universität
wien

D I S S E R T A T I O N

Building Blocks for Semantic Data Organization on the Desktop

Verfasser

Dipl.-Ing. Niko Popitsch

angestrebter akademischer Grad

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, im August 2011

Studienkennzahl lt. Studienblatt:	A 786 881
Dissertationsgebiet lt. Studienblatt:	Informatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas
Zweitbetreuer:	Univ.-Prof. Dipl.-Ing. DDr. Gerald Quirchmayr

Moji družini

Acknowledgments

Working as a Ph.D. student at the University of Vienna in the past four years was both a magnificent as well as a challenging experience. I wish to thank my supervisor Prof. Dr. Wolfgang Klas and my secondary advisor Prof. DDr. Gerald Quirchmayr for giving me the opportunity and the freedom to pursue my research and to write this thesis as well as for their guidance and all their support.

In addition, I am indebted to my colleagues Bernhard Haslhofer and Bernhard Schandl for the fruitful collaboration. Special thanks go also to my colleagues Stefan Leitich, Robert Mosser, Wolfgang Jochum, Ross King, Jan Stankovsky, Stefan Zander, Elaheh Momeni-Roochi and all the others with whom I had the pleasure to work with during the past years. Thank you all for sharing your excellent ideas with me and for all the fun we had at work.

Above all, however, my gratitude and my love go to my family, my friends and in particular to Maja for motivating and supporting me in the past years.

Abstract

The organization of (multimedia) data on current desktop systems is done to a large part by arranging files in hierarchical file systems, but also by specialized applications (e.g., music or photo organizing software) that make use of file-related metadata for this task. These metadata are predominantly stored in embedded file headers, using a magnitude of mainly proprietary formats. Generally, metadata and links play the key roles in advanced data organization concepts. Their limited support in prevalent file system implementations, however, hinders the adoption of such concepts on the desktop: First, non-uniform access interfaces require metadata consuming applications to understand both a file's format and its metadata scheme; second, separate data/metadata access is not possible, and third, metadata cannot be attached to multiple files or to file folders although the latter are the primary constructs for file organization. As a consequence of this, current desktops suffer, *inter alia*, from (i) limited data organization possibilities, (ii) limited navigability, (iii) limited data findability, and (iv) metadata fragmentation. Although there were attempts to improve this situation, e.g., by introducing semantic file systems, most of these issues were successfully addressed and solved in the Web and in particular in the Semantic Web and reusing these solutions on the desktop, a central hub of data and metadata manipulation, is clearly desirable.

In this thesis a novel, backwards-compatible metadata model that addresses the above-mentioned issues is introduced. This model is based on stable file identifiers and external, file-related, semantic metadata descriptions that are represented using the generic RDF graph model. Descriptions are accessible via a uniform Linked Data interface and can be linked with other descriptions and resources. In particular, this model enables semantic linking between local file system objects and remote resources on the Web or the emerging Web of Data, thereby enabling the integration of these data spaces. As the model crucially relies on the stability of these links, we contribute two algorithms that preserve their integrity in local and in remote environments. This means that links between file system objects, metadata descriptions and remote resources do not break even if their addresses change, e.g., when files are moved or Linked Data resources are re-published using different URIs. Finally, we contribute a prototypical implementation of the proposed metadata model that demonstrates how these building blocks sum up to constitute a metadata layer that may act as a foundation for semantic data organization on the desktop.

Zusammenfassung

Die Organisation von (Multimedia-) Daten auf Desktop-Systemen wird derzeit hauptsächlich durch das Einordnen von Dateien in ein hierarchisches Dateisystem bewerkstelligt. Zusätzlich werden gewisse Inhalte (z.B. Musik oder Fotos) von spezialisierter Software mit Hilfe Datei-bezogener Metadaten verwaltet. Diese Metadaten werden meist direkt im Dateikopf in einer Unzahl verschiedener, vorwiegend proprietärer Formate gespeichert. Allgemein nehmen Metadaten und Links die Schlüsselrollen in fortgeschrittenen Datenorganisationskonzepten ein, ihre eingeschränkte Unterstützung in vorherrschenden Dateisystemen macht die Einführung solcher Konzepte auf dem Desktop jedoch schwierig: Erstens müssen Anwendungen sowohl Dateiformat als auch Metadatenschema verstehen um auf Metadaten zugreifen zu können; zweitens ist ein getrennter Zugriff auf Daten und Metadaten nicht möglich und drittens kann man solche Metadaten nicht mit mehreren Dateien oder mit Dateiordnern assoziieren obgleich letztere die derzeit wichtigsten Konstrukte für die Dateiorganisation darstellen. Dies bedeutet in weiterer Folge: (i) eingeschränkte Möglichkeiten der Datenorganisation, (ii) eingeschränkte Navigationsmöglichkeiten, (iii) schlechte Auffindbarkeit der gespeicherten Daten, und (iv) Fragmentierung von Metadaten. Obschon es Versuche gab, diese Situation (zum Beispiel mit Hilfe semantischer Dateisysteme) zu verbessern, wurden die meisten dieser Probleme bisher vor allem im Web und im Speziellen im semantischen Web adressiert und gelöst. Das Anwenden dort entwickelter Lösungen auf dem Desktop, einer zentralen Plattform der Daten- und Metadatenmanipulation, wäre zweifellos von Vorteil.

In der vorliegenden Arbeit wird ein neues, rückwärts-kompatibles Metadatenmodell als Lösungsversuch für die oben genannten Probleme präsentiert. Dieses Modell basiert auf stabilen Datei-Identifikatoren und externen, semantischen, Datei-bezogenen Metadatenbeschreibungen welche im RDF Graphenmodell repräsentiert werden. Diese Beschreibungen sind durch eine einheitliche Linked-Data-Schnittstelle zugänglich und können mit anderen Beschreibungen und Ressourcen verlinkt werden. Im Speziellen erlaubt dieses Modell semantische Links zwischen lokalen Dateisystemobjekten und Netzressourcen im Web sowie im entstehenden "Daten Web" und ermöglicht somit die Integration dieser Datenräume. Das Modell hängt entscheidend von der Stabilität dieser Links ab weshalb zwei Algorithmen präsentiert werden, welche deren Integrität in lokalen und vernetzten Umgebungen erhalten können. Dies bedeutet, dass Links zwischen Dateisystemobjekten, Metadatenbeschreibungen und Netzressourcen nicht brechen wenn sich deren Adressen ändern, z.B. wenn Dateien verschoben oder Linked-Data Ressourcen unter geänderten URIs publiziert werden. Schließlich wird eine prototypische Implementierung des vorgeschlagenen Metadatenmodells präsentiert, welche demonstriert wie die Summe dieser Bausteine eine Metadatenschicht bildet die als Grundlage für semantische Datenorganisation auf dem Desktop verwendet werden kann.

Table of Contents

List of Figures

List of Tables

Part I

Background

A personal computer (PC) is any general-purpose computer whose size, capabilities, and original sales price make it useful for individuals, and which is intended to be operated directly by an end user, with no intervening computer operator. *Source: Wikipedia, http://en.wikipedia.org/wiki/Personal_computer, 28.08.2009, 09:53h*

Chapter 1

Introduction

A lot has changed since the "era of the personal computer" in the 80's and beginning 90's of the past century: The introduction of the World Wide Web (WWW) and associated networking technologies in 1990 radically changed the way how data is exchanged between computers and between humans that make use of them. The widespread adoption of the hypertext concept and the generic, open and extensible nature of the Web led to a colossal network of explicitly interlinked documents and data that can be traversed and consumed by humans and machines [JeWe04]. The subsequent development of machine-friendly description formats, such as XML and RDF, responded to the need to process, share and integrate these data in an automated fashion (Figure 1.1).

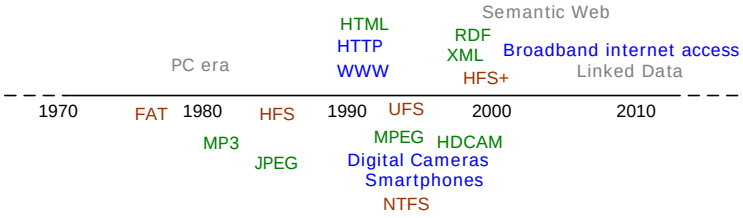


Figure 1.1: An approximate timeline of some key file formats and technologies that did or are expected to sustainably change the way how digital information is produced, processed and shared.

At the same time, the development of processing speed and storage capacities followed Moore's law to a large part. This applies not only to workstation or server hardware but also to end-user products like mobile phones or digital cameras (think, for example, about the number of pixels your mobile phone stores per image now and 5 years ago). The development of mobile devices for digital media production and consumption for the end consumer market led to a rapidly increasing amount of digital media that is created, processed and published on the Web every day.

Where media data originates from. Figure 1.2 sketches how media data are produced and exchanged between devices like digital cameras or scanners, desktop systems like PCs, working stations or laptops and the Web. Currently, most media data are produced by capturing real-world (analog) signals, e.g., by audio or video recording, photography, etc. Many of such media devices are nowadays enabled to directly consume and publish data to/from the Web. For example, modern smartphones are equipped with Web browsers, some digital cameras are enabled to directly upload images to Web applications like flickr or facebook. However, desktop systems still constitute the main hub for editing digital data and its metadata. Although media data

are often captured and consumed on other devices it is likely that they spend some time of their digital life on desktop systems where they are organized, edited and annotated before they are published and shared with other data consumers (e.g., by uploading them to a Web server or by broadcasting them via TV, radio, etc.). Sometimes these media processing and annotation tasks are done by multiple users, possibly in a collaborative fashion, and the media data are exchanged between various desktop systems before being published.

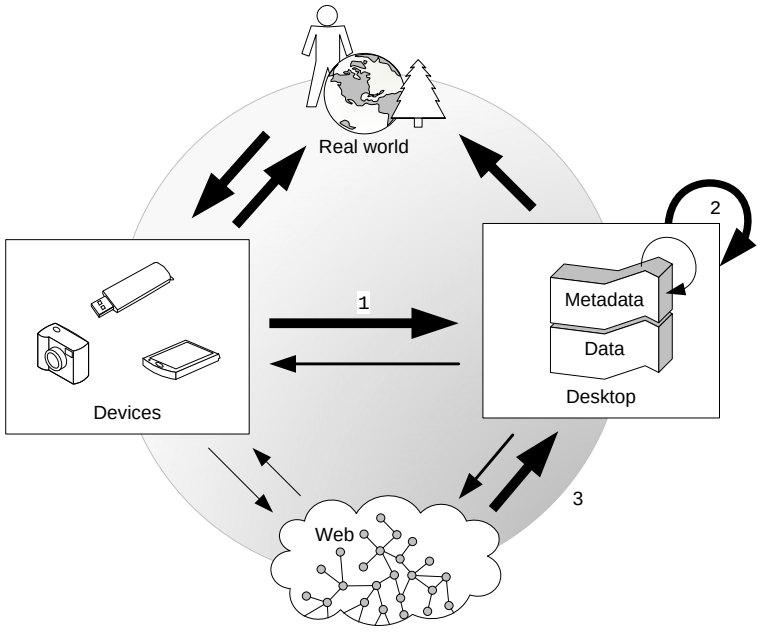


Figure 1.2: A simplified workflow of media and its metadata. Media data stored on today's desktops originates from external devices (1), the desktop computer itself (2) or from remote desktop computers or Web servers (3). This figure emphasizes the central role of desktop computers for data manipulation and metadata creation.

1.1 Data organization on the desktop

Major progress was made in the last decade in the area of Web information management. The Web 2.0 introduced collaborative, user-centered information platforms that resulted in huge social and media networks as well as novel communication possibilities between humans and machines. The emerging Web of Data that is built with technologies from the Semantic Web movement enables the machine friendly publication of interlinked data sets on the Web which results in new ways of data organization and integration. However, data organization on the desktop itself did benefit only partially from these advances so far and still suffers from a number of issues that are partially already solved on the Web:

Hierarchical organization. The predominant paradigm of data organization on desktop systems is the storage of files in a hierarchical file system. All currently prevalent file systems (like NTFS, ext, HFS, FAT; SMB, NFS) use the concept of files as their minimal units of data and rely on hierarchical file organization using some folder (aka directory) concept.

One major problem of exclusively hierarchical data organization is that data items can reside only at one particular location in this hierarchy. For example, folder hierarchies are often used to group semantically related files together as depicted in Figure 1.3 (cf. [QBHK03, Jon07, Sch09]). A file cannot be put into multiple such groups as it may reside only at a single location in the file-tree. Although all modern file systems support some kind of *shortcut* concept that could alleviate this limitation, these concepts are rarely used as they suffer from certain issues that render them unfeasible in practice (cf. Appendix B).

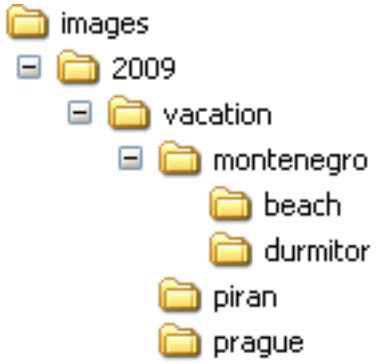


Figure 1.3: Exemplary folder hierarchy for grouping semantically related files.

Limited navigability. Further, only limited navigational functionality is provided by current file systems: traditionally, the strategy of users for locating files on desktop systems was to browse the folder hierarchy for them. But nowadays with the ongoing increase in available storage capacity and data stored on such systems, we face the situation that users simply do not find their local data any more in certain situations. This is one reason for the increasing popularity of alternative navigational concepts as provided, for example, by desktop search engines such as Apple's Spotlight or the Windows Desktop Search (cf. Section 9.3.2, [DB99, SG03, SG04, ABG+06, Jon07]). However, studies showed that even a perfectly accurate desktop search engine might not be sufficient to solve this problem as users often think in contexts (and not in things) and search engines are often only one part of a complex *orienteering* strategy that combines search, teleporting and navigation tasks to re-find digital information in a personal user-space or on the Web [TAAK04].

Limited metadata capabilities. Metadata and links between resources play the key roles in advanced data organization concepts. File-related metadata are very multifaceted and can be used to increase findability and accessibility, enables functionalities like versioning and preservation and is considered as the key-factor in data interoperability [Gia99]. Links are a special kind of metadata that enable the expression of explicit relationships between resources. They further allow the association of more complex metadata descriptions beyond name/value pairs with files or groups of files.

Files themselves are treated as simple byte sequences with some associated low-level metadata by current file systems, as described in Section 3.2. At this level, files "do not impose any formatting or structuring constraints" [VP96] which make the concepts of files and folders (containing files and/or other folders) very easy to understand and use [Ibid.] and many algorithms and data structures were developed in the past to organize these unstructured data very efficiently on storage devices. However, no higher-level metadata whatsoever is accessible on this level in order to improve the interfaces for accessing file data (exceptions are discussed in this work). A general mechanism for the association of arbitrary metadata with files across existing file system boundaries is currently missing on the desktop.

No uniform metadata interfaces. The actual *status quo* for associating higher-level metadata with files is to embed these metadata in the actual file itself, usually in the form of metadata headers. However, this requires all metadata consuming and writing actors to know how to actually access these data, i.e., they have to know how to read/write the actual data format of the file that contains these metadata. Due to the magnitude of different file formats this means a considerable constraint for metadata processing actors. The lack of uniform metadata interfaces consequently results in limited data interoperability and "information fragmentation" [KJ06] on the desktop. As a further consequence, data access based on higher-level metadata or data semantics is currently implemented entirely on the application level.

Lack of stable identifiers. Another current limitation is that stable links between file resources are not well-supported in current file system implementations. The main issue here is that current file systems and operating systems do not provide stable, location-independent identifiers for files: Files are usually identified on an application level via their path in the folder-hierarchy. This identification scheme is neither stable (as files might be renamed or moved) nor globally unique (as the same file paths might exists on many file systems). However, stable and globally unique identifiers play a key role for the stable association of arbitrary metadata and other resources with files which is essential for the implementation of more advanced data organization methods.

1.2 Existing solution strategies

Multiple developments were done in the past to overcome these issues. They share the common goal to enable access to files, groups of files and related resources based on their semantics rather than on their physical organization on some storage device. Such data organization was successfully introduced on an application level in recent years, in particular in ontology and folksonomy-based Semantic Web and Web 2.0 applications. It enables advanced search and navigational features like efficient content-based retrieval or dynamic views based on semantic metadata attributes (e.g., [XKTK03, SKL08]).

Semantic data organization... On a technical level, *semantic file systems* respectively semantic extensions of existing file systems introduce such advanced concepts to the desktop. Semantic file systems enable data access based on a file's (semantic) metadata and its relationships with other files rather than simply based on its location in a hierarchy. Such systems seem like the next logical step in file system development and major projects were started in this direction (e.g., WinFS or BeFS. A survey of major (research) projects in this area can be found in Section 9.3.1). The implementation of semantic data organization features directly on the file system level instead of the application level has many advantages ranging from performance to consistency and access control issues. It would provide a stable technological basis for applications built on top of this layer, disburdening them from the need to implement such basic functionality (cf. [MTX02]). However, file system development is a very complex task that requires a lot of knowledge and resources. Because of their importance in a computer system, file systems are among the most reliable software components in any operating system[1]. It can be argued that the enormous technical complexity is further increased by the need of commercial vendors to stay backward-compatible with previously released file system versions and applications that do not support certain new metadata capabilities. Although much effort has been put into the development of semantic data organization on the file system level, we are still far from a major breakthrough in this area. Current and past semantic file system prototypes are implemented to a large part as virtual file systems that serve mainly as proof-of-concepts. However, the semantic file system research of the past 20 years arguably influenced the actual file system developments of major vendors and a higher number of functional prototypes in this area is clearly desirable to make further progress.

[1] Simply consider that bugs in end-user applications are already unpleasing to users. However, unstable file system components may destroy or corrupt all local data. This is one reason why vendors of file systems put so much effort into file system reliability [Sir05].

...on the desktop. Currently, a large part of the data and metadata processed by various desktop applications is actually enclosed in so-called "data silos". Exchange and integration of these data items with other local as well as with remote applications is difficult or impossible. This *information fragmentation* problem [KJ06], i.e., the lack of data interoperability between the various data sources (e.g., email, calendars, file systems) on the desktop is the core problem addressed by current *Semantic Desktop* research (cf. Section 9.3.3). Semantic Desktop implementations try to increase data interoperability using metadata-rich representations of local resources. The availability of (semantic) file metadata is important for the automatic interpretation of these data. This could not only be exploited for improved data access but also for the integration of such data into existing data sets.

...and in the Web of Data. Further, publishing and sharing file data together with its (structured) metadata would be a valuable contribution to the emerging Web of Data, a global network of semantically related, structured data (even if data access has to be restricted and secured in such a scenario). This step would further open the mentioned "data silos" and would contribute to the ongoing integration of desktop and Web environments.

The central role of metadata

All of the above-mentioned solution approaches put metadata in the center of their strategies for improved data organization, access, processing and integration. Because data stored on desktop systems are predominantly available in an unstructured form (cf. Section 2), associated structured or semi-structured metadata plays a key role for such tasks. This raises the question where these metadata actually originate from.

Some metadata can be automatically extracted from data files or are assigned by the applications that created or recorded them (e.g., GPS-enabled cameras store GPS coordinates embedded in image files). However, these are often of low-level, technical nature and are not *per se* sufficient to implement the above-mentioned strategies (e.g., users do rather not search for GPS coordinates but for the respective names of places located at these coordinates).

Higher level (semantic) metadata that is especially useful for the above-mentioned strategies has to be assigned manually or semi-automatically to a large part[2]. Such metadata that cannot be reproduced automatically require a mechanism that stably associates them with the data items they describe. They should be expressed in standardized formats that can be interpreted by a large number of applications and should be available to local and remote actors via some standardized, uniform access interfaces in order to avoid the above-mentioned "data silos". Note that such metadata also include stable (semantic) relations between local and remote resources. Current file system implementations do not fulfill these requirements as discussed in this thesis.

[2]Note, however, that in the above-mentioned GPS example a software could make use of services such as `http://www.geonames.org/` to convert coordinates into actual place names.

1.3 Thesis and contributions

This thesis is motivated by the question how data stored on local desktops, central hubs of data and metadata manipulation, can be stably associated with semantic metadata, how these metadata can be captured/created and how data and metadata may be shared among multiple desktops and users. The main motivating scenario is to exploit these metadata for improved, semantic data organization on desktop systems.

In this thesis we present a novel, backward-compatible metadata model for the desktop (Section 4). This model is based on external, semantic, file-related metadata and introduces well-established methods from Web and Semantic Web research to the desktop. It allows the stable association of local file system objects, arbitrary complex metadata descriptions and Web resources and by that the integration of their respective data spaces.

Our metadata model relies on stable links between such resources and the central part of this thesis consequently focuses on how to preserve such stability in environments that do not ensure referential integrity. First, we contribute a formal definition of the related *broken link problem* and discuss existing solution strategies (Chapter 5). Then, two solutions (one for the Web of Data and one for the desktop) that are both based on the heuristic comparison of features derived from respective data items (Linked Data resources respectively local files; Chapters 6 and 8) are presented and evaluated. A study of low-level file feature distributions (Chapter 7) provides the fact basis for this work.

Finally, we discuss our own prototypical implementation of the proposed metadata model that is based on these two tools (Chapter 9). Our implementation shows how the proposed model can be used to establish one or multiple semantic metadata layers on a desktop system that could be exploited for semantic data organization as suggested by semantic file system and Semantic Desktop research. In the final Chapter 10 we summarize the presented contributions and discuss limitations and possible follow-up research directions.

Our work is founded on *state of the art* analyses of prevalent file systems (Appendix A), semantic file system and desktop implementations (Section 9.3) and associated topics (Appendices B and C). Based on this research we begin this thesis with definitions of important concepts and technologies (Chapter 2), followed by a discussion of core building blocks for semantic data organization that emphasizes the central role of metadata in this regard (Chapter 3).

Further information about this thesis and its contributions can be found at `http://purl.org/y2/`.

Chapter 2

Definitions

For a better understanding, we start this thesis with some clarifications and informal definitions of technologies, concepts and terms that are used throughout this document. The discussed concepts are fundamental for understanding the present work.

Desktop

Today's personal computers (PCs) have become increasingly powerful and complex systems. Personal computers include desktop computers but also mobile devices such as laptops or palmtops. For the sake of simplicity we will refer to all such systems as *desktop systems* or simply as *the desktop* throughout this thesis, also emphasizing the differences between such local environments and the distributed environment of the Web. Nowadays it is common to store gigabytes of digital data on such systems, use them for working and leisure purposes, for media creation, editing and management, etc. The term *personal computer*, seems less applicable today, as the data stored in such local systems are often authored by more than one single user: documents and media files are regularly exchanged between users, e.g., as email attachments, by networking protocols or via other file sharing infrastructure.

File systems

A central component of the desktop is the file system. File systems are responsible for storing and organizing data and provide methods for finding and accessing them. Although file systems constitute a central interface between data, applications and users, all prevalent implementations rely mainly on simple hierarchical organization schemes that were in principle developed in the early days of personal computers.

A common desktop system often contains more than one single file system. For example, the PC this thesis was written on is equipped with two NTFS formatted hard disks, a floppy disc drive (reading, e.g., FAT32 formatted floppies), a DVD drive (reading, e.g., UDF formatted DVDs) and a CD drive (reading, e.g., ISO 9660 formatted CDs). Sometimes, external storage devices were plugged in (e.g., FAT32 formatted USB memory sticks) and from time to time this desktop was connected with various remote file servers (e.g., via SMB and CIFS). Files were often copied or moved between these various storage devices and thus these data often crossed technical file system boundaries. This is noteworthy as the various file system implementations differ in their support for various metadata features which results in major interoperability issues as discussed in this thesis. A detailed discussion of various file systems and their metadata capabilities can be found in Appendix A.

Semantic file systems and Semantic Desktops

A semantic file system extends common file systems by alternative access possibilities: access to files stored in such systems is not based only on their location in a hierarchy but rather on their semantics or on the values of certain metadata attributes. A survey of important commercial and research projects in this area can be found in Section 9.3.1.

A Semantic Desktop is a system that extends the desktop's graphical user interfaces in order to improve data organization and personal information management (PIM) and to connect local and Web data using a unified data model and common interfaces [Sch09]. Often, Semantic Web technologies are used for metadata and link representation and processing. A survey of Semantic Desktop projects can be found in Section 9.3.3.

Virtual folders

Virtual folders are a common concept that is used in current desktop systems to overcome the limitations of pure hierarchical data organization. Such folders are collections of files that are computed on demand, they are so to say implementations of the corresponding file system's folder interface. The content of a virtual folder is not a fixed list of files stored in one particular file system folder but rather consists of shortcuts to files stored somewhere in the local file system or even in databases or some other repository. This concept is quite popular in software engineering and is considered wherever hierarchical organization schemes are in use, e.g., in LDAP servers, email clients, music players and also for most semantic file system prototypes. All modern operating systems contain implementations of the virtual folder concept (however with slightly different semantics) that can be used to organize file data in a location-independent fashion. Mac OS X *smart folders* and Windows 7 *libraries* are two examples for such virtual folder implementations. Virtual folders have further been extensively used in Unix-based file systems to increase system uniformity and utility (e.g., devices are listed in the virtual */dev* folder that is basically provided by the pseudo file system *devfs*). For client programs, such folders are normally indistinguishable from ordinary folders, but unlike those, virtual folders do not have to be explicitly created before they may be accessed [GJSJ91].

However, virtual folders as implemented in today's desktops cannot cure all problems of hierarchical file organization. They do not constitute real relationships between files but rather collections that are populated by dynamically querying the local file system. This means on the one hand that the information which files are related with each other is not explicitly modeled and can therefore not easily be re-used by other applications. On the other hand, this concept has been criticized because of usability concerns: regular desktop users are not used to work with shortcuts and therefore often confuse them with real files. Further, user-defined virtual folders would require easy-to-use interfaces for formulating complex queries which is a hard usability challenge. And last but not least, such a concept requires the re-execution of the corresponding queries whenever the data basis (e.g., the file system) changes which might result in high computational and I/O costs.

Forks, Alternate Data Streams and Extended Attributes

In file systems that support forks (or *alternate data streams* as they are called in NTFS), a file consists of a *data fork* (containing a file's data) and one or many additional *resource forks* (Figure 2.1). Forks allow it to associate large amounts of additional (meta) data with a file object. This concept was introduced by Apple with their HFS file system (Section A) and is supported by many current file systems albeit with varying semantics. File forks and alternate data streams are discussed in more technical detail in Appendix A.

In some file system implementations, forks can be referenced through extensions of the normal file system namespace which allows it to associate one file handle with each fork and apply common file system operations (like copy, delete, etc.) to them. On most file systems, the size of forks is restricted only by the maximum file size. See Appendix A for a detailed discussion of file forks in the context of respective file system implementations.

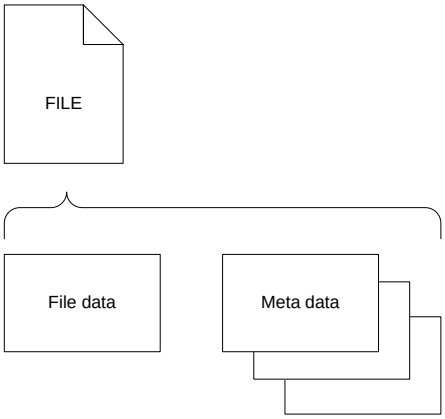


Figure 2.1: File fork / alternate data stream concept. A file is seen as a logical container of one (mandatory) data fork and multiple (optional) resource forks. Usually, these resource forks are used to store metadata about the file data.

Extended attributes. Extended attributes constitute an alternative way to store custom, file-related metadata in a more structured way. Originally, they were introduced by Microsoft and IBM with their file system HPFS, developed for the OS/2 operating system (Section A). Extended attributes are implemented differently on most file systems but in principle, they constitute a mechanism for associating name/value pairs with a file. In some implementations, the range of the attributes can be restricted (typed attributes), in others, the domain can be specified (e.g., with a namespace-like mechanism in the *ext file systems*). In contrast to forks, extended attributes are normally restricted in size (although on some file systems like NTFS they are actually implemented by making use of the alternate data stream mechanism). Extended attributes are mainly used to store lower management data as well as security related metadata like access control lists. Extended attributes are explained in the context of various file system implementations in Appendix A.

Problems with forks and extended attributes. The major problem with forks and extended attributes is that they are lost when files are either copied to file systems that do not support these features (e.g., FAT) or when files are processed by applications that are unaware of these concepts (e.g., when a file is being sent via email or when it is processed by some backup program).

We found two major strategies that were proposed to preserve metadata stored in forks or extended attributes on file systems that do not support these features. First, some systems use the strategy to store this information in regular but hidden files (e.g., Apple uses files prefixed by "._" for this purpose). Naturally, this means that data and metadata are not synchronized any more as there is no modeled connection between these two files. If one file is moved to another location on the volume, the other file will remain where it is and it is not possible to associate them any more.

A second proposed strategy is to automatically bundle files and their metadata (e.g., in zipped files) when they are copied/moved to volumes that do not support such metadata capabilities [Gia99]. This would strongly bind data and metadata together, however, such a strategy would result in major interoperability and performance issues.

For these reasons, forks and extended attributes are mainly used for storing non-critical and automatically reproducible metadata (such as icon positions or image thumbnails).

The Web

The World Wide Web (or simply *the Web*) can be seen as a huge information space [JeWe04] that is used by billions of people to publish and consume digital information. A large share of the documents and media published on the Web originates from desktop systems. Often, desktop systems are either the origin of these data or act as the first contact point for importing such data from external devices (e.g., digital cameras, scanners, etc.) as depicted in Figure 1.2. In particular, they act as *working stations* (sic!) for the creation and manipulation of such data before it is published on the Web. Much highly valuable metadata originates from such data manipulations, metadata that is a key for the efficient organization and integration of these data.

The Web 2.0

Web 2.0 is a popular buzzword these days. Well-known applications like facebook, youtube or twitter (cf. Figure 2.2) are known to have very large user communities and are regularly discussed in the media. A reasonable definition of the term Web 2.0 is given by Greaves:

> "It refers to a class of Web-based applications that were recognized *ex post facto* to share certain design patterns." [Gre07].

Typical Web 2.0 applications are rich Web applications that usually rely strongly on collaborative user contributions and annotations. Internally, most Web 2.0 applications are "based on flexible and lightweight data models that interlink their core elements (e.g., user profiles, wiki articles or blog posts) using hyperlinks [...]" [PMP10].

Figure 2.2: Some popular Web 2.0 applications.

Identifying, referencing and accessing things on the Web

In human communication, *names* are used to identify things. On the Web, *URIs* (Uniform Resource Identifiers) [BLFM05] are generally used for this purpose. In [HH08], the authors describe two distinct kinds of relationships between things and their names, namely *reference* and *access*. While *access* means that a name provides "a causal pathway to the thing" itself, *reference* is always part of a communication act and

is used to "mention the thing" [Ibid.]. Most things that are referred to by names are not *accessible* (e.g., the city of Vienna), so they have to be *referenced* when something should be stated about them. An example for such a reference could be a standard Web-page describing Vienna, identified, e.g., by the URI `http://example.org/Vienna`.

URLs and URNs. Two common special cases of URIs are URLs (Uniform Resource Locators) and URNs (Uniform Resource Names) [BLFM05, Arm01]. URLs provide means for locating and *accessing* a resource using some access mechanism that is specified by the URI scheme (e.g., HTTP, FTP, etc.). URNs can be seen as URIs conforming to the special `urn:` URI scheme. They are used only to name things and do not provide information about how to access them. Thus, URIs are used to identify both, accessible and non-accessible *resources*, URLs are used to *access* digital resources on the Web, URNs allow one to *reference* some (any) thing in this world that can be named.

Representations and content negotiation. Successfully accessing a resource that is identified by an HTTP URIs returns a *representation* of this resource (this is often referred to as *dereferencing* the URI) [BLFM05]. Things may have multiple representations (for example, an HTML representation and an RDF graph, both describing Vienna) that are accessible at the same URI. Content negotiation is an HTTP mechanism that allows clients to specify some properties (in the form of HTTP headers) that describe the *representation* of a thing they expect when accessing a URI [FGM+99]. It can, for example, be used to specify a sorted and weighted list of mime-types the returned resource representations should conform to. The server hosting the respective resource may then select a proper representation for answering such an HTTP request. By this, the client is able to retrieve different representations of a thing by accessing the same HTTP URI.

Ambiguity. However, there is also the case that a URI refers to multiple things: while *access* should be arguably unambiguous to allow for machine-processable retrieval, *referencing* things is unavoidably ambiguous as it is part of a communication act using descriptions of the referred thing [HH08]. Such descriptions are inherently ambiguous (for example, when using the name "Vienna" in a natural-language conversation, this may refer to the city, the city area, etc.). These ambiguities are at the very core of the ongoing, so-called "identity crisis" (aka "URI crisis") and the W3C "httpRange-14" discussions of (Semantic) Web experts about what a URI really refers to. For example, does `http://www.cs.univie.ac.at/Niko.Popitsch` refer to me as a person or to my Website? Such URIs (that refer to multiple things) are called *"overloaded"* and there is an ongoing discussion about their usefulness. The current strategy of the W3C TAG[1] regarding overloaded URIs is to use the HTTP 303 ("see other") mechanism to distinguish between accessible things and other resources. However, this method is criticized as it arguably misuses the HTTP code 303 as discussed in [HH08] where the authors propose the usage of special RDF properties in combination with hash URIs to solve this problem. Anyway, the W3C strategy seems widely accepted in the Semantic Web community and is currently propagated in the context of the evolving Linked Data initiative. A detailed discussion of this issue is beyond the scope of this work, it is however important to keep the fundamental discussions about how things can be named/referenced and accessed in mind as this is at the core of the whole Semantic Web idea that strongly influenced the present work.

The Semantic Web

Parallel to the Web 2.0, the so-called Semantic Web is evolving over time. The Semantic Web [BLHL01] co-exists with the traditional Web (the "Web of Documents") which is predominantly consumed by human users. The core Semantic Web idea is to publish data in a machine friendly way by including information (in a standardized format) that describes the *meaning* (semantics) of the published information, thereby enabling

[1] Cf. `http://www.w3.org/2001/tag/doc/httpRange-14/2007-05-31/HttpRange-14.html`

machines to automatically interpret these data. An often read statement is that Web 1.0 (aka the *document Web*) is about linking documents, Web 2.0 is about linking people and the Semantic Web is about linking data [Hau08].

RDF

A key technology of the Semantic Web is the *Resource Description Framework (RDF)* [KeCeMe04]. RDF is a generic, graph-based data model that enables the description of resources (basically anything can be a resource, this includes non-digital content) and their relations in the form of so-called *triples*. A triple is the core building block of RDF graphs and consists of a *subject*, a *predicate* and an *object*. Subjects and objects reference resources via URI references (objects may also be simple *literals*). Predicates describe how a subject and an object are related, and are also represented by URI references. A set of triples can then be interpreted as a graph by making use of these URIs.

Blank nodes. An RDF graph may also contain *blank nodes* (bnodes) that are not identified by an URI and are no literals. Bnodes are convenient as no "name" is required for them, they do however raise a number of issues when it comes to querying, publishing, reasoning or merging of RDF graphs that are discussed in this thesis where appropriate.

RDF links. Triples that semantically relate two resources with each other are also called *RDF links*. They can be used for the integration of multiple data sets and are the fundamental building blocks of the Web of Data [BCH08].

For example, an RDF triple could state that two persons (each referenced by some URI) are related by the fact that person *A* knows person *B*. Another example would be a triple stating that some document *X* is a precursor of another document *Y*. Let's imagine that those two facts ("*A knows B*" and "*X isPrecursorOf Y*") are expressed in two separated RDF graphs. An RDF link may now connect these two data spaces by, e.g., stating that person *A* is the *authorOf* document *X* which lays the foundation for the integration of these two data sets.

Semantic vocabularies. RDF, however, is just a graph-based data model and does not define the "meaning" of the expressed resources itself. For this, vocabularies and ontologies can be defined using languages like RDF Schema (RDFS) or the Web Ontology Language (OWL) [BeGe04, SeWeMe04]. Such knowledge representation languages differ in their expressiveness and in their complexity, but in principle they enable the modeling of a domain of interest by a formal definition of its concepts and relationships. By this, semantic and machine processable "vocabularies" can be defined that enable machine actors to "speak" with each other (e.g., because they want to exchange or integrate their data). We discuss some semantic vocabularies in Appendix C.

SPARQL. RDF graphs can be queried using the graph query language SPARQL [PeSe08]. SPARQL allows the specification of *graph patterns*, sets of triple patterns. A triple pattern is basically a triple that may contain variables in the subject, predicate or object position. These variables are then "bound" to RDF terms (IRIs[2], bnodes, literals) of the query graph during the query execution. SPARQL further allows grouping of such basic triple patterns and enables querying for optional patterns as well as for conjunctions or disjunctions of such patterns. A SPARQL query returns different types of result sets depending on the used query form (SELECT, CONSTRUCT, ASK, DESCRIBE).

[2]Internationalized Resource Identifier (IRIs) are generalized URIs that may be expressed in extended character sets while URIs are restricted to the ASCII character set.

For the CONSTRUCT query form, for example, the client has to provide an additional set of triple patterns. The variables in these patterns are replaced when the query is executed and the resulting graph is returned. A SPARQL CONSTRUCT query can thus be seen as a graph transformation function as it transforms the original RDF graph into the query result graph.

RDF serializations. An RDF graph can be serialized in several syntaxes, for example, in *RDF/XML*, a W3C-recommended[3] syntax for encoding RDF graphs as XML documents. Other widely-used syntaxes are *Turtle*[4] or *N3*[5]. Recent research brought a focus on the Linked Data style of publishing RDF on the Web that is described below.

Linked Data

In 2006, Tim Berners-Lee outlined simple guidelines for publishing data on the Web[6]. This set of best practices for publishing and connecting structured data on the Web is known as "the *Linked Data* principles". The goal of Linked Data is the creation of a huge "database" of interlinked data descriptions stemming from different data sources. This emerging global data space of structured metadata published in machine-friendly formats is also called the *Web of Data*, in contrast to the regular *Web of Documents* that mainly consists of semi-structured metadata intended for human consumption. Linked Data is technologically based on *URI*, *RDF*, *HTTP* and *RDF links* as described above [BHBL09]. The mentioned Linked Data principles are:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up ("dereference") those names.
3. When someone looks up a URI, provide useful information (i.e., a structured description), using the standards RDF, SPARQL.
4. Include links to other URIs so that they can discover more related things.

Linked Data and content negotiation. The core idea of the third Linked Data "principle" is that dereferencing an HTTP URI (i.e., accessing this URI using an HTTP GET request) leads to different *representations* (descriptions) of the respective resource that serve particular purposes such as human or machine consumption. Technically spoken, Linked Data services usually return either HTML or RDF resource representations. The decision what representation is returned depends usually on content negotiation based on HTTP header settings. Accessing a Linked Data source with a regular Web browser consequently returns an HTML representation of the data, accessing it via some API that allows manipulation of the sent HTTP headers might return other representations. The following cURL[7] request, for example, returns an RDF representation of a Linked Data resource in the DBpedia data set[8], accessing the same URI with a regular Web browser leads to the according HTML representation.

```
curl -L -H "Accept: text/rdf+n3" http://dbpedia.org/resource/Spellbound_%281945_film%29
```

[3] See http://www.w3.org/TR/rdf-syntax-grammar/
[4] The *Terse RDF Triple Language*, see http://www.w3.org/TeamSubmission/turtle/
[5] See http://www.w3.org/DesignIssues/Notation3
[6] http://www.w3.org/DesignIssues/LinkedData.html
[7] http://curl.haxx.se/. Note that the *-L* switch tells cURL to automatically follow HTTP redirects.
[8] http://dbpedia.org/

Linked Data and semantic vocabularies. It is recommended, for Linked Data, to rely as much as possible on existing, widely accepted RDF vocabularies such as FOAF, SIOC, vCARD or Dublin Core (these vocabularies are briefly described in Section C.2) but naturally anyone can publish new vocabularies/ontologies as required [BHBL09]. In a summary, one could state that Linked Data "re-uses and extends the Web infrastructure with technologies that allow to represent, transport, and access raw data over the network. In comparison to the traditional, document-centric Web it comprises the significant improvement that it associates resource identifiers (URIs) with structured descriptions that are represented in a unified format (RDF) and can be accessed by de-referencing their URIs." [PS10].

Linked Open Data. The *Linking Open Data Project*[9] is paying respect to these principles and constantly interlinks more and more publicly available data sets since 2007, resulting in the so-called *Linking Open Data Cloud* (Figure 2.3). The content of the interlinked data sets is very diverse, ranging from data about

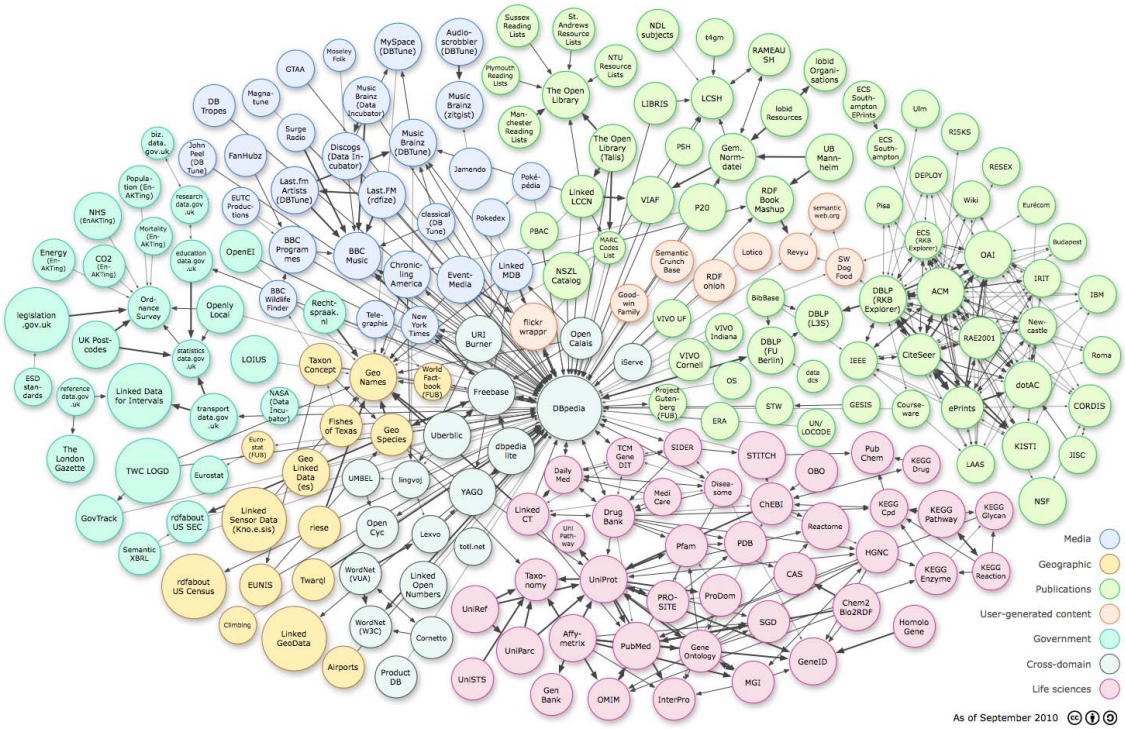


Figure 2.3: Linking Open Data cloud as of September 2010. Arrows represent links between data sets, the stroke size roughly corresponds with the number of links. Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/

geographic locations, books and scientific publications to films, music, TV, drugs, proteins, clinical trials and governmental data. Major institutions such as the Library of Congress, the BBC, Thomson Reuters or the UK government publish some of their data as Linked Data. One central data set of this cloud is DBpedia, the structured version of Wikipedia [BLK+09, ABK+07].

[9] See http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData

Structuredness of data

One way to look at data is to consider its grade of "structuredness": this can range from completely unstructured data that does not follow any predictable schema (like, e.g., text documents or images), to highly structured data as found in databases. Semi-structured data are in-between these two extremes: here the data are somehow structured, but the structure is not always explicit and often not as rigid as in fully structured data [Abi96, Bun97].

Most data on the Web is semi-structured, for example, in the form of XHTML documents. Some consider the Web even as "a collection of semi-structured multimedia documents in the form of Web pages connected through hyperlinks" [LSC02]. This hypertext-layer interrelates Web documents and references unstructured data like images or video files, the hyperlink structure as well as the textual context around these links in HTML documents are exploited by many current algorithms and tools concerned with information extraction, integration or organization on the Web.

Various emerging technologies as well as the increasing amount of Linked Data published on the Web of Data indicate that the structuredness of data on the Web is likely to increase in the near future. *RDFa*, for example, is a "serialization syntax" for embedding an RDF graph into a markup language, e.g., an XHTML (XHTML+RDFa) document. It defines a set of XHTML attributes that allow users "to augment visual data with machine-readable hints" [AeBeMePe08]. By this it is possible to embed machine-readable, structured metadata into XHTML pages, as depicted below. Listing 2.1 shows how to mark-up a regular XHTML file with FOAF (Appendix C.2.2) metadata using RDFa. Automatic parsing[10] of this XHTML file would, for example, easily allow the creation of a *Person* object with associated phone number and email address.

Listing 2.1: RDFa example, cf. [AeBeMePe08]

```
<div typeof="foaf:Person" xmlns:foaf="http://xmlns.com/foaf/0.1/">
    <p property="foaf:name">
        John Doe
    </p>
    <p>
        Email: <a rel="foaf:mbox" href="mailto:john@example.com">john@example.com</a>
    </p>
    <p>
        Phone: <a rel="foaf:phone" href="tel:+1-800-555-1234">+1 800.555.1234</a>
    </p>
</div>
```

Simple alternatives to RDFa for embedding structured, semantic metadata into HTML pages are *Microformats*[11] and *Microdata* [He10]. Microformats are a set of simple data formats inspired by existing and widely adopted metadata standards. An example is the *hCard* standard that provides means to integrate vCard descriptions into (X)HTML, Atom or RSS documents.

Microdata are, in contrast to this, not domain specific and introduce a simple set of attributes that can be used together with existing HTML attributes. Recently, the three big Web search engine providers (Microsoft, Google and Yahoo!) published a set of schemas for expressing microdata that are indexed by their crawlers at http://schema.org/.

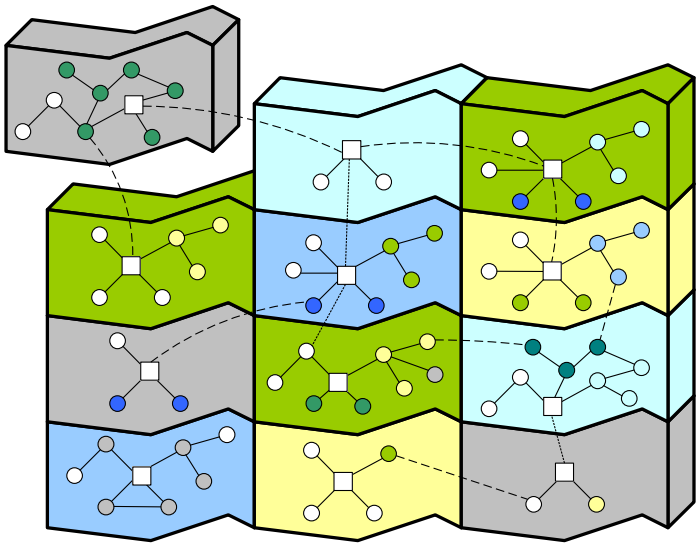
In contrast to the Web, data stored on the desktop such as text documents, images or movie files are predominantly unstructured. No hypertext-layer that models relations between these data is available. The lack of structure in these data emphasizes the role of metadata for processing and organizing them.

[10] The *Gleaning Resource Descriptions from Dialects of Languages* (GRDDL) mechanism can, for example, be used for parsing RDFa from XHTML documents. GRDDL describes method for extracting information from XML documents using XSL transformations: http://www.w3.org/TR/grddl/

[11] http://microformats.org/

Part II

Metadata model



Chapter 3

The role of metadata and links

Generally metadata has many forms and functions. It can be used to increase findability and accessibility, enables functionalities like versioning and preservation and is considered as the key-factor in data interoperability. As described in the introduction, metadata and links between resources play a central role in the implementation of semantic data organization. File-related metadata are very multifaceted and include, e.g., keywords in a document, access control lists, the last backup date of a file, a thumbnail of a video, the color depth of an image, the position of an icon associated with a file and many more [Gia99]. Such metadata describes various aspects of a respective file system object.

Links between such file system objects, on the other hand, enable the explicit modeling of (semantic) relations between data stored on desktop systems. They may further relate local and remote resources that are available, for example, in the Web of Data. Such links can be considered as a special kind of file-related metadata that annotate the file system objects they link from/to.

In this chapter we describe the central role of metadata and links for the implementation of semantic data organization on the desktop. We start by outlining the major building blocks of such a solution, introduce a pragmatic categorization scheme for metadata and give examples for the various categories. Finally, we conclude this chapter by focusing on links between local and remote resources and provide several examples for links on the desktop.

3.1 Core building blocks

Our state of the art analyses of research prototypes and commercial projects in the areas of file systems (Appendix A), semantic file systems (Section 9.3.1) and Semantic Desktop applications (Section 9.3.3) led to the identification of the following core building blocks required for implementing semantic data organization on the desktop. Throughout this thesis we discuss these core building blocks in detail and report on the author's contributions in the respective areas.

Methods to associate arbitrary metadata with file system objects. This includes the possibility to link file system objects with simple and complex semantic metadata records, with other file system objects as well as with remote resources. We discuss the role of semantic metadata and links in detail in this chapter (Sections 3.2 and 3.3) and propose a new model for the representation of such metadata in Chapter 4.

Methods to preserve link integrity. The major part of this work is devoted to the problem of preserving the integrity of links from local files to other files and to local and remote resources such as metadata records (Chapter 8) or Linked Data (Chapter 6).

Methods to generate file-related metadata. Semantic data organization depends to a large part on the availability of metadata. As discussed above, file-related metadata and links play a central role. Such metadata can be created (semi-) automatically or manually and we elaborate briefly on this topic by discussing some emerging, promising methods for metadata generation (Section 4.6.1). We then present a prototypical system that enables users to create a semantic hypertext layer on the desktop that describes and interlinks local and remote resources (Chapter 9).

Interfaces to access and exploit these metadata. File-related metadata can be exploited in many ways including search, navigation and data integration tasks. In Chapter 9, we present a prototypical implementation of our proposed metadata model that demonstrates some of the advantages of the availability of such metadata.

In the following, we first analyze what kinds of metadata and links do occur on the desktop, before discussing the first of these building blocks in Chapter 4.

3.2 Categories of metadata

Metadata can be classified using many different schemas/categories. Some classifications focus on the way how metadata are used (e.g., refer to [GGWe05] for an example classification into the categories: *administrative*, *descriptive*, *preservation*, *technical* and *use*), others on the way they are created and again others on structural properties of the data itself. For this thesis, a simple categorization scheme was chosen that classifies metadata on a very abstract level describing the different ways how these data can be accessed. This categorization scheme distinguishes between *implicit* and *explicit*, *intrinsic* and *extrinsic* and *internal* and *external* metadata (Figure 3.1).

3.2.1 Internal and external metadata

Internal metadata

Internal metadata are stored as part of the data file the metadata are about. Examples include ID3 tags in MP3 files, Exif data in JPEG images, keywords in MS-Word files or XMP tags in a PDF file. The actual schema used for the representation of internal metadata is determined by the respective file format. Besides fixed and strict schemas, some metadata standards, such as XMP, additionally support the inclusion of custom (sometimes even schema-less) metadata. We briefly discuss some popular standards for storing internal metadata in Appendix C.

Internal metadata has the major benefit that data and metadata are closely linked together in a single file which means that low-level I/O routines treat both in a uniform way. Thus no special measures need to be taken for making sure that data and their metadata remain associated. Basically three main mechanisms are used for storing internal metadata in file system objects:

i) By attaching simple name/value pairs using *extended attributes*

ii) By associating possibly large metadata descriptions using *file forks* and

iii) By storing internal metadata in file headers.

Extended attributes and file forks are rarely used by applications as a general metadata mechanism because of their limitations that are discussed in Section 2 and in more detail in Appendix A. These methods are, however, used to associate easily reproducible metadata such as image thumbnails or fingerprints with files.
Internal metadata in file headers is the predominant way of storing file-related metadata. This method is, however, not applicable in all situations. Some file formats, e.g., plain text files, do not support internal

metadata at all. Further, currently prevalent file systems include no concept for attaching internal metadata to file folders. Another issue is that access to metadata stored in file headers requires knowledge about the actual file format which is a practical limitation for metadata processing applications due to the magnitude of existing file formats.

External metadata

External metadata are stored "outside" of the file they describe and require some method to keep the association between data and metadata of a file stable. For example, when some metadata repository references local files via their file path, these references have to be updated when the files are moved or renamed. Further, when such a file is copied, its metadata have to be copied as well. If, for example, an email client sends a mail with an attachment and does not attach the file's external metadata in some form, this information would be lost for the receiver of the attached file. Although the explicit preservation of this relation between a file and its metadata constitutes a major challenge, external metadata also comprises some considerable advantages over internal metadata:

Separate access. The possibility for accessing data and metadata separately (by low-level I/O routines) results in improved performance, e.g., when external metadata indices are built or when metadata are updated by external services. Performance is for obvious reasons a key issue in a file system. Further, access to a file's data and its metadata could also be controlled (restricted) separately.

Expressiveness. Separating data and metadata records would enable association patterns beyond simple 1:1 relationships. It would, for example, be possible that one metadata record describes a whole set of files or that a single file is associated with multiple records that describe different aspects of it.

Uniform access and interoperability. External metadata enables the development of a general, generic representation model for metadata that would be shared by all files independent of their internal file format. Access to such metadata could be done via one single access interface respectively some standardized API. Such a generic representation model would make interoperability much easier: applications that are unaware of a certain file format could still read and write a file's metadata if they understand the semantics of the schemas used for metadata representation.

As one contribution, we present a metadata model that is based on external file-related metadata in Chapter 4 of this thesis.

3.2.2 Extrinsic, intrinsic, explicit and implicit metadata

One way to generate file-related metadata is by extracting them from the file data. We call metadata *intrinsic* if it can actually be extracted directly from a file's data by some extraction algorithm. Respectively, we call metadata *extrinsic* if this is not possible in an automated fashion, i.e., if human intervention is required to assign these metadata to a file (cf., [PR03, Mar04]).

Metadata that was actually extracted from (or assigned in some other way to) a file is called *explicit*: it is stored in some form (as internal or external metadata) and can be directly accessed by applications or users without any further extraction step. *Implicit* metadata was not yet extracted from a file (by an algorithm or a human user) and is thus not directly accessible. Only intrinsic metadata can be made explicit automatically. In consequence, the storage requirements for explicit extrinsic and explicit intrinsic metadata are different: storing intrinsic metadata is basically a caching operation with the goal of faster access. However, when such metadata are lost, they can automatically be derived from the file data. In contrast, lost extrinsic metadata cannot be reproduced, in some cases not even by the human being that created it in the first place.

[1] This refers actually to manually assigned tags that are, for example, stored in the data files of a photo management software.

	external	internal
explicit	thumbnail	Exif header
implicit	-	color distribution

(a) Explicit and implicit metadata

	external	internal
extrinsic	tags[1]	keywords in document header
intrinsic	thumbnail	number of pages

(b) Extrinsic and intrinsic metadata

Figure 3.1: Metadata categories and examples.

Sometimes, it is not straightforward to decide whether a particular metadata item should be considered as extrinsic or intrinsic. Consider, for example, the *temperature* that prevailed in the situation when a particular digital photograph was taken. If the camera had a sensor it would probably store the temperature value in the image headers as internal metadata and as this information is not directly extractable from the image data it would be considered as intrinsic. However, even in the absence of such a sensor, the temperature could still be assigned automatically by an application that exploits time and location information stored in the Exif headers of the photograph and queries an online weather database for this information [NHW+04]. In this case, the metadata would be considered as extrinsic; however, this depends on the availability of such an algorithm and the external data sources it makes use of. Figure 3.1 depicts some real-world examples for the described metadata categories.

3.2.3 Special metadata

In the following we discuss further categories of special file-related metadata that occur on the desktop and explain how they fit into our proposed categorization scheme.

Filesystem-level metadata

In this category we summarize metadata that are mainly used by the file system itself for housekeeping tasks, including i-node numbers, pointers to other metadata blocks, etc. This kind of (file system dependent) metadata is implemented by all existing file systems but due to its low-level semantics, it is not suitable for implementing a semantic data organization method. Furthermore it is hard for higher-level applications to access these metadata and manipulations should be done by the file system only for obvious reasons. We will not consider these metadata further in this thesis.

Low-level management metadata

Metadata of this category is used by all current file systems and includes meta information like *file byte size*, *timestamps* (e.g., file creation time), *type* (folder, file, socket, etc.) but also *file names*, *file extensions* or various flags indicating, for example, whether a file is visible or hidden. These metadata are partly intrinsic (e.g., the file size) and partly extrinsic (e.g., the file name) and we consider all of them as external metadata that are stably attached to the file system objects they describe by the file system. All current file systems support this kind of metadata even though they differ in the mechanisms for managing them as well as in the exact list of metadata attributes they provide (e.g., not all file systems store the last access time of files, a very valuable information for time-related functionality in data organization). These metadata may be accessed conveniently via high-level APIs and are therefore also used by most file-handling applications, including file explorer applications that are sometimes counted as part of a file system itself (e.g., the Windows *Explorer* or the Mac *Finder* application). The file system has universal access to these properties, keeps them consistent and imposes restrictions to them (e.g., the unique file path restriction or syntactical restrictions like maximum file name lengths).

Versioning and security related metadata.

Security related metadata such as *access control lists* or *security labels* are used for protecting file system resources but include also, e.g., *checksums* for preserving data integrity. This kind of metadata is usually managed by the file system/operating system itself and can be considered as a special kind of low-level management metadata. Versioning metadata can be modeled in many different ways and appears as both, internal or external metadata. External versioning metadata are well-known from version control systems, such as CVS, SVN or GIT[2]. A more detailed discussion of the specialties of these two kinds of metadata is beyond the scope of this thesis.

3.3 Links

A special type of metadata are links between file system objects. Links are explicitly modeled relationships between resources. They can be exploited for alternative navigational paths and may thus help to overcome many shortcomings of current hierarchically organized file systems. Many researchers including the author consider the support of inter-file relationships as the most important step towards semantic data organization [SG03, MC03, ABG+06, SH09]. Links between files can further be used to associate files with arbitrary external metadata that are, for example, stored in another file. We discuss links separately in this section due to the central role they play.

Before we focus on links on the desktop we briefly compare the differences in semantic expressiveness of links in the Web of Documents, the Web of Data and the desktop. As a simple example we consider two files, an image and a slide set that are related to each other (e.g., because the image is used in the slides). In the following we describe how the expression of such a relation is supported by currently prevalent linking schemes (cf. Figure 3.2):

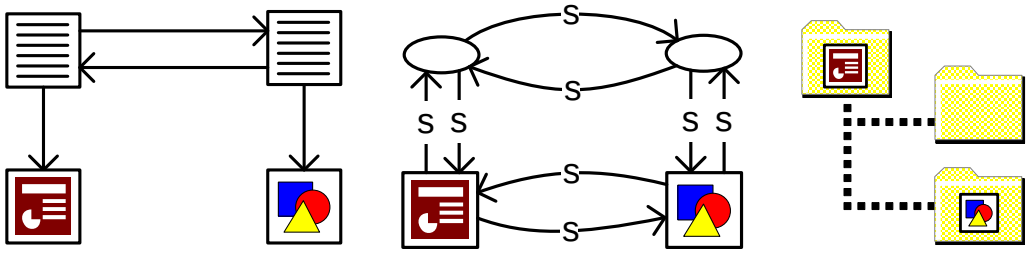
The Web. Binary HTML links between Web pages and from Web pages to embedded resources (e.g., embedded images, videos, etc.) express merely that those two resources are related (Figure 3.2a). No further explicit semantics can be attached to these links. Direct links between non-HTML resources cannot be modeled due to the used *embedded linking model* that is discussed below. The two media items in our example would thus be linked using one (or multiple) hyperlinked Web pages that embed these resources.

The Web of Data. In the graph-based RDF data model, links between resources are modeled by RDF properties and the semantics of such a link can be explicitly defined using semantic vocabularies/ontologies. In the Web of Data that is formed by publishing representations of resources as Linked Data, our media items could directly be represented by RDF resources and content-negotiation could be used to retrieve either the raw file data itself or its (interlinked) metadata representation. Links between such resources can be interpreted as direct semantic relations between them (Figure 3.2b).

The Desktop. Most file systems provide some sort of linking mechanism (shortcuts, soft and hard links, aliases, see Appendix B) that allows to reference a file or folder. To the end-user, such shortcuts look like the referenced file and at first sight this seems like a possibility to overcome the hierarchical organization of current file systems. However, as the integration of these mechanisms is currently not seamless, it requires the user to understand the difference between acting on a pointer to a document and acting on the document itself [DEL+00]. Furthermore the current linking implementations in the various file systems do not share common semantics and have technical issues (see Appendix B, [MC03]). Above that, such links cannot be used to model direct relations between files. They constitute

[2] File version information could, however, also be modeled as internal metadata. One proposed model was to store each version in an alternate stream of a file. Refer to [SFH+99] and [MTX02] for a discussion on this topic and two examples of special file systems that support file-versioning in internal metadata.

just alternative ways for accessing existing files/folders. Current implementations do not allow one to express the semantics of such "links" themselves. For these reasons, current linking mechanisms of file systems can be considered inadequate for the purpose of semantic file organization. Basically, the only explicit relations between file system objects in current file systems are modeled by the folder hierarchy itself [ABG+06] (Figure 3.2c).



(a) Explicit links between Web pages and embedded resources

(b) Semantic links between RDF resources

(c) Explicit hierarchical relations between folders in a file systems.

Figure 3.2: Different linking constructs on the Web (3.2a), between RDF resources in the Web of Data (3.2b) and between folders in a hierarchical file system (3.2c)

3.3.1 Examples for links on the desktop

Although not recognized by most users, desktop systems *are* already highly-interlinked networks of resources. However, due to the absence of appropriate mechanisms these links are stored mostly as internal, extrinsic metadata represented to a major part by proprietary metadata schemas. This makes access to these data and preservation of link integrity difficult.

Links on the desktop as considered in this work include for example: (i) links to embedded resources in text documents, (ii) links from PDF files to external files, (iii) links from spreadsheets to associated datasets, (iv) links from playlist to audio files, (v) links between source code files, or, (vi) links from a file folder to another one.

We further consider links from local to remote resources that are published on the Web or on the Web of Data. Such links can be found, e.g., embedded in text documents such as this thesis. Another example is a browser bookmark file that actually links the file itself with the referenced Web resources. Yet other examples are provenance links as discussed below in Section 3.3.1. In order to better understand the importance of local links on the desktop we end this chapter with four simple examples.

Example 1 : Links in music playlists

As a first, simple example for links on the desktop we consider links to MP3 files stored in a playlist file. M3U, for example, is a popular, simple, text-based file format for storing a list of references to multimedia files[3]. M3U stores references in the form of (platform dependent) file paths or as URIs. An example M3U file is depicted in listing 3.1.

Listing 3.1: An example M3U playlist containing relative and absolute path references as well as URIs and a reference to another playlist file. Comments are prefixed by hash-characters.

```
# relative path
leftfield-ftx-MJ.mp3
# absolute path
Q:\music\MP3\np\ftx\left_field_ftx_mj_tc.mp3
# HTTP URI
http://www.maupi.com/assets/sound_file/filename/533/01_Werke_fuer_Wutzler_No._1.mp3
# FILE URI
file://Q:/music/MP3/np/04_jack_und_jack_3_theme_song.mp3
# Other M3U files
otherplaylist.m3u
```

It can be seen, that file paths can be absolute or relative (where the base folder is unspecified; applications usually resolve such paths relative to the M3U file location). Local and remote files are also referenceable via URIs, other playlists may also be referenced. The links to local and remote resources in such M3U files are usually interpreted by a music player that *dereferences* them in order to retrieve and play the referenced audio files. This dereferencing fails in three general cases:

i) Referenced resources were moved, renamed or removed, which invalidated all relative and absolute paths as well as URIs.

ii) Local or remote resources become temporarily unavailable, e.g., due to plugged-out external hard discs or due to network connection failure.

iii) The M3U file itself was moved which breaks all relative paths in the playlist.

The first case (i) is probably the most common one. It occurs when users move or rename referenced files or parts of the folder tree containing these files, in the latter case possibly breaking multiple links. Such file system manipulations are quite common and current operating systems issue no warning whatsoever that links to the affected resources might become broken.

The second case (ii) becomes more and more common with the increasing popularity of external, portable storage media. Think, for example, of a memory stick containing your favorite MP3s and a couple of playlists with absolute file references (e.g., `Q:\music\nicetune.mp3`). When you unplug this memory stick and later plug it in again, the operating system might assign a new drive letter (say `W:` instead of `Q:`) to this device with the result that all links in your playlists will be broken. Note that remote resources on the Web can also become unavailable, for example, because a music provider changes its domain name or reorganizes its URI space.

The third case (iii) shows that relative references are only a partial solution for these problems: For example, when you move a playlist file from your memory stick to your desktop, all links using relative paths will be broken unless you also move the referenced media files.

[3] M3U (MP3 URL) was originally developed by *Nullsoft Inc.* for their *WinAmp Multimedia Player*. The simplicity of this file format has contributed to its popularity and it is supported by many other media players including *Apple's iTunes*, *Windows Media Player* or *VLC Media Player*.

As most other playlist formats (e.g., ASX, PLS or RAM), M3U does not provide any mechanism for avoiding/fixing broken links[4]. Music players usually deal with such broken links by simply skipping the respective entries in the playlist.

Example 2 : Links in a photo collection

In the previous example, links became broken because they were not updated when referenced resources changed their locations. This often happens when applications that are not aware of these links can manipulate resource locations. For example, a user using a file explorer may manipulate the folder tree containing MP3 files.

Some applications such as Apple's iTunes (a music player) or iPhoto (a photo management software) try to minimize such interventions by storing the resources they manage (i.e., audio files and images) in their own (automatically created) folder hierarchies[5]. It is the intention of these applications that all data-manipulations are done via their interfaces so that they receive all notifications required for preserving the integrity of links between the metadata they store (e.g., music collections or tags attached to images) and the respective files.

However, this kind of link integrity preservation is based on conventions rather than on a technical solution as there are no technical barriers that hinder a user or an application to manipulate these file trees. Considering that today's desktops are heterogeneous environments and that their central storage layer, the file systems, is used by many independent applications makes it unlikely that such a strategy may be successful on a larger scale[6]. A particular problem is that one application's predefinition of how its data is organized in the local file system may interfere with the organization schemes of other applications.

Example 3 : Embedded objects in documents

Many current document editors like word processors or slide authoring tools allow it to embed media objects (images, videos, etc.) into their documents. One possible way to do this is to copy the digital representation of the embedded object and include it in the document file itself, thereby really "embedding" this digital object. Although this is the standard method in most current word processors (like the popular Microsoft Word) it has major drawbacks: first, by copying the file the connection to the original object is lost which means that changes to this original object are not reflected in the embedded copy and vice versa. Second, this method does not scale very good as the file size of the authored document is increased by the amount of the embedded file's size. This may lead to major problems when authoring larger documents like, e.g., a PhD thesis.

An alternative method is to store just a reference to the embedded object in the authored document and resolve this link when the document is rendered (e.g., presented on the screen or printed). This method avoids unnecessary data copies and keeps the authored document and the referenced media object separated. This HTML-like *embed by reference* model is supported by most current document editors. However, this kind of embedding is as vulnerable for broken links as the playlists discussed in example 1. Word processors usually display placeholder images when such an embedded resource could not be dereferenced due to a broken link.

Note that documents (such as this thesis) are themselves highly interlinked structures. Consider, for example, references in a table of contents or references to bibliographic entries. These kinds of links, however, are usually handled exclusively by the authoring software itself. Therefore such software usually provides

[4] More advanced playlist formats like Kazaa's *Kapsule* format or *XSPF* (XML Shareable Playlist Format) may store hash values usable for identification of the referenced audio files. There are even some experimental solutions for keeping playlists up-to-date such as listfix (`http://listfix.sourceforge.net/`) or the DioneSS Playlist Editor.

[5] Although iTunes allows you to keep your own folder structure if desired, however, at the increased risk of broken links.

[6] A similar strategy can be observed, e.g., on smart phones where the folder structure is predefined to a large part. Although this arguably works for the current "generation" of phones, we believe that the conceivable increase of processing power and storage capacity on these devices will ultimately lead to the same problems as observable on heterogeneous desktop environments today.

mechanisms to ensure the referential integrity of such links to other document parts. For preserving the referential integrity to external resources no such mechanisms exist.

Example 4 : Provenance links

Provenance information generally describes the derivation history of resources, e.g., *where* the data comes from, *how* and *by whom* it was manipulated, etc. A very simple example for file-related provenance information is the HTTP URI a file was downloaded from. Another example is the sender of an email whose attachments are stored somewhere in the file system. Provenance metadata gets increasing attention recently, especially in the domains of digital preservation or scientific discovery reproducibility but its usefulness for data organization in general is also recognized [ABG+06, SSGN07, MFF+08, GeCeGe+10].

Some desktop applications already do preserve some of these metadata. For example, Apple's Safari browser stores the mentioned download URI of images in an extended attribute of the image file[7]. This entry actually links the local image file with its originating Web resource. This information may be exploited in multiple ways. An application could, e.g., automatically check whether the local image differs from its remote version (e.g., due to local changes to the image). An application could further search for all local files that originate from this particular URI, thereby finding all images that originated from this common "ancestor" image. Yet another use case is that an application could learn about additional data related to this image by querying some online service with this URI.

Obviously, such links between local files and online resources may break like any other link on the Web itself.

3.3.2 Integration of local file system objects with the Web of Data

In this section, we present a short data integration scenario that integrates local file resources with resources in the Web of Data. By this, we want to motivate why an integration of these data would be beneficial.

Links into the LOD cloud. The linked open data cloud consists of a large and growing variety of data sets that contain useful targets for links from the desktop (cf. Figure 2.3). Consider, for example, links from your local MP3s into the *DBTune*[8] data set that contains metadata about music-related resources. Such a link would model an explicit relationship between a local audio file and a comprehensive, community-maintained metadata set describing it. This kind of links could, for example, be automatically created by resorting to internal file metadata (e.g., stored in the ID3v2 headers of the MP3s) for querying the DBTune Web services.

A local music player could consume these remote metadata and present it to the user as additional information when she plays an audio file. It could further exploit these metadata to create automatic links into other LOD data sets such as the DBpedia, which could in turn be exploited to present even more metadata (e.g., artist biographies) to the user.

Further, remember that these remote metadata are also linked with each other and by this also indirectly interlink local music resources: consider, for example, two of your local MP3s being automatically interlinked with the respective metadata descriptions in the DBTune data set. Further, consider that both were created by the same artist and that this relation is explicit in the DBTune data. Local applications could exploit such relations easily, for example, to display a list of all local audio files that were created by the same artists.

Links to other Linked Data. The open access policies for LOD data sets are not applicable for all data. However, data can still be published conforming to all Linked Data principles (cf. Section 2) while being access-controlled or hidden behind firewalls, etc. Consider, for example, that a company decides to publish

[7] This URI is stored in the *kMDItemWhereFroms* field used by Spotlight and can be accessed via the Finder in the "More Info" section of the file. See https://discussions.apple.com/search.jspa?q=kMDItemWhereFroms.

[8] http://dbtune.org/

metadata about its projects *internally* as Linked Data. Access to these data would, for example, be possible only within the corporate network. As for the music scenario above, company employees could now link their local project-related files to remote metadata sets. By this, a company-wide network of project files would be established over time. Users could further publish their own, local, project-related metadata as Linked Data in the company network. In this case, building a central index of all file resources that are associated with a certain project would be trivial.

Links to other linked file systems. Note that publishing parts of a local file system as Linked Data enables also direct, stable links between files stored on different PCs. Applications could build local metadata indices of remote files and users could search these indices without being connected with this remote PC. By this, an employee could search the metadata of all company-wide, project related files (without actual file access) while being disconnected from the company network.

3.4 Discussion

In this chapter, we discussed the various shapes and categories of file-related metadata and discussed various issues regarding their representation as internal or external metadata. Today, most file-related metadata are stored in internal file headers that are defined by the various file formats. This means a practical limitation for metadata handling applications as access to these data requires knowledge of the actual file format (e.g., its byte layout). The ever-growing number of file formats makes the situation worse: implementing, for example, an application that accesses basic metadata (image width and height) of all images on a local desktop system is nearly impossible due to the large variety of existing image file formats[9]. A further problem of internal metadata is that they can be attached only to files (not folders) and that many files (e.g., plain text files) do not foresee any header structure for storing them. Another limitation of current file systems is their weak support for links between file system objects. Although links on the desktop are ubiquitous (which includes links to external (Web) resources), no satisfactory mechanisms for preserving the integrity of such links exist. Moving files on a hard disc may, for example, result in broken music playlists, incomplete photo albums or missing images in a slide set.

Dissolving local data silos. The magnitude of proprietary, internal metadata schemas and the weak linking support of current file systems contribute to the formation of "data silos". *Siloed data* (and metadata) are not easily exchangeable and accessible which hinders their integration with other data sources. This practically means that these data cannot contribute to a global information system and are hardly usable for the implementation of advanced data management strategies.

Storing metadata externally would help to dissolve such silos as it would enable the development of uniform access interfaces to these data. Such interfaces could further enable external data access which would in turn allow advanced methods of data exchange and integration. In a summary, an external metadata model would:

i) Allow separate access to data and metadata.

ii) Allow for more sophisticated and standardized metadata descriptions.

iii) Enable uniform access to these metadata.

iv) Improve interoperability between metadata handling applications.

v) Help to dissolve "data silos" on the desktop.

[9] Refer to http://en.wikipedia.org/wiki/Comparison_of_image_file_formats for an exemplary list of image file formats.

The need for stable links. However, such an external metadata model requires, in its simplest form, stable links between file system objects and their (external) metadata records. Integration with other (remote) data sources would further require a way to stably reference such external resources.

Currently, however, stable links between local file system objects and remote resources are limited in many ways. It is, for example, impossible to stably reference files stored on a remote computer unless these files are already accessible via some transport protocol (such as NFS or SMB). It is even impossible to stably associate files downloaded from the Web with their remote resources as the HTTP URIs of these resources may also change (e.g., due to URI space reorganizations). As a consequence, links to these resources can break as further discussed in Section 5. Thus, a precondition for the implementation of an external metadata model is a mechanism that either avoids or fixes such broken links on the desktop. We discuss our two solutions for this purpose in the Chapters 6 and 8 of this thesis.

Before this, however, we present a concrete model and implementation for storing external, file-related metadata. This model supports stable, semantic links between arbitrary file system objects and remote resource and provides a uniform interface for data access. In Section 3.3.2 of this chapter, we have sketched a concrete scenario that should provide an idea to the reader how semantic links between local and remote resources could enable advanced data organization, integration and publication.

Chapter 4

A model for external, semantic, file-related metadata

As discussed in the previous chapter, file-related metadata and links are central building blocks for the implementation of semantic data organization. Opening current "data silos" and reducing the *information fragmentation* problem by introducing unified access interfaces to such metadata on the desktop would have a major impact on the way how digital information is organized and exchanged in such environments. Karger and Jones analyzed the state of the art in personal information management (PIM) and the benefits of a unified metadata interface for local resources:

> "Whereas a traditional word processing program would store its documents as an opaque file, an XML representation can use the standard syntax to expose, as separate elements within the file, along with the creator(s) of the document, its title, its subject keywords, its citations, and its individual sections. Such a unified syntax for metadata representation would allow us to group or seek arbitrary information objects by shared metadata, regardless of their managing applications, much as we currently might group files independent of their managing applications" [KJ06].

Although they do not propose a concrete model in their work, they propose the usage of a generic, standardized format for the representation of metadata and links on the desktop such as XML or RDF.

In this chapter, we propose such a model by first informally discussing the *status quo* in metadata organization on the desktop and then formally defining a new model based on external RDF descriptions. These RDF descriptions are used to represent metadata and links associated with local file system objects. The descriptions (RDF graphs) are accessible as Linked Data, i.e., they are published according to the Linked Data principles presented in Section 2. We further discuss how our model allows these actors to execute CRUD (create, read, update, delete) operations on such metadata descriptions in a unified way. Differently from the current situation, our model avoids the formation of data silos and leads to a central, multi-faceted metadata graph that enables various actors to access different parts of this graph efficiently. File-related metadata may be described unambiguously with arbitrary semantic vocabularies, which makes data integration tasks a lot easier.

Note that a major part of this chapter was published in [Pop11].

4.1 Introduction

The currently predominant way of storing internal, file-related metadata is depicted in Figure 4.1a. As already discussed, a major problem with existing file metadata schemas is that they are often application-dependent and proprietary. Applications that access such metadata have to know the semantics of the particular schema in order to utilize the contained information. Further, if the metadata are stored internally as it is usually the case, these applications have to know how to read/write the particular file format in order to access its metadata headers. This means they have to know the byte-layout of each particular file format they read from and/or write to, which is a considerable practical problem when considering the magnitude of existing formats. More standardized schemas like Exif or XMP make things easier for obvious reasons as they enable a wider range of applications to access and manipulate these metadata. Some wide-spread standards for media-related, internal metadata (Exif, ID3v2 and XMP) are discussed in Appendix C.

A further disadvantage of the current *status quo* is that low-level file metadata like file ownership or access rights are defined over the whole file including the metadata headers. This means that it is not possible to specify different access policies for these two aspects of a file. It is, for example, impossible to define that some process has access to a file's metadata (e.g., for indexing them) but not its actual data.

External metadata. For these reasons, applications that need to attach metadata to a large variety of files commonly use external metadata approaches instead. The file versioning system SVN, for example, stores metadata about versioned file system objects in various auxiliary files stored in hidden ".svn" sub-folders; the development platform Eclipse (and similarly many other IDEs) stores meta information about a project (i.e., a set of files) in a ".project" file and in additional files in a ".settings" subdirectory; Apple's iTunes software organizes audio files in special folder hierarchies as discussed in Section 3.3.1. Metadata related to these audio files is stored in an extra file[1].

Such applications need to preserve the integrity of the associations between files and their external metadata records. As long as all file system manipulations are done through their respective user interfaces (e.g., via the Tortoise SVN plug-in, the Eclipse IDE or the iTunes GUI) this is straightforward. However, when a user moves a directory managed by SVN with the regular file explorer, the connection between files and their metadata becomes broken and SVN does not "know" anymore that a particular file in this directory corresponds to a particular record on the SVN server.

Linked Data publishing on the desktop

The currently emerging way how data is published on the Semantic Web as Linked Data (cf. Section 2) stands in clear contrast to the *status quo* in metadata handling on current desktops: Linked Data resources are usually described using a mix of multiple community-accepted or standardized vocabularies instead of file-type dependent proprietary metadata schemas. Access to (meta) data published in this way is done uniformly (HTTP, SPARQL, etc.), a standardized format (RDF) is used for their representation and any application can make claims about such resources independently. A Linked Data resource may further be backed by more than a single representation, e.g., an HTML-encoded Web page describing the resource in a human-readable way and an RDF graph describing it in a machine processable fashion.

The Linked Data way of describing, publishing and sharing resources may also be applied to (local) file data, simply by treating each file system object as a resource one can make RDF statements about [SP10]. Bringing this flexible, standardized and open approach to the desktop would comprise major benefits:

Multi-faceted, semantic description. Generic Semantic Web vocabularies that are independent of file-type and application can be used to describe various aspects of a file: for example, the FOAF vocabulary could be used to describe a document's authors, LINGVOJ could be used to describe the languages of

[1] This file is called "iTunes Library.itl" on Windows systems, see `http://support.apple.com/kb/ht1451` (Accessed May 2011).

the contained content or the COMM ontology could be used to describe an object's multimedia aspects (refer to Appendix C for a brief introduction to these vocabularies).

Such community-standardized, semantic vocabularies would make it possible to integrate these data easily with other (external) data sets. For example, the inverse functional `foaf:mbox` property could be used to link local documents with online resources (possibly containing RDFa descriptions) authored by the same person. Data integration of *information items* as described in [Jon07] (which includes, e.g., files, emails, Web resources, etc.) would become possible.

Additionally, more specialized, domain specific and custom vocabularies could be used to describe various aspects in more detail if required by particular applications.

Separate access policies. Content- and metadata independence would mean that applications that read or write metadata of files do not necessarily have to know how to read/write the actual file data. Access to a file's content and its metadata could be controlled separately.

Multiple representations. Different representations of a file's metadata could be offered in a standardized way. This could, for example, be useful when actors accessing the metadata of a file are allowed to read only a restricted subset of these metadata. It would further enable to deliver subsets of metadata records to actors if these do not require the whole record. Last but not least, this mechanism enables backward-compatibility for applications that are not RDF-aware as further explained in Section 4.4.4.

Any application that has the permission[2] to write such RDF metadata descriptions could either add its own metadata independently (by using custom and domain specific schemas/vocabularies, as already possible, e.g., with XMP) or modify existing data that was expressed using standardized, general-purpose vocabularies.

4.2 A novel metadata model for the desktop

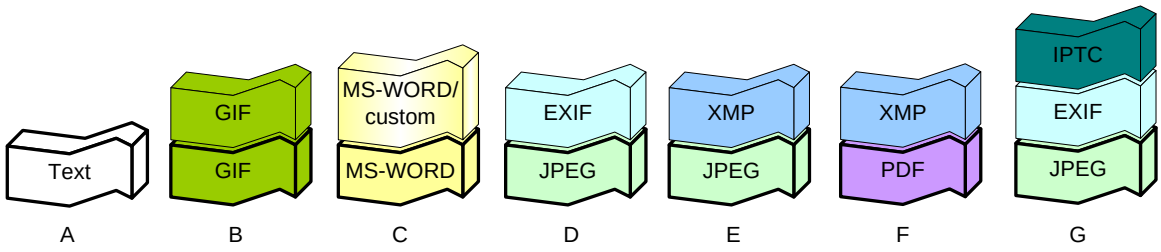
We propose a novel model for the management of file-related metadata on the desktop where data and metadata are treated consequently as distinct entities that are strongly related but may be accessed and modified independently. In our model, metadata are represented in the form of RDF sub-graphs that are connected to the file system objects they describe. The stability of this connection is of high importance in a desktop environment [SH10] and we discuss an algorithm for their maintenance later in this thesis (Chapter 8). Complex metadata descriptions, well beyond simple name/value pairs, can be created using the graph-based RDF data model which allows the description of resources with a mix of semantic vocabularies by design. Consequently, file system objects can be described with many, possibly orthogonal vocabularies of differing abstractness in our model. An image could, for example, be described by a generic vocabulary that contributes very general metadata about files such as their author(s) or versions; it could further be described by a general vocabulary for images that describes properties like an image's width or its color model. If the image is a photograph, it could further be described by a vocabulary derived from common metadata standards like Exif or XMP. Finally, custom semantic metadata describing the contents of the image (for example, what people are depicted), may be added. By this, the semantics of the stored metadata can be defined unambiguously using ontologies, which in turn enables reasoning and advanced data interpretation and integration tasks.

Other then the current situation, our model is based on external metadata descriptions. Figure 4.1b depicts how the situation of Figure 4.1a would change with our proposed metadata model: any file system object could be associated with one or more RDF graphs that describe its metadata. Thus, even file system objects that do not support internal metadata headers (e.g., plain text files or file folders) can be associated with arbitrary metadata records.

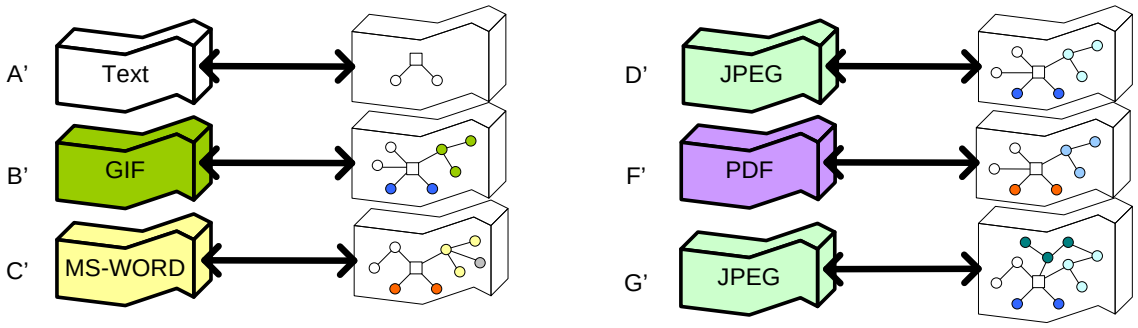
[2] In practice, such a generic metadata access method would obviously require a sophisticated security model that prohibits unauthorized access to (certain parts of) the metadata.

Figure 4.2 shows in more detail how a metadata record could look like. In this example, a digital photograph is stored as a JPEG file and general, image-specific and Exif-mapped metadata are available. Various tools could have contributed to this metadata description: An Exif extractor component could have extracted the internally stored Exif data, another extractor could have made the image width explicit and the software that copied the photograph from the camera to the desktop could have contributed the author information.

We continue this chapter by first discussing two principles that are required when using RDF graphs for metadata representation: First, we discuss some general approaches for extracting sub-graphs from RDF graphs. This is required for the definition of certain sub-graphs of a large metadata graph as the metadata descriptions of particular file system objects. After that, we describe how such sub-graphs can be merged and de-composed which enables access to partial aspects of a metadata description. Finally, we sketch how such metadata descriptions can be published as writable Linked Data before introducing and discussing our actual model in a formal way.

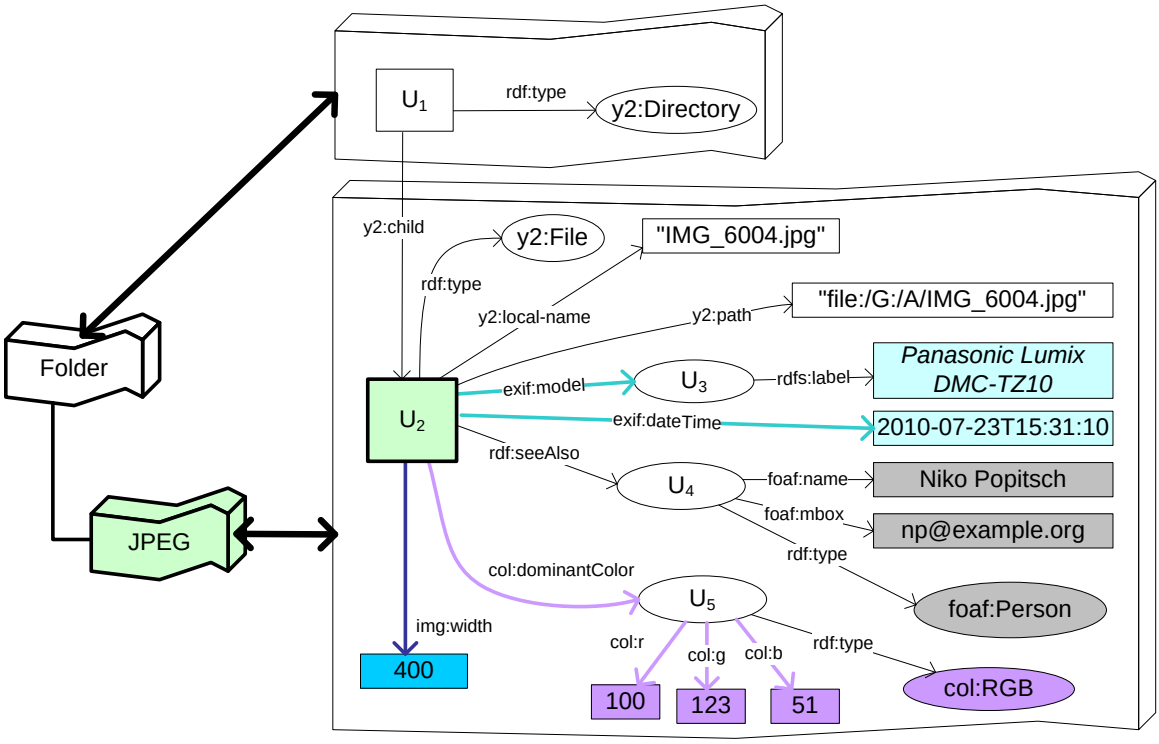


(a) Internal, proprietary metadata. Boxes with strong lines depict the *data* part of a file, boxes with light lines the metadata part. The figure shows the current situation: metadata are to a large part stored internally in proprietary formats: **(A)** a text file without internal metadata; **(B)** a GIF file with embedded GIF metadata headers. For reading/writing these metadata, applications have to know the GIF data format and its metadata format; **(C)** a MS-Word file with proprietary metadata header. It is possible to add custom, schema-less metadata to such a header; **(D)**, **(E)** and **(F)**: two JPEG and one PDF file, one with an EXIF, the other two with XMP headers. As both metadata formats are standardized, these metadata are accessible by a much larger group of software when compared to proprietary formats. Note that XMP data is already RDF compatible. Applications still need to know how to read/write the JPEG/PDF data format in order to access the embedded headers; **(G)** files may also contain multiple different metadata headers.



(b) External, standardized metadata. Boxes with strong lines depict the *data* part of a file, boxes with light lines the metadata part. The figure shows the situation corresponding to Figure 4.1a when our proposed metadata model is used: **(A')** the text file is linked to a metadata graph represented as an RDF graph. Applications need to know only how to read/write this format to access any file's metadata. The metadata itself is expressed using a mix of different semantic vocabularies/ontologies visualized in the figure by using a common fill color. The file resource itself is represented by a white square, the white fill color stands for a generic vocabulary that describes very general (core) metadata about files including its low-level features. **(B')** The GIF image's exemplary metadata graph contains general (white fill-color), image-specific (dark-blue) and GIF-specific (green) metadata. Applications that know how to handle the generic image-specific vocabulary can access this part of the metadata even if they don't know how to handle GIF files or their metadata. **(C')** The MS-Word file could be associated with general (white), text document-specific (red) and proprietary metadata. The addition of arbitrary custom metadata (gray) would be naturally supported by RDF. **(D')** and **(F')**: The JPEG image would be associated with image-specific and Exif metadata, the PDF with text document-specific and XMP metadata (analogously for **(E)**, not shown in figure). **(G')** Multiple metadata headers would simply result in a larger RDF graph.

Figure 4.1: Internal and external file-related metadata.



(a)

```
<U1>  a y2:Directory ;
      y2:child <U2> .

<U2>  a y2:File ;
      y2:local-name "IMG_6004.jpg"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:path "file:/G:/A/IMG_6004.jpg"^^<http://www.w3.org/2001/XMLSchema#string> ;
      exif:model <U3> ;
      exif:dateTime "2010-07-20T10:04:59"^^xsd:dateTime ;
      rdf:seeAlso <U4> ;
      col:dominantColor <U5> ;
      img:width "400"^^xsd:int .

<U3>  rdfs:label "Panasonic Lumix DMC-TZ10"^^<http://www.w3.org/2001/XMLSchema#string> .

<U4>  a foaf:Person;
      foaf:name "Niko Popitsch"^^<http://www.w3.org/2001/XMLSchema#string> ;
      foaf:mbox "np@example.org"^^<http://www.w3.org/2001/XMLSchema#string> .

<U5>  a col:RGB ;
      col:r "100"^^xsd:int ;
      col:g "123"^^xsd:int ;
      col:b "51"^^xsd:int .
```

(b)

Figure 4.2: Sketch of an exemplary metadata record of a JPEG file (U_2) and a partial record of its parent folder (U_1). Figure 4.2a shows a graph view, Figure 4.2b the corresponding RDF record serialized in RDF/N3. Squares are used for depicting the resources that represent the actual file system object. Ellipses are used for other RDF resources, rectangles for RDF literals. Namespace definitions are omitted for readability.

4.2.1 Resource-centered RDF subgraph extraction

When considering a global semantic graph for the representation of file metadata, the question arises what subgraphs of this graph actually represent the metadata of a particular file. In the following we discuss some possibilities for the extraction of resource-centered subgraphs from RDF graphs. A resource-centered subgraph is "centered" around some RDF resource of interest, in our case around a resource representing a particular file system object.

Resource and properties. A very simple way to extract a useful subgraph for describing such a resource is to consider all triples that have this particular resource as subject. This means that only the "outbound arcs" of the resource are included. Not including the corresponding "inbound" arcs results in an "asymmetric" (cf. [Sti05]) representation that does not include triples where the resource of interest is in the *object* position. Consider, for example, the statement (`f1, y2:child,` `f2`) that models a parent/child relationship between two file system objects. An asymmetric subgraph would include f_2 in the description of f_1 but not vice versa. In other words, only the children but not the parents of a file system object in question would be included in such a subgraph. A *symmetric representation* consequently includes all triples having the resource of interest as either subject or object, i.e., both inbound and outbound arcs would be included. For an example, consider Figure 4.2. An asymmetric resource/property subgraph of the metadata of the JPEG file would include the 8 triples shown in the RDF/N3 serialization next to the resource URI U_2. A symmetric subgraph would also include the (`U1, y2:child,` `U2`) triple.

Reification and quads. Another possibility to extract a subgraph from an RDF graph is by considering the data provenance of its contained triples. Although the standard RDF reification mechanism could be used for this purpose, there are several reasons why this is rather inconvenient and not used in practice. One obvious problem is that reification of a single triple requires the representation of five triples resulting in a much larger graph. A discussion of the practical limitations of the current state of RDF with respect to triple provenance can be found in [GC10].

It was proposed by some authors to extend the RDF triple model into a quad model for these reasons [CBHS05, GC10]. A quad extends a triple by an additional URI Reference (or blank node or ID) that can be used to identify the provenance of this particular RDF statement. It could, for our purposes, however also be used to associate such an RDF statement with a particular resource. For example, all statements in Figure 4.2 could be associated via this fourth URI reference with the file system object resource they express metadata about. Extracting a metadata subgraph for this file system object would then simply be a matter of extracting all quads having the respective URI as value of their fourth URI reference. However, this requires an RDF repository to store four instead of three values per arc in the RDF graph which means a considerable additional storage requirement.

Named graphs and RDF datasets. This storage need can be reduced when using *named graphs*. A *named graph* is a set of RDF triples that is identified by a URI reference. By asserting triples mentioning this name, one may now express statements describing this very (sub-)graph as any other RDF resource. Named graphs are considered useful for, e.g., provenance tracking, versioning and for specifying access rights for RDF sub-graphs [CBHS05]. The triples contained in a named graph can be seen as quads having the name of the graph as their fourth URI reference as described above. However, the above-mentioned storage requirements are reduced significantly under the precondition that all these triples share the same name[3]. One way to use named graphs for a resource-centered subgraph extraction would then be to define one named graph per file system object and use the file URI for naming this graph. This would, however, result in a large number of named graphs. Further, this model would enable only one named graph per file system object.

[3]Please note that most current RDF repository implementations do actually realize named graphs by using quads. In such a case, the storage requirements for named graphs and quads are obviously the same.

This latter restriction could be avoided by using one *RDF datasets* per file system object. An *RDF dataset* consists of a default graph and a set of named graphs having distinct names (IRIs) [PeSe08]. The definition of one such RDF dataset per file system object would allow it to include all its describing triples into the named graphs of this dataset. For example, the metadata contributed by application *X* could be in the first named graph, metadata contributed by a different application *Y* could be in the second named graph, etc. By merging these named graphs a complete metadata description of a resource could be obtained. Merging only a subset of these named graphs would result in a "filtered" view on this resource. This would be useful, for example, to restrict the size of this metadata description in certain situations. This approach may further be exploited for restricting access to certain aspects of a resource's metadata. Despite such beneficial properties, this model would result in a large number of RDF datasets, one per file system object. Unfortunately, this constitutes a practical restriction as current triple stores cannot handle very large numbers of RDF datasets efficiently[4].

SPARQL graph patterns. Another method to extract subgraphs from RDF graphs is the SPARQL query language. Triples that belong to the metadata description of a particular resource (file system object) could be selected by SPARQL graph patterns; the resulting subgraph could be built using a SPARQL *CONSTRUCT* query (cf. Section 2). This method requires, however, that these respective triple patterns are known beforehand.

To avoid such prior knowledge about the "model" of a queried RDF data set, SPARQL foresees so-called *DESCRIBE* queries. First, a graph pattern is used to query the data set. The SPARQL processor then determines a subgraph describing the resources contained in the result set of this query. This description is determined by the query service and may be arbitrary complex, possibly also containing metadata about other resources. There is no standardized way to further "configure" how these descriptions are created by the SPARQL service.

Both methods (CONSTRUCT, DESCRIBE) require the execution of one SPARQL query per subgraph extraction which might be costly.

Concise Bounded Descriptions. A *Concise Bounded Description* (CBD) of a resource is an RDF subgraph that is created by starting at this particular resource and including (i) all statements where the resource is the subject as well as (ii) recursively connected bnodes and their properties as well as (iii) the CBDs of all contained reifications. A CBD is also called the *bnode closure* of a resource. Analogously to the above-mentioned symmetry criteria, a *symmetric concise bounded description* includes both outbound and inbound arc paths [Sti05].

Other possibilities. Ding et al. propose *RDF molecules* as a useful decomposition of RDF graphs into connected subgraphs. "An RDF graph's molecules are the smallest components into which the graph can be decomposed into separate sub-graphs without loss of information." [DFP+05]. The authors propose an algorithm that decomposes any RDF graph into its molecules in a lossless way by exploiting functional and inverse functional properties. However, this method would require a background ontology that is not always available. Without a background-ontology only naïve decomposition is possible which includes basically a resource and its bnode closure.

The *Fresnel Selector Language (FSL)* is an XPath-inspired language for the selection of subgraphs in RDF graphs [Pe05]. It has been designed with a focus on usability and implementations for major triple stores exist. FSL expressions link graph entities (nodes or arcs) by an explicit path in the graph. Thus, a set of such FSL expressions could be used to associate all graph entities belonging to a metadata record with its centered resource.

[4] Our preliminary experiments with current triple store implementations that support named graphs/quads and scale to at least 100M triples showed that all of these have considerable performance problems with large numbers of RDF data sets (data not shown). Refer to `http://esw.w3.org/LargeTripleStores` for a continuously updated list of large triple store implementations.

Finally, Tumarello et al. introduced the concept of *Minimum Self-contained Graphs (MSG)* [TMPP05], the smallest self-contained subgraphs including a particular RDF statement. The authors show that any RDF graph can be decomposed into MSGs in an unique way. Further, they define the RDF Neighborhood (RDFN) of a resource as the subgraph that is composed by merging all MSGs mentioning this resource. As our model disallows bnodes (as explained below), the MSGs in our store are always single RDF statements and the resulting RDFNs are therefore symmetric resource/property subgraphs as described above[5].

4.2.2 Graph decomposition and merging

One of the major benefits of using RDF as a metadata model is that merging such graphs is straightforward. A merge of a set of RDF graphs that do not share blank nodes is simply their union. Figure 4.3 illustrates this by showing a decomposition of the graph from Figure 4.2. By superimposing the three depicted graphs one ends up with the original graph.

This decomposition into distinct graphs, however, has the advantage that these can be individually addressed meaning, for example, that different access rights can be granted to them or that a merge process may consider only a subset of graphs, thereby "filtering" some of the metadata out. This also enables scenarios where applications write/read exclusively to/from one or multiple of such subgraphs. It further enables to "lock" parts of the global metadata graph as *read-only* subgraphs.

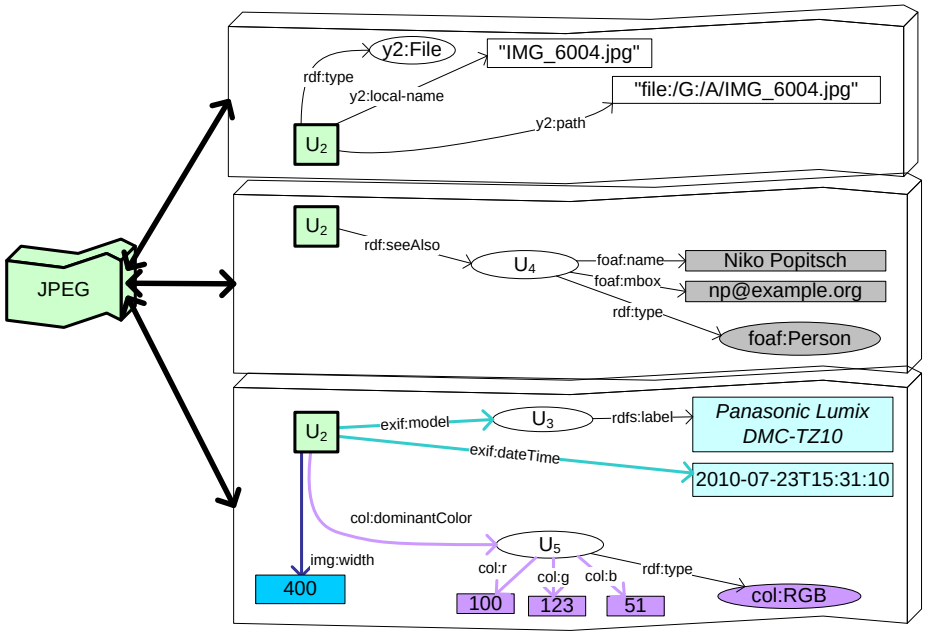


Figure 4.3: A decomposed metadata record. The figure shows one possible graph-decomposition of a metadata record. An RDF merge of the three shown graphs results in the metadata record depicted in Figure 4.2.

It is noteworthy, that even such a simple graph decomposition and merging approach may result in contradictory metadata. Consider, for example, two such subgraphs containing different values for the same property,

[5]Note that [TMPP05] defines an RDFN as "all the MSGs involving the resource itself" which includes MSGs containing statements mentioning the resource in the *predicate* position. In such a case, the RDFN is not equal to our resource/property subgraph but also includes such statements.

e.g., the *img:width* property. The merge of these graphs will then contain both values and metadata consumers have to decide which image size (or both) to use. In such a case, access to the individual graphs may be beneficial as this would enable such consumers to decide which image size is right for them based on data provenance features. However, a deeper discussion of this topic is out of the scope of this thesis.

4.2.3 A uniform Linked Data interface

So far, we have discussed how RDF metadata descriptions of file system objects can be extracted from a larger RDF graph that stores metadata about a whole set of such file system objects. We further discussed how such subgraphs can be (de-)composed which allows access to partial representations of a metadata description.

To uniformly access these descriptions now, we propose their publication according to the Linked Data principles (cf., Section 2). This basically means, that each file system object has a corresponding resource in the metadata RDF graph that is identified by an HTTP URI. Dereferencing this URI leads to either a human-readable HTML representation or to a machine-friendly RDF representation of the metadata "record" that describes the respective file system object. This approach is easily extensible to support other representation formats, e.g., for satisfying legacy interfaces when needed. The interface does, however, always support the delivery of metadata in a standardized, uniform format (RDF) that can be semantically interpreted if according semantic vocabularies were used for modeling the data.

The Linked Data approach has the further advantage that the published metadata could be made accessible to remote actors easily. Using HTTP as standardized transport protocol makes this interface accessible to a very broad range of actors. Using (stable) identifiers (HTTP URIs) for file system objects further enables references to such files, regardless whether these are stored locally or on some remote computer.

Shortcomings. However, the relatively young Linked Data approach still suffers from some major shortcomings. Above all, a standardized approach for *writing* Linked Data itself is currently missing, which may be considered a major issue [BLHL+08, GMG11]. Current Linked Data (e.g., in the LOD cloud) is usually created by some mapping process that converts "conventional" data sources (e.g., RDBs) to RDF, as in the case of DBpedia. Such data cannot be updated by sending RDF graphs to the Linked Data endpoint, but rather by changing the underlying data source, in the case of DBpedia by modifying data stored in a database using the Wikipedia Web interface. However, this forces applications to switch between two "worlds" for writing and reading data which adds undesirable complexity.

Further, descriptions of Linked Data resources are currently treated as atomic entities by current Linked Data servers. When a content-format was negotiated successfully, the whole resource description is delivered by the server in the respective format. In some situations it would, however, be beneficial if a server could deliver only *parts* of a (possibly large) resource description for reasons of restricted access or to reduce processing, I/O and transmission costs.

In the following, we formally introduce a model for a storage layer directly on top of Linked Data that enables CRUD (create, read, update, delete) operations on sub-graphs of Linked Data descriptions. This model enables to read and write selected aspects of Linked Data records and, by this, overcomes the above-mentioned shortcomings.

4.3 A formal definition of the Y2 metadata model

In this section we formally specify our proposed metadata model (the "Y2 metadata model") that is based on the considerations of the previous section. This model allows the efficient representation of multi-layered resource descriptions that are organized along two dimensions: Named graphs are used to separate logical "aspects" of the contained data into "horizontal" *layers*. Extraction functions are used to decompose these graphs into "vertical" *records*, collections of sub-graphs "centered" around some resource of interest.

The higher structuredness of the metadata stored with our model, when compared to arbitrary RDF graphs, brings practical advantages for implementing data or user interfaces. The strong data compartmentalization can be exploited in many ways, e.g., for more fine-grained access control or for concurrent write access to various layers of the store. It further enables actors to **c**reate, **r**ead, **u**pdate and **d**elete metadata that is organized into records using established technologies and formats from (Semantic) Web research.

We discuss a prototypical implementation of this model in Section 4.4. This prototype implements a REST interface for read/write access to the stored records. Further, clients can access partial records based on HTTP header settings. Note that this model is domain-independent and not restricted to local file-related metadata in any sense. It can thus be used to store record-like Linked Data of any kind. Here, however, we show how it can be used for the organization of explicit, file-related metadata represented by *external* RDF subgraphs.

Initial definitions. Our model disallows the use of blank nodes and strongly discourages the use of RDF reification in the used RDF graphs. Both are also discouraged for Linked Data publishing in general [BCH08]. The main reasons for this are that (i) bnodes are not referenceable and make graph (de-) composition much more complex and (ii) that reification results in increased storage requirements and makes querying more difficult. We start this section by introducing some derivations of well-known sets from the RDF specifications by excluding bnodes:

Definition 4.1 (Initial definitions) *Let $\mathbb{U}$ be the set of URI references, $\mathbb{B}$ the set of RDF blank nodes and $\mathbb{L}$ the set of literals as defined in [KeCeMe04]. Further let $\mathbb{T} = (\mathbb{U} \cup \mathbb{B}) \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{B} \cup \mathbb{L})$ be the set of RDF triples and $\mathbb{T}' = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ the set of triples without blank nodes.*
Further let $\mathcal{P}(X)$ be the powerset of an arbitrary set X.
Then $\mathbb{G} = \mathcal{P}(\mathbb{T})$ is the set of all RDF graphs and $\mathbb{G}' = \mathcal{P}(\mathbb{T}')$ is the set of all RDF graphs without bnodes. Further, analogously to the definitions in [CBHS05], we define a named graph without bnodes *as a pair $(u, g) \in (\mathbb{U} \times \mathbb{G}')$.*

Having introduced these basic sets, we continue by introducing our concept of *facets*, RDF graphs "centered" around a subject resource. This subject resource (also called *facet subject*) is identified in the subgraph by being the only one having an asserted OWL class `y2:RecordFacet`, that stems from our vocabulary introduced in Section 4.3.1. A second type of facets, so-called *extension facets*, are defined analogously using a second OWL class `y2:RecordExtensionFacet`. Facet graphs may not contain other facet subjects but apart from that they are not further restricted as shown in the Definitions 4.2 and 4.3. An example for core and extension facets is depicted in Figure 4.4.

Definition 4.2 (Core Facets) *We call $f^x_{core} \subseteq \mathbb{G}'$ a core facet of a resource identified by $x \in \mathbb{U}$ iff*
$y, z \in \mathbb{U}$ and $\exists!(y, \texttt{rdf:type}, \texttt{y2:RecordFacet}) \in f^{core}_x$ with $y = x$
and $\nexists(z, \texttt{rdf:type}, \texttt{y2:RecordExtensionFacet}) \in f^x_{core}$.

Definition 4.3 (Extension Facets) *We call $f^x_{ext} \subseteq \mathbb{G}'$ an extension facet of $x \in \mathbb{U}$ iff*
$y, z \in \mathbb{U}$ and $\exists!(y, \texttt{rdf:type}, \texttt{y2:RecordExtensionFacet}) \in f^x_{ext}$ with $y = x$
and $\nexists(z, \texttt{rdf:type}, \texttt{y2:RecordFacet}) \in f^x_{ext}$.

A so-called *slice* of a subject x is then a named graph created by merging[6] one core facet of x with an arbitrary number of extension facets (Definition 4.4). Merging the three subgraphs depicted in Figure 4.4 would thus result in a *slice* of subject X.

Definition 4.4 (Slices) *Let $\oplus$ be an RDF merge operator as defined in [HeMe04].*
We call the named graph $(n, S^x) \subseteq (\mathbb{U} \times \mathbb{G}')$ a slice with name n of the core facet f^x_{core} and a (possibly empty) set of extension facets $\{f^{y_1}_{ext}, \ldots, f^{y_m}_{ext}\}$ iff $S^x = f^x_{core} \oplus_{i=1}^{m} (f^{y_i}_{ext})$.

[6] A merge of a set of RDF graphs that do not share blank nodes is simply the set union of the contained triples.

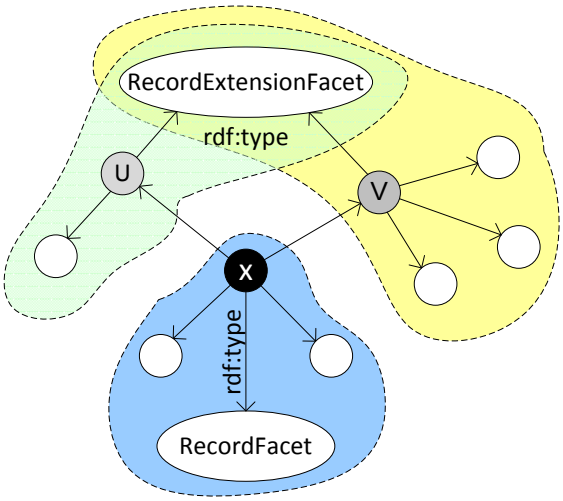


Figure 4.4: A core facet (with subject X) that is associated with two extension facets (with subjects U and V). The dashed lines circumscribe the respective facet subgraphs.

A *layer* is a named graph created by merging multiple slices sharing a common name (Definition 4.5).

Definition 4.5 (Layer) *We call the named graph* $(n, L) \subseteq (\mathbb{U} \times \mathbb{G}')$ *a layer with name n of the (possibly empty) set of slices* $\{(n, S_1^{x_1}), \ldots, (n, S_m^{x_m})\}$ *iff* $L = \oplus_{i=1}^{m}(S_i^{x_i})$.

A *store* is a set of layers including one mandatory *core layer* named by a special URI reference $n_{core} \in \mathbb{U}$ that is a configuration parameter of this store (Definition 4.6).

Definition 4.6 (Store) *A store* $\mathbb{S}^{n_{core}}$ *is a set of layers* $\{(n_{core}, L_{core}), (n_1, L_1), \ldots, (n_m, L_m)\}$ *with distinct names* n_i*. The named graph* (n_{core}, L_{core}) *is also called the core layer of the store.*

Finally, a *record* is an arbitrary set of slices sharing the same subject x. It thus represents multiple "aspects" of this subject resource. In order to integrate records into a store, its slices should be named according to the layers of the store. In particular, such records have to contain a *core slice* that is named as the store's core layer (i.e., n_{core}).

Definition 4.7 (Record) *A record* R^x *is a set of slices* $\{(n_{core}, S_{core}^x), (n_1, S_1^x), \ldots, (n_m, S_m^x)\}$ *with distinct names* n_i *and a common subject x. The named graph* (n_{core}, S_{core}^x) *is called the core slice of the record.*

Integration of such a record into a store is then achieved by simply merging all slice-graphs with the respective (equally named) layer-graphs in the store. We have prototypically implemented the above-described model. The central classes and interfaces of this implementation as well as their relationships are depicted in Figure 4.5. This implementation as well as additional documentation is available at `http://purl.org/y2/`.

4.3.1 Vocabulary

We have developed a simple OWL light vocabulary (the "Y2 model vocabulary"), depicted in Figure 4.6, for expressing the core concepts of our model in RDF.

A store is represented by a resource identified by a dereferenceable HTTP URI. Dereferencing this URI (with content-type text/rdf+n3) leads to an RDF description of this store that includes references to all its contained *layers*. Dereferencing a layer URI leads to a description containing:

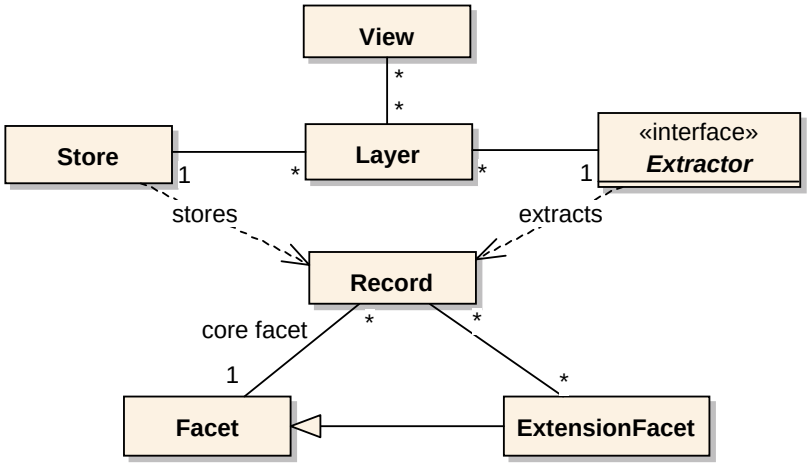


Figure 4.5: Main classes and interfaces of our prototypical implementation.

i) A `dataLocation` URI that can be used to link, for example, to a local file storing this layer's RDF model.

ii) A `readonly` property that, when set to `true`, disallows write access to this layer via the REST interface.

iii) A `recordExtractor` property that is used to link to a resource describing an extractor.

As we used Java for our prototype implementation, we foresaw an `implementingClass` property that stores the *full qualified name* (FQN) of the Java class that actually implements an extractor. This class, that complies with a simple extractor interface (cf. Figure 4.5), is loaded and instantiated dynamically by our store implementation. The extractor instance is then used to extract a collection of facets (including exactly one core facet) from a given layer RDF graph. Our implementation includes some basic but easily extensible extractor implementations that decompose a layer into non-overlapping facets as described above.

Two OWL classes are used to represent `RecordFacets` and `RecordExtensionFacets`. It is notable that resources contained in the merged RDF graph, that is returned when dereferencing a record URI, may be RDF-typed by both classes as a resource may be the center resource of a core facet in one layer and the center resource of an extension facet in another.

4.4 Realization of a file metadata store

For the implementation of a file metadata store using our model, we propose that its core layer contains one RDF resource per file system object and realizes the mapping between an actual file system object and its metadata record by storing the path of this file in a data type property. Additionally, we propose that this layer includes further "core metadata" that is useful for querying the graph and other tasks as described in detail in Chapter 9. These metadata are expressed using RDF properties that represent low-level metadata extracted from the file system itself, such as file names, sizes and hierarchical parent/child relationships.

The *core layer* is maintained by the store itself by extracting the respective metadata from a set of (usually crawled) file system objects and converting them to RDF using the vocabulary presented later in this thesis in Section 9.2.1. This layer is further *read-only* for all other actors, these may, however, add an arbitrary number

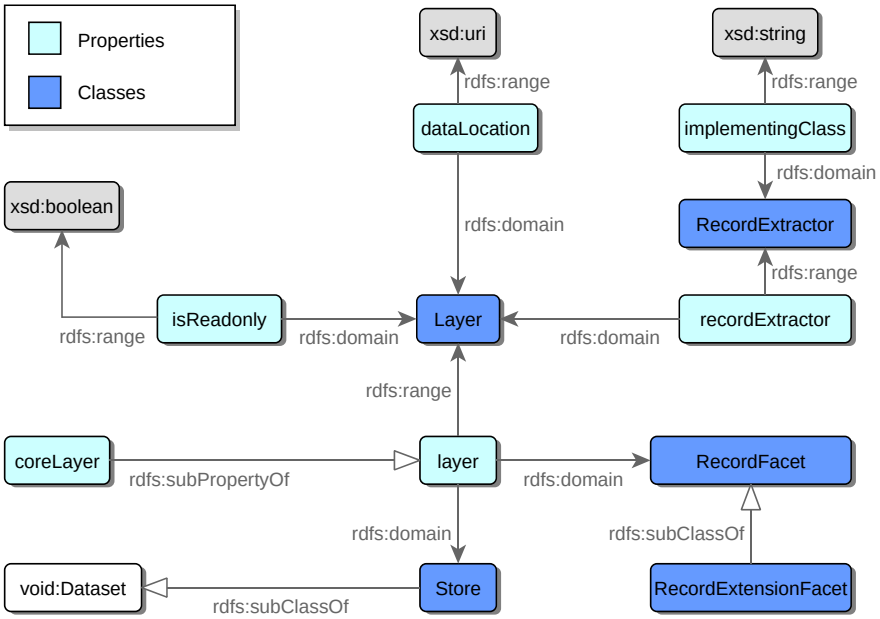


Figure 4.6: Y2 model vocabulary.

of additional layers to the store that contain their own metadata. Such *layers* can later also be removed from a store, which results in data loss, however[7].

For the practical implementation of such a metadata store, consider that a *store* in our model is basically a set of named graphs with some associated metadata. Such a store may thus be realized by a single RDF data set [PeSe08], where the default graph corresponds to the core layer of the store. Most current triple store implementations support this concept and provide a SPARQL service for querying such data sets. The named graphs in this data set are the *layers* that should be used to group "logical aspects" of the stored data.

4.4.1 Considered file system objects

A file metadata store holds metadata about some (not necessarily all) file system objects that can be accessed by it. As mentioned before, file system objects are usually referenced by their location in the file system hierarchy (their *path*) on an application level. Consequently, let $\mathcal{F}$ be the set of all possible file locations (paths) and $\mathcal{F}_{\mathbb{S}} \subset \mathcal{F}$ be the set of all locations that are actually considered by a store $\mathbb{S}$. We do not predefine how a particular store implementation builds the actual set $\mathcal{F}_{\mathbb{S}}$ as there are many useful possibilities to do this. For example, our prototype presented in Chapter 9 includes all file system objects stored in file trees defined by a user-specified set of root folders. This set is further filtered to exclude, for example, files with certain extensions.

File paths are represented syntactically different on various operating systems. They can, however, be uniformly represented by file URIs (hierarchical URIs using the file URI scheme) and a simple function $\upsilon : \mathcal{F} \longrightarrow \mathbb{U}$ may convert a file path to its respective file URI. Implementations of this function are available in most modern programming languages.

[7] For now, we do not specify the exact conditions under which *layers* can be added/removed and read/written as many different models for modeling this are possible. Practical experience will be needed to decide which is the best option in which scenario and so we assume that all but the *core layer* of a metadata store can be read, written, added or removed by any actor using the store and leave possible access control mechanisms as a topic for future research.

Representation of file system objects

We propose to represent the metadata record of an actual file system object in a store by an RDF subgraph centered around one single RDF resource. In order to assign a metadata record to its file system object and *vice versa*, a store has to realize some mapping function between file paths/URIs and the URIs of the corresponding resources in the metadata graph.

The problem with file URIs. A naïve approach would be to use file URIs also for the resources in the metadata graph. The above-mentioned function υ could then be used as a mapping function: The resource representing a file system object could be identified by applying υ to the file location. *Vice versa*, we may use an inverse function υ^{-1} to convert the file URI used in the metadata graph to the actual file path.

However, file paths are not globally unique (the path `C:\a.txt` could be valid in many file systems) and neither are their corresponding file URIs. This means that the integration of a local metadata graph into the global Web of Data would require yet another translation step as globally unique URIs are required in such a scenario. Further, file URIs are not stable: when the path of a file changes (e.g., because it was moved), the corresponding URI changes too. This would mean that the resource URIs around which metadata records are centered change whenever the path of a resource changes which is clearly not desirable as this would, for example, break links to these records that are not within the control of the store [SP10].

Opaque resource URIs. For these reasons we propose to use separate URIs for the identification of file system objects in the file system and in the metadata graph: Whenever a new file system object is added to a store (i.e., to the set $\mathcal{F}_{\mathbb{S}}$), a globally unique and preferably opaque URI is minted for representing its corresponding resource in the metadata graph. Opaqueness means that such URIs should be treated as identifiers only; no additional information should be parsed from their string representation. The file URI of the file system object is then associated with this resource via some RDF property. Mapping such a resource to the corresponding file system object is then simply done by retrieving the file URI via this property and applying υ^{-1} to this value. Mapping a file location to the resource in the graph is done by converting it to its file URI and retrieving the resource that has an outbound arc pointing to this URI[8]. The file URIs stored in this property have to be kept up-to-date by some actor in order to cope with the above-mentioned changes of file locations. We propose an algorithm for doing this in Chapter 8.

URI mappings

We define an actual mapping $\mathbb{M}_{\mathbb{S}}$ between file system objects and corresponding RDF resources in a store $\mathbb{S}$ as a set of ordered pairs of file locations (respectively their URIs) and the URIs of these RDF resources (Definition 4.8). We define such a mapping as *complete* if it contains entries for each considered file path in $\mathcal{F}_{\mathbb{S}}$, i.e., if the corresponding mapping function is injective.

Definition 4.8 (File/URI mapping) $\mathbb{M}_{\mathbb{S}} = \{(u_f, u_g) \in (\mathbb{U} \times \mathbb{U}) \mid u_f = \upsilon(f) \wedge f \in \mathcal{F}_{\mathbb{S}}\}$ *with distinct sets of URIs for u_f and u_g and a store $\mathbb{S}$.*

Such a mapping can be realized in RDF by a set of triples having file URIs as subject and global URIs as objects or *vice versa*. We actually propose to use global URIs as subjects and file URIs as objects for practical reasons explained later. Our proposed model foresees that the *core layer* of a store $\mathbb{S}$ contains such a complete mapping, i.e., we can assert that

Definition 4.9 $\forall(u_f, u_g) \in \mathbb{M}_{\mathbb{S}} \mid (u_g, x, u_f) \in L_{core} \wedge x \in \mathbb{U}$
where L_{core} is the core layer of $\mathbb{S}$, x is some RDF property and $\mathbb{M}_{\mathbb{S}}$ is a complete mapping as defined above.

[8] The file URIs could be represented as own RDF resources or simply as literals in the actual RDF graph. In the latter case, it would also be possible to store the actual file path in the literal. Note, however, that in practice some normalization steps for dealing uniformly with file paths may be required (e.g., uppercase/lowercase conversions) which could be done already by the function υ.

Again, the used RDF property stems from our vocabulary presented in Section 9.2.1. The *core layer* of our proposed file metadata store can be seen as the "backbone" of our model. It contains statements about all central information entities (i.e., the file system objects) and by querying it, it is possible to learn about all considered file system objects in a metadata store (i.e., $\mathcal{F}_{\mathbb{S}}$). All other layers in a store might contain data describing only a subset of $\mathcal{F}_{\mathbb{S}}$.

4.4.2 Two-dimensional compartmentalization

A file metadata store is structured along two dimensions. While layers structure a store "horizontally", *records* group resource-related descriptions from all layers, thus being a kind of orthogonal ("vertical") structuring concept. A (complete) record describes all available "aspects" of a resource and may thus be conceived as the current "view" on a resource in the respective linked data set[9]. However, records are also layered (layers of a record are called *slices* to make them distinguishable). This enables the retrieval and manipulation of partial records that represent only certain aspects of a resource.

The benefit of this data compartmentalization is that it is now straightforward to deliver or modify the various layers in a store separately or to attach differing access policies to them. A metadata store could, for example, introduce one separate layer for each application that writes to it. These applications would then write to their layers exclusively which avoids interoperability problems. It may, however, also be possible to share the access to a particular layer between applications that need (write) access to shared metadata aspects. A further, practical advantage is that a store may allow concurrent access to its layers which is clearly beneficial for a file metadata store where performance is an important issue.

Record facets

Records are further decomposable into so-called *facets*, sub-graphs of a particular slice of a record. A single, mandatory *core facet* (per slice) describes the data that is exclusively related to the particular subject of the record. A slice may, however, further contain an arbitrary number of extension facets that may be used to describe aspects of a file system object that are *shared* between multiple records.

Consider, for example, an application that extracts ID3 tags from MP3 files and stores them in our store. This component would create a core facet for describing the MP3 file, using literals to describe metadata directly related to this piece of music such as its title. For representing the MP3's artist, however, an own resource with associated metadata (e.g., including `owl:sameAs` links to respective DBpedia resources) could be created and linked via an object-property to the MP3 resource. This artist sub-graph represents data that is potentially shared between multiple records as many MP3s may share the same artist(s).

In this example, the sub-graph describing the MP3 would be the *core facet* while the artist sub-graphs would be extension facets. Figure 4.7 depicts such a situation. The resources X and Y represent two MP3 files that are associated with two artist resources (U, V) each. The difference between core and extension facets becomes clear when considering what happens when the respective records are deleted from the store. As core facets describe exclusive metadata about the record, they are deleted in any case. Extension facets, however, may only be deleted if they are not *referenced* by any other record anymore. Thus a deletion of record X in Figure 4.7 leaves its extension records untouched. A subsequent deletion of Y, however, would trigger the deletion of U and V as well.

Facet extractors

The core issue regarding our model is *how* (core and extension) facets are extracted from a given RDF graph (i.e., from a store layer or a record slice). More formally, we are looking for a function $\epsilon : \mathbb{U} \times \mathbb{G}' \longrightarrow \mathcal{P}(\mathbb{G}')$ that extracts a set of *facets* describing a record-resource (identified by a URI) from a given RDF graph

[9] Note that we actually do call a set of *layers* a "*view*" in our implementation.

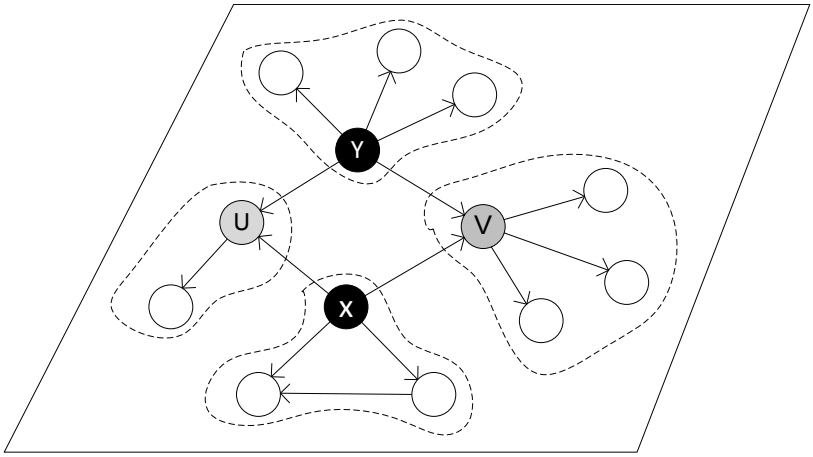


Figure 4.7: A store layer containing two records (X, Y) which have two core facets (denoted by the dashed lines around *X* and *Y*) and two extension facets (dashed lines around *U* and *V*) each. The assertions of their respective rdf-types are omitted for brevity. The reference count of these extensions is 2 each, in the depicted situation. Deleting one of the two records would remove only its core facet and reduce the reference counts to one. Consecutively removing the other record would then remove this record's core facet as well as both extension facets.

without bnodes. We have already discussed several proposed methods for the extraction of resource-centered subgraphs in Section 4.2.1 and briefly discuss their applicability here:

Simple extractors. A simple way to extract a useful sub-graph is to consider only triples that have the resource as subject. Stickler [Sti05] and others call this kind of representation "asymmetric", whereas a "symmetric" description would also include inbound arcs (i.e., triples that have the center-resource at the object position). Various other proposed methods, such as *Concise Bounded Descriptions (CBD)* [Sti05], *RDF molecules* [DFP+05] and *Minimum Self-contained Graphs (MSG)* [TMPP05] are basically reduced to these simple asymmetric or symmetric "shells" when considering graphs without bnodes. This simple extraction method can be implemented easily and efficiently but allows only simple (name/value pair) metadata descriptions.

Pattern-based methods. More complex graphs may be extracted using SPARQL CONSTRUCT queries where the graph patterns of the selection clause match the graph template used for constructing the result graph. The implementation of such extractors would be simple as most RDF repositories already support SPARQL access. However, using these extractors would require the execution of one SPARQL query per extraction process which might be too costly for a file metadata storage.

Explicit methods. Another possibility is to use named graphs: one could reuse the URI of the center-resource as the name of a named graph containing all triples describing a respective facet. This explicit method for defining a sub-graph does not restrict what triples are contained in a facet graph and does not require a special function for extracting them. It is, however, not applicable in a straightforward way for our scenario because of the following reasons: (i) We already use named graphs for representing layers. As facets are sub-graphs of layers, this scenario would require a concept for embedding named graphs into named graphs. We are currently not aware of an appropriate storage infrastructure for this. (ii) It would

further require a format for serializing such hierarchically organized named graphs which is currently not available. (iii) It would lead to a large number of named graphs (one per facet), which is unlikely to be efficiently handled by current triple store implementations. (iv) Further, compartmentalizing the layer graph into named sub-graphs would make queries to this compartment more complex and probably slow.

Custom algorithms. Finally, custom algorithms can be used to extract sub-graphs of arbitrary complexity. Such individual implementations have the benefit that they might also resort to external data and contextual information for this task and might be specialized for a particular domain.

Whatever extraction function is chosen for sub-graph extraction, the fundamental requirement is that it enables lossless graph decomposition and merging. This means that adding, removing or updating records in a store should not interfere with other records (except for updating shared descriptions, i.e., extension facets). One way to achieve this is to avoid that facets overlap within a layer, i.e., that they do not share triples. This is, for example, achieved when the simple *asymmetric shell* mentioned above is used (which, however, allows only the representation of name/value pairs and links). The *symmetric shell* would obviously not show this property as each extracted facet would contain all triples having the center-resource as subject or object.

Core slices. As distinct layers represent varying aspects of a data set, it is likely that they will require differing models for expressing their data. Allowing different extractors for distinct layers is thus a logical consequence from this as the various actors writing to a metadata store will use all sorts of metadata models that probably differ in complexity. While one application might get along with simple name/value pairs (and could use a simple *asymmetric shell* extractor for the layer it writes to), another, such as the above-mentioned MP3 handler, would probably require a custom-tailored extraction function that extracts, e.g., the extended "artist" facets from an RDF graph by resorting to a certain semantic vocabulary (e.g., a property from this vocabulary could be used to connect a MP3 resource to its artist resources).

We do not predefine the layers of a metadata store except for the mandatory core layer. For this core layer, our actual file metadata store implementation uses the described *symmetric shell* extractor for extracting slices. The extraction of such a *core slice* is done by a function $\rho_{core} : \mathbb{U} \times \mathcal{P}(\mathbb{U} \times \mathbb{G}') \longrightarrow \mathbb{G}'$ as defined in Definition 4.10. This function extracts all triples from a store's *core layer graph* where the passed URI reference is the subject or the object.

Definition 4.10 (Core slice) $\rho_{core}(u, \mathbb{S}) = \{(s, p, o) \in L_{core} \mid s = u \vee o = u\}$
where L_{core} is the core layer of store $\mathbb{S}$ and u is the URI of a file system object's metadata record.

Implementers of metadata stores, however, are free to implement other, more sophisticated extraction methods for core slices possibly based on the above-mentioned subgraph extraction methods. In some scenarios, it might even be useful to transform the extracted metadata graph before it is used by an application in order to satisfy existing interfaces (cf. Magkanaraki et al.'s work on RVL lenses [MTCP04] and Hung et al.'s work on RDF views [HDS05]).

Consistency via reference counting

It is important to understand that record extraction methods have to be implemented by the store itself as they define the "building blocks" of a metadata graph, i.e., the resource-centered subgraphs that CRUD (create, read, update, delete) operations are applied to. This also means that store implementers have to make sure that applied CRUD operations result in consistent states, for example, that no metadata "debris" is left behind when deleting metadata records from the store. This is trivial as long as there is no overlap between the record graphs which would, however, disallow shared resources such as the MP3 artists from above. To allow the consistent management of such shared resources[10], we propose a simple mechanism based on reference

[10] It is noteworthy, that no comparable mechanism for internal metadata exists.

counting of shared *extension facets*: In our model, each layer in a store is assigned exactly one extractor implementation that implements the above-mentioned extraction function ϵ as well as a "reference counting" function $\rho : \mathbb{G}' \times (\mathbb{U} \times \mathbb{G}') \longrightarrow \mathbb{N}^0$. This function accepts an extension facet and a layer as parameters and returns the number of core facets contained in the layer that reference this extension facet. This reference count (*refcount*) is then used by the store to decide whether an extension facet is removed together with its core facet or not. A simple method for this is to count the number of triples hat directly link core facet subjects with a particular extension facet subject. Reconsidering Figure 4.7, the *refcounts* of the two extension facets U and V would thus be 2 each. Deleting X, would decrease these refcounts to 1 and the store would therefore delete the extension records after deleting their last referencing core facet (Y). In case of the above-mentioned ID3 example, a *refcount* could be determined by counting only the triples with the predicate that was used to connect core and extension facets. Another possibility is to explicitly represent the reference count in a *literal* attached to the extension subject. In this case, however, respective functionality is required that keeps this value up-to-date.

4.4.3 REST interface

For accessing a metadata store, we propose an HTTP REST interface [Fie00] through which CRUD operations to its contained records can be applied. Reading records from a store can then be done by sending standard HTTP GET requests to their respective URIs. The store responds to these requests by serving the record descriptions in Linked Data style by[11]

i) extracting all according record facets from each layer using the respective extractor;

ii) merging these extracted record slices to one single RDF graph [12];

iii) converting/serializing the resulting graph to the requested content type;

iv) serving the results in the HTTP response.

Content and range negotiation. As with most other Linked Data services, clients can negotiate the content type of a returned representation (HTML or RDF) with our server. Due to the strong compartmentalization of our model, however, we added a second kind of "negotiation" possibility. Clients may request that the server constructs the returned description only from a sub-set (called a *view*) of layers. This is done by passing a list of layer names (i.e., URIs) in the HTTP *range* header of the request.

The HTTP range mechanism [FeGM+11] allows the access of partial resource representations. The main intention of this mechanism is the reduction of unnecessary data transfer. HTTP 1.1 enables clients to select byte-ranges of resource representations (e.g., 500-999/1234 selects the second 500 bytes of a 1234 bytes long representation) and a range-header aware server should answer these requests with a message containing an HTTP 206 (Partial Content) response code. The HTTP range mechanism explicitly foresees the possibility to define custom "range units" that can be used to specify a range[13]. We therefore propose this mechanism for selecting what layers a delivered record description should contain. The following command line, for example, requests the annotation layer representation of the record X in N3 notation:

[11] Note that this applies only for record center-resources. Our model does not specify what description is returned for other resources, the current implementation handles this case by simply returning their *asymmetric shell*.

[12] Note that it is also possible to access a decomposed version of a record serialized in TriG via content negotiation.

[13] Note that RFC 2616 (HTTP 1.1) is a bit vague regarding whether custom range units may be used for the Content-Range header. Although the running text states that such custom ranges can be used, they are not foreseen in the EBNF of section 14.16 Content-Range. We understand that this issue is work in progress and designed our solution with regard to the latest IETF draft regarding HTTP Range Requests and Partial Responses [FeGM+11].

```
curl -H "Accept: text/rdf+n3"
     -H "Range: layers=<http://mystore.com/y2/conf/anno>"
     -X GET http://mystore.com/y2/r/X
```

Writing records. Records can be updated by sending RDF descriptions to their HTTP URIs using HTTP PUT or HTTP POST requests[14]. As records are RDF datasets from a structural point of view, we serialize them using the TriG format [BC07], a simple text format for serializing named graphs. Updating a record in the store is thus done by, e.g., sending an HTTP PUT request containing a TriG document to the URI of the record's center resource. The server de-serializes the record from the TriG document in the following way:

i) for each named graph in the TriG document, a corresponding layer (i.e., a layer with the same name) is looked-up in the store;

ii) the record extractor of these layers is used to extract the record facets from the named graph;

iii) the extracted facets are merged to create the record slices.

Now, such de-serialized records may be integrated in our store by merging its slices with the respective store layers as described before.

A record might be *invalid* with respect to such a write operation, e.g., if not all named graphs have corresponding store-layers, if no core slice was found or if some layers are not writable. In such a case, the server answers with a respective 4XX HTTP status code (e.g., *416 Requested Range Not Satisfiable*, *400 Bad Request* or *403 Forbidden*). Otherwise, the record is updated in the store by first removing its old version and then merging its slice graphs with the respective layer graphs. Note that this may also result in an update of shared extension facets. This procedure of integrating records into a layered Linked Data store is depicted in Figure 4.8.

Our model supports also the creation of records via the REST interface: Sending a TriG document via HTTP PUT or POST to URIs that are under control of the store but are not yet mentioned in the core layer of the store results in the creation of a respective record. This, however, is disallowed in our file metadata store as the records in this store are created exclusively by the store implementation itself.

Record deletion. Analogously, the server accepts HTTP DELETE requests to the respective resource URIs. The server extracts the respective record from its layers and deletes these triples. Note that the above-mentioned range negotiation is used for the whole REST interface. Thus, replacing the HTTP method by "DELETE" in the above-mentioned *curl* request results in deleting only the annotation layer of resource *X*, leaving all other layers untouched.

Bulk creations/deletions. In order to support bulk creation and deletion operations (useful, e.g., for store synchronization, etc.), there is also the possibility to send a collection of records to a special resource of a store: its *RDF sink*. This resource is backed by a servlet that accepts HTTP POST and DELETE requests. Collections of records are parsed from the TriG document embedded in the HTTP request and the respective storage operations (create, update or delete) are executed on each contained record.

4.4.4 Other aspects

Synchronization of internal metadata

As described before, some file formats support internal metadata expressed by either proprietary formats or by some metadata standard. Consider the Exif properties in Figure 4.3. These metadata are stored internally in the metadata headers of the corresponding JPEG file.

[14] Note that this obviously works only if the respective URIs are part of this store's dereferenceable URI space.

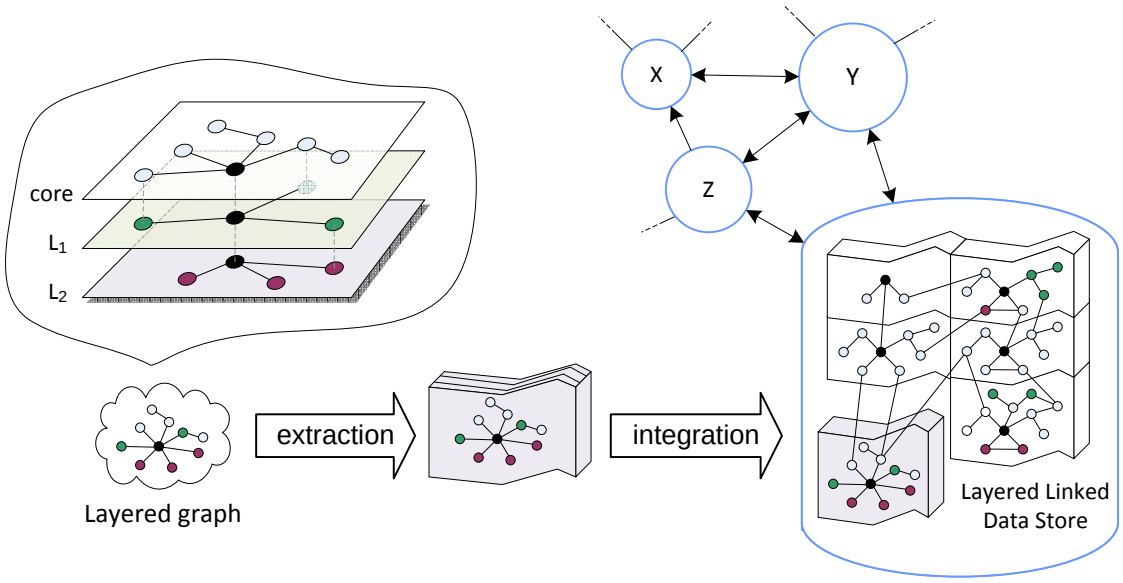


Figure 4.8: A record with three layers (*core*, L_1, L_2) is extracted and integrated into a layered store that is itself linked with other Linked Data sources. Record subjects (center resources) are depicted as black circles.

We do not propose to radically replace the current practice of internally stored metadata. In fact, internal metadata are successfully used in many domains and maintaining it provides a way to remain backward-compatible with a large variety of applications. We therefore propose a way to integrate these internal metadata in a stable manner into a store which enables clients to access these data in a uniform way, e.g., in searching or reasoning tasks. However, simply replicating them into the graph at the time a file metadata record is created is not sufficient as this would lead to inconsistencies between internal and external metadata over time. Instead, one or two-way synchronization between these metadata is required:

Read-only synchronization. A straightforward solution for this problem is to replicate these metadata (or parts of them) into a *read-only* subgraph. This graph would have to be updated whenever a change of the internal metadata is observed. A technology that supports this would have to know how to read this internal metadata from various file formats. It would further have to provide a mapping of these data to RDF. It is noteworthy that some file formats already use RDF as a model for internal metadata[15]. In this case, integrating this internal metadata into the global metadata graph might be as simple as an RDF graph merge. As described above, we propose such a read-only synchronization for mirroring low-level file metadata (such as file names, file hierarchy, etc.) into the core metadata layer.

Bidirectional synchronization. A more advanced approach would be to actually synchronize the metadata in both directions, i.e., to also update the internal metadata when the external metadata changes. A technology that supports this would additionally have to know how to actually *write* metadata headers of various supported file formats. Consequently, this would enable all actors that know how to write a certain layer of the store in order to update also the internal metadata of the respective file system objects without knowing how to access this particular file format. Further, such a system would be fully

[15]For example, the Adobe XMP standard (Section C.1.3) is based on a subset of RDF. Another example is the OpenDocument file format (ODF) used by the OpenOffice suite. Version 1.2 of the ODF supports the expression of metadata in RDF [EE11]. A number of already existing (partially prototypical) tools that RDF-ize the metadata of files stored in various file formats can be found at http://esw.w3.org/ConverterToRdf

backward-compatible with applications that cannot write to or read from the global metadata graph but would rather access only the internal metadata of files.

Application or store components that implement these synchronization strategies may replicate (parts of) the internal metadata of certain file formats into certain *layers* which would then be synchronized in a one-way (read only) or bidirectional (read/write) manner. An example for such a component is the above-mentioned MP3 handler that reads ID3 headers from audio files and converts them into an RDF representation using an appropriate vocabulary. This component would be notified by the metadata store whenever a file system object is created or updated that possibly contains such headers. It would then extract the respective RDF graph for this file system object and merge it with a certain *layer* of the store (e.g., a special "ID3" layer). If this module implements a bidirectional synchronization strategy it would further have to listen for notifications about changes in this layer. Whenever such changes take place, appropriate write operations to the ID3 headers of the respective file system object would have to be done. We have implemented such a bidirectional MP3 handler prototypically and describe it in Section 9.2.4.

Record index

Although the data in our model is strongly compartmentalized into layers, an index of all records contained in a store can easily be retrieved. Reconsider that each record contains a mandatory core facet which has to include a triple that asserts the RDF type of the record's core facet using a special OWL class. This core record slice will then be merged with the core layer of the store and thus a list of all contained records is easily retrievable from this graph using a trivial SPARQL query. Such an *index* is useful in many scenarios, e.g., for search engines or other indexing applications [FDP+05, TOD07, HHD+07a, PH11].

Record groups

Decoupling metadata from files would actually overcomes their 1:1 relationship and allow it to assign metadata records to groups of files as depicted in Figure 4.9. Arbitrary groups of files can be associated with an additional metadata record that describes this group. Note that our model does not require that all data stored in a metadata store can be decomposed into resource-centered, file-related *records*. This means, that a layer graph may also contain triples that are not contained in any record. Examples for such data are triples describing a semantic vocabulary or triples describing *groups of records*. We do not specify how such data has to be structured, e.g., what grouping constructs (RDF containers, RDF collections, etc.), should be used. The data can be stored in any layer as long as it is consistent with the above-mentioned definitions (e.g., it may not contain blank nodes).

Store implementers should foresee proper mechanisms (possibly also extractors) for preserving the consistency of such layers with the remaining store. For example, a layer that contains metadata describing a *group* of records as depicted in Figure 4.9 should be assigned an extractor that reacts on record removal events and in turn removes references to these records from all respective groups.

4.5 Related work

4.5.1 Writable RDF

SPARQL UPDATE. Obviously, consuming and updating Linked Data sets via SPARQL in general and the currently discussed SPARQL 1.1 Update extension [GePePe11] in particular are related work to ours. Our proposed model should, however, not be considered in any way as some kind of replacement for SPARQL but rather as a complementing strategy for storing and accessing Linked Data that is pre-structured in some form. The record extraction functions that determine the "silhouettes" of the various record slices in our

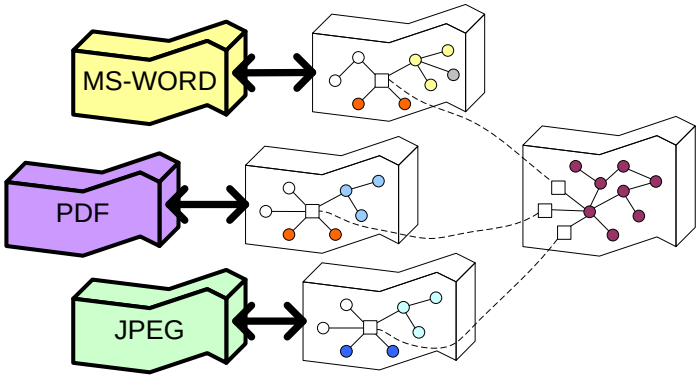


Figure 4.9: Metadata records assigned to groups of files. Note that the associated metadata graph (on the right) is not simply a merge of the individual files' metadata graphs. It may contain metadata specific for the defined group of files.

model enforce a common structure of the stored graphs respectively their facet sub-graphs. This may be beneficial when consuming these data as less-generic data handling interfaces (e.g., import interfaces, GUIs) are required. On the downside, this also restricts the modeling possibilities for the Linked Data in such stores. Although this restriction is alleviated by the possibility to provide multiple layers, each containing differently structured aspects of the data, our approach cannot be a general but rather a domain-specific solution for Linked Data. Generally spoken, the main difference between our solution and SPARQL is that the latter acts on the triple level of RDF graphs while we introduce logical, decomposable entities (layered records) and act on the descriptions of these entities.

The form of such entities is determined by extractors on the server side which is comparable to the SPARQL DESCRIBE semantics. In both approaches, the client requires no prior knowledge about the structure of the returned description. As our records are decomposable into layers, we additionally enable filtering of such descriptions by layer and thus also the retrieval of partial descriptions of a Linked Data resource. SPARQL also provides methods for filtering triples from a result set, even on a per-named-graph basis (namely, graph patterns and FILTER constraints) which could, arguably, also be used for the retrieval of partial records. The difference is, however, that in this case the data consumer explicitly decides how these partial records look like while in our approach the *writer(s)* actually determine the contents of a record slice and thereby also what is returned when this record is requested by a consumer. This is also reflected in the ways how such partial descriptions are requested: In SPARQL, the respective filter and selection statements would be part of the query itself (and thus part of the HTTP message payload), while our solution relies on negotiation via HTTP headers. This allows it to handle the availability or accessibility of partial records on the transport protocol layer which might be beneficial in certain situations, e.g., for caching purposes.

Writing to our store is also handled differently as in SPARQL UPDATE. Although clients may send any RDF data to our service, only properly extracted sub-graphs will be further processed. The parts of the transmitted descriptions that do not match the structure predefined by the given extractor functions are filtered out. This is different in SPARQL UPDATE where there are no restrictions for the client to specify what would be updated in a *graph store*.

Nevertheless, our store is in the end represented by a single RDF dataset that can be accessed via SPARQL like any other graph store. Actually, we do believe that a combination of both access methods could be beneficial in many scenarios where both levels of granularity (triple and record) are useful. For example, SPARQL could be used to query a store as usual (e.g., "return all URIs of JPEGs that are younger than x days"), accessing these URIs via the REST interface could then be used to exchange whole logical entities

(e.g., an image record as depicted in Figure 4.2), and SPARQL UPDATE could be used to update the store in situations where client-control over the data representation is useful (e.g., applications might want to send user-contributed RDF annotations that are not pre-structured to a particular graph layer via SPARQL UPDATE).

Tabulator. A generic user interface that uses SPARQL to build a kind of readable and writable "data wiki" is the Tabulator [BLHL+08]. *Tabulator* is not restricted to a single data source and treats Linked Data in a generic fashion on the triple level, i.e., no logical containers comparable to our records are available. The decision where modified triples are written to depends on their provenance; new item-related triples are added to the documents that mentioned them. Writable RDF documents are flagged by HTTP headers as such. Technically, WebDAV and SPARQL UPDATE are used to actually write RDF data.

Tabulator does not introduce any actual restrictions to the structure of the edited RDF graphs. This genericity is the main strength but also weakness of this approach in our opinion. Any data source can be filled with any triples which basically dissolves this compartmentalization mechanism as one cannot predict what data is found in what source anymore. This, however, raises questions of practicability. Besides the difficulties to develop appropriate read and write interfaces, the existing "hard" boundaries between data sources are usually exploited in many ways, e.g., for deciding on the provenance and reliability of the data or for access restrictions. It is further exploited to actually navigate to the data one wants to consume: certain data sources are "known" to contain certain kinds of data. For this reason one does not have to search the Web every time but may bookmark Web resources as consuming them at a later point in time is likely to return the same or comparable information.

Social rules may practically restrict what data is written in what "structure" to what data sources with Tabulator-like software and this might work quite well, e.g., for a data wiki. Other applications, however, might require some explicit and controllable mechanism for data structuring.

Not only the structure of these data but also their compartmentalization (respectively, the possibility to group data sharing some common feature) is relevant in real-world situations. One practically highly relevant application for this is access control. When many applications write into one common metadata store, it should be possible to manipulate and access the contributed metadata independently to avoid that the various applications mutually overwrite their data. Administrative services might, however, require full access to all metadata in a store, etc. Another relevant issue in our scenario is that metadata contributed by the various applications can be manipulated and read independently from all other data.

Finally, the generality of the Tabulator approach raises user interface design questions: How to build user-friendly GUIs for entering new/editing existing RDF data? One possible approach to this is shown by RDFauthor [THAF10], an approach for editing RDF graphs embedded in RDFa annotated HTML pages. RDFauthor also uses SPARQL UPDATE for propagating changes to a respective SPARQL endpoint. It consists basically of a JavaScript API that parses RDFa annotations from HTML pages and automatically builds a user interface using appropriate input widgets for the various properties. This user interface hides much of the complexity of the underlying RDF data model and is more user-friendly than the Tabulator GUI. Nevertheless, such generic user interfaces can never reach the usability of specialized, intuitive GUIs that can be developed when one knows about the basic structure of the edited data, especially when patterns more-complex than simple name/value pairs are used.

4.5.2 Related work from Semantic Web research

TripFS. The triple file system (TripFS), the author contributed to, was a first attempt to create a prototypical, linked file system [SP10, PS10]. It also represented file system objects by RDF resources and provided infrastructure to add extractor and linker plug-ins that could enrich such resources with semantic metadata. We made use of DSNotify, our software component for maintaining link integrity in the Web of Data, to synchronize the file paths stored in TripFS with the actual file locations. DSNotify is discussed in detail in

Section 6 of this thesis. However, TripFS had a much simpler data model that did not provide means for data organization beyond the standard RDF functionality. It can, to some extent, be seen as an early predecessor of the approach presented in this section.

Nepomuk Representational Language. The Nepomuk Representational Language (NRL) was originally designed to fit the needs of the Nepomuk Social Semantic Desktop applications [SvESH07]. It builds upon RDF and RDFS but introduces additional concepts for data modularization. NRL assumes a closed-world scenario, which means that facts that are not stated as true are automatically considered as being false. Similar to our model, NRL also proposes to use an RDF dataset for storing local metadata. The *default graph* of this data set contains all triples that are not explicitly assigned to another named graph. Roles that indicate the characteristics of the contained data can be assigned to named graphs. NRL keeps graph metadata separated from the graph itself in a second *metadata graph*. Further, NRL *Graph Views* are "interpretations" of a named graph that are defined by *View Specifications*. View specifications are executable components (e.g., a SPARQL query or some external application) that return the "view realization" (i.e., the triples) of a particular view. Our proposed model differs from the NRL model in several points:

i) Our metadata layers seem similar to the named graphs in the NRL model. They could, technically spoken, be realized by introducing a special graph role for NRL (e.g., derived from `nrl:InstanceBase`). On an informal semantic level, however, our *layers* were primarily designed to contain metadata describing the *records* of a store in various ways, while NRL graphs may contain arbitrary data (e.g., instance data, ontologies and knowledge bases). Consequently, these concepts differ in the actual metadata used for describing them (in our model, this includes, for example, what record extractor should be used with an aspect while this is not graph-bound in NRL).

ii) Our model defines a *core layer* that contains basic metadata about the records stored in our metadata store. This means that the URIs of all records stored in our store can be retrieved easily from this core layer.

iii) Further, the two models differ in the ways how the used RDF data sets are decomposed into useful subgraphs. In our model, resource-centered *records* are "built" by applying the configured subgraph extractors to the *layers* of the metadata store. These extractors are implemented by the store itself and make consistent CRUD operations for whole metadata records possible, however, at the cost of a restriction with regard to model expressiveness. A NRL graph view, on the other hand, can be any subgraph extracted from a named graph and basically realizes only the R of CRUD.

Siles. An alternative model to the classic file metaphor is the *Siles* model [Sch09, SH10]. Siles are uniquely identified containers consisting of arbitrary *content* and *annotations* which include tags, categories, attributes and semantic links to other siles. By that, they are conceptually close to standard files containing embedded metadata headers. By design, their data and metadata are self-contained, thus not directly depending on other data. Siles are identified by globally unique URIs, their metadata are pre-structured into the four mentioned categories. It is notable that the Siles concept does not impose any hierarchical constraints on the organization of siles. The self-containment constraint of the concept, however, restricts its expressiveness. Siles are, for example, restricted to an embedded link model, n-ary links or groups of siles are not supported by the model. A further problem with siles is that no provenance information is included in the model. When considering a particular sile it is not possible to find out what (kind of) application contributed what metadata. As a consequence it is also impossible to restrict access to this metadata in a clean way as advanced compartmentalization concepts are missing. Further, the reason for separating the metadata into the mentioned annotation categories remains unclear as the actual reference implementation stores this metadata in an RDF graph that is not subject to such restrictions.

4.5.3 Other related work

Windows registry. The Windows registry[16] is a hierarchically organized, local metadata repository for storing configuration settings of applications installed on a Windows PC[17]. Nodes in the registry hierarchy are called "keys" that may contain arbitrary numbers of sub-keys or typed name/value pairs. The registry is organized in so-called *hives* that compartmentalize it into logical sections. Such logical sections comprise, for example, all user-profile specific settings or all settings specific to the local computer. By this, different user profiles may, for example, resort to different registry settings. This registry is in a sense comparable to our proposed metadata store as it represents a local, shared metadata repository that is accessible via a uniform access interface. However, metadata organization in the registry is restricted to hierarchical organization and simple name/value pairs. Links between local and remote resources are not supported. Neither are advanced compartmentalization concepts beyond hives. Although the registry provides means to restrict the access to certain keys, it also suffers from the data debris problem mentioned above. There is, for example, no mechanism that ensures that data written to the registry by some application is completely removed when this application is de-installed. Missing or wrongly updated registry keys and values may lead to unexpected application behavior or even to crashes of the operating system. Although one might think that a shared registry without mechanisms to prevent these problems should quickly become corrupted this is rather seldom the case for the Windows registry nowadays. One reason is arguably that application developers put some effort into their own mechanisms to prevent data corruption in the registry. This is why we are confident that such a rather unrestrictive policy might also work for our proposed metadata store.

File forks and alternate data streams. Advanced file system concepts like file forks or alternate data streams could also be used for implementing an external metadata model. RDF subgraphs describing a file system object's metadata could be stored in its forks (as internal metadata). Unfortunately there are some major issues with the implementation of forks/ADS as discussed in Section 2 and in Appendix A. The main issue is that metadata stored in forks might become lost when such files cross file system boundaries. Further, storing the metadata in a file fork would not decouple the storage location of files and their metadata, i.e., file grouping would not be supported naturally by this approach. Another issue is that some additional, central indexing infrastructure would be required to enable queries over the whole set of file system objects. Forks could, however, be used for caching record slice graphs to enable faster access. Loosing this cached data would not be critical. Such an approach would, however, require a synchronization strategy similar to the ones described for mirroring internal metadata.

4.6 Discussion

Our proposed metadata model is fundamentally different from the *status quo* on the desktop as it cleanly separates files from their metadata. As discussed throughout this chapter, a logical and physical separation of metadata and file system objects has various advantages over the currently predominant model of internal metadata. These advantages include the possibility of separate access, more fine grained access control, or the possibility to associate multiple files with a single metadata description.

Our model does not define or restrict the semantics of the stored metadata, i.e., it is not a metadata schema. It rather defines concepts for multi-faceted data organization and compartmentalization. We introduced logical compartmentalization of resource descriptions in two dimensions: "horizontal" layers and "vertical" records. Records are layered RDF graphs, each layer representing some logical aspect of the description of a record's subject. This subject is a Linked Data resource: dereferencing its HTTP URI leads to a representation of this resource (the record) available in different content formats such as RDF or XHTML.

[16] See http://msdn.microsoft.com/en-us/library/ms724871(v=VS.85).aspx.
[17] On Mac OS X, comparable information is stored in binary property list (.plist) files.

Partial records can be requested by providing lists of layer names in HTTP range headers which leads to partial descriptions of only these aspects of a resource. An index of all records contained in a store, useful, e.g., for implementing semantic search services, is automatically maintained.

Our model also foresees the existence of a *core layer* that contains a file location/URI mapping that relates metadata records with actual files in the file system. This *core aspect* contains low-level features of file system objects and further describes their hierarchical organization in the actual file system. This enables semantic queries that incorporate file-related metadata as well as parent/child relationships between file system objects.

One important property of our model is that it introduces some structuring into Linked Data resource representations while not restricting the flexibility and expressiveness of the graph-based RDF model too much. Structures of metadata descriptions are determined by extraction functions that are pluggable into a Linked Data store. Extractors may define the space of allowed structures for a record description based on all levels of the Semantic Web Stack and even based on external (e.g., contextual) data. They must, however, together with the way the data is stored ensure that lossless graph decomposition is possible to enable CRUD operations on RDF subgraphs. Developing non-trivial, efficient extraction functions that fulfill this property is the hardest thing when implementing our model for real-world data sources. However, when this tasks succeeds, one is benefited with a store whose data structuredness lies somewhere between the fixed structures of object-oriented or frame-based systems and the extreme flexible structures achievable by arbitrary RDF graphs.

Our proposed model can be implemented using standard triple store and Web server technology and we present a prototypical Java implementation of it later in this thesis (Chapter 9). It does, however, not touch the complex topics of security, privacy, access control and related issues. Although these are clearly missing features for using our implementation in real-world scenarios as described in this thesis, we are confident that existing research on these topics may be integrated with our solution (e.g., [HPBL09, HZ10, SP11]).

We have discussed advantages and disadvantages of our model in comparison to generic methods that directly operate on the triple level based on SPARQL (UPDATE). Our solution is no replacement for such generic solutions. It operates on a different "level" of abstraction and may be used as a complementing strategy for manipulating Linked Data.

Further, we discussed general benefits of a graph-based metadata model. As file system objects are represented by RDF resources, they can also be related with each other using semantic links. Links can also connect local and remote resources. Further, our model supports the synchronized replication of internal file metadata into the metadata graph. In a summary, our approach leads to a kind of *semantic overlay graph* that interconnects semantically related local resources and is integrated into the global Web of Data as depicted in Figure 4.10. This overlay graph can then be exploited for improved navigation and other tasks and is amenable to existing Semantic Web technologies, access would be possible via unified and standardized interfaces. As an immediate effect, such a graph would enable sophisticated file queries, for example:

i) Search for all video files that are larger than a given byte size.

ii) Search for all high-resolution images (of various MIME types) older than three months in order to compress them automatically.

iii) Search for all JPEG images related to a particular person represented by some remote FOAF profile. Note that such a query can be executed even if the respective FOAF profile cannot be accessed (e.g., due to the lack of a network connection).

iv) Search for all project-related files and Web resources. A project could be modeled either by an explicit group of resources (cf. Figure 4.9) or by resources sharing a certain semantic tag (e.g., some SKOS concept).

Some actual SPARQL queries are presented later in this thesis in Section 9.1.5.

Finally, we have discussed in detail how our generic metadata model can be used for the organization of file-related metadata. In a summary, this approach is based on the following core ideas:

i) Each file system object is represented by an RDF resource and described by a layered RDF graph centered around this resource.

ii) A *core layer* stores a File/URI mapping and describes the low-level metadata of these file system objects as well as their hierarchical relationships.

iii) A set of named graphs (*layers*) is used to model different aspects of a file system object's metadata. Each *layer* may contain metadata for an arbitrary subset of the considered file system objects.

iv) Layers may also contain RDF data describing things that are not directly related to a single file system object, such as *aggregations of files* or semantic vocabularies.

v) Different *views* on these metadata are provided by merging different sets of *layers*.

vi) Metadata subgraph extraction for a particular file system object is done by applying *functions* that extract a resource-centered subgraph from such *views*. These extracted *records* are the building blocks of the store, CRUD operations are applied to them.

Although our model seems simple at first sight, it enables complex description and querying of file-related, semantic metadata using Semantic Web standards like RDF, OWL and SPARQL. It further benefits from all theoretical and technological advances in the area of Semantic Web research. This includes, for example, not only the possibility for the exact specification of metadata semantics using ontologies but also methods to deal with the evolution of these ontologies or to apply reasoning software such as Pellet or FaCT++ to file-related metadata.

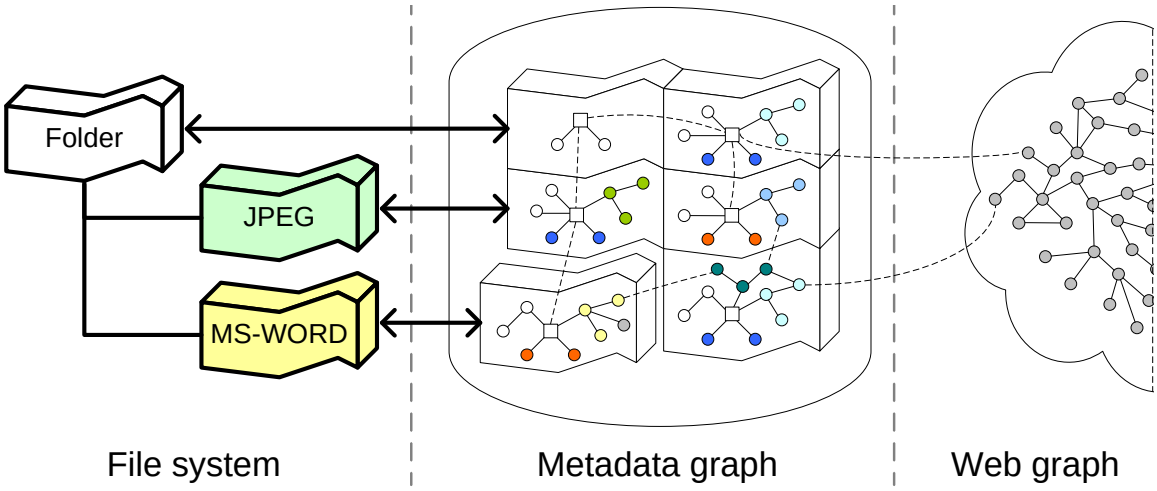


Figure 4.10: Continuous integration of individual file metadata graphs leads over time to one large metadata graph connecting semantically related resources in different file system and on the Web. RDF links between local files and remote Web resources are depicted as dashed lines.

4.6.1 Metadata generation

One question that remains is where the file-related metadata stored in such a store actually originate from. A general problem with custom, file-related metadata is that they are often not continuously updated by applications or users [MC03]. Many currently available metadata fields are left empty as it is either too cumbersome to manually enter/update this information or because the metadata are lost in file conversions or due to the loss of the connection between a file and it's externally stored metadata.

Besides proposing an approach for the manual annotation of local file system objects with hypertext presented in Chapter 9 we do not further contribute to the area of metadata generation in this work. We are, however, confident that the ongoing development of more advanced algorithms and applications for automatic, semi-automatic, manual and collaborative annotation of digital information will lead to a higher availability of file-related, semantic metadata in the future. Further, a whole variety of technologies developed for the (semantic) Web could become amenable if local resources were interlinked by the described metadata graph. Some exemplary advanced technologies for metadata generation and semantic annotation are:

Semantic wikis. Semantic wikis [OVBD06, Sch06, KVV+07] enrich traditional wikis with semantic annotations like tags and categories, attributes, and typed relationships. They comprise well-suited tools for collaborative knowledge capture, knowledge management, and learning. Semantic wikis could be used for the manual annotation of file resources, both in collaborative but also in personal knowledge management scenarios. The author has contributed to the area of semantic wikis [PSA+06] and experiences of using a semantic wiki in various real-world projects influenced design and implementation of the work presented in this thesis [PSK08]. A prototypical, wiki-like approach for local file annotation is presented in Chapter 9.

I/O and user activity tracing. In [GS08], the authors present *SeeTrieve*, a personal document retrieval and classification system that tends to improve desktop search accuracy by incorporating context information. Their tool traces low-level *read* and *write* operations as well as user GUI activities. SeeTrieve captures visible text snippets from the applications a user currently interacts with and stores relations between these snippets and related files in a context graph. The authors showed that search recall can be significantly improved by exploiting the information stored in this graph. This and comparable approaches could be used for the automatic creation of semantic relationships between files.

Data and link mining. Data mining techniques that put a special focus on links between considered data items are called *link mining techniques*. As files stored in local file systems are currently unrelated, these techniques including link analysis or graph mining are less applicable in this domain. This, however, changes when the described metadata graph would be in place. A survey on link mining methods can be found in [GD05], the authors explicitly name a combination of information extraction and link mining for building the Semantic Web. Other approached from the Semantic Web domain for automatic link discovery based on explicit similarity metrics are described, e.g., in [RSS08, VBGK09b].

Named entity recognition. Named entity recognition (NER) allows the extraction of information units of interest (e.g., names, numeric expressions, dates) from unstructured textual information as also found in local files. Such named entities could be exploited, for example, for automatic classification tasks of local text documents or for automatic interlinking of these resources. A comprehensive survey on named entity recognition can be found in [NS07].

Human computation. Human computation is a semi-automatic way to solve a task with the help of human intelligence. One well-known example is the collaborative labeling of images by players of the ESP game introduced by von Ahn and Dabbish in 2004 [vAD04]. In this game, two online users are randomly paired and are shown the same set of images. Independently (i.e., without further communication) they enter labels for these images and when these labels match, the users are awarded with

points. In the case of such an agreement, the labels both users agreed on are used for annotating the image[18]. Human computation has been used for various other tasks, such as the annotation of sounds, the collection of geospatial data, the classification of galaxies spotted by the Hubble Space Telescope of for weaving the Semantic Web [YCK09, SH08]. Human computation algorithms could also be used for the collaborative annotation of shared file resources.

Social Tagging. Social tagging was successfully introduced by Web 2.0 applications. In social tagging, a user community attaches freely definable keywords (tags) to some domain content. The set of created tags constitutes a so-called *folksonomy*. Social tagging constitutes a collaborative content annotation technique that might also be applicable to the annotation of local file system objects in small working groups.

Providing a uniform metadata access interface would make the development of such metadata generating tools arguably easier as it reduces technical complexity. Further, our model enables these applications to exploit explicit relationships between local file system objects and remote resources, something that currently cannot be done.

4.7 Conclusions

In [BLK08], the authors name five requirements for technologies that enable "the development of a global community of distributed but interconnected and interoperable information systems": (i) global unique identification, (ii) free mixing of terms from different ontologies, (iii) extensibility (add new terms/ontologies), (iv) possibility to drop portions of ontologies/data without affecting the meaning of the remaining concepts, and (v) mapping support between ontologies. These requirements are fulfilled by common Semantic Web technologies such as RDF, RDFS and OWL as described by the authors.

A file metadata store implementing our model, as sketched in this chapter, supports CRUD operations on file metadata records. By this, arbitrary file system objects can be annotated with multi-layered metadata descriptions expressed in the flexible RDF data model, semantic vocabularies can be used to describe the contained data unambiguously. All these data can be arbitrarily interlinked which includes links to external resources. We therefore believe that our proposed data model provides a way for integrating local file system objects stably into a Web of Data and by that into such a global information system.

[18] This collaborative labeling algorithm has been adopted by Google for labeling Web images, see http://images.google.com/imagelabeler/

The advantages of our proposed metadata model can be summarized as follows:

i) Arbitrary metadata can be assigned to any file or folder stored in the local file system. There are no restrictions in the semantics of these metadata (as defined by the respective vocabularies) or in their number. Vocabularies of varying abstractness may be used to describe files from many, possibly orthogonal viewpoints. These metadata enable sophisticated querying and integration of these data in turn.

ii) Explicit compartmentalization of metadata descriptions may be used to model and access partial aspects of the described resources which is useful, e.g., for grouping logically associated data or for modeling provenance information.

iii) The model breaks the current de-facto 1:1 relationship between files and metadata records. It allows the semantic annotation and interlinking of arbitrary sets of file system objects as well as (semantic) Web resources.

iv) As data and metadata are handled independently, access control is also done individually (but may as well be coupled automatically). Further, they can be accessed and manipulated concurrently which may allow fast parallel access to these data.

v) This independence disburdens applications that want to access the metadata of a file from the need to understand the file format which makes file metadata accessible to a much larger set of applications when compared with the current situation on the desktop.

vi) Metadata graphs can be merged and integrated with other metadata graphs. These graphs would be amenable to a large set of Semantic Web tools that enable sophisticated querying, reasoning and data integration tasks.

In a summary, we claim that our proposed metadata model would open the closed "data silos" of current desktop systems, enable the integration of file data into a global Web of Data and add a new way to share data *and* metadata among users (cf. [PS10]).

It is noteworthy that we do not envision one single metadata store per desktop as the only possible deployment possibility. As we use the RDF graph data model for data representation, metadata descriptions stored in different stores can be merged easily if they share common URIs. Note that the minimal store defined by our model is a simple RDF graph containing triples that realize the File/URI mapping. It is further important to emphasize that our proposed method for storing file-related metadata is fully backward-compatible as it does not require to radically change the current practice of storing metadata in file headers. One or many metadata stores can be used to augment this existing file-related metadata with additional external, semantic metadata descriptions.

The implementation of such a system is far from trivial, both from a theoretical and a practical point of view. Naturally, performance, concurrency and security demands to such a metadata store would be high which comprises a major technical challenge on its own. Further, a central issue with this model is that it crucially depends on a stable association of a file system object and its metadata record, i.e., on an accurate File/URI mapping as described in Definition 4.8. We describe possible solutions for preserving the link integrity between a local metadata record and a local or remote resource in the Sections 6 and 8. A prototypical implementation of a metadata store as described in this chapter is available at `http://purl.org/y2/`. Finally, a prototypical file-annotation application that also implements our proposed metadata model to a large part is described in Chapter 9.

Part III

Methods for preserving link integrity

The metadata model that we proposed in the previous part of this thesis crucially depends on stable links connecting local and remote resources. In this part we present two methods for preserving the integrity of such links between resources on the desktop and in the Web of Data (file system objects, metadata records, Linked Data resources). We start this part, however, with a detailed definition and discussion of the related *broken link problem* (Section 5) and explain what events taking place in local file systems or in remote Linked Data sources actually lead to broken links. In the following, we first present DSNotify (Section 6), a tool for preserving link integrity in Linked Data. We start with this historically older research on DSNotify as it strongly influenced our successive research on link preservation on the desktop. After that we present a study of low-level file system features (Section 7) that provided the fact basis for the design and the evaluation of Gorm (Section 8), an algorithm for preserving link integrity between local files and associated external metadata records.

Chapter 5

The broken link problem

5.1 Introduction

In the previous part we described several examples of links and references on the desktop and in Linked Data sources. As in the *Web of Documents*, such links may break when files or Linked Data resources are removed, moved, or updated. We already discussed some of the consequences of such broken links in the introductory examples in Section 3.3.1. Broken links are a considerable problem as they interrupt navigational paths in a network leading to the practical unavailability of information [ICL96, Ash00, LPF+01, MNI+09, SH09, VdSNB+10].

Broken links occur on the document Web, the Web of Data and on the desktop, however, the set of available methods and strategies for avoiding or repairing them are very different in each domain. In order to repair broken links, data providers have to (i) detect these links and (ii) fix them. While the detection of broken links on the Web is already supported by a number of tools, only few approaches for automatically fixing them exist [MNI+09, PH10]. In the Web of Data, the current approach is to rely on HTTP status codes (e.g., *HTTP 404 Not Found*) and assume that data-consuming actors can deal with the inaccessibility of data [PH10]. The situation is even worse for the local file system: No notifications whatsoever are sent to applications that need to detect and fix broken links in their data.

As we consider this as insufficient in many application scenarios, we propose our change detection tools DSNotify (Section 6) and Gorm (Section 8) for automatically fixing broken links in a data web respectively in local file systems. We do not contribute a method for fixing such broken links in the *Web of Documents* but do discuss the applicability of our approaches in this domain. In this chapter, however, we generally define the *broken link problem* and start by discussing what events actually lead to broken links. Clients that are aware of such events occurring in their considered data sources may undertake actions that cope with the problems resulting from *dataset dynamics*. Such actions include (i) repairing broken links (ii) re-indexing of resources or (iii) cache invalidation.

Note that the contents of this chapter were published to a large part already in [PH11], which is in turn an extended version of [PH10, PHR10, HP09].

5.2 Change events

Low-level change events that may occur in Linked Data sources or in the local file system include *create*, *remove* and *update* events. A fourth, more special type of event is the *move* event. Things can be moved only if they have some kind of *location* which is, for example, not *per se* the case in the RDF data model. In Linked Data, however, resources are identified by HTTP URIs and their representations are accessible at

these URIs. Therefore, we can state that the definition of resource movement is feasible in Linked Data as it relies on (location-bound) dereferenceable HTTP-URIs. Local file system objects obviously have a *location* defined by their path in the file system hierarchy.

In the following, we provide a formal definition of such low-level change events in the context of Linked Data and local file systems. The following preliminary definitions are to be considered throughout this chapter:

Definition 5.1 (Preliminary Definitions) *Let $\mathcal{R}$ and $\mathcal{D}$ be the set of resources and resource representations. Representations of Linked Data resources are, for example, RDF or HTML documents. Representations of file system resources are the file data themselves. Further let $\mathcal{P}(\mathcal{A})$ be the powerset of an arbitrary set $\mathcal{A}$.*

Now let $\delta_t : \mathcal{R} \longrightarrow \mathcal{P}(\mathcal{D})$, be a dereferencing function returning the set of representations of a given resource at a given time t.

Let $\mathcal{E}$ be the set of all events and $e \in \mathcal{E}$ be a quadruple $e = (r_1, r_2, \tau, t)$, where $r_1 \in \mathcal{R}$ and $r_2 \in \mathcal{R} \cup \{\emptyset\}$ are resources affected by the event, $\tau \in \{created, removed, updated, moved\}$ is the type of the event and t is the time when the event took place. Further let $\mathcal{L} \subseteq \mathcal{E}$ be a set of detected events.

Using these preliminary definitions we can assert creation, remove and update events as follows:

Definition 5.2 (Basic Events) *$\forall r \in \mathcal{R}$:*

$$\delta_{t-\Delta}(r) = \emptyset \wedge \delta_t(r) \neq \emptyset \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, created, t)\} .$$

$$\delta_{t-\Delta}(r) \neq \emptyset \wedge \delta_t(r) = \emptyset \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, removed, t)\} .$$

$$\delta_{t-\Delta}(r) \neq \delta_t(r) \Longrightarrow \mathcal{L} \longleftarrow \mathcal{L} \cup \{(r, \emptyset, updated, t)\} .$$

Where Δ is a time parameter that may become indefinitely small. Practically, this parameter is often related to the sampling frequency of an event detection tool.

Further, we define move events based on a *weak equality* relation between resource representations:

Definition 5.3 (Move Event)
Let $\sigma : \mathcal{P}(\mathcal{D}) \times \mathcal{P}(\mathcal{D}) \longrightarrow [0, 1]$ be some general similarity function between two sets of resource representations. Further let $\Theta \in [0, 1]$ be a threshold value. We now define the set of removed resources at time t as $\mathcal{R}^t_{old} \subseteq \mathcal{R} \,|\, \forall r_{old} \in \mathcal{R}^t_{old} : \delta_t(r_{old}) = \emptyset \wedge \delta_{t-\Delta}(r_{old}) \neq \emptyset$. We further define the set of new resources as $\mathcal{R}^t_{new} \subseteq \mathcal{R} \,|\, \forall r_{new} \in \mathcal{R}^t_{new} : \delta_t(r_{new}) \neq \emptyset \wedge \delta_{t-\Delta}(r_{new}) = \emptyset$.

We can now define the maximum similarity of a removed resource $r_{old} \in \mathcal{R}^t_{old}$ with any new resource as $sim^{max}_t(r_{old}) \equiv max_i(\sigma(\delta_{t-\Delta}(r_{old}), \delta_t(r_i)))$ with $r_i \in \mathcal{R}^t_{new}$, i.e., as the maximum similarity between this resource's representations and any new resource's representations.

Now we can assert that:
$$\exists! r_{new} \in \mathcal{R}^t_{new} \text{ with } s = \sigma(\delta_{t-\Delta}(r_{old}), \delta_t(r_{new})) \,|\, s = sim^{max}_t(r_{old}) \wedge s > \Theta \Longrightarrow \mathcal{L} \leftarrow \mathcal{L} \cup \{(r_{new}, r_{old}, moved, t)\} .$$

Thus, we consider a resource as *moved* from one location to another when its resource representations were *removed* from their "old" location, and very similar[1] representations were *created* at the "new" location.

[1] Note that in the case that there are multiple possible move target resources with equal (maximum) similarity to the removed resource r_{old}, no event is issued ($\exists!$ should be read as "there exists exactly one").

5.3 Broken links

Changes that potentially result in broken links can be described using the above-mentioned event notations. In this section we want to contribute a formal definition of broken links that can be applied to links in Linked Data sources and to links between local file system resources. We generally distinguish two types of broken links that differ in their characteristics and in the way how they can be detected and fixed: *structurally* and *semantically* broken links.

5.3.1 Structurally broken links

Formally, we define structurally broken (binary) links as follows:

Definition 5.4 (Broken Link) *We define a (binary) link as a pair* $l = (r_{source}, r_{target}) \in (\mathcal{R} \times \mathcal{R})$. *Such a link is called* ***structurally broken*** *at a given time* t *if* $\delta_{t-\Delta}(r_{target}) \neq \emptyset \wedge \delta_t(r_{target}) = \emptyset$.

That is, a link between Linked Data resources or between file system objects is considered *structurally broken* if its target resource had representations that are not retrievable anymore[2]. In the remainder of this thesis, we will refer to structurally broken links simply as "broken links" if not stated otherwise.

5.3.2 Semantically broken links

In a quantitative analysis of Wikipedia articles that changed their meaning over time, the authors found out that a small number of articles (6 out of a test set of 100 articles) underwent minor or major changes in their meaning [HSB07]. Although we do not think that these results are generalizable for arbitrary data sources, they do demonstrate the problem of semantic drift in Linked Data. Such a semantic drift is problematic when articles are, for example, used for the classification of other resources (e.g., by `rdf:type` triples).

Consequently, we consider a link also as broken if the human interpretation (the meaning) of the representations of its target resource differs from the one intended by the link author. We call such a link *semantically broken*. In contrast to structurally broken links, semantically broken links are much harder to detect and fix by machine actors. But they are fixable by human actors that may, in a semi-automatic process, report such events to a system that then forwards these events to subscribed actors.

5.3.3 Informal definitions

The following informal definitions summarize the formal definitions of events and broken links in this section:

- Resource **creation events** occur if some representation(s) for a resource become dereferenceable while there was no representation available in the past. This includes a Linked Data resource becoming available at a particular HTTP URI or a file becoming available at a particular path in the file system.
- Resource **deletion events** occur if no representation for a resource is dereferenceable anymore while some were available in the past.
- Resource **update events** occur if some representation(s) of a resource have changed.
- Resource **move events** occur if *very similar* representation(s) of a resource that were removed are published in a timely close fashion at a different location.
- Links are **structurally broken** when their targeted resource had some representation(s) in the past that are not dereferenceable anymore.

[2] Note that our definitions do not consider a link as broken if only *some* of the representations of the target resource are not retrievable anymore. Further, note that our definitions currently do not include blank nodes. As the utility of blank nodes in the context of Linked Data is discouraged [BCH08] we decided to leave this issue as a topic for further research.

Solution Strategy	Class	Broken link type A	B	C
Ignoring the Problem	-	○	○	○
Embedded Links	p	●	○	○
Relative References	p	●	◐	○
Indirection	p	●	●	◐
Versioned and Static Collections	p	●	●	●
Regular Updates	p	●	●	●
Redundancy	p	●	●	●
Dynamic Links	a	●	●	●
Notification	c	●	●	●
Detect and Correct	c	●	●	●
Manual Edit/Search	c	●	●	●

Table 5.1: Solution strategies for the broken link problem. The strategies are classified according to Ashman [Ash00]: *preventative methods* (p) that try to avoid broken links in the first place, *adaptive methods* (a) that create links dynamically thereby avoiding broken links and *corrective methods* (c) that try to fix broken links. Symbols: potentially fixes/avoids such broken links (●), does not fix/avoid broken links (○), partly fixes/avoids broken links (◐)

5.4 Solution strategies

Methods to preserve link integrity have a long history in hypertext research. We have analyzed existing approaches in detail, building to a great part on Ashman's work [Ash00] and extending it. In her comprehensive survey of methods to preserve reference integrity in hypertext systems, Ashman defines these terms as follows:

> "Referential integrity and link integrity are much the same, except that referential integrity is a measure of the reliability of a reference to an end point, whether a source or a destination, whereas link integrity measures the reliability of the whole link (i.e., both end points), plus any meta-data that may be associated with the link itself." [Ash00]

In consequence of our definitions for structurally and semantically broken links, we adapt this definition as follows:

Definition 5.5 (Link Integrity) *Link integrity is a qualitative measure for the reliability that a link leads to the representations of a resource that were intended by the author of the link.*

Further, we categorize broken links by the events that caused their breaking:

- Type **A**: links broken because source resources were moved[3].
- Type **B**: links broken because target resources were moved and
- Type **C**: links broken because source or target resources were removed.

The analyzed solution strategies are summarized in Table 5.1 and discussed in the following:

[3]Note that in our definitions we did not consider links of type *A* as we assumed an *embedded link* model as described in [Dav98].

Ignoring the problem. It is the *status quo* to simply ignore the problem of broken links on the desktop and in the Web of Data and shift it to higher-level applications that process the data, although this can hardly be called a "solution strategy".

Embedded links. The *embedded link model* [Dav98] is the most common way how links are modeled on the desktop and in the Web. As in HTML, the link is embedded in the source representation and the target resource is referenced using, e.g., an HTTP URI or a file URI reference. This method preserves link integrity when the source resource of a link is moved thus avoiding broken links of type *A*.

Relative references. Relative references prevent broken links in some cases (e.g., when a whole resource collection is moved). But neither does this method avoid broken links due to removed resources (type *C*), nor does it hinder links with external sources/targets (i.e., absolute references) from breaking.

Indirection[4]**.** Introducing a layer of indirection allows content providers to keep links to their Web resources up to date. Aliases refer to the location of a resource and special services translate between an alias and its referred location. Moving or removing a resource requires an update in these service's translation tables. *Uniform Resource Names* were proposed for such an indirection strategy, *Persistent URLs (PURLs)* and *Digital Object identifiers (DOIs)* are two well known examples [Arm01].

A special case of an indirection layer in Web applications can be created by so-called *permalinks*, a technique that originates from the domain of database-backed systems that publish contents on the Web (e.g., Content Management Systems, Blogs, etc.). It aims at solving the problem that dynamically generated content cannot be linked from other Web sites as it is not available under a specific URI. Web-based systems that support versioning, such as wikis, also use permalinks for versioning purposes, i.e., they include the version number in the URI. For permalinks, the above-mentioned translation step is performed by the content repository itself and not by a special (possibly central) service.

Another special case on the Web is the use of small ("gravestone") pages that reside at the locations of moved or removed resources and indicate what happened to the resource (e.g., by automatically redirecting the HTTP request to the new location).

The main disadvantage of the *indirection* strategy is that it depends on *notifications* (see below) for updating the translation tables. Furthermore it *"... requires the cooperation of link creators to refer to the alias instead of to the absolute address."* [Ash00]. Another disadvantage is that special "translation services" (PURL server, CMS, gravestone pages) are required that introduce additional latency when accessing resources (e.g., two HTTP requests instead of one). Nevertheless, indirection is an increasingly popular method on the Web. This strategy prevents broken links of type *A* and *B* and can also help with type *C* links: removed PURLs, for example, result in the *HTTP 410 Gone* status code, which allows an application to react accordingly (e.g., by removing all links to this resource).

Versioned and static collections. In this approach, no modifications/deletions of the considered resources are allowed. This prevents broken links of types *A*-*C* within a static collection (e.g., an archive); links with sources/targets outside this collection can still break. Examples are links between files on a read-only device, e.g., a CDROM. Further examples are HTML links into the *Internet Archive*[5]. Furthermore, semantically broken links may be prevented when, e.g., linking to a certain (unchangeable) revision of a Wikipedia article.

Regular updates. This approach is based on *predictable* updates to resource collections, so applications can easily update their links to the new (predictable) resource locations, avoiding broken links of types *A*-*C*).

[4] The "Dereferencing (Aliasing) of End Points" and "Forwarding Mechanisms and Gravestones" categories from [Ash00] are combined in this category.

[5] For example, the URI `http://replay.waybackmachine.org/19970126045828/http://www.archive.org/` references the 1997 version of the Internet archive main page.

Redundancy. Redundant copies of resource representations are kept and a service forwards referrers to one of these copies as long as at least one copy still exists. This approach is related to the versioning and indirection approaches. However, such services can reasonably be applied only to highly available, unchangeable data (e.g., collections of scientific documents). This method may prevent broken links of types *A*-*C*, examples include LOCKSS [RR00] or RepWeb [VF03].

Dynamic links. In this method links are created dynamically when required and are not stored, avoiding broken links of types *A*-*C*. The links are created based on computations that may reflect the current state of the involved resources as well as other (external) parameters, i.e., such links may be context-dependent. However, the automatic generation of links is a non-trivial task and this solution strategy is not applicable to many real-world problems.

Notification. Here, clients are informed about the events that may lead to broken links and all required information (e.g., new resource locations) is communicated to them so they can fix affected links. This strategy was, for example, used in the Hyper-G system [Kap95] where resource updates are propagated between *document servers* using a scalable, probabilistic algorithm ("*p-flood*"). It is also the strategy of some current approaches for solving the broken link problem for Linked Data sources as discussed in Section 6.7. This method resolves broken links caused by *A*-*C* but requires that the content provider can observe and communicate such events.

Detect and correct. The solutions for the broken link problem proposed in this work falls mainly into this category that Ashman describes in her work as: *"... the application using the link first checks the validity of the endpoint reference against the information, perhaps matching it with an expected value. If the validity test fails, an attempt to correct the link by relocating it may be made ..."* [Ash00]. As all other solutions in this category (Section 6.7), we propose heuristic methods to automatically fix broken links of the types *A*-*C*.

Manual edit/search. In this category we summarize manual strategies for fixing broken links or re-finding missing link targets. This "solution strategy" is currently arguably among the most popular ones on the Web and on the desktop. First, content providers may manually update links in their contents (perhaps assisted by automatic link checking software). Second, users may re-find the target resources of broken links, e.g., by exploiting search engines or by manual URI manipulations.

5.4.1 Discussion

Each of the discussed solution strategies for fixing or avoiding broken links has its advantages and disadvantages as briefly discussed. Furthermore, they differ in their applicability to real-world problems in general and in the contexts of Linked Data or local file systems in particular. For example, redundant collections are not very likely to be the strategy of choice for Linked Data, an indirection approach is probably unfeasible in a local environment where performance plays a major role.

We believe that the *detect and correct* approach is the preferable strategy in dynamic environments where broken links cannot be avoided due to missing mechanisms for link integrity preservation. Such a strategy is based on two fundamental stages: broken links have to be (i) detected and (ii) fixed.

One way to detect broken links is to listen for the described low-level change events that actually caused the links to break. This can be done, for example, by receiving the change events from the system that actually organizes the data. However, such notifications are not or only partially available in the environments discussed in this thesis. If no such event notification is available, as it is currently the case in the Web of Data, the data source can be periodically monitored as discussed in the next chapter. In the file system, the problem is that not all required event types are reported by current tools that notify listeners about changes. The detection of these missing event types can, however, be done by heuristic methods as will be shown in the following.

In the remainder of this part we discuss two algorithms (and two tools that actually implement them) that realize such a *detect and correct* strategy for preserving link integrity in the Web of Data and in local file systems. As mentioned before, we start with our historically older research on event detection in Linked Data sources.

Chapter 6

DSNotify – fixing broken links in a Web of Data

In this chapter, we present DSNotify, an event-detection framework that informs actors about various types of change events in Linked Data sources and allows them to maintain the integrity of links to resources in such distributed data sets. Later in this thesis we will describe how DSNotify can be used to preserve the integrity of links from local to remote Linked Data resources.

Note that the contents of this chapter were published to a large part already in [PH11], which is in turn an extended version of [PH10, PHR10, HP09]. Section 6.8.1 was published partly already in [SP10] and in [PS10]

6.1 Introduction

The early phase of Linked Data led to a large number of available data sets from various domains that form the so-called *Web of Data* [BHBL09]. With the increasing usage of these data by applications, questions about data quality and robustness of the supporting infrastructure get more and more weight in current research. One property that may affect data quality in the Web of Data is the dynamics of its data sets: Linked Data sets evolve over time and applications that make use of these data need to be aware of changes in order to update their local data dependencies. Not doing so may result in several issues such as broken links, invalid indices, or outdated data in client and server-side caches.

Research related to these matters has recently emerged under the term *dataset dynamics* [UHH+10]. It investigates how to deal with changes in various types of data sources (e.g., sensor data, archival data) and how to handle them at different levels of granularity (triple, resource, or graph-level). The goal is to develop strategies and build sophisticated yet simple solutions that address the mentioned problems at various levels, namely:

i) Vocabularies for describing the properties of a dataset with respect to its dynamics.

ii) Vocabularies for representing change information.

iii) Protocols for change propagation.

iv) Applications and algorithms for change detection.

Our work in dataset dynamics is motivated by the specific use case of *link maintenance* as discussed in the previous chapter. It derives from the requirement that applications that have linked their local datasets

with resources in remote datasets (e.g., DBpedia) need to be informed when remote resources and their representations change or disappear.

We built a tool called DSNotify, which is a general-purpose change detection framework that can be applied to Linked Data sources in order to inform data-consuming actors about various types of events (create, remove, move, update) that may occur in these data sources. By this, these actors are enabled to fix broken links in their local data, thus preserving link integrity, which is one aspect of data quality. DSNotify is open source software and publicly available at http://dsnotify.org.

In this chapter we describe the already formally introduced *broken link problem* in the context of the ongoing dataset dynamics research. We identify events that lead to broken links in the Web of Data in particular and present our method to deal with them. With our DSNotify Eventset Vocabulary we contribute a vocabulary for representing changes in dynamic data sets. We then describe the technical details of DSNotify, its core components and algorithms as well as its evaluation including the reusable infrastructure we used therefore. Finally, we discuss our work in the context of related work from various research areas, including hypertext research, database research and semantic Web research and conclude with a comprehensive discussion of our contributions in the context of dynamic Linked Data sources.

6.2 Dataset dynamics

Linked datasets change in the course of time: resource representations and links between resources are created, updated and removed. The frequency and dimension of such changes depends on the nature of a linked data source. Sensor data are likely to change more frequently than archival data. Updates on individual resources cause minor changes when compared to a complete reorganization of a data source's infrastructure such as a change of the domain name. Anyway, in many scenarios linked data consuming applications need to deal with these kinds of changes in order to keep their local data dependencies consistent. *Dataset dynamics* denotes a research activity that currently investigates how to deal with that problem.

6.2.1 Use cases and requirements

As part of our work in the Dataset Dynamics interest group[1] we identified three representative use cases in which applications need to be informed about changes in remote linked datasets.

- **UC1 Link Maintenance**: An application hosts resources that are linked with remote resources and uses remote data in its local application context. It needs to be informed when representations of these remote resources change or become unavailable under a given URI in order to keep these links valid.
- **UC2 Dataset Synchronization**: A dataset consumer wants to mirror or replicate (parts of) a linked dataset. The periodically running synchronization process needs to know which triples have changed at what time in order to perform efficient updates in the local dataset.
- **UC3 Data Caching**: An application that consumes data from one or more remote datasets uses a HTTP-level cache that stores local copies of remote data. These caches need to be invalidated when the remote data is changed.

These use cases require for a technical infrastructure comprising the following components:

i) A *dataset dynamics vocabulary* that can express meta-information about the dynamics of a data set (e.g., change frequency, dimension of changes, last update, etc.) and provide a link to the update notification source URI. A first draft of such a vocabulary is available at http://purl.org/NET/dady.

[1] http://groups.google.com/group/dataset-dynamics/

ii) A *change description vocabulary* to express the semantics of changes at different granularity levels. The DSNotify Eventset Vocabulary, which will be presented in Section 6.4 is one such vocabulary. Others are discussed in the related work section (Section 6.7).

iii) A *change notification protocol* that communicates changes from a remote linked dataset to a local client application.

iv) *Applications* for detecting changes. DSNotify, which we will discuss in the remainder of this article, is one example for such an application.

6.2.2 Change description vocabularies

Regarding the previously introduced use cases we need a change description vocabulary to express *what* data unit has changed, *how* it has changed, and in certain cases also *when* and *why* it has changed.

The description of *what* has changed depends on the use case: for dataset synchronization (UC2), changes need to be communicated on the triple-level. In such cases, we speak of a *triple-centric* perspective because the triple is the subject of change. If link maintenance (UC1) is the goal, it is sufficient to apply a *resource-centric* perspective and regard the resource as the subject of change. Umbrich et al. [UHH+10] also introduce the *entity-centric* perspective, which is a specialization of a *resource-centric* view. There is of course a connection between these perspectives: a change of a resource (entity) is a result of one or many triple changes.

The information *how* a certain data unit has changed is typically expressed by operations, *add* and *remove* being the most basic ones. From a set of atomic operations one can derive so-called *compound changes* [AH06] or even higher-level changes, such as the fact that ontology classes were merged or domains of properties changed (cf., [PFF+09]).

Timestamps or version numbers attached to change information cover the *when* aspect. This allows a client to reproduce the sequence of changes in a certain dataset over time, which is an important requirement for dataset synchronization (UC2).

The *why* aspect of a change allows dataset providers to express additional information about the nature of a change and allows clients to filter retrieved change information for certain criteria.

6.2.3 Change notification protocols

Several alternatives for propagating changes from a remote dataset to a client are currently being discussed. There is a consensus in the dataset dynamics working group that the applied protocol should be standards-based and widely supported by clients. The Atom Publishing Protocol [Gdh07] in combination with the Atom Syndication Format [NS05] is a possible candidate protocol. It provides a generic mechanism that allows clients to check for updates on Web resources, is widely implemented, and is the basis for existing publisher-subscriber protocols such as *pubsubhubbub*[2].

Alternative protocols that are currently considered are the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) [LdS02] and the Web of Data Link Maintenance Protocol [VBGK09a]. OAI-PMH supports selective harvesting (time and set based) and keeps track of deletions in the course of time, both features that are relevant for dealing with changes in linked datasets.

6.2.4 The dynamics of the DBpedia dataset

To illustrate that dataset dynamics is a real-world issue, we report on the changes we observed in DBpedia [ABK+07]. We observed how the instances of four OWL classes (*Person*, *Place*, *Organization*, *Work*)

[2] http://code.google.com/p/pubsubhubbub/

changed between two DBpedia snapshots (DBpedia 3.2 and 3.3). For this, we considered resource creations, removals and moves. For identifying moves, we exploited so-called DBpedia *redirect* links that link moved resources in DBpedia and are directly derived from Wikipedia redirection pages. These redirection pages are automatically created in Wikipedia when articles are renamed and usually forward users from the HTTP URI of the outdated wiki page to the new article location. Table 6.1 summarizes the results.

Class	Ins. 3.2	Ins. 3.3	MV	RM	CR
Person	213,016	244,621	2,841	20,561	49,325
Place	247,508	318,017	2,209	2,430	70,730
Organization	76,343	105,827	2,020	1,242	28,706
Work	189,725	213,231	4,097	6,558	25,967

Table 6.1: Changes in the numbers of instances between the two DBpedia releases 3.2 (October 2008) and 3.3 (May 2009). Ins. 3.2 and Ins. 3.3 denote the number of instances of a certain DBpedia class in the respective release data sets, *MV* the moved, *RM* the removed, and *CR* the number of created resources.

These data suggest that DBpedia has grown and was considerably reorganized within a period of about seven months. It must, however, be kept in mind that these changes do not necessarily originate in corresponding changes in the Wikipedia itself but can also result from modifications in the extraction process that is applied by DBpedia to extract resources from Wikipedia pages.

Nevertheless, our data indicate that a considerable number of resources either changed their types or were removed from DBpedia. An even higher number of class instances became newly available in the considered time interval. A less intuitive finding was that a noticeable number (about one magnitude smaller than the number of created and removed instances) of resources changed their URIs (i.e., were "moved") in this time period. Comparing "old" and "new" representations of moved resources revealed that most of these changes occurred because Wikipedia articles were renamed to better match the Wikipedia naming conventions[3]. In DBpedia, such *article rename* events result in the corresponding resources being re-published under a different HTTP URI. Although our observations cannot be generalized in any way, they give some idea why not all URIs in the Web of Data are as *cool*[4] (i.e., stable) as they should be.

When HTTP URIs of resources change, references to these resources must be updated in order to avoid broken links. In the following we focus on the previously described *Link Maintenance* Use Case (UC1): We describe the broken link problem in the Web of Data and discuss what kind of changes occurring in data sources may lead to broken links.

6.3 The broken link problem in the Web of Data

Reconsider the general broken link problem we discussed in Chapter 5. In this section we briefly discuss this problem again especially focusing on Linked Data.

In the example shown in Figure 6.1, an institution has linked a resource representing a band in their local data set with the corresponding resource in DBpedia in order to publish a combination of these data on its Web portal. At a certain point in time, the band changed its name and renamed the title of their Wikipedia entry from "Oliver_Black" to "Townline", with the result that the corresponding DBpedia resource "moved" from its previous URI to `http://dbpedia.org/resource/Townline`. Now, the institution that links to this DBpedia resource has to update its local data to avoid broken links.

[3] Wikipedia naming conventions can be found at `http://en.wikipedia.org/wiki/Wikipedia:Article_titles`, a list of currently requested article moves is available at `http://en.wikipedia.org/wiki/Wikipedia:RM`

[4] See `http://www.w3.org/Provider/Style/URI`

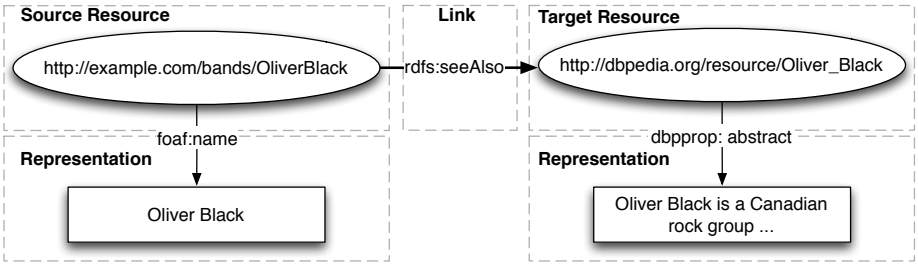


Figure 6.1: Sample link to a DBpedia resource.

Broken links in a machine-processable Web of Data are even worse than they are in the document Web: human users are able to find alternative paths to the information they are looking for. Such alternatives include directly manipulating the HTTP URI they have entered into a Web browser or using search engines to re-find the respective information. They may even decide that they do not need this information at this point in time. This is obviously much harder for machine actors (although the search-engine approach was proposed by some researchers as discussed in Section 6.7).

Linked data providers are usually able to preserve link integrity *within* their data source (i.e., with regard to links between resources in the same data source). It is considerably harder for them to avoid broken RDF links to remote data sources as they are normally not aware of changes that take place there. However, such RDF links between different data sources have a central role in the Linked Data approach in general and in the LOD project in particular.

In order to repair broken links, data providers have to (i) detect these links and (ii) fix them. The detection of broken links in the document Web is already supported by a number of tools but only few approaches for automatically fixing them exist (cf., [MNI+09, PH10]). The current approach in the Web of Data is basically to rely on *HTTP 404 Not Found* responses and assume that data-consuming actors can deal with inaccessibility of data.

As we consider this as insufficient in many application scenarios, we propose our change detection tool DSNotify that can be used to automatically fix broken links in a data web. Having defined events and broken links already in Section 5.2, we now first discuss our change description vocabulary for representing such change events and then present our tool that is capable of detecting *update*, *create*, *remove* and *move* events in Linked Data sources.

6.4 The DSNotify Eventset vocabulary

In order to describe sets of events we have developed an OWL-light vocabulary that conforms to the requirements presented in Section 6.2.2. This vocabulary allows the description of *what* Linked Data resources have changed *how*, *when* and *why*. Here we present an extended version of the vocabulary that was first introduced in [PHR10]. It is available at `http://dsnotify.org/vocab/eventset/`.

Our DSNotify Eventset Vocabulary, depicted in Figure 6.2, is built upon two vocabularies that are used for describing data sources and events: The Vocabulary of Interlinked Datasets (voiD) [ACHZ09], which is a vocabulary for describing `datasets` and `linksets` and the Linking Open Descriptions of Events (LODE) [STH09] vocabulary, which defines a core event model that was derived by considering certain relations from existing event models (such as CIDOC CRM, ABC or the Event Ontology) upon which a stable consensus has been reached.

Our vocabulary is centered around so-called `Eventsets` (themselves voiD datasets) that are sets of so-called `ResourceChangeEvents` (lode events). An eventset is a container of events (linked via the `hasEvent` property) that are related to two particular datasets: The `sourceDataset` is a particular voiD dataset that

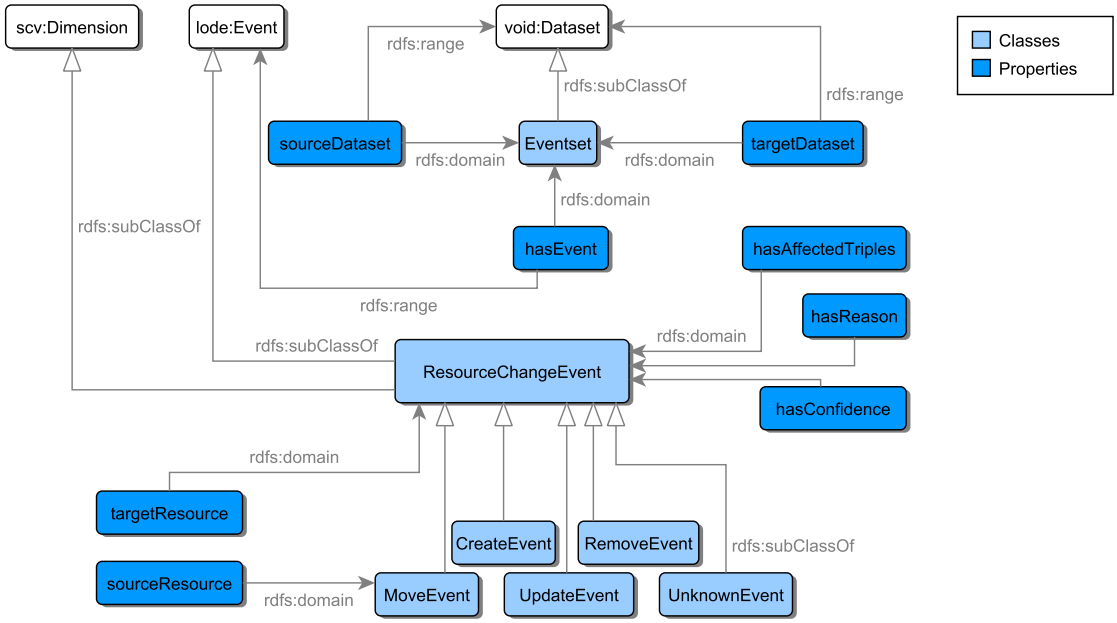


Figure 6.2: The DSNotify Eventset vocabulary.

was changed in the past, resulting in the `targetDataset`. The events that led to these changes are described by the eventset (an example for such datasets are snapshots of a particular linked dataset taken at different points in time).

Although any `lode:Event` can be part of an eventset, our vocabulary focuses on a particular subclass of such events called `ResourceChangeEvents`. These describe events that are directly associated with a particular resource (the `targetResource`) in the target respectively source dataset. We have sub-classed this class to model the basic events that may affect resources: creation (`CreateEvent`), update (`UpdateEvent`), and deletion (`RemoveEvent`). Furthermore, we defined a `MoveEvent` that describes that a particular resource in the `sourceDataset` (the `sourceResource`) was published under a different URI in the `targetDataset` (the `targetResource`). In order to capture events that are known to change resources but cannot be assigned to one of the previously listed event classes by the actor that creates an eventset, we have added a generic `UnknownEvent` class.

In order to further specify what triples were actually affected (i.e., removed or added) in the course of a `ResourceChangeEvent`, one may optionally link to a resource that specifies these triples (using the `hasAffectedTriples` property). This property intends to bridge the gap between a triple-centric view and our resource-based vocabulary (cf. Section 6.2.2). If necessary, `ResourceChangeEvents` can be directly linked to the created/modified instance data, for example, expressed using the Talis Changeset Vocabulary[5]. Further, we provide a `hasReason` property that can be used to link to resources that further specify the reasons for the respective event. For example, one could attach a textual description or link to another event here. We underspecified both, the `hasAffectedTriples` and the `hasReason` property on purpose as we expect new vocabularies for describing these event-facets to emerge in the near future due to currently ongoing research.

We further provide a simple data type property (`hasConfidence`) for capturing how confident an event detector is that a particular event actually took place; if omitted, applications should assume a default confidence value of 1.0 (i.e., 100% confident). As our method of event detection is based on normalized similarities

[5] http://vocab.org/changeset/schema.html

(as explained in Section 6.5), we use this value for this property to express how confident our algorithm is that a particular event took place.

For eventsets we have added the possibility to directly assert the number of contained events of a certain class using the `void:statItem` mechanism of the Statistical Core Vocabulary (SCOVO) [HHR+09]. We use ResourceChangeEvents as `scovo:dimensions` as depicted in Listing 6.1, an excerpt of an eventset that was used for the evaluation of DSNotify.

Listing 6.1: An excerpt of an eventset derived from DBpedia data. Namespace definitions are omitted for brevity; the complete eventset contained 8,380 events [PH10].

```
:created2490
  a     dsnotify:CreateEvent ;
  dsnotify:targetResource "http://dbpedia.org/resource/Heinrich_L%C3%BCtzenkirchen"
        ^^<http://www.w3.org/2001/XMLSchema#string> ;
  lode:atTime "2008-10-12T17:27:56" ^^<http://www.w3.org/2001/XMLSchema#dateTime> .

:moved43
  a       dsnotify:MoveEvent ;
  dsnotify:sourceResource "http://dbpedia.org/resource/Brian_Ashton_%28rugby_player%29"
         ^^<http://www.w3.org/2001/XMLSchema#string> ;
  dsnotify:targetResource "http://dbpedia.org/resource/Brian_Ashton_%28rugby_union%29"
         ^^<http://www.w3.org/2001/XMLSchema#string> ;
  lode:atTime "2008-10-25T09:28:51"
         ^^<http://www.w3.org/2001/XMLSchema#dateTime> .

:EDS1
 a       dsnotify:Eventset ;
 dsnotify:hasEvent :created2490, :moved43 ;
 dsnotify:targetDataset <http://dbpedia33.mminf.univie.ac.at> ;
 dsnotify:sourceDataset <http://dbpedia32.mminf.univie.ac.at> ;
 void:statItem
     [ a       scovo:Item ;
       rdf:value "179"^^<http://www.w3.org/2001/XMLSchema#int> ;
       scovo:dimension dsnotify:MoveEvent, void:numberOfResources
     ] ;
 void:statItem
     [ a       scovo:Item ;
       rdf:value "3810"^^<http://www.w3.org/2001/XMLSchema#int> ;
       scovo:dimension dsnotify:CreateEvent, void:numberOfResources
     ] .
```

We call an eventset *complete* if it contains all `ResourceChangeEvents` that, when executed in their timely order, result in the conversion of the `sourceDataset` into the `targetDataset`. We made use of complete eventsets in the evaluation of our tool, as described in Section 6.6. We have developed a Java API based on the Jena API for accessing our vocabulary. It allows the convenient creation and manipulation of eventsets and their contained events using Java objects that wrap respective RDF resources and properties in a Jena model.

6.5 Event detection with DSNotify

The DSNotify approach (cf. Figure 6.3) is based on an indexing infrastructure. A *monitor* periodically accesses data sources, creates an *item* for each resource it encounters and extracts a *feature vector* from this item's representations. The feature vector is stored together with the item's URI in an *index*. By comparing the feature vectors of currently monitored and already indexed items, DSNotify is able to detect create, remove and update events. The detection of resource move events is based on a heuristic comparison of indexed feature vectors of recently removed and recently added resource representations [PH10].

On a technical level, DSNotify comprises a generic data model that is shown in Figure 6.4. The depicted core components of DSNotify can be replaced easily with domain specific implementations using a plug-

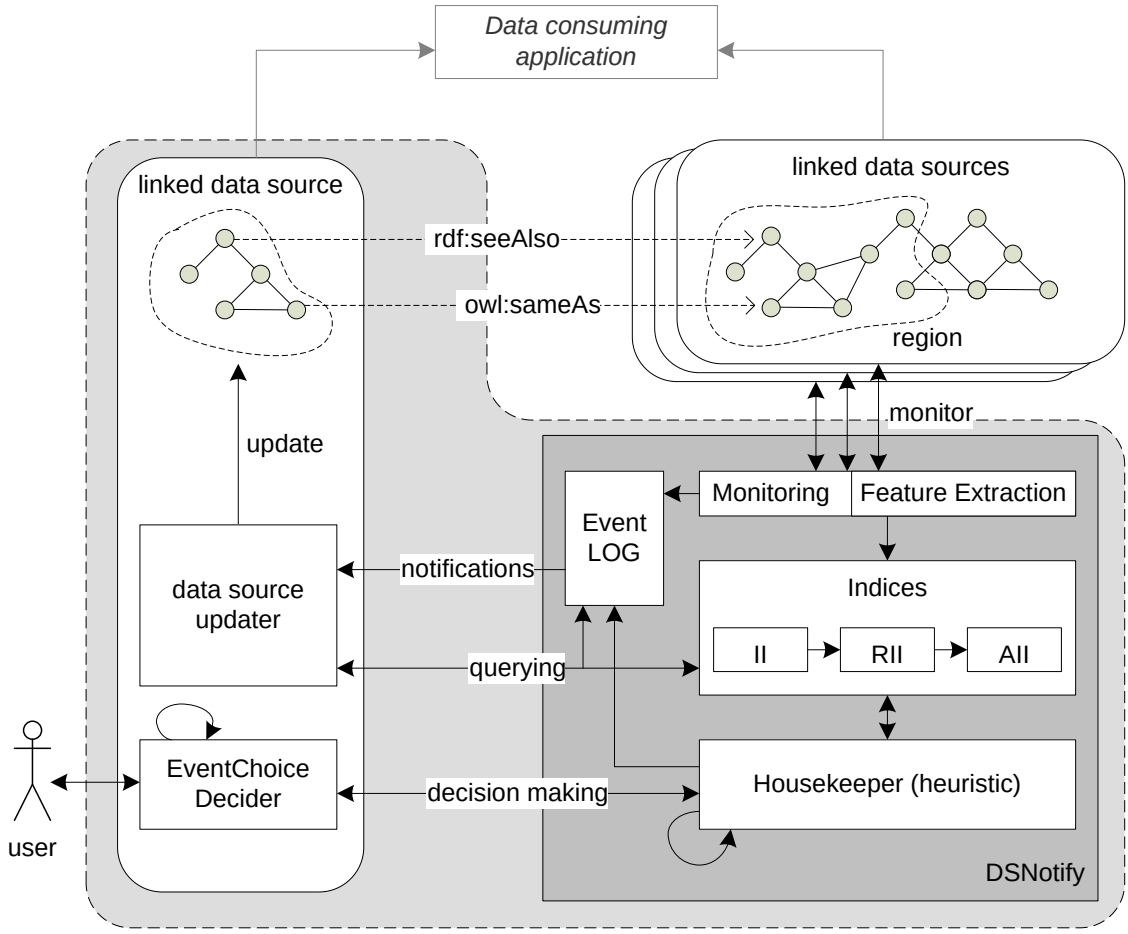


Figure 6.3: DSNotify architecture.

in concept. This enables, for example, the specialization of our tool for any *dataspace* [FHM05] that uses URIs as identifiers for its contained data items. A single DSNotify instance can detect changes in several dataspaces. The plug-in mechanism further allows to choose the exact composition of the used feature vectors or to replace the standard heuristic used for feature vector comparison.

6.5.1 Dataspace monitoring

The monitor component of DSNotify is responsible for the detection of created, removed and updated resource representations in the considered dataspaces. Monitors are usually periodically invoked by DSNotify and access so-called *observed regions* in the referenced dataspaces. A region corresponds to a subset of the data hosted in a dataspace. In the case of Linked Data sources, a region corresponds to a subgraph of the RDF graph hosted by that particular data source that is determined using arbitrary criteria: subgraphs may, for instance, be defined to contain only resources whose URIs match a certain syntactic pattern. A more general discussion of RDF subgraph selection was already provided in Section 4.2.1.

DSNotify currently contains two implementations of such regions for Linked Data sources: The first implementation is configured with an initial URI and a regular expression. When such a region is monitored, the monitor crawls the RDF graph starting from this URI and continues until there are no new URIs detected

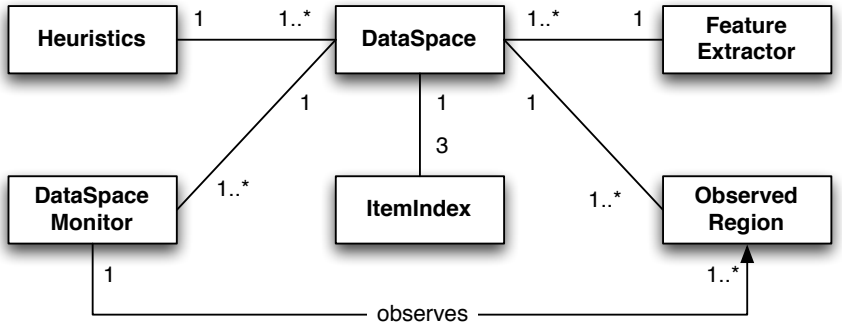


Figure 6.4: DSNotify generic data model.

that satisfy the regular expression. Our second region-implementation is configured by a SPARQL query and a SPARQL endpoint URI. Such a region is monitored simply by periodically querying the respective SPARQL service.

6.5.2 Feature vectors

Monitored resource representations are then converted to feature vectors. For this, a *feature extractor* component extracts configured property values from the considered representation and combines them to a feature vector (cf. Figure 6.5). These feature vectors are later used by a heuristic for the detection of moved resources.

Feature vectors may contain both, data type and object property values. Further, each feature in a feature vector may be individually weighted: for example, one would give features that have high entropy in the data higher weights to increase their influence in a vector comparison. Note that it is also possible to store features in a feature vector that are used by the heuristic for plausibility checks: we have, for instance, implemented a simple RDF hashing function that creates a fingerprint of all triples in the considered representation. The hash-value may be stored in a single feature of the feature vector and can be used by the heuristic to quickly decide whether a representation has changed (i.e., an update event occurred) or not. The set of extracted features, their implementation, and their extractors are configurable.

6.5.3 Indices and history

Feature vectors of monitored items are stored in three indices maintained by DSNotify:

- the *item index (II)* that contains the current state of the data source as known to DSNotify.
- the *removed item index (RII)* that contains the feature vectors of recently removed items
- the *archived item index (AII)* that contains the feature vectors of items that were removed a longer time ago.

These three indices are constantly updated by the monitor and the housekeeping components (see below) of DSNotify: items that are not found anymore are moved from the *II* to the *RII*. After a *timeout* period, they are moved from there to the *AII*.

DSNotify contains two alternative implementations for these indices: one is based on the Apache Lucene[6] framework and stores features in Lucene fields. The second implementation is a fast but non-persistent in-memory index implementation. Note that it is possible to mix both index implementations, e.g., by using the Lucene implementation for the *II* and the *AII* and a memory index for the *RII*.

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

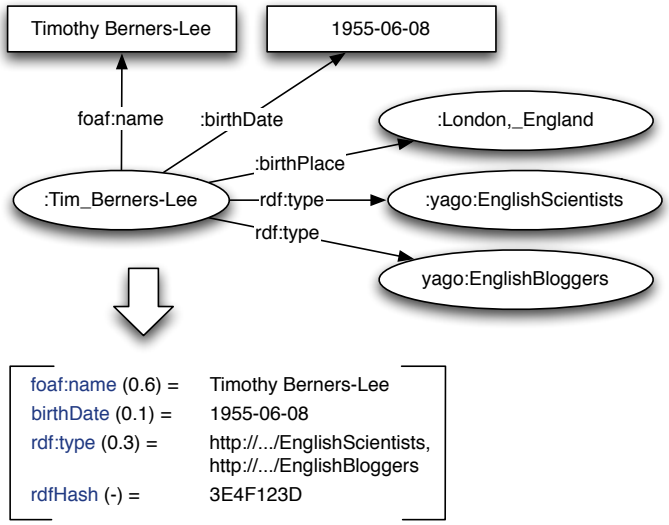


Figure 6.5: Deriving feature vectors from RDF representations. Note that features are weighted (weight given in parentheses) and that only a subset of the properties contained in a representation may be used for constructing a feature vector: in this example the *birthPlace* property is disregarded for constructing the vector.

6.5.4 Event detection

Recently added items in the *II* and the items in the *RII* are periodically examined by a *housekeeper* component. This housekeeper calls a heuristic algorithm that tries to detect *move* events by comparing the feature vectors of recently added and recently removed items.

The pairwise comparison of feature vectors is based on similarity measures for the individual features. These similarity measures are implemented in Java classes that comply with a certain Java interface. Examples for already implemented functions are exact string matching or the Levenshtein distance for comparing character data. The similarities of the individual features are then combined for calculating a similarity between the two considered vectors. The implementation of this heuristic algorithm itself is also easily replaceable using the above-mentioned plug-in mechanism.

The calculated similarities for the feature vector pairs are compared with a *lower threshold* value to determine possible candidates for a move event. The pairs with similarities above this threshold are grouped by their target item (i.e., the item that was created recently) and in a second step their similarity to possible predecessors (i.e., source items) for a newly created item are compared against a second, *higher threshold*. If only one such source/target pair has a similarity above this threshold, DSNotify decides that the considered source item is an actual predecessor of the target item and issues a *move* event. Otherwise, DSNotify does not decide automatically whether one of the candidate source items is a predecessor of the considered target item and issues a so-called *EventChoice*. Such event choices are representations of decisions that have to be made outside of our system by external actors (e.g., human users) that can resort to additional data/knowledge to make that decision. After this move event detection step is complete, DSNotify issues *create* events for all new items that had no detected predecessor.

Remove events are also issued by the housekeeper thread: when the housekeeper is scheduled, it first moves all items that resided longer than a timeout period in the *RII* to the *AII*. For each moved item, a *remove* event is issued. This means that items that were detected as missing by the monitor are represented in a “transient deletion state” in the RII before being moved to the AII (which corresponds to a “permanent

deletion state"). The timeout value determines for how long successor items for such transiently removed items may be found. Note that any newly detected item may be a successor of such an item which allows to detect move events of items being moved between different observed regions or even items that were first moved out of any observed region and later back in (as long as this happens within the timeout period). A pseudocode representation of the central housekeeping algorithm of DSNotify is depicted in Algorithm 1.

Algorithm 1: DSNotify event detection algorithm.

```
Data: Item indices II, RII, AII
Result: List of detected events L
begin
    L ⟵ ∅; PMC ⟵ ∅;
    t ⟵ currentTime();
    foreach toi ∈ RII.getTimeoutItems() do
        L ⟵ L + {(toi, ∅, removed, t)};
        move toi to AII;
    end
    foreach ni ∈ II.getRecentItems() do
        pmc ⟵ ∅;
        foreach oi ∈ RII.getItems() do
            sim ⟵ calculateSimilarity(oi, ni);
            if sim > lowerThreshold then
                pmc ⟵ pmc + {(oi, ni, sim)};
            end
        end
        if pmc = ∅ then
            L ⟵ L + {(ni, ∅, created, t)};
        else
            PMC ⟵ PMC + {pmc};
        end
    end
    foreach pmc ∈ PMC do
        if pmc ≠ ∅ then
            (oi_max, ni_max, sim_max) ⟵ getElementWithMaxSim(pmc);
            if sim_max > upperThreshold then
                L ⟵ L + {(oi_max, ni_max, moved, t)};
                move oi_max to AII;
                link oi_max to ni_max;
                remove all elements from PMC where pmc.oi = oi_max;
            else
                Issue an eventChoice for pmc;
            end
        end
    end
    return L;
end
```

Update events are issued by the monitor thread that simply compares the feature vector retrieved from a representation that was accessed in the current monitoring cycle with the current feature vector stored in the *II* if available. The detection of update events can be turned off using a configuration property for performance reasons: When considering highly dynamic data sources, it might be sufficient to get informed about new or removed items. Note that as the detection of update events is based on the feature vector derived from

a resource's representation, only updates that result in different feature vectors can be detected. However, we don't regard this as a shortcoming of our method as it is possible to add a feature to the feature vector that is calculated using the previously mentioned RDF hashing function. Such a feature is calculated at monitoring time by a hashing function over the whole representation and therefore captures all modifications to this representation. It is further possible to extend our system by own, more sophisticated plug-in hashing functions.

Monitoring, housekeeping and indices

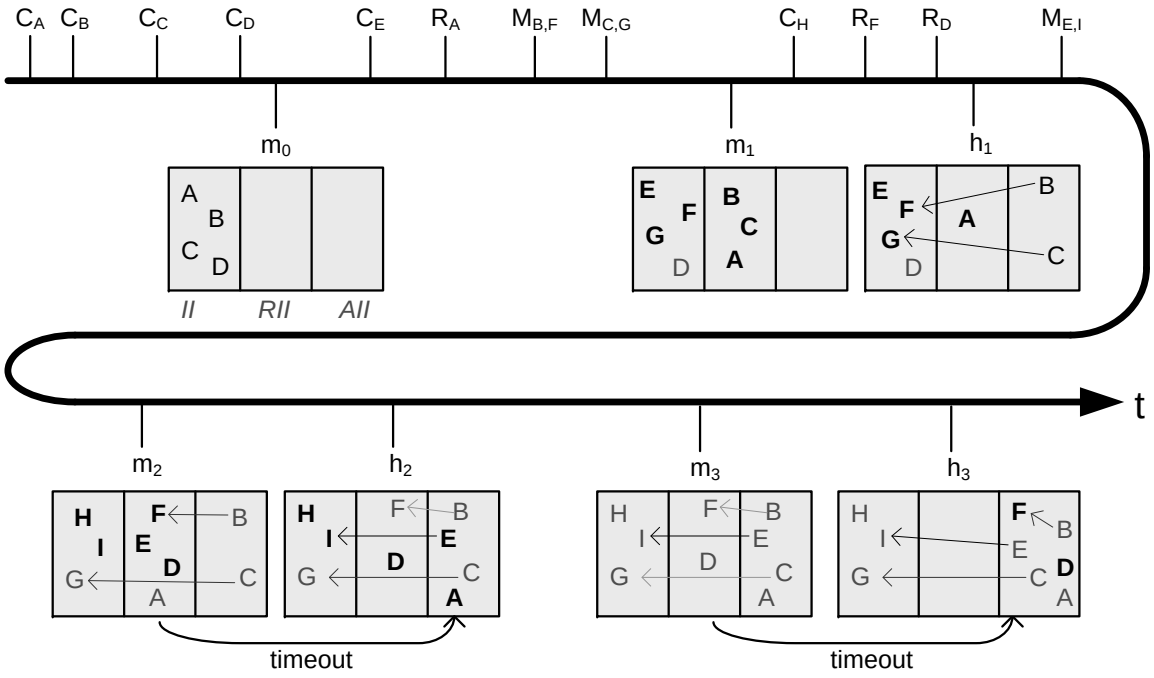


Figure 6.6: Example timeline illustrating the event detection mechanism of DSNotify. C_i, R_i and $M_{i,j}$ denote *create*, *remove* and *move events* of items i and j; m_x and h_x denote *monitoring* and *housekeeping* operations respectively. The current index content is shown in the gray boxes below the respective operation; the overall process is explained in the text.

The cooperation of monitor, housekeeper, and the indices is depicted in Figure 6.6. To simplify matters, we assume an empty dataset at the beginning. Then four items (A,B,C,D) are created before the initial DSNotify monitoring process starts at time m_0. The four items are detected and added to the item index (*II*)[7]. Then a new item E is created, item A is removed and the items B and C are "moved" to a new location becoming items F and G respectively. At time m_1 the three items that are not found anymore by the monitor are "moved" to the removed item index (*RII*) and the new items are added to the *II*. When the housekeeper is started for the first time at time h_1, it acts on the current indices and compares the recent new items (E,F,G) with the recently removed items (B,C,A). It does not include the "old" item D in its feature vector comparisons. The housekeeper detects B as a predecessor of F and C as a predecessor of G, moves B and C to the archived item index (*AII*) and links them to their successors. Between m_1 and m_2 a new item is created

[7] Actually their feature vectors are added, not the items themselves. But for reasons of readability we maintain this simplified notation throughout this section.

(H), two items (F,D) are removed and the item E is "moved" to item I. The monitor updates the indices accordingly at m_2 and the subsequent housekeeping operation at h_2 tries to find predecessors of the items H and I. But before this operation, the housekeeper recognizes that the retention period of item A in *RII* is longer than the *timeout* period and moves it to the *AII*. The housekeeper then detects E as a predecessor of I, moves it also to the *AII* and links it to I. Between m_2 and m_3 no events take place and the indices remain untouched by the monitor. At h_3 the housekeeper recognizes the timeout of the items F and D and moves them to the *AII* leaving an empty *RII*.

Time-interval-based blocking

The described algorithm is based on the efficient and accurate matching of pairs of feature vectors representing the same real-world item at different points in time. As in record linkage and related problems (cf. Section 6.7), the number of such pairs grows quadratically with the number of considered items resulting quickly in unacceptable computational effort. The reduction of the number of required comparisons is called *blocking* and various approaches, mostly from the record linkage domain, have been proposed in the past [Win06].

As described, our method needs to compare only the feature vectors derived from items that were recently removed or created, blocking out the vast majority of the items in our indices. We consider this method a *time-interval-based blocking* (TIBB) mechanism that efficiently reduces the number of feature vector pairs that need to be compared: If x is the number of feature vectors stored in our system, n is the number of new items and r is the number of recently removed items with $n + r \leq x$, then the number of comparisons in a single DSNotify housekeeping operation is $n \cdot r$ instead of x^2. It is intuitively clear that normally n and r are much smaller than x and therefore $n \cdot r \ll x^2$. The actual number of feature vector comparisons in a single housekeeper operation depends on the vitality of the monitored data source with respect to created, removed and moved items and on the frequency of housekeeping operations. We have analyzed and confirmed this behavior in the evaluation of our system (Section 6.6). As housekeeping and monitoring are separate operations in DSNotify, the number of feature vector pairs to be compared actually depends also on the monitoring frequency if lower than the housekeeping frequency. It determines the actuality of the *II* and the *RII* and thereby also the number of items stored in these indices. In our experiments we always chose equal monitoring and housekeeping frequencies.

6.5.5 Central data structures

DSNotify incrementally constructs three central data structures during its operation:

i) An event log containing all events detected by the system,

ii) A log containing all unresolved event choices and

iii) A linked structure of feature vectors constituting a history of the respective items.

These latter data are stored in the indices maintained by DSNotify. Note that we consider particularly this feature vector history as a very valuable data structure as it allows *ex post* analysis of the evolution of items with regard to their location in a data set and the values of the indexed features.

As these data structures may grow indefinitely, a strategy for pruning them from time to time is required. Currently we rely on simple timeouts for removing old data items from these structures but this method can still result in unacceptable memory consumption when monitoring highly dynamic data sources. More advanced strategies are under consideration.

6.5.6 Interfaces and implementation

DSNotify is implemented as a pure Java application that makes use of several open source libraries, the most central ones being *jena*, *apache lucene*, *quartz* and *jetty*. It can be used as a library or as a standalone application. Various interfaces are provided for accessing the data structures built by our tool:

i) DSNotify comprises a content-negotiated Linked Data interface that provides access to the above-mentioned data structures in HTML and RDF.

ii) It further starts a simple HTML interface. This interface provides human readable access to all DSNotify data including its configuration and allows querying of the indices. This interface can be disabled using a configuration flag.

iii) Applications that embed DSNotify as a Java library may subscribe directly to the event log to get actively informed about detected events. They may also subscribe to the log of event choices and may confirm or reject them. Further, methods for querying the indices are available.

iv) An XML-RPC interface for remote confirmation or rejection of event choices is also available.

6.5.7 Scalability

The scalability of our DSNotify approach is limited by three components:

i) The scalability of its indexing infrastructure

ii) The scalability of the housekeeping algorithm

iii) The scalability of the method how clients are informed about events detected by DSNotify.

Indexing. In terms of its indexing infrastructure, DSNotify is roughly comparable to common indexing architectures on the Web and in particular to other systems that build indices for semantic Web resources such as Swoogle [FDP+05], Sindice [TOD07] or SWSE [HHD+07a]. It is known that it is possible to build such services for very large numbers of RDF triples. The Lucene platform that we use for storing our feature vectors is known to scale up to several million index entries [HGM09]. However, as the DSNotify indexing approach stores also outdated feature vectors in its archived item index, this index may grow indefinitely and adequate strategies for pruning this index need to be implemented. Currently, we simply remove archived items after some timeout period.

Housekeeping. The housekeeping frequency of DSNotify as well as the dynamics of the considered data sources determine the number of feature vector pairs that have to be compared. Our evaluation, which we will present in the following section, shows that this number of feature vector comparisons as well as coverage and entropy of indexed features influence the accuracy of our method. A higher housekeeping frequency means less feature vector comparisons (when synchronized with the monitoring frequency) which means that the heuristic comparison algorithm is less likely to make a mistake. However, this increased accuracy of the method is paid with additional computational and I/O costs for the housekeeping operation.

Notification. Clients are informed about the events detected by DSNotify either by actively querying the service via HTTP requests, by notification via the XML-RPC interface (for remote notification) or directly via the Java interface. Remote notification may result in considerable network traffic (again, depending on the data source dynamics). DSNotify does not provide a sophisticated mechanism for solving this issue and we consider this as a weak point of our system. However, we are confident to be able to improve this in the future as mechanisms for the propagation of changes are a current research topic in the dataset dynamics domain.

6.5.8 Usage scenarios

Generally, we envision several scenarios for using DSNotify or a comparable tool that are described in the following:

1. As a **library** in an application that needs to be aware of changes in remote or local Linked Data sources.
2. As a **standalone service**: when a client notices that a resource's representations are not available at an expected URI, it could post the URI to this service which would automatically forward the HTTP request to the new URI of the resource as known by DSNotify. For human users, an automatically generated error page containing a link to the new URI of the resource could be returned by this service.
3. As an **indirection service**: As a special case of the above-mentioned scenario, it would be possible to build a PURL or DOI-like indirection service that automatically forwards requests to particular URIs to the current resource URIs as known by DSNotify. The advantage would be that data providers do not have to notify the service about changes in their data as DSNotify would detect them automatically.
4. As a **notification service** for a particular data source that reports changes in this data source to subscribers. This scenario requires further considerations regarding scalability which are a current subject of dataset dynamics research.

The DSNotify approach is applicable to event detection in various contexts, within and also outside the Linked Data domain. In the following we describe representative usage scenarios.

The BBC usage scenario

The BBC organizes parts of its information space according to the Linked Data principles. BBC Music (http://www.bbc.co.uk/music) assigns dereferenceable URIs to artists, bands, albums, etc. It further provides RDF descriptions for these resources, and links them with other related resources on the Web. Artists resources[8] are, for example, linked to DBpedia, Wikipedia, MySpace, the artists' blogs and fan pages, etc. A BBC Web page describing a particular artist is built by using the local resource representation of this artist as well as remote representations retrieved by dereferencing these links. Biographies of artists, for instance, are fetched from Wikipedia (DBpedia) and presented to the users of the BBC portal.

BBC could set up DSNotify as a notification service to monitor subsets of dependent data sources such as DBpedia person data and get informed about resource changes in order to keep their local data including the links to related resources up-to-date. Figure 6.7 illustrates the set-up for such a scenario.

Other scenarios

The DSNotify approach is applicable to event detection in various contexts, also outside the linked data domain. We have, for example, used DSNotify to enable stable URIs for files in a linked file system [SP10]. DSNotify is used for detecting change events in the file system and updating a mapping between (unstable) file URIs and (stable) UUID-based external URIs. DSNotify detects, e.g., when a file is moved on the local disk (and thus its file URI changes) and updates a mapping table accordingly. Low-level file metadata (name, creation date, etc.) was used in this case to build the feature vectors that represent local files. A more detailed discussion of this prototype is presented later in Section 6.8.1.

In principle, DSNotify is also applicable to the *Web of Documents*. However, the main issue here is the definition of proper *regions* (cf. Section 6.5.1) that are monitored by the system. While it seems common in Linked Data that data in a particular data source is linked with a defined set of other data sources (that can in turn be observed by DSNotify), this seems less common in the document Web.

[8] For example, http://www.bbc.co.uk/music/artists/084308bd-1654-436f-ba03-df6697104e19#artist.

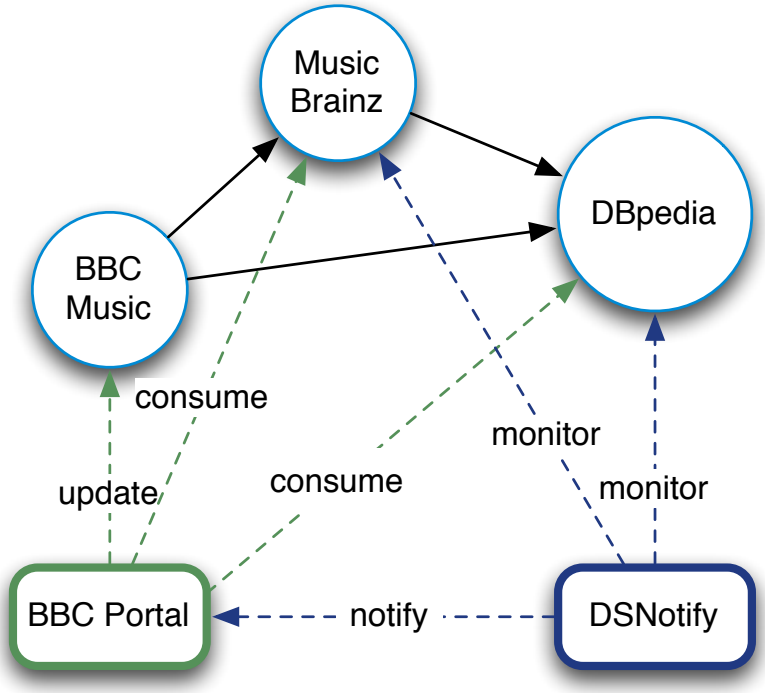


Figure 6.7: BBC usage scenario. The BBC Portal consumes data from DBpedia and MusicBrainz and exposes BBC music data as Linked Data. DSNotify is set up to monitor these dependent data sets and notifies the BBC Portal about changes. It can then react on these changes and update its local data, e.g., the RDF links pointing to remote resources.

Summary

In a summary, we can state that DSNotify is a flexible tool for detecting events in observed regions of specific dataspaces. Our tool detects create, remove, update and move events and can be used by applications, e.g., for fixing links to remote resources. Our move detection algorithm is based on heuristic feature vector comparisons and comprises a time-interval-based blocking approach for effectively reducing the number of required feature vector comparisons and increasing the detection accuracy. In the following section, we discuss the evaluation of this move event detection algorithm.

6.6 Evaluation

In the evaluation of our system we concentrated on two issues: first, we wanted to evaluate the system for its applicability for real-world Linked Data sources, and second, we wanted to analyze the influence of the housekeeping frequency on the overall effectiveness of our move event detection algorithm. However, first we had to develop an infrastructure for executing controlled experiments with our application and for measuring the results in order to evaluate it.

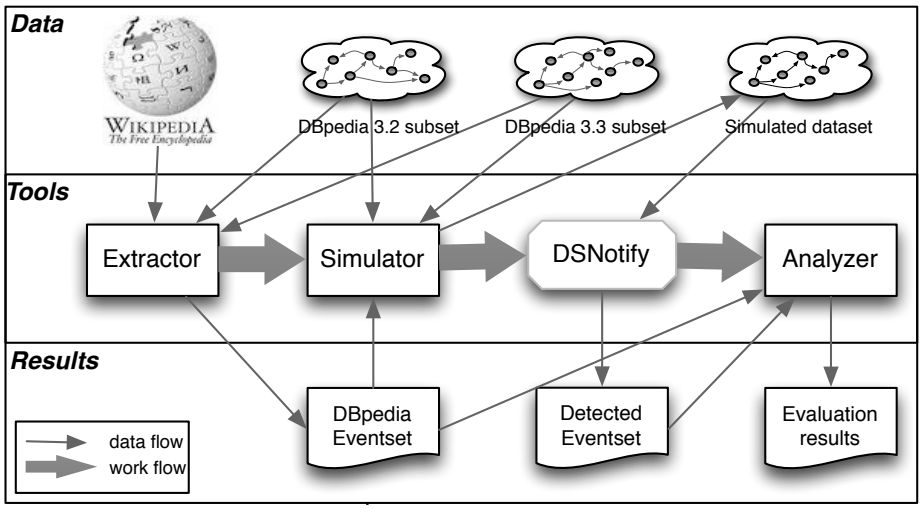


Figure 6.8: DSNotify evaluation infrastructure as used for the DBpedia Persondata Eventset evaluation.

6.6.1 Evaluation method

We have developed a set of extensible tools that can be combined to set up an evaluation infrastructure that allows the simulation of timely ordered change events in real-world datasets. This tool set consists of three components:

1. An **extractor** component that extracts a *complete* eventset from two versions of a data set
2. A **simulator** that re-plays the events in this eventset and applies the changes to the older of the two data set versions, gradually converting it into the newer version. These changes are observed by an event detecting application (in our case: DSNotify)
3. An **analyzer** component compares the eventset with the events that were detected by the application and calculates precision, recall and F_1-measure.

The co-operation of these core components of our evaluation infrastructure are depicted in Figure 6.8 and described in the following:

Extractor

The input data for our evaluation infrastructure consists of two versions of a particular data set (for example, two snapshots of the data source taken at different points in time). We call the "older" version the *source data set* and the "newer" version the *target data set*. An extractor then creates a *complete* eventset that was extracted from these data set versions and/or additional data. Remember that a complete eventset contains all events that, when executed in their timely order, convert the source- into the target data set. In some cases, an extractor needs to resort to versioning data or other external data in order to be able to extract a complete eventset.

The extracted *reference eventset* is the basis for the evaluation: in the end, event detecting applications are evaluated for how good they are at reconstructing this eventset without the external data/knowledge the eventset extractor could resort to. We did not develop a general tools set for the creation of versioned data sets, as such tools are available. Often creating such snapshots is as trivial as dumping an RDF model to a file. Some data set providers even provide snapshots of their data for download as it was the case for the DBpedia snapshots we used in our evaluation.

Simulator

The created reference eventset is the input for the simulator. The eventset contains links to the respective source and target data sets and can be configured to re-play the events contained in the eventset within a predefined time-span, which can be much shorter than the real-world event observation time. By doing this, the simulator continuously updates a *simulated dataset*. This simulated dataset is first initialized with the data from the source dataset and is then continuously updated by the simulator, ultimately becoming equal to the target data set (assuming a *complete* eventset).

By varying the time-span the simulation is allowed to run, one can vary the event frequency in the simulated data set which can be used to learn about the influence of this frequency on the effectiveness of an event detection mechanism.

Analyzer

The eventset detected by the evaluated application and the original reference eventset serve as input to an analyzer component that compares both eventsets and calculates the evaluation results in the form of a simple statistical analysis of precision, recall, and F_1-measure with respect to the detected events.

The recall for a particular event type is the number of correctly detected events of that type divided by the number of events of that type in the reference eventset. The precision is the number of correctly detected events of this type divided by the number of detected events of this type. The F_1-measure is the harmonic mean of precision and recall (i.e., the F-measure with $\beta = 1$).

Evaluation infrastructure implementation

The *extractor*, *simulator* and *analyzer* building blocks of our evaluation infrastructure correspond to respective abstract Java classes forming the foundation of our infrastructure: it comprises an abstract superclass of *eventset extractors* that provides some useful methods for eventset creation. Further, this class is able to create a histogram of the extracted eventset that can be used to analyze the distribution of events over the extraction period. We have implemented two concrete extractors for the data sets described below.

Our infrastructure further comprises two *simulator implementations*: a simple yet effective in-memory version and an OpenLink Virtuoso backed simulator for simulating larger eventsets on large RDF graphs (e.g., DBpedia snapshots). The simple simulator loads the snapshots in memory using the Jena API and creates a memory model for the simulated dataset. Then it re-plays the events in the eventset within a configured time interval and continuously updates the simulated dataset. The Virtuoso-backed simulator initializes the simulated dataset by creating a copy of all available data from the source dataset in Virtuoso. Then, as the simple simulator, it re-plays the events in the eventset.

Finally, an *analyzer implementation* is provided that loads a reference eventset and a detected eventset and calculates the precision, recall and F_1-measure as described above. It writes the results together with some statistical data to a file.

6.6.2 Evaluation data

We have evaluated our application with two types of eventsets extracted from existing datasets: the *iimb-eventsets* and the *dbpedia-eventset*[9]. For both types of data sets we first implemented specific extractors for deriving the reference eventsets. All experiments were carried out on a system using two Intel Xeon CPUs with 2.66 Ghz each and 8 GB of RAM. The used threshold values were 0.8 (*upperThreshold*) and 0.3 (*lowerThreshold*).

[9] All data sets are available at http://dsnotify.org/.

Name	$Cov.$	H	H_{norm}
tbox:cogito-Name	0.995	5.378	0.995
tbox:cogito-first_sentence	0.991	5.354	0.991
tbox:cogito-tag	**0.986**	**1.084**	**0.201**
tbox:cogito-domain	**0.982**	**3.129**	**0.579**
tbox:wikipedia-name	0.333	1.801	0.333
tbox:wikipedia-birthdate	0.225	1.217	0.225
tbox:wikipedia-location	0.185	0.992	0.184
tbox:wikipedia-birthplace	0.104	0.553	0.102

Namespace prefix tbox: <http://islab.dico.unimi.it/iimb/tbox.owl#>

Table 6.2: Coverage, entropy and normalized entropy of all properties in the *iimb* datasets with a *coverage* > 10%. The selected properties are written boldface in rows with gray background color. The rows are ordered by decreasing coverage.

The IIMB eventsets

The *iimb-eventsets* are derived from the ISLab Instance Matching Benchmark [FLMV08] which contains one (source) dataset containing 222 instances and 37 target datasets that vary in number and type of introduced modifications to the instance data. It is the goal of instance matching tools to match the resources in the source dataset with the resources in the respective target dataset by comparing their instance data. The benchmark contains an alignment file describing what resources correspond to each other that can be used to measure the effectiveness of such tools. We used this alignment information to derive 10 eventsets, corresponding to the first 10 iimb target datasets, each containing 222 *move* events. The first 10 iimb datasets introduce increasing numbers of value transformations like typographical errors to the instance data. We used random timestamps for the events (as this data is not available in this benchmark) that resulted in an equal distribution of events over the eventset duration.

We simulated these eventsets, monitored the changing dataset with DSNotify and measured precision and recall of the reported events with respect to the eventset information. For a useful feature selection we first calculated the entropy of the properties with a *coverage* > 10%, i.e., only properties were considered where at least 10% of the resources had instance values. The results are summarized in Table 6.2. As the goal of the evaluation was not to optimize the resulting precision/recall values but to analyze our blocking approach, we consequently chose the properties *tbox:cogito-tag* and *tbox:cogito-domain* for the evaluation because they have good coverage but comparatively small entropy in this dataset. We calculated the entropy as shown in Equation 6.1 and normalized it by dividing by $ln(n)$.

$$H(p) = -\sum_{i=1}^{n} p_i \, ln(p_i) \tag{6.1}$$

DSNotify was configured to compare these properties using the Levenshtein distance and both properties contributed equally (weight = 1.0) to the corresponding feature vector comparison. The simulation was configured to run for 60 seconds, thus the monitored datasets changed with an average rate of $\frac{222}{60} = 3.7$ events/s.

As stated before, the goal of this evaluation was to demonstrate the influence of the housekeeping frequency on the overall effectiveness of the system. For this, we repeated the experiment with varying housekeeping intervals of 1s, 3s, 10s, 20s, 30s (corresponding to an average rate 3.7, 11.1, 37.0, 74.0, 111.0 events/housekeeping cycle) and calculated the F_1-measure for each dataset (Figure 6.9).

Results. The results clearly demonstrate the expected decrease in accuracy when increasing the length of the housekeeping intervals, as this leads to more feature vector comparisons and therefore more possibilities

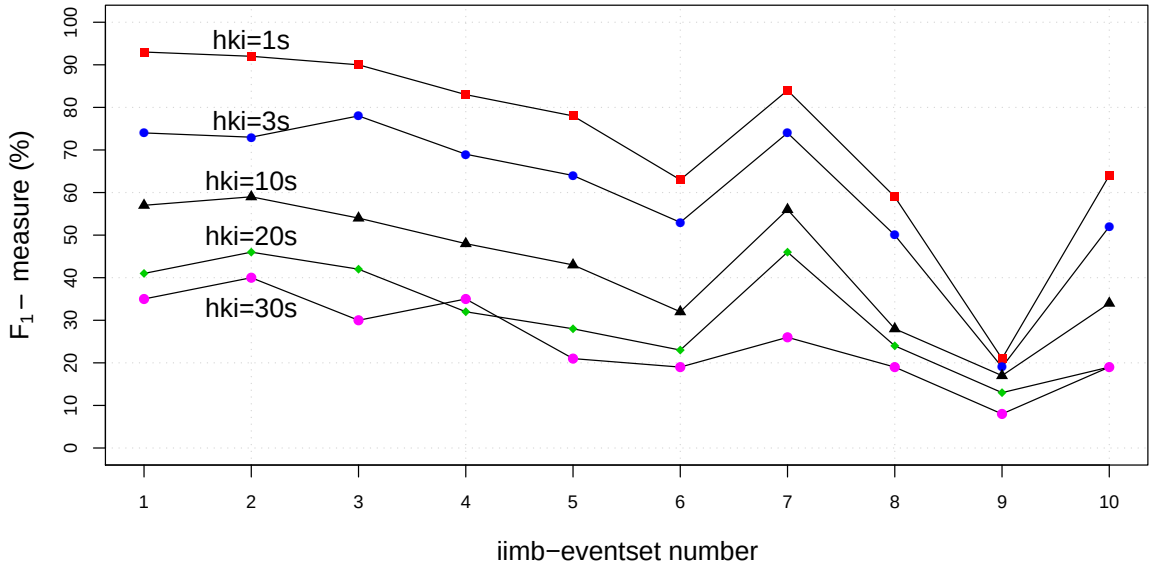


Figure 6.9: Influence of the housekeeping interval (hki) on the F_1-measure in the *iimb-eventset* evaluations.

to make wrong decisions. Furthermore, Figure 6.9 depicts the decreasing accuracy with the increasing dataset number. This is also expected as the benchmarks introduce more value transformations with higher dataset numbers, although there are two outliners for the datasets 7 and 10.

The DBpedia persondata eventset

In order to evaluate our approach with real-world data we have created a *dbpedia-eventset* that was derived from the person datasets of the DBpedia snapshots 3.2 and 3.3[10]. The raw persondata datasets contain 20,284 (version 3.2) and 29,498 (version 3.3) subjects typed as *foaf:Person* each having three properties *foaf:name*, *foaf:surname* and *foaf:givenname*. Naturally, these properties are very well suited to uniquely identify persons as also confirmed by their high entropy values (cf. Figure 6.10a). For the same reasons as already discussed for the *iimb* datasets an evaluation with only these properties would not clearly demonstrate our approach. Therefore we enriched both raw data sets with four properties (see the table in Figure 6.10a) from the respective DBpedia *Mapping-based Infobox Extraction* datasets [BLK+09] with smaller coverage and entropy values.

We derived the *dbpedia-eventset* by comparing both datasets for created, removed or updated resources. We retrieved the creation and removal dates for the events directly from Wikipedia as these data are not included in the DBpedia datasets. For the update events we used random dates. Furthermore, we used the DBpedia redirect dataset to identify and generate move events. This dataset contains redirection information derived from Wikipedia's *redirect* pages that are automatically created when a Wikipedia article is renamed (cf. Section 6.2.4). The dates for these events were also retrieved from Wikipedia.

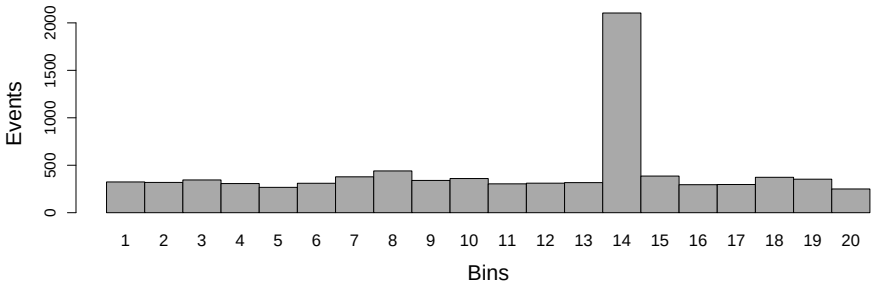
The resulting *eventset* contained 3810 *create*, 230 *remove*, 4161 *update* and 179 *move* events, summing up to 8380 events[11]. An excerpt of this eventset is depicted in Listing 6.1 in Section 6.4. The histogram of the eventset, depicted in Figure 6.10b, shows a high peak in bin 14. About a quarter of all events occurred within

[10] The snapshots contain a subset of all instances of type foaf:Person and can be downloaded from `http://dbpedia.org/` (filename: `persondata_en.nt`).

[11] Another 5666 events were excluded from the eventset as they resulted from inaccuracies in the DBpedia datasets. For example, there are some items in the 3.2 snapshot that are not part of the 3.3 snapshot but were not removed from Wikipedia (a prominent example is the resource `http://dbpedia.org/resource/Tim_Berners-Lee`). Furthermore several items from version 3.3 were not included in

Name	Property Type	Coverage	H	H_{norm}
foaf:name	d	1.00/1.00	9.91/10.28	1.00/1.00
foaf:surname	d	1.00/1.00	9.11/9.25	0.92/0.90
foaf:givenname	d	1.00/1.00	8.23/8.52	0.83/0.83
dbpedia:birthdate	d	0.60/0.60	5.84/5.96	0.59/0.58
dbpedia:birthplace	o	0.48/0.47	4.24/4.32	0.43/0.42
dbpedia:height	d	0.10/0.08	0.65/0.51	0.07/0.05
dbpedia:draftyear	d	0.01/0.01	0.06/0.05	0.01/0.01
Namespace prefix dbpedia: <http://dbpedia.org/ontology/>				
Namespace prefix foaf: <http://xmlns.com/foaf/0.1/>				

(a) Coverage, type, entropy and normalized entropy of all properties in the *enriched* dbpedia 3.2/3.3 persondata sets. The selected properties are written boldface in rows with gray background color. Symbols: object property (o), data type property (d).



(b) Histogram of the distribution of events in the *dbpedia-eventset*. A bin corresponds to a time interval of about 11 days.

Figure 6.10: DBpedia evaluation data.

this time interval. We think that such event peaks are not unusual in real-world data and were interested how our application deals with such situations.

We re-played the eventsets, monitored the changing dataset with DSNotify and measured precision and recall of the reported events with respect to the eventset information (cf. Figure 6.8). We repeated the simulation seven times varying the number of average events per housekeeping interval and calculated the F_1-measure of the reported *move* events. In this experiment, we fixed the housekeeping period for this experiment to 30s and varied the simulation length from 3600s to 56.25s. Thus the event rates varied between 1.17 to 75.00 events/second or 35.1 to 2250.0 events/housekeeping interval respectively. For these calculations we considered only move, remove and create events (i.e., 4219 events) from the eventset as only these influence the accuracy of the algorithm.

For each simulation, DSNotify was configured to index only one of the six selected properties in Figure 6.10a. To calculate the similarity between data type properties, we used the Levensthein distance. For object properties we used a simple similarity function that counted the number of common property values (i.e., objects of the triples) in both resources that are compared and divided it by the number of total values.

Additionally, we configured DSNotify to index only one special feature that contained a calculated *rdf-hash* value. Our RDF hashing function calculates an MD5 hashsum over all string-serialized properties of a resource and the corresponding similarity function returns 1.0 if the hash-sums are equal or 0.0 otherwise. Thus this *rdf-hash* is sensible to any modifications in a resource's instance data (cf. Section 6.5.4).

version 3.2 although the creation date of the corresponding Wikipedia article is before the creation date of the 3.2 snapshot. We decided to generally exclude such items.

Additionally we evaluated a combination of the dbpedia *birthdate* and *birthplace* properties, each contributed with equal weight to the weighted feature vector. The coverage of resources that had values for at least one of these attributes was 65% in the 3.2 snapshot and 62% in the 3.3 snapshot.

Results. The results, depicted in Figure 6.11, show a fast saturation of the F_1-measure with an decreasing number of events per housekeeping cycle. This clearly confirms the findings from our *iimb* evaluation. The accuracy of DSNotify increases with increasing housekeeping frequencies or decreasing event rates. From a pragmatic viewpoint, this means a tradeoff between the costs for monitoring and housekeeping operations (computational effort, network transmission costs, etc.) and accuracy. The curve for the simple *rdf-hash* is surprisingly good, stabilizing at about 80% for the F_1-measure. This can be attributed mainly to the high precision rates that are expected from such a function. The curve for the combined properties shows maximum values for the F_1-measure of about 60%.

The measured precision and recall rates are depicted in the Figures 6.11a and 6.11b. Both measures show a decrease with increasing numbers of events per housekeeping cycle. For the precision this can be observed mainly for low-entropy properties whereas the recall measures for all properties are affected.

It is, again, important to state that our evaluation had not the goal to maximize the accuracy of the system for these particular *eventsets* but rather to reveal the characteristics of our time-interval-based blocking approach. It shows that we can achieve good results even for attributes with low entropy when choosing an appropriate housekeeping frequency.

6.6.3 Evaluation discussion

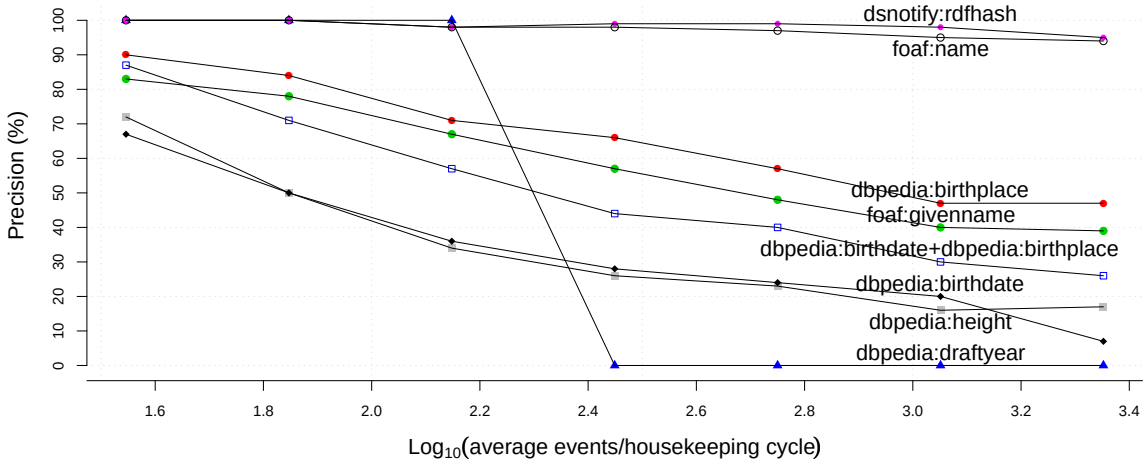
Our evaluation results confirm the feasibility of our approach for event detection. Our time-interval-based blocking method effectively reduces the number of feature vector pairs that have to be compared for the detection of move events: The housekeeping frequency of our tool as well as the dynamics of the considered data source determines the number of feature vector pairs that have to be compared. In turn, the number of these feature vector comparisons as well as the coverage and entropy of indexed features influence the accuracy of our method.

In a summary, one can state that the less comparisons our heuristic has to make, the less mistakes are possible. Just consider the special case where only one move event took place since the last monitoring/housekeeping cycle: our algorithm will then compare only one feature vector pair and predecessor identification is trivial[12].

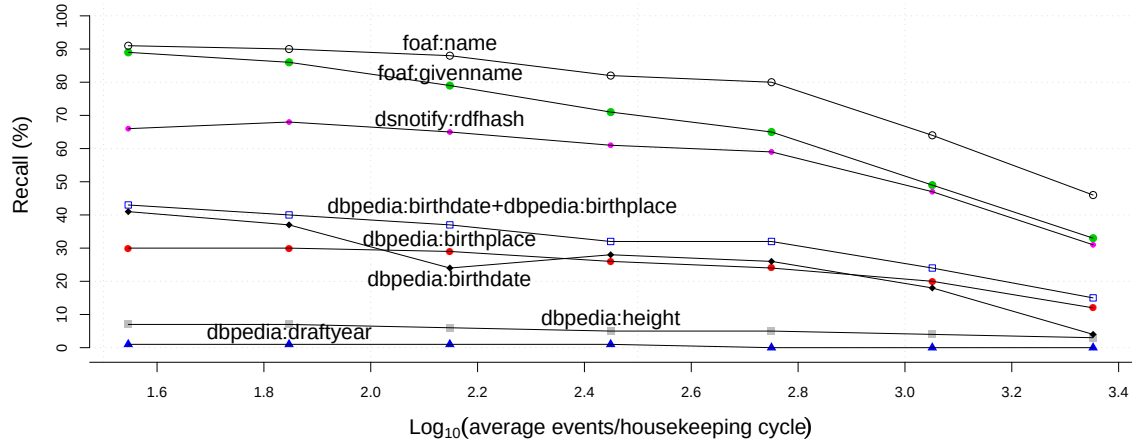
DSNotify can be configured in multiple ways, and the parameter estimation for configuring our system (including the used threshold values, housekeeping frequency, set of extracted features and their weight, etc.) was done manually for this evaluation. However, we used indicators such as the properties' coverage and entropy for the selection of indexed features. An automatic parameter estimation approach for our system could take this as well as information about the dynamics of a data set into account and could then train the application with a previously determined reference eventset.

However, entropy and coverage of the indexed features change over time in dynamic datasets as can be seen in Figure 6.10a. This could, for example, be taken into account by a function that automatically adjusts the weights of the various features in the feature vectors. However, first, the entropy calculation of the individual features is a computationally expensive operation and a periodic re-calculation of this measure would probably raise performance issues. Second, large changes of entropy and coverage could also lead to the need to include new features in the indexed feature vectors respectively to exclude others (as, e.g., their entropy sunk below a certain threshold) which would require re-indexing of the considered data source in the worst case. An algorithm that automatically determines and adjusts the various parameters of DSNotify is beyond the scope of this work.

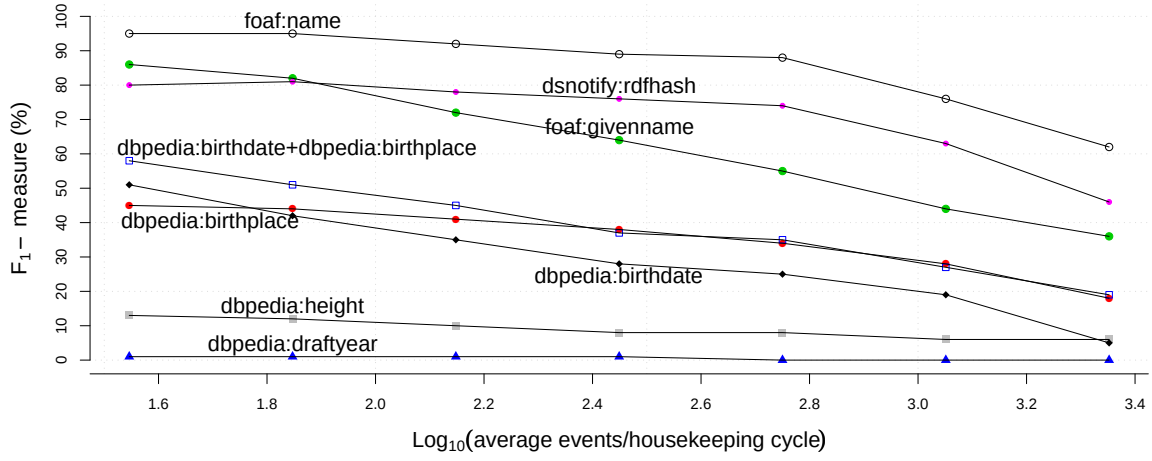
[12] Assuming that no other fundamental changes were made to the resource that resulted in similarity values below the upper threshold of the system.



(a)



(b)



(c)

Figure 6.11: Influence of the number of events per housekeeping cycle on the measured (a) precision (b) recall and (c) F_1-measure of detected *move* events in the DBpedia data set.

Our evaluation infrastructure can be reused for the evaluation of tools that, like ours, are concerned with the detection of events occurring in Linked Data sources. We consider this tool set as a first step towards a benchmark for Linked Data event detection tools: Note that our infrastructure may also be used in another way as described above: one could start with a particular source data set and create an artificial eventset (similar to the artificial ISLab instance matching benchmark alignments). This eventset could then be applied to the source data set (using the simulator) to get the respective target dataset. In this manner, a benchmark for event detection tools consisting of various event patterns could be constructed.

6.7 Related work

6.7.1 Solution strategies from hypertext research

Methods to preserve link integrity have a long history in hypertext research. In Section 5.4 we already discussed general solution strategies for this purpose. We have analyzed several existing approaches, building to a great part on Ashman's work [Ash00] and extending it. Here we want to discuss two of these methods in more detail as they are either widely used by the Linked Data community (*Indirection approaches*) or have been targeted by recent research (*Robust Hyperlinks*).

Indirection

Indirection services, such as persistent URLs (PURLs) and digital object identifiers (DOIs), introduce a layer of indirection between clients and content providers as already discussed in Section 5.4. URI aliases pointing to the indirection service are used to refer to a resource. When these alias URIs are accessed by a client, they are translated to the actual resource URIs and the indirection service forwards the HTTP request or directly delivers the representation retrieved from there. The content providers are responsible for keeping the translation table between actual resource URIs and aliases up-to-date by sending respective notifications to the indirection service. In principle, DSNotify could be used to automatically create such notifications if a content provider cannot provide these otherwise. However, in such a case DSNotify could alternatively be set-up as a standalone indirection service itself easily.

A disadvantage of current indirection services lies in their architecture. At first sight, they are designed to be highly scalable and to provide high quality of service: the Handle System (which is the technological basis of DOIs), for example, constitutes a collection of services that provide access to one or more replicated sites; for the PURL technology, activities for a federated architecture have been announced on the Web[13]. Nevertheless, logically centralized services that provide access to their translation services via a single service URI still constitute single points of failure. Furthermore, these services introduce additional latency when accessing resources (e.g., two HTTP requests instead of one). In situations where this is not acceptable, a decentralized service based on DSNotify that is built into existing data consuming applications might be the better choice.

Another issue with using indirection services is that they require client applications to actually use the alias-URIs instead of the original ones that are usually also accessible via HTTP. This may contribute to additional coreference problems on the Web of Data (cf. [GLMD07]). Nevertheless, indirection is an increasingly popular method on the Web and is often used for the identification of semantic Web vocabularies. Although we consider this a conceivable strategy for this purpose, we believe that a general use of such URI aliases for Linked Data resources is not feasible for the above-mentioned reasons.

[13] http://efoundations.typepad.com/efoundations/2010/03/federating-purlorg-.html

Robust hyperlinks

In [PW00], Phelps and Wilensky propose so-called *robust hyperlinks* based on URI references that are decorated with a small *lexical signature* composed of terms extracted from the referenced document. When a target document cannot be accessed, this lexical signature can be used to automatically re-find the resource using regular Web search engines, an approach often used by humans when faced with HTTP 404 responses (cf. Section 6.3). The authors found out that five terms are enough to uniquely identify a Web resource in virtually all cases. Robust URIs based on lexical signatures were extended by [PPGK04, HN06]. Another extension based on titles of HTML pages is described in [KN10]. A similar approach using key-phrases instead of single terms that exploits these terms to re-find moved HTML pages can be found in [DDD+04].

One major disadvantage of the robust hyperlink approach is that it requires existing URI references to be changed which is not the case with our approach. It is not clear what happens with such URIs when the lexical signature of a document actually changes. Furthermore, it is unclear how to extend this method to non-textual resources whereas our feature vector based approach could in principle be combined with most existing multimedia feature extraction solutions.

Other approaches for the document Web

In [MNI+09], Morishima et al. describe *Link Integrity Management* tools that focus on fixing broken links in the Web that occurred due to moved link targets. Similar to DSNotify, they have developed a tool called *PageChaser* that uses a heuristic approach to find missing resources based on indexed information (namely URIs, page content and redirect information). An *explorer* component, which makes use of search engines, redirect information, and so-called *link authorities* (Web pages containing well-maintained links), is used to find possible URIs of moved resources. They also provide a heuristics-based method to calculate such link authority pages. A major difference to our approach is that PageChaser was built for fixing links in the (human) Web exploiting some of its characteristics (like locality or link authorities), while DSNotify aims at becoming a general framework for event detection based on domain-specific content features.

Peridot is a tool developed by IBM for automatically fixing links in the Web. It is based on the patents [BF04, BF07] and the basic idea is to calculate *fingerprints* of Web documents and repair broken links based on their similarity. The method differs from DSNotify in that we consider the structured nature of Linked Data and support domain-specific, configurable similarity-heuristics on a property level which allows more advanced comparisons methods. Furthermore, DSNotify introduces the described time-interval-based blocking approach and detects also create, remove and update events.

The XChange language [BP05] provides the necessary means to implement reactive behavior on the Web. Web site providers can formulate so-called event queries on specific aspects of their data using common query languages such as XQuery, incrementally evaluate those queries and send event messages to their subscribers. The design of DSNotify, in contrast to XChange, is driven by the practical experience that consumers of Linked Data sources cannot rely on data providers to implement such a notification infrastructure on top of their existing data source. At the time of this writing, to the best of our knowledge, not a single Linked Data source provides such a service. Therefore, for the majority of use cases, we expect DSNotify to be set up as a standalone monitoring service that detects change events in remote data sources, which are not under the control of the consuming application provider (cf. the BBC usage scenario described in Section 6.5.8).

6.7.2 Related work from Semantic Web research

Similar to our eventset vocabulary, the Talis Changeset Vocabulary can be used to describe *changesets* that encapsulate the differences between two versions of a resource description. Our design differs from the *changeset* vocabulary mainly because (i) we do not consider the triple as subject of change but apply a resource-centric view (cf., Section 6.2), (ii) we preserve the timely order of changes (iii) our model also defines a special `MoveEvent` that can be considered as a simple composite event (a remove and a create

event). By our `hasAffectedTriples` property we bridge the gap between the resource- and triple-centric view. If required, `ResourceChangeEvents` can be directly linked to the created/modified instance data.

The Web of Data Link Maintenance Protocol (WOD-LMP) [VBGK09a] is a SOAP-based protocol for communicating changes from a server to a client. It delivers *change notifications* that contain sequences of resources that were created, removed or updated. The proposed XML schema does not provide means to include what data associated with these resources was changed (e.g., what triples were added/removed), a *move* event type is not foreseen.

Triplify [ADL+09] is a system that maps HTTP requests to SQL queries and exposes data retrieved from relational databases as Linked Data. It also provides so-called *Linked Data Update Logs*: The updates occurring in a particular *triplified* data source are logged and exposed as *nested (RDF) update collections* which are described using the Triplify update vocabulary[14]. This vocabulary, however, covers only resource updates and deletions, the changed data itself is not associated with such *UpdateCollections*.

In [VHC10], the authors propose an alternative approach for dealing with the broken link problem in the Web of Data: by exploiting coreference and redundancy in Linked Data resource descriptions, they try to find and return Linked Data about an URI of interest. Their proposed algorithm is based on so-called URI discovery endpoints that are able to provide "equivalent" URIs to a considered URI. However, such services (e.g., sameas.org[15]) are not generally available for Linked Data, neither are redundant resource representations. Further, their approach cannot be used to re-find moved Linked Data resources and fix broken links.

PingtheSemanticWeb[16] is a central registry service that does not index the accessed data but offers a periodically updated list of new or updated (changed) RDF documents. This service reports creation and update events of registered URIs on a document level but cannot be used as an event detection framework as presented in this thesis.

Finally, in [VdSNB+10], Van de Sompel et al. discuss a protocol for time-based content negotiation that can be used to access archived representations ("Mementos") of resources. By this, the protocol enables a kind of time-travel when accessing archived resources. DSNotify could be used to build such archives when a monitor implementation was used that would store not only a feature vector derived from a resource representation but also the representation data itself.

6.7.3 Related work from database research

The data dynamics problem is already well known in the database domain. Chawathe et al. [CRGMW96], for instance, proposed a change detection algorithm that calculates the minimum distance between two hierarchically structured data sets. Ipeirotis et al. [INCG07] analyzed changes in text databases, proposed a model for representing changes over time, and predicted schedules for updating content summaries.

Research in the area of active database systems — a survey of which is provided by Paton and Diaz [PD99] — investigated how database systems can automatically respond to events that are taking place either inside or outside the database system itself. Active databases make use of monitoring devices to detect events and formulate so-called ECA-rules (event-condition-action) to describe the runtime strategy for specific events. In today's DBMS such rules are known as *triggers*.

DSNotify shares with active database systems the event-based approach for describing changes. It supports the detection of primitive events (insert, update, delete) and, at the moment, a single composite event (move). The reaction on changes under given conditions, however, is out of the scope of DSNotify because this is highly application-dependent and hardly generalizable for the use cases described in Section 6.2.1. Furthermore, certain assumptions that hold in closed (distributed) database settings, such as global event detectors that monitor inter-site composite (events), are inappropriate for open environments such as Linked Data or the Web in general. Therefore, we designed DSNotify to be a pragmatic solution for detecting

[14] http://triplify.org/vocabulary/update
[15] http://www.sameas.org
[16] http://pingthesemanticweb.com

changes in a set of pre-defined dataspaces assuming that the application designers know which dataspaces are relevant to be monitored for changes.

6.7.4 Other related work

Besides the already cited works, our research is also closely related to the areas of record linkage, a well-researched problem in the database domain, and instance matching, which is related to ontology alignment and schema matching.

Record linkage is concerned with finding pairs of data records (from one or multiple datasets) that refer to (describe) the same real-world entity. This information is useful, e.g., for joining different relations or for duplicate detection. Record linkage is also known under many other names, such as *object identification*, *data cleaning*, *entity resolution*, *coreference resolution* or *object consolidation* [Win06, HHD07b, EIV07]. Record linkage is trivial where entity-unique identifiers (such as ISBN numbers or OWL inverse-functional properties like *foaf:mbox*) are available. When such additional identifiers are missing, tools often rely on probabilistic distance metrics and machine learning methods (e.g., HMMs, TF-IDF; SVM). A comprehensive survey of record linkage research can be found in [EIV07].

The *instance matching* problem is closely related to record linkage but requires certain specific methods when dealing with structural and logical heterogeneities as pointed out in [FLMV08].

6.8 Discussion

Some data sources like the Wikipedia (and its Linked Data version DBpedia) already do track all changes of their data. In the case of the Wikipedia this includes, for example, *article rename* events which are then exploited for creating the DBpedia redirect links. This information may already be used by Linked Data consuming applications to fix links to respective DBpedia resources. However, not all Linked Data sources are able to provide this information. In such cases event detection tools such as DSNotify are required to avoid broken links to resources in remote data sets.

We presented the broken link problem in the context of Linked Data as a special case of the instance matching problem and showed the feasibility of a time-interval-based blocking approach for systems that aim at detecting and fixing such broken links.

In the end, the various parameters of DSNotify like monitoring- and housekeeping frequency, feature vector size, thresholds, etc. can be used to configure the tradeoff between the system maintenance costs (e.g., network transmission costs for monitoring operations, CPU and I/O costs for housekeeping, storage costs for feature vectors, etc.) and the accuracy and timeliness of the event detection. Additionally, this tradeoff is influenced by the dynamics and the composition of the considered data as demonstrated by the discussed influence of entropy, coverage and number of events per housekeeping cycle. The automatic determination of such parameter sets is left for future research.

We presented our tool for event detection in Linked Data sources and a vocabulary for describing the resulting eventsets. Further, we described a reusable infrastructure for the evaluation of event detection tools similar to ours. On the one hand, this infrastructure may be used by developers of dynamics-aware applications to create testbeds that are able to reproduce the inherent dynamics of Linked Data sources. On the other hand, it could be used to develop a first benchmark for tools dealing with dataset dynamics issues.

The various interfaces for accessing the data structures built by DSNotify facilitate the integration with existing applications. Further, our approach is by design a semi-automatic solution that is capable of integrating human intelligence in the sense of human-based computation and we plan to further elaborate on this issue. However, DSNotify cannot “cure” the Web of Data from broken links. It may rather be used as an add-on for particular data providers that want to keep a high level of link integrity and thereby quality in their data.

Feature	Data type	Similarity	Weight
Last access	Date	Plausibility	
Last modification	Date	Plausibility	
IsDirectory	Bool	Plausibility	
Checksum	Integer	Plausibility	
Name	String	Levensthein	3.0
Extension	String	Major MIME type equality	1.0
Path	String	Levensthein	0.5
Size	Long	Equality	0.1
Permissions	Bitstring	Equality	0.1

Table 6.3: Extracted file system features, their data type and the strategy used to calculate a similarity between them. Features that are used only in plausibility checks have a value *Plausibility* here. Table reprinted from [SP10].

6.8.1 Applicability for the file system

The flexibility of our DSNotify tool is founded in its generic nature and its customizability. Thus, the development and evaluation of additional *monitors*, *features* and *extractors*, *heuristics* and *indices* allows the application of DSNotify in other domains, such as the file system or the *Web of Documents*.

As discussed, our proposed metadata model requires stable links between remote and local resources and DSNotify could be set-up to ensure stable links to remote Linked Data resources. In a next step we need to ensure also stable links between local file system objects and their metadata records. Consequently, we specialized the generic event-detection approach of DSNotify for the detection of file system events and used these detected events for the maintenance of a stable mapping between local file locations and globally unique URIs (such a mapping is described later in this thesis in detail in Section 8) serving as identifiers for such metadata records.

In our first attempt, published in [SP10], we implemented a generic file-feature extractor for DSNotify that extracts low-level features from local files (Table 6.3). Further, we have developed a simple heuristic function that calculates the plausibility that a file (described by the feature vector X) was moved to another location (the file there being described by the feature vector Y).

This heuristic consisted of two parts: first, plausibility checks were performed. For example, if the last modification date of file Y is before the one of file X, it cannot be a successor of X. Another example is that a file cannot become a directory or *vice versa*. Second, a similarity metric between the remaining features was calculated by using the strategies listed in Table 6.3. The resulting similarities were weighted (e.g., the name similarity was considered more important than equal file sizes), summed up, and normalized. These similarities were then used by DSNotify to distinguish between *move*, *remove* and *create* events in the local file system. Furthermore, DSNotify reported update events based on changes in the extracted feature vectors.

We exploited the generic design of DSNotify and developed a special *monitor* implementation that periodically monitored a subtree of the file system. Using the Java file API we extracted feature vectors composed of the file features described above and these vectors were stored in the DSNotify indices. This implementation worked good at first sight; however, in an informal evaluation (data not shown) it became evident that this first, naïve approach suffered from several issues that influenced its accuracy:

Feature selection. First, the selection of features as well as their weight was based solely on our own subjective choice (which was, however, influenced by several test-runs with the system). However, from the DSNotify evaluation in Section 6.6 it became evident that coverage and entropy of the considered

features have a major impact on the accuracy of the overall event-detection algorithm. Thus, one very important criterion for deciding what features should be included in a feature vector is knowledge about its value distributions in real-world data sets. Because of this, we conducted a study of the value distributions of low-level file system features that is presented in Chapter 7 of this thesis.

Feature comparison. A further crucial factor for reaching high event detection accuracies is obviously the method for comparing features with each other. The key for choosing the proper method here is knowledge about the value-dynamics of the considered features. If we know, for example, that file sizes more often increase than decrease over time, a comparison function of this feature could take this into account. We propose one possible set of features and corresponding comparison functions that were designed based on the above-mentioned file system study in Section 8.3.1.

Special file system characteristics. Our initial approach did not take an inherent characteristic of hierarchical file systems into account which led to reduced overall accuracies. This characteristic is that one single folder move operation in the file system results in changed locations for all file system objects contained in this folder and its subfolders. A method that takes this coherence into account may save itself a lot of unnecessary feature vector comparisons which corresponds with less computational costs but also with increased move detection accuracy. We describe this aspect and our solution for this in detail in Section 8.3.3.

Performance issues. Certain bulk operations in the file system may result in high event peaks. Although the implementation we describe above already used a native component that informed DSNotify about changes in the file system, we ran into performance issues in some test scenarios due to the high number of file system events in certain situations. We do not consider this as a general issue of DSNotify as we currently do not observe data sources with such high dynamics in the Linked Data domain.

We concluded from these issues that a specialized algorithm for the detection of low-level events in the file system might be required as our two main data sources (Linked Data vs. file system) differ in various characteristics that include (i) coverage and value distributions of low-level features, (ii) real-world dynamics of these features, (iii) special data organization characteristics and (iv) performance requirements.

In order to deal with these issues in the file system, we first conducted a study of the value distributions of low-level file system features that is presented in Chapter 7. Based on these results as well as on our experiences with DSNotify, we developed a specialized algorithm (Gorm) for the detection of file system events that is presented and evaluated in Chapter 8. Finally, we present Y2, a prototypical implementation of our proposed metadata model that makes use of DSNotify and Gorm for the maintenance of stable links between local and remote resources in Chapter 9.

Chapter 7

A study of low-level file system features

7.1 Introduction

As outlined in the previous chapters, our proposed method for file move detection is based on feature vector comparison of low-level file features (e.g., file name, file extension, size). However, the careful design of such a method requires several questions to be answered beforehand, such as (i) what features should be included into a feature vector, (ii) what similarity functions should be used to compare these features, (iii) how to design an algorithm that can efficiently solve the file move-detection problem based on similarities between feature vectors.

In order to approach these questions we were interested in the value-distributions of low-level file system features in real-world data sets. Knowledge and data about how these values are spread over file systems and in particular about how they change in the course of time can be exploited in multiple ways: First, it can be used to decide on the feasibility of an algorithm that relies to a large part on the comparison of such features. Second, these data can be used to guide the design process of such an algorithm. Third, such data can be exploited for a first estimation about what magnitudes of feature vector comparisons have to be done in what time and about expected storage and computation costs.

In this chapter, we present a study of the real-world distributions of low-level file features and compare the results with previously published data. We analyze the value distributions of low-level file features in multiple ways (extreme values, distinct and unique values, collision probabilities, entropies) and discuss the results in the context of our proposed file move detection method. The outcome of this study served as input for the design of similarity functions for comparing various file features. It influenced the overall algorithmic design and provided the fact basis for performance considerations. Finally, it was considered for the careful design of the evaluation method of our algorithm that is presented in the next chapter.

7.2 Related work

A classical yet a bit outdated study of file sizes and lifetimes was conducted in 1981 and published in [Sat81]. Some main conclusions were that (i) file size distributions are strongly skewed towards small sizes, (ii) most files are used very few times, (iii) older files are used less frequently than younger files.

Another early study on the distribution of file sizes on Unix file systems was done by Mullender and Tanenbaum in 1984 [MT84]. This study was repeated 21 years later in 2005 [THB06]. As main conclusions, it turned out that the median file size in the considered data sets has increased from 1.0 kB to 2.4 kB and that they consider 4 kB as the optimal block size for file systems as with this number nearly 60% of all files fit

into a single disk block while "wasted" storage size when compared to smaller block sizes is acceptably low. Their reported median file size compares well to our results (2.3 kB), see below.

The study presented in [BBK92] also confirms that most files are small. The authors further concluded from their data that text files slowly increase in their average size and that the overall number of files is increasing over time. They also show that most files (over 80%) in their data set were last updated more than one month before their measurement, confirming our results.

A file size survey from 1993 reports on size statistics from over 12 million files stored in Unix file systems. The author reports on a median of 2 kB and an average file size of 22 kB. Further it was found that 89% of files take up 11% of disk space [Irl93].

[Vog99] studies the usage of NTFS by analyzing long-term kernel traces of I/O events (cf. Section 8.5.4). This study considers low-level I/O operations that cannot be trivially mapped to the file API events considered in this thesis (file create, file delete, file move and file update). However, these and related studies complement our work.

[Dow01] examined file sizes from 562 file systems and concluded that file sizes follow a lognormal distribution. A feature X is lognormally distributed if $ln(X)$ is normally distributed. Another study focusing on multimedia files showed that a lambda distribution might be better-suited to fit the size distribution in their data [EK02].

In 2007, Microsoft conducted a large study on file system metadata published in [ABDL07]. In this study, the authors analyzed crawls of Windows PCs over a period of five consecutive years. They collected data from 63,000 Windows company PCs in total, for 18 file systems they had scans for all five years. Although their study was done on a large dataset and contains many valuable insights on the value distributions of low-level file features, we conducted our own (much smaller) study for the following reasons: First, the scans in this study were anonymized due to the applied privacy policy. This means, that they had no access to the real filenames and paths of the scanned files and therefore did not study their value distributions. As our method, however, considers both features we were interested in such data. Second, their datasets are biased as only Windows company PCs were considered. We were interested whether we would find considerable deviations on other operating systems/file systems. Third, the comparison of snapshots that are taken one year apart cannot reliably reveal the change frequencies of file features on the short term as required by our algorithm. We do, however, compare our results with the findings of this study throughout the following section and report on notable accordance and disagreement where appropriate.

In [Day08], the author reports on file attribute statistics collected from 13 supercomputing file systems. Their report includes a comparison of file size distributions of various scientific publications.

In [HDPM08], the authors describe how mechanical engineers manage their personal electronic files. They present a study in which they scanned 40 file systems of engineers from a variety of industrial and academic backgrounds and analyzed extreme values and means of several properties of their scans, such as directory depth, numbers and sizes of files and folders and percentages of duplicate folder and file names. Further, they analyzed the distribution of file modification dates. Their published data compares well with our results, however, as the Microsoft study discussed above, their study does not contain an analysis of all features considered in this thesis.

Further related work can be found in the area of computer forensics where it is common to scan whole file systems and analyze various features of the contained file system objects. A noteworthy project is the Real Data Corpus, "[...] a collection of raw data extracted from data-carrying devices that were purchased on the secondary market around the world." [1] In [RG10], the file times of this corpus (over 1000 disc images, over 5 million files) were analyzed with the goal to identify the primary drive usage and to detect atypical time patterns. The authors used this information, for example, to identify file time patterns that could be exploited for the identification of downloaded or copied files and log files.

[1] http://simson.net/page/Real_Data_Corpus, accessed June 2011.

7.3 Data sets

Basic data sets. We scanned 15 volumes of various file systems containing around $4 \cdot 10^6$ file system objects (FOs) representing approximately 1 TByte of data. These 15 basic data sets are summarized in the table in Figure 7.1a, data sets sharing a common number (e.g., 1a and 1b) correspond to volumes of a single PC (e.g., for data set 1 this was a Mac OS X laptop with a Windows Bootcamp partition).

We scanned mostly Windows volumes as this is the currently predominant operating system on the PC market (cf. Section A.2.1); however, also Mac OS X machines and one Linux PC were considered. Data sets 10a and 10b correspond to two USB hard discs that were used as external storage devices for Windows PCs. Ten data sets correspond to scans of PCs that were primarily used for business purposes in a computer science department. The others were primarily used for private purposes.

Special folder data sets. Besides the data sets in Figure 7.1a, we further analyzed derived data sets summarized in the tables in the Figures 7.1b and 7.1c. The data sets in Figure 7.1b were created by considering only the contents of the special folders *My Documents*, *My Pictures*, *My Music*, *My Videos* and *Desktop* of the respective Windows data sets. For Windows 7 data sets (data sets 5-7) we considered the respective *Users/profile/Documents*, *Users/profile/Music*, etc. folders. These folders are recommended by Microsoft for the organization of user-generated documents and multimedia data and usage is encouraged by special OS integration (e.g., the Windows file browser displays them as the first entries in the file tree). Thus we were interested to what extent users actually make use of these folders.

Merged data sets. Further, we created merged data sets of all basic data sets (s1), all basic Windows data sets (s2), all Mac OS X data sets (s3) and all special folder data sets (s4). Statistics about these data sets are provided in the table in Figure 7.1c.

7.4 Methodology

Besides calculating several statistical standard measures such as means, medians and standard deviations of the collected data, we further calculated the normalized *Shannon entropy* and the *collision probabilities* for the considered features as discussed below. These measures gave us valuable information about the value distributions of the respective features and provided us with well-grounded evidence about what features qualify to what extent for FO identification via feature comparison.

Coverage and entropy. As discussed in Section 6.6.3, entropy and coverage of features have a major impact on the accuracy of a similarity function based on feature vector comparison. Shannon entropy is a measure of the randomness of a distribution; coverage describes what fraction of a considered set of items actually possesses a particular feature.

As we rely on low-level file features, coverage is not an issue as the file system provides values for all the features of a particular FO type (file or folder). Entropy, however, is considerably different for the various features. Low entropy of a feature means that it is not well-suited for an identification task via distance/similarity comparison as there are large clusters of FOs sharing the same value for this feature. This does not necessarily mean that this feature should not be considered by a move detection algorithm in another form, e.g., for plausibility checks as discussed below.

ID	OS	FS Type	Primary usage	Volume Fullness	Size (GB)	Files	Folders	Folder (%)	FOs per Folder
1a	WinXP	NTFS	business	21.1%	29.2	73,006	18,050	20%	5.04
1b	OSX	HFS+	business	51.8%	8.4	17,968	9,573	35%	2.88
2	OSX	HFS+	business	54.7%	98.7	586,258	169,917	22%	4.45
3a	WinXP	NTFS	business	60.6%	69.2	629,337	145,990	19%	5.31
3b	WinXP	NTFS	business	54.4%	18.4	39,381	8,389	18%	5.69
3c	WinXP	NTFS	business	20.9%	142.0	258,655	91,338	26%	3.83
4	WinXP	NTFS	business	11.9%	47.2	300,673	81,516	21%	4.69
5	Win7	NTFS	private	99.7%	123.4	269,536	92,225	25%	3.92
6	Win7	NTFS	private	22.4%	62.6	121,175	22,292	16%	6.44
7	Win7	NTFS	private	22.7%	67.2	237,225	28,241	11%	9.40
8	OSX	HFS+	business	72.7%	142.9	207,338	251,057	55%	1.83
9	OSX	HFS+	business	26.5%	25.3	116,281	35,139	23%	4.31
10a	-	NTFS	private	94.2%	111.5	57,725	3,726	6%	16.49
10b	-	NTFS	private	47.6%	112.3	40,530	1,986	5%	21.41
11	Linux	EXT3	business	13.3%	12.1	100,977	22,749	18%	5.45
			sum		**1,070.4**	**3,056,065**	**982,188**		
			average	45.0%	71.36	203,738	65,479	21%	6.74
			std.dev.	28.4%	47.60	188,401	73,653	12%	5.32

(a) Basic data sets. The table summarizes the data set identifier (as used in this thesis), the operating system of the scanned PC (data sets 10a and 10b correspond to external devices), the file system type, the primary usage of the volume, the relative fullness of the volume, the total size in GBytes, the number of files and folders, the fraction of folders of all file system objects, and the average number of file system objects per folder.

ID	Size (GB)	Files	Folders	Folder %	FOs/folder
1a.1	0.33	92	23	20%	4.61
3b.1	0.93	1,423	284	17%	5.98
4.1	1.42	12,810	2,194	15%	6.84
5.1	40.75	5,956	245	4%	25.11
6.1	25.96	9,036	550	6%	17.35
7.1	0.34	608	167	22%	4.36
sum	**70**	**29,925**	**3,463**		
average	12	4,988	577	14%	11
std.dev.	17	5,184	811	7%	9

(b) Special folder data sets. These data sets were created by extracting only FOs contained in the Windows special folders *My Documents* and *Desktop* of the corresponding basic data sets. The respective Windows 7 folders were considered for the data sets 5.1, 6.1 and 7.1.

ID	FS Type	Files	Folders	Folder %	FOs/folder	Description
s1	mixed	2,893,686	950,676	25%	4.02	merge of all data sets
s2	NTFS	1,864,864	462,241	20%	4.98	merge of all Windows data sets
s3	HFS+	927,845	465,686	33%	2.99	merge of all Mac OS X data sets
s4	NTFS	29,821	3,413	10%	9.70	merge of all special folder data sets

(c) Merged data sets. Note that duplicates (i.e., FOs sharing the same file URI) were removed, thus there are deviations in the FO counts when compared to the table in Figure 7.1a

Figure 7.1: Data sets used in the file feature study.

In practice, we estimated the entropy of the considered features by replacing the discrete probabilities in the Shannon entropy (cf. Formula 6.1) with their *maximum likelihood* estimates $\hat{p}_i$. That is, when considering N observations with n_i being the number of observations (FOs) sharing a common feature value then $\hat{p}$ is calculated as in Equation 7.1.

$$\hat{H}(p) = -\sum_{i=1}^{M} \hat{p}_i \, ln(\hat{p}_i) \quad with \quad \hat{p}_i = \frac{n_i}{N} \tag{7.1}$$

For comparison, the entropy is usually normalized by dividing by the logarithm of the number of distinct clusters (M) sharing the same feature value (Equation 7.2).

$$\hat{H}_{norm}(p) = \frac{\hat{H}(p)}{ln(M)} \tag{7.2}$$

Note that it is known that maximum likelihood estimation of entropy leads to a systematic underestimation of the real entropy H [Sch04, BDKR05].

Theoretical collision probabilities. Besides entropy estimation, we further consider the *theoretical collision probability* of a file feature as a related, yet more intuitive measure to decide whether a file feature is a good candidate for an identification task using feature comparison. The theoretical collision probability is the probability that two randomly chosen FOs share the same value for a particular feature (e.g., the probability that two files have the same file extension). This may serve as an indicator for how probable it is that such an identification algorithm compares a newly created file with another, recently removed file that is not its predecessor but shares the same file feature value. The probability of drawing two files with the same feature value can be calculated with help of the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{(n-k)! \quad k!} \qquad with \quad 0 \leq k \leq n \tag{7.3}$$

The binomial coefficient denotes the number of possible combinations to choose k FOs from a set of n FOs. Consider that we have split up the set of all files N (with cardinality n) into k distinct subsets E_i (with cardinalities e_i) where $\cup_{i=0}^{k} E_i = N$. Each subset E_i contains all files sharing the same file feature value, e.g., all files with the same file extension. We can then calculate the probability to draw two files with the same file extension by the formula given in Equation 7.4.

$$p_{collision} = \frac{\sum_i \binom{e_i}{2}}{\binom{n}{2}} = \frac{\sum_i \frac{1 \cdot 2 \cdot \ldots \cdot (e_i - 2) \cdot (e_i - 1) \cdot e_i}{1 \cdot 2 \cdot \ldots \cdot (e_i - 2)} \cdot \frac{1}{2!}}{\frac{1}{2!} \cdot \frac{1 \cdot 2 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n}{1 \cdot 2 \cdot \ldots \cdot (n-2)}} = \frac{\sum_i e_i \cdot (e_i - 1)}{n \cdot (n-1)} \tag{7.4}$$

Feature	Data type	*API*	*f*	*d*	Description
name	String	+	+	+	File name
extension	String	-	+	+	File extension
size	Long	+	+	-	File size
FO type	Boolean	+	+	+	Flag that indicates whether a FO is a folder or a file
treedepth	Integer	-	+	+	Depth in the file system hierarchy
lastModified	Long	+	+	+	Last modification date, time in milliseconds since 00:00:00 GMT, January 1, 1970
uri	String	-	+	+	File URI
canExecute	Boolean	+	+	+	Flag that indicates whether a FO is executable
canRead	Boolean	+	+	+	Flag that indicates whether a FO is readable
canWrite	Boolean	+	+	+	Flag that indicates whether a FO is writable
hidden	Boolean	+	+	+	Flag that indicates whether a FO is hidden
FO/folder	Integer	-	-	+	Number of FO entries in a folder
mime	String	-	+	-	MIME-type, determined by using a predefined extension/mime-type mapping.

Table 7.1: Low-level and calculated file features contained in our data sets. The boldface features were analyzed in detail in our study. The column *API* indicates whether the respective feature values were retrievable directly via the standard file API (+) or had to be calculated from other low-level features (-). The columns *f* and *d* indicate whether this feature is available (+) or not (-) for files (f) respectively folders (d).

Considered file features. In this study, we considered low-level file features that are available on all major file system implementations via standard APIs, such as file name or file size. We further considered special "features" that can easily be calculated from these low-level features, such as the depth in the file system hierarchy. The considered features are listed and briefly described in Table 7.1.

7.5 Results

In the following section, we discuss the analyzed value distributions of the considered features in detail.

The FO/folder feature. The FO/folder feature describes how many file system objects are contained in a folder. The table in Figure 7.3a summarizes some statistics about this feature. The mean number of FOs per folder is about 4 in our merged data set s1. The extreme values show that there is a considerably wide range of possible values for this feature. However, folders with large numbers of entries are an exception as also confirmed by the data published in [ABDL07]. The measured median value of this feature was 1, indicating that most of the folders were either empty or contained only one FO. The histogram in Figure 7.2 shows that the vast majority (97%) of all folders have less than 10 entries.

FO type feature and boolean flags. The boolean features listed in the table in Figure 7.3b describe whether a FO is (i) a directory, (ii) readable, writable, executable and/or (iii) marked as hidden by the file system. The *FO type* feature is a boolean flag indicating whether a file system object is a regular file or a folder. In our data (see Figure 7.1a), up to a third of a volume's FOs are folders, data set 8 being an outliner containing more folders than files. The merged data set s1 consisted of 25% folders and 75% files. Naturally, the *canRead* flag was set for all scanned FOs. 83% of them were writable and 76% were marked as executable. This latter flag is not only used for real executables (programs) but also to indicate, e.g., whether the entries of a folder can be accessed. Note that our scan programs could not traverse folders that are not marked as executable either.

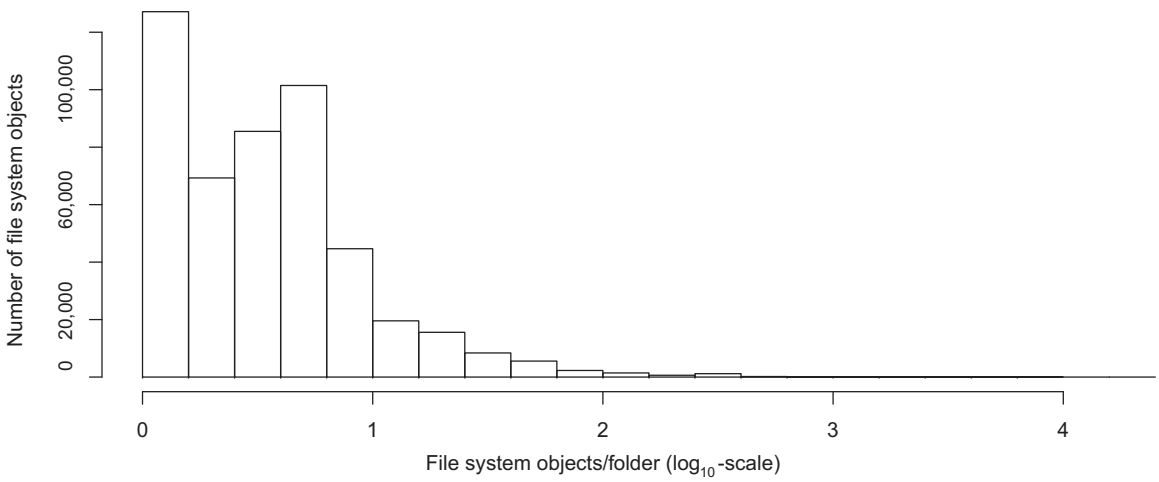


Figure 7.2: Histogram of the number of file system objects per folder. Note the logarithmic scale (base: 10) on the x-axis.

2% of all scanned files were marked as *hidden*. Such files are usually hidden from users by applications or by the operating system.

File size. Our measured distribution of file sizes compares well with the results published in [ABDL07], Fig. 2. Please note that our histogram in Figure 7.4 was calculated over all files in our data sets (i.e., it is not a histogram of the means of the individual data sets), hence the much larger range on the y-axis. Additionally, our histogram contains a bar for the files with zero size which make up about 1.8% of all files ([ABDL07] reported 1-1.5%). We measured a mean file size of 351.3 kB and a median file size of 2.3 kB. This means that most files in our data sets are small (2.3 kB or smaller) as confirmed by all studies mentioned in Section 7.2.

File names and extensions. We normalized all file extensions to lowercase letters before we analyzed them as we were interested in a distribution of various groups of files (e.g., video files, audio files, images) in our data. In total, we found 19,067 distinct filename extensions in our data sets; however, the majority of these were unique: 13,854 (73%) extensions were used only by a single file. This can mainly be contributed to automatically created files. For example, we found a large number of files created by one application that uses sequential numbers as filename extensions for its data files.

For a further analysis, we considered only a subset of popular filename extensions that is summarized in the table in Figure 7.5b. We grouped these files along their major mime type categories into audio and video files, images and documents (Figure 7.5a). We were mainly interested in this classification as we consider these files as good candidates users might want to attach user-contributed metadata to. As evident in Figure 7.5a, the majority (about 70%) of the scanned files does not fall in any of these categories. About 97% of the files belonging to one of the categories of Figure 7.5b are images or documents. On average, we classified about 10% of all files as images and 12% as documents ([HDPM08] report 20% images in their data set). Only small numbers of audio and video files were found. It is further notable that a considerable percentage (about 10%) of files had no extension at all.

Data set s4 shows that there seems to be an accumulation of media and document files in the Windows special folders. A Fisher's exact test confirms the statistical significance ($p\text{-}value = 0.0001305$) of this correlation between the relative media proportions in data set s2 (merged Windows scans) and data set s4

Feature	Min	Max	Mean	Median	SD
FO/folder	0	18,954	4	1	47.9
File size	0 bytes	12.8 GB	351.3 kB	2.3 kB	17.7 MB
Tree depth	0	29	11	10	4.3
Name length	0	255	13	9	12.7

(a) Statistics of the measured numerical features (FOs per folder, file size, depth in the file hierarchy and FO name length) in the merged data set s1. The table shows minimum, maximum, mean, median and standard deviation of the respective feature values.

Feature	Value	Percentage
FO type	"folder"	25%
canRead	true	100%
canWrite	true	83%
canExecute	true	76%
hidden	true	2%

(b) Statistics of the measures boolean features in the merged data set s1.

Figure 7.3: Statistics of numerical and boolean features.

(Windows special folders). The independence between the relative differences of the Windows (s2) and the Mac OS X (s3) scans, however, cannot be rejected with a Fisher test (p - *value* = 0.1606).

For an analysis of the FO names in our data set, we pruned the filename extension and the separating dot from them. A histogram of the lengths of these names is depicted in Figure 7.6, the extreme values of this feature are given in the table in Figure 7.3a. It can be seen that most FO names are shorter than 10 characters. A considerable number (about 2%) of the FOs had no name (i.e., these FOs were "*.extension*" files or folders). The longest name we found was 255 characters long which is also the respective file system maximum.

Last modification date. An analysis of the dates of last modification of the scanned FOs enabled us to learn about what fraction of the FOs in a file system were recently updated respectively remained unchanged for a longer time. The histogram in Figure 7.7 shows that the vast majority of FOs was last modified longer than a month before the date of the respective file system scan. Similar results are reported in [HDPM08].

Depth in the hierarchy. We were further interested in the depth of FO locations in the file system hierarchy. For example, files residing directly in the root folder of a file hierarchy have a depth value of 1, per subfolder this depth feature increases by 1 (e.g., a file /home/root/mydir/test.dat has a depth value of 4). The distribution of this property can be seen in Figure 7.8.

The extreme values of this feature (cf. Figure 7.3a) show that file system hierarchies can become quite deep (our observed maximum was 29 levels). However, the median shows that most FOs reside within the first 10 levels of these hierarchies. The average depth of a FO in the hierarchy was 11 levels. The histogram in Figure 7.8 shows a peak at depth 7. [ABDL07] report such a peak at the same level 7 where the Windows Web cache folders are found. Our data shows a second peak at level 4 that is also reported in [HDPM08].

Shannon entropies and collision probabilities. Finally, we analyzed six file features (Table 7.1) in more detail and calculated their absolute and normalized entropies, their collision probabilities, the average number of distinct values and the average number of unique values (i.e., the number of distinct feature values that are assigned to a single FO).

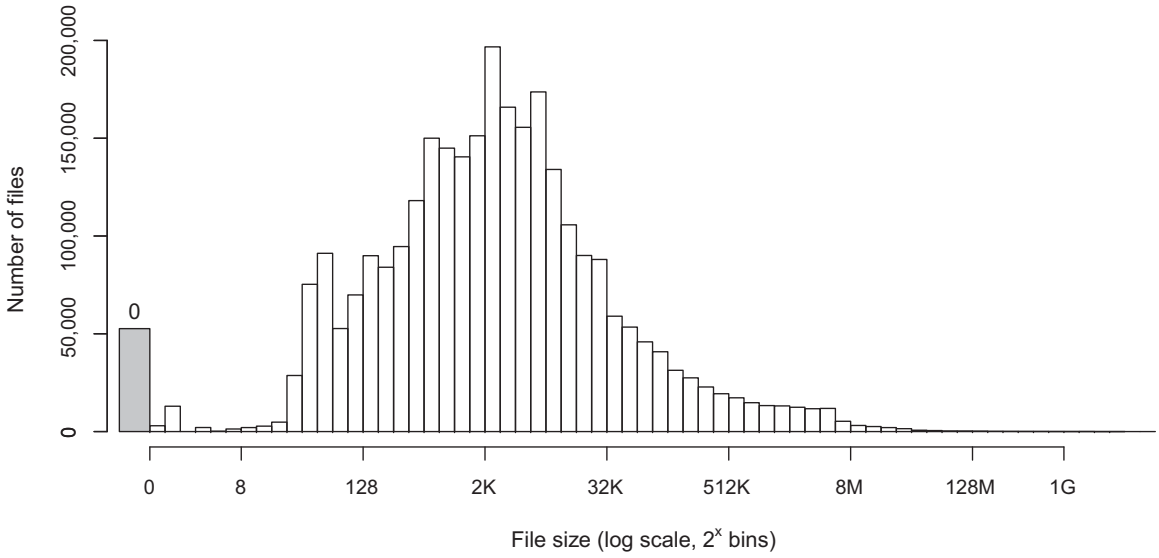


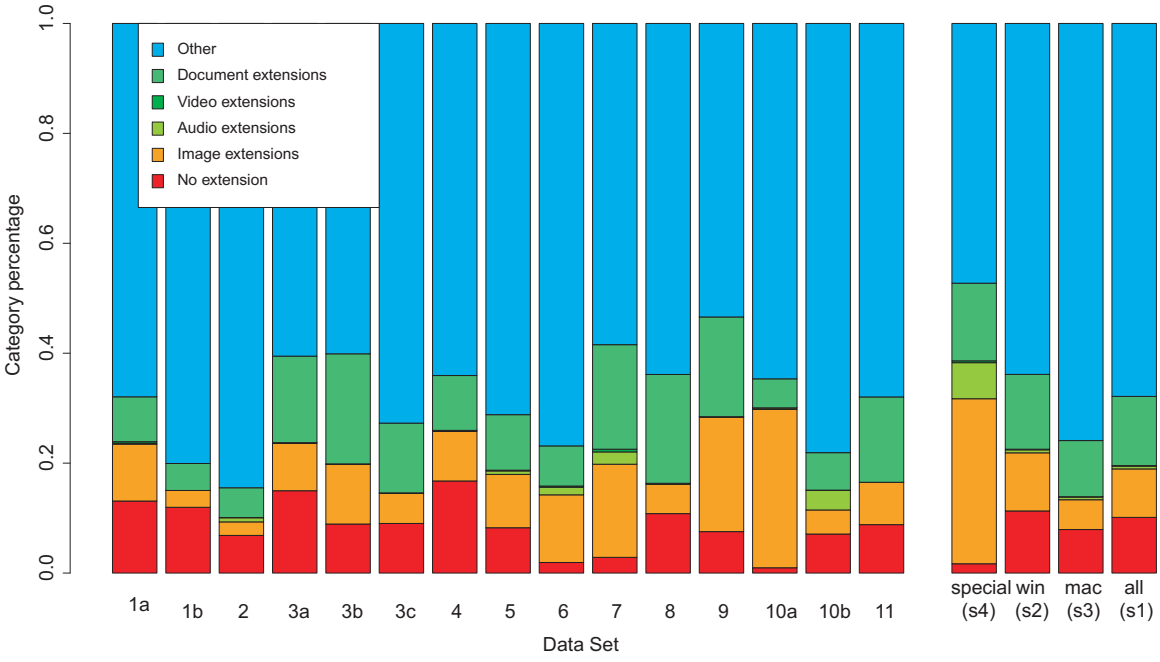
Figure 7.4: Histogram of file sizes. The leftmost gray bar depicts files with zero file size. N = 2,893,686.

Note that the theoretical collision probability is a more intuitive measure here than the normalized Shannon entropy. Consider, for example, the *FO type* feature above. Data set 8 (which is rather non-typical) contains nearly as much folders as files and thus the normalized Shannon entropy is near to 1, indicating a well-suited feature at first sight. However, the FO type feature is apparently not a good feature for this purpose as discussed above. As the normalized Shannon entropy is not sensitive to the number of value clusters in comparison to the total number of considered FOs, it is not well-suited for the comparison of features with high divergence in this measure. Therefore we considered only absolute Shannon entropies and theoretical collision probabilities in our analysis. The results of this analysis are summarized in the table in Figure 7.9c, Shannon entropies and collision probabilities are further shown in the box diagrams in Figure 7.9.

7.6 Discussion

It can be concluded from Figure 7.9 that three features (file name, file size and last modification date) are well-suited for the identification of FOs by feature value comparison due to the high disorder of their values in real-world data. Their low (< 1%) collision probabilities make it very unlikely to find two FOs sharing the same values for these features. Especially the file name feature stands out as the most unique feature of all. On average, we found $44.0\% \cdot 68.5\% = 30.1\%$ unique file names in our data sets. [HDPM08] report on nearly 70% of their files/folders having unique names on average. Such high percentages are plausible considering that file names are on average 13 characters long in our data set, resulting in a very large number of possible names.

Somewhat surprisingly, file size and last modification dates are much less unique (10.9% and 16.8% respectively). We believe this can be contributed to fixed sizes of application data files on the one hand and bulk file modification on the other hand. It has to be considered that these latter two features show higher dynamics in their values when compared to the file name feature: they change whenever a FO is updated



(a)

Category	Extensions
Image	'gif', 'jpg', 'jpeg', 'jp2', 'png', 'tiff', 'bmp', 'svg'
Audio	'wav', 'wma', 'mp3', 'mpeg3', 'ogg', 'aac', 'aif', 'ra', 'mpa', 'oga', 'spx'
Video	'avi', 'mov', 'mpg', 'mp4', 'rm', 'swf', 'vob', 'wmv'
Document	'txt', 'rtf', 'docx', 'doc', 'odt', 'pdf', 'ps', 'eps', 'xls', 'ods', 'xlsx', 'htm', 'html' , 'ppt', 'pptx', 'odc', 'odf', 'xml'

(b)

Figure 7.5: File categories. Figure 7.5a shows the fractions of files (no folders) per data set that were assigned to the categories *Image*, *Audio*, *Video* and *Document*. Additionally, the fractions of files without an extension are shown. The file extensions in the table in Figure 7.5b were used to assign files to categories. The four rightmost data sets in Figure 7.5a correspond to the merged data sets described in Figure 7.1c.

respectively when its content changes[2] whereas file rename events are much less frequent. Nevertheless, both features are well-suited to be incorporated into a feature vector in our proposed method.

We cannot conclude much regarding the dynamics of the file size feature from our data. However, Agrawal et al. report an annual growth rate of the measured mean file size of 15% [ABDL07]. This might support our hypothesis that there is tendency for files to rather grow than shrink. [BBK92] also report that file sizes tend to increase over time. We contribute further evidence supporting this claim by a small timeline experiment that we have conducted. This experiment is presented later in Section 8.4.4 as it makes use of the algorithm presented in the next chapter of this thesis.

Only a small fraction (about 2%) of the FOs in our data sets was recently (less than a week ago) modified. The vast majority (about 87%) remained untouched for more than a month before the scan date. Similar

[2]The last modification date might even change without any changes to the actual file content (e.g., by using the Unix tool *touch*).

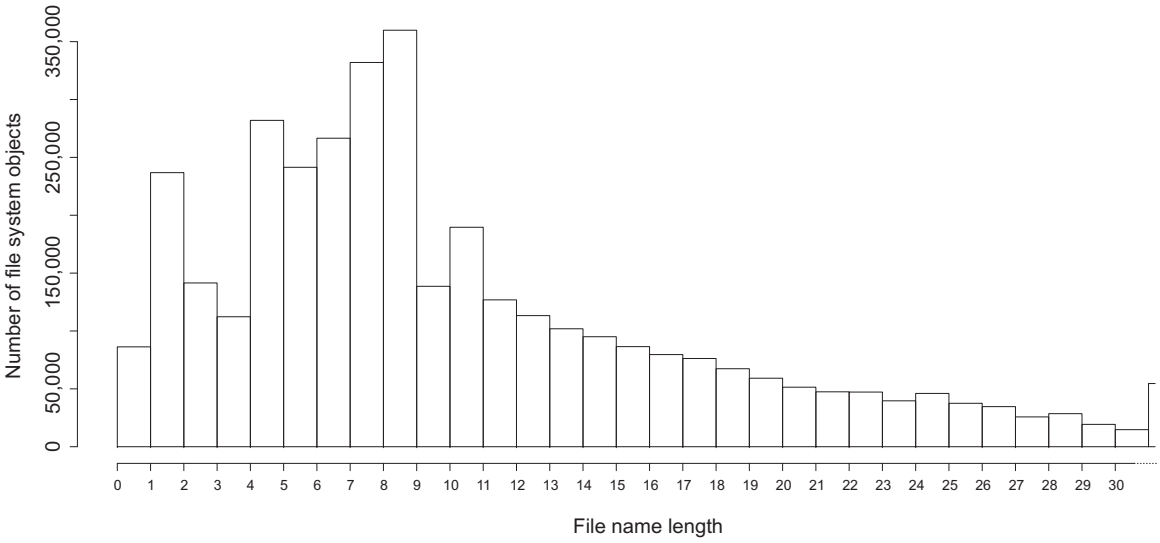


Figure 7.6: Histogram of the file name lengths (file and folder names, no extensions). Note that only lengths smaller or equal to 30 characters are plotted in this diagram. Only 7% of the names in our data set were longer than 30 characters.

results are published also by Hicks et al. who report that "An engineer will modify 17 files from 8 different directories in a day, 50 files from 22 directories in a week and 1500 files from 295 directories in a year." [HDPM08]. Although we do not consider these numbers as generalizable, they give an idea about the magnitude of expected modified FOs per day.

Whenever one of the considered file features (Table 7.1) changes due to some event, this leads also to an update of the last modification date feature of a FO. This means that from an unchanged modification date one may conclude that the whole feature vector of such a FO remained unchanged[3]. Further, the last modification date feature of a FO can only grow over time (not considering the possibilities to modify this date with special tools). This means that for a predecessor/successor pair, there must be a partial order of this feature's values.

The number of file extensions was quite variable throughout our data sets. However, most of them are unique. File extensions should still be considered as a good feature for file identification as they are not expected to change often.

The FO type feature does not qualify for feature vector comparison. It is, however, the most stable feature of all: a file system object cannot change its type. Thus, it is well-qualified for a plausibility test that efficiently rejects all predecessor/successor pairs with differing FO type feature value. For the other boolean features (canRead, canWrite, canExecute, hidden) nothing is known about their dynamics. It is, however, unlikely that these features change often for FOs, thus they could also be used in a feature vector.

The number of FOs per folder seems to be a valuable feature for the comparison of folders. Although we found a high maximum value for this feature, its low median and mean values and its distribution show that most folders contain only a few (0-4) FOs. One possible implication of this could be that when comparing a pair of folder representations it might be computationally feasible to actually compare their contained FOs as well (e.g., by file name).

[3]Disregarding that last modification timestamps have limited granularity depending on file system and operating system.

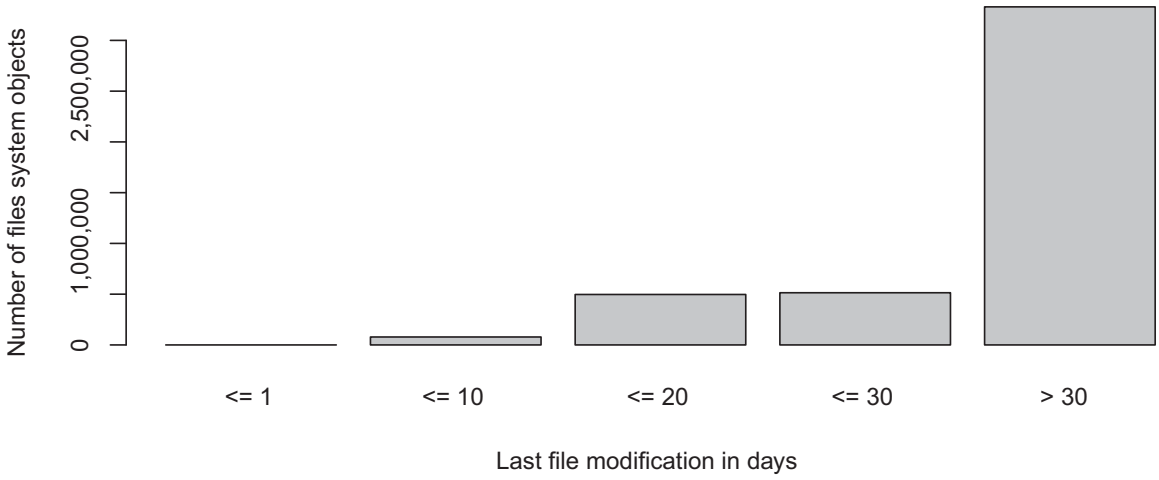


Figure 7.7: Distribution of last modification dates in the data set *s1*.

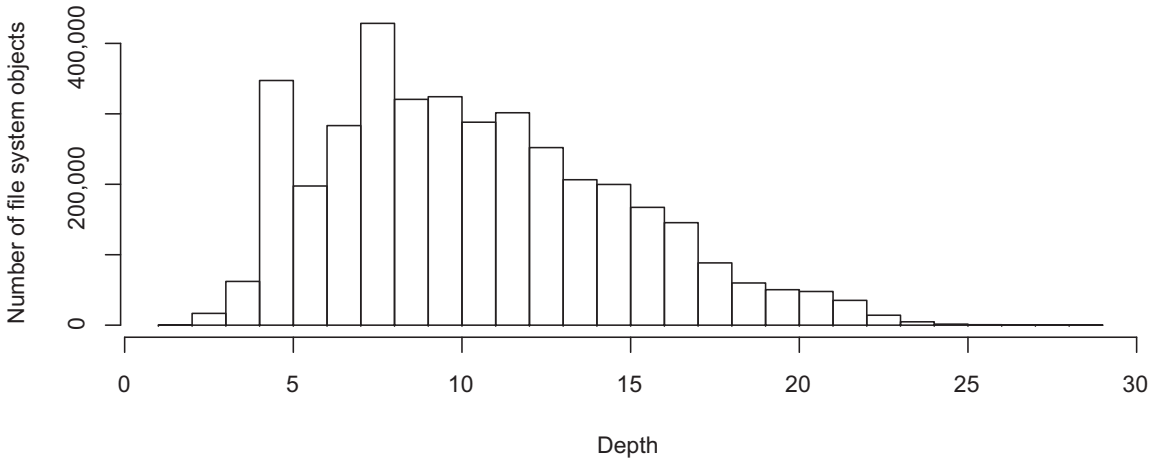


Figure 7.8: Histogram of the depth feature of file system objects (i.e., their depth in the file system hierarchy). N = 3,844,362.

The tree depth feature, however, seems less suited for the discussed purposes as it is likely to change often in move operations when FOs are moved up/down the hierarchy. Further, the discussed distribution of its values makes it quite probable that two unrelated FOs share a common value for this feature. Additionally, calculating this feature for a large number of FOs might be quite costly as it requires to, e.g., parse the file URI which might contain up to 29 levels in our data.

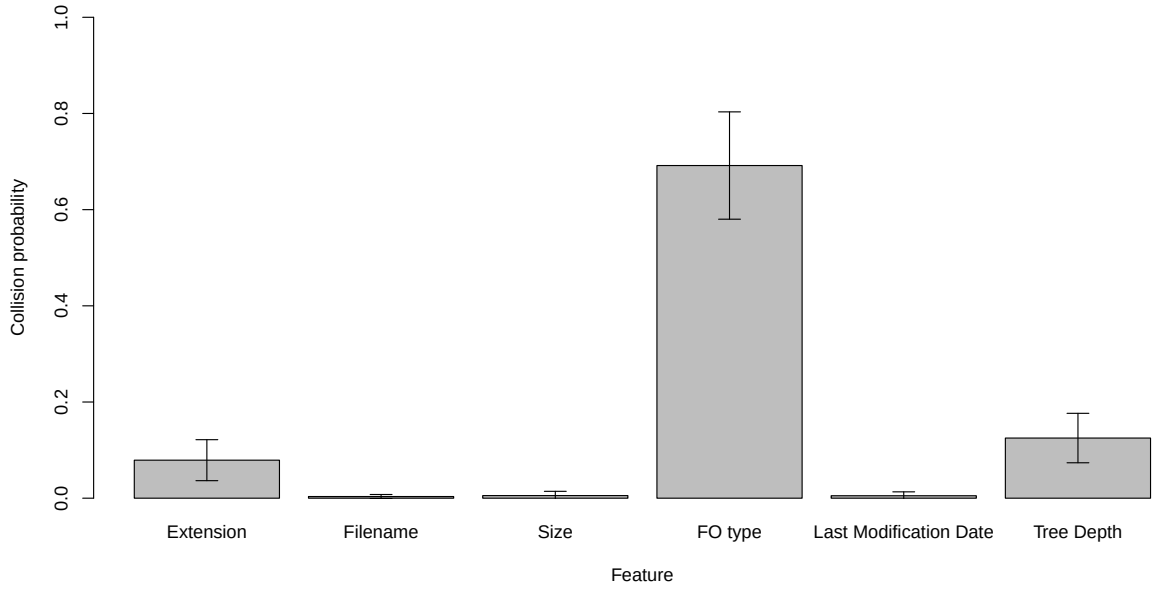
7.7 Conclusions

In this section we reported on a small-scaled study of file feature value distributions. Where possible, we compared our results with related work. Other than most related studies we had access to the actual file paths/names of the scanned FOs. By this, we were able to calculate collision probabilities and Shannon entropies for these and several other features.

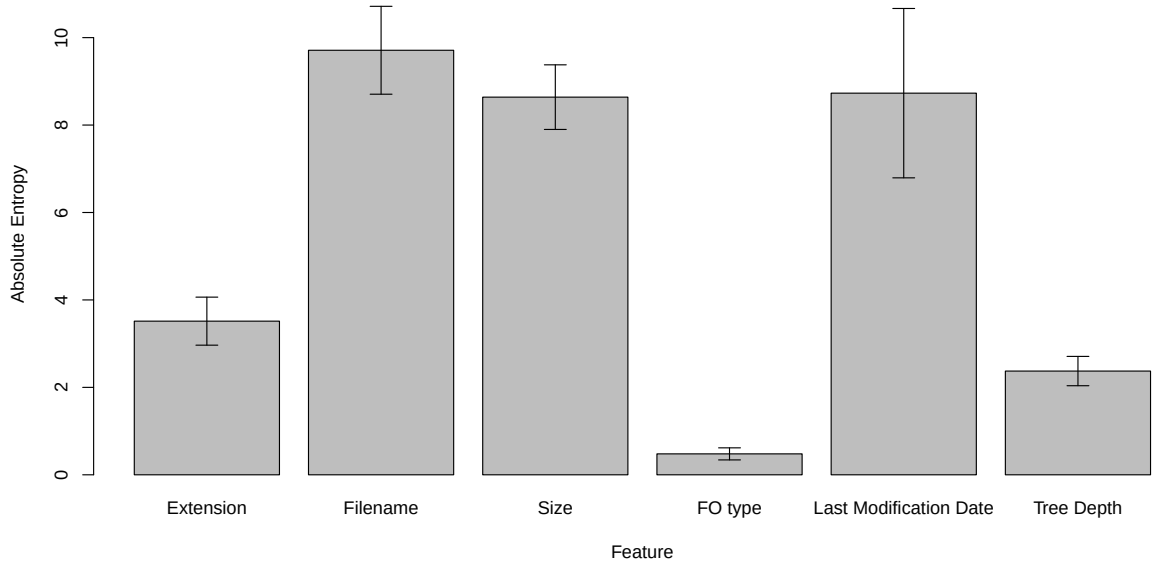
Our study shows that not all considered features are equally applicable for our proposed method of file identification by feature comparison. While some of them seem well-suited to be incorporated into a feature vector (e.g., file name, size, last modification date), others are useful for plausibility checks (e.g., FO type) or applicable only to a particular FO type (e.g., FO/folder for folders, MIME type for files).

Overall, we conclude that our proposed method for file system object identification based on low-level file features is feasible as the probability to find two files sharing the same set of feature values on a volume is very small. Further, large fractions of a file system are untouched for longer time periods (weeks) which means that only small fractions of a volume have to be considered when solving the move detection problem.

We consider the extreme values and the studied value distributions as the fact basis for the design of similarity functions for the respective features. Further, these data enabled us to estimate storage and computation costs for storing and comparing these features. Other results, such as the average file name length, provide valuable insights, e.g., in this case for the GUI design of a system that displays file names.



(a) File feature collision probabilities. Mean values ± standard deviation.



(b) Absolute file feature entropies. Mean values ± standard deviation.

Feature	Collision probability	Entropy $\hat{H}$	Avg. distinct values	Avg. unique values
Extension (files only)	7.90% (σ : 4.26%)	3.51 (σ : 0.55)	0.9% (σ : 0.6%)	40.1% (σ : 14.2%)
Filename	0.34% (σ : 0.42%)	9.71 (σ : 1.01)	44.0% (σ : 15.2%)	68.5% (σ : 10.2%)
Size (files only)	0.54% (σ : 0.89%)	8.64 (σ : 0.74)	21.3% (σ : 8.9%)	51.3% (σ : 12.1%)
Last modification date	0.51% (σ : 0.81%)	8.73 (σ : 1.94)	27.9% (σ : 19.2%)	60.3% (σ : 10.0%)
Treedepth	12.51% (σ : 5.14%)	2.37 (σ : 0.34)	21 (σ : 5)	1 (σ : 0)
FO type	69.17% (σ : 11.16%)	0.48 (σ : 0.14)	2 (σ : 0)	0 (σ : 0)

(c) Statistics of file features ($n_{FOs} = 4,038,253$, $n_{files} = 3,056,065$). Collision probability is the probability to draw two file system objects randomly that share the same feature value (see text). Shannon entropy is an estimate of the randomness of the respective value distribution. Distinct values are given as percentage of the considered numbers of FOs for the upper four features and as absolute value for the lower two. Unique values correspond to feature values that uniquely identify a single FO. Unique values are given as percent of the distinct values for the upper four features. For example, the number of distinct extensions per data set is on average 0.9% of the number of FOs in this data set. Of these distinct extensions, 40% are assigned only to one single FO. Values are averaged over the 15 datasets shown in the table in Figure 7.1a.

Figure 7.9: Resulting statistics of low-level file features.

Chapter 8

Gorm – a heuristic, user-space method for event detection in the file system

8.1 Introduction

In Chapter 6 we presented a framework for event detection in Linked Data sources and discussed how it can be used to provide stable links from local metadata records to resources in such remote data sets. In this chapter, we now focus on stable links from metadata records to local file system objects as required by our metadata model (cf. Figures 4.1b and 4.10). Again, we base our strategy on the accurate detection of events that take place in the respective data space, the file system. A metadata storage can then react on these events accordingly for link integrity preservation.

Several tools for the detection of low-level file system events based on file system APIs exist and we discuss some of them in Section 8.5. However, none of these tools do support the detection of file or folder *move* events. Missing this kind of events means that moving a file system object might lead to the loss of its associated external metadata record, which is clearly not desirable.

Based on the study presented in the previous chapter as well as on our experience with DSNotify we have developed a specialized algorithm for the heuristic detection of file system events that may also detect such move events. This algorithm is able to preserve the integrity of a mapping between (unstable) file URIs pointing to local file system objects and stable, globally unique identifiers and by this constitutes a major building block for the implementation of our proposed metadata model.

We start this chapter by specializing our introduced definitions for the broken link problem in this regard and by formally describing the problem of detecting file move events.

8.2 The file move detection problem

In the following, we describe the move detection problem for file system objects. We use the term *item representation* to describe some arbitrary data structure that is used to describe a file system object. An example for such a representation is a feature vector holding metadata about a file. Note that this term is different from the actual *data representation* which is the file data itself as discussed before.

Let $\mathbb{F}$ be the set of file system objects in a local system, $\mathcal{I}$ be the set of item representations and $\mathcal{F}$ be the set of file system object locations (i.e., file paths). Then there exist two time-dependent functions $\phi_t : \mathbb{F} \longrightarrow \mathcal{F}$ and $\delta_t : \mathcal{F} \longrightarrow \mathcal{I}$ that assign a location respectively an item representation to a given file respectively location at a particular time t.

An item representation could, for example, be assigned to a file $f \in \mathbb{F}$ by accessing the data representation of the file at location $\phi_t(f)$ and then extracting a feature vector from there. Further, let $\delta_t(x) = \emptyset$ if no data representation is available at time t at location $x \in \mathcal{F}$. Note the analogy to the definitions in Section 5.3.1, however, in this case we consider only one *data representation* per item which allows us to define an inverse function of δ. To simplify matters, we assume that there exist two inverse functions[1] $\phi_t^{-1} : \mathcal{F} \longrightarrow \mathbb{F}$ and $\delta_t^{-1} : \mathcal{I} \longrightarrow \mathcal{F}$. In other terms, ϕ_t^{-1} returns a file system object for a given location and δ_t^{-1} is a function that returns a single file location for a given item representation at a particular time t.

Now let $\mathcal{R}_t$ be a set of recently removed items at a particular time t (Definition 8.1) and $\mathcal{N}_t$ be a set of recently created items at the same time t (Definition 8.2).

Definition 8.1 (Recently removed items) $\mathcal{R}_t = \{r \in \mathcal{I} \mid l = \delta_{t-\Delta}^{-1}(r) \wedge \delta_{t-\Delta}(l) \neq \emptyset \wedge \delta_t(l) = \emptyset\}$

Definition 8.2 (Recently removed items) $\mathcal{N}_t = \{n \in \mathcal{I} \mid l = \delta_t^{-1}(n) \wedge \delta_{t-\Delta}(l) = \emptyset \wedge \delta_t(l) \neq \emptyset\}$

Given these two sets of recently removed and recently created item representations, the file move detection problem is to compute a (possibly empty) set $\mathcal{M}_t$ of pairs $(r, n) \in \mathcal{R}_t \times \mathcal{N}_t$ that are representations of the same (moved and possibly updated) file system object at a particular point in time t, see Definition 8.3.

Definition 8.3 (Moved items) $\mathcal{M}_t = \{(r, n) \in \mathcal{R}_t \times \mathcal{N}_t \mid \phi_{t-\Delta}^{-1}(\delta_{t-\Delta}^{-1}(r)) = \phi_t^{-1}(\delta_t^{-1}(n)) = f \wedge \phi_{t-\Delta}(f) \neq \phi_t(f)\}$

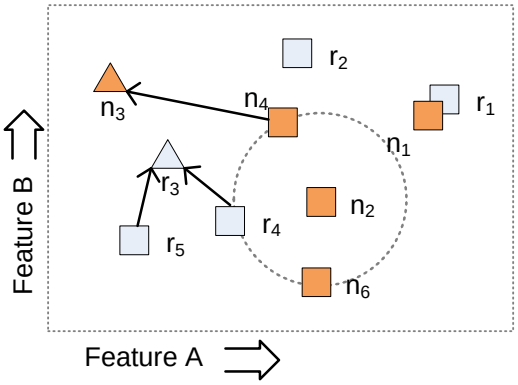
A solution for this problem is obviously trivial if implementations of the inverse functions ϕ^{-1} and δ^{-1} were available. Such implementations would be possible, for example, when item representations contain a unique and stable identifier for the respective file system object. In this section, however, we consider a function that cannot rely on such an identifier and is based on a heuristic feature vector comparison between all available (r, n) pairs at a particular time t for detecting $\mathcal{M}_t$ as good as possible. That is, analogously to the broken link problem formulated in Section 5.3.1, we detect move operations based on some similarity measure applied to item representations of recently removed and recently added file system objects.

A binary classification task? At first sight, the problem to decide for a pair of item representations (r, n) whether they correspond to the same (moved and/or updated) file system object or not might look like a binary classification task. A simple algorithm that uses a binary classification strategy could solve the described file move detection problem by first classifying each pair (r, n), adding all pairs that are classified as being a possible predecessor/successor pairs to the result set and then removing all pairs that share the same r or n as these represent ambiguous decisions (a file can only be moved to one other location when disregarding digital copies for now; further, a file can originate only from one location).

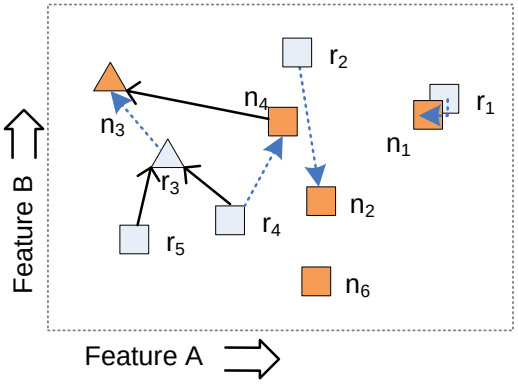
However, considering the described file move detection problem as a simple binary classification task of (r, n) pairs disregards valuable information. In the non-trivial cases (i.e., when there is more than one possible predecessor/successor pair), the move detection problem can be seen as choosing the best predecessor candidates from $\mathcal{R}_t \cup \emptyset$ for all available possible move targets from $\mathcal{N}_t$. Consider Figure 8.1a for a visual representation of this problem. The Figure shows recently removed and recently created items arranged in a 2-dimensional space (i.e., these item representations consist of two features only). The closer two items, the more similar their feature vectors. The true set of (r, n) pairs is in this case
$\mathcal{M}_t = \{(r_1, n_1), (r_2, n_2), (r_3, n_3), (r_4, n_4), (r_5, \emptyset), (\emptyset, n_6)\}$, as shown in Figure 8.1b.

In order to calculate all required distances, each removed item representation has to be compared to all created item representations, which requires $|\mathcal{R}_t| \cdot |\mathcal{N}_t|$ feature vector comparisons. However, many of these pairs can already be excluded beforehand due to some plausibility checks. For example, all pairs with differing type (file/folder) can be excluded as a file cannot turn into a folder and *vice versa*. For example, $n3$ in Figure 8.1a can only be a new item or a successor of item $r3$ as there are no other removed folders in this

[1] Note that $\delta^{-1}(\emptyset)$ is left undefined for now as there is not practical use case that requires such a functionality.



(a) Blue boxes (r_i) correspond to recently removed files, orange boxes (n_i) to recently created ones. Triangles correspond to removed/created folders analogously. Parent/child relationships between folders and files are indicated by solid, black arrows. The items are arranged in a 2-dimensional feature space, the inverse of their Euclidean distance corresponds to their similarity. The dashed circle centered at n_2 represents a minimum similarity threshold. If such a threshold is introduced, only item r_4 would be considered as a possible predecessor of n_2.



(b) Solved move detection problem. Matching removed and created items of a moved file system object are connected by dashed, blue arrows, r_5 corresponds to a removed item, n_6 to a created item.

Figure 8.1: Example for the file move detection problem.

example. Further plausibility checks are discussed in the next section. Such plausibility checks may reduce the effort of calculating a $|\mathcal{R}_t| \times |\mathcal{N}_t|$ similarity matrix considerably.

Further, the file system hierarchy itself can be exploited. Let us reconsider Figure 8.1b. The first true item representation pair (r_1, n_1) is easy to detect due to its high similarity. The second pair (r_2, n_2), however, is much more difficult to detect. First, the similarity between r_2 and n_2 is rather low. Second, n_4 is actually more similar to r_2 than n_2. Third, r_4 is more similar to n_2 than r_2. Thus, it is obvious that the true (r, n) pairs cannot be detected by considering only the item representation similarities in this case. It can, however, be found when also considering the given parent/child relationships between the folders r_3 and n_3 and their contained files r_5, r_4 and n_4. When a detection algorithm takes into consideration that r_3 is a predecessor of n_3 and that r_4 is a child of r_3 then it might conclude that r_4 is an actual predecessor of n_4 which is a child of n_3. Consequently, r_2 would be detected as predecessor of n_2. We conclude that pure binary classification is not a good method to solve the file move detection problem as it disregards explicit (hierarchical) and implicit relationships between the considered items.

Analogously to the described DSNotify system (Section 6), we base our approach for maintaining a stable file/URI mapping on the accurate detection of file system events. Solving the described *file move detection problem* in a timely close fashion to the actual move events can obviously be used to detect these events. The file move detection problem is closely related to the detection of resource move events in Linked Data sources described in Section 6 and by this also to alignment problems like instance matching or similar [CFLM08]. However, several special file system characteristics can be exploited for its solution. This includes the given file system hierarchy, the 100% coverage of low-level file features and various features that can be used in plausibility checks, such as the type (folder or file) of a file system object.

8.2.1 Event detection and stable file URIs

The accurate detection of the file system events in Table 8.1 can be exploited to provide a stable mapping between local file system object paths and globally unique, stable URIs. We can describe such a mapping as

a set of pairs of the local file URI ($\upsilon(f)$) and some globally unique URI (u_g) for this file system object (see Definition 4.8):
$\mathbb{M} = \{(\upsilon(f), u_g) \in \mathbb{U} \times \mathbb{U}\}$.

Such a mapping may require to be updated when the local file system is updated, that is when file system events occur. The required update operations for this mapping are given in Table 8.1 (cf. [SP10]). The detection of create, update and remove events is straightforward as explained in the following section. Thus, the core problem that has to be solved in order to provide a stable File/URI mapping as described in this section is the file move detection problem.

Event	Description	Map Update Operation
$c(f)$	Creation of a file at location f	Mint globally unique URI u_g and add mapping $(\upsilon(f), u_g)$
$r(f)$	Deletion of a file at location f	Remove mapping $(\upsilon(f), \circ)$
$u(f)$	Update of the file at location f	No mapping update required.
$m(f_1, f_2)$	Move of file at f_1 to its new location f_2. Note that the rename operation is a special case of this operation.	Replace the mapping $(\upsilon(f_1), \circ)$ with the mapping $(\upsilon(f_2), \circ)$

Table 8.1: File system events and required updates to a mapping between locally and globally unique URIs. Note that a circle ($\circ$) in a mapping pair stands for an arbitrary URI.

Feature vector index

We need the sets $\mathcal{R}_t$ and $\mathcal{N}_t$ in order to apply a move detection algorithm to possible predecessor/successor pairs. These sets can easily be calculated by comparing the current state of the considered file system with an index of item representations $\mathcal{X}_t \subset \mathcal{I}$ (Equations 8.1 and 8.2). Such an index is a set of item representations (i.e., feature vectors) that represent the last known state of the considered file system objects. The sets $\mathcal{R}_t$ and $\mathcal{N}_t$ are calculated by comparing the indexed item representations with the ones that resulted from a crawl $\mathcal{C}_t \subset \mathcal{I}$ of the file system at time t.

Item representations are compared by their file URI as we can claim that item representations within a local system that share a common URI represent the same file system object. In other words, $\upsilon(f_1) = \upsilon(f_2) \iff f_1 = f_2$.

$$\mathcal{R}_t = \{r \in \mathcal{I} \mid r \in \mathcal{X}_t \wedge r \notin \mathcal{C}_t\}. \tag{8.1}$$

$$\mathcal{N}_t = \{n \in \mathcal{I} \mid n \notin \mathcal{X}_t \wedge r \in \mathcal{C}_t\}. \tag{8.2}$$

Practically, such an index also contains the URI mapping described in Definition 4.8, i.e., it stores file system object locations and assigned, stable URIs. After the sets $\mathcal{R}_t$ and $\mathcal{N}_t$ are calculated, the file move detection algorithm is applied to them. Then, the detected events are reported and the index X is updated accordingly.

8.3 Proposed heuristic approach

We propose a heuristic algorithm that takes the specific characteristics of the considered low-level file system properties into account. Our algorithm is based on pairwise similarity measures between file system object (item) representations. In the following section we describe our proposed feature selection and similarity

measures. We then describe our file move detection algorithm that is based on these similarity measures and discuss our approaches for detecting subtree move operations and for dealing with the unfavorable cubic complexity of our algorithm.

8.3.1 File features and similarity functions

By resorting to the results from our file system study (Chapter 7) as well as some practical/performance considerations, we selected the set of features in Table 8.2a that together constitute an item representation in our system. Table 8.2b summarizes the influence of certain file system operations on the values of these features. It can be seen that there are more and less "stable" features in the selected set.

Feature	**Data type**	*API*	*f*	*d*	**Description**
uri	String	+	+	+	File URI
name	String	+	+	+	File name (including extensions)
extension	String	-	+	+	File extension
size	Long	+	+	-	File size
type	Boolean	+	+	+	Flag that indicates whether a file system object is a folder or a file
lastModified	Long	+	+	+	Last modification date, time in milliseconds since 00:00:00 GMT, January 1, 1970
parent	Folder	+	+	+	Parent folder of this file system object
mime type	String	-	+	-	File mime type
checksum	Integer	-	+	-	16-bit checksum of the file content. Note that our algorithm can be configured to not calculate this feature for performance reasons.

(a) File features that are included in our *item representations*, italic features are optional. The column *API* indicates whether the respective feature values were retrievable via the standard file API or had to be calculated from the other features. The columns *f* and *d* indicate whether this feature is available (+) or not (-) for files (f) respectively folders (d).

Operation	uri	name	extension	size	type	lastModified	parent	mime type	checksum
Touch	○	○	○	○	○	●	○	○	○
Update	○	○	○	◐	○	●	○	○	◐
Move	●	◐	◐	○	○	●	●	◐	○
Rename	●	●	◐	○	○	●	○	◐	○

(b) Influence of file system operations on various file features. Symbols: The value of the respective feature changes (●), does not change (○), possibly changes (◐) when the respective operation is applied to a file. Operations: touch (Unix file touch operation), update (file content update), move (file move, no rename), rename (file rename).

Table 8.2: File features and how they are affected by certain file system operations.

As we do not consider only numerical features, we needed functions that convert values from non-numerical domains into numbers in order to calculate an overall similarity between two such item representations. One way to do this is to consider the distance or similarity between two feature values (of a considered representation pair) in a numerical space. A trivial example for a boolean feature would be to assign a zero distance to two matching feature values and a distance of 1.0 to non-matching values. However, for several considered features (e.g., file name, extension, path) there are many ways how to convert them to numerical (distance or similarity) values. For example, the classical Levensthein (edit) distance[2] applied to

[2]In the following we use *edit distance* and *Levensthein distance* interchangeably if not stated otherwise

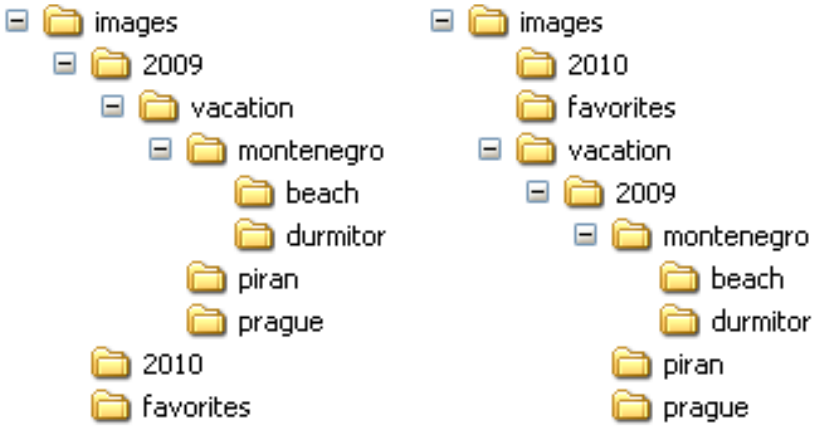
the file names of the "older" and the "newer" item representations is one such measure, the length of their longest common substring (LCS) would be another. These distance/similarity measures have a major influence on the resulting accuracy of any classification algorithm that incorporates them and should be designed with care. Taking the results of our file system study (Chapter 7) into account, we specified the following similarity functions for comparing the features in Table 8.2a with each other.

URI and file path. File URIs can be used to derive the local file system object path (simply by the inverse function $\upsilon^{-1} : \mathbb{U} \longrightarrow \mathcal{F}$). When we consider file move events in a file system hierarchy, it seems likely that most files are moved only a few levels up/down in the hierarchy. In other words, the percentage of common path elements of two item representations of the same file system object is expected to be high in most cases. In the case of a rename operation, for example, all path elements except for the file name itself are equal. Consequently, we aim at a comparison function that assigns higher similarity values if two compared file paths differ in few path elements and lower similarities if they differ in many path elements. Further, this function should not penalize simple hierarchy reorganizations as shown in Figure 8.2a too much. Note that in such a case a simple edit distance between the file paths/URIs is not a well-suited measure. For example, the edit distance between the file paths describing the "beach" folder in Figure 8.2a is 10.0 as ten characters were removed/added (five operations for removing the characters "2009/" and another five for inserting them). However, this measure depends on the length of the names of the reorganized folders which is not what we want. Therefore we decided to compare file paths based on the Levensthein (edit) distance of their path components. That is, we first calculate the minimum set of edit operations that was required to rearrange one file path into another. In the example in Figure 8.2a, this *component edit distance* would be 2.0 (one operation to remove the component "2009" and one for inserting it below the folder "vacation"). This component edit distance can simply be calculated by treating each path component as a single letter in an extended alphabet and then calculating the standard Levensthein distance for these strings. We propose the simple formula in Definition 8.4 for converting this edit distance into a normalized similarity measure, this function is plotted in Figure 8.2b. Note that the influence of this similarity on the overall similarity is weakened by limiting its values to the interval [0.5,1].

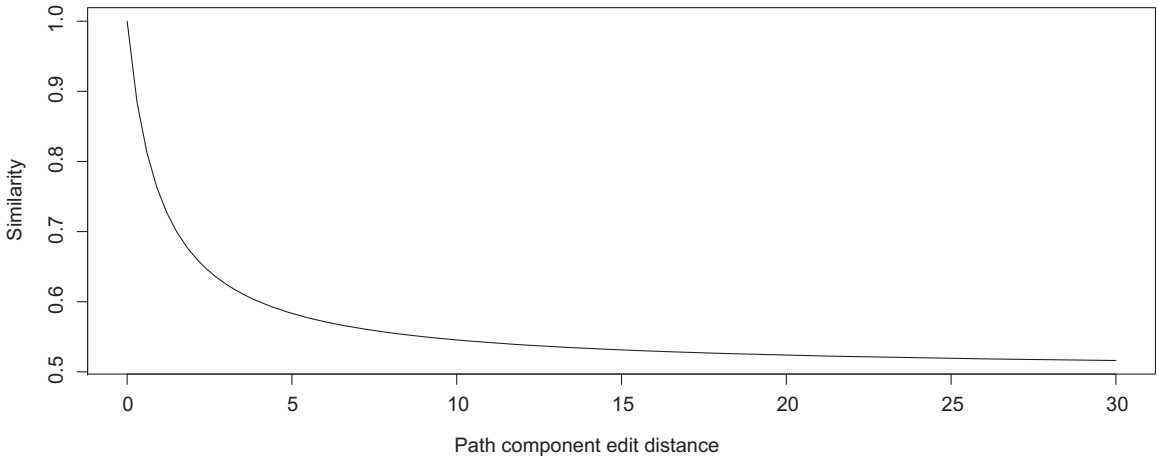
Definition 8.4 (Path component similarity)

$$\sigma_{path}(r_{path}, n_{path}) = 0.5 + \frac{1}{(1 + component\_edit\_distance(r_{path}, n_{path})) \cdot 2}$$

Name and extension. As our study showed that file names are particularly good features for a direct feature comparison, we designed the similarity function for this feature with special care. Such a similarity function should be good at modeling how users actually rename files. The first, trivial case we considered is that users rename files simply for correcting upper/lowercase letters, for example, by renaming "mydocument.PDF" to "MyDocument.pdf". As a consequence, we propose a high similarity for file names / extensions that differ only in the case of their characters. We further considered the case that users include multiple "components" such as categories, dates and abbreviations in one single file name. An example would be the filename "2009_durmitor_rafting_001.jpeg" which includes information about a certain date, place and activity. We argue that, similar to the comparison of file paths above, the edit distance is not a well-suited measure for rename operations. Imagine, for example, that the user imported the above-mentioned JPEG image from her camera. This import process created a file, say, "rafting_001.jpeg" on the user's disc. She now renames this file to "2009_durmitor_rafting_001.jpeg" which would give an edit distance of 14 for the added characters. However, with 14 edit operations a huge number of other names could be created. It would thus be considered rather unlikely that these two names belong to the same file. In our study of real file data that is presented in Chapter 7, we found a mean file name length of 13 which means that the edit distance to the majority of these file names is also 13 which is the number of edit operations required to substitute/remove all characters from the original file name. To deal with this problem, we consider the Levensthein distance only for small changes



(a) Folder hierarchy reorganization. The path component edit distance between the file paths for the "beach" folder is 2.0 as explained in the text.



(b) File path similarity.

Figure 8.2: Component similarity.

(edit distance ≤ 4), as done, e.g., when correcting spelling errors, and use the length of the longest common subsequence (LCS) between two file names for broader changes. The length of the LCS is the maximum number of common subsequent characters between two strings. In the above-mentioned example, the LCS between the file names "rafting_001" and "2009_durmitor_rafting_001" is of length 11. We normalize this length by dividing by the length of the longer of the two names (in this case: 25). The overall formula for calculating the similarity between two file names is given in Definition 8.5[3] and plotted in Figure 8.3. Note that we propose the same measure for file extensions ($\sigma_{extension}$).

[3]In Definition 8.5, $len(x)$ refers to the length of a string x and $LCS(n1, n2)$ refers to the LCS-length between two strings $n1$ and $n2$

Definition 8.5 (Name similarity)

$$\sigma_{name}(r_{name}, n_{name}) = \begin{cases} 1.00, & \textit{if names are equal} \\ 0.95, & \textit{if names are equal except for their case} \\ 0.95 - editdistance(r_{name}, n_{name}) \cdot 0.05 & \textit{if } editdistance(r_{name}, n_{name}) \leq 4 \\ 0.70 \cdot \dfrac{LCS(r_{name}, n_{name})}{\max(len(r_{name}), len(n_{name}))} & \textit{otherwise} \end{cases}$$

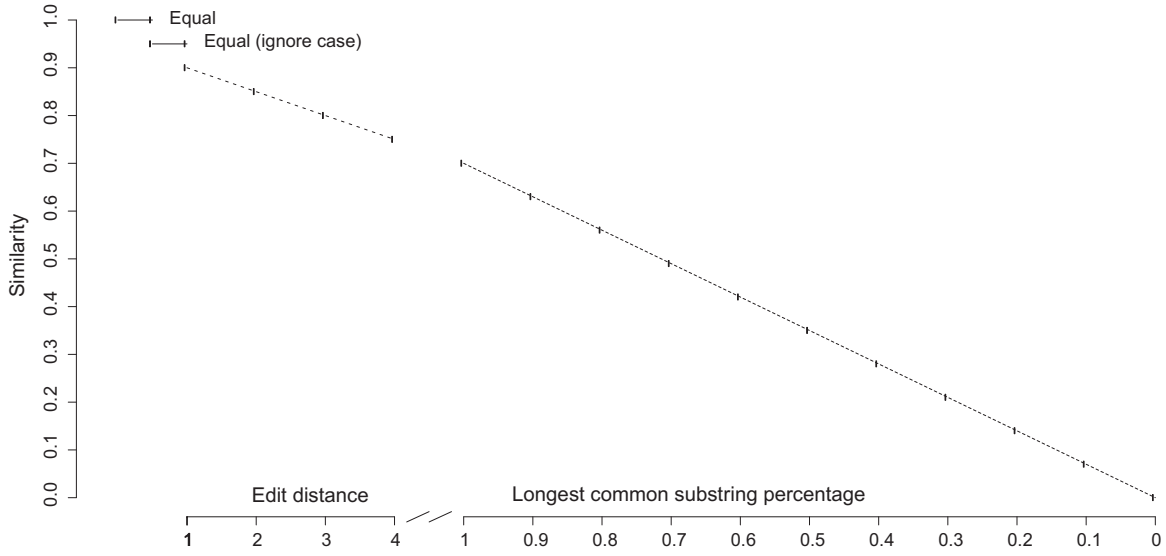


Figure 8.3: File name and file extension similarity.

File size. The file size feature is defined only for file objects (not folders) and our observations of its value distributions concluded in our working hypothesis that file sizes tend to grow, rather than shrink, slowly over time. We present a small experiment that supports this claim in Section 8.4.4.

One usage scenario we considered was that a user starts to work on a text document and iteratively adds, changes and deletes text in this document until a certain version of it is finished. In this scenario, the file size would mostly grow while size shrinkage would be rather rare. Further, the file would not grow for several mega bytes per update but rather slowly[4].

Accordingly, we propose the formula in Definition 8.6 for determining the file size similarity between two files. Note that this function first normalizes the file size to kilobytes, thus the function is by intention not sensible to smaller file size changes. Further note that, as this function is asymmetrical, the order of its parameters matter: first, the assumed predecessor size (r_{size}) and then the assumed successor size (n_{size}) should be passed. This similarity function is plotted in Figure 8.4

[4]One scenario for a large change in file size might be, for example, a log file that constantly grows until its maximum size is reached when it is "rolled over" which usually means that it is renamed and a new, empty file is created with the log file's original name.

Definition 8.6 (File size similarity) $\sigma_{size}(r_{size}, n_{size}) = \begin{cases} 1 & , \lfloor \frac{n_{size}-r_{size}}{1000} \rfloor = 0 \\ \frac{1}{\lfloor \frac{n_{size}-r_{size}}{1000} \rfloor} & , r_{size} \leq n_{size} \\ -\frac{1}{\lfloor \frac{n_{size}-r_{size}}{1000} \rfloor \cdot 2} & , r_{size} > n_{size} \end{cases}$

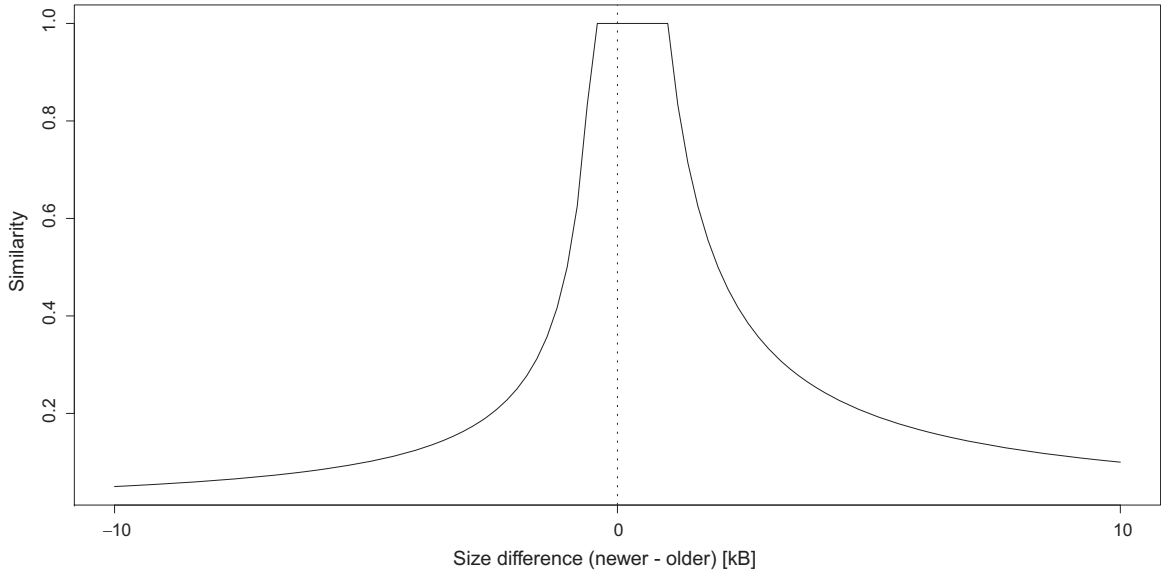


Figure 8.4: File size similarity. Note the asymmetrical shape of the function that is founded in our hypothesis that files tend to grow rather than shrink.

Type. The file system object type is used for plausibility checks that may greatly speed up the similarity calculations. If two file system objects differ in their type, a general similarity of 0 between these two file system objects is assumed, see Definition 8.7.

Definition 8.7 (Type similarity) $\sigma_{type}(r_{type}, n_{type}) = \begin{cases} 0, & r_{type} \neq n_{type} \\ 1, & r_{type} = n_{type} \end{cases}$

Last modification date. The last modification date is also used for a plausibility check in our model. Remember that we assume a pair (r, n) as an input for our similarity function where r is a recently removed and n is a recently added file. If r is a predecessor of n, its last modification date has to be *before or equal* the last modification date of n. Thus, if the last modification date of r is after the last modification date of n, we can conclude not only a zero similarity between r and n regarding this particular feature (Definition 8.8) but can further immediately reject this item pair as possible predecessor/successor pair.

Definition 8.8 (Last modification date similarity) $\sigma_{lastmod}(r_{moddate}, n_{moddate}) = \begin{cases} 0, & r_{moddate} > n_{moddate} \\ 1, & r_{moddate} \leq n_{moddate} \end{cases}$

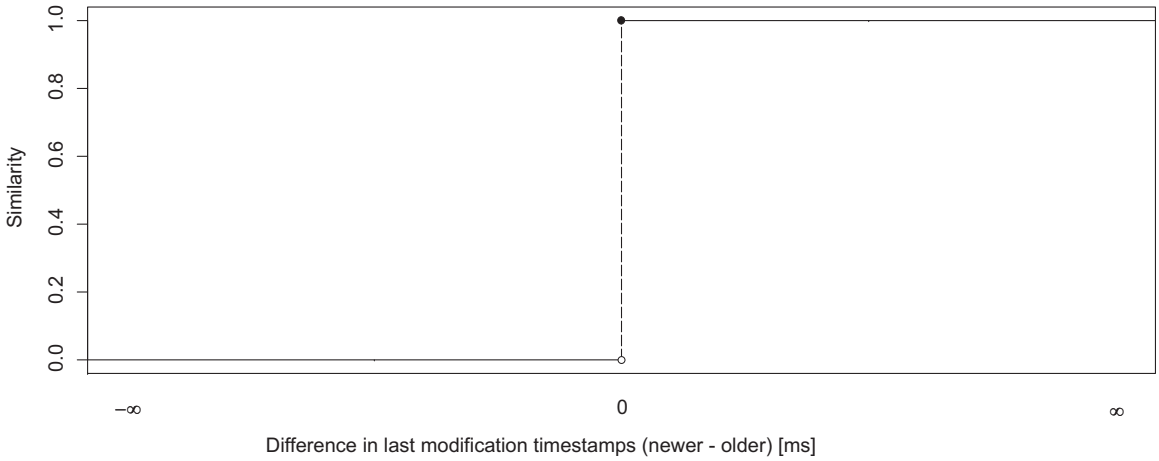


Figure 8.5: Last modification date similarity.

Mime type. The mime type of files is assigned using a standard API method[5]. The mime-type RFC defines "five discrete top-level media types" [FB96]. These *primary types* ("text", "image", "audio", "video", "application") are not expected to change throughout a file's life. We consider, however, scenarios where the secondary mime type changes. For example: when the file extension of a file is changed from "txt" to "html", the mime type changes from "text/plain" to "text/html". We propose the simple formula in Definition 8.9 for calculating the mime type similarity of two files.

Definition 8.9 (Mime type similarity) $\sigma_{mime}(r_{mime}, n_{mime}) = \begin{cases} 0 & \textit{primary mime types differ} \\ 0.5 & \textit{secondary mime types differ} \\ 1 & \textit{mime types are equal} \end{cases}$

Parent/child relationships. Some of the features in Table 8.2a are not applicable to folders as these file system objects lack a data representation (and thus also a size, a mime type or a checksum feature). The unavailability of these features would result in a more error prone move detection for folders as it would have to resort to a reduced feature set. To compensate for this, we propose the introduction of a special feature for folders that takes the hierarchical parent/child relationships into consideration. This feature can easily be calculated from the *parent* feature (cf. Table 8.2a) and represents the (possibly empty) set of children of a folder. Our proposed similarity function (Definition 8.10) then simply considers the percentage of common children of two folders as their similarity with regard to this feature.

Reconsidering the folder rearrangement shown in Figure 8.2a it can be seen that the folders "vacation", "2009" and "images" have differing child sets and by that similarities $\neq 1$. The two switched folders "vacation" and "2009" have different child folders in both scenarios. The *children-similarity* between these folders depends on the number of their contained files. The "images" folder shares two of three sub-folders in both hierarchies. Even without any contained files, the *children-similarity* between these folders will thus be $\frac{2}{3}$.

The question remains how to actually decide how many files are shared by both child-sets. In principle, we would have to apply our move detection algorithm to these sets in order to decide what file system objects are predecessors/successor pairs and divide the number of these pairs by the number of files contained in the removed folder. However, for performance reasons, we propose to simply compare the child sets by the

[5]We use the standard Java class `javax.activation.MimetypesFileTypeMap` for assigning mime-types to files.

names of the contained files. Thus, the sets $r_{children}$ and $n_{children}$ in Definition 8.10 refer to the sets of names of the children of the respective folders.

Definition 8.10 (Children similarity) $\sigma_{children}(r_{children}, n_{children}) = \dfrac{|r_{children} \cap n_{children}|}{|r_{children}|}$

Checksum. The checksum of a file is a hash sum of its byte content. It is the only content-derived feature we consider (as we determine the mime-type based on a mapping from a file's extension). When we compare two item representations that share a common checksum, we can conclude that (i) these items represent the same file, or (ii) these items represent copies of a file, or (iii) a hash collision took place.

Note that the calculation of the checksum feature might be an expensive operation as it usually requires I/O operations for loading the whole file content into memory. As this might be inappropriate in some scenarios, we consider this feature as optional in our feature vectors. Our implementation can be configured whether this feature is calculated (and included in the feature vector comparison) or not. When the checksum is considered, we use it as described in Definition 8.11 and plotted in Figure 8.6.

Definition 8.11 (Checksum similarity) $\sigma_{checksum}(r_{checksum}, n_{checksum}) = \begin{cases} 0, & r_{checksum} \neq n_{checksum} \\ 1, & r_{checksum} = n_{checksum} \end{cases}$

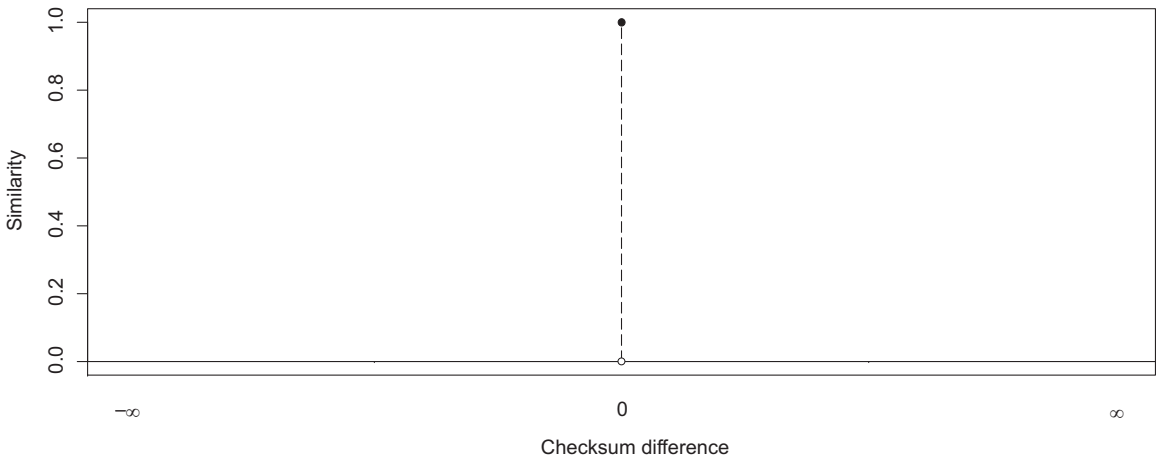


Figure 8.6: Checksum similarity.

Overall similarity calculation

The overall similarity calculation is summarized in Algorithm 2. After the two described plausibility checks, the algorithm calculates the sum of the various applicable features for file or folder (r, n) pairs. This sum is then normalized to the interval $[0, 1]$ and used as a measure for the similarity between the predecessor item representation r and its possible successor item representation n. Note that our actual implementation allows to exclude particular features (e.g., the checksum feature) from the calculation for performance reasons (not shown in the listing of the algorithm).

Algorithm 2: File similarity calculation algorithm.

Data: pair (r, n)
Result: similarity σ

begin
  if $(\sigma_{type}(r_{type}, n_{type}) = 0)$ **then**
    return 0.0;
  end
  if $(\sigma_{lastmod}(r_{moddate}, n_{moddate}) = 0)$ **then**
    return 0.0;
  end
  $\sigma_{sum} \longleftarrow 0.0$;
  $weight \longleftarrow 0.0$;
  if $(r_{type} = FILE)$ **then**
    $\sigma_{sum} \longleftarrow \sigma_{path}(r_{path}, n_{path}) + \sigma_{name}(r_{name}, n_{name}) + \sigma_{extension}(r_{extension}, n_{extension}) + \sigma_{size}(r_{size}, n_{size}) + \sigma_{mime}(r_{mime}, n_{mime}) + \sigma_{checksum}(r_{checksum}, n_{checksum})$;
    $weight \longleftarrow 6.0$
  end
  else
    $\sigma_{sum} \longleftarrow \sigma_{path}(r_{path}, n_{path}) + \sigma_{name}(r_{name}, n_{name}) + \sigma_{extension}(r_{extension}, n_{extension}) + \sigma_{children}(r_{children}, n_{children})$;
    $weight \longleftarrow 4.0$;
  end
  $\sigma \longleftarrow \frac{\sigma_{sum}}{weight}$;
  return σ;
end

Algorithm 3: File move detection algorithm.

Data: Sets of recently removed $(\mathcal{R}_t)$ and recently created $(\mathcal{N}_t)$ item representations; thresholds lt, ut, dt;
Result: Set of moved files $(\mathcal{M}_t)$

begin
  $\mathcal{M}_t \longleftarrow \emptyset$;
  $\mathbb{SIM} \longleftarrow [s_{ij}]_{|\mathcal{N}_t| \times |\mathcal{R}_t|}$;
  $s_{ij} \longleftarrow \begin{cases} \sigma(r_j, n_i), & \sigma(r_j, n_i) > lt \\ \emptyset, & \sigma(r_j, n_i) \leq lt \end{cases}$
  repeat
    $(a, x) \longleftarrow (i, j) \,|\, s_{ij} = max(\mathbb{SIM})$;
    $(b, x) \longleftarrow (i, x) \,|\, s_{ix} = max(\mathbb{SIM}[\cdot, x] \setminus (a, x))$;
    if $s_{ax} \geq ut$ **then**
      if $s_{ax} - s_{bx} > dt$ **then**
        $\mathcal{M}_t \longleftarrow \mathcal{M}_t + (r_x, n_a)$;
        $s_{aj} \longleftarrow \emptyset \; \forall \; 0 \leq j < |\mathcal{R}_t|$;
      end
      $s_{ix} \longleftarrow \emptyset \; \forall \; 0 \leq i < |\mathcal{N}_t|$;
    end
  until $s_{ax} < ut$;
  return $\mathcal{M}_t$;
end

8.3.2 File move detection

Our proposed file move detection method, described in Algorithm 3, is based on the introduced similarity Algorithm 2. First, a $|\mathcal{N}_t| \times |\mathcal{R}_t|$ similarity matrix is created by applying Algorithm 2 to each (r, n) pair (cf. Figure 8.7b). The algorithm sets matrix cells to *null* if the calculated similarity for a (r, n) pair is below a configured *lower threshold* (*lt*). This lower threshold excludes pairs with low similarity from the move detection algorithm beforehand. In Figure 8.7 it is depicted by the outer dashed circles.

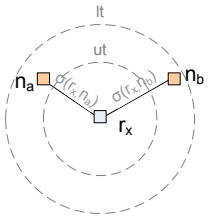
In the next step, the entry with the highest similarity value $\sigma(r_x, n_a)$ is searched in the matrix. The respective pair represents the most similar predecessor/successor pair (r_x, n_a) (cf. Figure 8.7a). Next, the second-highest similarity value for the respective removed item is searched in the respective column of the matrix. This pair (r_x, n_b) contains a newly created item that is the second-most similar to the considered removed item r_x.

The proposed algorithm now accepts or rejects the considered pair (r_x, n_a) as predecessor/successor pair of the same real file system object (i.e., as result of a file move event) based on these two similarity values $\sigma(r_x, n_a)$ and $\sigma(r_x, n_b)$ as well as on two thresholds, an *upper threshold* (*ut*) and a *difference threshold* (*dt*). Three possible scenarios depicted in Figure 8.7 explain the effects of these thresholds in more detail.

Scenario 1: items not similar enough. Figures 8.7a and 8.7b show a scenario where the maximum similarity value is smaller than the upper threshold *ut*. In this case, our algorithm rejects the pair (r_x, n_a) as its similarity is not high enough. Further, the algorithm terminates as all other entries in the matrix are smaller than this value.

Scenario 2: multiple possible predecessor candidates. Figures 8.7c and 8.7d show a scenario where the maximum similarity value is greater than the upper threshold *ut* but its difference to the second-highest similarity value is smaller or equal to the difference threshold ($\sigma(r_x, n_a) - \sigma(r_x, n_b) \leq dt$). In this case, our algorithm rejects the pair (r_x, n_a) as it cannot decide unambiguously between the two successor candidates for r_x. The algorithm fills the respective column of the similarity matrix with null values (cf. Figure 8.7d) and continues by searching for the new maximum values in the matrix. One effect of the difference threshold is that it reduces the number of false positive pairs while at the same time increasing the number of false negatives. By varying this threshold it is thus possible to balance these measures to some extent.

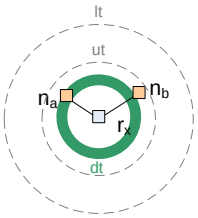
Scenario 3: move detection. Figures 8.7e and 8.7f finally show the scenario when our algorithm detects a move event. The two considered items r_x and n_a are similar enough (above upper threshold) and the second most similar created item n_b is less similar than the difference threshold *dt* ($\sigma(r_x, n_a) > ut \wedge \sigma(r_x, n_a) - \sigma(r_x, n_b) > dt$). In this case, a move event is issued and the respective column and row of the matrix are filled with null values (cf. Figure 8.7f) to avoid that these items can be part of another (r, n) pair. The algorithm continues with the next iteration until the exit condition (scenario 1) is reached.



(a) Move detection scenario 1

	r_1	r_2	...	r_x	...	r_m
n_1						
⋮						
n_a				$\sigma(r_x, n_a)$		
⋮						
n_b				$\sigma(r_x, n_b)$		
⋮						
n_n						

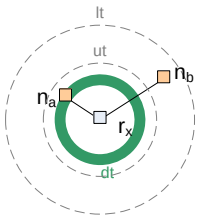
(b) No created item is within the *upper similarity threshold* of r_x. The algorithm decides on a *remove* event for r_x and terminates as all other matrix values are smaller than $\sigma(r_x, n_a)$.



(c) Move detection scenario 2

	r_1	r_2	...	r_x	...	r_m
n_1						
⋮						
n_a				$\sigma(r_x, n_a)$		
⋮						
n_b				$\sigma(r_x, n_b)$		
⋮						
n_n						

(d) The maximum similarity value $\sigma(r_x, n_a)$ lies within the *upper similarity threshold* of r_x, the difference between largest and second-largest similarity values is smaller or equal to the *difference threshold* (green circle). In this case, the algorithm cannot decide on a *move* event for r_x and a remove event is issued. The r_x column (blue background-color in the table) is removed from the similarity matrix (by setting its values to $\emptyset$).



(e) Move detection scenario 3

	r_1	r_2	...	r_x	...	r_m
n_1						
⋮						
n_a				$\sigma(r_x, n_a)$		
⋮						
n_b				$\sigma(r_x, n_b)$		
⋮						
n_n						

(f) The maximum similarity value $\sigma(r_x, n_a)$ lies within the upper threshold *ut* and the difference to the second-largest similarity value is greater than the difference threshold *dt*. The algorithm decides on a move event for the pair r_x, n_a and removes the r_x column and the n_a row (blue background-color in the table) from the similarity matrix (by setting its values to $\emptyset$).

Figure 8.7: Three scenarios of the move detection algorithm. The left-handed Figures (8.7a, 8.7c, 8.7e) show a graphical representation of three considered items r_x, n_a and n_b (see text). The inverse of their similarities is used as their graphical distance, i.e., the closer two items the more similar they are. Upper and lower threshold values are depicted as dashed circles, the distance threshold is depicted as green circle. The right-handed figures show the respective similarity matrices.

8.3.3 Subtree move operations

A subtree move operation means that a whole subtree of the folder hierarchy is moved to another location in the file system tree. Such an operation is quite common when users reorganize their folder hierarchies. For example, the hierarchy reorganization in Figure 8.2a could have been achieved by a sequence of such operations. Figure 8.8 shows a single, exemplary subtree move operation that is triggered by moving the subtree's root folder (B) to a new location (as a child of C). This operation changes the locations of all file system objects contained in the moved subtree. This means that a single move event may trigger a large set of dependent move events. Algorithm 2 is used to calculate pairwise similarities between file system objects. In the case of a subtree move operation it would have to calculate the similarities between all file system objects in the moved subtree as all of them changed locations. This is clearly not desirable as it would result in many unnecessary (r, n) pairs to be compared. In the example in Figure 8.8, 8 file system objects have changed locations which would result in 64 (r, n) pairs.

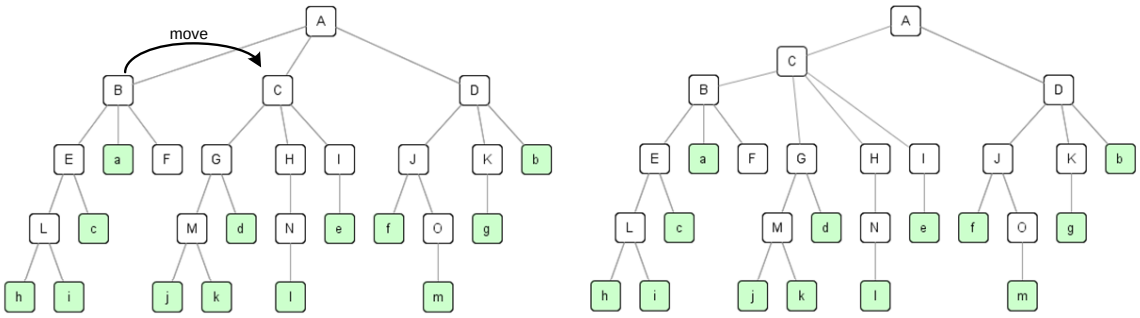


Figure 8.8: Subtree move operation. The figure shows a file hierarchy where folders are represented as white boxes labeled by uppercase letters and files as green boxes labeled by lowercase letters. Folder B in the left hierarchy is moved to folder C resulting in the hierarchy depicted in the right half of the figure. Note that this single operation results in changed locations of the folders *B, E, F, L* and of the files *a, c, h, i*.

For this reason we split our file move detection algorithm into two stages: first, folder tree reorganizations are detected by applying Algorithm 2 only to the elements of $\mathcal{R}_t$ and $\mathcal{N}_t$ that represent folders (Definitions 8.12 and 8.13). Whenever a folder move event is detected, we propagate the changed path also to the child file system objects of this folder in our index. Consider the example in Figure 8.8: four folder move events should be detected:
$\mathcal{M} = \{m(A/B,\ A/C/B), m(A/B/E,\ A/C/B/E), m(A/B/F,\ A/C/B/F), m(A/B/E/L,\ A/C/B/E/L)\}$
For each detected folder move event, the changed path prefix is propagated to all child file system objects of this folder[6]. For example, the move event of item B results in the old path prefix *A/B/* being replaced by the new prefix *A/C/B/* in all path/URI feature values of all child file system objects of this folder. By this, the new paths are propagated to all file system objects in the B-subtree in Figure 8.8.

After this first stage, the move detection algorithm is applied to all recently removed or created files (Definitions 8.14 and 8.15). In the example in Figure 8.8 these sets would be empty: all new locations for the files a, c, h, i would already be updated in the index due to the above-mentioned path propagation operation. The overall event detection procedure is described in Algorithm 4.

[6]Note that it is not sufficient to do this operation only for the subtree root folders as child folders of these roots might themselves be root folders of other subtree move operations.

Definition 8.12 (Recently removed folders) $\mathcal{R}_t^{dir} = \{r \in \mathcal{R}_t \,|\, r_{type} = FOLDER\}$

Definition 8.13 (Recently created folders) $\mathcal{N}_t^{dir} = \{n \in \mathcal{N}_t \,|\, n_{type} = FOLDER\}$

Definition 8.14 (Recently removed files) $\mathcal{R}_t^{file} = \{r \in \mathcal{R} \,|\, r_{type} = FILE\}$

Definition 8.15 (Recently created files) $\mathcal{N}_t^{file} = \{n \in \mathcal{N} \,|\, n_{type} = FILE\}$

8.3.4 Time interval based blocking of file system events

The described Algorithm 3 can be used to detect file and subtree move events by considering an arbitrary set of (r, n) pairs. One could, for example, use the algorithm to calculate a diff between two snapshots of a file tree (cf. Section 8.5.3). However, the time complexity of our algorithm is cubic ($\mathcal{O}(N^3)$) as it calculates the similarities of all (r, n) pairs and, in the worst case, finds $|\mathcal{R}_t|$ predecessor/successor pairs. Thus, large similarity matrices result in long run times of our algorithm. A further practical disadvantage of a large number of (r, n) pairs is that there are more possibilities to make mistakes. As our algorithm is heuristic by design it will not always make the right decision. The more possible move targets for a particular r_x our algorithm has to consider, the more errors are expected.

We consider a time interval blocking (TIBB) approach similar to the one presented in Section 6.5.4 to address both problems. TIBB is based on the fact that file system change events are usually spread over time, i.e., file system content changes *gradually*. This means that the number of (r, n) pairs our algorithm has to consider at a particular time t depends on when we last synchronized our index X with the considered file system. Generally spoken, the more often we apply our algorithm to synchronize the index with the file system, the faster and more accurate it works.

Figure 8.9 graphically represents this TIBB approach: Imagine that our algorithm synchronized the index $\mathcal{X}_t$ with the file system at time t_0. The blue-shaded area of the outer rectangle now represents the number of (r, n) pairs that have to be compared when synchronizing again at time t_4. The much smaller area of the white rectangles represents the number of (r, n) pairs that have to be considered when synchronizing also at times t_1, t_2 and t_3.

There is a natural tradeoff between increased accuracy and reduced number of similarity calculations on the one hand and the overhead costs for file system crawling and index synchronization on the other hand. Now, the question arises how to find the optimal points in time when index and file system should be synchronized. Different from the TIBB approach presented for DSNotify, we do not consider crawling the file system periodically as we do for Linked Data sources. We rather exploit that tools such as *inotify* are available that may be used to listen for create, remove and update events via standard file system APIs. Note that these tools can detect and report removed, created and updated but not moved file system objects as discussed in Section 8.5. This means that the move detection algorithm is triggered whenever a remove or create event happens in the considered file system. However, this does not necessarily mean that only a single event has to be handled by our algorithm as the execution of the move detection takes some time during which other events may take place. These events have to be handled in a subsequent iteration of the move detection algorithm then, as described in the following section.

8.3.5 Overall event detection

The overall event detection algorithm is depicted in Figure 8.10. After an *initial synchronization* of the considered file system and the index $\mathcal{X}_t$ the algorithm waits for a trigger from a *native change detection* module. This module runs in an own thread and listens for change events via common file system APIs. When a change is detected, the respective file system parts are *crawled* and the required sets of recently removed and recently created items are determined by comparing with the actual index state. These sets are then used for the accurate detection of the mentioned file system events. After this, the index is updated accordingly and

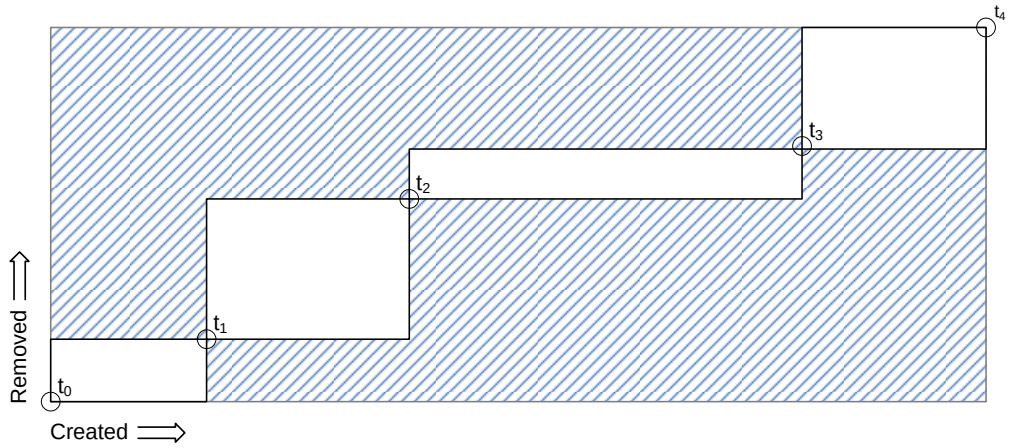


Figure 8.9: Time interval based blocking. The numbers of *create* respectively *remove* events since the last synchronization of the index $\mathcal{X}_t$ with the file system at time t_0 are plotted on the horizontal respectively vertical axis. The area of the white boxes represents the number of similarity calculations required when synchronizing with the file system at t_1, t_2, t_3 and t_4. The blue-shaded area represents the number of additional similarity calculations required when synchronizing only at t_4.

the detected events are published to a central event log before the algorithm waits again for further file system events. Note that file system events occurring during the execution of this main synchronization loop are queued by the native change detection module. Thus, the system immediately starts a next synchronization iteration when some event happened during the last iteration.

Algorithm 4: File system event detection overview.

```
1  begin
2      initialization();
3      while not finished do
4          waitForNativeChangeDetection(); // blocks until some monitored file system area was changed
5          R_t^dir, N_t^dir ⟵ crawlFolders(); // returns the sets R and N for folders only
6          inactivateRemovedRootTrees( R_t^dir ); // "inactivate" subtrees where the root cannot be accessed anymore
7          M_t^dir ⟵ moveDetection( R_t^dir, N_t^dir ); // see Algorithm 3
8          moveItemsInIndex( M_t^dir ); // propagates changed paths to child items
9          R_t^file, N_t^file ⟵ crawlFiles(); // now crawl the left files
10         M_t^file ⟵ moveDetection( R_t^file, N_t^file ); // see Algorithm 3
11         moveItemsInIndex( M_t^file );
12         updateIndices(); // update all indices
13         issueEvents(); // issue all detected events (create, remove, update, move)
14     end
15 end
```

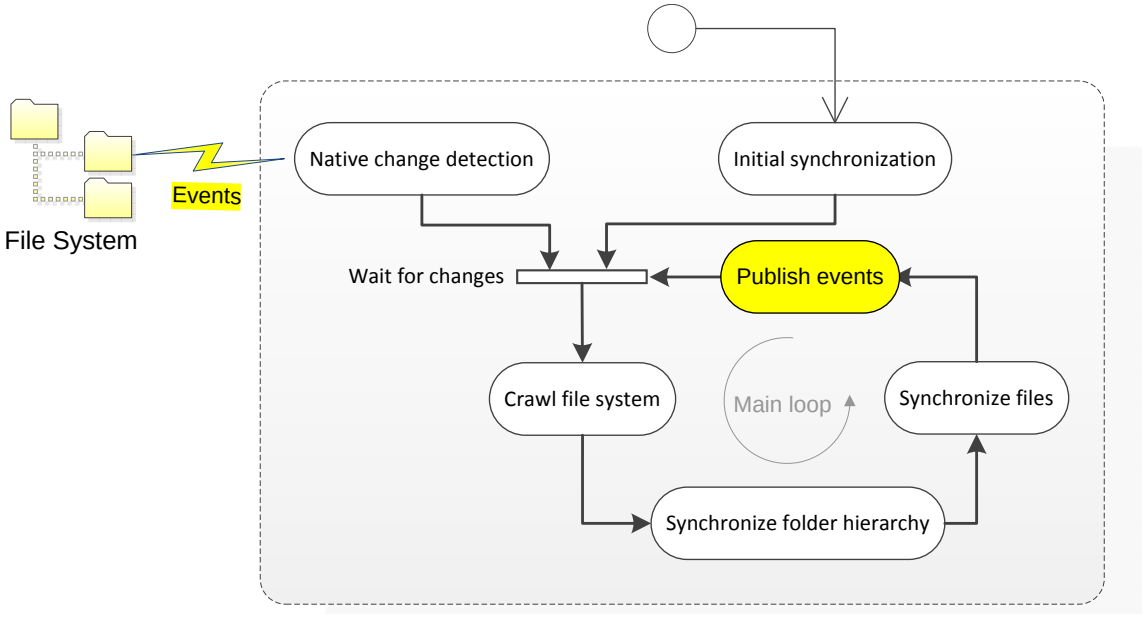


Figure 8.10: File system event detection overview.

8.4 Evaluation

We developed a simulator application in order to evaluate our solution with regard to the ability to provide stable file URIs. This simulator is able to create and execute a valid file event set (cf. Section 6.4) in order to simulate the various events that take place in a regular file system. A valid event set consists of file system operations (see Table 8.1) that can be applied to the file system. Validity means, that all required source and target file paths do exist when the event set operations are executed in their timely order. This is obviously not the case for arbitrary sets of file system events. Consider, for example, an empty file system. While the event set $E_1 = \{c_t(f), r_{t+\Delta}(f)\}$ is valid for this file system, the event set $E_2 = \{r_t(f), c_{t+\Delta}(f)\}$ is not as no file exists at location f at time t.

Because some file events "overwrite" the effects of prior ones, they make these prior events undetectable at a later point in time. For example, the effects of the create event $c_t(f)$ in E_1 is canceled out by the subsequent remove event $r_{t+\Delta}(f)$. These events are not detectable by an application that knows only the states of the file system before t and after $t + \Delta$. Further, these "overloading" effects might lead to the detection of events that never actually happened. Consider, for example, a create event $c_t(f_1)$ followed by a move event $m_{t+\Delta}(f_1, f_2)$. An event detection algorithm that observes the file system at time $t+\Delta$ would not be able to detect the original events but would detect a creation event $c_{t+\Delta}(f_2)$.

For these reasons we evaluated our algorithm not for its ability to reconstruct the actual sequence of file system events but with respect to its ability to provide stable URIs for file system objects as this is the intended usage scenario of our method. Our evaluation simulator first creates a random file tree with a configurable number of files and folders. Several statistical parameters like the average number of file system objects per folder or the average file size can be configured. During its operation, the simulator applies a randomly created, valid file event set to this random file tree, thus constantly changing it by creating/removing/moving/renaming and updating its file system objects. The simulator waits (pauses) for a configurable time between subsequent events which allowed us to measure the effect of the proposed TIBB approach on the accuracy of our solution as explained below. While applying the event set, the simulator creates and

updates a (gold-standard) mapping between original file URIs (i.e., the ones that were created when the simulator started) and their final file URIs (i.e., the URIs these files had when the whole event set was applied). Note that this map might also contain entries for newly created (original file URI field set to *null*) or removed (final file URI field set to *null*) file system objects. The constantly updated file tree was monitored at the same time by our algorithm during the simulation. The file system events detected by our tool were sent back to the simulator that created a second, detected, mapping between original and final file URIs.

Finally, we compared the gold-standard mapping with the detected mapping and determined true positives, true negatives, false positives and false negatives with regard to detected file move events. These measures were used to determine precision, recall, F-measure and accuracy of our solution.

8.4.1 Definitions

Let $\mathbb{G}$ be the gold-standard mapping and $\mathbb{D}$ the mapping detected by our algorithm. Let such a mapping be a set of pairs (u_o, u_n) between original (u_o) and final (u_n) file URIs, similar to the mapping in Definition 4.8. However, in this case we also allow *null* values for u_o or u_n. The pair $(\emptyset, u_n)$ represents a created file while the pair $(u_r, \emptyset)$ represents a removed file that originally had the URI u_r.

In the following, we define the basic measures *true positives*, *true negatives*, *false positives* and *false negatives* with regard to the ability to provide a correct mapping between original and final file URIs. Note again, that we do not measure the accuracy of event detection but the accuracy of providing the proper original URI for an existing file URI. This URI is *null* for created files and unequal *null* for all other file system objects that exist in the final mapping. Thus, a *true negative* is actually a properly detected create event as defined in the following.

True positives (i.e., properly detected move events) are mappings representing a move operation (i.e., original and final URI are different but not *null*) that were found in both, the gold-standard and the detected event set:

Definition 8.16 (True positives) $tp = |(f_o, f_n) \in (\mathbb{G} \cap \mathbb{D})| \quad with \quad \emptyset \neq f_n \neq f_o \neq \emptyset$

True negatives are mappings that represent (properly detected) create events (i.e., the original URI is *null* while the final URI is unequal *null*) that were found in both, the gold-standard and the detected event set:

Definition 8.17 (True negatives) $tn = |(f_o, f_n) \in (\mathbb{G} \cap \mathbb{D})| \quad with \quad f_n \neq f_o = \emptyset$

False positives (i.e., wrongly detected move events) are mappings representing a move operation (i.e., original and final URI are different but not *null*) that were found in the detected event set but not in the gold-standard.

Definition 8.18 (False positives) $fp = |(f_o, f_n) \in (\mathbb{D} \setminus \mathbb{G})| \quad with \quad \emptyset \neq f_n \neq f_o \neq \emptyset$

False negatives (i.e., missed move events) are mappings representing a move operation (i.e., original and final URI are different but not *null*) that were found in the gold-standard but not in the detected event set.

Definition 8.19 (False negatives) $fn = |(f_o, f_n) \in (\mathbb{G} \setminus \mathbb{D})| \quad with \quad \emptyset \neq f_n \neq f_o \neq \emptyset$

Note that the proper detection of remove events is not covered by these definitions. This means that we do not measure the number of (wrong) entries in the detected mapping that correspond to actually removed files. However, we consider such wrong entries as a minor problem as they are easily detectable by applications by simply checking whether the referenced file system object actually exists.

The above-mentioned definitions were then used to measure *precision* and *recall* using the Definitions 8.20 and 8.21.

Definition 8.20 (Precision) $precision = \frac{tp}{tp+fp}$

Definition 8.21 (Recall) $recall = \frac{tp}{tp+fn}$

Finally, we calculated the F-measure (Definition 8.22) and the accuracy (Definition 8.23) of our solution.

Definition 8.22 (F-measure) $F\text{-}measure = 2 \cdot \frac{precision \cdot recall}{precision+recall}$

Definition 8.23 (Accuracy) $accuracy = \frac{tp+tn}{tp+tn+fp+fn}$

The F-measure is the harmonic mean of precision and recall. Accuracy is the fraction of true results in the proportion to the total number of results.

8.4.2 Evaluation experiments

We evaluated our solution with seven different test runs. For each of these test runs we varied the number of milliseconds the simulator waits between two executed operations (*TBE*) from 0 to 4s. We varied this property as shorter pauses result in more file system events taking place during our event detection algorithm executes its main loop (cf. Figure 8.10). This would result also in more (r, n) pairs that have to be considered in the subsequent move detection loop. As explained in Section 8.3.4 we expect a decrease in accuracy due to more possibilities to make mistakes when more (r, n) pairs are available. We wanted to confirm this expected behavior in this evaluation.

Each of the seven test runs was repeated 50 times, thus we executed a total of 350 experiments. We created random file trees containing 155 folders and 1550 files each. The average number of file system objects per folder was 11 which is equal to the mean value found in the special folder data sets of our study (cf. Figure 7.1b). A total of 100 file system operations were executed per experiment. The type of these operations was chosen randomly (equally distributed) among the four types *create*, *remove*, *update* and *move*. Thus, on average, 25% of all operations were file move events. Note that some of these move events were actually rename events.

The files were automatically and uniquely named by concatenating a prefix with a numerical value from an increasing counter. The file extension was randomly chosen from a set of five different extensions (uniformly distributed). The byte size of the files was determined using a lognormal distribution as proposed in [Dow01, AADAD09]. We used the parameters from the table in Figure 7.3a for this distribution. When files were updated, the new file size was again randomly chosen from this lognormal distribution. Folder updates were simple *touch* commands that updated the folder's time-stamp. Target directories of move operations were chosen randomly (equally distributed) among all available folders in the current experiment's file tree.

The evaluation was done on a standard Windows PC. The used threshold values were $lt = 0.35$, $ut = 0.7$ and $dt = 0.01$. Table 8.3 summarizes the results of these experiments. It shows the averages of the number of detected events, the measured true and false positives and negatives and the measured precision, recall, F-measure and accuracy. These latter measures are further plotted in the Figures 8.11a, 8.11b, 8.12a and 8.12b.

8.4.3 Evaluation results

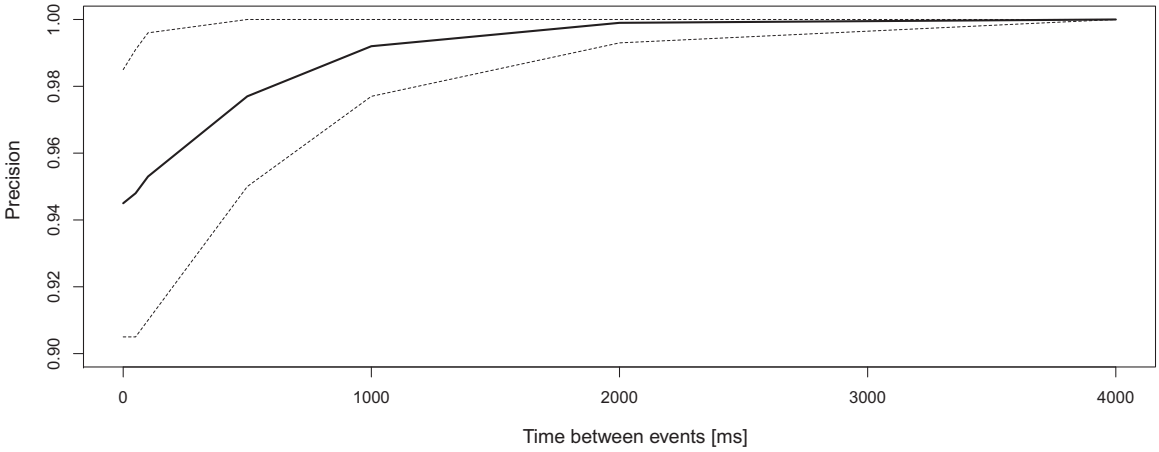
The throughout high values for the determined performance measures (precision, recall, F-measure and accuracy) show how well the algorithm worked for the described evaluation scenario (Table 8.3). These numbers increase constantly with increasing time between events (TBE) clearly showing the influence of the TIBB method on the achieved accuracy. This effect can be seen well in the F-measure and accuracy plots in Figure 8.12. More time between events results in more time for Gorm to execute its main loop and to wait for new events, thus keeping the number of (r, n) pairs low. When considering only user-triggered events (e.g., when a user moves a file to another location) we can expect higher average TBE times and thus high accuracy of the algorithm. However, even at maximum event rate ($TBE = 0ms$) the algorithm reached an accuracy of

TBE	*ADE*	*fp*	*fn*	*tp*	*tn*	*precision*	*recall*	*F - measure*	*accuracy*
0 ms	282	3.00	6.12	63.24	20.28	0.95	0.94	0.94	0.92
50 ms	302	2.36	2.94	58.64	21.54	0.95	0.98	0.96	0.95
100 ms	299	1.82	0.52	53.86	22.44	0.95	0.99	0.97	0.96
500 ms	364	1.14	0.24	62.6	23.06	0.98	1.00	0.99	0.98
1000 ms	344	0.42	0.02	66.86	22.92	0.99	1.00	1.00	0.99
2000 ms	351	0.02	0.02	62.22	23.76	1.00	1.00	1.00	1.00
4000 ms	400	0.00	0.04	69.52	23.54	1.00	1.00	1.00	1.00

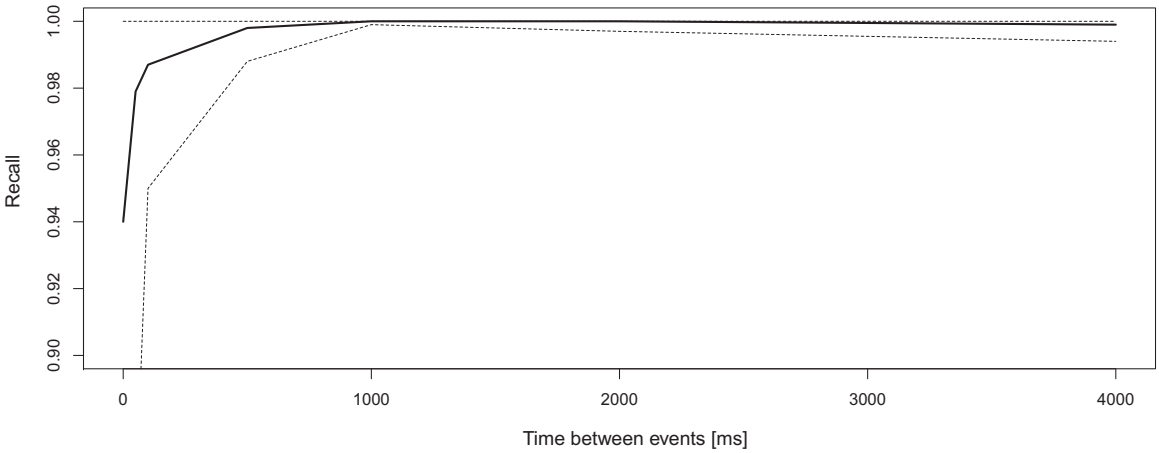
Table 8.3: Evaluation results. The values are averaged over 50 experiments. Columns: time between events (TBE), average number of detected events (ADE), averages of false positives (fp), false negatives (fn), true positives (tp), true negatives (tn), precision, recall, F-measure and accuracy.

92%. The absolute numbers (false negatives, false positives) show that Gorm made only few wrong decisions that resulted in wrongly detected mappings in the evaluation experiments.

The average number of detected events (ADE) shows that the original 100 file system operations resulted in an about 3 to 4 times higher number of detected events. One reason for this is that updates to the contents of a folder result also in changes of this folder's last modification time, which in turn leads to the detection of an additional update event. Further, note that a directory move operation may result in multiple "move mappings" detected by Gorm as explained above. Thus, the values for true and false positives and negatives that are calculated with respect to the gold-standard mapping may well exceed the number of actual events that took place.

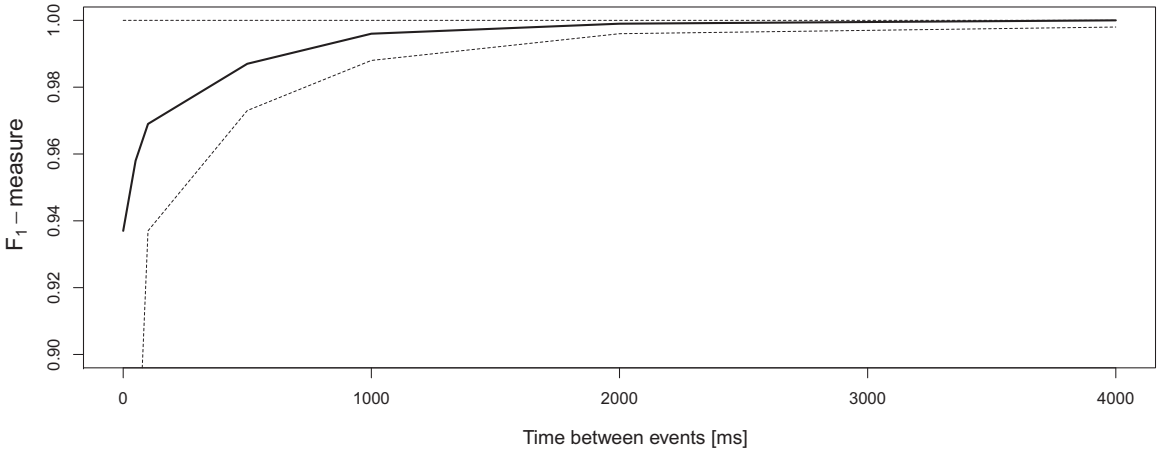


(a) Measured precision. Mean values ± standard deviation (dashed lines), N=50

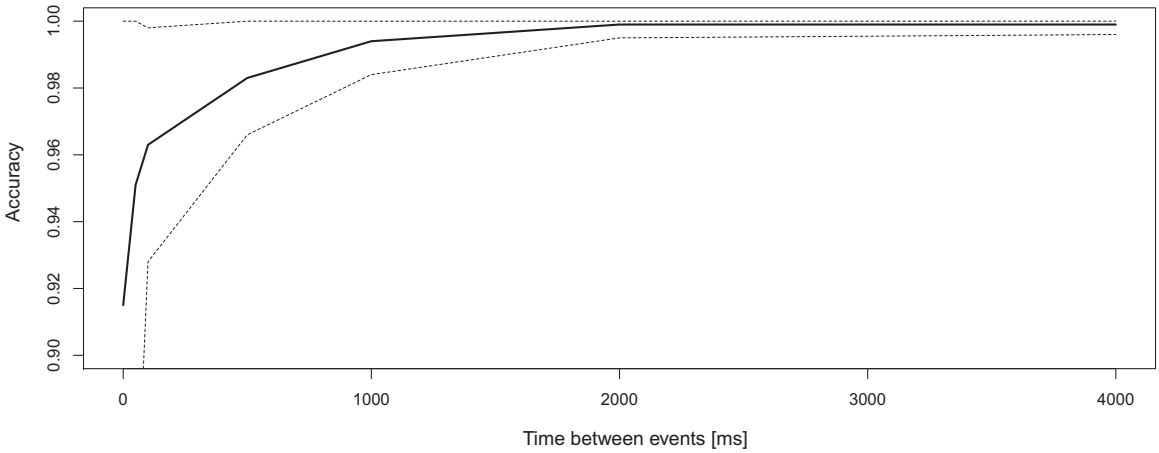


(b) Measured recall. Mean values ± standard deviation (dashed lines), N=50.

Figure 8.11: Measured precision and recall in the evaluation experiments.



(a) F-measure. Mean values ± standard deviation (dashed lines), N=50



(b) Accuracy. Mean values ± standard deviation (dashed lines), N=50

Figure 8.12: Measured F-measure and accuracy in the evaluation experiments.

8.4.4 Timeline experiment

Additional to the functional evaluation described above, we also conducted a small experiment in order to learn about the expectable event rates in real-world situations, i.e., on regular PCs. In a small self-experiment, the author of this thesis created weekly scans (snapshots) of one of his hard drives in his business PC for a period of 10 sequent weeks[7]. The PC was used every day for about 9-10 hours regularly, the main activities were the editing of documents and small programming tasks. The scanned hard disc did not contain any operating system files, other (possibly higher) event rates might be expected on such drives. The disc snapshots contained the values for all file system features shown in Table 7.1.

The snapshots were named T_1-T_{10} and are summarized in the table in Figure 8.13a. The last snapshot contained a much larger number of file system objects than the first one (+26,287), the total file size increased by about 6 GB of data. The data shows a continuous growth in the numbers of files and folders with one exception where the difference in the number of folders was slightly negative. The observation that the total number of file system objects grows over time is supported by the data reported in a large Microsoft study [ABDL07].

Using these data we also tried to find evidence that individual files tend to grow rather than shrink over time. For this, we first needed to assign corresponding files from two subsequent snapshots to each other, including moved and renamed files. As the scans contained all feature values required by our Gorm algorithm, we were able to apply it in order to detect what files were created, removed, updated and moved between two snapshot dates. Note that for the same reasons as explained above, these numbers do not directly reflect the events that actually took place but rather what is detectable. Now we compared the size features of all updated and moved files in two consecutive snapshots and counted the number of files that grew respectively shrunk in size. The results are summarized in Table 8.13b.

Although this experiment cannot be generalized due to its small data basis and the fuzziness inherent to the used method, the data still gives an idea about the magnitude of expectable (create, remove, move) event rates in real-world data sets. As can easily be seen, our algorithm properly determined the total numbers of created/removed file system objects. Further, these data show that the numbers of remove and move events are much smaller when compared to create and update events. They also show that files are more likely to grow rather than shrink in byte size. We found about twice as much files that grew than files that shrunk which fits very good with our proposed size similarity function defined in Definition 8.6.

8.5 Related work

The following section discusses related work for our approach. Note that the related work for DSNotify discussed above in Section 6.7 is also related to this research but will not be reprinted here.

8.5.1 Instance matching

The goal of solving the instance matching (IM) problem lies in "detecting whether two different resource descriptions refer to the same real-world entity" [CFLM08]. As the broken link problem described in Section 6, the file move detection problem is also related to instance matching. However, instance matching is defined as calculating the similarity between two instances originating from two ontologies O_1 and O_2. Semantic relationships in these ontologies are exploited by current IM solution approaches, e.g., in the form of matching properties at the schema level. However, such information is unavailable in a regular file system. Here, other specific properties can be exploited such as the hierarchical organization schema or the specific characteristics of low level file features as shown in our approach. Note further, that current IM approaches

[7] One snapshot was actually made 2 weeks after the prior one. However, we do not believe that this affects the results as the PC was not used for one week in this period.

ID	Date [days]	Files	Folders	FOs	Total size [MB]
T_1	2011-01-13	258,655	91,338	349,993	141,996.5
T_2	+7	+7,049	+5,841	+12,890	+2,572.9
T_3	+7	+166	-8	+158	+1,724.6
T_4	+7	+409	+82	+491	-760.3
T_5	+7	+4,781	+1,307	+6,088	+436.3
T_6	+7	+748	+222	+970	+294.9
T_7	+7	+3,289	+1,059	+4,348	+1,419.0
T_8	+7	+803	+344	+1,147	+279.3
T_9	+7	+121	+52	+173	+9.8
T_{10}	+7	+11	+11	+22	+36.2
sum		17,377	8,910	26,287	6,012.6

(a) Timeline experiment data sets. The first row shows the absolute numbers for the first snapshot, the rows below show the value differences to the respective prior snapshot. The value -8 in row 3, column 4, for example, means that snapshot T_3 contained eight folders less than snapshot T_2. The last row contains the sums of the relative values.

IDs	Created	Removed	Updated	Moved	Shrunk	Grew
T_1/T_2	12,945	55	2,690	68	42	138
T_2/T_3	221	63	727	43	100	151
T_3/T_4	507	16	82	12	2	20
T_4/T_5	6,116	28	1,178	89	79	175
T_5/T_6	1,032	62	1,395	62	72	149
T_6/T_7	4,353	5	1,029	44	18	115
T_7/T_8	1,199	52	857	40	70	114
T_8/T_9	176	3	677	4	19	80
T_9/T_{10}	52	30	398	25	38	75
average	2,956	35	1,004	43	49	113
std.dev.	4,303	24	746	27	33	48

(b) Counts of events and grown/shrunk files in the timeline data sets. The Gorm algorithm was used to detect events in two consecutive snapshots. The rightmost columns contain the counts of files that decreased/increased in their sizes.

Figure 8.13: Timeline experiment data sets.

are not applied incrementally (in a TIBB manner) but rather try to find a mapping between instances from two snapshots annotated by unaligned ontologies.

8.5.2 Machine learning

One possible solution to solve the file move detection problem would be to classify (r, n) pairs by machine learning algorithms. As the dimensions (i.e., number of features) in our considered feature vectors are quite low, such classification problems could efficiently be solved using, e.g., supervised machine learning algorithms like support vector machines (SVM) or k-nearest neighbors (kNN). Most learning algorithms, like SVM or kNN, require the input vectors to be numerical and normalized to similar ranges (e.g., $[0, 1]$) which could be achieved using our proposed similarity functions. However, some learning algorithms use dis-

tance metrics (e.g., SVM with Gaussian kernels) which are quite sensitive to heterogeneities in the input data (mainly as a metric has to fulfill the triangle inequality). If such an algorithm was chosen, this should be taken into account when designing such conversion functions. The file move detection problem is not a simple binary classification problem as discussed above in Section 8.2. One of the challenges for applying a machine learning approach is that the situations such an algorithm faces vary a lot depending on the dynamics of a considered file system: sometimes, there will be only a few feature vectors to compare, sometimes, there will be hundreds. A further challenge is the incorporation of the mentioned plausibility checks into standard machine learning algorithms as neglecting them will most likely have to be paid with reduced accuracy and increased computational costs. However, the major obstacle for applying supervised machine learning approaches to the described problem is the current lack of available training data.

8.5.3 Diff algorithms

The presented algorithm is roughly comparable to so-called diff-algorithms that calculate differences between two or more data representations. The classical diff algorithm is a simple file comparison utility based on line-by-line comparison using the Longest Common Subsequence (LCS) algorithm. It is used, for example, in the common GNU diffutils[8] or in version control systems such as CVS, SVN, Git, Mercurial or Bazaar.

XML-diff algorithms. XML-diff Algorithms try to detect the differences between two XML documents. Such algorithms take the tree-structure of XML documents, their syntactic restrictions and sometimes even DTD or Schema information into account. XML-diff algorithms are normally based on a fixed set of operations (e.g., add/remove node, move subtree, etc.) that are considered when detecting change operations and their order that convert one documents into another. Usually, cost functions are associated with these operations and a sequence of operations that transform two XML documents into each other with minimum costs is searched. Cobéna et al. discuss important aspects of such algorithms, such as correctness, minimality, semantics, performance and complexity as well as the ability to detect "move" operations [CAH02]. In their comprehensive survey, they compare several XML-diff algorithms where some are based on LCS (such as DeltaXML and MMDiff) while others are based on pattern matching techniques for comparing the two tree-structures such as XyDiff [CAM02] and LaDiff [CRGMW96] that detects changes in LaTeX documents. This latter group of algorithms also supports the detection of "move" operations.

Comparison of Gorm with diff algorithms. Our algorithm is not directly comparable with the classical diff algorithm that does line-by-line comparison of text documents. However, the output of this diff algorithm (i.e., a list of detected change operations that convert a particular file into another one) could be used as a content-based feature to disambiguate between move candidates detected by our algorithm: one could argue that it is more likely for a document to be the successor of another document if their line-by-line diff is smaller than with any other considered candidate. Such content-based features could further decrease the false-positives found by our algorithm. A similar strategy is pursued by the GIT version control system[9]. However, the diff calculation is itself a computationally expensive operation and it would require further experiments to determine whether such an extension would pay off.

Our algorithm compares better with algorithms that detect changes in two versions of hierarchically structured information such as the XML-diff algorithms discussed above. Although it is possible to adapt these algorithms to calculate a diff between two file trees, it is harder to adapt them to work in the incremental fash-

[8] http://www.gnu.org/software/diffutils/

[9] The GIT version control system can detect moved files based on contained source code. However, according to several reports published on the Web, GIT has problems when files are moved and updated before the move detection is done, see http://kerneltrap.org/node/11765 (Accessed May, 2011). The incorporation of content-based features, as done by GIT, could be a valuable extension of our proposed algorithm as discussed above.

ion of our TIBB approach that splits the problem into many small pieces thereby making it computationally feasible.

Further, our algorithm supports the comparison of unordered trees (i.e., trees where no left/right ordering exists and only the parent/child relationships are significant) which is supported by few existing algorithms (KF-Diff+, Delta-XML, MH-Diff, VM Tools, or X-Diff and its extension BioDIFF [SB04]). The heuristic detection of move operations in unordered trees is supported by even less algorithms, we are aware only of one algorithm (MH-Diff [CGM97]) that supports this per design (worst case computational complexity is also $O(N^3)$).

A particular difference between XML-diff algorithms and our proposed algorithm is that we are able to base our solution on the value distributions of low-level file features as discussed in Section 7. The considered file features can be used for plausibility checks as well as for specific similarity calculations. Such features are not available in XML-trees, thus these algorithms focused, for example, on incorporating the schema level which is absent in current file system implementations on the other hand. Nevertheless, the design of our heuristic algorithm that is based on domain-specific features (low-level file attributes) was influenced by the existing XML-diff algorithms.

8.5.4 File system API methods

Current file system APIs (or related APIs) usually provide basic methods for file system change notification.

Windows FileSystemWatcher. One example for this is the *FileSystemWatcher* class of the Windows .NET API[10]. This class can be used to watch specified file system folders and files. The class notifies registered applications of *rename*, *remove* and *create* events of files and folders, *move* events are not detected.

Mac OS X FSEvents and kqueue. The *file system events API* of Mac OS X provides methods for watching particular registered file system subtrees. The library notifies registered applications when some folder changes, it does not provide information what file actually changed or what event actually took place. This API is, for example, used by Apple's spotlight service and by its backup technology. Apple recommends the usage of *kernel queues* for the actual detection of *create*, *remove* and *rename* events[11].

Linux dnotify and inotify. Dnotify is a Linux module that runs in kernel-space and executes a command whenever some specific directory changes. It made use of the Linux kernel directory notification feature and is by that a purely event-based module. Dnotify was superseded by the inotify[12] module that was merged into the Linux kernel with version 2.6.13. Inotify is by now implemented on top of the *fsnotify* file system notification backend. Note that inotify reports two special event types (moved_from, moved_to) that, when properly correlated, can be used for the detection of move events. Inotify is used, e.g., by the Linux desktop search engine beagle (see 9.3.2) and some semantic file system prototypes.

Jpathwatch. Jpathwatch[13] is a platform-independent Java library for the detection of *create*, *delete*, *update* and partially *rename* events. We used this library for the implementation of the native change detection module (Figure 8.10).

[10] http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx

[11] http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html

[12] http://edoceo.com/creo/inotify

[13] http://jpathwatch.wordpress.com/

Analysis of I/O traces

Another possibility to learn about file system events is to directly observe traces of I/O requests passing through the file system's I/O manager. Such traces could be captured by kernel-space modules that hook into the I/O manager and report them to some user-space module (see Section A.2 for a further discussion). However, a particular problem when considering such low-level I/O traces is that the identification of what actually happened on a file API level is difficult. Note, for example, that simply typing a few characters of text into the Windows notepad application will result in 26 NTFS system calls [Vog99]. As a further example, it was reported that up to 80% of newly created files in Windows NT were removed within a few seconds [Ibid.]. This may be contributed to the common behavior of many programs to create copies of files when they are opened to avoid concurrency problems.

Confluence [GSV07] is a system that tries to learn semantic relationships from low-level I/O events. A kernel-space module monitors low-level I/O events (read, write) and uses their temporal locality to derive a weighted relation graph interconnecting local files. This relation graph was successfully used to enhance local search and ranking tasks. In [SSGN07], this method is improved by an algorithm that infers file-to-file relationships based on the causality between I/O events.

8.6 Discussion

Our proposed algorithm for the detection of low-level events and in particular *move* events in the file system is based on similarity measures between feature vectors derived from file system objects at different points in time. The features in these vectors are compared by specific similarity functions and the returned similarity values are combined to an overall similarity value. This overall similarity is then used to decide whether the two feature vectors were derived from the same or from two different file system objects. Some of the feature comparisons are used for plausibility checks that may quickly reject a pair of vectors as such a predecessor/successor pair. Other similarity functions try to model the dynamics of the respective file feature and return a value that represents the probability that a particular feature of a particular file system object changed from the value in vector 1 to the value in vector 2. These similarity functions take several properties of real-world data into account as analyzed in our study (Chapter 7). Further, our algorithm considers file system-specific characteristics such as the folder hierarchy that greatly help to increase the move detection accuracy while reducing the number of required feature vector comparisons considerably.

Complexity. One drawback of the proposed method is its worst-case computational complexity. Large numbers of (r, n) pairs quickly make the method unfeasible. Practically spoken, the time to extract the feature vectors is the main factor here, while the actual runtime of Algorithm 3 depends also on how many values in the similarity matrix are above the upper threshold which is normally a much smaller number than $|\mathcal{R}_t| \cdot |\mathcal{N}_t|$.

We consider it unlikely that very large sets of removed and created items will be detected in real-world scenarios as our chosen TIBB approach starts to calculate the sets $\mathcal{R}_t$ and $\mathcal{N}_t$ immediately after some change in the observed file system was detected. Furthermore, our timeline experiment showed easily manageable magnitudes of events in snapshots that were taken one week apart.

One special case, however, where $|\mathcal{R}_t| \cdot |\mathcal{N}_t|$ might become very large is when external storage devices are re-mounted to a PC under a different path as discussed in the introductory example in Section 3.3.1. For example, when a USB stick is plugged out of a Windows PC and later plugged in again, its volume character might change and by this also all paths of the contained file system objects. We deal with this special case in our implementation (that is presented in the following chapter) by first checking whether the root folder of a particular considered file tree was removed and, if this is the case, "inactivating" all metadata records of the respective items. By this, they are also excluded from the upcoming (r, n) pair determination (cf. Algorithm 4).

Our method requires an index containing the feature vectors of all considered file system objects. As the number of proposed features is relatively small (8-9 features) we are confident that such an index is manageable in real-world environments. The actual byte size depends on several factors (data type implementation, overheads, platform, index data structure, length of the stored strings, etc.), we estimated the average byte size of a feature vector based on the data of our file system study to be below 1 kB and thus the total index size may be in the range of a few hundred MBytes.

Feasibility. The evaluation of our method evidences the feasibility of the approach. Our implementation reached nearly-perfect move-detection accuracies resorting only on low-level file features that were quickly accessible via the file system API. We based our evaluation on randomly created test data due to the absence of a real gold-standard. We are, however, confident that our system shows comparable accuracies in real-world scenarios as we based our evaluation method to a large part on real-world feature value distributions. We further conducted a small timeline experiment consisting of 10 consecutive, timely close (1 week) scans of a local file system. The goal of this small experiment was to learn about the magnitude of file system events that may be expected in such an environment. The results support our hypothesis that the byte sizes of files are more likely to grow rather than shrink. Further, the detected numbers of move events were small but not to be neglected. Not detecting move events would, in our small experiment, lead to an average loss of 43 metadata records per week which would not be acceptable in a real-world scenario. Further, it has to be considered that the measured event counts for create, remove and move events differ from the real number of events because of the mentioned "overwrite" effects.

Future work. An obvious extension of our method is to include support for storing an item's global URI in certain internal metadata headers in the file itself. For the reasons explained in this thesis, this would work only for certain file system objects. For these, however, the move detection would be errorless. Another possibility would be to cache such URIs in file forks/extended attributes and use the Gorm algorithm to refresh these caches when they are lost for the reasons described in Section 2.

Another possibility to further increase the detection accuracy of our method is the inclusion of content-specific features such as internal metadata (e.g., JPEG Exif headers or MP3 ID3v2 tags) or diff information. This would, however, require additional I/O requests per comparison and would require the actual file data at hand, while our proposed method requires only the values for the discussed features. A metadata repository might however decide to mirror such content-specific metadata into a *facet* of the metadata graph[14] anyway as described in Section 4.4.4. In this case, it would be feasible to extend the Gorm algorithm to include such features when available.

The high accuracy of a file/URI mapping maintained by Gorm is the basis for the implementation of a store that implements our proposed metadata model presented in Chapter 4 as it enables a stable connection between a file system object and its external RDF metadata record. We now continue by presenting Y2, our prototypical implementation of such a metadata store in the final chapter of this thesis.

[14] For example, our implementation mirrors certain ID3v2 tags from indexed MP3 files into the metadata graph.

Part IV

Implementation

In this final part we present an implementation of the proposed metadata model that makes use of DSNotify and Gorm for preserving the integrity of links between local file system objects, metadata records and remote Linked Data resources. The described system uses these tools as well as common Semantic Web technologies to maintain a faceted metadata graph that represents a semantic overlay over local file system hierarchies. Users can add arbitrary hypertext annotations to any file system object stored in the system, including, for example, folders or empty text files. They may add semantic links between these resources and may formulate complex queries on this metadata graph. The system further exposes the stored data as Linked Data and can also consume such Linked Data from remote stores, thereby opening local data silos and integrating them with remote sources. Our prototypical implementation further demonstrates how existing metadata stored in file headers can be integrated seamlessly into such a metadata graph in order to remain backwards-compatible with the current situation of metadata handling on the desktop.

Chapter 9

Y2 – a prototype for semantic file annotation

In this section the presented building blocks are assembled and Y2, an implementation of the proposed metadata model (Chapter 4), is described. The goal of this prototype is to demonstrate how this model can be used to incrementally develop a semantic "overlay" graph that interconnects local file system objects, associated metadata descriptions and remote resources. The integrity of the links between the various resources in this emerging graph is preserved by the presented Gorm (Chapter 8) and DSNotify (Chapter 6) algorithms.

Y2 demonstrates how arbitrary file system objects can be augmented with semantic annotations without the need to modify existing data organization practices. It contains a wiki-like user interface that can be used for the textual annotation of local resources with external metadata records. Users may add arbitrary hypertext to file system objects that links them with other resources, including local files or folders, arbitrary Web resources and remote files stored in other Y2 instances. Additionally, users may create their own simple SKOS concept scheme for annotating their local content with it. Our system further demonstrates how users may share the resulting metadata descriptions and how they can author them in a collaborative manner.

Our current implementation of Y2 contains a component that mirrors selected ID3 tags from MP3 files into the metadata store and implements a bi-directional synchronization strategy as discussed in Section 4.4.4. This component demonstrates how existing internal metadata can be integrated into our model which illustrates a way how both metadata handling approaches can co-exist. This allows it to remain backward-compatible with applications that work only with these internal metadata.

Ultimately, Y2 enables users to create a local, semantic hypertext layer that annotates and interconnects local and remote resources. By this, additional navigational paths for traversing these data are created. Further, all created annotations are searchable which consequently increases the findability of the annotated file system objects themselves. All this can also be done by machine actors due to Y2's machine-friendly and uniform access interfaces. Additionally, the possibility to specify the semantics of annotations and links further improves the findability and the machine friendliness of the described data. Finally, Y2 is able to provide stable references from and to local file system objects due to its built-in algorithms for the detection of change events on the desktop and in the Web of Data.

The Y2 implementation as well as related resources, data sets and additional documentation are available at `http://purl.org/y2/`.

9.1 User interface

We start our report on Y2 with a detailed description of its graphical user interface (GUI) which demonstrates most of its functionality. The main GUI of Y2 is depicted in Figure 9.1a.

9.1.1 File selection

Users can add folders from the locally available file tree (e.g., folders on the local disk, on a mounted network drive or on a memory stick) to Y2 easily by dragging them to a certain area on the GUI. Y2 will then integrate the selected subtree into its metadata store. This is done by recursively crawling this subtree and extracting an RDF representation for all contained file system objects. These RDF data are expressed using the vocabulary presented in Section 9.2.1 and are stored in the *core* layer of the Y2 metadata store.

Note that Y2 can further be configured to exclude certain file system object from this crawling process. This can be done by providing regular expressions that are applied to the name of each crawled file system object. If a name matches an expression, it is either included into or excluded from the remaining crawling process. This makes it possible to include, for example, only files with certain extensions (e.g., only image files) or to exclude certain file system objects (e.g., ".svn" folders; ".log" files) from the crawling process.

9.1.2 Hypertext annotation

The Y2 GUI allows users to browse and extend the RDF descriptions of file resources. Each file system object can be annotated with formatted hypertext as shown in Figure 9.1. These hypertexts are represented in RDF by (i) storing the plain text entered into the input field and (ii) storing text-ranges that mark-up certain text regions with one (or multiple) of the following styles:

i) Text format: boldface, italic, underline and strikeout

ii) Text foreground color: green, red, blue, black

iii) Text background color: yellow (e.g., for highlighting certain text regions)

iv) Semantic link (marked by a pair of URIs that denote the target resource and the semantic type of the link)

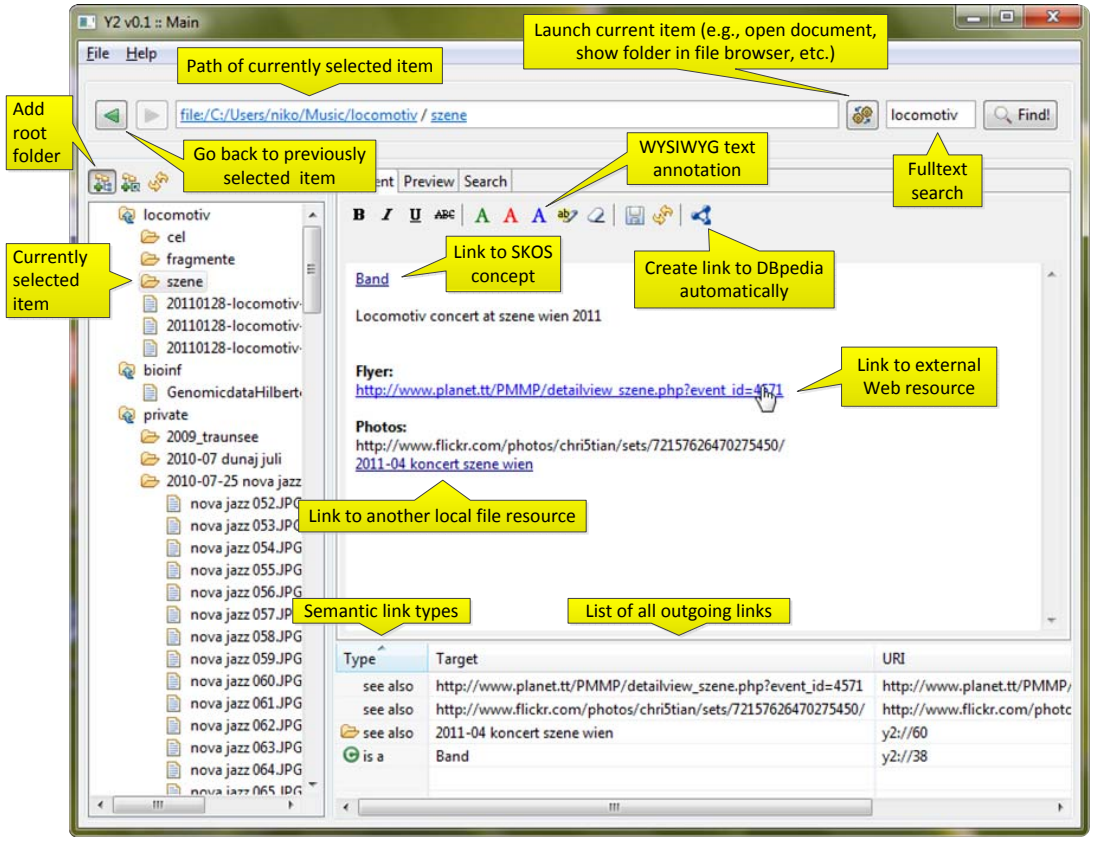
The plain-text is stored in the RDF graph using the `rdfs:comment` property, the style information by using a Y2-specific property. Semantic links are not only contained in this "style" text, but are also explicitly modeled by appropriate RDF properties. Such explicit relations can thus be queried easily, e.g., with SPARQL queries as shown below.

All resulting RDF triples, however, are not added to the store's core layer but to a distinct "annotation" layer. This means that these data are logically separated from the Y2 core layer and can thus also be accessed separately. The actual update of the RDF graph (i.e., the storage of the annotation texts) is triggered either manually when the user presses the "*save*" button or automatically when she navigates to another file system object (autosave functionality) .

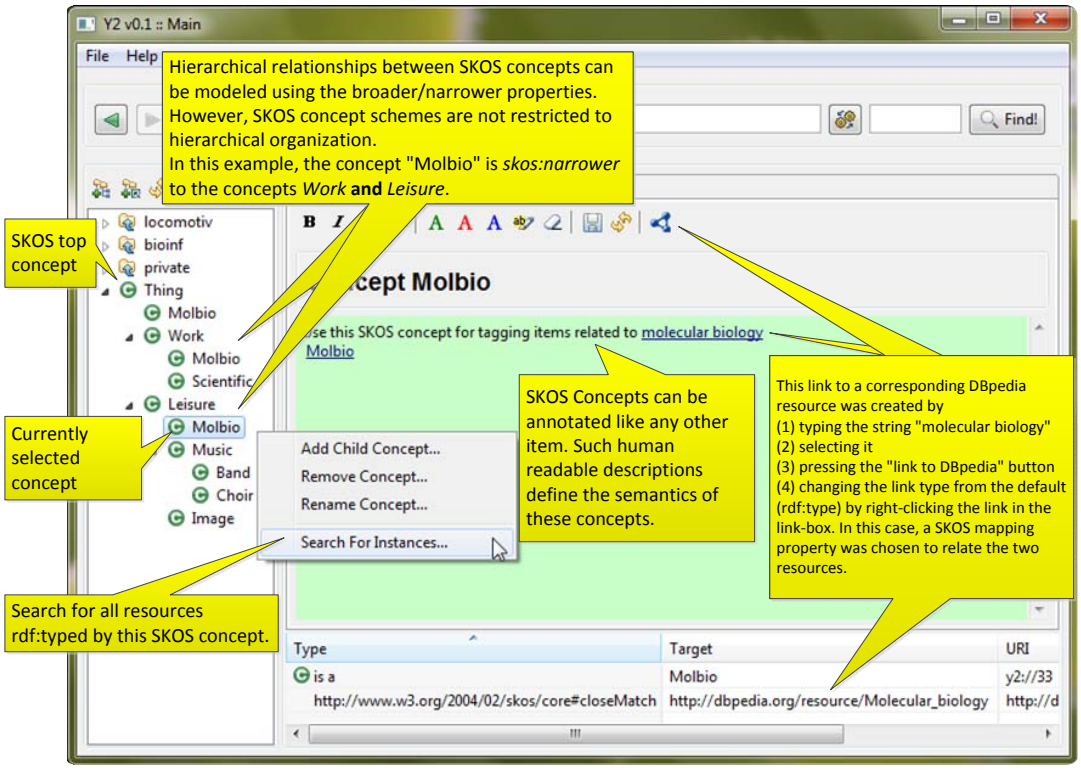
Drag-and-drop support

It is further possible to drag-and-drop or copy/paste the following objects from any other desktop application to the Y2 editing area where they are interpreted accordingly:

File system objects: If the dragged file system object is managed by Y2, a link to the respective resource is created.



(a) The GUI is split into four areas: (i) the top area provides access to browser-like functionality (address bar, forward/back navigation, full-text search). (ii) the left side shows the currently handled file trees, (iii) the central area enables users to enter hypertext in WYSIWYG style, and, (iv) the bottom area lists all links outgoing from the currently displayed item. An N3-serialized RDF representation of this item is shown in Appendix D.



(b) SKOS editing with Y2. The SKOS concepts can then, for example, be used for tagging local file system objects. Semantic filtering of search results by such SKOS concepts is shown in Figure 9.2a.

Figure 9.1: Main graphical user interface of Y2.

URIs: HTTP URIs that are dragged to the content area result in the insertion of a link to the respective (external) resource. When clicking an external link, Y2 will try to display the respective resource in the system's standard Web browser.

Text snippets: Regular strings are simply inserted into the content area. Dropped rich-text snippets are parsed and the formatting instructions are mapped to the respective Y2 markup. This enables comfortable copying/pasting of formatted content descriptions stemming, e.g., from wordprocessors.

9.1.3 Navigation

Navigation to a particular file system object can be done by several GUI interactions:

i) By selecting an item in the Y2 file tree.

ii) By using the "go back/go forward" buttons to navigate the history of displayed items.

iii) By clicking a link to another local item.

iv) By dragging a file (e.g., from the file explorer) to the Y2 file tree.

v) On Windows systems: by pressing "Show in Y2" in the context menu of the regular file browser (works only if the respective shell extension was installed), see Figure 9.2b.

9.1.4 Semantic linking

The GUI allows users to annotate file system objects with links to local or remote resources. Links between local file system objects are created by simply dragging a file item either from the Y2 file tree or from the system's file explorer to the editing area. Y2 reacts on such drag-and-drop events by inserting a properly formatted link-text into the input area which consequently results in adding an appropriate RDF property to the current resource. The RDF property that is used for this (by default it is `rdfs:seeAlso`) can later be changed manually by right-clicking a link and entering the new property URI.

SKOS editor

Y2 includes also a simple SKOS [MeBe09] editor that can be used to create concept schemes useful for the semantic organization of local content. Figure 9.1b shows how SKOS concept scheme editing in Y2 works.

A SKOS top concept ("Thing") is predefined by Y2. By right-clicking a concept, users may "add a child concept" which results in the creation of a new SKOS concept that is related to its "parent" using `skos:narrower` and `skos:broader` properties respectively. As SKOS concept schemes are not restricted to hierarchical organization, a concept can also be `skos:narrower` than multiple other concepts, as shown for the "Molbio" concept in Figure 9.1b. SKOS concepts are one example for non file-related resources maintained by Y2, their RDF representations are held in a distinct layer in the metadata store. This local SKOS concept scheme can now be used for the organization of local file data. Users may simply drag-and-drop concepts to the Y2 editing area. In this case, Y2 inserts an `rdf:type` link to such a concept which "types" the respective item with this SKOS concept. In the example in Figure 9.1a, the folder "szene" is rdf-typed with the SKOS concept "Band". The link-box in the bottom user interface area shows this semantic link (instead of "rdf:type", the GUI renders "is a" as the link type). The property used for such a link can be changed later manually as described above.

Note that in this example a folder representing a location ("Szene Wien") was rdf-typed by a SKOS concept describing a band which is arguably semantically incorrect. It is, however, one way how we expect users to make use of our system. In several semantic wiki-related projects, the author observed that non-expert users tend to develop broad yet shallow concept schemes for annotating their content [PSK08]. These

concept schemes were intuitively used for content-tagging as shown in this example as this enabled simple querying of all resources that are "related" in some way to the tagging concept. However, the ability to "strengthen" respectively adjust the semantics of the created content at a later point in time was considered as beneficial by all interviewed users. We believe that this use-case is well-covered by the Y2 interface, a deeper discussion of this issue is, however, beyond the scope of this work.

Automatic linking

Y2 contains an exemplary function that demonstrates one simple way how semantic linking in a Web of Data can be done in a (semi-) automatic fashion. Semantic links to DBpedia can be created automatically by first selecting a text in the editing area and then pressing the "Link to DBpedia" button (Figure 9.1). Y2 will now query the DBpedia data space via SPARQL for the selected term and the selected text is then converted to an `rdf:type` link if a resource with a corresponding URI is dereferenceable from there. This link enables a user to quickly relate a local file with "concepts" from DBpedia. Again, the property used for such RDF links can be manually changed afterwards. Figure 9.1b shows the result of this linking process; the link to the DBpedia resource `http://dbpedia.org/resource/Molecular_biology` was created in the following way:

i) The user entered and selected the text "molecular biology" in the input area.

ii) The user pressed the "Link to DBpedia" button.

iii) Y2 now queried the DBpedia data space for an appropriate resource and as it exists, created an `rdf:type` link to this remote resource.

iv) The user right-clicked this link and changed the link to use the `skos:closeMatch` property.

As a result, the user expressed that the local SKOS concept "Molbio" is very similar to the respective DBpedia resource that may also be interpreted as a concept (cf. [HSB07]). Further, a navigational path was established that allows the user to quickly navigate to the respective DBpedia description and from there to the respective Wikipedia article (by following a `foaf:primaryTopic` link) which both include a number of links leading to other, related resources. Additionally, the Y2 metadata store can now be searched for all resources that are related to this resource/concept. The Y2 GUI allows users to search a store by combining full-text search with subsequent semantic filtering.

9.1.5 Semantic search

Full-text search

Searching in Y2 always starts with a full-text search that was implemented using Apache Lucene[1]. The search input box in the upper right corner of the GUI (Figure 9.2a) is nowadays well-known from regular Web browsers. Users may enter their text queries there, using the Lucene query syntax that supports wildcard characters such as the question mark (matches zero or one arbitrary character) or the asterisk (matches an arbitrary sequence of characters).

Lucene does not support wildcards as query prefixes as this would result in very long execution times due to the design of the Lucene indexing structures. In Y2, however, we introduced two such special queries to support the following two use cases:

"*" queries. Users that do not want to formulate a full-text query may enter a single asterisks which will return a result-set that matches all available items in the store. The Y2 GUI supports result set paging (see Figure 9.2a) which allows users to deal with such long result sets. This list can then be filtered as described below (e.g., by showing only the items that are typed with a particular SKOS concept).

[1] `http://lucene.apache.org/`

"*.XXX" queries. A second wide-spread use case is that users want to search for files with a particular extension (e.g., all ".mp3" files). For this reason, Y2 allows "*.XXX" queries that are internally rewritten to valid Lucene queries that search only file extension values that are stored in separate fields in the text index. For example, the query "`*.t??`" returns all files whose extension is three characters long and starts with a "t" (e.g., "txt", "tex" or "tmp").

Semantic filtering

The Y2 GUI allows to filter the result set of a full-text query (cf. Figure 9.2a) by (i) selecting a SKOS concept created with Y2's simple SKOS editor, (ii) selecting a file system object type (file, folder or item which is the superclass of both), or, (iii) selecting both.

All three cases are treated uniformly by Y2: First, the respective concept URIs[2] are used to assemble a SPARQL query that is used to retrieve a set of resource URIs that fulfill the respective criteria. Then, this set is used for intersecting the full-text query result set, thereby filtering all items from there that are not typed by the respective concepts.

SPARQL endpoint

Our prototypical user interface does not support the formulation of more sophisticated queries. Machine actors and advanced users may, however, formulate such complex queries by directly accessing Y2's SPARQL endpoint. By this, they may query a metadata store in various ways, e.g., by accessing low-level file features as shown in the Listings 9.1, 9.2 and 9.3.

Listing 9.1: SPARQL query that returns the names of all known files.

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>

SELECT ?fn WHERE {
  ?f a y2:File;
      y2:local-name ?fn .
  }
```

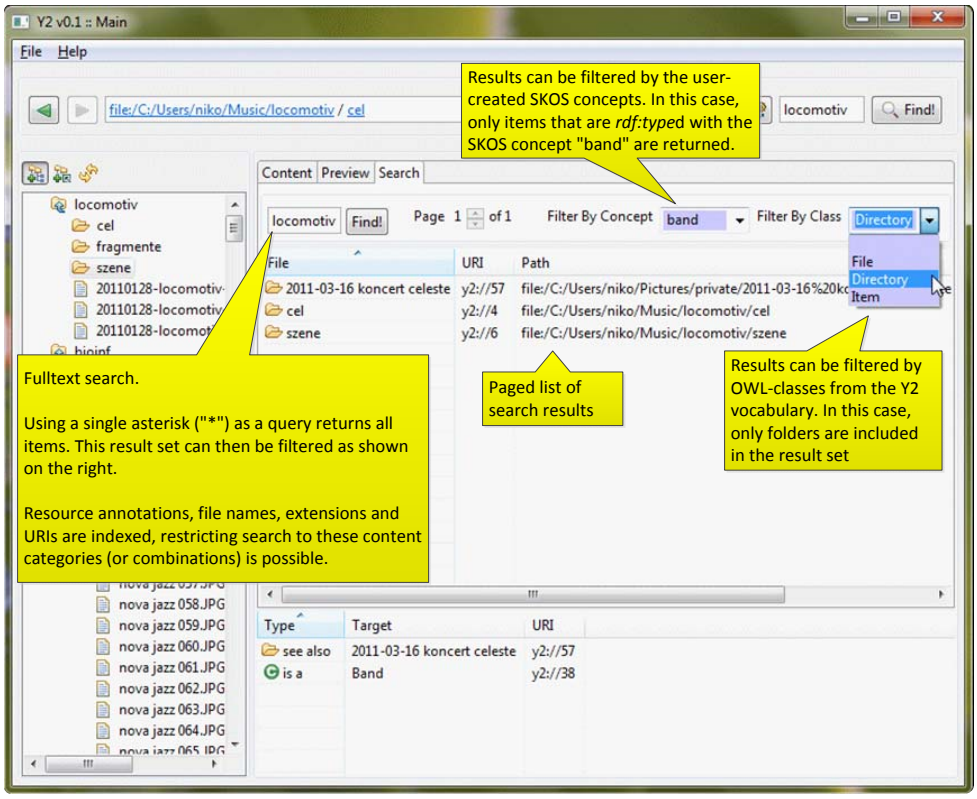
Listing 9.2: SPARQL query that returns all distinct file extensions.

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>

SELECT DISTINCT ?fn WHERE {
  ?f a y2:File ;
       y2:extension ?fn .
}
```

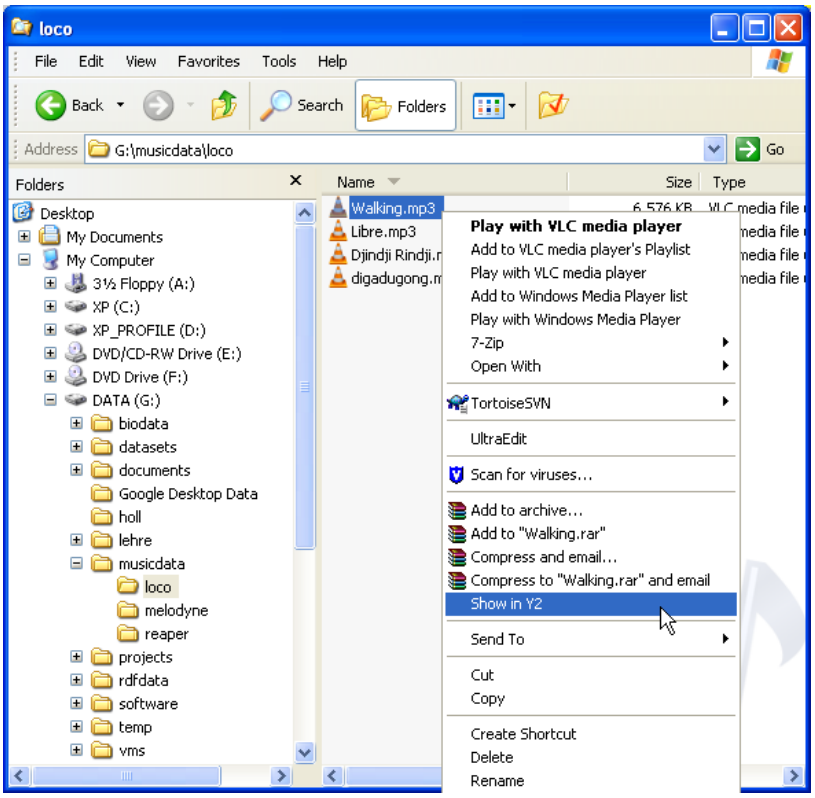
Listing 9.3: SPARQL query that returns the URI mapping of all files with the extension "mp3".

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT ?f ?p WHERE {
?f a y2:File ;
     y2:extension "mp3"^^xsd:string ;
     y2:path ?p .
}
```

[2]Either the ones used for the SKOS concepts or the ones for the respective OWL classes from the vocabulary presented in Section 9.2.1



(a) Y2 search GUI. This interface combines full-text search and semantic filtering.



(b) Y2 shell extension for quickly selecting a file system object in the Y2 GUI. The shell extension is automatically installed when Y2 is started and integrates with the context menu of the regular Windows file explorer. Right-clicking a file system object and selecting “Show in Y2” results in the respective file path being transferred to Y2 where the respective item is shown (if available).

Figure 9.2: Y2 search interface and shell integration.

Remember, however, that the Y2 metadata store comprises actually an RDF data set consisting of distinct *layer* graphs, e.g., the mentioned *core* and the *annotation* layers. Clients may access the data in these layers independently and may therefore also reason on their provenance (e.g., what application/component wrote what parts of the overall metadata graph). Listing 9.4 and 9.5 show how these layers (i.e., named graphs) can be accessed via SPARQL. These examples make it easy to see how applications that have access to a Y2 SPARQL endpoint may query its various layers of metadata.

Listing 9.4: SPARQL query that returns all folders that have a comment in the annotation layer.

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?s ?comm
FROM NAMED <core>
FROM NAMED <y2annotation>
WHERE {
  GRAPH ?core { ?s a y2:Directory . }
  GRAPH ?anno{ ?s rdfs:comment ?comm. }
}
```

Listing 9.5: SPARQL query that returns the names of all file system objects annotated by the SKOS concept named “Molbio”.

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>
prefix skos: <http://www.w3.org/2004/02/skos/core#>

SELECT ?s ?c ?n
FROM NAMED <y2annotation>
FROM NAMED <vocab>
FROM NAMED <core>
WHERE {
  GRAPH ?core  { ?s y2:local-name ?n . }
  GRAPH ?anno  { ?s a ?c . }
  GRAPH ?vocab { ?c a skos:Concept ;
                   skos:prefLabel "Molbio"^^<http://www.w3.org/2001/XMLSchema#string> . }
}
```

Listing 9.6: SPARQL query for all files that are somehow related to the DBpedia resource or the Wikipedia page describing the band “Gotan Project”. The query returns the resource URIs and the RDF properties used for the relations (e.g., `rdfs:seeAlso`).

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?s ?x
FROM NAMED <core>
FROM NAMED <y2annotation>
WHERE {
  GRAPH ?core { ?s a y2:File . }
  GRAPH ?anno {
    { ?s ?x <http://dbpedia.org/resource/Gotan_Project>. }
    UNION
    { ?s ?x <http://en.wikipedia.org/wiki/Gotan_Project>. } } }
```

9.1.6 Editing of remote descriptions

Y2 can also be used for editing metadata records that are stored in other, remote Y2 instances. This can be done by “mounting” a file subtree of a remote Y2 instance as explained in more detail below in Section 9.2.1. Mounted resource descriptions can be edited like any local metadata record, they are, however, stored and retrieved via the Linked Data interface of the hosting Y2 instance. This enables a group of users to collaboratively annotate a set of file resources, stored, e.g., on a shared file server.

9.2 Implementation of the proposed metadata model

In this section, we focus on some implementation details that explain how our metadata model presented in Chapter 4 was realized. The Y2 prototype was implemented as a platform-independent Java application that makes use of several open-source libraries. Among others these include the *Jena Tuple Database*[3] (TDB) as a triple store implementation, *Jetty*[4] as a Servlet container and *Apache lucene*[5] as a text search engine.

We start this section by first introducing the semantic vocabulary that we developed for expressing low-level file metadata. These metadata are stored in the core layer of the Y2 metadata store which is basically a named TDB graph in our implementation. Each file system object is represented in this graph by an RDF resource that uses a property for storing the respective file path that references the actual object in the file system. The Gorm algorithm (Chapter 8) that is directly implemented in Y2 uses these metadata to keep such file reference up-to-date. This means that the *core layer* graph of Y2 actually implements the item index $\mathcal{X}_t$ as required by Gorm.

9.2.1 Y2 core vocabulary

Figure 9.3 depicts the OWL-light vocabulary we used for the description of the file system data represented in Y2 instances. This vocabulary extends previous work published in [SP10].

File system resources are typed by the OWL classes `File` and `Directory` respectively, which are both subclasses of the `Item` class. `Items` can be hierarchically organized using `parent` and `child` properties. `Items` can further be described using various data type properties (e.g., `local-name` of this file system object, `path`, `extension` and last `modified` timestamp). Further, an `item` can be marked by an `inactive` flag that indicates that this item should not be monitored/accessed actively. `Files` can be described by additional properties (file `size`, `mime type`, `checksum`) and are disjoint from `directories`.

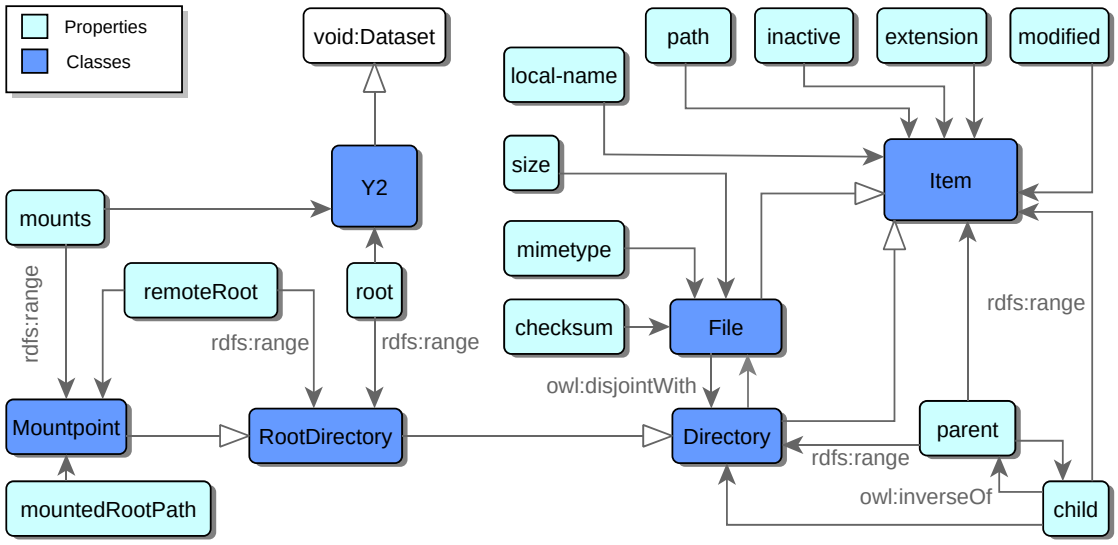


Figure 9.3: Proposed vocabulary for describing a Y2 metadata storage. Dark-blue boxes represent OWL classes, unlabeled arrows represent `rdfs:domain` relationships (filled arrowheads) or `rdfs:subClassOf` properties (empty arrowheads).

[3] http://openjena.org/wiki/TDB
[4] http://jetty.codehaus.org/
[5] http://lucene.apache.org/

A Y2 instance is represented also as a (dereferenceable) RDF resource typed by the OWL class "`Y2`". Using the `root` property, the `RootDirectories` that are actually indexed by this Y2 instance are asserted. `RootDirectories` represent the root folders of the file system subtrees that are monitored and indexed. Note that the resource that is RDF-typed by the `Y2` class can also be annotated using the `Store` class from the vocabulary presented in Section 4.3.1 which enables the specification of the storage structure (i.e., what layers using what extractors are supported by this Y2 instance).

Mounting

Y2 enables users to *mount* such roots from remote Y2 instances in order to allow collaborative file annotation. A `Mountpoint` describes what remote `RootDirecory` is mounted. It may further describe at what *local* path these remote file data are accessible, e.g., because they were mounted via some remote file protocol such as SMB or WebDAV. If this is done (via the `mountedRootPath` property), the file data itself may also be accessed directly via Y2. The metadata are, however, stored in the remote Y2 repository.

To further illustrate this, consider the scenario in Figure 9.4: Here, a local system (depicted on the left side) with a single hard disc ("C:") mounted a folder on a remote file server using a standard file sharing protocol (e.g., SMB). The remote files are locally available on the (logical) file system volume "S:". The figure further depicts one local ("X") and one remote ("Y") file. Now consider that the local and the remote Y2 instances contain metadata about their respective "C:" volumes, i.e., the local Y2 store describes the file *X*, the remote one describes the file *Y*. A user may now add a `Mountpoint` to the local Y2 instance that (i) links to the respective root directory in the remote Y2 instance and (ii) describes the local path ("S:") under which the respective resources are available. This user may then access both, metadata and data representations of the remote file *Y* directly via the Y2 GUI: double-clicking the file in the Y2 file-browser or dragging-and-dropping it from the file explorer to the Y2 GUI will display the respective metadata record; clicking the "launch" button will open the file with the respective application it is associated with by the operating system.

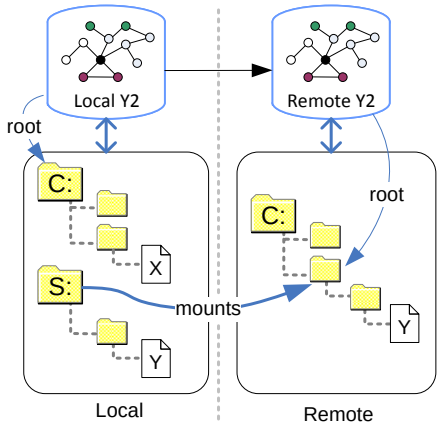


Figure 9.4: Mounting remote Y2 metadata storages.

Modifying such a remote record will store the metadata in the remote store (using the REST interface described in Section 4.4.3). If other members of a user group mount these file resources in the same way, they could collaboratively annotate the shared files. As an example, imagine that such a file server contains project documentation such as project plans, budget sheets, contracts, emails, etc. Users that cooperate on such a project may now annotate and interlink these resources collaboratively in a semantic-wiki-like fashion.

9.2.2 Gorm and DSNotify integration

Y2 integrates our two methods for link preservation:

Gorm. Gorm is used for informing Y2 about events in the local file system. The algorithm is directly integrated into the Y2 kernel and makes use of a platform-independent Java library for the detection of changes in the local file system (cf. Section 8.3.5). Gorm reacts on such detected changes by reasoning heuristically what actual events took place as described in Chapter 8. It then reports these events to Y2's central event log. Various Y2 components can subscribe to this log and are notified via callback routines whenever a new event (create, delete, update, move) is detected. These components then react accordingly, e.g., by updating file path references when files or folders are moved. By this, the connection between a file system object and its metadata record is kept stable.

DSNotify. DSNotify is not directly included in the Y2 application. Y2 can rather be configured to start DSNotify as an additional service (that is configured separately) as described in Section 6.5.8. It will then listen for move events reported by DSNotify and reacts on such by querying its metadata store for all resources that actually RDF-link to the original (old) URI contained in the event description. All such URI references will then be updated automatically with the target URI from the event description, thereby fixing broken links that would otherwise occur.

The responsibilities of Gorm and DSNotify for preserving the link integrity in a Y2 metadata store are depicted in Figure 9.5.

Temporary file unavailability

Deleting files on the local disc that are monitored by Y2 results in the detection of these events by Gorm and consequently in the removal of their metadata records from the Y2 store. However, this would also be the case for file resources that become only temporary unavailable, such as files stored on a memory stick when this stick is unmounted (cf., Section 3.3.1). Y2 implements a simple strategy to avoid data loss in such a case. As our Gorm implementation may distinguish between actual file system remove events and events that result in the unavailability of resources, it publishes a special event type ("inactivate") in the latter case. When Y2 detects this kind of event, it does not remove the respective resource descriptions from the metadata store but rather marks them as *inactive*. Such descriptions are rendered differently in the Y2 GUI and cannot be modified by the users. They can, however, be read and included in any query as usual. Thus, the semantic metadata describing such (currently unavailable) files can still be accessed, something that is, e.g., impossible for internal metadata.

When the respective file tree is accessible again, Y2 synchronizes its index with these files and reports all change events it heuristically detects. Note that this functionality is seamlessly supported by the Gorm approach. It basically means only that a much larger Δt than usual was used for synchronizing this particular file system subtree. As discussed, such longer synchronization times also lead to a loss in accuracy and in turn in more wrongly detected or undetected events. This is, however, only an issue if a lot of changes actually took place on such an external storage medium.

Metadata archiving

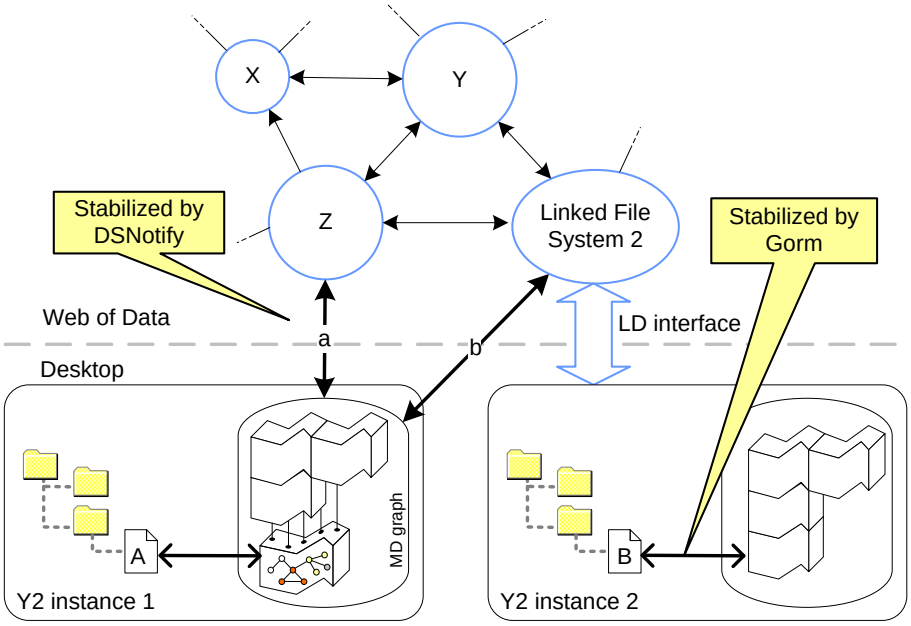
Our current Y2 implementation further includes an experimental component that archives removed metadata records in a local file database (implemented using the H2 database engine[6]). Such archived metadata descriptions are then automatically assigned to newly created metadata records when (i) the names of the removed and the newly created items match and (ii) the similarity between their metadata records is higher than a special "long-term move detection threshold".

[6] http://www.h2database.com/

This feature is useful, e.g., when files are moved out of and later back into the regions that are monitored by a local Y2 instance. In such a case, Gorm detects a deletion event and, at a later point in time, a creation event but cannot associate these two as the time between their occurrences is greater than the used Δt. In this case, metadata loss can still be avoided by restoring the archived record when the above-mentioned conditions are met and the respective file system object did not accumulate too many changes in its associated feature vector.

9.2.3 Linked Data publishing

All the metadata records that describe file system objects managed by Y2 are published as Linked Data. This Linked Data interface basically works as described in Section 4.4[7]. Accessing, for example, the item shown in the screenshot in Figure 9.1a using an "Accept: text/rdf+n3" HTTP header would result in the HTTP response shown in Listing D.1 in Appendix D. This description is an actual merge of the two layers ("core" and "annotation") that Y2 creates by default. As can be seen, the description is rather long as it includes all incoming and outgoing links of the center resource (which includes all parent and child resources). A client that is interested only in a partial description (e.g., only the annotation properties), may negotiate this with the Y2 server using HTTP range headers. Note that the actual file path reference stored in the `y2:path` property is exported relative to the root folder known by Y2 (in this case, the root folder is called "locomotiv"). Not publishing the absolute path here was done to avoid disclosing the local disc layout "below" the root folder to external data consumers.



(a) This figure depicts two Y2 instances. Instance 1 links to the Linked Data cloud, instance 2 is itself remotely accessible via its Linked Data interface and thus becomes a regular "bubble" in this cloud. DSNotify can be used by Y2 to stabilize links into the Linked Data cloud. References to local file system objects are kept stable by the Gorm algorithm implemented in Y2.

Figure 9.5: DSNotify and Gorm enable stable Linked Data publishing of file system resources.

[7] Please note that there might be slight differences between the actual Y2 implementation and the metadata model implementation presented in Chapter 4 as the model was re-implemented to increase implementation modularity and robustness after the Y2 prototype was finished.

9.2.4 MP3 extension

In order to demonstrate how existing, internal metadata can be integrated into a metadata store, we have implemented a simple extension for Y2. This extension subscribes to the central Y2 event log and listens for file creation and update events reported by Gorm. When such events are applied to MP3 or MP4 files, the extension reads the ID3 tags "Album", "Artist" and "Title" from the file headers and updates/creates them in an additional "id3" layer in the Y2 store where they are modeled by RDF properties from the current "Ontology for Media Resources" draft [LeBeBe+11]. MP3 albums and artists are modeled as own RDF resources using automatically generated URIs from the Y2 URI space. A respective RDF representation of an annotated MP3 file can be found in Listing D.3 in Appendix D. Consequently, these metadata are accessible through all Y2 interfaces, including its SPARQL endpoint. The following exemplary SPARQL query can now be applied to such a store in order to retrieve all titles of resources representing an MP3 file that was created by a certain band.

Listing 9.7: SPARQL query that returns the titles of all MP3s that were created by the band "Locomotiv".

```
prefix y2: <http://purl.org/y2/vocab/20110503/y2core#>
prefix mont: <http://www.w3.org/ns/ma-ont#>

SELECT ?t
FROM NAMED <y2id3>
WHERE {
    ?s mont:hasCreator <http://localhost:8081/y2/r/artist_Locomotiv> ;
       mont:title ?t.
}
```

Our simple MP3 extension is depicted in Figure 9.6. Besides a button for actively writing ID3 to the respective file, this extension further provides two buttons for linking this resource automatically to other corresponding resources in other data sources, namely the DBTune[8] and the DBpedia data spaces. This is achieved by using the extracted ID3 tags for the creation of appropriate SPARQL queries that are then sent to the respective endpoints. Y2 then links the local resource to the corresponding remote resources when these requests were successful. By this, the local data is automatically interwoven with the Web of Data.

9.3 Related work

We continue our discussion of Y2 now with an extensive related work section that is grouped by various research areas. We start by providing a general overview of the research related to semantic file systems and continue by relating our work to desktop search engines and Semantic Desktop applications.

9.3.1 Semantic file systems

Semantic file systems enable data access based on an item's semantics rather than on its location in a single hierarchy. This is usually achieved by exploiting (semantic) metadata associated with file system objects. We can distinguish between augmented and integrated semantic file systems respectively semantic extensions to existing file systems [VP96]. Integrated semantic file systems introduce semantic access features directly in the file system interface. As all file manipulations are done via this interface it is straightforward to preserve the integrity between file system objects and associated metadata in such a system: all file system events are known and the system can react accordingly. However, these modified interfaces are not backward-compatible any more. This forces users to migrate to new operating systems and applications that can deal with these new interfaces.

[8] http://dbtune.org/

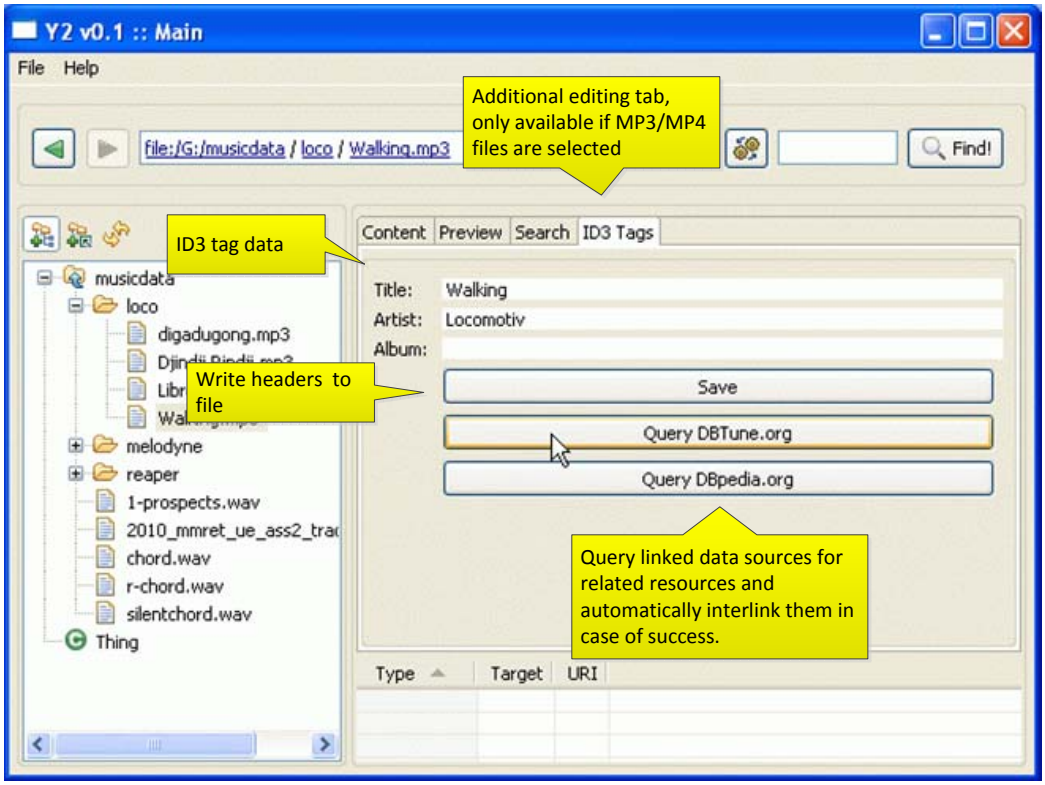


Figure 9.6: Experimental ID3 extension for Y2. Focusing on an MP3 or MP4 file will add an additional tab to the main area of the GUI for accessing some of the ID3 tags stored in the file. These ID3 tags are read by Y2 whenever the file is updated and mirrored into the metadata store. Pressing the "Query ..." buttons triggers a background SPARQL query to the respective data spaces. If a matching resource is returned by this query, Y2 automatically interlinks these two, thereby integrating local and remote content.

Augmented approaches form the significantly larger group of implementations. They augment existing file system interfaces with semantic access methods, while still maintaining traditional access possibilities. This enables a coexistence of new, semantics-aware applications and legacy applications that make use of the file system as usual. This backward-compatibility constitutes a major benefit as it allows for a gentle transition and does not require to (immediately) update existing applications. Furthermore, augmented approaches benefit from all updates/improvements done to the traditional file system layers and related operating system dependent components directly.

In the following, we provide a chronological overview of important projects and prototypes developed in the field of semantic file systems within the past two decades. Most prototypes were implemented as *virtual file systems* (see Section A.2) that are integrated into a desktop environment via network-file system interfaces that are supported by the target operating system's virtual file system layer (common examples are NFS, SMB or CIFS).

SFS – Semantic file system (1991). The term "semantic file system" was probably coined by the authors of [GJSJ91] in 1991. In their paper, the authors define semantic file systems (SFS[9]) as "an information storage system that provides flexible associative access to the system's contents by automatically extracting attributes from files with file type specific transducers" [GJSJ91]. Transducers are components that extract (metadata) attributes from files that are in turn used to access these files. The extracted file attributes are used to determine the contents of virtual directories (cf. Section 2). This is done by mapping a tree-structured (hierarchical) path into a conjunctive query over these attributes. One major limitation of the SFS approach is that it constitutes a read-only file system. Users are not allowed to modify the contents of virtual directories but have to do any modifications in the underlying (traditional) file system. Gifford et al. further listed some other properties a semantic file systems should fulfill (like query consistency, extensibility, etc.). Access to the SFS was achieved by implementing an NFS interface.

Prospero (1992). Prospero [Neu92], was a virtual file system with a strong focus on customizable views of a global, distributed file system. Multiple views on this global file space were possible and these views could be created dynamically as functions of other views. This mechanism allowed grouping of logically related files in a location-free manner.

IFS – Inversion file system (1993). The *Inversion file system* [Ols93] was built on top of the free Postgres[10] RDBMS and stored a file's (chunked) data directly into a uniquely-named table. Inversion provided functions directly derived from respective database functions (transactions, consistency checks, etc.) as well as the possibility to query for files based on associated metadata attributes. Like SFS, the author intended to implement the NFS interface for backward-compatible access.

Nebula (1994). Nebula [BDB+94] was a file system prototype that implemented files as sets of attributes, a concept that is also found, e.g., in Microsoft's NTFS file system. The file content was also stored in a special attribute. Nebula replaced the common file system interface with a database-like interface and enabled dynamic collections of files based on their metadata attributes (*views*). Such views could be named and included in other views, thereby overcoming some of the restrictions of a single-hierarchy organization.

AttrFS – Attribute-based file system (1995). In 1995, Wills et al. developed a prototypical system (AttrFS) that allowed attribute-based access to files [WGM95]. It integrated into a Unix file system using a user-level NFS server. It allowed full read and write access to the stored files and their associated attributes. Users could query for files by building conjunctive as well as disjunctive logical expressions (something not available in previous work). AttrFS provided also access to derived attributes, i.e., attributes that contain computed values (like, e.g., the file *age in days*).

Lifestreams (1996). Other systems such as Lifestreams [Fre97] focused on particular attributes (e.g., creation time) for document organization. In Lifestreams, all documents were organized along a timeline according to the time when they entered the system. This system tried to replace the common desktop metaphor with a general timeline metaphor (a concept that is further developed in [Rek99]). The Lifestream approach was based on simple name/value attributes that were associated with a particular file. Actually, a so-called document record contained attributes that described a file along with a *data* attribute that pointed to the corresponding file on the server (the original Lifestreams implementation was client-server based). Attributes were automatically extracted from documents and users could update these and add additional attributes. Attribute values were indexed for faster search and retrieval.

[9] Not to be confused with the *self-certifying file system* or the *secure file system* or the *SFS file system development toolkit*, all also abbreviated as SFS.

[10] http://www.postgresql.org/

HAC – Hierarchy And Content (1999). HAC [GM99] combines name- and content-based access in the same file system. Related to the work of [SM92], HAC extends the hierarchical organization scheme with a semantic access model while preserving access to the ordinary folder structure. Thereby, the HAC design favors a smooth transition path from traditional file organization towards a semantic access method, a development that is still under way today. HAC was originally implemented on top of the *Glimpse*[11] search engine running on a Unix file system (SunOS) as a user-space DLL which, however, resulted in some performance issues.

Presto (1999). Presto [DELS99] was developed in the course of the Xerox "Placeless Documents" project in 1999. It uses metadata attributes (name/value pairs) that describe documents to provide content-based access. Presto focuses on "high-level document attributes" for the semantic description of objects (examples include "word file", "important", "shared with Jim"). Attribute values are assigned either manually or automatically (active properties) by some software component. Presto uses attributes "as a uniform mechanism for document interaction" [DELS99], this means that no explicit *linking* concept exists in this system. Although simple links could be modeled with attributes (e.g., by storing the target of a link), the system lacks functional support for them and advanced linking concepts (n-ary links, semantically decorated links, etc.) cannot be implemented easily.

Presto supports an advanced collection concept called "fluid collections". Collections are defined by a query, a file-inclusion and a file-exclusion list whereas all three components are optional. A traditional folder concept would, e.g., be modeled with a single file-inclusion list. A virtual folder that contains all recently accessed files (documents) would be modeled by using a query over the last-access attribute of all files. The Presto prototype was implemented with Java, an NFS server provided access via standard file system interfaces. The Presto architecture allows decomposition of the system into a client and a server component, enabling distributed access scenarios. Metadata attributes are stored in a relational DBMS.

Libferris (2001). Libferris[12] is an open-source virtual file system for the Linux operating system. Libferris can treat normal files as folders if it understands their internal format (e.g., it is possible to "mount" an XML file or a GZIP-ed file as a folder subtree). By this, it starts to dissolve the atomicity of single files, similar to the LISFS approach (see below). Such parts of files are also writable via the libferris API. Libferris makes it, for example, possible to mount DOMs (document object models) as libferris file systems. It is also possible to expose, *vice versa*, any libferris file system as a DOM which makes it possible to apply XSLT (eXtended stylesheet transformations) to file systems. Further, a number of applications can be mounted as file systems[13] and the metadata of a file (as stored by libferris) can also be mounted as a so-called branch file system. This makes it possible to access single attributes via the file system API. Libferris can be accessed either by native libferris clients or via SMB; a FUSE variant is also available [Mar04, Mar05, Mar06].

WinFS – Windows Future Storage (2005 - beta). WinFS [Rod04] was Microsoft's attempt to develop an RDBMS-based file system. WinFS was built on top of NTFS and used components derived from Microsoft's SQL Server for storing file-related metadata. A prototype was first demonstrated in 2003 and a beta release running on Windows XP was made available to MSDN subscribers in 2005. However, Microsoft apparently stopped the development due to technical issues and although they claimed in 2006 that development continues[14], the project seems dead by now. Although WinFS was never released in a final version, some architectural considerations were. The basic idea was to add a relational storage layer on top of the well-established NTFS file system that holds additional file-related metadata. WinFS was designed to store relationships as well as metadata attributes that are described by schemas that are easily extensible by software developers.

[11] http://webglimpse.net/
[12] http://www.libferris.com/
[13] See http://www.libferris.com/features for an actual list.
[14] http://blogs.msdn.com/winfs/, accessed August 2008

Developer support seemed to be important for the WinFS designers – several APIs (especially for the .NET platform) were foreseen. It was further planned to support synchronization of WinFS across a network with other WinFS stores as well as other applications and devices.

Relationships were at the core of the WinFS design. They were used, e.g., for modeling the relation between a folder and its files (*FolderMember* relationship). This latter type of relationship was called a *holding relationship*. Holding relationships enabled WinFS to support the concept that a file resides in multiple folders at the same time. A second type of relationship, so-called *referencing relationships*, could be used to model semantic or structural relationships between items. Such relationships were exploited, for instance, for searching and for navigational purposes.

LiFS – Linking File System (2005). LiFS [ABG+06, AMB+05] focuses primarily on attributed inter-file relationships. Attributes are key/value pairs that may contain arbitrary (binary) data such as thumbnails or preview video clips. Attributes may even be *executable*. Special cases are file triggers, executable code that is ran when certain registered operations are invoked on a file. LiFS allows multiple links between files as long as they may be disambiguated. The relationship model of LiFS goes that far that file folders are actually emulated in LiFS by zero-byte files that link to all "contained" files.

LiFS is accessible via an enhanced POSIX[15] interface that extends the standard interface by methods for link and attribute administration. The LiFS prototype was developed as a FUSE (Section A.2) file system for Linux and was designed to be used with magnetic RAM (MRAM) to make the metadata access efficient.

LISFS – Logical Information System as a File System (2003-2006). LISFS [PR03, PSR06] uses file paths as logical formulas that query a logical information system and return the results as directory listings. For example, the path "ls /lfs/music/year:[1980..1990]" would list all available music files from the '80s. The required metadata used for answering this query is extracted from the music files (e.g., by parsing ID3 tags of MP3 files) and indexed in a local database. LISFS was implemented as a FUSE-based Linux virtual file system that augments the underlying ext2 file system. It contains a plug-in mechanism for *logic*- and *transducer* plug-ins. Logic plug-ins are used for the description, querying and classification of objects, transducers play the same role as in the SFS: they extract intrinsic file metadata. Furthermore, LISFS allows explicit addressing of parts or sections of files through a special service (PofFS – *Part of file file system* [PR05]), an attempt to dissolve the atomicity of files on file system level.

TagFS (2006). TagFS [BGSV06] is a virtual file system that focuses on the file-tagging paradigm. Files and folders can be associated with semantic tags that may then be exploited for file management purposes by existing desktop applications or for browsing the file store in an SFS-like fashion. Tags and some basic file metadata are stored in an RDF model. TagFSs can be mounted as WebDAV drives.

9.3.2 Desktop search engines

Desktop search engines directly address the problem that users are unable to recall the exact locations of documents on their desktops and are thereby unable to navigate there. Studies have shown that in principle users prefer location-based (visual) navigation over searching for files in local environments [MC03, BN95]. However, it was also shown that users do integrate keyword-based search into a greater "orienteering strategy" for finding resources on the desktop [TAAK04]. Users want to locate resources based on context, contained text, metadata or semantic information and desktop search engines constitute a reaction to the general shift from pure browsing-based to combined browsing-and-search based access models on the desktop. With more and more data stored on local PCs, desktop search engines gain increasing importance, also, because a robust

[15] POSIX is a family of IEEE standards that define the API, shell and utility interfaces for software compatible with variants of the Unix operating system [Wal95].

and scalable local indexing infrastructure can be used for advanced data organization as demonstrated in projects like *Stuff I've Seen* [DCC+03], where an index of the data a user has seen on her PC is maintained and exploited to help users re-find their data.

Most current desktop search tools work with the file/folder concept and do not directly integrate semantic information about the indexed data items. This basically reduces them to full-text search engines that lack advanced search and retrieval functions as known from Web search. Although desktop search engines basically use the same techniques and data structures for document indexing, they still have to work under different assumptions when compared to their "relatives" from the Web:

Completeness. Web search engines do not index the whole Web. Although the major search engines index a considerable fraction of the so-called "surface Web", the deep Web and many other parts of the Web are not indexed at all. This situation seems acceptable for the Web as people do not know about the other part of the Web they cannot find via online search engines (and many people seem even unaware of the fact that Google does not index the whole Internet). However, incomplete searches are less acceptable in a local environment and local search engines should index all accessible files when they become available (e.g., when a memory stick is plugged in).

Timeliness. Web search engines index the Web normally by using so-called *crawler* (aka *spider*, aka *robot*) applications that irregularly navigate the hyperlink structure of the Web and (re-) index the accessed resources. Such an approach naturally suffers from timeliness-problems: the index and the indexed objects are sometimes *out-of-sync* for days. Such time spans are obviously not acceptable in a local environment where users expect immediate (re-) indexing when files are changed or added.

Lack of contextual information. While Web search engines can resort to a rich layer of interlinked hypertext documents and their embedded media, such a layer is completely missing in today's file systems. Modern Web search engines exploit this hypertext layer, for example, to increase their search and ranking accuracy (consider, for example, the well-known PageRank algorithm). Local desktop search algorithms cannot resort to comparable information.

Generally spoken, local desktop search engines have a much higher need to maximize search precision and recall when compared to Web search engines. However, the lack of contextual information as given on the Web makes this task considerably more difficult. In the following we discuss some prominent desktop search engines.

Windows Desktop Search. Windows (Desktop) Search[16], formerly known as MSN Desktop Search, is the indexed search used in the Windows operating system since Windows 2000. In its latest implementation for Windows 7 it supports features like Natural Query Syntax (NQS) or nested query composition. By using IFilters[17], it is able to extract and index metadata from certain file types (MS Word, XML, and many more). IFilters are plug-ins for the Windows indexing service that know how to read the metadata of particular file formats so that these metadata can be indexed.

Apple Spotlight. Spotlight [Sir05] is Apple's desktop search tool. It indexes files based on their metadata as well as their data and provides a powerful query language on the API level. On the user interface, however, it is (by design) not possible to express complex queries. Spotlight can return lists of file system objects that match the formulated query but can also send notifications to registered system components/applications whenever the result set of a query changes.

One feature of Spotlight is to enable users to store searches as so-called *smart folders*. A "smart folder" is a regular file with a ".savedSearch" extension containing an XML-serialized Spotlight query. The *Finder*

[16] http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx
[17] http://www.ifilter.org/

application, however, displays such files as read-only folders containing the Spotlight query result set. Note that this simple virtual directory implementation requires applications to be aware that a ".savedSearch"-file actually represents a query that has to be resolved by Spotlight.

On an implementation level, Spotlight builds a simple file metadata index based on name/value pairs retrieved by importer plug-ins (one for each file type). This index is then updated by Spotlight based on low-level I/O events handled by the OS kernel.

Linux Beagle. Beagle[18] is an open source desktop search engine for the Linux operating system. It can index documents from various data sources like the file system or an e-mail repository. Metadata are extracted from files using plug-in *filters*, various filters for common file formats including office and text documents, help documents or images (for images, the Exif tags are indexed), are already available. Beagle uses a C# port of Apache Lucene for full text indexing.

Beagle is one of the few Linux applications that make use of the extended attribute mechanism (Section 2). It uses them to keep track of the files that have already been indexed. Beagle further receives notifications from the kernel-space inotify module (see 8.5.4) when files are updated and need to be re-indexed.

Beagle++. Beagle++ [MPC+10] is an extension of the Beagle search engine that uses semantic technologies to improve the accuracy of its search algorithm. It assigns a GUID-based URI to each desktop resource that is indexed and makes use of inotify to keep its indices up-to-date. Metadata are extracted from desktop resources, described using the Nepomuk ontologies (Section C.2.4) and stored in a central RDF repository. Beagle++ also builds full-text indices using Lucene and enables combined (metadata based/fulltext based) queries – the ranking of the results is based on a combination of a full-text TF-IDF score and an ObjectRank [BHP04] score. By exploiting the semantic metadata in search and ranking algorithms, Beagle++ clearly outperforms the Beagle search engine with regard to precision.

9.3.3 Semantic Desktop applications

The main goal of Semantic Desktop applications is to improve *personal information management* on the desktop. This is achieved, for example, by improving search and retrieval capabilities with the help of metadata or by opening local "data silos", e.g., by interconnecting local and remote data such as calendar entries, emails or digital photos [Sch09]. Most current Semantic Desktop applications make use of technologies and standards developed by the Semantic Web community and most implementations employ graph-based data models like RDF or TopicMaps[19] for metadata representation. One central aspect of Semantic Desktop applications is user interface design as the interaction with a (human) user is in the focus of these systems. In the following we briefly discuss important research projects in this area, a detailed survey of current Semantic Desktop projects can be found in [Sch09]. All current implementations are user-space applications that are not tightly integrated with the operating system or the underlying file system.

Haystack (2005). The MIT Haystack project was one of the first to make use of Semantic Web technologies for personal information management (PIM). In [KBH+05], the authors explain how they use RDF for semantically interlinking local items of interest (such as documents, persons or tasks) that are identified by URIs. The main goal of this application is "to improve end-users' ability to store, examine, manipulate, and find their information." [KBH+05]. Haystack extracts metadata (such as ID3 tags or email headers) from considered items for this purpose and integrates them into an RDF metadata graph. This information is then presented to the user by so-called *view prescriptions* that describe what particular properties of an item should be rendered in what visual area. Haystack further supports advanced grouping constructs, such as *lenses* and *collections*.

[18] http://beagle-project.org/
[19] http://www.topicmaps.org/

Semex (2005). The core idea of Semex [DH05] is to improve PIM by exploiting semantic associations between objects. These associations are extracted automatically from multiple sources, such as Word documents, email repositories or LaTeX files. Semex uses a simple, extensible schema for the semantic annotation of objects and their associations. It builds a database of objects and their associations that can be queried by (i) a keyword search, (ii) searching for objects by making use of classes and properties from the schema and (iii) by "association queries" that are similar to SPARQL triple patterns. One particular problem the authors describe in their paper is the need to decide whether extracted references refer to the same real-world object (e.g., to the same person). They propose a simple algorithm that is based on (i) shared keys, (ii) string similarity and (iii) "global knowledge" such as information retrieved from the Web using the Google search engine.

MyLifeBits (2006). MyLifeBits [GBL06] is a Microsoft research project inspired by the famous Memex vision of Vannevar Bush [Bus45]. It is based on a relational database that stores content items including contacts, documents, emails, events, photos, music, and video files as well as related metadata. Items can be related to each other using semantic links, a GUI logger observes mouse and keyboard activities to enable MyLifeBits to reason upon how content items are related to each other automatically. A monitoring process synchronizes local files stored on NTFS discs or Outlook emails with the MyLifeBits storage to enable backward-compatibility.

Nepomuk (2007). The Nepomuk [GHM+07] project aimed at building standards and a reference architecture for a *Social Semantic Desktop*. This project puts a strong focus on collaboration, thus the term "social". The reference implementation stores metadata about local information items in a local RDF graph and further provides a full-text index of these data. Local desktop applications can make use of this local repository for metadata storage and retrieval. Metadata can be extracted from multiple file formats using the Aperture[20] framework. Nepomuk can be distributed in a p2p fashion: an event-based network communication layer exchanges messages between the peers that carry RDF graphs as a payload [RGSH07]. Various semantic vocabularies for describing the stored metadata were developed in this project. We report briefly on some of them in Section C.2.4.

DeepaMehta (2008). DeepaMehta [RP08] is a Semantic Desktop platform that interlinks local items of interest in a large TopicMap. Applications running on the DeepaMehta platform may define arbitrary topic types and properties for describing its domain. They can then add, remove or update respective topics in the TopicMap. Such topics may describe various resources such as text notes, emails or files. DeepaMehta presents a graph-view that visualizes subgraphs of this TopicMap and users can perform operations on the various "topics" via context menus. Users may, for instance, search for related resources and store the query and its result set in the system. They may further directly manipulate the TopicMap via this user interface.

SemDAV (2006-2009). SemDAV [Sch09] is an implementation of the RDF-based Siles concept discussed in Section 4.5.2. This advanced data representation concept was implemented as a virtual file system on top of an RDF store that is accessible via WebDAV and XML-RPC interfaces. A second implementation is based on an in-memory RDF graph storing metadata about files in the local hierarchical file system.

9.3.4 Other related work

WikiFolders. In [VG09], the authors describe a system that allows users to modify the way how folders are rendered by Mac OS X. Their WikiFolders system enables users to annotate the listing of the files contained in such a folder by using a simple wiki markup language. Thereby, it is possible to mark-up important files

[20] http://aperture.sourceforge.net/

visually in order to get a user's attention which is especially useful in environments where user groups share common file resources. Their solution is directly integrated into the Mac OS X file browser which makes this browsing experience seamless. WikiFolders can be used for the annotation of folders only; the markup is stored in a hidden file in the respective folder.

9.4 Discussion

In this chapter we presented Y2, a prototypical application for the semantic annotation of file system resources. Y2 adopts an augmented strategy when considering the related work discussed in the previous section.

9.4.1 An augmented approach

Some approaches for improved data organization on the desktop propagate a radical change to the *status quo*: they completely replace the traditional interfaces of file access (e.g., the file browser) with their own access interfaces. While this simplifies the implementation of advanced user interfaces that exploit the additional semantic information, such systems require users and applications to completely switch to this new way of data access and organization. Although such systems constitute very valuable research prototypes for the development of advanced methods in this field, it is unlikely that such a drastic change will be rewarded with sufficient user acceptance to become successful as also confirmed by the low adoption of respective projects in the past.

Other applications emulate traditional file system interfaces, e.g., via network file protocols or as virtual file systems. One reason for this is that these applications need to be aware of the events that take place in the data sources they integrate. Such events are easily observed when all interactions are done via such an emulated interface. This comes at the cost of reduced I/O performance which is, however, crucial in this regard as the file system constitutes the central storage back-end of current desktop systems. Solutions that considerably slowdown low-level I/O access to file system resources are therefore also unlikely to be successful. Consider in this regard that all attempts to store file data and metadata in DBMS systems have failed in the past despite the enormous effort that was put in systems such as WinFS.

In our work, we have proposed a slightly different architectural concept. Due to the lack of appropriate event notification mechanisms in the file system and in the Web of Data, we have developed user-space components that are able to observe and report events in such data sources. This is in turn used to provide stable identifiers for the respective files or Linked Data resources. These resources can then be annotated by arbitrary metadata descriptions that are stored in multi-faceted semantic metadata graphs.

Our solution provides a gentle transition path for users/applications in the sense of an augmented semantic file system as we do not replace traditional data access interfaces. Applications and users may still access file system objects and their low-level metadata as usual (e.g., via its API or via a file explorer application). They may also resort to higher-level metadata stored in file headers as they are used to. Additionally, however, they may access one or multiple Y2 instances in order to retrieve additional semantic metadata describing a file system object or a group of such. Such semantic metadata as well as the ability to stably and semantically link the respective resources with each other is at the core of most research prototypes that were presented in the previous sections.

9.4.2 Applications for Y2

Y2 enables users to annotate their local file data in a stable way using external, RDF-based metadata descriptions. The Y2 GUI, however, is just one example for an application that makes use of a metadata store as described in this thesis, many others are possible. We briefly describe two alternative application scenarios in the following:

Music player. Consider a music player that implements an extended version of the MP3 component presented in Section 9.2.4. This player could exploit internal and external metadata for presenting local music files to the user.

It could further easily integrate local audio files with remote Linked Data sources as demonstrated. This could then be exploited, e.g., for the retrieval of textual, music-related descriptions from DBpedia. An example for exploiting links into a linked music data set could be that such a music player may retrieve the names of all tracks belonging to a local audio file's album from there and present this list to the user, highlighting all entries that are found on the local system.

Play-lists (cf. Section 3.3.1) in such an audio player would be expressed by record groups as described in Section 4.4.4, broken links in these play-lists would be avoided by Gorm.

Email application. As another example, consider a mail program that would have access to a metadata store as described in this thesis. This email application could easily store links between local emails and, e.g., attachments stored as external files in one layer of this store. By this, it could enable users to quickly access these files and may avoid unnecessary digital copies of email attachments in the local file system. This would be especially beneficial in environments with strongly limited storage space (e.g., on mobile devices). Users could then immediately search the store for emails with attachments of a particular mime-type and size or for emails that are related to other emails; they could further query the store for all emails that have a particular file as attachment.

Many scenarios like these two are possible and should motivate how a Y2-like file metadata store can be used as a foundation for applications that require semantic, file-related metadata on the desktop. As another example, semantic file system and Semantic Desktop applications as discussed above would benefit greatly from such a store that liberates them from the need of persisting file-related semantic information.

9.4.3 Limitations

The spectrum of desktop applications that may use such a metadata store is basically limited by performance and scalability of the metadata store implementation.

Scalability. The number of triples contained in a Y2 metadata store can be estimated by multiplying the number of considered files, i.e., $|\mathcal{F}_{\mathbb{S}}|$, with the average size (in triples) of a metadata record. The results of our study presented in Chapter 7 gives an idea about the expected numbers of file system objects on regular PCs. For example, annotating 200,000 files on such a system with a record of, say, 500 triples each would require such a store to efficiently manage 100 mio. triples which is not an issue for today's triple store implementations that claim to scale up to several billion triples[21]. It is noteworthy, that all layers of the current Y2 metadata graph were implemented as named graphs managed by a regular triple store implementation. It would, however, also be possible to store the respective metadata in other storage systems (e.g., in a RDBMS or in a NoSQL storage) if a mapping to the corresponding RDF descriptions could be defined. The higher scalability of such systems would be paid by the loss of native SPARQL support.

A further limiting factor is the execution of Gorm. Although the described domain-specific methods (TIBB, native change detection, subtree move operations, etc.) reduce the required efforts significantly, event detection by Gorm is not for free. A particular problem for this algorithm are constantly changing files (e.g., log files) that force it to cycle through its main loop all the time (cf. Figure 8.10). A strategy for avoiding this might be to actually exclude such files from the set of files considered by Gorm (there might be few use-cases for attaching semantic metadata to such files anyway). For now, such files may be excluded manually via the configuration (e.g., by excluding all ".log" files), as discussed.

[21] See http://www.w3.org/wiki/RdfStoreBenchmarking for a list of up-to-date benchmarking results.

Performance. Performance is a central issue for file system-related functionality. This obviously includes access to file-related metadata as described in this thesis and we do not claim that our implementation can compete on these grounds with any low-level file system implementation. It is, however, not our intention to replace such tools but to rather augment them with various layers of semantic metadata (mirroring low-level file metadata into the core layer of Y2 is done to (i) implement a feature vector index as required by Gorm and (ii) to enable SPARQL queries on these data). Accessing semantic metadata in Y2 is considerably slower than access to the metadata stored in the file system itself; however, this is also the case for the current strategy of storing these metadata in internal file headers.

Gorm accuracy. Yet another limitation is the accuracy of the used algorithm for the detection of local file system events. As described, our heuristic identification of files based on feature vectors derived from their low-level metadata is not error-free, although we measured high accuracies in our evaluation in Section 8.4. This means that our algorithm might report false positive and false negative move events to Y2. False negative move events would result in lost metadata records (which can partly be avoided by the metadata archive discussed above) while false positives result in the association of existing metadata records with wrong file system objects.

9.4.4 Summary

Y2 assembles our proposed building blocks and acts as a prototypical demonstrator of how semantic data organization on the desktop might be implemented on top of our data model presented in Chapter 4. In a summary, the main use cases handled by the Y2 GUI are:

i) Management of a set of local file system objects that may be augmented with semantic metadata.

ii) Annotation of these resources with formatted hypertext.

iii) Creation of arbitrary semantic links between local and remote resources.

iv) Management of a SKOS concept scheme useful for organizing local content.

v) Querying these metadata via the GUI and via a SPARQL endpoint.

vi) Linked Data access to all stored metadata descriptions.

vii) Editing of mounted remote resource descriptions.

In a summary, this functionality should demonstrate a way how to overcome the current limitations for data organization on the desktop (Section 1.1) with the help of semantic metadata:

First, Y2 allows users to incrementally create a hypertext layer on top of their local, hierarchically organized file resources. This layer enables alternative navigational paths between local file system objects but also remote resources in the Web. The integration with operating system components (e.g., the file explorer) enables users to quickly switch between the traditional navigation in the hierarchical file system and this additional, associative navigation in the hypertext layer. Furthermore, the Y2 GUI allows users to search this hypertext layer which, in a summary, enables an *orienteering* strategy of small navigational steps as described in [TAAK04].

Second, the layered metadata model based on Semantic Web technologies that is implemented in Y2 enables advanced access to the contained metadata via uniform and standardized interfaces. All kinds of file system objects can be associated with arbitrary, external metadata records which overcomes the limited metadata support of current file systems.

Finally, the mentioned hypertext layer constitutes a kind of semantic overlay graph. Users may describe their local file system objects semantically and unambiguously by using standardized semantic vocabularies.

Stable identifiers are assigned to local file system objects irrespective of their location in the local file system hierarchy. This leads to a continuous integration of local file data with various data spaces in the Web of Data and ultimately into one global information system.

Other than currently proposed solutions that deal with file system objects, we favor an augmented approach that complements rather than replaces the current ways of metadata handling on the desktop. This is possible due to the integration of algorithms that preserve the integrity of relationships between files and their external metadata records independent of the used operating system or file system implementation. The presented solution cannot, however, be considered as a semantic file systems or a Semantic Desktop application on its own. Neither does Y2 itself implement a file system interface nor does it consider other information items occurring on the desktop, such as emails or contacts, for now. The presented technology can rather be seen as a foundation for such applications that may use the multi-layered metadata store of Y2 for storing and accessing the semantic metadata they need. The Y2 prototype presented in this chapter, however, demonstrates how the possibility to uniformly associate and access arbitrary semantic metadata descriptions with file system objects can be exploited for advanced data organization on the desktop.

In the following final chapter, we summarize and discuss our contributions and their limitations as discussed in this thesis.

Chapter 10

Conclusions and Outlook

10.1 Summary of contributions

In this thesis we identified and discussed several building blocks for the implementation of semantic data organization methods on the desktop. We presented own implementations of these building blocks and discussed advantages and limitations of our approaches. In a summary, the following contributions have been presented in this dissertation:

i) We introduced a novel metadata model for the desktop that allows the semi-structured representation of complex, file-related metadata records using Semantic Web technologies. This model:

- supports semantic links between resources,
- enables separate access to data and metadata,
- introduces an explicit compartmentalization concept that may be used to model and access partial aspects of the described resources, and,
- enables data integration into a global information system.

ii) We developed two methods that preserve stable links between resources in distributed (Web of Data) and local (desktop) environments. Our contributions:

- preserve link integrity between files, metadata records and Linked Data resources,
- show the feasibility to identify resources based on a heuristic comparison of explicit metadata in the discussed domains, and,
- show the feasibility of a time-interval-based blocking approach to deal with expected event frequencies in the respective environments.

iii) We presented a prototypical demonstrator that makes use of all developed components and:

- represents a centralized storage of external, file-related metadata records,
- implements a wiki-like interface for the textual annotation of arbitrary file system objects,
- enables semantic interlinking of arbitrary desktop and Web resources,
- allows the development of semantic vocabularies and their adoption for data organization,
- implements a machine-friendly Linked Data read/write interface that enables data exchange with other Y2 instances and other services,
- demonstrates backward-compatibility by integration of existing, internal metadata, and,
- shows how such a system can be exploited for complex data organization and querying.

10.2 Discussion

Desktop systems are the central hubs of data and metadata consumption and manipulation today[1]. They are, however, also the origin of many valuable metadata describing files and their semantic contexts. Such semantic metadata can be exploited in manifold ways:

i) For improved search and retrieval interfaces on the desktop.

ii) For the implementation of novel, semantic access methods.

iii) For the integration and exchange of local and remote data.

iv) For data provenance related applications.

In a summary, they can be used for the realization of advanced data organization concepts on the desktop as they are already known from the (semantic) Web. However, these metadata are to a large part extrinsic in nature, which means that they have to be captured at the time they are created, stored and stably associated with the files they describe.

A common low-level basis for data storage and access on desktops is the file system: data is partitioned into files that are organized on a storage medium in a certain way that allows their efficient access. Different devices, however, differ in the used file systems (e.g., because of the used storage hardware, operating system, etc.). As a consequence of this, files often have to cross file system boundaries. Consider, for example, a photo that is copied from your digital camera or smartphone (SD cards are usually formatted in FAT) to your PC's hard disc (e.g., formatted in HFS+ or NTFS) where it is manipulated and then uploaded to a Web server that stores the data in an ext4 file system.

The currently predominant way how metadata are associated with such files is by storing them in internal file headers, despite the multiple shortcomings of this solution as elaborately discussed in this work. The different support of prevalent file systems for advanced file (meta) data concepts such as *extended attributes* or *file forks*, however, hinders the development of *external* metadata models: moving files between file systems that differ in their support for these mechanisms results in (meta) data loss.

This thesis describes an alternative model for capturing and representing external, extrinsic metadata on desktop systems. The presented model is based on external metadata records stored in a central repository that are linked to the respective file system objects but also to remote resources. The stability of these links is a precondition for such a solution. However, we showed under which conditions the locations and by that the available resource identifiers (file, respectively, HTTP URIs) of considered resources do change. The main part of this thesis is dedicated to this problem and we proposed two algorithms that enable stable links to resources based on the detection of change events that take place in their respective data spaces. At their core, both discussed algorithms are based on a heuristic comparison of feature vectors that are derived at different points in time from respective resource metadata descriptions. They further have in common that both split the task they have to solve into many small subtasks which enables their efficient execution. Both components are, however, specialized in their respective domains which differ in their data characteristics and dynamics as well as in coverage and entropy of the available metadata.

Our model allows capturing and exchange of extrinsic metadata in multi-layered, semantic graphs that may be accessed via uniform, standardized interfaces. It does not completely change the existing ways of metadata handling on the desktop, neither does it require changes of existing file system interfaces. It rather demonstrates how a user-space application can be used for augmenting the desktop with semantic metadata that is stably attached to local resources. Our results presented in this thesis indicate that such a strategy is feasible. In the long run, however, the author believes that the support for arbitrary, external metadata has to

[1] It is important to remember that by desktop system we consider not only classical desktop computers (i.e., stationary devices) but any general-purpose computer that is directly operated by an end user as defined at the very beginning of this thesis (page 3). This includes also mobile computers such as laptops or palmtops and even modern smartphones.

become directly integrated into the lower levels of a computing system, e.g., into the file system. Metadata and links play the central roles in advanced data organization and interoperability concepts. Therefore, end-user applications should be able to resort to a stable, flexible and efficient metadata storage that disburdens them to implement all basic metadata access and management functionality themselves. The availability of such metadata stores would constitute a major step towards more advanced concepts for dealing with the ever-increasing amounts of data stored in desktop systems as well as towards the integration of these data into a global, semantic information system.

10.3 Limitations and future research directions

In this final section, we list limitations along with possible future research directions related to our presented work. Note that specific limitations of the individual components are described in more detail also in the respective chapters of this thesis.

Content-based features. A central limitation of our heuristic event detection approaches is that errors cannot be completely excluded which might not be acceptable in certain scenarios. As briefly discussed in this thesis, content-based features might be used to further increase the accuracy of these methods. The feature vector considered by Gorm for the identification of local file system objects, for example, is currently built solely by low-level features that are directly accessible via the file system interface. One way to further increase the accuracy of the employed method is to additionally resort to higher-level, intrinsic metadata of the considered files. These could be values stored in internal metadata headers as well as features directly extractable from an item's data representation (e.g., the headline texts of this document). The incorporation of such features for some file system objects requires changes to the similarity estimation algorithm that allow it to deal with incomplete feature vectors.

Parameterization. Note that the introduction of new features will also result in additional method parameters that have to be determined somehow. As described in Section 6.6.3, the current DSNotify algorithm already suffers from this problem. Consequently, one future research direction could be the development of an automatic parameter estimation approach that takes the described measures (coverage, entropy/collision probability) as well as content-specific data set dynamics into account.

Co-reference resolution. This thesis and its related work on semantic file systems and Semantic Desktops show how the Semantic Web technology stack (including RDF, OWL and SPARQL) can also be used for metadata organization in local environments. The integration of the desktop and the (semantic) Web is an ongoing process and at some point in time the borders between these two "worlds" might become completely dissolved. It is often read that the ability to uniquely (re-)identify each thing in such a global data space with one, single digital identifiers (e.g., a URI) is a precondition for this. It is, however, the author's opinion that such a state will never be reached and that co-reference will always be an issue in a global information system. This requires different ways for the re-identification of items respectively for identifying the same "referent" of multiple identifiers. The approach discussed in this thesis is to consider not only a single (identifier) property, but a whole set of properties of an item and compare them with some memory state of an item's representation. In some way, this might be close to how humans deal with this problem: things are usually identified based on a whole set of (visual, tactile, etc.) characteristics rather than based on a single, unique identifier (e.g., their DNA sequence). Co-reference, i.e., the problem that multiple such descriptions refer to the same *thing* is an inherent issue of such approaches. As a consequence, all research contributed to the issue of co-reference resolution stemming, e.g., from linguistics, computer vision but also from semantic Web research (cf. Glaser et al.'s work on this topic in [GLMD07, GJM09]) should be considered when further developing a global data integration scenario.

File feature dynamics. Knowledge and statistics about the dynamics of the features considered by our methods is crucial to reach satisfactory accuracies as shown in this thesis. The study presented in Chapter 7 constitutes a first step in this direction; however, further observations of feature value distributions and their modifications in real-world environments might provide additional insights in how to design accurate feature value comparison functions.

Human computation support. Our algorithms can be extended to include human intelligence when required. An application that makes use of our algorithms could, for example, ask the user for assistance in the disambiguation of move-event predecessor/successor pairs in scenarios where high accuracies are required. We have discussed such a mechanism briefly in Section 6.5.4.

File type identification. Not all file system objects in a system might be annotation-worthy. Consider, for example, application data files: an application might be interested in events related to such files, but not in actually attaching metadata to them. This could have an impact on the design of user-interfaces that distinguish between such files and, e.g., user-created documents. The automatic detection of whether a file was created by a user or by some application might be interesting in order to find out what parts of a local file system actually contain annotation-worthy files. Further, we discussed that highly dynamic files such as log files lead to continuous event (update) streams and basically keep the Gorm algorithm in a busy-loop. Currently, such files have to be excluded manually from Gorm, an automatic detection and classification of such files would be desirable.

Duplicate detection. Digital copies of files do usually *not* result in equal feature vectors (e.g., different file names, creation dates, etc.). Nevertheless, it might be straightforward to include a duplicate detection algorithm based on file checksums into Y2 that could interlink the respective RDF resources using, e.g., `owl:sameAs` links. Such links could then, for instance, be exploited to automatically display related versions of a file.

File fragment support. Our model allows attaching arbitrary complex metadata descriptions to file system objects or groups of such. However, it does not support to reference information entities embedded in a file. Although multimedia data and documents are often "self contained", i.e., they are stored in one single file that does not contain any other information items, it might be useful to annotate fragments such as a region in a digital image, a cell in a spreadsheet or a paragraph in a text document. Such functionality might be equally relevant for data files (e.g., an electronic address book, a file containing emails, a database file).

Advanced querying and ranking. It might further be possible to use our event detection algorithms for the implementation of advanced search methods such as *query by example*; however, this might succeed only when content-derived features are also included in the feature vectors. In such a scenario, a user would possibly drag-and-drop a file to a search interface. Then, the feature vector of this file would be assembled and compared to all existing vectors in the index. As a result, all file system objects above a certain threshold would be presented, possibly ranked by their similarity to the query file.

Discovery. A currently missing feature of our presented implementation is the automatic discovery of other (local or remote) metadata stores. A store should thus register as a service to some registry (e.g., to a Zeroconf service[2]), including a pointer to its configuration resources. We have discussed some possibilities for this purpose, including the use of PURLs as stable service identifiers, in [PS10].

[2]Zero Configuration Networking, `http://www.zeroconf.org`

Implementation. In the end, however, the ability to implement the described metadata layer in a stable, scalable and high-performance way will decide whether such an approach will gain sufficient user acceptance. The author considers this thesis as a first starting point for such an implementation. One obvious improvement to the presented design is the introduction of caching mechanisms for item descriptions, both on the server side but possibly also embedded in a file's forks or metadata headers. In distributed environments, more advanced caching strategies will be needed. Further, concurrency and transaction control as well as other data integrity and security issues have to be considered for a real-world implementation of such a system. All this might, however, be rather straightforward when resorting to common database technology as a storage layer that already implements respective features. Further, support for these features is already available in some triple store implementations and further improvements in this direction can be expected.

10.4 Epilogue

Some parts of this thesis are based on research papers that were published in peer-reviewed journals, conference and workshop proceedings during the past four years. Chapter 4 is based to a large part on [Pop11]. Chapter 5 and Chapter 6 are mainly based on [PH11], which is in turn an extended version of [PH10, PHR10, HP09]. Section 6.8.1 was published partly already in [SP10] and in [PS10]. Once again, the author wishes to thank all contributing co-authors, all reviewers and all persons specially acknowledged in these papers for their collaboration as well as for all the invaluable discussions on the topics discussed in this thesis and, last but not least, for all the good times we had together.

Niko Popitsch,
Vienna, August 2011

Part V

Addenda

Appendix A

Prevalent file systems

In this chapter, we briefly discuss current file systems relevant to private computer users with a focus on their metadata capabilities and relevant implementation details. File systems are used for data organization on physical storage devices (e.g., a hard disk or an optical storage medium), they may provide access to remote resources via network protocols (NFS, AFS, SMB, CODA, etc.) or they may be completely virtual such as the process file system *procfs*, a pseudo file system that is used to access process metadata from Unix-like kernels (cf. Section 2).

All modern file systems provide support for the explicit representation of file-related metadata of various kinds, ranging from file names to image thumbnails. Particularly, most modern file systems support concepts similar to file forks and extended attributes that were introduced already in Section 2. As these concepts can be used for the association of arbitrary metadata with file system objects, we continue here by listing important file systems and discussing how these concepts are implemented.

FAT – File Allocation Table (1976)

FAT was introduced in 1976[1] and was the primary file system for older DOS/Windows systems developed by Microsoft. Nowadays it stands for a whole family of file systems (FAT12, FAT15, FAT16, FAT32, VFAT, exFAT). FAT still constitutes a wide-spread file system (mostly in the form of FAT32), mainly because of its compatibility with all prevalent operating systems and because of the little management overhead it introduces. For these reasons it is nowadays used on most portable digital devices, including USB sticks, flash drives or memory cards for digital cameras and there is still ongoing research that tries to overcome certain shortcomings of FAT (e.g., FATTY [AKXH07]).

Microsoft has further introduced the extended FAT file system (exFAT aka FAT64) that resolves some major limitations of FAT32 (e.g., it increased the file size limit and scales better on large storage volumes) [Mic08]. ExFAT is especially well suited for embedded devices, it was introduced with the Windows Embedded CE 6.0 and is being recommended for flash memory storage and other external devices. As all other FAT implementations it does neither support alternate data streams nor extended attributes. This raises the question how to deal with data copied from file systems that make use of such features and different strategies were developed to solve this issue. Mac OS X, for example, stores the resource fork data of HFS/HFS+ files in hidden files on a FAT volume, OS/2 and Windows NT make use of unused bytes in FAT directory entries for linking to a chain of clusters that hold the extended attributes for that file[2].

[1] Dates are official release dates of stable file system versions. Sometimes file systems are pre-released (e.g., in an open-source community), so the dates may differ by about one year from known release dates.

[2] See `http://www.tavi.co.uk/os2pages/eadata.html` (Accessed June 2008) for a more detailed discussion of this topic.

HPFS – High Performance File System (1989)

The HPFS [Dun89] was developed for the OS/2 operating system and included significant enhancements compared to FAT, among them support for longer file names and generally for larger random access devices. While HPFS stores all standard file-attributes of FAT (like the read only, hidden or archive flags), it also supports the storage of file-related name/value pairs that were called "extended attributes" (cf. Section 2) and are accessible via a simple API. One intended purpose for extended attributes was to store references to related/dependent files or binary information like file icons. HPFS supported 64 KiB extended attributes per file but had no support for a concept comparable to forks/alternate data streams.

NTFS – New Technology File System (1993)

In 1993, Microsoft presented NTFS [RC98, Esp00, Tan01, Mea03, BB04], the arguably most wide-spread file system on the PC market today. In NTFS, files are containers for multiple *streams*, each stream including a data section (for storing, e.g., a file's data) and an attribute section (for storing stream-related metadata). Additionally, some metadata are used to describe the whole file (i.e., the set of streams), including the file names[3], extended attributes, security descriptors, reparse[4] and indexing information and standard file system metadata (timestamps, flags).

The "primary (unnamed) data stream" of a file is used to store the data that is usually processed by data-consuming applications while alternate streams (ADS) may contain related data such as thumbnails or any other binary data. NTFS alternate data streams can be compared directly to HFS' file forks (see below) on a functional level. In fact, they have been used to enable NTFS machines to act as servers for Macintosh clients. ADS may also be associated with directories and empty files (i.e., files with an empty primary data stream), nested streams are currently not supported by NTFS. All *file system level metadata* that is required for their manipulation (e.g., allocation size, actual data size, etc.) are associated with each primary or alternate data stream. On this lower level, ADS support is transparently implemented by the Windows API, but higher-level utilities like *DIR* or even the *Windows Explorer* are not ADS aware, meaning that they do not provide functionality for accessing each alternate data stream. Newer Windows versions ship with several tools for handling ADS and extended attributes[5]. ADS have been used since Windows 2000 to store some general file metadata, such as file title, subject, author, and, in the case of images, even thumbnails. These metadata can be accessed via a tab in the file-properties view (cf. Figure A.1). They are further used to store the NTFS change journal. Alternate data streams can be accessed by expanding the name of a file in the following way:

```
<filename>:<ADSname>:<ADStype> (where ADStype is optional).
```

A simple way to add/edit a file's ADS is, for example, to use the Windows notepad tool:

```
notepad ADS-textfile.txt:hidden.txt
```

Although a very powerful feature, ADS are rarely used in practice. This includes Microsoft's own tools and APIs (e.g., the .NET framework) as well as, e.g., many anti-virus utilities[6]. There have been quite a lot of discussions whether ADS constitute a considerable security leak, as it is possible to "hide" valuable information or even malicious executables in ADS. Another noteworthy issue with ADS is that most file integrity tools do not include them in their algorithms and thus provide equal hash-sums for files sharing the same primary stream content but having different alternate data streams. However, the most important reason why ADS are not widely used is the lack of compatibility with older file systems that are still in use. When

[3] A file can have multiple names, e.g., when hard links (see Appendix B) are used.

[4] See http://msdn.microsoft.com/en-us/library/aa365503(VS.85).aspx

[5] *RoboCopy*, for example, is a Windows file copy tool that preserves extended attributes, see http://technet.microsoft.com/en-us/magazine/ee851678.aspx.

[6] Some anti-virus scanners include an ADS-scanning option that is usually disabled per default and has to be actively turned on by the user.

data is copied from an NTFS to a FAT volume, for example, only the primary data streams are copied, the alternate streams are lost[7]. The same data loss occurs when, e.g., sending such files as e-mail attachments or reading and writing them with other applications that are not ADS-aware. Nevertheless, Microsoft itself uses ADS successfully for caching recoverable metadata, such as image thumbnails as mentioned above.

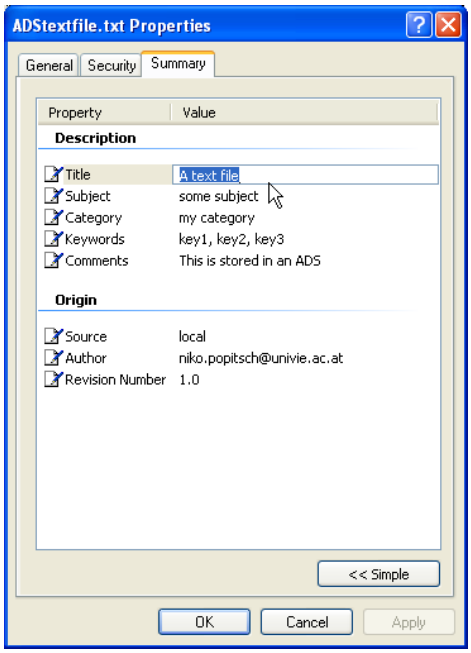


Figure A.1: Windows NTFS summary metadata, stored in an ADS.

HFS – Hierarchical File System (1984)

Apple introduced the concept of file forks with their HFS file system [Mon92]. Each file in HFS has two *forks*, a *data* and a *resource fork*. The resource fork contains a file's resources, e.g., icons or windows locations, and has a predefined structure consisting of a resource map followed by the actual resource data. The data fork contains the file's data.

HFS supports empty data forks as well as empty resource forks. The maximum number of possible resources is restricted to 2700 and they are searched linearly making resource access slow when many resources are attached to a file. Another limitation is that the resource fork cannot be accessed by multiple users of a shared volume (which is possible for the data fork).

HFS+ (1998). HFS+ was introduced with Mac OS 8.1 in 1998 and contains a number of improvements when compared to HFS, such as long (255 characters) file names, smaller allocation blocks or larger maximum file size [Sir05, App04]. More importantly for this discussion, HFS+ prepared the way for the association of arbitrary numbers of metadata to files by introducing a mechanism for storing extended attributes held in a B-tree. Starting with Mac OS X 10.3, so-called *attribute files* contain *fork data attributes* or *extension*

[7] As regular Windows tools such as the file Explorer or the `del` command are unable to remove ADS, copying files to a FAT volume and back is the actual method recommended by Microsoft to get rid of a file's ADS. See `http://support.microsoft.com/kb/319300` (Accessed July 2010).

attributes. Fork data attributes are used for storing large amounts of associated metadata, extension attributes extend such fork data attributes to allow for an arbitrary number of forks. Because of technical issues with heavy-weight named forks, Apple made other use of attribute files and finally introduced *extended attributes* with their next OS version "Tiger" (10.4) in 2005. Unfortunately, some major limitations regarding extended attributes in HFS+ still seem to exist: attribute names cannot be longer than 128 bytes (UTF-8 characters) and the values they store are 4KB of arbitrary data (maximum size recommended by Apple). As applications share a single, flat attribute namespace, the length of the attribute names might become a practical problem: conventions between applications that make use of extended attributes are needed to avoid collisions. Further, API access to the extended attribute features of Mac OS X is possible only at the low-level BSD level. Nevertheless, HFS+ extended attributes were used by Apple for the implementation of file access control lists (ACLs). On volumes that do not support extended attributes, these data are stored in "._"-files [Ibid.].

BFS – Be File System (1997)

BFS is a 64-bit journaling file system developed by Dominic Giampaolo and Cyril Meurillon for the Be operating system [Gia99]. In BFS, any number of (extended) attributes can be associated with a file. The attributes are typed and if integral data types (int32, int64, float, double, string) are used, these attributes can be indexed for efficient search and retrieval purposes. BeOS takes multitudinous advantage of this mechanism. It is, for example, used by the email client to store metadata about emails that are then indexed by BFS[8]. BFS also creates indices for the name, size and last modification time of all files, making queries of the form "last modified since yesterday" fast and reliable. Indices are stored as B+ trees in regular files in a (hidden) index directory. When no integral data type is used, attributes may contain simply "raw data" of arbitrary size. In this regard, such attributes are comparable to forks/alternate data streams.

UFS – Unix File System (1993)

The Unix file system (UFS; aka BSD FFS) can be considered as the ancestor of most current file systems. It introduced major performance, reliability and functional improvements when compared to older Unix file systems. BSD FFS introduced the concept of symbolic links (see Appendix B) as well as long file names and file locking [MJLF84, Gia99].

Ext2 – Second Extended File System (1993) and related Linux file system

Due to its robustness, the *Second Extended File System* [CTT94] file system is the current *de-facto* standard of Linux file systems. Ext2 provides standard Unix file system semantics and added access control lists (ACLs) for more fine-grained access control. Ext2 supports extended attributes when a certain package is installed and the file system is prepared accordingly. Such extended attributes are restricted in size (64KB).

Ext3, ext4, XFS, JFS, ReiserFS. Ext3 is a further development of the ext2 file system. It is a journaling file system that allows the file system to grow dynamically and adds special support for large directories. Ext3 is fully backward-compatible with ext2. An enhanced version of ext3 called *ext4* was released in 2008. Ext4 is also backward-compatible with ext3 and ext2 and mainly improves it by better support for large files and file systems. Other commonly used Linux file systems include the journaling file systems *XFS*[9], *JFS*[10] and *ReiserFS*[11], all of which support extended attributes for any regular file[12].

[8] In BeOS, emails are stored in individual files. The email daemon creates indices for the *From*, *To* and *Subject* attributes attached to these files.

[9] http://oss.sgi.com/projects/xfs/

[10] Journaling file system; http://jfs.sourceforge.net/

[11] Except for the unfinished Reiser4 which introduced its own interface for extended attributes.

[12] Extended attribute (*xattr*) support in Linux is only available if the *libattr* option is enabled in the kernel configuration. Further information about extended attribute support on Linux file systems is available at http://acl.bestbits.at/

ZFS – Zetabyte File System (2006)

A quite recent file system implementation is SUN's ZFS [Sun06], the first industrial-strength 128 bit file system. One of its major contributions is that it moves away from the (traditional) volume-based storage model to a more flexible concept based on *data pools*. With data pools it is possible to add/remove storage space to a pool, comparable to adding/removing memory via the well-known *malloc/free* operations.

The ZAP (ZFS Attribute Processor) is a file system module that manages *ZAP objects*, used for storing attribute values in a name/value form (extended attributes). *Names* are zero-terminated 256 byte (maximum) strings; *values* are arrays of integer values limited in size only by the maximum ZAP data block size (128KB). There are two ZAP object implementations in ZFS, designed for fast access (*microzap* objects) respectively large number of attribute values (*fatzap* objects).

ZFS provides a layer (ZPL - ZFS POSIX layer) that makes it look like a POSIX file system and although ZFS was designed for the server segment it could become interesting for the end-user market in the future due to the ever growing storage requirements.

CD-ROM and DVD file systems

ISO 9660 (1987). ISO 9660 is the standard file system for CD-ROMs but is used on other optical disks as well. ISO 9660 supports extended attributes in the form of *extended attribute records*. An extended attribute record can be associated with any file or directory and stores predefined file-related metadata like file creation/modification dates and UNIX-style access permissions as well as application-specific metadata attributes. The standard does not constrain this "application use" section except for its maximum size of 2^{16} bytes.

ISO 9660 supports also an *associated file* concept that is compatible with the file fork concept. An associated file has a relationship to another file in the same directory – both files share the same file identifier, but one is marked as the associated file, the other is not. A file can have only one associated file, the concept is analogous to the HFS data/resource fork model [ISO98].

Several OS providers have developed extensions to ISO 9660, mainly for compatibility reasons: Microsofts *Joliet* (1995) is an extension that supports longer file names and non-ASCII character sets (e.g., Unicode) and is supported by most current PC operating systems. Further extensions include *Rock Ridge* (long filenames, Unix-style symbolic links), *El Torito* (for bootable CDs/DVDs) and *Apple's ISO 9660 extension*.

UDF – Universal Disk Format (1996). The UDF file system is the industry-standard format for storing information on DVD. It is an implementation of the ISO/IEC 13346 standard (also known as ECMA-167) and should supersede ISO 9660 as the standard file system for optical disks in the future.

UDF supports extended attributes as well as so-called "named streams", a concept comparable to file forks and ADS. UDF named streams are not limited in size, hierarchical named streams are not supported and named streams cannot have extended attributes [OST05].

A.1 Special purpose file systems

DCF – Design rule for Camera File system (1998)

DCF [JEI98] is a specification by the Japanese electronics and information technology industries association (JEITA) that also developed the Exif standard (see section C.1.1). DCF aims at simplifying the exchange of images between digital still cameras (DSC) and other equipment (e.g., printers). It is based on the Exif 2.2 specification and can be considered as the *de-facto* industry standard for DSCs.

DCF is not a real file system but rather specifies how files and directories should be named and structured on (removable) FAT formatted storage volumes. In DCF, related files (e.g., an image file, its thumbnail and a related audio file) may be associated with a common file number that is part of the file name (but the files have distinct extensions). While this strategy works in this specific, closed domain (software for DSCs is mainly provided by the manufacturers themselves) it also introduces several limitations (only one thumbnail per object, a limited number of files per directory, etc.) that follow from this design that is based on directory layout and filename conventions.

JFFS, JFFS2 – Journalling Flash File System (1999, 2001)

The Journaling Flash File System [Woo01] is a log-structured file system that was developed to fit the specific needs of flash technology including "wear leveling" (a technique for prolonging the lifetime of EEPROM and flash memory devices), small footprint and reliability. Currently, JFFS does not support forks, but there is some support for extended attributes according to the JFFS mailing list[13].

A.2 File system virtualization

VFS – Virtual File Systems (1985). Most modern operating systems implement a so-called Virtual File System (VFS) concept (e.g., the Windows *Installable File System*, the Linux *Virtual File System* [CTT94] or the BeOS *vnode layer* [Gia99]), basically consisting of a kernel-level software layer that handles all file system calls and translates them before sending them to the mounted (real) file systems. Sun Microsystems introduced their VFS interface in 1985 in order to be able to mount multiple diverse file systems in a single tree. Figure A.2 shows the integration of two file systems (UFS and NFS, see also below) using a virtual file system layer.

A VFS introduces a "common file model" that is supported by all mountable file systems thereby also reducing the possible functionalities to the lowest common denominator. Virtual file systems constitute today one of the most important technologies for the integration of computer systems using different operating and file systems.

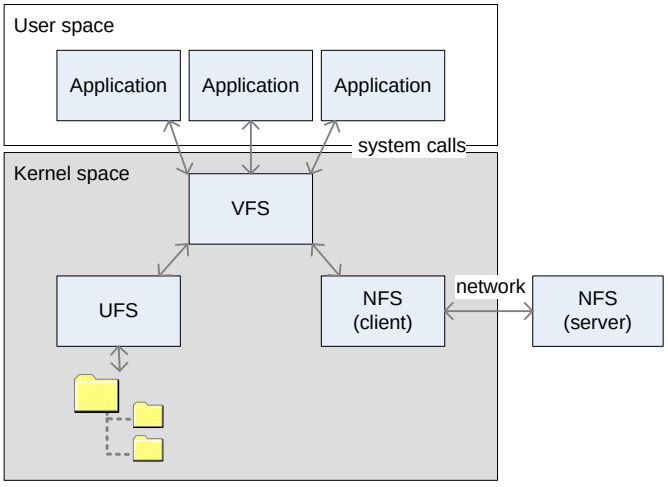


Figure A.2: Integrating two file systems (UFS and NFS) using a virtual file system layer.

[13] http://lists.infradead.org/mailman/listinfo/linux-mtd (Accessed August 2008).

NFS – Network File System (1985). NFS [SCR+03] is a secure and reliable network protocol layered above TCP/IP[14] that implements a network file system and is supported by most operating systems. The NFS protocol basically defines a set of message formats and types for accessing a remote file system. It is mountable in most operating systems through a virtual file system layer.

NFS is not a semantic file system but it was historically used by many projects for the implementation of semantic file system prototypes as it is much easier to write an NFS-server that augments an existing file system with a semantic access model than to implement a whole new file system. Naturally, this approach raises some major performance issues.

There are several other protocols for realizing a distributed file system, such as the Andrew File System (AFS), the Server Message Block (SMB) protocol and its Microsoft extension CIFS or the WebDAV protocol that allows file access via HTTP.

Device drivers. Another approach used for the development of file system extensions is the implementation of custom device drivers that reside between a storage device and the actual file system implementation. However, the development of device drivers is a complex and error-prone task that requires a lot of knowledge and resources[15].

User-space driver development is much easier as it is less susceptible to the many exceptions that have to be considered when programming in kernel space (concurrency, blocking functions, locks, page faults, address invalidation, etc.). This led to the development of frameworks for user-level driver development, including, e.g., Microsoft's User Mode Driver Framework[16] or the Linux Userspace-I/O (UIO) module. On the downside, pure user-level drivers suffer from poor performance because they cannot access all performance-optimized functions and kernel services available to kernel-space modules. Recently, hybrid driver frameworks were proposed that run partly in kernel- and partly in user-space to overcome this issue, refer to [GRB+08] for a sound discussion on this topic.

The idea of programming user-level file systems is not new. Early versions of the Linux NFS implementation ran in user-space, in 1997, Microsoft Research developed a kernel-mode proxy driver that forwarded I/O requests into user-mode [Hun97]. Today, the two most popular frameworks for the development of file system extensions are Microsoft's *IFS Kit* and the open source project *FUSE* (see below) that enables the simple development of Linux file systems in user-space.

IFS Kit – Installable File System Kit (1997). The IFS Kit[17] is a commercial file system development kit and provides all required interfaces for developing file systems and file system filters for several Windows versions. It enables developers to write *file system filter drivers* that insert themselves in the driver stack and intercept requests to a file system or another filter driver. The filter drivers may modify or drop such requests or may react by executing their own routines. A filter driver already provided by Microsoft is the *Filter Manager*. This driver greatly simplifies the development of 3rd party drivers, so-called *minifilters*. Minifilters may define their own position in the I/O-stack and provide an easy-to-use interface for the development of file system extensions for Windows.

FUSE – File System in User Space (2005). FUSE[18] is a virtual file system that provides generic interfaces for the development of custom Linux file systems (prototypes) that are then running as user-space applications. FUSE consists of a kernel-space module that is mounted as a virtual file system (it hooks into the

[14] Most older NFS implementations use UDP datagram transport for low overhead but this is not supported by the latest version (v4) any more.

[15] Microsoft reported that about 89% of the Windows XP crashes were caused by (buggy) device drivers, Linux driver code has up to seven times the bug density when compared to other kernel modules. Reasons for such increased bug-frequencies include the unavailability of sophisticated kernel-space development tools as they are available for user-space development [GRB+08]

[16] User-Mode Driver Framework, http://www.microsoft.com/whdc/driver/wdf/UMDF.mspx

[17] http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx

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

Linux VFS code). This component provides a virtual device (/dev/fuse), that is in turn used by a user-space daemon. This daemon then sends requests to the custom user-space component developed with the FUSE development kit and the returned results are sent back to the kernel.

FUSE has a large and active user community and was successfully used for the development of numerous file systems, including some interesting prototypes that implement semantic access models like the Logic File System (LisFS) [PSR06]) or LiFS (see 9.3.1). A comprehensive list of FUSE-based file systems can be retrieved from the project homepage. Although FUSE was developed for Linux (it is part of the kernel since version 2.6.14), various ports and comparable projects for other operating systems (MacOSX, Windows[19], NetBSD, FreeBSD, OpenSolaris, etc.) exist.

A.2.1 Spreading of file systems

Unfortunately, we are not aware of a source for getting reliable numbers about the distribution of file systems among PC users. However, some impression can be derived from the regularly updated browser statistics collected by w3schools (http://www.w3schools.com/browsers/browsers_os.asp)[20]. From the published numbers, it is evident that Windows is currently by far the most popular operating system and as NTFS is the standard file system of recent Windows versions, it is safe to infer that NTFS is clearly the most wide-spread file system in the private user sector.

[19] http://dokan-dev.net/en/ (Accessed June 2011).
[20] Similar numbers can be found at http://www.netmarketshare.com/

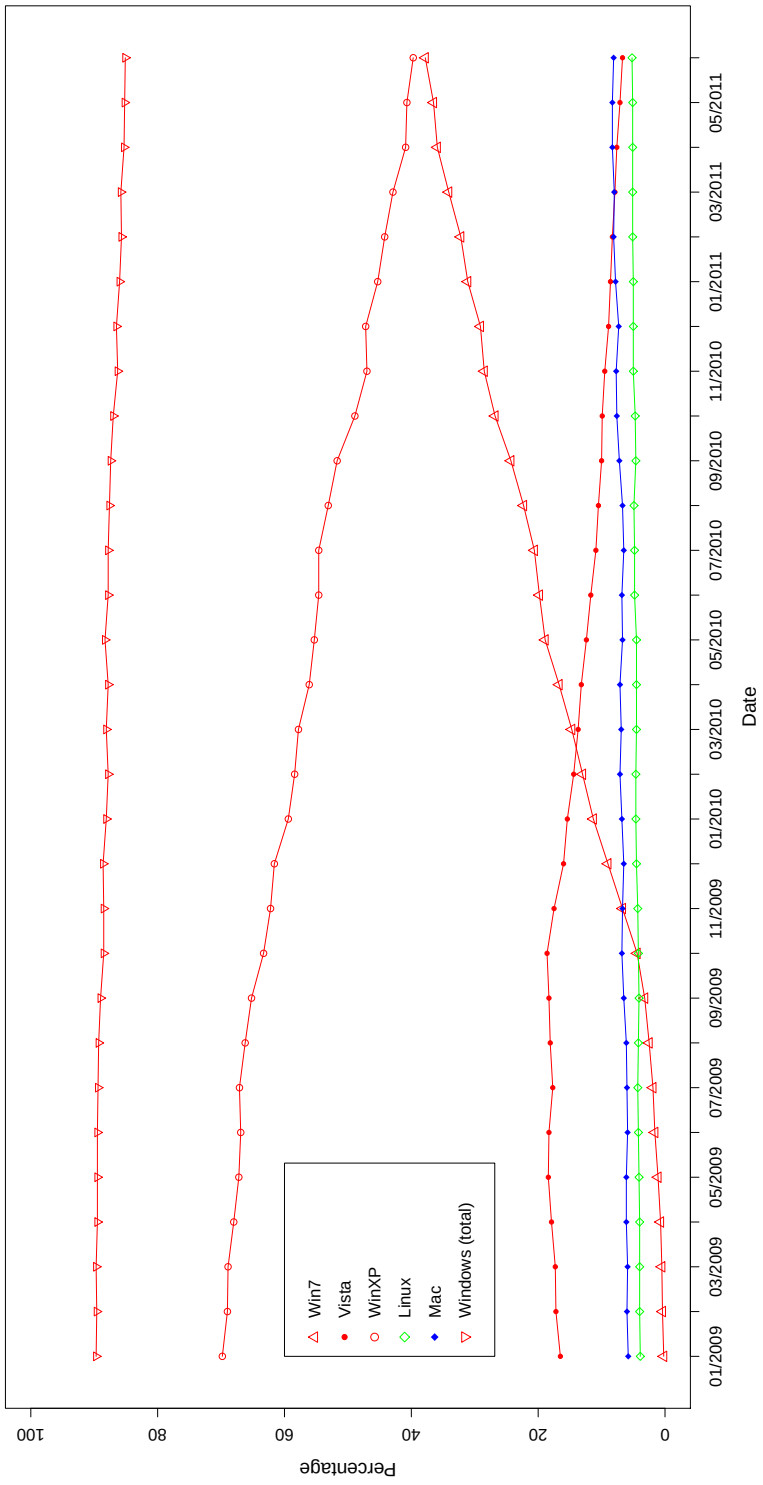


Figure A.3: Browser access statistics by operating system for the website http://www.w3schools.com/ as of June 2011. The Windows family counts for nearly 90%, platforms that count for less than 2% are not shown. Source: http://www.w3schools.com/browsers/browsers_os.asp

Appendix B

Current file system linking concepts

This chapter describes the semantics of linking concepts in current file systems, namely *hard links*, *symbolic links*, *shortcuts* and *aliases*.

Hard links. Hard Links are physical pointers to a file in the file system. In most file systems, the name of a file is itself a hard link and additional hard links (i.e., alternative "names" for the same resource) can be added. When hard links are removed, the file remains accessible until the last hard link is removed (and by that also the file data). Tools for creating hard links are, e.g., the NTFS *fsutil* tool or the Linux *ln* tool. Hard links to folders are often forbidden as they may introduce cycles in the folder hierarchy [CTT94, RC98, Esp00, App04].

Symbolic links. Symbolic links are part of the POSIX[1] operating-system standard and are supported by most Unix-like operating systems but also by Mac OS X or Windows 7. NTFS symbolic links (since Windows Vista) are file system objects that point to other files or folders, linking to remote paths (e.g., a SMB network path) is supported[2]. When the file they point to is removed, the symbolic link remains in the file system but now points to a non-existing file and is called "orphaned" (or sometimes "dangling").

Linux ext2 and UFS symbolic links are files that contain the (relative or absolute) pathname of the target file. An enhanced implementation called *fast symbolic links (fast symlinks)* improved the performance and disc-space usage of Unix/Linux symbolic links. While hard links cannot be used for cross-file system links, the implementation of symbolic links makes this possible [MJLF84, CTT94].

Windows shortcuts and Mac OS X aliases. Windows *shortcuts* are small binary files with the *.lnk* extension containing, among other metadata, a pointer to the referenced file. Shortcuts are interpreted at the Windows API level. This means, that applications that do not use this API for accessing them (although most applications do), will open the shortcut file instead of the referenced file. A similar concept is the Mac OS X Alias that is also realized by regular files containing some metadata about a referenced file. Both concepts maintain the references to their target files when these are moved. On the Windows platform, this is done by the *Distributed Link Tracking* (DLT) service[3], on Mac OS X by the *Alias Manager* [Mon92]. Both mechanisms, however, are limited by system boundaries (e.g., DLT works only on NTFS volumes) and are not applicable as a general mechanism for preserving stable references to all file system objects on a desktop system.

[1] POSIX is a family of IEEE standards that define the API, shell and utility interfaces for software compatible with variants of the Unix operating system [Wal95].

[2] Prior implementations that used NTFS junction points were limited to local paths and could link only to folders.

[3] `http://msdn.microsoft.com/en-us/library/aa363997%28v=vs.85%29.aspx` (Accessed June 2011).

Appendix C

Popular metadata standards

> "The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model."
> *Andrew S. Tanenbaum, Computer Networks, 2nd edn., p. 254*

C.1 Multimedia metadata standards

In this chapter we briefly introduce the metadata standards mentioned in this thesis starting with some popular standards for the description of multi-media data.

C.1.1 Exif – Exchangeable image file format (1998)

Exif is a simple, embeddable metadata format developed especially for the description of digital still camera (DSC) pictures [EA02]. Exif is supported by most camera manufacturers and is able to capture the following types of image-related metadata:

i) Metadata describing the image data structure: width, height, orientation, compression scheme, image format specific metadata, etc.

ii) Metadata describing the used digital camera: manufacturer, model; orientation, shutter speed, focal length, etc.

iii) Metadata describing the recording event: GPS related attributes (e.g., longitude, latitude and measurement precision), temporal information (date, time), etc.

iv) Image thumbnails: for on-camera preview purposes

v) Custom descriptions and metadata: image title, artist, notes, copyright information, etc.

Furthermore, Exif can also be used for the description of (RIFF WAVE form) audio files. The Exif standard further describes how to embed Exif data into the header of TIFF (tagged image file format) and JPEG images.

C.1.2 IIM, IPTC – Information Interchange Model

The International Press and Telecommunications Council (IPTC) developed the Information Interchange Model (IIM) in the 90ties as a "multimedia exchange format". In 1995, Adobe partly implemented IIM and integrated it with their popular Photoshop tool. Metadata was now being directly embedded in the file

headers of JPEG, TIFF and PSD files. The successor project was called *IPTC core* and is part of the *IPTC Photo Metadata 2008* standard that is by now fully integrated with Adobe's XMP format[1].

C.1.3 XMP – Extensible Metadata Platform (2001)

In 2001, Adobe presented the first version of their eXtensible Metadata Platform (XMP), a standard for storing and processing file-related metadata that was overworked since then several times [Ado05]. Today, XMP makes use of a subset of the Resource Description Framework (RDF) for the representation of metadata properties that describe a document or parts (e.g., an embedded image) of a document.

Although not required by XMP, it is recommended to *embed* XMP metadata directly in the file they describe (i.e., as internal metadata). Serialized XMP metadata are embeddable in many popular file formats including PDF, HTML, JPEG or TIFF and may co-exist with other embedded metadata: XMP supports multiple parallel namespaces (an important feature when multiple applications write data to such a file), simple data structures and a possibility to store properties that describe other properties (e.g., if a document has two authors that are represented as XMP properties, a property "role=main author" could be attached to the property representing the main author of the document). XMP defines several *extensible* metadata schemas for the description of files including (i) a *Dublin Core schema*, (ii) a *Dynamic Media* schema (e.g., for video and audio data), (iii) an *Exif* schema that implements the Exif 2.2 specification and (iv) a *Media Management* schema that is useful for digital asset management systems.

XMP has been adopted by other major vendors as well. Since Windows Vista, users may, for example, manipulate XMP metadata (including tags) in image files directly via the Windows Explorer application and these metadata are indexed by the Windows desktop search. An open-source software development kit for XMP, developed by Adobe, is available at http://www.adobe.com/devnet/xmp/.

C.1.4 ID3v2 (1998)

ID3v2 is an informal, open standard for storing metadata attributes in the headers of digital mpeg3 audio files [Nil99]. ID3v2 is wide-spread and supported by most MP3 processing tools, including Apple's iTunes, Microsoft's Windows Media Player and Nullsoft's WinAmp. It defines fixed header and footer metadata fields as well as some variable-length *frames* that may contain descriptive metadata. ID3v2 supports many types of audio-related metadata, including:

i) Textual information: album title, composer name, genres, etc.

ii) User defined textual information and links: additional, custom metadata, including URI references to Web resources containing, e.g., legal or publisher information.

iii) Synchronized and unsynchronized lyrics.

iv) Attached pictures and encapsulated (binary) objects.

v) Links to other ID3v2 tags stored in other files: here it is recommended to use this feature only in settings where the risk of file separation is low (e.g., on CD-ROMs)

The design of ID3 foresees that applications access certain metadata *frames* without having to know the structure of other frames. This mainly contributes to the flexibility and extensibility of the ID3v2 standard. Many software tools/APIs support the creation and/or extraction of ID3v2 metadata from audio files. For a list of stable tools and libraries, please refer to http://www.id3.org/Implementations. Although ID3 was developed for use with the MP3 format, other audio file formats like Ogg Vorbis (using Xiph Comments) or AAC (Advanced Audio Coding is a part of the MPEG2 standard and supports tagging as well) support comparable mechanisms.

[1] See http://www.iptc.org/IPTC4XMP/

C.2 Semantic vocabularies

In the following, we briefly describe some popular Semantic Web vocabularies mentioned in this thesis. Note that this list represents just a subjective choice of the author and is by all means incomplete. An online repository that manages references to existing LOD data sets and their used vocabularies can be found at http://ckan.net/group/lodcloud. Some statistics of vocabulary usage are available at http://www4.wiwiss.fu-berlin.de/lodcloud/state/#terms. A recently published data set that describes available Semantic Web vocabularies is the Linked Open Vocabularies (LOV) data set (http://labs.mondeca.com/dataset/lov/index.html).

C.2.1 DC Terms – Dublin Core

The *Dublin Core Metadata Initiative (DCMI) Metadata Terms*[2] are a Semantic Web compatible version of all terms maintained by DCMI. This includes the fifteen core terms of the well-known, standardized *Dublin Core Metadata Element Set* that can be used for a basic description of physical and digital objects. DC / DC-terms is currently arguably the most wide-spread semantic vocabulary in use.

C.2.2 FOAF – Friend of a Friend

FOAF[3] is a simple, RDF/OWL based vocabulary for describing a *social network* of persons, their activities and, most importantly, their relations to other persons. One can use FOAF, for example, to describe a person (name and nickname, etc.), its email and online accounts and what other persons this person knows.

As FOAF is a quite popular vocabulary, many tools for creation, validation and/or manipulation of FOAF data were developed. FOAF data is also already recognized by search engines like Yahoo's SearchMonkey or Google. A comparable XML-based technology is XFN[4] that is often used in blogs.

C.2.3 vCARD – Versitcard

vCard is a simple, standardized format for describing/exchanging electronic business cards. vCard entries may contain person-related metadata such as name, address, phone number, email addresses, URLs but also logos, photographs or even audio files that contain the proper pronunciation of the person's name. vCard supports the expression of these metadata in multiple languages. Various extensions to the vCard standard have been proposed by several vendors in order to support their specific needs (e.g., an Apple address book UUID or alternative name-spellings for Japanese names).

vCard entries can be serialized to a simple text-based format, an XML vCard format as well as an RDF-based encoding[5] have been developed. vCards can also be embedded into Web pages using the hCard microformat (see Section 2). The vCard version 2.1[6] is widely spread and supported by most email clients, the latest version 3.0 is published in RFC 2425 (A MIME Content-Type for Directory Information) and RFC 2426 (vCard MIME Directory Profile).

[2] http://dublincore.org/documents/dcmi-terms/
[3] http://www.foaf-project.org/
[4] XHTML Friends Network, http://gmpg.org/xfn/
[5] http://www.w3.org/TR/vcard-rdf/ (Accessed June 2011)
[6] http://www.imc.org/pdi/vcard-21.rtf

C.2.4 Nepomuk Ontologies

In the course of the Nepomuk project (see Section 9.3.3), several vocabularies for the semantic description of desktop data were developed[7]. These vocabularies can be used for the annotation of data on a semantic desktop including, e.g., files, contacts, emails, and multimedia data.

Some of these ontologies are RDF mappings to existing metadata standards like Exif and ID3. Others are rather high-level Semantic Desktop vocabularies that may have to be specialized to a particular domain for practical reasons (cf., [MPC+10]). Again others, like the Nepomuk File Ontology (NFO), were developed to fulfill the specific requirements of the Nepomuk project. The NFO can be used to describe files and other desktop resources and is related work to our presented Y2 core vocabulary (Section 9.2.1). It contains most properties for the description of low-level file metadata (e.g., file name, size, etc.) required by our method. However, some are missing and some differ slightly in their semantics when compared to our vocabulary.

C.2.5 SKOS – Simple Knowledge Organization System

The Simple Knowledge Organization System (SKOS) is a W3C recommended data model for expressing linked knowledge organization systems (KOS) [MeBe09]. Typical KOS (thesauri, taxonomies or classification schemes) can be expressed by the simple SKOS vocabulary (basically an *OWL Full* ontology) in a machine-friendly way. SKOS organizes knowledge in *concept schemes* comprising sets of *concepts*. Such concepts are represented by RDF resources that can be related with *multilingual labels and notes* as well as with so-called *notations* that comprise secondary identifiers within a KOS (e.g., ISBN numbers). SKOS further predefines properties for the semantic linking of concepts that allow, for example, the expression of hierarchical relationships. For further organization of a concept scheme, its contained concepts can be grouped into labeled and ordered *collections*. Finally, SKOS provides properties for mapping concepts in different concept schemes to each other in order to integrate them.

SKOS was designed to be easily extensible and customizable. It comprises an increasingly popular vocabulary for the description of KOS, probably due to its simplicity, and more and more tools for editing SKOS models become available (such as plug-ins for the popular Protégé editor or collaborative online tools like Poolparty[8]).

C.2.6 Lingvoj/Lexvo

Lingvoj (meaning "languages" in Esperanto) was a project "dedicated to the publication and use of multilingual RDF descriptions of human languages, to be used as Linked Data." [9]. It consisted of RDF descriptions about 522 languages (including all ISO 639-1 and most ISO 639-2 languages). Lingvoj described how languages are called in other languages, e.g., `http://www.lingvoj.org/lang/de` is a resource representing the German language.

The lingvoj language data was integrated with the lexvo[10] dataset in 2010, lingvoj URIs redirect to this data set by now. Lexvo serves language representations that are linked with a large variety of other Linked Data sets and vocabularies, e.g., WordNet, DBpedia and geographic regions.

The lingvoj project further defines an OWL ontology that defines several classes and properties useful for linking FOAF resources to languages (e.g., for stating that a person speaks a certain language and has interest in another one).

[7] `http://nepomuk.semanticdesktop.org/ontologies/`
[8] `http://poolparty.punkt.at/`
[9] See `http://www.lingvoj.org/`
[10] `http://www.lexvo.org/`

C.2.7 SIOC – Semantically-Interlinked Online Communities

SIOC[11], aims at interlinking online communities, such as wikis, weblogs or message boards by using Semantic Web technologies. It is partly based on FOAF[12] for describing user-related issues and on SKOS for knowledge organization. SIOC can be used for a high-level description of online-community content by using an RDF/OWL ontology that contains the central concepts SIOC is concerned with, such as user accounts, forums or postings. SIOC's original (a bit outdated) description is published in [BHBD05].

There exist several SIOC plug-ins (exporters) for popular blogging engines, content management systems and discussion forums, such as WordPress, Drupal, and phpBB[13]. APIs in various programming languages are available. SIOC is developed by an active developer community, it was published as a W3C Member Submission that was submitted by 16 organizations[14].

C.2.8 Semantic multimedia ontologies

COMM. Multimedia ontologies address the issue of finding and reusing multimedia content on the Web by supporting fine-grained (fragment based) description of multimedia objects like images, audio files or videos [HTRB09]. The COMM multimedia ontology, for example, is an MPEG-7 based OWL DL ontology that covers most parts of the MPEG-7 standard [ATS+07]. It is Semantic Web compatible and can easily be combined with other existing Web ontologies. COMM is also extensible and a Java API is available at http://comm.semanticweb.org/.

Music Ontology. The Music Ontology[15] provides a formal framework for describing various music-related data, including editorial, cultural and acoustic information [RASG07]. Audio signals (encoded in audio files) can be linked to their timeline and an event-ontology can be used for the description of particular timeline regions [HTRB09]. The Music Ontology has been used in various projects like Zitgist[16], DBTune[17] or EASAIER[18]. It is interlinked with other ontologies like FOAF and can be extended using SKOS [RSS09].

Ontology for Media Resources. The emerging "Ontology for Media Resources" [LeBeBe+11] is a W3C effort to create a semantic vocabulary for the description of media objects. This vocabulary defines basic properties for the description of such media and is published together with a number of defined mappings to existing media metadata standards, including Dublin Core, Exif, IPTC, ID3, XMP and MPEG-7.

[11] http://rdfs.org/sioc/spec/
[12] For instance, the *sioc:User* is an OWL-subclass of *foaf:OnlineAccount*.
[13] See http://sioc-project.org/exporters
[14] See http://www.w3.org/Submission/2007/02/
[15] See http://musicontology.com/
[16] See http://zitgist.com/
[17] See http://dbtune.org/
[18] See http://www.elec.qmul.ac.uk/easaier/

Appendix D

Y2 RDF examples

Listing D.1: N3-serialized Y2 RDF representation of the item shown in Figure 9.1a.

```
@prefix :        <http://localhost:8081/y2/r/6> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix void:    <http://rdfs.org/ns/void#> .
@prefix y2:      <http://purl.org/y2/vocab/20110503/y2core#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .

<http://localhost:8081/y2/r/25>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/3>
      y2:child <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/26>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/27>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/60>
      rdfs:seeAlso <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/6>
      a       <http://localhost:8081/y2/r/38> , y2:Directory ;
      rdfs:comment """␣Band

␣Locomotiv␣concert␣at␣szene␣wien␣2011


␣Flyer:
␣http://www.planet.tt/PMMP/detailview_szene.php?event_id=4571

␣Photos:
␣http://www.flickr.com/photos/chri5tian/sets/72157626470275450/
␣2011-04␣koncert␣szene␣wien

"""^^<http://www.w3.org/2001/XMLSchema#string> ;
      rdfs:seeAlso <http://www.planet.tt/PMMP/detailview_szene.php?event_id=4571> , <http://
           localhost:8081/y2/r/60> , <http://www.flickr.com/photos/chri5tian/sets/72157626470275450/> ;
      <http://purl.org/y2/vocab/20110503/y2annotation#textStyle>
              "1,4,260,y2://38␣http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                   ;56,8,1;134,7,1;211,26,260,y2://60␣http://www.w3.org/2000/01/rdf-schema#seeAlso;"^^<
                   http://www.w3.org/2001/XMLSchema#string> ;
      y2:child <http://localhost:8081/y2/r/25> , <http://localhost:8081/y2/r/26> , <http://
           localhost:8081/y2/r/27> , <http://localhost:8081/y2/r/18> , <http://localhost:8081/y2/r/20>
           , <http://localhost:8081/y2/r/22> , <http://localhost:8081/y2/r/23> , <http://localhost:8081
```

```
          /y2/r/19> , <http://localhost:8081/y2/r/30> , <http://localhost:8081/y2/r/21> , <http://
          localhost:8081/y2/r/28> , <http://localhost:8081/y2/r/24> , <http://localhost:8081/y2/r/29>
          ;
      y2:extension ""^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:local-name "szene"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:modified 1302765003193 ;
      y2:parent <http://localhost:8081/y2/r/3> ;
      y2:path "/szene"^^<http://www.w3.org/2001/XMLSchema#string> .

<http://localhost:8081/y2/r/18>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/20>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/22>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/30>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/19>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/23>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/21>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/28>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/24>
      y2:parent <http://localhost:8081/y2/r/6> .

<http://localhost:8081/y2/r/29>
      y2:parent <http://localhost:8081/y2/r/6> .
```

Listing D.2: N3-serialized Y2 RDF representation of the SKOS concept shown in Figure 9.1b.

```
@prefix :        <http://localhost:8081/y2/r/33> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix void:    <http://rdfs.org/ns/void#> .
@prefix y2:      <http://purl.org/y2/vocab/20110503/y2core#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .

<http://localhost:8081/y2/r/2>
      skos:narrower
              <http://localhost:8081/y2/r/33> .

<http://localhost:8081/y2/r/36>
      skos:narrower
              <http://localhost:8081/y2/r/33> .

<http://localhost:8081/y2/r/35>
      skos:narrower
              <http://localhost:8081/y2/r/33> .

<http://localhost:8081/y2/r/33>
      a       skos:Concept , <http://localhost:8081/y2/r/33> ;
      rdfs:comment """Use␣this␣SKOS␣concept␣for␣tagging␣items␣related␣to␣molecular␣biology
␣Molbio␣"""^^<http://www.w3.org/2001/XMLSchema#string> ;
      <http://purl.org/y2/vocab/20110503/y2annotation#textStyle>
              "51,17,260,http://dbpedia.org/resource/Molecular_biology␣http://www.w3.org/2004/02/skos/
                   core#closeMatch;71,6,260,y2://33␣http://www.w3.org/1999/02/22-rdf-syntax-ns#type;"^^
                   <http://www.w3.org/2001/XMLSchema#string> ;
      skos:broader
              <http://localhost:8081/y2/r/2> , <http://localhost:8081/y2/r/36> , <http://localhost:8081
                   /y2/r/35> ;
      skos:closeMatch <http://dbpedia.org/resource/Molecular_biology> ;
      skos:prefLabel "Molbio"^^<http://www.w3.org/2001/XMLSchema#string> .

<http://localhost:8081/y2/r/32>
      a       <http://localhost:8081/y2/r/33> .
```

Listing D.3: N3-serialized Y2 RDF representation of the MP3 file shown in Figure 9.6.

```
@prefix :        <http://localhost:8081/y2/r/13> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix void:    <http://rdfs.org/ns/void#> .
@prefix y2:      <http://purl.org/y2/vocab/20110503/y2core#> .
@prefix skos:    <http://www.w3.org/2004/02/skos/core#> .

<http://localhost:8081/y2/r/4>
      y2:child <http://localhost:8081/y2/r/13> .

<http://localhost:8081/y2/r/13>
      a       y2:File ;
      y2:checksum 58535 ;
      y2:extension "mp3"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:local-name "Walking.mp3"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:mimetype "application/octet-stream"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:modified 1308730282404 ;
      y2:parent <http://localhost:8081/y2/r/4> ;
      y2:path "/loco/Walking.mp3"^^<http://www.w3.org/2001/XMLSchema#string> ;
      y2:size 6733346 ;
      <http://www.w3.org/ns/ma-ont#hasCreator>
              <http://localhost:8081/y2/r/artist_Locomotiv> ;
      <http://www.w3.org/ns/ma-ont#title>
              "Walking"^^<http://www.w3.org/2001/XMLSchema#string> .
```

Bibliography

[AADAD09] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. *Trans. Storage*, 5(4):16:1–16:30, December 2009.

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3):9, 2007.

[ABG+06] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. Lifs: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2006.

[Abi96] Serge Abiteboul. Querying semi-structured data. Technical Report 1996-19, Stanford InfoLab, 1996.

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: a nucleus for a web of open data. In *ISWC'07*, pages 722–735, 2007.

[ACHZ09] Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. Describing linked datasets - on the design and usage of void, the 'vocabulary of interlinked datasets'. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, 2009.

[ADL+09] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumüller. Triplify: light-weight linked data publication from relational databases. In *WWW '09*, pages 621–630, 2009.

[Ado05] Adobe. XMP: Adding intelligence to media. Technical report, Adobe Systems Incorporated, September 2005.

[AeBeMePe08] Ben Adida (ed.), Mark Birbeck (ed.), Shane McCarron (ed.), and Steven Pemberton (ed.). Rdfa in xhtml: Syntax and processing, w3c recommendation 14 october 2008. Technical report, W3C, October 2008. http://www.w3.org/TR/rdfa-syntax/.

[AH06] Sören Auer and Heinrich Herre. A versioning and evolution framework for RDF knowledge bases. In *Ershov Memorial Conference*, pages 55–69, 2006.

[AKXH07] Liang Alei, Liu Kejia, Li Xiaoyong, and Guan Haibing. Fatty: A reliable fat file system. In *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, pages 390–395, 2007.

[AMB+05] Alexander Ames, Carlos Maltzahn, Nikhil Bobb, Ethan L. Miller, Scott A. Brandt, Alisa Neeman, Adam Hiatt, and Deepa Tuteja. Richer file system metadata using links and attributes. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 49–60, 2005.

[App04] Apple. Technical note tn1150 - hfs plus volume format. Technical report, Apple Inc., March 2004. http://developer.apple.com/technotes/tn/tn1150.html (Accessed August 2008).

[Arm01] William Y. Arms. Uniform resource names: handles, purls, and digital object identifiers. *Commun. ACM*, 44(5):68, 2001.

[Ash00] Helen Ashman. Electronic document addressing: dealing with change. *ACM Comput. Surv.*, 32(3):201–212, 2000.

[ATS+07] Richard Arndt, Raphaël Troncy, Steffen Staab, Lynda Hardman, and Miroslav Vacura. COMM: Designing a well-founded multimedia ontology for the web. In *ISWC '07*, pages 30–43, 2007.

[BB04] Hal Berghel and Natasa Brajkovska. Wading into alternate data streams. *Commun. ACM*, 47(4):21–27, 2004.

[BBK92] J. Michael Bennett, Michael A. Bauer, and David Kinchlea. Characteristics of files in nfs environments. *SIGSMALL/PC Notes*, 18(3-4):18–25, 1992.

[BC07] Chris Bizer and Richard Cyganiak. The trig syntax. http://www.wiwiss.fu-berlin.de/suhl/bizer/TriG/Spec/TriG-20070730/, July 2007.

[BCH08] Chris Bizer, Richard Cyganiak, and Tom Heath. How to publish linked data on the web. http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/, July 2008. Accessed april 2011.

[BDB+94] Mic Bowman, Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A File System for Information Management. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, March 1994.

[BDKR05] Tuugkan Batu, Sanjoy Dasgupta, Ravi Kumar, and Ronitt Rubinfeld. The complexity of approximating the entropy. *SIAM J. Comput.*, 35:132–150, July 2005.

[BeGe04] Dan Brickley (ed.) and Ramanathan V. Guha (ed.). Rdf vocabulary description language 1.0: Rdf schema, w3c recommendation 10 february 2004. Technical report, W3C, 2004. http://www.w3.org/TR/rdf-schema/.

[BF04] Margaret Beynon and Andrew Flegg. Hypertext request integrity and user experience, 2004. US Patent 0267726A1.

[BF07] Margaret Beynon and Andrew Flegg. Guaranteeing hypertext link integrity, 2007. US Patent 7290131 B2.

[BGSV06] Stephan Bloehdorn, Olaf Görlitz, Simon Schenk, and Max Völkel. Tagfs - tag semantics for hierarchical file systems. In *6th International Conference on Knowledge Management (I-KNOW'06), Special Track on Advanced Semantic Technologies*, 9 2006.

[BHBD05] John G. Breslin, Andreas Harth, Uldis Bojars, and Stefan Decker. Towards semantically-interlinked online communities. In *European Semantic Web Conference (ESWC)*, volume 3532 of *LNCS*, pages 500–514, 2005.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst*, 5(3):1–22, 2009.

[BHP04] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: authority-based keyword search in databases. In *VLDB '04: Proceedings of the 30th international conference on very large data bases*, pages 564–575, 2004.

[BLFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. RFC 3986, Internet Engineering Task Force, January 2005.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[BLHL+08] Tim Berners-Lee, James Hollenbach, Kanghao Lu, Joe Presbrey, and mc Schraefel. Tabulator redux: Browsing and writing linked data. In *WWW 2008 Workshop: Linked Data on the Web (LDOW2008)*, 2008.

[BLK08] Tim Berners-Lee and Lalana Kagal. The fractal nature of the semantic web. *AI Magazine*, 29(3):29–34, 2008.

[BLK+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semantics*, 7(3):154–165, 2009.

[BN95] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: file organization from the desktop. *SIGCHI Bull.*, 27(3):39–43, 1995.

[BP05] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the web: paradigms and applications of the language xchange. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1645–1649, 2005.

[Bun97] Peter Buneman. Semistructured data. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121, 1997.

[Bus45] Vannevar Bush. As we may think. *Atlantic Monthly*, 176(1):641–649, March 1945.

[CAH02] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study for xml change detection. Technical report, INRIA, 2002.

[CAM02] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in xml documents. In *ICDE '02*, pages 41–52, 2002.

[CBHS05] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 613–622, 2005.

[CFLM08] Silvana Castano, Alfio Ferrara, Davide Lorusso, and Stefano Montanelli. On the ontology instance matching problem. In *Dexa '08 Workshops*, pages 180–184, 2008.

[CGM97] Sudarshan Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *ACM International Conference on Management of Data (SIGMOD 1997)*, 1997.

[CRGMW96] Sudarshan Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25:493–504, June 1996.

[CTT94] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[Dav98] Hugh C. Davis. Referential integrity of links in open hypermedia systems. In *HYPERTEXT '98*, pages 207–216, 1998.

[Day08] Shobhit Dayal. Characterizing hec storage systems at rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University Parallel Data Lab, 2008.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99*, pages 59–70, 1999.

[DCC+03] Susan Dumais, Edward Cutrell, JJ Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *ACM SIGIR*, SIGIR '03, pages 72–79, 2003.

[DDD+04] Zubin Dalal, Suvendu Dash, Pratik Dave, Luis Francisco-Revilla, Richard Furuta, Unmil Karadkar, and Frank Shipman. Managing distributed collections: evaluating web page changes, movement, and replacement. In *JCDL '04: 4th ACM/IEEE-CS joint conference on Digital libraries*, pages 160–168, 2004.

[DEL+00] Paul Dourish, W. Keith Edwards, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry, and James Thornton. Extending document management systems with user-specific active properties. *ACM Trans. Inf. Syst.*, 18(2):140–170, April 2000.

[DELS99] Paul Dourish, W. Keith Edwards, Anthony LaMarca, and Michael Salisbury. Presto: an experimental architecture for fluid interactive document spaces. *ACM Trans. Comput.-Hum. Interact.*, 6(2):133–161, 1999.

[DFP+05] Li Ding, Tim Finin, Yun Peng, Paulo Pinheiro da Silva, and Deborah L. McGuinness. Tracking rdf graph provenance using rdf molecules. In *Proceedings of the 4th International Semantic Web Conference*, November 2005.

[DH05] Xin Dong and Alon Y. Halevy. A platform for personal information management and integration. In *CIDR*, pages 119–130, 2005.

[Dow01] Allen B. Downey. The structural cause of file size distributions. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '01, pages 328–329, 2001.

[Dun89] Roy Duncan. Design goals and implementation of the new high performance file system. Microsoft Systems Journal 4:5, Microsoft Corperation, September 1989.

[EA02] Japan Electronics and Information Technology Industries Association. Exchangeable image file format for digital still cameras: Exif version 2.2. Technical report, JEITA, April 2002.

[EE11] Patrick Durusau (Ed.) and Michael Brauer (Ed.). Open document format for office applications (opendocument) version 1.2, part 1: Opendocument schema. Technical Report Committee Specification Draft 07 / Public Review Draft 03, OASIS, January 2011.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):1–16, 2007.

[EK02] Kylie M. Evans and Geoffrey H. Kuenning. A study of irregularities in file-size distributions. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '02)*, 2002.

[Esp00] Dino Esposito. A programmer's perspective on ntfs 2000 part 1: Stream and hard link. http://msdn.microsoft.com/en-gb/ms123402.aspx, 2000. Accessed May 2008.

[FB96] Ned Freed and Nathaniel Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. RFC 2046, The Internet Society, November 1996.

[FDP+05] Tim Finin, Li Ding, Rong Pan, Anupam Joshi, Pranam Kolari, Akshay Java, and Yun Peng. Swoogle: Searching for knowledge on the semantic web. In *National Conference on Artificial Intelligence (AAAI)*, July 2005.

[FeGM+11] Roy Fielding (ed.), J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Y. Lafon (ed.), and J. Reschke (ed.). HTTP/1.1, part 5: Range Requests and Partial Responses draft-ietf-httpbis-p5-range-14, April 18th 2011.

[FGM+99] Roy Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[Fie00] Roy T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, 2000.

[FLMV08] Alfio Ferrara, Davide Lorusso, Stefano Montanelli, and Gaia Varese. Towards a benchmark for instance matching. In *Ontology Matching (OM 2008)*, volume 431 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[Fre97] Eric T. Freeman. *The lifestreams software architecture*. PhD thesis, Yale University, New Haven, CT, USA, 1997.

[GBL06] Jim Gemmell, Gordon Bell, and Roger Lueder. Mylifebits: a personal database for everything. *Commun. ACM*, 49(1):88–95, 2006.

[GC10] Fabien Gandon and Olivier Corby. Name that graph - or the need to provide a model and syntax extension to specify the provenance of rdf graphs. In *W3C Workshop - RDF Next Steps, June 26-27 2010, hosted by the National Center for Biomedical Ontology (NCBO), Stanford, Palo Alto, CA*, 2010.

[GD05] Lise Getoor and Christopher P. Diehl. Link mining: a survey. *SIGKDD Explor. Newsl.*, 7(2):3–12, 2005.

[Gdh07] Joe Gregorio and Bill de hOra. The atom publishing protocol. RFC 5023, The Internet Society, 2007.

[GeCeGe+10] Yolanda Gil (ed.), James Cheney (ed.), Paul Groth (ed.), Olaf Hartig (ed.), Simon Miles (ed.), Luc Moreau (ed.), and Paulo Pinheiro da Silva (ed.). Provenance xg final report. Technical report, W3C, December 2010. http://www.w3.org/2005/Incubator/prov/XGR-prov-20101214/.

[GePePe11] Paul Gearon (ed.), Alexandre Passant (ed.), and Axel Polleres (ed.). SPARQL 1.1 update, W3C working draft 12 may 2011. Technical report, W3C, 2011. http://www.w3.org/TR/2011/WD-sparql11-update-20110512/.

[GGWe05] Tony Gill, Anne J. Gilliland, Mary S. Woodley, and Murtha Baca (ed.). *Introduction to Metadata - Pathways to digital information*. Getty Education Institute for the Arts, U.S., version 2.1, online edition, Dec 2005.

[GHM+07] Tudor Groza, Siegfried Handschuh, Knud Moeller, Gunnar Grimnes, Leo Sauermann, Enrico Minack, Cedric Mesnage, Mehdi Jazayeri, Gerald Reif, and Rosa Gudjonsdottir. The nepomuk project – on the way to the social semantic desktop. In *Proceedings of the Third International Conference on Semantic Technologies (I-SEMANTICS 2007)*, 2007.

[Gia99] Dominic Giampaolo. *File System Design with the Be File System*. Morgan Kaufmann Publishers Inc., 1999.

[GJM09] Hugh Glaser, Afraz Jaffri, and Ian Millard. Managing co-reference on the semantic web. In *WWW 2009 Workshop: Linked Data on the Web (LDOW 2009)*, April 2009.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Association for Computing Machinery SIGOPS, 1991.

[GLMD07] Hugh Glaser, Tim Lewy, Ian Millard, and Ben Dowling. On coreference and the semantic web. Technical report, University of Southampton, 2007.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 265–278, 1999.

[GMG11] Antonio Garrote and María N. Moreno García. Restful writable apis for the web of linked data using relational storage solutions. In *WWW 2011 Workshop: Linked Data on the Web (LDOW2011)*, 2011.

[GRB+08] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *ASPLOS'08: Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, March 2008.

[Gre07] Mark Greaves. Semantic web 2.0. *IEEE Intelligent Systems*, 22(2):94–96, 2007.

[GS08] Karl Gyllstrom and Craig Soules. Seeing is retrieving: building information context from what the user sees. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 189–198, 2008.

[GSV07] Karl Anders Gyllstrom, Craig Soules, and Alistair Veitch. Confluence: enhancing contextual desktop search. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 717–718, 2007.

[Hau08] Michael Hausenblas. *Building Scalable and Smart Multimedia Applications on the Semantic Web*. Doctoral thesis, Technical University of Graz, 2008.

[HDPM08] Ben Hicks, Andy Dong, R. Palmer, and Hamish C. Mcalpine. Organizing and managing personal electronic files: A mechanical engineer's perspective. *ACM Trans. Inf. Syst.*, 26:1–40, October 2008.

[HDS05] Edward Hung, Yu Deng, and V. S. Subrahmanian. Rdf aggregate queries and views. In *ICDE*, pages 717–728, 2005.

[He10] Ian Hickson (ed.). Html microdata - working draft 24 june 2010. Technical report, W3C, 2010. http://www.w3.org/TR/microdata/.

[HeMe04] Patrick Hayes (ed.) and Brian McBride (ed.). Rdf semantics. Technical report, W3C Recommendation 10 February 2004, 2004. http://www.w3.org/TR/rdf-mt/.

[HGM09] Erik Hatcher, Otis Gospodnetic, and Michael McCandless. *Lucene in Action*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[HH08] Patrick J. Hayes and Harry Halpin. In defense of ambiguity. *Int. J. Semantic Web Inf. Syst.*, 4(2):1–18, 2008.

[HHD+07a] Andreas Harth, Aidan Hogan, Renaud Delbru, Jürgen Umbrich, Sean O'Riain, and Stefan Decker. Swse: Answers before links! In *Semantic Web Challenge*, volume 295 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[HHD07b] Aidan Hogan, Andreas Harth, and Stefan Decker. Performing object consolidation on the semantic web data graph. In *Proceedings of the 1st I^3: Identity, Identifiers, Identification Workshop*, 2007.

[HHR+09] Michael Hausenblas, Wolfgang Halb, Yves Raimond, Lee Feigenbaum, and Danny Ayers. Scovo: Using statistics on the web of data. In *ESWC 2009*, pages 708–722, 2009.

[HN06] Terry L. Harrison and Michael L. Nelson. Just-in-time recovery of missing web pages. In *HYPERTEXT '06*, pages 145–156, 2006.

[HP09] Bernhard Haslhofer and Niko Popitsch. DSNotify – detecting and fixing broken links in linked data sets. In *8th International Workshop on Web Semantics (WebS 09), co-located with DEXA 2009*, pages 89–93, 2009.

[HPBL09] James Hollenbach, Joe Presbrey, and Tim Berners-Lee. Using rdf metadata to enable access control on the social semantic web. In *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, 2009.

[HSB07] Martin Hepp, Katharina Siorpaes, and Daniel Bachlechner. Harvesting wiki consensus: Using Wikipedia entries as vocabulary for knowledge management. *IEEE Internet Computing*, 11(5):54–65, 2007.

[HTRB09] Michael Hausenblas, Raphaël Troncy, Yves Raimond, and Tobias Bürger. Interlinking multimedia: How to apply linked data principles to multimedia fragments. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, 2009.

[Hun97] Galen C. Hunt. Creating user-mode device drivers with a proxy. In *NT'97: Proceedings of the USENIX Windows NT Workshop*, pages 8–8, 1997.

[HZ10] Olaf Hartig and Jun Zhao. Publishing and consuming provenance metadata on the web of linked data. In *IPAW*, pages 78–90, 2010.

[ICL96] David Ingham, Steve Caughey, and Mark Little. Fixing the "broken-link" problem: the w3objects approach. *Comput. Netw. ISDN Syst.*, 28(7-11):1255–1268, 1996.

[INCG07] Panagiotis G. Ipeirotis, Alexandros Ntoulas, Junghoo Cho, and Luis Gravano. Modeling and managing changes in text databases. *ACM Trans. Database Syst.*, 32(3):14, 2007.

[Irl93] Gordon Irlam. Unix file size survey. http://www.gordoni.com/ufs93.html, 1993. Accessed april 2011.

[ISO98] ISO/ECMA. Volume and file structure of CD-ROM for information interchange. International standard; ISO 9660 ISO 9660: 1988 (E), International Organization for Standardization and European Computer Manufacturers Association, 1998.

[JEI98] JEITA. Design rule for camera file system version, v. 1.0. http://www.exif.org/dcf.PDF, December 1998. Standard of Japan Electronics and Information Technology Industries Association.

[JeWe04] Ian Jacobs (ed.) and Norman Walsh (ed.). Architecture of the world wide web, volume one. Technical report, W3C, December 2004. http://www.w3.org/TR/webarch/.

[Jon07] William Jones. Personal information management. *Annual Rev. Info. Sci & Technol.*, 41:453–504, December 2007.

[Kap95] Frank Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *Journal of Universal Computer Science*, 1(2):84–104, 1995.

[KBH+05] David R. Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A customizable general-purpose information management tool for end users of semistructured data. In *Proceedings of CIDR*, 2005.

[KeCeMe04] Graham Klyne (ed.), Jeremy J. Carroll (ed.), and Brian McBride (ed.). Resource description framework (RDF): Concepts and abstract syntax, W3C recommendation 10 february 2004. Technical report, W3C, 2004. http://www.w3.org/TR/rdf-concepts/.

[KJ06] David R. Karger and William Jones. Data unification in personal information management. *Commun. ACM*, 49:77–82, January 2006.

[KN10] Martin Klein and Michael L. Nelson. Evaluating methods to rediscover missing web pages from the web infrastructure. *CoRR*, abs/0907.2268, 2010.

[KVV+07] Markus Krötzsch, Denny Vrandecic, Max Völkel, Heiko Haller, and Rudi Studer. Semantic wikipedia. *Journal of Web Semantics*, 5:251–261, September 2007.

[LdS02] Carl Lagoze and Herbert Van de Sompel. The open archives initiative protocol for metadata harvesting — version 2.0. http://www.openarchives.org/OAI/openarchivesprotocol.html, june 14th 2002. Accessed april 2011.

[LeBeBe+11] WonSuk Lee (ed.), Werner Bailer (ed.), Tobias Bürger (ed.), Pierre-Antoine Champin (ed.), Véronique Malaisé (ed.), Thierry Michel (ed.), Felix Sasaki (ed.), Joakim Söderberg (ed.), Florian Stegmaier (ed.), and John Strassner (ed.). Ontology for media resources 1.0, W3C working draft 10 june 2011. Technical report, W3C, 2011. http://www.w3.org/2008/WebVideo/Annotations/drafts/ontology10/CR/mediaont-1.0.html.

[LPF+01] Steve Lawrence, David M. Pennock, Gary William Flake, Robert Krovetz, Frans M. Coetzee, Eric Glover, Finn Arup Nielsen, Andries Kruger, and C. Lee Giles. Persistence of web references in scientific research. *Computer*, 34(2):26–31, 2001.

[LSC02] Wen-Syan Li, Junho Shim, and K. Selcuk Candan. Webdb: A system for querying semistructured data on the web. *Journal of Visual Languages & Computing*, 13(1):3 – 33, 2002.

[Mar04] Ben Martin. Formal concept analysis and semantic file systems. In *ICFCA*, pages 88–95, 2004.

[Mar05] Ben Martin. Filesystem indexing with libferris. *Linux J.*, 2005(130):7, 2005.

[Mar06] Ben Martin. The world is a libferris filesystem. *Linux J.*, 2006(146):7, 2006.

[MC03] Gary Marsden and David E. Cairns. Improving the usability of the hierarchical file system. In *SAICSIT '03*, pages 122–129, Republic of South Africa, 2003.

[Mea03] Ryan L. Means. Alternate data streams: Out of the shadows and into the light. Technical report, SANS Institute, 2003.

[MeBe09] Alistair Miles (ed.) and Sean Bechhofer (ed.). SKOS simple knowledge organization system reference, w3c recommendation 18 august 2009. Technical report, W3C, 2009. http://www.w3.org/TR/skos-reference/.

[MFF+08] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. Mcgrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In *Provenance and Annotation of Data and Processes*, pages 323–326. Springer-Verlag, Berlin, Heidelberg, 2008.

[Mic08] Microsoft. Extended fat file system. http://msdn.microsoft.com/en-us/library/aa914353.aspx, August 2008. Accessed August 2008.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[MNI+09] Atsuyuki Morishima, Akiyoshi Nakamizo, Toshinari Iida, Shigeo Sugimoto, and Hiroyuki Kitagawa. Bringing your dead links back to life: a comprehensive approach and lessons learned. In *20th ACM conference on Hypertext and hypermedia*, pages 15–24, 2009.

[Mon92] Tim Monroe. *Inside Macintosh - Filesystem*. Addison-Wesley, 1992.

[MPC+10] Enrico Minack, Raluca Paiu, Stefania Costache, Gianluca Demartini, Julien Gaugaz, Ekaterini Ioannou, Paul-Alexandru Chirita, and Wolfgang Nejdl. Leveraging personal metadata for desktop search: The beagle++ system. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(1):37 – 54, 2010.

[MT84] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Softw. Pract. Exper.*, 14:365–368, June 1984.

[MTCP04] Aimilia Magkanaraki, Val Tannen, Vassilis Christophides, and Dimitris Plexousakis. Viewing the semantic web through rvl lenses. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):359–375, October 2004.

[MTX02] Mallik Mahalingam, Chunqiang Tang, and Zhichen Xu. Towards a semantic, deep archival file system. Technical report, HP Laboratories Palo Alto, 2002.

[Neu92] Clifford Neuman. The prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, 1992.

[NHW+04] Mor Naaman, Susumu Harada, QianYing Wang, Hector Garcia-Molina, and Andreas Paepcke. Context data in geo-referenced digital photo collections. In *12th ACM international conference on Multimedia (MM 2004)*, October 2004.

[Nil99] Martin Nilsson. Id3 tag version 2.3.0. http://www.id3.org/id3v2.3.0, February 1999. Accessed april 2011.

[NS05] Mark Nottingham and Robert Sayre. The atom syndication format. RFC 4287, The Internet Society, 2005.

[NS07] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Linguisticae Investigationes*, 30(1):3–26, January 2007.

[Ols93] Michael A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–218, 1993.

[OST05] OSTA. Universal disk format specification. Technical Report Rev. 2.60, Optical Storage Technology Association, March 1 2005.

[OVBD06] Eyal Oren, Max Völkel, John G. Breslin, and Stefan Decker. Semantic wikis for personal knowledge management. In *DEXA, ser. Lecture Notes in Computer Science*, pages 509–518, 2006.

[PD99] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31:63–103, March 1999.

[Pe05] Emmanuel Pietriga (ed.). Fresnel selector language for rdf (fsl). Technical report, W3C, November 18th 2005. http://www.w3.org/2005/04/fresnel-info/fsl/.

[PeSe08] Eric Prud'hommeaux (ed.) and Andy Seaborne (ed.). SPARQL query language for RDF. W3C recommendation 15 january 2008. Technical report, W3C, 2008. http://www.w3.org/TR/rdf-sparql-query/.

[PFF+09] Vicky Papavassiliou, Giorgos Flouris, Irini Fundulaki, Dimitris Kotzinos, and Vassilis Christophides. On detecting high-level changes in rdf/s kbs. In *ISWC '09: Proceedings of the 8th International Semantic Web Conference*, pages 473–488, 2009.

[PH10] Niko Popitsch and Bernhard Haslhofer. Dsnotify: handling broken links in the web of data. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 761–770, 2010.

[PH11] Niko Popitsch and Bernhard Haslhofer. Dsnotify – a solution for event detection and link maintenance in dynamic datasets. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9:266–283, 2011.

[PHR10] Niko Popitsch, Bernhard Haslhofer, and Elaheh Momeni Roochi. An evaluation approach for dynamics-aware applications using linked data. In *9th International Workshop on Web Semantics (WebS 10), co-located with DEXA 2010*, 2010.

[PMP10] Niko Popitsch, Robert Mosser, and Wolfgang Philipp. Urobe: A prototype for wiki preservation. In *7th International Conference on Preservation of Digital Objects (iPRES2010)*, 2010.

[Pop11] Niko Popitsch. Keep your triples together: Modeling a resttful, layered linked data store. In *I-Semantics*, 2011.

[PPGK04] Seung-Taek Park, David M. Pennock, C. Lee Giles, and Robert Krovetz. Analysis of lexical signatures for improving information persistence on the world wide web. *ACM Trans. Inf. Syst.*, 22(4):540–572, 2004.

[PR03] Yoann Padioleau and Olivier Ridoux. A logic file system. In *USENIX Annual Technical Conference, General Track*, pages 99–112, 2003.

[PR05] Yoann Padioleau and Olivier Ridoux. A parts-of-file file system. In *USENIX Annual Technical Conference, General Track*, pages 359–362, 2005.

[PS10] Niko Popitsch and Bernhard Schandl. Ad-hoc file sharing using linked data technologies. In *International Workshop on Personal Semantic Data (PSD 2010)*, October 2010.

[PSA+06] Niko Popitsch, Bernhard Schandl, Arash Amiri, Stefan Leitich, and Wolfgang Jochum. Ylvi - multimedia-izing the semantic wiki. In *1st Workshop on Semantic Wikis - From Wiki to Semantics*, 2006.

[PSK08] Niko Popitsch, Bernhard Schandl, and Ross King. Semantic wikis: Conclusions from real-world projects with ylvi. In *Proceedings of the 16th European Conference on Information Systems (ECIS)*, 2008.

[PSR06] Yoann Padioleau, Benjamin Sigonneau, and Olivier Ridoux. Lisfs: a logical information system as a file system. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 803–806, 2006.

[PW00] Thomas A. Phelps and Robert Wilensky. Robust hyperlinks cost just five words each. Technical Report UCB/CSD-00-1091, EECS Department, University of California, Berkeley, 2000.

[QBHK03] Dennis Quan, Karun Bakshi, David Huynh, and David R. Karger. User interfaces for supporting multiple categorization. In *INTERACT*. IOS Press, 2003.

[RASG07] Yves Raimond, Samer Abdallah, Mark Sandler, and Frederick Giasson. The music ontology. In *Proceedings of the 8th International Conference on Music Information Retrieval (ISMIR)*, September 2007.

[RC98] Jeffrey Richter and Luis Felipe Cabrera. A file system for the 21st century: Previewing the windows nt 5.0 file system. *Microsoft Systems Journal*, November 1998.

[Rek99] Jun Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 45–54, 1999.

[RG10] Neil C. Rowe and Simson L. Garfinkel. Global analysis of drive file times. In *Proceedings of the 2010 Fifth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE '10, pages 97–108, 2010.

[RGSH07] Gerald Reif, Tudor Groza, Simon Scerri, and Siegfried Handschuh. Nepomuk deliverable d6.2.b - final nepomuk architecture. Technical report, NEPOMUK Consortium, 2007.

[Rod04] Jesus Rodriguez. Winfs data model. http://www.longhorncorner.com/UploadFile/jrodriguez/WinFSDataModel03072005041249AM/WinFSDataModel.aspx, Feb 2004. Accessed August 2008.

[RP08] Jörg Richter and Jurij Poelchau. Deepamehta - another computer is possible. *Emerging Technologies for Semantic Work Environments: Techniques, Methods, and Applications*, 2008.

[RR00] David S. H. Rosenthal and Vicky Reich. Permanent web publishing. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 129–140, 2000.

[RSS08] Yves Raimond, Christopher Sutton, and Mark Sandler. Automatic interlinking of music datasets on the semantic web. In *WWW 2008 Workshop: Linked Data on the Web (LDOW2008)*, April 2008.

[RSS09] Yves Raimond, Christopher Sutton, and Mark Sandler. Interlinking music-related data on the web. *IEEE MultiMedia*, 16(2):52–63, 2009.

[Sat81] Mahadev Satyanarayanan. A study of file sizes and functional lifetimes. *SIGOPS Oper. Syst. Rev.*, 15(5):96–108, 1981.

[SB04] Yang Song and Sourav S. Bhowmick. Biodiff: an effective fast change detection algorithm for genomic and proteomic data. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 146–147, 2004.

[Sch04] Thomas Schürmann. Bias analysis in entropy estimation. *Journal of Physics A: Mathematical and General*, 37(27):L295, 2004.

[Sch06] Sebastian Schaffert. Ikewiki: A semantic wiki for collaborative knowledge management. In *Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 388–396, 2006.

[Sch09] Bernhard Schandl. *An Infrastructure for the Development of Semantic Desktop Applications*. Doctoral dissertation, University of Vienna, 2009.

[SCR+03] Spencer Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (nfs) version 4 protocol. http://tools.ietf.org/html/rfc3530, April 2003. Accessed April 2011.

[SeWeMe04] Michael K. Smith (ed.), Chris Welty (ed.), and Deborah L. McGuinness (ed.). Owl web ontology language guide. Technical report, W3C Recommendation 10 February 2004, 2004.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, 1999.

[SG03] Craig A. N. Soules and Gregory R. Ganger. Why can't i find my files? new methods for automating attribute assignment. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 115–120, 2003.

[SG04] Craig A. N. Soules and Gregory R. Ganger. Toward automatic context-based attribute assignment for semantic file systems. Technical Report CMU-PDL-04-105, Carnegie Mellon University, June 2004.

[SH08] Katharina Siorpaes and Martin Hepp. Games with a purpose for the semantic web. *IEEE Intelligent Systems*, 23(3):50–60, 2008.

[SH09] Bernhard Schandl and Bernhard Haslhofer. The sile model — a semantic file system infrastructure for the desktop. In *6th European Semantic Web Conference (ESWC2009)*, 2009.

[SH10] Bernhard Schandl and Bernhard Haslhofer. Files are siles: Extending file systems with semantic annotations. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 6(3):1–21, 2010.

[Sir05] John Siracusa. Mac os x 10.4 tiger: Metadata revisited. *Ars Technica - Operating System Reviews*, 1:6–12, April 28th 2005. http://arstechnica.com/reviews/os/macosx-10-4.ars/6 (Accessed August 2008).

[SKL08] Marc Spaniol, Ralf Klamma, and Mathias Lux. Imagesemantics: User-generated metadata, content based retrieval & beyond. *Journal of Universal Computer Science*, 14(10):1792–1807, 2008.

[SM92] Stuart Sechrest and Michael McClennen. Blending hierarchical and attribute-based file naming. *International Conference on Distributed Computing Systems*, 1:572–580, June 1992.

[SP10] Bernhard Schandl and Niko Popitsch. Lifting file systems into the linked data cloud with TripFS. In *WWW 2010 Workshop: Linked Data on the Web (LDOW2010)*, volume 628 of *CEUR Workshop Proceedings*, 2010.

[SP11] Owen Sacco and Alexandre Passant. A privacy preference ontology (ppo) for linked data. In *WWW 2011 Workshop: Linked Data on the Web (LDOW2011)*, 2011.

[SSGN07] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using provenance to aid in personal file search. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2007.

[STH09] Ryan Shaw, Raphaël Troncy, and Lynda Hardman. Lode: Linking open descriptions of events. In *ASWC*, volume 5926 of *LNCS*, pages 153–167. Springer, 2009.

[Sti05] Patrick Stickler. Cbd - concise bounded description, w3c member submission. http://www.w3.org/Submission/CBD/, June, 3rd 2005. Accessed april 2011.

[Sun06] Sun. Zfs on-disk specification (draft). Technical report, Sun Microsystem Inc., 2006.

[SvESH07] Michael Sintek, Ludger van Elst, Simon Scerri, and Siegfried Handschuh. Nepomuk representational language specification. Technical report, Nepomuk Consortium, 2007.

[TAAK04] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. The perfect search engine is not enough: a study of orienteering behavior in directed search. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 415–422, 2004.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[THAF10] Sebastian Tramp, Norman Heino, Sören Auer, and Philipp Frischmuth. Making the semantic data web easily writeable with rdfauthor. In *ESWC (2)*, pages 436–440, 2010.

[THB06] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems: then and now. *SIGOPS Oper. Syst. Rev.*, 40:100–104, January 2006.

[TMPP05] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. Signing individual fragments of an rdf graph. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 1020–1021, 2005.

[TOD07] Giovanni Tummarello, Eyal Oren, and Renaud Delbru. Sindice.com: Weaving the open linked data. In *ISWC*, volume 4825 of *LNCS*, pages 547–560, November 2007.

[UHH+10] Jürgen Umbrich, Michael Hausenblas, Aidan Hogan, Axel Polleres, and Stefan Decker. Towards dataset dynamics: Change frequency of linked open data sources. In *WWW 2010 Workshop: Linked Data on the Web (LDOW2010)*, volume 628 of *CEUR Workshop Proceedings*, 2010.

[vAD04] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 319–326, 2004.

[VBGK09a] Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Discovering and maintaining links on the web of data. In *ISWC '09*, volume 5823 of *LNCS*, pages 650–665, 2009.

[VBGK09b] Julius Volz, Christian Bizer, Martin Geadke, and Georgi Kobilarov. Silk - a link discovery framework for the web of data. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, 2009.

[VdSNB+10] Herbert Van de Sompel, Michael Nelson, Lyudmila Balakireva, Harihar Shankar, and Scott Ainsworth. An HTTP-based versioning mechanism for linked data. In *WWW 2010 Workshop: Linked Data on the Web (LDOW2010)*, volume 628 of *CEUR Workshop Proceedings*, 2010.

[VF03] Luís Veiga and Paulo Ferreira. Repweb: replicated web with referential integrity. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1206–1211, 2003.

[VG09] Stephen Voida and Saul Greenberg. WikiFolders: Augmenting the Display of Folders to Better Convey the Meaning of Files. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2009)*, pages 1679–1682, 2009.

[VHC10] Rob Vesse, Wendy Hall, and Leslie Carr. Preserving linked data on the semantic web by the application of link integrity techniques from hypermedia. In *WWW 2010 Workshop: Linked Data on the Web (LDOW2010)*, volume 628 of *CEUR Workshop Proceedings*, 2010.

[Vog99] Werner Vogels. File system usage in windows nt 4.0. *SIGOPS Oper. Syst. Rev.*, 33:93–109, December 1999.

[VP96] Venu Vasudevan and Paul Pazandak. Semantic file systems. http://www.objs.com/survey/OFSExt.htm, 1996. Accessed July 2008.

[Wal95] Stephen R. Walli. The posix family of standards. *StandardView*, 3:11–17, March 1995.

[WGM95] Craig E. Wills, Dominic Giampaolo, and Michael Mackovitch. Experience with an interactive attribute-based user information environment. *Computers and Communications*, 1:359–365, Mar 1995.

[Win06] William E. Winkler. Overview of record linkage and current research directions. Technical report, U.S. Bureau of the Census, 2006.

[Woo01] David Woodhouse. Jffs : The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*, pages 177–182, 2001.

[XKTK03] Zhichen Xu, Magnus Karlsson, Chunqiang Tang, and Christos Karamanolis. Towards a semantic-aware file store. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 31–31, 2003.

[YCK09] Man-Ching Yuen, Ling-Jyh Chen, and Irwin King. A survey of human computation systems. *IEEE International Conference on Computational Science and Engineering*, 4:723–728, 2009.

Curriculum Vitae

Niko Popitsch

University of Vienna
Research Group Multimedia Information Systems
Liebiggasse 4/3-4, 1010 Wien
Phone: +43-1-4277-39626
E-Mail:: niko.popitsch@univie.ac.at
WWW: http://www.cs.univie.ac.at/niko.popitsch

Personal

Born:	February 2nd, 1976 in Graz, Austria
Citizenship:	Austria

Education

Since 2005	Studies of molecular biology at the University of Vienna
2007–2011	PhD in computer science, University of Vienna
1994–2000	Diploma in informatics, Vienna University of Technology

Research and professional experience

2007–2011	Research assistant at the University of Vienna, research group *Multimedia Information Systems*
2003–2007	Senior researcher at ARC Seibersdorf research GmbH, Studio Digital Memory Engineering
2001–2003	Uma information systems
2000–2001	Blue-C Internet GmbH
Since 1999	Project-related collaboration with the artist collective maupi, Web development