TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Medieninformatik

Diplomarbeit

Effiziente Ad-Hoc-Abfragen in Objektdatenbanken am Beispiel der ZODB

Sebastian Wehrmann

Halle, den 23. April 2008

Prüfer: Prof. Dr. Maximilian Eibl
Dipl. Inf. Christian Zagrodnick

Betreuer: Christian Theune

gocept

gocept gmbh & co. kg
Forsterstraße 29
D-06112 Halle (Saale)

www.gocept.com

Danksagung

Bedanken möchte ich mich in besonderem Maße bei Herrn Prof. Dr. Maximilian Eibl und Herrn Christian Theune für die Unterstützung und die umfassende und gute Betreuung dieser Arbeit.

Ich bedanke mich ausserdem bei Herrm Thomas Lotze für die zahlreichen Tipps zu LaTeX und das finale Korrekturlesen.

Weiterhin danke ich der Firma gocept gmbh & co. kg, die mir diese interessante und spannende Diplomarbeit ermöglicht hat und mir bei Fragen und Problemen immer zu Seite stand.

Aufgabenstellung

Die Zope Object Database, kurz ZODB, ist eine Open-Source-Datenbank für Python. Im Gegensatz zu den meisten relationalen Datenbanken verfügt die ZODB allerdings nicht über eine Anfragesprache zur gezielten Selektion von Objekten.

Aufgabe dieser Diplomarbeit ist es, effiziente Ad-Hoc-Anfragemöglichkeiten zu evaluieren und eine geeignete als Zusatzprodukt in Python zu implementieren.

Folgende Themen sind zu bearbeiten:

- Vergleich und Auswahl einer Anfragesprache für Objektgraphen
- Auswahl von Indexstrukturen zur Unterstützung der gewählten Anfragesprache
- Implementation eines Zusatzprodukts zur ZODB, die eine Anfragesprache sowie unterstützende Indizes bereitstellt
- Testen und Bewerten der Implementierung

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis iv

Listings v

1 Grundlagen 1

1.1 Hintergrund . . . 1
1.2 Begriffsklärung . . . 1
1.2.1 Datenbank . . . 1
1.2.2 Datenbankmanagementsystem . . . 2
1.2.3 Datenbankmodell . . . 2
1.2.4 Datenbanksprachen . . . 3
1.3 Prinzipien objektorientierter Anfragesprachen . . . 4
1.4 Persistenz in Python . . . 5
1.4.1 Aufbau und Funktion der ZODB . . . 6
1.5 XML und DOM . . . 9

2 Anfragesprachen für Objektgraphen 10

2.1 Beispieldatenbank . . . 10
2.2 Kriterien für Anfragesprachen . . . 11
2.3 Anforderungskatalog und Bewertungsraster . . . 12
2.3.1 Anforderungen an Anfragesprachen für Objektgraphen . . . 13
2.3.2 Bewertungsraster für die Anforderungen . . . 15
2.4 Evaluation der Anfragesprachen . . . 19
2.4.1 XML Query Language (XML-QL) . . . 19
2.4.2 XML Path Language (XPath) . . . 22
2.4.3 Regular Path Expressions (RPE) . . . 27
2.4.4 XML-Graphical-Language (XML-GL) . . . 30
2.4.5 Quilt . . . 34
2.4.6 XQuery - An XML Query Language . . . 40
2.4.7 Object Query Language (OQL) . . . 47

2.4.8 Native Queries 52
2.5 Auswahl einer Sprache für die Implementierung 55

3 Indexstrukturen **58**
3.1 Indexstrukturen im Überblick 58
3.1.1 Einstufige geordnete Indexstrukturen 58
3.1.2 Mehrstufige Indexstrukturen 61
3.1.3 B- und B^+-Bäume 63
3.1.4 Hashing 65
3.2 Indexstrukturen für den Objektgraphen der ZODB 67
3.2.1 Indexorganisation 68
3.2.2 Datenorganisation 68

4 Implementierung **75**
4.1 Die ObjectQuery 75
4.1.1 Der QueryProcessor 76
4.1.2 Die ObjectCollection 81
4.2 Anbindung an die ZODB 86
4.3 Probleme und Lösungen 87
4.3.1 Mehrfaches Hinzufügen gleicher Objekte 87
4.3.2 Kreisreferenzen 88

5 Bewertung und Ausblick **90**
5.1 Testen und Bewerten der Implementierung 90
5.1.1 Testen in Python 91
5.1.2 Testen der ObjectQuery 92
5.2 Ausblick und weiterführende Gedanken 95

Literaturverzeichnis **96**

A Quelltexte **102**

Abbildungsverzeichnis

1.1 Schematischer Aufbau der ZODB 8

2.1 ERD der Beispieldatenbank 11
2.2 Auslesen aller student-Objekte in XML-GL 31
2.3 Notationsmöglichkeiten für die Aufbereitung der Ergebnismenge in XML-GL . . 32
2.4 XML-GL Anfrage mit match-Bedingungen 32
2.5 Datenfluß in einem FLWR-Ausdruck 36
2.6 XQuery Prozessmodell 41

3.1 Sekundärindex mit Indirektionsstufe. 59
3.2 Dichter Sekundärindex auf nicht ordnendes Schlüsselfeld. 61
3.3 Zweistufiger Primärindex 62
3.4 Knoten eines Suchbaumes 64
3.5 B-Baumstrukturen 64
3.6 Struktur des erweiterbaren Hashing 67
3.7 Erweitertes Nummerierungsschema nach [LM01] 72
3.8 Weiteres Nummerierungsschema 73

4.1 Aufbau der ObjectQuery 75
4.2 Die EBNF für RPE-Ausdrücke 79
4.3 Aufrufbaum für Query Q2 (aus [LM01]) 80

5.1 Ergebnis des Performance-Tests in graphischer Darstellung 94

Tabellenverzeichnis

2.1 Ergebnis der Evaluation von XML-QL 20
2.2 Verkürzte und unverkürzte Lokalisierungspfade in XPath 23
2.3 Wichtige Achsen und ihre Beschreibung in XPath 24
2.4 Ausgewählte Funktionen in XPath 25
2.5 Ergebnis der Evaluation von XPath 26
2.6 Mögliche Notationen in RPE 28
2.7 Ergebnis der Evaluation der RPE 29
2.8 Ergebnis der Evaluation der XML-GL 33
2.9 Wichtige Symbole in Quilt-Pfadausdrücken 35
2.10 Ergebnis der Evaluation von Quilt 38
2.11 Beispiele für Listentypen in XQuery 43
2.12 Ergebnis der Evaluation von XQuery 45
2.13 Grundlegender Aufbau einer Anfrage in OQL 47
2.14 Gebräuchliche Vergleichsoperatoren in OQL 49
2.15 Ergebnis der Evaluation von OQL 51
2.16 Gesamtergebnis der Evaluation 56

5.1 Ergebnis des Performance-Tests für die ObjectCollection (Auswahl) 93

Listings

2.1 Beispieldatenbank als XML DTD 10
2.2 Beispieldatenbank als Python Code 11
2.3 XML-QL: Auslesen des Alters einer bestimmen Person 19
2.4 XML-QL: Anfrage von geschachtelten Strukturen 19
2.5 XML-QL: Beispielresultat der Anfrage aus 2.4 20
2.6 RPE: Filtern nach Vorname der Studenten 27
2.7 RPE: Anfrage mit Vereinigung 28
2.8 Quilt: Differenzierte Nutzungsmöglichkeiten von Prädkatausdrücken 35
2.9 Quilt: Konstruktion von Elementen mit Quilt 36
2.10 Quilt: Beispielanfrage für Gruppierung 38
2.11 XQuery: Auslesen des Alters einer bestimmen Person 44
2.12 XQuery: Auslesen aller Kurse, die von mehr als 10 Studenten besucht werden. . . 44
2.13 XQuery: Deklaration eigener Funktionen 44
2.14 OQL: Pfadausdrücke und Extend am Beispiel 47
2.15 OQL: Beeinflussen der Ergebnismenge über Pfadausdrücke 48
2.16 OQL: Bedingungen mit Hilfe der WHERE-Klausel 48
2.17 OQL: Sortierung am Beispiel 49
2.18 OQL: Gruppierung am Beispiel 49
2.19 Native Queries in Python 53
3.1 Laufzeittest des Klassenindex mittels Dictionary 69
3.2 Ergebnis des Laufzeittests mittels Dictionary 69
3.3 Laufzeittest des Klassenindex mittels OOBTree 70
3.4 Ergebnis des Laufzeittests mittels OOBTree 70
4.1 Beispiele für Pfadelemente in RPE 77
4.2 SimpleParse-Grammatik für RPE (gekürzt) 78
4.3 Query Q2 (aus [LM01]) 79
4.4 Queryplan für Query Q2 80
4.5 Mapping für die Vergleichsoperationen des $\mathcal{EA}$-Joins 83
5.1 Doctest zum Testen des korrekten Parsens eines Pfadausdrucks 91
5.2 Testabdeckung der Objectquery 92
A.1 Python-Script zum Testen der Performance der ObjectQuery 102

Abkürzungsverzeichnis

API Application Programming Interface

BNF Backus-Naur-Form

DBM Datenbankmodell

DBMS Datenbankmanagementsystem

DB Datenbank

DDL Data Definition Language

DML Data Manipulation Language

DOM Document Object Model

DTD Document Type Definition

EBNF Erweiterte BNF

ERD Entity-Relationship-Diagramm

MVCC Multiversion Concurrency Control

ODMG Object Database Management Group

OQL Object Query Language

ORM Object-Relational Mapping

RPE Regular Path Expressions

SQL Structured Query Language

UAT User Acceptance Test

URL Uniform Resource Locator

UoD Universe of Discourse

VDL View Definition Language

W3C World Wide Web Consortium

XDM XML Data Model

XISS XML Indexing and Storage System

XML Extensible Markup Language

XML-GL XML-Graphical-Language

XML-QL XML Query Language

XSD XML Schema Definition

XPath XML Path Language

XSLT Extensible Stylesheet Language Transformations

ZEO Zope Enterprise Objects

ZODB Zope Object Database

1 Grundlagen

1.1 Hintergrund

Datenbanken und Datenbanksysteme spielen heutzutage eine wesentliche Rolle in unserer Gesellschaft. Dabei ist den Meisten nicht bewusst, wann sie mit einer Datenbank interagieren, zum Beispiel beim Einkauf im Supermarkt, beim Geldabheben am Bankautomaten oder beim Bestellen eines Bahntickets im Internet. Für diese und andere ähnlich geartete Anwendungsfälle existieren Datenbankanwendungen, die Informationen in Textform oder numerisch speichern. Weitere Datenbanken sind unter anderem Multimedia-Datenbanken, geografische Informationssysteme oder Echtzeitdatenbanken.

1.2 Begriffsklärung

1.2.1 Datenbank

Allgemein ausgedrückt ist eine Datenbank (DB) "eine Sammlung von Daten, die einen Ausschnitt der realen Welt beschreiben. Unter Daten verstehen wir bekannte Tatsachen, die aufgezeichnet werden können und eine implizite Bedeutung haben." [EN02, Seite 24] Diese allgemein gehaltene Definition kann eingeschränkt werden. So hat eine DB folgende Eigenschaften[1]:

- Es werden Aspekte der realen Welt in einer DB dargestellt. Man bezeichnet sie auch als *Miniwelt* oder Universe of Discourse (UoD). Änderungen in der Miniwelt spiegeln sich in der DB wieder.
- Daten hängen in einer DB logisch zusammen. Eine zufällige Datensammlung wird nicht als DB bezeichnet.
- Die DB hat einen Zweck. Ihre Daten werden von bestimmten Benutzern in zweckbezogenen Anwendungen genutzt.

Datenbanken haben einige Vorteile gegenüber konventioneller Speicherung (zum Beispiel in Dateien). Zu diesen Vorteilen zählen unter anderem[2]:

[1] Vgl. [EN02, Seite 24f]
[2] Vgl. [EN02, Seite 34ff]

- *Redundanzkontrolle*
  Verschiedene Sichten (View) auf einen Datenbestand erlauben das redundante Speichern von Daten für mehrere Benutzer.

- *Zugriffsbeschränkung*
  Bestimmten Benutzern kann der partielle Zugriff auf Daten in der DB verwehrt werden.

- *Integritätsbedingungen*
  Daten lassen sich mit Bedingungen versehen, welche für die Speicherung erfüllt sein müssen. Dies erlaubt einerseits eine Validierung an zentraler Stelle und andererseits eine effiziente Speicherung der Daten.

1.2.2 Datenbankmanagementsystem

[EN02, Seite 25] definiert ein Datenbankmanagementsystem (DBMS) als "ein Softwaresystem (oder auch Sammlung von Programmen), das dem Benutzer das Erstellen und die Pflege einer DB ermöglicht." Das DBMS vereinfacht die **Definition** (das heißt das Spezifizieren der Datentypen, Beziehungen und Einschränkungen für die in der DB zu speichernden Daten) und die **Manipulation** (das heißt das Lesen und Schreiben von Daten in beziehungsweise aus der DB) von Datenbanken für verschiedene Anwendungen.

Eine schematische Beschreibung eines DBMS ist die **Drei-Schema-Architektur** (oder auch ANSI/SPARC-Architektur). Sie unterteilt das DBMS in folgende drei Schichten[3]:

- *Externe Ebene*
  Die externe Ebene stellt dem Benutzer und Anwendungen individuelle Sichten auf die Daten zur Verfügung, die keine Details über die Art und Weise der Speicherung der Daten enthalten. Die externe Ebene ist die Schnittstelle des DBMS mit seiner Umwelt.

- *Konzeptionelle Ebene*
  In der konzeptionelle Ebene wird beschrieben, welche Daten in der DB gespeichert und wie deren Beziehungen untereinander sind.

- *Interne Ebene*
  Die interne Ebene stellt die physische Sicht der DB dar. Sie beschreibt, wie und wo die Daten in der Datenbank gespeichert werden.

1.2.3 Datenbankmodell

Nach [HS99, Seite 49] ist ein Datenbankmodell (DBM) "ein System von Konzepten zur Beschreibung von Datenbanken. Es legt (...) Syntax und Semantik von Datenbeschreibungen für ein Da-

[3] Vgl. [EN02, Abschnitt 2.2.1]

tenbanksystem fest, die sogenannten Datenbankschemata." Zu den bekanntesten Vertretern zählen das **relationale** und das **objektorientierte DBM**.

Beide Datenbankmodelle verwenden Konzepte wie **Entitäten**, **Attribute** und Beziehungen, wobei eine Entität ein Objekt aus der realen Welt und ein Attribut eine Eigenschaft eines solchen Objektes darstellt. Beziehungen (**Relationen**) bestehen zwischen zwei oder mehr dieser Entitäten. Zur Beschreibung dieser Modelle kommt in der Regel das Entity-Relationship-Diagramm (ERD)[4], eine graphische Darstellung der Entitäts- und Beziehungstypen, zum Einsatz.

Der Unterschied zwischen dem relationalen und dem objektorientierten DBM liegt in der Art der Repräsentation der Daten. Im relationalen DBM werden Entitäten in Tabellen organisiert, deren Spalten (jeweils eine Entität) eine Menge von Attributen enthalten. Relationen werden über Fremdschlüssel implementiert, die Beziehungen zwischen Entitäten gewährleisten. Das objektorientierte DBM verwaltet Entitäten im Sinne der Objektorientierung als Objekte und nicht wie im relationalen DBM in Tabellen. Als ein Objekt wird die Zusammenfassung von zugehörigen Attributen bezeichnet, die dieses näher beschreiben.

1.2.4 Datenbanksprachen

Es existieren eine Reihe von Sprachen, die zur Manipulation eines DBMS genutzt werden können. So gibt es die **Datendefinitionssprache** Data Definition Language (DDL), die der Datenbankadministrator und der Datenbankdesigner zur Definition der konzeptuellen und internen Schemata nutzt. Benutzer wiederum nutzen die **Datenmanipulationssprache** Data Manipulation Language (DML) zum Lesen, Einfügen, Löschen und Ändern von Daten.

Ein typisches Beispiel für eine umfassende Datenbanksprache, die DDL, DML und eine **View Definitionssprache** View Definition Language (VDL) enthält, ist die Structured Query Language (SQL).

Bei der DML wird zwischen **prozeduralen** und **nicht prozeduralen** DML unterschieden. Prozedurale DML müssen in eine Programmiersprache eingebettet werden und rufen in der Regel einzelne Datensätze oder Objekte aus der DB ab und verarbeitet sie getrennt[5]. Man nennt sie deshalb auch **satzbasierte DML**. Nicht prozedurale oder auch **mengenorientierte DMLs** können eigenständig benutzt werden, um komplexe Datenbankoperationen präzise und kompakt zu spezifizieren.

Wird eine nicht prozedurale DML in einer Programmiersprache einzeln und interaktiv verwendet, so bezeichnet man sie auch als **Anfragesprache** (query language). Eine deklarative Anfragesprache (wie SQL) stellt die aktuell gebräuchlichste Möglichkeit, DBMS zu benutzen, dar.

[4] Vgl. [EN02, Kapitel 3]
[5] Vgl. [EN02, Seite 53]

Für die Kopplung der Datenbanksprache an die Wirtssprache lassen sich grundsätzlich vier Formen wählen[6]:

- *Prozedurale Schnittstelle*
  Die Datenbankfunktionen werden über eine externe Prozedur in einer beliebigen Programmiersprache aufgerufen. (Beispiel: die JDBC-Schnittstelle für Java-Programme)

- *Spracherweiterung*
  Die Wirtssprache wird um die Konstrukte der Datenbanksprache erweitert.

- *Spracheinbettung mit Vorübersetzer*
  Die Datenbankoperationen stehen in der Wirtssprache, werden mit einem Symbol (Präfixsymbol) gekennzeichnet und müssen von einem Vorübersetzer (Precompiler) bearbeitet werden. (Beispiel: Embedded SQL)

- *Integrierter Ansatz*
  Die Wirtssprache wird um ein Datenmodell und die zugehörigen Datenbankoperationen erweitert (Datenbankprogrammiersprache). (Beispiel: 4GL-Systeme)

Die größte Bedeutung haben die **prozedurale Schnittstelle** und die **Spracheinbettung mit Vorübersetzer** erlangt. Die prozedurale Schnittstelle hat den Vorteil, dass die Datenbankaufrufe dynamisch zur Laufzeit zusammengestellt werden können, während sie bei der Spracheinbettung mit Vorübersetzer zur Übersetzungszeit der Wirtssprache als externe Prozeduren zur Verfügung gestellt werden. Im Gegensatz zur prozeduralen Schnittstelle können hier allerdings Fehler schon zur Übersetzungszeit festgestellt werden.

1.3 Prinzipien objektorientierter Anfragesprachen

Die Vielfalt der Strukturkonzepte im objektorientierten Bereich führt zu dem Problem, den Ergebnistyp einer Anfrage festzulegen. Relationale Datenbanksprachen liefern als Ergebnis immer eine Menge von Daten. Für objektorientierte Sprachen bieten sich zur Bestimmung des Typs prinzipiell drei Möglichkeiten[7]:

1. *Relationale Semantik*
   Aus den gespeicherten Zuständen der Objekte werden deren Daten extrahiert. Da das Ergebnis keine Menge von Objekten, sondern eine Menge von Daten, ist, können klassenspezifische Methoden auf die Einträge der Ergebnismenge nicht angewendet werden.

[6] Vgl. [HR01, Seite 352] und [DSB04]
[7] Vgl. [HS99, Seite 329f]

2. *Objekterzeugende Semantik*
   Aus den Objekten in der DB werden neue Objekte mit Zuständen erzeugt, die von vorhandenen Objekten in der DB extrahiert wurden. Sie gehören zu einer neuen, dynamisch erzeugten Klasse, für die neue Methoden definiert, aber keine Methoden von Klassen des DB-Schemas geerbt werden können.

3. *Objekterhaltende Semantik*
   Man erhält eine Auswahl der in der DB vorkommenden Objekte mit neuen Zuständen. Die Objekte sind entweder vom gleichen Typ, wie die Objekte in der Datenbank (statische Klassifizierung und Typisierung), oder von einem neuen Typ, nämlich einer Unterklasse der schon bestehenden Klasse (dynamische Klassifizierung und Typisierung). Die dynamische Klassifizierung und Typisierung hat den Vorteil, dass für die neuen Unterklassen Methoden definiert werden können, die neben den geerbten Methoden verwendet werden können.

1.4 Persistenz in Python

Der Begriff Persistenz geht auf Atkinson[8] zurück. Danach haben Objekte, die im Programm erzeugt und benutzt werden, unabhängig von der Blockstruktur des Programms und Gültigkeitsbereichen von Variablen eine beliebige Lebensdauer. Ein Persistenzmechanismus soll dabei folgende zwei Prinzipien erfüllen:

- *Typ-Orthogonalität*
  Es können Programm-Objekte beliebigen Typs persistent gemacht werden.

- *Programm-Unabhängigkeit*
  Explizite Speicherbefehle sind verboten. Die Angabe, ob ein Objekt persistent ist oder nicht, erfolgt deskriptiv.

Neben **persistenten** Objekten gibt es die **transienten** Objekte, deren Lebensdauer an die Laufzeit des Programmes gebunden ist. Die Programmiersprache Python arbeitet nativ nur mit solchen transienten Objekten. Weiterhin gibt es zwei Realisierungstechniken für Persistenzmechanismen[9]:

- *Klassenabhängige Persistenz*
  Die klassenabhängige Persistenz macht unterhalb einer persistenten Wurzelklasse einen vollständigen Teilgraphen der Klassenhierarchie mit allen Objekten persistent. Der Teilgraph bestimmt das Datenbankschema.

[8] Vgl. [Atk91, Seite 453ff]
[9] Vgl. [HS99, Seite 363]

- *Objektabhängige Persistenz*
  Bei der objektabhängigen Persistenz gibt es kein Datenbankschema im eigentlichen Sinn. Objekte können persistent oder transient sein, was in der Regel beim Erzeugen entschieden wird. Es können daher sowohl persistente als auch transiente Objekte einer Klasse existieren.

Die Zope Object Database (ZODB) ist ein Vertreter der **objektabhängigen Persistenz**.[10] Für die Persistenz-Fortpflanzung, das heißt die Wirkung der Persistenz auf typgleiche Objekte, gibt es zwei Implementierungsmöglichkeiten[11]:

- *Explizite Persistenz*
  Jedes Objekt muss einzeln persistent gemacht werden. Zu einem persistenten Objekte gehörende Komponenten sind nicht automatisch persistent und werden nicht langfristig mitgespeichert, solange sie nicht als persistent gekennzeichnet wurden.

- *Persistenz durch Erreichbarkeit*
  Ist ein Objekt persistent, so sind auch alle Komponentenobjekte rekursiv persistent. Das bedeutet, dass zum Beispiel bei einem persistenten Mengenobjekt automatisch alle Elemente der Menge persistent sind.

Die ZODB implementiert **Persistenz durch Erreichbarkeit**. Im folgenden Abschnitt wird die Funktionalität der ZODB detailliert erklärt.

1.4.1 Aufbau und Funktion der ZODB

Die ZODB arbeitet nach einem Prinzip, das auch in vielen anderen objektorientierten Programmiersprachen verfügbar ist: das **Serialisieren**. Beim Serialisieren werden die Daten der Objekte in eine minimale Datenstruktur (Byte-Folge) umgewandelt, um dann zum Beispiel in eine Datei geschrieben zu werden. Für kleine Anwendungen ist dies eine Alternative, Objekte ohne DBMS persistent zu machen: beim Start der Anwendung werden die serialisierten Daten eingelesen (unpickle) und beim Beenden wieder serialisiert (pickle). Allerdings stößt man bei diesem Verfahren schnell an Grenzen: das Programm muss sich um die Art und Weise, wie die Objekte gespeichert und gelesen werden, selbst kümmern.

Die ZODB bringt eine Reihe an Funktionalität eines DBMS mit und bietet neben einer effizienten Speicherung der Objekte auch Transaktionen mit Multiversion Concurrency Control (MVCC), Savepoints und Undo-Funktionalität.

[10] Detailierte Informationen zur Funktionsweise der ZODB liefert Abschnitt 1.4.1
[11] Vgl. [HS99, Seite 364]

Transaktionen bieten dem Anwender die Möglichkeit, mehrere DB-Operationen zu bündeln und Änderungen an der DB vom Erfolg aller gebündelten Operationen abhängig zu machen. Transaktionen sollten die folgenden Eigenschaften aufweisen, die unter der Abkürzung *ACID*[12] bekannt sind. ACID steht dabei für die englischen Begriffe Atomicity, Isolation, Consistency und Durability (Atomizität, Konsistenz, Isolation und Dauerhaftigkeit) und bedeutet:

- *Atomizität*
  Eine Transaktion wird entweder vollständig oder garnicht ausgeführt. Änderungen an der DB sind atomar. Inkonsistenzen durch Teiländerungen treten nicht auf.
- *Konsistenz*
  Die Transaktion hinterlässt nach Beendigung einen konsistenten Datenzustand. Transaktionen, die in einen inkonsistenten Datenzustand führen, werden nicht gespeichert.
- *Isolation*
  In Ausführung befindliche Transaktionen dürfen sich gegenseitig nicht beeinflussen. Die ZODB nutzt hierfür den MVCC-Mechanismus [Wik07b].
- *Dauerhaftigkeit*
  Das Ergebnis einer Transaktion ist dauerhaft. Erfolgreiche Änderungen bleiben dauerhaft in der DB erhalten, insbesondere nach Fehlern oder Systemabstürzen.

Zusätzlich stellt die ZODB noch folgende Funkionen bereit[13]:

- *Transparenz*
  Persistenz ist transparent gegenüber der Umwelt. Es müssen keine speziellen Funktionen gerufen werden, um Daten zu speichern oder zu lesen. Jedes Objekt, das serialisierbar ist, kann in der ZODB gespeichert und aus der ZODB geladen werden. (siehe *Programm-Unabhängigkeit* auf Seite 5)
- *Wiederherstellung*
  Die ZODB kann jede Änderung an einem Objekt wiederherstellen, solange diese Aktion nicht zu einem inkonsistenten Zustand der DB führt. Üblicherweise gilt dies aber nur für sehr kurze Zeiträume.
- *Verschiedene Speicherschichten*
  Die ZODB bietet die Möglichkeit, unterschiedliche Speicherschichten (sogenannte Storages) zu nutzen. In der Regel kommt der FileStorage, der Daten als Transaktionslog speichert, zum Einsatz. Auch ein Netzwerk-Storage (Zope Enterprise Objects (ZEO)) auf Basis einer Client/Server Architektur, steht zur Verfügung.

[12] Vgl. [EN02, Seite 687f], [vW07, Seite 84f] und [Wik08a]
[13] Vgl. [vW07, Seite 85]

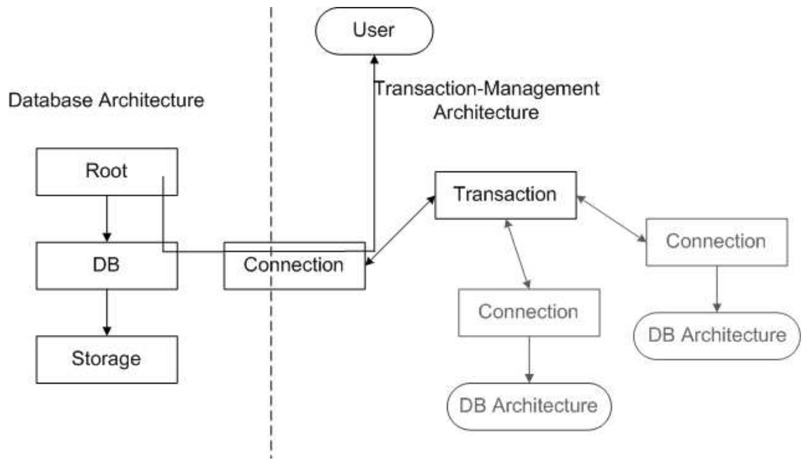

Abbildung 1.1: Schematischer Aufbau der ZODB

Der schematische Aufbau der ZODB ist in Abbildung 1.1 dargestellt. Die einzelnen Komponenten werden nachfolgend erläuert.

- **Datenbank-Architektur**
  - *Root*
    Die Wurzel (root) ist der Zugang zur DB, die die Objekte enthält. Von der Wurzel aus können alle anderen Objekte erreicht und es kann mit ihnen interagiert werden.
  - *DB*
    Die Datenbank (DB) enthält das Wurzel-Element und repräsentiert die *Konzeptionelle Ebene* (siehe Abschnitt 1.2.2). Sie speichert und serialisiert Objekte.
  - *Storage*
    Die Speicherschicht (Storage) repräsentiert die *Interne Ebene* (Abschnitt 1.2.2). Sie bestimmt, wie serialisierte Objekte physisch gespeichert werden.

- **Transaktions-Management-Architektur**
  - *Connection*
    Die Connection repräsentiert eine Interaktion zwischen der Anwendung und der DB. Sie ist das Verbindungsstück zwischen Transaktions-Management und Datenbank. Sie enthält weiterhin eine Methode, um das Wurzel-Element, welches nicht direkt von der DB angefragt werden kann, zu holen.
  - *Transaction*
    Eine Connection und jedes Objekt, das von einer Connection angefragt wird, ist mit einer Transaktion verbunden. Transaktionen erlauben weiterhin die Anbindung mehrerer Connections (und damit DBs).

1.5 XML und DOM

Neben den vorgestellten klassischen DBMS gibt es weitere Möglichkeiten, Daten strukturiert zu speichern. Ein verbreiteter Vertreter zur Speicherung von vorwiegend Textdaten ist XML: "Die Extensible Markup Language (XML) (engl. für *erweiterbare Auszeichnungssprache*), ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien." [Wik08d] Für XML existieren eine Reihe von Anfragesprachen, die das Traversieren der Struktur, Filtern nach Eigenschaften und Manipulieren von Struktur und Eigenschaften von XML ermöglichen. Viele dieser Sprachen greifen nicht direkt auf XML-Code zu, sondern nutzen eine vom World Wide Web Consortium (W3C) entwickelte Schnittstelle: das Document Object Model (DOM).

"Im Sinne der objektorientierten Programmierung besteht das DOM aus einem Satz von Klassen zusammen mit deren Methoden und Attributen. Es erlaubt Computerprogrammen, dynamisch den Inhalt, die Struktur und das Layout eines Dokuments zu verändern." [Wik08c] Das DOM ist ein Application Programming Interface (API). Es strukturiert die Elemente des XML-Dokumentes hierarchisch und hält für jedes spezifische Methoden und Attribute vor. Dies entspricht dem DOM Level-1[14], welches sprachunabhängige Schnittstellen für den Zugriff auf und die Manipulation von Objekten eines Dokuments definiert. DOM Level-2[15] erweitert die Level-1 Spezifikation unter anderem um Schnittstellen für Namensräume und verschiedene Sichten auf Teilbereiche eines Dokumentes. Level-3[16] enthält unter anderem Funktionalitäten zur Serialisierung von Dokumenten, eine verbesserte Ausnahmebehandlung und den Umgang mit Zeichenkodierungen.

Da die ZODB die gespeicherten Objekte mit einem Objektgraphen verwaltet, sind XML-Anfragesprachen, die mit dem DOM arbeiten, generell auch als Anfragesprachen für die ZODB geeignet. Einige dieser Sprachen werden im Abschnitt 2.4 evaluiert, um ihre Eignung genauer zu spezifizieren.

[14]Vgl. `http://www.w3.org/TR/REC-DOM-Level-1/`
[15]Vgl. `http://www.w3.org/TR/DOM-Level-2-Core/`
[16]Vgl. `http://www.w3.org/TR/DOM-Level-3-Core/`

2 Anfragesprachen für Objektgraphen

Ziel dieses Kapitels ist die Evaluation von Anfragesprachen für Objektgraphen. Ein Objektgraph ist ein gerichteter Graph, der "aus den zwei Mengen V – eine beliebige, endliche Menge von Knoten – und E – eine Menge von Kanten, wobei $E \subseteq \{(u,v)|u,v \in V, u \neq v\}$ gilt – besteht." [Goe08]

Der Objektgraph der ZODB erlaubt zusätzlich Mehrfachkanten zwischen Objekten und Kanten mit $u = v$. Für die Menge seiner Kanten gilt demnach $E \subseteq \{(u,v,i)|u,v \in V, i \in \mathbb{N}\}$.

Es existieren eine Reihe Sprachen, die auf Objektgraphen arbeiten. In Abschnitt 2.3 werden Anforderungen an Anfragesprachen vorgestellt. Mit Hilfe dieses Anforderungskataloges wird im Abschnitt 2.3.2 ein Bewertungsraster für die im Abschnitt 2.4 durchgeführte Evaluation vorgestellt. Abschnitt 2.5 beendet die Evaluation mit der Auswahl einer Sprache für die Implementierung.

2.1 Beispieldatenbank

In diesem Kapitel werden zur Veranschaulichung von Sachverhalten und der Syntax von Anfragesprachen einige Beispielanfragen gezeigt. Um diese Beispiele untereinander vergleichbar zu machen, stelle ich in diesem Abschnitt eine Datenbank vor, mit der die Beispielanfragen arbeiten.

```
<!ELEMENT kurs (student*)>
<!ELEMENT student EMPTY>
<!ATTLIST kurs
    titel CDATA #REQUIRED
    professor CDATA #REQUIRED>
<!ATTLIST student
    vorname CDATA #IMPLIED
    nachname CDATA #REQUIRED
    alter CDATA #REQUIRED>
```

Listing 2.1: Beispieldatenbank als XML DTD

Die Beispiel-DB besteht aus zwei Entitäten: `Student` und `Kurs`. Beide Entitäten besitzen Attribute, von denen einige zwingend und andere nicht zwingend sind. So besitzen Studenten einen Nachnamen und ein Alter, ein Vorname muss allerdings nicht angegeben sein. Kurse besitzen einen Titel, einen Professor und eingeschriebene Studenten. Diese Beziehung – *eingeschriebene Studenten* – entspricht einer $1 \rightarrow n$ Relation, da an *einem* Kurs n Studenten teilnehmen können.

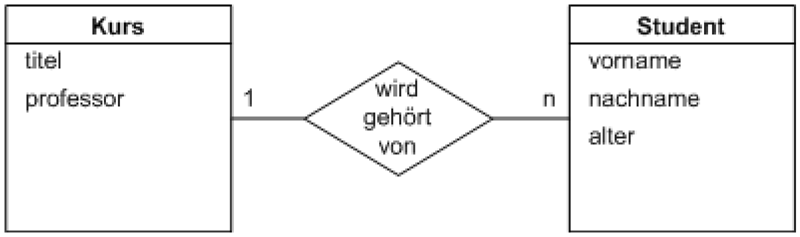

Abbildung 2.1: ERD der Beispieldatenbank

Die Beispieldatenbank wird bewusst nicht mit (Pseudo-)Daten gefüllt, um die Übersicht zu wahren. Listing 2.1 zeigt die Document Type Definition (DTD) dieser Datenbank. Sie ist die Grundlage für Anfragesprachen, die auf XML-Dokumenten arbeiten. Alle anderen Beispielanfragen richten sich nach dem ERD in Abbildung 2.1 und den Klassendefinitionen in Listing 2.2. Relationen werden hier direkt über die Kanten des Objektgraphen implementiert.

```
class Student (Persistent):
    def __init__(self, nachname, alter, vorname=""):
        self.nachname = nachname
        self.vorname = vorname
        self.alter = alter
class Kurs (Persistent):
    def __init__(self, titel, professor, studenten):
        self.titel = t
        self.professor = p
        self.studenten = []
        self.studenten.extend(s)
```

Listing 2.2: Beispieldatenbank als Python Code

2.2 Kriterien für Anfragesprachen

In diesem Abschnitt stelle ich Kriterien für objektorientierte Anfragesprachen für die ZODB vor. Diese richten sich nach [HMH91] und [HS99, Abschnitt 6.1].

- *Ad-Hoc-Formulierung*
  Der Benutzer soll eine Anfrage formulieren können, ohne ein vollständiges Programm schreiben zu müssen.

- *Deskriptivität*
  Der Benutzer soll angeben, *was* er anfragen will, nicht *wie* es aus der DB auszulesen ist.

- *Mengenorientiert*
  Eine Anfragesprache sollte Operatoren zur Verfügung stellen, die auf Mengen von Objekten arbeiten und nicht nur auf einzelnen Elementen (one-tuple-at-a-time).

- *Abgeschlossenheit*
  Das Ergebnis einer Anfrage ist im Datenmodell darstellbar.

- *Orthogonalität und Erweiterbarkeit*
  Sprachkonstrukte sollten frei kombinierbar sein. Desweiteren sollte es einfach möglich sein, neue Sprachkonstrukte und -elemente der Sprache hinzuzufügen.

- *Vollständigkeit*
  Die Sprache sollte vollständig sein, das heißt sie muss mindestens die Anfragen der objekterhaltenden Semantik (statische Klassifizierung und Typisierung, siehe Abschnitt 1.3) ausdrücken können. Diese werden im Anforderungskatalog (Abschnitt 2.3.1) näher erläutert.

- *Sicherheit*
  Anfragen, die syntaktisch korrekt sind, müssen in endlicher Zeit verarbeitbar sein und mit endlichem Speicher auskommen.

- *Optimierbarkeit*
  Für die Operationen gibt es mathematische oder anders geartete Optimierungsregeln (die nicht Gegenstand dieser Diplomarbeit sind).

- *Effizienz*
  Die Operationen sollten effizient implementierbar sein. Im Relationenmodell hat zum Beispiel jede Operation eine Komplexität von maximal $O(n^2)$ (n Anzahl der Tupel einer Relation). Die Komplexität im Objektmodell ist höher, da die Daten aufgrund der Speicherung in Objekten für Anfragesprachen ineffizient zugreifbar sind. Die Effizienz kann hier durch Indexstrukturen (siehe Abschnitt 3.2) gesteigert werden.

- *Eingeschränktheit*
  Die Anfragesprache darf keine komplette Programmiersprache sein. Diese Eigenschaft folgt aus *Ad-Hoc-Formulierung*, *Sicherheit*, *Optimierbarkeit* und *Effizienz*.

2.3 Anforderungskatalog und Bewertungsraster

Im Folgenden wird ein Anforderungskatalog sowie ein dazugehöriges Bewertungsraster für Anfragesprachen vorgestellt. Der Anforderungskatalog stellt dabei eine Liste von Eigenschaften dar, die eine geeignete Anfragesprache implementieren sollte. Das Bewertungsraster beschreibt anschließend, wie die Anforderungen bewertet werden können, um die evaluierten Sprachen miteinander vergleichen zu können.

2.3.1 Anforderungen an Anfragesprachen für Objektgraphen

Zur Bewertung der unter 2.4 vorgestellten Anfragesprachen werden Anforderungen an eine solche Sprache benötigt. Diese richten sich nach den Kriterien aus Abschnitt 2.2 sowie nach [CFMR00], [Qua98] und [Kö01].

2.3.1.1 Anforderungen an die Syntax

Verständlichkeit

Einfache Anfragen müssen einfach formulierbar sein. Die Anfragesprache sollte für erfahrene Benutzer und nicht ausschließlich Datenbankexperten verständlich sein. Diese Anforderung folgt aus *Ad-Hoc-Formulierung* und *Deskriptivität*.

Deklarativ

Es wird eine deklarative Anfragesprache gesucht. Diese Anforderung folgt aus *Deskriptivität*. Es werden nur Sprachen evaluiert, die diese Anforderung erfüllen, sodass sie im Bewertungsraster nicht explizit erwähnt wird.

2.3.1.2 Anforderungen an die Funktionalität

Projektion

Ein Objektgraph besteht aus Objekten und Kanten. Es wird eine Möglichkeit benötigt, Pfade[1] im Objektgraphen sowie die Beziehungen zwischen den Objekten (zum Beispiel die *Vater-Sohn-*Beziehung) anzufragen. Diese Anforderung folgt aus *Vollständigkeit*.

Beispiel: "Studenten, die in Kursen eingeschrieben sind."

Selektion

Es wird eine Möglichkeit benötigt, nach den Werten von Attributen der Objekte zu selektieren. Diese Anforderung folgt aus *Vollständigkeit*.

Beispiel: "Studenten, die älter als 30 Jahre alt sind."

Quantifizierung

Die Anfragesprache sollte es erlauben, Selektionen über die Relationen zwischen den Objekte zu spezifizieren. Folgt aus *Vollständigkeit*.

[1] Ein Pfad in einem Graphen ist der Weg von einem Knoten u über die Knoten $u_1, u_2, \ldots u_n$ zum Knoten v.

Beispiel 1: "Kurse, die von mindestens 10 Studenten besucht werden."

Beispiel 2: "Kurse, die von keinem Studenten besucht werden."

Aggregation

Aggregation bezeichnet die Reduktion von mehreren Objekten auf einen Wert oder ein Objekt.

Beispiel 1: "Der Student, der am Ältesten ist."

Beispiel 2: "Der Kurs mit den meisten Studenten."

Verbund

Eine weitere wichtige Operation ist die Vereinigung von Ergebnismengen, um mehrere Teilanfragen zu komplexeren Anfragen zu kombinieren.

Beispiel 1: "Studenten, die älter als 25 Jahre sind und den Kurs Softwaretechnik besuchen."

Beispiel 2: "Studenten, die älter als 25 Jahre sind oder den Kurs Softwaretechnik besuchen."

Gruppierung und Sortierung

Die Objekte in der Ergebnismenge sollte gruppiert und sortiert werden können.

Beispiel: "Studenten absteigend nach Alter sortiert."

2.3.1.3 Weitere Anforderungen

Robustheit

Die Anfragesprache sollte robust gegenüber unerwarteten Zuständen der Objekte und fehlerhaften Anfragen sein. Es muss im Fehlerfall mindestens eine aussagekräftige Meldung ausgegeben werden. Folgt aus *Effizienz* und *Sicherheit*.

Beispiel: "Gib mir alle Studenten, die einen grünen Pullover tragen."[2]

Objektorientiert

Die Anfragesprache sollte direkt mit objektorientierten Datenstrukturen (im Idealfall mit Objektgraphen) arbeiten. Das bedeutet unter anderem, dass der Zugriff auf eine solche Datenstruktur nicht über ein Object-Relational Mapping (ORM)[3], stattfindet. Richtet sich nach *Mengenorientiert*, *Abgeschlossenheit*, *Orthogonalität und Erweiterbarkeit* und *Vollständigkeit*.

Mächtigkeit

Eine Anfragesprache A ist mächtiger als eine Anfragesprache B, wenn "A den Datenbestand schärfer trennt als B, wenn also die Menge der in A bildbaren Suchergebnismengen die Menge der in B

[2] Die Anfrage greift auf ein Attribut zu, welches im Objekt nicht existiert. Die Implementation muss diesen fehlerhaften Zugriff abfangen und eine aussagekräftige Fehlermeldung ausgeben.

[3] Object-Relational Mapping (zu deutsch: Objektrelationale Abbildung) "ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann." [Wik07c]

bildbaren Suchergebnismengen umfasst.” [Wik07a] Die Bestimmung der Mächtigkeit einer Sprache A geht also nur über den Vergleich ihrer Funktionalität mit der der Sprache B. Da die Evaluation genau dies tut, entspricht deren Ergebnis der Mächtigkeit der evaluierten Anfragesprache.

2.3.2 Bewertungsraster für die Anforderungen

2.3.2.1 Allgemeine Informationen zum Bewertungsraster

Im Zuge der Bewertung muss jede Anfragesprache mit den Anforderungen aus 2.3.1 evaluiert werden. Hierfür wird nun ein Bewertungsraster vorgestellt, welches für jede Anforderung einen Standard definiert, der der gewünschten Funktionalität je Anforderung entspricht und mit 0 Punkten (neural) bewertet wird.

Weiterhin werden Gesichtspunkte aufgezeigt, welche eine Erhöhung um maximal **(+2)** oder eine Verringerung um maximal **(–2)** Punkte rechtfertigen. Anschließend werden die Anforderungen mit einem Multiplikator zwischen **(1)** und **(3)** gewichtet.

Für Eigenschaften (positiv als auch negativ), die so speziell sind, dass sie im Bewertungsraster nicht auftauchen, gibt es im Bewertungsraster das zusätzliche Kriterium **Zusatzpunkte**. Für diese können, wie bei allen anderen Anforderungen auch, maximal ± 2 Punkte vergeben werden. Zusatzpunkte haben einen Multiplikator von **(1)**.

2.3.2.2 Das Bewertungsraster im Detail

Verständlichkeit der Syntax

Als Standard dient eine strukturierte und verständliche Syntax, zum Beispiel die `SELECT-FROM-WHERE`-Syntax von SQL.

(+2) Syntax ist weit verbreitet und standardisiert.

(+1) Syntax ist leicht verständlich oder gut dokumentiert.

(–1) Syntax ist unverständlich oder schlecht dokumentiert.

(–2) Syntax ist unvollständig.

Projektion

Als Standard wird eine Sprache definiert, die folgende Funktionalitäten unterstützt:

- Anfrage von (Teil-)Pfaden im Graphen
- Anfrage von Teilpfaden, die über bestimmte Attribute erreichbar sind

- Anfrage von Beziehungen zwischen Objekten

(+2) Funktionalität geht über die Anforderungen hinaus

(–1) Funktionalität bleibt hinter den Anforderungen

(–2) Strukturanfragen sind nicht möglich

Selektion

Als Standard wird definiert, dass die Sprache eine Möglichkeit bietet, nach bestimmten Werten der Attribute zu filtern und mindestens Vergleiche auf (Un-)Gleichheit, Größer-als und Kleiner-als ermöglicht.

(+2) Vergleichsmöglichkeiten gehen über die Anforderungen hinaus

(–1) Vergleichsmöglichkeiten bleiben hinter den Anforderungen

(–2) Selektionen während der Anfrage sind nicht möglich.

Quantifizierung

Als Standard wird definiert, dass die Anfragesprache die Quantifizierer “mindestens `N`”, “maximal `N`”, “alle” und “kein” implementiert.

(+2) Weitere Quantifizierer stehen zur Auswahl

(–1) Es sind nicht alle angeforderten Quantifizierer verfügbar

(–2) Quantifizierung ist nicht möglich

Aggregation

Als Standard wird definiert, dass die Aggregatfunktionen “Minimum” und “Maximum” verfügbar sind.

(+2) Aggregatfunktionen gehen über die Anforderungen hinaus

(–1) einige Aggregatfunktionen fehlen

(–2) keine Aggregatfunktionen vorhanden

Verbund

Als Standard wird definiert, dass mindestens die Verbundoperationen Union (Vereinigungsmenge)[4] und Intersection (Schnittmenge)[5] zur Verfügung stehen.

(+2) Es existieren weitere Verbundoperationen

(–1) Eine der geforderten Verbundoperationen ist nicht verfügbar

(–2) Keine Verbundoperation implementiert

Gruppierung und Sortierung

Es wird als Standard definiert, dass das Ergebnis gruppiert oder sortiert werden kann.

(+2) Gruppierung und Sortierung sind möglich

(+1) Sortierung nach mehreren Attributen ist möglich

(–2) keine Gruppierungs- und Sortierungsmöglichkeit

Robustheit

Als Standard wird definiert, dass die Sprache bei Anfragen auf unerwartete Daten nicht in unerwartete Zustände fällt sondern mindestens eine aussagekräftige Fehlermeldung ausgibt. Hierbei spielt auch die Dokumentation der Fehlerzustände eine große Rolle.

(+2) Es gibt eine große Menge an Fehlerzuständen, die abgefangen werden. Der Anwender wird über verständliche Meldungen auf Fehler hingewiesen.

(–2) Fehler werden nicht abgefangen. Die Implementierung stürzt in einen unbenutzbaren Zustand ab.

Objektorientiert

Als Standard wird definiert, dass die Anfragesprache mit objektorientierten Daten umgehen kann.

(+2) Die Anfragesprache ist für objektorientierte Datenbanken konzipiert und unter diesem Gesichtspunkt ideal geeignet.

(–2) Die Anfragesprache kann nicht für objektorientierte Datenbanken eingesetzt werden.

[4] Die Vereinigungsmenge zweier Mengen enthält alle Elemente aus beiden Mengen.
Beispiel: A = (1,2,3), B = (3,4,5), A ∪ B = (1,2,3,4,5)

[5] Die Schnittmenge zweier Mengen enthält alle Elemente, die in beiden Mengen vorkommen.
Beispiel: A = (1,2,3), B = (3,4,5), A ∩ B = (3)

2.3.2.3 Gewichtung des Bewertungsraster

Wie unter 2.3.2.1 erwähnt soll in diesem Kapitel jedem Punkt im Bewertungsraster eine Gewichtung zwischen 1 und 3 zugeordnet werden. Diese Gewichtung richtet sich nach dem Stellenwert im Gesamtkontext.

- *Verständlichkeit* – Multiplikator: **1**
  Die Verständlichkeit spielt eine eher untergeordnete Rolle. Sie ist nicht entscheidend für die Funktionalität der Sprache.

- *Projektion* – Multiplikator: **3**
  Die Art und Weise, wie auf Beziehungen im Objektgraphen zugegriffen wird, ist für die Funktionalität von enormer Bedeutung.

- *Selektion* – Multiplikator: **3**
  Das Selektieren ist unerlässlich für eine Anfragesprache.

- *Quantifizierung* – Multiplikator: **1**
  Die Quantifizierung ist unterstützend für die Selektion, aber nicht zwingend nötig.

- *Aggregation* – Multiplikator: **1**
  Aggregationsfunktionalität ist unterstützend, aber für die Anfragen nicht zwingend.

- *Verbund* – Multiplikator: **2**
  Der Verbund ist ein wichtiges Instrument, mehrere Anfragen zu verknüpfen und damit erweiterte Anfragen zu ermöglichen.

- *Gruppierung und Sortierung* – Multiplikator: **2**
  Gruppierung und Sortierung sind nützliche Werkzeuge, um die Ergebnismenge aufzubereiten.

- *Robustheit* – Multiplikator: **2**
  Robustheit ist sehr wichtig für eine Anfragesprache, ist aber stark von der Implementierung abhängig und lässt sich somit im Vorfeld schwer abschätzen. Deshalb nur ein Faktor von 2, um Sprachen, die in diesem Bereich schlecht dokumentiert sind, nicht zu stark zu benachteiligen.

- *Objektorientiert* – Multiplikator: **3**
  Die Sprache muss mit Objekten und deren Eigenschaften umgehen können. Dieses Kriterium ist sehr wichtig.

2.4 Evaluation der Anfragesprachen

2.4.1 XML Query Language (XML-QL)

2.4.1.1 Hintergrund

Die XML Query Language (XML-QL) [DFF+98] wurde von Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy und Dan Suciu als Anfragesprache für XML vorgestellt und 1998 beim W3C eingereicht. Heute spielt sie nur noch eine untergeordnete Rolle in der Anfrage von XML-Daten. Allerdings flossen viele Ideen in andere Sprachen wie zum Beispiel XQuery ein.

2.4.1.2 Aufbau

XML-QL besteht aus zwei Hauptteilen, dem `WHERE`- und dem `CONSTRUCT`-Block. Im `WHERE`-Block wird definiert, was angefragt werden soll, während im `CONSTRUCT`-Block die Struktur des Ergebnisses festgelegt wird. Innerhalb dieser Blöcke wird eine XML-ähnliche Syntax verwendet. Ein einfaches Beispiel zeigt Listing 2.3.

```
WHERE <student alter=$alter></student>
      nachname = "Mustermann", vorname = "Max"
CONSTRUCT $alter
```

Listing 2.3: XML-QL: Auslesen des Alters einer bestimmen Person

Variablen beginnen mit einem `$`-Zeichen, um sie von Zeichenketten (wie zum Beispiel 'Mustermann') unterscheiden zu können. Der Query aus Listing 2.3 sucht alle Personen mit dem Namen 'Max Mustermann' und gibt deren Alter zurück.

In Listing 2.4 wird die XML-ähnliche Struktur am Beispiel der Anfrage von geschachtelten Strukturen deutlicher: Es werden alle Kurse gesucht, in denen 'Max Mustermann' eingeschrieben ist. Listing 2.5 zeigt ein mögliches Ergebnis der Anfrage aus 2.4.

```
WHERE <kurs titel=$ergebnis>
        <student></>
        nachname = "Mustermann", vorname = "Max"
      </>
CONSTRUCT <kurse_von_Max_Mustermann>
            $ergebnis
          </>
```

Listing 2.4: XML-QL: Anfrage von geschachtelten Strukturen

Kriterium	**Bewertung**	**Multiplikator**	**Gesamt**
Syntax			
Verständlichkeit	$+1$	1	$+1$
Funktionalität			
Projektion	± 0	3	± 0
Selektion	± 0	3	± 0
Quantifizierung	-2	1	-2
Aggregation	-2	1	-2
Verbund	-1	2	-2
Gruppierung und Sortierung	± 0	2	± 0
weitere Anforderungen			
Robustheit	-2	2	-4
Objektorientiert	± 0	3	± 0
Zusatzpunkte	$+1$	1	$+1$
Gesamt			**−8**

Tabelle 2.1: Ergebnis der Evaluation von XML-QL

```
<kurse_von_Max_Mustermann>
  Medienwerkzeuge
  Softwarepraktikum
</kurse_von_Max_Mustermann>
```

Listing 2.5: XML-QL: Beispielresultat der Anfrage aus 2.4

XML-QL bietet weiterhin die Möglichkeiten, mit Hilfe geschachtelter Anfragen Ergebnisse zu gruppieren (wie das Statement `GROUP BY` in SQL), Anfragen mittels Joins zu verbinden sowie eigene Funktionen zu definieren und zu nutzen. Diese sollen hier allerdings nicht näher vorgestellt werden, eine ausführliche Erklärung sowie weitere Informationen findet man unter [DFF+98].

2.4.1.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.1. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die Syntax von XML-QL ist sehr strukturiert und verständlich. Anfragen werden über XML-ähnliche-Tags realisiert, welche von XML-QL-spezifischen Befehlen umschlossen sind. Die Dokumentation ist verständlich, aber weniger ausführlich. **(+1)**

- *Projektion:*
  Eine Anfrage von (Teil-)Pfaden sowie Pfaden über Attribute ist möglich, da XML-QL mit hierarchisch strukturiertem Markup im Query arbeitet. Auch Beziehungen lassen sich damit anfragen.

- *Selektion:*
  Eine Filterung in XML-QL ist implizit, da der `WHERE`-Block fester Bestandteil einer Anfrage ist. Vergleiche auf (Un-)Gleichkeit sowie Relationen sind möglich.

- *Quantifizierung:*
  Quantifizierungsoperatoren sind nicht vorhanden. **(–2)**.

- *Aggregation:*
  Operatoren für Aggregationen sind nicht vorhanden, sollen aber in einer der nächsten Versionen implementiert werden. **(–2)**

- *Verbund:*
  Es ist möglich, über Joins Schnittmengen von Anfrageergebnissen zu bilden. Weitere Verbundoperatoren sind nicht verfügbar. **(–1)**

- *Gruppierung und Sortierung:*
  Ergebnisse können nach Attributen gruppiert werden, eine Sortierung ist nicht vorgesehen.

- *Robustheit:*
  Wird in der Dokumentation komplett außen vor gelassen. Es kann daher nichts darüber gesagt werden, wie robust sich XML-QL verhält. **(–2)**.

- *Objektorientierung:*
  XML-QL unterstützt keine reine Objektorientierung, aber die XML-Syntax der Anfragen und Ergebnisse ähnelt der von Objekten. Es müssen daher Anpassungen vorgenommen werden, damit XML-QL mit Objektgraphen umgehen kann.

- *Zusatzpunkte:*
  Es können über den Operator `FUNCTION` eigenen Funktionen definiert werden. **(+1)**

2.4.1.4 Fazit

Die Evaluation stellt klar heraus, dass XML-QL als Anfragesprache für die ZODB **nicht geeignet** ist. Es kristallisieren sich drei entscheidende Schwachstellen heraus:

- Mangelnde Informationen über die Robustheit gegenüber unerwarteten Daten[6].

[6] Siehe: Dokumentation unter [DFF+98]

- Die fehlende Unterstützung für Quantifizierung und Aggregationen.

- Der Ein- und Ausgabe von XML-Code anstelle von Objekten.

2.4.2 XML Path Language (XPath)

2.4.2.1 Hintergrund

XML Path Language (XPath) [CD99] wurde 1999 von James Clark und Steve DeRose vorgestellt und vom W3C als Recommendation[7] befürwortet. Die Hauptfunktionalität von XPath ist die Adressierung von (Teil-)Pfaden eines XML-Dokumentes. Zur Unterstützung gibt es Funktionalität für String-, Zahlen- und Wahrheitswertmanipulationen. XPath besitzt seine eigene nicht-XML-Syntax, die stark an die Pfadangaben in UNIX-Dateisystemen oder URLs erinnert.

XPath nutzt für das Testen von Elementen (ob ein Element einem Ausdruck entspricht oder nicht) die Extensible Stylesheet Language Transformations (XSLT). XSLT ist in [Cla99] beschrieben und wird in der Diplomarbeit nicht evaluiert. Es genügt zu wissen, dass diese Funktionalität existiert und XPath sie nutzen kann.

2.4.2.2 Aufbau

Lokalisierungspfade

Lokalisierungspfade bezeichnen das wichtigste Konstrukt in XPath. Man unterscheidet die verkürzte und unverkürzte Syntax. Tabelle 2.2 stellt beide Varianten anhand von Beispielen gegenüber. Die verkürzte Syntax ist sehr stark an XSLT angelehnt.

Bei den Lokalisierungspfaden wird zwischen zwei Varianten unterschieden:

- *Relativer Lokalisierungspfad*
  Er besteht aus einem oder mehreren Lokalisierungsschritten, die durch das Pfadtrennzeichen '`/`' von einander separiert sind. Lokalisierungsschritte werden von links nach rechts zusammengesetzt. Es wird in jedem Schritt eine Menge von Knoten relativ zum Kontextknoten gewählt. Dabei ist der Kontextknoten der Knoten, der sich aus dem Vorgängerschritt ergibt.

- *Absoluter Lokalisierungspfad*
  Der absolute Lokalisierungspfad besteht aus der Wurzel '`/`' gefolgt von einem optionalen relativen Lokalisierungspfad.

[7] Das W3C nennt seine fertigen Arbeiten W3C-Recommendations, also W3C-Empfehlungen, da es keine zwischenstaatliche Organisation und daher nicht berechtigt ist, Standards festzulegen. Vgl. [Wik08k, Abschnitt 1]

Syntax (verkürzt)	Beschreibung
`child::person` `(person)`	Wählt alle `person`- Kindelemente des Kontextknotens
`child::*` `(*)`	Wählt alle Kindelemente des Kontextknotens
`attribute::name` `(@name)`	Wählt das `name`- Attribut des Kontextknotens
`child::person[position()=1]` `(person[1])`	Wählt das erste `person`-Kind-Element des Kontextknotens
`child::person/descendant::email` `(person//email)`	Wählt alle `email`-Element Nachkommen der `person`-Kind- Elemente des Kontextknotens
`self::node()` `(.)`	Wählt den Kontextknoten
`parent::node()` `(..)`	Wählt das Eltern-Element des Kontextknotens
`/` `(/)`	Wählt den Wurzelknoten (root)

Tabelle 2.2: Verkürzte und unverkürzte (in runden Klammern) Lokalisierungspfade in XPath

Lokalisierungsschritte

Lokalisierungsschritte bestehen aus drei Teilen:

- einer Achse, die die Beziehung zwischen dem Knoten, der im Lokalisierungsschritt ausgewählt werden soll, und dem Kontextknoten ausdrückt
- einem Knotentest, der den Knotentyp und einen Namen des Knotens angibt
- keinem oder mehreren Prädikaten, die über Ausdrücke die Menge an Knoten einschränken

Die Syntax eines Lokalisierungsschrittes besteht aus einem Achsennamen und Knotentest, beide getrennt durch zwei Doppelpunkte (`::`), gefolgt von den optionalen Prädikaten, die in eckigen Klammern stehen (`[]`). Zum Beispiel ist in `child::student[alter=25]` 'child' der Achsennamen, 'student' der Knotentest und '[alter=25]' das Prädikat. Gelesen wird der Lokalisierungsschritt von links nach rechts, das bedeutet im aktuellen Beispiel, dass zuerst alle Kinder mit dem Namen 'student' des aktuellen Kontextknotens ermittelt und anschließend all jene Kinder, die das Attribut 'alter' mit dem Wert '25' besitzen, in die Ergebnismenge aufgenommen werden. Tabelle 2.3 zeigt eine Liste mit den wichtigstens Bezeichnern für Achsen[8].

[8]Eine ausführliche Liste ist in [CD99] unter `http://www.w3.org/TR/xpath#axes` verfügbar.

Bezeichner	Beschreibung
`child`	enthält die direkten Nachkommen des Kontextknotens
`descendant`	enthält alle Nachkommen des Kontextknotens
`parent`	enthält den direkten Vorgänger des Kontextknotens
`ancestor`	enthält alle Vorgänger des Kontextknotens
`attribute`	enthält alle Attribute des Kontextknotens (nur bei Elementen)
`self`	enthält den Kontextknoten

Tabelle 2.3: Wichtige Achsen und ihre Beschreibung in XPath

Funktionen

XPath unterstützt eine ganze Reihe von Funktionen zum Umgang mit Knotenmengen, für Zeichenkettenmanipulation und -vergleiche sowie Wahrheits- und Nummerische Werte. Übersicht 2.4 zeigt ausgewählte Funktionen und ihre Beschreibung. In ihr fehlen unter anderem die Funktionen für Namensräume, die in XML eine wichtige, in Objektgraphen allerdings keine Rolle spielen[9].

2.4.2.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.5. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die Syntax von XPath ist sehr umfangreich und trotzdem gut verständlich. Man kann mit wenig Wissen und einer kleinen Menge an Befehlen eine Vielzahl an Anfragen formulieren und sich für ausführlichere Anfragen im Pool der mitgelieferten Funktionen bedienen. Weiterhin ist die Syntax vom W3C als Empfehlung verabschiedet und dadurch als quasi-Standard etabliert. **(+2)**

- *Projektion:*
  XPath bietet mit den *Lokalisierungspfaden* eine flexible und verständliche Möglichkeit, XML-Strukturen anzufragen. Sie bieten dem Anwender unterstützt durch die *Funktionen für Knotenmengen* (siehe 2.4 eine Fülle an Möglichkeiten, den Graphen des DOM zu zerlegen. **(+2)**

[9] Die Funktionen für Namensräume können bei Bedarf unter `http://www.w3.org/TR/xpath#corelib` nachgelesen werden.

[10] Das nodeset ist eine unsortierte Liste von Knoten.

Funktion	**Beschreibung**
Funktionen für Knotenmengen	
num `last()`	Liefert eine Zahl entsprechend der *Kontextgröße* des aktuellen Ausdrucks ($\widehat{=}$ der Anzahl der Knoten der Knotenmenge).
num `position()`	Liefert die *Position* des aktuellen Kontextknotens.
num `count(`*nodeset*`)` [10]	Liefert die *Anzahl Knoten* der übergebenen Knotenmenge.
str `name(`*nodeset?*`)`	Liefert den *Namen* des ersten Knotens in der übergebenen Knotenmenge bezüglich der Dokumentenordnung. Wird kein Argument übergeben, wird der Name des aktuellen Kontextknotens zurückgegeben.
Funktionen für Zeichenketten	
str `string(`*obj*`)`	Wandelt ein Objekt in eine Zeichenkette um.
str `concat(`*str,str,str**`)`	Liefert die Verkettung der Argumente zurück.
bool `contains(`*str,str*`)`	Liefert *True*, falls die erste Zeichenkette die zweiten enthält, ansonsten *False*.
num `string-length(`*str?*`)`	Liefert die Anzahl Zeichen des übergebenen Argumentss. Wird kein Argument übergeben, so wird die Länge des Namens des aktuellen Kontextknotens zurückgeliefert.
Funktionen für Wahrheitswerte	
bool `boolean(`*obj*`)`	Wandelt ein Objekt in einen Wahrheitswert um.
bool `not(`*bool*`)`	Liefert *True*, wenn das Argument *False* ist, ansonsten *False*.
bool `true()`	Liefert *True*
bool `false()`	Liefert *False*
Funktionen für numerische Werte	
num `number(`*obj*`)`	Wandelt ein Objekt in eine Zahl um.
num `sum(`*nodeset*`)`	Berechnet die Summe der in eine Zahl umgewandelten Namen der Knoten der übergebenen Knotenmenge.
num `floor(`*num*`)`	Liefert den kleinsten ganzzahligen Wert, der nicht größer als das Argument ist.
num `ceiling(`*num*`)`	Liefert den kleinsten ganzzahligen Wert, der nicht größer als das Argument ist.
num `round(`*num*`)`	Rundet das Argument auf einen ganzzahligen Wert

Tabelle 2.4: Ausgewählte Funktionen in XPath

Kriterium	**Bewertung**	**Multiplikator**	**Gesamt**
Syntax			
Verständlichkeit	+2	1	+2
Funktionalität			
Projektion	+2	3	+6
Selektion	±0	3	±0
Quantifizierung	±0	1	±0
Aggregationen	+2	1	+2
Verbund	±0	2	±0
Gruppierung und Sortierung	−2	2	−4
weitere Anforderungen			
Robustheit	+2	2	+4
Objektorientiert	+2	3	+6
Zusatzpunkte	±0	1	±0
Gesamt			**+16**

Tabelle 2.5: Ergebnis der Evaluation von XPath

- *Selektion:*
  Mittels der Prädikatausdrücke (`[]`) können Objekte über Eigenschaften direkt selektiert werden. Alle geforderten Vergleichsmöglichkeiten sind verfügbar.

- *Quantifizierung:*
  Über die Aggregatfunktion `count()` kann die Anzahl von Objekten in einem nodeset bestimmt werden. So wählt zum Beispiel `/kurs[count(student) > 10)]` alle Kurse, in denen mehr als 10 Studenten eingeschrieben sind. Es ist in XPath möglich, alle geforderten Quantifizierungen anzufragen.

- *Aggregation:*
  Es werden die Aggregatfunktionen `avg()`, `max()`, `min()` und `sum()` unterstützt. XPath geht daher über die Anforderungen hinaus. **(+2)**

- *Verbund:*
  Es ist möglich, sowohl Schnitt- als auch Vereinigungsmengen in XPath zu bilden. Schnittmengen werden über Unteranfragen in Prädikaten, Vereinigungen über den Operator `|`, realisiert.

- *Gruppierung und Sortierung:*
  XPath liefert standardmäßig eine Liste von Knoten (sogenannte *nodesets*) zurück. Diese ist unsortiert. Eine Gruppierfunktion ist nicht vorhanden. **(–2)**

- *Robustheit:*
  XPath geht mit unerwarteten Daten sehr differenziert um, versucht aber, möglichst jeden Fall fehlerhafter Anfragen/Parameter abzufangen. Die Dokumentation [CD99] zeigt zum Beispiel für jede Funktion eine Übersicht an Rückgabewerten, falls ein Parameter nicht dem erwarteten Format oder Wert entspricht. **(+2)**

- *Objektorientierung:*
  XPath arbeitet auf dem DOM eines XML-Dokumentes. Die Unterstützung für den Umgang mit einem Objektgraphen ist daher sehr gut. **(+2)**

2.4.2.4 Fazit

XPath ist sehr gut **geeignet** als Anfragesprache für die ZODB. Alle Kriterien (bis auf Gruppierung und Sortierung) entsprechen den Forderungen oder sind über diese hinaus erfüllt. Positiv aufgefallen sind die Lokalisierungspfade, die eine intuitive Möglichkeit darstellen, Strukturen im Objektgraphen anzufragen.

2.4.3 Regular Path Expressions (RPE)

2.4.3.1 Hintergrund

Die Regular Path Expressions (RPE) wurden im Rahmen der Vorstellung eines neuen Systems zur Indizierung und Anfrage von XML-Daten, dem XML Indexing and Storage System (XISS) [LM01], ausgearbeitet und vorgestellt. Sie sind vordergründig eine Demonstrationssprache, die sehr stark an die Lokalisierungspfade aus XPath erinnern.

2.4.3.2 Aufbau

Hauptbestandteil der RPE-Syntax sind Pfadausdrücke, welche sehr stark an die verkürzten Lokalisierungspfade in XPath angelehnt sind. Diese bieten die Möglichkeit, direkte (Teil-)Pfade in Objekthierarchien sowie Attributwerte von Objekten anzufragen. Listing 2.6 zeigt ein einfaches Beispiel, bei dem alle Studenten mit dem Vornamen ‘Christian’ aus der Beispieldatenbank gelesen werden.

```
/student[@vorname="Christian"]
```

Listing 2.6: RPE: Filtern nach Vorname der Studenten

Attribut-Zugriffe stehen wie in XPath in eckigen Klammern (Prädikatausdrücke). Der Attributname wird durch ein vorangestelltes `@`-Zeichen klassifiziert, der Attributwert von doppelten

Symbol	Funktion des Symbols
_	Bezeichnet irgendeinen einzelnen Knoten.
/	Trennzeichen zwischen Knoten in einem Pfad.
|	Vereinigung (Union) von Knoten.
?	Kein oder ein Vorkommen eines Knotens.
+	Ein oder mehrere Vorkommen eines Knotens.
*	Kein oder mehrere Vorkommen eines Knotens.
[]	Umschließt einen Prädikatausdruck.
@	Bezeichnet Attribute.
()	Bezeichnet Vorrang.

Tabelle 2.6: Mögliche Notationen in RPE

Anführungsstrichen umschlossen. Tabelle 2.6 zeigt eine Liste mit Notationen, die in RPE möglich sind.

Listing 2.7 zeigt ein Beispiel für Vereinigung in RPE. Die dargestellte Anfrage liefert alle Studenten, die auf den Nachnamen ‘Mustermann’ hören oder 25 Jahre alt sind.

```
(/_*/student[@nachname="Mustermann"])|(/_*/student[@alter="25"])
```

Listing 2.7: RPE: Anfrage mit Vereinigung

2.4.3.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.7. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  RPE besitzen eine recht einfache, leicht zu erlernende Syntax. Diese basiert auf den aus XPath bekannten Pfadausdrücken. **(+2)**

- *Projektion:*
  Die aus XPath bekannten Lokalisierungspfade bieten eine effiziente Möglichkeit, Strukturanfragen für Objektgraphen zu formulieren.

- *Selektion:*
  Mit Hilfe der Prädikate kann nach bestimmten Objekten während der Anfrage gefiltert werden. Es ist nur die Möglichkeit vorgesehen, auf Gleichheit zu prüfen. **(–1)**

- *Quantifizierung:*
  Es sind die Quantifizierer ‘mindestens 1’ (+), ‘0 oder 1’ (?) und ‘beliebig viele’ (*) ver-

Kriterium	**Bewertung**	**Multiplikator**	**Gesamt**
Syntax			
Verständlichkeit	$+2$	1	$+2$
Funktionalität			
Projektion	± 0	3	± 0
Selektion	-1	3	-3
Quantifizierung	-1	1	-1
Aggregation	-2	1	-2
Verbund	-1	2	-2
Gruppierung und Sortierung	-2	2	-4
weitere Anforderungen			
Robustheit	-2	2	-4
Objektorientiert	$+2$	3	$+6$
Zusatzpunkte	0	1	0
Gesamt			**- 8**

Tabelle 2.7: Ergebnis der Evaluation der RPE

fügbar. RPE bleiben daher hinter den Anforderungen, da die Quantoren ‘mindestens N’, ‘maximal N’ sowie ‘kein’ nicht unterstützt werden. **(–1)**

- *Aggregation:*
  Es ist keine Aggregationsfunktionalität vorhanden. **(–2)**

- *Verbund:*
  Mit Hilfe des Union-Operators (|) können einfache Joins implementiert werden. Intersection wird nicht unterstützt. **(–1)**

- *Gruppierung und Sortierung:*
  RPE enthalten keine Möglichkeiten, Ergebnismengen zu sortieren oder zu gruppieren. **(–2)**.

- *Robustheit:*
  Hinweise für eine robuste Implementierung werden im XISS-Paper[11] nicht erwähnt. **(–2)**

- *Objektorientierung:*
  RPE arbeiten auf dem DOM eines XML-Dokumentes und können daher mit Objektgraphen umgehen. **(+2)**

[11] Vgl. [LM01]

2.4.3.4 Fazit

RPE sind offensichtlich als produktive Anfragesprache für Objektgraphen aufgrund der sehr geringen Funktionalität **nicht geeignet**. Positiv ist aufgefallen, dass sie aufgrund ihrer schlichten Syntax eine ideale Demonstrationssprache darstellen. Die im XISS-Paper [LM01] vorgestellten Algorithmen und Indexstrukturen kombiniert um diese einfache Sprache stellen ein gutes Konzept als Machbarkeitsstudie zur Anfrage und Indizierung von Objektgraphen dar.

2.4.4 XML-Graphical-Language (XML-GL)

2.4.4.1 Hintergrund

Die XML-Graphical-Language (XML-GL) [CCD+99] ist eine grafische Anfragesprache für XML-Dokumente. Sie basiert auf visuellen Beschreibungen, die auch bei komplexen Anfragen intuitiv benutzbar sein sollen. XML-GL wurde von Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi und Letizia Tanca entwickelt.

2.4.4.2 Aufbau

XML-GL addressiert zwei Schwerpunkte: Formulieren von Anfragen zur Datenextraktion aus XML-Dokumenten und Restrukturieren von Informationen (in der Regel die Ergebnismengen) in neue XML-Dokumente. Anfragen werden über visueller Graphen formuliert, die aus drei Bestandteilen bestehen:

- *Objekte*
  Objekte werden als *Rechtecke* dargestellt. Sie bezeichnen abstrakte Dinge, die keinen darstellbaren Wert besitzen.

- *Eigenschaften*
  Eigenschaften werden als mit einem Objekt verbundene *Kreise* dargestellt. Sie bezeichnen darstellbare Werte (zum Beispiel ein Zeichen oder Zeichenketten). Eigenschaften haben einen Namen und einen Typ.

- *Beziehungen*
  Beziehungen werden als *Pfeile* zwischen Objekten dargestellt. Sie stellen semantische Verbindungen dar und sind gerichtet.

Eine XML-GL-Anfrage besteht aus vier Teilen: extract, match, clip und construct. Diese werden in der Anfrage nebeneinander dargestellt, getrennt durch eine vertikale Linie. Die linke Seite stellt den extract und match Teil dar, die rechte Seite die Teile clip und construct. Damit sind auf der linken Seite alle Informationen zum Auslesen der Informationen und auf der rechten alle zum Aufbauen des Ergebnisses zu finden.

- *Der extract Block*
  Der extract Teil beschriebt den Zuständigkeitsbereich der Anfrage, indem die anzufragenden Elemente angegeben werden. Dies entspricht zum Beispiel dem *FROM*-Teil in SQL.

- *Der match Block*
  Im match Teil, welcher optional ist, werden logische Bedingungen formuliert, die die Elemente erfüllen müssen, um Teil des Ergebnisses zu sein. Entspricht dem *WHERE*-Block in SQL.

- *Der clip Block*
  Im clip Block werden all jene (Unter-)Elemente spezifiziert, die in das Ergebnis übernommen werden sollen. Entspricht dem *SELECT*-Block in SQL.

- *Der construct Block*
  Im optionalen construct-Block wird beschrieben, wie das Ergebnis aufgebaut werden soll.

Im Folgenden werden nun Beispielanfragen vorgestellt. Abbildung 2.2 zeigt eine extract-clip-Anfrage, die alle `student`-Objekte anfragt. Eine Manipulation der Ergebnismenge findet nicht statt, weiterhin werden alle hierarchischen Unterobjekte mit in die Ergebnismenge aufgenommen.

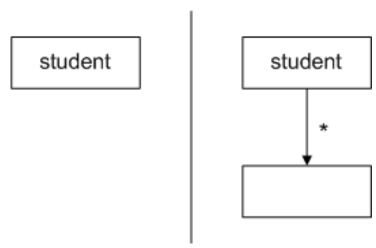

Abbildung 2.2: Auslesen aller student-Objekte in XML-GL

Abbildung 2.3 zeigt einige Möglichkeiten, wie festgelegt werden kann, welche Unterelemente wie ausgelesen werden sollen. Die einzelnen Anfragen werden im Folgenden erklärt:

- alle Unterobjekte der ersten Ebene werden behalten (Abbildung 2.3.a)

- alle Unterobjekte aller Ebenen werden behalten (Abbildung 2.3.b)

- nur Unterobjekte der ersten Ebene vom Typ `PCDATA` werden behalten (Abbildung 2.3.c)

- alle Unterobjekte aller Ebenen vom Typ `PCDATA` werden behalten (Abbildung 2.3.d)

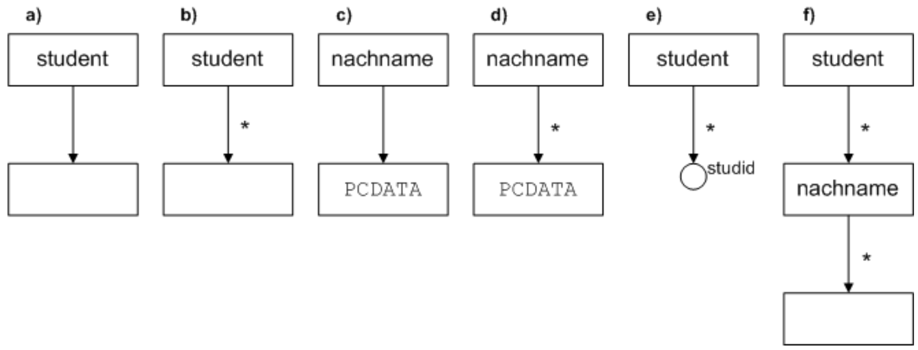

Abbildung 2.3: Notationsmöglichkeiten für die Aufbereitung der Ergebnismenge in XML-GL

- alle Vorkommen eines gegebenen Objektes werden ohne umgebende Objekte behalten (Abbildung 2.3.e und Abbildung 2.3.f)

Eine extract-match-clip-Anfrage ist in Abbildung 2.4 dargestellt. Diese erlaubt durch Angabe von Prädikatbedingungen auf der linken Seite der Anfrage eine Filterung nach Eigenschaften. Es wird in diesem Beispiel das Alter der Studenten gesucht, die älter als 25 sind und den Vornamen 'Max' tragen.

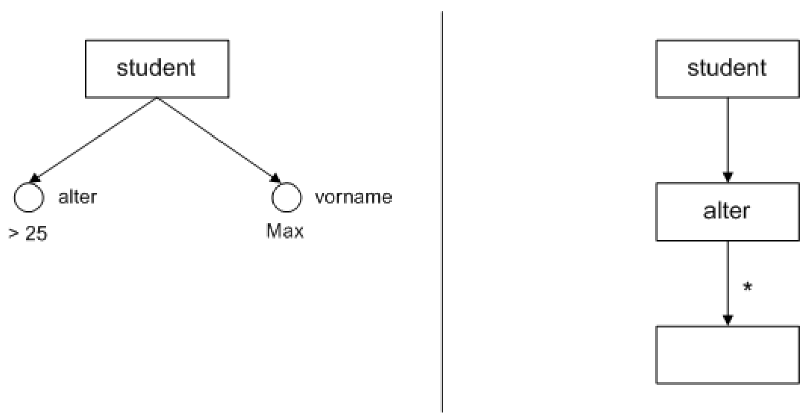

Abbildung 2.4: XML-GL Anfrage mit match-Bedingungen

2.4.4.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.8. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die XML-GL-Syntax besteht aus einer Reihe grafischer Primitive, welche, in Relation gesetzt, eine Anfrage bilden. Diese sind leicht verständlich und gut dokumentiert. **(+1)**

Kriterium	Bewertung	Multiplikator	Gesamt
Syntax			
Verständlichkeit	$+1$	1	$+1$
Funktionalität			
Projektion	± 0	3	± 0
Selektion	$+2$	3	$+6$
Quantifizierung	-2	1	-2
Aggregation	-2	1	-2
Verbund	-1	2	-2
Gruppierung und Sortierung	± 0	2	± 0
weitere Anforderungen			
Robustheit	-2	2	-4
Objektorientiert	$+2$	3	$+6$
Zusatzpunkte	± 0	1	± 0
Gesamt			**+ 3**

Tabelle 2.8: Ergebnis der Evaluation der XML-GL

- *Projektion:*
  Es besteht die Möglichkeit, mit Hilfe der Objekte und Beziehungen Pfadanfragen zu formulieren. Objekteigenschaften können über den match-Block in gefordertem Maß angefragt werden.

- *Selektion:*
  Es sind im match-Block Vergleichs- (>, <, >=, <=, = und <>) sowie Zeichenkettenoperatoren (_ und %) möglich. **(+2)**

- *Quantifizierung:*
  Quantifizierungen werden nicht unterstützt. **(–2)**

- *Aggregation:*
  Es fehlen Aggregatoperatoren wie Minimum und Maximum. **(–2)**

- *Verbund:*
  Schnittmengen können im match-Block gebildet werden. Das Bilden von Vereinigungsmengen ist hingegen nicht möglich. **(–1)**

- *Gruppierung und Sortierung:*
  Im match-Block gibt es die Möglichkeit, mittels `GROUP BY` Anfragen zu gruppieren. Sortierung der Ergebnisobjekte wird nicht unterstützt.

- *Robustheit:*
  Die Dokumentation geht nicht auf das Thema Robustheit ein. Es wird auch nicht erklärt, wie XML-GL robust implementiert werden kann. **(–2)**

- *Objektorientierung:*
  XML-GL arbeitet direkt auf Objekten und deren Eigenschaften (DOM). **(+2)**

2.4.4.4 Fazit

Die Idee, Anfragen auf eine völlig neue Art und Weise, nämlich nicht text- sondern graphenbasiert zu implementieren, ist sehr interessant. Allerdings spricht diese Eigenschaft gegen eine Implementierung von XML-GL als Anfragesprache für die ZODB. Die Evaluation ergab zumindest eine theoretische Eignung, praktisch ist XML-GL allerdings **nicht geeignet**.

2.4.5 Quilt

2.4.5.1 Hintergrund

In die Entwicklung von Quilt [RFC00] flossen die positiven Erfahrungen aus anderen XML-Querysprachen ein. So enthält Quilt unter anderem die Pfadausdrücke aus XPath und die Variablenbindung aus XML-QL. Quilt wurde von Jonathan Robie, Daniela Florescu und Don Chamberlin entwickelt.

2.4.5.2 Aufbau

Grundsätzlich besteht Quilt aus den folgenden Strukturen:

- Pfadausdrücke
- Elementkonstruktoren
- FLWR-Ausdrücke[12]
- Operatoren und Funktionen
- Bedingungen
- Quantoren

Weiterhin sind Kommentare möglich, die, wie in SQL, mit einem doppelten Bindestrich beginnen und bis zum Zeilenende als Kommentar interpretiert werden.

[12]FLWR steht für die Synonyme For, Let, Where und Return. Details folgen auf Seite 36.

Symbol	Funktion des Symbols
.	Zeigt auf den aktuellen Knoten.
..	Zeigt auf den Vater des aktuellen Knotens.
/	Zeigt auf die Wurzel oder auf die Kinder des aktuellen Knotens.
//	Zeigt auf die Nachfolger des aktuellen Knotens.
@	Zeigt auf Attribute des aktuellen Knotens.
*	Zeigt auf “beliebige” (Knoten mit freiem Namen).
[]	Klammern umschließen einen Bool’schen Ausdruck, der als Prädikat für einen Schritt dient.
[n]	Wenn ein Prädikat aus einem Integer besteht, dient es als Selektor aus einer Liste von Knoten.

Tabelle 2.9: Wichtige Symbole in Quilt-Pfadausdrücken

Pfadausdrücke

Die Pfadausdrücke basieren auf der abgekürzten Form der Lokalisierungspfade in XPath. Das Ergebnis eines Pfadausdrucks ist ein sortierter Wald[13] der Knoten, die dem Ausdruck entsprechen, und ihren Nachfolgern.

Wie in XPath besteht ein Pfadausdruck aus einer Reihe von Schritten, die eine Bewegung durch den XML-Baum ermöglichen und in jedem Schritt über nicht zutreffende Prädikatausdrücke den Ergebniswald verkleinern. Das Ergebnis jedes Schrittes ist eine Menge von Knoten, die wiederum den Startpunkt für den nächsten Schritt darstellen. Tabelle 2.9 erklärt die wichtigsten Symbole, die in Quilt-Pfadausdrücken benutzt werden.

Listing 2.8 verdeutlicht den Unterschied zwischen Bool’schen Ausdrücken (`[]`) und Selektoren (`[n]`) in Prädikaten an einem Beispiel. Im ersten Beispiel (Zeile 1) wird der 2. Student im DOM ausgewählt, wohingegen in der Anfrage in Zeile 2 der Student mit dem Nachnamen ‘Mustermann’ angefragt wird.

```
1 /student[2]
2 /student[nachname="Mustermann"]
3 /student[RANGE 2 TO 5]
```

Listing 2.8: Quilt: Differenzierte Nutzungsmöglichkeiten von Prädkatausdrücken

Es gibt ferner die Möglichkeit, über den `RANGE`-Operator (Zeile 3) den Selektor auf einen Zahlenbereich zu erweitern. Weiterhin ist es in Quilt-Pfadausdrücken möglich, mit Hilfe des Dereferenz-Operators (->) Attribute anzugeben, über die im Graph traversiert werden soll.

[13] “Als Wald bezeichnet man in der Graphentheorie einen ungerichteten Graphen ohne Zyklus. Ist dieser zusammenhängend, so spricht man von einem (ungerichteten) Baum. Die Zusammenhangskomponenten eines Waldes stellen in diesem Sinne für sich einen Baum dar, so dass ein Wald aus einem oder mehreren Bäumen besteht. [Wik08j]”

Elementkonstruktoren

Elementkonstruktoren werden vornehmlich zum Erzeugen und Aufbereiten der Ausgabe genutzt. So können in Elementkonstruktoren auch Variablen verwendet werden, die in der vorangegangenen Anfrage gebunden wurden. Beim Erzeugen von Elementen aus Variablen ist darauf zu achten, dass Start- und End-Tag des Elementes den selben Variablennamen enthalten (Listing 2.9).

```
<$variable id=$tagid>
  <name> $vorname $nachname </name>
  <alter> $tel </alter>
</$variable>
```

Listing 2.9: Quilt: Konstruktion von Elementen mit Quilt

FLWR-Ausdrücke

Ein FLWR (sprich: “flower”) Ausdruck besteht aus `FOR`, `LET`, `WHERE` und `RETURN` Klauseln. Ein FLWR Ausdruck wird benötigt, wenn über Elemente einer Menge iteriert werden soll.

Abbildung 2.5 zeigt den prinzipiellen Aufbau eines FLWR Ausdrucks. Ein FLWR Ausdruck beginnt mit einer `FOR` Klausel, die eine oder mehrere Variablen erzeugt. Jede dieser Variablen gehört zu einem Ausdruck (zum Beispiel einem Pfadausdruck), welcher wiederum eine Liste von Knoten zurückliefert. Der `FOR` Klausel können wiederum eine oder mehrere `LET` Klauseln und weitere `FOR` Klauseln folgen. Die `LET` Klausel bindet eine oder mehrere Variablen an das Ergebnis eines oder mehrerer Ausdrücke.

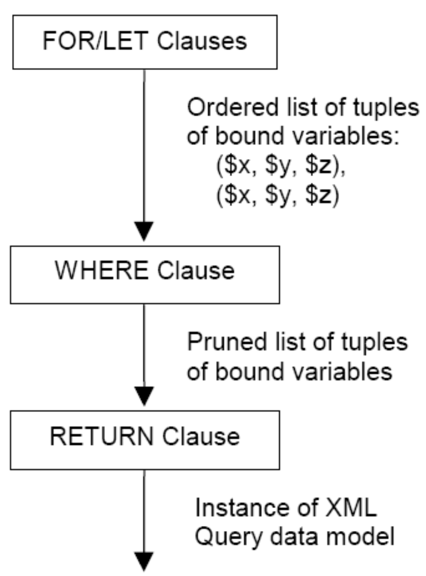

Abbildung 2.5: Datenfluß in einem FLWR-Ausdruck

Das Ergebnis der `FOR` und `LET` Klauseln dient als Eingabe für eine optionale `WHERE` Klausel. Nur jene Tupel, für die die Bedingung der `WHERE` Klausel *wahr* ergibt, werden an die `RETURN` Klausel weitergereicht. Die `WHERE` Klausel kann mehrere Bedingungen, separiert durch `AND`, `OR` und `NOT`, enthalten. Die `RETURN` Klausel generiert die Ausgabe des FLWR Ausdrucks, welche ein Knoten, ein geordneter Knotenwald oder ein einfacher Wert sein kann.

Operatoren und Funktionen

Quilt erlaubt das Erstellen von Ausrücken mittels infix und prefix Operatoren sowie verschachtelte Ausdrücke in Klammern. Quilt unterstützt den üblichen Satz an logischen und arithmetischen Operatoren und die Mengenoperatoren `UNION`, `INTERSECT` sowie `EXCEPT`. Weiterhin gibt es den `FILTER` Operator, welcher eine Untermenge von Knoten aus einer Knotenmenge herauslöst. Quilt enthält außerdem eine Menge vordefinierte Funktionen, unter anderem alle Funktionen aus XPath, eine Vielzahl an Aggregatfunktionen (zum Beispiel avg, sum, count, max und min) und einige andere nützliche Funktionen (zum Beispiel distinct, empty). Eigene Funktionen können zusätzlich definiert werden, sind aber nur im "lokalen Rahmen", das heißt in der Anfrage, in der sie definiert werden, verfügbar.

Bedingungen

Bedingungen sind nützlich, wenn die Informationen, die zurückgegeben werden, von einigen Bedingungen abhängen. Bedingungen können, wie in fast jeder Sprache, auch in Quilt geschachtelt sein. Sie haben die Form `IF` *Bedingung* `THEN` *Bedingung ist wahr* `ELSE` *Bedingung ist unwahr*.

Quantoren

Zum effizienten Testen, ob Elemente in einer Liste eine Bedingung erfüllen, existieren in Quilt die Existenzquantor (`SOME $element IN $liste SATISFIES $bedingung`) und der Universalquantor (`EVERY $element IN $liste SATISFIES $bedingung`). Beide Quantoren liefern *wahr* zurück, wenn mind. eine bzw. alle Elemente in der Liste der Bedingung entsprechen, ansonsten *falsch*.

Variablen

Einige Anfragen benutzen den gleichen Ausdruck mehrmals. In diesem Fall ist es sinnvoll, den Ausdruck an eine Variable zu binden, damit der Ausdruck nicht wiederholt eingegeben und ausgeführt werden muss. Dies geschieht mit Hilfe der `LET` Klausel eines FLWR-Ausdrucks. So bindet zum Beispiel der Ausdruck `LET $a := avg(//student/alter)` das Durchschnittsalter (`avg()` steht für die Durchschnittsfunktion) aller Studenten an die Variable `$a`.

weitere Möglichkeiten

Quilt unterstützt auch das Gruppieren von Daten. Es existieren hierfür, anders als in SQL mit `GROUP BY` und `HAVING`, keine speziellen Operatoren. Listing 2.10 zeigt ein Beispiel zur

Kriterium	Bewertung	Multiplikator	Gesamt
Syntax			
Verständlichkeit	+1	1	+1
Funktionalität			
Projektion	+2	3	+6
Selektion	+2	3	+6
Quantifizierung	+2	1	+2
Aggregation	+2	1	+2
Verbund	+2	2	+4
Gruppierung und Sortierung	+2	2	+4
weitere Anforderungen			
Robustheit	+2	2	+4
Objektorientiert	+2	3	+6
Zusatzpunkte	+2	1	+2
Gesamt			**+37**

Tabelle 2.10: Ergebnis der Evaluation von Quilt

Gruppierung in Quilt. Es werden alle Kurse mit mehr als 20 eingeschriebenen Studenten gesucht.

```
FOR $kurs IN kurs
LET $stud := $kurs/student
WHERE count($stud) >= 20
RETURN
    <gut_besuchte_vorlesung>
        <titel>$kurs/titel</titel>
        <anz_stud> count($stud) </anz_stud>
    </gut_besuchte_vorlesung>
```

Listing 2.10: Quilt: Beispielanfrage für Gruppierung

Desweiteren wird auch das Verknüpfen mehrere Quellen mittels Joins unterstützt. In Quilt können sowohl *inner*- als auch diverse Formen des *outer*-Joins formuliert werden.

2.4.5.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.10. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die Quilt Syntax ist gut strukturiert und dokumentiert. Sie ist überaus umfangreich, daher nur **(+1)**.

- *Projektion:*
  Es wird die selbe Funktionalität wie in XPath genutzt. **(+2)**

- *Selektion:*
  Mit Hilfe der `WHERE` Klausel kann während der Anfrage selektiert werden. Zusätzlich gibt es den Bedingungsoperator `IF`. Weiterhin gibt es mehr Vergleichsmöglichkeiten als gefordert. **(+2)**

- *Quantifizierung:*
  Es existieren die Quantoren `some` und `every`. Zusätzlich existieren die Quantifizierungsmöglichkeiten aus XPath. **(+2)**

- *Aggregation:*
  Aggregatfunktionen gehen über die Anforderungen hinaus. **(+2)**

- *Verbund:*
  Neben den Operatoren `UNION` und `INTERSECT` existiert der Operator `EXCEPT`, der eine Menge A um die Elemente einer Menge B verkleinert. **(+2)**

- *Gruppierung und Sortierung:*
  Gruppierung und Sortierung von Ergebnislisten ist möglich. **(+2)**

- *Robustheit:*
  Im Anhang des Quilt-Papers [RFC00] ist die Quilt-Grammatik sowie die Ressourcen zu den Funktionen, die aus anderen Sprachen implementiert sind, detailliert aufgelistet. Damit ist eine robuste Implementierung möglich. **(+2)**

- *Objektorientierung:*
  Quilt arbeitet auf dem DOM der XML-Dokumente. **(+2)**

- *Zusatzpunkte:*
  In Quilt ist die Definition von eigenen Funktionen möglich. Weiterhin kann über Knotenmengen mittels des Operators `FOR` iteriert werden. **(+2)**

2.4.5.4 Fazit

Quilt erweist sich als eine äußerst taugliche Anfragesprache. Quilt besticht durch seine umfangreiche Funktionalität sowohl in der Breite als auch Tiefe. Quilt ist als Anfragesprache für die ZODB **geeignet**.

2.4.6 XQuery - An XML Query Language

2.4.6.1 Hintergrund

XQuery [BCF+07] wurde 2007 von Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie und Jérôme Siméon vorgestellt und vom W3C als Recommendation befürwortet. XQuery ist eine Anfragesprache für die unterschiedlichste Arten von XML-Daten und ist, im Gegensatz zu zum Beispiel XPath (siehe 2.4.2), eine "echte" Anfragesprache mit Verarbeitungs-, Optimierungs- und Queryfunktionalität.

2.4.6.2 Aufbau

Ausdruckskontext

Der Ausdruckskontext (engl: expression context) für einen gegebenen Ausdruck besteht aus allen Informationen, die das Ergebnis des Ausdrucks beeinflussen. Er ist in zwei Kategorien untergliedert:

- *statischer Kontext*
  Der statische Kontext eines Ausdrucks entspricht all jenen Informationen, die während der Analyse, also vor der Evaluation, zur Verfügung stehen. Diese Informationen können genutzt werden, um den Ausdruck auf statische Fehler (zum Beispiel Syntax-Fehler) zu überprüfen.

- *dynamischer Kontext*
  Der dynamische Kontext eines Ausdrucks entpricht all jenen Informationen, die während der Evaluation des Ausdrucks zur Verfügung stehen. Diese Informationen können genutzt werden, um den Ausdruck auf dynamische Fehler (zum Beispiel einen numerischen Überlauf) zu überprüfen. Er enthält zusätzlich alle Informationen des statischen Kontext.

Prozessmodell

Abbildung 2.6 zeigt den schematischen Aufbau des Prozessmodells. Alle Schritte außerhalb des schwarzen Rahmens sind eigenständige Themen, die nicht zur Sprache XQuery gehören. Dies sind *externe Prozesse* wie zum Beispiel das Erzeugen einer XML Data Model (XDM)-Instanz oder der Import einer XML Schema Definition (XSD).

Der innere Bereich wird **Query Processing** genannt und beschreibt den eigentlichen Ablauf einer XQuery-Anfrage. Er wird in zwei Phasen untergliedert:

- *statische Analysephase*
  Die statische Analysephase stützt sich auf den Ausdruck selbst und den *statischen Kontext*.

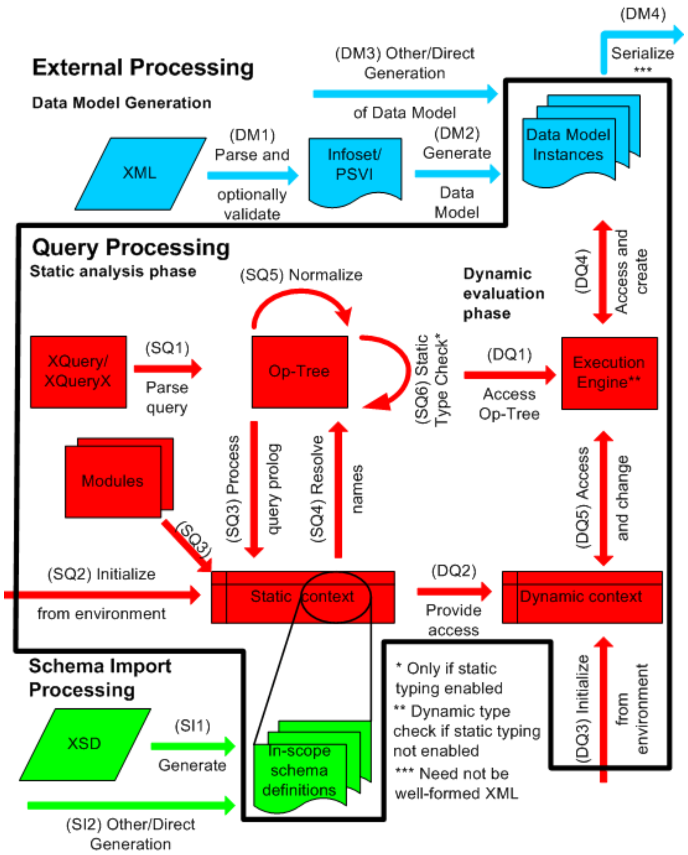

Abbildung 2.6: XQuery Prozessmodell

Sie hängt nicht von Eingabedaten (außer Schemata) ab. Während der statischen Analysephase wird die Anfrage in einen Aufrufbaum zerlegt (siehe `(SQ1)` in Abbildung 2.6). Der *statische Kontext* wird von der Implementation initialisiert (Schritt `(SQ2)`) und mit Schemadefinitionen aus importierten Schemata und Funktions- beziehungsweise Variablendeklarationen `(SQ3)` gefüllt. Der statische Kontext wird dann benutzt, um den Aufrufbaum zu verifizieren `(SQ4)`. Abschließend wird der Aufrufbaum normalisiert `(SQ5)` und jedem Ausdruck ein fester Typ `(SQ6)` zugewiesen.

- *dynamische Evaluationsphase*
  Nach der statischen Analysephase wird in der dynamischen Evaluationsphase für jeden Ausdruck des Aufrufbaumes der Wert bestimmt. Sie hängt von dem Aufrufbaum, der in der statischen Analysephase aufgebaut wurde (siehe `(DQ1)`), den Eingabedaten `(DQ4)` und vom *dynamischen Kontext* `(DQ5)`, welcher wiederum von der Umwelt `(DQ3)` und dem *statischen Kontext* (siehe `(DQ2)`) abhängt, ab. Es können neue Datenmodelle erzeugt `(DQ4)` und der *dynamische Kontext* erweitert `(DQ5)` werden.

Konsistenzbedingungen

Damit XQuery klar definiert ist, müssen die XDM-Instanz sowie der statische und dynamische Kontext zueinander konsistent sein. Hierfür werden Konsistenzbedingungen definiert, die eine korrekt funktionierende XQuery Implementation garantieren. Sie beschreiben unter anderem Voraussetzungen für type-matching oder type-conversion[14].

Fehlerbehandlung

XQuery definiert drei Hauptfehlerarten:

- Der **statische Fehler** ist ein Fehler, der während der statischen Analysephase entdeckt werden muss. Ein Syntax-Fehler ist zum Beispiel ein statischer Fehler.
- Der **dynamische Fehler** ist ein Fehler, der während der dynamischen Evaluationsphase entdeckt werden muss und während der statischen Analysephase entdeckt werden kann. Ein numerischer Überlauf ist zum Beispiel ein dynamischer Fehler.
- Ein **Typenfehler** kann sowohl in der statischen Analyse- als auch der dynamischen Evaluationsphase auftreten.

Das Ergebnis einer Anfrage ist

- in der statischen Analysephase entweder "Erfolg" oder einer oder mehrere Typenfehler, statische Fehler oder statisch-erkannte dynamische Fehler, und
- in der dynamischen Evaluationsphase entweder ein Ergebniswert oder ein oder mehrere Typfehler oder dynamische Fehler.

Zusätzlich zu Fehlern kann eine XQuery-Implementation **Warnungen** ausgeben. Die Spezifikation und der Umgang mit diesen ist implementationsabhängig. Außerdem können in implementationsspezifischen Situationen weitere dynamische Fehler erzeugt werden.

Die XQuery-Spezifikation hält eine Liste mit möglichen Fehlercodes und deren Beschreibung vor[15]. Die Ausgabe zusätzlicher Informationen wie Fehlerort und Ausführungsphase ist in eigenen XQuery-Implementationen möglich.

Typen

XQuery basiert auf den Typen aus XML Schema[16]. Von entscheidender Rolle ist hier der Typ *Liste*, welcher immer benutzt wird, wenn der Typ eines XQuery-Ausdrucks bestimmt werden soll. Listentypen sind nie verschachtelt, das heißt verknüpft man die Werte `1`, `(2, 3)` und `( )` zu

[14] siehe `http://www.w3.org/TR/xquery/#id-consistency-constraints`
[15] Vgl. `http://www.w3.org/2005/xqt-errors/`
[16] Vgl. `http://www.w3.org/TR/xquery/#XMLSchema`

Beispiel	Erklärung
`xs:date`	bezieht sich auf den eingebauten, atomaren Schematyp `xs:date`
`attribute()?`	bezieht sich auf einen optionalen Attributknoten
`element()`	bezieht sich auf jeden Elementknoten
`schema-element(customer)`	bezieht sich auf einen Elementknoten mit dem Bezeichner `customer`
`node()*`	bezieht sich auf keinen oder mehr Knoten jeglichen Typs
`node()+`	bezieht sich auf einen oder mehr Knoten oder atomare Werte

Tabelle 2.11: Beispiele für Listentypen in XQuery

einer Liste, entsteht `(1, 2, 3)` und nicht `(1, (2, 3), )`. Weiterhin wird zwischen dem Typ-Wert und String-Wert unterschieden, wobei Ersterer den Typ und Letzter die Stringrepräsentation eines Elementes widerspiegelt. So ist zum Beispiel der Typ der obigen Liste `(1, 2, 3)` und die Stringrepräsentation `"1 2 3"`.

Listen bestehen aus dem Typ jedes Eintrags und der Kardinalität, das heißt der Anzahl von Einträgen in dieser Liste. Der Typ eines Eintrages wiederum ist unterteilt in den Knotentyp (zum Beispiel `element()`) und den atomaren Typ (zum Beispiel `xs:integer`). Tabelle 2.11 zeigt einige Beispiele von Listentypen.

Weiterhin bietet XQuery Funktionen, um einen Ausdruck mit einem erwarteten Listentyp zu vergleichen. Für eigene Typen können eigene Vergleichsmechanismen implementiert werden[17].

Kommentare werden in XQuery über die Symbole `(:` und `:)` eingeschlossen. Sie haben keinen Einfluß auf die Anfrageprozedur.

Ausdrücke

Jede Anfrage besteht aus (mit runden Klammern geschachtelten) Ausdrücken, die durch Kommata getrennt werden. Ausdrücke bestehen aus Literalen, Funktionsaufrufen, Pfadausdrücken, FLWOR-Ausdrücken[18], Bedingungen, Quantoren, Logischen Ausdrücken sowie eigenen vom Benutzer definierten Funktionen. Weiterhin gibt es Vergleichsoperatoren für Werte, Listen und Knoten.

Module

XQuery bietet die Möglichkeit, Funktions- und Variablendeklarationen in Bibiliotheken zu definieren und in andere Module zu importieren. Diese werden unter anderem im Hauptmodul importiert, welches auch die eigentliche Anfrage enthält.

[17] Vgl. `http://www.w3.org/TR/xquery/#id-sequencetype-matching`

[18] FLWOR steht für die Konstrukte `for`, `let`, `where`, `order`, `by` und `return`

Konformität

Weiterhin sind für einen XQuery-Prozessor Bedingungen definiert, die einen funktionsfähigen Zustand garantieren. Hierbei gibt es die *minimale Konformität*, welche eine Reihe von Bedingungen enthält, die erfüllt sein *müssen*. Weiterhin gibt es Features, die implementiert werden *können* (optionale Features) und *sollten* (Features, die unter Umständen nicht benötigt werden – eine genaue Abwägung über deren Implemntierung ist hier nötig).

Beispiele

An dieser Stelle sollen einige Beispiele vorgestellt werden, die die Syntax und Funktionsweise von XQuery veranschaulichen. Listing 2.11 zeigt eine einfache Anfrage, welche das Alter von ‘Max Mustermann’ ausliest. Als Ergebnis wird in diesem Beispiel <alter>25</alter> zurückgeliefert.

```
for $x in student
return
    if (data($x@vorname)="Max") AND (date($x@nachname)="Mustermann")
        then <alter>{data($x@alter)}</alter>
```

Listing 2.11: XQuery: Auslesen des Alters einer bestimmen Person

Angenommen, es seien alle Kurse gesucht, die von mehr als 10 Studenten besucht werden. Listing 2.12 zeigt eine entsprechende Anfrage in XQuery.

```
for $x in kurse
where count($x/student) > 10
order by $x@titel
return $x
```

Listing 2.12: XQuery: Auslesen aller Kurse, die von mehr als 10 Studenten besucht werden.

Man beachte die unterschiedlichen Möglichkeiten, wie in XQuery Bedingungen formuliert werden können: in Listing 2.11 über ein einfaches `IF..THEN` Statement; in Listing 2.12 über den `WHERE` Block der FLWOR-Ausdrücke.

```
declare function minAlter(
  $alter1 as xs:integer?, $alter2 as xs:integer?) AS xs:integer? {
    return if ($alter1 < $alter2)
        then $alter1
        else $alter2
};
```

Listing 2.13: XQuery: Deklaration eigener Funktionen

Kriterium	Bewertung	Multiplikator	Gesamt
Syntax			
Verständlichkeit	+2	1	+2
Funktionalität			
Projektion	+2	3	+6
Selektion	+2	3	+6
Quantifizierung	+2	1	+2
Aggregation	+2	1	+2
Verbund	+2	2	+4
Gruppierung und Sortierung	+2	2	+4
weitere Anforderungen			
Robustheit	+2	2	+4
Objektorientiert	±0	3	±0
Zusatzpunkte	+1	1	+1
Gesamt			**+31**

Tabelle 2.12: Ergebnis der Evaluation von XQuery

Schließlich noch ein Beispiel, wie in XQuery Funktionen definiert und aufgerufen werden. In Listing 2.13 wird eine Funktion `minAlter(alter1, alter2)` definiert, welche das minimale der beiden übergebenen Alter zurückliefert.

2.4.6.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.12. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die Syntax von XQuery ist sehr umfangreich. XQuery bezieht einen Teil seiner Funktionalität von anderen Sprachen wie XPath oder XML Schema. Die Dokumentation ist äußerst detailliert. **(+2)**

- *Projektion:*
  XQuery nutzt für die Strukturanfrage des Objektgraphen die gleiche Funktionalität wie XPath. **(+2)**

- *Selektion:*
  Während der Anfrage kann über den Bedingungsoperator `where` konditionale Anweisungen ausgedrückt werden. Zusätzlich gibt es den Bedingungsoperator `IF`. Vergleichsoperatoren gehen über die Anforderungen hinaus. **(+2)**

- *Quantifizierung:*
  In XQuery gibt es nur den Existenziellen (`some`) und Universellen (`every`) Quantifizierer. Weiterhin sind die Quantifizierer 'mindestens 1' (+), '0 oder 1' (?) und 'alle' (*) verfügbar. **(+2)**

- *Aggregation:*
  XQuery nutzt die selben Aggregatfunktionen wie XPath. **(+2)**

- *Verbund:*
  Mit Hilfe geschachtelter Iterationen über mehrere Dokumente können Joins nachgebaut werden. Unter [BCF+07], Abschnitt `I.1`, sind die Möglichkeiten für differenzierte Verbundoperationen erklärt. **(+2)**

- *Gruppierung und Sortierung:*
  Es ist sowohl das Gruppieren als auch das Sortieren von Ergebnismengen möglich. Weiterhin kann in XQuery, ähnlich wie in XML-QL, das Ergebnis auch in eine XML-Struktur verpackt werden. **(+2)**

- *Robustheit:*
  Die XQuery-Spezifikation regelt ganz klar, welche Features eine XQuery-Implementation für eine reibungslose Funktion implementieren muss (minimal Conclusion) und was optional implementiert werden kann. Weiterhin sind alle Funktionalitäten sowie deren Verhalten erklärt. **(+2)**

- *Objektorientierung:*
  XQuery beherrscht direkt keine Objektorientierung, allerdings sind die Voraussetzung für die Implementierung in dieser sehr gut. Es müssen lediglich ein paar Anpassungen in den Funktionen gemacht werden, wie Objekteigenschaften ausgelesen werden.

- *Zusatzpunkte:*
  Eigene Funktionen können definiert, in Modulen zusammengefasst und in anderen Implementationen wiederverwendet werden. **(+1)**

2.4.6.4 Fazit

Die Evaluation stellt klar heraus, dass XQuery als Anfragesprache für die ZODB **geeignet** ist. XQuery besticht durch seine äußerst umfangreiche Funktionalität und Robustheit.

Komponente	Beschreibung
`SELECT`	Wähle Objekte aus, die in die Ergebnismenge aufgenommen werden sollen, durch Angabe von Klassennamen.
`FROM`	Gibt an, wo gesucht werden soll.
`WHERE`	Optionale Bedingungen, um die Ergebnismenge einzugrenzen.
`ORDER BY`	Optionale Angabe von Sortierparametern.

Tabelle 2.13: Grundlegender Aufbau einer Anfrage in OQL

2.4.7 Object Query Language (OQL)

2.4.7.1 Hintergrund

Die Object Query Language (OQL) [Gro03] ist eine standardisierte Anfragesprache für Objektdatenbanken. Sie wurde von der Object Database Management Group (ODMG) basierend auf der Syntax von SQL entwickelt. Anfragen werden mit Hilfe von Objekten und ihren Attributen formuliert und liefern Mengen von Objekten zurück.

2.4.7.2 Aufbau

Die grundlegende Syntax entspricht der von SQL: `SELECT ... FROM ... WHERE`. Tabelle 2.13 zeigt die einzelnen Komponenten einer Anfrage im Überblick.

Pfadausdrücke

Pfadausdrücke erlauben den Zugriff auf Attribute von Objekten. Hierfür ist die (`.`) als auch die (->) Notation möglich. Listing 2.14 zeigt die Anwendung eines Pfadausdruckes an einem Beispiel. Über den (`.`) Operator wird auf das Attribut ‘nachname’ der Student-Objekte zugegriffen.

```
1 SELECT Student
2     FROM StudentExtend AS st
3     WHERE st.nachname = "Mustermann"
```

Listing 2.14: OQL: Pfadausdrücke und Extend am Beispiel

Das Extend

Ein Extend enthält alle Objekte in einer Datenbank, die zu einer Klasse gehören. OQL Anfragen arbeiten auf Extends, während SQL auf Relationen arbeitet. Das Extend einer Klasse `Student` wird angesprochen, indem der String `Extend` an den Klassennamen angehängt wird (`StudentExtend`, siehe Listing 2.14 Zeile 2 – das Statement `AS` definiert die `SET`-Variable “`st`”, welche ein Alias für das Extend darstellt.).

Die Ergebnismenge

Der Inhalt der Ergebnismenge wird in der `SELECT` Klausel definiert. Die Elemente der Menge sind entweder vom Typ des gelesenen Extend oder von dem einer seiner Attribute. Es werden zwei Auswahlmöglichkeiten unterstützt:

- *Alle Elemente des Extend*
  Auswahl aller Elemente des Extend über $\star$, Klassenname oder `SET`-Variable.

- *Ein Attribut der Objekte des Extend*
  Auswahl eines Attributes der Elemente des Extends über Pfadausdruck

Der Unterschied soll an zwei Beispielen verdeutlicht werden. Während Listing 2.14 eine Menge Studenten-Objekte zurückliefert, gibt Listing 2.15 eine Menge von String-Objekten zurück, nämlich die Namen aller Studenten.

```
SELECT st.nachname
    FROM StudentExtend AS st
```

Listing 2.15: OQL: Beeinflussen der Ergebnismenge über Pfadausdrücke

Bedingungen

Mit Hilfe der `WHERE` Klausel können Bedingungen definiert werden. Um einen Teilstring in einem String zu suchen, benötigt man den `LIKE` Operator in Kombination mit der Wildcard $\star$. Listing 2.16 Zeile 1 zeigt ein Beispiel für diese Anfrage. Die Anfrage in Listing 2.16 Zeile 2 hingegen sucht, aufgrund des verwendeten Gleichheitsoperators =, nach allen Personen, die den Nachnamen “*mann” tragen.

```
1 ...WHERE Student.nachname LIKE ‘‘*mann’’...
2 ...WHERE Student.nachname = ‘‘*mann’’...
3 ...WHERE Student.nachname LIKE ‘‘*mann’’ AND Student.alter = 25...
4 ...WHERE Student.alter >= 18 AND Student.nachname != "Mustermann"...
```

Listing 2.16: OQL: Bedingungen mit Hilfe der WHERE-Klausel

Mit Hilfe der Bool’schen Operatoren `AND` und `OR` ist es möglich, mehrere Bedingungen zu verknüpfen (Listing 2.16 Zeile 3).

Tabelle 2.14 zeigt eine Liste mit den gebräuchlichsten Vergleichsoperatoren. Ein Beispiel zur Anwendung zeigt Listing 2.16 Zeile 4.

Sortierung und Gruppierung

Mit Hilfe der `ORDER BY` Klausel kann eine Ergebnismenge aufsteigend (`ASC`) oder absteigend (`DESC`) sortiert werden. Listing 2.17 zeigt eine Anfrage, die die Kurse nach ihrem Titel aufstei-

Komponente	Beschreibung
= bzw. ==	Vergleich auf Gleichheit.
!= bzw. <>	Vergleich auf Ungleichheit.
> bzw. >=	Größer (gleich) als.
< bzw. <=	Kleiner (gleich) als.
`IN extend`	Existenz eines Objektes in einem Extend.

Tabelle 2.14: Gebräuchliche Vergleichsoperatoren in OQL

gend sortiert (Zeile 3).

```
1 SELECT *
2     FROM KursExtend as k
3     ORDER BY k.titel ASC
```

Listing 2.17: OQL: Sortierung am Beispiel

Gruppierung ist über die `GROUP BY` Klausel möglich. In OQL wird Gruppierung wie folgt durchgeführt:

1. *Initialisierung*
   Initialisieren einer collection mit dem Ergebnis aus der `FROM` und `WHERE` Klausel.

2. *Anpassung*
   Anpassen der collection mit den Funktionswerten aus der `GROUP BY` Klausel.

3. *Ausgabe*
   Ausgabe der collection entsprechend der Bedingungen in der `SELECT` Klausel.

Das Beispiel in Listing 2.18 verdeutlicht die Gruppierung an einem Beispiel: Anfrage des Durchschnittsalters der Studenten in jedem Kurs.

```
SELECT kursTitel, avgAlter as AVG(
    SELECT s.alter
        FROM k.studenten as s)
    FROM KursExtend as k
    GROUP BY kursTitel as kurs.titel
```

Listing 2.18: OQL: Gruppierung am Beispiel

Weitere Anweisungen

Weiterhin sind noch folgende wichtige Funktionen in OQL definiert:

- `int COUNT`
  Liefert die Anzahl Objekte zurück.

- `bool EXISTS`
  Prüft auf Existenz von Objekten.

- `bool FOR ALL`
  Formulieren von Bedingungen für mehrere Objekte.

Wichtig sind zusätzlich folgende Anweisungen:

- `DEFINE`
  Erlaubt die Speicherung von Anfragen zur Wiederverwendung.

- `ELEMENT`
  Holt das Objekt aus einer einelementigen Menge.

- `DISTINCT`
  Holt eine Ergebnismenge ohne Redundanzen.

2.4.7.3 Bewertung

Eine Übersicht des Evaluationsergebnisses gibt Tabelle 2.15. Die Zusammensetzung der Punkte wird im Folgenden detailliert aufgeschlüsselt:

- *Verständlichkeit der Syntax:*
  Die Syntax von OQL ist weit verbreitet und standardisiert. **(+2)**

- *Projektion:*
  OQL bietet mit den `.` beziehungsweise `->` Operatoren eine mit den Pfadausdrücken aus XPath vergleichbare Funktionalität für die Projektion. **(+2)**

- *Selektion:*
  Selektionen werden in der `WHERE` Klausel angegeben. Es existieren Operatoren für Vergleiche (`<=, >=, !=, ==`) sowie Zeichenketten (`LIKE`). **(+2)**

- *Quantifizierung:*
  Es existiert der Existenzquantor `EXISTS` sowie `ALL`, `SOME`, `ANY` und `IN`. Diese werden in Kombination mit Subqueries oder direkten Mengenangaben verwendet und ermöglichen auf vielfältige Weise Quantifizierungsanfragen, die über die Anforderungen hinaus gehen. **(+2)**

Kriterium	**Bewertung**	**Multiplikator**	**Gesamt**
Syntax			
Verständlichkeit	+2	1	+2
Funktionalität			
Projektion	+2	3	+6
Selektion	+2	3	+6
Quantifizierung	+2	1	+2
Aggregation	+2	1	+2
Verbund	±0	2	±0
Gruppierung und Sortierung	+2	2	+4
weitere Anforderungen			
Robustheit	+2	2	+4
Objektorientiert	+2	3	+6
Zusatzpunkte	±0	1	±0
Gesamt			**+ 32**

Tabelle 2.15: Ergebnis der Evaluation von OQL

- *Aggregation:*
  Es werden alle gängigen Aggregationfunktionen wie `min`, `max`, `sum`, `count` und `avg` unterstützt. **(+2)**

- *Verbund:*
  Über Joins und Subqueries lassen sich Intersection und Union ausdrücken.

- *Gruppierung und Sortierung:*
  Mit Hilfe der Klausel `ORDER BY` kann die Ergebnismenge sortiert werden. Die `GROUP BY` Klausel erlaubt die Gruppierung nach Attributen. **(+2)**

- *Robustheit:*
  Eine Liste mit Fehlercodes ist in der OQL Referenz [Gro03] verfügbar. Diese betreffen unter anderem Syntaxfehler, Typfehler und Zugriffsfehler. **(+2)**

- *Objektorientierung:*
  OQL arbeitet rein objektorientiert. **(+2)**

2.4.7.4 Fazit

OQL ist als Anfragesprache für die ZODB aufgrund der sehr großen Funktionalität absolut **geeignet**.

2.4.8 Native Queries

2.4.8.1 Hintergrund

Native Queries [CR06] wurden von William R. Cook und Carl Rosenberg 2006 vorgestellt und basieren auf dem *Processing Framework for Object Comprehensions* [CT97] von Daniel K.C. Chan und Philip W. Trinder. Native Queries erfüllen nicht alle Kriterien für Anfragesprachen[19], werden aber trotzdem vorgestellt, da sie einen interessanten, anderen Ansatz für Anfragen an Objektdatenbanken demonstrieren.

2.4.8.2 Aufbau

Während das Abspeichern von Objekten in Objektdatenbanken für den Entwickler transparent im Hintergrund abläuft, sehen Anfragen in einer objektorientieren Anwendung eher "fremd" aus. Sie werden meist entweder mit Hilfe von einfachen Zeichenketten oder Objektgraphen mit eingebetteten Zeichenketten formuliert. So teilen alle bisher vorgestellten Anfragesprachen folgende Probleme:

- *Syntaxvalidierung*
  Moderne Entwicklungsumgebungen prüfen eingebettete Zeichenketten nicht auf semantische oder syntaktische Fehler.

- *Refactoring*
  Beim Refactoring des Quelltextes werden die Objekte und Attribute in den Query-Zeichenketten nicht berücksichtigt. Sie müssen manuell überarbeitet werden – ein fehleranfälliger und aufwändiger Prozess.

- *Multiple Syntax*
  Entwickler müssen beim Programmieren umschalten zwischen 'ihrer' Programmiersprache und der Anfragesprache. Anfragen können keinen Code benutzen, der schon in der Programmiersprache existiert.

- *Werteübergabe*
  Die Übergabe eines Parameter an die Query-Zeichenkette gestaltet sich ungeschickt und fehleranfällig.

- *Sicherheit*
  Eingebettete Zeichenketten können für Injection-Angriffe missbraucht werden.

[19] Vgl. Abschnitt 2.4.8.3

Um diese Probleme zu umgehen, ist die Idee der Native Queries, Anfragen direkt in der Programmiersprache zu formulieren und dabei auf die Objekte in der DB direkt zuzugreifen. Hierfür wird ein ausgeklügeltes Indizierungs- und Optimierungsverfahren benötigt, damit eine Anfrage nicht auf alle Objekte der DB zugreifen muss.

Native Queries verfolgt die folgenden Ziele:

- *100% nativ*
  Anfragen sollten vollständig in der Programmiersprache formuliert werden.
- *100% objektorientiert*
  Anfragen sollten in der Sprache selbst ausgeführt werden können, um unoptimierte Anfragen auf Objektgraphen oder -mengen zuzulassen.
- *100% typsicher*
  Anfragen sollten von der Entwicklungsumgebung vollständig zugreifbar für Syntax- und Typprüfung sowie das Refactoring sein.
- *optimierbar*
  Es sollte möglich sein, Native Queries in eine ‘Metasprache’ oder ein API zu übersetzen, um die Geschwindigkeit zu optimieren. Dies kann über Quelltextanalyse oder Übersetzung bewerkstelligt werden.

Listing 2.19 zeigt eine Beispielanfrage in der Spache Python. Es werden alle Studenten gesucht, die jünger als 20 Jahre sind und deren Nachname ein “f” enthält. Sie ist in dieser Form sehr langsam, denn es fehlt jegliche Optimierung.

```
1 def match(ObjectCollection):
2     for object in ObjectCollection:
3         if not isinstance(object, Student):
4             return False
5         if object.alter < 20 AND find(object.nachname, ‘‘f’’) >= 0:
6             yield object
```

Listing 2.19: Native Queries in Python

Die Bedingung in Zeile (3,4) stellt sicher, dass nur *Student*-Objekte verarbeitet werden. Hier ist eine erste Optimierung möglich, indem man der Funktion `match` nur Student-Objekte übergibt, welche zuvor über einen entsprechenden Klassenindex herausgefiltert wurden.

Weiterhin beschreibt [CT97] die Möglichkeit, Native Queries in eine andere Anfragesprache umzuwandeln und diese für den Zugriff auf der DB zu verwenden.

Idealerweise ist jede Art von Code in einer Native Queries-Anfrage erlaubt. Um eine stabile und robuste Umgebung zu garantieren werden in [CR06] einige Restriktionen vorgeschlagen:

- *Variablen*
  Variablendeklarationen sollten erlaubt sein.
- *Erstellen neuer Objekte*
  Temporäre Objekte sind für komplexe Anfragen unerlässlich. Ihre Erstellung sollte unterstützt werden.
- *Statische Aufrufe*
  Statische Aufrufe sind Teil des objektotientierten Konzeptes und sollten erlaubt sein.
- *Sicherheitsbeschränkungen*
  Da die Anfragen mit echten Objekten in der DB arbeiten, muss definiert werden, was sie mit diesen tun dürfen. Es ist vernünftig, das Ausführen von Methoden oder Erstellen von Objekten in einigen Bereichen zu erlauben oder zu verweigern.
- *reiner Lesezugriff*
  Veränderungen an gespeicherten Objekten sollten nicht erlaubt sein. Dies garantiert wiederholbare Ergebnisse und hält Transaktionsprobleme aussen vor.
- *Timeouts*
  Langwierige Anfragen müssen zur Schonung der Ressourcen von Datenbankseite abgebrochen werden können.
- *Speicherbeschränkung*
  Speicherbeschränkungen haben den gleichen Hintergrund wie Timeouts. Ein einstellbares oberes Speicherlimit pro Anfrage wird empfohlen.
- *Undefinierte Aktionen*
  Solange etwas nicht direkt verboten ist, sollte alles erlaubt sein.

2.4.8.3 Bewertung

Als Anfragesprache für die ZODB scheiden Native Queries aus. Unter 2.3 wurden formale Kriterien für Anfragesprachen definiert, die von Native Queries teilweise nicht erfüllt werden. So brechen Native Queries ganz klar mit dem Kriterium *Sicherheit*, denn syntaktisch korrekte Anfragen können ohne Weiteres zum Beispiel in eine Endlosschleife geraten. Da quasi beliebige Konstrukte in einer Anfrage formuliert werden können, ist auch eine *Optimierbarkeit* schwer erreichbar und die *Effizienz* aller möglichen Anfragen nicht zu gewährleisten.

Und schließlich genügen Native Queries nicht dem Kriterium der *Eingeschränktheit*, den Native Queries entsprechen einer kompletten Programmiersprache.

Eine Bewertung der Native Queries wird daher nicht durchgeführt.

2.4.8.4 Fazit

Native Queries bestechen durch die spezielle Art und Weise, Anfragen nicht mittels Querystrings sondern nativ in der Programmiersprache zu formulieren. Dadurch ist ein enormer Funktionsgehalt gegeben, den konventionelle Anfragesprachen nur bedingt erreichen. Allerdings scheiden Native Queries aus, da sie den formalen Kriterien für Anfragesprachen nicht genügen.

2.5 Auswahl einer Sprache für die Implementierung

Die Evaluation der Anfragesprachen hat ergeben, dass **Quilt** und OQL den größten Funktionsumfang bereitstellen. Den dritten Platz belegt XQuery gefolgt von XPath.

Für Python existiert bisher keine Implementierung der vorgestellten Sprachen. Es muss daher im Rahmen der Diplomarbeit eine der Sprachen implementiert werden. Hierfür wird ein Parser[20] benötigt, welcher die Anfragen evaluiert, zerlegt und in ein geeignetes Format für die Weiterverarbeitung umwandelt.

Eine vollständige Implementierung einer funktionsreichen Anfragesprache wie Quilt ist für die Diplomarbeit nicht zielführend und scheidet aufgrund der Zeitbeschränkung aus. Weiterhin erfüllen Native Queries, die ‘einfach’ zu implementieren sind, wichtige formalen Anforderungen an eine Anfragesprache nicht und werden daher auch nicht implementiert[21].

Die in Kapitel 2.4.3 vorgestellten Regular Path Expressions sind aus meiner Sicht sehr gut geeignet, im Rahmen der Diplomarbeit als Demonstrationssprache implementiert zu werden, denn das Thema der Diplomarbeit definiert sich nicht nur aus der *Implementierung* einer Sprache, sondern auch aus den Indexstrukturen, die die Sprache benutzen, und der möglichst effizienten Gesamtfunktionalität, die gegeben sein sollte. Daher ist eine minmale Sprache, die die grundlegenden funktionalen Aspekte wie Projektion und Selektion unterstützt, für eine Testimplementierung aus meiner Sicht absolut ausreichend.

Für die Implementierung sollen die RPE noch etwas optimiert werden: so sehen sie für Prädikatausdrücke nur den Vergleich auf Gleichheit vor. Eine Verbesserung ist hier die Implementierung *zusätzlicher Vergleichsoperatoren* auf Größer(–gleich), Kleiner(–gleich) sowie Ungleichheit. Die-

[20] Vgl. [Pul93] und [Wik08g]
[21] Vgl. Abschnitt 2.3 und Abschnitt 2.4.8.3

Kriterium	**XML-QL**	**XPath**	**RPE**	**XML-GL**	**Quilt**	**XQuery**	**OQL**
Syntax							
Verständlichkeit	+1	+2	+2	+1	+1	+2	+2
Funktionalität							
Projektion	±0	+6	±0	±0	+6	+6	+6
Selektion	±0	±0	−3	+6	+6	+6	+6
Quantifizierung	−2	±0	−1	−2	+2	−1	+2
Aggregation	−2	+2	−2	−2	+2	+2	+2
Verbund	−2	±0	−2	−2	+4	+4	±0
Gruppierung und Sortierung	±0	−4	−4	−2	+4	+4	+4
weitere Anforderungen							
Robustheit	−4	+4	−4	−4	+4	+4	+4
Objektorientiert	±0	+6	+6	+6	+6	±0	+6
Zusatzpunkte	+1	±0	±0	±0	+2	+1	±0
Gesamt	**−8**	**+16**	**−8**	**+4**	**+37**	**+28**	**+32**

Tabelle 2.16: Gesamtergebnis der Evaluation

se Maßnahme verbessert auch das Evaluationsergebnis: im Kriterium *Selektion* erreichen RPE nun **(+6)** Punkte.

Zusätzlich ist die Implementierung eines Dereferenzierungsoperators wie in OQL denkbar, mit dem in Pfadausdrücken über bestimmte Attribute verzweigt werden kann. Auch die Implementierung von Aggregatfunktionen ist möglich. Beide Funktionalitäten werden hier nur erwähnt, aber nicht implementiert. Sie erweitern die Sprache nicht um wichtige Funktionalität und können später implementiert werden.

Ein weiterer Schwachpunkt der RPE ist die fehlende Dokumentation der Robustheit. Da RPE allerdings XPath sehr ähnlich ist, kann man sich aus meiner Sicht an der Dokumentation unter [CD99] orientieren und die dort getätigten Anmerkungen übernehmen. Damit ergibt sich in der Evaluation für das Kriterium *Robustheit* eine Punktzahl von **(+4)**.

Es ergibt sich nun eine Gesamtpunktzahl von **(+9)**, ein gutes Ergebnis für eine minimale Demonstrationssprache.

3 Indexstrukturen

Für eine effiziente Anfrage einer Objektdatenbank sind Indexstrukturen unerlässlich. Ein Index ist "eine von der Datenstruktur getrennte Indexstruktur in einer Datenbank, die die Suche und das Sortieren nach bestimmten Feldern beschleunigt." [Wik08b] Ohne diese müssten sämtliche Objekte in der Datenbank einzeln deserialisiert und die Anfragekriterien überprüft werden. Indexstrukturen halten Informationen über Objekte in der DB vor und ermöglichen ein mehrmaliges, schnelles Auslesen ohne direkten Zugriff auf die Objekte. Hierfür müssen die Daten beim Laden der DB einmalig in den Index gespeichert und bei jeder Veränderung der DB aktualisiert werden.

Dieses Kapitel beginnt mit einer allgemeinen Einführung in Indexstrukturen. Anschließend werden in Abschnitt 3.2 verschiedene Indizes, die für die Implementierung der Diplomarbeit in Frage kommen, vorgestellt und deren Implementierung erklärt.

3.1 Indexstrukturen im Überblick

In diesem Abschnitt werden verschiedene Indextypen beschrieben. Abschnitt 3.1.1 beginnt mit einstufigen Indizes. Mehrstufige Indexstrukturen werden in Abschnitt 3.1.2, B- und B^+-Bäume in 3.1.3 und das erweiterbare Hashing in 3.1.4 erklärt.

3.1.1 Einstufige geordnete Indexstrukturen

Der einstufige[1] Index speichert jeden Wert des Indexfeldes zusammen mit einer Liste von Verweisen auf die Speicherorte der Datensätze mit diesem Feldwert. Die Werte im Index sind *geordnet*, sodass für den Zugriff die binäre Suche[2] angewendet werden kann.

Es gibt mehrere Arten einstufiger Indexstukturen. Für geordnete, eindeutige Indexwerte eignet sich der **Primärindex**. Ist der Indexwert nicht eindeutig, das heißt er kann für mehrere Datensätze gleich sein, so eignet ich der **Clusterindex**. Für nicht-ordnende Werte gibt es den **Sekundärindex**. Die drei einstufigen Indexstrukturen werden in den folgenden Unterabschnitten beschrieben.

[1] Vgl. [EN02, Seite 186]

[2] Die binäre Suche hat eine Komplexität von $O(\log_2 n)$, da die Suchmenge bei jedem Suchschritt halbiert wird.

3.1.1.1 Primärindex

Der Primärindex[3] ist ein geordneter Index, dessen Datensätze eine feste Länge und zwei Felder besitzen. Das erste Feld (der Schlüssel) hat den gleichen Datentyp wie der zu indizierende Wert, das zweite Feld ist ein Verweis auf einen Plattenblock (Blockadresse). Für jeden Block in der zu indizierenden Datendatei gibt es einen Indexeintrag. Primärindizes werden in relationalen Datenbanken automatisch für neue Relationen angelegt. Abbildung 3.3 veranschaulicht den Primärindex an einem Beispiel.

Man unterscheidet dichte (dense) und nicht dichte (sparse) Indizes. Ein **dichter Index** hat einen Sucheintrag für *jeden Suchschlüsselwert* (das heißt jeden Datensatz) in der Datendatei. Ein **nicht dichter Index** hat dagegen Indexeinträge für nur *einige Suchwerte*. Der Primärindex ist ein nicht dichter Index, da er einen Eintrag für jeden Plattenblock und nicht für jeden Datensatz beinhaltet.

Die ZODB besitzt intern einen Primärindex, der den effizienten Zugriff auf die `OID`, das heißt die eindeutige ID ("Primärschlüssel") eines persistenten Objektes, ermöglicht. Es existiert für jedes gespeicherte Objekt ein Eintrag in diesem Primärindex.

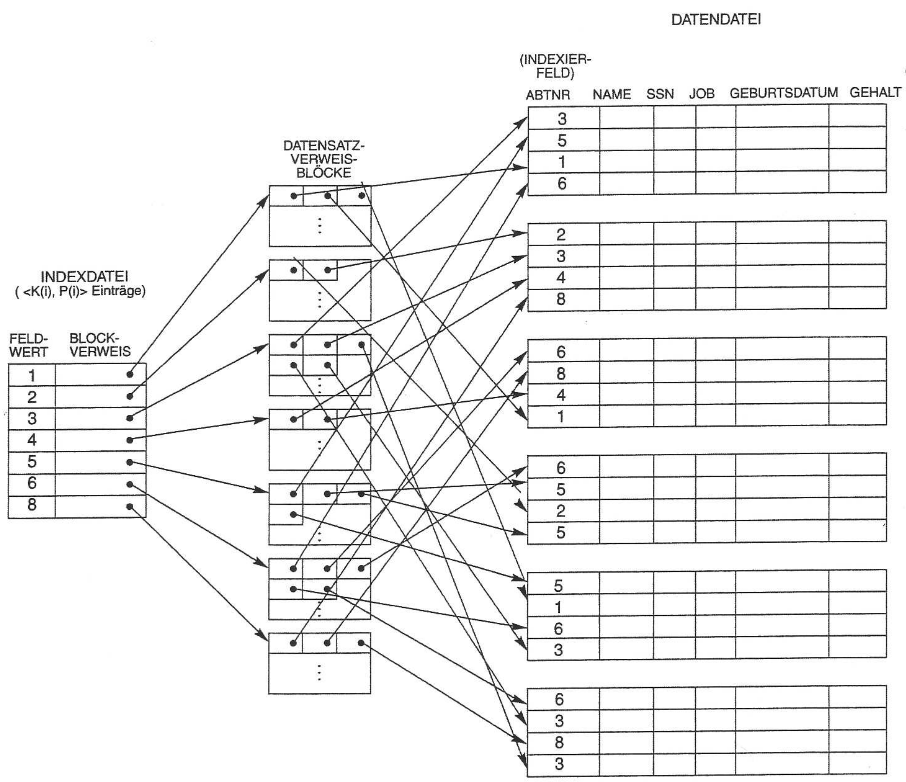

Abbildung 3.1: Ein Sekundärindex mit Indirektionsstufe. Die Indexeinträge erhalten eine feste Länge und eindeutige Feldwerte. (aus [EN02, Seite 194])

[3]Vgl. [EN02, Seite 186ff]

Ein Problem des Primärindex entsteht, wie bei jeder geordneten Datenstruktur, als Folge des Einfügens und Löschens von Datensätzen, da die Ordnung (Sortierung) im Index erhalten bleiben muss. Überlauflisten können hier helfen.

3.1.1.2 Clusterindex

Der Clusterindex[4] erlaubt, im Gegensatz zum Primärindex, einen nicht-eindeutigen Schlüsselwert. Ein Clusterindex ist ebenfalls ein geordneter Index mit zwei Feldern. Er enthält *einen* Eintrag für jeden Wert, der auf die Speicherstelle mit dem ersten Vorkommen des Wertes zeigt. Da die Werte geordnet sind, sind von diesem ersten Vorkommen alle nachfolgenden mit dem gleichen Wert sequentiell erreichbar.

Auch der Clusterindex ist ein nicht dichter Index, da er einen Eintrag für jeden zu indizierenden Wert anstatt für jeden Datensatz enthält.

3.1.1.3 Sekundärindex

Ein Sekundärindex[5] ist ebenfalls geordnet und besteht aus zwei Feldern. Im Gegensatz zum Cluster- und Primärindex ist er allerdings *dicht*, da er für das zu indizierende, *nicht ordnende* Feld für jeden Wert einen Indexeintrag erzeugt, welcher auf den Block oder den Datensatz selbst verweist.

Ein Sekundärindex benötigt in der Regel mehr Speicherplatz und Suchzeiten als ein Primärindex. Gegenüber der *linearen* Suche eines Datensatzes ohne Sekundärindex bringt die Suche mit Index jedoch erhebliche Geschwindigkeitsvorteile.

Für die Indizierung eines *Nichtschlüsselfeldes*, für das zahlreiche Datensätze in der Relation den gleichen Wert im Indexfeld haben können, stehen mehrere Möglichkeiten zur Verfügung[6]:

- *Möglichkeit 1*
  Mehrere Indexeinträge werden in den gleichen Indexwert einbezogen, das heißt einer für jeden Datensatz. (Abbildung 3.2)

- *Möglichkeit 2*
  Für jeden Indexwert gibt es einen Indexeintrag, aber es wird eine Liste mit Verweisen auf die Datensätze angehängt. Möglichkeit 1 und 2 sind beides dichte Verfahren. Der binäre Suchalgorithmus muss zusätzlich auf den Index erweitert werden.

- *Möglichkeit 3*
  Das einstufige Verfahren wird durch ein zweistufiges ersetzt. Es kommt eine zusätzliche

[4] Vgl. [EN02, Seite 189f]
[5] Vgl. [EN02, Seiten 190ff]
[6] Vgl. [EN02, Seite 193]

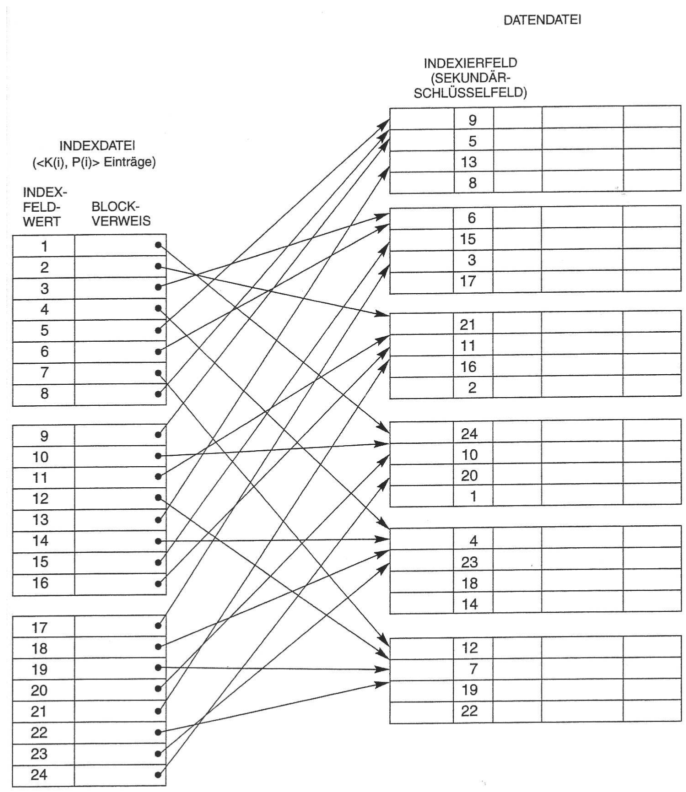

Abbildung 3.2: Ein dichter Sekundärindex auf ein nicht ordnendes Schlüsselfeld. (aus [EN02, Seite 192])

Indirektionsstufe, der *Datensatzverweisblock*, zum Einsatz, welcher alle Verweise auf die Datensätze zu einem Indexwert enthält. Der Index selbst enthält jeden Indexwert einfach mit dem dazugehörigen Verweis auf den Datensatzverweisblock. (siehe Abbildung 3.1)

3.1.2 Mehrstufige Indexstrukturen

Die bisher vorgestellten Indexverfahren basieren auf dem Prinzip, dass die Indexdatei geordnet ist und mit einem mittleren Aufwand von $O(\log_2 b_i)$ bei b_i Blöcken durchsucht wird (binäre Suche). Der mehrstufige Index[7] reduziert den noch gesuchten Indexteil um bfr_i, den *Index-Blockungsfaktor*, der größer als 2 ist. Er wird auch als **Fan-out** des mehrstufigen Index bezeichnet und durch das Symbol **fo** gekennzeichnet. Es ergibt sich also ein mittlerer Suchaufwand bei mehrstufigen Indexstrukturen von $O(\log_{fo} b_i)$ Zugriffen, also viel weniger als bei der binären Suche, falls der Fan-out größer 2 ist.

Beim mehrstufigen Index wird auf der ersten Stufe ein Primärindex für die Indexdatei, die nun als **Basisdatei** bezeichnet wird, erstellt. Dieser wird als die **zweite Stufe** des mehrstufigen Index

[7] Vgl. [EN02, Seiten 195ff]

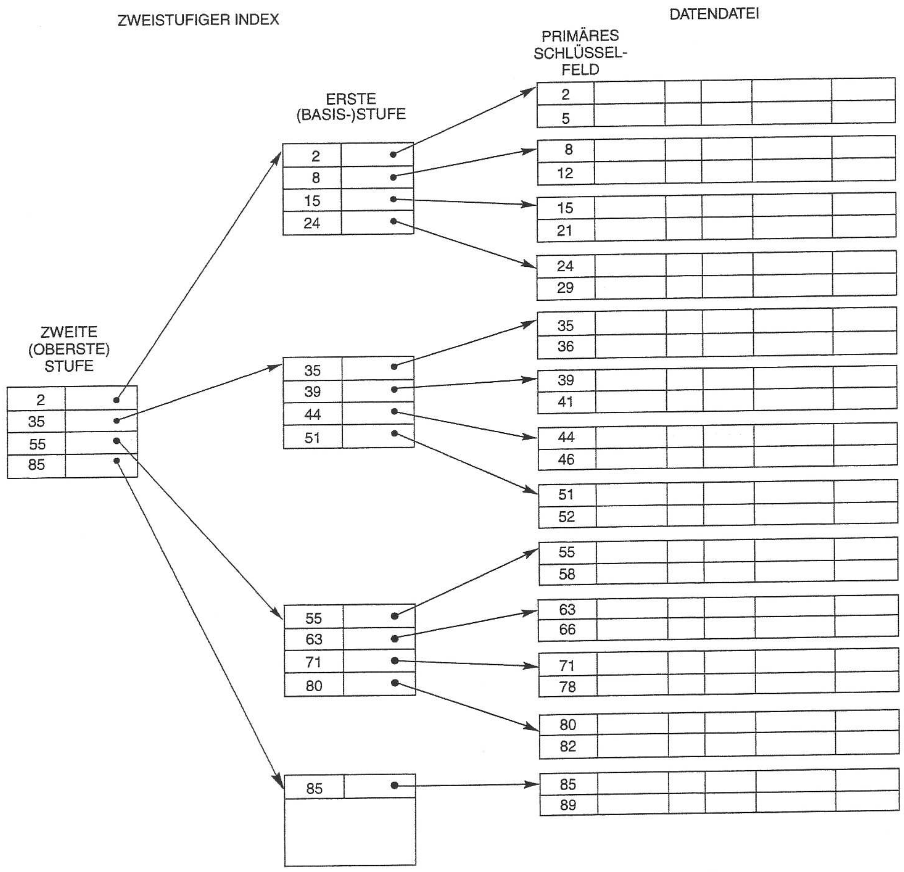

Abbildung 3.3: Ein zweistufiger Primärindex (aus [EN02, Seite 197])

bezeichnet. Die **dritte Stufe**, die einen Primärindex für die zweite Stufe darstellt, hat einen Eintrag für jeden Block der zweiten Stufe. Jede Stufe reduziert die Anzahl der Einträge der vorhergehenden Stufe um den Faktor *fo*.

Abbildung 3.3 verdeutlicht das Prinzip beispielhaft an einem zweistufigen Primärindex. Das mehrstufige Schema kann aber auch für jeden weiteren Indextyp (also Cluser- oder Sekundärindex) verwendet werden, solange der Index der ersten Stufe unterschiedliche Indexwerte und Einträge fester Länge beinhaltet.

Das Problem beim Einfügen und Löschen im Index (alle Indexstrukturen müssen physikalisch geordnet sein) bleibt auch hier weiterhin bestehen. Ihm wird im **dynamischen mehrstufigen Index** begegnet, welcher in jedem Block Platz für neue Einträge lässt und oft mittels *B- bzw. B^+-Bäumen* implementiert wird.

3.1.3 B- und B$^+$-Bäume

Baumstrukturen zählen zu den bekanntesten Indexstrukturen in der Informatik. Zuerst werden die in diesem Zusammenhang verwendeten Begriffe eingeführt: Ein Baum besteht aus **Knoten**. Jeder Knoten hat einen **Elternknoten** oder auch **Vorgänger** (außer die **Wurzel**) und mehrere oder keine **Kindknoten** beziehungsweise **Nachfolger**. Ein Knoten ohne Nachfolger ist ein **Blattknoten**, alle anderen heißen **innere Knoten**.

Ein Knoten enthält neben den Verweisen auf seine Kinder **Daten**. Bei einem mehrstufigen Index als Baumstruktur sind dies die Werte der Indexfelder, nach denen gesucht wird.

Suchbäume

Ein Suchbaum[8] ist vergleichbar mit einem mehrstufigen Index. Er besitzt eine **Ordnung p**, die die maximale Anzahl Verweise pro Knoten eines Suchbaums angibt. Ein Knoten im Suchbaum hat die Form $< P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q >$, wobei $q \leq p$ gilt und P_i den Verweis i und K_i den Suchwert i darstellt. Suchwerte haben eine definierte Ordnung und sind eindeutig. Für den Suchbaum gelten immer zwei Bedingungen:

- *lokales Ordnungskriterium*
  Innerhalb jedes Knotens gilt $K_1 < K_2 < ... < K_{q-1}$.

- *globales Ordnungskriterium*
  Für alle Werte X im Unterbaum, auf den P_i zeigt, gilt: (siehe Abbildung 3.4)
  - $K_{i-1} < X < K_i$ für $1 < i < q$
  - $X < K_i$ für $i = 1$
  - $K_{i-1} < X$ für$i = q$

Jeder Indexwert ist mit einem Verweis auf den Datensatz assoziiert, der den Wert enthält. Weiterhin muss beim Einfügen und Löschen im Baum auf die Wahrung der oben genannten Bedingungen geachtet werden. Diese garantieren allerdings nicht, dass der Suchbaum **ausgeglichen** ist, das heißt dass alle seine Blattknoten auf einer Ebene liegen. Je tiefer ein Baum ist, um so höher ist der Suchaufwand, wenn der gesuchte Wert auf einer tiefen Ebene liegt. Ein ausgeglichener Baum stellt eine minimale Tiefe und damit einen minimalen Suchaufwand sicher.

Ein Spezialfall des Suchbaumes ist der **binäre Suchbaum**, welcher eine Ordnung von **p = 2** hat und zu den bekanntesten Vertretern der Suchbäume zählt.

[8] Vgl. [EN02, Seite 199f]

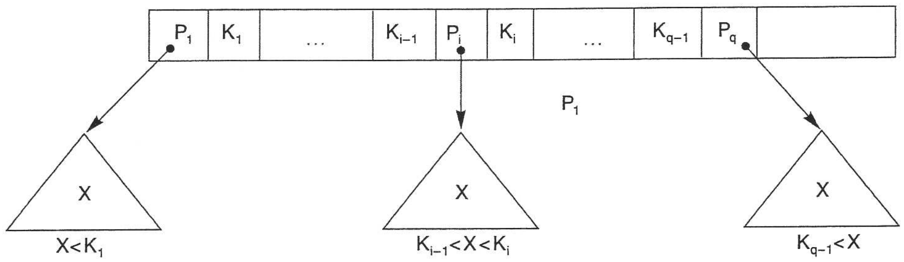

Abbildung 3.4: Ein Knoten eines Suchbaumes. (aus [EN02, Seite 200])

B-Bäume

B-Bäume[9] sind erweiterte Suchbäume, die durch zusätzliche Bedingungen sicherstellen, dass sie immer ausgeglichen sind und der Platz beim Löschen von Knoten nicht verschwendet wird.

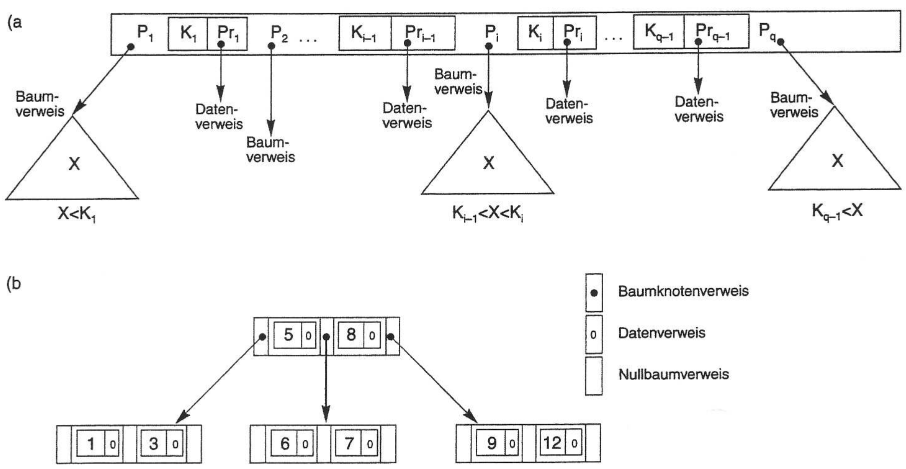

Abbildung 3.5: B-Baumstrukturen: (a) ein B-Baumknoten mit $q-1$ Suchwerten; (b) ein B-Baum der Ordnung $p = 3$. Die Werte wurden in der Reihenfolge 8, 5, 1, 7, 3, 12, 9, 6 eingefügt. (aus [EN02, Seite 202])

Ein B-Baum der **Ordnung p** ist wie folgt definiert:

- Jeder innere Knoten eines B-Baumes besteht aus q Baumverweisen (Verweis auf einen Knoten tiefer im Baum), $q \leq p$ und $q-1$ Tupeln bestehend aus dem Suchschlüsselfeldwert und einem Datenverweis (siehe Abbildung 3.5(a))

- Es gilt das lokale und globale Ordnungskriterium im Suchbaum.

[9] Vgl. [EN02, Seiten 201ff]

- Jeder Knoten hat maximal p Baumverweise.
- In jedem Knoten (außer Wurzel- und Blattknoten) werden mindestens $(p/2)$ Baumverweise gespeichert.
- Alle Blattknoten liegen auf einer Ebene im Baum. Die Baumverweise der Blattknoten haben den Wert Null.

Ein Beispiel für einen B-Baum der Ordnung $p = 3$ zeigt Abbildung 3.5(b). In diesem Beispiel sind alle Werte eindeutig. Für uneindeutige Schlüsselwerte wird eine zusätzliche Indirektionsstufe benötigt, die im Abschnitt 3.1.1 als Möglichkeit 3 der Sekundärindizes erwähnt wurde.

B^+-Bäume

Während im B-Baum (siehe oben) Datenverweise in jedem Knoten vorkommen können, werden in einem B^+-Baum[10] Datenverweise *nur in den Blattknoten* des Baumes gespeichert. Deshalb ist die Struktur von Blatt- und inneren Knoten unterschiedlich: Blattknoten haben einen Eintrag für *jeden* Suchfeldwert ergänzt um einen Verweis auf den entsprechenden Datensatz, während innere Knoten nur aus Suchfeldwerten und Baumverweisen bestehen.

Die Struktur eines *inneren Knotens* eines B^+-Baumes der Ordnung p entspricht im Wesentlichen der Knotenstruktur im Suchbaum, ausser dass für die Suchfeldwerte gilt, dass $K_{i-1} < X \leq K_i$ für $1 < i < q$, $X \leq K_i$ für $i = 1$ und $K_{i-1} < X$ für $i = q$ ist.

Der *Blattknoten* eines B^+-Baumes der Ordnung p hat die Form $<< K_1, Pr_1 >, < K_2, Pr_2 >, ..., < K_{q-1}, Pr_{q-1} >, P_{next} >$ mit $q \leq p$, Pr als Datenverweis und P_{next} als Verweis auf den nächsten Blattknoten im Baum. Der Verweis P_{next} erlaubt das sequentielle Durchlaufen aller Blattknoten entsprechend des Indexfeldes.

Eine Erweiterung des B^+-Baumes ist der **B^*-Baum**, bei dem die Knoten mindestens zu zwei Dritteln statt mindestens zur Hälfte gefüllt sein müssen.

3.1.4 Hashing

Eine Abbildung $h : K \rightarrow S$ heißt "Hash- oder Streuwertfunktion, wenn $|K| \geq |S|$ gilt." [Wik08e, Abschnitt 3] K entspricht den Daten, die gehasht werden. S ist die Menge der Schlüssel, der Adressraum. Eine Hashfunktion bildet also eine Eingabe beliebiger Länge auf eine kurze, möglichst eindeutige Identifikation fester Länge (den *Hash-Wert*) mit dem Ziel, Daten effizient auf eine höchstwahrscheinliche Gleichheit beziehungsweise sicherer Ungleichheit zu vergleichen, ab. Hash-Funktionen müssen vier Kriterien genügen[11]:

[10] Vgl. [EN02, Seiten 204ff]
[11] Vgl. [Wik08e, Abschnitt 4]

- *Reduzierent*
  Der Hashwert soll möglichst viel kleiner sein als das Quellelement.

- *Chaotisch*
  Ähnliche Quellelemente sollen zu völlig verschiedenen Hash-Werten führen. Dadurch wird eine große Streuung der Hash-Werte erreicht.

- *Surjektiv*
  Der Adressraum muss von der Hash-Funktion vollständig ausgefüllt werden können. Es darf keine Hash-Werte im Adressraum geben, die die Hash-Funktion nicht erzeugen kann.

- *Effizient*
  Die Hash-Funktion muss schnell und ressourcenarm arbeiten und sollte das Quellelement nur einmal einlesen müssen.

Interessant für die Indizierung sind die **HashtabellenVgl. [?, Abschnitt 5.9.2f]** oder kurz Hashing. Das Hashing basiert auf dem Prinzip, dass eine Hashfunktion die Position eines Objektes in einer Tabelle errechnet, wodurch sich die Suche nach diesem Objekt 'erübrigt'. Das Objekt wird an die errechnete Stelle (in einen Behälter, **Bucket**) gespeichert und teilt sich dort in der Regel den Platz mit weiteren Objekten, für die die Hashfunktion das gleiche Ergebnis liefert (**Kollision**[12]).

Das Finden von Objekten in der Hashtabelle gestaltet sich analog: Für den Suchschlüssel wird mit der Hashfunktion ein Hashwert ermittelt. Dieser entspricht dem Bucket des gesuchten Objektes. Befinden sich mehrere Objekte in diesem Bucket, muss durch direkten Vergleich der Objekte mit dem Suchschlüssel das gesuchte Objekt ermittelt werden.

Die vorgestellte Technik wird auch als **statisches Hashing** bezeichnet. Sie hat den Nachteil, dass der Hash-Adressraum statisch festgelegt ist und dynamisch nicht vergrößert oder verkleinert werden kann. Hier knüpft das **erweiterbare Hashing** an, bei dem ein Array der Größe 2^d zur Verwaltung der Bucket-Adressen genutzt wird, wobei d die sogenannte *globale Tiefe* dieses Arrays ist. Dieses Array, auch Verzeichnis genannt, fungiert als Index auf die Buckets, welche die Objekte (beziehungsweise Referenzen auf die Objekte) halten. Dabei entspricht d den ersten höherwertigen Bits des Hash-Wertes. Läuft ein Bucket über, muss d um eins erhöht werden. Dadurch verdoppelt sich die Anzahl der Einträge im Verzeichnis sowie die Anzahl Buckets, die Objekte aufnehmen können. Wird durch Löschungen die globale Tiefe d größer als die lokale Tiefe d' in allen Buckets, wird d um eins verringert, was zu einer Halbierung der Anzahl Buckets führt. Abbildung 3.6 illustriert die Struktur des erweiterbaren Hash-Verfahrens für ein Verzeichnis mit $d = 3$.

[12]Zur Vermeidung von Kollisionen gibt es das bereits vorgestellte Verfahren der *Kollisionsauflösung durch Verkettung* mittels Buckets sowie das *offene Hashing*, bei dem lineares Sondieren, quadratisches Sondieren oder doppeltes Hashing eine andere freie Stelle ermitteln.

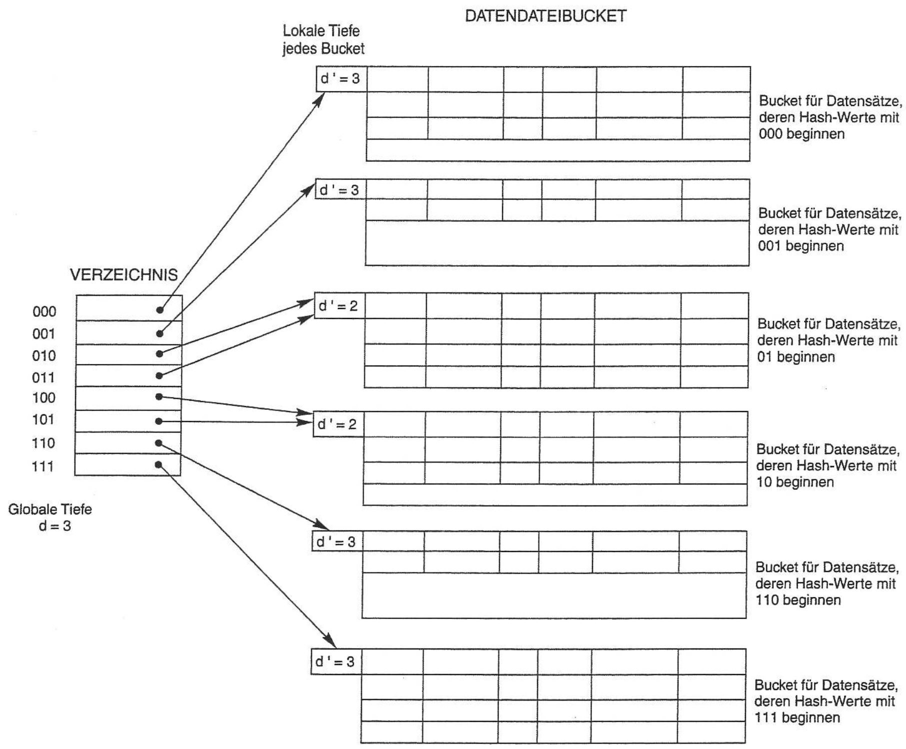

Abbildung 3.6: Struktur des erweiterbaren Hashing. (aus [EN02, Seite 173])

Der wesentliche Vorteil des erweiterbaren Hashing ist die Tatsache, dass sich das Leistungsverhalten mit zunehmender Größe nicht verschlechtert. Weiterhin ist die dynamische Speicherplatzverwaltung vorteilhaft gegenüber dem statischen Hashing, auch wenn die Reorganisation beim Verdoppeln beziehungsweise Halbieren des Verzeichnisses aufwändig ist. Nachteilig ist der zusätzliche Schritt für das Durchsuchen des Verzeichnisses, bevor auf das Bucket und die darin enthaltenen Objekte zugegriffen werden kann.

3.2 Indexstrukturen für den Objektgraphen der ZODB

Dieser Abschnitt gibt einen Überblick über die zu implementierenden Indexstrukturen. Abschnitt 3.2.1 stellt mögliche Indizes für den Objektgraphen der ZODB vor. Wie diese Indizes implementiert werden können, wird in Abschnitt 3.2.2 erläutert.

3.2.1 Indexorganisation

Indexstrukturen sollen den Zugriff der RPE (siehe Abschnitt 2.5) auf den Objektgraphen der ZODB optimieren. Grundsätzlich gibt es hierfür drei Fälle, die bei nahezu jeder RPE-Anfrage ausgeführt werden[13]:

- *Klassenbezeichner*
  Innerhalb der Pfadausdrücke ist die Suche nach Objekten zu optimieren. Es wird eine Liste mit Objekten, die zu der gesuchten Klasse gehören, zurückgegeben.

- *Attributnamen*
  Innerhalb der Pfadausdrücke ist die Suche nach Attributnamen möglich. Es wird eine Liste mit Objekten, die das gesuchte Attribut enthalten, zurückgegeben.

- *Beziehungen*
  Für zwei gegebene Objekte muss ermittelt werden, ob diese eine Vorfahren- beziehungsweise Nachkommensbeziehung im Objektgraphen haben.

Für die Suche nach Klassenbezeichnern wird ein **Klassenindex** benötigt, der einer Abbildung von Namen auf eine Liste mit Objektreferenzen entspricht. Details zu Implementierung des Klassenindex werden in Abschnitt 3.2.2.1 erläutert.

Für die Suche nach Attributnamen wird ein **Attributindex** benötigt, der einer Abbildung von Namen auf eine Liste mit Objektreferenzen entspricht. Details zu Implementierung des Attributindex werden in Abschnitt 3.2.2.2 erläutert.

Für die Suche nach Beziehungen wird ein **Strukturindex** benötigt, der für zwei gegebene Objekte angibt, ob diese im Objektgraphen eine Vorfahren- oder Nachkommensbeziehung haben. Details zur Implementierung des Strukturindex werden in Abschnitt 3.2.2.3 erläutert.

3.2.2 Datenorganisation

In diesem Abschnitt wird für den Klassen-, Attribut- und Strukturindex eine möglichst effiziente Implementierung ausgearbeitet. Dabei wird auf eine flexible, erweiterbare und performante Umsetzung geachtet.

3.2.2.1 Klassenindex

Der Klassenindex entspricht einer $n \rightarrow m$ Relation, wobei n die Anzahl Klassenbezeichner X und m die Anzahl Objekte der Klasse X darstellen. Eine einfache Implementierung dieser Relation

[13] Vgl. [LM01]

ist in Python über ein *Dictionary*[14] mit angehängter *Liste*[15] möglich. Dafür werden alle in der DB gespeicherten Objekte nach ihren Typen gruppiert und in Listen gespeichert. Anschließend wird jede Liste in das Dictionary unter dem Klassenbezeichner als Schlüssel eingefügt. Eine Anfrage nach dem Schlüssel ‘Student’ liefert dann eine Liste mit allen Objekten vom Typ ‘Student’[16].

```
1 struct_index = {}                    # create index
2 for i in range(1000000):
3     struct_index[str(i)] = []        # insert an empty list
4
5 for i in range(1000000):
6     struct_index[str(i)]             # get list from index
```

Listing 3.1: Laufzeittest des Klassenindex mittels Dictionary

```
sweh@atlantis:~/devel/dicttest$ bin/test
Running unit tests:
  Ran 1 tests with 0 failures and 0 errors in 18.885 seconds.
```

Listing 3.2: Ergebnis des Laufzeittests mittels Dictionary

Das Prinzip des Klassenindex mittels Dictionary verdeutlicht Listing 3.1 an einem Performance-Test. In das Dictionary `struct_index` werden 1000000 leere Listen mit dem Schlüssel *i* eingefügt (Listing 3.1 Zeile 2f). Die Zeilen 5f in Listing 3.1 verdeutlichen den Zugriff auf die Einträge des Indexes. Der Test ergibt eine Laufzeit von ca. **19 Sekunden** (siehe 3.2).

Eine alternative Möglichkeit zu Dictionaries bieten *BTrees*[17]. Diese B^+-Bäume gibt es in vier verschiedenen Varianten:

- *IIBTree*
  Assoziiert Integer-Schlüssel auf Integer-Elemente.
- *IOBTree*
  Assoziiert Integer-Schlüssel auf Objekte als Elemente.
- *OIBTree*
  Assoziiert Objekte als Schlüssel auf Integer-Elemente.

[14] Das Dictionary ist eine als Hashtabelle implementierte Assoziation, welche nichtnumerische Schlüssel zur Adressierung der enthaltenenen Elemente nutzt. Vgl. [LA07, Seiten 111ff]

[15] Listen speichern Elemente sequentiell. Elemente können hinzugefügt oder gelöscht werden, ein Zugriff erfolgt über numerische Schlüssel.

[16] In der Praxis würde man nicht die Objekte selbst in der Liste speichern, sondern ihre ObjectIDs – eine eindeutige ID, die jedes persistente Objekt in der ZODB hat.

[17] BTrees sind in Python implementierte B^+-Bäume. Sie werden wie ein Dictionary benutzt, speichern ihre Daten aber in einer Baumstruktur und nicht in einer Hashtabelle.

- *OOBTree*
  Assoziiert Objekte als Schlüssel auf Objekte als Werte.

Während für die Schlüssel von Dictionaries nur die Bedingungen gelten, dass sie hashbar und auf Gleichheit vergleichbar sind, muss auf die Schlüssel der BTrees eine *strenge Totalordnung*[18] gelten. Das bedeutet:

- *Reflexivität*
  R ist reflexiv: $\Longleftrightarrow \forall x \in M : xRx$

- *Transitivität*
  R ist transitiv: $\Longleftrightarrow \forall x, y, z \in M : xRy \wedge yRz \Longrightarrow xRz$

- *Trichotomie*
  Es gilt für $a, b \in M$ genau eine der Beziehungen $a < b$, $a = b$ oder $a > b$.

In Python erfüllen diese Bedingungen unter anderem die Datenstrukturen Integer, Float sowie 8-Bit- und Unicode-Zeichenketten.

Für den Klassenindex empfiehlt sich der **OOBTree**, da er der Einzige ist, welcher Objekte (Zeichenketten sind in Python Objekte) als Schlüssel auf Objekte (in unserem Fall Listen) assoziiert. Eine Implementierung gestaltet sich analog zum Dictionary-Fall. Einen Performance-Test zeigt Listing 3.3. Dieser hat eine Laufzeit von ca. **28 Sekunden** (Listing 3.4), ist also langsamer als die mittels Hashing implementierten Dictionaries. Das bestätigt die unter 3.1.4 getätigte Aussage, dass sich beim Hashing das Leistungsverhalten mit zunehmender Größe nicht verschlechtert.

```
from BTrees.OOBTree import *
struct_index = OOBTree()             # create index
for i in range(1000000):
    struct_index.insert(str(i), [])  # insert empty list

for i in range(1000000):
    struct_index[str(i)]             # get list from index
```

Listing 3.3: Laufzeittest des Klassenindex mittels OOBTree

```
sweh@atlantis:~/devel/btreetest$ bin/test
Running unit tests:
  Ran 1 tests with 0 failures and 0 errors in 25.726 seconds.
```

Listing 3.4: Ergebnis des Laufzeittests mittels OOBTree

[18] Vgl. [Wik08f]

Damit der Index nicht bei jedem Einlesen der ZODB neu aufgebaut werden muss, ist es angedacht, ihn direkt in der Datenbank zu speichern. Dadurch ergeben sich Probleme bei den Dictionaries, die bei BTrees nicht auftauchen:

- Dictionaries werden in einem Stück alloziiert. Änderungen ziehen dadurch eine komplette Speicherung des Indizes nach sich. BTrees hingegen sind mehrstufige Baumstrukturen, die aus mehreren Objekten aufgebaut wird. Eine partielle Speicherung der geänderten Daten ist dadurch möglich.

- Da Dictionaries in einem Stück alloziiert werden, müssen sie komplett in den Arbeitsspeicher geladen werden. Bei besonders großen Indizes kann es hier zur Auslagerung kommen, welche erhebliche Performanceeinbußen mit sich bringen. BTrees hingegen laden die Teilbäume partiell nach, sobald sie gebraucht werden.

Diese Probleme ergeben sich auch bei den nachgelagerten Listen, die die Objekte aufnehmen sollen. Hier muss stattdessen auf **TreeSets** zurückgegriffen werden, die wie BTrees aus einer mehrstufigen Baumstruktur bestehen.

3.2.2.2 Attributindex

Der Attributindex gestaltet sich analog zum Klassenindex (siehe 3.2.2.1). Hier müssen die Attributnamen (Zeichenketten) auf die Objekte, die dieses Attribut enthalten, assoziiert werden. Auch diese Relation ist eine $n \rightarrow m$ Relation, wobei n die Anzahl Attributbezeichner X und m die Anzahl Objekte mit dem Attribut X darstellen.

Die Implementierung sollte aus den gleichen Gründen wie beim Klassenindex dargelegt mit einem OOBTree als Suchindex und einem nachgelagerten TreeSet als Objektcontainer vollzogen werden.

3.2.2.3 Strukturindex

Um Reguläre Pfadausdrücke zu beschleunigen, ist eine schnelle Bestimmung von Vorfahren- und Nachkommensbeziehungen innerhalb der Hierarchie des Objektgraphen unumgänglich. [LM01] stellt einen Ansatz vor, der für den Objektgraphen ein Nummerierungsschema beinhaltet, mit dem Beziehungen zwischen Objekten ohne Navigation durch den Graphen bestimmt werden können. Es basiert auf Dietz's Nummerierungsschema[19], welches folgenden Sachverhalt herausstellte:

[19] Vgl. [Die82, Seiten 122–127]

> *Für zwei gegebene Knoten x und y aus dem Baum T ist x dann und nur dann ein Vorfahre von y, wenn x vor y in der Preorder- und nach y in der Postorder-Traversierung steht.*

Der Vorteil liegt auf der Hand: Vorfahren- und Nachkommensbeziehungen können in linearer Zeit durch Vergleich der Pre- und Postorderwerte ermittelt werden. Allerdings müssen bei jedem Einfügen und Löschen von Objekten aus dem Graphen für viele Knoten die Werte neu berechnet werden, worunter die Flexibilität leidet.

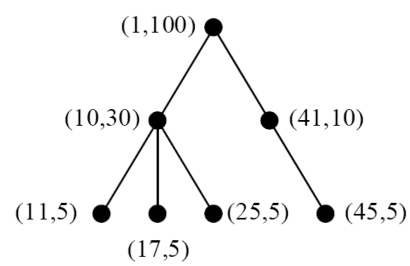

Abbildung 3.7: Erweitertes Nummerierungsschema nach [LM01]

[LM01] setzt hier an und stellt ein erweitertes Nummerierungsschema vor, welches anstatt einfacher Pre- und Postorderwerte je Baumknoten einen *erweiterten Preorderwert* und einen *Wertebereich von Nachfolgern* definiert. Das bedeutet, dass jeder Knoten ein Tupel mit zwei Werten enthält, von denen der erste die *Position* des Knotens in Preorder und der zweite die *mögliche Menge* an Nachfolgern wiederspiegelt. Für die Bestimmung der Vorfahren- und Nachkommensbeziehung ergibt sich nach [LM01] folgendes Lemma:

> *Für zwei gegebene Knoten x und y aus T ist x dann und nur dann Vorfahr von y, wenn* $order(x) < order(y) \leq order(x) + size(x)$ *gilt.*

Zur Verdeutlichung wird das Lemma an einem Beispiel nachvollzogen. Abbildung 3.7 zeigt einen im erweiterten Schema nummerierten Baum, bei dem jeder Knoten mit besagtem `<order, size>` Tupel versehen ist. Der Knoten $(10,30)$ ist ein Vorfahr des Knotens $(25,5)$, da $10 < 25 \leq 40$. Weiterhin ist auch der Knoten $(1,100)$ ein Vorfahr von $(25,5)$, da $1 < 25 \leq 101$. Noch ein Gegenbeispiel zur Verdeutlichung: $(41,10)$ ist kein Vorfahr von $(25,5)$, denn $41 \nless 25$.

Das erweiterte Nummerierungsschema erhöht im Vergleich zu Dietz's Nummerierungsschema die Flexibilität beim Einfügen neuer Knoten durch die Einführung des `size`-Wertes, welcher Platz für weitere Nachkommen bietet. Doch auch diese Variante hat ihre Grenzen, denn beim Erreichen des `size`-Limits muss entweder mit einer größeren Spanne komplett neu durchnummeriert oder ein *Unter-Tupel* mit neuem `order`- und `size`-Wert erzeugt werden, welches für alle weiteren Nachkommen eine neue Nummerierung 'unterhalb' des letzten Tupels erzeugt.

Nimmt man nur die Variante der *Unter-Tupel* und lässt die `order`- und `size`-Werte aussen vor, so erhält man eine weiteres Nummerierungsschema, welches durch seine Einfachheit und Flexibilität bei der Erweiterung in die Breite als auch in die Tiefe bestricht. Es macht sich die Rekusivität, das heißt die Tatsache, dass alle Nachkommen eines beliebigen Knotens einen Teilbaum des gesamten Baumes darstellen, zu Nutze. Dabei werden neue Teilbäume von links nach rechts durchnummeriert und jeder Knoten "erbt" die Nummerierung seiner Vorfahren. Dadurch ergibt sich pro Knoten ein Tupel der Länge x, wobei x der Ebene des Knotens entspricht. Abbildung 3.8(a) zeigt diese Variante an einem Beispiel.

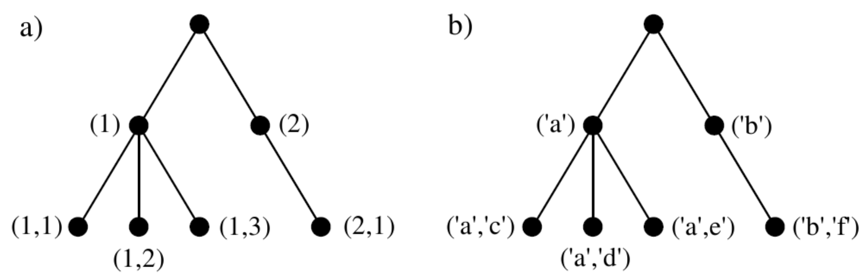

Abbildung 3.8: Weiteres Nummerierungsschema

Wird in dieser Variante anstatt einer Nummerierung der *Suchschlüssel* (für den Strukturindex ist dies die OID[20] der persistenten Objekte) genommen, entsteht ein flexibles und erweiterbares Nummerierungsschema zur Bestimmmung der Vorfahren- und Nachkommensbeziehung von Objekten im Objektgraphen. Abbildung 3.8(b) stellt dieses finale Nummerierungsschema für den Strukturindex beispielhaft dar.

Zur Bestimmung der Vorfahren- und Nachkommensbeziehung sei folgendes Lemma gegeben:

> *Sei v das Tupel des Knotens x, w das Tupel des Kontens y.*
> *Für zwei gegebene Knoten x und y aus T ist x dann und nur dann Vorfahr von y, wenn $len(v) < len(w)$ und $\forall i, i \in \mathbb{N}, 0 \leq i < len(v) : v_i = w_i$ gilt.*[21]

3.2.2.4 Weiterführende Anmerkungen

Object Identifier

Jedes Objekt in der ZODB hat eine feste, eindeutige ID: den Object Identifier (OID). Zur besseren Verwaltung und um Speicherplatz zu sparen empfiehlt es sich, innerhalb der Indexstrukturen mit diesen OIDs anstatt mit den Klassenbezeichnern zu arbeiten.

[20]siehe Abschnitt 3.2.2.4

[21]die Funktion $len(t)$ bestimme die Länge des Tupels t

Weitere Indizes

Die vorgestellten Indizes stellen eine minimale Anforderung für die Funktionalität der zu implementierenden Anfragesprache dar. Das bedeutet, dass mit diesen Indizes alle geforderten Funktionalitäten implementiert und effizient ausgeführt werden können. Trotzdem gibt es eine Reihe weiterer, möglicher Indizes, die die Performance der Implementierung verbessern können. Zu diesen zählen:

- *Werteindex*

  Der Werteindex hält für alle Werte von Attributen der Objekte die dazugehörige OID vor. Dadurch müssen bei Attributvergleichen die Objekte nicht mehr angefasst werden, es kann direkt über den Index verglichen werden, ob ein Wert größer oder kleiner als der Vergleichswert ist. Weiterhin bietet er die Möglichkeit, Aggregatfunktionen wie *Minimum* und *Maximum* performant und einfach zu implementieren: Da die Bäume automatisch eine Ordnung auf ihre Schlüsselwerte (und damit auf die Attribute) anlegen, kann der minimale (erster Schlüsselwert) sowie maximale (letzter Schlüsselwert) Attributwert in linearer Zeit ermittelt werden.

- *Referenzindex*

  Für die Implementierung des Dereferenzierungsoperators (siehe Kapitel 2.5) ist ein Index nötig, welcher zu einem Attributbezeichner und der OID des Objektes die OIDs der nachfolgenden Objekte speichert. Hierfür empfiehlt sich ein zweistufiger Index.

4 Implementierung

Dieses Kapitel beschreibt die Implementierung der Anfragesprache, der Indexstrukturen und der Algorithmen zur Behandlung der Anfragen. Abschnitt 4.1 beginnt mit einer Übersicht über das entwickelte Pythonprodukt *ObjectQuery* und beschreibt den Aufbau der einzelnen Komponenten. In Abschnitt 4.1.1 wird die Implementierung der Anfragesprache erklärt, Abschnitt 4.1.2 schließt mit der Implementierung der anfrageunterstützenden Komponenten.

4.1 Die ObjectQuery

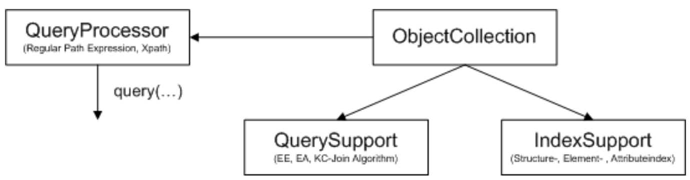

Abbildung 4.1: Aufbau der ObjectQuery

Für die Implementierung des Pythonproduktes ObjectQuery, welches die Funktionalitäten zum Zerlegen (Parsen) der Anfrage und den Zugriff auf die Indexstrukturen vereint, nutze ich einen modularen Aufbau, der die Entwicklung der Komponenten als 'eigenständige Programme' ermöglicht. So steht an erster Stelle der *QueryProcessor*, welcher mit dem *QueryParser* die Anfrage zerlegt und einen Queryplan erstellt. Mit diesem Plan ermittelt er über die *ObjectCollection* das Ergebnis der Anfrage. Die ObjectCollection greift auf die Indexstrukturen (*IndexSupport*) und Algorithmen (*Querysupport*) zurück.

Eine schematische Übersicht des Aufbaus gibt Abbildung 4.1, die einzelnen Komponenten werden im Folgenden detailliert erklärt.

- *QueryProcessor*
  Der QueryProcessor kümmert sich um die Verarbeitung der Anfrage. Er verarbeitet den Queryplan, den der QueryParser liefert, mit Hilfe der ObjectCollection. Weiterhin kümmert er sich um die Aufbereitung der Ergebnismenge.

- *QueryParser*
  Der QueryParser kümmert sich um das Verarbeiten der übergebenen Anfrage. Im Rahmen

der Diplomarbeit wird ein RPE-QueryParser implementiert. Er kann später flexibel durch einen QueryParser einer anderen Sprache ersetzt werden.

- *ObjectCollection*
  Die ObjectCollection muss vor dem Instanziieren des QueryProcessor angelegt und diesem übergeben werden. Sie unterstützt den QueryProcessor beim Suchen durch Algorithmen und Indexstrukturen. Sie bündelt Query- und IndexSupport und ist die Schnittstelle zur ZODB.

- *QuerySupport*
  Der QuerySupport bietet der ObjectCollection Algorithmen, die die Operatoren der RPE umsetzen. Diese sind zum Beispiel Attributvergleiche, Pfadausdrücke und die Vereinigung.

- *IndexSupport*
  Der IndexSupport unterstützt die ObjectCollection mit dem im Abschnitt 3.2.2 vorgestellten Klassen-, Attribut- und Strukturindex. Weitere Indexstrukturen können hier problemlos hinzugefügt werden.

4.1.1 Der QueryProcessor

Der QueryProcessor nimmt die Anfrage entgegen und leitet sie an den QueryParser (Abschnitt 4.1.1.1) weiter. Dieser liefert einen Queryplan zurück. Der QueryProcessor ermittelt anschließend mit Hilfe des Queryplans und der ObjectCollection (Abschnitt 4.1.2) die Ergebnismenge.

Der QueryProcessor arbeitet dabei rekursiv auf dem Queryplan und ruft die Joins mit Hilfe eines dynamischen Mappings. Dadurch ist eine Erweiterung flexibel durchführbar, es ist angedacht, dass der Processor beim Hinzufügen weiterer Algorithmen in den Queryplan nicht refactored werden muss.

4.1.1.1 Der QueryParser

Es gibt zwei grundsätzliche Möglichkeiten, regulärer Pfadausdrücke zu verarbeiten: den *Lexer* (Kurzform für 'lexikalischer Scanner') und den *Parser*. Ein Lexer zerlegt eine Eingabe in Folgen zusammengehöriger Einheiten (sogenannte Token). Der Parser ist eine Erweiterung des Lexers. Er liefert keine Token zurück, sondern eine individuelle Datenstruktur. Lexer als auch Parser benötigen eine reguläre Grammatik, nach welcher die Eingabe validiert und zerlegt wird. In Python gibt es für beide Möglichkeiten fertige Module:

- *Reflex*
  Reflex [Tal06] ist ein Python-Lexer, welcher mit Hilfe einer Backus-Naur-Form (BNF) eine Eingabe validiert und in eine Token-Datenstruktur umwandelt. Bei der Benutzung von Reflex stellte sich heraus, dass dieser nicht rekursiv arbeitet: *nach* dem Bearbeiten einer

Regel springt Reflex nicht automatisch an den Einsprungpunkt der übergeordneten Regel zurück, sondern erwartet entweder eine Sprunganweisung oder fährt mit der Validierung der sequenziell nächsten Regel fort.

- *SimpleParse*
  SimpleParse [Fle06] ist ein Parser für Python. Er erwartet während der Initialisierung eine Grammatik in Erweiterte BNF (EBNF) und liefert eine Datenstruktur, die der Grammatik-Abbildung der geparsten Anfrage entspricht. Da diese Abbildung für eine Weiterverarbeitung im QueryProcessor kaum geeignet ist, muss sie anschließend noch in eine universelle Form (den Queryplan) umgewandelt werden.

Da Reflex aufgrund seiner nicht-rekursiven Validierung für reguläre Pfadausdrücke ungeeignet ist, wird nachfolgend SimpleParse erweitert um eine geeignete Ausgabedatenstruktur implementiert. Es werden zwei Dinge benötigt:

- *Erweiterte BNF*
  Die Syntax der RPE muss dem Parser in einer EBNF vorliegen. Da hierfür noch keine Form existiert, wird in Kapitel 4.1.1.2 eine solche hergeleitet.

- *Queryplan*
  Die von SimpleParse erzeugte Ergebnis ist sehr speziell an die EBNF der Anfragesprache angelehnt. Es muss daher in einen allgemeineren Queryplan umgewandelt werden, damit der Parser für die Zukunft austauschbar ist.

4.1.1.2 Herleitung einer EBNF für SimpleParse

Grundlage einer RPE ist ein Pfadelement, welches aus einem Bezeichner oder einer Wildcard (_), einem optionalen Quantor (`?`, `+` oder `*`) und einem optionalen Prädikatausdruck besteht. Ein Prädikatausdruck hat die feste Form `[@ID="VALUE"]`. `ID` besteht aus einer Zeichenfolge ohne Leer- und Sonderzeichen (äquivalent zum Bezeichner des Pfadelements), `VALUE` wiederrum aus einer Zeichenfolge mit möglichen Leer- und Sonderzeichen. Beispiele einfacher Pfadelemente zeigt Listing 4.1.

```
/Kurs?
/Kurs[@titel="Medientechnik"]
/Student[@alter>="12"]
```

Listing 4.1: Beispiele für Pfadelemente in RPE

Diese einfachen Pfadausrücke können über Pfadtrennzeichen separiert (`/Kurse/Student?`) oder Vereinigungsmengen (`/Kurs | /_*/Student`) kombiniert werden. Desweiteren kann mit Hilfe von runden Klammern Vorrang (precedence) ausgedrückt werden.

Die vollständige EBNF der RPE zeigt Abbildung 4.2 schematisch, Listing 4.2 veranschaulicht die Grammatikdefinition für SimpleParse. Diese enthält die in Abschnitt 2.5 vorgestellten Erweiterungen der Funktionalität.

```
rpe            := expr, ((UNION / PATH_SEPARATOR), expr)?
expr           := ((bracket / normal), occurence?)+
bracket        := open_bracket, rpe+, close_bracket
normal         := PATH_SEPARATOR?, pathelem,
                  (PATH_SEPARATOR, pathelem)*
pathelem       := (WILDCARD / ELEM), occurence?, predicate?
predicate      := (PREDICATE_BEGIN, ID, COMPARER, '"', ATTRVALUE,
                  '"', PREDICATE_END)
occurence      := OCC_NONE_OR_ONE / OCC_ONE_OR_MORE / OCC_MULTI
ID             := ELEM
ELEM           := [a-zA-Z0-9_]+
ATTRVALUE      := text / quoted_text
text           := [a-zA-Z0-9 ]*, quoted_text*
quoted_text    := [a-zA-Z0-9 ]*, '\"', [a-zA-Z0-9 ]+, '\"',
                  [a-zA-Z0-9 ]*
COMPARER       := COM_EQ / COM_LO_EQ / COM_GR_EQ / COM_GR /
                  COM_LO / COM_NOT_EQ
```

Listing 4.2: SimpleParse-Grammatik für RPE (gekürzt)

SimpleParse liefert als Ergebnis eine komplex verschachtelte, aus Tupeln und Listen bestehende Datenstruktur. In dieser wird der Weg, den SimpleParse durch die Grammatik genommen hat, abgebildet. Diese Datenstruktur ist für eine weitere Verwendung im QueryProcessor ungeeignet, da sie zum Einen nicht den Query selbst, sondern eine “Wegbeschreibung” durch die EBNF darstellt, und zum Anderen keine *Werte* von Pfad- oder Attributbezeichnern und dergleichen enthält, sondern nur Start- und Endpunkt dieser Werte im ursprünglichen Ausdruck. Außerdem ist eine allgemeinere Form gewünscht, um die Austauschbarkeit der Anfragesprache zu gewährleisten. Insofern muss das Ergebnis des Parsers in eine weitere Datenstruktur umgewandelt werden: den Queryplan.

Die Methode `_build_queryplan(self, result, expression, output)` des QueryParser wandelt das SimpleParse-Ergebnis in einen Queryplan um, indem unter anderem überflüssige Zwischenschritte entfernt und Positionsangaben durch die repräsentierten Werte er-

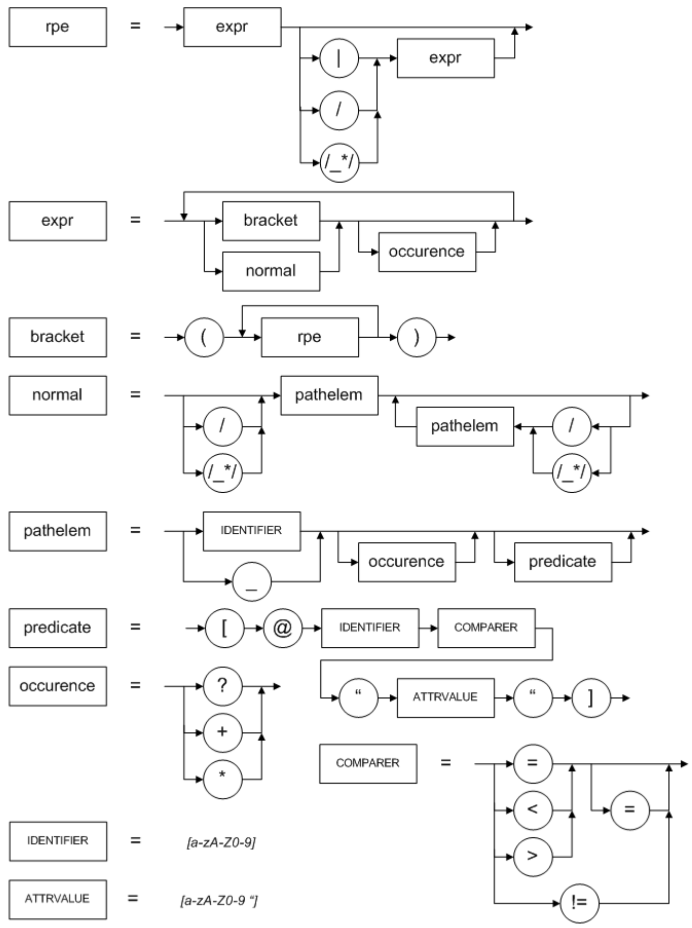

Abbildung 4.2: Die EBNF für RPE-Ausdrücke

setzt werden. Weiterhin wird die Reihenfolge der Elemente für eine rekursive Verwendung im QueryProcessor optimiert.

Der Queryplan besteht aus dem Format `[Funktion, Argument, Argument]`, wobei `Argument` entweder ein Tupel der Form `(Art, Wert)` oder einen weiteren Funktionsaufruf repräsentiert.

Beispielhaft liefert der Parser für den Query `Q2` (Listing 4.3) den in Abbildung 4.3 grafisch und in Listing 4.4 programmatisch dargestellten Aufrufbaum.

```
(Q2): (E1/E2)*/E3/((E4[@A=v])|(E5/_*/E6))
```

Listing 4.3: Query Q2 (aus [LM01])

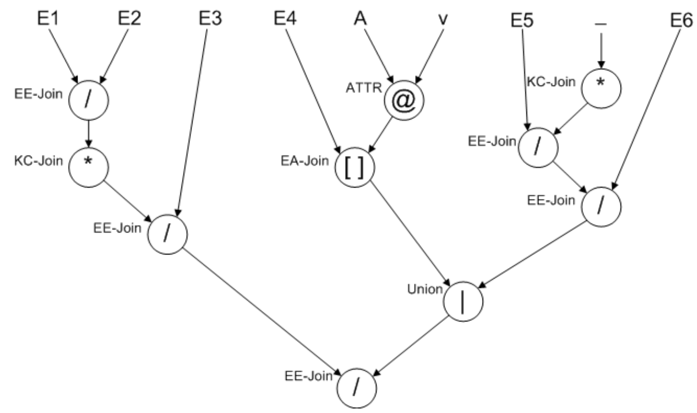

Abbildung 4.3: Aufrufbaum für Query Q2 (aus [LM01])

Dieser Queryplan kann nun im Processor flexibel eingesetzt werden, um die Join-Algorithmen in der richtigen Reihenfolge und mit den richtigen Argumenten anzuwenden.

```
['EEJOIN',
  ['EEJOIN',
    ['KCJOIN',
      '*',
      ['PREC',
        ['EEJOIN',
          ('ELEM', 'E1'),
          ('ELEM', 'E2')
  ] ] ],
    ('ELEM', 'E3')
  ],
  ['PREC',
    ['UNION',
      ['PREC',
        ['EAJOIN',
          ['ATTR', ('A', 'v', '=')],
          ('ELEM', 'E4')
      ] ],
      ['PREC',
        ['EEJOIN', ('ELEM', 'E5'), '/_*/', ('ELEM', 'E6')]
] ] ] ]
```

Listing 4.4: Queryplan für Query Q2

4.1.2 Die ObjectCollection

Die ObjectCollection hat zwei wesentliche Aufgaben: sie bündelt die Indextabellen und Join-Algorithmen und sorgt für die Kommunikation mit der ZODB. Die Implementierung der Indextabellen werden in Kapitel 4.1.2.2 erläutert, die Join-Algorithmen in Kapitel 4.1.2.1.

Für die Kommunikation mit der ZODB stehen zwei Methoden zur Verfügung:

- *add(oid, poid=None, cycle_prevention=None)*
  Die Methode add ist für das Hinzufügen des Objektes mit der ID *oid* in die Indexstrukturen verantwortlich. Es werden weiterhin alle Objekte, die von diesem Objekt aus referenziert sind, *rekursiv* hinzugefügt. Wird die Variable *poid* übergeben, so wird das Objekt 'unterhalb' des Objektes mit der ID *poid* abgelegt, ansonsten 'unter' die Wurzel. Die Variable *cycle_prevention* ist eine interne Liste, die die OIDs aller hinzugefügten Objekte speichert und für die Validierung auf Kreisreferenzen benötigt wird.

- *delete(oid, poid=None)*
  Die Methode delete ist für das *rekursive* Löschen des Objektes *oid* und aller seiner referenzierten Objekte aus den Indexstrukturen verantwortlich. Es werden nur Objekte gelöscht, die nach dem Löschen der Referenz keine eingehenden Referenzen haben. Wird die Variable *poid* übergeben, so wird das Objekt nur unter dem Objekt *poid* gelöscht, was dazu führen kann, dass das Objekt erhalten bleibt, weil ein anderes Objekt noch auf dieses referenziert.

Diese beiden Methoden müssen von der ZODB aufgerufen werden, sobald ein Objekt hinzugefügt beziehungsweise gelöscht wird. So können die Indexstrukturen aktuell gehalten werden. Für das Initialisieren der Indexstrukturen mit einer bestehenden Datenbank reicht es, die add-Methode mit dem Wurzelobjekt der Datenbank einmal aufzurufen.

Details zur Speicherung in den Indexstrukturen liefert Kapitel 4.1.2.2.

4.1.2.1 Der QuerySupport

Ein Pfadausdruck kann zur Verarbeitung in mehrere, einfache Unterausdrücke zerlegt werden. Jeder dieser Unterausdrücke liefert Zwischenergebnisse, welche anschließend über Verbund- ($|$) oder Join-Algorithmen kombiniert werden, um das Endergebnis zu erzielen. Generell gibt es in den RPE die folgenden minimalen Unterausdrücke[1]:

1. *Element oder Attribut*
   Der Unterausdruck besteht entweder aus einem einzelnen Element oder einem einzelnen

[1] Vgl. [LM01]

Attribut.
Beispiel: `nachname`

2. *Element und Attribut*
   Der Unterausdruck besteht aus einem Element mit einem Prädikatausdruck.
   Beispiel: `Student[@vorname="Max"]`

3. *zwei Elemente*
   Der Unterausdruck besteht aus einem Pfadausdruck minimaler Länge.
   Beispiel: `Kurs/Student`

4. *Kleensche Hülle*
   Der Unterausdruck ist eine Kleensche Hülle (`*`, `?`, `+`) eines anderen Unterausdrucks.
   Beispiel: `Student+`

5. *Vereinigung*
   Der Unterausdruck ist eine Vereiningung (Union) zweier Unterausdrücke.
   Beispiel: `Kurs|Student`

Unterausdrücke vom Typ (1) werden über den Zugriff auf den entsprechenden Index – Klassenindex (Kapitel 3.2.2.1) oder Attributindex (Kapitel 3.2.2.2) – ausgeführen. Typ (5) entspricht einer Vereinigung der übergebenen Ergebnismengen. Für die Unterausdrücke vom Typ (2), (3) und (4) werden in [LM01] folgende drei Algorithmen vorgestellt:

- der $\mathcal{EA}$-Join Algorithmus für Ausdrücke vom Typ (2)
- der $\mathcal{EE}$-Join Algorithmus für Ausdrücke vom Typ (3)
- der $\mathcal{KC}$-Join Algorithmus für Ausdrücke vom Typ (4)

Der $\mathcal{EA}$-Join-Algorithmus

Die Implementierung des $\mathcal{EA}$-Join entspricht der Bildung einer Schnittmenge zweier Listen. Zur besseren Erklärung folgendes Beispiel: `Student[@vorname="Max"]`. Der Algorithmus bekommt zwei Objektlisten übergeben. Die erste Liste enthält alle Objekte, die von der Klasse `Student` instantiiert sind (erhältlich über den Klassenindex). Die zweite Liste enthält alle Objekte, welche das Attribut `vorname` mit dem Wert ‘Max’ besitzen. Diese erhält man über den Attribut- und Werteindex[2]. Der $\mathcal{EA}$-Join berechnet aus beiden Listen die Schnittmenge.

[2]Ist kein Werteindex implementiert, müssen alle Objekte, die der Attributindex liefert, nach dem Wert ‘Max’ gefiltert werden.

Diese Implementierung bildet die ursprüngliche Funktionalität des $\mathcal{EA}$-Joins ab. Im Kapitel 2.5 habe ich vorgeschlagen, zusätzlich zur Möglichkeit, in Prädikatausdrücken auf Gleichheit zu vergleichen, auch die Vergleiche auf `<`, `>`, `<=`, `>=` sowie `!=` (Ungleich) zu implementieren.

Python ermöglicht hier eine elegante Variante: ein Dictionary[3] wird so implementiert, dass es einer Zuordnung Operator <-> Funktion entspricht. Als Funktion kommen sogenannte Lambda-Methoden[4] zum Einsatz. Listing 4.5 zeigt das implementierte Dictionary.

```
self.comp_map = {
    "==": lambda x, y: x == y,
    "<": lambda x, y: x < y,
    "<=": lambda x, y: x <= y,
    ">": lambda x, y: x > y,
    ">=": lambda x, y: x >= y,
    "!=": lambda x, y: x != y }
```

Listing 4.5: Mapping für die Vergleichsoperationen des $\mathcal{EA}$-Joins

Der Aufruf `self.comp_map["!="](23, 42)` vergleicht beispielweise die Werte 23 und 42 auf Ungleichheit und liefert das Ergebnis `True` zurück. Hierbei wird diejenige `lambda`-Methode, die sich im Dictionary hinter dem Schlüssel `!=` verbirgt, aufgerufen, und das Ergebnis von `x != y` zurückgegeben.

Der $\mathcal{EE}$-Join-Algorithmus

Der Element-Element-Join bekommt zwei Listen mit Objekten und gibt all jene Objekte aus der zweiten Liste zurück, die direkte Nachfolger (Kinder) der Objekte der ersten Liste sind. Hierfür wird der Strukturindex (Kapitel 3.2.2.3) benötigt, der eine schnelle Überprüfung dieser Beziehung ermöglicht.

Ein Spezialfall des $\mathcal{EE}$-Joins ist die Möglichkeit, nicht nur auf direkte Nachfolgerschaft, sondern auch auf generelle Nachfolgerschaft (mittels '`/_*/`') zu prüfen. Die Anfrage `Kapitel/_*/Abbildung` bestimmt zum Beispiel alle direkten und indirekten Nachfolger vom Typ 'Abbildung' der Objekte vom Typ 'Kapitel'. Diese Operation ist normalerweise sehr viel aufwändiger als die oben genannte direkte Nachfolgerschaft, lässt sich allerdings mit Hilfe des Strukturindexes vergleichsweise schnell berechnen.

Der $\mathcal{KC}$-Join-Algorithmus

Der $\mathcal{KC}$-Join ist für die Auflösung der Operatoren **, ?* und + verantwortlich. $\mathcal{KC}$ steht für die Abkürzung Kleenesche Hülle (Kleene Closure), welche die Menge all jener Pfade ist, die durch beliebige Verknüfung aller Objekte der Eingabemenge entstehen. Das bedeutet am Beispiel der

[3] Vgl. Kapitel 3.2.2.1

[4] Lamdba-Funktionen sind kleine anonyme Funktionen, die direkt verwendet oder einer Variable zugeordnet werden können.

Kleeneschen Hülle `Kapitel*`, welche eine beliebige Anzahl Kapitel darstellt, dass diese in die Pfade `Kapitel`, `Kapitel/Kapitel`, `Kapitel/Kapitel/Kapitel` usw. aufgelöst wird. Bei den Operatoren * und *?* kommt noch der leere Pfad hinzu, da eine Nichtexistenz des Objektes in der Definition (siehe hierzu Tabelle 2.6 auf Seite 28) enthalten ist.

Der Algorithmus nutzt den $\mathcal{EE}$-Join, um für zwei Objekte der Eingabemenge zu überprüfen, ob diese in einer direkten Nachbarschaftsbeziehung stehen. Im ersten Ausführungschritt entsteht dadurch eine Menge zweistufiger Pfade. Diese wird im zweiten Schritt analog gegen die Objekte der Eingabemenge gejoint, wodurch eine Menge dreistufiger Pfade entsteht. Dieser Vorgang wird solange wiederholt, bis keine Pfade mehr erstellt werden können.

Für eine Objektmenge x entstehen somit im ersten Schritt maximal $x*(x-1)$ Pfade der Länge zwei. Dies entspricht allen Pfaden, die durch beliebige Verknüpfung der Objekte entstehen abzüglich der Pfade von Objekten mit sich selbst (diese entsprechen den Objekten der Eingabemenge).

Im zweiten Schritt entstehen dann durch die Verknüpfung der $x*(x-1)$ Pfade des ersten Schrittes mit den x Objekten der Eingabemenge $x*(x-1)^2$ Pfade der Länge 3. Es werden wieder alle Kreise der Länge 1 eliminiert. Kreise der Länge > 1 sind allerdings erlaubt.

Im dritten Schritt ergeben sich dann maximal $x*(x-1)^3$ Pfade der Länge 4.

Der $\mathcal{KC}$-Join hat demnach eine worst-case Laufzeit von $O(x*(x-1)^{y-1})$, wobei x der Menge an Objekten der Eingabemenge und y der Höhe des Objektgraphen und damit der maximal möglichen Länge von Pfaden entspricht. Diese Laufzeit wird allerdings nur erreicht, wenn der Objektgraph aus sehr vielen Kreisen der Länge > 1 besteht.

4.1.2.2 Der Indexsupport

Der Indexsupport bündelt alle wichtigen Indexstrukturen, die den Querysupport beim Ermitteln der gesuchten Objekte unterstützen. Alle Indexstrukturen bestehen aus einem `OOBTree`, also einem B-Baum, der Objekte als Schlüssel und als Werte aufnimmt. Eine detaillierte Erklärung, welche Indexstrukturen sich eignen und warum ein B-Baum die beste Indexstruktur darstellt, wurde in Kapitel 3.1 erläutert.

Folgende Indexstrukturen sind für die Algorithmen des Querysupport wichtig:

- *Klassenindex*
  Der Klassenindex mappt Klassennamen auf eine Liste von `OID`s von Objekten, welche vom Typ dieser Klasse sind.

- *Attributindex*
  Der Attributindex mappt Attributnamen auf eine Liste von `OID`s von Objekten, die diese Attribute besitzen.

- *Strukturindex*
  Der Strukturindex ermöglicht die effiziente und schnelle Bestimmung von Nachbarschaftsbeziehungen zwischen Objekten. Er mappt `OID`s auf eine Liste von Pfadtupeln.

Genauere Informationen zu den einzelnen Indizes sind im Kapitel 3.2 und die Implementierung im Folgenden erklärt.

Klassen- und Attributindex

Der Klassen-(Attribut-)index mappen Klassen-(Attribut-)namen (Zeichenketten, Strings) auf Listen. Beides sind in Python Objekte, weshalb als B-Baum nur ein `OOBTree` in Frage kommt. Der Index selbst wird in die Datenbank mitgespeichert, damit dieser beim erneuten Laden der DB nicht neu aufgebaut werden muss. Für die nachgelagerten Listen kommen `OOTreeSets` zur Verwendung. Diese sind für das Arbeiten in der ZODB optimiert, denn beim Lesen und Schreiben muss nicht die gesamte Liste in den Arbeitsspeicher geschrieben, sondern Elemente können einzeln selektiert und gespeichert werden.

Der Klassen- und der Attributindex haben folgende Methoden:

- *insert(key, value)*
  Der Wert *value* (die `OID` eines Objektes) wird der Klasse bzw. dem Attributnamen *key* zugeordnet. Ist *key* noch nicht im Index, so wird er eingefügt und ein `OOTreeSet` mit dem vorerst einzigen Wert *value* erzeugt. Andernfalls wird *value* in das `OOTreeSet` des *key* eingefügt.

- *delete(key, value)*
  Löscht *value* aus dem `OOTreeSet` des *key*. Ist das `OOTreeSet` nach dem Löschen leer, so wird der Schlüssel *key* aus dem Index gelöscht.

- *get(key)*
  Gibt eine Liste gefüllt mit den Werten des `OOTreeSet` des Schlüssels *key* zurück.

- *has_key(key)*
  Gibt `True` zurück, wenn *key* im Index vorhanden ist, sonst `False`.

- *all()*
  Gibt eine distincte Liste mit allen Werten *aller* `OOTreeSets` im Index zurück.

Strukturindex

Der Strukturindex wird ähnlich dem Klassen- und Attributindex implementiert. Er mappt `OIDs` auf Listen von Pfadtupeln ab. Für die interne Organisation ist weiterhin eine Datenstruktur nötig, die zu jeder `OID` die Kinder speichert. Sie wird benötigt, um beim Löschen eines Tupels auch rekursiv die entsprechenden Tupel der Kinder löschen zu können, denn diese sind dann über den gelöschten Knoten nicht mehr erreichbar. Da der Strukturindex unabhängig arbeiten soll, ist ein Zugriff auf den Objektgraphen hierfür nicht vorgesehen, sodass eine interne Zuordnung *OID -> Liste von OIDs der Kinder* nötig ist.

Der Strukturindex hat folgende Methoden:

- *insert(key, parent=None)*
  Erstellt für *key* ein neues Pfadtupel mit dem Vater *parent*. Ist *parent* `None`, so wird die Wurzel als Vater genommen. Weiterhin werden rekursiv alle Nachfolger mit neuen Pfadtupeln aktualisiert.

- *delete(key, parent=None)*
  Löscht für *key* das Pfadtupel, welches die Erreichbarkeit über *parent* gewährleistet. Ist *parent* `None`, so werden alle Pfadtupel von *key* gelöscht. Weiterhin werden rekursiv von allen Nachkommen die entsprechenden Tupel entfernt.

- *is_child(key1, key2)*
  Prüft durch Vergleich der Pfadtupel, ob *key1* ein direkter Nachfolger (ein Kind) von *key2* ist. Gibt `True` bzw. `False` zurück.

- *is_successor(key1, key2)*
  Gibt `True` zurück, wenn *key1* ein Nachfolger von *key2* ist, ansonsten `False`. Dies geschieht über den Vergleich der Pfadtupel, eine (unter Umständen langwierige) Traversierung des Objektgraphen wird vermieden[5].

- *get(key)*
  Gibt eine Liste von Pfadtupeln für *key* zurück.

- *validate(key, indexlist)*
  Gibt `True` zurück, wenn *key* über einen der in `indexlist` enthaltenen Pfadtupel erreichbar ist, ansonsten `False`.

4.2 Anbindung an die ZODB

Für die Anbindung der Indexstrukturen an die ZODB existieren zwei Szenarien:

- *einmaliges Indizieren einer bestehenden Datenbank*
  Eine bestehende, unindizierte Datenbank muss vor dem ersten Query einmal komplett indiziert werden. Dies ist auf zwei Arten implementierbar:

[5] Vgl. Abschnitt 3.2.2.3

  - *aktive Indizierung*
    Der Anwender muss, nachdem er die ObjectCollection initialisiert hat, das Indizieren der Indexstrukturen manuell starten.
  - *passive Indizierung*
    Das Indizieren ist Bestandteil der Initialisierung der ObjectCollection und muss nicht aktiv vom Anwender gestartet werden.

- *kontinuierliches Aktualisieren der Indexstrukturen*
  Hierfür gibt es drei Implementierungsmöglichkeiten:
  - *aktive Anbindung*
    Die Indexstrukturen lauschen an der Connection zur ZODB und aktualisieren ihre Strukturen, sobald sich der Datenbestand ändert.
  - *passive Anbindung*
    Die ZODB benötigt eine Funktionalität, die Änderungen am Datenbestand propagiert, damit die Indexstrukturen aktualisiert werden können.
  - *direkte Anbindung*
    Die ZODB ruft bei Änderungen am Datenbestand direkt die Methoden der ObjectCollection auf.

Für die initiale Indizierung einer bestehenden Datenbank ist die Implementierung der **passiven Indizierung** vorzuziehen, da eine Benutzung der ObjectCollection ohne diesen Vorgang nicht möglich ist und daher beim Initialisieren der ObjectCollection automatisch gestartet werden sollte. Allerdings muss eine entsprechende Statusmeldung ausgegeben werden, da der Vorgang bei besonders großen Datenbanken recht lange dauern kann[6].

Für das kontinuierliche Indizieren ist die **direkte Anbindung** zu bevorzugen. Die ZODB muss dahingehen angepasst werden, dass sie die von der ObjectCollection bereitgestellten Methoden (add, delete, update) aufruft, sobald sich ihr Datenbestand ändert.

4.3 Probleme und Lösungen

Dieses Kapitel beschreibt einige Probleme bei der Implementierung und diskutiert deren mögliche Lösungen.

4.3.1 Mehrfaches Hinzufügen gleicher Objekte

Durch mehrfaches Einfügen eines Objektes in den Objektgraphen entstehen sogenannte Crosskanten. In einem Objektgraphen, der nur aus Baumkanten besteht, haben alle Objekte das gleiche

[6] für die Performance der Indexstrukturen siehe Abschnitt 5.1.2

Gewicht, das heißt die gleiche Anzahl Vorgänger (nämlich genau einen). Crosskanten verändern die Gewichtung der Objekte, so dass einige Objekte mehrere Vorgänger haben.

Da der Strukturindex pro Objekt im Objektgraphen nur ein Pfadtupel ermöglicht, ein Objekt mit n Vorgängern allerdings mindestens n Pfadtupel benötigt, muss der Strukturindex erweitert werden. Diese Erweiterung wurde so implementiert, dass pro Objekt eine *Liste* von Tupeln im Index angelegt wird. Diese besteht beim ersten Einfügen des Objektes aus genau einem Element, bei jedem weiteren Hinzufügen muss nur das neue Pfadtupel an die Liste angehängt werden. Beim Löschen einer Objektreferenz muss darauf geachtet werden, dass das *richtige* Pfadtupel aus der Liste entfernt wird.

4.3.2 Kreisreferenzen

Beim mehrfachen Einfügen kann es zu Kreisreferenzen über Rückwärtskanten kommen. Diese entstehen, wenn das Objekt X unter einen seiner Nachfahren Y gehängt wird. Y zeigt dann auf X (Rückwärtskante) und damit auf die Wurzel seines Teilbaumes, der mit X beginnt. Kreise führen zu unendlichen Ausführungsschritten bei der Traversierung.

Eine Lösungsmöglichkeit besteht in der frühzeitigen Prävention von Kreisen. Vor jedem Einfügen eines Objektes in die Indexstrukturen wird geprüft, ob ein Kreis entsteht und bei einem positiven Ergebnis das Einfügen abgebrochen. Geprüft wird folgendermaßen: Ist der einzufügende Knoten in der Liste aller seiner Vorgänger enthalten, so entsteht beim Hinzufügen ein Kreis. Diese Variante der Prävention hat den Nachteil, dass Kreise prinzipiell nicht unterstützt werden. Die ZODB lässt allerdings Kreisreferenzen zu, sodass im Falle der Implementation dieser Variante ZODB-Funktionalität außen vor gelassen würde.

Die Indexstrukturen müssen also Kreisreferenzen unterstützen. Probleme hierbei können an drei Stellen auftreten:

1. *Hinzufügen*
   Beim Hinzufügen neuer Knoten zum Index darf der ObjectParser[7] nicht in eine Endlosschleife laufen.

2. *Löschen*
   Beim Löschen von Knoten aus dem Index darf der ObjectParser nicht in eine Endlosschleife laufen.

3. *Anfragen*
   Beim Anfragen von Beziehungen zwischen zwei Objekten darf der Strukturindex nicht in eine Endlosschleife laufen.

[7]Der ObjectParser ist für das Durchsuchen eines Objektes nach referenzierten Objekten (Kindern) zuständig. Er ist im QuerySupport enthalten.

Um Problem (1) zu lösen, muss beim Hinzufügen jedes Objekt als 'schon besucht' markiert werden. Hierfür wird eine Liste, die alle hinzugefügten Objekte enthält, mitgeführt. Ist das hinzuzufügende Objekt schon in dieser Liste enthalten, wird das rekursive Hinzufügen abgebrochen und mit dem nächsten Objekt weitergemacht.

Problem (2) rührt daher, dass in der Regel zum Löschen eines Objektes erst alle Nachfolger rekursiv gelöscht werden müssen. Hier entsteht das gleiche Problem wie unter (1), und auch die Lösung gestaltet sich exakt genau so.

Der Strukturindex greift für die Anfrage von Beziehungen zwischen Objekten nicht auf die Objekte selbst, sondern auf die Pfadtupel im Index zurück. Der Index kann also nicht in eine Endlosschleife laufen. Trotzdem muss sichergestellt werden, dass Strukturanfragen, die Kreise enthalten, korrekte Ergebnisse liefern. Das lässt sich am Besten an einem Beispiel erklären: Gegeben seien drei Knoten `A, B` und `C`. `B` ist das Kind von `A`, `C` das Kind von `B` und `A` das Kind von `C`. Die Pfadtupel sehen in diesem Beispiel wie folgt aus (unter der Bedingung, dass `A` als erstes eingefügt wurde):

A: `[(A, ), (A, B, C, A)]`

B: `[(A, B), (A, B, C, A, B)]`

C: `[(A, B, C), (A, B, C, A, B, C)]`

Für jedes Element ist also der einfache Weg (ohne Kreis) und ein "einfacher Kreisumlauf" als Pfad gespeichert. Dieser einfache Kreisumlauf genügt für Kreisanfragen beliebiger Länge, denn für den QueryProcessor ist immer das Zwischenergebnis der einzelnen Beziehungsanfragen entscheidend. Somit können Anfragen wie zum Beispiel `query('A/B/C/A/B/C/A/B')` ohne Weiteres angefragt werden, denn der Strukturindex liefert für alle Beziehungsanfragen `A/B`, `B/C` sowie `C/A` *True*.

5 Bewertung und Ausblick

Dieses Kapitel befasst sich mit dem Testen und Bewerten der Implementierung. Hierfür wird in Abschnitt 5.1 ein Einblick in die Möglichkeiten der Softwaretests gegeben und anschließend ein Performance-Test durchgeführt. Abschnitt 5.2 schließt die Diplomarbeit mit einem Ausblick und weiterführenden Gedanken.

5.1 Testen und Bewerten der Implementierung

Der Softwaretest ist der "überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen." [Den92] Er soll die dauerhafte Qualität und Funktionalität der Software sicherstellen. Man unterscheidet zwei wichtige Arten von Softwaretests[1]:

- *Akzeptanztest*
  Der User Acceptance Test (UAT) ist ein Black-Box-Test[2] durch den Anwender (funktionaler UAT) und Hersteller (Produktions-UAT).

- *Systemtest*
  Beim Systemtest wird das gesamte System gegen seine Spezifikation getestet. Man unterscheidet den funktionalen und den nicht-funktionalen Systemtest.
  - *funktionaler Systemtest*
    Überprüft ein System auf funktionale Eigenschaften wie *Korrektheit* und *Vollständigkeit*.
  - *nicht-funktionaler Systemtest*
    Überprüft ein System auf nicht-funktionale Eigenschaften wie *Benutzbarkeit*, *Sicherheit* oder *Zuverlässigkeit*.

Softwaretests haben zum Ziel, das Restrisiko verbleibender Fehler zu minimieren und damit eine Bewertung der Qualität des Produktes zu ermöglichen. Dabei sollen Tests Fehler aufdecken und nicht die Abwesenheit von Fehlern beweisen. Deshalb ist ein erfolgloser Test auch kein Beweis für ein fehlerloses Programm.

[1] Vgl. [Wik08h, Abschnitt 2.1 und 2.2]
[2] Das Black-Box-Testen beschränkt sich auf das funktionsorientierte Testen mit Hilfe der Spezifikation, aber ohne Kenntnis der inneren Funktionsweise der Software.

5.1.1 Testen in Python

Python enthält zum Testen das *unittest* Framework[3]. Unit bezeichnet die kleinste testbare Einheit eines Programms, in der Regel Methoden des Programms. Python-Unittests bestechen durch Automatisierung, gemeinsame Benutzung von Initialisierungs- und Aufräumaufwand, Gruppierung von Tests und Unabhängigkeit der Tests vom Framework. Sie zählen zu den Akzeptanztests. Dabei unterstützen Unittests einige wichtige Konzepte:

- *Testverankerung*
  Spiegelt den Aufwand, der nötig ist, um Tests auszuführen, sowie alle damit verknüpften Aufräumarbeiten, wider.

- *Testfall*
  Ist die kleinste Testeinheit. Testet eine spezifische Antwort gegen eine Menge an Eingaben.

- *Testfolge*
  Eine Sammlung von Testfällen, Testfolgen oder beidem. Dient der Gruppierung von Tests, die zusammen ausgeführt werden sollen.

- *Testlauf*
  Der Testlauf organisiert die Ausführung der Tests und die Ausgabe des Ergebnisses an den Benutzer.

Ein weiteres Modul zum Testen von Python-Programmen sind *Doctests*[4]. Sie unterstützen die gleichen Konzepte wie Unittests, nutzen aber zusätzlich die Vorteile der Docstrings[5] und der Introspektion[6]. Doctests können in Testfolgen gruppiert und in externen Testläufen ausgeführt werden.

```
Importieren und Instantiieren des QueryParser:
    >>> from gocept.objectquery.pathexpressions import RPEParser
    >>> p = RPEParser()
Aufruf der parse()-Methode und Validierung des Ergebnisses:
    >>> p.parse('Buch/Kapitel')
    ['EEJOIN', ('ELEM', 'Buch'), ('ELEM', 'Kapitel')]
```

Listing 5.1: Doctest zum Testen des korrekten Parsens eines Pfadausdrucks

[3] Vgl. [Fou08b]
[4] Vgl. [Fou08a]
[5] Ein Docstring ist eine Zeichenkette, die einen bestimmten Codeabschnitt im Quelltext dokumentiert.
[6] Introspektion (auch als Reflexion bekannt) dient der Gewinnung von Informationen über Klassen und deren Instanzen zur Laufzeit.

Ein Doctest entspricht syntaktisch der interaktiven Python Shell. Einen einfachen Doctest aus dem QueryParser-Modul zeigt Listing 5.1. In diesem wird getestet, ob die Methode `parse()` (Zeile 5) einen einfachen Pfadausdruck in den richtigen Queryplan umwandelt. Liefert diese nicht den geforderten Ausdruck in Zeile 6, wird eine Fehlermeldung ausgegeben.

5.1.2 Testen der ObjectQuery

Für das ObjectQuery-Modul habe ich während der Implementierung über 100 Doctests geschrieben. Diese untergliedern sich in 5 Testfolgen:

- *pathexpressions.txt*
  Enthält 25 Testfälle zum Testen der Funktionalität des QueryParsers.
- *indexsupport.txt*
  Enthält 20 Testfälle zum Testen der Funktionalität des IndexSupport.
- *querysupport.txt*
  Enthält 16 Testfälle zum Testen der Funktionalität des QuerySupport.
- *collection.txt*
  Enthält 22 Testfälle zum Testen der Funktionalität der ObjectCollection.
- *processor.txt*
  Enthält 21 Testfälle zum Testen der Funktionalität des QueryProcessor.

```
sweh@krabappel:~/dev/objectquery$ bin/test --coverage
Running unit tests:
  Ran 106 tests with 0 failures and 0 errors in 2.638 seconds.
lines   cov%   module
   65   100%   gocept.objectquery.collection
  137   100%   gocept.objectquery.indexsupport
   71   100%   gocept.objectquery.pathexpressions
   53   100%   gocept.objectquery.processor
  121   100%   gocept.objectquery.querysupport
   36   100%   gocept.objectquery.testobjects
   11   100%   gocept.objectquery.tests
```

Listing 5.2: Testabdeckung der Objectquery

Für die Qualität der Tests spielt die *Testabdeckung* eine entscheidende Rolle. Als Testabdeckung bezeichnet man “das Verhältnis an tatsächlich getroffenen Aussagen eines Tests gegenüber den theoretisch möglich treffbaren Aussagen bzw. der Menge der gewünschten treffbaren Aussagen.” [Wik08i]

Höhe	Objekte	bi	q1	q2	q3	q4	q5
8	511	0.15s	0.00s	0.01s	0.03s	0.01s	0.03s
10	2047	0.70s	0.01s	0.03s	0.13s	0.03s	0.10s
12	8191	2.92s	0.06s	0.14s	0.57s	0.11s	0.14s
14	32767	12.57s	0.23s	0.68s	2.68s	0.53s	0.30s
16	131071	54.76s	1.24s	3.70s	13.01s	2.45s	1.63s
17	262143	120.50s	3.21s	10.32s	30.08s	7.25s	4.02s
18	524287	260.00s	10.07s	28.29s	68.41s	20.49s	10.85s

Tabelle 5.1: Ergebnis des Performance-Tests für die ObjectCollection (Auswahl)

Listing 5.2 zeigt die Testabdeckung für alle Module der ObjectQuery auf Basis der ausgeführten Codezeilen. Es ist eine Abdeckung von 100% angegeben, das heißt, dass die geschriebene Testsuite jede Codezeile mindestens einmal ausführt. Ein Programm definiert sich allerdings nicht nur aus seinen Codezeilen, sondern auch aus seiner Logik. Es ist gewissermaßen ein Automat mit einer (unbekannten) Anzahl an Zuständen. Eine Testfolge, die eine vollständige Testabdeckung auf einen solchen Automaten erreichen will, muss nicht nur alle Codezeilen des Programms ausführen (wie die Testsuite in Listing 5.2), sondern auch alle möglichen Zustände des Programmes und deren Ergebnis testen. Für das vollständige Testen dieser Programmlogik existieren drei Kriterien[7]:

1. Werden bei den Tests alle Zustände durchlaufen?
2. Werden alle Zustandsübergänge durchlaufen?
3. Werden alle Ereignisse, die Zustandsübergänge hervorrufen, getestet?

Die Vollständigkeit dieser zustandsbasierten Testfälle ist aufgrund ihrer Komplexität nicht meß- und erreichbar. Da die Testfälle für die ObjectQuery parallel zur Implementierung geschrieben wurden, kann allerdings davon ausgegangen werden, dass eine signifikant hohe Anzahl Zustände und Zustandsübergänge innerhalb der ObjectQuery durch Testfälle abgedeckt sind.

Ein weiteres Qualitätsmerkmal ist die *Performance* der ObjectQuery. Diese hängt von der Geschwindigkeit der Indexstrukturen und der Join-Algorithmen ab. Zum Testen dieser beiden Komponenten habe ich ein Script geschrieben, welches die Indexstrukturen (inklusive ZODB) mit einer gewissen Anzahl Objekte füllt und anschließend einige Testanfragen durchführt. Sowohl für das Füllen der Indizes als auch das Anfragen wird die beanspruchte Zeit gemessen.

Dieses Script ist im Anhang A.1 hinterlegt. Es füllt die ZODB mit einem binären Baum der Höhe 2 bis 20. Dieser enthält `Dummy`-Objekte mit dem Attribut `id` nach Postorder[8] nummeriert.

[7] Vgl. [Wik08l]

[8] Bei der Postorder-Traversierung wird zuerst der linke, dann der rechte Teilbaum durchlaufen und anschließend die Wurzel betrachtet.

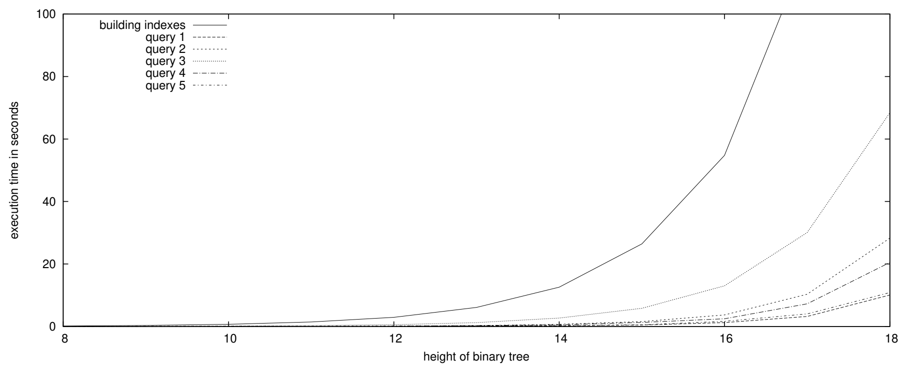

Abbildung 5.1: Ergebnis des Performance-Tests in graphischer Darstellung

Anschließend wird die Dauer für das Füllen der Indexstrukturen gemessen (`bi`). Danach werden fünf Anfragen abgeschickt, für die die Ausführungszeit ermittelt wird (`q1` bis `q5`). Die Zeiten, gemessen auf einem T7200[9] Prozessor mit 2 Gigabyte Arbeitsspeicher, können Tabelle 5.1 entnommen werden. Die Anfragen `q1` bis `q5` sind im Folgenden erklärt:

- `q1: 'Dummy[@id="%i"]' %(objects/2)`
  Holt das Objekt, das in der Postorder-Darstellung des Baumes in der Mitte liegt. Demonstriert die Performance eines einfachen Prädikatausdrucks.

- `q2: '/Dummy'`
  Holt alle Objekte, die unterhalb der Wurzel liegen. Demonstriert die Performance eines einfachen $\mathcal{EE}$-Join.

- `q3: '/Dummy/Dummy/Dummy'`
  Holt alle Objekte, die in der dritten Ebene liegen. Demonstriert die Performance geschachtelter $\mathcal{EE}$-Joins.

- `q4: 'Dummy[@id="%i"]/_*/Dummy[@id="1"]' %objects`
  Holt das Objekt mit der kleinsten `id`, welches unterhalb des Objektes mit der höchsten `id` (der Wurzel) liegt. Demonstiert die Performance des $\mathcal{KC}$-Joins, da dieser über die gesamte Höhe des Baumes geht.

- `q5: 'Dummy[@id<"1000"]'`
  Holt alle Objekte, deren `id` kleiner 1000 ist. Demonstriert die Performance, wenn viele Objekte in die Ergebnismenge aufgenommen werden.

[9]Einzelheiten zur Spezifikation des T7200 Prozessors sind im Internet unter der Adresse `http://www.intel.com/products/processor/core2duo/specifications.htm` verfügbar.

Das Ergebnis der Performance-Tests ist in Abbildung 5.1 graphisch dargestellt. Es ist gut zu erkennen, dass der Zeitaufwand für die Anfragen weit unter dem Aufwand für den Aufbau der Indexstrukturen liegt.

Anzumerken ist, dass die erzielten Zeiten der Anfragen als worst-case Zeiten zu interpretieren sind, da der gesamte Objektgraph mit Objekten von genau einem Typ (Dummy) gefüllt wird. Dadurch enthält zum Beispiel der Klassenindex für eine DB mit 1000 Objekten vom gleichen Typ genau einen Knoten mit einer Liste von 1000 Elementen. Dementsprechend bekommt auch der $\mathcal{EE}$-Join für jede Elementanfrage 1000 Objekte, die er verarbeiten muss. Bei einer 'durchschnittliche' DB mit 1000 Objekten gleichverteilt auf 10 Typen enthält der Klassenindex 10 Knoten mit 100 Elementen. Die Performance ist also nicht von der Größe der Datenbank, sondern von der Anzahl Elemente, die bei den Joins bearbeitet werden müssen, abhängig. Analog verhält es sich beim Attribut- und Strukturindex.

Die Ergebnisse der Tests bestätigen eine gute Performance der Implementierung. Ein produktiver Einsatz ist möglich.

5.2 Ausblick und weiterführende Gedanken

Um die in der Diplomarbeit geleistete Arbeit zu einem produktiven Einsatz zu führen, sollten meiner Meinung nach folgende Schritt folgen:

1. *Anbindung an die ZODB*
   Es fehlt noch die automatische Anbindung an die ZODB. Diese muss als erstes implementiert werden, um die Benutzbarkeit für den Anwender zu verbessern.

2. *Betatest in der Community*
   Die aktuelle Version sollte als Testversion in der Zope-Community vorgestellt werden mit dem Ziel, möglichst viel Feedback zu erhalten. Dieses betrifft unter anderem Fehler (Bugs), Probleme und Anregungen.

3. *Implementieren einer Anfragesprache und Indexstrukturen*
   Die RPE haben nicht den Funktionsumfang, um als produktive Anfragesprache eingesetzt zu werden. Daher sollte eine Sprache implementiert werden, die sich aufgrund der in der Diplomarbeit durchgeführten Evaluation als geeignet herausgestellt hat. Die neue Sprache wird unter Umständen neue Indexstrukturen benötigen. Außerdem wurden im Rahmen der Diplomarbeit auch nicht alle vorgestellten Indexstrukturen implementiert (zum Beispiel der Werteindex).

Die Diplomarbeit hat gezeigt, dass eine Anfragesprache für die ZODB implementierbar ist. Durch sie wird es erst möglich, gezieht nach Objekten in der ZODB zu suchen.

Literaturverzeichnis

[Atk91] Malcom Atkinson. *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases*. 1991.

[BCF+07] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, and Jonathan Robie Jérôme Siméon. Xquery - an xml query language. `http://www.w3.org/TR/xquery/`, 2007.

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. a graphical language for querying and restructuring xml documents. `http://citeseer.ist.psu.edu/ceri99xmlgl.html`, 1999.

[CD99] James Clarc and Steve DeRose. Xml path language (xpath). `http://www.w3.org/TR/xpath/`, 1999.

[CFMR00] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. Xml query requirements. `http://www.w3.org/TR/2000/WD-xmlquery-req-20000815`, 2000.

[Cla99] James Clarc. Xsl transformations. `http://www.w3.org/TR/xslt#patterns`, 1999.

[CR06] William R. Cook and Carl Rosenberger. Native queries for persistent objects. `http://www.cs.utexas.edu/~wcook/papers/NativeQueries/NativeQueries8-23-05.pdf`, 2006.

[CT97] Daniel K.C. Chan and Philip W. Trinder. A processing framework for object comprehensions. `http://citeseer.ist.psu.edu/384122.html`, 1997.

[Den92] Ernst Denert. *Software–Engineering*. 1992.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Lavy, and Dan Suciu. Xml-ql: A query language for xml. `http://www.w3.org/TR/NOTE-xml-ql/`, 1998.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, 1982.

[DSB04] TU Berlin Dr. Susanne Busse. Kopplung von programmiersprachen und datenbanken. http://cis.cs.tu-berlin.de/Lehre/WS-0304/Sonstiges/db-pages/Inhalt/kopplung_pl_dbs/, 2004.

[EN02] Ramez Elmasri and Shamkant B. Navathe. *Grundlagen von Datenbanksystemen*. 2002.

[Fle06] Mike C. Fletcher. Simpleparse - a parser generator for python. http://simpleparse.sourceforge.net/, 2006. Version 2.1.0a1.

[Fou08a] Python Software Foundation. doctest – test interactive python examples. http://docs.python.org/lib/module-doctest.html, 2008.

[Fou08b] Python Software Foundation. Unit testing framework. http://docs.python.org/lib/module-unittest.html, 2008.

[Goe08] Prof. Dr. Andreas Goerdt. Theoretische informatik i - vorlesungsscript. http://www.tu-chemnitz.de/informatik/TI/ti1_ws20072008/skript_ti1_ws20072008.pdf, 2008.

[Gro03] Object Database Management Group. Fastobjects oql reference. http://www.fh-wedel.de/~apf/oodb-uebung/infomaterial/OQLReference.pdf, 2003.

[HMH91] Andreas Heuer and Scholl Marc H.˙ Principles of object-oriented query languages. http://www.inf.uni-konstanz.de/dbis/publications/download/HS:BTW91.ps.gz, 1991.

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*. 2001.

[HS99] Andreas Heuer and Gunter Saake. *Datenbanken - Konzepte und Sprachen*. 1999.

[Kö01] Dirk Köhler. Xml - query sprachen. http://lips.informatik.uni-leipzig.de/pub/2001-7, 2001.

[LA07] Mark Lutz and David Ascher. *Einführung in Python*. 2007.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions (en). http://xml.coverpages.org/BongkiMoonVLDB2001.pdf, 2001.

[Pul93] Stephen G. Pulman. *Grammars and Parsers*. 1993.

[Qua98] Dallan Quass. Ten features necessary for an xml query language (en). http://www.w3.org/TandS/QL/QL98/pp/quass.html, 1998.

[RFC00] Jonathan Robie, Daniela Florescu, and Don Chamberlin. Quilt - an xml query language. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf, 2000.

[Tal06] Talin. A simple, lightweight lexical analyzer. http://cheeseshop.python.org/pypi/reflex, 2006. Version 0.1.

[vW07] Philipp von Weitershausen. *Web Component Development with Zope3*. 2007.

[Wik07a] Wikipedia. Abfragesprache (de). http://de.wikipedia.org/wiki/Abfragesprache, 2007.

[Wik07b] Wikipedia. Multiversion concurrency control. http://de.wikipedia.org/wiki/Multiversion_Concurrency_Control, 2007.

[Wik07c] Wikipedia. Object-relational-mapping (de). http://de.wikipedia.org/wiki/Object-Relational_Mapping, 2007.

[Wik08a] Wikipedia. Acid (en). http://en.wikipedia.org/wiki/ACID, 2008.

[Wik08b] Wikipedia. Datenbankindex. http://de.wikipedia.org/wiki/Datenbankindex, 2008.

[Wik08c] Wikipedia. Document object model (de). http://de.wikipedia.org/wiki/Document_Object_Model, 2008.

[Wik08d] Wikipedia. Extensible markup language (de). http://de.wikipedia.org/wiki/Extensible_Markup_Language, 2008.

[Wik08e] Wikipedia. Hash-funktion. http://de.wikipedia.org/wiki/Hash-Funktion#Definition_einer_Hash-Funktion, 2008.

[Wik08f] Wikipedia. Ordnungsrelationen (de). http://de.wikipedia.org/wiki/Ordnungsrelation, 2008.

[Wik08g] Wikipedia. Parser. http://de.wikipedia.org/wiki/Parser, 2008.

[Wik08h] Wikipedia. Testabdeckung (de). http://de.wikipedia.org/wiki/Softwaretest, 2008.

[Wik08i] Wikipedia. Testabdeckung (de). http://de.wikipedia.org/wiki/Testabdeckung, 2008.

[Wik08j] Wikipedia. Wald (graphentheorie). http://de.wikipedia.org/wiki/Wald_(Graphentheorie), 2008.

[Wik08k] Wikipedia. World wide web consortium. http://de.wikipedia.org/wiki/World_Wide_Web_Consortium, 2008.

[Wik08l] Wikipedia. Zustandsbasierte testmethoden (de). http://de.wikipedia.org/wiki/Testmethoden#Zustandsbasierte_Testmethoden, 2008.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Halle, den 23. April 2008

Sebastian Wehrmann

Anhang

A Quelltexte

```
import transaction, time
from gocept.objectquery.testobjects import Dummy
from ZODB import MappingStorage, DB
from gocept.objectquery.pathexpressions import RPEQueryParser
from gocept.objectquery.collection import ObjectCollection
from gocept.objectquery.processor import QueryProcessor

def create_btree(n, c=1):
    if n < 0:
        raise ValueError("n should be >= 1")
    if n == 0:
        temp = Dummy(); temp.id = c
        return temp
    if n > 0:
        left = create_btree(n-1, c)
        right = create_btree(n-1, left.id+1)
    temp = Dummy([left, right])
    temp.id = right.id + 1
    return temp

heights = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

storage = MappingStorage.MappingStorage()
db = DB(storage)
conn = db.open()
dbroot = conn.root()
parser = RPEQueryParser()

for n in heights:
    dbroot.clear()
    oq = ObjectCollection(conn)
    query = QueryProcessor(parser, oq)
    objects = pow(2, n+1)-1

    dbroot[0] = create_btree(n)
    transaction.commit()
    t0 = time.time()
    oq.add(dbroot[0]._p_oid)
```

```
    transaction.commit()
    t1 = time.time()
    print "%2i %8i | %7.2fs" %(n, objects, (t1-t0)),
    queries = ['Dummy[@id="%i"]' %(objects/2), '/Dummy',
               '/Dummy/Dummy/Dummy',
               'Dummy[@id="%i"]/_*/Dummy[@id="1"]' %objects,
               'Dummy[@id<"1000"]']
    for q in queries:
        t0 = time.time()
        r = query(q)
        t1 = time.time()
        print "%6.2fs" %(t1-t0),
    print ""
```

Listing A.1: Python-Script zum Testen der Performance der ObjectQuery