Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

THE ANH PHAM

Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht

ZIB-Report 05-41 (September 2005)

Diplomarbeit

Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht

vorgelegt von

The Anh Pham

an der

Technischen Universität Berlin

Betreuer:

Prof. Dr. Hans-Ulrich Heiß

Prof. Dr. Alexander Reinefeld

August 2005

Selbständigkeitserklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt. Die Arbeit wurde nur mit den angegebenen Quellen verfasst und noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Berlin, den 04.08.2005 _____

Danksagung

Mein Dank gilt Herrn Prof. Dr. Alexander Reinefeld, Leiter des Bereichs Computer Science am Konrad-Zuse-Zentrum für Informationstechnik Berlin und Professor für Parallele und Verteilte Systeme am Institut für Informatik der Humboldt-Universität zu Berlin, und Herrn Prof. Dr. Hans-Ulrich Heiß vom Institut für Kommunikations- und Betriebssysteme an der TU Berlin. Ganz besonders bedanken möchte ich mich bei Herrn Florian Schintke, Konrad-Zuse-Zentrum für Informationstechnik Berlin, für seine kompetente Betreuung und konstruktiven Hinweise zu dieser Arbeit.

Außerdem danke ich allen ZIBDMS-Entwicklern für die Bereitstellung benötigter ZIBDMS-Komponenten sowie für die spannenden „Pair Programming"-Stunden, die wir gemeinsam an ZIBDMS gearbeitet und viel Spaß dabei hatten.

Ein herzliches Dankeschön an meine Freunde und Kollegen für die Zeit, die sie mit dem Korrekturlesen dieser Arbeit verbracht haben.

Schließlich möchte ich mich ganz herzlich bei meinen Eltern für ihre unbegrenzte Unterstützung bedanken, durch die ich bei ihnen jederzeit einen starken Rückhalt gefunden habe.

Kurzfassung

Die hierarchische Organisation von Dateien beherrscht Computersysteme seit vielen Jahren und wird sich vermutlich auch in Zukunft weiterhin durchsetzen. Dennoch stößt diese Verwaltungsmethode in einem klassischen hierarchischen Dateisystem bei dem jährlichen rasanten Zuwachs an neuen Daten an ihre Grenzen, auch wenn Nutzer eine ausgefeilte Verzeichnisstruktur mit einer disziplinierten Namensgebung konsequent auf- und ausbauen. Um den Überblick über die Dateien zu behalten, werden in dieser Arbeit neue Verwaltungsmethoden vorgestellt und in die hierarchische Dateisystemsicht eingebettet.

Auf der Basis des hierarchischen Dateisystems bietet ein metadatenbasiertes Dateisystem neben dem hierarchischen Zugriff noch einen flexiblen, assoziativen Zugriff auf Dateien über virtuelle Objekte, indem Suchmethoden in Form einer Anfrage auf in einer Datenbank gespeicherte Index- und Metadaten angewendet werden. Das Ergebnis dieser Abfrage wird mit Hilfe von virtuellen Verzeichnissen und virtuellen Dateien in einer übersichtlichen Form dargestellt, so dass eine Nutzung der hierarchischen Sicht weiterhin intuitiv fortgesetzt wird.

Das Datenmanagementsystem ZIBDMS, welches einen verteilten Metadatenkatalog und einen Dateireplikationskatalog beinhaltet, ermöglicht es, die oben genannten Verwaltungsmethoden für die NFS-Schnittstelle, CORBA-Middleware und Web Services transparent zu implementieren. Zudem stellt ZIBDMS weitere neue Dateiverwaltungsmethoden zur Verfügung, die sich in die hierarchische Dateisystemsicht integrieren lassen. Metadaten in Form von Attribut-Wert-Paaren lassen sich als virtuelle Datei darstellen und editieren. Eine Collection bietet die Möglichkeit, Dateien in einem logischen virtuellen Ordner zu organisieren. Mit einem Dependency-Graph lassen sich Dateien zueinander in Relation stellen, so dass eine Linkstruktur zwischen Dateien ausgedrückt werden kann. Das Verweiskonzept eines klassischen hierarchischen Dateisystems wird im ZIBDMS um eine neue Verweisart Weak-Link erweitert, um einen aktualisierbaren, konsistenten und zyklenfreien symbolischen Link anzubieten. In einer hierarchischen Sicht lässt sich durch Verweise ein Baum bilden, in dem eine Navigation und Verweisauflistung möglich sind, was außer ZIBDMS noch kein anderes System bietet.

Abstract

The hierarchical organization of files has dominated computer systems for many years and this will probably not change in the near future. However, with the rapid annual growth of new data this classical hierarchical file system management method reaches its limitations, even when the user consistently sets up and continually expands an elaborated directory structure with well-disciplined naming. This paper introduces new management methods to maintain an overview of all files which can be embedded in the hierarchical file system.

A metadata-based file system, which is based on the hierarchical file system, offers, in addition to hierarchical access, a more flexible and associative access to files via virtual objects by providing query capabilities of index- and metadata which are stored in a database. The query results are presented in a clearly-arranged form of virtual directories and virtual files so that the intuitive hierarchical view continues to be maintained.

The data management system ZIBDMS, which contains a distributed meta-data catalog and a file replication catalog, permits a transparent implementation of these management methods for the NFS interface, CORBA-middleware or web services. Furthermore, ZIBDMS provides other new file management methods that can be integrated into the hierarchical file system view. Metadata in form of attribute-value-pairs can be presented and edited with a virtual file. A Collection makes it possible to organize files in logical virtual folders. Using a Dependency graph, files can be related to each other and expressed through a link structure between these files. In the ZIBDMS the reference concept of a classical hierarchal file system is extended with a new kind of reference called Weak Link in order to provide up-to-date, consistent and cycle-free symbolic links. With the references a tree can be established in a hierarchical view in which navigation and listing of references are possible; this capability is only possible in the ZIBDMS system.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

In der Studie „How Much Information? 2003“ an der University of California, Berkeley [LyVa03] wurde das Jahr 2002 untersucht: wie groß ist der jährliche Zuwachs an neuen Informationen weltweit? Es wurden im Jahr 2002 rund fünf Exabyte[1] neuer Daten auf Papier, Film sowie magnetischen und optischen Medien weltweit geschätzt; davon seien 92 Prozent dieser neuen Daten auf magnetischen Medien, größtenteils auf Festplatten [LyVa03] gespeichert. Im Vergleich zu dem Ergebnis des vorherigen Forschungsberichts für das Jahr 1999/2000 betrage die Zuwachsrate jährlich ca. 30 Prozent an neuen Daten [LyVa03]. Vor diesem Hintergrund ist die Notwendigkeit einer effizienten Datenverwaltung für ein schnelleres Auffinden stark gewachsen.

Bisher werden Daten in Form von Dateien auf Datenträgern gespeichert und meistens in einer Hierarchie von Verzeichnissen organisiert und sie werden in einem hierarchischen Dateisystem verwaltet. In dieser Arbeit sollte untersucht werden, warum die Verwaltung der stetig wachsenden Datenmenge ein klassisches hierarchisches Dateisystem überfordert und welche neuen Verwaltungsmethoden in das hierarchische Dateisystem eingebettet werden können, damit bestehende und neue Daten in der bewährten Hierarchie effizient genutzt werden können. Folgende Anforderungen müssen bei dem Einsatz neuer Verwaltungsmethoden erfüllt werden:

1. Ein Exabyte ist ca. eine Million Terabyte

- Daten sollen dynamisch an jeder Stelle verfügbar sein, wo sie benötigt werden. Der Zugriff auf Dateien soll möglichst schnell und unkompliziert erfolgen.
- Assoziation & Relation: Dateien sollen wie die menschliche Denkweise organisiert werden können.

Am Konrad-Zuse-Zentrum für Informationstechnik Berlin wird derzeit ein neues Datenmanagementsystem namens ZIBDMS[2] entwickelt. Das ZIBDMS verfügt über einen verteilten Metadatenkatalog, in dem Metadaten in Form von Attribut-Wert-Paaren gespeichert werden, und einen verteilten Dateireplikationskatalog, in dem Adressen heterogener Datenquellen aufbewahrt werden. Mit Hilfe dieser beiden Kataloge wird eine einheitliche hierarchische Sicht auf global replizierte Dateien geschaffen.

Am Beispiel des ZIBDMS wird die Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht implementiert, die die o. g. Anforderungen erfüllen. Das Ziel dieser Einbettung ist ein hohes Maß an Transparenz, das die Nutzung neuer Methoden für den Benutzer möglichst einfach und in sich geschlossen macht; ein Nutzer stellt die Verbindung mit dem System auf eine gleiche Art und Weise her, wie die Verbindung mit einem gewöhnlichen entfernten Netzlaufwerk; er kann die vom System zur Verfügung gestellten Funktionalitäten mit den existierenden, unmodifizierten Programmen benutzen und dies bietet zusätzlich den Vorteil, dass eine Automatisierung mit den üblichen Tools ohne zusätzliche Anpassung möglich ist.

Das ZIBDMS hat für die Zielsetzung dieser Arbeit passende Voraussetzungen; der Entwurf des Systems orientiert sich nach dem Design und Prinzipien von UNIX; für die Kommunikation nach außen verfügt das ZIBDMS über eine Vielfalt von Schnittstellen. Besonders nennenswert und für diese Arbeit von Bedeutung ist die Integration von der weit verbreiteten NFS-Schnittstelle ins ZIBDMS [Witt05]. All diese Bedingungen waren Motivation für den Beginn dieser vorliegenden Arbeit.

2. **Z**use-**I**nstitute-**B**erlins **D**ata **M**anagement **S**ystem

1.2 Aufbau dieser Arbeit

Mit der Motivation und den Zielen aus diesem Kapitel werden Einschränkungen der bestehenden Verwaltungsmethoden in einem klassischen hierarchischen Dateisystem diskutiert. Als eine Lösung wird das metadatenbasierte Dateisystem im Detail beschrieben. Hier werden die Indizierung und Metadatenerfassung als Datenbasis für die Anwendung von Suchmethoden und Zugriffsmechanismen betrachtet. Es werden Algorithmen und Suchverfahren aus der Information-Retrieval in einer kompakten Form erläutert, die für ein metadatenbasiertes Dateisystem von besonderer Bedeutung sind. Weitere Verfeinerungen der Techniken sowie Details zu Algorithmen sind der angegebenen Literatur zu entnehmen. Für die Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht werden in dieser Arbeit virtuelle Objekte konzipiert, die auf dem vorhandenen virtuellen Verzeichniskonzept basieren, damit ein assoziativer Zugriff und eine übersichtliche Ergebnisdarstellung möglich werden.

Kapitel 3 gibt einen Einblick in die Entwicklung existierender Systeme, welche verschiedene Verwaltungsmethoden des metadatenbasierten Dateisystems repräsentieren. Am Ende der Beschreibung jedes Systems werden Stärke, Schwäche sowie offene Punkte in die Diskussion gebracht. Abschließend folgt ein kurzer Überblick über weitere metadatenbasierte Systeme, die in vieler Hinsicht mit den in dieser Arbeit beschriebenen Systemen verwandt sind.

Im Kapitel 4 wird das Datenmanagementsystem ZIBDMS mit einem neuen Dateikonzept vorgestellt. In dem ersten Abschnitt dieses Kapitels werden die Architektur des ZIBDMS und die von ZIBDMS-Entwickler zur Verfügung gestellten Komponenten kurz erläutert. Im zweiten Abschnitt werden in dieser Arbeit entstandene Ideen und Konzepte implementiert. In der Komponente `directory-view` wird die Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht des ZIBDMS ausführlich beschrieben. In diesem Abschnitt werden Einzelheiten jeder Unterkomponente erklärt, die jeweils für eine neue Semantik zuständig sind. Am Ende dieses Kapitels erfolgt ein Vergleich von ZIBDMS zu den im Kapitel 3 vorgestellten Systemen.

Abschließend werden Leistungsmessungen und die dazugehörigen Ergebnisse vorgestellt und bewertet. In der Zusammenfassung wird ein Resümee der einzelnen Kapitel gezogen und weitere mögliche Entwicklungen an ZIBDMS diskutiert.

2 Datenorganisation und Zugriffsmethoden im Dateisystem

Ein Dateisystem ist eine Sammlung von Daten, die sich auf einem Permanentspeicher befinden. Die Hauptaufgabe eines Dateisystems besteht darin, einen geeigneten Abstraktionsmechanismus bereitzustellen, um Daten auf einem Speichermedium auf komfortable Art und Weise zugänglich zu machen. Der Benutzer muss keine Details der physischen Organisationsform kennen. Die Daten in einem Dateisystem können in unterschiedlicher Form organisiert sein, die die physische Struktur und logische Struktur umfasst.

Physische Struktur: Als physische Struktur wird ein Schema, das Daten auf die Geometrie eines Datenträgers abbildet, und damit verbundene Algorithmen, um auf Dateien zugreifen zu können, bezeichnet. Beispielsweise werden Festplatten mittels magnetischer Linien in Sektoren eingeteilt, die jeweils 512 Bytes Daten enthalten können. Für die Adressierung großer Festplatten werden mehrere Sektoren logisch in einem vom Dateisystem verwalteten Block zusammengefasst. Bei dem Entwurf eines neuen Dateisystems müssen verschiedene physische Strukturen mit ihren spezifischen Vor- und Nachteilen abgewogen werden, denn eine getroffene Entscheidung beeinflusst die Leistung und Datensicherheit eines Dateisystems in höchstem Maße.

Logische Struktur: Für diese Arbeit ist die logische Struktur bedeutsam, denn sie stellt das Dateisystem dem Anwender dar. Die logische Struktur

katalogisiert Daten, die sich auf einem Massenspeicher befinden, in sogenannten Dateien, die wiederum eine komfortable Datenverwaltung bieten. Die einfachste Form einer solchen Organisation ist die sog. flache Struktur, bei der alle Dateien in einem einzigen Verzeichnis zusammengefasst sind. In diesem flachen Dateisystem sind Dateien eindeutig und werden durch ihre Namen identifiziert. Diese Art Dateisystem wurde in früheren Dateisystemen wie Macintosh File System (MFS) [RHA+85] benutzt. Obwohl sich das hierarchische Schema als Standard in modernen Betriebssystemen [Tann02] durchgesetzt hat, wird das flache Dateisystem heute immer noch in kleinen eingebetteten Systemen eingesetzt.

Auf eine detaillierte Abhandlung der verschiedenen Aspekte bezüglich der physischen Struktur wird in dieser Arbeit verzichtet, da Zugriffsmethoden die logischen Ebene betreffen. Die Arbeit konzentriert sich deshalb auf die logische Struktur, inbesondere die hierarchische Sicht.

In diesem Kapitel werden das traditionelle hierarchische Dateisystem und seine Einschränkungen diskutiert. Die entstehenden Probleme sollen mit Hilfe eines metadaten-basierten Dateisystems gelöst werden, ohne die hierarchische Dateisystemsicht verlassen zu müssen. Dabei werden mehrere Konzepte verschiedener Suchmethoden, Objekte und Zugriffsmechanismen vorgestellt.

2.1 Einschränkungen des hierarchischen Dateisystems

Die am weitesten verbreitete Form des Dateisystems ist das hierarchische Dateisystem, welches Verzeichnisse baumartig abgehend vom Wurzelverzeichnis organisiert. Unterhalb des Wurzelverzeichnisses gibt es ein übergeordnetes Verzeichnis und eventuell ein oder mehrere untergeordnete Verzeichnisse. Jedes dieser Unterverzeichnisse kann wiederum sowohl weitere Dateien als auch eigene Unterverzeichnisse enthalten. Durch ihren hierarchischen Aufbau eignen sich Verzeichnisse zur logischen Organisation von Dateien. Diese Hierarchie reflektiert die Struktur

von Objekten in der realen Welt, denn Menschen organisieren ihre Sachen gerne hierarchisch. Wir versuchen Dokumente, die zusammenhängen, in einem Ordner abzuheften, dann mehrere Ordner gleicher Kategorie bzw. mit gleichem Thema in ein Regal einzuordnen. Die verwandten Regale werden logischerweise in einem Zimmer gelagert. Diese Denkweise wird sich in Zukunft genauso wie die Verbreitung hierarchischer Dateisysteme nicht ändern, dennoch hat diese Art von Dateisystem mehrere Einschränkungen, die über Jahre hinweg noch nicht behoben wurden.

Eine Datei befindet sich nur an einem Ort des Dateisystems

Eine Datei in einem hierarchischen Dateisystem ist eindeutig zugeordnet und lässt sich durch einen Zugriffspfad innerhalb des Verzeichnisbaums identifizieren. Das bedeutet auch, dass eine Datei sich nur an einem Ort befindet. Diese Eigenschaft entspricht nicht immer der Realität, wenn eine Datei nach aktuellem Kontext an mehreren Stellen erscheinen soll. Manche Systeme wie UNIX versuchen das Problem zu umgehen, indem symbolische Links und Hardlinks eingeführt werden. Aber auch dies hat Einschränkungen zur Folge. Ein Hardlink verweist zwar auf eine Datei aber nicht über Partitionsgrenzen hinweg. Außerdem ist es nicht möglich, mit einem Hardlink auf ein Verzeichnis zu verweisen. Dagegen hat ein symbolischer Link nicht die Einschränkungen eines Hardlinks, aber aufgrund der Abhängigkeit der hierarchischen Struktur und deren üblicher Abbildung auf der physischen Ebene hat eine Datei keine Information über die symbolischen Links, die auf sie zeigen. Wenn diese Datei gelöscht oder verschoben wird, werden alle symbolischen Links weiterhin auf den alten Pfad zeigen und werden auch nicht vom System aufgeräumt, denn es besteht eine Hoffnung, dass die Datei irgendwann zurückverschoben bzw. eine neue Datei an ihre Stelle gesetzt wird. Obwohl diese Semantik Vorteile hat, verliert die Datei ihre Multiexistenz-Eigenschaft. Es kann dazu führen, dass diese Datei unter Umständen nicht mehr gefunden wird.

Schlechte Reorganisation

Eine andere Einschränkung des hierarchischen Dateisystems ist die Reorganisation der Daten. Für eine vernünftige Struktur verlässt sich das hierarchische Datei-

system auf die Geschicklichkeit des Benutzers, dass er seine Dateien thematisch nach gewissen semantischen Kriterien im Verzeichnisbaum anordnet und dies auch systematisch weiter verfolgt. Er muss also von Anfang an entscheiden, ob er seine Daten z. B. nach Autor, Projekt, Jahrgang oder selbst definierter Kategorien strukturieren wird. Es kommt aber vor, dass eine Datei mit der Zeit oder durch eine Änderung eine neue Bedeutung gewinnt und daher inhaltlich nicht mehr in ihrem aktuellen Verzeichnis richtig zugeordnet ist. Die Datei muss also an einen anderen Ort oder in ein neues Verzeichnis verschoben werden. Wenn mehrere Dateien und Verzeichnisse in einer komplexen Verzeichnisstruktur neu gestaltet werden sollen, was in der Praxis in vielen Fällen notwendig ist, wird dies recht aufwendig.

Names should mean what, not where [OtGi92]

Das hierarchische Dateisystem (HFS[1]) nimmt keinen Einfluss auf den Namen einer Datei und ihren Speicherort. HFS erlaubt dem Benutzer, seine Dateien in beliebigen Verzeichnissen abzuspeichern. Ebenso überlässt HFS dem Benutzer die Benennung seiner Dateien. Weil HFS keine Information über den Inhalt einer Datei speichert, und die Position einer Datei im Verzeichnisbaum keine vom System definierte semantische Information über die Datei liefert, versucht der Anwender oft mehr Informationen über die Datei in den Dateinamen einzubetten, damit hinterher ein schnelleres Finden möglich wird. Dafür wurde der Dateiname nicht vorgesehen; der Name einer Datei sollte kurz und aussagekräftig vergeben werden.

Assoziation, Kontext und Relation

Im traditionellen Dateisystem werden Dateien nach einer gemeinsamen Eigenschaft in einem Verzeichnis organisiert. Es bietet noch keine weitere Möglichkeit, zwischen ihnen mehrere Verbindungen herzustellen. Assoziationen zwischen Dateien sind wie oben beschrieben eingeschränkt, der Kontext wird in vielen Fällen nicht berücksichtigt und Dateien lassen sich nicht zueinander in Relation stellen. Relationen unter der Berücksichtigung des Kontexts stellen Verbindungen zwischen Dateien automatisch her und eröffnen einen neuen Weg für die Organisation und Wiederfindung der Daten. Mit der neuen Methode werden Dateien miteinan-

1. **H**ierachical **F**ile **S**ystem

der verknüpft und lassen sich beispielsweise mit folgender Abfrage wiederfinden:

```
Gesucht werden PDF-Dokumente als Anlagen aus einer im Jahr
2000 gelöschten Email von John, die während des Herunterladens
von "Windows 2000 SP1" gespeichert wurden.
```

Suchen vs. Finden

Um eine Datei wiederfinden zu können, muss der Benutzer in einem hierarchischen Dateisystem den Dateinamen mit seinem Pfad kennen und das ist für das menschliche Gehirn nicht ganz selbstverständlich und nicht leicht nachvollziehbar. Bei einer kleinen Menge an Daten kann der Anwender die Übersicht noch behalten, aber wenn mit der Zeit ein starker Zuwachs an Informationen entsteht, verliert er leicht den Überblick und kann sich an die gesuchte Datei nur noch undeutlich erinnern. Das Suchen nach einer vergessenen Datei kostet in der Regel viel Zeit, ist sehr unangenehm und führt unter Umständen zu keinem Ergebnis. Das menschliche Gedächtnis lässt aus Informationen wie dem Inhalt der Datei, seine Eigenschaften (z. B. den Typ, die Dateigröße) und den dazugehörigen Kontext (also zu welcher Zeit, Situation und an welchem Ort) zurückerinnern. Dies ist nur möglich, wenn zusätzliche semantische Informationen zu der Datei gespeichert werden.

2.2 Metadatenbasiertes Dateisystem

In einem metadatenbasierten Dateisystem werden Metadaten zusätzlich verwaltet, um die Möglichkeiten des Dateizugriffs zu erweitern. Das metadatenbasierte Dateisystem in dieser Arbeit legt das hierarchische Datenmodell zu Grunde und übernimmt die Aufgaben eines klassischen Dateisystems. Es wird grundsätzlich zwischen den zwei folgenden Ansätzen [VaPa97] unterscheiden:

Integrierter Ansatz

Bei dem integrierten Ansatz hängt das metadatenbasierte Dateisystem mit dem traditionellen Dateisystem integriert in einem System zusammen. Dieses System verwaltet sowohl Dateidaten im traditionellen Dateisystem

als auch semantische Daten im metadatenbasierten Dateisystem und hält sie konsistent durch eine implizite Synchronisation. Es wird zwischen zwei folgenden Arten unterschieden:

- *Lose gekoppelte Systeme:* Das metadatenbasierte Dateisystem ist eine separate Speicherung des traditionellen Dateisystems (z. B. Spotlight [Appl05a], SHORE[2] [CTW+94]). Der Zugriff auf die Daten erfolgt durch einen eigenen Mechanismus.
- *Eng gekoppelte Systeme:* Semantische Daten und Dateidaten werden in einem Dateisystem zur Verwaltung vereinigt, so dass ein neues Dateisystem wie im Beispiel von BeFS [GiBe99] oder WinFS[3] [Grim04, Rect04] entsteht.

Erweiterter Ansatz

Es handelt sich um ein Middleware-File-System, in dem semantische Daten auf einer Abstraktionsschicht über dem traditionellen Dateisystem zur Verfügung gestellt werden. Systeme, die diesen Ansatz verfolgen, können unabhängig von dem darunterliegenden, existierenden Dateisystem entwickelt werden. So ist es möglich, die existierenden Daten ohne weitere Änderungen zu verwalten. Allerdings haben solche Systeme mit dem Konsistenzproblem zu kämpfen, weil sie über Änderungen des darunterliegenden Dateisystems nicht informiert sind. Nach der Art der Abfragebearbeitung werden diese Systeme in einer der zwei Architekturen entworfen:

- *Query-Shipping-Architektur:* Die Abfragen werden direkt an die Datenquellen weitergeleitet, damit die Daten unmittelbar aus den Datenquellen geholt werden. Diese Architektur ist für Systeme wie mediatorbasierte Systeme [Buss02] geeignet, die die Daten aus heterogenen Quellen auslesen. Für jede Datenquelle ist ein sogenannter Wrapper zuständig, der die Anfrage entgegennimmt und in die für die Datenquelle verständliche Abfragesprache umwandelt (*Mapping*). Die

2. A High-Performance, Scalable, Persistent Object Repository. Projekt Homepage: http://www.cs.wisc.edu/shore/
3. **Win**dows **F**uture **S**torage

Ergebnisse einzelner Datenquellen werden durch die Wrapper wiederum in ein einheitliches Mediator-Schema [Buss02] zur Verfügung gestellt. Der Mediator filtert redundante Ergebnisse und stellt sie zu einem Endergebnis zusammen.

- *Index-Shipping-Architektur:* Die Metadaten werden aus Dateidaten extrahiert und auf eine Abstraktionssebene gelegt, auf der die Abfragen verarbeitet werden. Diese Architektur erlaubt eine unkomplizierte Integration des Systems in bestehende Umgebungen und wird deshalb von vielen Systemen [GJSO91, AFMM02] gewählt.

2.2.1 Metadatenerfassung

Metadaten von einer Datei sind Daten, die die Datei beschreiben. Neben dateispezifischen Informationen enthalten Metadaten weitere semantische Angaben zur Dateieigenschaft und zum Inhalt der Datei. In einem metadatenbasierten Dateisystem sind Metadaten wertvolle Informationen, die aus folgenden Quellen erfasst werden können:

- *Aus der Datei selbst:* Metadaten lassen sich bei vielen Dateiformaten[4] zusammen mit den Dateidaten in strukturierter Form[5] speichern.
- *Aus den Dateiattributen:* Es sind Verwaltungsinformationen für die Datei und weitere Attribute (z. B. BeFS [GiBe99], HFS+ [Sure05]).
- *Manuell eingegebene Metadaten:* Es sollte ein Mechanismus zur Verfügung gestellt werden, damit der Benutzer Metadaten selbst eintragen kann.

Eine automatische Metadatenerfassung für jedes Dateiformat kann mit einem Extraktor wie Transducer [GJSO91] oder Importer [Appl05a] durchgeführt werden. Dabei werden Metadaten in Form von Attribut-Werten mit Hilfe einer einzubindenden Programmierbibliothek (z. B. `libextractor`[6]) gesammelt und die gewonnenen Daten werden im metadatenbasierten Dateisystem gespeichert.

4. Sammlung zu diversen Formaten: http://www.wotsit.org
5. beispielsweise Dublin Core Metadata Element Set: http://dublincore.org/documents/dces/
6. http://www.gnu.org/software/libextractor/

2.2.2 Indizierung

Als Basis für eine spätere erfolgreiche Suche wird jedes Dokument manuell, halbautomatisch oder vollautomatisch mit aussagekräftigen Begriffen indiziert. Dies bildet eine komprimierte Version des Dokuments, die den wichtigen Inhalt reflektiert. Zu einem metadatenbasierten Dateisystem müssen die Indizierung und die damit schwer zu bewältigenden Probleme nicht gehören. Es wird aber den Wert eines System erhöhen, wenn Metadaten und Indexe qualitativ kombiniert in die Informationsaufbereitung einfließen. Metadaten sind zwar qualitativ hochwertig, haben aber folgende Schwachpunkte:

- Metadaten sind noch nicht in allen Dateiformaten vorhanden. Für sie ist eine manuell/intellektuelle Eintragung notwendig.
- Metadaten spiegeln nicht immer den kompletten Inhalt eines Dokuments wider.
- Die Erstellung von Metadaten ist zeit- und kostenintensiv. Außer vom Autor selbst sollten die Metadaten eines Dokuments nur von erfahrenen Fachleuten erzeugt und nachträglich geändert werden.

Aus den o. g. Gründen sollten die Indexe als eine sinnvolle Ergänzung zu den Metadaten qualitativ ausgewählt und gespeichert werden, denn sie sind nur nutzbar, wenn sie den Inhalt des Dokuments repräsentieren und bei der Suche zutreffend wiedergeben.

Indexdaten sind eine kompakte stichwortartige Form des Dokuments [SaMc83]. Je nach Art, Anzahl und Position wird die Granularität eines Indexes bei dem Entwurf des Systems entschieden. Dafür werden ausgefeilte Sprachanalyseverfahren eingesetzt [BaRi99, FrBa92, SaMc83, WiMB99], um die intellektuelle Indexierung ersetzen zu können.

Groß-/Kleinschreibung

Anders als bei den Metadaten, wo eine Unterscheidung zwischen Groß- und Kleinschreibung wünschenswert ist, werden bei der Indexierung alle Großbuchstaben in Kleinbuchstaben konvertiert, bevor ein Stichwort in die

Indexmenge aufgenommen wird. Das bringt den Vorteil, dass der Benutzer sich über die genaue Schreibweise eines Suchbegriffes keine Gedanken machen muss.

Stoppliste

Mit einer Stoppliste lässt sich die Anzahl der Indexe schnell reduzieren. In der Stoppliste sind die Hochfrequenzwörter enthalten, die in der Sprache vor allem grammatikalische/syntaktische Funktionen übernehmen und daher keine Relevanz für die Erfassung des Dokumentinhalts besitzen. Sie werden bei der Indexierung nicht indiziert und somit bei der Suche aus den Suchbegriffen entfernt. Eine Negativliste kann je nach Bereich auch unerwünschte Wörter beinhalten. Beispielsweise kann eine Stoppliste für Dokumente aus der Informatik definiert werden und diese Begriffe werden nur einmal indiziert.

Wortstammreduzierung

Wortstammreduzierung[7] ist ein weiterer Ansatz, die aufzunehmenden Indexe zu verbessern. Ein Wort im Dokument befindet sich meistens in einer anderen grammatikalischen Form mit einem Affix[8], als Teil einer Komposition oder in einer anderen Flexionsform. Mit geeigneten Transformationsregeln lassen sich viele dieser morphologischen Varianten eines Wortes auf eine Repräsentation reduzieren. Selbstverständlich braucht man einen geeigneten Algorithmus und Listen wie Suffixliste für jede Sprache. Zu den bekanntesten Stemming Algorithmen [FrBa92] gehört der Porter-Stemming-Algorithmus, der sich mit einer kleinen String-Behandlungssprache Snowball[9] für Englisch, Deutsch und viele andere europäische Sprachen in verschiedenen Programmiersprachen erfolgreich implementieren lässt.

Wenn man mit der Stemming-Technik die Dokumente indiziert, muss man die Suchbegriffe in der Suchabfrage auf ihren Wortstamm

7. engl.: Stemming
8. Bildungssilbe, die als Präfix, Suffix oder Infix zu einem Wortstamm hinzugefügt wird.
9. http://snowball.tartarus.org

zurückführen und nur mit diesen Wortstämmen suchen. Es können somit unter Umständen sehr große Ergebnismengen entstehen.

Thesaurus

Mit dem Thesaurus lassen sich Begriffe untereinander assoziieren. Ein Thesaurus klassifiziert Begriffe nach Thesaurusklassen, also die inhaltlich verwandten Begriffe werden unter einem Repräsentant indiziert. Unter eine Thesaurusklasse können auch alle phonologischen Varianten eines Wortes gruppiert werden. Der Einsatz von Thesauri ist meist nur in speziellen Wissensgebieten sinnvoll und erfordert eine aufwändige Erstellung und Pflege, denn hier wird das beste Wissen über Wortbedeutungen im Zusammenhang mit dem Kontext benötigt.

Wie beim Stemming ist Thesaurus sprachabhängig und hat hinsichtlich der Suchabfragen dieselben Zusammenhänge. Deshalb muss nur ein Konzept für die Verarbeitung entwickelt und implementiert werden.

2.2.3 Suchmethoden

Bevor man mit der Suche beginnt, sollte der Informationsbedarf in einer Fragestellung konkret formuliert werden. Der Inhalt der Fragestellung sollte das Ziel unter der Berücksichtigung des Kontexts möglichst genau widerspiegeln. In diesem Abschnitt werden verschiedene Suchmethoden als Werkzeuge vorgestellt, mit denen präzise und vollständige Ergebnisse entsprechend der Fragestellung erzielt werden können. Bei einer Volltextsuche wird die Fragestellung in Suchbegriffe reduziert, die aber mit weiteren Hilfstechniken präzise Ergebnisse ermöglichen. In einer anderen Suchmethode wird versucht, die Suchfrage in natürlicher Sprache mit linguistischen Verfahren zu verarbeiten. Eine Suchfrage in Form einer Abfragesprache oder logischen Syntax erzielt exakte Treffer, welche für ein metadatenbasiertes System von Bedeutung sind.

2.2.3.1 Volltextsuche

Bei einer Volltextsuche wird die Suchfrage in ein oder mehrere Stichwörter umgesetzt, die repräsentativ für das gesuchte Thema sind. Dafür lassen sich erfolgversprechende Stichwörter sowie Fachbegriffe mit Hilfe von Nachschlagewerken wie Wörterbücher, Lexika, Enzyklopädien abklären. Je mehr präzise Suchbegriffe der Suchtext enthält, desto besser wird das Resultat gefiltert. Es werden oft folgende Erweiterungen kombiniert angeboten:

Phrasensuche

Bei einer Phrasensuche wird die exakte Wortfolge gesucht. Sie ist für das Suchen in den Metadaten geeignet, weil die Begriffe direkt aufeinanderfolgend gespeichert werden. Für eine Suche nach einem Satz in einem Dokument ist die Phrasen-Suche sehr aufwändig. Zum einen müssen alle Suchbegriffe vorhanden sein, zum anderen müssen sie in einer bestimmten Reihenfolge vorkommen. Daher muss auch die Position des jeweiligen Begriffs berechnet und gespeichert werden.

Trunkierungssuche[10]

Bei einer Trunkierungssuche [BaRi99] handelt es sich um das Suchen nach Teilworten. Es werden Universalzeichen in der Suchabfrage benutzt, um ein bestimmtes Zeichenmuster auszudrücken. Dabei fungiert dieses Universalzeichen als ein Platzhalter für ein oder beliebig viele Zeichen. Folgende Zeichen werden häufig verwendet:

- * steht für eine Zeichenkette beliebiger Länge.
- ? ersetzt kein oder ein Zeichen. Man kann ein oder mehrere Fragezeichen verwenden, um eine genaue Anzahl von Zeichen zu ersetzen.

Trunkierung wird vor allem für die morphologische Suche verwendet. Hier werden einzelne Wortformen, Präfixe und Suffixe erkannt und verarbeitet. Dies ist besonders nützlich, wenn man die Schreibweise längerer Begriffe

10. Wildcard-Suche

nicht genau kennt. Man unterscheidet zwischen Linkstrunkierung (Trunkierung am Wortanfang), Innentrunkierung (in der Wortmitte) und Rechtstrunkierung (am Wortende).

Boolesche Suche

Bei der booleschen Suche werden Dokumente nach einzelnen Wörtern durchsucht. Mehrere Suchbegriffe können angegeben und durch logische Operatoren verknüpft werden:

- `AND` alle Suchbegriffe müssen im Ergebnis vorkommen. In vielen Systemen wird dieser Operator mit dem Inklusionsoperator „+“ alternativ angeboten.
- `OR` Mindestens einer der Suchbegriffe muss im Dokument enthalten sein. Dieser Operator sollte möglichst aufgrund der großen Ergebnismenge zuletzt ausgeführt werden.
- `NOT` Das Wort darf nicht im Ergebnis vorkommen. Es wird oft mit dem Exklusionsoperator „-“ realisiert.
- `NEAR` Die Wörter stehen in einem geringen Abstand zueinander. Je näher Wörter beieinander im Dokument stehen, desto besser wird das Dokument in der Ergebnisliste gewichtet. Der Abstandsoperator bietet die Möglichkeit, nahe zusammenliegende Begriffe zumeist im gleichen Zusammenhang stehend, als Alternative zur Phrasensuche ausdrücken zu können. Diese Art Suche wird oft Umgebungssuche genannt.

Zusätzlich werden oft Klammerausdrücke eingesetzt, um eine Verschachtelung der booleschen Ausdrücke und somit eine präzisere Suchabfrage zu formulieren. Die boolesche Suche zeichnet sich durch logische Klarheit aus, bietet eine flexible Eingrenzung der Ergebnismenge und bedarf keiner zusätzlichen Aufbereitung der zu durchsuchenden Dokumente. Deshalb ist sie die meistverwendete Suchmethode und wird als eine Erweiterung der Volltextsuche angeboten.

Fuzzy[11]-Suche

Wenn die Indexierungstechniken wie Stemming und Thesaurus nicht verwendet wurden, kann man die Ergebnismenge mit weiteren relevanten Treffern erweitern, indem die Suchbegriffe mit computerlinguistischen Methoden in verschiedenen grammatikalischen Formen[12] oder leicht variierenden Schreibweisen[13] gesucht werden. Fuzzy-Suche ist in vielen Fällen nützlich, weil der Benutzer seinen Informationsbedarf häufig nur vage formulieren kann.

Fehlertolerante Suche

Falsch geschriebene Wörter in Dokumenten oder Suchabfragen werden bei dieser Methode mitberücksichtigt. Sie sind in der richtigen Schreibweise abzubilden und fehlertolerant abzugleichen. Es werden approximative Suchtechnologien angewendet, um die Treffer im Text melden zu können, auch wenn gewisse Abweichungen vom gesuchten Muster bestehen. Der Levenshtein[14] Algorithmus [Hirs97] ist eine sprachunabhängige Lösung für das approximative Zeichenkettensucheproblem. Bevor der Algorithmus angewendet wird, wird ein ähnliches Wort aus dem Wörterbuch unter Berücksichtigung des aus der Tastaturbelegung möglichen Tippfehlers ausgewählt, um es mit dem vermutlich falsch geschriebenen Wort vergleichen zu können. Der Unterschied zwischen dem fehlerhaften Wort und einem Korrekturvorschlag aus den Wörterbucheinträgen wird mit der Levenshtein-Distanz berechnet, um das fehlerhafte Wort mit möglichst wenigen Editierungsoperationen[15] in den Korrekturvorschlag überführen zu können. Dieser Algorithmus wird auch angewendet, um potentielle Syntaxfehler in der Suchabfrage zu tolerieren.

11. deut.: unscharf
12. z. B. Thesaurus/Flexion
13. phonetische Schreibweise oder alte/neue Rechtschreibung
14. 1965 stellte Vladimir I. Levenshtein den Algorithmus vor.
15. Einfügen, Löschen, Ersetzen

2.2.3.2 Abfrage mit natürlicher Sprache

Menschen formulieren ihren Informationsbedarf bevorzugt in natürlicher Sprache. Es stellt sich die Frage, ob diese natürlichsprachige Abfrage mit Hilfe von linguistischer Verfahren verarbeitet werden kann und korrekte Ergebnisse zurückgeliefert werden können. Gnome Storage[16] ist ein Dokumentmanagementsystem, das in der Lage ist, Abfragen wie „`movies that were directed by spielberg`“ oder „`messages from Brian`“ zu beantworten. Die englische Grammatik[17] basiert auf der generativen HPSG-Grammatik[18], die sich von dem PET Parser [Camm00] parsen lässt. Allerdings werden die Feinheiten der natürlichen Sprache wie Homonyme oder Nuancen in der Bedeutung noch nicht erkannt. Es besteht jedoch die Hoffnung, dass neue Forschungsergebnisse die linguistischen Verfahren in Zukunft attraktiver werden lassen. Die derzeitige Entwicklung erlaubt zumindest den Einsatz automatischer Frage-Antwort-Systeme [SaMc83].

2.2.3.3 Abfragesprache

Mit einer Abfragesprache kann man die Metadaten direkt aus der Datenbank holen. Eine Abfragesprache hat meistens eine strukturierte Syntax (z. B. SQL[19] [KeEi04]) oder die an einer Programmiersprache angelehnte Syntax (z. B. XQuery[20] [ChRZ03]). Diese Syntax ist schwer zu erlernen und deswegen wird sie selten als direkte Suchmethode in einem metadatenbasierten Dateisystem verwendet; vielmehr wird sie beispielsweise von der folgenden logischen Syntax als Übergangssprache benutzt.

2.2.3.4 Logische Syntax

Im Abschnitt 2.2.3.1 wurde die boolesche Suche als eine erweiterte Volltext-Suchmethode vorgestellt. Bei der booleschen Suche werden logische Operatoren eingeführt, um das Vorkommen der Stichwörter in der Datei auszudrücken. Eine Abfrage mit logischer Syntax beinhaltet auch dieselben logischen Operatoren; sie dienen

16. Projekt's Homepage: http://www.gnome.org/~seth/storage/
17. English Resource Grammar, http://lingo.stanford.edu/erg.html
18. Head-Driven Phrase Structure Grammar, http://hpsg.stanford.edu
19. **S**tructured **Q**uery **L**anguage
20. **X**ML **Query** Language, http://www.w3.org/TR/xquery/

aber dazu, mehrere Abfragen als Terme miteinander zu verknüpfen, wobei jeder Term einen booleschen Wert durch einen Vergleichsoperator zugewiesen bekommt. Diese Art von Abfragen ist die meist verwendete Suchmethode in den in dieser Arbeit vorgestellten Systemen, da sie folgende entscheidende Vorteile bietet:

Sprachunabhängigkeit: Eine logische Syntax gilt als Kompromiss zwischen beiden letzten o.g. Suchmethoden. Eine natürliche Sprache ist nicht universal, so dass der Rechercheur die Sprache mit der Grammatik beherrschen muss, um eine Abfrage formulieren zu können. Eine Abfragesprache hängt in der Regel von der darunterliegenden Datenquelle ab und hat oft eine komplizierte Syntax, so dass das Lernen etwas Zeit in Anspruch nimmt.

Präzise Formulierung: Mit Hilfe von Vergleichsoperatoren kann eine Abfrage genau ausgedrückt werden. Mehrere Abfragen lassen sich mit einer Iteration von logischen Operatoren zu einer komplexen Abfrage bilden, so dass Exact-Match[21]-Ergebnisse erzielt werden können.

Intuitive Notation: Die logische Syntax ist insofern einfach zu lernen, weil die Notation auf mathematischen Grundkenntnissen basiert und eine klare Struktur hat. Die Formulierung der gewünschten Ergebnisse lässt sich aus der menschlichen Denkweise schnell ableiten.

Diese Suchmethode hat jedoch eine gewisse Einschränkung bezüglich der Ergebnismenge. Wenn der Rechercheur mit dem Datenbestand nicht vertraut ist, kann er die Ergebnismenge schwer kontrollieren: zu viel erzielte Treffer erzwingen eine weitere Restriktion, zu wenig Treffer erfordern eine Erweiterung der Abfrage. Außerdem muss der Rechercheur seinen Informationsbedarf sorgfältig formulieren und exakt beschreiben, denn jede Bedingungsverletzung kann zu einer falschen bzw. leeren Ergebnismenge führen.

Die Notation der Syntax umfasst folgende Ausdruckkomponenten:

Vergleichoperator

- =, == prüft, ob der linke und rechte Operand gleich sind.

21. Exaktes Suchen in der Retrievaltechnik

- `!=, <>` prüft, ob der linke und rechte Operand ungleich sind.
- `>` prüft, ob der linke Operand größer als der rechte Operand ist.
- `<` prüft, ob der linke Operand kleiner als der rechte Operand ist.
- `>=, =>` prüft, ob der linke Operand größer oder gleich wie der rechte Operand ist.
- `<=, =<` prüft, ob der linke Operand kleiner oder gleich wie der rechte Operand ist.

Logischer Operator

Es können logische Operatoren verwendet werden, um die Formulierung einer Abfrage zu spezifizieren und die Ergebnisse zu verfeinern. Ein logischer Operator beruht auf der booleschen Algebra und gibt einen booleschen Wert zurück.

- `AND, &&` verknüpft zwei Bedingungen miteinander und liefert eine Ergebnismenge zurück, die sowohl die linke als auch rechte Bedingung erfüllt.
- `OR, ||` verknüpft zwei Bedingungen miteinander und liefert eine Ergebnismenge zurück, die mindestens eine Einzelbedingung erfüllt.
- `NOT, !` ist ein unärer Operator[22] und hat eine negierte Funktion, die einen umgekehrten boolschen Wert einer Bedingung zurück liefert.

Neben den o. g. Operatoren können weitere logische Operatoren wie XOR bzw. logische komplementäre Operatoren verwendet werden.

22. Operator, der genau einen Operanden erwartet.

Auswertung und Gruppierung von logisch verknüpften Termen

Über die Reihenfolge der Auswertung von logischen binären Operatoren kann man keine Aussage treffen, denn `AND` und `OR` bilden beispielsweise untereinander gleich stark [EMC+01]. Der Ausdruck *(A `OR` B `AND` C)* steht weder für *((A `OR` B) `AND` C)* noch für *(A `OR` (B `AND` C))*. Hier müssen also runde Klammern zur Gruppierung von logisch verknüpften Elementen gesetzt werden, um die Reihenfolge von Operatoren in Ausdrücken zu ändern. Klammern dienen außerdem dazu, den Ausdruck eindeutig lesbar zu machen.

Doch besitzen Operatoren in der Umsetzung eine Rangfolge, die angibt, in welcher Reihenfolge die Auswertungen durchgeführt werden, wenn mehrere Operatoren ohne Klammern in einem Ausdruck vorkommen. Ähnlich wie die aus der Mathematik bekannte Multiplikation und Division vor Addition und Subtraktion wird `AND` beispielsweise vorrangig vor `OR` abgehandelt. Eingeklammerte Ausdrücke, also die eingeschlossenen Operatoren, werden zuallererst ausgewertet. Dann schließen sich in der Rangfolge die Vergleichoperatoren an, meist gefolgt von den logischen unären Operatoren und danach von den logischen binären Operatoren.

2.2.4 Konzept des virtuellen Objekts

In der realen Welt hat sich der Mensch daran gewöhnt, mit seinen eigenen Sachen virtuell umzugehen und das wird immer beliebter werden, da dies praktisch, schnell und in vielen Fällen auch sicherer ist. Es gehört bereits jetzt zum Alltag, mit Geld virtuell über elektronische Wege zu bezahlen oder es zu transferieren. Der Bedarf an einem virtuellen Zugang zu vielen realen Objekten wird immer größer, insbesondere zu Daten. Medien, die bisher in einer Form von gedruckten Büchern oder Musik-CDs verfügbar sind, werden in der elektronischen Welt zu Dateien, organisiert in Verzeichnissen, zur Verfügung gestellt. Hier könnte man sie auch virtuell verwalten, so dass auf sie aus einem persistenten Speicher über einen virtuellen Pfad (siehe Abschnitt 4.1.4.2, Seite 72) zugegriffen werden kann.

Diese Art des Zugriffs ist sehr flexibel und intuitiv. Es ist weiterhin möglich, ein Objekt über den bisherigen Pfad zu erreichen. Dazu kann auf das Objekt an mehreren Stellen über einen virtuellen Pfad bzw. je nach Semantik über ein virtuelles Objekt zugegriffen werden.

Über die Lebensdauer eines virtuellen Objekts kann man keine Aussage treffen. Wenn das Objekt als temporäres Objekt benutzt wird, hat es meist eine kurze Lebensdauer. Es kann virtuelle Objekte geben, die semantische Informationen zu einer Datei tragen und quasi immer da sind, wenn man sie aufruft. Aus der Sicht des Systems hat dieses Objekt eine kurze Lebensdauer, aus der Benutzersicht hat das Objekt die Lebenslänge einer Datei.

2.2.4.1 Virtuelles Verzeichnis

Im Dateisystem sind Dateien reale Objekte und werden in realen Verzeichnissen organisiert. Doch kann man Dateien aus verschiedenen Orten für eine kurze Zeit in einem Sonderverzeichnis zusammenstellen, um eine bestimmte Semantik zu erfüllen. In der Literatur bezeichnet man ein solches Verzeichnis virtuelles Verzeichnis [GJSO91], semantisches Verzeichnis [GoMa99] oder intelligenten Ordner [Appl05a, Sure05]. Im traditionellen Dateisystem wurde der Zugriff über einen expliziten Pfadnamen oder eine Navigation angeordnet. Diese Navigation beschränkt sich auf nur zwei Richtungen. Man kann sich im Verzeichnisbaum nur nach oben oder nach unten bewegen. In dieser Arbeit wird das Konzept des virtuellen Verzeichnisses erweitert, um einen flexibleren Zugriff auf Dateien zu bieten. Ein virtuelles Verzeichnis zeichnet sich durch folgende Eigenschaften aus:

- Es formuliert eine Abfrage an den darunterliegenden Verzeichnisbaum (*Query-Konsistenz* [GJSO91]) und liefert das Ergebnis mit entsprechendem Inhalt zurück. Der Inhalt dieses semantischen Verzeichnisses ist eine Teilmenge von dem Inhalt des aktuellen Verzeichnisses und seiner darunterliegenden Verzeichnisse. Falls es im System globale Objekte wie `collection` und `dependency-graph` (siehe Abschnitt 4.2) gibt, die zu keiner Hierarchie gehören aber mit einem virtuellen Pfad (siehe Abschnitt 4.1.4.2, Seite 72) ansprechbar sind, werden sie auch als Ergebnis zurückgeliefert.

- Es kann an einem beliebigen Ort des Verzeichnisbaums eingebettet werden und hat das aktuelle Verzeichnis als übergeordnetes Verzeichnis. Hier wird das virtuelle Verzeichnis nicht angezeigt, man kann aber jederzeit hineinwechseln. Wenn das aktuelle Verzeichnis auch ein semantisches Verzeichnis ist, wird der Inhalt des virtuellen Verzeichnisses zu einer Verfeinerung der übergeordneten Abfrage, denn das Verhältnis zwischen einem aktuellen und einem untergeordneten Verzeichnis wird in der Regel mit dem logischen `UND` verknüpft. Da diese Operation eine kommutative Eigenschaft besitzt, lassen sich die virtuellen Verzeichnisse beliebig umordnen. Das Verzeichnis „`/author==Gifford/TYPE==pdf`“ ist äquivalent zu „`/TYPE==pdf/author==Gifford`“ und liefert auch die gleiche Ergebnismenge. Es ist also nicht möglich, es auf einer festen Baumstruktur abzubilden, wie das bei konventionellen Dateisystemen der Fall ist.
- Wenn das System normale Dateien und Verzeichnisse mit dem Namen eines virtuellen Objektes erlaubt, so dass der absolute Pfad gleich dem virtuellen Pfad ist, werden sie auch in der Ergebnismenge zusammengestellt. Diese Mischung zwischen Name und Inhalt für den Pfadnamen ist eine mächtige Methode für die Organisation von Informationen. Beispielsweise kann der Benutzer ein reales Verzeichnis mit dem Pfadnamen von einer gewünschten Abfrage erzeugen und die Dateien, die relevant aber nicht in der Ergebnismenge der Abfrage enthalten sind, in dieses neu erzeugte Verzeichnis verschieben oder verlinken. Ebenso ist es vorstellbar, dass der Benutzer gezielt die irrelevanten Treffer herausfiltern möchte, indem eine Verlinkung der unerwünschten Dateien von der Ergebnismenge in das reale Verzeichnis erstellt wird. Diese Entfernung von nicht relevanten Treffern kann in der Löschoperation von Dateien im semantischen Verzeichnis, aber nicht im realen Verzeichnis implementiert werden. So wird es beim Zugriff mit dem virtuellen Pfad erkannt und die gleichen Dateien, die sich sowohl im realen Verzeichnis als auch im semantischen Verzeichnis befinden, werden herausgefiltert. Auf diese Weise kann der Anwender das Ergebnis nach seinem eigenen Wunsch individuell gestalten. Diese Möglichkeit ist besonders interessant, wenn der

Inhalt des semantischen Verzeichnisses ständig aktualisiert wird und der Anwender dem Ergebnis seiner Abfrage manuell weitere Dateien hinzufügen bzw. die irrelevanten Dateien ausblenden will.

- Der Zugriff auf ein virtuelles Verzeichnis erfolgt auf gleiche Art und Weise wie auf ein reguläres Verzeichnis. Idealerweise sollte der Zugriff auf seinen Inhalt genauso wie auf eine normale Datei oder ein normales Verzeichnis erfolgen. Wenn man ungewöhnliche Methoden benutzt, solle man dafür sorgen, dass sie intuitiv und leicht zu nutzen sind.
- Wenn ein System keine Navigation für semantische Verzeichnisse unterstützt, ist das System nicht viel mehr als eine Suchmaschine. Eine Navigationsfunktion macht die Abfrage und deren Ergebnisse wesentlich flexibler und schneller, weil man das Ergebnis beliebig mit einem neuen virtuellen Verzeichnis verfeinern kann. Es sollte in allen generierten Verzeichnissen innerhalb eines virtuellen Verzeichnisses hin- und zurückgewechselt werden können, soweit die Zugriffsrechte es erlauben.

Da ein virtuelles Verzeichnis im Dateisystembaum platziert werden kann und wie ein normales Verzeichnis ansprechbar sein sollte, hat es alle normalen Attribute wie ein normales Verzeichnis. Allerdings sind die meisten Attribute individuell selbst definierte Attribute, um eine bestimmte Semantik auszudrücken. Beispielsweise wird die Größe des virtuellen Verzeichnisses nicht den vordefinierten Wert bekommen, sondern es wird die Anzahl der beinhalteten Objekte zugewiesen.

2.2.4.2 Virtuelle Datei

Virtuelle Dateien werden im UNIX-Betriebssystem in dem `/proc`-Verzeichnis verwendet, um den einzelnen Programmen und dem Kernel selbst Zugriff auf Geräte zu ermöglichen (siehe Abschnitt 4.2.2). Sie werden zur Laufzeit generiert und liefern einen aktuellen Einblick in die technische Umgebung des Systems. In dieser Arbeit werden weitere interessante Nutzungen für virtuelle Dateien vorgestellt.

Virtuelle Dateien befinden sich nicht auf dem persistenten Speicher, sondern sie werden meist zur Laufzeit generiert, um die ständig aktualisierten Informationen auf dem Laufenden halten zu können. Im Vergleich zu virtuellen

Verzeichnissen haben virtuelle Dateien folgende einfache Eigenschaften:

- Das übergeordnete Verzeichnis einer virtuellen Datei kann ein normales Verzeichnis oder ein semantisches Verzeichnis sein. Die Dateien, die in einem virtuellen Verzeichnis erscheinen, werden als Ergebnis der Abfrage vom System generiert. Ansonsten werden sie vor dem Benutzer vollständig verborgen und nur bei Bedarf mit entsprechender Syntax aufgerufen.
- Obwohl durch den virtuellen Pfadnamen Abfragen an das System gestellt werden, formuliert der Name einer virtuellen Datei selbst keine direkte Abfrage, lediglich eine Semantik. Bei jedem Zugriff wird die virtuelle Datei umgehend mit aktuellen Informationen neu erzeugt. Der Inhalt einer virtuellen Datei kann durch normale Anzeigeprogramme gelesen, ebenso mittels normalen Dateieditoren geschrieben werden. Für dieses Verhalten würde man in vielen Systemen wieder zusätzliche Programme benötigen, die aber logischerweise entfallen sollten.

Eine virtuelle Datei kennzeichnet sich durch ihren Namen, der auch bestimmt, was für eine Semantik sie ausdrückt und wie das System diese interpretieren und verarbeiten soll. Üblicherweise leistet eine virtuelle Datei folgende Aufgaben:

Repräsentation eines realen Objekts

Wenn man ein reales Objekt auf eine gewünschte Art und Weise repräsentieren will, benutzt man virtuelle Dateien als Platzhalter für den generierten Inhalt. Das System erkennt anhand des Namens, was für ein Inhalt zurückgeliefert werden soll, holt die Informationen aus dem realen Objekt heraus und stellt das Ergebnis dem Client zur Verfügung. Ein Beispiel ist die Repräsentation von Dateiattributen im ZIBDMS, attribute-file-layer (Abschnitt 4.2.2). Die Namen „`foo..attr`“ und „`foo..sysattr`“ können benutzt werden, um die getrennte Darstellung mit unterschiedlichen Verarbeitungsmerkmalen von User-Attributen und System-Attributen der Datei „`foo`“ abzubilden. Eine virtuelle Datei muss nicht unbedingt einen Inhalt haben, um ein reales Objekt zu vertreten, sondern der Name selbst kann als eine weitere Art von Repräsentation verwendet werden. „`collectionname{}`“

oder „`collectionname{\foo, \bar}`“ wird im ZIBDMS für ein nicht im hierarchischen Dateisystem taugliches Objekt vom Typ „`COLLECTION`“ mit dem Namen „`collectionname`“ und eventuell seinen Mitgliedern „`/foo`“ und „`/bar`“ dargestellt.

Verweis auf ein anderes Objekt

Viele Systeme [GJSO91, GoMa99] verwenden symbolische Links, um auf eine andere Datei zu verweisen. Den Namen einer virtuellen Datei kann man auch als eine erweiterte Verweisauffassung konstruieren. Hier wird der Name als ein einfacher Zeiger auf eine andere Datei oder ein Verzeichnis benutzt. Im ZIBDMS gibt es eine solche Datei mit dem Namen „`\zibdms\foo`“, die einen Verweis auf einen absoluten realen Pfad „`/zibdms/foo`“ zeigt. Es ist weiterhin möglich, den Namenzeiger für einen virtuellen symbolischen Link (z. B. „`\zibdms\foolink -> foo`“) zu verwenden.

Semantische Information über eine andere Datei

Selbstverständlich kann man semantische Informationen über ein Objekt als Metadaten im Objekt selbst speichern. Es gibt aber viele zusätzliche Informationen, die nicht für alle Objekte vorhanden und nur in bestimmtem Kontext interessant sind. Es geht hier um mehr Auskünfte über ein Objekt, ohne das Objekt in Betracht ziehen zu müssen. Wenn man ein Verzeichnis im normalen Dateisystem auflistet, möchte man beispielsweise wissen, ob ein Objekt verlinkt wird, wie viele Verweise es dazu gibt und welche diese sind. Diese Informationen geben weitere Aussagen darüber, wie wichtig beispielsweise das Objekt ist und es ist gut zu überlegen, ob man das Objekt löscht. Solche semantischen Informationen lassen sich idealerweise über virtuelle Dateien ausgeben. Oft wird der Dateiname verwendet, um bei der Verzeichnisauflistung besondere Auskünfte über bestimmten Einträge zu geben. Zum Beispiel wird im ZIBDMS „`foo@depgrpname.src`“ als Name einer virtuellen Datei zusätzlich zu „`foo`“ bei der Verzeichnisauflistung ausgegeben, wenn „`foo`“ zu den Quellen in dem Dependency-Graph

„depgrpname“ (siehe Abschnitt 4.2.6) gehört. Ein anderes Beispiel ist der Dateiname „colname{\foo, \dir\bar}“. Hier sieht man, dass „foo“ zu einem Collection „colname“ (siehe Abschnitt 4.2.5) gehört und „colname“ zwei Mitglieder „/foo“ und „/dir/bar“ hat. Dies hat einen guten Grund, denn sonst müsste man sich fragen, wie man aus der Liste von Objekten solche Beziehungen bzw. Zusammenhänge zu „foo“ erfahren würde.

2.2.4.3 Virtuelle Operation

Wenn in einem System virtuelle Objekte wie virtuelle Dateien und virtuelle Verzeichnisse definiert sind, muss man Operationen zur Verfügung stellen, um sie zu verarbeiten und zu verwalten. Die Semantik dieser Operationen lässt sich mit den Standard-Befehlen wie im Beispiel von `collection` und `dependency-graph` (siehe Kapitel 4) realisieren. Wenn es im System weitere Objektdaten gibt, die sich semantisch nicht als virtuelle Dateien oder virtuelle Verzeichnisse abbilden lassen, bleibt die Frage offen, wie man mit ihnen umgeht.

Eine Antwort auf diese Frage ist eine neue Entwicklung von Programmen auf der Client-Seite und eine entsprechende Implementierung auf der Serverseite. Wenn man für die Handhabung von Objektdaten nicht auf unmodifizierte Kommandos verzichten will, ist der Einsatz von virtuellen Operationen eine mögliche Alternative. Obwohl virtuelle Operationen nicht über Standard-Befehle erhältlich sind, haben sie die gleiche Semantik wie normale Operationen. Auf dem attribute-file-layer (siehe Abschnitt 4.2.2.4) im ZIBDMS werden Objektdaten wie Attribute oder NSLs mit Hilfe von virtuellen Operationen geschrieben. Für die Verarbeitung dieser Objektdaten sind mindesten zwei elementare Operationen notwendig:

- ***Einfügen:*** Neues Attribut oder NSL[23] wird von Befehlszeilenparametern geparst und an das Objekt im Katalog hinzugefügt. Hier werden virtuelle Dateinamen wie „foo..attr.add“ oder „foo..nsl.add“ als Träger für die virtuelle Operation benutzt.
- ***Löschen:*** Das zu löschende Attribut oder NSL aus Befehlszeilenparametern wird mit Hilfe der virtuellen Dateioperation „foo..attr.rm“ oder

23. **N**ative **S**torage **L**ocation: Adresse, wo das Objekt physisch gespeichert wird.

„`foo..nsl.rm`“ aus dem Katalog entfernt.

Eine weitere virtuelle Operation übernimmt das Assimilieren von Dateien oder Verzeichnissen (siehe Abschnitt 4.2.2.4). Es werden Dateien oder Verzeichnisse übers Netz geholt und die Metadaten automatisch in den Katalog geschrieben.

2.2.4.4 Virtueller Parameter

Ein Kommando in einer Kommandozeile kann mit mehreren Optionen aufgerufen werden. Diese Art Schalter für die Kommandos wird Parameter[24] genannt und ermöglicht dem Benutzer eine Anpassung an seine speziellen Bedürfnisse. Ein virtueller Parameter hat in erster Linie die Funktion eines normalen Parameters, mit dem Unterschied, dass er nicht in der Parameterliste des Kommandos zu finden ist. Durch die Angabe eines virtuellen Parameters kann das Kommando eine neue Semantik verarbeiten, die ursprünglich nicht für das Kommando vorgesehen ist. Die Syntax für virtuelle Parameter erfolgt meist versteckt in einem normalen Parameter.

Ein solcher Parameter wird im ZIBDMS beispielsweise auf der `weak-link-layer` eingeführt, um einen `weak-link` zu erzeugen. Weitere Details werden im Abschnitt 4.2.3 beschrieben.

2.2.5 Zugriffsmechanismen

Ein metadatenbasiertes Dateisystem bietet gegenüber einem hierarchischen Dateisystem weitere flexible Zugriffsarten. Auf der Kommandozeile oder einer graphischen Oberfläche kann man zu Objekten in verschiedenen Wegen navigieren bzw. auf sie zugreifen:

Hierarchischer Zugriff

Bei einem hierarchischen Zugriff dient der Pfadname als Zugriffsmittel, um ein Objekt zu adressieren. Dazu zählen der absolute Pfad, der relative Pfad und Pfadabkürzungen (z. B. die Tilde ~), die leider eine genaue Ort- und Namensangabe

24. auch Argumente genannt

benötigen. Deswegen wird der hierarchische Zugriff in Verbindung mit Suchmethoden und virtuellen Objekten gebracht, um einen assoziativen Zugriff anzubieten.

Assoziativer Zugriff

Ein assoziativer Zugriff ist ein wahlfreier, direkter Zugriff in Form einer Abfrage. Mit den in Abschnitt 2.2.3 vorgestellten Suchmethoden können Objekte im Dateisystem nach Attributen [Hupf03], dem Inhalt (vorgestellte Systeme in Kapitel 3 und 4), dem Kontext (siehe Abschnitt 3.4) oder der Relation [ABB+05] gefunden werden. Das Ergebnis dieser Abfrage wird oft mit Hilfe virtueller Objekte so organisiert und dargestellt, dass Objekte ohne weiteres verfügbar sind.

2.2.6 Ergebnisorganisation

In einem Suchsystem kann der Anwender Informationsbedarf oft nur vage formulieren. Deswegen kann er selten die gewünschten Resultate bei der ersten Abfrage erhalten; sondern er muss die Abfrage nach der ersten Ergebnisinterpretation weiter präzisieren, um die unzutreffenden Ergebnisse zu filtern. Dieser Suchprozess kann mit einer übersichtlichen Darstellung der relevanten Ergebnisse optimiert werden, um Resultate mit hoher Suchgenauigkeit schnell zu liefern.

2.2.6.1 Präzision und Relevanz

Die Qualität einer Suchmethode besteht darin, Suchergebnisse zurückzuliefern, auf die man sich verlassen kann. Die Präzision und die Relevanz sind zwei wichtige Faktoren, womit man die Ergebnisse entsprechend filtern, gruppieren und sortieren kann. Die Präzision ist die Anzahl der zutreffenden Objekte, die relevant sind [SaMc83]. Es muss also sichergestellt werden, dass die zurückgelieferten Ergebnisse auch bei einer unscharfen Suche inhaltlich mit den gewünschten Ergebnissen übereinstimmen. Da in einem großen System viele Abfragen eine sehr große Trefferanzahl nach sich ziehen, wird die Relevanz, also die Einschätzung, wie passend

und aussagekräftig ein Treffer ist, oft eingeführt, um die wichtigsten Treffer an einem am besten zugreifbaren Ort (*Ranking*) einzuordnen. Diese Ergebnisgewichtung kann aus folgenden Methoden errechnet werden:

- ***Indexgewichtung:*** Wenn das System die Indizierung unterstüzt, kann man die Indexgewichtung mit verschiedenen Methoden [BaRi99, FrBa92, SaMc83, WiMB99] berechnen. Eine einfache Methode beruht darauf, dass die Gewichtwerte nach der Worthäufigkeit bzw. Keyword-Dichte (innerhalb einer Datei und der gesamten Datenmenge), nach Wortpositionen (im Titel, am Anfang der Datei usw.) oder nach der relativen Nähe von Worten zueinander berechnet werden können. Ein Beispiel dafür ist das bekannteste Vektorraum-Modell [BaRi99, FrBa92, SaMc83, WiMB99], das in vielen Systemen wie PlanetP [CPMN03] erfolgreich eingesetzt wird.

- ***Popularität:*** Im Gegensatz zur Indexgewichtung, die auf dem Inhalt des Objekts basiert, wird hier die Relevanz nach der Popularität des Objekts bewertet, also wie bekannt und beliebt das Objekt in der Umgebung ist. Dieser Ansatz setzt einen Maßstab für die Wichtigkeit des Objekts in der Umgebung. Die Bewertung kann über manuelles Votum durch den Benutzer, die Anzahl der Zugriffe[25] bzw. Zitate erfolgen. Allerdings muss sichergestellt werden, dass die Relevanzwerte objektiv und unter einer gewissen Toleranz manipulationssicher sind.

  Das PageRank-Konzept [PBMW99] betrachtet die umfassende Link-Struktur des World Wide Webs als eine einzigartige demokratische Einheit und stellt einen relativ trivialen Algorithmus [PBMW99] vor, mit dem die Qualität einer Seite objektiv eingeschätzt werden kann. Eine Seite wird als wichtig eingestuft, wenn sie von mehreren relevanten Seiten verlinkt ist. Die Relevanz einer Seite wird aus der Summe der PageRank-Werte aller eingehenden Links (Backlinks) berechnet. Der PageRank-Wert dieser Seite wird wiederum an die ausgehenden Links gleichmäßig verteilt. Es entsteht dadurch eine rekursive Struktur, die die Bedeutsamkeit aller Seiten untereinander widerspiegelt. Diese zu Patenten [Page01, Bhar03] angemeldete

25. Ranking nach Nutzung

Technologie hat allerdings für das komplexe World Wide Web eine zu langsame Berechnung des PageRank-Wertes, so dass weitere effiziente Techniken zur Beschleunigung benötigt werden. Beispielsweise wird unter der Annahme, dass rund 80 Prozent der Links auf einer beliebigen Seite auf andere Pages der selben Seite verweisen, der PageRank-Wert solcher Seiten in einem Block (BlockRank [KHMG03]) berechnet.

Dieses Verfahren kann auch in einem metadatenbasierten System eingesetzt werden, wenn Dateien untereinander verlinkt sind. Für die Verlinkung zu einer Datei können beispielsweise Zitate für eingehende Links und Referenzen für ausgehende Links verwendet werden. Unter Verwendung der Dateirelation wie Dependency-Graph im ZIBDMS (siehe Abschnitt 4.2.6) kann man die benötigte Linkstruktur herstellen.

2.2.6.2 Ergebnisdarstellung

Eine gute Darstellung der Ergebnisse verschafft dem Anwender einen schnellen Überblick, ob die Treffer zu einer Abfrage abdecken und ob ein Treffer in der Ergebnisliste das gewünschte Ergebnis ist, ohne das Ergebnis öffnen zu müssen. Die Erfolgsaussicht hängt davon ab, wie die Ergebnisse zur Darstellung vorbereitet werden und was für eine Darstellungstechnik angewendet wird.

Organisationsvorbereitung

In der Orgarnisationsvorbereitung werden die Ergebnisse nach einem Kriterium sortiert, wobei eine Relevanzbewertung eine wichtige Rolle spielt. Für eine große Trefferanzahl werden die Ergebnisse in eine Hierarchie gruppiert oder in Listen partitioniert.

- *Partitionierung in Listen:* Am besten sollten die Listen wiederum in ein Array organisiert werden, so dass ein direkter Zugriff auf die einzelne Liste möglich ist. Wenn dies nicht der Fall ist, sollte eine Zeigerstruktur eingeführt werden, um mehrere Listen überspringen zu können und somit die gewünschte Liste mit wenigen Schritten zu erreichen.

- *Hierarchie:* Mit einer Hierarchie werden Gruppen schrittweise immer feiner unterteilt. Es kann die Ähnlichkeit der Ergebnisse vollautomatisch (*Clusteranalyse*) untersucht und eine hierarchische Baumstruktur erzeugt werden. Als eine weitere Hilfe kann man Dateien aus dem Datenbestand schon bei der Indizierung in vordefinierten Themengebieten (z. B. Chemie, Physik) klassifizieren oder Themenkatalogen (z. B. Taxonomie) zuordnen.

Die o. g. Organisationsarten können miteinander kombiniert werden. Als einfacher Ansatz werden Ergebnisse in Listen partitioniert und dann die Listen wenigen (vordefinierten) Ebenen hierarchisch zugeordnet.

Darstellungstechnik

Bei einer Informationsrepräsentation kommt eine graphische Oberfläche oft zum Einsatz, weil man damit komplexe Informationsstrukturen visualisieren kann. Dennoch verliert eine einfache und übersichtliche textuelle Anzeige ihre Bedeutung nicht, solange die mächtige, beliebte Kommandozeile noch im Einsatz ist.

- *Textuelle Anzeige:* Bei einer textuelle Anzeige müssen die Ergebnisse so sortiert, gruppiert und formatiert werden, so dass die potenziellen Treffer an einer veranschaulichten Stelle zugreifbar sind.
- *Graphische Oberfläche:* Mit einer graphischen Oberfläche kann man verschiedene Visualisierungstechniken anbieten. Im Spotlight (siehe Abschnitt 3.3) wird eine benutzerfreundliche Ergebnisdarstellung präsentiert. Im ZIB-DMS-GUI werden die Ergebnisse mit ihren Metadaten in einem Matrix Browser (siehe Abschnitt 4.1.5) visualisiert. Eine graphische Darstellung ist intuitiv, leicht erlernbar und bietet weitere interaktive Nutzungen.

3 Einbettung neuer Verwaltungsmethoden in existierende Systeme

Viele Forschungsgruppen haben die Einschränkungen vom hierarchischen Dateisystem seit langem erkannt. Es gab auch viele Vorstellungen darüber, was ein modernes Dateisystem mit einem flexiblen und schnellen Zugang an Informationen leisten sollte. Die Tatsache, dass das hierarchische Dateisystem über viele Jahre hinweg die Computersysteme beherrscht hat und sich vermutlich auch in Zukunft durchsetzen wird, hat eine Behinderung für den Entwurf eines völlig neuen Dateisystems zur Folge. Schließlich möchte man die bestehenden Informationen, die meist hierarchisch auf das klassische Dateisystem verteilt sind, ohne großen Aufwand an Reorganisation weiter verwenden. Viele Ansätze basieren auf einem Middleware-Dateisystem (User-Level-Dateisystem) und versuchen einen effizienten Zugriff auf eigene Art und Weise zu realisieren.

Zu Beginn erlaubte das im Abschnitt 3.1 beschriebene Semantic-File-System unter Verwendung des virtuellen Verzeichniskonzepts einen assoziierten Zugriff auf indizierte Dateien über die NFS-Schnittstelle. Das XMLFS im Abschnitt 3.2 nutzt die semistrukturierte Eigenschaft der XML-Dokumente, um den in der Form eines Querys navigierbaren Zugriff auf eine normale graphische Oberfläche zu erweitern. Spotlight (Abschnitt 3.3) importiert Metadaten aus Dateien und indiziert den Inhalt getrennt. Auf dieser Grundlage bietet Spotlight eine Volltextsuche mit Stichwörtern oder gemischt mit einer logischen Syntax auf einfachste Art und

Weise. Das brauchbare Ergebnis wird in einer übersichtlichen Darstellung für einen direkten Zugriff bereitgestellt. Als eine Repräsentanz für einen kontextsensitiven Zugriff ermöglicht das im Abschnitt 3.4 beschriebene Context-File-System vernetzten Geräten, in einem Active-Space miteinander zu kommunizieren, wobei der Kontext über Sensoren erkannt oder durch Regeln definiert werden kann.

3.1 SFS - MIT Semantic-File-System

1991 wurde am Massachusetts Institute of Technology (MIT) ein neues Speichersystem entwickelt, das auf dem klassischen Dateisystem basiert aber die Einschränkung des hierarchischen Datenmodells beseitigen soll. Es wurde versucht, semantische Daten aus Dateien zu extrahieren und in einer anderen Abstraktionsschicht dem System zur Verfügung zu stellen. Das MIT-Semantische Dateisystem [GJSO91] (siehe Abbildung 3.1) erlaubt einen attribut-basierten Zugriff auf den Inhalt des klassischen Dateisystems. Dazu wurde das virtuelle Verzeichnis (siehe Abschnitt 3.1.3) konzipiert.

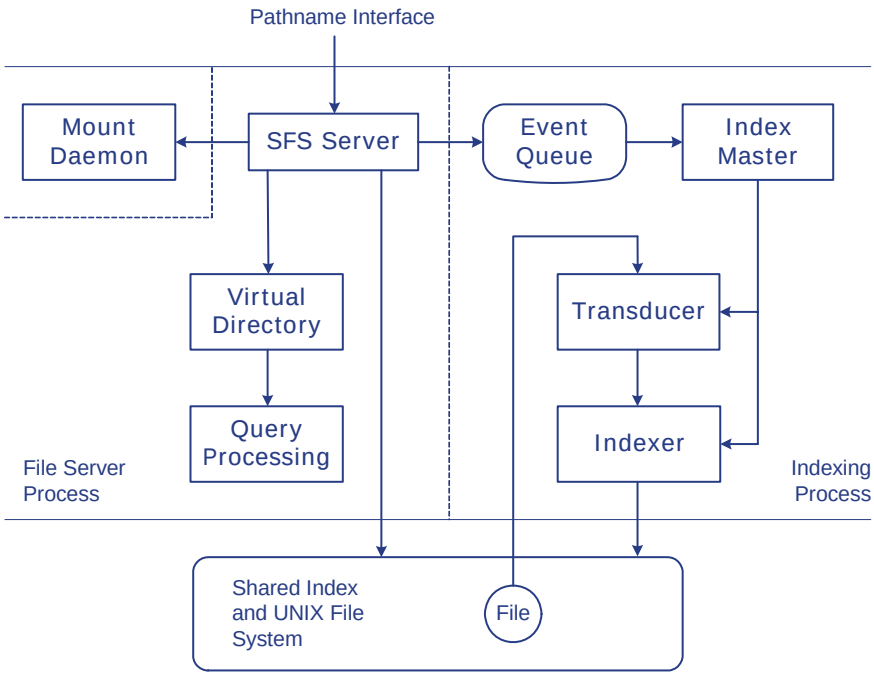

Abb. 3.1 SFS Blockdiagramm [GJSO91]

3.1.1 Indizierung und Metadatenerfassung

Als Grundlage für das neue Speichersystem verwendet SFS das Attribut-Wert-Paar, um die Eigenschaften einer Datei zu beschreiben. Für jeden Dateityp, die Metadaten explizit enthält, wurde ein sogenannter `Transducer`[1] definiert, um die dateispezifischen Metadaten zu extrahieren. Für solche gängigen Dateiformate wie `eMail`, `PDF` oder TEX wurden entsprechende `Transducer`, je nach Dateityp, vom System oder vom Benutzer vorprogrammiert. Es wird zusätzlich ein `Indexer` benutzt, um den Inhalt der Datei zu indizieren. Ein `Index Master` ist zuständig für das Sammeln der Metadaten, steuert den `Transducer`, `Indexer` und aktualisiert diese in regelmäßigen Zeitabständen.

3.1.2 Schnittstelle

Das primäre Ziel von SFS ist die Verwendung von existierenden Dateisystemen, ohne deren Daten und Struktur modifizieren zu müssen. Dazu benötigt SFS eine Schnittstelle, die eine vollständige Kompatibilität mit existierenden baumstrukturierten Dateisystemen gewährleistet und über Rechnergrenzen hinweg interagiert. Es bleibt offen, welches Protokoll man verwenden soll. Es wurde jedoch empfohlen, ein weit verbreitetes Protokoll wie NFS [SUN95] oder AFS [HKM+88] einzusetzen. Für SFS selbst wurde das NFS-Protokoll ausgewählt und implementiert. Alle Zugriffe auf das Dateisystem sind benutzertransparent und geschehen über gewöhnliche NFS-Abfragen.

3.1.3 Abfrage

SFS gehört zu den ersten Dateisystemen, die die Abfrage und Navigation kombinieren. Dies erlaubt einen flexiblen Zugriff auf das darunterliegende Dateisystem. Man kann auf verschiedenen Wegen auf eine Datei zugreifen. Außer dem normalen Weg über die hierarchische Struktur unterstützt SFS einen assoziativen Zugriff, der auf Metadaten und dem Inhalt basiert. Um die Nutzung der Standardschnittstelle

1. Meta data extractor

des hierarchischen Dateisystems nicht verlassen zu müssen, haben die Schöpfer von SFS das Konzept des virtuellen Verzeichnisses vorgestellt.

Ein virtuelles Verzeichnis verhält sich für den Benutzer wie ein normales Verzeichnis. Der Pfadname [GJSO91] wird als Abfrage über die NFS-Schnittstelle an das native Dateisystem interpretiert und das Ergebnis wird als Inhalt des virtuellen Verzeichnisses zur Laufzeit generiert. Eine Abfrage im SFS hat folgende Merkmale:

- Wenn es aus mehreren Termen besteht, werden sie untereinander mit dem booleschen Operator „`AND`“ verknüpft. Syntaktisch wird dies durch das „`/`“-Trennzeichen ausgedrückt.
- Jeder Term ist eine Gleichheitsabfrage, die einen Stringvergleich zwischen einem Attribut und einem Wert formuliert.
- Der Kontext einer Abfrage wird berücksichtigt. Das Ergebnis einer Abfrage innerhalb eines realen Verzeichnisses ist eine Teilmenge des Inhalts dieses realen Verzeichnisses.
- Es gibt ein distinguiertes virtuelles Verzeichnis namens „`field:`“, womit alle verfügbaren Attribute abgefragt werden können:

```
> ls -F /sfs/field:
author:/      dir:/        imports:/    priority:/    title:/
category:/    exports:/    name:/       subject:/     type:/
date:/        ext:/        owner:/      text:/
```

Die Abbildung 3.2 demonstriert die Ergebnisdarstellung einer Abfrage in der Form eines virtuellen Verzeichnisses. Es wird sowohl nach Metadaten als auch nach dem Inhalt aller Dateien innerhalb des realen Verzeichnisses „`/sfs`“ gesucht, die die logische Abfrage „`author==gifford AND text==semantic`“ erfüllen:

```
> ls –F /sfs/author:/gifford/text:/semantic
bio.txt@     mail.txt@     prop.tex@     sfs.tex@
```

Abb. 3.2 Auflistung eines virtuellen Verzeichnisses

Jede gefundene Datei wird als ein symbolischer Link im virtuellen Verzeichnis abgebildet. Das ermöglicht dem Benutzer eine normale Interaktion bezüglich des Ergebnisses, ohne zusätzliche Kommandos benutzen zu müssen.

3.1.4 Bewertung

SFS ist zwar ein guter Ansatz, aber noch keine ausgereifte Lösung für ein Daten-Management-System. SFS bleibt als Grundlage für weitere Entwicklungen:

- Es wird nur String als Datentyp in einer Gleichheitsabfrage unterstützt. Weitere Datentypen wie Zahlen mit weiteren Vergleichsoperatoren sind wünschenswert.
- Ebenso werden alle Terme nur mit dem booleschen UND-Operator verknüpft. Abfragen in der Realität lassen sich nur in der Kombination mit weiteren booleschen algebraischen Operatoren formulieren.
- Weitere Metadaten können nicht vom Benutzer hinzugefügt werden.
- Das Ergebnis in einem virtuellen Verzeichnis besteht unorganisiert aus symbolischen Links. Eine große Ergebnismenge bedarf einer Struktur zur übersichtlichen Ergebnisdarstellung.
- Das Resultat in einem virtuellen Verzeichnis sollte durch weitere Abfragen verfeinert werden.

In der Tat wurde die Idee eines semantischen Dateisystems weiterverfolgt, das Konzept des virtuellen Verzeichnisses erweitert, so dass mehrere Systeme mit interessanten Funktionalitäten und Einsatzgebieten, wie im Folgenden beschrieben, entstanden sind.

3.2 XMLFS

XML File System [AzFM00, AFMM02] wurde als eine Kombination aus hierarchischem Dateisystem und der Informationsretrievaltechnik am IBM Research Lab entwickelt. Anders als andere Systeme, die eine unstrukturierte Information wie SFS [GJSO91] und Presto [DELS99] oder strukturierte Information in einer relationalen Datenbank [Grim04] speichern, basiert XMLFS auf XML Technologie, die eine strukturierte Speicherung [Bune97] der semantischen Information unterstützt, was eine größere Flexibilität beim Erfassen von Daten erlaubt. Im XMLFS

werden die Metadaten in einem Repository für XML Dokumente [ChRZ03] organisiert, um effizientes Durchsuchen und Wiederfinden zu erreichen. Die Entwickler des Projekts haben den Ansatz von MIT Semantic File System verfolgt und bieten einen assoziativen Zugriff unter Verwendung des virtuellen Verzeichniskonzepts an.

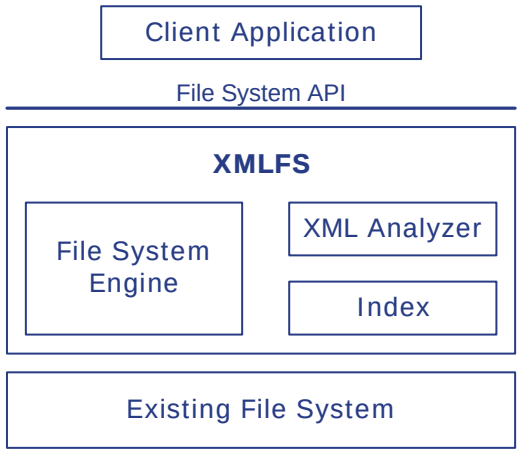

Abb. 3.3 XMLFS-Architektur [AzFM00]

3.2.1 Dateninfrastruktur

Ein einzigartiges Merkmal des XMLFSs ist die automatische Organisation von XML-Dokumenten entsprechend ihrem Kontext. Ein Repository [AFMM02] besteht aus einer Menge von XML Dateien, zieht die Organisation in Betracht, den Kontext, Elemente und Attribute, in die der Inhalt geschrieben wird. Um die Indexierung durchführen zu können, werden zwei Komponenten benötigt:

XML-Index

Wenn ein XML-Dokument in das System neu hinzugefügt oder geändert wird, übernimmt ein mehrstufiges XML-Indexing-Engine die Indexierung und gibt die gewonnenen Indexe an den XML-Analyzer weiter. Es wird nach Wort-Indexen und strukturierten Indexen gesucht und sie werden als invertierte Indexe gespeichert. Ein Wort-Index wird aus dem Inhalt des Dokuments geholt und in einer doppelten Linked-Trie[2] gespeichert. Jeder Wort-

Index hat einen Zeiger zu einer Posting-Liste, in der DateiID, Offset innerhalb der Datei und die Länge des Wortes für jeden Eintrag benötigt werden. Ein strukturierter Index repräsentiert ein Element in dem DOM[3]-Baum und wird mit Hilfe von DTDs[4] erkannt. Außer einer Posting-Liste besitzt jeder strukturierte Index zusätzlich einen Zeiger zu einer Value-Liste, der auf die Position des letzten Buchstabens des Wortes zeigt. Die Einträge der beiden Listen werden über eine Schnittstelle [AFK+02] mit Operationen zur Erzeugung, Aktualisierung und Query-Erkennung in dem Komponenten `XML-Index` (siehe Abbildung 3.3) verwaltet.

XML-Analyzer

Der XML-Analyzer benutzt einen generischen XML-Parser `xml4j`[5] zur Verarbeitung von XML-Dokumenten. Mit einer vorgegebenen DTD präsentiert der Analyzer die allgemeine Struktur eines beliebigen Dokuments. Der XML-Analyzer wird angesprochen, wenn ein Dokument im Repository eingefügt oder geändert wird. Dabei wird das Dokument geparst und aktuelle Indexe geschrieben und anschließend validiert.

Zur Speicherung von XML-Dokumenten berücksichtigt XMLFS zwei unterschiedliche Typen von Repositories: selbstständige (standalone repository) und eingebettete (embedded repository).

Selbstständiges Repository

Ein selbstständiges Repository ist ein komplettes Repository, unterstützt eine direkte Speicherung des Dokuments und liefert an eine externe Schnittstelle. Wenn ein XML Dokument in das Repository eingefügt werden soll, werden die indizierten Informationen, die vom XML Analyzer bereitgestellt wurden, in diesem Repository physikalisch persistent gespeichert. Ebenso wird das Dokument bei einer Delete-Operation zuerst vom Indexierungssystem aufgelöst und dann aus dem persistenten Speicher entfernt. Eine

2. Eine Variante der Indexed-Trie-Datenstruktur zur Speicherung von Worten.
3. **D**ocument **O**ject **M**odel [ChRZ03]
4. **D**ocument **T**ype **D**efinition [ChRZ03]
5. Wird als Apache's Xerces-J Parser weiterentwickelt. Siehe http://xml.apache.org/xerces-j/

Update-Operation ist eine Folge von einer Delete-Operation und einer Additions-Operation.

Eingebettetes Repository

Um mit der Schnittstelle nach außen oder mit der Speicherverwaltung interagieren zu können, braucht das eingebettete Repository ein Host-Repository. Das eingebettete Repository selbst unterstützt nur Verzeichnisse und hat nur zwei Komponenten, den XML-Index und XML-Analyzer. Die Antwort auf alle Operationen des Clients übernimmt das Host-Repository und präsentiert die gespeicherten Dokumente in hierarchischer Sicht.

3.2.2 Querying-Capabilities

Aus dem Repository lassen sich Daten über eine Abfrage abrufen. Jedes Query [AFK+02] ist einen Attribut-Wert-Vergleich. Ein Query kann folgende Komponenten enthalten:

Wild-Card

Der String eines Querys kann Wild-Card-Zeichen beinhalten, allerdings auf Grund der baumartigen Indexspeicherungsstruktur unterstützt XMLFS nur eine Rechtstrunkierung, die einen Bereich am Ende des Strings offen lässt.

Query-Operator

Der Attribut-Wert-Vergleich muss einen Query-Operator beinhalten. Es kann sowohl ein exakter Vergleich mit einem „=“ als auch eine Suche in einem Bereich mit den {<, >} -Operatoren erfolgen.

Im XMLFS lassen sich mehrere Queries logisch miteinander verknüpfen. Boolesche Operatoren wie „AND“ und „OR“ sind mächtige Unterstützungen für eine genauere Formulierung einer Abfrage. Die Abbildung 3.4 zeigt ein Ergebnis eines kontext-sensitiven Querys aus hierarchischer Sicht. Man kann in diesem Baum mit Hilfe von DTDs navigieren. Dieses Query beinhaltet einen logischen Operator „AND“ als Ergebnisverfeinerung und liefert das Ergebnis in einem virtuellen

Verzeichnis unter Berücksichtigung des spezifischen Kontexts. Der Kontext in einem Query startet von dem obersten Element des XML-Dokuments und wird in einer XPath-ähnlichen Art [AFK+02] spezifiziert.

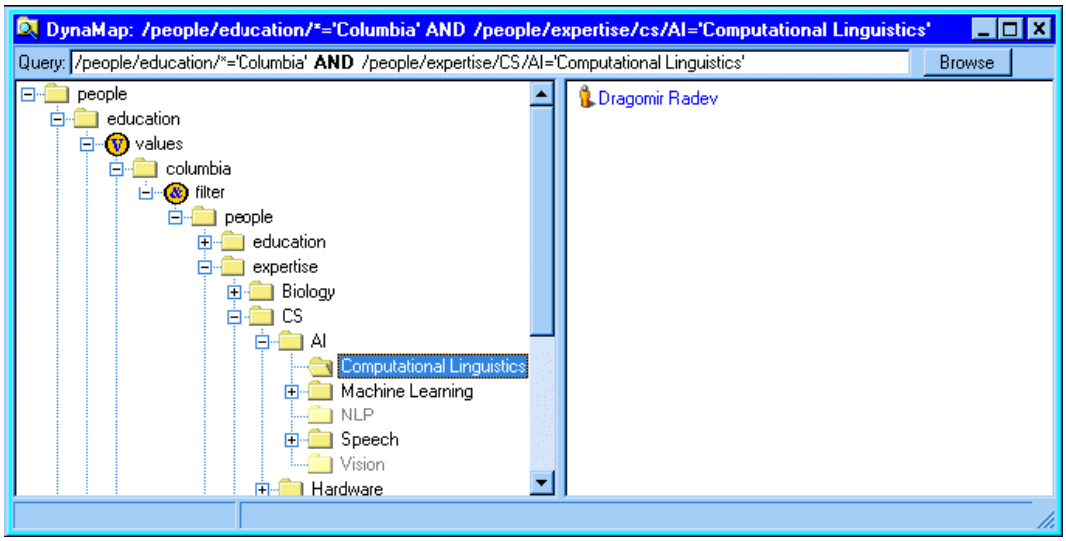

Abb. 3.4 Beispiel eines Querys im XMLFS [AFMM02]

3.2.3 Schnittstelle

Für die Kommunikation mit dem Client setzt sich XMLFS auf das NFS-Protokoll Version 2 (RFC1094)[6]. Um XML-Dokumente zu speichern, abzurufen oder zu modifizieren wird ein unveränderter, standardisierter NFS-Server implementiert, der die Abfrage (NFS Request) entgegennimmt und entweder in eine Operation auf das Speichersystem tranformiert oder als ein Query interpretiert. Das Ergebnis wird als NFS-Response zurückgegeben, so dass keine Modifizierung auf der Client-Seite für die Kommunikation mit dem XML-Repository notwendig ist. Einzelne Funktionen im NFS-Protokoll werden wie folgt verarbeitet:

- `mkdir()` und `rmdir()` sind nicht erlaubt.
- `readdir()` wird an das Indexing-Engine weitergeleitet. Das Lesen eines Verzeichnisses wird als ein Query behandelt.

6. http://www.ietf.org/rfc/rfc1094.txt

- `lookup()` und `getattr()` werden je nach Objekttyp unterschiedlich behandelt. Für eine normale Datei wird die Operation an das Speichersystem geleitet. Für ein Verzeichnis wird eine arbiträre Antwort zurückgegeben.
- `setattr()`: Bei einer Datei wird die Operation an das Speichersystem weitergeleitet, dagegen bekommt `setattr()` auf ein Verzeichnis eine Fehlermeldung zurückgeliefert.
- `remove()`: Zuerst werden die Indexe aktualisiert, danach wird die Anfrage an das Speichersystem transferiert.
- `rename()`: Bei einer Datei wird die Operation an das Speichersystem transferiert und die Indexe werden entsprechend der Dateinamenänderung aktualisiert. `rename()` ist nicht gültig für ein Verzeichnis.
- `read()`, `create()` und `statfs()` werden an das Speichersystem geleitet.
- `write()` wird an das Speichersystem weitergeleitet. Es wird ein Thread für die Aktualisierung der Indexe gestartet.

Das Beispiel aus der Abbildung 3.5 betrachtet ein Interaktionsszenario zwischen einem unveränderten Windows-Explorer und XMLFS über einen Mount-Point des NFS-Servers (das Netzlaufwerk „`F:`“). Es wird gezeigt, dass eine logische Abfrage hierarchisch mit einer graphischen Oberfläche strukturiert werden kann. Der Benutzer kann im Verzeichnisbaum navigieren und auf ein einzelnes virtuelles Verzeichnis zugreifen bzw. Dateien öffnen. Für eine Abfrage wie beispielsweise

```
name=atlantic and ticker=nasdaq
```

öffnet der Benutzer zunächst das „`Nasdaq`“-Verzeichnis, das alle Dateien innerhalb des Kontexts „`/profile/ticker`“ beinhaltet. In diesem Verzeichnis gibt es die Möglichkeit, das vorläufige Ergebnis mit einem „`and`“-Unterverzeichnis einzuengen. Darunter wird er sein gewünschtes Ergebnis in einem virtuellen Verzeichnis mit dem kompletten Pfad

```
/profile/ticker/Nasdaq/and/profile/name/Atlantic
```

finden. Dieses Verzeichnis listet alle Dateien auf, die das Wort „`Nasdaq`“ mit dem

Kontext „`/profile/ticker`“ und das Wort „`Atlantic`“ mit dem Kontext „`/profile/name`“ enthalten. Diese Dateien lassen sich mit einer normalen Anwendung wie `Windows Wordpad` problemlos öffnen.

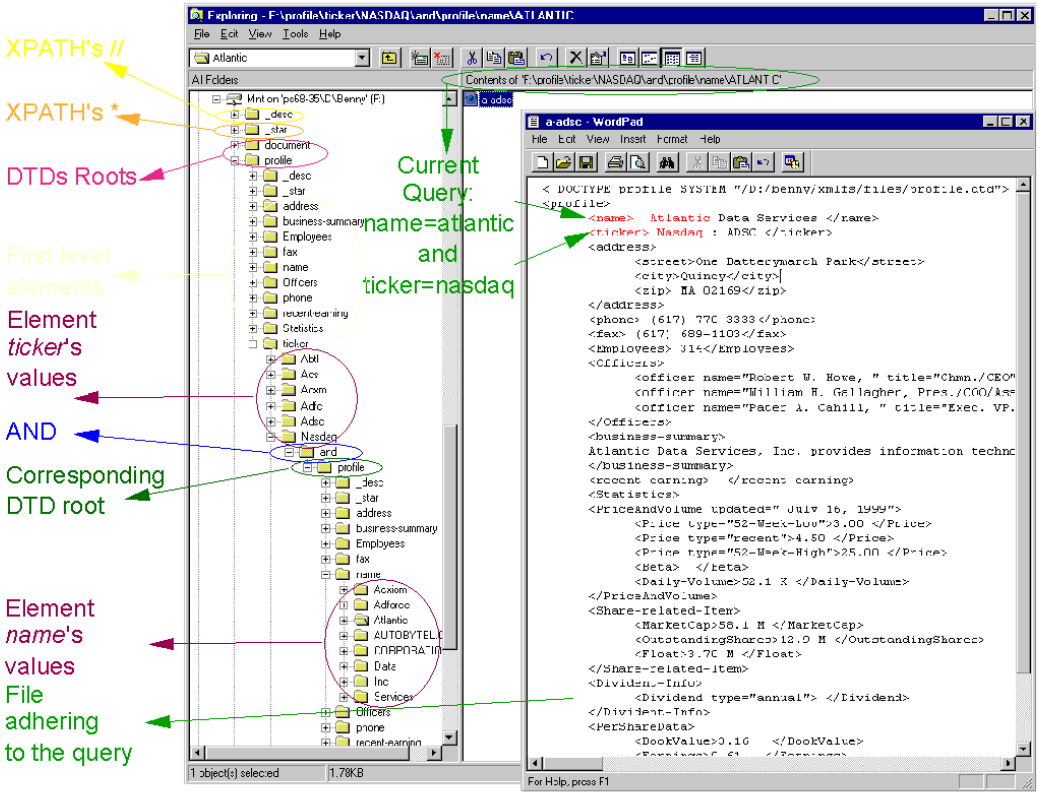

Abb. 3.5 XMLFS-Interaktionsszenario über NFS-Schnittstelle [AFMM02]

3.2.4 Bewertung

XMLFS benutzt die Information-Retrievaltechnik in einem Dateisystem, um den assoziativen Zugriff auf XML-Dokumente zu ermöglichen. Der Inhalt von Dateien wird indiziert und in einer hierarchischen Sicht von virtuellen Verzeichnissen zur Verfügung gestellt. Obwohl XMLFS eine effiziente Methode für das Suchen und Browsern von Daten zeigt, hat es jedoch einen Nachteil bezüglich der Performance von der XML-Technologie. Daher kann dieser Ansatz in großen Systemen problematisch werden.

3.3 Spotlight

Anfang der 90er Jahre begann Apple mit der Entwicklung eines neuen ambitionierten Betriebssystems namens Copland, das als Nachfolge von Mac OS Classic antreten sollte. Ein wichtiger Bestandteil von Copland war die Volltextsuche Sherlock, die nach der Einstellung des Projekts separat in Mac OS 9 eingeflossen hat. Leider konnte der ursprüngliche Plan nicht durchgesetzt werden. Zum einen waren die Macs nicht schnell genug, um Dateien ständig im Hintergrund zu indizieren, zum anderen lieferte Sherlock oft unbefriedigende Ergebnisse. Für die heutigen, leistungsfähigen Macs wurde diese Suchtechnologie komplett überarbeitet und bekam einen neuen Namen: Spotlight.

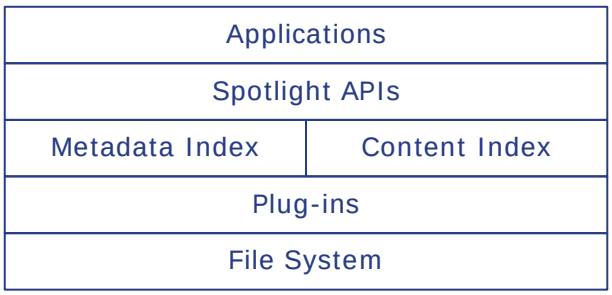

Abb. 3.6 Spotlight-Architektur [Appl05a]

Spotlight [Sure05, Appl05a] bietet eine einfache Volltextsuche nach Dateiattributen und Dateiinhalten, die so schnell arbeitet, dass man die ersten Ergebnisse schon beim Tippen des Suchbegriffs (*live queries*[7]) bekommt. Die Ergebnisse werden mit einer hohen Suchgenauigkeit zurückgeliefert und in einer sehr übersichtlichen, gruppierbaren Darstellung präsentiert. Die Qualität der Ergebnisse wird durch die im Folgenden beschriebene komplexe Technik gewährleistet.

3.3.1 Indizierung und Metadatenerfassung

Apple setzt die Such-Engine Spotlight benutzertransparent auf Systemebene an, in dem ein Hintergrundprozess die intern geführten Indexdaten und Metadaten für neu hinzugefügte, bearbeitete oder gelöschte Dateien ständig aktualisiert und

7. up-to-the-moment Ergebnisse, schon im BeOS [GiBe99] vorhanden.

optimiert, so dass die Suche eine lineare Performance [Appl05a] erzielen kann. Spotlight sammelt Indexdaten und Metadaten aus Dateien getrennt (siehe Abbildung 3.6) und legt zwei Datenbanken separat im Verzeichnis „`/.Spotlight-V100`“ an, um das zugrunde liegende Dateisystem HFS+ [Sure05] nicht verändern zu müssen. Die Metadatenerfassung im Spotlight hängt wie alle anderen Systemen davon ab, ob das Format vom System unterstützt wird. Jedes neue Format lässt sich durch Plug-ins (*Metadata-Importers*) erweitern. Spotlight selbst unterstützt eine Vielzahl von Dateiformaten, die für den alltäglichen Gebrauch von Bedeutung sind. Die Indizierung extrahiert den Dateiinhalt in optimierten Indexdaten und sorgt dafür, dass genaue und relevante Ergebnisse zurückgeliefert werden können. Leider ist es nicht bekannt, was für Indizierungstechniken Spotlight benutzt, aber aus [Appl05a, Appl05b, Appl05e] lassen sich folgende Techniken erkennen, die im Kapitel 2 beschrieben wurden:

- *Groß-/Kleinschreibung:* Metadaten werden unverändert in Form von Attribut-Wert-Paaren erfasst. Dagegen sei eine Unterscheidung von Groß- und Kleinschreibung bei Indexdaten nicht nötig, weil dies bei der Volltextsuche ignoriert wird.
- *Keine Stoppliste:* Es wurde keine Stoppliste eingeführt. Spotlight liefert zu den Hochfrequenzbegriffen im Englischen wie „`a`“ oder „`the`“ eine entsprechend hohe Anzahl von Ergebnissen zurück.

Spotlight verfolgt den intergrierten Ansatz eines lose gekoppelten Systems (mehr dazu im Abschnitt 2.2, Seite 10). Dies bringt den großen Vorteil, dass das System über alle Dateiänderungen implizit informiert wird. Außerdem verfügt Spotlight aufgrund der systemnahen Realisierung auf der Systemebene eine enge Kooperation mit anderen Anwendungen. Die Anwendung kann die von Spotlight zur Verfügung gestellten APIs benutzen, um die Suche innerhalb der Anwendung zu ermöglichen. Dagegen kann eine Anwendung sinnvolle Metadaten im bestimmten Kontext für Spotlight liefern. Die Abbildung 3.7 zeigt eine Auflistung der Attribut-Wert-Paare einer Datei, wobei die Werte bei Mehrwert-Attributen in eine Liste dargestellt werden. Die Aufmerksamkeit liegt hier auf der letzten Zeile, in der das Attribut „`kMDItemWhereFroms`“ mit seinem Wert angezeigt wird. Dieser Eintrag

wurde automatisch durch den Internet Browser `Safari` [Sure05] eingetragen, als die Datei „`anatomy.pdf`“ im Laufe dieser Arbeit mit `Safari` von der gelisteten Adresse heruntergeladen wurde. Solche Attribute können in vielen Situationen wie beim in der Abbildung 3.8 geschilderten Problem sehr hilfreich sein.

```
> mdls anatomy.pdf
anatomy.pdf -------------
kMDItemAttributeChangeDate     = 2005-07-29 17:11:19 +0200
kMDItemAuthors                 = (carl)
kMDItemContentCreationDate     = 2005-04-06 00:12:07 +0200
kMDItemContentModificationDate = 2005-04-06 00:12:07 +0200
kMDItemContentType             = "com.adobe.pdf"
kMDItemContentTypeTree         = (
    "com.adobe.pdf",
    "public.data",
    "public.item",
    "public.composite-content",
    "public.content"
)
kMDItemCreator                 = "Pscript.dll Version 5.0"
kMDItemDisplayName             = "anatomy.pdf"
kMDItemEncodingApplications    = ("Acrobat Distiller 4.05 for
Windows")
kMDItemFSContentChangeDate     = 2005-04-06 00:12:07 +0200
kMDItemFSCreationDate          = 2005-04-06 00:12:07 +0200
kMDItemFSCreatorCode           = 0
kMDItemFSFinderFlags           = 0
kMDItemFSInvisible             = 0
kMDItemFSLabel                 = 0
kMDItemFSName                  = "anatomy.pdf"
kMDItemFSNodeCount             = 0
kMDItemFSOwnerGroupID          = 501
kMDItemFSOwnerUserID           = 501
kMDItemFSSize                  = 308123
kMDItemFSTypeCode              = 0
kMDItemID                      = 571569
kMDItemKind                    = "PDF-Dokument"
kMDItemLastUsedDate            = 2005-04-05 23:12:07 +0200
kMDItemNumberOfPages           = 25
kMDItemPageHeight              = 792
kMDItemPageWidth               = 612
kMDItemSecurityMethod          = "None"
kMDItemTitle                   = "Microsoft Word - p_final.doc"
kMDItemUsedDates               = (2005-04-05 23:12:07 +0200)
kMDItemVersion                 = "1.3"
kMDItemWhereFroms              = ("http://www.globus.org/alliance/
publications/papers/anatomy.pdf")
```

Abb. 3.7 Auflistung von Metadaten einer Datei im Terminal

3.3.2 Suchmethode

Spotlight bietet sowohl eine einfache Volltextsuche als auch eine Suche mit logischer Syntax (siehe Abschnitt 2.2.3). Abfragen der beiden Suchmethoden werden im Hintergrund in entsprechende Methode der Spotlight-APIs [Appl05b, Appl05c] transformiert. Im Folgenden werden Einzelheiten der beiden Methoden erläutert.

3.3.2.1 Volltextsuche

Eine Volltextsuche im Spotlight ist einfach, beinhaltet aber viele intelligente, transparente Mechanismen (siehe Abschnitt 2.2.3.1), so dass Resultate mit einer hohen Treffgenauigkeit ohne weiteres erzielt werden können:

- *Implizite Trunkierungssuche:* Für einen Suchbegriff folgen eine Linkstrunkierungssuche und eine Rechtstrunkierungssuche automatisch, ohne ein Jokerzeichen anhängen zu müssen. Spotlight ist intelligent genug, um eine Entscheidung automatisch zu treffen, ob und welche Trunkierungssuche gestartet werden soll. Beispielsweise für die Suche nach „Paris“ werden keine Ergebnisse zu „com*paris*on“ zurückgeliefert [Appl05a].
- *Fuzzy-Suche:* Es werden viele Schreibvarianten zu einem Suchbegriff mitgesucht. Beispielsweise sind „ß“ und „ss“ [Appl05a] sowie „Frédéric“ und „Frederic“ [Appl05b] äquivalent.
- *Boolesche Suche:* Wenn ein logischer Operator für einen Suchbegriff benutzt wird, wird dieser Suchbegriff nur in der Index-Datenbank gesucht; der Suchbegriff wird also nicht als Metadaten verstanden. Folgende Syntaxen können miteinander kombiniert werden:

+	Der Suchbegriff muss im Inhalt der Datei vorkommen.
-	Der Suchbegriff darf nicht im Inhalt der Datei vorkommen.
&	AND-Verknüpfung.
|	OR-Verknüpfung.
( )	Gruppierung von Verknüpfungen.

Wenn die Volltextsuche viele Resultate zurückliefert, kann man sie mit Schlüsselwörtern eingrenzen. In [Sure05] werden Schlüsselwörter wie „`kind:email`“ ode „`date:yesterday`“ aufgelistet, die speziell für die Volltextsuche vorgesehen sind.

3.3.2.2 Logische Syntax

Die Suche mit einer logischen Syntax kann über die Kommandozeile oder mit einem intelligenten Ordner folgen. Die Suche mit einem intelligenten Ordner wird im Abschnitt 3.3.4 beschrieben und die Schwäche der Abfrageformulierung aufgezeigt. Dagegen ist eine logische Syntax auf der Kommandozeile sehr mächtig und wesentlich flexibler.

Für eine Abfrage mit der logischen Syntax bietet Spotlight neben die im Abschnitt 2.2.3.4 beschriebene Syntax noch eine Kombination mit der Trunkierungssuche sowie neue Vergleichsmodifikatoren [Appl05b].

- *Wild-Card:* Das Jokerzeichen „*“ kann am Anfang, am Ende oder innerhalb der Zeichenkette vorkommen, um die Vertretung eines Teilwortes auszudrücken.
- *Vergleichsmodifikator:* Der Vergleich bei der logischen Syntax ist eine Exact-Match-Suche, deswegen erlaubt Spotlight eine Fuzzy-Suche, indem weitere Suffixe an dem Wert des Attributes angehängt werden:

`c`	Keine Unterscheidung der Groß-/Kleinschreibung.
`d`	Weitere Schreibvarianten werden berücksichtigt.
`w`	Prüft, ob das Ergebnis den Wert als ein eigenständiges Wort beinhaltet. Zum Beispiel für die Suche nach „sicher“ werden „sicher“, „nicht sicher“, „sicherstellen“, „zielsicher“ aber nicht „versicherung“ zurückgeliefert.

Um eine Abfrage mit der logischen Syntax auf der Kommandozeile zu formulieren, braucht man eine Liste von Attributnamen [Appl05d]. Aus dieser Liste der verfügbaren Attributen werden die benötigten Attributnamen ausgewählt und bilden zusammen mit den Operatoren [Appl05e] eine Abfrage. In diesem Abschnitt werden die Mächtigkeit und die Flexibilität einer Abfrage auf der Kommandozeile anhand

eines Beipiels erläutert. Die Abbildung 3.8 beschreibt eine Suche nach Dateien, an die man sich nur im Zusammenhang mit einem Kontext erinnern kann und nicht weiss, wo und unter welchem Namen diese Dateien gespeichert wurden.

```
Gesucht werden alle veröffentlichten Publikationen von Herrn
Carl Kesselman, die das Wort "Grid" beinhalten und von
"globus.org" heruntergeladen wurden.
```

Abb. 3.8 Eine zu lösende Aufgabe mit der Spotlight-Kommandozeile

Aufgrund der relationalen Zuordnung des menschlichen Gedächtnisses sind zumeist nur solche Informationen für eine Suche verfügbar. Für die o. g. Aufgabe kann eine entsprechende Abfrageformulierung in die logische Syntax umgewandelt werden

```
> mdfind -live "kMDItemAuthors='*Carl*'cd\
&&(kMDItemContentType=*pdf||kMDItemContentType=*postscript)\
&&kMDItemTextContent=Grid&&kMDItemWhereFroms=*globus.org*"
/Users/zibdms/Desktop/anatomy.pdf
[Type ctrl-C to exit]
```

Abb. 3.9 Abfrage mit einer logischen Syntax auf der Kommandozeile

Wenn man eine Metadatenauflistung eines Dokument wie in der Abbildung 3.7 schon mal gesehen hat, lässt sich diese ziemlich kryptische Syntax leicht erklären. Das Kommando „`mdfind -live`“ in der ersten Zeile gibt an, dass es sich um ein Live-Query (siehe Abschnitt 3.3.3) handelt. Anschließend fängt die erste Angabe mit dem Namen des Autors an; „`kMDItemAuthors='*Carl*'cd`“ sucht alle Resultate mit verschiedenen Schreibweisen, die „`Carl`“ beinhalten. Die zweite Zeile enthält eine weitere Angabe zur Publikation, die mit Hilfe einer `OR`-Gruppe ausgedrückt wird. Die letzte Zeile wird der Inhalt mit dem Attribut „`kMDItemTextContent`“ gesucht und die Herkunft des Herunterladens benutzt eine Trunkierungssuche, weil die URL das Teilwort „`globus.org`“ sicher enthalten muss. Die Treffgenauigkeit des Spotlights kann man beispielsweise anhand der Metadaten in der Abbildung 3.7 bestätigen.

Auf einer Kommandozeile präsentiert Spotlight die Ergebnisse in Form einer

einfachen Liste mit den absoluten Pfaden. Es handelt sich hier um eine globale Suche; das aktuelle Verzeichnis wird dabei nicht berücksichtigt, dass die Suche nur im aktuellen Verzeichnis und in Unterverzeichnissen stattfindet. Dafür bietet Spotlight die Option „`-onlyin`„ um die Suche in einem bestimmten Verzeichnis zu veranlassen. Die Ergebnisliste ist eine normale textuelle Darstellung und bietet keinen direkten Zugriff auf die Ergebnisse, weil sie nicht in einem virtuellen Verzeichnis wie in einem intelligenten Ordner (Abschnitt 3.3.4) organisiert werden.

3.3.3 Ergebnisdarstellung

Spotlight verwendet die Live-Queries [GiBe99], um Ergebnisse schon beim Eingeben des ersten Buchstabens zu präsentieren und danach zu aktualisieren. Ein Live-Query im Spotlight hat die zwei folgenden Phasen, die sich über Spotlight-APIs [Appl05c] steuern lassen:

- *Ergebnisermittlungsphase:* Beim Tippen der Suchbegriffe werden die ersten Ergebnisse geliefert. Die Suche beim Spotlight geht weiter, bis alle möglichen Ergebnisse aus der Datenbank nach und nach gefunden werden; dann gibt Spotlight eine Mitteilung an die Anwendung und beendet diese Phase.
- *Live-Update-Phase:* In diese Phase werden alle Änderungen der Ergebnisse aktualisiert. Dateiänderungen oder neue passende Dateien fließen automatisch in das Ergebnis ein. Die Anwendung bekommt ständig entsprechende Meldungen, um die Ergebnisdarstellung zu aktualisieren.

In dem vorherigen Abschnitt wird das Ergebnis eines Live-Querys textuell dargestellt; in diesem Abschnitt wird eine graphische Darstellung mit einem direkten Zugriff vorgestellt. Die Abbildung 3.10 zeigt die Ergebnisdarstellung der Abfrage „`+Grid Carl Kesselman`“, über die alle Dateien mit „`Grid`“ aus der Indexdatenbank und mit „`Carl`“ und „`Kesselman`“ aus beiden Datenbanken gefunden werden sollen. Das Ergebnis wird zeilenweise oder als Miniaturansicht in einer überschaubaren Darstellung angezeigt. Diese graphische Oberfläche bietet mehrere Darstellungsoptionen zur Gruppierung oder zur Sortierung. Jedes Resultat lässt sich direkt zugreifen, ohne den Ort des gefundenen Objekts wissen zu müssen.

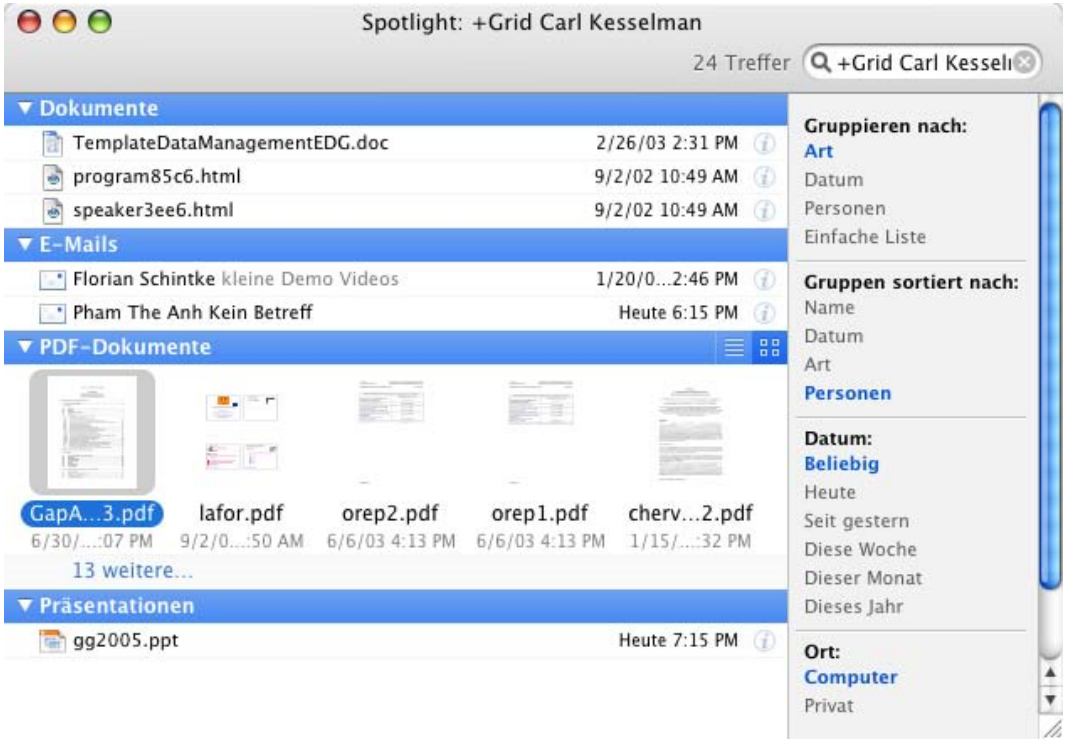

Abb. 3.10 Spotlight-Ergebnisdarstellung mit einer graphischen Oberfläche

3.3.4 Query-Speicherung

Spotlight bietet die Möglichkeit, das Live-Query in Form eines intelligenten Ordners zu speichern, der auf dem ursprünglichen Konzept eines virtuellen Verzeichnisses [GJSO91] basiert. Im Vergleich zum normalen Spotlight-Suchfeld ermöglicht ein intelligenter Ordner dem Benutzer, eine komplexe Abfrage mit logischen Verfeinerungen auszudrücken. Die logische Abfrageformulierung dieser Aufgabe im Spotlight ist relativ selbsterklärend. Alle Attribute, die im [Appl05d] aufgelistet sind, werden übersichtlich in einer graphischen Auswahlliste abgebildet. Ebenso werden Vergleichsoperatoren und logische AND-Verknüpfung in zusammenhängenen Bedingungen bereitgestellt. Leider kann keine Abfrage mit einer OR-Verknüpfung oder einer logischen Gruppierung gebildet werden. Für sie muss man auf die logische Syntax auf der Kommandozeile zurückgreifen. Die Abbildung der logischen Syntax auf der graphischen Oberfläche beschränkt sich auf die Eingrenzung der Abfrage mit dem logischen AND.

Als Beispielsaufgabe wird das in der Abbildung 3.8 beschriebene Problem genommen und analysiert. Ein einzelnes Attribut lässt sich aus der Liste sämtlicher Metadaten schnell finden. Zu jedem Attribut bekommt der Benutzer eine Liste der möglichen Bedingungen im natürlichen Ausdruck zur Auswahl. Die Abbildung 3.11 zeigt eine Abfrage aus zusammengestellten Kriterien, die der logischen Syntax in der Abbildung 3.9 entspricht.

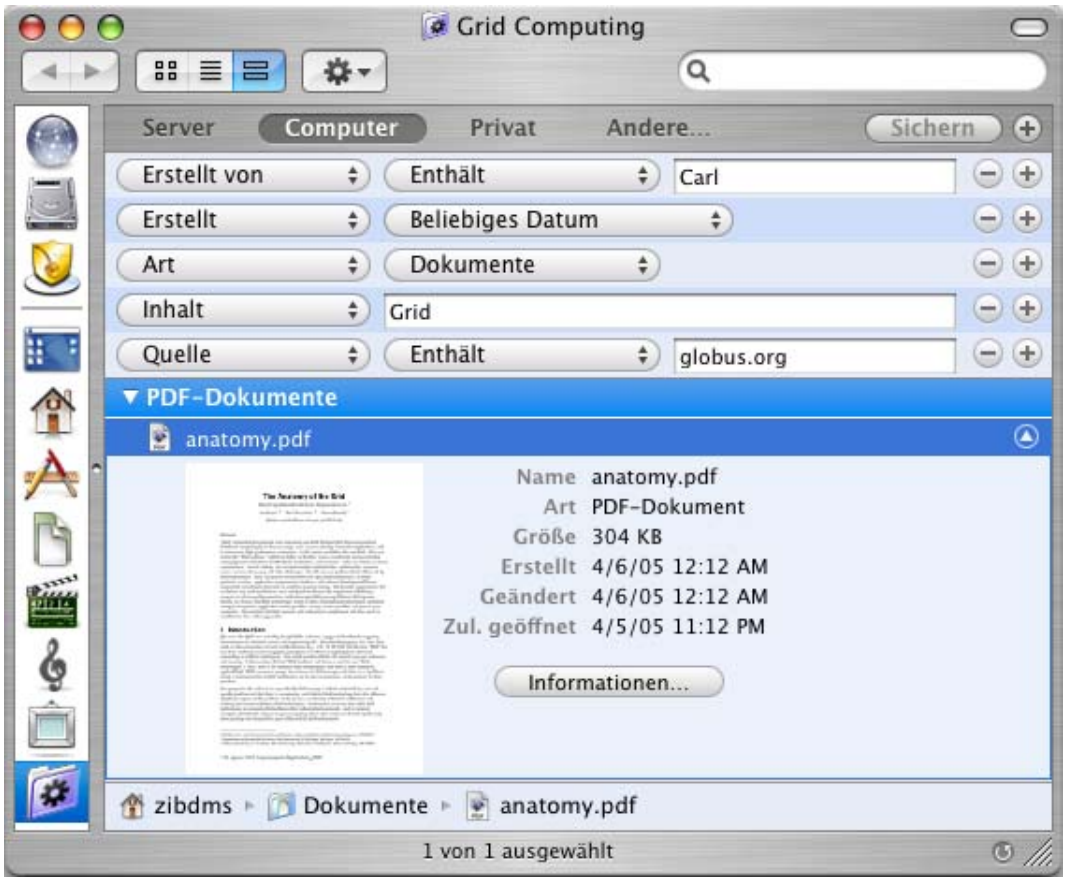

Abb. 3.11 Logische Formulierung und Speicherung eines Querys mit einem intelligenten Ordner im Spotlight

Das Kriterium „`Art`“ bekommt den Wert „`Dokumente`“ zugewiesen. Es ist unklar wie Spotlight diesen Wert vordefiniert. Zum einen hat man auf der Kommandozeile zwei Attribute „`kMDItemContentType`“ und „`kMDItemKind`“, die die Dateiart bezeichnen. Zum anderen ist es nicht bekannt, wie viele Dateiformate zu „`Dokumente`“ gehören. Es wäre hilfreich, eine Möglichkeit gäbe, wie eine Abfrage mit einem intelligenten Ordner auf der Kommandozeile formuliert wird.

Spotlight erlaubt eine komplexe Abfrage mit mehreren Kriterien als intelligenten Ordner abzuspeichern. Ein intelligenter Ordner kann in jedes Verzeichnis des graphischen Verzeichnisbaums hinzugefügt oder verschoben werden und wird somit über den hierarchischen Pfad erreichbar. Das Wechseln in einen intelligenten Ordner löst die in dem Ordner gespeicherte Abfrage auf und man bekommt demzufolge immer aktuelle Ergebnisse. Auf der Kommandozeile wird ein intelligenter Ordner über eine normale Datei mit dem Suffix „`.savedSearch`“ gespeichert. Es handelt sich um ein normales XML-Dokument, in dem die Abfrage im XML-Format gespeichert wird. Der Einblick in die Beispielsdatei „`Grid Computing.savedSearch`“ erklärt die o. g. Frage, mit welchen Ausdrucksmittels der logischen Syntax ein rohes Query (*RawQuery*) im Spotlight intern formuliert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
...
     <key>RawQuery</key>
     <string>(kMDItemAuthors = '*Carl*'cd)
((kMDItemContentTypeTree = 'public.composite-content') ||
(kMDItemContentTypeTree = 'public.audiovisual-content') ||
(kMDItemContentTypeTree = 'public.image'))
(kMDItemTextContent = "Grid*"cd)
(kMDItemWhereFroms = '*globus.org*'cd)
(kMDItemContentType != com.apple.mail.emlx)
(kMDItemContentType != public.vcard)
     </string>
...
```

3.3.5 Bewertung

Mit der neuen Suchtechnologie Spotlight hat Apple ein schlüssiges Konzept vorgestellt: Indexdaten und Metadaten werden automatisch im Hintergrund gesammelt und aktualisiert; es wurden verschiedene Suchmethoden angeboten, die wesentlich ausgefeilter, schneller und vor allem genauer arbeitet; die Integration von intelligenten Ordnern in der Verzeichnishierarchie bietet eine neue Organisationsmöglichkeit, Suchabfragen und Dateien zukünftig nach ihrem Inhalt oder ihrer Eigenschaft zu verwalten.

3.4 Context-File-System

Ein Context-File-System (CFS) [Hess03] wurde im Rahmen des Gaia-Projekts[8] [RHC+02, CBAC05] an der University of Illinois in Urbana-Champaign entwickelt. Gaia ist eine ereignisbasierte Kommunikations-Middleware-Plattform (siehe Abbildung 3.12), um Kontext-Informationen zu verwalten, die für sogenannte Active-Spaces zur Verfügung gestellt werden sollen. Ein Active-Space modelliert einen physischen Raum, in dem sich eine Vielzahl von heterogenen und vernetzten Geräten befinden und miteinander kommunizieren. Der Benutzer kann die existierenden Anwendungen mit einem Application-Framework [Roma03] zu Active-Space-Applications anpassen oder neu konstruieren und danach in unterschiedlichen physischen Räumen problemlos ausführen. Die technischen Daten jedes Gerätes und die darauf laufende Software werden in einem Space-Repository gespeichert und sind über einen Space-Repository-Service abrufbar. Wenn beispielsweise die Auflösung eines PDA-Geräts im Repository bekannt ist, können die zu übertragenden Bilder automatisch verkleinert werden. Das Presence-Service erkennt die Präsez von sowohl Personen als auch von Geräten mit verfügbaren Diensten in einem Active-Space. Über Sensoren wie den Fingerprint-Leser oder Bewegungsmelder wird ein neuer Benutzer authentifiziert und registriert. Über solche Zustandsänderungen kann der Event-Manager-Service informieren. Mit dem Context-Service kann eine Anwendung einzelne Kontextinformation registrieren und abrufen.

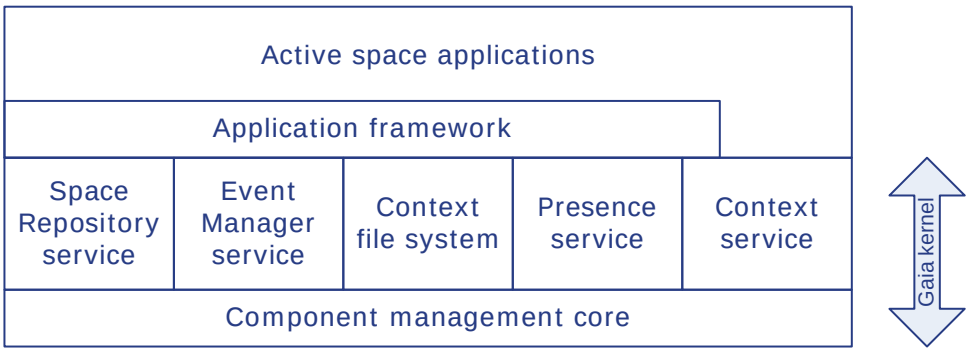

Abb. 3.12 Gaia-Architektur [RHC+02]

8. http://gaia.cs.uiuc.edu

3.4.1 Kontext

Ein atomarer Kontext im Context-Service wird mit einem Quadrupel-Prädikat definiert:

```
Context(<ContextType>, <Subject>, <Relater>, <Object>)
```

- `ContextType` ist der Typ des Kontext, bezieht sich auf das beschriebene Prädikat.
- Unter `Subject` versteht man eine Person, einen physischen Raum, was für den Kontext wichtig ist.
- `Object` ist der mit dem Subjekt assoziierte Wert. Ein Objekt von einem Kontext kann sich als ein Subjekt in einem anderen Kontext figurieren.
- Ein `Relater` assoziiert ein Subjekt und ein Objekt mit einem Verb, einer Präposition oder einer booleschen Operation (wie z. B. =, >, oder <)

Mit dieser Klausel wurde eine Situation von einem Objekt in der Umgebung charakterisiert. Es wird zwischen High- und Low-Level-Kontext unterschieden:

- Low-Level-Kontext bekommt die Information über Sensoren wie `Context(temperature, room 3231, is, 23)` oder `Context(location, chris, entering, room 3231)`.
- Ein High-Level-Kontext kann aus den Klauseln der Low-Level-Kontexte durch ein Regelwerk abgeleitet werden. Erlaubte Logikoperationen sind Quantifikation, Implikation, Konjunktion, Disjunktion und Kontextprädikat-Negation. Zum Beispiel:

  ```
  Context(number of people, room 2401, >, 4) AND
  Context(application, Powerpoint, is, running) => Context(social activity, room 2401, is, presentation)
  ```

Ein Context-File-System wird für diese kontextbasierte Umgebung mit speziellen Aufgaben entworfen:

- Wenn ein Benutzer in einen interaktiven Raum eintritt, wird eine Sitzung dynamisch mit den Active-Space-Ressourcen zugewiesen und dem Benutzer

seine Daten entsprechend dem Kontext automatisch zur Verfügung gestellt.

- Das Dateisystem soll das Auffinden von für die Anwendung und den Benutzer relevanten Daten erleichtern.
- Je nach Vorlieben des Benutzers oder der Gerätespezifikation werden Daten in einem vordefinierten Format über einen dynamischen Typ ausgeliefert.

3.4.2 Kontextverzeichnis

Der transparente Zugriff auf die Dateien innerhalb eines Kontexts erfolgt über ein virtuelles Verzeichnis, das genau einen Kontext im CFS [Hess03] repräsentiert. Dieses Kontextverzeichnis kann man mit einem Pfad wie bei einem klassischen Dateisystem erreichen. Die Abbildung 3.13 versucht, den im [Hess03] beschriebenen Pfad mit der Grammatik in EBNF [ISO01] aufzustellen:

```
path         ::= "/" [path-element] ["/"]
path-element ::= context { "/" context } | name
context      ::= type | type "/" value | "current:" | "user:"
type         ::= char { char } ":" | name
value        ::= char { char }
name         ::= char { char }
```

Abb. 3.13 EBNF-Grammatik für CFS-Path

Aus der formalen Sprache lässt sich erkennen, dass das virtuelle Verzeichnis nicht in die darunterliegende Verzeichnishierarchie integriert ist. Es ist nicht möglich, innerhalb eines normalen Verzeichnisses ein Kontextverzeichnis einzubetten. Wenn man aus einem normalen Verzeichnis in ein Kontextverzeichnis wechseln will, muss man in den Modus eines absoluten kontextbezogenen Pfades wechseln. Diese Trennung zwischen dem Dateimodus und dem Kontextmodus erschwert die Navigation und die Handhabung von Dateien und Verzeichnissen.

Erwähnenswert sind zwei Sonder-Kontextverzeichnisse: „`current:`“ und „`user:`“, die jeweils Daten entsprechend dem Kontext in aktueller Umgebung oder für den aktiven Benutzer zur Verfügung stellen. Diese beiden Verzeichnisse kann

man, ausgenommen das `root`-Verzeichnis, beliebig innerhalb eines virtuellen Verzeichnisses im Kontextmodus ansprechen.

CFS erlaubt dem Benutzer, seine Daten in einem Kontextverzeichnis zu organisieren. Wenn eine Anwendung ein Kontextverzeichnis mit einem `<type:>/<value>` beinhaltenden Pfad erzeugt, kreiert das System einen *Mount-Point* auf einem Auto-Mount-Server, der für jedes Active-Space einen Namensraum verwaltet [Roma03]. Dieser Namensraum ändert sich, wie der Benutzer physisch in einen interaktiven Raum eintritt oder den Raum verlässt.

Der Mount-Point auf dem Mount-Server fungiert als Platzhalter, so dass ein Kontextverzeichnis ohne Inhalt auch aufgelistet werden kann. Zum Beispiel,

```
/location:/BBAW/situation:/conference
```

präsentiert ein virtuelles Verzeichnis in der Form einer Abfrage, die wiederum aus zwei Teilabfragen besteht. Diese Abfrage lässt sich in die logische Syntax

```
location==BBAW && situation==conference
```

transformieren. CFS interpretiert also das „/“-Zeichen als eine UND-Verknüpfung zwischen Pfadkomponenten und unterstützt keine disjunktiven Abfragen. Intern wird ein Mount-Point für dieses Kontextverzeichnis im XML-Format auf dem Mount-Server generiert:

```
<CFS:Storage>
     <CFS:Owner>johnson</CFS:Owner>
     <CFS:Context>
          <CFS:Type>situation</CFS:Type>
          <CFS:Value>conference</CFS:Value>
     </CFS:Context>
     <CFS:Context>
          <CFS:Type>location</CFS:Type>
          <CFS:Value>BBAW</CFS:Value>
     </CFS:Context>
</CFS:Storage>
```

Abb. 3.14 Beispiel für einen generierten Mount-Point, wenn ein Kontextverzeichnis erzeugt wurde

Die Zuweisung von Dateien zu einem Kontext kann mit der `copy`-Operation über eine primitiv verfügbare CFS-Schnittstelle erfolgen. Bei der `copy`-Operation

von Dateien wird zuerst ein normales Verzeichnis auf der lokalen Festplatte temporär angelegt und darin die Verknüpfung zu jeder Datei verlinkt. Danach wird das Kontextverzeichnis mit dem temporären Verzeichnis und der Maschine assoziiert. Die Abbildung 3.15 zeigt eine assoziative Beziehung zwischen dem Kontextverzeichnis aus der Abbildung 3.14 und einem generierten nativen Verzeichnis auf einem Rechner, auf dem sich die Dateien befinden.

```
<CFS:Storage>
     <CFS:Owner>johnson</CFS:Owner>
     <CFS:Host>guest.bbaw.de</CFS:Host>
     <CFS:Path>C:\Temp\4711</CFS:Path>
     <CFS:Context>
          <CFS:Type>situation</CFS:Type>
          <CFS:Value>conference</CFS:Value>
     </CFS:Context>
     <CFS:Context>
          <CFS:Type>location</CFS:Type>
          <CFS:Value>BBAW</CFS:Value>
     </CFS:Context>
</CFS:Storage>
```

Abb. 3.15 Beispiel für einen generierten Mount-Point, wenn Dateien zu einem Kontextverzeichnis assoziiert wurden

Über einen Auto-Mount-Mechanismus kann man das Kontextverzeichnis und die verlinkten Dateien mit dem normalen Kommando erreichen:

```
> cd /location:
> ls
BBAW      CCGrid2002-Program.pdf
> cd /location:/BBAW
> ls
situation:     time:        group:
space:         current:     users:
> cd /location:/BBAW/situation:/conference
> ls
johnson_ieee_grid02.pdf
```

Abb. 3.16 Zugriff auf ein Kontextverzeichnis

Wenn man den Inhalt der Datei `johnson_ieee_grid02.pdf` aus der Auflistung in der Abbildung 3.16 mit einem Programm liest, wird diese Kontextdatei über eine Verknüpfung in dem nativen Verzeichnis `C:\Temp\4711` auf dem Rechner `guest.bbaw.de` (siehe Abbildung 3.15) ausgelesen.

3.4.3 Bewertung

Für einen kleinen interaktiven Raum kann Gaia seine kontextbezogenen Dienste mit einer großen Akzeptanz einsetzen. Es bleibt leider ein spezielles System mit konkreten Aufgaben, denn das vorgestellte Konzept kann man nur bedingt in ein kontextbasiertes System übertragen. Der Erfolg des eingesetzten Sensors, mit dem man den Kontext automatisch festlegt, bringt auch Probleme mit sich. Die Anwendungen müssen einen Kontext für nur eine bestimmte Semantik interpretieren und sich entsprechend verhalten, was in der Praxis nicht immer der Fall ist; denn ein komplexer Kontext könnte in verschiedenen Semantiken für unterschiedliche Einsatzgebiete interpretiert werden.

Das Context-File-System bietet keine Relation zwischen Ereignissen und Daten sowie zwischen den Dateien. Außerdem stellt das Dateisystem keinen Mechanismus für eine meta-datenbasierte Suche zur Verfügung. Der Zugriff auf ein Kontextverzeichnis ist zwar mit einer Modusumschaltung versehen, jedoch wie oben beschrieben nicht flexibel genug. Ein nicht vordefiniertes Kontextverzeichnis muss dann manuell angelegt und wieder aufgeräumt werden. Aus diesen Gründen wird das CFS im Gaia nicht in einem anderen als dem dafür vorgesehenen Active-Space verwendet werden können.

3.5 Überblick weiterer Systeme

Das Semantic-File-System [GJSO91] ist das erste metadatenbasierte Dateisystem, in dem Index- und Metadaten mit einer Abfrage in Form eines virtuellen Verzeichnisses zur Verfügung gestellt werden können. Der assoziative Zugriff auf die Ergebnisse erfolgt über eine standardisierte Schnittstelle wie NFS, um die bestehenden

Tools auf der Benutzerseite ohne weitere Änderung verwenden zu können. SFS benutzt symbolische Links, um die Resultate in einem virtuellen Verzeichnis zu präsentieren. Wenn eine Ursprungsdatei der Ergebnismenge verschoben oder gelöscht wird, wird der symbolische Link im virtuellen Verzeichnis ins Leere zeigen. Diese Inkonsistenz versucht das HAC[9]-Dateisystem [GoMa99] zu beheben, indem das Query in regelmäßigen Zeitabständen ausgeführt wird, um die symbolischen Links im virtuellen Verzeichnis zu aktualisieren. HAC bietet eine Kombination zwischen dem hierarchischen Zugriff und einem inhaltsbasierten Zugriff mit Hilfe eines semantischen Verzeichnisses. Man muss leider dieses virtuelle Verzeichnis mit einem speziellen Kommando zuerst erzeugen, um das Verzeichnis in die Hierarchie einzubetten. Zudem unterstützt HAC keine Metadaten und lässt den Benutzer keine Attribute zu einer Datei zuweisen. Das Synopsis-File-System (SynFS) [BoJo96] ist ein attributbasiertes Dateisystem [SoGa03], in dem Metadaten zu einer Datei durch den Benutzer oder automatisch erfasst und bei jeder Änderung synchronisiert werden. Diese Attributmenge ist eine Zusammenfassung der Datei (*Synopsis*) und mehrere Dateien können in einer Gruppe (*Digest*) organisiert werden. Dies ist die Grundlage für eine attributbasierte Sicht (*View*), die mit Hilfe von Queries eine Hierarchie wie bei Storagebox [Hupf03, Hupf04] bildet, wo jedes Unterverzeichnis eine Verfeinerung der Abfrage zu dem übergeordneten Verzeichnis ist.

Presto [DELS99] ist ein Dokumentmanagementsystem und verfolgt das Ziel, dem Benutzer seine Dokumente unabhängig von einem Ort (*Placeless Documents*) zu verwalten. Für die Organisation der Dokumente werden benutzerdefinierte Attribute zu Dokumenten hinzugefügt und Dokumente können wie bei SynFS in einer Collection gruppiert werden. Da jedes Attribut des Dokument eine eigene Semantik hat, können Dokumente damit schnell sortiert und reorganisiert werden. Presto bietet eine graphische Oberfläche Vista [DELS99], die dem Benutzer eine direkte Interaktion mit seinen Dokumenten über Collections oder Attributabfragen ermöglicht.

Zu den eng gekoppelten Systemen des integrierten Ansatzes (siehe Seite 10) gehören Windows Vista (Codename Longhorn) mit WinFS und BeOS[10] mit BeFS[11]

9. **H**ierarchy **a**nd **C**ontent
10. Das eingestellte Projekt hat viele Nachfolge, u. a. Zeta, http://www.yellowtab.de

[GiBe99]. WinFS [Rect04, Grim04] basiert auf einem SQL-Datenbanksystem und soll das traditionelle Dateisystem unter Windows ablösen. Mit der Indizierung erlaubt WinFS die Suche nach Dateien mit beliebigen Inhalten oder Metadaten und bietet gleichzeitig eine neue Dateiverwaltungsmethode mit virtuellen Ordnern. Das Be-File-System ist ein 64-Bit Journaling-Dateisystem, dessen Geschwindigkeit unabhängig von der Größe der verwalteten Daten sei. Attribute im BeFS werden assoziativ mit der Datei in einem Attributverzeichnis gespeichert. BeFS benutzt einen B+-Baum [GiBe99], um den Inhalt eines Verzeichnisses und Indexdaten zu speichern; diese Daten können mit einem Live-Query abgefragt werden. Zusätzlich zu Hardlinks und Symlinks bietet BeFS dem Benutzer weitere auf Symlinks basierende Dynamic-Links mit einer besonderen Eigenschaft, dass die gespeicherte Zeichenkette (`linkinfo`) in einem Symlink dynamisch wie eine Variable interpretiert werden können.

PlanetP [Cuen04, CPMN03] ist ein Information Retrievalsystem in einem P2P-Netzwerk [MKL+02, AnSp04]. Dateien werden indiziert und die gewonnene Indexdaten in Form einer invertierten Liste werden mit dem Vektorraum-Modell [BaRi99, FrBa92, SaMc83, WiMB99] gewichtet, so dass eine Suche mit Ranking nach relevanten Inhalten ermöglicht wird. Jeder Peer verfügt über lokale und globale Indexdaten. Globale Indexdaten werden in einem Verzeichnis eingetragen, das auf allen Peers mit Hilfe des Gossip-Algorithmus [CPMN03] repliziert wird. Eine Abfrage auf einem Peer wird zuerst an dieses globale Verzeichnis auf dem selben Peer geleitet. Das Ergebnis ist eine nach Ranking sortierte Liste der Peers, die vermutlich relevanten Ergebnisse verfügen. Diese Peers bearbeiten die Abfrage lokal und liefern das Ergebnis zurück; die Abfrage wird von diesen Peers nicht weitergeleitet. Obwohl dieses Verfahren bei PlanetP das Auffinden aller relevanten Dokumente nicht garantieren kann, bleibt PlanetP ein interessanter Ansatz für diese Arbeit.

Semantic Web[12] ist eine Erweiterung des World Wide Webs und fußt im Prinzip auf RDF (Resource Description Framework). RDF ist eine Beschreibungssprache für Metadaten, die auf dem Tripel <Subject, Predicate[13], Object[14]> basiert. Mit

11. **Be F**ile **S**ystem
12. http://www.w3.org/2001/sw/

diesem Tupel wird die Metadatenbeschreibung in Form von Attribut-Wert-Paaren erweitert, weil das Objekt in dem Tripel mit anderen Metadaten in Beziehung gesetzt werden kann. Semantic Web dient dazu, semantische Informationen über Webinhalte zu beschreiben, wobei XML für den Datenaustausch und die Darstellung verwendet wird. Viele mediatorbasierte Systeme nutzen das Konzept des Semantic Webs, um Daten aus mehreren Quellen mit den jeweiligen Wrappern in eine einheitliche Ergebnisdarstellung zusammenzustellen [Buss02].

13. oder Property
14. oder Value

4 Hierarchische Dateiverwaltungsmethoden im ZIBDMS

In dem vorherigen Kapitel wurden verschiedene für diese Arbeit bedeutende Systeme beschrieben, in die neue Methoden zur Dateiverwaltung eingebettet werden. In diesem Kapitel wird das Datenmanagementsystem ZIBDMS vorgestellt, dem ein neues Dateikonzept zu Grunde liegt. Mit diesem Konzept ist es möglich, weitere Dateiobjekte wie Weak-Link, Collection, Dependency-Graph oder sogar virtuelle Dateien und virtuelle Verzeichnisse eingebettet in die hierarchische Sicht des ZIBDMS einzuführen. Auf solche Dateien bietet ZIBDMS neben dem hierarchischen Zugriff noch einen assoziativen Zugriff mit einer leicht erlernbaren, sprachunabhängigen Abfragesprache in Form eines virtuellen Verzeichnisses, so dass die Dateiverwaltung intuitiv mit den bewährten, unmodifizierten Programmen weiterhin erfolgt.

4.1 Design des ZIBDMS

Das ZIBDMS verfolgt das Ziel, heterogene verteilte Daten in einem Datenmanagementsystem zu verwalten. Mit der Annahme, dass diese Daten sehr häufig gelesen und selten geschrieben werden [Schi04b], wurde ZIBDMS nach der Index-Shipping-Architektur (siehe Abschnitt 2.2) entworfen. Der Entwurf von ZIBDMS berücksichtigt folgende Aspekte, die in weiten Teilen auch bereits realisiert sind:

- ***Metadatenbasierend:*** Metadaten zu einem Objekt im ZIBDMS werden in Form von Attribut-Wert-Paaren repräsentiert, wobei ein Attribut ein oder mehrere Werte besitzt und jeder Wert eine 64-Bit Ganzzahl oder eine Zeichenkette beinhalten kann.
- ***Inhärente Verteilung:*** Metadaten werden in einem verteilten Katalog gespeichert, um Ausfälle wie Single-Point-of-Failure zu verhindern. Die Verteilung erfolgt in einem P2P-Netzwerk, so dass Metadaten in jedem Knoten verwaltet werden können.
- ***Datenverfügbarkeit:*** In eine Datei im ZIBDMS können verschiedene heterogene Datenquellen eingetragen werden. Je mehr Replikate eingetragen werden, desto höher ist die Verfügbarkeit der Daten innerhalb des ZIBDMS [ScRe03]. Allerdings hängt die Verfügbarkeit des einzelnen Replikats weiterhin von der Quelle ab, auf der das Replikat physisch gespeichert wird. Im ZIBDMS sind die Ausfälle der Quellen nicht bemerkbar (*fehlertransparent*), solange mindestens eine Kopie der Datei erreichbar ist. Ebenso erfährt der Benutzer nicht, von welchen und wie vielen Replikaten (*replikationstransparent*) er bedient wird.
- ***Transfergeschwindigkeit:*** Wenn eine Datei im ZIBDMS gelesen wird, wird die Datei aus den eingetragenen Datenquellen geholt. Es wird dabei von mehreren erreichbaren Kopien gelesen, so dass der Zugriff mit der maximal möglichen Transfergeschwindigkeit zwischen der Benutzerseite und allen Quellen erfolgt.
- ***Ortstransparenz:*** Durch die Verteilung der Metadatenkataloge und die Eintragung heterogener Datenquellen kann eine Datei über einen von dem physischen Speicherort unabhängigen Namen (*namenstransparent*) angesprochen werden, auch wenn der Speicherort danach geändert wird.
- ***Flexibler Zugriff:*** Der Zugriff auf eine Datei ist insofern flexibel, weil man die Datei über den üblichen hierarchischen Pfad oder über einen assoziativen Pfad mit Hilfe von virtuellen Objekten erreichen kann.

- ***Sprachunabhängige Abfrage:*** Der assoziative Zugriff erfolgt über eine im Abschnitt 2.2.3.4 vorgestellte logische Syntax, die leicht erlernbar und sprachunabhängig ist. Mit dieser intuitiven Notation ist es möglich, den Informationsbedarf vom Benutzer präzise zu formulieren und die Ergebnismenge einzugrenzen oder zu erweitern.
- ***Portierbarkeit und Interoperabilität:*** Das ZIBDMS stellt eine Vielfalt von Benutzerschnittstellen zur Verfügung, mit denen der Benutzer die Funktionalitäten des ZIBDMS aus unterschiedlichen Betriebssystemen nutzen kann. Die Portierbarkeit des ZIBDMS zeichnet sich durch die für das ZIBDMS wichtigen NFS-Schnittstelle aus, über die sich die Nutzung des entfernten ZIBDMS automatisieren lässt und auf Dateiobjekte mit unmodifizierten Programmen in gleicher Weise wie im Lokal (*zugriffstransparent*) zugegriffen werden kann.

In diesem Abschnitt werden von ZIBDMS-Entwicklern implementierte Konzepte und Komponente vorgestellt, die der vorliegenden Arbeit zugrunde liegen und für ZIBDMS relevant sind.

4.1.1 Dateikonzept

Das ZIBDMS-Dateikonzept [Schi04a, Schi04b] verfolgt die Unix-Philosophie: „Alles ist eine Datei, und wenn nicht, sollte es eine Datei sein“. Jedes Objekt im UNIX wird also als Datei durch eine Datenstruktur, I-Node (*Index-Node*) genannt [Tann02, Kofl04], repräsentiert. In einem I-Node werden Dateiattribute und Addressen der Dateidaten, jedoch nicht der Dateiname, gespeichert; der Name wird in den Verzeichniseinträgen notiert und wird mit dem Datei-I-Node über eine eindeutige I-Node-Nummer verknüpft. Zu jedem Element dieses Konzepts gibt es ein Pendant im Dateikonzept des ZIBDMS, so dass jede Information im ZIBDMS als Datei verfügbar wird. Ein Dateiobjekt im ZIBDMS besteht aus den drei folgenden Teilobjekten, die jeweils ein Attribut UFI besitzen:

- ***Hierarchical File Name (HFN):*** Anders als beim Unix-Dateikonzept wird die Information über das übergeordnete Verzeichnis mit dem Dateinamen gespeichert. Der absolute Pfad setzt sich aus diesen beiden Einträgen zusammen. Diese beide Attribute könnte man im UFI-Objekt speichern, wenn man auf Hardlinks im ZIBDMS verzichten würde.
- ***Unique File Identifier (UFI):*** Im UFI-Objekt werden Systemattribute und Userattribute gespeichert. Systemattribute sind notwendige Metadaten zu einer Datei, die das System benötigt, und Userattribute sind frei definierbare Metadaten, die vom Benutzer angelegt und verwaltet werden können.
- ***Native Storage Location (NSL):*** Das NSL-Objekt beinhaltet Adressen aller Replikate. Bei Operationen auf Dateidaten wie Lesen oder Schreiben werden Dateidaten von diesen Datenquellen zur Verfügung gestellt. Jede URL[1] enthält eine übliche Adresse mit einem Zugangsprotokoll (Übertragungsprotokoll) oder eine einfache Zeichenkette, die vom ZIBDMS intepretiert wird.

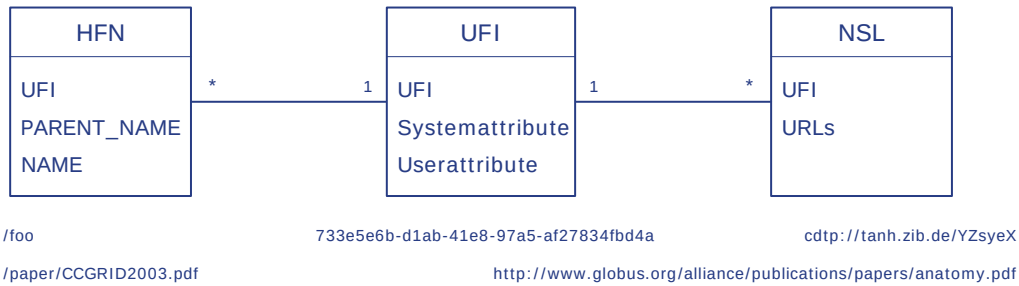

Abb. 4.1 Modellierung einer Datei im ZIBDMS [Schi04a]

Die Abbildung 4.1 zeigt die Verbindung zwischen drei Teilobjekten einer Datei, die miteinander über ein UFI verknüpft sind. Ein UFI ist eine global eindeutige ID, die zur Zeit mit einem 128-Bit generierten Wert garantiert wird. Das UFI dient als Schlüssel, um von einem Teilobjekt aus andere Teile finden zu können.

1. **U**niform **R**esource **L**ocator

4.1.2 Architektur

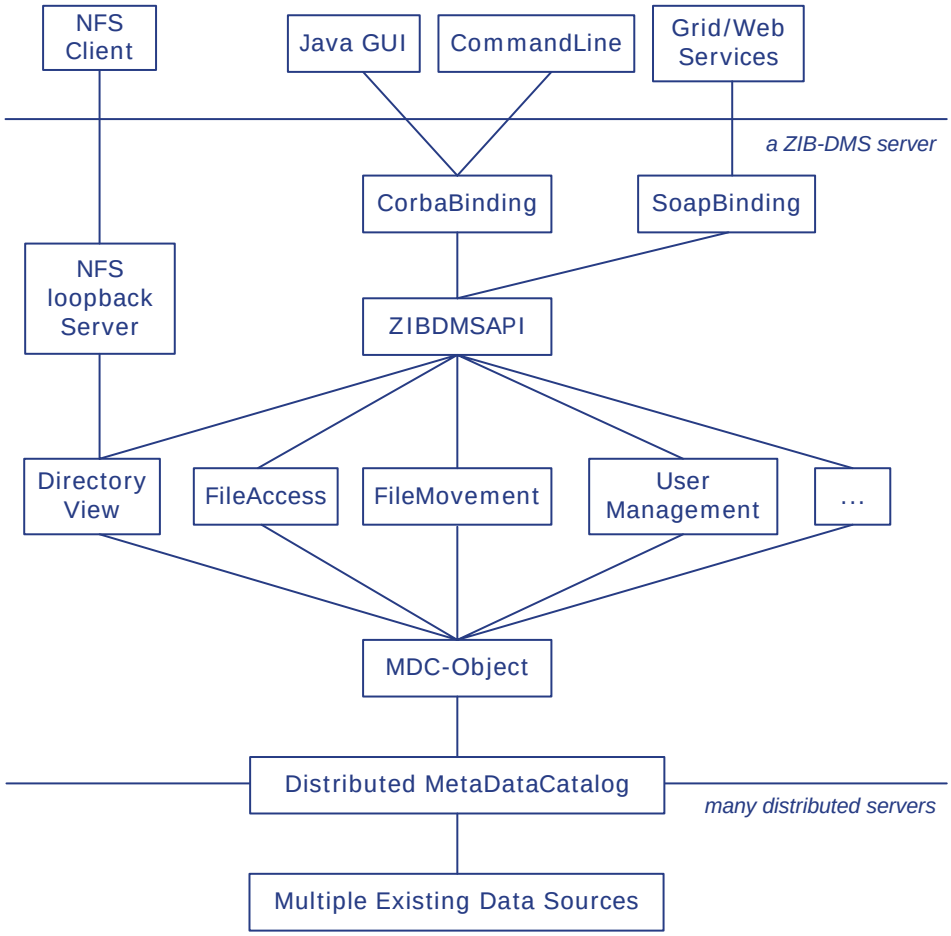

Abb. 4.2 ZIBDMS Systemarchitektur, angelehnt an [Schi04a]

Die in der Abbildung 4.2 dargestellte Architektur des ZIBDMS [Schi04a, Schi04b] implementiert die Index-Shipping-Architektur des erweiteren Ansatzes (siehe Abschnitt 2.2) und umfasst folgende wesentliche Komponenten:

- *User-Interfaces:* ZIBDMS besitzt einen NFS-loopback-Server [Witt05], um einen unmodifizierten NFS-Client zu unterstützen. CorbaBinding und SoapBinding stellen die Funktionalitäten des ZIBDMSAPIs für Java-GUI (siehe Abschnitt 4.1.5), Command-Line und Grid/Web Services zur Verfügung. Command-Line umfasst viele kleine Programme, die über Kommandozeile

ausführbar sind und ergänzen weitere Semantiken, die mit normalen Programmen über die NFS-Schnittstelle nicht abgedeckt sind.

- *ZIBDMSAPI:* Über diese ZIBDMS-APIs wird eine konsistente Programmierschnittstelle des gesamten ZIBDMS zur Verfügung gestellt.
- *Directory-View:* Hier werden elementare Operationen implementiert; sie werden in dieser Arbeit im Abschnitt 4.2 ausführlich beschrieben.
- *File-Access*, *File-Movement:* Mit diesen beiden Komponenten ist es möglich, Dateidaten aus verschiedenen Datenquellen zu lesen, Replikate zwischen verschiedenen Speicherorten zu bewegen und lokale Kopien zu schreiben. Für den Zugriff auf lokale Kopien wird ein eigenes Zugangsprotokoll „`cdtp://`“ implementiert.
- *User-Management:* Hier werden die Benutzerkennungen und dazugehörige Zugriffsrechte verwaltet.
- *MDC-Object:* Metadaten zu einer Datei in Form von Attribut-Wert-Paaren werden über diese Komponente als Objekte (siehe Abschnitt 4.1.3) zur Verfügung gestellt.
- *Distributed-MetaDataCatalog:* In dem `mdc`-Katalog werden Metadaten in einer Datenbank gespeichert und über mehrere Knoten (siehe Abschnitt 4.1.3) verteilt.

4.1.3 Verteilter Metadatenkatalog

Metadaten im ZIBDMS werden in dem Metadatenkatalog `mdc` gespeichert. Der Katalog organisiert Attribut-Wert-Paare in einer Backend-Datenbank wie `storagebox`[2] [Hupf03], `sqlite`[3] oder `mysql`[4] und kann mit einem Schema über mehrere Rechner verteilt werden. Die Verteilung des Katalogs sollte in einem P2P-Netzwerk [MKL+02, AnSp04] erfolgen, in dem die Komponenten selbständig oder lose

2. http://www.storagebox.org
3. http://www.sqlite.org
4. http://www.mysql.com

gekoppelt [ScSR03] sind. Um ein dynamisches Routing innerhalb des P2P-Netzes effektiv zu erreichen, entwickeln die ZIBDMS-Entwickler zur Zeit einen Chord#-Algorithmus, der auf dem Chord-Algorithmus [AnSp04] basiert.

In dem Katalog werden Metadaten in Form von Attribut-Wert-Paaren gespeichert und können von Directory-View aus theoretisch direkt abgefragt werden. Jedoch werden sie in der Implementierung über eine Zwischenschicht `MDC-Object` bereitgestellt. Die MDC-Object-Ebene reflektiert das im Abschnitt 4.1.1 beschriebene Dateikonzept und bietet eine komfortable Dateiverwaltung (siehe Abbildung 4.3). Im ZIBDMS gibt es weitere globale Objekte wie Collection (siehe Abschnitt 4.2.5) und Dependency-Graph (siehe Abschnitt 4.2.6), die zu keiner Hierarchie gehören und deshalb keinen Pfad besitzen; sie werden vom UFI-Objekt vererbt und infolgedessen als Datei verfügbar.

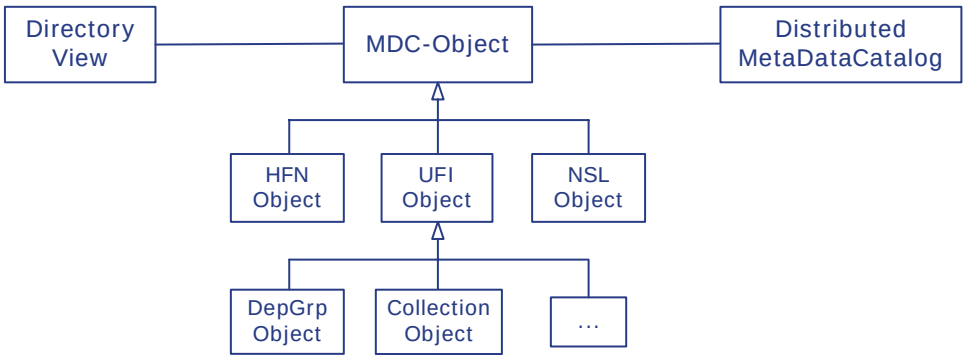

Abb. 4.3 Interaktion zwischen Directory-View, MDC-Object und Metadatenkatalog

Mit der MDC-Object-Ebene werden Metadaten aus dem Katalog geholt und in entsprechende Objekten zusammengefasst, so dass Directory-View sie als Dateien betrachten und darauf operieren kann.

4.1.4 NFS Server

Der NFS Server im ZIBDMS implementiert das ursprünglich von der Firma SUN entwickelte NFS Protokoll Version 3 [SUN95]. Es erlaubt einen transparenten Zugriff auf Dateien und Verzeichnisse über ein virtuelles Dateisystem und eine

verteilte Verarbeitung auf vernetzten Rechnern. Dies ist für ZIBDMS eine wünschenswerte Eigenschaft. Das NFS-Protokoll ist zustandslos, das heißt jede Operation ist in sich abgeschlossen. Zur Übertragung nutzt der implementierte NFS Server nur das UDP-Protokoll [Witt05], ein verbindungsloses Protokoll, das keinen Erfolg der Übertragung garantiert. Die Implementation des NFS-Protokolls dient als eine Schnittstelle zum ZIBDMS, die die NFS-Semantik in die ZIBDMS-Semantik überträgt. Demzufolge werden NFS-Objekte in Form von regulären Dateien, Verzeichnissen, Hardlinks und symbolischen Links auf ZIBDMS-Objekten abgebildet.

4.1.4.1 File-Handle

Der zustandslose NFS-Server speichert keine Information über den Zustand des Clients, und umgekehrt. Es ist also möglich, bei einem Serverabsturz und anschließendem Neustart des Servers hinterher weiter zu arbeiten. Für diese Persistenzeigenschaft wird eine Datenstruktur, sogenannter File-Handle, auch magic-cookie genannt, bei einer Lookup-Anforderung an den Server definiert, der die einzige Referenz auf die jeweilige Datei bei allen zukünftigen Zugriffsanforderungen durch den Client darstellt. Der File Handle ist für jede Datei eindeutig und solange gültig, bis das Dateisystem des Servers neu initialisiert wird. Es gibt keine Möglichkeit, ein File-Handle für ungültig zu erklären. Ein File Handle im internen NFS Server wurde aus dem UFI und eine aus dem HFN errechnete Hash-Summe zusammengesetzt.

„`Stale NFS file handle`“-Meldung wird nur an den Client geschickt, wenn ein laufender Prozess weiterhin versucht, auf eine Datei zuzugreifen, nachdem die Datei nicht mehr existiert. Das ist ein üblicher Fehler auf NFS-Clients nach einem Server-Absturz oder einem Neustart des Servers, bevor die NFS-Dateisysteme von den Clients ausgehängt wurden. Die Fehlermeldung lässt sich beheben, indem man alle Dateien schließt und anschließend das Dateisystem aus- und wieder einhängt.

4.1.4.2 Pfadnamen

Im NFS-Server werden Dateien und Verzeichnisse über ihren Pfad adressiert. Dies ist ein normaler UNIX-Pfadname, der in eine kontextfreie Grammatik durch die erweiterte Bakkus Naur Form (EBNF) [ISO01] spezifiziert werden kann:

```
path            ::= [root] [relative-path]
root            ::= [root-name] [root-directory]
root-directory  ::= "/"
relative-path   ::= path-element { "/" path-element } ["/"]
path-element    ::= name | parent-directory |
                    directory-placeholder
name            ::= char { char }
directory-placeholder::= "."
parent-directory     ::= ".."
```

Abb. 4.4 EBNF-Grammatik für NFS-Pfad

Absoluter Pfad

Ein absoluter Pfad beschreibt den Weg zu einer Datei oder einem Verzeichnis immer ausgehend vom Wurzelverzeichnis. Das Wurzelverzeichnis ist gleichzeitig der Mount-point zum NFS-Server und beginnt immer mit einem Schrägstrich „/“.

Relativer Pfad

Ein relativer Pfad beschreibt den Weg zu einer Datei oder einem Verzeichnis, ausgehend vom aktuellen Verzeichnis des Benutzers im Dateisystem. Das aktuelle Verzeichnis ist immer jenes, dessen Inhalt man gerade bearbeitet. Das aktuelle Verzeichnis lässt sich jederzeit wechseln und der NFS-Server bekommt keinerlei Informationen darüber. Deshalb bekommt der NFS-Server immer den absoluten Pfad vom Client und leitet ihn an weitere zuständige Komponenten zum Bearbeiten weiter. Auf dieser Grundlage wurde der virtuelle Pfadname konzipiert.

Eine besondere Erwähnung verdienen die beiden Verzeichnis-Einträge „.“ und „..“. Das sind Stellvertreter für Verzeichnisnamen, die man statt der realen

Namen abkürzend benutzen kann. Und zwar bezeichnet „.“ das jeweils aktuelle Verzeichnis, und „..“ das dem aktuellen Verzeichnis übergeordnete Verzeichnis. Obwohl man die beiden Einträge mit normalen Kommandos wie ein Verzeichnis ansprechen kann, werden sie bei keiner Auflistung des Verzeichnisses in der Ergebnismenge zurückgegeben. Diese Besonderheiten hat das NFS-Protokoll auch nicht vorgeschrieben.

Virtueller Pfadname

Im Gegensatz zu den beiden o. g. Pfadnamen verweist ein virtueller Pfadname weder auf eine existierende Datei noch auf ein Verzeichnis. Er wurde dafür konzipiert, um auf Dateien und Verzeichnisse auf andere Art und Weise zuzugreifen oder um eine bestimmte Semantik auszudrücken. Ein virtueller Pfad im ZIBDMS hat folgende Merkmale:

- Er wird als Zeiger benutzt, um virtuelle Dateien oder virtuelle Verzeichnisse zu adressieren. Ein virtueller Pfad kann vom System generiert oder vom Benutzer explizit angegeben werden.
- Er kann eine festgelegte Syntax beinhalten, um eine Abfrage an das ZIBDMS zu formulieren oder um eine gewünschte Operation bzw. einen Parameter auszudrücken.

Für den NFS Server hat der Pfad eine besondere Bedeutung. Er ist das einzige Element, das man von dem Client über einen RPC-Aufruf immer übergeben bekommt. Deswegen wurden semantische Informationen in dem Pfadnamen untergebracht, was einen virtuellen Pfad als ein mächtiges Mittel der Semantik ausprägt.

Syntaxzeichen ausschließen[5]

Wenn die Interpretation von Leerzeichen oder sonstigen Sonderzeichen (wie z. B. *, +, <, >, &, |, \ usw.) durch die Shell verhindert werden soll, kann man ein solches Syntaxzeichen durch einen vorangestellten Backslash „\“ quoten. Um mehrere Sonderzeichen vor der Interpretation zu schützen, wird der Pfadname mit zwei Anführungszeichen am Anfang und Ende eingeschlossen.

5. engl.: quoting

4.1.5 Graphische Oberfläche

Außer der im letzten Abschnitt beschriebenen NFS-Schnittstelle kann der Benutzer die Funktionalitäten des ZIBDMS über eine Vielfalt von Benutzerschnittstellen (*User-Interfaces*) nutzen. Für die Visualisierung des Systems stellt ZIBDMS eine graphische Oberfläche (ZIBDMS-GUI) zur Verfügung, die in Java [Krüg04] programmiert wird, so dass die Nutzung von ZIBDMS plattformunabhängig ermöglicht wird. Das ZIBDMS-GUI ist zwar nicht so mächtig und automatisierbar wie bei einer Kommandozeile über die NFS-Schnittstelle, bietet aber eine unersetzbare Darstellung komplexer Informationsstruktur. Die Funktionalitäten beider Schnittstellen ergänzen sich hervorragend und bieten eine optimale Nutzung des ZIBDMS.

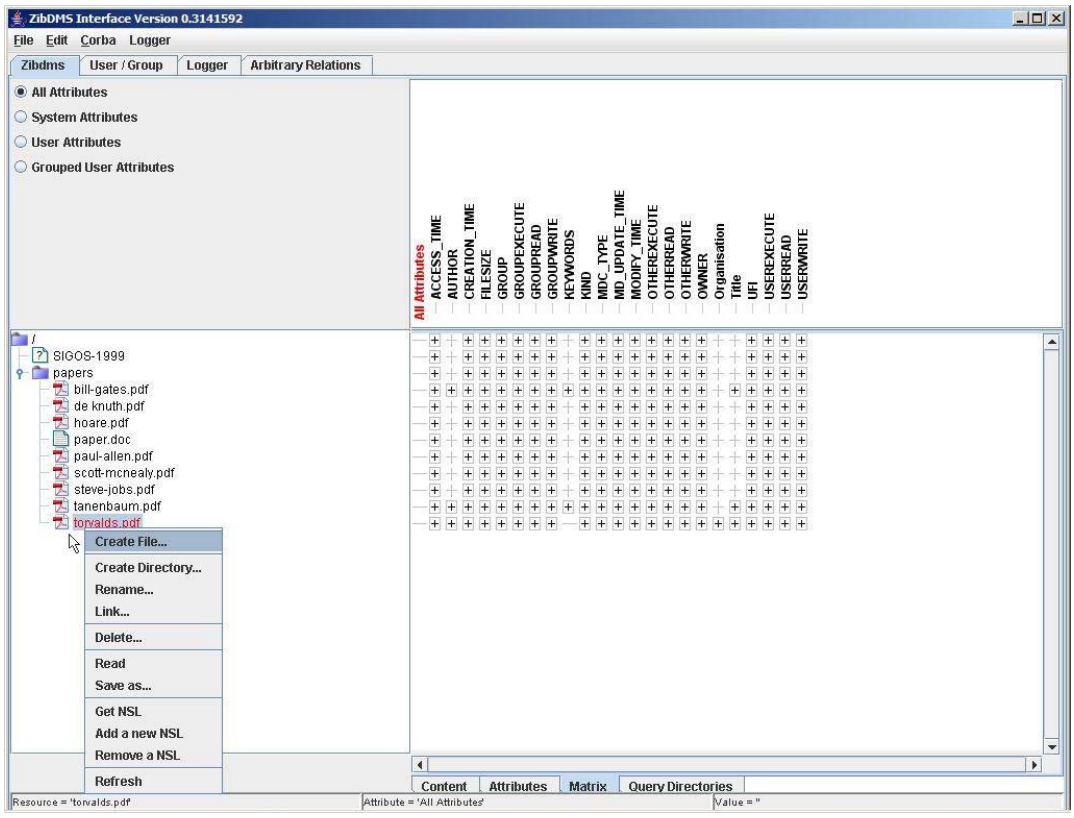

Abb. 4.5 Graphische Darstellung des Verzeichnisbaums und der dazugehörigen Attribute im ZIBDMS-GUI

Das ZIBDMS-GUI interagiert mit Directory-View über das ZIBDMSAPI und benutzt dieselben Methoden wie der interne NFS-Server. Die Abbildung 4.5 präsentiert eine Übersicht aller Metadaten einer Datei im Verzeichnisbaum mit Hilfe eines Matrix Browsers [ZiKB02] und das Kontextmenü zu der Datei. Auf der vertikalen Achse wird die hierarchische Sicht des Verzeichnisbaums durch die bekannte Baummetapher dargestellt. An der horizontalen Achse werden Attribute in eine Liste sortiert angeordnet. Aus der Zelle dieser Adjazenzmatrix kann der Attributwert abgelesen und interaktiv geändert werden.

Abgesehen von einer übersichtlichen Abbildung des Dependency-Graphs (siehe Abschnitt 4.2.6) wird in dieser Arbeit auf weitere Beschreibungen des ZIBDMS-GUIs verzichtet. Es wird sich im Folgenden auf Details von `directory-view` konzentriert, die auch für das ZIBDMS-GUI relevant sind.

4.2 directory-view

Das `directory-view` ist eine der zentralen Komponenten im ZIBDMS und wird in mehreren Ebenen organisiert. Jede neue Semantik wird in einer Unterkomponente (siehe Abbildung 4.6) untergebracht, die durch `directory-view` gesteuert wird. Durch diese Unterteilung ist es möglich, Ergebnisse von einzelnen Unterkomponenten zu kombinieren und ein Endergebnis für andere ZIBDMS-Komponenten bereitzustellen.

Im Directory-View werden atomare Operationen implementiert, die als Grundlage für andere Schnittstellen dienen. Jede atomare Operation wird in einer Methode in der C++-Programmiersprache [Josu99, Stro00] implementiert. Auf jeder separaten Ebene befindet sich die eigentliche Implementierung dieser Methode, die eine bestimmte Semantik erfüllt. Beim jeden Aufruf im `directory-view` wird dieselbe Methode in einer oder mehreren Unterkomponenten je nach Semantik unter einer bestimmten Koordination aufgerufen und die Ergebnisse werden zusammengestellt. Die Anwendung setzt mehrere dieser Semantiken zusammen und stellt damit eine Programmlogik her.

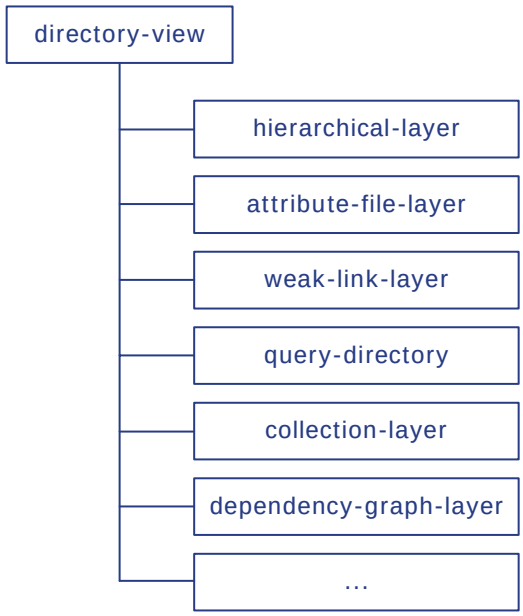

Abb. 4.6 Directory-Views Komponente

4.2.1 hierarchical-layer

Das `hierarchical-layer` ist zuständig für alle nativen Dateien und Verzeichnisse, die im `mdc`-Katalog gespeichert werden. Diese Ebene wurde im Rahmen der Implementierung der NFS-Schnittstelle [Witt05] implementiert. Deswegen wird in diesem Abschnitt nur die `write()`-Funktion beschrieben, die in [Witt05] noch offen war.

Schreiben einer Datei im ZIBDMS

Das Design des ZIBDMS basiert auf dem Ansatz, dass Dateien aus heterogenen Datenquellen häufig gelesen und selten geschrieben werden. Mit dem Entwurf des Dateikonzepts ist ein Schreibvorgang in einer Datei im ZIBDMS zwar möglich aber verläuft aus folgenden Gründen nicht so einfach wie bei einem klassischen Dateisystem:

- *Keine Originale:* Die Dateidaten befinden sich, wie schon mehrfach erwähnt, nicht innerhalb des ZIBDMS. Es werden nur Adressen der Replikate im ZIBDMS eingetragen.

- *Schreibverbot:* Bei einer Adresse eines Replikats handelt es sich oft um eine URL mit einem Zugangsprotokoll. Die meisten aller weit verbreiteten Protokolle erlauben keinen oder nur einen eingeschränkten Schreibzugriff. Eine Schreibberechtigung beim entfernten Server und ein das Schreiben erlaubendes Protokoll sind die Voraussetzungen für das Schreiben in einer Datei im ZIBDMS.
- *Konsistenz und Aktualität:* Hier werden mehrere Fragen gestellt. Wenn eine Datei mehrere Replikate hat, hat man Schreibzugriffe auf alle Replikate. Wenn ja, welche Replikate werden geschrieben, und wenn nein, werden sie überhaupt geschrieben? Wenn während des Schreibens ein Replikat nicht erreichbar ist oder ein Fehler auftritt, wie wird dieses Replikat danach weiter geschrieben, synchronisiert oder für ungültig (Invalidierung) erklärt? Diese Probleme führen auf die Algorithmen für das Schreiben von Replikaten in verteilten Systemen [CoDK02] zurück, in denen der Schreibvorgang wie eine Transaktion behandelt wird. Solche Algorithmen wie Write-All-Available-Algorithmus[6], Primary-Copy-Algorithmus[7] oder Quorum-Algorithmus[8] könnten als Basis für einen neu entwickelten Algorithmus für die o. g. Probleme im ZIBDMS dienen.

In dieser Arbeit wird das Schreiben in eine Datei vereinfacht implementiert. Es geht hier darum, die Semantik des Schreibens auf ein Replikat ins ZIBDMS, inbesondere über die NFS-Schnittstelle, einzuführen. Eine Datei wird also nur geschrieben, wenn sie keine oder nur eine NSL hat, die das von ZIBDMS-Entwicklern spezifizierte `cdtp`-Zugangsprotokoll beinhaltet. Mit diesem Protokoll ist es möglich, eine logische Datei im ZIBDMS mit einer Datei auf dem lokalen Datenträger zu assoziieren und somit in den Inhalt zu schreiben. Mit Hilfe von der FileAccess-Komponente wird der Zugang zu der lokalen Datei zuerst geschaffen, entsprechend in die richtige Stelle positioniert, dann werden die übermittelten Daten auf den Datenträger geschrieben. Zum Schluss wird die Datei geschlossen. Diese einfache

6. deut.: Schreiben aller verfügbaren Replikaten.
7. Die Primärkopie wird zuerst geschrieben, das Schreiben anderer Kopien folgt erst nach der Bestätigung der Transaktion.
8. Eine qualifizierte Mehrheit der Replikaten werden geschrieben.

Implementierung ermöglicht dem Benutzer, eine logische Datei im ZIBDMS mit den unmodifizierten Kommandos zu kopieren oder zu editieren:

```
> cp foo bar
> nano bar
```

Das Schreiben mit dem `cdtp`-Protokoll erfordert die Verwaltung von lokalen Dateien durch FileAccess oder manuell durch den Benutzer. Eine Alternative könnte eine Implementierung der WebNFS[9]-Client-Spezifikation sein, so dass der Schreibzugriff auf einen entfernten NFS-Server mit einer normalen NSL ohne Mounten erfolgt.

4.2.2 attribute-file-layer

Das Prozessdateisystem unter UNIX/LINUX ist ein Pseudo-Dateisystem, weil ihm kein physikalisches Dateisystem zugrunde liegt. Die Dateien im `/proc`-Verzeichnis werden virtuelle Dateien genannt, weil sie keinen Platz auf dem Datenträger belegen, sie sind nur im Arbeitsspeicher vorhanden und werden aus den Daten vom Kernel zur Laufzeit als normales Dateisystem zur Verfügung gestellt. So wird es möglich, den Inhalt dieser Dateien mit einem Standardkommando auszulesen oder zu editieren. Die Komponente `attribute-file` ist zuständig für solche Dateien im ZIBDMS, um Metadaten verwalten zu können.

Objekte im ZIBDMS haben Metadaten in Form von Attributen, die aus einem Attributnamen und einem oder mehreren Werten bestehen. Sie sind in einer relationalen Datenbank aufbewahrt und werden nicht in Objektdaten mitgespeichert. Außerdem kann jede Datei im ZIBDMS mehrere NSLs besitzen und sie werden ebenfalls in der Datenbank gespeichert. Zu diesen Eigenschaften einer Datei soll der Benutzer einen Zugang zum Lesen bekommen und bei Bedarf auch Hinzufügen bzw. Verändern können. ZIBDMS fasst die Metadaten einer Datei in folgenden virtuellen Dateien (auch Pseudodateien genannt) zusammen:

`<filename>..sysattr` präsentiert alle Systemattribute einer Datei und kann nur vom System geändert werden.

9. http://www.ietf.org/rfc/rfc2054.txt

`<filename>..attr`	beinhaltet die Benutzerattribute und ist editierbar.
`<filename>..nsl`	enthält die NSLs zu der Datei. Ein NSL ist vom System eingetragen, kann aber auch vom Benutzer geändert werden.

Der Vorteil dieses Konzeptes besteht darin, dass eine virtuelle Datei wie eine gewöhnliche Datei ohne zusätzliche Anpassung an die Anwendung behandelt werden kann. In der Tat wirkt es wie eine normale Datei, hat aber einige interessante Besonderheiten:

- Der Dateiname enthält eine spezielle Zusatzendung je nach Semantik. Im Verzeichnis wird diese Art von virtuellen Dateien nicht aufgelistet, sondern ihre Präsenz muss explizit aufgerufen werden.
- Die Zeit und das Datum der virtuellen Dateien spiegeln die aktuelle Zeit und das aktuelle Datum wider. Es muss aber nicht bedeuten, dass das Objekt ständig geändert wurde, sondern nur, weil die Datei erst zur Laufzeit generiert wurde. Die genaue Zeit des Entstehungszeitpunkts, des letzten Zugriffs sowie der letzten Änderung ist aus dem Dateiinhalt der `<filename>..sysattr` zu entnehmen.
- Die Größe der Datei gibt nur Auskunft über die gegenwärtige Größe der virtuellen Datei. Es entspricht nicht unbedingt der Größe des rohen Inhalts. Beispielsweise wird die aktuelle Dateigröße viel mehr verändert, wenn nur ein Byte hinzugefügt wurde. Hier handelt es sich um die zusätzlichen Zeichen, die für eine übersichtliche Darstellung des Dateninhalts sorgen.
- Ebenso können andere Systemattribute für verschiedene Zwecke modifiziert werden. Eine manuelle Veränderung des Eigentümers oder Gruppenbesitzers kann in bestimmter Situation nützlich sein. Zugriffsrechte können dadurch individuell festgelegt werden, um zu einer bestimmten Semantik zu gelangen, was im normalen Dateisystem nicht möglich ist. In `attributefile` werden alle Schreibrechte für `<filename>..sysattr` entzogen, denn diese Datei sollte nicht vom Anwender modifiziert werden. Eine

nachträgliche Änderung ist nicht möglich, auch wenn der Benutzer einen `root`-Zugang besitzt.

Folgendes Beispiel demonstriert die o. g. Merkmale:

```
> ls
bar    foo
> ls -la foo..sysattr
-r--r--r--  1 root root 439 Jun 20 12:07 foo..sysattr
```

4.2.2.1 Lesen einer virtuellen Datei

Da der Inhalt einer virtuellen Datei eine Abbildung von semantischen Informationen darstellt, besteht die Möglichkeit, weitere Informationen in die Datei einzubetten und sie in einer übersichtlichen Darstellung zu formatieren. Um eine virtuelle Datei zu lesen, wird zuerst eine Leseoperation in Form einer Abfrage an das System gesendet. Das System empfängt diese Anforderung, öffnet die Datei, verarbeitet und bereitet ihren Inhalt vor und übermittelt die generierten Daten zum Client. Diese Lesetransaktion ist von System zu System unterschiedlich und teilweise auch protokollabhängig. Im ZIBDMS ist das Lesen einer virtuellen Datei derzeit nur über die NFS-Schnittstelle interessant. Im `attribute-file` wurde eine atomare, also nicht zerlegbare Leseoperation entsprechend implementiert:

```
ssize_t read(const string& pathname, off_t offset,
             void *buf, size_t count);
```

- `pathname` ist ein virtueller Absolutpfad und wird vom NFS-Server über eine virtuelle UFI in einen Filehandle tranformiert.
- `offset` gibt an, ab welcher Position gelesen werden soll.
- `buf` fungiert als ein Platzhalter, um das Ergebnis in den Zwischenspeicher zurückzukopieren.
- `count` gibt an, wie viele Bytes gelesen werden sollen.

Und `ssize_t` erwartet die Anzahl der gelesenen Bytes als Rückgabe.

In der Implementierung der read-Prozedur wird der Inhalt der Datei immer zunächst komplett aus dem Katalog (mdc oder replication-catalog) geholt und in eine lesbare Form formatiert und dann ab dem `offset` `count`-Bytes abgeschnitten. Dies wird zum Schluss in den Puffer `buf` kopiert und `count` als Anzahl der erfolgreichen Bytes zurückgegeben.

Wenn es voraussehbar ist, welches `offset` als nächste Abfrage kommt, kann man einen Cache für den nächsten und vielleicht auch für den übernächsten Puffer zwischenspeichern. Der NFS-Server unterstützt derzeit leider nur UDP als Transportprotokoll. Deshalb wird auf eine Beschleunigung des Lesevorgangs durch Cachen verzichtet.

Eine andere Überlegung besteht darin, die Abfrage an den Katalog so zu formulieren, dass nur der angeforderte Teil aus dem Katalog geholt und nur dieser verarbeitet wird. Die Backend-Datenbank liefert leider nur das komplette Objekt zurück, so dass diese effizientere Lösung nicht viel weiter bringt. Es verbleibt an dieser Stelle ein Optimierungsbedarf.

```
> cat foo..sysattr
# ZIBDMS system attributes for /foo
# NOT WRITEABLE
ACCESS_TIME = 20050620T100709
CREATION_TIME = 20050620T100708
FILESIZE = 308123
GROUP = 0
GROUPEXECUTE = FALSE
GROUPREAD = TRUE
GROUPWRITE = FALSE
KIND = REG_FILE
MDC_TYPE = MDC_UFI
MD_UPDATE_TIME = 20050620T100709
MODIFY_TIME = 20050620T100709
OTHEREXECUTE = FALSE
OTHERREAD = TRUE
OTHERWRITE = FALSE
OWNER = 0
UFI = 733e5e6b-d1ab-41e8-97a5-af27834fbd4a
USEREXECUTE = FALSE
USERREAD = TRUE
USERWRITE = TRUE
```

Abb. 4.7 Beispiel für das Lesen einer virtuellen Datei

Auf der Client-Seite kann man ein unmodifiziertes Standard-Anzeigeprogramm benutzen, um den Inhalt einer virtuellen Datei wiederzugeben. Die Abbildung 4.7 zeigt ein Beispiel für das Auflisten aller Systemattribute einer Datei mit dem UNIX-Kommando `cat`.

4.2.2.2 Schreiben einer virtuellen Datei

In dem vorherigen Abschnitt wurde beschrieben, wie eine virtuelle Datei mit normalem Kommando gelesen werden kann. Logischerweise sollte angeboten werden, diese Datei mit einem neuen Inhalt zuschreiben bzw. eine Änderung darin vorzunehmen. Leider ist der Schreibvorgang einer virtuellen Datei aus folgenden Gründen nicht einfach:

- Eine virtuelle Datei existiert nicht auf irgendeinem Datenträger. Es gibt also keine Möglichkeit eine Adresse festzustellen, an welcher Stelle die Datei anfängt und wo das Ende ist.
- Der Inhalt wurde aus den Daten einer oder mehrerer Quellen geholt und mit weiteren Informationen zur Darstellung gemischt. Virtuelle Dateien haben meistens das Problem, dass die Position innerhalb der kodierten Ausgabedaten nicht entsprechend auf das zugehörige interne Datum zugewiesen werden kann. Das ist aber notwendig, wenn umfangreiche Informationen über die virtuelle Datei ausgegeben werden sollen. Wenn eine Änderung gemacht wird, muss man jede Abweichung erkennen, entsprechend der hinzugefügten Informationen parsen und nur die tatsächliche Änderung in die ursprünglichen Quellen schreiben.

In der Praxis wird eine virtuelle Datei nur im einfachsten Fall schreibend erlaubt, und zwar nur mit einem vorgegebenen Format. Ein interessantes Beispiel dafür ist das Schreiben von Dateien im `/proc`-Verzeichnis [Kofl04], wodurch Kernelparameter im laufenden System eingestellt werden können:

```
> echo 4 2 45 > /proc/sys/kernel/acct
```

Da ZIBDMS das Lesen einer virtuellen Datei über eine NFS-Schnittstelle unterstützt, sollte also das Schreiben auch über NFS folgen. Der Inhalt einer

virtuellen Datei wird in Blöcke zerlegt und an ZIBDMS übertragen. Jeder Block wird dann von einer atomaren Schreiboperation verarbeitet.

```
ssize_t write(const string& pathname, off_t offset,
              void *buf, size_t count);
```

- `pathname` ist ein virtueller Absolutpfad und wird vom NFS-Server über eine virtuelle UFI in einen Filehandle tranformiert.
- `offset` gibt an, ab welcher Position geschrieben werden soll.
- `buf` wird als Zwischenspeicher benutzt, um die zu schreibenden Daten an `write()` zu übergeben.
- `count` gibt an, wie viele Bytes gelesen werden sollen.

Und `ssize_t` erwartet die Anzahl der geschriebenen Bytes als Rückgabe.

Die Implementation von `write()` stößt neben den o. g. Problemen auf die Konsequenz des UDP-Transportprotokolls. UDP ist im Gegensatz zu TCP verbindungslos und nicht zuverlässig. Demzufolge besitzen die Blöcke weder eine sequenzielle Konsistenz noch eine kausale Konsistenz.

Eine erste Implementierung von `write()` in [Witt05] schrieb vor, dass jede virtuelle Datei ein Wort „`EOF`“[10] (*end-of-file)* am Ende der Datei beinhaltet. Dadurch wurde der letzte Block gekennzeichnet und somit angenommen, dass keine Blöcke mehr geschrieben werden sollten. Es wurden zuerst die Blöcke in einem Cache gesammelt, bis ein Block mit dem „`EOF`“ am Ende erscheint, dann werden alle Daten zu der virtuellen Datei im Katalog gelöscht und mit den geparsten Daten neu geschrieben. Diese Lösung ist zwar einfach zu implementieren, ergibt aber folgende zwei Probleme: Zum einen brauchte es bei einer großen Datei sehr viel Zeit, um alle Blöcke zu cachen. Wird die Zeit der maximalen Lebensdauer des Schreib-Caches [Witt05] überschritten, wird der alte Eintrag aussortiert. So gehen die ersten Blöcke verloren. Zum anderen ist es sehr wahrscheinlich, dass viele Blöcke nicht auf dem Schreib-Cache mitgespeichert werden. Es wurde zwar erkannt aber nicht

10. Es ist nicht mit dem EOF-Sonderzeichen zu verwechseln, das das Ende der Dateneingabe mit Hilfe von "Strg-D" signalisiert.

sorgfältig berücksichtigt, dass die Blöcke in beliebiger Reihenfolge ankommen. Denn es gibt keinerlei Garantie darüber, dass der Block mit „`EOF`“ als letzter Block ankommt.

Es muss also eine neue Lösung für die o. g. Probleme gefunden werden. Das Ziel ist eine schnelle Verarbeitung der Schreiboperation und ein effektives Schreiben in den Katalogen. Dabei sind folgende Rahmenbedingungen von der vorhandenen NFS-Implementierung und ZIBDMS-Katalogen zu berücksichtigen:

- Zwischen Client und Server gibt es eine zu definierende Puffergröße, die die maximale Länge einer atomaren Schreiboperation bestimmt und mit einem Parameter `wsize` (*write-size* in Bytes) beim Mounten gesetzt werden kann. `wsize` ist dann gleichzeitig die Transfergröße bei Schreiboperationen und wird automatisch durch eine `wtmult` (4096 Bytes, [Witt05]) teilbare Zahl angepasst. Der implementierte NFS-Server [Witt05] arbeitet einwandfrei nur mit einer Puffergröße von 8192 Bytes, obwohl das NFS-Protokoll Version 3 [SUN95] `wsize` bis zu einer Größe von 32768 Bytes erlaubt.
- Zwar stellt der interne NFS-Server die Puffer in einer nicht voraussehbaren Reihenfolge zur Verfügung, aber es ist feststellbar, welcher Puffer zuerst übertragen wird. Wenn die virtuelle Datei in mehrere Puffer zerlegt wird, wird der Puffer mit dem `offset=wsize` als erster übermittelt.
- Die Metadaten sind als Attribut-Wert-Paare in einer relationalen Datenbank gespeichert. Es ist somit leider keine logische Struktur zu erkennen. Außerdem darf ein Attribut mehrere Werte vom Typ `string` oder `int` haben, die wiederum als eigenständige Attribute exportiert werden. Das hat zur Folge, dass alle Metadaten bei jeder Änderung in einer virtuellen Datei gelöscht und mit einem neuen Inhalt geschrieben werden müssen. Es stellt sich die Frage, wann man die Metadaten löscht: Dies am Ende zu tun ist ineffizient, weil man alle Puffer zuerst sammeln muss, dazwischen ist aufgrund der Pufferzerlegung und zufälligen Pufferreihenfolge nicht möglich. Es bleibt nur noch der Beginn und zwar ganz am Anfang, bevor die Daten in den ersten Puffer geschrieben werden. Dieser Ansatz setzt allerdings ein vollständiges Schreiben bei jeder Modifizierung des Client-Editors voraus.

- NSL und Attribut-Wert-Paare lassen sich zeilenweise übersichtlich darstellen. Dieses Merkmal bringt den Vorteil, dass man sie auch zeilenweise verarbeiten kann. Aus einem Puffer lassen sich Attribute bzw. NSLs durch einen `newline` ('\n' unter UNIX/LINUX) als Trennungszeichen leicht parsen. Dieses Zeichen ist am Ende der Datei nur notwendig, wenn die Größe der Datei durch die Puffergröße teilbar ist und der Editor kein `newline` am Ende automatisch hinzufügt.

Eine neue Implementierung im Rahmen dieser Arbeit behebt die Probleme der ersten Implementation. Dafür wurde ein Algorithmus entwickelt:

(1) Lösche alle Metadaten des Objekts. Zuerst muss festgestellt werden, ob der aktuelle Puffer der erste Puffer ist. Aus `offset` und `count` lässt sich ein für alle Puffer gültiger Algorithmus entwickeln, um den Zeitpunkt des Löschens zu berechnen:

```
if (count - offset == 0) {
  object.removeAllMetaData();
  insert objectCache.isFirstBuffer;
  insert objectCache.metaDataDeleted;
} else if (count - offset < 0) {
  if ( not objectCache.isFirstBuffer.exists()) {
      object.removeAllMetaData();
      insert objectCache.metaDataDeleted;
  }
} else if (count - offset > 0) {
  if (objectCache.metaDataDeleted.exists()) {
      delete objectCache.metaDataDeleted;
  } else {
      object.removeAllMetaData();
  }
}
```

(2) Schreibe die nicht vollständigen Metadatenteile am Anfang und am Ende des Blocks in den Cache. Wenn das Token am Ende des Blocks nicht mit

`newline`-Zeichen abgeschlossen ist und kein dazu passender Teil am Anfang des Nachbarblocks aus dem Cache gefunden werden kann, wird dies als ein nicht vollständiger Metadatenteil erkannt. Ausgenommen von dem Puffer mit `offset=0` wird jedes Token am Anfang des Blocks in den Cache geschrieben, sofern kein passender Teil am Ende des Nachbarblocks aus dem Cache gefunden werden kann.

(3) Setze die nicht vollständigen Metadatenteile am Anfang und am Ende des Blocks mit den aus dem Cache gefundenen Teilen der Nachbarblöcke zusammen. Diese Metadaten sind jetzt vollständig und das Schreiben dieser Metadaten ist ein verzögertes Schreiben (*write-delayed*).

(4) Schreibe alle vollständigen Metadaten des Puffers zeilenweise in den Katalog.

In der neuen Umsetzung ist jede Schreiboperation in sich geschlossen. Jeder Puffer wird sofort verarbeitet und in die Datenbank geschrieben. Es wurde nur ein minimaler Cache benutzt, um die nicht vollständigen Metadatenteile mit dem Rest aus den Nachbarblöcken zusammensuchen zu können. Die Anforderungen werden erfüllt, man kann eine virtuelle Datei mit einem unmodifizierten Editor wie `vi`, `pico, nano` oder `emacs` bearbeiten.

```
> vi foo..attr
```

oder

```
> nano foo..nsl
```

4.2.2.3 Löschen einer virtuellen Datei

Wenn eine native Datei gelöscht wird, entfernt das System den Verweis auf den Speicherplatz dieser Datei und fügt den frei gewordenen Speicherplatz zum freien Speicherplatz des Dateisystems hinzu. Es überschreibt dieses Segment des Datenträgers aber nicht wirklich mit Nullen. Obwohl auf die Datei zwar tatsächlich nicht mehr zugegriffen werden kann, gibt es sie noch so lange, bis eine neue Datei über diese Datei geschrieben wird. Diese Semantik des Löschens kann allerdings nicht auf eine virtuelle Datei übertragen werden, weil sie nicht auf einem physischen Datenträger existiert, sondern sie wird nur über ein virtuelles Objekt angesprochen.

Deshalb wird kein Entfernen eines Verweises aus dem Dateisystem erforderlich. Es wird lediglich der rohe Inhalt aus dem Katalog gelöscht. Folgende Tabelle zeigt einen Vergleich der Löschsemantik:

Semantik des Löschens	Native Datei	Virtuelle Datei
Verweisentfernung im Dateisystem	ja	nein
Löschen des Inhalts auf dem Datenträger	nein	ja
Wiederherstellung	teilweise	nein

Tab. 4.1 Vergleich der Löschsemantik

In der Realisierung dieser Semantik wird eine atomare Operation `unlink()` im `attribute-file` implementiert:

```
void unlink(const string& pathname);
```

Es werden alle Metadaten quasi aus dem Inhalt der virtuellen Datei gelöscht. Der `pathname` enthält einen absoluten Pfadnamen einer nativen Datei und eine Semantikendung, die bestimmt, welche Metadaten aus der nativen Datei im Katalog gelöscht werden sollen. Das Löschen verläuft einfach. Anhand der Semantikendung werden entsprechende Metadaten aus dem Katalog geholt und dann nach und nach alle gelöscht. Wenn man danach versucht, die virtuelle Datei zu lesen, bekommt man einen leeren Inhalt zurück, weil es keine Metadaten mehr zu generieren gibt.

```
> cat foo..attr
Title = The Anatomy of the Grid: Enabling Scalable Virtual Organizations
author = Foster
author = Kesselman
author = Tuecke
> cat foo..nsl
cdtp://tanh.zib.de/YZsyeX
http://www.globus.org/alliance/publications/papers/anatomy.pdf
> rm foo..attr
> cat foo..attr
> rm foo..nsl
> cat foo..nsl
```

4.2.2.4 Ausführen einer Dateioperation

Im Abschnitt 2.2.4.3 wurde virtuelle Operation konzipiert, um eine bestimmte Operation auf einem Objekt ausführen zu lassen, die nicht über ein unmodifiziertes Kommando erhältlich ist. In diesem Abschnitt werden virtuelle Operationen benutzt, um Metadaten bzw. Adressen eines Replikats ein- oder auszutragen. Eine virtuelle Operation im ZIBDMS ist in der Regel mit einer Datei verbunden (*Dateioperation*). Zur Identifikation einer virtuellen Operation wird eine vordefinierte Endung wie „`.add`“ oder „`.rm`“ an die virtuelle Datei angehängt. Anhand dieses Merkmals kann die `write()`-Funktion entsprechende Aktionen auslösen. Es wird zwar die selbe `write()`-Funktion wie beim Schreiben einer virtuellen Datei benutzt aber hier handelt es sich um einen einzeiligen Puffer; damit werden Objektdaten korrekt in den Katalog geschrieben.

Auf der Kommandozeile wird ein Attribut oder eine NSL nur in eine Zeile angegeben. Diese Angabe kann Sonderzeichen, die der Shell als Syntaxzeichen interpretiert und eventuell (in dem Fall von „`/`“) nicht ausschließen lässt. Aus diesem Grund wird das Ausführen einer Dateioperation in der `write()`-Funktion implementiert, um die rohe Zeichenkette des `echo`-Befehls in eine virtuelle Datei umzulenken.

Objektdaten einfügen:

Objektdaten lassen sich mehrfach mit dem `echo`-Befehl einfügen. Dadurch kann ein Attribut mehrere Werte eingetragen bekommt, ebenso können mehrere Replikate zu einem Objekt eingetragen werden.

Das Hinzufügen eines Attribut-Wert-Paars erfolgt mit einem einfachen `echo`-Befehl:

```
> echo author=Foster > foo..attr.add
```

Um eine NSL eines Replikats in dem Dateireplikationskatalog einzutragen, kann man sich ebenfalls mit dem `echo`-Befehl bedienen:

```
> echo http://anywhere.com > foo..nsl.add
```

Objektdaten löschen:

Ein Attribut-Wert-Paar lässt sich intuitiv mit dem echo-Befehl löschen:

```
> echo author=Foster > foo..attr.rm
```

Wenn ein Attribut mehrere Werte hat, wird nur eine Angabe des Attributnamens benötigt:

```
> echo author > foo..attr.rm
```

Die eingetragene NSL kann mit dem echo-Kommando ebenfalls entfernt werden:

```
> echo http://anywhere.com > foo..nsl.rm
```

Datei assimilieren:

Der Assimilationsvorgang ist gleichzeitig ein Registrierungsvorgang einer Datei oder eines Verzeichnisses ins ZIBDMS. Das Objekt wird übers Netz geholt und analysiert. Die gewonnenen Metadaten und die NSL werden in die Kataloge geschrieben. Die Voraussetzung dafür ist, dass das Zugangsprotokoll (z. B. HTTP, FTP) vom ZIBDMS unterstützt werden muss.

```
> echo http://www.globus.org/alliance/publications/papers\
/anatomy.pdf > foo..nsl
```

In dem obigen Beispiel wird die PDF-Datei von der URL „http://www.globus.org/alliance/publications/papers/anatomy.pdf“ geholt. Zu der Datei „foo“ im ZIBDMS werden diese URL-Adresse in den Dateireplikationskatalog und Metadaten wie Dateigröße, Erstellungsdatum usw. in den Metadatenkatalog geschrieben.

4.2.3 weak-link-layer

Im Abschnitt 2.1 wurde eine Einschränkung des hierarchischen Dateisystems diskutiert, da sich eine Datei nur an einem bestimmten Ort im Dateisystem befindet. Wenn man diese Datei aus einem anderen Ort ansprechen will, muss man oft einen langen und unbequemen Pfad eingeben. Aus diesem Grund wurde ein Verweismechanismus konzipiert, wodurch Objekte an verschiedenen Orten gemeinsam genutzt werden können.

4.2.3.1 Symlink und Hardlink

Verweise in einem hierarchischen Dateisystem werden durch Hardlinks und Symlinks [Kofl04] realisiert. Im ZIBDMS werden sie in Objekten [Witt05] implementiert:

- **Hardlink:** Für jede neue Instanz einer Datei im ZIBDMS wird entsprechend ein `HFNObject` (siehe die Beschreibung von `link()` im Abschnitt 4.2.1) erzeugt, das einen neuen Pfad für die Datei beinhaltet. Leider kann ein Hardlink nicht auf Verzeichnisse hinweisen und nur innerhalb des ZIBDMS[11] benutzt werden.
- **Symlink:** Diese Einschränkungen werden mit dem Konzept des symbolischen Links behoben, bei dem es sich um einen Verweis auf den Eintrag im Verzeichnis handelt, das das Zielobjekt enthält. Wird das Zielobjekt verschoben oder gelöscht, weist der Link anders als bei einem Hardlink ins Leere.

4.2.3.2 Weak-Link

Das Verweiskonzept in einem klassischen Dateisystem hat sich über Jahre hinweg bewährt. Es wurden jedoch folgende Mängel noch nicht behoben:

- *Konsistenz und Aktualität eines Verweises:* Wenn das Objekt verschoben oder gelöscht wird, weisen alle symbolischen Links des Objektes ins Leere. Die Funktionstüchtigkeit der Symlinks hängt dann davon ab, ob diese Stelle wieder besetzt wird. Es existiert bisher noch kein Mechanismus, um die Konsistenz der Verweise zu halten bzw. die Verweise mit dem aktuellen Stand des Objektes zu aktualisieren.
- *Behandlung von Verklemmung*[12]*:* In einem klassischen Dateisystem wie unter LINUX/UNIX kann man einen symbolischen Link zu einem Objekt erzeugen, bevor das Objekt überhaupt existiert. Es ist auch erlaubt, mit einem symbolischen Link auf einen anderen symbolischen Link zu verweisen. Das führt dazu, dass dadurch ein Zyklus entstehen kann. Auf der Dateisystem-

11. also nicht partitionsübergreifend
12. deadlock

ebene werden keine Gegenmaßnahmen zur Behandlung einer möglichen Verklemmung bereitgestellt. Deswegen versucht das Betriebssystem bzw. der Shell den Zyklus auf unterschiedliche Art und Weise zu entdecken und die Verklemmung aufzulösen.

- *Liste der Verweise zu einem Objekt:* Aufgrund der Datenorganisation in sequentiellen Blöcken in einem klassischen Dateisystem kann man für ein Objekt keine Verweisliste ausstellen. Eine Liste der Verweise kann bei der Entscheidung helfen, ob die unnötigen Symlinks vorher gelöscht werden oder ob das Objekt überhaupt gelöscht oder verschoben werden soll.

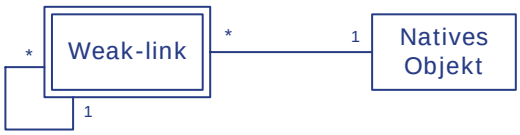

Abb. 4.8 Modellierung des Weak-Link-Konzepts

Aus den o. g. Problemen kann man den Zusammenhang zwischen einem symbolischen Link und einem Objekt feststellen, dass der Link in einer Abhängigkeit zum Objekt stehen soll. Aus dieser Kenntnis wird in dieser Arbeit ein sogenannter Weak-Link[13] eingeführt, der folgende Eigenschaften besitzt:

- *Symlink basierend:* Weak-Link ist eine Erweiterung des Symlinks, denn Symlinks sollten mit seinen Vorteilen weiterhin zur Verfügung stehen. Im Dateisystem erscheint ein Weak-Link als ein Symlink und wird auch wie ein Symlink behandelt. Der Mehrwert des Konzeptes besteht darin, dass ein Weak-Link exakt die Lebensdauer und immer den aktuellen Stand des Zielobjekts hat. Es werden also die Konsistenz und die Aktualität der Links sichergestellt.
- *Verklemmungsvorbeugung:* Wie in der Abbildung 4.8 skizziert, kann ein natives Objekt viele Weak-Links bekommen, auf die wiederum viele Weak-Links verweisen. Es kann aber kein Zyklus entstehen, weil die Existenz eines Weak-Links von der Existenz des nativen Objekts abhängig ist.

13. Die Idee ähnelt dem Konzept einer schwachen Entität im Datenbanksystem [KeEi04].

Die Funktionalität eines Weak-Links gegenüber Symlinks und Hardlinks kann man der Tabelle 4.2 entnehmen:

Verhalten	**Hardlinks**	**Symbolische Links**	**Weak-Links**
Verweis auf eine Datei	ja	ja	ja
Verweis auf ein Verzeichnis	nein	ja	ja
Mehrstufig	gleichberechtig	ja	ja
Partitionsübergreifend	nein	ja	ja
Zyklusbildung	nein	möglich	nein
Löschen des Objekts	eine Instanz weniger	Links->Leere	rekursiv gelöscht
Auskunft über andere Instanzen	im ZIBDMS möglich	nur aktive Links	möglich

Tab. 4.2 Verhalten von Verweisen im ZIBDMS

Um ein Weak-Link zu erzeugen, kann man das Kommando zur Erzeugung eines Symlinks wie gewohnt verwenden. Es werden zwei gleichnamige atomare Operationen aufgerufen:

```
void symlink(const string& pathname, const string& linkinfo);
```

Der erste Aufruf wird in `hierarchical-layer` durchgeführt und sorgt für eine normale Symlinkobjekterzeugung. Die Weak-Link-Semantik wird mit einem zweiten Aufruf in `weak-link-layer` behandelt. Dabei wird der absolute Pfad des Zielobjekts aus dem absoluten Pfad des Symlinkobjekts `pathname` und die Symlinkangabe `linkinfo` berechnet, denn `linkinfo` wird üblicherweise mit einem relativen Pfad des Zielobjekts übergeben. In diesem zweiten `symlink()`-Aufruf wird ein Systemattribut dem erzeugten Symlinkobjekt hinzugefügt:

```
DESTINATION = [absolute Pfad des Zielobjektes]
```

Dieses Attribut bei einem Symlinkobjekt besagt, dass das Symlinkobjekt ein Weak-Link-Objekt ist; ein Symlinkobjekt ohne dieses Attribut bleibt also ein normaler Symlink. Das Attribut dient außerdem noch dazu, die Aktualität und Konsistenz eines Weak-Links abzusichern. Wenn das Zielobjekt verschoben wird, kann

man mit Hilfe vom DESTINATION-Attribut alle Weak-Links des Zielobjekts aussuchen und mit dem neuen Pfad in DESTINATION und in SYMLINKINFO[14] aktualisieren. Wenn das Zielobjekt gelöscht wird, werden alle zu dem Objekt gezeigten Weak-Links und deren rekursiven Weak-Links gelöscht.

Auf der Kommandozeile wird diese Weak-Link-Semantik mit einem virtuellen Parameter „..weaklink“ (siehe Abschnitt 2.2.4.4) ausgedrückt:

```
> ln -s foo foowl..weaklink
```

4.2.3.3 Navigation und Verweisauflistung

Das weak-link-layer erweitert nicht nur das Verweiskonzept sondern stellt auch weitere Mechanismen zur Verfügung, um die Verweise eines Objektes aufzulisten und eine Navigation im Verzeichnisbaum zu ermöglichen. Es werden alle Hardlinks, funktionstüchtige Symlinks und alle Weak-Links zu einem Objekt im System in einem virtuellen Verzeichnis aufgelistet:

- Bei Hardlinks werden als Dateien dargestellt, weil Hardlinks nur Verweise auf Dateien sind. Es werden alle HFNObjects zu einem Objekt geholt und daraus die Pfade berechnet.
- Funktionstüchtige Symlinks und alle Weak-Links werden als virtuelle Symlinks angezeigt, sofern sie keine mehrstufigen Links sind, denn die mehrstufigen Links werden als virtuelle Unterverzeichnisse dargestellt, um weitere tiefere Navigationen zu ermöglichen.

Das untere Beispiel weist eine Auflistung der Links eines Objekts auf. Wenn man zu einem Objekt „foo“ wissen will, ob es dazu Verweise gibt und welche sie sind, dann kann man einfach versuchen, in das dazugehörige Verzeichnis „@foo“ hineinzuwechseln und es aufzulisten. Im Beispiel kann man erkennen, dass „\dir\foolink“ als virtueller Symlink dargestellt wird, weil keine weiteren Verweise auf ihn zeigen. „\bar“ ist ein Hardlink von „foo“, daher wird er als normale Datei angezeigt. Dagegen hat „\foo2“ weitere Verweise, die auf „\foo2“ verweisen, deswegen ist „\foo2“ als virtuelles Verzeichnis anzusehen.

14. Systemattribut, dessen Wert aus dem übergebenen linkinfo gesetzt wird (siehe Abschnitt 4.2.1).

```
> cd @foo
> ls -l
total 3
lrwxrwxrwx  1 root root 6 Oct 24 19:22 \dir\foolink -> ../foo
-rw-r--r--  1 root root 0 Oct 24 19:21 \bar
drwxrwxrwx  1 root root 3 Oct 24 19:23 \foo2
```

Zu diesem virtuellen Verzeichnis kann man logischerweise auch wechseln. Die folgende Abbildung listet den Unterverweis auf:

```
> cd \\foo2
> ls -l
total 1
lrwxrwxrwx  1 root root 4 Oct 24 19:23 \dir\underfoo -> foo2
```

Diese Struktur bildet einen zweistufigen Link „`foo2 -> foo`“ und „`/dir/underfoo -> foo2`“ ab. Der Verweisbaum kann beliebig erweitert werden und er lässt sich wie ein Verzeichnisbaum auflisten und man kann darin navigieren.

4.2.4 query-directory

Eine Abfrage an die Datenbank im ZIBDMS wird als virtuelles Verzeichnis formuliert. Als Eingabe bekommt `query-directory` einen Pfad, der durch einen Parser lexikalisch und syntaxtisch analysiert wird und als entsprechende Abfrage an die Datenbank geschickt wird. Das Ergebnis dieser Abfrage wird in einem Query-Verzeichnis präsentiert. Ein Query-Verzeichnis ist ein virtuelles Verzeichnis und besitzt alle der im Abschnitt 2.2.4.1 beschriebenen Eigenschaften.

Für jede Abfrage in der Form eines Query-Verzeichnisses werden die entsprechenden Ergebnisse aus der Datenbank geholt und in einer übersichtlichen und direkt zugreifbaren Darstellung zur Verfügung gestellt. `query-directory` ist in der Lage, folgende Beispielabfrage zu beantworten:

```
Gesucht werden alle im Jahr 1991 veröffentlichten
Publikationen von Herrn Gifford mit einem "Semantic"-
angefangenen Titel und nicht größer als 200 KBytes.
```

Abb. 4.9 Eine zu lösende Aufgabe mit einem Query im ZIBDMS

4.2.4.1 Query

Ein elementares Query drückt einen Attribut-Wert-Vergleich aus und hat folgendes Format:

```
Attribut op Wert
```

- `Attribut` ist ein String und erlaubt die Abfrage sowohl mit Systemattributen als auch mit Benutzerattributen.
- `op` ist der logische Vergleichsoperator. Verwendbare Vergleichsoperatoren sind in der Tabelle 4.3 definiert.

Symbol	Semantik
==	gleich
>	größer
<	kleiner
>=	größer oder gleich
<=	kleiner oder gleich

Tab. 4.3 Zulässige Vergleichsoperatoren im query-directory

- `Wert` hat einen ganzzahligen oder alphanumerischen Datentyp. Aus dem als Zeichenkette übergegebenen Wert wird der Typ von einem Parser erkannt, ob es sich um einen String oder eine ganze Zahl handelt. Wenn man eine ganze Zahl als String explizit ausdrücken will, muss man diese Zahl mit umschließenden Anführungszeichen schreiben.

Das o. g. Query im `query-directory` benutzt je nach Semantik die von der `mdc`-Komponente zur Verfügung gestellten Queries. `mdc` unterstützt grundsätzlich zwei Query-Typen (siehe Abbildung 4.10):

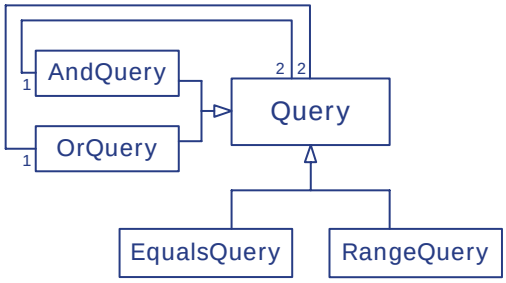

Abb. 4.10 Spezialisierung eines komplexen Querys

EqualsQuery

EqualsQuery ist ein einfacher Attribut-Wert-Vergleich. Es werden alle Objekte aus der Datenbank geholt, die das Attribut mit dem angegebenen exakten Wert besitzen.

```
QueryPtr query (new EqualsQuery(attribute, value));
```

RangeQuery

RangeQuery kümmert sich um die Bereichabfrage. Üblicherweise werden alle Queries mit einem der {>, >=, <, <=}-Operatoren in RangeQuery mit entsprechendem Bereich transformiert. Es werden alle Objekte aus der Datenbank geholt, die das Attribut mit dem angegebenen Wert innerhalb eines Bereichs besitzen. Ein Bereich im RangeQuery besteht aus einer unteren Schranke und einer oberen Schranke:

```
QueryPtr query (new RangeQuery(attribute, begin, end));
```

Bei einem Gleichheitsoperator wird RangeQuery vor allem benutzt, um mehr Ergebnisse mit einer Rechtstrunkierungssuche (mehr dazu im Abschnitt 2.2.3.1) aus einer Zeichenkette zu erzielen. Diese Option ist besonders nützlich, wenn man nur den Anfang des Strings kennt. Es wird

eines der Jokerzeichen[15] in der Tabelle 4.4 am Ende des gesuchten Werts angehängt:

Symbol	Semantik
*	findet alle Attributwerte, die genau den angegebenen Wert haben oder mit dem angegebenen Wert anfangen.
+	findet alle Attributwerte, die exklusiv mit dem angegebenen Wert anfangen.

Tab. 4.4 Zulässige Jokerzeichen am Ende des gesuchten Wertes

4.2.4.2 Logische Verknüpfung mehrerer Queries

In dem vorherigen Abschnitt wurde ein Query als einzelne Abfrage vorgestellt. Dieses einfache Query drückt ein Kriterium aus, um eine Teilmenge von Objekten aus der gesamten Menge im Katalog auszuwählen. Diese Teilmenge entspricht nicht immer dem gewünschten Ergebnis. Deswegen sollen weitere Kriterien eingeführt werden, um diese Teilmenge zu verfeinern oder zu erweitern. Dies entspricht dem Durchschnitt und der Vereinigung in der Mengenlehre. Die Beziehung zwischen Teilmengen wird in der booleschen Algebra mit den booleschen Operatoren verknüpft. Diese Verknüpfung erfolgt nach der logischen Syntax, die als eine Suchmethode im Abschnitt 2.2.3.4 vorgestellt wurde. Um eine Abfrage auszudrücken, werden folgende Notationen verwendet:

Symbol	Semantik
&&	Logischer AND-Operator, wird mit höherem Vorrang ausgewertet und über `AndQuery` verarbeitet.
||	Logischer OR-Operator, wird über `OrQuery` verarbeitet.
( )	Gruppierung von logisch verknüpften Termen.

Tab. 4.5 Syntaxnotation von logisch verknüpften Queries

Für jeden logischen Operator in der Tabelle 4.5 wird ein entsprechendes Query zur Verfügung gestellt. Aus der Abbildung 4.10 ist zu erkennen, dass

15. auch Metazeichen genannt

AndQuery oder OrQuery zwei Queries miteinander logisch verknüpft, die wiederum AndQuery bzw. OrQuery beinhalten können. Durch diese rekursive Eigenschaft entsteht ein binärer Query-Baum, wobei EqualsQuery und RangeQuery auf den Blättern zu finden sind. Eine Query-Verknüpfung mit AndQuery und OrQuery ist primitiv zu erzeugen:

AndQuery

```
QueryPtr query (new AndQuery(query1, query2));
```

OrQuery

```
QueryPtr query (new OrQuery(query1, query2));
```

4.2.4.3 Query-Grammatik

Wie oben schon erwähnt, wird die Formulierung eines Querys mit Hilfe der o. g. Syntaxnotation in einem Pfad ausgedrückt. Dieser Pfad akzeptiert eine kontextfreie Grammatik, die in der Abb. 4.4 auf Seite 71 definiert wurde. Das Query-Verzeichnis ist zwar in einem normalen Pfad integriert, beinhaltet aber weitere Regeln, die sich in eine weitere kontextfreie Grammatik übertragen lassen. Im Folgenden wird für die logischen Verknüpfungen im Pfad eine Query-Grammatik in EBNF vorgestellten Produktionsregeln [ISO01] aufgestellt:

```
expression      ::= directory { '/' directory }
directory       ::= term { "||" term }
term            ::= query-directory { "&&" query-directory }
query-directory ::= char { char } | '(' expression ')'
```

Abb. 4.11 EBNF-Grammatik für boolesche Verknüpfungen im Pfad

In dem Abschnitt 2.2.3.4 wurde beschrieben, dass die logisch verknüpften Terme in der Implementierung in einer Rangfolge ausgewertet werden. Aus der o. g. Grammatik kann die Rangfolge der logischen Verknüpfungen bei der späteren Query-Verarbeitung abgelesen werden:

- Ausdrücke in runden Klammern werden zuallererst ausgewertet.

- Die Grammatik bietet die Möglichkeit, „/“ gleichwertig wie „&&“ zu behandeln.
- „&&“ hat eine stärkere Bildung als „||“.

4.2.4.4 Query-Parser

Bevor eine Abfrage in Form von Zeichenketten an die Datenbank geschickt werden kann, muss sie zuerst über einen Parser syntaktisch analysiert und entsprechend semantisch verarbeitet werden. Parser sind konkrete Implementierungen von abstrakten Automaten, die eine Grammatik akzeptieren. Die Implementierung eines Parsers in einer Programmiersprache erfolgt je nach Komplexität der Eingabe durch einen der folgenden Ansätze:

- **Manueller Parser:** Für einen einfachen Automat kann man den Parser selbst schreiben. Der Programmierer kann frei entscheiden, wie der Parser gestaltet wird und wie die Ausgabe verarbeitet werden soll. Diese Lösung ist allerdings aufwändig und fehleranfällig, weil der Programmierer alle auftretenden Fälle ganz am Anfang und bei jeder Änderung abdecken muss.
- **Parser-Generator:** Ein Parser kann aus einer gegebenen Grammatik mit einem Parser-Generator automatisch generiert werden. Zuerst wird eine lexikalische Analyse mit Hilfe eines Scanners durchgeführt, in der die Eingabe in separate Tokens[16] zerlegt wird. Dann folgt eine syntaktische Analyse des Tokenstroms durch einen Parsergenerator. In dieser Phase werden die einzelnen Tokens zu Regeln gruppiert und ein Syntaxbaum aufgebaut. Nach einer Prüfung auf syntaktische Richtigkeit wird ein Code in der Programmiersprache zur Verfügung gestellt. Der Programmierer muss diesen Code lediglich mit dem eigentlichen Programmcode einbinden und die semantischen Aktionen entsprechend anpassen. Zu den bekanntesten Scanner/Parser-Werkzeugen gehören `lex`/`yacc`[17] [Hero03, LeMB95] und die erweiterte GNU-Version `Flex`[18]/`Bison`[19] [Hero03, LeMB95], die aus dem Compiler-

16. syntaktische Einheiten
17. **Y**et **A**nother **C**ompiler-**C**ompiler
18. http://www.gnu.org/software/flex/
19. http://www.gnu.org/software/bison/

bau zuverlässig über Jahre hinweg für sich sprechen. Neben der auffälligen Größe des generierten Codes ist die Programmierung bei dieser Lösung auch ziemlich kryptisch. Denn die semantischen Aktionen müssen bei der Regelbeschreibung mit Syntaxspezifikationen vom Scanner- und Parser-Werkzeug und Programmiersyntax gemischt programmiert werden.

- **Inline-Parser:** Man kann den Parser im Programmcode in der Programmiersprache schreiben. Anders als die Programmierung eines manuellen Parsers wird hier eine Grammatik benötigt. Ein Inline-Parser sollte vollständig in der Programmiersprache wie eine Bibliothek integriert werden und die Eigenschaften der Programmiersprache besitzen, so dass sowohl die Grammatik als auch die entsprechenden semantischen Aktionen in dem Hauptprogrammcode geschrieben und übersetzt werden können, ohne einen Zwischenschritt bzw. eine Einbindung explizit veranlassen zu müssen. Im Vergleich zu beiden o. g. Lösungen hat dieser Ansatz den flexiblen Programmierungsaspekt des manuellen Parsers und die Korrektheit eines Generators, weil die Funktionsfähigkeit des zu implementierenden Automaten durch die Grammatik zuverlässig garantiert wird.

Die o. g. Lösungen wurden bei der Implementierung des Query-Parsers untersucht, inwieweit sich dies ins ZIBDMS effektiv implementieren lässt. Aufgrund der Trennung zwischen dem Query-Typ und der logischen Verknüpfung in der Abfrage besteht der Parser im ZIBDMS aus einer Kombination von einem kleinen manuellen Parser zur Erkennung beider Query-Typen und einem Inline-Parser für die logische Syntaxverknüpfung.

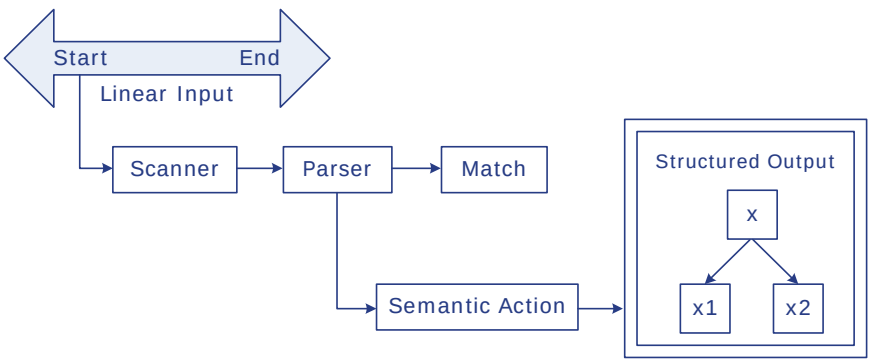

Abb. 4.12 Spirit-Basiskonzept [Guzm03]

Der Inline-Parser wird mit dem Spirit-Parser-Generator-Framework [Guzm03] realisiert, welches ein Bestandteil der Boost[20]-Bibliothek ist. Spirit-Parser ist ein Recursive-Descent-Parser[21], der die objektorientierte Eigenschaft von C++ besitzt und sich durch das Überladen von Operatoren in dem Programmcode vollständig integrieren lässt. Um die Eingabe zu parsen, benötigt Spirit lediglich eine kontextfreie Grammatik, die durch die transparenten Scanner und Parser (siehe Abbildung 4.12) verarbeitet wird. Danach werden vom Programmierer die entsprechenden semantischen Aktionen durchgeführt.

```
#include <boost/spirit.hpp>
struct queryGrammar : public grammar<queryGrammar> {
   template<typename S>
   struct definition {
      rule<S> dir, expr, term, query_dir;
      definition(const queryGrammar& self) {
         using namespace query_action; //for semantic actions
         expr = dir       >> *( ('/' >> dir)[&AND] );
         dir  = term      >> *( ("||" >> term)[&OR] );
         term = query_dir >> *( ("&&" >> query_dir)[&AND] );
         query_dir = ( +( alnum_p
                        | '.' | '\\' | '_' | '/' | '-'
                        | '=' | '>'  | '<' | '*' | '+' )
                     )[PUSH()]
                     | '(' >> expr >> ')';
    }
    const rule<S>& start() const {
      return expr;
    }
  }; //struct definition
}; //struct queryGrammar
```

Abb. 4.13 C++-Implementierung der Query-Grammatik im Programmcode mit dem Spirit-Parser-Framework

Eine kontextfreie EBNF-Grammatik kann mit der Spirit-Parser-Bibliothek in dem normalen C++-Code gemischt implementiert werden. Die Abbildung 4.13 präsentiert eine quasi 1:1-Transformation von einer in der Abb. 4.11 auf Seite 97 aufgestellten Grammatik in den C++-Programmiercode. Es müssen nur noch die

20. http://www.boost.org
21. Ein Top-Down-Parser

semantischen Aktionen angepasst werden.

Der implementierte Parser benutzt ein *Stack*[22], um ein elementares Query auf dem Zwischenspeicher zu halten. Im Folgenden wird die Arbeitsweise der o. g. semantischen Aktionen aufgelistet:

- `PUSH()` Hier wird ein elementares Query erkannt. Dieses Query wird in der Funktion `PUSH()` manuell geparst. Aus den Tokens (Attribut, Vergleichsoperator, Wert, Jokerzeichen) werden entsprechend ein Equalsquery bzw. RangeQuery erzeugt und auf dem Stack für eine eventuelle weitere Verknüpfung mit einem anderen Query gelegt.
- `&AND` Es wird eine logische `AND`-Verknüpfung erkannt. Dieser logische Operator verknüpft die beiden letzten Queries auf dem Stack mittels AndQuery. Das daraus resultierende Query wird auf dem Stack für eine eventuelle weitere Verknüpfung mit einem anderen Query gelegt. Es ist festzustellen, dass „`/`“ und „`&&`“ gleichwertig und vor „`||`“ ausgewertet werden.
- `&OR` Es wird eine logische `OR`-Verknüpfung erkannt. Dieser logische Operator verknüpft die beiden letzten Queries auf dem Stack mittels OrQuery. Das daraus resultierende Query wird auf dem Stack für eine eventuelle weitere Verknüpfung mit einem anderen Query gelegt.

Die Implementation des kompletten Query-Parsers umfasst nur ca. 200 Zeilen Code, wobei die Hälfte davon aus dem manuellen Parser stammt. Der Spirit-Inline-Parser und die entsprechenden semantischen Aktionen sind klein und einfach zu halten, so dass eine effiziente und schnelle Verarbeitung ermöglicht wird.

4.2.4.5 Query-Verarbeitung

Um Ergebnisse eines Querys zurückliefern zu können, müssen Daten aus der Backend-Datenbank geholt werden. Datensätze in der Datenbank lassen sich nur mit

22. Eine nach dem LIFO-Prinzip organisierte Datenstruktur

einer Abfragesprache abfragen. Das vom Parser zurückgelieferte Query muss also in die Abfragesprache umgewandelt werden. Zuerst leitet `query-directory` das Query an die Query-Unterkomponente der mdc-Komponente weiter. Hier erfolgt dann ein Mapping zwischen dem vom Parser zurückgelieferten Query und der von der Datenbank akzeptierten Abfragesprache. Das Ergebnis wird für die Ergebnisdarstellung zur Verfügung gestellt.

Query-Konsistenz [GJSO91]

Das Query im ZIBDMS ist konsistent. Das bedeutet, dass die Ergebnisse eines Querys mit dem aktuellen Verzeichnisinhalt übereinstimmen. Es wird nur nach Dateien und Unterverzeichnissen unterhalb des aktuellen Verzeichnisses abgefragt. Dafür wird eine zusätzliche Bedingung an das Query gehängt:

```
newQuery ="PARENT_NAME==" + currentDirectory + "*&&" + query;
```

Query-Optimierung

Durch den Parser wird das Query in der Form eines binären Baums zurückgeliefert. Es sind logisch verknüpfte Terme (elementare Queries), die für eine schnellere Verarbeitung optimiert werden sollen. Es sollten folgende offene Punkte untersucht werden:

- Inwieweit ist ein OrQuery kostenintensiver als ein AndQuery?
- Zu welcher Komplexitätsklasse gehört der Algorithmus zur Umformung und Minimierung einer disjunktiven Normalform des ZIBDMS-Querys?

Query-Speicherung

Wenn in dem aktuellen Verzeichnis ein natives Verzeichnis mit dem Namen des Query-Verzeichnisses existiert, wird die Trefferliste eines Querys aus der Ergebnisliste des Querys und dem Inhalt des nativen Verzeichnisses zusammengesetzt. Dieses Merkmal kann verwendet werden, um ein Query zu speichern. Man erzeugt ein normales Verzeichnis mit dem Namen eines gewünschten Querys:

```
> mkdir AUTHOR==Gifford
```

In diesem Verzeichnis kann man weitere Dateien manuell organisieren, die

nicht unbedingt zu dem Query passen. Dies erlaubt eine flexible Datenorganisation auf eine einfache Art und Weise.

4.2.4.6 Ergebnisdarstellung

Ein GUI zu einer graphischen Gestaltung der Trefferliste wie im [Appl05a] bietet dem Benutzer eine flexible und angenehme Sicht. Dieser Komfort kann bei einer textuellen Anzeige im Terminalfenster nicht eingerichtet werden. Es sollte trotzdem eine übersichtliche Darstellung geben, wodurch ein direkter flexibler Zugriff auf das Resultat ermöglicht wird. Die Ergebnisse werden in einem von zwei Darstellungsmodi zur Verfügung gestellt:

Repräsentanzmodus

Die Ergebnisse eines Querys werden unter Berücksichtigung des aktuellen Verzeichnisses als virtuelle Dateien oder virtuelle Verzeichnisse in einem Query-Verzeichnis zusammengefasst. Diese virtuellen Dateien und Verzeichnisse werden nach dem Konzept des virtuellen Objekts (Abschnitt 2.2.4) erzeugt. Sie sind virtuelle Instanzen der entsprechenden nativen Objekte und tragen ihren absoluten Pfad kodiert im Namen. Dieser kodierte Name wird beim Zugriff intern innerhalb des Systems zum ursprünglichen absoluten Pfad enkodiert, so dass der Zugriff auf ein virtuelles Objekt wie auf ein natives Objekt erfolgt.

In der Abbildung 4.9 wurde eine Abfrage in der natürlichen Sprache formuliert. Diese Abfrage lässt sich in ein Query wie in der Abbildung 4.14 transformieren. Das Ergebnis dieses Querys wird in Form eines virtuellen Verzeichnisses im ZIBDMS präsentiert. Man kann in dieses virtuelle Verzeichnis wie in ein normales Verzeichnis hineinwechseln und mit den darin befindlichen virtuellen Dateien wie mit normalen Dateien mit einem unmodifizierten Programm lesen. Beim Lesen der virtuellen Datei „`\semFS\sfs.pdf`“ wird der im Namen kodierte absolute Pfad identifiziert und das Lesen der originalen Datei „`/semFS/sfs.pdf`“ wird veranlasst.

```
> cd "AUTHOR==Gifford&&TITLE==Semantic*\
&&CREATION_TIME==1991*&&(FILETYPE==pdf||FILETYPE==ps)"
> pwd
/AUTHOR==Gifford&&TITLE==Semantic*&&CREATION_TIME==1991*&&
(FILETYPE==pdf||FILETYPE==ps)
> ls
\semFS\sfs.pdf        \semFS\sfs.ps
> cd "FILESIZE<=204800"
> ls
\semFS\sfs.pdf
> acroread \\semFS\\sfs.pdf
```

Abb. 4.14 Zugriff auf ein virtuelles Verzeichnis im ZIBDMS

In diesem Modus kann eine Abfrage sehr viele Ergebnisse ergeben, so dass die Darstellung unübersichtlich wird. Diese Ergebnismenge kann disjunkt in mehrere virtuelle Unterverzeichnisse unterteilt werden. Diese getrennte Gruppierung kann dynamisch um eine der folgenden Optionen erweitert werden:

- Wenn ein wie im Abschnitt 2.2.6.1 beschriebenes Ranking-Konzept realisiert wird, kann eine konfigurierbare Anzahl der relevanten Treffer aufgelistet werden. Der Rest ist in weiteren virtuellen Unterverzeichnissen zu finden.
- Die Ergebnismenge kann nach Dateiarten[23], Kategorien usw. gruppiert werden.
- Es kann aber auch automatisch in den Navigationsmodus umgeschaltet werden, in dem die Treffer hierarchisch schrittweise verfeinert werden können.

Navigationsmodus

Das Kriterium für eine automatische Umschaltung ist die Anzahl der zurückgegebenen Objekte. Wenn es größer als eine einstellbare Zahl ist, werden nur Dateien und Verzeichnisse (ausgenommen deren Unterverzeichnisse) innerhalb des aktuellen Verzeichnisses in originalen Namen zurückgegeben. Das Wechseln in ein Ergebnisverzeichnis bedeutet eine weitere Eingrenzung des Querys, denn der Kontext, also das aktuelle Verzeichnis, wird immer berücksichtigt. Es ist allerdings

23. Bilder, Musik, Dokumente usw.

nicht vorhersehbar, ob sich das gewünschte Ergebnis in diesem Verzeichnis befindet. Man kann aber immer mit dem üblichen Befehl „`cd ..`“ zurückwechseln.

```
> ls -ld AUTHOR==Gifford
dr-xr-xr-x  15 root root 5084 Jul  2 23:07 AUTHOR==Gifford
> cd AUTHOR==Gifford
> ls
semFS        www        zibdms
> cd semFS
> pwd
/AUTHOR==Gifford/semFS
> ls
sfs.pdf        sfs.ps
> acroread sfs.pdf
```

Abb. 4.15 Navigation in einem virtuellen Verzeichnis im ZIBDMS

Die Abbildung 4.15 zeigt eine Navigation innerhalb des virtuellen Verzeichnisses. Durch das Wechseln in das Ergebnisverzeichnis „`semFS`“ erhält man weitere Treffer und sie können wie gewohnt gelesen werden.

4.2.5 collection-layer

Das hierarchische Dateisystem erlaubt, eine Datei nur in einem Verzeichnis zu ordnen. Es gibt jedoch Verweismechanismen, wonach man die Datei in mehreren Verzeichnissen instanziieren kann. Leider ergibt sich hierdurch eine Verbreiterung des Verzeichnisbaums. Es gibt keine Möglichkeit, eine Verbindung zwischen den einzelnen Blättern im Verzeichnisbaum herzustellen. ZIBDMS stellt jedoch ein Konzept vor, Dateien flexibler in einer benutzerdefinierten Gruppe zu verwalten. Collection ist eine Art von logischem Ordner (siehe Abbildung 4.16), in dem Dateien virtuell organisiert werden können. Eine Collection im ZIBDMS zeichnet sich durch die folgenden Eigenschaften aus:

- Ein globales eindeutiges Objekt, identifiziert sich durch seinen Namen.

- Wird als virtuelles Verzeichnis abgebildet.
- Stellt eine Verbindung zwischen den Dateien her. Dateien mit gemeinsamen Eigenschaften aus verschiedenen Verzeichnissen können in eine Collection zusammengefasst werden. So stehen sie in Relation.
- Wenn eine Operation auf einer Datei in der Collection ausgeführt wird, wird dieselbe Operation bei allen Collectionmembers ausgeführt. Dieser Ansatz steht je nach Semantik für die Anwendung offen. Es macht z. B. im Normalfall beim Löschen einer Datei nicht viel Sinn, alle anderen Mitglieder aus der Collection zu entfernen.

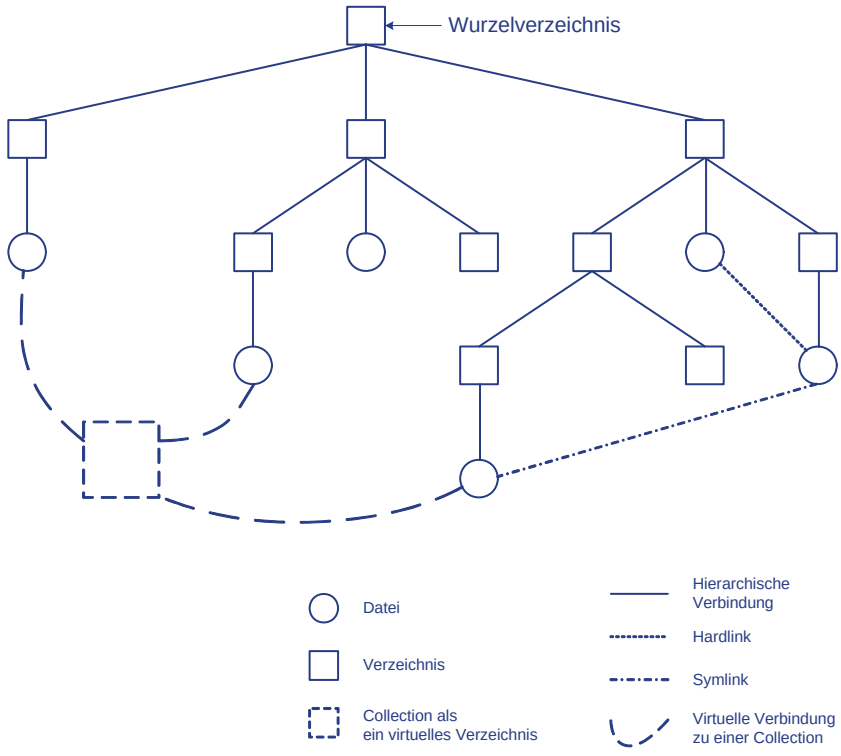

Abb. 4.16 Einbettung einer Collection als virtuelles Objekt ins hierarchische Dateisystem

Da eine Collection als ein virtuelles Verzeichnis abgebildet werden soll, muss man alle Funktionalitäten eines Verzeichnisses bereitstellen. Es werden im Collection-Komponent des Directory-Views entsprechende Methoden implementiert, so dass man mit normalen UNIX-Befehlen die Collections wie gewohnt verwalten kann:

- **Collection anlegen:**

```
> mkdir colnew{}
```

Beim Erzeugen einer Collection wird ein Objekt in dem `mdc` vom Typ `COLLECTION` zuerst angelegt und dann mit dem übergebenen, global eindeutigen Namen zugewiesen. In dem obigen Beispiel wird eine Collection mit dem Namen `colnew` erzeugt, mit der Voraussetzung, dass im ganzen System noch kein Collectionobjekt `colnew` existiert.

- **Collection umbenennen:**

Da eine Collection im hierarchischen Dateisystem als ein virtuelles Verzeichnis und nicht ortsgebunden abgebildet wird, lässt sich keine Verschiebungssemantik modellieren. Stattdessen wird die elementare Operation `rename()` als eine Namensumbenennungsoperation implementiert:

```
void rename(const string& oldpath,const string& newpath);
```

Es wird in der Implementation dieser Funktion anhand des Pfades geprüft, ob die Collection im `oldpath` existiert und sowohl `oldpath` als auch `newpath` Collections sind. Wenn dies der Fall ist, wird das Collectionsobjekt aus dem Katalog geholt und der Name geändert. Auf der Anwendungsebene kann ein normales Kommando diese Semantik ausführen:

```
> mv colnew{} coltest{}
```

- **Mitglieder hinzufügen bzw. verschieben mit einer Verschiebungssemantik:**

Nach der Erzeugung einer Collection sollte eine Möglichkeit geschaffen werden, Dateien dieser Collection hinzufügen zu können. Jede Datei, die der Collection hinzugefügt wird, bekommt ein Attribut

```
IN_COLLECTIONS = collectionname
```

angehängt. Es gab eine Überlegung, dass man die UFIs der Mitglieder als Mehrwerte-Attribut an das Collectionobjekt hängt, damit die Collection einfacher und schneller (weniger Datenbankzugriff bei der Collectionsbearbeitung) zu verwalten ist. Leider wurde der Ansatz nicht weiter verfolgt, weil

auf Dateien häufiger zugegriffen wird und Collection-Semantik einer Datei wichtiger als die Dateisemantik einer Collection ist. Deshalb wird `IN_COLLECTIONS` in der Datei als Referenz zu Collection realisiert, so lässt sich schnell aus der Datei herausfinden, zu welchen Collections sie gehört.

Das Einfügen einer normalen Datei in eine Collection bzw. das Verschieben einer Datei aus einer Collection (Collection-Datei) in eine andere Collection (Collection-Verzeichnis) wird mit der o. g. `rename()`-Funktion realisiert. Durch jedes Argument dieser Funktion kann sowohl eine Datei oder ein Verzeichnis als auch eine Collection-Datei oder ein Collection-Verzeichnis übergeben werden. Die Abbildung 4.17 zeigt eine mögliche Implementierung verschiedener Semantiken des `rename()`-Methodes:

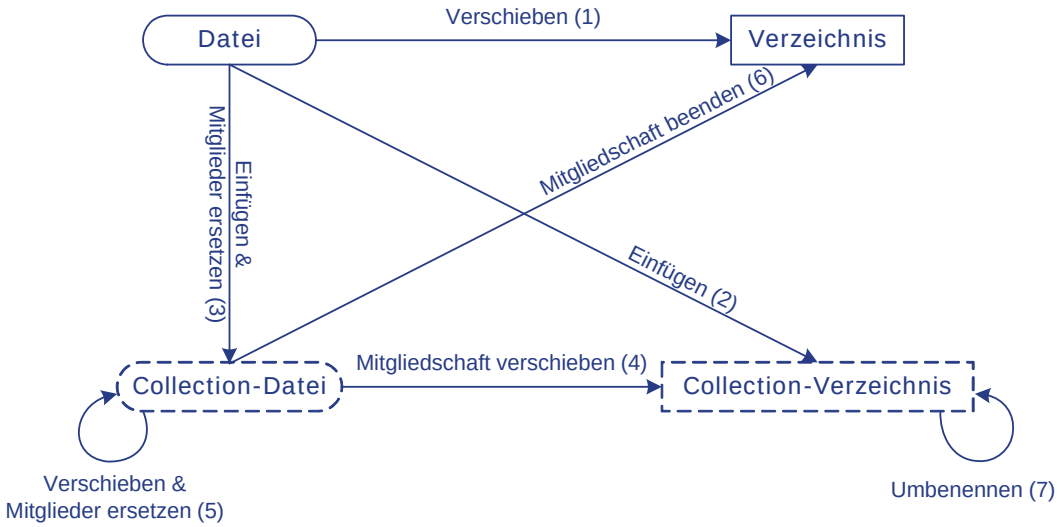

Abb. 4.17 Collection-Verschiebungssemantik

(1) Das Verschieben einer Datei in ein Verzeichnis ist ein normales Verschieben im hierarchischen Dateisystem.

(2) Das Einfügen einer Datei in ein Collection-Verzeichnis modelliert eine Dateiverschiebungssemantik in die Collection. Die Datei wird als Mitglied der Collection hinzugefügt. Es wird aber nur eine virtuelle Instanz der Datei verschoben. Die Datei bleibt an ihrem Ursprungsort erhalten.

```
> mv foo coltest{}
```

(3) Wenn eine Datei in eine existierende Collection-Datei verschoben wird, wird die Datei in die Collection eingefügt und die Mitgliedschaft der existierenden Collection-Datei beendet. Das neue Mitglied ersetzt die Mitgliedschaft der existierenden Collection-Datei.

```
> mv foo coltest{}/\\bar
```

(4) Ein Mitglied einer Collection kann als Mitglied in eine andere Collection wechseln. Die Datei hat dann keine Verbindung mehr zu der alten Collection, stattdessen wird sie in die neue Collection aufgenommen. Bei `rename()` wird nur das Dateiattribut `IN_COLLECTIONS` durch den neuen Collectionsnamen ersetzt.

```
> mv coltest{}/\\foo newcoltest{}
```

(5) Die Umbennenung einer Collection-Datei in eine andere Collection-Datei modelliert eine Ersetzungssemantik. Die zu verschiebende Collection-Datei wird aus ihrer alten Collection entfernt und ersetzt die Mitgliedschaft der neuen Collection-Datei, wenn sie schon existiert. In dem Fall, dass die zu ersetzende Collection-Datei nicht existiert, wird keine Semantik implementiert, weil die Datei dadurch keinen Namen bekommen soll.

```
> mv coltest{}/\\foo newcoltest{}/\\bar
```

(6) Das Entfernen einer Datei aus einer Collection kann man entweder mit einem Löschbefehl wie `rm` oder mit einer Implementierung der Verschiebungssemantik realisieren. Wenn eine Collection-Datei in ein beliebiges reales Verzeichnis verschoben werden soll, wird ihre Mitgliedschaft an der Collection beendet. So wird die Verbindung einer Datei zu einem virtuellen Verzeichnis abgebrochen und quasi der realen Hierarchie zurückgegeben. Das Endergebnis dieser Verschiebungssemantik ist unter anderem Aspekt gleich dem Löschen einer Collection-Datei.

```
> mv coltest{}/\\foo /
```

(7) Collection-Verzeichnisse sind virtuelle Objekte, daher können sie untereinander nicht verschoben werden. Statt dessen wird eine Umbenennungssemantik wie in dem o. g. Abschnitt „Collection umbenennen“ implementiert,

wenn das Zielverzeichnis nicht existiert. Zur Zeit werden Verzeichnisse in einer Collection sowie Collections in einer Collection nicht erlaubt. Deshalb sollte diese Operation keine Wirkung auf ein vorhandenes Collection-Verzeichnis haben.

- **Collections anzeigen bzw. aufrufen:**

  Der Inhalt einer Collection lässt sich mit einem normalen Auflistungsbefehl anzeigen. Intern wird eine elementare Operation `ls()` mit einem Collection-Verzeichnis als Parameter aufgerufen, die eine Abfrage nach allen Dateien mit dem Attribut „`IN_COLLECTIONS=collectionname`“ an den Katalog schickt. Das Ergebnis dieser Abfrage beinhaltet alle Mitglieder der Collection und wird als virtuelle Dateien in dem Collection-Verzeichnis aufgelistet.

```
> cd coltest{}
> ls
\bar     \foo     \testdir\bar
```

  Obwohl Collection-Dateien und Collection-Verzeichnisse virtuelle Objekte sind, ist die Existenz einer Collection in einem realen Verzeichnis spürbar, wenn mindestens eine Datei in diesem Verzeichnis zu der Collection gehört. Es wird eine virtuelle Datei generiert, um mehr semantische Information über eine Datei auszudrücken. Der Name dieser virtuellen Datei gibt Auskunft über die Mitgliedschaft in einer Collection und andere Mitglieder der Collection.

```
> ls
bar     coltest{\bar, \foo, \testdir\bar}      foo     testdir
```

  Collections sind virtuelle globale Objekte, die einzeln mit dem Namen überall aufrufbar sind. Wenn man den Namen einer Collection nicht kennt, kann man ihn aus der vollständigen Liste aller Collections über ein `query-directory` abrufen. Jede Collection hat ein spezielles Attribut `KIND=COLLECTION`, das einen eindeutigen Objekttyp im ZIBDMS kennzeichnet. Deshalb beinhaltet eine Liste aller Collections auch eine Liste aller Objekte vom

Typ `COLLECTION`.

```
> ls KIND==COLLECTION
coltest{}     colname{}
```

- **Mitglieder entfernen:**

  In dem vergangenen Abschnitt wurde bereits erwähnt, dass das Entfernen eines Mitglieds aus der Collection unter verschiedenen Aspekten semantisch unterschiedlich interpretiert werden kann. Mit einem Löschbefehl lässt sich die Verbindung einer Datei zu einer Collection trennen. Dabei wird nur das Attribut „`IN_COLLECTIONS=collectionname`“ aus dem Dateiobjekt gelöscht, die native Datei bleibt weiterhin unverändert.

```
> rm coltest{}/\\foo
```

- **Collection löschen:**

```
> rmdir coltest{}
```

  Das Löschen einer leeren Collection erfolgt ebenfalls intuitiv mit einem Standardkommando. Die Collection wird danach vollständig aus dem `mdc` gelöscht. Ein rekursives Löschen entfernt alle Mitglieder der Collection, also alle Bindungen zwischen der Collection und ihren Dateien und erst zum Schluss wird die Collection gelöscht.

Collection stellt dem Benutzer einen Mechanismus zur Verfügung, Dateien gemeinsamer Eigenschaft explizit in einem logischen Ordner zu verwalten. Durch die Verwendung von Collections kann man sich viele interessante Anwendungen vorstellen. Ein Programm hat normalerweise eine ausführbare Datei, viele Hilfs- und Bibliotheksdateien. Wenn sie in einer Collection sind, können sie auf der Anwendungsebene als einziges Objekt erscheinen, denn für den Benutzer sind sie irrelevant, er braucht nur ein Objekt, um ein Programm ausführen zu können. Eine andere Anwendung kann die Relationseigenschaft einer Collection nutzen. Beispielsweise hat ein wissenschaftliches Dokument mehrere Bestandteile: Bilder, Notizen, Bibliotheksangabe, Hauptdokument im Original und in verschiedenen Veröffentlichungsformaten. Wenn sie alle in einer Collection gebündelt werden, kann man die Bestandteile schnell finden, auch wenn sie sich an verschiedenen Orten befinden.

Eine weitere Anwendung kann auf der Zusammengehörigkeit aller Mitglieder einer Collection basieren. Wenn eine Operation bei einem Mitglied einer Collection ausgeführt wird, wird dieselbe Operation auch bei allen restlichen Mitgliedern durchgeführt. Beispielsweise kann man dadurch sicherstellen, dass alle Mitglieder einer Collection immer über gleiche bestimmten Metadaten verfügen, so dass eine Änderung dieser Metadaten nur einmal bei einem Mitglied vorgenommen werden muss.

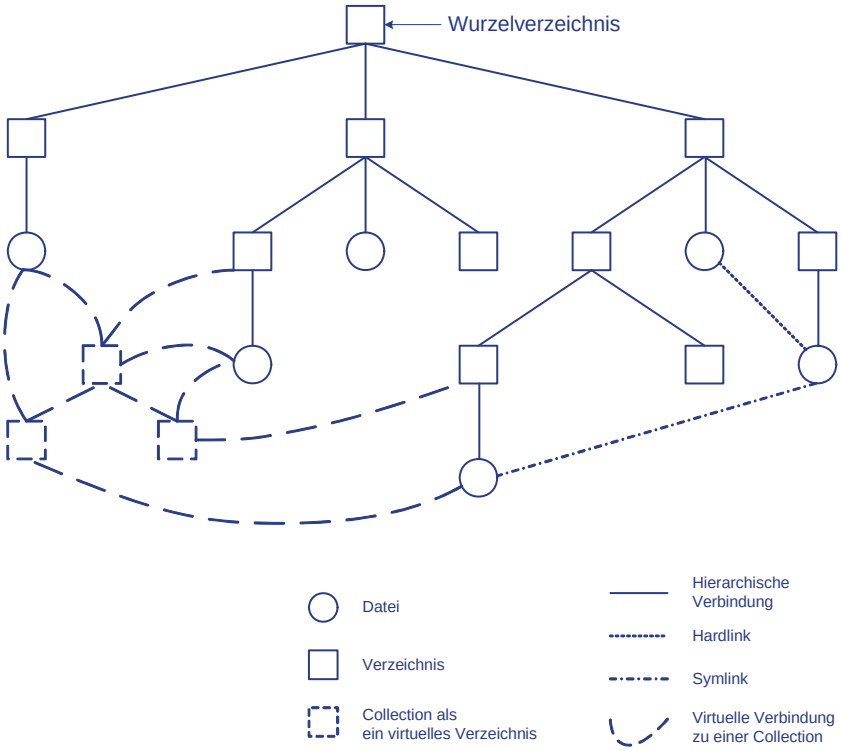

Abb. 4.18 Einbettung von Collections als virtueller Verzeichnisbaum ins hierarchische Dateisystem

Die derzeitige Implementierung lässt aufgrund ihrer Komplexität weder ein Verzeichnis in einer Collection, noch eine Collection innerhalb einer anderen Colletion zu, sonst bilden Collections wie in der Abbildung 4.18 einen virtuellen Verzeichnisbaum auf das hierarchische Dateisystem ab. Hier kann jeder beliebige Knoten des realen Dateisystems mit dem virtuellen Verzeichnisbaum verbunden werden, so dass Überlappungen und Redundanzen berücksichtigt werden müssen. Dies bleibt infolgedessen für die zukünftige Arbeit offen.

4.2.6 dependency-graph-layer

Im Abschnitt 2.1 wurde beschrieben, dass es im hierarchischen Dateisystem keine Möglichkeit gibt, eine Beziehung zwischen Dateien auszudrücken. Dies ist in einem Datenmanagementsystem aber notwendig, da Objekte Verhältnisse zueinander haben und oft untereinander referenzieren. Das Problem wurde in [Schi04b] erkannt und entsprechend eine Lösung, das sog. Arbitrary-Relation-Konzept vorgestellt, auf dem eine Implementierung als `dependency-graph` im ZIBDMS basiert. Fast zeitgleich wurde interessanterweise ein ähnliches Konzept als `link` im Design des LiFS[24] [ABB+05] untergebracht. LiFS befindet sich im Anfangsstadium und wurde als eine Erweiterung des traditionellen Dateisystems entworfen. Die Metadaten im LiFS werden in einer relationalen Datenbank gespeichert und können mit der RDF-Abfragesprache[25] ausgelesen werden.

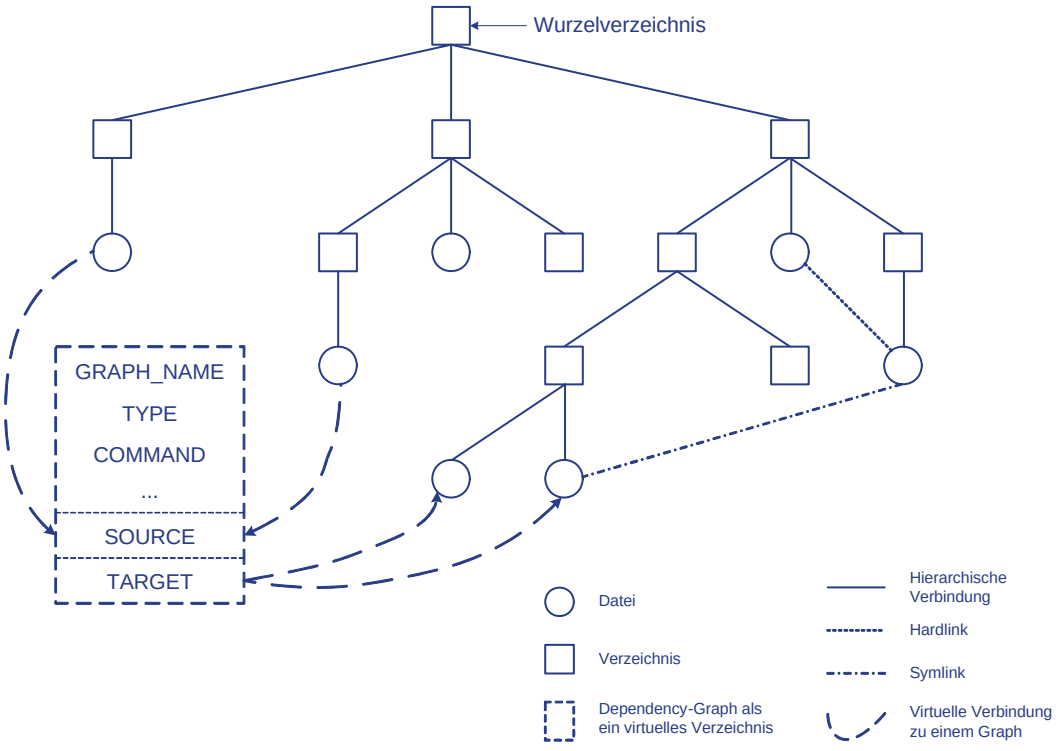

Abb. 4.19 Einbettung eines Dependency-Graphs als virtuelles Objekt ins hierarchische Dateisystem

24. **Li**nking **F**ile **S**ystem
25. http://www.w3.org/RDF/

Die Abbildung 4.19 zeigt die Einbettung eines Dependency-Graphs als virtuelles Verzeichnis in den Verzeichnisbaum. Ein Dependency-Graph-Objekt hat folgende Merkmale:

- Ein globales eindeutiges Objekt und identifiziert sich durch seinen Namen wie `collection` im ZIBDMS.
- Wird als virtuelles Verzeichnis abgebildet.
- Ein Graph-Objekt besteht aus einer Quelle (source), einem Ziel (target) und Attributen. Eine Quelle oder ein Ziel kann mehrere Dateien enthalten und Attribute sind normale ZIBDMS-Attribute. Zu den Attributen gehören optional folgende Attribute:
  - `COMMAND` kann angegeben werden, mit welchem Kommando und dazugehörigen Parametern man aus den Quelldateien die Zieldateien erzeugt.
  - `DG_TYPE` lässt sich die Art der Relation wie `cited_by`, `referenced_by`, `InProceedings` usw. zuweisen.

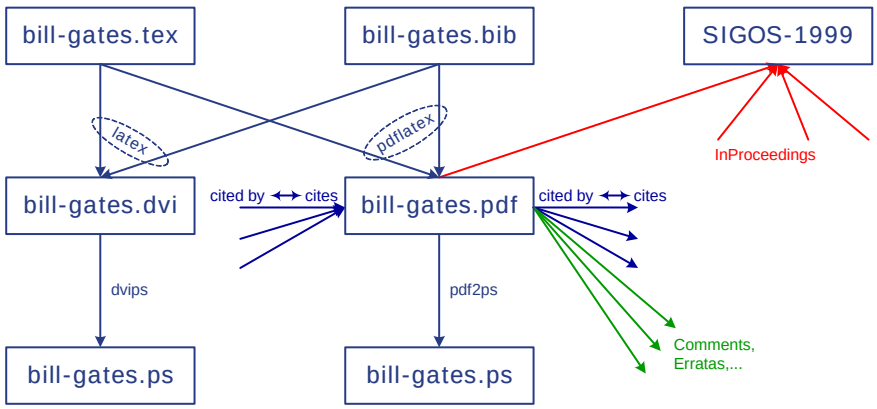

Abb. 4.20 Eine Beispielanwendung von Arbitrary Relations [Schi04b]

Wie oben erwähnt, dient ein Dependency-Graph-Objekt dazu, Dateien in eine Relation zu stellen. Mehrere Graph-Objekte bilden eine Linkstruktur in Form eines gerichteten Dependency-Graphs. In diesem Graph sind Zyklen möglich, weil Dateien untereinander referenziert werden können. Die Abbildung 4.20 präsentiert ein Anwendungsbeispiel, in dem die Erzeugung von Dateien aus mehreren

Dateien einen Entstehungsgraph bildet. Beachtenswert steht die Datei „`bill-gates.pdf`“ im Zentrum, die aus den Quellen „`bill-gates.tex`“ und „`bill-gates.bib`“ durch das Kommando `pdflatex` erzeugt wird. In diesem Fall kann man das Graph-Objekt ebenfalls mit einem eindeutigen Namen `pdflatex` vergeben. Diese Publikation wird mit der Zeit von anderen Veröffentlichungen zitiert und kommentiert. Auch diese Verbindungen lassen sich mit Graph-Objekten herstellen, so dass eine übersichtliche graphische Darstellung wie in der Abbildung 4.21 möglich wird.

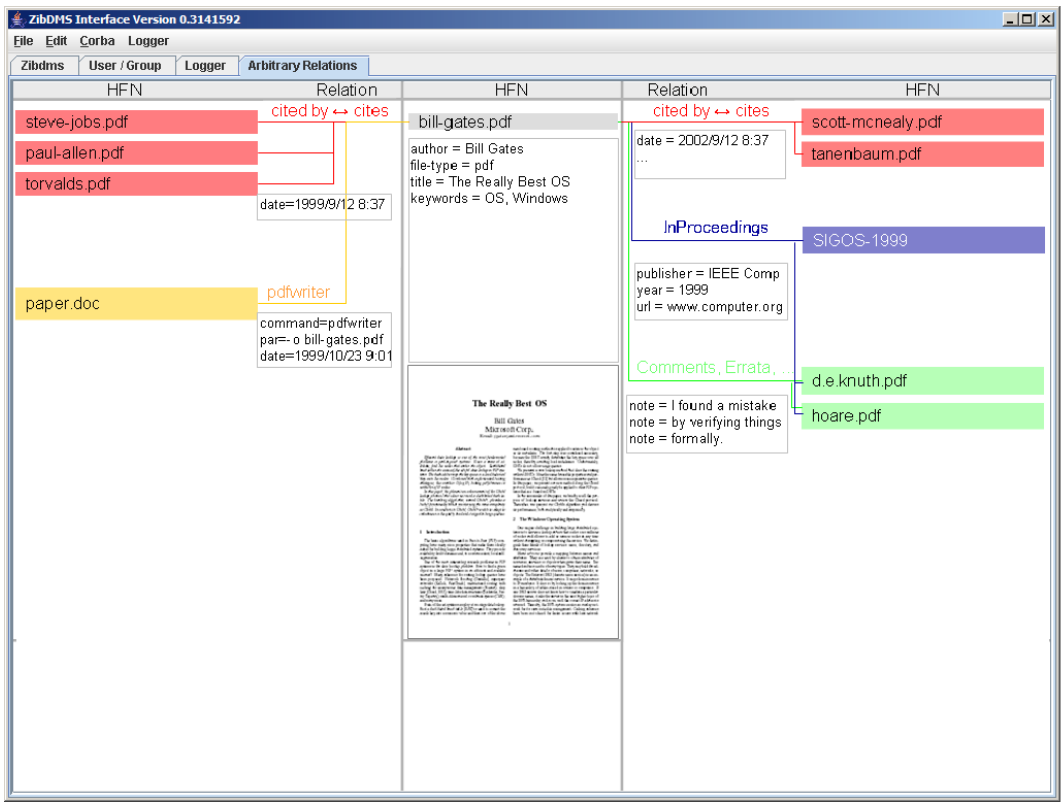

Abb. 4.21 Ein zu realisierendes Beispiel von Arbitrary Relations über ZIBDMS-GUI

Die durch den Dependency-Graph entstehende Linkstruktur kann als Grundlage für das PageRank-Konzept (im Abschnitt 2.2.6.1 genauer beschrieben) dienen, wodurch die Kanten gewichtet sind. Der Relevanzwert einer Publikation wird aus Gewichtungen aller Zitate (eingehende Links) summiert. Dieser Wert wird weiter an die in der Publikation referenzierten anderen Veröffentlichungen (ausgehende Links) gleichmäßig verteilt. So wird die Popularität einer

Veröffentlichung mit einem relativen Wert gemessen.

Zur Behandlung vom Dependency-Graph stellt das `dependency-graph-layer` folgende Befehle zur Verfügung:

- **Dependency-Graph anlegen:**

```
> mkdir newgraph..dg
```

Ein Dependency-Graph wird im Verzeichnisbaum als virtuelles Objekt eingebettet, so dass die Erstellung eines Dependency-Graphs gleich der Erzeugung eines virtuellen Verzeichnisses ist. Es wird ein Objekt in den `mdc` vom Typ `DEP_GRAPH` mit einem eindeutigen Namen eingetragen, sofern im ganzen System noch kein gleichnamiges Dependency-Graph-Objekt existiert.

- **Dependency-Graph umbenennen:**

Zur Namensumbenennung eines Graphen wird die elementare Operation `rename()` wie bei einer Collection aufgerufen. Auf der Kommandozeile wird diese Funktion analog mit einem Verschiebungsbefehl realisiert:

```
> mv newgraph..dg dgname..dg
```

- **Quelldateien / Zieldateien hinzufügen:**

Jeder Graph besteht aus einer Quelle, einem Ziel und Attributen. Eine Quelle oder ein Ziel kann mehrere Dateien enthalten. Sie werden deswegen als vordefinierte Unterverzeichnisse des virtuellen Dependency-Graph-Verzeichnisses („`src`“ für Quelle und „`dest`“ für Ziel) untergebracht. Zum Hinzufügen von Dateien in dieses Verzeichnis auf der Kommandozeile gibt es keinen eigenen Befehl. Deshalb wird diese Semantik mit folgendem `mv`-Befehl ausgeführt, der eigentlich dem Verschieben von Dateien dient:

```
> mv foo dgname..dg/src
> mv bar dgname..dg/dest
```

Abhängig von der Quelle oder dem Ziel wird jede Datei in dem Dependency-Graph-Objekt entsprechend mit dem Attribut `SRC` oder `DEST` vermerkt. Dabei wird das UFI der Datei dem Wert dieses Attributes zur Identifikation zugewiesen.

- **Dependency-Graph anzeigen bzw. aufrufen:**

  In der hierarchischen Sicht kann man Dateien mit ihren normalen Attributen anzeigen. Aus der Verzeichnisauflistung werden keine Informationen sichtbar, ob eine Datei zu einem Dependency-Graph gehört und wenn ja, zu Quelldateien oder Zieldateien? Aus diesem Grund wird eine virtuelle Datei zu jeder nativen Datei erzeugt, die zu einem Graph gehört, um mehr semantische Information über die Datei auszudrücken:

```
> ls

main.bib           main.dvi               main.tex
main.bib@dgname    main.dvi@dgname.dest   main.tex@dgname.src
```

  In der obigen Abbildung kann man erkennen, dass die Datei „`main.tex`“ zu den Quelldateien und „`main.dvi`“ zu den Zieldateien des „`dgname`“-Graphs gehören. Die Datei „`main.bib`“ gehört sowohl zu Quelldateien als auch Zieldateien. Um eine vollständige Liste der Quelldateien bzw. Zieldateien zu bekommen, wechselt man in das virtuelle Graph-Verzeichnis:

```
> cd dgname..dg

> ls -la

total 3
lrw-r----- 1 root root 65534 Mai 24 20:18 dest -> dgname..dgdest
lrw-r----- 1 root root   632 Mai 24 20:18 info -> dgname..dgattr
lrw-r----- 1 root root 65534 Mai 24 20:18 src -> dgname..dgsrc
```

  In diesem Verzeichnis werden symbolische Links auf Quellverzeichnis, Zielverzeichnis und Graph-Attributdatei aufgelistet. Wenn man beispielsweise in das „`src`“-Verzeichnis geht, sind alle Quelldateien zu sehen:

```
> cd src

> ls -la

total 2
-rw-r-----  1 root root  5534 Mai 24 20:18 \main.bib
-rw-r-----  1 root root 98534 Mai 24 20:18 \main.tex
```

  Ein Dependency-Graph ist ein virtuelles globales Objekt und wird nicht im Verzeichnisbaum angezeigt. Wie bei einer Collection hat jedes Graph-Objekt ein spezielles Attribut „`KIND=DEP_GRAPH`“ zur Kennzeichnung im

System. Man kann diese Eigenschaft nutzen und eine Abfrage nach dem Attribut veranlassen, um die Liste aller Graph-Objekte zu bekommen.

```
> ls KIND==DEP_GRAPH
dgtest..dg     dgname..dg
```

- **Quelldateien / Zieldateien entfernen:**

  Um Dateien aus der Quelle oder aus dem Ziel zu entfernen, kann man sich eines normalen Löschbefehls bedienen. Intern wird die UFI der zu entfernenden Datei entsprechend aus der Werteliste des Attributs SRC oder DEST im Katalog gelöscht.

```
> rm dgname..dg/src/foo
> rm dgname..dg/dest/bar
```

- **Dependency-Graph löschen:**

```
> rmdir dgname..dg
```

  Beim Löschen eines Dependency-Graphs wird das Graphobjekt vollständig aus dem mdc gelöscht. Es muss bei Quelldateien und Zieldateien nicht ändern, da alle Verbindungsinformationen zu ihnen im Graphobjekt gespeichert werden.

4.3 Vergleich mit anderen Systemen

Im Kapitel 3 wurden die Eigenschaften und Funktionalitäten SFS, XMLFS, Spotlight und CFS beschrieben. Da jedes dieser Systeme für einen Aspekt des metadatenbasierten Systems ausgewählt wurde, hat es eigene Stärken sowie Schwächen. Beispielsweise beschränkt sich Spotlight nur auf eine Desktop-Lösung. Spotlight kann zwar Index- und Metadaten zu Dateien auf entfernten Rechnern sammeln aber kein anderer Rechner kann diese Daten nutzen, weil die Spotlight-Datenbanken nicht verteilt sind. Die Tabelle 4.6 zeigt solche Schwächen und Stärken aller Systeme im Überblick:

Systeme / Funktionalität	SFS	XMLFS	Spot-light	CFS	ZIBDMS
Klassifikation	erweitert	erweitert	integriert	erweitert	erweitert
Schnittstelle	NFS	NFS	beliebig	CFS	NFS, CORBA, SOAP
Art der Datenbank	BTree	XML Repo-sitory	k. A.	XML-Space Repository	storagebox, sqlite, mysql
Verteilung der Datenbank	nein	nein	nein	nein	ja, in Arbeit
Umgebungskontext	nein	nein	nein	ja	nein
Metadatenerfassung	explizit	explizit	automatisch	nein	manuel
Manuelles Hinzufügen von Metadaten	nein	k. A.	nur als Stichwörter	nein	ja
Indizierung	explizit	explizit	automatisch	nein	nein
Attribut mit mehreren Werten	nein	k. A.	ja	nein	ja
Wild-Cards	nein	Rechtstrun-kierung	ja, mit Fuzzy-Suche	nein	Rechtstrun-kierung
Vergleichsoperatoren	=	{=, >, <}	alle	=	alle außer „!“
Logische Operatoren	AND	AND, OR	AND, OR, NOT	AND	AND, OR
Logische Gruppierung	nein	nein	ja	nein	ja
Volltextsuche	ja	ja	ja	nein	nein
Ergebnissortierung	nein	ja	ja	nein	nein
Ergebnisgruppierung	nein	nein	ja	nein	nein
Direkter Zugriff auf die Ergebnisse	ja	ja, im GUI	nur im GUI	ja	ja
Graphische Darstellung	nein	ja	ja	über GUI	über GUI
Query-Speicherung	nein	nein	nur im GUI	nein	ja
Virtuelles Verzeichnis	virtuelles Verzeichnis	virtuelles Verzeichnis	intelligenter Ordner	Kontextver-zeichnis	query-direc-tory
Andere virtuelle Objekte	nein	nein	nein	nein	ja
Konsistenter Verweis	nein	nein	teilweise mit alias	nein	Weak-Link
Logischer Ordner	nein	nein	nein	nein	Collection
Relationsunterstützung	nein	nein	nein	nein	Dependency-Graph

Tab. 4.6 ZIBDMS im Vergleich zu anderen Systemen

Aus der Tabelle kann man vieles über ZIBDMS im Vergleich zu anderen Systemen erkennen. Die Aufmerksamkeit liegt hier auf der fehlenden Indizierung und manuellen Metadatenerfassung des ZIBDMS. Es ist auch zu beachten, dass die Tabelle nur die Angaben über Systemfunktionalitäten beinhaltet. Ein Leistungsvergleich ist aufgrund der fehlenden vergleichbaren Daten nicht möglich.

5 Performance und Ergebnisbewertung

Im letzten Kapitel wurde die Einbettung neuer Verwaltungsmethoden in die hierarchische Sicht des ZIBDMS ausführlich beschrieben. Es wurde jedoch nicht behandelt, wie die neuen Methoden sich auf die Leistung des ZIBDMS auswirken. In diesem Kapitel werden mehrere Leistungsmessungen im Directory-View durchgeführt, um das Leistungsvermögen neuer Objekte mit den bekannten Objekten zu vergleichen.

Die Messungen laufen auf einem Rechner mit dem Betriebssystem SuSE Linux 9.3 und der Datenbank `storagebox` [Hupf04]. Der Rechner verfügt über einen Pentium 4 Prozessor bei einer Taktfrequenz von 3 GHz, 2 GB Arbeitsspeicher und eine SATA-Festplatte mit einer Cache-Größe von 8 MB und einer Umdrehungsgeschwindigkeit von 7.200 rpm. Dabei wird die durchschnittliche verbrauchte CPU-Zeit[1] ermittelt, damit Zeiten anderer quasi parallel laufender Prozesse nicht in die Zeitmessung mit einfließen. Die Rohdaten der Leistungsmessung ist den Tabellen im Anhang zu entnehmen.

Da Objekte im ZIBDMS in einer Datenbank gespeichert sind, wurde die vier folgenden Operationen ausgewählt, die entsprechende Datenbankoperationen im Metadatenkatalog hervorrufen:

1. Die Laufzeitmessung nähert sich bei vielen Messungen der ermittelten CPU-Zeit an.

Einfügeoperation

Es werden in eine leere Datenbank schrittweise 1000 Objekte eingefügt. Für native Objekte werden verschiedene Messungen für die Erzeugung von Dateien, Verzeichnissen oder symbolischen Links durchgeführt. Weak-Links sind auch native Objekte, werden aber mit eigenen Zeiten dargestellt, um einen Vergleich mit den normalen Dateien zu ermöglichen. Für virtuelle Objekte wird die Zeit der Erzeugung von Collections oder Dependency-Graphs gemessen.

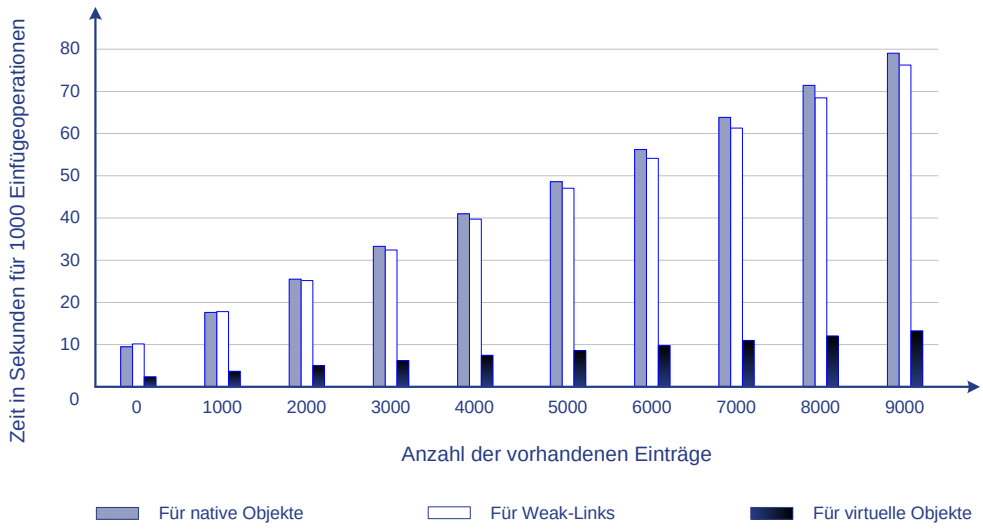

Abb. 5.1 Zeitsmessung im Directory-View für das Einfügen von 1000 Objekten

Es ist in der Abbildung 5.1 zu erkennen, dass die Erzeugung von Weak-Links genau so lang wie die Erzeugung von anderen nativen Objekten dauert. Bemerkenswert sind die ein Fünftel kürzeren Zeiten bei virtuellen Objekten. Zum einen wird nur ein UfiObjekt mit weniger Attributen für ein virtuelles Objekt in die Datenbank eingetragen, zum anderen wird die Modifikationszeit des übergeordneten Verzeichnisses nicht aktualisiert, weil virtuelle Objekte zu keiner Hierarchie gehören. Dadurch lässt sich die höhere Geschwindigkeit erklären.

Auflistungsoperation

Bei der Auflistungsoperation werden Objekte in einem Verzeichnis aufgelistet. Zum Vergleich werden native Dateien in einem nativen Verzeichnis aufgelistet;

dateien in verschiedenen Verzeichnissen mit dem RangeQuery „`ls("/NAME==*")`“ abgefragt; Collections oder Dependency-Graphs werden anhand des Attributs `KIND` wie z. B. `ls("/KIND==DEP_GRAPH")` aufgelistet. Das Ergebnis in der Abbildung 5.2 zeigt bessere Werte für die Auflistung mit einer Abfrage. Der Grund liegt hier bei den semantischen Informationen, die bei einer normalen Verzeichnisauflistung angezeigt werden sollen und bei virtuellen Objekten zum großen Teil mit Defaultwerten belegt sind, die nicht aus der Datenbank geholt werden müssen. Es werden die Ergebnisse der Unterkomponenten `hierarchical`, `collection` und `dependency-graph` in das Endergebnis einbezogen.

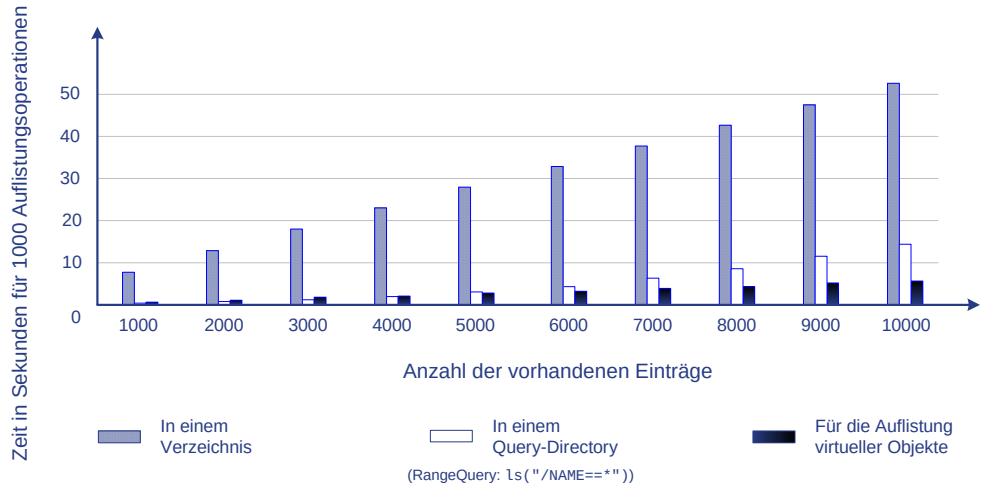

Abb. 5.2 Zeitsmessung im Directory-View für die Auflistung von 1000 Objekten

Verschiebeoperation

Es werden schrittweise Dateien erzeugt und in ein Verzeichnis, ein Dependency-Graph oder eine Collection verschoben. So kann das Verschieben einer Datei in ein natives Verzeichnis und in einen virtuellen logischen Ordner verglichen werden.

Bei der Betrachtung der Werte in der Abbildung 5.3 fällt auf, dass die Verschiebung in ein Verzeichnis wesentlich langsamer ist. Ähnlich wie bei der Verzeichnisauflistung werden bei einer Dateiverschiebung die Aktualität der Weak-Links und Collectionsanzeige im Verzeichnis zusätzlich aktuell gehalten. Es ist hier auch zu beachten, dass die Aktualisierung der Modifikationszeit des

übergeordneten Verzeichnisses einige Zeit in Anspruch nimmt. Beim Hinzufügen von Dateien in einem Graph, entspricht die Zuweisung des UFI-Werts einer Datei in `SRC` oder `DEST`, verursachte die `Storagebox`-Datenbank bei mehr als 2000 Objekte ein Systemabsturz. Die Datenbank hat vermutlich ein Problem mit der Verwaltung von einem Attribut mit mehr als 2000 Werten.

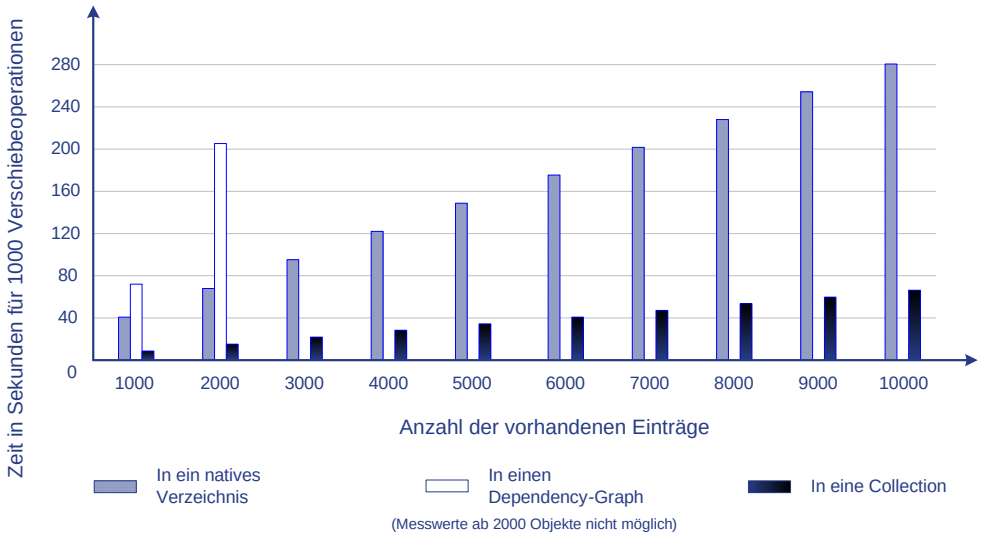

Abb. 5.3 Zeitsmessung im Directory-View für das Verschieben von 1000 Objekten

Löschoperation

Das Löschen von Dateien, Weak-Links oder virtuellen Objekten wird ebenfalls mit dem gleichen Verfahren durchgeführt. Zu Beginn waren in der Datenbank 10000 Objekte gespeichert, die verbrauchte CPU-Zeit für die Löschung von wiederholten 1000 Objekten nimmt linear (siehe Abbildung 5.4) ab. Hier werden auch virtuelle Objekte wie Collections oder Dependency-Graphs, die jeweils nur ein UfiObjekt im Katalog haben, schneller aus der Datenbank ausgetragen. Außerdem muss die Modifikationszeit des übergeordneten Verzeichnisses nicht aktualisiert werden.

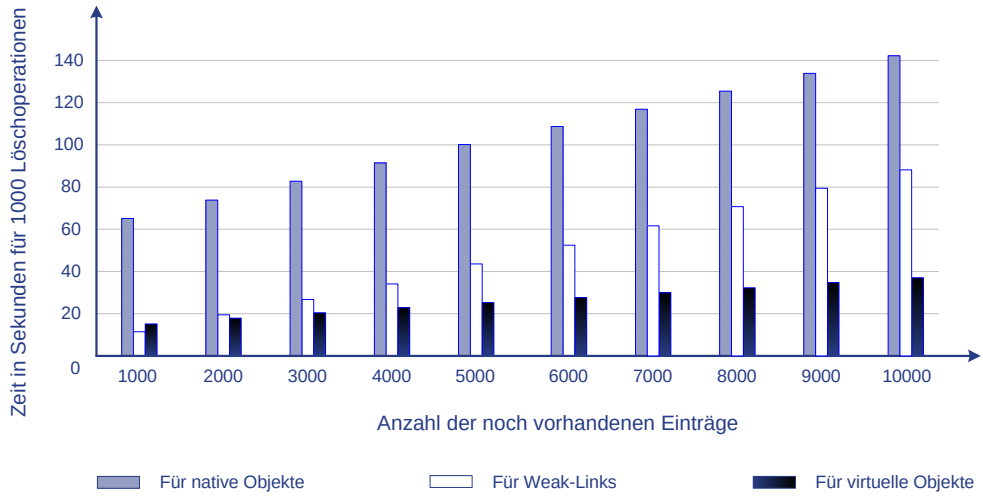

Abb. 5.4 Zeitsmessung im Directory-View für das Löschen von 1000 Objekten

Die oben durchgeführten Messungen haben gezeigt, dass die Einführung der neue Datenverwaltungsmethoden in die hierarchische Dateisystemsicht keinen großen Einfluss auf die Leistung des ZIBDMS. Beim Vergleich haben Operationen auf neue Objekte mit besseren Werten abgeschnitten.

6 Zusammenfassung und weitere Entwicklungen

Das Ergebnis der Studie „How much information? 2003“ [LyVa03] ist eine Bestätigung dafür, dass die Menge an Daten, Informationen und Dokumenten in den letzten Jahren weltweit weiterhin mit hohen Zuwachsraten zunimmt. Der weitaus größte Teil dieser Daten wird in magnetischen oder optischen Datenträgern gespeichert, die üblicherweise mit der hierarchischen Struktur eines Dateisystems organisiert werden. Im Zusammenhang mit dem Zuwachs an Daten wurden Einschränkungen dieser Organisationsform diskutiert: Dateien werden oftmals an zwei oder mehr Stellen in der Hierarchie untergebracht; aufgrund des hierarchischen Aufbaus wird das Suchen nach Dateien mit einem bestimmten Inhalt komplexer und eine Reorganisation von Dateien ist in der Regel schwierig. Hinzu kommen weitere Mängel an Dateiverwaltungsmethoden, so dass mehrere Assoziationen zwischen Dateien nicht möglich sind und Dateien sich nicht in eine Relation unter der Berücksichtigung des Kontextes stellen lassen.

Als eine Erweiterung des klassischen hierarchischen Dateisystems wurde das metadatenbasierte Dateisystem vorgestellt, in dem Index- und Metadaten als Grundlage für unterschiedliche Suchmethoden und flexible Zugriffsmechanismen dienen. Es wurde das Konzept des virtuellen Verzeichnisses erweitert, so dass neue Verwaltungsmethoden mit virtuellen Objekten in die bestehende hierarchische Dateisystemsicht eingebettet werden können. Es ist somit auch möglich, das Ergebnis einer Suche in der Hierarchie zu organisieren und eine übersichtliche Darstellung für einen assoziativen Zugriff bereitzustellen.

Im Kapitel 3 wurden einige Systeme vorgestellt, welche besondere Aspekte des metadatenbasierten Dateisystems repräsentieren. Die ursprüngliche Idee dieses neuartigen Dateisystems wurde mit dem Projekt Semantic-File-System am MIT realisiert. Es werden Indexdaten und Metadaten aus Dateien extrahiert, die mit einer einfachen logischen Systax über die NFS-Schnittstelle abgefragt werden können. Für die Ergebnisdarstellung wurde das virtuelle Verzeichnis konzipiert, in dem Ergebnisse mit symbolischen Links zu den eigentlichen Dateien aufgelistet werden.

XMLFS verwendet dieses Konzept, um XML-Dokumente für einen effektiven Zugriff über eine Abfrage bereitzustellen. Dies ermöglicht beispielsweise unter der Verwendung vom Windows Explorer, den Zugang an XML-Dokumente im XML-Dateisystem über die NFS-Schnittstelle zu schaffen. Im Verzeichnisbaum kann man in dem in der hierarchischen Sicht eingebetteten virtuellen Verzeichnis mit Hilfe von spezifischen DTDs navigieren.

Spotlight ist der Vertreter von dem integrierten Ansatz eines metadatenbasierten Systems und verwaltet die Indexdaten und Metadaten getrennt, die bei jeder Dateiänderung auf der Betriebssystemebene automatisch aktualisiert werden. Die innovative Suchtechnologie von Apple bietet eine Kombination aus einer Volltextsuche und einer logischen Syntax in einem einfachen Suchfeld. Das Ergebnis wird in einem überschaubaren intelligenten Ordner mit verschiedenen sinnvollen Darstellungsoptionen organisiert. Um eine immer aktuelle Sicht der Dateien zu erhalten, kann man diesen Ordner in der Verzeichnishierarchie abspeichern.

Das Context-File-System wurde für eine ereignisbasierte Kommunikations-Middleware-Plattform entwickelt, in der ein Kontext für einen interaktiven Raum mit Hilfe von Kontextregeln definiert werden kann. Dateien in dieser kontextbasierten Umgebung werden in Kontextverzeichnissen zur Verfügung gestellt, auf die ebenfalls als virtuelles Verzeichnis über einen Mount-Point auf einem Mount-Server zugegriffen werden kann.

Der praktische Teil dieser Arbeit begann im Kapitel 4 mit der Implementierung neuer Verwaltungsmethoden für die hierarchische Sicht des ZIBDMS. Das ZIBDMS ist ein Datenmanagementsystem, in dem Metadaten in Form von

Attribut-Wert-Paaren im verteilten Metadaten- und Dateireplikationskatalog verwaltet werden. Für diese Arbeit wurden mehrere ZIBDMS-Komponenten benötigt, die im Abschnitt „Design des ZIBDMS“ (Seite 63) mit dem neuen Dateikonzept kurz beschrieben wurden. Anschließend wurde auf das `directory-view` mit seinen Unterkomponenten im Detail eingegangen. Als erste Unterkomponente wurde das `attribute-file-layer` vorgestellt, auf dem das Lesen einer virtuellen Datei sowie das Problem mit dem Schreiben einer virtuellen Datei über die mit UDP implementierte NFS-Schnittstelle ausführlich behandelt wird. Um das Problem zu lösen, wurde in dieser Arbeit einen Algorithmus für die Behandlung der zufällig ankommenden Schreibpuffer entwickelt. In dieser Ebene wird auch das Eintragen bzw. Entfernen einzelner Metadaten mit virtuellen Operationen über die Kommandozeile erlaubt, so dass eine Automatisierung mit den unveränderten Kommandos möglich ist.

Weak-Links wurden in dieser Arbeit konzipiert, um die Aktualität- und Konsistenzprobleme eines symbolischen Links zu beheben sowie einer Verklemmung vorzubeugen. Das `weak-link-layer` stellt außerdem weitere Informationen über Verweise zur Verfügung. Im ZIBDMS ist es möglich, alle Verweise (Symlinks, Hardlinks und Weak-Links) zu einem Objekt in einem virtuellen Verzeichnis aufzulisten und im Verweisbaum zu navigieren bzw. über die Liste der Unterverweise zu informieren.

Die Abfrage mit der logischen Syntax im ZIBDMS wird in der Ebene `query-directory` behandelt. Zuerst wird die Abfrage mit dem Inline-Parser Spirit geparst und dann werden entsprechende Abfragen an den Metadatenkatalog ausgeführt. Das Ergebnis wird in einem virtuellen Verzeichnis angezeigt. In Abhängigkeit von der Anzahl der Ergebnisse werden Resultate in einem der zwei Darstellungsmodi präsentiert.

Die Beziehung zwischen Dateien wird im ZIBDMS mit Collections und Dependency-Graphs realisiert. Eine Collection fasst Dateien in einem logischen Ordner zusammen, und mit einem Dependency-Graph lassen sich Dateien in eine Relation stellen. In `collection-layer` und `dependency-graph-layer` werden

das Konzept und grundlegende Operationen für die Verwaltung zur Verfügung gestellt, auf denen sinnvolle Anwendungen aufgebaut werden können.

Die in der Motivation (Kapitel 1) angesetzten Ziele wurden in dieser Arbeit verfolgt und verwirklicht. Darüber hinaus konnten im Rahmen dieser Arbeit folgende weitere Optimierungs- und Erweiterungsmöglichkeiten festgestellt werden:

- Zur Zeit werden Metadaten ins ZIBDMS manuell eingetragen. Eine angemessene Indizierung und eine automatische Metadatenerfassung (siehe Kapitel 2) würden die Nutzbarkeit des ZIBDMS in der Praxis erhöhen.
- Im Moment benutzt die NFS-Implementierung UDP als Transportprotokoll. Eine Unterstützung von TCP und eine Implementierung des WebNFSs würden den Einsatz des ZIBDMS weiter verbreiten.
- Für ein effektives Suchen sollte die Abfrage unter Berücksichtigung der Speicherungsstruktur optimiert werden. Bei einer großen Anzahl der Ergebnisse wären weitere Techniken wie Ranking (siehe Abschnitt 2.2.6) für die Darstellung sehr hilfreich.
- Am Ende des Abschnitts 4.2.5 wurde die Erweiterung von Collection erwähnt. Eine Unterstützung von Verzeichnissen oder Collections in einer Collection könnte in Betracht gezogen werden.

Im letzten Kapitel wurden Leistungsmessungen von eingeführten Objekten im ZIBDMS vorgenommen. Elementare Operationen wie Einfügen, Auflisten, Verschieben und Löschen wurden einzeln im `directory-view` gemessen. Die Auswertung dieser Ergebnisse zeigte, dass neue Verwaltungsmethoden die Leistung des ZIBDMS im Allgemein nicht beeinträchtigen.

A Rohdaten der Leistungsmessung

A.1 Messung der Einfügeoperation

Einträge	Zeit(s) für native Objekte	Zeit(s) für Weak-Links	Zeit(s) für virtuelle Objekte
0	9,45	10,14	2,36
1000	17,63	17,82	3,69
2000	25,53	25,15	5,02
3000	33,26	32,43	6,26
4000	41,02	39,71	7,43
5000	48,64	47,02	8,56
6000	56,25	54,11	9,78
7000	63,85	61,29	10,98
8000	71,44	68,46	12,06
9000	79,06	76,23	13,27

A.2 Messung der Auflistungsoperation

Einträge	Zeit(s) für native Objekte	Zeit(s) für Query-Directory	Zeit(s) für virtuelle Objekte
1000	7,68	0,56	0,52
2000	12,83	0,73	1,08
3000	17,93	1,17	1,74
4000	22,97	1,93	2,04
5000	27,89	3,02	2,76
6000	32,81	4,29	3,19
7000	37,70	6,32	3,88
8000	42,60	8,52	4,28
9000	47,42	11,52	5,15
10000	52,57	14,35	5,61

A.3 Messung der Verschiebeoperation

Einträge	Zeit(s) für native Objekte	Zeit(s) für Dependency-Graphs	Zeit(s) für virtuelle Objekte
1000	40,74	71,82	8,59
2000	67,80	205,24	15,04
3000	95,12		21,57
4000	122,05		28,13
5000	148,83		34,35
6000	175,32		40,76
7000	201,58		47,17
8000	227,91		53,40
9000	254,11		59,69
10000	280,50		66,06

A.4 Messung der Löschoperation

Einträge	Zeit(s) für native Objekte	Zeit(s) für Weak-Links	Zeit(s) für virtuelle Objekte
1000	65,04	11,35	15,11
2000	73,83	19,47	17,79
3000	82,68	26,69	20,36
4000	91,43	34,06	22,82
5000	100,13	43,60	25,28
6000	108,60	52,44	27,60
7000	116,92	61,58	29,94
8000	125,48	70,64	32,27
9000	133,82	79,44	34,69
10000	142,16	88,11	37,01

Literaturverzeichnis

[ABB+05] A. Ames, N. Bobb, S. A. Brandt, A. Hiatt, C. Maltzahn, E. L. Miller, A. Neeman und D. Tuteja. Richer file system metadata using links and attributes. In: *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)*, Monterey, CA, April 2005.

[AFK+02] A. Azagury, M. Factor, N. Kraus, I. Loy und B. Mandler. Index Infrastructure for an XML repository. In: *ACM SIGIR 2002 XML and IR workshop notes*, Finland, August 2002.

[AnSp04] S. Androutsellis-Theotokis und D. Spinellis. A survey of peer-to-peer content distribution technologies. In: *ACM Computing Surveys (CSUR)*, Volume 36 , Issue 4, pp. 335-371, Dezember 2004.

[Appl05a] Inc. Apple Computer. Spotlight: Find anything on your Mac instantly. Technology Brief - Mac OS X: Spotlight, Inc. Apple Computer, April 2005.

http://images.apple.com/macosx/pdf/MacOSX_Spotlight_TB.pdf
(Zugriff am 30. 07. 2005)

[Appl05b] Inc. Apple Computer. Introduction to Spotlight Query Programming Guide. Inc. Apple Computer, April 2005.

http://developer.apple.com/documentation/Carbon/Conceptual/SpotlightQuery/SpotlightQuery.html
(Zugriff am 30. 07. 2005)

[Appl05c] Inc. Apple Computer. MDQuery Reference. Inc. Apple Computer, Juni 2005.

http://developer.apple.com/documentation/Carbon/Reference/MDQueryRef/MDQueryRef.pdf
(Zugriff am 30. 07. 2005)

[Appl05d] Inc. Apple Computer. Spotlight Metadata Attributes Reference. Inc. Apple Computer, Juni 2005.

http://developer.apple.com/documentation/Carbon/Reference/MetadataAttributesRef/MetadataAttributesRef.pdf
(Zugriff am 30. 07. 2005)

[Appl05e] Inc. Apple Computer. Predicates Programming Guide. Inc. Apple Computer, Juli 2005.

http://developer.apple.com/documentation/Cocoa/Conceptual/Predicates/Predicates.pdf
(Zugriff am 30. 07. 2005)

[AzFM00] A. Azagury, M. Factor und B. Mandler. XMLFS: an XML-aware file system. In: *ACM SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, Juli 2000.

[AFMM02] A. Azagury, M. Factor, Y. Maarek und B. Mandler. A Novel Navigation Paradigm for XML Repositories. In: *Journal of the American Society for Information Science and Technology*, vol. 53, no. 6, pp. 515-525, 2002.

[BaRi99] R. Baeza-Yates und B. Ribeiro-Neto. Modern Information Retrieval. Addison-Wesley, ISBN 0-201-39829-X, Januar 1999.

[Bhar03] K. Bharat. Ranking search results by reranking the results based on local inter-connectivity. United States Patent 6526440. Erteilt am 25. Februar 2003.

[Bune97] P. Bunemann. Semistructured Data. In: *16th ACM Symposium on Principles of Database Systems PODS*, Tuscon, Arizona, pp. 117-121, Mai 1997.

[Buss02] S. Busse. Modellkorrespondenzen für die kontinuierliche Entwicklung mediatorbasierter Informationssysteme. Dissertation, TU Berlin, Fakultät IV Elektrotechnik und Informatik, Logos Verlag Berlin, ISBN 3-89722-964-1, 2002.

[BoJo96] M. Bowman und R. John. The Synopsis File System: From Files to File Objects. In: *Workshop on Distributed Object and Mobile Code*, Boston, Massachusetts, June 1996.

[Call00] B. Callaghan. NFS Illustrated. Professional Computing Series. Addison Wesley, ISBN 0-201-32570-5**,** 2000.

[Camm00] U. Callmeier. PET - A Platform for Experimentation with Efficient HPSG Processing Techniques. In: *Natural Language Engineering*, 6 (1):99 – 108, Cambridge University Press, 2000.

[CBAC05] E. Chan, J. Bresler, J. Al-Muhtadi und R. Campbell. Gaia Microserver: An Extendable Mobile Middleware Platform. In: *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PERCOM'05)*, Volume 00, pp. 309-313, Kauai Island, Hawaii, 08-12 März 2005.

[ChRZ03] A. Chaudhri, A. Rashid, R. Zicari. XML Data Management: Native XML and XML-Enabled Database Systems. Addison Wesley Professional, Boston, Massachusetts, ISBN 0-201-84452-4, März 2003.

[CoDK02] G. Coulouris, J. Dollimore, T. Kindberg. Verteilte Systeme - Konzepte und Design. 3., überarbeitete Auflage, Pearson Studium, ISBN 3-8273-7022-1, 2002.

[CPMN03] F. M. Cuenca-Acuna, C. Peery, R. P. Martin und T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In: *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, vol. 00, p. 236, IEEE Computer Society, Juni 2003.

[CTW+94] M. J. Carey, O. Tsatalos, S. White, M. Zwilling, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon und C. K. Tan. Shoring Up Persistent Applications. In: *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data*, Minneapolis, Minnesota, United States. ACM SIGMOD Record, 23(2):383-394, June 1994.

[Cuen04] F. M. Cuenca-Acuna. A Probabilistic Approach to Building Large Scale Federated Systems. Ph.D. Thesis, Department of Computer Science, Rutgers University, April 2004.

[DELS99] P. Dourish, W. K. Ed-wards, A. LaMarca und M. Salisbury. Presto:an experimental architecture for fluid interactive documentspaces. In: *ACM Transactions on Computer-Human Interaction*, 6 (2):133-161, Juni 1999.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy und D. Suciu. XML-QL: A Query Language for XML. August 1998.

http://www.w3.org/TR/NOTE-xml-ql/
(Zugriff am 30. 07. 2005)

[EMC+01] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode und P. Zeitz. Mathematisch-strukturelle Grundlagen der Informatik. 2. überarbeitete Auflage, Springer-Verlag Berlin Heidelberg, ISBN 3-540-41923-3, April 2001.

[FrBa92] W. B. Frakes und R. Baeza-Yates. Information Retrieval: Data Structures and Algorithms. Prentice-Hall, Englewood Cliffs, New Jersey, ISBN 0-13-463837-9, 1992.

[GiBe99] D. Giampaolo, Inc. Be. Practical File System Design with the Be File System. Morgan Kaufmann, San Francisco, California, ISBN 1-55860-497-9, 1999.

[GJSO91] D. K. Gifford, P. Jouvelot, M. A. Sheldon und J. W. O'Toole: Semantic file systems. In: *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, pp. 16-25, Oktober 1991.

[GoMa99] B. Gopal und U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In: *Proceedings of the 3rd ACM Symposium on Operating Systems Design and Implementation*, pp. 265–278, New Orleans, LA, Februar 1999.

[Grim04] R. Grimes. Revolutionary File Storage System Lets Users Search and Manage Files Based on Content. MSDN Magazine, Januar 2004.

http://msdn.microsoft.com/msdnmag/issues/04/01/WinFS/
(Zugriff am 30. 07. 2005)

[Guzm03] J. de Guzman. Spirit v1.8.1 User's Guide. 2003.

http://www.boost.org/libs/spirit/
http://spirit.sourceforge.net
(Zugriff am 30. 07. 2005)

[HeCa03] C. K. Hess und R. H. Campbell. A Context-Aware Data Management System for Ubiquitous Computing Applications. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Providence, Rhode Island, 19-22 Mai 2003.

[Hero03] H. Herold. Lex & yacc: Die Profitools zur lexikalischen und syntaktischen Textanalyse. Addison-Wesley, ISBN 3-8273-2096-8, März 2003.

[Hess03] C. K. Hess. The design and implementation of a context-aware file system for ubiquitous computing applications. Ph.D. Thesis, Computer Science Department, University of Illinois at Urbana-Champaign, 2003.

[Hirs97] D. S. Hirschberg. Serial Computations of Levenshtein Distances. In: Apostolico, Galil (Hrsg.), *Pattern matching Algorithms*, pp. 123-141, Oxford University Press, 1997.

[HKM+88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham und M. J. West. Scale and Performance in a Distributed File System. In: *ACM Transactions on Computer Systems*, Volume 6 , Issue 1, pp. 51–81, Februar 1988.

[Hupf03] F. Hupfeld. Hierarchical Structures in Attribute-based Namespaces and their Application to Browsing. ZIB Report 03-06, Zuse Institute Berlin, März 2003.

[Hupf04] F. Hupfeld. Log-Structured Storage for Efficient Weakly-Connected Replication. In: *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, Workshops 2004.

[ISO01] International Organization for Standardization. Information Technology — Syntactic Metalanguage — Extended BNF. ISO/IEC 14977:1996, August 2001.

[Josu99] N. M. Josuttis. The C++ Standard Library - A Tutorial and Reference. Addison-Wesley Professional, ISBN 0-201-37926-0, August 1999.

[KeEi04] A. Kemper und A. Eickler. Datenbanksysteme - Eine Einführung. 5., aktualisierte und erweiterte Auflage, Oldenbourg Verlag, ISBN 3-486-27392-2, November 2004.

[KHMG03] S. D. Kamvar, T. H. Haveliwala, C. D. Manning und G. H. Golub. Exploiting the Block Structure of the Web for Computing PageRank. Technical Report, Stanford University, März 2003.

[Kofl04] M. Kofler. Linux - Installation, Konfiguration, Anwendung. 7. Auflage, Addison-Wesley, ISBN 3-8273-2158-1, November 2004.

[Krüg04] G. Krüger. Handbuch der Java-Programmierung. 4. Auflage, Addison-Wesley, ISBN 3-8273-2201-4, 2004.

[LeMB95] J. R. Levine, T. Mason und D. Brown. Lex & yacc. 2., korrigierte Auflage, O'Reilly, ISBN 1-56592-000-7, Oktober 1992.

[LyVa03] P. Lyman und H. R. Varian. "How Much Information?" 2003. University of California at Berkeley, Oktober 2003.

http://www.sims.berkeley.edu/research/projects/how-much-info-2003/
(Zugriff am 30. 07. 2005)

[MKL+02] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins und Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Laboratories Palo Alto, März 2002.

[OtGi92] J. O' Toole und D. Gifford. Names should mean what, not where. In: *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, ACM Press, September 1992.

[Page01] L. Page. Method for node ranking in a linked database. United States Patent 6285999. Erteilt am 04. Sept. 2001.

[PBMW99] L. Page, S. Brin, R. Motwani und T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report, Stanford University, November 1999.

[Rect04] B. Rector. Introducing Longhorn for Developers. Wise Owl Consulting, 2004.

http://msdn.microsoft.com/windowsvista/community/books/rector/default.aspx
(Zugriff am 30. 07. 2005)

[RHC+02] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell und K. Nahrstedt. A middleware infrastructure for active spaces. In *IEEE Pervasive Computing*, Volume 1, Issue 4, pp. 74-83, Okt.-Dez. 2002

[RHA+85] C. Rose, B. Hacker und Inc. Apple Computer. Apple Inside Macintosh. Addison-Wesley, ISBN 0201177374, Juli 1985.

[Roma03] M. Roman. An Application Framework for Active Space Applications. Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[SaMc83] G. Salton und M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, New York, NY, USA, ISBN 0-07-054484-0, September 1983.

[Schi04a] F. Schintke. ZIBDMS: A System to Organize Scientific Data. Vortrag im Forschungsseminar „Grid Computing“, Zuse-Institut Berlin, Juli 2004.

[Schi04b] F. Schintke. The ZIB Distributed Data Management System. Vortrag beim Dagstuhl Seminar *„Future Generation Grids“ (FGG 2004, Seminar Nr. 04451)*, Schloss Dagstuhl, Wadern, Germany, 01-05 November 2004.

[ScRe03] F. Schintke, A. Reinefeld. Modeling Replica Availability in Large Data Grids. In: *Journal of Grid Computing*, 1(2):219-227. Kluwer Academic Publisher, Juni 2003.

[ScSR03] F. Schintke, T. Schütt, A. Reinefeld. A Framework for Self-Optimizing Grids Using P2P Components. In: *14th Intl. Workshop on Database and Expert Systems Applications (DEXA'03)*, pp. 689-693, IEEE Computer Society, September 2003.

[SoGa03] C. A. N. Soules und G. R. Ganger. Why can’t I find my files? New methods for automating attribute assignment. In: *Proceedings of the 9th Workshop on Hot Topics in Operating systems (HotOS IX)*, USENIX Association, Lihue, Hawaii, USA, 18–21 Mai 2003.

[Stro00] B. Stroustrup. Die C++-Programmiersprache. 4. Auflage, Deutsche Übersetzung der Special Edition, Addison-Wesley, ISBN 3-8273-1660-X, Mai 2000.

[SUN95] Inc. Sun Microsystems. RFC 1813: NFS Version 3 Protocol Specification. Juni 1995.

http://www.ietf.org/rfc/rfc1813.txt
(Zugriff am 30. 07. 2005)

[Sure05] K. Surendorf. Das Praxisbuch Mac OS X 10.4 Tiger - Die Version 10.4 im professionellen Einsatz. Galileo Design, ISBN 3-89842-621-1, Juni 2005.

[Tann02] A. S. Tannenbaum. Modern Operating Systems. 2., überarbeitete Auflage, Prentice Hall, ISBN 0-13-031358-0, Februar 2001.

[VaPa97] V. Vasudevan und P. Pazandak. Semantic File Systems. Technical Report, Inc. Object Services and Consulting, März 1997.

[WiMB99] I. H. Witten, A. Moffat und T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. 2. Auflage, Morgan Kaufmann, San Francisco, California, ISBN 1-55860-570-3, 1999.

[Witt05] J. Witte. Implementierung einer NFS Schnittstelle im Kontext eines Grid-Datenmanagement-Systems. Diplomarbeit, Humboldt-Universität zu Berlin, Januar 2005.

[ZiKB02] J. Ziegler, C. Kunz und V. Bosch. Matrix browser: visualizing and exploring large networked information spaces. In: *Proceedings of the International Conference on Computer Human Interaction SIGCHI 2002*, Minneapolis, Minnesota, USA, pp. 602-603, ACM Press, 2002.

Stichwortverzeichnis

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z