Declarative Access to Filesystem Data

New application domains for XML database management systems

Alexander Holupirek

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Fachbereich Informatik und Informationswissenschaft
Mathematisch-Naturwissenschaftliche Sektion
Universität Konstanz

Referenten:
Prof. Dr. Marc H. Scholl
Prof. Dr. Marcel Waldvogel

Tag der mündlichen Prüfung:
17. Juli 2012

Abstract

XML and state-of-the-art XML database management systems (XML-DBMSs) can play a leading role in far more application domains as it is currently the case.

Even in their basic configuration, they entail all components necessary to act as central systems for complex search and retrieval tasks. They provide language-specific indexing of full-text documents and can store structured, semi-structured and binary data. Besides, they offer a great variety of standardized languages (XQuery, XSLT, XQuery Full Text, etc.) to develop applications inside a pure XML technology stack. Benefits are obvious: Data, logic, and presentation tiers can operate on a single data model, and no conversions have to be applied when switching in between.

This thesis deals with the design and development of XML/XQuery driven information architectures that process formerly heterogeneous data sources in a standardized and uniform manner. Filesystems and their vast amounts of different file types are a prime example for such a heterogeneous dataspace. A new XML dialect, the Filesystem Markup Language (FSML), is introduced to construct a database view of the filesystem and its contents. FSML provides a uniform view on the filesystem's contents and allows developers to leverage the complete XML technology stack on filesystem data.

BaseX, a high performance, native XML-DBMS developed at the University of Konstanz, is pushed to new application domains. We interface the database system with the operating system kernel and implement a database/filesystem hybrid (BaseX-FS), which is working on FSML database instances. A joint storage for both the filesystem and the database is established, which allows both developers and users to access data via the conventional and proven filesystem interface and, in addition, through a novel declarative, database-supported interface. As a direct consequence, XML languages such as XQuery can be used by applications and developers to analyze and process filesystem data. Smarter ways for accessing personal information stored in filesystems are achieved by retrieval strategies with no, partial, or full knowledge about the structure, format, and content of the data ("Query the filesystem like a database").

In combination with BaseX-Web, a database extension that facilitates the development of desktop-like web applications, we present a system architecture that makes it easier for application developers to build content-oriented (data-centric) retrieval and search applications dealing with files and their contents. The proposed architecture is ready to drive (expert) information systems that work with distinct data sources, using an XQuery-driven development approach. As a concluding proof of concept, a complete development cycle for an OPAC (Online Public Access Catalogue) system is presented in detail.

Zusammenfassung (German Abstract)

XML einerseits und moderne XML-Datenbank-Management-Systeme (XML-DBMS) andererseits können als Basistechnologie weit mehr leisten, als ihnen derzeit zugetraut wird.

Bereits in ihrer Grundausstattung beinhalten sie alle notwendigen Komponenten, die für den Aufbau und den Betrieb komplexer Such- und Informationsdienste notwendig sind. Der Umgang mit Volltexten und deren sprachspezifische Indexierung gehört ebenso zu den Aufgaben eines modernen XML-DBMSs wie die Speicherung von strukturierten, semi-strukturierten oder binären Daten.

Sie verfügen über ein reichhaltiges Arsenal an XML verarbeitenden Sprachen (XQuery, XSLT, XQuery Full Text, etc.) und bieten damit einen kompletten Technologiezweig an, der es erlaubt, innerhalb einer reinen, also nur auf XML Technologie basierenden Umgebung Applikationen zu entwickeln. Die Vorteile liegen auf der Hand: Von der Speicherung über die Verarbeitung bis hin zur Ergebnispräsentation kann das gleiche Datenmodell ohne Transformation zwischen den einzelnen Schichten einer Systemarchitektur erfolgen.

Die vorliegende Arbeit erprobt die Verwendung von XML-DBMSs auf bisher unbekanntem Terrain und untersucht deren Einsatzmöglichkeiten innerhalb moderner Betriebssysteme. Wir zeigen, wie über den Einsatz von XML-DBMSen eine deklarative Schnittstelle zur Abfrage von Dateisystem-Inhalten mittels XQuery geschaffen werden kann und implementieren ein hybrides Datenbankdateisystem (BaseX-FS). Die Technologiestudie erlaubt es, auf den Daten des Dateisystems, sowohl konventionell, also über die vom Betriebsystem angebotenen *system calls* und den *filesystem namespace*, zu arbeiten, als auch mit Hilfe der vom Datenbanksystem angebotenen deklarativen Zugriffsmethoden. Das heisst insbesondere, dass die in BaseX-FS gespeicherten Dateien semantisch und inhaltsbezogen über XQuery abgerufen und verarbeitet werden können, als auch, dass über die Verzeichnishierarchie inhaltsbezogene Daten einer Datei exportiert und mit konventionellem File I/O bearbeitet werden können.

Unter Verwendung von BaseX-FS als Basisarchitektur lässt sich zeigen, dass zahlreiche Dienste, wie zum Beispiel Desktopsuchmaschinen sehr viel leichtgewichtiger implementiert und funktional erweitert werden können, als dies bisher der Fall ist.

Zusammen mit BaseX-Web, einer Datenbankerweiterung, die es erlaubt, desktop-ähnliche Web-Applikationen zu entwickeln, zeigen wir, dass sich die vorgestellte erweiterte Datenbankarchitektur sehr gut für den Aufbau von Expertensuchsystemen, wie zum Beispiel eines Online Public Access Catalogues (OPAC), eignet.

Contents

1 Introduction

Today, almost exactly 14 years after the W3C released the first XML Recommendation on February 10, 1998 [8], XML has become an integral part of modern information systems. The markup language was originally envisioned as a language for defining new document formats and is suited especially well for that purpose. Besides it offers a rich set of accompanying standards[1], languages and processing techniques dealing with XML data:

XQuery 1.0: An XML Query Language "that uses the structure of XML intelligently *[to]* express queries across all *[...]* kinds of data, whether physically stored in XML or viewed as XML via middleware. XQuery is a full declarative programming language, and supports user-defined functions, external function libraries (modules) referenced by URI, and system-specific *native* functions." [53]

XQuery Update Facility (XQUF) "provides expressions that can be used to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model." [54]

XQuery and XPath Full Text (XQFT) "a language that extends XQuery 1.0 and XPath 2.0 with full-text search capabilities. XML documents may contain highly structured data (fixed schemas, known types such as numbers, dates), semi-structured data (flexible schemas and types), markup data (text with embedded tags), and unstructured data (untagged free-flowing text). Where a document contains unstructured or semi-structured data, it is important to be able to search using Information Retrieval techniques such as scoring and weighting." [12]

XSLT (XSL Transformations) 2.0 "a language for transforming XML documents into other XML documents. The term stylesheet reflects the fact that one of the important roles of XSLT is to add styling information to an XML source document, by

[1] `http://www.w3.org/standards/xml/`

transforming it into a document consisting of XSL formatting objects (see Extensible Stylesheet Language (XSL)), or into another presentation-oriented format such as HTML, XHTML, or SVG. However, XSLT is used for a wide range of transformation tasks, not exclusively for formatting and presentation applications." [43]

XSL: The Extensible Stylesheet Language. "Given a class of arbitrarily structured XML documents or data files, designers use an XSL stylesheet to express their intentions about how that structured content should be presented; that is, how the source content should be styled, laid out, and paginated onto some presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book." [4]

XProc: An XML Pipeline Language "for describing operations to be performed on XML documents. An XML Pipeline specifies a sequence of operations to be performed on zero or more XML documents. Pipelines generally accept zero or more XML documents as input and produce zero or more XML documents as output." [66] Operations can be of different nature but typically include validation, transformation, or querying of XML data.

XQSE: XQuery Scripting Extensions. "XQuery is a functional language that is Turing-complete and well suited to write code that ranges from simple queries to complete applications. However, some categories of applications are more easily implemented by combining XQuery capabilities with some imperative features, such as the ability to explicitly manage internal states. The same issue stands for XQuery enriched with the XQuery Update Facility *[...]*. The scripting extension is intended to overcome this problem, and allow programmers to write such applications without relying on embedding XQuery into an external language." [15], [60]

Fourteen years ago, a key feature, the easy definition of new document formats, paved the way for the huge success of XML as a data exchange format. Compared to ASN.1—a standard for the abstract definition of data types and an, at that time, established method to communicate between heterogeneous systems—the uniform description of data in XML and its subsequent processing is a straight-forward task. XML, as a textual format, is easy to edit, simple to parse and may represent structured, semi-structured or unstructured data. Associating XML files with a schema allows to validate

XML contents, but a less complicated ad-hoc approach—the so-called schema-agnostic processing—is possible as well and widely-used in practice.

While XML, in its early years, has been mostly used for data exchange—for example as a replacement of older formats in Electronic Data Interchange (EDI)—it was soon accepted as a suitable data storage format by many applications. In the beginning, only small files like the famous `.ini` or other textual configuration files have been replaced. But other applications, such as Apache's `ant(1)` software build tool[2], chose XML from the start. Integration of XML parsers in just about any common programming language made this a comprehensible choice. It allows the use of a standardized toolchain to parse files, check their validity against a Document Type Definition (DTD) or XML Schema Definition (XSD) and subsequently process the data accordingly. The files are readable by both humans and machines and can therefore easily be modified and adapted manually or automatically.

As the story goes on, more applications joined the party and chose XML as a storage format. A prime example is the OASIS Open Document Format (ODF). It "is an open XML-based document file format for office applications to be used for documents containing text, spreadsheets, charts, and graphical elements. The file format makes transformations to other formats simple by leveraging and reusing existing standards wherever possible." [67]

What we can observe in general today is an ever increasing number of XML collections emerging in different areas of application. Best practice, storing XML files in the filesystem, is more and more becoming a bottleneck, and an increasing interest in supporting database technologies can be observed, especially in the industrial sector.

During the first hype of XML, processing XML with dedicated database systems could not fulfill people's expectations. Systems were unstable and not ready for production or did not meet the demands in terms of processing speed or scalability. After a poor start, the situation has changed. Now, a decade later, we face market-ready XML databases in just about every big players database portfolio. Besides well established database providers, such as Oracle, IBM and Microsoft, several smaller companies and open source projects, solely focussed on XML, emerged and matured:

[2] Apache Ant is very similar to the popular Unix `make(1)` tool. Its mission is to orchestrate processes (described in build files as targets and extension points) dependent upon each other. XML is used in the build files to define the rules to compile, assemble, test and run applications.

MarkLogic is the leading company in the niche market of XML database management systems. Their credo is to provide “21st century technology for 21st century challenges” [45]. For MarkLogic “traditional relational databases were built for another era and organizations are seeking alternatives to address today’s information management challenges” [45].

“Organizations are struggling to manage and leverage Big Data. Unstructured information and other complex, valuable data can be particularly difficult to capitalize on. Examples of unstructured information include: documents, rich media like images or videos, metadata, content, user-generated content, RSS feeds, e-mail, geospatial data, and XML among others. Typically, unstructured information has one or more of the following characteristics:

- Heterogeneous (different formats, varying standards, irregular lengths, etc.)
- Constantly evolving in ways that may be unanticipated
- Growing exponentially

These characteristics make it difficult to manage unstructured information using previous technologies, such as relational databases, which typically expect reasonably-sized data that is normalized and conforms to a pre-defined schema.

MarkLogic 5 is the company’s flagship product: a next generation database for managing and leveraging Big Data and unstructured information. Such information may be textual, irregular, hierarchical, de-normalized, time-varying, or structured in an unexpected way.” [46]

Documentum xDB is offered by EMC Corporation as a “high-performance and scalable native XML database designed for software developers who require advanced XML data processing and storage functionality within their applications. xDB enables high-speed storage and manipulation of very large numbers of XML documents. Using xDB, programmers can build custom XML content management solutions and store XML documents in an integrated, highly scalable, high-performance, object-oriented database.” [14]

Quizx is “a fast XML database engine fully supporting XQuery. Qizx is designed from the ground up to perform fast queries, without requiring specific efforts from users. Queries run at full speed out of the box without the need to manually define indexes, tweak parameters, or add a new index." [51]

eXist-db "is an open source database management system. It stores XML data according to the XML data model and features efficient, index-based XQuery processing. It supports many Web 2.0 technology standards, making it an excellent platform for developing web-based applications." [48]

Sedna "is a free native XML database which provides a full range of core database services - persistent storage, ACID transactions, security, indices, hot backup. Flexible XML processing facilities include W3C XQuery implementation, tight integration of XQuery with full-text search facilities and a node-level update language." [36]

BaseX "is a very light-weight, high-performance and scalable XML Database engine and XPath/XQuery Processor, including full support for the W3C Update and Full Text extensions. Various interactive and user-friendly graphical user interfaces give great insight into stored XML documents. BaseX is developed at the Chair of Databases and Information Systems at the University of Konstanz as an open source system under the terms of the BSD license." [13]

In addition there are **Saxon**[3] and **Zorba**[4], powerful XQuery processors of high renown in the community.

Given these developments, we want to tap the full potential of current XML-DBMSs and put them to the test in somewhat unfamiliar territories. Our bold statement is that XML databases with their current characteristics can serve as core components for search and retrieval systems on heterogeneous data sources. Filesystems are prime examples. They store vast amount of heterogeneous data formats. Providing means to programmatically query, process and analyze personal data stored in filesystem would be a major improvement over current filesystem abilities.

[3] `http://www.saxonica.com/`

[4] `http://zorba-xquery.com`

1.1 Motivation

1.1.1 Intrinsic Motivation - Personal Data Mess

Trying to find things—I definitely know I have—is a common task for me. This is true for my real life (but, lucky me, there are always nice people around helping me out) and even more for my digital self.

As a matter of fact, it is getting worse all the time. Cloud storage and multiple mobile devices do not simplify matters in this respect. With every new machine my disk space to mislay things increases.

Of course, it may be considered a bad habit to just copy data over from my old machine to the new laptop instead of curating, archiving and purging data from the working system. But I seem to be in good company and in-line with established practice: Already back in 2002, Jim Gray, while talking about data curation, 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." [20]

In fact, mere storage of (personal) data in state-of-the-art filesystems is a markedly well done job in current operating systems. Convenient access to and information retrieval from such data, however, is crucial to leverage the stored information. Recent variants of operating systems therefore come with integrated search capabilities[5] or can easily be equipped with a third-party desktop search application. These tools clearly offer a smarter way to access personal information stored in the filesystem (and tell me that I'm not alone needing help to recover once stored assets).

1.1.2 Professional Challenge - Retrieval Support for Filesystems

Working in a database group, however, I can not be satisfied. *We*, occupationally, want to store anything we consider useful in a database and have it ready to be queried.

[5] (*e.g.*, Instant Search on the Windows platform, the Spotlight architecture on Macintosh, or Tracker and Beagle on Linux systems)

The way *we* want to explore our data is via a *standardized and established database query language (DQL)*. Finding things using a keyword-based search expression is just the beginning of what *we* would expect from an information system that keeps track of our (personal) data.

Since the beginning of database management systems, there is a desire to store all data in a database and have it ready to be queried. Several industrial and research efforts such as WinFS or the Be Filesystem have been made to push the filesystem into a database. None made it to technical production quality.

Offshoots, like Microsoft's Instant Search or Apple's Spotlight Architecture, however, can be found in all of the recent operating system variants, and a user demand for products helping to find relevant content can be derived from the increasing popularity of Desktop Search Engines, such as Google's or Yahoo's Desktop Search.

While these tools offer a smarter way to access personal information stored in the filesystem, the keyword-driven search approach, as used by today's search engines, is—while perfectly suitable for the everyday business—just the beginning of what can be expected.

An additional support for database style query languages to *"filter, select, search, join, sort, group, aggregate, transform, and restructure"*, in short, analyze and programmatically process, stored data, would be a consequential further development.

1.2 Problem Description

We generally face the fact that the amount of data stored in filesystems on personal computers is growing steadily. This comes as no surprise since—against current opinion—data gets copied from old machines to new ones instead of being archived. 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. Therefore filesystems 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 filesystems and their proven, but basic interface (VFS) support neither.

Donald Norman coined the phrase *"Attractive things work better"* [50]. While Norman's statement, in the first place, aimed 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 is determined by the ability to search/find and access/use stored data.

In fact, the challenge we now face (and will even more in the future) is to enhance storage systems in a way that users can make full use of their data. Finding and programmatically process *relevant content* in this ever growing amount of data is a major aspect. Filesystems still focus on mere storage and tend to be conservative regarding feature enhancements [70]. Consequently, they do not offer solutions to this demanding task.

Current solutions to find files are developed outside the filesystem as separate, concurrent systems. Redundant storage of metadata is common practice in modern operating systems and applications. Integrated file indexing services, such as Windows Search or Apple's Spotlight, crawl the filesystem in order to harvest metadata. Domain-specific applications, such as audio players or e-mail applications, harvest relevant information for *their* file types and store them in accompanying index structures. The extracted information is used to provide retrieval and search functionality to the user. In times of ever increasing personal data masses, this obviously is a frequently demanded and useful feature.

Today's solutions do not develop the exploration of the collected metadata to the maximum. Application-specific solutions fail to reoffer the extracted information via a public interface. As such, they hide relevant data, and peer applications have to perform the same work again.

System-wide APIs to access the stored information use an imperative programming style only and, while suited to access single data items, do not allow for sophisticated declarative programming. While we consider keyword-driven search a suitable approach for end-users and ad-hoc queries, we postulate that it is not enough to cope with the explosive growth of personal information and the full variety of present and future search and retrieval tasks.

A more general and ideally standardized storage facility for the harvested data would make it easier for applications and developers to profit from the tediously collected information.

We opt for XML database technology to provide such an infrastructure and to establish a system-wide service to export harvested metadata in a standardized, well-defined format that is suitable for further processing. Choosing XML allows to leverage the complete and feature rich *X-technology stack* developed and standardized by the World Wide Web Consortium (W3C).

Therefore, we propose the description of filesystem's contents and metadata applying an XML dialect, to use a high-performant and scalable XML store, together with a full-fledged and highly compliant XQuery processor to system-wide expose the collected data.

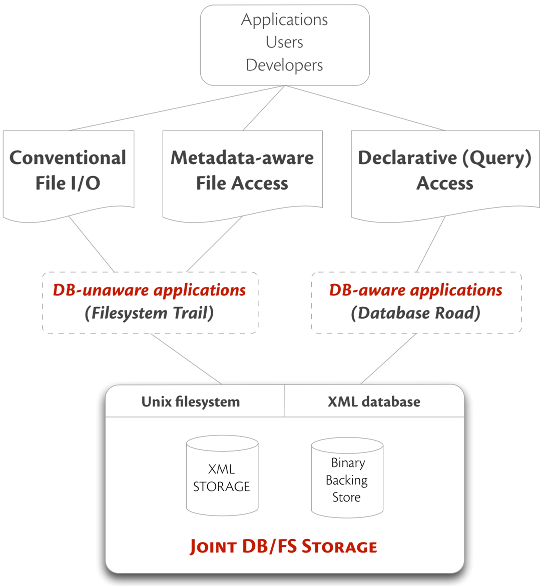

Applications
Users
Developers
Conventional File I/O
Metadata-aware File Access
Declarative (Query) Access
DB-unaware applications (Filesystem Trail)
DB-aware applications (Database Road)
Unix filesystem
XML database
XML STORAGE
Binary Backing Store
JOINT DB/FS STORAGE

Figure 1.1: Dual access to filesystem data

The approach has two main effects: It will provide an additional semantic, content-related view on the filesystem and its stored content. Using XML it becomes possible to express the logical structure of files (as also proposed by semantic desktop or filesystem approaches). That way, the system *knows* about the insides of a data source and can retrieve parts of it (for instance just the subject of an e-mail). This stands in stark contrast to current filesystems, that treat files as a mere sequence of bytes. *Database-aware applications* are able to directly query and process this information and have a more fine-grained view on the system. Interconnections between data assets of different kinds, for example, can be explored more easily using a declarative query language that is designed to this end. We will re-export the collected data back in the filesystem namespace, so that legacy applications can profit as well. Finally, conventional File I/O for database-unaware applications as well as database-enhanced access to the same data is provided. Figure 1.1 on the preceding page illustrates the concept.

1.3 Research Approach

Despite the fact that several database-driven filesystem attempts have already failed, the advent of XML brought some significant enhancements to DBMS that inspired us to dare another attempt.

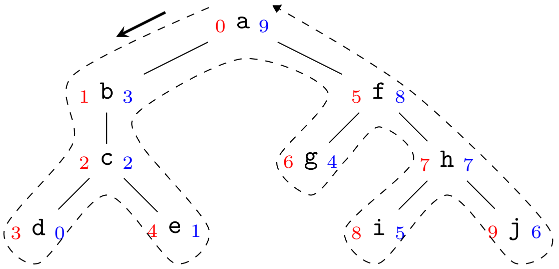
In a preliminary study [34], we evaluated the mapping of a file hierarchy and its content to XML and emulated filesystem operations using XPath/XQuery/XQUF operations. We found it possible to perform basic filesystem commands, as well as content-based retrieval, in interactive time on the constructed filesystem mappings with an off-the-shelf XML database. Motivated by these results we pushed the idea forward.

The tree-based XML model has spawned efforts on relational storage and processing techniques for hierarchically structured data and meanwhile, DBMSs have learned to work with tree-shaped data (*e.g.*, [5, 6, 22–24]). This is of direct benefit, as the hierarchic nature of filesystems can now consistently be mapped to the relational storage (see Figure 1.2 on the next page) and leverage the associated algorithms (an elaborate discussion of relational XML storage and algorithms can be found in [64, Chapter 2]).

BaseX, the database we use within this project, is also built on a relational encoding scheme as will be discussed in Chapter 2 on page 25.

A major problem of storing files in a DBMS (apart from using BLOBs) has been the basic necessity of providing a schema first. With an unmanageable amount of file formats this appears to be impossible. Schema-oblivious storage techniques made it possible for XML

```
0<a>
  1<b>
    2<c>
      3<d/>0
      4<e/>1
    </c>2
  </b>3
  5<f>
    6<g/>4
    7<h>
      8<i/>5
      9<j/>6
    </h>7
  </f>8
</a>9
```



pre	post	*n*
0	9	a
1	3	b
2	2	c
3	0	d
4	1	e
5	8	f
6	4	g
7	7	h
8	5	i
9	6	j

Figure 1.2: Basic (simplified) idea of storing trees (such as file hierarchies, XML documents) in a RDBMS [21]

data to be stored in the database without previous knowledge of its interior structure[6].

As mentioned, more and more applications use XML as their native storage format anyway. Data of this kind is already prepared to be handled with database technology. From our point of view, these documents are nothing else but serialized database instances. In consequence, they are not only stored as plain text, but directly shredded[7] into the DBMS. Legacy applications are still able to process them conventionally by requesting them in their serialized, *i.e.*, textual representation. In direct communication with the database, however, XML processing languages such as XPath/XQuery can be used on the data. Going through with the concept, and as filesystems are structured hierarchically, it seems to be a natural thing to also map the file hierarchy into tree-aware DBMSs.

[6] An additional XML Schema specification for the file type may be of advantage to formulate queries against the document, but is not mandatory.

[7] A terminus technicus used to indicate the conversation of XML in its textual representation to an internal format used by the database. Read it as "import".

1.4 Contribution and Outline

We will design and develop an XML/XQuery driven information architecture that works on formerly heterogeneous data sources in a standardized and uniform manner, leveraging semi-structured database techniques. The system will provide both proven and stable access to the data using filesystem techniques and query support for all stored files. As a consequence, our architecture will provide the following novel features:

- Database query capabilities on filesystem data as a general system service
- Unified view on (formerly heterogeneous) filesystem contents
- Declarative API to work with file objects
- Metadata-aware file access through the filesystem namespace

Furthermore, we will present work at different layers of a suitable DBMS architecture and show how application development inside a pure *X-Technology Stack* can be achieved.

Foundation. In a first step, we provide an xmlified, database-centric view of the filesystem's content. We gather file contents and express them in a new XML dialect designed for that purpose: FSML, the Filesystem Markup Language. The result is stored as a BaseX-FS database instance and ready to be queried via XQuery and related languages. That way we provide an unified view on filesystem data. It is the base for processing heterogeneous filesystem data with semi-structured database technology and will be described in the next Chapter.

The Database as Filesystem in Chapter 3 will dig down and contribute an implementation that establishes a link between DBMS and OS and shows how a database with XML/XQuery support can be used as a user-level filesystem. As a result, the DBMS is mounted as a Unix filesystem by the operating system kernel. Consequently, access via the established filesystem interface as well as database-enhanced access to the same data is provided (joint storage for filesystem and database). The database filesystem hybrid will provide metadata-aware file access (*"deep access"*) over the conventional filesystem interface.

Declarative Application Programming Interface. Having established the uniform view on heterogeneous filesystem data, we move up to the database frontend and show how the new database filesystem infrastructure can be used to facilitate, support and, in the final analysis, change application development. The system now provides a declarative application programming interface, and database-aware applications can directly profit from the database infrastructure. A selection of user interfaces implemented as BaseX views will demonstrate this and show how databases can be turned into primary processors for users and application developers dealing with information stored in files.

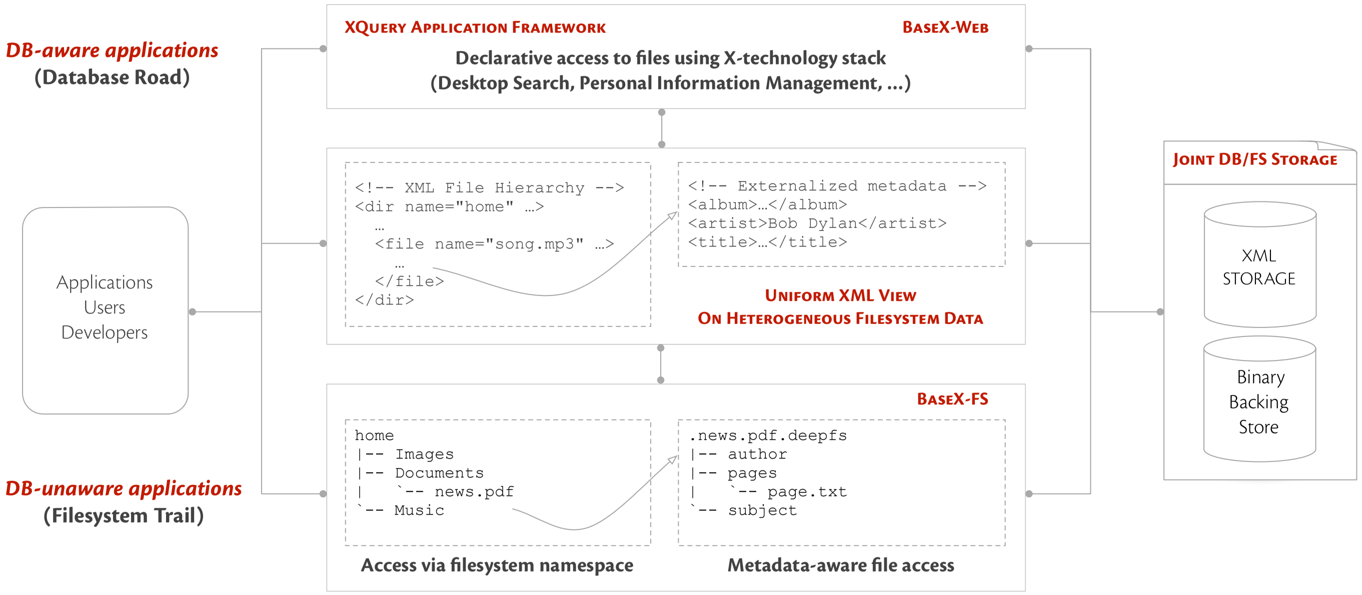
An XQuery Application Framework. In Chapter 4, we push the idea even further and develop an XQuery application framework to enable developers to implement database-aware applications inside a clean XML technology stack. Our expectation is that implementations developed on top of a pure W3C technology stack will show simpler, more flexible, and more efficient application code.

Kickstarting an Infrastructure. To finally demonstrate the feasibility of our approach and to put our architecture to the test we provide an actual example application and describe the development of an expert retrieval system from scratch. It leverages declarative access on documents stored in the filesystem (BaseX-FS), makes use of our application framework (BaseX-Web), and is built solely on XML technology.

Big picture. Figure 1.3 on page 23 illustrates the big picture. Users, developers, and applications gain two access paths to filesystem data. *Database-enhanced (declarative, "queryable") access* to data is provided as illustrated in the upper half of the figure entitled as *"Database Road"*. It allows for a uniform view on formerly heterogeneous data stored in the filesystem (Chapter 2). An XQuery Application Framework (Chapter 4) permits application development using a declarative programming style and allows developers to stay inside a clean standardized technology stack approved by the W3C.

Legacy access to filesystem data is still provided. A joint storage for filesystem and database is set up and the database is mounted as a filesystem by the operating system kernel (Chapter 3). That way both stable and proven access via the established filesystem interface as well as database-enhanced access can be achieved since the filesystem is the database and the database is the filesystem. This additionally allows for improved, metadata-aware (so-called *deep*) access—as it allows to navigate into the file—via the filesystem namespace.

Generalization. We will apply XML technology to implement what is typically considered to be solved by a variety of different languages and concepts, including low-level system programming. As such, we target new application domains for XML database management systems and propose enhancements for XML database architectures. The existing BaseX XML database management system is taken as a representative. Finally, the techniques discussed will serve as a general blueprint on how to design and develop XML/XQuery driven information architectures that work on formerly heterogeneous data sources in a standardized and uniform manner.


DB-aware applications
(Database Road)
Applications
Users
Developers
DB-unaware applications
(Filesystem Trail)
XQuery Application Framework
BaseX-Web
Declarative access to files using X-technology stack
(Desktop Search, Personal Information Management, ...)
Uniform XML View
On Heterogeneous Filesystem Data
BaseX-FS
Access via filesystem namespace
Metadata-aware file access
Joint DB/FS Storage
XML
STORAGE
Binary
Backing
Store

```
<!-- XML File Hierarchy -->
<dir name="home" …>
  …
  <file name="song.mp3" …>
    …
  </file>
</dir>
```

```
<!-- Externalized metadata -->
<album>…</album>
<artist>Bob Dylan</artist>
<title>…</title>
```

```
home
|-- Images
|-- Documents
|   `-- news.pdf
`-- Music
```

```
.news.pdf.deepfs
|-- author
|-- pages
|   `-- page.txt
`-- subject
```

Figure 1.3: Big picture and ultimate goal: Applications, users, and developers gain two access paths to file contents. Proven and stable access via the filesystem interface is retained. An enhanced, *metadata-aware (deep) file access* is provided as the data is stored in a joint storage for filesystem and database. The database is mounted as a filesystem by the operating system kernel *("Filesystem Trail")*. *Database-enhanced (declarative, "queryable") access* can leverage the complete range of XML technologies on filesystem data. Additionally an application framework to build software inside a unified W3C stack is proposed *("Database Road")*

2 The BaseX Filesystem View

Ever-growing data volumes demand for storage systems beyond current filesystems 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 filesystems are structured hierarchically they can be mapped to XML and as such stored in and queried by an XML-aware database system. Filesystems typically store vast amounts of heterogeneous file and data formats. The lack of a unified representation, however, makes it difficult for query languages to work through the data.

In the following, we present FSML, the Filesystem Markup Language, a novel XML dialect that maps filesystem entities to XML nodes. BaseX, a native XML database system, is used to store FSML instances in order to provide a standardized, uniform, and high-level representation of a filesystem. The proposed mapping will later on be used to:

Work on filesystem data using a database query language. XQuery, for example, can be used to search through, program with or analyze the data of a filesystem.

Mount the database as a filesystem by the operating system. We will establish a link between database and operating system. The database will be mounted as a conventional filesystem by the operating system kernel.

Implement applications using a declarative/functional programming style whenever it comes to the processing of file data. Traditionally, files are roughly classified as either text or binary. We add XML as a third type and expose formerly locked away content of files in a well-defined format with both, its structure and content.

The unified representation of a filesystem can be leveraged by applications, developers and users. It is, however, in first place targeted at application developers to finally offer a new declarative way of dealing with filesystem data. In Chapter 3 we show how XQuery can be used in the domain of system programming. We will hook the database into the operating system and re-export the database content via the filesystem namespace.

2.1 Joint Storage for Filesystem and Database

In Figure 1.3 on page 23 we gave a high-level overview of the system's architecture. A key element is the joint storage system used by both the filesystem and the database. BaseX supports the storage of semi-structured XML and binary data. We will use its storage layer to assemble all data necessary to drive a filesystem (file hierarchy, filesystem metadata, user data).

The XML Store supports updates and, beside the usual name, path, and value indexes, maintains two full-text index structures: A *fuzzy index* is centered on specialized approximate matches, and a *trie index* supports wildcard queries. Both versions yield fast results for exact queries [28]. The system is an early adopter of the XQuery Full Text Recommendation [12] and supports sequential scanning, index-based, and hybrid processing of full-text queries. The support for textual retrieval at the core of the database engine makes BaseX a good choice to power our content-aware filesystem representation.

2.1.1 The pre/distance/size Encoding

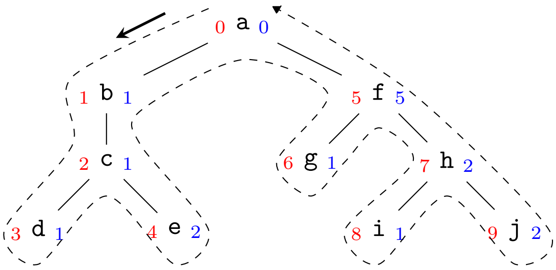
BaseX' storage layer uses a pre/distance/size encoding for XML data with various compactification techniques, such as attribute and integer inlining [25]. It is derived from the XPath Accelerator encoding [21], which is used in the MonetDB/XQuery system[1]. Those flat tree encodings have proven to show excellent query performance [6, 23, 25, 26].

Figure 2.1 shows a pre/distance/size encoded tree. The `pre` value is dense and ordered for the complete tree structure, and it is implicitly given by its position. `dist` defines the

[1] `http://www.monetdb-xquery.org/`

```
$ tree ./a
0a0
|-- 1b1
|    `-- 2c1
|        |-- 3d1
|        `-- 4e2
`-- 5f5
    |-- 6g1
    `-- 7h2
        |-- 8i1
        `-- 9j2
```

```
<a>
 <b>
   <c>
     <d/>
     <e/>
   </c>
 </b>
 <f>
   <g/>
   <h>
     <i/>
     <j/>
   </h>
 </f>
</a>
```



pre	dist	size	*n*
0	0	9	a
1	1	3	b
2	1	2	c
3	1	0	d
4	2	0	e
5	5	4	f
6	1	0	g
7	2	2	h
8	1	0	i
9	2	0	j

Figure 2.1: Storing trees (such as file hierarchies, XML documents) in the pre/distance/size encoding

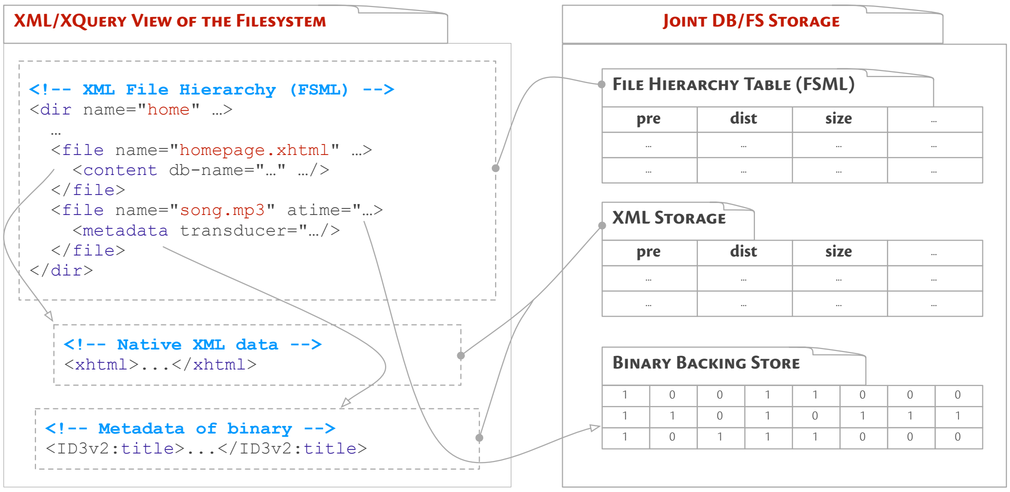
relative distance to the parent `pre` value, and `size` contains the number of descendants of a node.

To facilitate updates, the table structure is organized in disk blocks. A block directory references the first `pre` value of each block. The `dist` and `size` values have to be modified if deletions/insertions are performed: The `size` values are updated for all ancestors of that node—which means that a maximum of $log(n)$ nodes in the tree has to be accessed—and the `dist` values are updated for the following siblings and the following siblings of the ancestor nodes. In comparison, *e.g.*, the storage of absolute parent references would ask for a complete renumbering of all nodes in the tree table that follow a deleted/inserted node, rendering it inapt for updates in filesystems.

2.1.2 The Encoded File Hierarchy

As the pre/distance/size encoding is essentially a storage for tree structures, it can be seamlessly used to store the file hierarchy of a filesystem. The hierarchical mapping of filesystems is straight-forward, as illustrated in Figure 2.1.

A more detailed view of the joint storage [33] is shown in Figure 2.2.


XML/XQuery View of the Filesystem
Joint DB/FS Storage
File Hierarchy Table (FSML)
XML Storage
Binary Backing Store

```
<!-- XML File Hierarchy (FSML) -->
<dir name="home" …>
  …
  <file name="homepage.xhtml" …>
    <content db-name="…" …/>
  </file>
  <file name="song.mp3" atime="…>
    <metadata transducer="…/>
  </file>
</dir>
```

```
<!-- Native XML data -->
<xhtml>...</xhtml>
```

```
<!-- Metadata of binary -->
<ID3v2:title>...</ID3v2:title>
```

pre	dist	size	…
…	…	…	…
…	…	…	…

pre	dist	size	…
…	…	…	…
…	…	…	…

1	0	0	1	1	0	0	0
1	1	0	1	0	1	1	1
1	0	1	1	1	0	0	0

Figure 2.2: Joint storage for filesystem and database. Uniform XML representation of filesystem content

From the XML/XQuery perspective, the system stores an XML representation of the filesystem, which is valid against a W3C XML Schema Definition. A BaseX-FS database instance consists of three components:

- The FSML database that contains the file hierarchy tree.
- XML databases for well-formed native XML documents and extracted information from files.
- The binary backing store that stores raw data for any file in the file hierarchy, except those XML files turned into a database.

Any information relevant to operate a traditional filesystem is stored in the “File Hierarchy Table” and is accessible for the XQuery processor as well as for operating system requests, as we will see later. Following the Unix tradition there are block and character, directory, fifo, (symbolic) link, socket and regular file types. Each file type is expressed as XML element, *e.g.*, `<file/>`, and augmented with its file attributes (file size, access time, protection mode, …).

```
<file name="05_like_a_rolling_stone.mp3" suffix="mp3" st_mode="100644"
      st_uid="501" st_gid="20" st_size="8943004" st_nlink="1"
      st_mtime="1323101051" st_ctime="1323168268" st_atime="1324470592"/>
```

Listing 1: FSML file element with file attributes

When the database is mounted as a filesystem at a later point, those attributes are used to provide file status information to the operating system kernel (as, for instance, necessary to process the `stat(2)` system call).

2.2 Leverage Tacit Information Hidden in Files

Another view at Figure 2.2 on the facing page reveals that content and inherent metadata of files is taken into account and explicitly represented in the BaseX-FS database instance. Our mapping breaks with the long tradition to consider files as just a sequence of bytes. A central point of the integration of contents into the XML representation of the filesystem is to allow the full range of XQuery retrieval features on the data. While XML files are ready to be included without additional effort (unless schema validation is demanded), commonly used binary files, such as images or audio files, contain metadata, which is quite relevant for querying.

Our basic approach is that any information is considered an *asset of interest* that may, at a later point, be useful for information retrieval, personal information management, or related tasks. Assets of interest are to be defined for file types, *e.g.*, audio files, e-mails, pictures, etc. and typically belong to one of the four categories:

- Inherent file metadata (encapsulated within the file, *e.g.*, ID3 information for music data, EXIF annotations for image files)
- System metadata (filename, file size, modification times, …)
- File content (full-text contained in office documents, e-mails etc.)

- User annotations, like tags, etc.

Often, a number of different assets of interest exist for a given file. Those are bundled together and form what we call a *metadata entry (MDE)*. A metadata entry is the XML encoded view of a file and is suited well for querying. Together with the original regular file in the backing store it forms the database view of a file.

2.2.1 Transducers – Filetype-specific Data Extractors

Metadata entries are constructed by so-called *transducers* (which first appeared in the context of the Semantic File System [17]). Transducers are file-specific metadata extractors. Transducers exist for various file types and can be plugged into the system to expose file-specific metadata. An extensible and configurable architecture has been chosen for the implementation of transducers to

- facilitate the support for new filetypes
- enable developers and organizations to produce metadata entries that contain exactly the (meta)data of files they want to query. BaseX-FS provides the framework in which transducers can feed metadata entries in order to build user-defined views of the filesystem in XML

Transducers are triggered by the detected file MIME type. Transducer plugins can register for various MIME types and will be invoked once a file of that type is processed.

The detected metadata is added as separate XML documents to the database and the file hierarchy mapping is augmented by a reference to the metadata entry.

Listing 2 on the next page is an example of what is stored in the native XML database. A transducer for audio files has detected some ID3 information and the `<file>` element is augmented with file attributes taken from the operating system.

```
<!-- File Hierarchy encoded in FSML (fsml.xml) -->
<fsml version="1.0">…
  <dir name="Music" st_size="…">
    <file name="05_Like_A_Rolling_Stone.mp3" suffix="…">
      <metadata transducer="exiftool" db-size="10144" db-nodes="73"
                db-name="fsml-522dd6df-169d-4edf-aaf3-e2396e18dfab"
                db-timestamp="16.01.2012 10:57:48"
                doc-size="2499 Bytes" doc-encoding="UTF-8"
                whitespace-chopping="true"/>
    </file>
  </dir>…
</fsml>

<!-- for $metadata in doc("fsml")//file/metadata/@db-name
     return doc($metadata) -->
<metadata transducer-toolkit="Image::ExifTool 8.68"
          xmlns:MPEG="http://ns.exiftool.ca/MPEG/MPEG/1.0/"
          xmlns:ID3v2_3="http://ns.exiftool.ca/ID3/ID3v2_3/1.0/"
          xmlns:Composite="http://ns.exiftool.ca/Composite/1.0/">
  <MPEG:MPEGAudioVersion>1</MPEG:MPEGAudioVersion>
  <MPEG:AudioLayer>3</MPEG:AudioLayer>
  <MPEG:AudioBitrate>192 kbps</MPEG:AudioBitrate>
  <MPEG:SampleRate>44100</MPEG:SampleRate>
  <MPEG:ChannelMode>Stereo</MPEG:ChannelMode>
  <MPEG:MSStereo>Off</MPEG:MSStereo>
  <MPEG:IntensityStereo>Off</MPEG:IntensityStereo>
  <MPEG:CopyrightFlag>False</MPEG:CopyrightFlag>
  <MPEG:OriginalMedia>False</MPEG:OriginalMedia>
  <MPEG:Emphasis>None</MPEG:Emphasis>
  <ID3v2_3:Title>Like A Rolling Stone</ID3v2_3:Title>
  <ID3v2_3:Artist>Bob Dylan</ID3v2_3:Artist>
  <ID3v2_3:Composer>Bob Dylan</ID3v2_3:Composer>
  <ID3v2_3:Album>Greatest Hits</ID3v2_3:Album>
  <ID3v2_3:Track>5/10</ID3v2_3:Track>
  <ID3v2_3:PartOfSet>1/1</ID3v2_3:PartOfSet>
  <ID3v2_3:Year>1965</ID3v2_3:Year>
  <ID3v2_3:Genre>Folk</ID3v2_3:Genre>
  <ID3v2_3:Comment>(iTunPGAP) 0</ID3v2_3:Comment>
  <ID3v2_3:EncodedBy>iTunes 8.0.2</ID3v2_3:EncodedBy>
  <ID3v2_3:Comment>(iTunNORM)  00 … 00042A05</ID3v2_3:Comment>
  <ID3v2_3:Comment>(iTunSMPB)  00 … 000000</ID3v2_3:Comment>
  <ID3v2_3:Comment>(iTunes_CDDB_IDs) 10 … 750289</ID3v2_3:Comment>
  <Composite:DateTimeOriginal>1965</Composite:DateTimeOriginal>
  <Composite:Duration>0:06:12 (approx)</Composite:Duration>
</metadata>
```

Listing 2: Metadata extracted for .mp3 file using ExifTool transducer

Transducers externalize data formerly siloed in filesystems. The extraction of tacit information, encapsulated in various file formats, leads to a standardized and easily accessible representation. Content and structure of file data is exposed and can now be queried together, as the extracted data is presented in a homogeneous manner. The data is indexed and we can search on anything that has been loaded without knowing questions ahead of time.

Think, for instance, about finding an e-mail with a known sender, a big attachment and some keywords:

```
for $mail in //file/Mail
let $attach := $mail/Attachment
where $mail/From = 'jim.walker@mail.com'
  and $mail/Section
      contains text 'Hansson' ftand 'report'
  and $attach/@size > 3000000
return fsml:path($mail)
```

Listing 3: XQuery pseudo-code to retrieve relevant e-mails

Queries may combine filesystem metadata (such as file size, directory names) with file content and use both filesystem commands and languages for semi-structured data, such as XQuery, to request and manipulate data. In the case of e-mails, comparable functionality is already offered by advanced e-mail applications. However, each application has to provide its own implementation, leading to highly redundant code for similar functionality. Our approach strives to provide such capabilities as a basic system service. Furthermore, the search is not restricted to application-defined communication paths (such as the often connected e-mail, calendar, address book applications), but can include any stored data.

2.2.2 Implementation of a Transducer

Several sophisticated tools in the open-source domain focus on metadata extraction. *ExifTool* [30] is a good example. It is in operation and under constant development since 2003 and supports an astonishing amount of more than 130 file types. Following established software engineering practice we want to put those tools to use for our goal to externalize information in a homogeneous manner.

A plugin architecture has been chosen for that purpose.

While it can be quite difficult to write extraction code to get information from raw data, it is easy to deploy a new transducer and to integrate it into our architecture, as it boils down in supporting a simple interface:

- `register(list of mime types supported by transducer)`
- `<metadata/> extract(fileref)`
- `inject(fileref, <updates/>)`

The implementation has to be thread-safe, the functions are called back by the system when appropriate. Plugins are initially loaded into the BaseX-FS Database Server on startup. They may be provided as external dynamic libraries or included into the project code. Additional plugins can be loaded and removed from the server during runtime. If multiple transducers are registered for the same MIME type, they are executed in sequence.

2.3 A Deeper Filesystem – The Metadata Hierarchy

Back in 1998 Simon St. Laurent published a short essay [61] that contained the following Figure 2.3:

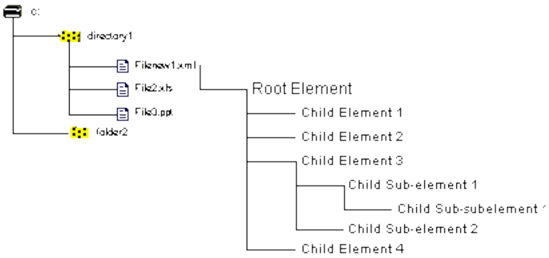


Figure 2.3: Simon St. Laurent's vision of an enhanced, "deeper" filesystem

St. Laurent writes: "Implementing this requires a drastic rethinking of the file system and database structures as well. Supporting retrieval at the element level breaks down formerly monolithic binary files (or, in database terms, Binary Large Objects or BLOBs) into separate, often tiny chunks which may themselves continue other chunks, which contain other chunks, and so forth. At this point, the file system is no longer a file system in the traditional sense, but an object store which is capable of storing large chunks of information as well as hierarchies built of tiny data sets. The document still exists - but only as one layer of the object store, an object containing other objects much as directories contain files at present." [61]

We were excited about the idea of letting the filesystem immerse into files to have an enhanced, deeper, and more fine-grained access to data. And given a BaseX-FS instance, other—more specific or application-tailored—views on a filesystem can be created easily. We came up with another XML representation, called *DeepFS*, that integrates selected items of metadata entries into the file hierarchy.

Beside the well-known *file and directory* hierarchy, DeepFS establishes a second *metadata* hierarchy. Assets of interest are structured in `<fact/>` and `<folder/>` elements. *Facts* are leaf nodes in the metadata hierarchy and contain values, such as *'Bob Dylan'* in an *'artist'* fact of an audio file or the full-text of a PDF in the *'page'* fact.

Folders recursively contain, analogous to directories, zero or more facts or folders. They, for instance, group the individual *page facts* of a PDF document to a *pages folder*. An example is given in Listing 4 on the next page.

```
$ tree -a /var/tmp/mnt/
/var/tmp/mnt/
|-- a.mp3
`-- .a.mp3.deepfs
    |-- artist
    |-- sub
    |   `-- genre
    `-- title

    2 directories, 4 files
```

Facts and folders form the metadata hierarchy that is exposed in the Unix filesystem namespace. Per convention, a known file type is expected to expose its metadata in a folder with an annotation of `type="metadata"`. It denotes the root of the metadata hierarchy along which deep access to the regular file is established. DeepFS prolongs the conventional file hierarchy with a metadata hierarchy. When mounting the database as filesystem this metadata hierarchy is reflected in the filesystem namespace again in order *to navigate into the file*.

```
<dir name="Documents" st_mode="040755" ...="...">
  <file name="BBC_News-Mars_Nasa_images.pdf" suffix="pdf"  ... >
    <folder name=".BBC_News-Mars_Nasa_images.pdf.deepfs" type="metadata">
      <fact name="pagecount">2</fact>
      <fact name="title">Mars: Nasa images show signs of flowing water</fact>
      <fact name="author">Hamish Pritchard (Science Reporter)</fact>
      <fact name="subject">Science &amp; Environment</fact>
      <fact name="keywords"/>
      <fact name="creator">Google Chrome</fact>
      <fact name="producer">Mac OS X 10.6.8 Quartz PDFContext</fact>
      <fact name="creationdate">2011-08-10T15:11:03.000Z</fact>
      <fact name="modificationdate">2011-08-10T15:11:03.000Z</fact>
      <folder name="pages">
        <fact name="page" number="1">SCIENCE &amp; ENVIRONMENT
4 August 2011 Last updated at 18:11 GMT
Mars: Nasa images show signs of flowing water

Striking new images from the mountains of Mars may be
the best evidence yet of flowing, liquid water, an essential
ingredient for life. The findings, reported today in the journal
Science, come from a joint US-Swiss study. ...
        </fact>
        <fact name="page" number="2">Salty water ...
```

Listing 4: DeepFS with facts and folder elements that establish a metadata hierarchy. Navigation into the file along the metadata hierarchy can be achieved once the database is mounted as a filesystem

Folders naturally appear as regular directories with the access rights of the original file inherited. The root folder of the metadata hierarchy is a hidden Unix directory (dot notation) and named after the corresponding file name suffixed with `.deepfs`. Facts show up as regular files and can be treated as such, *i.e.*, a write to a “file” in the metadata hierarchy translates into an update of its `<fact/>` element. To go through with the concept of a *deeper filesystem*, updates of facts in the DeepFS view propagate back into the original files. For that purpose we introduced the concept of **bi-directional transducers**.

Pushing metadata updates back into files. As indicated by the `inject(fileref, <update/>)` function, we, in contrast to existing metadata harvesters, allow users and applications to actually work on the extracted data. This means, while Desktop Search

Engines or application-specific indexes collect metadata in order to provide search functionality and lock away the metadata otherwise, we maintain a strong relationship between the XML view and the original data file. Whenever the file is updated, its redundant, externalized XML representation is updated as well. The same holds vice versa: if a metadata entry is updated by a database query those changes are propagated back into the original file.

Since the original, raw file is kept in a backing store, the homogeneous representation comes with the cost of storing data redundantly: The original metadata in the file and its counterpart in the XML representation.

2.4 Related Work

Various ideas have been proposed for including file contents into information systems.

One of the earliest attempts, the Semantic File System (SFS) [17], 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 filesystem projects, including Shore [11], HAC [19].

An interesting approach to bring XML and filesystems together was presented by IBM's XMLFS [3]. 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 a full XPath/X-Query support, and document storage was apparently limited to XML instances and to the existence of DTDs.

The visionary paper "From databases to dataspaces: a new abstraction for information management" [16] proposes dataspaces as a new data management abstraction. It led to various promising research efforts regarding the development of software platforms to facilitate a heterogeneous and distributed mix of personal information, such as Semex [10]. Approaches like this are far more prospective and target the development of so-called

DataSpace Support Platforms (DSSPs). These are supposed to meet the criteria defined in “Principles of dataspace systems” [29].

IBM’s Virtual XML Garden [55] and the draft of File System XML (FSX) [68] share the common idea to have a unified view over heterogeneous data sources. Since filesystems are structured hierarchically, they can easily be mapped to an XML structure as sketched in [68]. Together with the idea to let the filesystem immerse into the file [61], these provide the basis for the construction of our representations.

An extensive discussion focused on semantic technologies to the problem of personal information management is to be found in [56, Chapter 2].

2.5 In a Nutshell

`mkfs.basexfs(1)` takes an existing file hierarchy as input and creates a BaseX-FS database instance. This *bulk loading* operation serves well as a short summary of the points discussed so far. A depth-first preorder tree traversal, starting from the topmost directory, is performed in order to produces a unified representation of the file hierarchy.

While traversing the file hierarchy, each file is visited and analyzed. The following operations take place:

- Encountered files are represented as XML elements in the FSML database. They are augmented with operating system specific metadata attributes, such as file access time, file size, file protection mode and the like. This is done for all file types, incl. regular files, directories, links, etc. When, at a later date, mounting the database as filesystem, those attributes are used to obtain information about the file.
- File-type specific metadata (such as EXIF information for images, ID3 data for audio files, …) is stored in separate databases using an XML representation constructed by transducers.
- File-type specific content, such as the full-text of a PDF file, or an e-mail message, is included as well. The transducers are responsible for deciding what assets of interest should be represented.

- The original data file is copied to the binary backing store of BaseX and a unique reference is added to the corresponding file element (`<file bsid='uuid'>`).
- XML files are treated the same way. Metadata about the document is added to FSML, *i.e.*, statistics about the document (how many nodes, how many elements of a specific tagname, etc.). The document itself is *shredded* into the database. This holds for any well-formed XML instance. In the case of an incorrect, corrupt XML document, the document is put into the backing store and the FSML metadata entry contains information about the problem.

We created a suitable format to store and operate on filesystem data using a DBMS. While being straight-forward, it adds semantics and exposes formerly hidden contents of files with both, its structure and values. The approach allows to leverage all components (storage, indexes, query capabilities) of an XML-DBMS. XML processing languages, such as XPath and XQuery, allow for unprecedented search capabilities and flexibility on the data. Conventional approaches using full-text engines can only perform full-text queries, using proprietary syntax. With XQuery and its Full-Text extension we can easily combine full-text search criteria and queries based on values of any XML element or attribute.

In the next chapter, we will explore the integration of BaseX-FS instances to Unix operating systems in order to build filesystems on top of the unified XML representation. Since the database will be mounted as a conventional filesystem by the operating system kernel, access via the established (virtual) filesystem interface as well as database-enhanced access to the same data will be provided.

3 An XML Database as Filesystem

Given an instance of a BaseX-FS database, and given the database is connected to the operating system, metadata of files normally only accessible with dedicated tools, can now be represented as regular files and directories.

Applications completely unaware of the database can utilize the Unix filesystem interface to gain access to the uniformly stored file data in the DBMS. The database is mounted as a filesystem, and its data appear in the filesystem namespace. The database *becomes* the filesystem and the filesystem *is* the database.

Establishing a link between database management system and operating system kernel is crucial to achieve our ultimate goal: Provide both proven and stable access to the data leveraging filesystem techniques for database-unaware applications and enhanced, declarative access (including query support) to all stored files for database-aware applications.

Via the Unix filesystem interface we can provide:

- Conventional file I/O to all files in the FS/DB Server *(legacy interface)*
- Access to the formerly locked-in metadata of files via a proven and well-known interface *(metadata-aware filesystem)*
- Manipulation of database content in a BaseX-FS instance, using any tool capable of reading and writing files *(file I/O to database)*

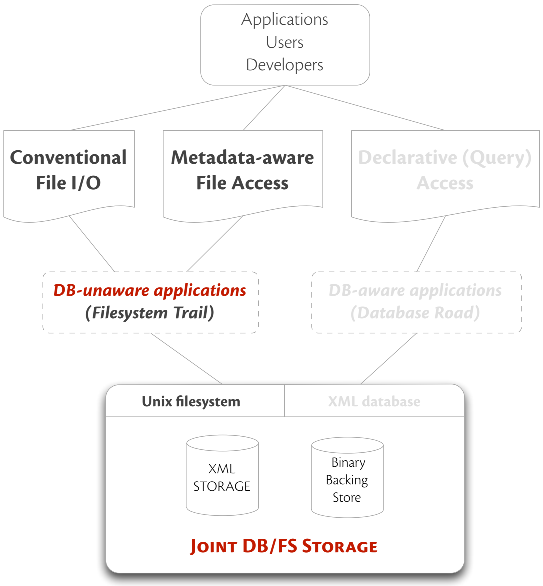

Applications
Users
Developers
Conventional File I/O
Metadata-aware File Access
Declarative (Query) Access
DB-unaware applications (Filesystem Trail)
DB-aware applications (Database Road)
Unix filesystem
XML database
XML STORAGE
Binary Backing Store
Joint DB/FS Storage

Figure 3.1: Ultimate goal: Database-enhanced *("Database Road")* and conventional access *("Filesystem Trail")* to filesystem data

3.1 On Filesystem Prototyping

Developing a filesystem from scratch is reported to be difficult and error-prone [71], [52]. Rajgarhia et al. from Stanford University summarize it as follows:

"Developing in-kernel file systems for Unix is a challenging task, due to a variety of reasons. This approach requires the programmer to understand and deal with complicated kernel code and data structures, making new code prone to bugs caused by programming errors. Moreover, there is a steep learning curve for doing kernel development due to the lack of facilities that are available to application programmers. For instance, the kernel code lacks memory protection, requires careful use of synchronization primitives, can be written only in C, and that too without being linked against the standard C library. Debugging kernel code is also tedious, and errors can require rebooting the system. Even a fully functional in-kernel file system has several disadvantages. Porting a file system written for a particular flavor of Unix to a different one can require significant changes in the design and implementation of the file system, even though the use of similar file system interfaces (such as the VFS layer) on several Unix-like systems makes the task somewhat easier. Besides, an in-kernel file system can be mounted only with superuser privileges. This can be a hindrance for file system development and usage on centrally administered machines, such as those in universities and corporations." [52]

At least two approaches strive to overcome this burden and provide frameworks suitable to rapidly prototype new filesystem concepts and ideas:

- Stackable Filesystems (paired with the FiST framework)
- Filesystem in USErspace (FUSE) approach

3.1.1 Stackable Filesystems

Stackable filesystems offer a way to add new functionality to existing filesystems without modifying kernel or existing filesystem code.

The basic idea of stacking can be summarized as follows: Most operating systems separate their filesystem code in two components, a native filesystem and a general-purpose layer, the Virtual File System (VFS). The VFS provides a uniform access mechanism to filesystems at a higher abstraction level and is unaware of the underlying filesystems' details. When filesystems are initialized in the kernel, a set of function pointers is installed in the VFS. The VFS, in turn, generically calls these pointer functions without knowing which specific filesystem the pointers represent.

For example, an `unlink` system call gets translated into a service routine `sys_unlink`. It invokes the VFS function (`vfs_unlink`), which in turn invokes the filesystem specific method by using its installed function pointer: `ext4_unlink` for ext4, `nfs_unlink` for NFS or the appropriate function for other filesystems.

This allows yet another filesystem to be inserted right between the existing VFS and base filesystem. Figure 3.2 shows such an inserted filesystem (CryptFS). It is called stackable, because it is stacked on top of another, the underlying filesystem.

If the stackable filesystem approach is applied, new functionality is layered on top of existing filesystems. Before the lower-level filesystem is called, a stackable filesystem can modify an operation and/or its arguments, and perform arbitrary operations before, after or instead of the underlying filesystems actions. Thereby, the underlying filesystem could

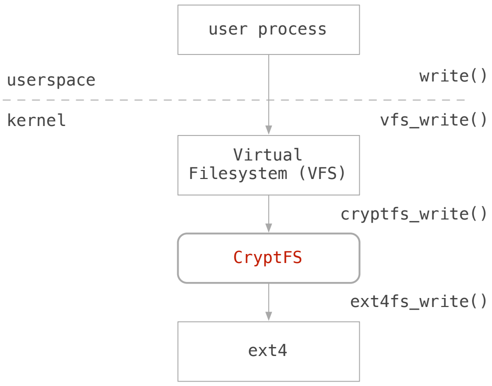


Figure 3.2: Information and execution flow in a stackable filesystem

be any other filesystem (Ext4, NFS, another stackable FS). The features implemented in the stackable filesystem are separate from the filesystem module, thus a stackable filesystem allows for portability to different environments.

The *File-System Translator (FiST)* [69] is a high-level language developed by Erez Zadok from Stony Brook University to describe stackable filesystem. If a FiST description is taken as input, a dedicated compiler can generate kernel filesystem modules for different platforms.

Erez Zadok and Jason Nieh explain in [70] why they consider using FiST a good choice to prototype new filesystems using the stackable filesystem approach:

“To ease the problems of developing and porting stackable file systems that perform well, we propose a high-level language to describe such file systems. There are three benefits to using a language:

1. Simplicity: A file system language can provide familiar higher-level primitives that simplify file system development. The language can also define suitable defaults automatically. These reduce the amount of code that developers need to write, and lessen their need for extensive knowledge of kernel internals, allowing even non-experts to develop file systems. 2. Portability: A language can describe file systems using an interface abstraction that is common to operating systems. The language compiler can bridge the gaps among different systems’ interfaces. From a single description of a file system, we could generate file system code for different platforms. This improves portability

considerably. At the same time, however, the language should allow developers to take advantage of system-specific features. 3. Specialization: A language allows developers to customize the file system to their needs. Instead of having one large and complex file system with many features that may be configured and turned on or off, the compiler can produce special-purpose file systems. This improves performance and memory footprint because specialized file systems include only necessary code."

The "Anti-Virus File System (AVFS)" [49] implemented by Zadok et al. perfectly conveys an idea of what can be achieved using stackable filesystems. Figure 3.3 shows the main components and interactions of AVFS.

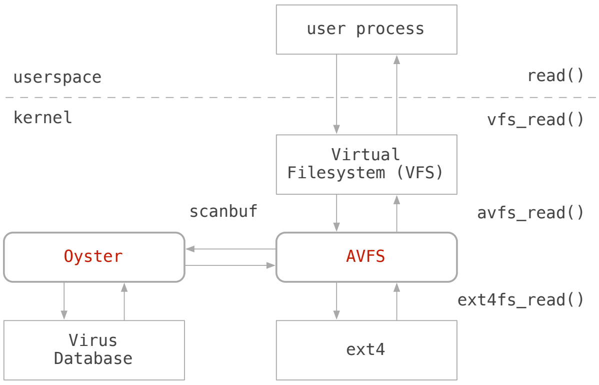


Figure 3.3: The Anti-Virus Stackable Filesystem [49]

AVFS is a stackable filesystem that provides protection against viruses. AVFS—as a stackable filesystem—is mounted over an existing filesystem, thus providing a bridge between VFS and the underlying filesystem. The VFS calls various AVFS operations, and AVFS in turn calls the corresponding operations of the underlying filesystem. AVFS performs virus scanning and state updates during these operations. Analogous to the explanation in the previous section, a user process that wants to do a `read` calls the VFS layer's `vfs_read` as usual (through, for example, glibc). The VFS calls AVFS (`avfs_read`) instead of directly addressing the "real" filesystem. AVFS performs scanning and calls an arbitrary filesystem (for example `ext3_read`). Throughout the process, a page is the fundamental data unit. Within AVFS, a modified and enhanced version of the open-source virus scan engine ClamAV (now called *Oyster*) is used. The filesystem has become "a page-based *on-access* virus scanner that scans in real time as opposed to conventional scanners that operate during `open` and `close` operations." [49]

3.1.2 Filesystem in Userspace

In recent years, the Filesystem in USErspace framework (FUSE), has entered the world of filesystem development.

FUSE is a framework for implementing filesystems outside the operating system kernel in a separate protection domain as a user process.

It was first implemented for and integrated into the Linux kernel [62]. There are reimplementations for the Mac OS X [58], FreeBSD, and NetBSD [40, 41] kernels.

The FUSE Framework

Userspace filesystems operate by connecting an in-kernel filesystem module to the virtual filesystem layer. This kernel component has a counterpart in userspace. The kernel part picks up VFS requests and transforms them to be suitable for delivery to userspace. After sending the request to userspace, the module waits for a response, interprets the result, and feeds it back to the caller in the kernel.

The FUSE user-level library interface closely resembles the in-kernel virtual filesystem interface. Function callbacks can be registered by the user-level implementations, which get executed once a corresponding request is issued by the OS kernel (Table 3.1).

A FUSE kernel module and the FUSE library communicate via a special file descriptor: `/dev/fuse`. This file can be opened multiple times, and the obtained file descriptor is passed to the mount system call, to match up the descriptor with the mounted filesystem.

Figure 3.4 on the next page depicts the FUSE framework and illustrates request handling of a filesystem call (*e.g.*, `stat(2)`) during the execution of an `ls(1)` command.

Filesystem operations

A FUSE implementation is a program listening on a socket `/dev/fuse` for operations to perform. The FUSE library (`libfuse`) transparently communicates with the socket

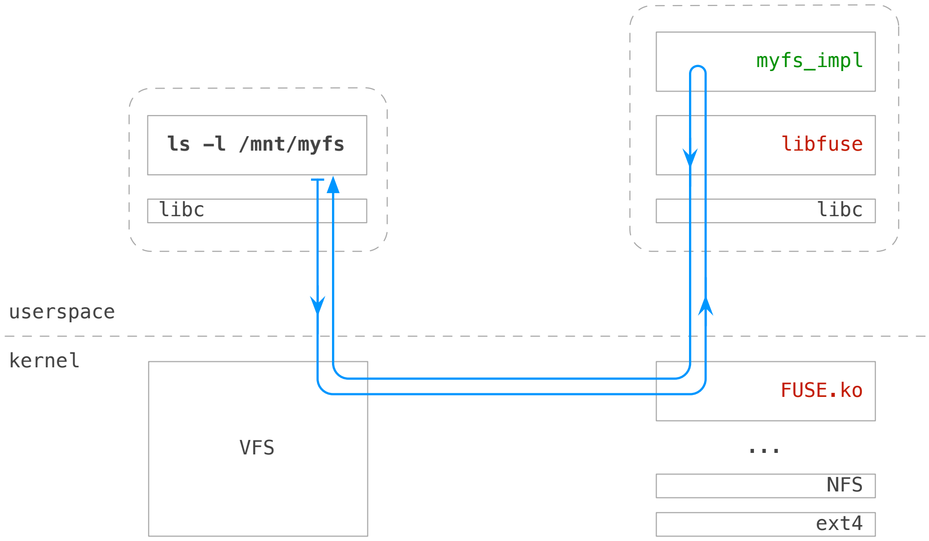


Figure 3.4: The FUSE framework. FUSE kernel module (.ko), `libfuse` user library and an implementation (`myfs_impl`). Request handling of a filesystem call (*e.g.*, `stat(2)`) during the execution of an `ls(1)` command. (Figure redrawn from the FUSE project documentation at `http://fuse.sourceforge.net/`)

and provides a *callback* mechanism to the filesystem developer. The callbacks *(request handlers)* are a set of functions for file operations. Table 3.1 provides an overview of the request handlers (file operations, callbacks) to give an impression of what can be achieved by a high-level FUSE implementation.

The FUSE APIs

Most FUSE request handlers work very similarly to the well known Unix filesystem operations and system calls. FUSE provides two basic APIs for developers to implement filesystems: A low-level and a high-level API. If the low-level API is used, almost all operations take an inode as first argument to identify the object the operation should work upon, while the high-level interface uses path names.

In principle, both interfaces could be used for our purpose. The low-level interface could connect inodes with database IDs of XML nodes. The high-level interface, however, is recommended by the FUSE developer community. It generally performs better because

Request handler	**Short Description**
`getattr()`	Get file attributes
`readlink()`	Read the target of a symbolic link
`mknod()`	Create a file node
`mkdir()`	Create a directory
`unlink()`	Remove a file
`rmdir()`	Remove a directory
`symlink()`	Create a symbolic link
`rename()`	Rename a file
`link()`	Create a hard link to a file
`chmod()`	Change the permission bits of a file
`chown()`	Change the owner and group of a file
`truncate()`	Change the size of a file
`open()`	File open operation
`read()`	Read data from an open file
`write()`	Write data to an open file
`statfs()`	Get file system statistics
`flush()`	Possibly flush cached data
`release()`	Release an open file
`fsync()`	Synchronize file contents
`setxattr()`	Set extended attributes
`getxattr()`	Get extended attributes
`listxattr()`	List extended attributes
`removexattr()`	Remove extended attributes
`opendir()`	Open directory
`readdir()`	Read directory
`releasedir()`	Release directory
`fsyncdir()`	Synchronize directory contents
`init()`	Initialize filesystem
`destroy()`	Clean up filesystem
`access()`	Check file access permissions
`create()`	Create and open a file
`ftruncate()`	Change the size of an open file
`fgetattr()`	Get attributes from an open file
`lock()`	Perform POSIX file locking operation
`utimens()`	Change the access and modification times of a file
`bmap()`	Map block index within file to block index within device
`ioctl()`	Manipulate the underlying device parameters of special files
`poll()`	Poll for IO readiness events

Table 3.1: FUSE request handlers a high-level implementation can choose to register for

several system calls are grouped to form a single filesystem call. In our case an additional, even more compelling reason is of importance: *path names* play an important role. And path-based navigation in filesystems and navigation in XML documents have quite a lot in common. For XML, path expressions are the core construct of XPath; they represent a fundamental part of XQuery. For filesystems, 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, which are syntactically separated by slashes (’/’): s_0/s_1/.../s_n. Each step $s_1 \dots s_n$ 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 δ_0/.../δ_n/f with directory names (δ_i) and a device/socket/file (f) is equivalent to ./δ_0/.../δ_n/f, where ’.’ denotes the current working directory.

Given the proposed mapping, filesystem path names (ρ_{fs}) naturally translate to path expressions (ρ_{xq}) as shown in 3.2.

path names	path expressions
.	self::fs:dir
..	parent::fs:dir
δ_0/···/δ_n	child::fs:dir[@fs:name="δ_0"]/…/child::fs:dir[@fs:name="δ_n"]
/···	fsml:fsRoot()/···
···/f	···/child::fs::*[@fs:name="f"]

Table 3.2: Filesystem path names to XPath/XQuery path expressions

Choosing FUSE as a prototyping framework

Given an efficient conversion from path names to XPath, its path-based API makes the FUSE framework a good choice to prototype our new filesystem concept. Apart from that, it is actively developed, maintained inside the Linux kernel tree and has a supporting community. Several FUSE-based filesystems (SSHFS [63], NTFS-3G [65], GlusterFS [18]) have already proven that FUSE is able to power non-trivial filesystems in real-world scenarios.

The obvious drawbacks stem from the performance overhead user-level filesystems have to encounter due to their nature. FUSE introduces two context switches for each filesystem call. There is a context switch from the user application that issued the system call to the FUSE user space library, and another one in the opposite direction. This and further aspects—splitting read and write requests into chunks and further memory copies—are discussed in [52, Section 3].

The stackable filesystem approach.

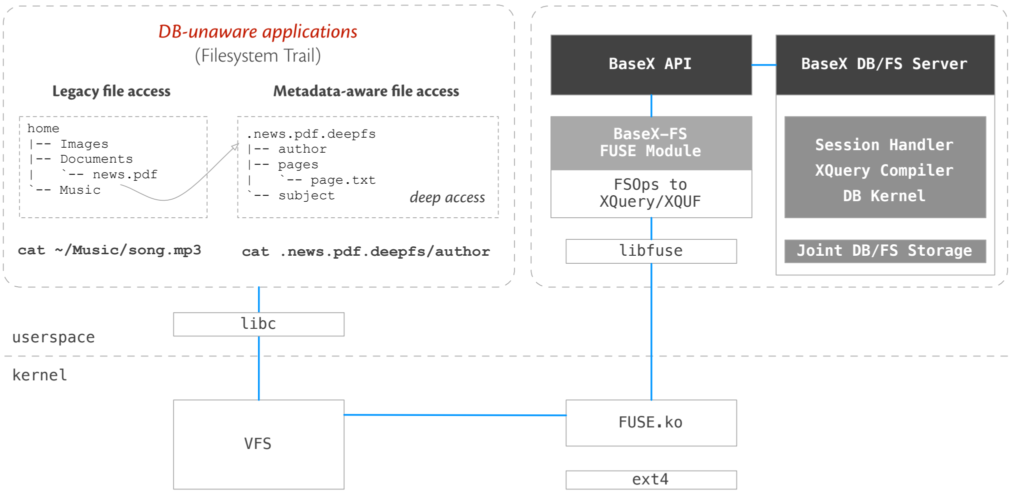
If we, at a later date, decide to push the filesystem further down the system stack, the stackable approach may become more important again, as it is closer to a "real" filesystem. Under the term "Breaking database technology out-of-the-box", one could think about isolating the relevant techniques to implement a storage layer that suits both, filesystem and database demands. A numbering scheme, inspired by the XPath accelerator (such as pre/dist/size), could be used as a basis for a filesystem implementation. Interlinking database node IDs with filesystem vnodes may be an approach worth further investigations. The general idea would be a storage that is tuned for filesystem access, but able to serve as a storage layer to the database as well. This could finally lead to an implementation of an in-kernel filesystem that is capable of interacting with a DBMS in userspace.

3.2 Mounting the Database as a Filesystem

3.2.1 System Architecture

The joint storage of BaseX-FS is capable of delivering any information needed to model and run a filesystem [32, 33]. The database server handles requests of multiple clients and manages concurrent read and write operations. FUSE, on the other hand, allows any implementation to organize the data the way it likes. The virtual filesystem operations initiated by applications are looped back into userspace and captured by the functions registered with the callback interface of the FUSE user-level library. BaseX-FS translates kernel requests into equivalent database queries. It comes in handy that the high-level interface of FUSE is path-based as those requests naturally translate into efficient XPath operations.

Figure 3.5 illustrates the system architecture and shows how the FUSE framework is used to connect the database system to the OS kernel.



```
home
|-- Images
|-- Documents
|    `-- news.pdf
`-- Music
```

```
.news.pdf.deepfs
|-- author
|-- pages
|    `-- page.txt
`-- subject
```

```
cat ~/Music/song.mp3
```

```
cat .news.pdf.deepfs/author
```

Figure 3.5: System architecture to mount the database as conventional filesystem into the operating system

For database-unaware applications, conventional as well as metadata-aware file access is achieved. The BaseX-FS FUSE code communicates as a client with a running server, leveraging the internal API of BaseX for optimal performance [31].

The filesystem, from this point of view, is yet another database client.

Apart from the necessary code to glue the database, FUSE and the kernel together, it turns out that the main logic is implemented in XQuery, feeding the kernel with data taken from the database.

The general workflow for all file operations is depicted in Figure 3.6 and can be summarized as follows:

- Kernel requests information about `path_name`
- BaseX-FS FUSE module dispatches kernel request to database (XQuery)
- Relevant information is retrieved from database (XML)
- Response is sent to kernel

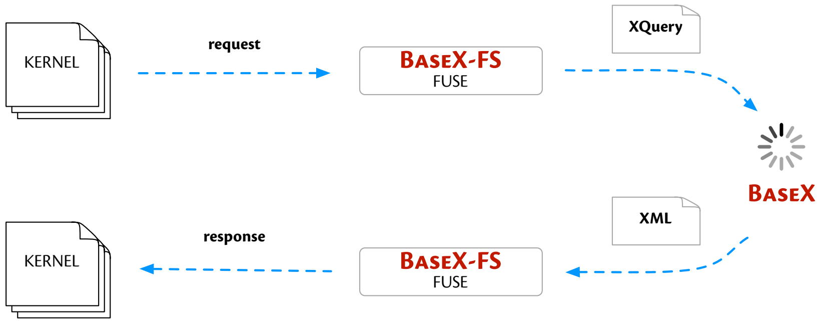


Figure 3.6: Kernel - FUSE - BaseX-FS communication. Logic in XQuery

3.2.2 Implementation Details

FUSE supports a variety of Unix systems. While `libfuse` already provides a good abstraction layer, it still has some system-specific elements. To further reduce native dependencies, we made the decision to split BaseX-FS into two parts: A thin, native low-level layer that is implemented against `libfuse` and a platform-independent high-level part interfacing with the database.

The low-level part compiles to a platform-specific shared library `libbasexfs` and is dynamically loaded by its high-level counterpart. Figure 3.7 illustrates the architecture. Since BaseX is a pure Java-based database system, the native `libbasexfs` uses Java Native Interface (JNI) for bi-directional communication with its platform-neutral part. In this way, the following three objectives could be realized:

1. The platform-specific code is reduced to a minimum
2. The "natural" Java-based BaseX API can be used for optimal performance
3. An arbitrary XML database can be connected to the native part, provided that it offers the system prerequisites described in Chapter 2.

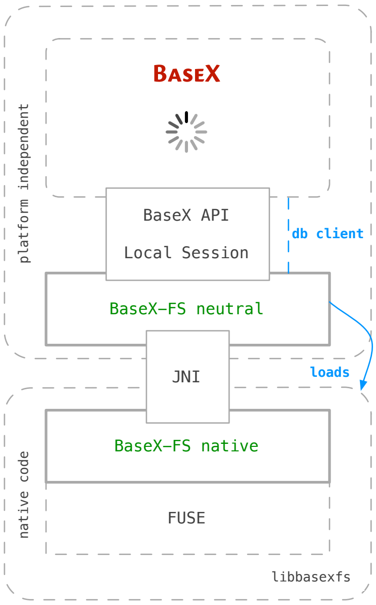


Figure 3.7: Implementing an XML database as filesystem in userspace. FUSE acts as a database client

Get file attributes (`stat ← fuse_getattr(path)`).

As an example for all other request handlers, we describe the implementation of one central filesystem operation. In a running filesystem, various attributes of files—so-called *file metadata*—are of central interest. That information about a file, including file timestamps, file ownership, and file permissions, is typically retrieved by the `stat(2)` family of system calls:

```
/* return information about named file */
int stat(const char *path, struct stat *buf);
/* return information about file refered to by an open file descriptor */
int fstat(int fd, struct stat *buf);
/* similar to stat(), except that if the named file is a symbolic link,
 * information about the link itself is returned */
int lstat(const char *path, struct stat *buf);
```

Listing 5: Retrieve file attributes. `stat(2)` family of system calls

To a large extent, the attributes are derived from the file inode and returned in a C stat structure, which contains the fields as depicted in Listing 6. The various data types used to type the fields are specified in the Single UNIX Specification [35]. Some of them are irrelevant for the implementation of a FUSE filesystem; for instance, `st_dev` and `st_blocksize` are ignored.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Listing 6: File attributes. Fields of a stat structure

The request handler of FUSE, called `getattr()`, is similar to the `stat(2)` system call. It obtains information on the file pointed to by path:

```
int (*getattr) (const char *path, struct stat *stat_buffer);
```

Listing 7: FUSE operation to get file attributes

In our implementation the `getattr/stat` filesystem call is sent through the JNI bridge to BaseX. The native path name is converted to a corresponding XPath expression, and finally evaluated by the database. An XML fragment is returned that contains the requested information:

```
<dir st_mode="040755" st_size="2278" st_uid="501" st_gid="20"  ...
     st_atime="1322218109" st_mtime="1322218081" st_ctime="1322218081"/>
```

Listing 8: File attributes are returned from the database as XML fragment. The values are filled into the stat buffer subsequently passed to the OS kernel

Given these details, the general chain of operations can now be stated more precisely:

- Kernel requests information about `path_name`
- VFS request is looped back into userspace and received by `libfuse`
- A corresponding request handler, the registered callback operation, is triggered in the native, low-level part of `libbasexfs`
- Native `libbasexfs` part interfaces with its platform-independant, high-level counterpart
- The kernel request is converted to an equivalent database operation (XQuery)
- Relevant information is retrieved from the database (XML)
- Result is transformed into applicable native data structures
- Response is sent back to the kernel

This communication path corresponds to Figure 3.7 on page 51 and illustrates that BaseX-FS, in a sense, acts as yet another database client.

3.2.3 Assessment

The substitution of path resolution with XPath resolution on a database instance is central to our approach. The Linux Kernel Documentation [44] describes path resolution as the finding of a *dentry* corresponding to a path name string, by performing a path walk. A *dentry* is an object with a string name (`d_name`), a pointer to an inode (`d_inode`), and a pointer to the parent dentry (`d_parent`). A filesystem is represented in memory using dentries and inodes. Typically, for every `open(2)`, `stat(2)` etc., the path name will be resolved.

Since it is a frequent operation for workloads like multiuser environments and web servers, it is important to optimize this code. Next, it is backed by existing kernel caching techniques. Runtime of the FUSE-based filesystem is also typically dominated by the costs of `getattr` (*i.e.*, `stat(2)`) requests.

To get an impression on how BaseX-FS performs in this regard, we compare it with two other FUSE-based implementations.

As a baseline, we chose a `nullfs` implementation using the native C API of FUSE. All FUSE request handlers are implemented by their corresponding system calls as exemplified in Listing 9. That way, an already existing file hierarchy can be exported via a FUSE mount point again.

```c
int fusefs_access(const char *path, int mask) /* FUSE request handler */
{
  int res;

  /* path is absolute from mountpoint */
  res = access(path, mask);  /* system call into ext4 */
  if (res == -1)
    return -errno;

  return 0;
}
```

Listing 9: Implementation of a `nullfs` request handler using system calls on the native filesystem

Consider, for instance, a mounted ext4 filesystem. We additionally mount `nullfs` on

/mnt/fuse. The actual implementation exports the complete file hierarchy, stored in the ext4 instance, now delivered and under the control of FUSE again. As such, the commands ls /home and ls /mnt/fuse/home produce the identical result: Both list the contents of directory /home, which is physically stored in the ext4 filesystem. The FUSE implementation acts as an intermediate layer, which uses system calls to retrieve data from the original ext4 source.

As an upper bound, we chose SSHFS [63] since it can be compared with our implementation: both filesystems act as clients to a server (sshd(8) and XML-DBMS, respectively). SSHFS is widely used as a substitute for NFS, serves well for that purpose, and can be seen as a filesystem of practical use in real-world scenarios. By connecting to the locally running SSH daemon[1] we serve an existing file hierarchy in the same way as nullfs. In practice, we can expect even slower results, since SSHFS is normally used to mount a remote resource via untrusted connections.

BaseX-FS uses a DeepFS representation of the existing file hierarchy and exports it back into the filesystem namespace.

As test data we used the home directory of a work station in our department. It contains a typical file hierarchy with 30.619 directories, 186.955 files, and 16.312 links. We performed a recursive traversal of the complete file hierarchy by the invocation of /usr/bin/time tree [mountpoint]/home >/dev/null. tree(1) is a recursive directory listing program that produces a depth-indented listing of files on all three instances. 233.886 files are served by all three implementations, the list of currently mounted filesystems is depicted in Listing 10, and the respective timings are listed in Table 3.3 on the next page.

```
$ mount | grep fuse
fusectl on /sys/fs/fuse/connections type fusectl (rw)
holu@localhost:/home/holu/ on /mnt/sshfs type fuse.sshfs (rw,nosuid,...
fusefs on /mnt/fusefs type fuse.fusefs (rw,nosuid,nodev)
deepfs on /mnt/bxfs type fuse.deepfs (rw,nosuid,nodev)
```

Listing 10: List of currently mounted FUSE implementations

[1] $ sshfs user@localhost:/home /mnt/sshfs

Filesystem	**Timings (Average of 1000 runs)**	**Difference**
`nullfs`	28,82ms	100%
`basexfs`	44,52ms	∼ 154%
`sshfs`	110,11ms	∼ 382%

Table 3.3: Timings of Fuse-based filesystems performing a recursive directory listing

We argue that our implementation is an adequate proof-of-concept and serves well as a domain-specific filesystem. The argumentation is along the lines of using an encrypted filesystem. The administrator/user decides and accepts to pay an acceptable performance penalty for a more advanced, respectively otherwise missing, functionality.

Metadata-aware file access (*"deep access"*) surely adds new functionality to filesystems. It is quite convenient and more efficient to directly `find(1)` "artist" files and `grep(1)` for "Bob Dylan" instead of writing a shell script to find all .mp3 files, pipe them through an ID3 decoder and analyze the output.

However, while it adds new functionality, it is still just an improvement of a legacy concept. What we really want to achieve is to empower developers to implement database-aware applications and to use declarative programming on the filesystem data (*"Database Road"*).

In the following we will give an insight on how database-aware applications can be developed based on our architecture. Chapter 4 will afterwards introduce a framework to enable developers to exploit theses concepts with ease and at a higher abstraction level. And Chapter 5 will finally present how to implement a non-trivial application, *i.e.*, an expert retrieval system on top of BaseX-FS and the proposed application framework.

3.3 Database-aware Applications

Nowadays, nearly every application provides its own indexing and retrieval component for the domain it is responsible for. Even so, the metadata in question has often already been collected by the desktop search engines, it is not reused. Specialized applications create their own, proprietary index structures: audio players index audio and video data in order to display and arrange the media library of a user, and to enable efficient searches across the media's metadata. The same holds for personal information management tools: mail clients index e-mails and their attachments, browsers index bookmarks and browsing history, photo editing and management software keeps track on image files and the like.

3.3.1 XQuery your Filesystem

With an FSML database view of the filesystem, numerous inquiries can be supported that can hardly be achieved with available tools. It can be used by application developers to reduce code complexity and redundant implementation of similar functionality. In our scenario, metadata from different sources is exposed in a unified metadata namespace, and a common interface to access the formerly heterogeneous data is provided. Application developers can profit from:

- a unified view on the filesystem's content and metadata
- a standardized and generic way to retrieve the metadata and work with it
- a clear description of what data is available and in what form (transducers provide a schema definition of the data they extract)
- valid data that can be expected

Retrieval capabilities are not restricted to application-defined communication paths (such as the often interconnected e-mail, calendar and address book applications), but can include any data stored in the filesystem.

Transducers externalize information useful for analysis and retrieval that has formerly been hidden and encapsulated in the filesystem. Content and structure can be queried together. The extracted data is presented in a homogeneous manner.

As such, an FSML database view provides a solid foundation to work on filesystem data with XPath and XQuery, as illustrated by the following examples:

- Compute disk usage of file hierarchy with XPath:

  ```
  sum(//file/@size)
  ```

- Search for a file name:

  ```
  //file[@name contains  text  "Rolling Stone"]
  ```

- Find all e-mail files in which the e-mail body contains ’A’ and ’B’ within a maximum distance of five words.

  ```
  //file[@mime  =   ’ x-mail ’ ] [.//body
        contains  text  { ’ A ’ , ’ B ’ } distance at most 5 words ]
  ```

Typically, an XQuery module is used that provides utility functions to work on FSML instances in the `deepfs` namespace. For a first impression, we simulate Unix command-line requests. The task is to compute the disk usage in bytes of images with a resolution of 240dpi and an aperture of 9. We chain XQuery functions on the analogy of Unix pipes.

```
deepfs:du(
  deepfs:find(
      deepfs:find( //file, "XResolution", "240" )
    , "FNumber", "9.0"
  )
)
```

Listing 11: Chaining XQuery functions on the analogy of Unix pipes

The following two examples—taken from the domain of personal information management—try to give some insight in how both XQuery users and application developers can benefit from a system-wide metadata repository.

Think, for instance, about the extension of an e-mail application. The task is to create reports about the e-mail correspondence and to display specific details as it is offered by various e-mail reporting and analysis tools. The data to be displayed can be retrieved by a query over the e-mails stored in the filesystem.

Show my most frequent e-mail contacts

“Who is sending the most e-mails?”, such a report can be constructed by the following query:

```
1  (: find the most chatty senders throughout all your mails :)
2  for $email in deepfs:mime( //file, "x-mail" )
3  let $sender := deepfs:value( $email, "From:" )
4  group by $sender
5  order by count($email) descending
6  return
7    <collection>{
8     attribute from   { $sender },
9     attribute size   { count($email) },
10    element   latest { $email[ count($email) ] }
11   }</collection>
```

Listing 12: XQuery: Who is sending the most e-mails?

For each e-mail found in the database/filesystem, a list of sender e-mail addresses is maintained `(lines 1,2)`. The e-mails are grouped by their sender, such that each group contains all e-mails per unique sender `(3)`. Afterwards, the groups are sorted according to the total number of e-mails contained in each group `(4)`. Finally, a `<collection />`-element is returned for each group. It contains information on which sender the mails in this group originate from, how many mails are in that group, and the most recent mail that belongs to that group `(6-10)`.

The result is returned as an XML fragment and can instantly be used as a data source for a graphical user interface (GUI) widget.

Show all e-mails from people that are not listed in my address book

Another report may be generated to help cleaning up overfilling inboxes. In order to provide a pre-selection of e-mails, the report compares the address book and e-mails to *show all e-mails from people that are not listed in the address book*. Those e-mails may be good candidates for deletion or to add senders to the personal contact list.

```
1 let $vCards  := deepfs:mime( //file, "vcard" ),
2     $friends := distinct-values( deepfs:value( $vCards, "email" ) )
3 (: now find all mails from unknown senders :)
4 return
5  for $mail in deepfs:mime( //file, "x-mail" )
6  let $from := deepfs:value( $mail, "From:")
7  where not( some $person in $friends satisfies
8    ( contains( $from, $person ) ) )
9  return $mail
```

Listing 13: XQuery: Show all e-mails from people not listed in my address book

First, we generate a list of all known e-mail addresses by storing the distinct values for the `email`-key of all vCards available in the database (`1-3`). Then we iterate through all e-mails (`6`) and store the sender (`7`). All e-mails are returned that are sent by an address that is not in the list of known people (`8-10`).

Both examples show how to leverage the uniform representation of files in XML. Different file formats (e-mails, vCards, …) can be addressed in a declarative way, and their content can be uniformly accessed. The possibility of consistently working on the complete filesystem content provides the unprecedented opportunity to interrogate and combine any number of files in a single query.

3.3.2 Visual Access to Large Filesystem Data

Based on the database/filesystem, and with the power of XQuery, it is not only simple to construct a desktop search engine, but to even go a step further and implement a *desktop query engine*. When using common desktop search engines, users expect immediate responses and visual presentation of documents. Since our research is supported by the German Research Council (DFG) Research Training Group GK-1042 “Explorative Analysis and Visualization of Large Information Spaces” we happily meet those demands and visually present result sets which allow to instantly access the files stored in the database. In a selection of screenshots, we want to illustrate how visual access to filesystem data is implemented in the graphical user interface of BaseX.

From the beginning, BaseX offered an innovative, interactive graphical frontend to visualize large XML data instances. Apart from the conventional client/server mode and its command line interface, a graphical frontend, which allows the visual exploration of XML data, has been part of BaseX since version 1.0 [27].

In an early stage of development, it was realized that the storage structure and parts of the query algorithms perfectly match for the construction of hierarchic visualizations [27]. The TreeMap, a space-filling visualization for hierarchic information [39], is a good example to demonstrate the close relationship.

A simplified algorithm is shown in Figure 3.8:

```
calcMap(nodes, rect):

if #nodes = 0 or rect < fixed minimum:
   return

if #nodes = 1:
   paint(nodes[0], rectangle)
   newRect := rect + fixed offset
   children := xpath(nodes, "child::node()")
   calcMap(children, newRect)

else:
   descendants := xpath(nodes, "//node()")
   split := descendants.length / 2
   orientation := horizontal/vertical,
     dependent on rect.width and rect.height

   firstRect := left/upper half of rect,
     dependent on split and orientation
   calcMap(nodes[0..split], firstRect)

   secondRect := right/lower half of rect,
     dependent on split and orientation
   calcMap(nodes[split..#nodes], secondRect)
```

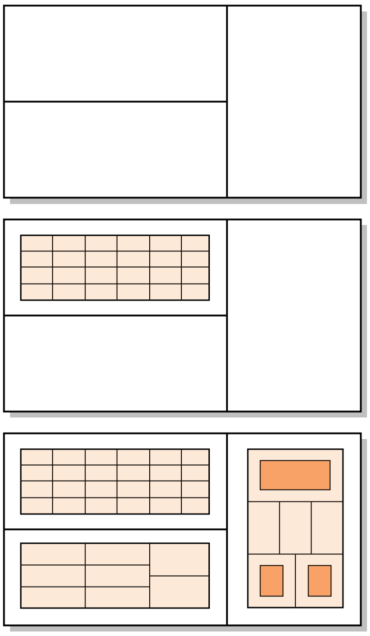
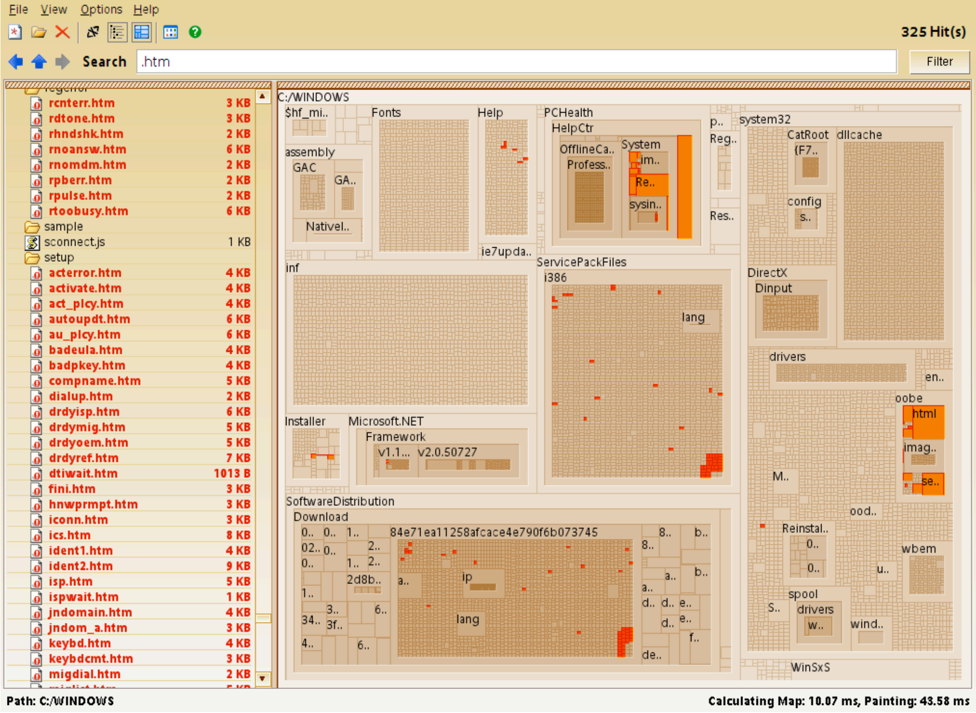


Figure 3.8: TreeMap algorithm (left), recursive visualization (right)

The algorithm is initially invoked with the root node of the database and the visible rectangle bounds as arguments. If the node set contains only one node (as is the case after the first call), the current rectangle is painted to the panel. Next, the algorithm is recursively called with the child nodes that have been retrieved via an XPath expression. If several nodes are found in the node set, the current rectangle is split in the middle, a new orientation (horizontal vs. vertical) is chosen for the child nodes, and the algorithm is called with the first and second node subset as arguments. If the node set is empty or if the calculated area is too small to be painted, the recursive traversal is stopped.

The two XPath expressions in the example code serve as an actual link between the visualization and the database: data access is realized via simple numeric node references and node sets, and extra data structures can be largely avoided.

Figure 3.9 shows a TreeMap as one of two *views* that show results for a specific *search* query. The conventional search slot is used to scan an FSML instance for HTML files. The entered search term is internally converted to a full-text query. Matches are highlighted and visualized. On the left hand side, a generic tree view is shown, as known from website navigation and file explorers. On the right hand side, a TreeMap is depicting the structure of the file hierarchy in its space-filling visualization.



325 Hit(s)

Search: .htm

rcnterr.htm	3 KB
rdtone.htm	3 KB
rhndshk.htm	2 KB
rnoansw.htm	6 KB
rnomdm.htm	2 KB
rpberr.htm	2 KB
rpulse.htm	2 KB
rtoobusy.htm	6 KB
sample	
sconnect.js	1 KB
setup	
acterror.htm	4 KB
activate.htm	4 KB
act_plcy.htm	4 KB
autoupdt.htm	6 KB
au_plcy.htm	6 KB
badeula.htm	4 KB
badpkey.htm	4 KB
compname.htm	5 KB
dialup.htm	2 KB
drdyisp.htm	6 KB
drdymig.htm	5 KB
drdyoem.htm	5 KB
drdyref.htm	7 KB
dtiwait.htm	1013 B
fini.htm	3 KB
hnwprmpt.htm	3 KB
iconn.htm	3 KB
ics.htm	8 KB
ident1.htm	4 KB
ident2.htm	9 KB
isp.htm	5 KB
ispwait.htm	1 KB
jndomain.htm	4 KB
jndom_a.htm	3 KB
keybd.htm	4 KB
keybdcmt.htm	3 KB
migdial.htm	2 KB

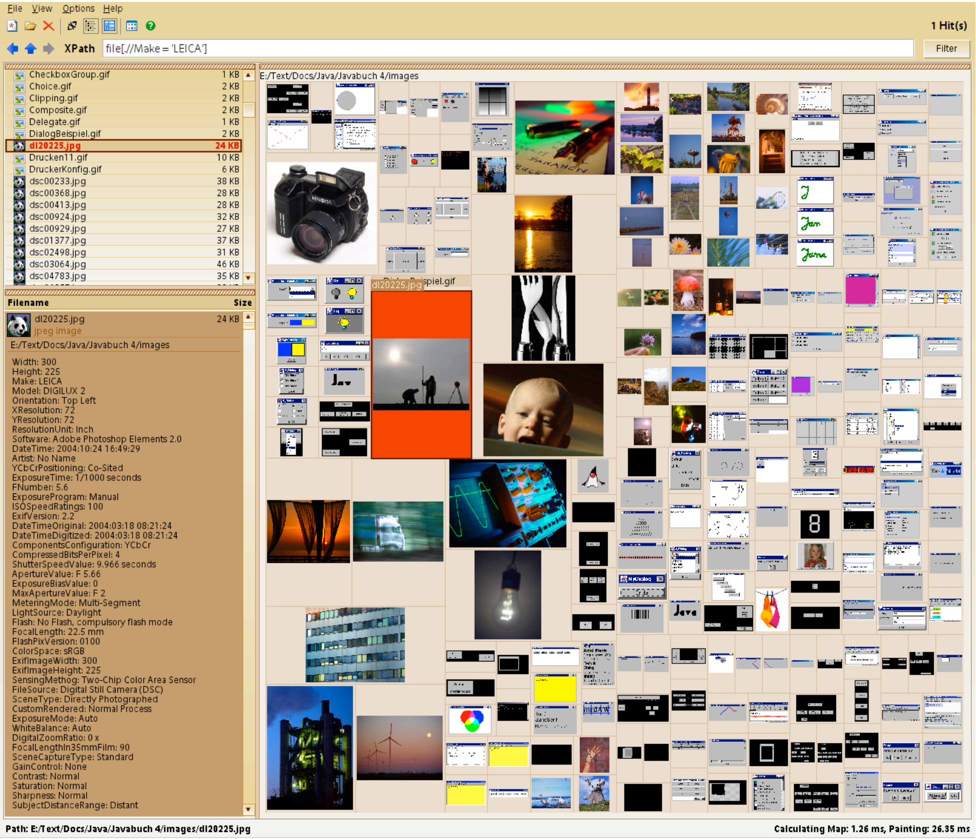
Path: C:/WINDOWS

Calculating Map: 10.07 ms, Painting: 43.58 ms

Figure 3.9: Simple Search Mode. Searching for `.htm` files. Results are shown and highlighted in both views, a generic tree view and the space-filling tree map

The GUI architecture provides various *interactive* visualizations, so-called views, to be freely combined in a single window. All views are synchronized and work on the same data instance. Users get instant result feedback while they visually explore, analyze or browse the filesystem in order to get a quick overview or to dive into the details.

In Figure 3.10, an XPath expression is used to search through image files. Images are loaded as thumbnails into the TreeMap, and their metadata is displayed in a separate view on the lower left side.



XPath: file[.//Make = 'LEICA']

CheckboxGroup.gif	1 KB
Choice.gif	2 KB
Clipping.gif	2 KB
Composite.gif	2 KB
Delegate.gif	1 KB
DialogBeispiel.gif	2 KB
dl20225.jpg	24 KB
Drucken11.gif	10 KB
DruckerKonfig.gif	6 KB
dsc00233.jpg	38 KB
dsc00368.jpg	28 KB
dsc00413.jpg	28 KB
dsc00924.jpg	32 KB
dsc00929.jpg	27 KB
dsc01377.jpg	37 KB
dsc02498.jpg	31 KB
dsc03064.jpg	46 KB
dsc04783.jpg	35 KB

E:/Text/Docs/Java/Javabuch 4/images

Filename	Size
dl20225.jpg jpeg image	24 KB

E:/Text/Docs/Java/Javabuch 4/images

Width: 300
Height: 225
Make: LEICA
Model: DIGILUX 2
Orientation: Top Left
XResolution: 72
YResolution: 72
ResolutionUnit: Inch
Software: Adobe Photoshop Elements 2.0
DateTime: 2004:10:24 16:49:29
Artist: No Name
YCbCrPositioning: Co-Sited
ExposureTime: 1/1000 seconds
FNumber: 5.6
ExposureProgram: Manual
ISOSpeedRatings: 100
ExifVersion: 2.2
DateTimeOriginal: 2004:03:18 08:21:24
DateTimeDigitized: 2004:03:18 08:21:24
ComponentsConfiguration: YCbCr
CompressedBitsPerPixel: 4
ShutterSpeedValue: 9.966 seconds
ApertureValue: F 5.66
ExposureBiasValue: 0
MaxApertureValue: F 2
MeteringMode: Multi-Segment
LightSource: Daylight
Flash: No Flash, compulsory flash mode
FocalLength: 22.5 mm
FlashPixVersion: 0100
ColorSpace: sRGB
ExifImageWidth: 300
ExifImageHeight: 225
SensingMethog: Two-Chip Color Area Sensor
FileSource: Digital Still Camera (DSC)
SceneType: Directly Photographed
CustomRendered: Normal Process
ExposureMode: Auto
WhiteBalance: Auto
DigitalZoomRatio: 0 x
FocalLengthIn35mmFilm: 90
SceneCaptureType: Standard
GainControl: None
Contrast: Normal
Saturation: Normal
Sharpness: Normal
SubjectDistanceRange: Distant

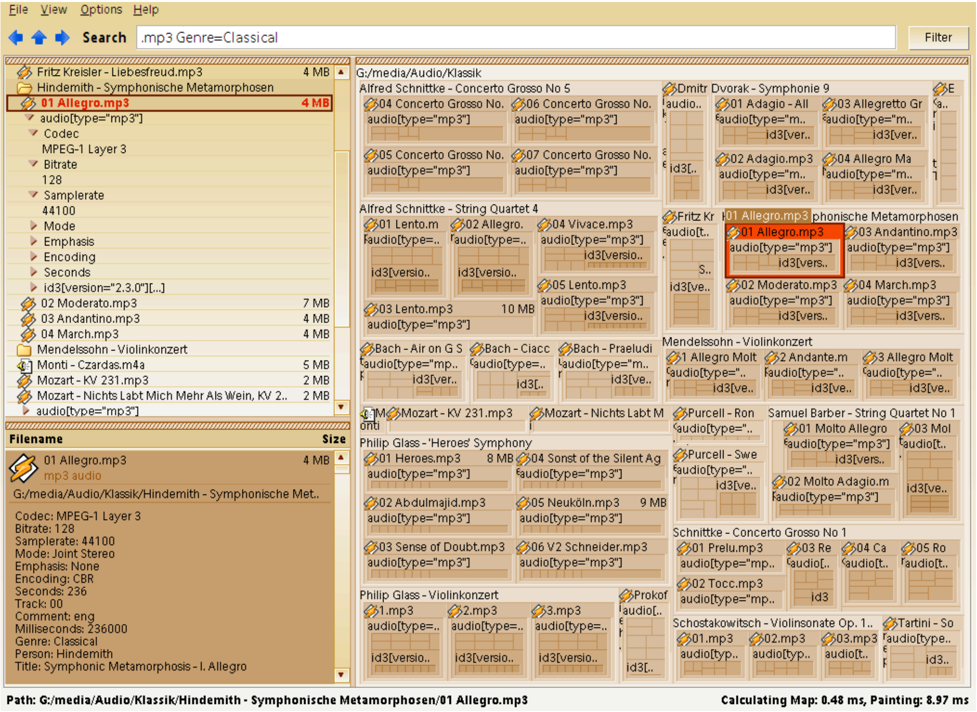
Path: E:/Text/Docs/Java/Javabuch 4/images/dl20225.jpg

Calculating Map: 1.26 ms, Painting: 26.35 ms

Figure 3.10: Using XPath to search through image files

The TreeMap makes extensive use of semantic zooming. In our filesystem context, semantic zoom is to be looked upon as a form of details-on-demand technique, which lets users explore different amounts of detail in one view by zooming in and out. The visualization is closely related with the description of data in the filesystem markup language, as the *"zoom in"* operation correlates with *"navigating into"* a file.

Figures 3.11 illustrates the concept and shows the extension of the file hierarchy along the file's inherent structure. The metadata hierarchy is shown for an audio file, and both views allow the user to navigate (zoom) into the file.


File
View
Options
Help
Search
.mp3 Genre=Classical
Filter
Fritz Kreisler - Liebesfreud.mp3 4 MB
Hindemith - Symphonische Metamorphosen
01 Allegro.mp3 4 MB
audio[type="mp3"]
Codec
MPEG-1 Layer 3
Bitrate
128
Samplerate
44100
Mode
Emphasis
Encoding
Seconds
id3[version="2.3.0"][...]
02 Moderato.mp3 7 MB
03 Andantino.mp3 4 MB
04 March.mp3 4 MB
Mendelssohn - Violinkonzert
Monti - Czardas.m4a 5 MB
Mozart - KV 231.mp3 2 MB
Mozart - Nichts Labt Mich Mehr Als Wein, KV 2.. 2 MB
audio[type="mp3"]
Filename
Size
01 Allegro.mp3 4 MB
mp3 audio
G:/media/Audio/Klassik/Hindemith - Symphonische Met..
G:/media/Audio/Klassik
Alfred Schnittke - Concerto Grosso No 5
Dmitr
Dvorak - Symphonie 9
Alfred Schnittke - String Quartet 4
Hindemith - Symphonische Metamorphosen
Mendelssohn - Violinkonzert
Philip Glass - 'Heroes' Symphony
Samuel Barber - String Quartet No 1
Schnittke - Concerto Grosso No 1
Philip Glass - Violinkonzert
Schostakowitsch - Violinsonate Op. 1..
Tartini - So
Prokof
Purcell - Ron
Purcell - Swe
Bach - Air on G S
Bach - Ciacc
Bach - Praeludi
Mozart - KV 231.mp3
Mozart - Nichts Labt M
audio[type="mp3"]
id3[versio..

Codec: MPEG-1 Layer 3
Bitrate: 128
Samplerate: 44100
Mode: Joint Stereo
Emphasis: None
Encoding: CBR
Seconds: 236
Track: 00
Comment: eng
Milliseconds: 236000
Genre: Classical
Person: Hindemith
Title: Symphonic Metamorphosis - I. Allegro

Path: G:/media/Audio/Klassik/Hindemith - Symphonische Metamorphosen/01 Allegro.mp3

Calculating Map: 0.48 ms, Painting: 8.97 ms

Figure 3.11: Zooming into the file. The continuation of the file hierarchy along the file's inherent structure ("Semantic Zoom")

With the tight coupling of different query strategies (keyword-based, full-fledged XPath/XQuery) and the result presentation, we try to establish a query cycle, which allows to refine, *i.e.*, filter/select/modify the (intermediate) results in a user-system feedback loop. This allows to start a search with a simple keyword-based query, to browse the resulting items, and to refine a selected context set by issuing an XQuery.

3.4 Considerations

Tailor-made BaseX GUIs are a suitable way to make the database accessible to end users. They allow for special-purpose visualizations of data and achieve very good performance results when large databases are to be processed. The approach, however, provides only limited means to application developers in terms of implementing their own XML and XQuery-driven applications. The visualizations to graphically work on BaseX-FS instances have two significant drawbacks:

- The GUI directly operates on the underlying storage and is not capable of handling multiple users simultaneously
- The visualizations access the low-level APIs of BaseX. On account of this, their use as a public interface can not be recommended

In general, leveraging BaseX as core infrastructure to implement graphical applications on top of the current architecture would force developers to fiddle with internal Java-based APIs. Profound knowledge of the BaseX code base would be a mandatory prerequisite. Given a steep learning curve, accompanied by the dynamic development of BaseX and the considerable maintenance overhead involved, developers are surely not too keen to opt in. Consequently, it can be stated that taking the current implementations of the BaseX GUIs as a blueprint for customized application development application development is not advisable.

So, what is it that we are looking for? Once we became aware of the benefits stemming from the declarative access on filesystem data, and once we implemented the first application prototypes as views inside the BaseX GUI, we thought about a more high-level and generic way to develop applications based on BaseX-FS instances (or XML in general).

We strive to provide a framework that simplifies the implementation of graphical applications with BaseX as a core component. To use of BaseX as a general-purpose application framework, we have been investigating platform independent ways to benefit from the BaseX storage and processing capabilities without resorting to internal APIs.

The conceptually most tempting idea was to use XML technologies on all layers of a three-tier architecture. The benefits are obvious:

- The storage layer of BaseX provides persistence per se, and XQuery allows us to perform declarative and domain-specific queries.
- When it comes to business logic, the XML ecosystem unites concepts of functional, general-purpose programming languages (XQuery and its extensions XQUP, XQFT, XQSE) with transformations (XSLT). Arbitrary functionality can be implemented via XQuery modules, ready to be used by developers.
- For the presentational layer, we decided to choose (X)HTML, while developers are still free to use every other flavor of XML.

With the proposed architecture, database-aware applications will be enabled to leverage FSML with ease and new applications can be build on the solid foundation of BaseX-FS and a modern XML database management system.

The following chapter will introduce BaseX-Web, an XQuery application framework, as a further extension to the BaseX database management system. It will allow database-supported application development in a clean XML technology stack.

4 XQuery Application Framework

XML, XQuery and XHTML are an ideal match to present and process information resources in a platform neutral way.

In order to facilitate application development in BaseX, we built a service infrastructure to implement and deploy XQuery-based applications.

Since data is natively stored in XML, XQuery is the standard processing language that can be applied to produce XHTML output with no additional effort. That way, XML technology is applied on all three layers of a classical three-tier-architecture. The persistence layer is provided by a native XML database management system, business logic is implemented in XQuery, and the presentation layer is primarily driven by XHTML. As such, a single data model is used throughout the architecture and no conversions have to be applied between the layers. By using this “unified technology stack”, problems such as the object-relational impedance mismatch [37] can be avoided at all. Figure 4.1 illustrates the general idea.

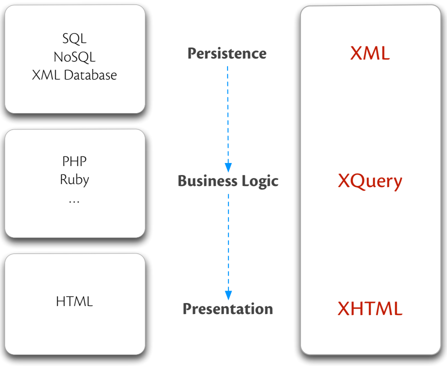


Figure 4.1: Uniform Application Stack: XML technology on all three tiers of a system architecture

4.1 Maturity of Web Applications

Web-based result presentation became the predominant way in both consumer and expert retrieval systems. The web is full of hosted solutions that provide almost any information need that might possibly arise. A plethora of machine processable resources exist, such as RSS feeds, web services via SOAP, or REST. In the same way, numerous websites and applications are available which can be used with any browser—most prominently Google Search, serving as an everyday retrieval tool at a large scale. Thanks to the tremendous success of Google,

(a) we are used to start the browser, whenever we search for something, and
(b) we are convinced that it is both feasible and convenient to use web applications as alternatives to desktop applications.

Applications such as GoogleMail have the look and feel of desktop applications; in terms of functionality, they even outperform most of their native counterparts, due to the tight integration of services.

The rise of AJAX [38] leads to the type of web applications we are used to today. Coming from a past in which JavaScript was mainly used to swap images on mouse-over actions or validate forms before sending them, applications that have been developed for a single target architecture and operating system can now be run on virtually any device and OS that is connected to the internet.

Most notably, two companies tried to make web applications first class citizens—on their mobile devices—early on, Palm with webOS and Apple with their first iPhone. The latter even lacked support for a *native* development kit for over a year.

Obviously, the companies' motivation was to provide developers with tools to build and deliver their applications directly to their customers' handsets, without having to worry about device fragmentation or other low-level specifics.

Ranging from mail clients and spreadsheets to fully blown multimedia applications like YouTube or Google Maps, it seems that there are effectively no limits on what can be achieved inside a user's browser today. The gap between a user's browser and her desktop will vanish even more, as HTML5 devotes a whole lot of its new functionality to actually mimic the behavior of native applications.

Some of the key features include:

Offline Support gives HTML5 applications a dedicated storage—provided by the runtime environment—to locally cache their data. In addition, developers may subscribe to certain events, check for online connectivity, and perform synchronization.

File Access provides developers with access to a user's file system, such that HTML5 applications may store and retrieve files locally. This represents a major advantage over the status quo, which was often based on the proprietary FLASH format.

Graphics enables developers to make use of 3D acceleration hardware with very low effort.

In this respect, it is quite impressive to see how far open-source frameworks, such as Cappuccino[1] or SproutCore[2], have pushed the state-of-the-art: these packages devote their existence *"to build desktop-caliber applications that run in a web browser"* in order to give *"a native experience—on the web"*. Figure 4.2 on the following page shows exemplary screenshots, illustrating desktop-caliber web applications with a native look and feel. On recent machines, these applications run smoothly and seamlessly, and it is quite easy to forget that they are running in a browser and not directly on the desktop. SproutCore, for example, is backed by Apple, who in turn implemented their latest `icloud.com` browser frontend using the open-source framework.

4.2 Related Work

The idea of using server-side XQuery implementations to foster application development is not new and has been around for quite some time in competing open-source implementations such as the Sausalito project[3], which claims to bring XQuery to the cloud, or eXist-db[4], one of the early native XML database systems. Both implementations use different approaches and focuses to achieve this goal.

[1] `http://cappuccino.org/`

[2] `http://www.sproutcore.com/`

[3] `http://www.28msec.com`

[4] `http://exist.sourceforge.net/`

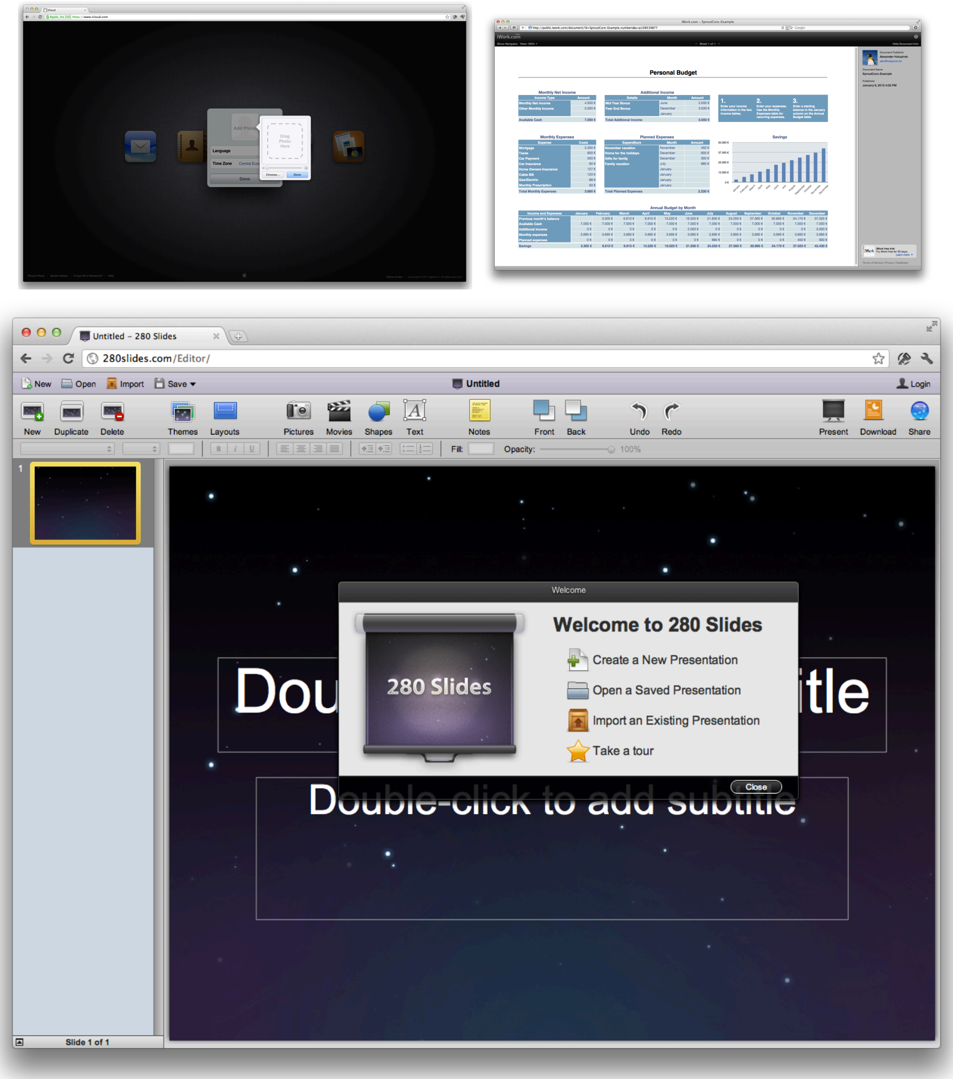


Figure 4.2: Examples of desktop-caliber web applications. Above, left hand: the login screen of `http://icloud.com`, which is broadly similar to native login and configuration scenarios on the OS X. Right hand: an `http://iwork.com` frontend reproducing its native counterpart, the Numbers office application (both using the SproutCore framework). Below: a presentation application built with the Cappuccino framework running on `http://280slides.com/`

They mainly differ in two aspects: while the latter is more database-centric, the former is about XQuery-powered application logic (exposing data-centric services through a RESTful interface and delegating storage considerations to backend systems).

4.2.1 Sausalito – XQuery in the Cloud

Kaufmann and Kossmann were the first to describe the development of enterprise web applications based on the W3C technology stack, as depicted in Figure 4.1 on page 67.

They concluded "that the W3C family of standards is very well suited for this task and has important advantages over the state-of-the-art (*e.g.*, J2EE, .Net, or PHP). Most importantly, using XQuery and W3C standards only ensures a uniform technology stack and avoids the technology jungle of mixing different technologies and data models. As a result, the application architecture becomes more flexible, simpler, and potentially more efficient." [42]

A commercial offspring from this research is Sausalito, "a suite of tools that allow to write, test, and deploy full-fledged web-based applications, entirely written in XQuery" [1].

The company behind, 28msec, directly follows their line of argument. They agree that "XQuery has an extremely powerful support for database queries, scripting, and full-text search. By using a single programming on all tiers, Sausalito is collapsing web servers, application servers, and databases into a single stack." [1]

As the project further documents, XQuery is used as a language for

- writing the application code
- defining the data and access structures (*i.e.*, collections and indexes)
- accessing the data.

In a Sausalito project, all XQuery code is structured around XQuery modules. Handler modules contain functions that are directly exposed using REST. Their main task is to dispatch and orchestrate calls to library or external modules. Library modules provide a rich set of functions, which help to implement application logic. They are not publicly exposed and thus can not be called from the outside. External modules contain additional XQuery modules, provided by any third party. Once a Sausalito project is set up it is, generally, deployed on Amazon Web Services.

As such, Sausalito can be seen as a solution for building RESTful services with XQuery running on cloud infrastructure. The framework uses sophisticated distributed commit protocols [7] and exposes handler modules through HTTP. Its functions can be invoked with any HTTP client, first and foremost a web browser. These functions constitute the REST-based interface of a Sausalito application and trigger access to data stored in a storage facility, i.e, an XML database, a JSON store, etc. Application logic is implemented in XQuery and evaluated using the Zorba XQuery Processor[5].

Figure 4.3 on the facing page illustrates Sausalito's integrated application stack. An XQuery application server is provided, which integrates a web server and database system placed in a cloud environment. It leverages Zorba as a query processor and makes use of its accompanying function libraries. On top of Zorba, a web server is responsible for mapping REST requests to XQuery expressions. Beneath, the XQuery processor Sausalito implements Zorba's Store API in order to manage XML data (*i.e.*, instances of the XQuery Data Model). Data is stored in Amazon's Simple Storage Service (S3), and the Sausalito application server orchestrates access to it. Rich applications entirely written in XQuery (including the update and scripting extensions) can thus be deployed on the Amazon Web Service infrastructure.

4.2.2 eXist – the XQuery Servlet

eXist-db has been one of the first open-source native XML database implementations. It pursues the goal of providing developers with an easy to use, out-of-the-box package:

> *eXist [...] a native XML database system, which can be easily integrated into applications dealing with XML in a variety of possible scenarios, ranging from web-based applications to documentation systems running from CDROM. The database is completely written in Java and may be deployed in a number of ways, either running as a stand-alone server process, inside a servlet-engine or directly embedded into an application.*
>
> *Wolfgang Meier (project leader) on the goals of eXist [47]*

[5] http://zorba-xquery.com/

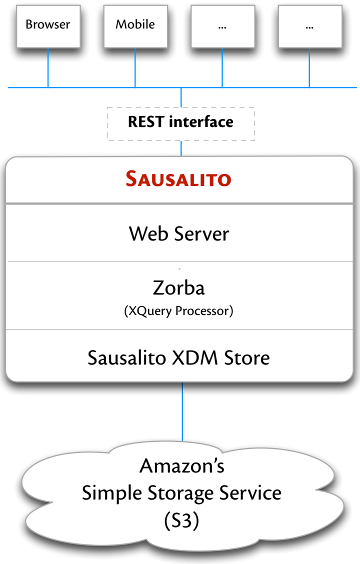


Figure 4.3: Sausalito's integrated application stack

eXist exposes all of its functionality, in addition to traditional programmatic APIs, through web-enabled services: `XQueryServlet` and `REST`.

In order to generate a web page, eXist leverages standard XQuery to generate XHTML. The XQuery runtime is bundled into a servlet implementation, which maps URLs to XQuery script files residing in the filesystem. This very closely resembles the *traditional* way of developing web applications and, as such, has a low-entry barrier for implementers that come from scripting languages such as PHP, Python, Ruby or Perl.

Predefined modules exist to handle request parameters (*e.g.*, GET-, POST- and cookie data), sessions, and authentication. This already allows developers to build fully featured applications relying solely on XQuery and eXist's integrated XML storage.

The significance and versatility of eXist's approach is even more emphasized by the fact that the whole administrative user interface is provided as a web application itself. Beside the administration interface, the eXide XQuery IDE[6] succeeds at providing an excellent in-browser IDE for developing XQuery. The REST-style interface to the database works similar to the XQueryServlet; it only differs with respect to the location of

[6] Accessible online at `http://demo.exist-db.org/exist/eXide/index.html`

the actual XQuery script file: for REST, it is stored directly inside the database and not accessible via the filesystem.

As XQuery is a very young language, the current implementations lack the richness of third-party libraries that other language families and frameworks offer. Hence, developers are often challenged to either implement their own libraries or rely on community efforts. For added functionality, eXist comes with a rich set of prepackaged extension modules that assist developers in creating interactive applications: date & time, file, mail, or image functions, to name only a few.

⋆ ⋆ ⋆

To get the best of both of these worlds—eXist's straight forward approach and Sausalito's service oriented architecture—we present a framework that enables developers to implement a distinct project structure that directly maps information resources to URLs, combined with a direct way to provide end-user views in HTML or RESTfully exposed data encoded in either XML oder JSON.

The main goals we set up for BaseX-Web were flexibility and service orientation, supplied by a pure X-technology stack. Due to the appeal and ease of use that MVC architectures provide, our framework follows these principles to clearly separate concerns regarding storage, processing, and rendering of data (→ 4.3.1 on page 76). But first of all, we will give an overview of our approach to connect the BaseX database management system to the World Wide Web in order to establish an XQuery application framework.

4.3 System Overview

To eliminate binary dependencies, BaseX-Web is implemented as *yet another database client*, which issues requests in XQuery and expects the server to return results serialized as XML, XHTML or JSON.

The application server holds the "business logic", *i.e.*, the functions we want to perform on the stored data. In general, this involves retrieving data from the database, process any input data, whether queries or updates, accordingly (validate and verify results,

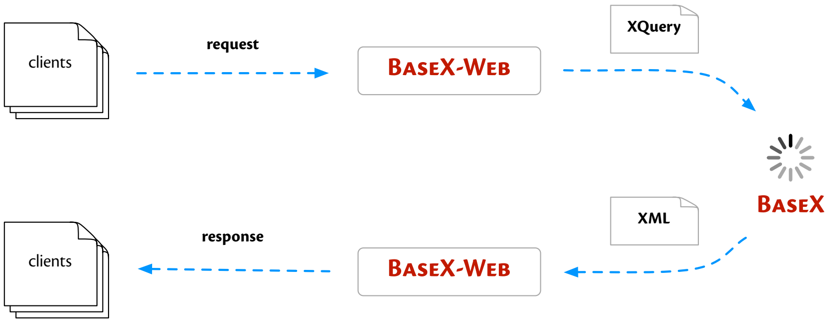


Figure 4.4: System overview: BaseX-Web's general operating sequence

ensure consistency) and send those back to the user interface.

In order to process web-related tasks (cookie handling, redirects and caching options), BaseX has been extended in a non-intrusive way. Extensions for XQuery processors are implemented as separate libraries in conformance with the EXPath packaging specification[7]. They are to be loaded into the database management system once with the `repo install` database command. As such, they can extend the database's functionality whenever their service is needed, but do not encumber the base system or pollute namespaces, otherwise.

The internal API of BaseX is used for the generation of results, which means that our infrastructure directly benefits from all optimizations performed by the BaseX query processor.

So as to provide a light-weight application server, the architecture is based on established components. We used Java Servlet Technology, which provides "developers with a simple, consistent mechanism for extending the functionality of a web server"[8]. The frameworks logic is encapsulated in a Java servlet and deployed as server-independent web application archive (`war`). A web application archive can be thought of as a program that has to be run inside a dedicated environment, namely the servlet container. Currently, the `jetty://` web server[9] is used as a HTTP server and servlet container, as depicted in Figure 4.5:

[7] `http://expath.org/spec/pkg`
[8] `http://www.oracle.com/technetwork/java/javaee/servlet/index.html`
[9] `http://www.eclipse.org/jetty/`

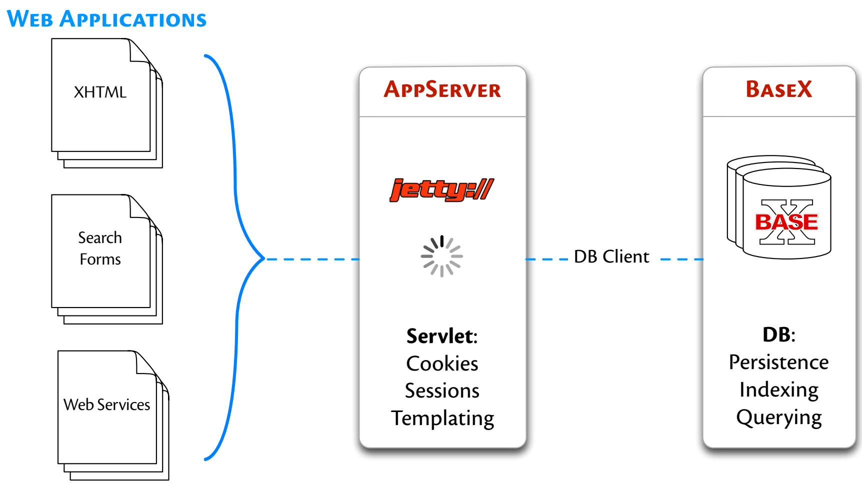


Figure 4.5: System overview: The main building blocks of BaseX-Web. `jetty://` is used as web server and servlet container. The server itself connects to the BaseX XML-DBMS as a database client

4.3.1 Model-View-Controller

MVC is an architectural pattern. It splits software into three distinct development components that interact with each other with respect to specific interfaces. Its goal is to make program design more flexible, thus allowing to change or extend and reuse functionality easily later on.

This idea is based on one simple observation: while user interfaces are subject to change at a much higher speed—one might think of targeting different platforms and types of applications, web applications, mobile apps & desktop applications—the domain logic behind the application is usually much more constant over time. Using MVC even fosters collaboration in development, as one developer might start implementing views and presentational logic while another one copes with data abstraction and domain logic.

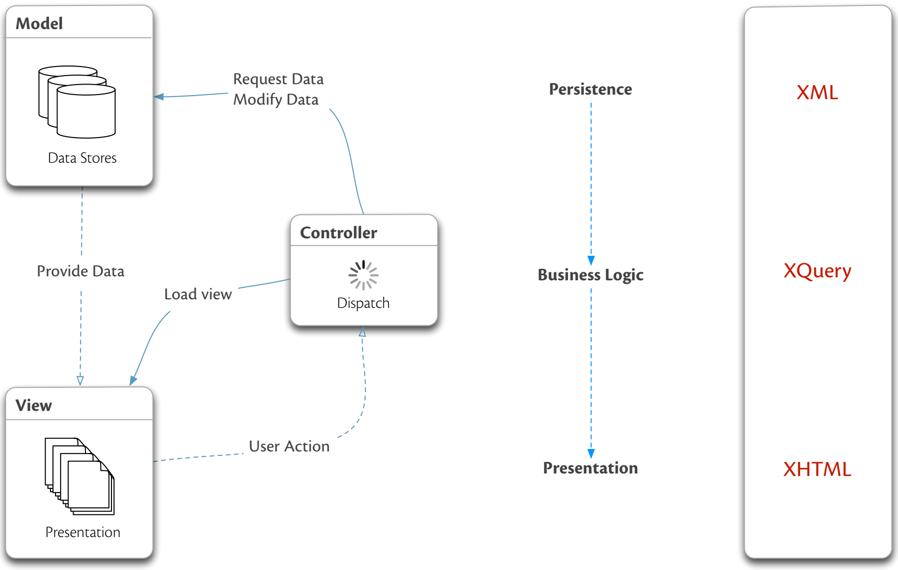


Figure 4.6: Using the Model-View-Controller paradigm to build a uniform X-technology stack

The Model

The model encapsulates all data-related tasks and logic. In traditional software development, this often relates directly to tables or views located in a DBMS. The model may also be responsible for enforcing constraints on data structures, such as having certain attributes present or conform to certain conditions. Some publications [9, 59] also differentiate between

- the *active model*, which has a notification mechanism—usually implemented with the Observer pattern—and notifies its views or controller of changes, and
- the *passive model*, which is completely unaware of the fact that it is part of an MVC architecture.

As web applications must obey a strict request-response cycle, and are usually stateless, we will refer to passive models in the following unless mentioned otherwise.

The View

The view does nothing but obtain data from the model, and present it to the user. Usually, a view is instantiated by the controller, which also passes the needed data in. Views are not supposed to change or modify, or in any other way interfere with the model. All this is done by the controller.

The Controller

Controllers maintain the state and business logic of the application; they act as glue between the models and their views. Controllers react to user actions and provide their respective views with requested data that is obtained from the model. Originally designed for the implementation of GUIs of desktop applications, controllers have been the interface which is responsible for dispatching the event loop of a particular view. For example, controllers receive keyboard, timer, or mouse events from a view and change the state of a model. In its original definition, a controller was not meant to act as a mediator between the view and the model, but this gradually changed with web application frameworks. The special characteristics of web applications led to the introduction of various controller patterns, most notably that of the:

Front Controller as a single point of entry for dispatching HTTP requests,

Page Controller coordinating the logic on a single web page, and

Application Controller that defines the business logic of the whole application.

These controllers are often cascaded, in a such way that the *front controller* accepts incoming requests, dispatches them to a page controller's action, which in turn renders a view, handles this view back to the *page controller*, which in return sends a complete page back to the client.

This further extends the *separation of concerns* adequately for the special case of web applications as, from an architectural point of view, different actors in the application are driven by different demands. As an example, the HTML representation, which is the view in the eyes of the web server, becomes the model once it is sent to the client and rendered inside the browser, and may be part of an MVC architecture on the client side, if used, for example, in conjunction with JavaScript.

4.3.2 Application Layout

BaseX-Web was developed with the MVC paradigm as guiding principle in mind. Applications implemented on top of BaseX-Web have to follow some basic rules that directly derive from it. The general application layout and its project's directory structure are an example:

The `models` folder may contain XML Schema definitions to validate data before it is inserted into a database.

The `controllers` folder contains an arbitrary number of controllers that encapsulate reusable business logic.

The `layouts` folder contains predefined (X)HTML pages inside which evaluated content is placed before it is served.

The `views` folder contains a folder for each controller available inside the application and has XQuery files—named *actions*—that respond to a unique URL.

```
DeepWebApplication
|-- models
|   `-- fsml.xsd
|-- controllers
|   `-- deepweb.xq
|-- layouts
|   |-- ajax.html
|   `-- default.html
`-- views
    `-- deepweb
        |-- search.xq
        |-- ls.xq
        |-- dir.xq
        `-- action.xq
```

Developers are forced to explicitly categorize their project's files, depending on their functionality, as this fosters collaboration: once learnt, developers know where to put and find the functions they are searching for. In addition, we are able to map this directory layout to meaningful URLs that contain information resources.

4.3.3 Request-Response Cycle

The following part deals with implementation and engineering details, which were necessary to make our application server "talk" XQuery. We will follow a full request-response loop from a user's browser to our server and back (Figure 4.7 on the facing page).

Please note that, even though we chose `jetty://` as our web container, the results are as well applicable to any other implementation conforming to the servlet specification.

Incoming HTTP requests that address a web application inside BaseX-Web must adhere to the following pattern: `/app/$controller/$action`. The servlet will parse this URL and ensure that the following preconditions are given:

- an XQuery file `controllers/$controller.xq` *may* exist inside the project directory; if it does, a flag is set
- an XQuery file `views/$controller/$view.xq` *must* exist inside the project directory

If the view exists, it is read into memory. If not, an `HTTP/404` error is sent to the client.

In case the specified controller module exists, it is imported into the view. The servlet injects the module import into the view, such that all functions defined inside the controller are available to the view in the `$controller` namespace. This convenient *automagic* frees the user from manually importing the view's controller for each view she implemented.

Subsequently, the view and, optionally, the embedded controller are passed on to the BaseX client `QueryProcessor` instance and wait until they are sent to the `BaseX` server for evaluation.

Side effects that are caused by parameters being sent to the requested URL, such as `GET`, `POST`, `COOKIE` or `SESSION`, are bound in a map[10] to the `QueryProcessor` as external XQuery variables. These variables are accessible inside the *view* via `$GET('param')` and allow users to pass on parameters in order to send specific information to the website.

[10] XQuery's notion of Key-Value-Paris

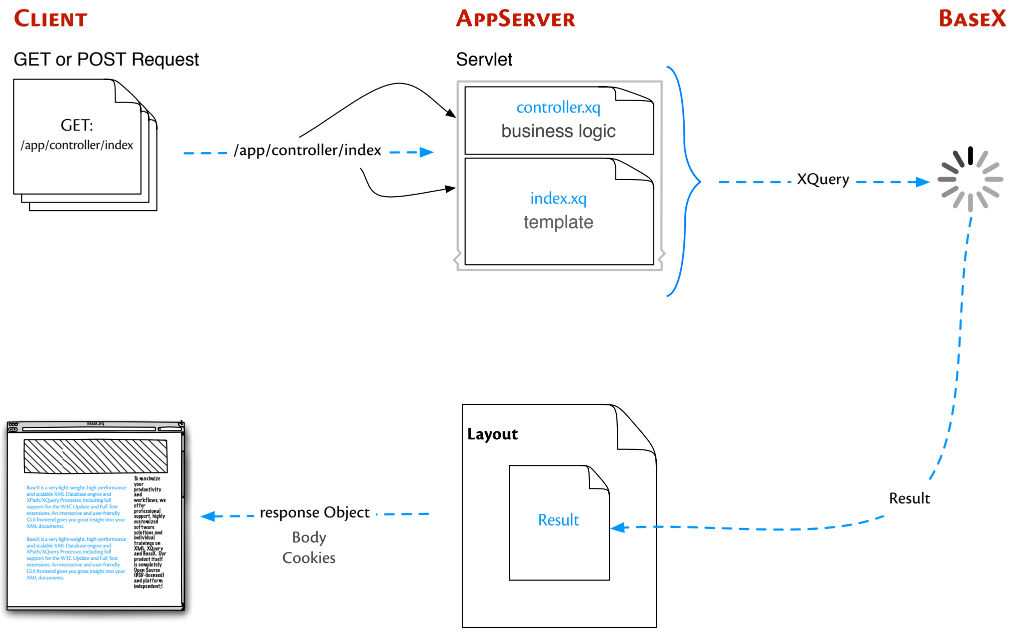


Figure 4.7: Complete request-response cycle. User requests trigger the "XQuery construction" step, *i.e.*, the controller gets embedded into a page template. Processing of the result page is done by the BaseX query processor, which accesses databases if needed

At this stage, the query and its arguments are fully specified, and the query is dispatched to the BaseX database server. The server processes the query—like any other incoming query—and returns the result back to the servlet. A dedicated XQuery extension module provides the implementor with special-purpose functions to modify parameters that are related to the servlet `HttpServletResponse`[11] object.

The invocation of any of these functions triggers a notification from BaseX to the servlet. The notification adheres to the command pattern, as depicted in Listing 14 on the next page.

The servlet watches for these events and invokes the appropriate methods with the given arguments in order to modify the response accordingly. Meanwhile, the servlet waits for BaseX to return the evaluated query results. Once a result is received, the

[11] `http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html`

```
<command name="set-cookie">
        <session>21d87371-9e01-4b3c-936a-4c80bad47019</session>
        <arg>Cookiename</arg>
        <arg>This is the cookie's value</arg>
</command>
```

Listing 14: XML fragment notifying the servlet to add a cookie to the response

servlet embeds the resulting string into `layouts/ default.html`, unless asked by the implementer to do otherwise, and flushes it into the response body. Afterwards, the response will be sent to the client. It includes all received headers and the complete body, which represents the full web page.

An extensive discussion of further implementation details can be obtained from the Master Thesis [57] that accompanied this work.

4.4 Summary

BaseX-Web sprung from the desire to develop graphical user interfaces and user applications that do not depend on the internal APIs of a database, such as *e.g.*, the native visualizations of BaseX.

Our considerations have finally led to an architecture that allows the user to develop and run data-driven applications. While the development can be carried out on a more high-level and generic abstraction layer, it still provides full-fledged database support. Direct access to indexes and other internal data structures have been wrapped into XQuery modules and can be integrated whenever required. BaseX-Web is a powerful web application framework for creating uniform search and retrieval services without being a slave to a multitude of languages and concepts. The implementation of services exclusively relies on the W3C technology family, eliminating unnecessary paradigm shifts.

Kaufmann and Kossmann concluded their work "Developing an Enterprise Web Application in XQuery" with the words: "Today, the biggest concern in adopting this approach [*using the uniform W3C technology stack (author's note)*] is that there are no mature application servers available, but we believe that the situation will change soon in this regard. [...] In the future, more experience with other applications [*others, than* *the evaluated demo application in the paper (author's note)*] is needed." [42]

BaseX-Web is our contribution to address these issues. Although a larger portion of convenience code, such as scaffolding, is not available yet, BaseX-Web—as a light-weight and highly extensible framework—is already used in production, as it represents a solid basis for complex web applications. It is also available online as a public open-source project.

An application like the presented "Desktop Query Engine" at the end of Chapter 3 clearly illustrates the advantage of having an extended database architecture as a service within a modern operating system. Similar to desktop search engines like Spotlight, the application provides methods to manage personal information.

The database architecture can be applied to implement such a service by exclusively relying on the XML technology stack. While XQuery is used to query the filesystem contents, the frontend can be realized as a desktop-caliber web application, which enables the user to perform keyword searches, visually explore the results, or steer a faceted search navigation.

In the following chapter, we will describe the realization of another search system. In the course of its evaluation, the second addressed issue will be solved, namely that "more experience with other applications is needed". We will discuss the necessary steps to bootstrap an expert retrieval system that directly benefits from BaseX-FS and BaseX-Web.

Next, a generic solution for an OPAC (Online Public Access Catalogue) system will be illustrated as one possible use case for a purely XQuery-driven web application. It will demonstrate how an information system works with distinct data sources. Ideally, there should be support for queries on library-oriented metadata (author, title, publisher, ISBN) and for queries that directly operate on the content (full-texts for publications, metadata for multimedia documents).

By setting up a concrete example application, we want to illustrate how our XML database architecture can form a strong foundation for the construction of data driven search and retrieval systems in general.

5 Kickstarting an Infrastructure

Online Public Access Catalogs (OPAC) are online databases that provide user access to a library catalog, usually in the form of a web application. As today's libraries have great amounts of all kinds of different media—think of books, newspapers, magazines, journals, DVDs and software—sophisticated retrieval systems are crucial.

Kickstarting such a retrieval system on top of our architecture is not only elegant, but also straightforward. In the following paragraphs, we will illustrate the process by first sketching the general system setup and then evaluating it against real-world data.

Therefore, we will take the raw data of the Konstanz Online Publication System (KOPS), an OPAC run by the Library of the University of Konstanz, and bootstrap a basic OPAC system solely powered by BaseX and the extensions discussed in the previous chapters.

We want to demonstrate that we are able to kickstart a performant retrieval system realized with significantly less implementation effort (compared to state-of-the-art solutions), that, in addition, is easy to extend and customize towards future information demands.

5.1 Online Public Access Catalog (OPAC)

An OPAC enables users to conduct searches for *traditional* bibliographic metadata, such as authors, titles, keywords, institutes or publishers.

This rather blurry definition of OPACs led researchers to refining this concept, to what they now name a *digital library*. In "The DELOS Digital Library Reference Model. Foundations for Digital Libraries" [2], the authors refer to a digital library as a three-tier framework providing services and infrastructure, which consists of three core layers:

> **Digital Library (DL)**
> An organisation, which might be virtual, that comprehensively collects, manages and preserves for the long term rich digital content, and offers to its user communities specialised functionality on that content, [...].
> **Digital Library System (DLS)**
> A software system that is based on a defined (possibly distributed) architecture and provides all functionality required by a particular Digital Library. Users interact with a Digital Library through the corresponding Digital Library System.
> **Digital Library Management System (DLMS)**
> [...] that provide[s] the appropriate software infrastructure both (i) to produce and administer a Digital Library System [...] and (ii) to integrate additional software offering more refined, specialised or advanced functionality.
> c.f. [2, p. 17]

5.2 Konstanz Online Publication System (KOPS)

The Library of the University of Konstanz provides an institutional repository called "Konstanz Online-Publikations-System" (KOPS). It is an information platform supporting the Open Access[1] initiative. Members of the University can publish digital documents and make them available on the internet. An online search interface provides simple and advanced keyword search options. Simple search is restricted to keywords, while advanced search allows for boolean queries on author, title, project, keywords, DDC (Dewey Decimal Classification), and full-text retrieval.

KOPS is driven by the Open Source Software DSpace[2]. DSpace is a major player in the field of institutional repositories. It supports all kinds of media types, such as text corpora, scans, photographies, video contents and many others. In KOPS, it is used to

[1] Open Access (OA) stands for unrestricted, toll-free online access to scientific and scholarly knowledge and information.

[2] www.dspace.org

- search for one or more keywords in metadata or extracted full-texts
- browse through author, title, project, keywords, date or category

DSpace is organized into a classical three-tier architecture with each layer consisting of a number of components[3].

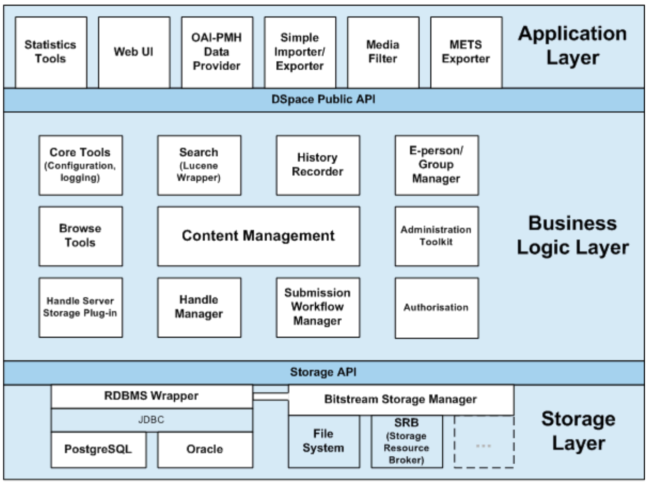


Figure 5.1: DSpace System Architecture Overview

Behind the scenes, the main components are Lucence[4], the open source Java search engine, a storage layer with an RDBMS, and a so-called Bitstream Store[5] to store binary content. Lucene allows categorized searches, stopword removal, stemming, and the ability to incrementally add new indexed content without regenerating the entire index.

At the time of writing, KOPS contained 12,312 library entries. 4,163 entries were only library-oriented metadata with no document attached, and 8,149 were available with the complete full-text.

[3] https://wiki.duraspace.org/display/DSDOC18/Architecture

[4] http://lucene.apache.org/

[5] Either filesystem or a Data Grid Management System called DICE Storage Resource Broker http://www.sdsc.edu/srb/index.php/Main_Page

5.3 XML OPAC

5.3.1 Intention

In the following, we demonstrate that BaseX—with its extensions for rich-media indexing (BaseX-FS) and web application framework (BaseX-Web)—is capable of bootstrapping an infrastructure such as KOPS

- by exclusively utilizing a state-of-the-art XML-DBMS, and
- without ever leaving the unified X technology stack

Advantages we expect from our approach:

- **low entry barrier: reduced design, setup and administration effort**
  - load information "as is". No need to define a data model in advance
  - data driven approach: index and search on anything that has been loaded without knowing the questions ahead of time
- **low implementation effort/high expressiveness**
  - no glue code, no impedance mismatch (data is XML, business logic is XQuery, result presentation is XHTML)
  - full-text functions, for example, are first-class citizens, while in comparable systems this functionality is either vendor-specific or has to be added by additional, external subcomponents
- **lean system architecture**
  - Instead of having a combined system architecture (like the one depicted in Figure 5.1 on the preceding page, *i.e.*, a RDBMS, a concurrent full-text index engine …), XML-DBMS are document-centric information stores tuned to operate on semi-structured data.

    A major advantage stemming from this fact is that programmatic access to all system components (such as full-text indexes) is provided in a consistent and transparent way through XQuery[6]. It is sufficient to master a single system instead of being an expert in a multitude of subcomponents and their interweavements.

[6]
```
(: Returns all index entries for text nodes starting with "Germ" :)
index:texts("factbook", "Germ")
```

The following two sections will give a brief and concise overview of the steps necessary to bootstrap the base system and to configure it towards an online retrieval system.

5.3.2 Foundation: General System Setup

Consider a vanilla server machine. The proposed architecture is built upon an arbitrary Linux distribution with Java and FUSE support. In the next step, the BaseX database server with its BaseX-FS extension is installed.

Two ways of bringing the content into the database system can be considered. As mentioned, the library data set consists of 8,149 publications. The simplest approach is to just make use of the generic BaseX-FS' transducers (Section 2.2 on page 29).

The PDF transducer will break down the binary information silo and aggregate metadata, full-text, annotations and embedded images into a unified XML representation. Original raw data is incorporated into the database system as well. Bulk-loading the initial data set now boils down to:

- Mount an empty database as filesystem
- Copy the documents into the database/filesystem
- Let transducers construct an XML view of the data

The resulting database will contain a uniform view on the metadata, formerly available only to dedicated programs; it will allow us to handle these data in a standardized and generic way. Throughout the whole process, our mapping does not contain any information specific to our use case. Instead, we extract all *information as is*, leaving alone any assumption about which data we are going to need afterwards.

Listing 15 on the following page shows how full-text of a document is stored in the database[7].

The second approach takes into account that each publication in KOPS has already encountered extensive bibliographic tagging. In order to leverage this—by librarians, manually processed data—a specialized transducer can be plugged into the transducer

[7] A more detailed database excerpt is shown in Appendix Listing 22 on page 124

```
      <folder name="pages">
        <folder name="page" number="1">
          <fact name="text">
Interactive exploration of fuzzy clusters using Neighborgrams
Michael R.Berthold - Bernd Wiswedel - David E.Patterson

Department of Computer and Information Science,University of Konstanz,Box M712,78457 Konstanz,Germany

Data Analysis ResearchLab,Tripos Inc.,USA

Abstract
We describe an interactive method to generate a set of fuzzy clusters for classes of interest of a
given,labeled data set.

The presented method is therefore best suited for applications where the focus of analysis
lies on a model for the minority class or for small  to medium-sized data sets.

The clustering algorithm creates one dimensional models of the neighborhood for a set of patterns
[…]
          </fact>
        </folder>
```

Listing 15: KOPS-FSML.xml: Extracted full-text from online resource

chain. Besides the conventional, generic PDF metadata and full-text extraction, it adds bibliographic metadata as opacinfo to the result (example shown in Listing 16).

```
<file name="1896748.pdf" suffix="pdf" st_size="533883">
  <folder name=".1896748.pdf.deepfs">
    <folder name="opacinfo">
      <fact name="pagecount">17</fact>
      <fact name="author">Berthold, Michael</fact>
      <fact name="author">Wiswedel, Bernd</fact>
      <fact name="author">Patterson, David E.</fact>
      <fact name="title">Interactive exploration of fuzzy clusters using Neighborgrams</fact>
      <fact name="town">Konstanz</fact>
      <fact name="publisher">Bibliothek der Universität Konstanz</fact>
      <fact name="year">2005</fact>
      <fact name="format">Online-Ressource</fact>
      <fact name="note">Article</fact>
      <fact name="signature">|004</fact>
      <fact name="language">Englisch</fact>
      <fact name="category">Informatik</fact>
      <fact name="url">http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-65525</fact>
      <fact name="creation-date">November 17, 2004 21:34:22 (UTC)</fact>
      <fact name="modification-date">October 13, 2008 14:42:40 (UTC +02:00)</fact>
    </folder>
    <folder name="fulltext">
```

Listing 16: KOPS-FSML.xml: Bibliographic metadata about online resource

Briefly summarized, the *xmlified* and raw content is stored inside BaseX and can be queried, using a standardized, declarative API written in XQuery. Having completed this initial step we are ready to configure our online retrieval application. Using BaseX-Web as application framework a developer can easily leverage the database to analyze, search, and discover all data at various levels of granularity since every asset of interest is now indexed and homogeneously represented in the database.

5.3.3 Configuration: Shaping a Retrieval Application

After having completed the initial steps, extracting and preloading the OPAC data, we are ready to set up our application skeleton.

We start conceptually, by defining which user requests our system will respond to, and how search results are going to be represented internally. As XQuery—contrary to its relational counterparts—is a fully fledged programming language, we face a much higher level of expressiveness and at the same time we eliminate the need of scripting language *glue code* when processing user input.

Listing 17 shows a query that performs a search for all works that match a given `key, value` combination and `<file />` elements for further processing. To parametrize the function we pass two strings, the `key` we are searching for and its wanted `value`. Once we have defined the search functionalities in terms of XQuery functions, we are ready to leverage the frameworks capabilities. We add those expression as functions to an OPAC controller, located in `controllers/opac.xq`.

```
(: Search works matching a given key/value combination: :)
declare function opac:keyValue($key, $value){
  //file[.//fact[ ./@name eq  $key and . eq $value]]
};
```

Listing 17: An XQuery function returning all `file` elements matching a specific `key, value` combination

In order to extend the search capabilities—more elaborate examples will be given in Section 5.4—implementers have to do little more than adding more XQuery functions to the controller. All defined functions may as well be used in any context besides an web application, so far we have only defined the search process, not its representation.

Afterwards we define how these internal results are to be transformed to a more browser —and user-friendly—(X)HTML representation, by providing a view. A view will not only request the search results and conduct the transformation, but as well represents an unique, machine accessible resource that provides an interface to underlying data.

In order to return a list of all elements that matched a particular search request, we create `views/opac/simple-search.xq`, a *view* that:

- receives user input, the search parameters
- leverages the controller's function(s) to obtain result data from the database and
- passes results back to the view, which in return transforms these to XHTML

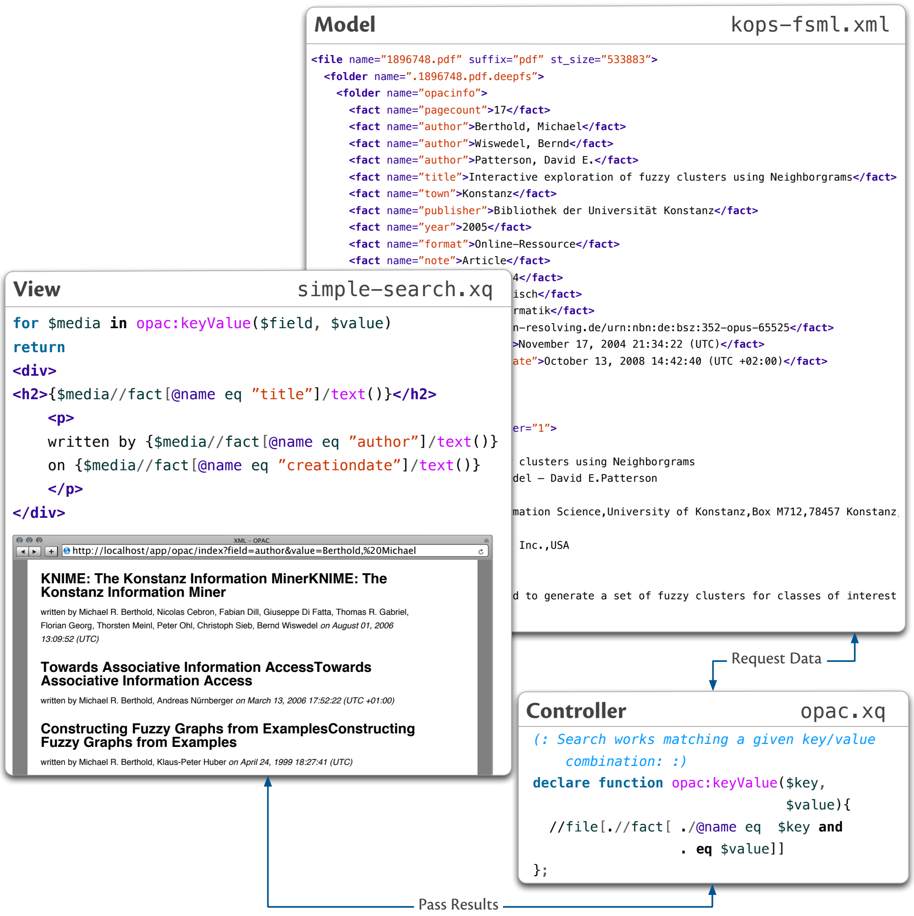
Once created, that particular view is immediately accessible at `http://xmlopac/app/opac/simple-search`.

A result page view based on the previous definition of `opac:keyValue` is shown in Listing 18. This very generic approach may be used to render `file`-elements regardless which transducer produced their metadata.

```
for $media in opac:keyValue($field, $value)
return
  <div>
    <h2>{ $media//fact[@name eq "title"]/text() }</h2>
    <p>
    written by { $media//fact[@name eq "author"]/text() }
    on { $media//fact[@name eq "creationdate"]/text() }
    </p>
  </div>
```

Listing 18: The XQuery OPAC simple search view

The BaseX-Web server will automatically take care of importing the controller inside the view, thus making its defined functions available inside the `opac` namespace. Figure 5.2 depicts how the components work together to form the basic infrastructure.



```
<file name="1896748.pdf" suffix="pdf" st_size="533883">
  <folder name=".1896748.pdf.deepfs">
    <folder name="opacinfo">
      <fact name="pagecount">17</fact>
      <fact name="author">Berthold, Michael</fact>
      <fact name="author">Wiswedel, Bernd</fact>
      <fact name="author">Patterson, David E.</fact>
      <fact name="title">Interactive exploration of fuzzy clusters using Neighborgrams</fact>
      <fact name="town">Konstanz</fact>
      <fact name="publisher">Bibliothek der Universität Konstanz</fact>
      <fact name="year">2005</fact>
      <fact name="format">Online–Ressource</fact>
      <fact name="note">Article</fact>
                               4</fact>
                               isch</fact>
                               rmatik</fact>
                               n–resolving.de/urn:nbn:de:bsz:352–opus–65525</fact>
                               >November 17, 2004 21:34:22 (UTC)</fact>
                               ate">October 13, 2008 14:42:40 (UTC +02:00)</fact>



                               er="1">


                                clusters using Neighborgrams
                               del – David E.Patterson

                               mation Science,University of Konstanz,Box M712,78457 Konstanz,

                                Inc.,USA


                               d to generate a set of fuzzy clusters for classes of interest
```

```
for $media in opac:keyValue($field, $value)
return
<div>
<h2>{$media//fact[@name eq "title"]/text()}</h2>
    <p>
    written by {$media//fact[@name eq "author"]/text()}
    on {$media//fact[@name eq "creationdate"]/text()}
    </p>
</div>
```

XML – OPAC
http://localhost/app/opac/index?field=author&value=Berthold,%20Michael

KNIME: The Konstanz Information MinerKNIME: The Konstanz Information Miner

written by Michael R. Berthold, Nicolas Cebron, Fabian Dill, Giuseppe Di Fatta, Thomas R. Gabriel, Florian Georg, Thorsten Meinl, Peter Ohl, Christoph Sieb, Bernd Wiswedel *on August 01, 2006 13:09:52 (UTC)*

Towards Associative Information AccessTowards Associative Information Access

written by Michael R. Berthold, Andreas Nürnberger *on March 13, 2006 17:52:22 (UTC +01:00)*

Constructing Fuzzy Graphs from ExamplesConstructing Fuzzy Graphs from Examples

written by Michael R. Berthold, Klaus-Peter Huber *on April 24, 1999 18:27:41 (UTC)*

```
(: Search works matching a given key/value
    combination: :)
declare function opac:keyValue($key,
                                $value){
  //file[.//fact[ ./@name eq  $key and
                . eq $value]]
};
```

Figure 5.2: The core components of the web architecture:
Model contains the complete data extracted from KOPS
View represents an URL and orchestrates user requests to parameterized XQuery function calls
Controller holds the logic to retrieve and return search results

The screenshot shows the computed result rendered inside a browser when opening `http://xmlopac/app/opac/simple-search?field=author&value=Berthold,%20Michael`

5.4 Evaluation Setup

In the previous section we have shown how to quickly setup a basic infrastructure to drive a search and retrieval system. We now want to put our system to the test and examine if it's equally fast when it comes to the evaluation of common search requests.

To conduct our real-world data study, we obtained a full dump of all data available online in KOPS and transformed it into an XML representation. As already shown, the resulting XML database instance contains all entries of the original data, the bibliographic metadata and, whenever available, the full-texts of the actual PDF documents.

Some key characteristics for the input data and the resulting database are displayed in Table 5.1.

Input statistics		**Index statistics**	
Size of input data	`17 GB`	Size of full-text index	`614 MB`
# files	`8,149`	# full-text index entries	`1,984,734`
# PDF pages	`254,299`	# XML nodes	`3,671,331`
# authors	`25,793`	# `<fact/>` elements	`668,191`

Table 5.1: Statistics on the original KOPS library resources and the resulting database `kops-fsml.xml`

All queries were benchmarked against BaseX Version 7.1 with the following settings: `java -server -Xmx4096m -Xms1024m`. To make results more reliable, we restarted BaseX before each test, then every query was run 20 times to warm the caches. Next, the actual measurements were performed by running the query again for 10 times and storing the average response times, which include all evaluation steps (parsing, compilation, evaluation of the query, and serialization of the results).

5.5 Queries and Performance Results

5.5.1 Keyword Search

Due to its simplicity, keyword search has turned out to be one of the most dominant approaches for expressing one's information needs on the internet. Keywords are filled in by users into search fields, then matched against inverted indexes for an underlying text corpus, and all documents are returned that contain the keywords and potentially related terms. Related terms may be derived by stemming the text corpus or enhancing the full-text with thesauri and language specific features.

As a consequence, high performance in keyword search scenarios is crucial for a system's acceptance, and the full-text extension of XQuery [12] provides a standardized way of formulating such requests for XML. A keyword search in XQuery Full Text that retrieves relevant document files can, for example, be expressed as shown in Listing 19.

Query: Keyword search using XQuery Full Text

```
let $words := ("problem", "properties")
return //*[text() contains text { $words } all words]/ancestor::file
```

Listing 19: A keyword search function for the OPAC XQuery module (`opac.xq`)

So as to benchmark the keyword search performance, we randomly selected 10 keywords from the text corpus and performed a keyword search for all possible combinations ($\binom{10}{2} = 45$) of those 10 words. Each query was run 10 times against a document corpus containing 250, 500, 1000, 2000, 4000 and 8000 source documents.

Results: Table 5.2 on the following page shows the runtime statistics for each of the six database instances. The results can be read as follows: For the largest database containing 8000 documents all 45 keyword search queries could be evaluated in a total time of 706.32ms. Thereby the fastest query took 8.92ms and the slowest 36.75ms. On average the evaluation could be performed in 15.70ms. Adding up all matching documents this results in a total number of 27, 169 hits (the single number of hits for each query can be derived from Table 5.3 on the next page).

Corpus size	250	500	1000	2000	4000	8000
Total time	12.04	30.55	63.59	164.67	367.61	706.32
Min	0.06	0.14	0.43	1.62	4.59	8.92
Max	0.66	2.28	5.33	9.07	14.09	36.75
AVG	0.27	0.68	1.41	3.66	8.17	15.70
Total # of Hits	709	1,570	3,135	6,813	14,037	27,169

Table 5.2: Runtime statistics for the keyword search queries against six differently sized corpora

Figure 5.3 is showing two graphs depicting these results again. For each of the six database instances on the abscissa it shows

(a) how many documents have been evaluated as matches and
(b) how much time in ms, this evaluation took on average.

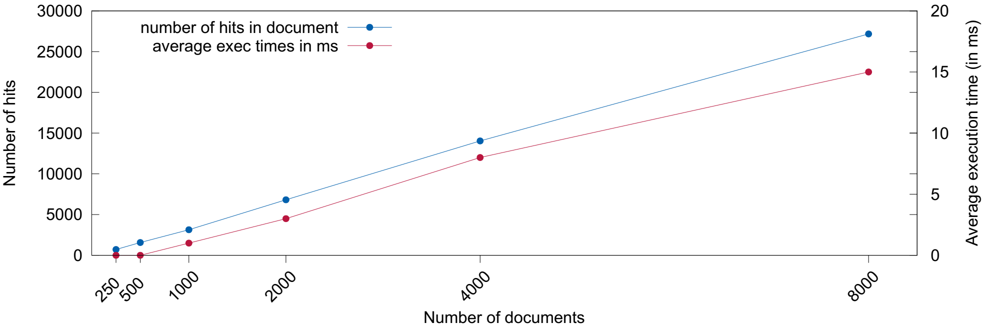


Figure 5.3: Average runtime in ms (red line/right y-axis) to evaluate 45 keyword queries on each of the six corpora (x-axis). Blue line/left y-axis shows the accumulated number of matching documents

Detailed performance results for the 8000 source documents are shown in Table 5.3 on the facing page. It depicts all 45 possible combinations of the keywords in question, and shows the absolute number of hits and the time needed for returning the results of particular keyword combinations.

	germany	problem	change	science	formation	situation	space	properties	material	power
germany	—	645 / 15.28	919 / 22.07	1387 / 36.75	976 / 20.06	438 / 12.26	554 / 13.17	1010 / 18.98	777 / 17.58	506 / 13.64
problem	—	—	813 / 21.14	669 / 17.13	417 / 11.59	795 / 21.61	724 / 21.57	625 / 16.98	438 / 12.53	508 / 15.12
change	—	—	—	860 / 21	842 / 20.49	666 / 18.04	628 / 17	863 / 21.41	541 / 14.84	612 / 18.25
science	—	—	—	—	799 / 18.55	338 / 9.72	492 / 13.07	793 / 17.95	393 / 10.73	483 / 13.34
formation	—	—	—	—	—	386 / 10.63	380 / 10.65	912 / 21.65	608 / 15.36	291 / 8.92
situation	—	—	—	—	—	—	338 / 10.24	371 / 10.07	376 / 10.66	321 / 10.53
space	—	—	—	—	—	—	—	628 / 17.54	314 / 9.09	404 / 11.65
properties	—	—	—	—	—	—	—	—	622 / 16.53	407 / 11.45
material	—	—	—	—	—	—	—	—	—	300 / 9.5
power	—	—	—	—	—	—	—	—	—	—

Table 5.3: Number of results and time for generating the results for a keyword search against the 8000 file database

Analysis: Due to the exploitation of the full-text indexes, all query runtimes scale linearly for the tested database instances. Index lookup itself is negligible and the most limiting factor in terms of performance is the number of the results, as this determines the amount of data to be serialized. Hence, very large corpora may be searched yielding very fast response times. In our specific case, the slowest query, searching for the keywords *germany* and *science*, needs 36.75ms on the largest corpora, yielding 1.387 result documents.

5.5.2 Phrase Search

There are numerous cases in which plain keyword searches alone are too limiting. Phrase search is a highly needed functionality for most current retrieval systems. It enables users to search for, *e.g.*, compound names, terms and sentences containing words in a fixed order. Phrase searching removes *noise*, added by documents that contain the keywords but not necessarily in the order requested by the user.

Table 5.4 lists phrases of lengths two to five, that have been manually selected from the `kops-fsml.xml` corpus. The phrases themselves consist of keywords that—concerning their number of index entries—cover a range from rare to very frequent. The runtime statistics show a much higher variance than in the previous test case: query runtimes do not increase with the number of hits; an explanation will be given in the analysis. An XQuery script, shown in Listing 23, has been used to generate the results in Table 5.4.

Query: Phrase search using XQuery Full-Text

```
//*[text() contains text "with respect to" phrase ]
```

Results. Table 5.4 shows the conducted phrase searches and their average execution time for ten runs. The number of matching nodes is given for each phrase, and each phrase's keyword is listed with the number of occurrences in the Full-Text index.

	T(ms)	# matching node	chosen phrase, with number of index entries per keyword
Q 1	0.45	0	minor: 2218; drawback: 450
Q 2	1.25	2	major: 8553; deficiency: 368
Q 3	2.88	79	particularly: 4800; strong: 9900
Q 4	5.33	18	special: 5669; interest: 7380; group: 15147
Q 5	6.10	51	major: 8553; contribution: 4139
Q 6	11.10	593	Related: 17695; Work: 17362
Q 7	30.36	1107	Experimental: 12858; results: 36192
Q 8	42.57	2	Stabilisieren: 203; konnte: 18118; sich: 73862; dieses: 18674; System: 28553
Q 9	81.98	50	We: 53641; conclude: 2958; with: 102476
Q 10	167.86	48	I: 87473; would: 19880; like: 17708; to: 119519; express: 2142
Q 11	222.91	8571	with: 102476; respect: 10168; to: 119519
Q 12	248.23	5	major: 8553; advantage: 3319; of: 148306; our: 26799
Q 13	276.81	2949	As: 96236; shown: 23813; in: 228856
Q 14	367.56	5105	in: 228856; contrast: 12264; to: 119519

Table 5.4: Phrase searches: The average runtime per phrase query is shown. Queries 9–14 clearly stand out in terms of time taken

Analysis. As shown in Table 5.4 more than half of the selected phrases evaluate under 50ms due to exploitation of the full-text index.

Most phrases are evaluated in interactive time. We were especially interested in the limits of the presented architecture, thus we considered the phrases (Q9-Q14), that took much longer than the other queries, c.f. Figure 5.4, for more thorough analysis.

As the results indicate, there is no direct relationship between the query times and the number of results. Instead, queries with large result sets may be evaluated fast while other, slower evaluated queries yield much smaller result sets.

One general observation that can be derived from the resulting times is, that phrases

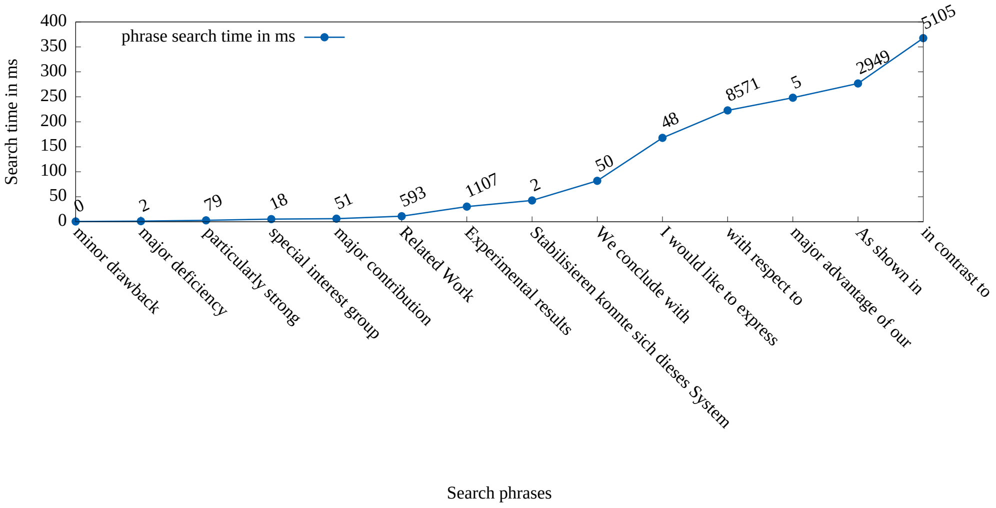


Figure 5.4: Phrase search: Result graph showing the average runtime needed to search for each phrase and the total number of matching nodes

containing a frequent word (*i.e.*, *stopwords* that happen to occur frequently in natural language corpora, or proper names) tend to be evaluated an order of an magnitude slower than phrases made up only of rare words.

This is mainly due to BaseX' evaluation strategy:

- all index hits for each keyword are evaluated as intermediate result lists containing `node ids`, and
- subsequently merged in order to produce a complete result set.

Thus, in the worst case, lots of large intermediate result lists per keyword have to be sorted and merged, often producing only very small final results.

Increasing the total number of keywords may pose an additional penalty on the runtime as each keyword adds an intermediate result list to the workload.

Possible solutions to overcome the problems with large intermediate result sets could involve the following suggestions:

- The pipelining concept could be pushed down to the index access operation: in-

w1∧¬(w2)	germany	problem	change	science	formation	situation	space	properties	material	power
germany	–	4574 165.79 ms	4483 154.87 ms	4168 140.47 ms	4382 141.89 ms	4731 156.94 ms	4585 139.52 ms	4202 169.72 ms	4528 140.8 ms	4673 136.14 ms
problem	3251 173.73 ms	–	3252 175.13 ms	3306 167.65 ms	3325 170.35 ms	3321 166.29 ms	3281 152.81 ms	3286 165.41 ms	3336 163.97 ms	3348 155.28 ms
change	3439 159.77 ms	3486 171.13 ms	–	3458 156.73 ms	3414 156.08 ms	3555 162.56 ms	3504 148.84 ms	3439 153 ms	3528 154 ms	3528 144.54 ms
science	3786 143.55 ms	4106 162.19 ms	4023 154.2 ms	–	4023 143.71 ms	4204 162.9 ms	4137 138.74 ms	4014 144.41 ms	4185 145.08 ms	4176 136.19 ms
formation	2734 136.45 ms	2862 153.72 ms	2781 151.56 ms	2737 133.34 ms	–	2904 148.74 ms	2876 126.66 ms	2745 127.8 ms	2850 128.21 ms	2894 125.16 ms
situation	2768 154.08 ms	2747 156.02 ms	2714 154.75 ms	2837 152.14 ms	2790 153.56 ms	–	2843 138.8 ms	2796 148.51 ms	2850 142.76 ms	2859 135.29 ms
space	2155 125.63 ms	2176 130.55 ms	2162 128.62 ms	2176 122.56 ms	2193 119.82 ms	2262 125.89 ms	–	2176 115.86 ms	2221 116.93 ms	2222 108.28 ms
properties	2808 142.32 ms	2974 153.58 ms	2917 146.36 ms	2895 138.09 ms	2811 130.64 ms	3064 154.07 ms	2960 126.4 ms	–	2956 131.86 ms	3023 128.93 ms
material	2375 133.97 ms	2554 148.37 ms	2503 145.77 ms	2516 133.83 ms	2457 126.44 ms	2576 136.36 ms	2581 122.7 ms	2427 126.19 ms	–	2576 120.46 ms
power	2066 121.55 ms	2085 129.79 ms	2040 122.01 ms	2063 117.08 ms	2127 117.49 ms	2155 120.91 ms	2089 105.56 ms	2088 115.57 ms	2131 112.28 ms	–

Table 5.5: Boolean search performance results and hits. Each combination of two keywords has been executed against the database

stead of materializing all resulting node references in one run, they could be returned blockwise or one by one. This way, only those nodes will be requested that are actually required by a query, and retrieval can be skipped if it is clear that the remaining references will not be part of the final result.

- The pipelined retrieval could also be used to skip node retrieval whenever a query requests only parts of the result. As an example, a limitation to the first n results means that retrieval can be skipped as soon as those results have been evaluated.

5.5.3 Boolean Search

Boolean search is another basic technique supported by many retrieval systems, and considered in the context of this evaluation. It introduces the operators AND, OR plus NOT, which allow users to exclude or include terms, or combine them in an arbitrary fashion. These operators are commonly used to cut down result sizes and filter unwanted hits from result listings.

Query: Boolean search using XQuery Full Text

```
//*[text() contains text "germany" ftand ftnot "problem"]/ancestor::file
```

Results: Query results for the fully-sized OPAC corpus are depicted in Table 5.5.

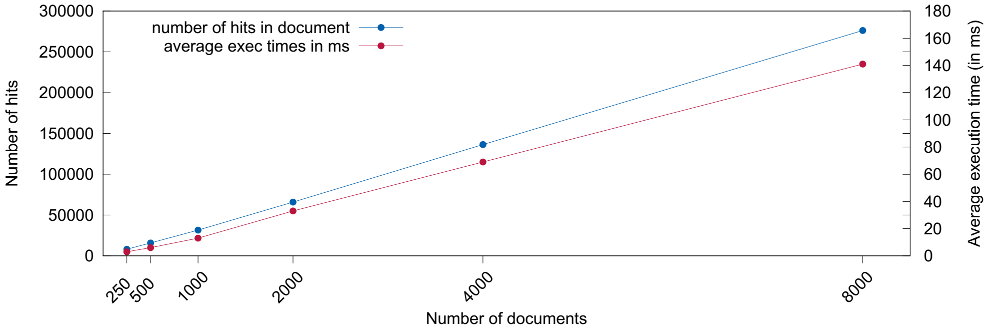


Figure 5.5: Average runtime for the boolean full-text queries, ran against six different sized corpora

Analysis: Compared to the keyword search shown before, we see a *linear* degradation in performance. This again is explained by the fact, that possibly large intermediate results will have to be merged by BaseX in order to produce the result set. Once more the queries perform fast enough regarding interactivity constraints. An upper bound in our example is set by query `change`∧¬(`problem`) yielding 3,486 result nodes in `175.13`ms.

5.6 Summary

The previous observations show that the X-technology stack is ready to cope with state-of-the-art requirements and able to deliver retrieval infrastructure needed to build even complex systems. XML-DBMS may be considered mature enough to drive retrieval systems ready for production.

Yet XQuery with its various extensions is capable of delivering more than just state-of-the-art: due to the hierarchical nature of XML and its ability to contain structured as well as unstructured data, users are able to exploit these characteristics in order to improve the relevance of their search results. The languages's expressiveness can be applied to numerous problems. To give an idea of what kind of questions may be answered, consider a search for documents, that contain the words “substrate” & “transformation” in a maximum distance of at most 4 words, followed by another page containing the word “compound”.

```
(: Exploiting structural and textual proximity. :)
let $words:= ("substrate", "transformation"),
    $following := "compounds"
return
  //*[ text() contains text {$words} distance at most 4 words
       and
        . /following-sibling::*/text() contains text {$following}
     ]/ancestor::file
```

Listing 20: XQuery example exploiting structure and textual proximity

Skilled experts can steer the system from within a single language. XQuery gives transparent access to underlying system components such as the full-text engine (hard to achieve in a traditional general purpose system) and allows implementors to work directly on the underlying data.

All of that can be done in a single domain specific technology stack reducing the complexity of both system components involved and technologies to be mastered by developers.

6 Conclusion

In a nutshell, this thesis was about innovative ways to leverage the powerful infrastructure of modern XML database management systems (XML-DBMS) for cleaner, more efficient and compact application development and system architectures.

XML-DBMS provide sophisticated technology that, today, is generally underestimated and can be used in far more application domains than it is currently the case. Used to full capacity, they can provide lean system architectures with less components involved, a clean and pure technology stack and reduced amounts of code involved.

The thesis was focused on the question in how far the tree-awareness of recent DBMSs can be used to enhance filesystems with database technology. The main goal was to provide means to query the data stored in filesystems and to enhance/combine the data storage and query capabilities of operating systems using XML database technology.

BaseX, a high performance, native XML-DBMS, built on top of a improved XPath accelerator numbering scheme [25], laid the foundation for our work. Throughout the work BaseX was applied to new and unanticipated application domains.

In Chapter 2, we introduced a new XML dialect, the Filesystem Markup Language (FSML), to construct a database view of the filesystem and its contents. FSML provides a uniform view on the filesystem's content and allows developers to leverage the complete XML technology stack on filesystem data. Raw files, along with their metadata exposed in XML, are stored in the database. Together with a representation of the file hierarchy, the database contains all necessary data to model a filesystem.

For *database-unaware applications*, we established a link between DBMS and OS in Chapter 3. We contributed a filesystem in userspace, backed by the BaseX DBMS system. The DBMS is mounted as a conventional filesystem by the operating system kernel. The architecture provides access via the established filesystem interface as well as database-enhanced access to the same data. By doing that, we have demonstrated the possibility of providing legacy filesystem access while storing the data in the database and have it ready to be queried. As the database itself can now be accessed via the filesystem namespace, the system exposes formerly siloed data via the filesystem interface. The concept has been introduced as metadata-aware, deep access to files.

For *database-aware applications*, the architecture reveals an additional declarative API to work on file objects. While it is still possible to use the established and proven interface with an imperative programming style (using system calls and processing byte streams), we additionally can use XQuery to gain semantic, declarative access to file objects (using database queries and processing file objects in XML). That way, we contributed a system architecture that makes it easier for application developers to build content-oriented (data-centric) retrieval and search applications dealing with files and their contents.

The standardized and established XML technology stack, specified by the World Wide Web Consortium (W3C), can now be used without the need to resort to other programming languages and concepts. In doing so, we achieve a low-entry barrier for developers and data. Developers must not learn software-specific retrieval languages or APIs to regain stored data. Data can be (transparently) stored in a conventional Unix filesystem and processed as such (*i.e.*, backups, Unix commands). However, files' inherent metadata is externalized and stored, and is generically accessible via a formerly non-existent declarative API. Relevant content for search tasks is indexed and optimized for later retrieval without additional tools to master and maintain.

In summary, the thesis contributed a facility for representing filesystem data system-wide in a uniform way and, at the same time, offering domain-specific query and processing languages to work with filesystem data at a higher abstraction level. The extended XML database now entails all components needed to act as central system for search and retrieval tasks on heterogeneous file data.

In order to let developers benefit most from the proposed architecture, we contributed a powerful XQuery application framework in Chapter 4. It allows for the development of applications relying on the W3C technology family. Services that profit from a uniform search and retrieval service can now be implemented on a more high-level and generic abstraction layer, while still being able to benefit from full-fledged database support. As a proof of concept, we conducted a complete development cycle for an OPAC (Online

Public Access Catalogue) system in Chapter 5. The implementation revealed that the proposed architecture is ready to drive expert information system that work with distinct data sources using an XQuery-driven development approach.

We consider the discussed techniques as a general blueprint appropriate to design and develop XML/XQuery driven information architectures that work on formerly heterogeneous data sources in a standardized and uniform manner.

List of Figures

List of Listings

List of Tables

Bibliography

[1] **28msec**, *Sausalito: XQuery in the Cloud*, 2012. [Online]. Available: `http://www.28msec.com/documentation/overview` (visited on 02/03/2012).

[2] **G. Athanasopoulos, L. Candela, D. Castelli, P. Innocenti, Y. Ioannidis, A. Katifori, A. Nika, G. Vullo, and S. Ross**, "The Digital Library Reference Model," ISTI-CNR, Tech. Rep., 2007.

[3] **A. Azagury, M. Factor, Y. S. Maarek, and B. Mandler**, "A novel navigation paradigm for XML repositories," *JASIST*, vol. 53, no. 6, pp. 515–525, 2002.

[4] **A. Berglund**, *Extensible Stylesheet Language (XSL) Version 1.1*, 2006. [Online]. Available: `http://www.w3.org/TR/2006/REC-xsl11-20061205/` (visited on 01/23/2012).

[5] **P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner**, "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine," in *SIGMOD Conference*, 2006, pp. 479–490.

[6] —, "Pathfinder: XQuery - The Relational Way," in *VLDB*, 2005, pp. 1322–1325.

[7] **M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska**, "Building a database on S3," in *SIGMOD Conference*, 2008, pp. 251–264.

[8] **T. Bray, J. Paoli, and C. M. Sperberg-McQueen**, *Extensible Markup Language (XML) 1.0*, 1998. [Online]. Available: `http://www.w3.org/TR/1998/REC-xml-19980210` (visited on 01/23/2012).

[9] **S. Burbeck**, *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*, 1987. [Online]. Available: `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html` (visited on 12/19/2011).

[10] **Y. Cai, X. L. Dong, A. Y. Halevy, J. M. Liu, and J. Madhavan**, "Personal information management with SEMEX," in *SIGMOD Conference*, 2005, pp. 921–923.

[11] **M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling**, "Shoring Up Persistent Applications," in *SIGMOD Conference*, 1994, pp. 383–394.

[12] **P. Case, M. Dyck, M. Holstege, S. Amer-Yahia, C. Botev, S. Buxton, J. Doerre, J. Melton, M. Rys, and J. Shanmugasundaram**, *XQuery and XPath Full Text 1.0*, 2011. [Online]. Available: `http://www.w3.org/TR/2011/REC-xpath-full-text-10-20110317/` (visited on 01/23/2012).

[13] **DBIS Group, University of Konstanz**, *BaseX - The XML Database*, 2012. [Online]. Available: `http://basex.org/` (visited on 02/07/2012).

[14] **EMC Corporation**, *Documentum xDB*, 2012. [Online]. Available: `http://germany.emc.com/products/detail/software2/documentum-xdb.htm` (visited on 02/07/2012).

[15] **D. Engovatov, D. Florescu, and G. Ghelli**, *XQuery Scripting Extension 1.0 Requirements*, 2010. [Online]. Available: `http://www.w3.org/TR/2007/WD-xquery-sx-10-requirements-20070323` (visited on 02/03/2012).

[16] **M. J. Franklin, A. Y. Halevy, and D. Maier**, "From databases to dataspaces: a new abstraction for information management," *SIGMOD Record*, vol. 34, no. 4, pp. 27–33, 2005.

[17] **D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr.**, "Semantic file systems," in *SOSP*, New York, NY, USA: ACM, 1991, pp. 16–25. DOI: `http://doi.acm.org/10.1145/121132.121138`.

[18] **Gluster, Inc.**, *GlusterFS - A cluster filesystem capable of scaling to several peta-bytes*, 2012. [Online]. Available: `http://www.gluster.org/` (visited on 02/05/2012).

[19] **B. Gopal and U. Manber**, "Integrating Content-Based Access Mechanisms with Hierarchical File Systems," in *OSDI*, 1999, pp. 265–278. DOI: `http://doi.acm.org/10.1145/296806.296838`.

[20] **J. Gray, A. S. Szalay, A. Thakar, C. Stoughton, and J. vandenBerg**, "Online Scientific Data Curation, Publication, and Archiving," *CoRR*, vol. cs.DL/0208012, 2002. [Online]. Available: `http://arxiv.org/abs/cs.DL/0208012`.

[21] **T. Grust**, "Accelerating XPath Location Steps," in *SIGMOD Conference*, 2002, pp. 109–120. DOI: `10.1145/564691.564705`.

[22] **T. Grust and M. van Keulen**, "Tree Awareness for Relational DBMS Kernels: Staircase Join," in *Intelligent Search on XML Data*, 2003, pp. 231–245. [Online]. Available: `http://www.springerlink.com/content/ltdcgv2t680w/`.

[23] **T. Grust, M. Mayr, J. Rittinger, S. Sakr, and J. Teubner**, "A SQL:1999 Code Generator for the Pathfinder XQuery Compiler," in *SIGMOD Conference*, 2007, pp. 1162–1164. DOI: `10.1145/1247480.1247642`.

[24] **T. Grust, J. Rittinger, and J. Teubner**, "Why off-the-shelf RDBMSs are better at XPath than you might expect," in *SIGMOD Conference*, 2007, pp. 949–958. DOI: `10.1145/1247480.1247591`.

[25] **C. Grün**, "Storing and Querying Large XML Instances," Ph.D. Thesis, University of Konstanz, Konstanz, Germany, 2011. [Online]. Available: `http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-127142`.

[26] **C. Grün, A. Holupirek, M. Kramis, M. H. Scholl, and M. Waldvogel**, "Pushing XPath Accelerator to its Limits," in *ExpDB*, 2006.

[27] **C. Grün, A. Holupirek, and M. H. Scholl**, "Visually exploring and querying XML with BaseX," in *BTW*, 2007, pp. 629–632.

[28] **C. Grün, S. Gath, A. Holupirek, and M. H. Scholl**, "XQuery Full Text Implementation in BaseX," in *XSym*, 2009, pp. 114–128. DOI: `10.1007/978-3-642-03555-5_10`.

[29] **A. Y. Halevy, M. J. Franklin, and D. Maier**, "Principles of dataspace systems," in *PODS*, 2006, pp. 1–9. DOI: `10.1145/1142351.1142352`.

[30] **P. Harvey**, *ExifTool - Read, Write and Edit Meta Information*, 2012. [Online]. Available: `http://www.sno.phy.queensu.ca/~phil/exiftool/` (visited on 02/22/2012).

[31] **A. Holupirek and M. H. Scholl**, "An XML Database as Filesystem in Userspace," in *Grundlagen von Datenbanken*, 2008, pp. 31–35.

[32] —, "Implementing filesystems by tree-aware DBMSs," *PVLDB*, vol. 1, no. 2, pp. 1623–1630, 2008. [Online]. Available: `http://www.vldb.org/pvldb/1/1454237.pdf`.

[33] **A. Holupirek, C. Grün, and M. H. Scholl**, "BaseX & DeepFS joint storage for filesystem and database," in *EDBT*, 2009, pp. 1108–1111. DOI: `10.1145/1516360.1516489`.

[34] —, "Melting Pot XML: Bringing Filesystems and Databases One Step Closer," in *BTW*, Aachen, Germany, 2007.

[35] **IEEE, The Open Group, and ISO/IEC JTC 1/SC22/WG15**, "Single UNIX Specification, Version 3," The Open Group, Tech. Rep.

[36] **Institute for System Programming RAS**, *Sedna Native XML Database System*, 2012. [Online]. Available: `http://www.sedna.org/` (visited on 02/07/2012).

[37] **C. Ireland, D. Bowers, M. Newton, and K. Waugh**, "A Classification of Object-Relational Impedance Mismatch," in *DBKDA*, IEEE Computer Society, 2009, pp. 36–43. DOI: `10.1109/DBKDA.2009.11`. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1545012.1545492`.

[38] **Jesse J. Garrett**, *Ajax: A New Approach to Web Applications*, 2005. [Online]. Available: `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications` (visited on 12/13/2011).

[39] **B. Johnson and B. Shneiderman**, "Tree maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures," in *VIS*, IEEE Computer Society, 1991, pp. 284–291.

[40] **A. Kantee**, "puffs - Pass-to-Userspace Framework File System," in *AsiaBSDCon*, 2007.

[41] **A. Kantee and A. Crooks**, "ReFUSE: Userspace FUSE Reimplementation Using puffs," in *EuroBSDCon)*, 2007.

[42] **M. Kaufmann and D. Kossmann**, "Developing an Enterprise Web Application in XQuery," in *ICWE*, 2009, pp. 465–468. DOI: `10.1007/978-3-642-02818-2_39`.

[43] **M. Kay**, *XSL Transformations (XSLT) Version 2.0*, 2007. [Online]. Available: `http://www.w3.org/TR/2007/REC-xslt20-20070123/` (visited on 01/23/2012).

[44] **Linux Kernel Developers**, *Path walking and name lookup locking*, 2005. [Online]. Available: `http://www.mjmwired.net/kernel/Documentation/filesystems/path-lookup.txt` (visited on 02/22/2012).

[45] **MarkLogic Corporation**, *MarkLogic: The Operational Database for Big Data*, 2012. [Online]. Available: `http://www.marklogic.com/solutions/overview.html` (visited on 01/23/2012).

[46] —, *MarkLogic: The Operational Database for Big Data*, 2012. [Online]. Available: `http://www.marklogic.com/products/overview.html` (visited on 01/23/2012).

[47] **W. Meier**, “eXist: An Open Source Native XML Database,” in *Web, Web-Services, and Database Systems*, vol. 2593, LNCS, 2003, pp. 169–183. DOI: 10.1007/3-540-36560-5_13.

[48] —, *eXist-db Open Source Native XML Database*, 2012. [Online]. Available: http://exist.sourceforge.net/ (visited on 02/07/2012).

[49] **Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok**, “Avfs: An On-Access Anti-Virus File System,” *USENIX Security 2004*, 2004. [Online]. Available: www.fsl.cs.sunysb.edu/docs/avfs-security04/avfs.pdf.

[50] **D. A. Norman**, *Emotional Design: Why We Love (Or Hate) Everyday Things.* Basic Books, 2004, ISBN: 0465051359.

[51] **Pixware**, *Qizx, a fast XML database engine fully supporting XQuery*, 2012. [Online]. Available: http://www.xmlmind.com/qizx/ (visited on 02/07/2012).

[52] **A. Rajgarhia and A. Gehani**, “Performance and extension of user space file systems,” in *SAC*, 2010, pp. 206–213. DOI: 10.1145/1774088.1774130.

[53] **J. Robie, D. Chamberlin, M. Dyck, and J. Snelson**, *XQuery 3.0: An XML Query Language*, 2011. [Online]. Available: http://www.w3.org/TR/2011/WD-xquery-30-20111213/ (visited on 01/11/2012).

[54] **J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon**, *XQuery Update Facility 1.0*, 2011. [Online]. Available: http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/ (visited on 01/23/2012).

[55] **K. H. Rose, S. Malaika, and R. J. Schloss**, “Virtual XML: A toolbox and use cases for the XML world view,” *IBM Systems Journal*, vol. 45, no. 2, pp. 411–424, 2006. DOI: 10.1147/sj.452.0411.

[56] **B. Schandl**, “An Infrastructure for the Development of Semantic Desktop Applications,” Ph.D. Thesis, Universität Wien, Wien, Austria, 2009.

[57] **M. Seiferle**, “Implementing Web Applications Using XQuery,” Master’s thesis, University of Konstanz, Germany, 2012.

[58] **A. Singh**, *A FUSE-Compliant File System Implementation Mechanism for Mac OS X*, 2011. [Online]. Available: http://code.google.com/p/macfuse/ (visited on 01/11/2012).

[59] **R. Singh and H. S. Sarjoughian**, “Software Architecture for Object-Oriented Simulation Modeling and Simulation Environments: Case Study and Approach,” Computer Science & Engineering Dept., Arizona State University, Tempe, AZ, Tech. Rep., 2003.

[60] **J. Snelson, D. Chamberlin, D. Engovatov, D. Florescu, G. Ghelli, J. Melton, and J. Siméon**, *XQuery Scripting Extension 1.0*, 2010. [Online]. Available: `http://www.w3.org/TR/2010/WD-xquery-sx-10-20100408` (visited on 02/03/2012).

[61] **S. St.Laurent**, *Bringing the File System into the File: Making Information More Accessible Through Object Stores*, 1998. [Online]. Available: `{http://www.simonstl.com/articles/filesyst.htm}` (visited on 02/23/2012).

[62] **M. Szeredi**, *Filesystem in USErspace*, 2012. [Online]. Available: `http://fuse.sourceforge.net/` (visited on 12/05/2012).

[63] —, *SSH Filesystem*, 2012. [Online]. Available: `http://fuse.sourceforge.net/sshfs.html` (visited on 02/05/2012).

[64] **J. Teubner**, “Pathfinder: XQuery Compilation Techniques for Relational Database Targets,” Ph.D. Thesis, Technische Universität München, Munich, Germany, 2006.

[65] **Tuxera Ltd**, *Tuxera NTFS-3G and Ntfsprogs*, 2012. [Online]. Available: `http://www.tuxera.com/community/ntfs-3g-download/` (visited on 02/05/2012).

[66] **N. Walsh, A. Milowski, and H. S. Thompson**, *XProc: An XML Pipeline Language*, 2010. [Online]. Available: `http://www.w3.org/TR/2010/REC-xproc-20100511/` (visited on 01/23/2012).

[67] **R. Weir**, *Developing an XML-based file format specification for office applications*, 2011. [Online]. Available: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office` (visited on 01/23/2012).

[68] **E. Wilde**, “Merging trees: file system and content integration,” in *WWW*, 2006, pp. 955–956. DOI: `10.1145/1135777.1135962`.

[69] **E. Zadok**, “FiST: A System for Stackable File-System Code Generation,” PhD thesis, Columbia University, 2001.

[70] **E. Zadok and J. Nieh**, “FiST: A Language for Stackable File Systems,” in *USENIX Annual Technical Conference, General Track*, 2000, pp. 55–70.

[71] **E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright**, "On incremental file system development," *TOS*, vol. 2, pp. 161–196, 2 2006. DOI: `10.1145/1149976.1149979`.

Appendix

```xml
<phrase>
  <search>minor drawback</search>
  <ms>0.45</ms>
  <hits>0</hits>
  <index>
    <ftcount name="minor">2218</ftcount>
    <ftcount name="drawback">450</ftcount>
  </index>
</phrase>
<phrase>
  <search>major deficiency</search>
  <ms>1.25</ms>
  <hits>2</hits>
  <index>
    <ftcount name="major">8553</ftcount>
    <ftcount name="deficiency">368</ftcount>
  </index>
</phrase>
<phrase>
  <search>Stabilisieren konnte sich dieses System</search>
  <ms>42.57</ms>
  <hits>2</hits>
  <index>
    <ftcount name="Stabilisieren">203</ftcount>
    <ftcount name="konnte">18118</ftcount>
    <ftcount name="sich">73862</ftcount>
    <ftcount name="dieses">18674</ftcount>
    <ftcount name="System">28553</ftcount>
  </index>
</phrase>
```

Listing 21: Phrase search query result

```xml
<file name="1896748.pdf" suffix="pdf" st_size="533883">
  <folder name=".1896748.pdf.deepfs">
    <folder name="opacinfo">
      <fact name="pagecount">17</fact>
      <fact name="author">Berthold, Michael</fact>
      <fact name="author">Wiswedel, Bernd</fact>
      <fact name="author">Patterson, David E.</fact>
      <fact name="title">Interactive exploration of fuzzy clusters using Neighborgrams</fact>
      <fact name="town">Konstanz</fact>
      <fact name="publisher">Bibliothek der Universität Konstanz</fact>
      <fact name="year">2005</fact>
      <fact name="format">Online-Ressource</fact>
      <fact name="note">Article</fact>
      <fact name="signature">|004</fact>
      <fact name="language">Englisch</fact>
      <fact name="category">Informatik</fact>
      <fact name="url">http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-65525</fact>
      <fact name="creation-date">November 17, 2004 21:34:22 (UTC)</fact>
      <fact name="modification-date">October 13, 2008 14:42:40 (UTC +02:00)</fact>
    </folder>
    <folder name="fulltext">
      <folder name="pages">
        <folder name="page" number="1">
          <fact name="text">
Interactive exploration of fuzzy clusters using Neighborgrams
Michael R.Berthold - Bernd Wiswedel - David E.Patterson

Department of Computer and Information Science,University of Konstanz,Box M712,78457 Konstanz,Germany

Data Analysis ResearchLab,Tripos Inc.,USA

Abstract
We describe an interactive method to generate a set of fuzzy clusters for classes of interest of a
given,labeled data set.

The presented method is therefore best suited for applications where the focus of analysis
lies on a model for the minority class or for small  to medium-sized data sets.

The clustering algorithm creates one dimensional models of the neighborhood for a set of patterns
[…]
          </fact>
        </folder>
        <folder name="page" number="2">
          <fact name="text">[…]</fact>
        </folder>
        […]
      </folder>
    </folder>
  </folder>
</file>
```

Listing 22: Kopsmedia bibliographic metadata stored as FSML

```
let $phrases:= ("minor drawback",
  "major deficiency",
  "major contribution",
  "particularly strong",
  "special interest group",
  "Related Work",
  "Experimental results",
  "Stabilisieren konnte sich dieses System",
  "We conclude with",
  "I would like to express",
  "major advantage of our",
  "with respect to",
  "As shown in",
  "in contrast to"
)

for $phrase in $phrases
  let
    $find := function($p){//*[text() contains text {$p} phrase ] },
    $hits := count($find ( $phrase)),
    $ms   := util:ms($find ( $phrase))
  order by $hits
return <phrase>{
  <search>{$phrase}</search>,
  <ms>{$ms}</ms>,
  <hits>{$hits}</hits>,
  <index>{
    for $w in tokenize($phrase," ")
    return
      <ftcount name="{$w}">{
        count(db:fulltext(., $w))
      }</ftcount>
  }</index>
}</phrase>
```

Listing 23: Functions to benchmark the phrase search performance