Vereinfachung der Entwicklung von Geschäftsanwendungen durch Konsolidierung von Programmier-konzepten und -technologien

Lenoid Berov, Johannes Henning, Toni Mattis,
Patrick Rein, Robin Schreiber, Eric Seckler,
Bastian Steinert, Robert Hirschfeld

Technische Berichte Nr. 71

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 71

Lenoid Berov | Johannes Henning | Toni Mattis | Patrick Rein |
Robin Schreiber | Eric Seckler | Bastian Steinert | Robert Hirschfeld

Vereinfachung der Entwicklung von Geschäftsanwendungen durch Konsolidierung von Programmierkonzepten und -technologien

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6404/
URN urn:nbn:de:kobv:517-opus-64045
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-64045

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-231-5

Zusammenfassung. Die Komplexität heutiger Geschäftsabläufe und die Menge der zu verwaltenden Daten stellen hohe Anforderungen an die Entwicklung und Wartung von Geschäftsanwendungen. Ihr Umfang entsteht unter anderem aus der Vielzahl von Modellentitäten und zugehörigen Nutzeroberflächen zur Bearbeitung und Analyse der Daten. Dieser Bericht präsentiert neuartige Konzepte und deren Umsetzung zur Vereinfachung der Entwicklung solcher umfangreichen Geschäftsanwendungen.
Erstens: Wir schlagen vor, die Datenbank und die Laufzeitumgebung einer dynamischen objektorientierten Programmiersprache zu vereinen. Hierzu organisieren wir die Speicherstruktur von Objekten auf die Weise einer spaltenorientierten Hauptspeicherdatenbank und integrieren darauf aufbauend Transaktionen sowie eine deklarative Anfragesprache nahtlos in die selbe Laufzeitumgebung. Somit können transaktionale und analytische Anfragen in der selben objektorientierten Hochsprache implementiert werden, und dennoch nah an den Daten ausgeführt werden.
Zweitens: Wir beschreiben Programmiersprachkonstrukte, welche es erlauben, Nutzeroberflächen sowie Nutzerinteraktionen generisch und unabhängig von konkreten Modellentitäten zu beschreiben. Um diese abstrakte Beschreibung nutzen zu können, reichert man die Domänenmodelle um vormals implizite Informationen an. Neue Modelle müssen nur um einige Informationen erweitert werden um bereits vorhandene Nutzeroberflächen und -interaktionen auch für sie verwenden zu können. Anpassungen, die nur für ein Modell gelten sollen, können unabhängig vom Standardverhalten, inkrementell, definiert werden.
Drittens: Wir ermöglichen mit einem weiteren Programmiersprachkonstrukt die zusammenhängende Beschreibung von Abläufen der Anwendung, wie z.B. Bestellprozesse. Unser Programmierkonzept kapselt Nutzerinteraktionen in synchrone Funktionsaufrufe und macht somit Prozesse als zusammenhängende Folge von Berechnungen und Interaktionen darstellbar.
Viertens: Wir demonstrieren ein Konzept, wie Endnutzer komplexe analytische Anfragen intuitiver formulieren können. Es basiert auf der Idee, dass Endnutzer Anfragen als Konfiguration eines Diagramms sehen. Entsprechend beschreibt ein Nutzer eine Anfrage, indem er beschreibt, was sein Diagramm darstellen soll. Nach diesem Konzept beschriebene Diagramme enthalten ausreichend Informationen, um daraus eine Anfrage generieren zu können. Hinsichtlich der Ausführungsdauer sind die generierten Anfragen äquivalent zu Anfragen, die mit konventionellen Anfragesprachen formuliert sind. Das Anfragemodell setzen wir in einem Prototypen um, der auf den zuvor eingeführten Konzepten aufsetzt.

Schlüsselwörter: Geschäftsanwendungen, Programmierkonzepte, Datenbank, Hauptspeicherdatenbank, Python, Spaltenlayout, Nebenläufigkeit, Transaktionen, Anfragesprache, Embedded Query Language, SQL, LINQ, domänenspezifisch, modellintensiv, Ansichten, Abläufe, Prozesse, Statistiken, Anfragen.

Inhaltsverzeichnis

Teil I

Vorwort

I.1 Einleitung

Die Komplexität von Geschäftsabläufen ist heutzutage nicht mehr ohne den Einsatz von Geschäftsanwendungen beherrschbar. Geschäftsanwendungen bilden die Gesamtheit betrieblicher Prozesse ab. Sie unterstützen das Unternehmen unter anderem bei der Auftrags- und Kundenverwaltung, der Finanzbuchhaltung und der Verwaltung des Personalwesens. Aufgrund des Umfangs der zu verwaltenden Daten kann man Geschäftsanwendungen als modell- und datenintensive Anwendungen bezeichnen. Obwohl die Komplexität der einzelnen Aufgaben solcher Anwendungen oft nur gering ist, haben wir festgestellt, dass die Entwicklung von vollständigen modell- und datenintensiven Anwendungen mit den gängigen Methoden aufwendig ist. Als Ursachen dafür haben wir zwei Arten von Problemen identifiziert:

1. Probleme, die durch die Trennung von Anwendungen in Schichten mit unterschiedlichen Abstraktionen entstehen.
2. Probleme, die durch die redundante und implizite Beschreibung von Funktionalität innerhalb der Anwendungsschichten entstehen.

Abschnitt I.2 führt die erste Art von Problemen am Beispiel der Trennung von Anwendung und Datenbank in unterschiedliche Laufzeitumgebungen ein. Kapitel II bis IV beschreiben dann eine sprachintegrierte Hauptspeicherdatenbank, welche die Trennung von Anwendung und Datenbank aufhebt und die beschriebenen Probleme auf diese Weise löst.

Die zweite Art von Problemen tritt vor allem bei der Beschreibung von Ansichten und Abläufen in modell- und datenintensiven Anwendungen auf. In Abschnitt I.3 führen wir in die Probleme ein, die bei der üblichen Art, diese Beschreibungen zu formulieren, auftreten. In Kapitel V und VI beschreiben wir alternative Lösungen für die Beschreibung dieser Ansichten und Abläufe, die diesen Problemen nicht unterliegen.

Die beschriebenen Lösungen sollen die Entwicklung von Geschäftsanwendungen effizienter gestalten. Kapitel VII schlägt ein alternatives Anfragemodell zur Beschreibung analytischer Anfragen durch Endnutzer vor und evaluiert die in den anderen Arbeiten vorgestellten Programmierkonzepte, indem sie die Implementierung des Anfragemodells vorstellt, welche auf unserer prototypenhaften Umsetzung der Konzepte aufbaut.

I.2 Laufzeitumgebungen für Datenbanken und Anwendungen

Die Idee der Trennung von Anwendung und Datenbank ist heutzutage weit verbreitet. Die Arbeit innerhalb der jeweiligen Technologie (Anwendung bzw. Datenbank) ist optimiert – an ihren Schnittstellen gestaltet sie sich jedoch problematisch. So ist zum Beispiel das Debugging über Technologiegrenzen hinweg nicht immer möglich, und die Optimierung von Algorithmen kompliziert, da unklar ist, ob sie auf der Datenbank- oder Anwendungsebene zu geschehen hat. Dies hat außerdem zur Folge, dass die Programmierer mehrere Technologiestacks beherrschen müssen, um an einer Anwendung arbeiten zu können, was die Einarbeitungszeit erhöht. Zusätzlich muss der Aufbau von Modellen sowohl bei der Darstellung in Ansichten als auch der Verarbeitung in objektorientierten Programmiersprachen und der Speicherung in Datenbanken jeweils explizit beschrieben werden, was zu Codeduplikation zwischen Datenbank und Datenmodell der Anwendung führt.

Ein gänzlich anderer, schon seit mehreren Jahrzehnten existierender Ansatz ist es, die Objekte der Anwendung einfach als solche in einer Datenbank zu speichern. Dafür müssen diese im Speicher liegenden Objekte serialisiert und dann als Binärobjekt in der Datenbank abgelegt werden. Hiermit geht man der Diskrepanz zwischen relationalem Modell und Objekten aus dem Weg. Zudem sind Wechsel zwischen verschiedenen Technologien nicht mehr von Nöten: Der Programmierer formuliert alles in einer Sprache. Alle Objekte, mit denen man zur Laufzeit gearbeitet hat, werden in genau dieser Form festgehalten. Man persistiert also gewissermaßen den Zustand des Programms.

Bei objektorientierten Datenbanken ergeben sich allerdings zwei große Nachteile: Das Speicherlayout von serialisierten Objekten lässt sich in der Regel nicht komprimieren, was heißt, dass man tendenziell weniger Daten in der gleichen Menge an Speicher ablegen kann. Ein zweiter Nachteil ist, dass man auf den serialisierten Daten keinerlei Anfragen ausführen kann, und die Objekte deshalb einzeln wiederbeleben muss, was zusätzlichen Mehraufwand für das ausführende System darstellt.

Wünschenswert wäre es daher, die Vorteile des Arbeitens mit Objekten direkt in der Datenbank beizubehalten, aber gleichzeitig keinerlei Kompromisse bei der Effizienz der Speicherung und der Performanz der Ausführung einzugehen. Wir schlagen daher eine Lösung vor, die sich zwischen diesen beiden Ansätzen einordnen lässt. In Kapitel II wird der von uns implementierte Prototyp einer sprachintegrierten Hauptspeicherdatenbank vorgestellt. Bei diesem gibt es keinen Unterschied mehr zwischen Datenbank und aktiv verwendeten Objekten, da Lesen und Schreiben von Attributen direkt in ein modernen Hauptspeicherdatenbanken nachempfundenes Speicherschema umgeleitet wird. Da man in einer solchen Umgebung noch immer nebenläufiges Arbeiten ermöglichen

muss, wird in Kapitel III der Entwurf und die Implementierung eines optimistischen Systems zur Nebenläufigkeitskontrolle für den Prototypen beschrieben. Zuletzt muss es dem Anwender ermöglicht werden, große analytische Anfragen auf den Objekten in der Datenbank auszuführen. Diese müssen in kürzester Zeit ausgeführt werden, da sie oft weitreichende Geschäftsentscheidungen beeinflussen. Dafür haben wir eine dedizierte deklarative Anfragesprache entwickelt, die in Kapitel IV vorgestellt wird.

I.3 Programmierkonzepte für Ansichten und Abläufe

Ansichten und Abläufe in modellintensiven Anwendungen sind aufwändig zu programmieren, zu erweitern und zu warten. Das grundlegende Problem bei der Beschreibung von Anwendungen ist, dass die Information, welche Funktionalität die Anwendung bietet, implizit in der technischen Umsetzung formuliert ist. Solche technischen Beschreibungen sind zum Beispiel HTML-Elemente oder Methoden in einem Controller.

Durch die technischen Umsetzungen lassen sich selbst Standardfunktionalitäten teilweise nicht mehr wiederverwenden, obwohl sie mehrfach ähnlich umgesetzt werden. Außerdem ist das Verhalten einer Anwendung teilweise schwer zu verstehen, da eine einzige Funktionalität unter Umständen über mehrere technische Einheiten verteilt beschrieben ist.

Um die Verständlichkeit, Wartbarkeit und Erweiterbarkeit von modellintensiven Anwendungen zu verbessern, müssen diese Probleme gelöst werden. Da sie aus der impliziten Beschreibung der Funktionalität entspringen, ist unser allgemeiner Ansatz, Programmierkonzepte zu schaffen, mit denen diese vormals implizit erfassten Informationen explizit formuliert werden können. Wir haben solche Programmierkonzepte zur Beschreibung von Ansichten und zur Beschreibung von Abläufen in einem Prototypen umgesetzt.

Kapitel V beschreibt das Problem der impliziten Informationen über Ansichten genauer. Die Beschreibung, welche Funktionalität eine Ansicht bietet, ist implizit, da sie in technischen Abstraktionen erfasst ist. Diese technischen Abstraktionen haben zur Folge, dass die Funktionalität jeder Ansicht von Grund auf neu beschrieben wird, auch wenn sich die Ansichten eigentlich ähneln. Dadurch ist redundant gespeichert, welche Ansichten welche Funktionalität bieten sollen. Domänenspezifische Programmierkonstrukte für Ansichten modellintensiver Anwendungen können diese Information explizit erfassen. Die vorgeschlagenen Konzepte fußen auf einem Datenmodell, das um Darstellungsinformationen erweitert ist. Darauf baut eine modellunabhängige Beschreibung von Attributdarstellung, Strukturen und Interaktionen auf. Die bereitgestellte Funktionalität der Ansichten kann dann implementierungsunabhängig be-

schrieben werden. Zur weiteren Optimierung nutzt die Umsetzung aus, dass modellintensive Anwendungen einen hohen Anteil an Standardfunktionalität bieten. Diese kann einmal modellunabhängig gefasst werden und gilt anschließend für alle Modelle. Nötige Abweichungen müssen nur noch inkrementell pro Modell formuliert werden.

Die Probleme der impliziten Formulierung von Abläufen der Anwendung und die passenden Programmierkonstrukte sind in Kapitel VI dargestellt. Die essentielle Information eines Ablaufs ist die darin beschriebene Geschäftslogik. Diese Bedeutung des Ablaufs entsteht durch den Zusammenhang zwischen den einzelnen Schritten, die entweder Nutzerinteraktionen oder Veränderungen des Datenmodells sein können. Diese Logik geht verloren, da die Schritte auf einzelne Programmierelemente, wie Methoden oder Ansichten, verteilt sind und damit der Zusammenhang nicht mehr auf einen Blick erkennbar ist. Diese verteilte Beschreibung sorgt auch dafür, dass man einem einzelnen Programmierelement nicht ansieht, ob es Teil eines Ablaufs ist und wenn ja, an welcher Stelle es sich einfügt. Die Stellen, an denen Nutzerinteraktionen passieren, sind zudem nicht explizit. Diese Probleme werden durch ein Programmierkonstrukt gelöst, mit dem Entwickler die Geschäftsabläufe zusammenhängend beschreiben können. Die konkrete Umsetzung ermöglicht es, den Ablauf in einer Prozedur zu beschreiben, in der Nutzerinteraktionen Aufrufe spezieller Funktionen sind, die die Eingabe des Nutzers zurück geben. Diese Funktionen können genutzt werden, als ob sie synchrone Aufrufe wären, obwohl sie tatsächlich asynchron ausgeführt werden. Da alle Abläufe, von CRUD-Operationen bis zu umfangreichen Geschäftsprozessen, in Prozeduren beschrieben sind, lassen sie sich wiederverwenden.

Zusammen erlauben diese zwei Ansätze, große Teile der Funktionalität von Standardanwendungen explizit zu formulieren und sie somit modularisiert zu fassen. Die resultierende Wiederverwendbarkeit führt zu besserer Wartbarkeit und Erweiterbarkeit der Anwendung, sowie zu besserer Verständlichkeit des Quelltextes.

Teil II

Sprachintegrierte Hauptspeicherdatenbank für Python

II.1 Einleitung

Die Entwicklung von Datenbanken und Laufzeitumgebungen für Anwendungen verlief über Jahrzehnte nahezu vollständig getrennt. Einerseits strebte die Evolution von Datenbank-Managementsystemen (DBMS) letztendlich einem relationalen Datenmodell mit einer deklarativen Anfragesprache (heute meist SQL-Dialekte) entgegen. Diese hatte den Vorteil, in Anbetracht der relativ langsamen Festplattenspeicher die Optimierung der Datenzugriffe dem DBMS selbst zu überlassen, indem der Entwickler lediglich formuliert, was er haben möchte, und nicht wie.

Andererseits sind moderne Geschäftsanwendungen in objektorientierten (OOP) Sprachen geschrieben, allen voran Java, C++, C#, PHP und Python. Objekte definieren sich durch ihre Identität, einen von der Identität unabhängigen Zustand und ihr Verhalten. Sie kapseln die genaue Implementierung ihres Zustands und kommunizieren über Nachrichten.

Wenn es zur Speicherung von Objekten in einer Datenbank kommt, treten eine Reihe konzeptueller Unverträglichkeiten auf, welche unter dem Begriff *Object-Relational Impendance Mismatch* gruppiert sind: Beispielsweise besitzen Tupel keine Identität – sie definieren sich allein über ihre Werte. Ebensowenig lässt sich Kapselung, Verhalten und Nachrichtenaustausch direkt auf ein relationales Modell abbilden. Um dennoch objektorientierte Schnittstellen zu Datenbanken bereitzustellen, wurden Objektrelationale Mapper (ORM) entwickelt, die als Zwischenschicht objektorientierte Konzepte in einer relationalen Datenbank simulieren. Durch weitestgehend transparente Übersetzung dämpfen sie die Effekte des Impendance Mismatch auf den Programmierer. Dennoch bedürfen sie meist einer Konfiguration in Form eines Schemas und eines auf die Datenbank abgestimmten Konnektors und erfordern gelegentlich Wissen über ihre Implementierung, beispielsweise in welcher Reihenfolge Objekte eines Objektbaums abgelegt werden müssen, erzeugen mehr Raum für Fehler und Einbußen in der Performanz.

Als es technisch und wirtschaftlich möglich wurde, eine Unternehmensdatenbank vollständig im Arbeitsspeicher aufzubewahren, setzte die Entwicklung von hauptspeicherresidenten Datenbanken einen neuen Meilenstein. Nun liegen die Daten und die darauf arbeitende Anwendung im gleichen Ökosystem, doch die historische Trennung besteht weiterhin. Das wirft die Frage auf, warum bereits im Arbeitsspeicher vorhandene Daten durch ein ORM kopiert oder durch Sprachen manipuliert werden müssen, die von der Anwendungssprache verschieden sind; und welche Sprachkonzepte und Implementierungsstrategien die Performanz- und Konsistenz-Vorteile eines DBMS auch ohne Trennung erhalten können.

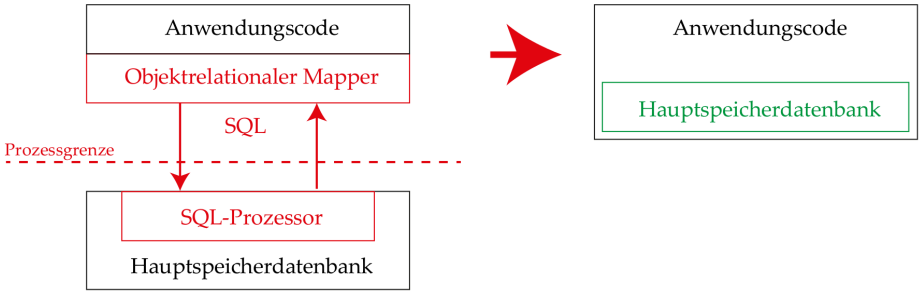


Abb. 1: *Ziel des Prototypen ist die Abschaffung von Schichten zwischen Anwendung und Daten.*

Gegenstand dieser Arbeit ist der Entwurf und die Implementierung einer Hauptspeicherdatenbank innerhalb der Skriptsprache Python, indem wir die grundlegenden Konzepte einer Hauptspeicherdatenbank in die Python-Laufzeitumgebung integrieren. Diese sollen objektorientierte Programmierung nahtlos unterstützen und gleichzeitig auf die bisherige Schichtenarchitektur verzichten, wodurch der Umgang mit Daten direkter, schneller und durchweg in einer einzigen Sprache geschehen sollen.

Im folgenden Abschnitt II.2 werden zunächst die Nachteile objektorientierter Sprachen zur Datenspeicherung untersucht und wodurch hauptspeicherresidente Datenbanken diese Schwierigkeiten überwinden. Abschnitt II.3 formuliert konkrete Anforderungen an eine Integration einer Datenbank in die Programmiersprache Python, deren konzeptuelle und programmatische Implementierung in Abschnitt II.4 detailiert beschrieben wird. In Abschnitt II.5 wird die Plausibilität des gewählten Ansatzes empirisch nachgewiesen sowie Schwachstellen dokumentiert. Vorschläge zur Behebung dieser und weiterer Unzulänglichkeiten sowie Ideen für eine Weiterführung sind in Abschnitt II.6 aufgeführt. In Abschnitt II.7 erfolgt ein Vergleich und eine Abgrenzung zu bestehenden, ähnlichen Projekten.

II.2 Vergleich der Speicherorganisation in Datenbanken und OOP-Laufzeitumgebungen

Dieser Abschnitt soll einen kurzen Abriss darüber geben, warum sich klassische objektorientierte Laufzeitumgebungen für sich genommen nicht als Datenbank eignen und inwiefern moderne Hauptspeicherdatenbanken einer bloßen Ansammlung von Objekten überlegen sind. Dazu wird im Folgende die Speicherorganisation von Python betrachtet und im Anschluss die einer Hauptspeicherdatenbank. Zum Schluss wird herausgestellt, was die Kernunterschiede letztendlich für Effekte auf die Datenhaltung und -verarbeitung haben.

II.2.1 Die Speicherorganisation von Python

Die Speicherorganisation objektorientierter Sprachen bildet zur Laufzeit einen gerichteten Graphen: Ein Objekt selbst stellt einen Knoten dar und beinhaltet die primitiven Attribute seines Zustands sowie Zeiger auf die Speicherstellen referenzierter Objekte. Objektidentität ist i.d.R. über die Speicherstelle des Objekts festgelegt.

Während statisch getypten Sprachen das bloße Hintereinanderschreiben der Attribute und Zeiger genügt und durch die Klasse fest definiert ist, welche Daten wo liegen, verwenden dynamische, insbesondere auf *Duck-Typing* ausgelegte Sprachen z.T. komplexere Strukturen um Attribute zur Laufzeit hinzuzufügen oder zu entfernen. Ein Beispiel hierfür ist die Speicherorganisation von Python in Abbildung 2.

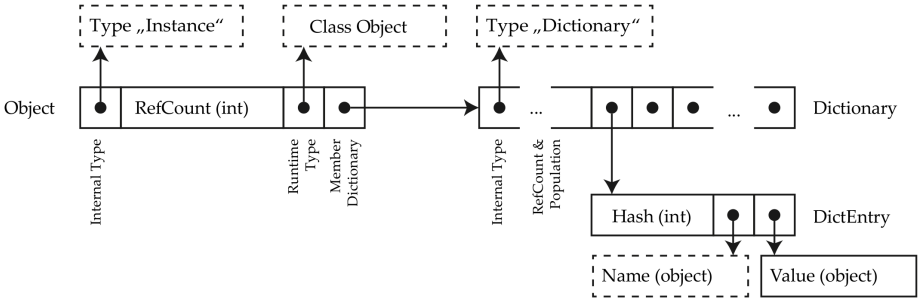


Abb. 2: *Vereinfachte Speicherorganisation eines Objekts in Python 2.7. Gestrichelte Strukturen werden von mehreren Objekten gemeinsam genutzt.*

Neben einem konstanten internen Typen, welcher die nativen C-Methoden bereitstellt, wird der für die Garbage Collection wichtige Referenzzähler, ein Zeiger auf die eigentliche Klasse und ein Zeiger auf ein Attribut-Dictionary gehalten. Dieses ist als Hashmap realisiert.

Möchte man große Ansammlungen von Objekten flexibel verwalten, bietet Python Listen, Dictionaries und Mengen als Datentypen an. Während Mengen ähnlich wie Dictionaries funktionieren und lediglich keinen Wert hinter einem Schlüssel ablegen (s. Dictionary in in Abbildung 2), arbeiten Listen nicht Hash- sondern Indexbasiert in Arrays, wie Abbildung 3 illustriert. Neben dem eigentlichen Array von Zeigern wird die Länge des allozierten Zeiger-Arrays sowie der tatsächlich verwendete Füllstand gespeichert.

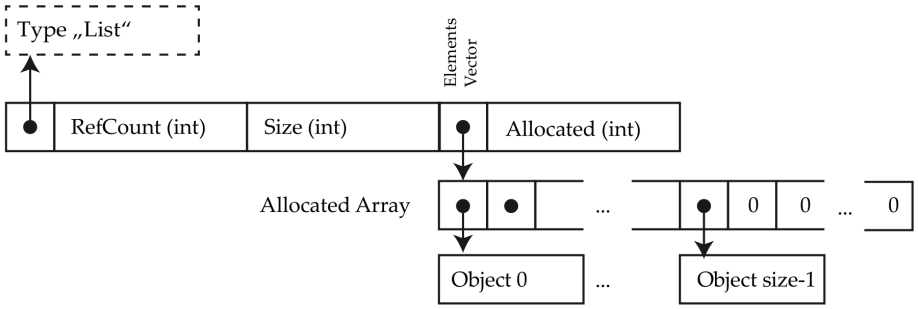


Abb. 3: *Speicherorganisation einer Liste von Objekten in Python 2.7.*

II.2.2 Relationale Hauptspeicher-Datenbanken

Moderne hauptspeicherresidente Datenbanken unterliegen anderen Bedingungen als festplattenbasierte Speicher. Zum einen sind die Kosten pro Gigabyte erheblich höher, was zum Einsatz platzssparender Datenstrukturen motiviert, zum anderen sollen die Geschwindigkeitsvorteile optimal genutzt werden, d.h. anstatt auf möglichst sequentielle Festplattenzugriffe zu achten, sollte nun auf eine optimale Ausnutzung der CPU-Caches Wert gelegt werden.

Moderne Hauptspeicherdatenbanken nutzen daher anstelle des alten Tupel- oder Zeilenbasierten Layouts (Abbildung 4) eine auf Spalten basierende Speicherung (Abbildung 5). Diese hat die Eigenschaft, dass Daten gleichen Typs (und teilweise auch gleiche Daten) hintereinander im Speicher liegen. Wie bei Datenbanken üblich werden für Tupel meist eindeutige Kennziffern als *Primärschlüssel* oder *ID* festgelegt und die Beziehung von Daten untereinander durch Angabe der ID eines referenzierten Tupels, dem sog. *Fremdschlüssel*, hergestellt.

108	Jean	Dupont	1986
109	Piet	Pompies	1990
110	Jon	Doe	1974

Abb. 4: *Speicherorganisation in Zeilen. Was auf Festplatten für schnelles Auslesen zusammengehörender Daten beiträgt, bringt im RAM keine Vorteile mehr.*

108	Jean	Dupont	1986
109	Piet	Pompies	1990
110	Jon	Doe	1974

Abb. 5: *Speicherorganisation in Spalten. Homogene Daten füllen zusammenhängende Speicherbereiche, was Cache-Prefetching und Komprimierbarkeit begünstigt.*

II.2.3 Vorteile des Spalten-Layouts gegenüber Objekten

Die oben geschilderte Speicherorganisation von Python-Objekten ist auf maximale Flexibilität ausgelegt, d.h. beliebig benannte Attribute können zur Laufzeit angelegt, verändert, gelesen und wieder entfernt werden. Datenbanken tauschen diese Flexibilität aufgrund der Gleichförmigkeit vieler Daten gegen ein starres, aber anderweitig sehr effizientes Speicherlayout.

Bei Hauptspeicherdatenbanken sind Werte des gleichen Attributs in einer Spalte hintereinander im Speicher angeordnet, wodurch diese *komprimiert* werden können und den teuren Hauptspeicher besser nutzen als Zeiger-gestützte Listen von Objekten. Einen viel schwerwiegenderen Vorteil haben die Spalten jedoch bei *Aggregationen* über Daten. Während bei normalen Python-Objekten für das Auslesen eines einzigen Attributs mehreren Zeigern gefolgt werden muss, liegen in einer Spalte bereits die Attribute aller Instanzen nebeneinander. Das errechnen einer Summe o.ä. über diese Attribute geht damit konsequenterweise sehr viel schneller. Darüberhinaus greifen hierbei auch Hardware-Aspekte, so kann die CPU bei der Verarbeitung der Daten im Hintergrund bereits die nächsten Speicherstellen aus dem RAM in den CPU-Cache laden (*Prefetching*) und die Aggregation letztendlich auf dem Cache noch schneller arbeiten.

Es soll also im Folgenden versucht werden, eine solche Speicherorganisation für Python-Objekte nutzbar zu machen, um einerseits möglichst viel Flexibilität beizubehalten und andererseits für eine Teilmenge der Programmdaten – nämlich jene, die normalerweise in Datenbanken abgelegt werden – oben genannte Vorteile zu erreichen.

II.3 Anforderungen an eine sprachintegrierte Datenbank

An dieser Stelle werden zunächst eine Reihe an Anforderungen aufgestellt, denen die von uns entwickelte, sprachintegrierte Datenbank genügen soll.

II.3.1 Performanz und Speichereffizienz

Objekte, die in der integrierten Datenbank residieren, sollen von Geschwindigkeit und kompakter Datendarstellung moderner Hauptspeicherdatenbanktechnologien profitieren können. Gegenstand dieser Arbeit ist zunächst die Implementierung einer den Hauptspeicherdatenbanken nachempfundenen Speicherorganisation inkl. Kompression. Dies soll auf eine Weise geschehen,

die in aufbauenden Arbeiten die Implementierung einer deklarativen, aber objektorientierte Anfragesprache, wie in Kapitel IV beschrieben, ermöglicht. Darüberhinaus soll die Umsetzung von Nebenläufigkeit und transaktionalem Verhalten in Kapitel III unterstützt werden.

II.3.2 Identität und Referenz

Wichtiger Bestandteil von Objektorientierung ist das Konzept der Identität. Während Objekte zur Laufzeit ihren Zustand ändern können, ändern sie ihre Identität nie, d.h. sind zu jedem Zeitpunkt immer eindeutig benennbar. Diese „Benennung“ innerhalb eines Programms erfolgt durch Referenzen. Referenzen sind Daten, welche die Identität eines Objekts, jedoch nicht seinen Zustand beschreiben und kommunizieren können.

In objektorientierten Sprachen ist die Identität eines Objekts meist durch seine Speicheradresse gegeben, d.h. Referenzen beinhalten lediglich die Speicheradresse in Form eines Zeigers. In relationalen Datenbanken ist Identität durch Wertgleichheit gegeben, d.h. dort muss zustandsunabhängige Objekt-Identität simuliert werden, was i.d.R. durch einführen einer relationsweit eindeutigen ID geschieht. Referenzen werden vielmehr zu einer Angabe, die Tabelle und ID benennt.

Anforderungen Unsere Impementierung soll sich nahtlos in das objektorientierte Python eingliedern. Referenzen auf normale Objekte sollen sich in ihrer Handhabung nicht von Referenzen auf Datenbank-Objekte unterscheiden, obwohl sie anders implementiert sein werden. Es ergeben sich folgende Design-Ziele:

- DB-Referenzen können beliebigen Variablen zugewiesen werden, auch Funktionsargumenten.
- DB-Referenzen können auf Referenzgleichheit (Identität) geprüft werden

II.3.3 Klassen, Nachrichten und Methoden

Objekte kommunizieren in der Regel über Nachrichten. Beim Empfänger löst das Eintreffen einer Nachricht eine bestimmte Methodenausführung aus. Prinzipiell kann (auch in Python) jedes Objekt seine eigene Methode definieren, in den gängigen OOP-Sprachen wird gleichartiges Verhalten jedoch in Klassen zusammengefasst, sodass die Klasse eines Objekts bestimmt, wie die Nachricht zu behandeln ist. Da sich unsere Datenbank-Objekte auf die gleiche Weise wie normale Laufzeitobjekte verhalten sollen, ergeben sich weitere Anforderungen:

- DB-Objekte sind Instanzen spezieller DB-Klassen, welche genau wie normale Klassen Methoden besitzen können.
- DB-Klassen sollen bei der Definition explizit gekennzeichnet werden, um sie von Nicht-DB-Klassen zu unterscheiden.
- Bezüglich Nachrichtenaustausch und -interpretation sollen sich die Instanzen von DB-Klassen nicht von Instanzen normaler Klassen unterscheiden.

II.3.4 Attribute und Assoziationen

Während in statisch getypten Sprachen die Klasse bereits Attribute und Assoziationen definiert, d.h. den vollständigen Rahmen für den Objektzustand vorgibt, werden Attribute in Python zur Laufzeit im Konstruktor `__init__` oder einer beliebigen anderen Methode erstellt und können beliebigen, auch zwischenzeitlich variierenden Typen/Klassen angehören. Jede Instanz nach ihrem Konstruktoraufruf oder gar weiteren Methodenaufrufen auf ihre angelegten Attribute inspizieren zu müssen um die Notwendigkeit neuer Spalten festzustellen, kann die Erzeugung von Instanzen stark verlangsamen. Daher ist es ratsam, Datenbank-Attribute statisch in der Klasse zu verankern. An die Programmierung mit DB-Klassen ergehen daher neue Anforderungen:

- DB-Klassen müssen Attribute, die in der Datenbank gespeichert werden sollen, explizit mit ihrem Typen definieren. Alle anderen, zur Laufzeit dynamisch erzeugten Attribute sollen sich wie auf normalen Python-Klassen verhalten und werden demnach auch nicht optimiert gespeichert.
- Assoziationen zu DB-Klassen müssen explizit gemacht werden mitsamt deren Multiplizität, d.h. ob es sich um eine einzige Referenz handelt oder eine Menge von Referenzen.

II.4 Implementierung der Spracherweiterung

II.4.1 API

Das folgende Quelltext-Beispiel 1 soll darstellen, wie o.g. Anforderungen in die Sprache integriert werden sollen. Dabei wird das in Abbildung 6 dargestellte Modell mit Datenbank-Klassen implementiert.

```
1 @column_layout      # Kennzeichnung der DB-Klassen
2 class Product(object):
3     name = str      # Statische Typisierung
4     price = int
```

```
5
6       # normaler Konstruktor
7       def __init__(self, name, price):
8           self.name = name
9           self.price = price
10
11  @column_layout
12  class Customer(object):
13      name = str
14      address = str
15
16  @column_layout
17  class OrderItem(object):
18      count = int
19      # Referenz auf genau ein Objekt
20      product = One(Product)
21
22      # Methoden funktionieren wie an normalen Klassen
23      def price(self):
24          return self.product.price * self.count
25
26  @column_layout
27  class Order(object):
28      items = Many(OrderItem) # Menge von Referenzen
29      customer = One(Customer)
30
31      year  = int
32      month = int
33      day   = int
34
35      def total(self):
36          sum = 0
37
38          # Mengen sind lesbar
39          for item in self.items:
40              sum += item.price()
41          return sum
42
43      def add_item(product, count = 1):
44
45          # Einfuegen in die DB geschieht durch instantiierung
46          item = OrderItem()
47          item.product = product
48          item.count = count
49
50          # Mengen sind modifizierbar
51          self.items_add(item)
```

Quelltext 1: *Beispielquelltext in dem die Anforderungen durch neue Sprachelemente realisiert werden.*

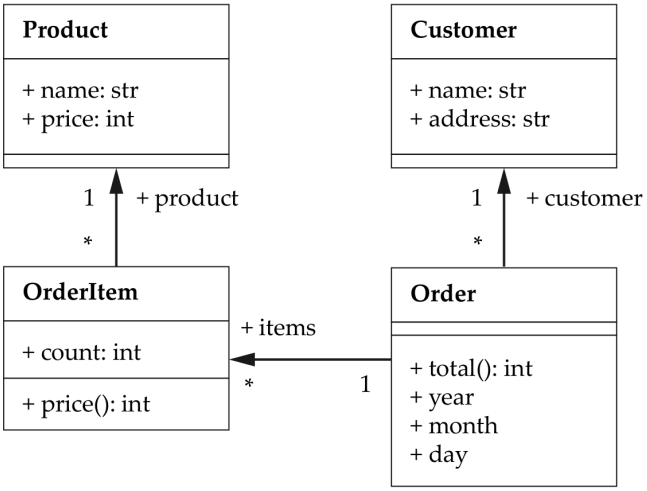


Abb. 6: *Ein Beispielmodell, welches in der Datenbank realisiert werden soll.*

Die Kennzeichnung der DB-Klassen soll durch einen Klassendekorator `column_layout` geschehen. Dieser ist im Wesentlichen eine Methode, welche zum Definitionszeitpunkt der Klasse diese als Argument erhält (s. PEP 3129, [31]), die in folgenden Abschnitten vorgestellten Klassenvariablen interpretiert und zu Gettern und Settern für ein neues Speicherlayout umwandelt. Es muss explizit von `object` geerbt werden, damit ein Klassendekorator Getter und Setter dynamisch überschreiben kann (s. *New-Style Classes*, [24])

Attribute Attribute werden durch Klassenvariablen deklariert, deren Name der Attributname ist und deren Wert dem Typ entspricht, den das Attribut annehmen kann. Unser Prototyp unterstützt die Python-Typen `int`, `bool`, `float` und `str`, wobei die speziellen Typen `One`, `Many` und `ManyToMany` Referenzen auf andere DB-Klassen angeben. Der Platzhalter `Self` steht immer für die Klasse, in der er verwendet wird – dies ist nötig, da Python den Namen einer Klasse zum Zeitpunkt der Deklaration ihrer Klassenvariablen noch nicht kennt.

Assoziationen zu einer Instanz

```
referenzname = One(referenzierte_klasse)
```

Bezeichnet eine Referenz auf genau ein Objekt der angegebenen Klasse. Dieses kann durch `instanz.referenzname` gelesen und `instanz.referenzname = ziel` geschrieben werden. Die Multiplizität der deklarierenden Klasse ist beliebig, die der referenzierten Klasse eins.

Exklusive Mengenassoziation
`referenzname = Many(referenzierte_klasse)`

Bezeichnet eine Menge von Referenzen auf Objekte, wobei jedes Ziel-Objekt maximal einmal (exklusiv) referenziert werden kann. D.h. die von allen Instanzen der Klasse referenzierten Mengen sind paarweise disjunkt. Dieses Verhalten kann gewünscht sein, beispielsweise können Kunden mehrere Bestellungen aufgeben, jede Bestellung aber nur zu einem Kunden gehören. `instanz.referenzname` gibt die Menge der Referenzen zurück. Die durch den Klassendekorator automatisch generierten Methoden `referenzname_add` und `referenzname_remove` fügen Objekte der Menge hinzu oder entfernen sie daraus.

`children = Many(Person, through='parent')` ist eine Möglichkeit, die eindeutige Rückbeziehung auf Seite der referenzierten Klasse `parent` zu nennen. Würde man mittels `a.children_add(b)` ein Objekt `b` zu `a.children` hinzufügen, hat das den gleichen Effekt wie `b.parent = a`. Ohne `through`-Parameter wird ein generierter Name verwendet.

Allgemeinere Mengenassoziation Die Many-Assoziation an sich besitzt die Einschränkung, dass assoziierte Objekte von nur einer Instanz gleichzeitig referenziert werden können. Soll allerdings auch die Rückbeziehung eine Menge darstellen und damit die referenzierte Instanz zu mehreren referenzierenden Instanzen gehören können, kommt die `ManyToMany`-Assoziation zum Einsatz.

Diese legt, mittels `through`-Parameter auf referenzierter Seite benannt, beidseitig eine Schnittstelle zum Ermitteln der Menge mittels `instanz.referenzname` an sowie die Methoden `instanz.referenzname_add` und `instanz.referenzname_remove` zum Hinzufügen und Entfernen einer Referenz.

II.4.2 Änderung der Speicherorganisation von Python-Objekten

Die grundlegende Idee der Umsetzung besteht nun darin, den Zustand eines Objekts modernen Hauptspeicherdatenbanken nachempfunden in Spalten abzulegen, um die Vorteile einer Hauptspeicherdatenbank für Objekte ohne Indirektion über SQL-Anfragen nutzbar zu machen.

Konkret erhält jedes Attribut einer Klasse eine eigene Spalte. Die Klasse kennt alle ihre Attribut-Spalten und nimmt damit aus Datenbanksicht die Rolle einer Tabelle ein. Instanzen einer Klasse verfügen über eine ID, mithilfe der sie die zu ihnen gehörenden Attributwerte in der dazugehörigen Attribut-Spalte auffinden können, s. dazu Abbildung 7.

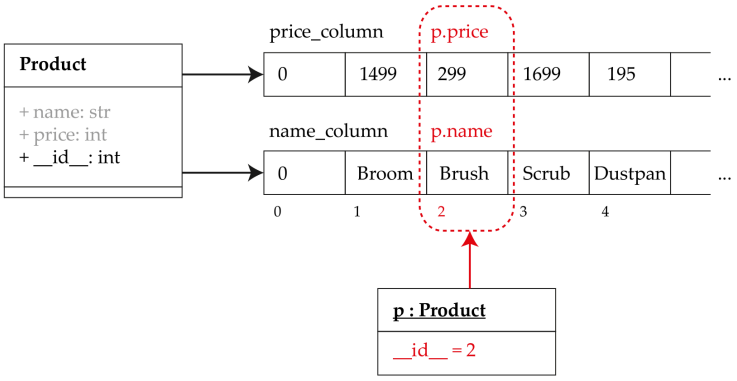


Abb. 7: *Eine DB-Klasse. Attribute sind in Spalten organisiert, Instanzen lediglich Referenzen, welche Attribut-Zugriffe auf ihren entsprechenden Spalteneintrag umleiten.*

Objektidentität und Primärschlüssel Die Objektidentität stellen wir angelehnt an herkömmliche Datenbanken durch eine eindeutige Kombination aus Primärschlüssel (ID) und Klasse dar.

Im Gegensatz zur zeigerbasierten Implementierung von Referenzen ist die abstrakte ID nicht mit einer festen Speicherstelle verknüpft, was Referenzen unabhängig von der Speicherorganisation macht. Leider benötigen wir eine zeitaufwändige Übersetzung von ID zu Speicherstelle – diese kann dafür auch Faktoren wie Kompression oder verschiedene Versionen eines Objekts berücksichtigen.

In unserer Implementierung sind Instanzen der DB-Klassen strenggenommen Referenzen, welche den Typ der DB-Klasse haben und lediglich die ID als einziges Attribut namens `__id__` speichern. Alle anderen Attribute werden automatisch durch *Getter* und *Setter* repräsentiert, welche die Spalten-Einträge an Position der ID lesen und schreiben.

Instanzerzeugung Wir nehmen, wie in Python üblich, an, dass ein expliziter Aufruf von `Klasse(...)` ein neues Objekt der Klasse anlegen soll, d.h. hier wird eine neue Position in allen Spalten reserviert. Zurückgegeben wird ein Referenzobjekt, dessen ID auf die neu reservierte Position verweist. Erreicht wird dies durch Überschreiben der `__new__`-Fabrikmethode einer Klasse, in der ein Threadsicherer Zähler inkrementiert und als ID verwendet wird.

Im Ergänzung dazu existiert in unserer Implementierung eine weitere Fabrikmethode, welche unter Angabe einer bekannten ID eine Referenz auf das Datenbankobjekt erzeugt. Diese Methode heißt `at`, sodass der Aufruf von `Klasse.at(i)` eine Referenz auf das Objekt mit ID i zurückgibt. Besonders bei der Implementierung von Assoziationen ist diese Methode relevant.

Attribute als Spalten Die Werte primitiver Attribute sind in einer Spalte hintereinander angeordnet. Da sie den gleichen Typen haben, bietet sich die Verwendung einer Array-Struktur als Spalte an, die indexbasierten Zugriff auf ihre Elemente erlaubt. Nun bietet Python die Möglichkeit, mittels sog. *Properties* den Zugriff auf Attribute manuell zu implementieren bzw. zu überschreiben.

```
1  # Methode zur Generierung einer
2  # Attribut-Umleitung auf Spalten
3  def property_for(self, column):
4
5      def getter(self):
6          return column[self.__id__]
7
8      def setter(self, value):
9          column[self.__id__] = value
10
11     return property(getter, setter)
12
13 # Erstellen eines Beispiel-Attributs
14 price_column = Column(int)
15 Product.price = property_for(price_column)
```

Quelltext 2: *Property-Dispatch von primitiven Attributen auf Spalten*

Quelltext 2 zeigt, wie zur Laufzeit neue Properties erstellt werden: Eine Methode `property_for` nimmt ein Spaltenobjekt entgegen und definiert einen *Getter* und *Setter* als innere Methoden. Deren `column`-Variable wird bei Methodenausführung an die übergebene Spalte der äußeren Methode gebunden, d.h. es werden zwei neue Funktionen `getter` und `setter` zur Laufzeit erzeugt, die lediglich den für Instanzmethoden nötigen `self`-Parameter entgegen nehmen. Ermöglicht wird dies durch die Fähigkeit von Python, Funktionen bei ihrer Definition an Variablen des umgebenden Namensraums zu binden und damit sog. *Closures* zu erzeugen.

Insbesondere bietet das `property`-Konzept die Möglichkeit, Getter und Setter hinter einem Attributzugriff zu verbergen. Der Getter wird beim Lesen und der Setter beim Schreiben des Attributs auf einer Instanz gerufen. Zugewiesen wird er der Klasse, wie die letzten beiden Zeilen in Quelltext 2 beispielhaft zeigen.

One-Assoziationen Da die Klasse einer Assoziation im Voraus festgesetzt ist, reicht für die Assoziation die Angabe der ID des referenzierten Objekts. Ähnlich wie bei Fremdschlüsseln in einer Datenbank wird auch für eine Assoziation eine Spalte als Ganzzahl-Array angelegt, beim Lesen wird jedoch nicht die Zahl zurückgegeben, sondern eine Referenz vom Typ der Klasse, welche die gelesene Zahl als ID beinhaltet. Umgekehrt führt das Schreiben einer Referenz

dazu, dass nach einer Typprüfung die in der Referenz gespeicherte ID in die entsprechende Spalte geschrieben wird.

```
1  def association_for(column, target_class):
2
3      def getter(self):
4          return target_class.at(column[self.__id__])
5
6      def setter(self, value):
7          column[self.__id__] = value.__id__
8
9      return property(getter, setter)
```

Quelltext 3: *Vereinfachter Property-Dispatch auf Assoziationen*

Quelltext 3 illustriert, wie die für primitive Attribute in Quelltext 2 verwendeten Getter und Setter im Falle einer Assoziation abgeändert werden. Während der Getter mittels `at`-Fabrikmethode die Referenz aus dem Spalteneintrag erzeugt, schreibt der Setter die ID des Objekts in die Spalte. Die Typprüfung ist der Einfachheit halber hier nicht zu sehen und kann mittels Prüfung auf `isinstance(value, target_class)` erfolgen.

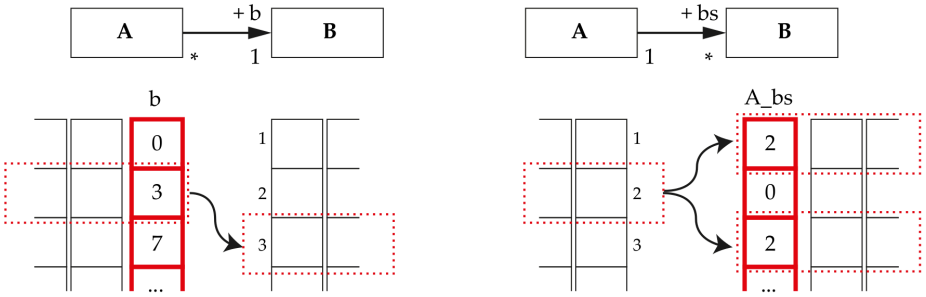


Abb. 8: *One- und Many-Assoziation im Vergleich. Links eine One-Assoziation, in der eine Instanz von Klasse A die dritte Instanz von B via Fremdschlüssel-Spalte referenziert. Rechts eine Many-Assoziation, in der die Instanz mit ID 2 zwei Instanzen der Klasse B referenziert.*

Many-Assoziation Bei Many-Assoziationen existiert ebenfalls eine Fremdschlüssel-Spalte, diese ist jedoch der referenzierten Klasse zugeordnet (vgl. Abbildung 8). Zu der Menge der referenzierten Objekte gehören all jene, welche die ID des referenzierenden Objekts als Fremdschlüssel beinhalten. Beim Hinzufügen und Entfernen eines referenzierten Objekts wird dessen Fremdschlüssel demnach auf die des referenzierenden Objekts respektive auf Null gesetzt.

Da die Many-Assoziation im Wesentlichen eine umgekehrte Betrachtung der One-Assoziation darstellt und auch auf Basis derer implementiert wurde, ist es naheliegend, die Rückbeziehung mittels `through=`-Parameter zu benennen. Standardmäßig hat sie den Namen `a_of_K`, wobei `a` der Assoziationsname und `K` der Name der Klasse ist, an welcher die Assoziation definiert wurde. Im rechten Beispiel aus Abbildung 8 würde die Rückbeziehung ohne explizite Benennung automatisch als `bs_of_A` bereitgestellt.

Auf Seite der referenzierenden Klasse wird also ein Getter erstellt, der die Menge zurückgibt, sowie eine Methode `attributname_add`, welche ein neues Objekt hinzufügt und `attributname_remove`, welches es entfernt. Implementiert wird der Getter im Prototypen noch mittels linearer Suche. Da diese Aufgrund ihres asymptotischen Laufzeitverhaltens von $O(n)$ als Proof-of-Concept zwar genügt, jedoch keine zufriedenstellende Lösung darstellt, wird an dieser Stelle der Einsatz einer Indexstruktur empfohlen. Hinzufügen und Entfernen geschieht durch setzen des Fremdschlüssels der referenzierten Instanz auf die eigene ID oder auf null. Null wird in Assoziationen generell als *Null-Referenz* gehandhabt.

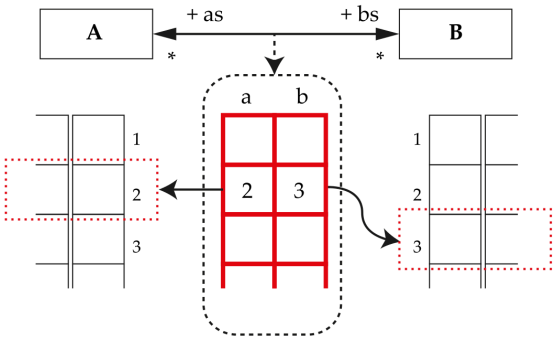


Abb. 9: *Implementierung einer Many-To-Many-Assoziation durch eine Relationstabelle, welche Paare von zusammengehörigen IDs speichert.*

Many-To-Many-Assoziation Bislang konnte entweder auf der referenzierenden oder auf der referenzierten Seite nur ein einziger Fremdschlüssel eingetragen werden. Für beliebige Multiplizitäten auf beiden Seiten wird – angelehnt an das relationale Datenbank-Modell – eine Zwischentabelle erstellt, welche Paare zusammengehöriger IDs speichert (s. Abbildung 9). Beide Seiten erhalten einen Getter, welcher wieder mittels linearer Suche die Menge der referenzierten Instanzen ermittelt, sowie eine `attributname_add`- und `attributname_remove`-Methode. Es ist nicht möglich, den `through=`-Parameter wegzulassen, beiderseits werden die Attribute demnach zwangsläufig so benannt, wie der Nutzer es spezifiziert.

II.4.3 Implementierung der Spalten

Die getypten Spalten, von denen in dem vorausgegangenen Abschnitt die Rede war, können als wachsende Arrays betrachtet werden. Hierbei ist jedoch zu berücksichtigen, dass häufiges Reallozieren bei Erschöpfung des vorallozierten Speichers und etwaiges Kopieren der vorhandenen Array-Elemente quadratisch wachsenden Laufzeitaufwand mit sich bringt. Aus diesem Grund sind Spalten in unserem Prototypen segmentiert.

Segment-Offset-Adressierung Der Zugriff auf eine Spalten-Position erfordert zunächst auf Bit-Ebene die Aufspaltung in Segment- und Offset-Teil, wobei die Bit-Längen eines jeden Teils konfigurierbar sind. Mit dem Segment-Teil wird das Segment aus einer Segment-Tabelle gewählt, während der Offset-Teil die Position innerhalb des Segments angibt. Wird über ein Segment hinaus geschrieben, so kann das nächste Segment alloziert werden. Eine Segment-Tabelle übersetzt den Segment-Teil in einen konkreten Segment-Zeiger.

Für Segmente aus den primitiven Typen `int` (32-Bit-Ganzzahl), `bool` (8-Bit boolscher Wert) und `float` (64-Bit-Gleitkommazahl) kommen Arrays aus dem Python-Modul `array` zum Einsatz. Für Strings kommen vorallozierte Python-Listen zum Einsatz, was nicht optimal, aber für den Protoypen ausreichend ist. Durch nativ unterstütztes String-Interning werden gleiche Strings nicht mehrfach gespeichert, sondern lediglich der entsprechende Zeiger.

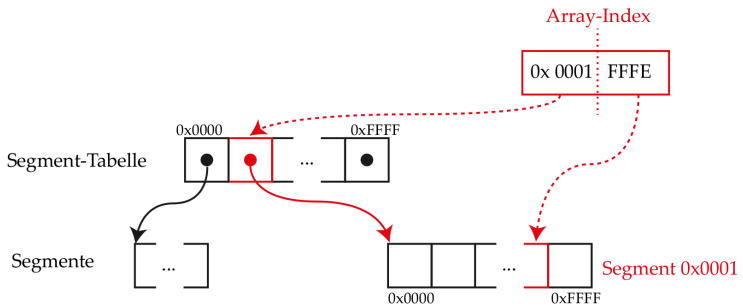


Abb. 10: *Adressierungsschema der Spalten. Der Index wird in einen oberen und einen unteren Teil gespalten, der obere bestimmt das Segment, der untere den Offset innerhalb des Segments.*

In unserem Prototypen wurde von einem 32-Bit-Spaltenindex ausgegangen und dieser in zwei 16-Bit-Hälften gespalten, was die maximale Elementzahl auf $2^{32} = 4.294.967.296$ begrenzt. Denkbar sind Anwendungsfälle, in denen diese Zahl nicht ausreicht und die Segmentgröße und/oder die Segmentanzahl (und damit die Größe der Segment-Tabelle) erhöht werden muss.

Lokale Optimierung Ein weiterer Vorteil der Segmentierung besteht darin, dass jedes Segment für sich genommen je nach Nutzungsfrequenz komprimiert werden kann. So bleiben öfter genutzte Daten unkomprimiert, während seltener verwendete Segmente auf geringen Speicherverbrauch bei gleichzeitig sehr viel langsamerer Zugriffszeit optimiert werden können. Darüberhinaus kann die Kompression auf den Daten lokal arbeiten, d.h. sich an lokale Schwankungen in der Datenverteilung anpassen.

Unser Prototyp nutzt beispielhaft eine sehr einfache Heuristik: Wird ein Segment gefüllt und damit das nächste alloziert, so wird das dem gefüllten Segment vorangegangene Segment komprimiert. Mindestens die letzten 2^{16} Einträge bleiben somit schnell editierbar, ältere Daten hingegen sehr kompakt. Bessere, auf tatsächlichen Laufzeit-Daten basierende Strategien sind denkbar.

II.4.4 Kompression

Da Hauptspeicher gegenüber Festplattenspeicher erheblich teurer ist, sind Hauptspeicherdatenbanken auf eine bessere Ausnutzung ihres Speichers angewiesen, um wirtschaftlich zu bleiben. Daten in Datenbanken enthalten z.T. immense Redundanzen, seien es mehrfach gespeicherte Werte oder eine große Menge leerer Felder, welche durch Kompression eingespart werden kann.

Bit-Kompression Oft sind die Daten innerhalb einer Spalte stark korreliert. Kommen beispielsweise nur wenige, voneinander verschiedene Werte zum Vorschein (z.B. Jahreszahlen zwischen 2000 und 2012), bietet es sich an, für jeden eindeutigen Wert eine kürzestmögliche, unterscheidbare Bitfolge zu wählen und in einem Wörterbuch dem tatsächlichen Wert zuzuordnen. Die Darstellung von n verschiedenen Werten benötigt mit dieser Kompression lediglich $\lceil \log_2 n \rceil$ Bits pro Eintrag, d.h. jede Jahreszahl von 2000 bis einschl. 2012 ist in 4 Bits codierbar, während eine gewöhnliche Ganzzahl in Python-Arrays 32 Bits und damit das 8-fache an Speicher belegen würde.

Die Einfachheit dieser Kompression macht sie darüber hinaus sehr schnell bei lesenden Zugriffen. So kann die Speicheradresse eines Spalteneintrags bei bekannter Bit-Länge eines jeden Eintrags trivial errechnet werden. Im Anschluss ist nur noch eine Rückübersetzung der Zahl in den ursprünglichen Wert durch das Wörterbuch nötig.

Unser Prototyp unterstützt zunächst nur Bit-Kompression, weitere Verfahren werden im Ausblick vorgestellt.

II.4.5 Versionierung und Indexstrukturen

Solange jedes Objekt genau einmal in der Datenbank gespeichert ist, ist seine Position innerhalb der Spalten eindeutig. Dadurch könnte die ID eines Objekts semantisch der absoluten Position seiner Attribut-Werte in den jeweiligen Spalten entsprechen.

Problematisch wird dieser einfache Ansatz, wenn Objekte mehrfach in der Datenbank gespeichert werden. Dies ist bei uns der Fall, da zur Nebenläufigkeits- und Transaktionssteuerung *Multiversion Concurrency Control*(Kapitel III) zum Einsatz kommt oder wir sog. *Time-Travel Queries* über vergangene Datenbank-Zustände ermöglichen wollen (Kapitel IV). Hierbei erzeugt jede Modifikation eines Objekts eine neue Version an aufsteigenden Spalten-Position. Da die Identität dabei aber erhalten bleiben soll, muss die konstante ID auf die sich ändernde, aktuelle Spaltenposition übersetzt werden.

Für solche Schlüssel-Wert-Zuordnungen existieren eine Reihe von Datenstrukturen: In Hauptspeicherdatenbanken werden bevorzugt CSB+ Bäume [25] eingesetzt, welche sich im Wesentlichen wie B+ Bäume verhalten, jedoch statt Zeiger auf jeden einzelnen Kindknoten nur einen einzigen Zeiger auf ein Array von Kindknoten enthalten. Dadurch wird weniger Speicher durch Zeiger belegt und der CPU-Cache besser genutzt.

In unserer Implementierung wird für das Nachschlagen der ID jedoch bewusst eine andere Struktur eingesetzt, die ursprünglich auf Strings optimiert wurde: Ein modifizierter Trie. Gegenüber (CS)B+ Bäumen hat dieser neben einer deutlich einfacheren Implementierung die nützlichen Eigenschaften, dass keine Duplikation der Daten in den Knoten geschieht und das ermitteln eines Kindknotens nicht durch Suche, sondern eine direkte Berechnung geschieht. Darüberhinaus können Tries gegenüber B+ Bäumen relativ einfach für nebenläufige Zugriffe erweitert werden, wie im Rest dieses Unterabschnitts erläutert wird.

Grundlegender Aufbau der Trie-Struktur Tries gehören zu den Präfix-Bäumen, d.h. sie zerlegen den Schlüssel in kleinere Einheiten und arbeiten diesen von vorne nach hinten ab. Im Falle eines 256-ären Tries, wie er in unserer Datenbank zum Einsatz kommt, handelt es sich bei diesen Einheiten um Bytes, die die Werte 0 bis 255 annehmen. Das höchstwertige Byte ist der Index des ersten Kindknotens, dort wird an Position des zweithöchsten Bytes der nächste Kindknoten ausgewählt usw., bis das niederwertigste Byte des Schlüssels schließlich im Blattknoten den konkreten Wert ausliest.

Abbildung 11 illustriert einen Trie mit 3 Ebenen. Soll beispielsweise der Schlüssel 65025 nachgeschlagen werden, wird er als dreistellige Zahl zur Basis

256 dargestellt: $65025 = (0, 254, 1)_{256}$. Beginnend von der Wurzel schlägt man rekursiv den 0-ten, 254-ten und 1-ten Eintrag nach. Während die ersten beiden inneren Knoten wiederum auf einen Knoten verweisen, wird im Blattknoten an Position 1 direkt die Zahl 14 gelesen, welche die Speicherposition des gesuchten Elements mit ID 1 ist.

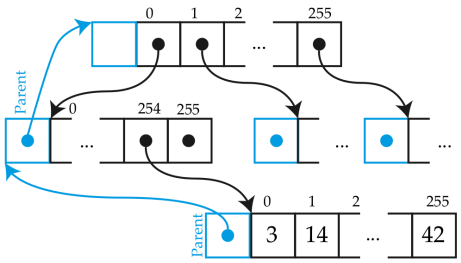


Abb. 11: *Ein 256-ärer Trie mit Tiefe 3 und Rückreferenzen auf Elternknoten. Zum Nachschlagen wird der Schlüssel als dreistellige Zahl zur Basis 256 dargestellt und ein dreifacher Array-Lookup durchgeführt.*

Hat ein Trie-Knoten n Ebenen Abstand zu den Blättern (es gilt $n = 0$ für Blätter), errechnet sich der Kindknoten, den er wählen muss, aus $i_{child} = key/(256^n) \mod 256$. Ein Trie mit Tiefe t hat konsequenterweise einen Definitionsbereich von 0 bis einschließlich $256^t - 1$ und kann zu jedem Schlüssel genau einen Wert speichern.

Schreiben in der Trie-Struktur Da die Menge der Primärschlüssel von vornherein nicht bekannt ist, muss die Trie-Struktur auf eine Überschreitung ihres Definitionsbereichs reagieren können. Dies geschieht wie folgt:

Zunächst besteht der Trie aus einem einzigen Blattknoten und ist demnach ein 256-elementiges Array. Soll ein Wert hinter einem Schlüssel größer als 255 geschrieben werden, so wird ein neuer Wurzelknoten erstellt, dessen 0-tes Element der bisherige Blattknoten wird (s. Abbildung 12). Diese Transformation verändert die gespeicherte Abbildung nicht, nun können jedoch Werte bis $256^2 - 1 = 65535$ geschrieben werden. Existiert ein Kindknoten noch nicht, wird er alloziert und die Referenz dorthin gesetzt.

Trie-Versionierung Transaktionales Lesen und Schreiben in der Datenbank ist nicht Bestandteil dieser, sondern der hierauf aufbauenden Arbeit über Nebenläufigkeit (Siehe Kapitel III). Um die Wahl der vorliegenden Trie-Struktur vollständig zu begründen, wird im folgenden dennoch deren Fähigkeit zur Versionierung vorgestellt.

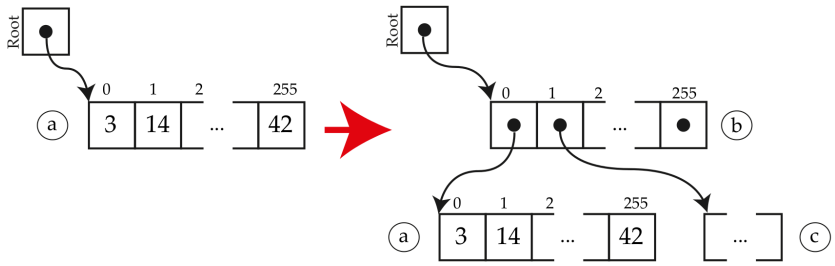


Abb. 12: *Abbildungserhaltende Transformation eines Tries der Tiefe 1 in einen Trie der Tiefe 2. Der Knoten (a) bleibt dabei unverändert und wird zum nullten Element der neuen Wurzel (b), die nun 256-mal so viele Schlüssel adressieren kann. Das Schreiben von Schlüssel 256 würde nun einen neuen Blattknoten (c) allozieren und dort an nullter Position den Wert platzieren.*

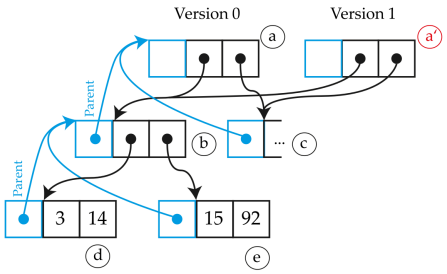


Abb. 13: *Eine flache Kopie des Tries wurde angelegt.*

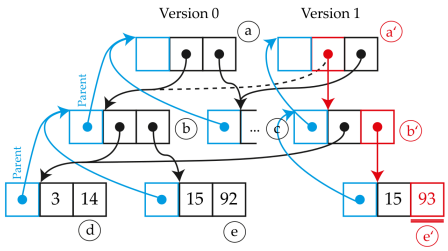


Abb. 14: *Ein Blattknoten wurde verändert. Die Änderung erfordert einen nach oben propagierenden Copy-On-Write-Prozess.*

Wann immer die Indexstruktur einer Transaktion, d.h. mehreren zusammenhängenden Änderungen unterzogen werden muss, welche entweder vollständig oder gar nicht ausgeführt werden sollen, empfiehlt sich das Arbeiten auf einer Kopie der Indexstruktur. Da eine vollständige Kopie aufgrund der Größe der Indexstruktur sowohl zeitlich als auch speichertechnisch unmöglich werden kann, wird in unserer Implementierung nur eine *flache Kopie* des Wurzelknotens erzeugt. Alle Referenzen zeigen noch immer auf die Kindknoten des ursprünglichen Baums, solange bis ein Wert geschrieben wird (s. Abbildung 13).

Bei einem Schreibvorgang, illustriert in Abbildung 14, wird der Blattknoten (e) dupliziert und nur das Duplikat (e') an der betroffen Stelle verändert. Da auch die Referenz auf den neuen Blattknoten in (b) umgesetzt werden muss, propagiert der Kopiervorgang bis zu einem Knoten, der bereits kopiert wurde (hier a') nach oben.

Beim Kopieren eines Knotens wird die Eltern-Referenz auf den ebenfalls kopierten Elternknoten gesetzt. So kann jederzeit entschieden werden, ob ein Kind-Knoten mit einem älteren Trie geteilt wird (Eltern-Referenz zeigt auf eine vorhergegangene Version) und kopiert werden muss, oder bereits kopiert wurde.

Das Übernehmen oder Abbrechen der Änderungen ist im nicht-nebenläufigen Fall trivial: Zur Übernahme genügt das Einsetzen des neuen Tries anstelle des alten. Zum Abbrechen genügt das Deallozieren der kopierten Knoten.

Wenn nebenläufige Änderungen möglich sind, muss vor der Übernahme verifiziert werden, ob die nicht-kopierten Knoten geändert wurden und diese ggf. aus der neuesten Version übernommen werden müssen. Der in unserer Implementierung eingesetzte Algorithmus ist an den Befehl `git rebase` des Versionskontrollsystems GIT angelehnt und führt einen sog.*Three-Way Merge* zur Linearisierung der Änderungshistorie durch. Da jeder Trie das Ergebnis einer vollständigen Transaktion widerspiegelt, können Merge-Konflikte durch überschreiben mit der neuesten Änderung konsistent gelöst werden.

II.4.6 Wahl des Interpreters

Aus Gründen der Performanz und Speichereffizienz ergibt die Implementierung einer Datenbank in einer Skriptsprache nicht sofort Sinn, jedoch unterstützt der aktiv weiterentwickelte Python-Interpreter PyPy [23] unsere Implementierung mit einer Reihe wichtiger Optimierungen, darunter *Just-In-Time-Kompilierung*. Dadurch konnten wir letztendlich eine wesentlich aufwändigere Implementierung in einer Systemprogrammiersprache wie C oder C++ vermeiden und dennoch eine angemessene Performanz erreichen.

Betrachtet man Python 2.7.3 und PyPy 1.9 im direkten Laufzeitvergleich bei Durchführung realistischer Szenarien im Beispielmodell aus Abbildung 17 auf dem in Abschnitt II.5 vorgestellten Testsystem, zeigt sich der in Abbildung 15 dargestellte Performanzvorteil von PyPy gegenüber Python. Die auf der Datenbank ausgeführten Tests sind folgende:

1. **OLTP Lesen:** 100.000x Lesen der Kundenadresse des einer beliebigen Bestellung zugeordneten Kunden. Codeausschnitt:
   `result = Order.at(i).customer.address`.
2. **OLTP Einfügen:** Instantiieren von 100 Produkten, 1.000 Kunden, 10.000 Bestellungen und 100.000 Posten mit Zufallswerten.
3. **OLTP Ändern:** 100.000x Modifizieren eines Attributs. Codeausschnitt:
   `OrderDetail.at(i).count = 1`
4. **OLAP:** Analytische Anfrage, die den Gesamtumsatz aller verkauften Produkte gruppiert nach Produkt und Jahr ermittelt. Implementiert wird diese mit der in Kapitel IV vorgestellten eingebetteten Anfragesprache.

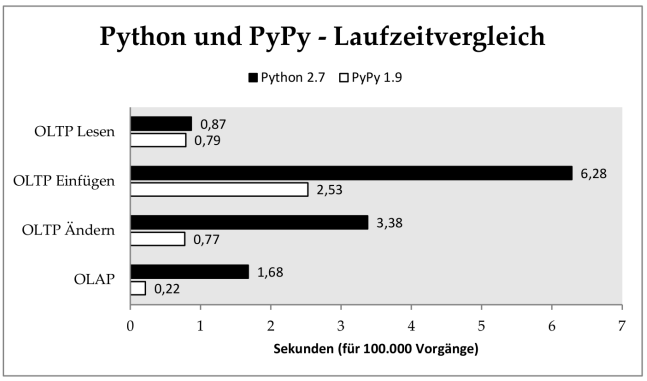


Abb. 15: *PyPy erweist sich in typischen Datenbank-Szenarien als schneller als Python*

Es ist außerdem bekannt, dass PyPy in Einzelfällen die Geschwindigkeit äquivalenter C-Programme überbieten kann. [11] Genauere Analysen der PyPy-Optimierungen auf der sprachintegrierten Datenbank, und wie man diese für analytische Anfragen (OLAP) explizit nutzen kann, sind in Kapitel IV ausgeführt.

II.5 Evaluierung

Die neue, spaltenbasierte Speicherorganisation von Python-Objekten hat das Ziel, eine Reihe von Vorteilen aus der Welt der Datenbanken in eine objektorientierte Laufzeitumgebung zu übertragen. Der Grad der Zielerreichung soll in diesem Abschnitt auf Basis von Messungen dargelegt werden.

Dazu vergleichen wir einerseits die Speichereffizienz unserer neuen Objekte mit der herkömmlicher Objekte in der Python-Laufzeit und der von Tupeln in SQLite, um zu demonstrieren, dass die spaltenorientierte Speicherung bereits in der wenig optimierten Implementierung des Prototypen kompakter ist. Andererseits vergleichen wir die Geschwindigkeit, mit der unser Prototyp eine Reihe von normalerweise in SQL implementierten Aufgaben ausführt mit der Geschwindigkeit einer verbreiteten Datenbank. SQLite teilt sich den Adressraum mit dem Prozess und kann ebenfalls vollständig im Hauptspeicher operieren, weshalb wir den Vergleich mit dieser Datenbank einer in separaten Prozessen arbeitenden Datenbank vorgezogen haben.

II.5.1 Testvoraussetzungen

Alle hier aufgeführten Tests wurden mit einer Version des Prototypen durchgeführt, welche die in Kapitel III erläuterten Mechanismen zur Transaktions- und Nebenläufigkeitskontrolle unterstützt und dadurch gegenüber der hier erläuterten Implementierung aufwändigere (transaktionale) Lese- und Schreibvorgänge ausführt.

Als Interpreter kommt die vorkompilierte 32-Bit-Version PyPy 1.9.1 JIT mit SQLite 3 auf Windows 7 64-Bit zum Einsatz. Als Hardware stand eine Intel Core i7-2720QM CPU mit 4 x 3,30 GHz, Hyper-Threading und 16 GB DDR3-RAM zur Verfügung. Die Auslagerungsdatei wurde deaktiviert, sodass die Tests ausschließlich im RAM stattgefunden haben.

Weder in SQLite noch in unserer Datenbank werden explizit Indizes angelegt, die Operationen in bestimmten Fällen beschleunigen oder bremsen könnten. Lediglich Primär- und Fremdschlüssel sind definiert. SQLite wurde durch die folgenden beiden Optionen auf Hauptspeicherbetrieb optimiert: `PRAGMA synchronous = OFF` aktiviert asynchrone Verarbeitung, `temp_store = 2` verlagert sämtliche Caches ausschließlich in den Hauptspeicher.

II.5.2 Speicherverbrauch

Dieser Benchmark misst die Anzahl an Entitäten, die wahlweise mit unserer Datenbank mit und ohne Kompression bzw. mit normalen Python-Objekten oder in SQLite gespeichert werden können, bevor ein `MemoryError` von Python erzeugt wird. In der 32-Bit-Version tritt dieser auf, sobald 2 GB RAM aufgebraucht wurden.

```
1 @column_layout
2 class Vertex(object):
```

```
3       drain = One(Self)
4       weight = int
```

Quelltext 4: *Kleine Benchmark-Entität*

Der erste Test arbeitet mit kleinen Entitäten, die lediglich eine Ganzzahl zwischen 0 und 10 sowie eine Referenz auf eine andere Entität gleichen Typs speichern. Quelltext 4 zeigt die Beispielklasse.

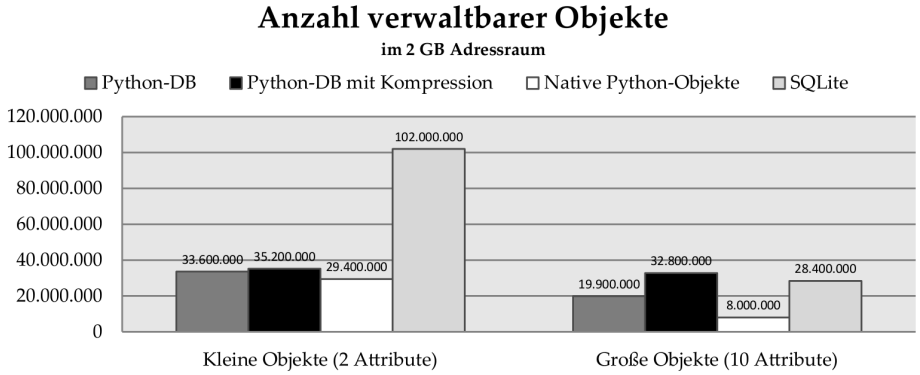


Abb. 16: *Gemessene Anzahl erzeugbarer Objekte in 2 GB Hauptspeicher. Die spaltenbasierte Speicherorganisation ermöglicht gegenüber normalen Python-Objekten eine höhere Packungsdichte der Daten im Speicher.*

Der Overhead, welcher durch Verwaltungs-Datenstrukturen (insbesondere durch die Transaktionen und MVCC aus Kapitel III) anfällt, hält sich gegenüber der Dictionary-basierten Speicherung normaler Python-Objekten die Waage, Kompression nutzt nur geringfügig etwas, da Metadaten und Fremdschlüssel nicht effektiv komprimiert werden. SQLite, welches weder MVCC noch Kompression benutzt, kann die kleinen Tupel noch wesentlich effizienter Ablegen.

Im zweiten Test wurden Objekte mit 10 ganzzahligen Attributen angelegt, deren Wertebereich jeweils 10 Elemente umfasst. Hier wiegt der Einfluss der Metadaten deutlich weniger, sodass die Kapazität der neuen Speicherorganisation gegenüber nativen Python-Objekten sich bereits ohne Kompression um Faktor 2,4 erhöht hat. Durch Kompression kann sogar die tupelbasierte Speicherung von SQLite übertroffen werden.

Insgesamt zeigt sich also, dass die spaltenorientierte Speicherung in Hinblick auf Speichereffizienz deutlich besser als normale Python-Objekte abschneidet und bei größer werdenden Entitäten mit nicht-komprimierenden Datenbanken wie SQLite mithalten kann.

II.5.3 Performanz

Wir werden im Folgenden die Anzahl von Einfüge-, Schreib- und Lesevorgängen pro Sekunde in unserer Datenbank mit SQLite und nativen Python-Objekten vergleichen.

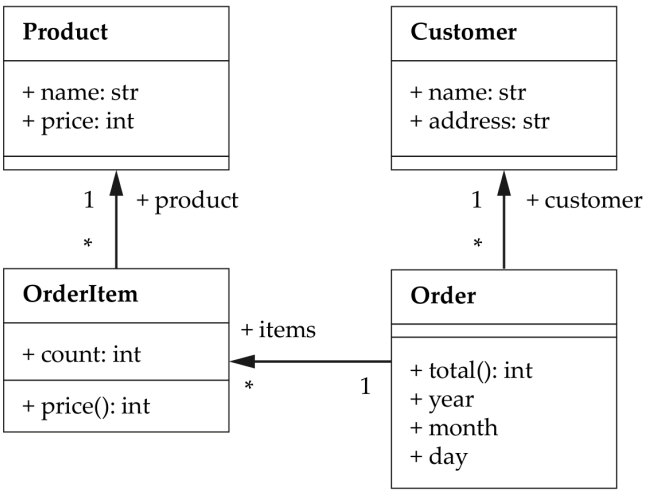


Abb. 17: *Das Beispiel-Modell, welches den Benchmarks zu Grunde liegt*

Um einen Eindruck von der OLTP-Performanz zu bekommen, führen wir nacheinander die folgenden drei Szenarien aus und messen die Performanz der sprachintegrierten Datenbank sowie einer äquivalenten Anfrage an SQLite.

Einfügen Eingefügt werden (dem Modell in Abbildung 17 entsprechend) 10 Mio. Posten (`OrderDetail`) mit zufälliger Anzahl und zufälligem Produkt, 1 Mio. Bestellungen mit je 10 Posten, 100.000 Kunden mit je 10 Bestellungen und 10.000 Produkten. Insgesamt sind danach 11.110.000 Objekte angelegt. Während SQLite 1,25 Mio. Elemente pro Sekunde via SQL-Insert-Statement anlegen kann, gelingen der sprachintegrierten Datenbank nur 667.000 Instanzerzeugungen (s. Abbildung 18). Dies weist u.a. auf die Notwendigkeit eines Bulk-Insert-Modus hin, denn bei der normalen Instanzerzeugung wird stets ein Referenzobjekt miterzeugt, welches sofort wieder verworfen wird.

Einfache Anfragen Als Beispiel für eine einfache Anfrage soll eine objektorientiert prägnant auszudrückende Anfrage dienen: Das Ermitteln der Kundenadresse zu einer Bestellung, z.B. via `order.customer.address`. In SQL er-

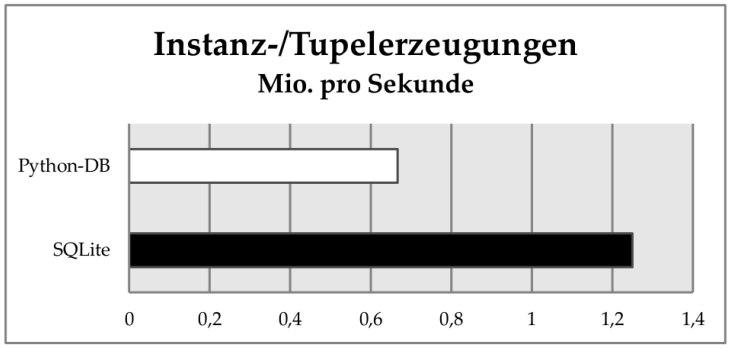


Abb. 18: *SQLite kann pro Sekunde knapp doppelt so viele Tupel in eine Datenbank eintragen als die sprachintegrierte Datenbank Objekte erzeugen.*

fordert dies eine komplexere relationale Anfrage mit Join-Bedingungen, s. Abschnitt II.5.5. SQLite erkennt scheinbar die Semantik der Anfrage, nämlich dass nur ein einziges Tupel selektiert werden soll, nicht mehr und führt einen *Join* an dieser Stelle aus. Der entstehende Overhead verlangsamt SQLite um Faktor 20, wie in Abbildung 19 dargestellt.

Bei einfachen, objektorientierten Zugriffen ist die gewählte Implementierung der Assoziationsauflösung offenbar dem relationalen Modell überlegen.

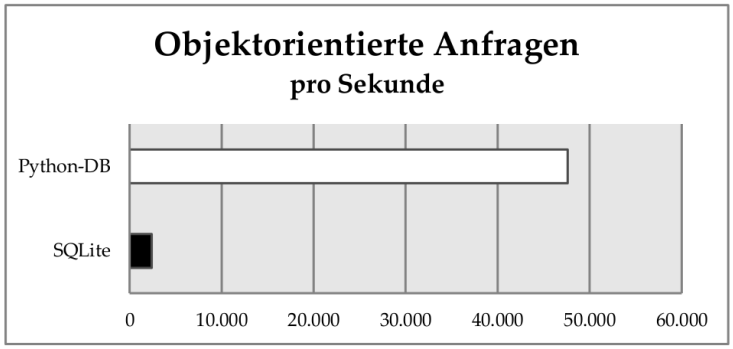


Abb. 19: *Das Folgen von Assoziationen erfordert in SQLite einen Join und lässt die Performanz gegenüber Python einbrechen.*

Verschachtelte Anfragen Das folgende Szenario soll ausgehend von der eben eingefügten Datenbasis zu 100 verschiedenen Kunden den Umsatz ermitteln. Die genaue Anfrage ist im Abschnitt II.5.5 beschrieben. Im Wesentlichen soll zu einem beliebigen Kunden die Werte aller seiner Bestellungen ermittelt werden, wobei der Wert einer Bestellung seinerseits aus der Summe aller Posten besteht. Ein Posten-Wert errechnet sich aus Produktpreis multipliziert mit der Menge.

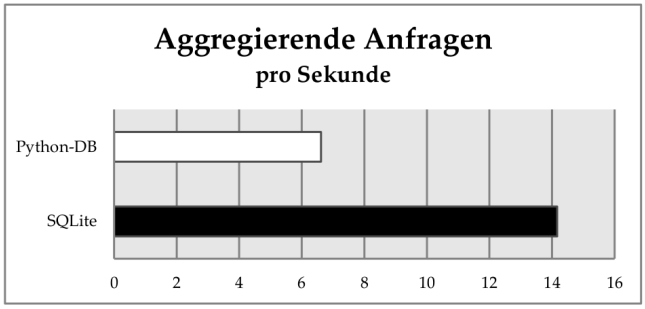


Abb. 20: *Beinhalten Anfragen Mengenoperationen, ist relationales SQL konzeptuell im Vorteil.*

Da Mengenoperationen (z.B. das Abrufen aller Posten einer Bestellung) im Prototypen mittels linearer Suche implementiert sind, ergibt sich bei verschachtelten Anfragen (z.B. über alle Bestellungen eines Kunden) ein quadratisches Laufzeitverhalten. SQL kann dies durch eine *Join*-Operation mengenbasiert mit günstigerem Laufzeitverhalten lösen. Abbildung 20 bestätigt diesen Nachteil der objektorientiert formulierten Anfrage gegenüber dem relationalen Modell von SQLite; unsere Python-DB schafft mangels Optimierungen hier nur 47% der SQLite-Leistung.

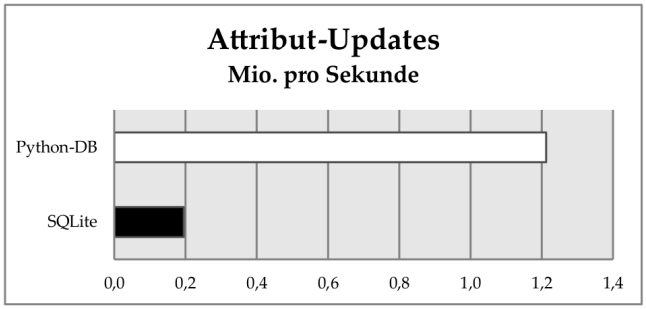


Abb. 21: *Schreibvorgänge sind in der integrierten Datenbank deutlich schneller als via SQL in SQLite.*

Updates Im letzten Szenario soll die Mengenangabe eines Postens auf 1 gesetzt werden. Dieses einfache Update wird für 1 Mio. verschiedene Posten ausgeführt. Die direkte Übersetzung der Attribut-Zuweisung in unserer sprachintegrierten Datenbank schneidet trotz Einsatz von MVCC in diesem Test gegenüber eines SQL-Statements um Faktor 6,2 performanter ab, wie in Abbildung 21 zu sehen. Die konkrete Anfrage ist in Abschnitt II.5.5 aufgeführt.

II.5.4 Zusammenfassung der Messergebnisse

Die Tatsache, dass eine in der Skriptsprache Python implementierte Datenbank in Verbindung mit einem modernen JIT-Compiler in einigen Szenarien sowohl Performanz- als auch Speichervorteile gegenüber einer etablierten Datenbank bietet, ist ein klares Indiz für das Potential der von uns untersuchten Sprachintegration.

Die Eliminierung von Schichten zahlt sich besonders bei einfachen Vorgängen aus, z.B. bei kleinen Änderungen auf einzelnen Objekten oder dem Folgen von Assoziationen, wie sie in OOP-Sprachen allgegenwärtig sind. Benachteiligt ist die sprachintegrierte Datenbank bei Aggregaten oder Mengenoperationen, allerdings ist die Speicherorganisation auch auf solche Operationen ausgelegt und beweist bei der Implementierung einer Anfragesprache in Kapitel IV, dass sie auch hier besser als andere Datenbanklösungen abschneiden kann, wenn sie richtig eingesetzt wird.

Mit dem Nachweis der Effektivität einer einfachen Kompression ist außerdem gezeigt, dass wir eine ursprünglich für Hauptspeicherdatenbanken eingesetzte Technologie erfolgreich auf die Speicherorganisation von Objekten übertragen haben. Hieran anknüpfend können weitere Optimierungen stattfinden, die das teure Speichermedium noch effizienter mit Objekten als Datenbank nutzbar machen.

II.5.5 Benchmarks

Die oben stehenden Benchmarks basieren auf den folgenden Anfragen.

Einfache Anfragen Dieser Python-Code, der die Kundenadresse zu einer Bestellung ausgibt, ist in SQL wesentlich aufwändiger zu formulieren:

```
1 def address(order_id):
2     customer = Order.at(order_id).customer
3     return customer.name, customer.address
```

Das äquivalente SQL-Statement:

```
1 SELECT customers.name, customers.address
2 FROM customers, orders
3 WHERE orders.id = %s
4 AND orders.customer_id = customers.id
```

Verschachtelte Anfragen Folgender Python-Code ermittelt zum Beispielmodell aus Abbildung 17 zu einem beliebigen Kunden mit gegebener ID den von ihm gemachten Umsatz. Die Verwendung einer Menge (`self.details`) sowie mehrerer Summen macht die Anfrage ohne Optimierung sehr kompliziert, da die Mengen im Prototypen mit linearer Zeitkomplexität gesucht werden.

```
class Order:
    # ...
    def total(self):
    '''Berechnet den Gesamtwert dieser Bestellung'''
        return sum(
            (d.product.price * d.count for d in self.details)
        )

# Anfrage
def customer_revenue(customer_id)
    return sum(
            (o.total() for o in \
                        Customer.at(customer_id).orders)
        )
```

Äquivalent dazu folgende SQL-Anfrage:

```
SELECT SUM(order_details.count * products.price)
FROM order_details, orders, products
WHERE order_details.order_id = orders.id
AND order_details.product_id = products.id
AND orders.customer_id = ?
```

Updates Folgendes Update in Python ist sehr viel schneller als das äquivalente SQL-Statement in SQLite:

```
def update(detail_id):
        OrderDetail.at(detail_id).count = 1
```

Ein äquivalentes SQL-Statement:

```
UPDATE order_details
SET count = 1
WHERE id = ?
```

II.6 Ausblick

II.6.1 Persistenz

Hauptspeicherdatenbanken sind im allgemeinen sehr Empfindlich, da u.a. Strom- oder Hardwareausfälle erheblichen Datenverlust zur Folge haben, wenn die Daten nicht kontinuierlich auf ein persistentes Speichermedium geschrieben werden. Die Empfindlichkeit wird in unserer Implementierung zusätzlich dadurch erhöht, dass ein Fehler im Anwendungscode mangels Prozess-Isolation auch die laufende Datenbank zerstören kann.

Da das persistente Abbild der Datenbank auf eine Weise geschrieben werden muss, sodass ein etwaiger Ausfall eben dieses nicht beschädigt oder inkonsistent machen kann, ist die Implementierung einer solchen transaktionalen Persistenz nicht trivial. Sie fand noch keinen Weg in diese Arbeit, da sie keinen wesentlichen Beitrag zur Laufzeit-Speicherorganisation oder zu Sprachkonzepten leistet.

II.6.2 Indizes

Im Prototypen ist die Zeitkomplexität zum Auffinden einzelner Werte in einer Spalten aktuell noch linear. Dies wird problematisch beim Auflösen von Mengen-Assoziationen, wenn alle Einträge gefunden werden müssen, die einen bestimmten Fremdschlüssel beinhalten. Dies beeinträchtigt auch den Bedienkomfort der Mengenassoziationen erheblich und könnte Entwickler eher zu Workarounds motivieren als zur Nutzung bereitgestellter, optimierter Schnittstellen.

Lösen lässt sich dieses Problem durch die Einführung von Indizes, die etwa mittels eines B+ oder CSB+ Baums implementiert werden können. Besonders auf Fremdschlüssel-Spalten von Many- und ManyToMany-Assoziationen könnten solche Indizes automatisch erstellt werden. Die vorhandene Trie-Implementierung ist für diesen Anwendungsfall nicht geeignet, da sie keine Schlüssel-Duplikation und keine anderen Schlüsseltypen als Ganzzahlen zulässt.

II.6.3 Kompression

Die eingesetzte Kompression ist vergleichsweise schwach und nicht auf tatsächliche Geschäftsdaten ausgelegt. Spalten, die größtenteils leer sind, z.B.

Namenszusätze in einer Personentabelle, lassen sich mit einer Lauflängenkompression deutlich besser kompaktieren. Textdaten lassen sich beispielsweise durch den LZW-Algorithmus sehr schnell komprimieren und extrahieren oder in einer anderen Darstellung (z.b. mit Patricia-Tries) redundanzfreier aufbewahren.

II.7 Verwandte Arbeiten

II.7.1 Gemstone

Eine erfolgreiche Integration zwischen Anwendungssprache und Datenbank gelang mit Gemstone[7]. Der wesentliche Unterschied besteht allerdings in der Speicherorganisation: Während Gemstone Objekte zusammenhängend speichert, speichert die sprachintegrierte Python-Datenbank Werte des gleichen Attributs (Spalten) zusammenhängend. Dies liegt in der Zeit begründet, zu der Gemstone entstand. Geschäftstaugliche Hauptspeicherdatenbanken waren zu dieser Zeit noch nicht in Betracht zu ziehen, sodass Aspekte wie Kompression und schnelle Aggregationen unter Ausnutzung des CPU-Caches nachrangig waren. Nichtsdestotrotz hat Gemstone die Entwicklung dieser sprachintegrierten Datenbank stark motiviert und inspiriert.

II.7.2 SAP HANA

Während Gemstone eine für Datenbanken revolutionäre Sprachintegration aufwies, ist die aktuelle hauptspeicherresidente Datenbanklösung von SAP ein eigenständiges System. Die in HANA[28] (*High Performance Analytic Application*) eingesetzten Technologien, u.a. die Spaltenorganisation für schnellere Aggregationen und bessere Platzausnutzung dienten letztendlich als Vorlage für die Speicherorganisation unseres Prototypen. Hervorzuheben ist auch die Möglichkeit, in HANA ähnlich wie mit Gemstone sehr datennah programmieren zu können, allerdings in den eigens entwickelten Sprachen *SQLScript* und *L*, welche von der Anwendungssprache verschieden sind, auf relationalen respektive imperativen Konzepten basieren und keine derart hilfreiche Toolunterstützung wie verbreitete Anwendungssprachen aufweisen.

Aufgrund der technischen Nähe unseres Prototypen zu HANA sollte es möglich sein, auf dieser Arbeit aufsetzende Konzepte, insbesondere jene aus Kapitel III und Kapitel IV auch auf HANA zu übertragen.

II.8 Fazit

Ziel dieser Arbeit war die Integration einer *Hauptspeicherdatenbank* in die *Laufzeitumgebung* von Python, um objektorientiertes Programmieren auf den Daten ohne Umwege über *Objektrelationale Mapper*, *SQL* und Prozessgrenzen zu ermöglichen. Gleichzeitig sollte gezeigt werden, dass die dazu eingeführten Sprachkonzepte auch auf der Speicherorganisation moderner Hauptspeicherdatenbanken aufsetzen können.

Gelungen ist die Umsetzung dieser Ziele durch die Änderung der Speicherorganisation bestimmter Python-Objekte in eine *spaltenbasierte* Darstellung. Wir konnten erfolgreich die Grundfunktionalität einer Datenbank in der objektorientierten Sprache Python bereitstellen, indem wir Entitäten als Klassen mit einem speziellen Dekorator darstellten und Attribute sowie Assoziationen via Klassenvariablen bereitstellten. Das darunterliegende Spaltenlayout nutzten wir ähnlich wie *Objektrelationale Mapper* (ORM), um Objektidentität mittels automatisch vergebenen, indizierten *Primärschlüsseln* und Referenzen mittels *Fremdschlüsseln* auszudrücken. Die dadurch gewonnene Unabhängigkeit der Objektidentität von der konkreten Speicherstelle ermöglichte es, die Speicherorganisation weiter zu verfeinern und Kompression oder Segmentierung zur effizienteren Speicherausnutzung einzubauen. Darüber hinaus bietet diese Interpretation von Objektidentität sowie die Implementierung der primären Indexstruktur Anknüpfungspunkte für die in [Berov 2012] implementierten Mechanismen zur Transaktions- und Nebenläufigkeitskontrolle via *Multiversion Concurrency Control* (MVCC).

Bis auf einige in der Evaluierung festgestellte, behebbare Schwachstellen kann die prototypische Implementierung sogar mit der verbreitetsten In-Process-Datenbank SQLite in Sachen Performanz und Speichereffizienz mithalten, was unsere Entwurfsentscheidungen sowie die Wahl einer Skriptsprache mit JIT-Compiler gegenüber einer nativen Spracherweiterung in C/C++ weitestgehend bestätigt. Letztendlich ist die in dieser Arbeit vorgestellte Speicherorganisation auch auf schnelle analytische Anfragen (OLAP) ausgelegt, wobei eine erfolgreiche und sehr performante Implementierung einer API und eines Anfrageprozessors für OLAP in Kapitel IV vorgestellt wird.

Teil III

Nebenläufigkeitskontrolle in einer sprachintegrierten Hauptspeicherdatenbank

III.1 Einleitung

Wenn Anwendungen Daten aus einer Datenbank abfragen, dann werden diese in den privaten Hauptspeicher der Anwendung kopiert. Mit dem Aufkommen von Hauptspeicherdatenbanken befinden sich diese Daten allerdings bereits im Hauptspeicher. Unser Bachelorprojekt (siehe auch Kapitel II, Kapitel IV und Kapitel VII) untersucht die Vorteile einer Verschmelzung der Adressräume einer Hauptspeicherdatenbank und der Laufzeitumgebung einer höheren Programmiersprache, einer s.g. *Sprachintegrierte Hauptspeicherdatenbank*.

Durch diese Verschmelzung bedeutet jeder Zugriff auf Daten durch eine Anwendung gleichzeitig eine Anfrage an die Datenbank. Da Anwendungen zunehmend nebenläufig programmiert werden, und Datenbanken potentiell mehrere Anwendungen gleichzeitig unterstützen, muss eine sprachintegrierte Hauptspeicherdatenbank die Verarbeitung nebenläufiger Anfragen unterstützen. Dies birgt Probleme, da Anwendungen nicht mehr isoliert ausgeführt werden. Eventuell auftretende inkonsistente Zwischenstände einer Anwendung können somit die Funktion einer Anderen beeinträchtigen. Bei einer sprachintegrierten Hauptspeicherdatenbank ist dies besonders problematisch, da jede Operation einer Anwendung sich sofort in der Datenbank widerspiegelt und sofort alle anderen Ausführungskontexte beeinflusst. Gleichzeitig sind inkonsistente temporärer Zustände selten vermeidbar, da Anwendungen oft mehrschrittige Bearbeitungsvorgänge implementieren. Um das Propagieren dieser inkonsistenter Zwischenzustände zu vermeiden, muss ein System zur Nebenläufigkeitskontrolle implementiert werden. Die vorliegende Arbeit stellt die Umsetzung eines solchen Systems zur Nebenläufigkeitskontrolle für den Prototypen einer Sprachintegrierte Hauptspeicherdatenbank (SDB) vor.

Gegliedert ist die Arbeit dabei wie folgt: Im nächsten Kapitel wird der Aufbau der SDB grob umschrieben, da er den Rahmen für die restliche Arbeit setzt. Grundlagen sowie unterschiedliche Prinzipien der Nebenläufigkeitskontrolle in Datenbanksystemen werden in Kapitel III.3 eingeführt. Darauf aufbauend wird in Kapitel III.4 der Entwurf eines Systems zur Nebenläufigkeitskontrolle in der SDB vorgestellt. Seine Implementierung wird im Anschluss in Kapitel III.5 erläutert. Eine Auswertung dieser Implementierung unter Betrachtung von Nebenläufigkeitsproblemen, Laufzeitverhalten und veränderter Sprachnutzung für den Anwender erfolgt in Kapitel III.6. Im Ausblick, Kapitel III.7, werden ausgehend von der Auswertung Verbesserungsmöglichkeiten vorgestellt und Erweiterungen skizziert, die durch die Besonderheiten des Prototypen möglich werden. Ein Fazit wird in Kapitel IV.9 gezogen.

III.2 Eine Sprachintegrierte Hauptspeicherdatenbank in Python

Der SDB-Prototyp ist eine Erweiterung der Python Laufzeitumgebung, die Attribute ausgewählter Objekte in eine insert-only Hauptspeicherdatenbank auslagert. Dadurch erfolgt eine (teilweise) Verschmelzung der Adressräume von Python-Laufzeitumgebung und Datenbank. Durch den geteilten Hauptspeicher sind keine Anfragesprachen wie SQL nötig, um Daten aus der Datenbank abzufragen. Da der Prototyp automatisch Tabellen und Spalten zur Persistierung von Klassen anlegt und diese mit Methoden zum Zugriff auf sie ausstattet, ist außerdem kein zusätzlicher Object Relational Mapper nötig, um die gelesenen Daten in Objekte zu übersetzen. Die Architektur des Systems ist optimiert für den PyPy-Interpreter[1], der die Ausführung mithilfe von Just-in-Time Compiling optimiert.

In Abschnitt III.2.1 wird kurz die Nutzung des Prototypen erläutert und im Anschluss wird in Abschnitt III.2.2 vereinfacht auf Details der Umsetzung eingegangen, die für die vorliegende Arbeit relevant sind. Eine detaillierte Abhandlung der Implementierung kann in Kapitel II gefunden werden.

III.2.1 Nutzung der SDB

Nicht alle Daten in einem Softwaresystem müssen gespeichert werden. Klassen, deren Instanzen persistiert werden sollen, müssen deswegen explizit mit der Annotation `@column_layout` versehen werden. Da auch nicht unbedingt alle Daten-Attribute[2] persistiert werden sollen, ist es außerdem notwendig, diejenigen Instanzvariablen typisiert zu deklarieren, die zu persistieren sind. Dazu muss der Instanzvariable der primitive Datentyp, den sie später speichern soll, zugewiesen werden. Speichert die Variable eine Assoziation zu einer anderen Klasse, so muss ihr statt dessen eine Instanz der Assoziations-Klassen `One`, `Many`, oder `HasAndBelongsToMany`, parametrisiert mit der Zielklasse, zugewiesen werden, je nach dem ob es sich um eine n:1, 1:n oder n:m Assoziation handelt (siehe Quelltext 5). Der Umgang mit Instanzen annotierter Klassen verläuft wie üblich. Es ist jedoch zu beachten, dass das Lesen und Schreiben persistierter Attribute immer innerhalb eines Transaktionskontexts zu geschehen hat. Um eine Methode als Transaktion zu kennzeichnen, muss sie mit `@transactional` oder `@retrying` dekoriert werden.

[1] `http://www.pypy.org/` (Stand: 10.06.2012)

[2] Im Python-Sprachgebrauch sind Daten-Attribute auf Instanzen das Äquivalent zu Instanzvariablen in Smalltalk. Im weiteren Verlauf werden diese Begriffe synonym verwendet.

Um deklarative Anfragen auf der Gesamtheit der Daten in der Datenbank führen zu können, hat jede Klasse eine Methode `all_instances`, die ein Stellvertreterobjekt zurück gibt, das wiederum Methoden für solche Anfragen und ihre Auswertung anbietet. Details zu der deklarativen Anfragesprache, die daraus resultiert, können Kapitel IV entnommen werden.

```
1  @column_layout # annotation for persistence
2  class Person(object):
3      # declaration of persisted attribut
4      age = int
5      # declaration of persisted association
6      car = One(Vehicle)
7
8      def mature(self, years):
9          self.age += years # reading and writing attributes
```

Quelltext 5: *Deklaration einer persistierenden Klasse in Python mit SDB*

III.2.2 Umsetzung der SDB

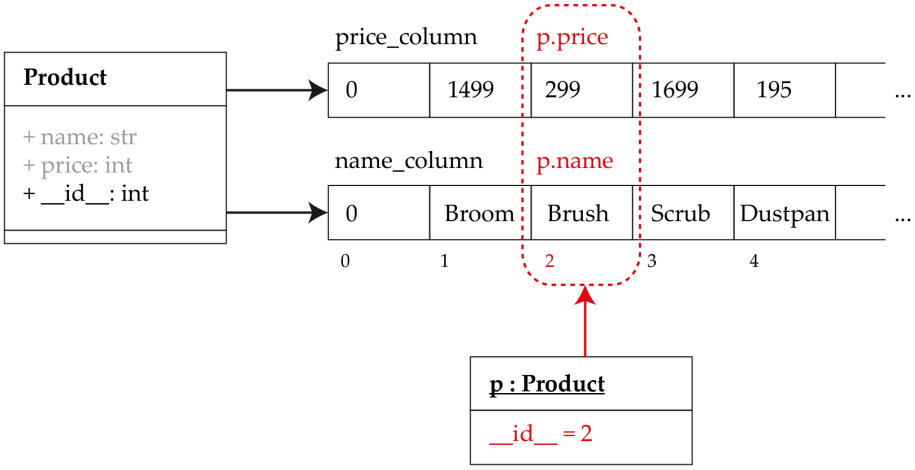


Abb. 22: *Property-Dispatch der SDB anhand eines Beispiels: Die Getter- und Setterproperties werden so überschrieben, dass sie auf Arrays operieren. Der Index des Zugriffs wird aus der ID der Instanz berechnet. Abbildung entnommen aus Kapitel II.*

Die Instanzen persistierender Klassen der SDB bestehen de facto lediglich aus einer eindeutigen ID größer Null, die bei der Instanziierung vergeben wird. Alle Methoden zum Zugriff auf Attribute werden klassenseitig verwaltet. Der Aufbau dieser Methoden erfolgt beim Laden des Moduls der Klasse. Für jede

Instanzvariable – die einen primitiven Datentyp persistieren soll – wird ein Array angelegt das von einer Instanz der `Column`–Klasse verwaltet wird. Der Klasse werden im Anschluss Getter- und Settermethoden hinzugefügt, um auf die `Column`s zugreifen zu können. Diese Methoden arbeiten abhängig von der ID der rufenden Instanz auf einem Wert aus dem Array des angefragten Attributs (siehe Abb. 22). Die Übersetzung von Objekt-IDs in Array-Indizes übernimmt eine Indexstruktur (für Details siehe Kapitel II). Diese Indirektion ist nötig, da zu einem Objekt mehrere Indizes gehören können, was eine einfachere Implementierung von Nebenläufigkeit ermöglicht. Dieses Multiversion Concurrency genannte Prinzip wird in Abschnitt III.3.3 vorgestellt. Die Gesamtheit der `Column`s einer Klasse wird von ihrer `Table` verwaltet.

Die Aufbau von `One`–Assoziationen erfolgt analog zu dem von Attributen. Das zugehörige Array speichert dabei aber nicht die primitiven Werte der Instanzvariablen, sondern die IDs der referenzierten Objekte. Beim Aufbau von `Many`–Assoziationen wird ebenfalls ein Fremdschlüssel gespeichert, allerdings auf Seiten der referenzierten Klasse. Eine `Many` Getter-Methode gibt also diejenigen Instanzen der Zielklasse zurück, deren Fremdschlüssel-Eintrag die ID des referenzierenden Objekts enthält. Entsprechend setzt eine `Many`–Settermethode den Fremdschlüssel des referenzierten Objekts auf die ID des referenzierenden Objekts. Für die Auflösung von `HasAndBelongsToMany`–Assoziationen (im Folgenden `HBTM`) wird eine Tabelle mit zwei Spalten, eine `DoubleColumn`, aufgebaut. Diese speichert in ihren Spalten jeweils die IDs von Instanzen einer der beiden beteiligten Klassen. Eine `HBTM`–Settermethode fügt dieser Tabelle eine neue Zeile hinzu, die die ID der referenzierenden und die der referenzierten Instanz enthält. Eine Gettermethode gibt dem entsprechend alle IDs der Fremdspalte zurück, die sich an einem Index befinden, an dem in der eigenen Spalte die ID der referenzierenden Instanz steht.

III.3 Nebenläufigkeit

Der SDB-Prototyp setzt Multiversion-Concurrency-Control (MVCC) ein. Dieser Ansatz erlaubt nebenläufiges Operieren auf Daten, ohne Anfragen verzögern oder blockieren zu müssen. Er baut dabei auf dem Transaktionskonzept auf, dass dafür sorgt, dass Anwendungsprogrammierer von Nebenläufigkeit und ihren Folgen abstrahieren können. Dieses Konzept wird in Abschnitt III.3.1 beschrieben. Im Anschluss wird in Abschnitt III.3.2 erläutert warum das nebenläufige Ausführen von Transaktionen problematisch ist, und welche Arten von Ansätzen existieren um diese Probleme zu lösen. Darauf aufbauend wird in Abschnitt III.3.3 Multiversion-Concurency-Control als eine konkrete Art der Nebenläufigkeitskontrolle vorgestellt.

III.3.1 Transaktionen

Die Transaktions-Definition in dieser Arbeit folgt Weikum [30, S. 22–25]. Laut ihr sind Transaktionen ein Konzept, das es Anwendungsprogrammen erlaubt, beim Lesen und Modifizieren von Daten von Nebenläufigkeit und Fehlerbehandlung zu abstrahieren. Die Transaktionsschnittstelle sieht vor, das Anwendungsprogramme Transaktionsgrenzen festlegen, in dem sie `begin` und `commit` oder `abort` rufen. Die Datenbank betrachtet alle Anfragen innerhalb dieser Grenzen als Teil der selben Transaktion.

Um Fehler bei Nebenläufigkeit zu vermeiden, müssen die vier ACID-Kriterien garantiert werden:

1. **Atomicity**: Eine Transaktion wird entweder ganz oder gar nicht ausgeführt. Die Änderungen durch die Transaktion werden erst beim erfolgreichen Erreichen des `commit`-Befehls sichtbar gemacht.
2. **Consistency**: Eine Transaktion überführt die Daten von einem konsistenten Zustand in einen Anderen. Temporäre inkonsistente Zustände sind lediglich während einer Transaktion erlaubt.
3. **Isolation**: Eine Transaktion muss von anderen Transaktionen komplett isoliert sein. Jede Transaktion verhält sich so, als würde sie alleine auf den Daten operieren.
4. **Durability**: Die Änderungen einer Transaktion, deren `commit`-Befehl erfolgreich ausgeführt wurde, müssen persistiert werden.

Die Einhaltung dieser Kriterien obliegt zum Großteil dem Datenhalter, allerdings muss z.B. das Konsistenzkriterium auch von der Endanwendungen beachtet werden. Die für ein Datenbanksystem relevanten Operationen innerhalb von Transaktionen sind Lese- und Schreibzugriffe, sowie deren Reihenfolge [29, S. 918]. Außerdem sind die Befehle zum Start und zum Ende relevant. Die Berechnungen, die zu diesen Operationen führen sind für ein Datenbanksystem dagegen nicht von Bedeutung.

III.3.2 Nebenläufigkeitskontrolle

Nach den in Abschnitte III.3.1 vorgestellten Prinzipien überführt eine Transaktion die Daten dann von einem konsistenten Zustand in einen anderen konsistenten Zustand, wenn sie in Isolation ausgeführt wird. Um dabei eine gute Performanz, sprich hohen Durchsatz und geringe durchschnittliche Antwort-

zeiten[3] [30, S. 26] zu erreichen, muss eine Datenbank Transaktionen aber auch nebenläufig ausführen. In diesem Fall ist das Isolations-Kriterium nicht mehr a priori gegeben, sondern muss durch Algorithmen zur Nebenläufigkeitskontrolle garantiert werden.

Die zeitlich sortierte Abfolge der relevanten Operationen nebenläufig ausgeführter Transaktionen wird *Schedule* genannt. Ein Schedule S wird *seriell* genannt, wenn für alle beteiligten Transaktionen T und T' gilt, dass wenn eine Operation von T vor allen Operationen von T' ausgeführt wird, dann alle Operationen von T vor denen von T' ausgeführt werden [29, S. 919]. Ein serieller Schedule führt also erst alle Operationen einer Transaktionen aus, bevor die Nächste operieren kann. Ein *nebenläufiger Schedule* mischt Operationen mehrerer Transaktionen, diese werden also gleichzeitig ausgeführt. Wenn er dabei die Daten in den selben Endzustand überführt, wie ein beliebiger serieller Schedule mit den selben Transaktionen, so wird er *serialisierbar* genannt [29, S. 920].

Die Aufgabe eines nebenläufigen Datenbanksystems ist es also zu gewährleisten, dass aus den eingehenden Lese- und Schreiboperationen serialisierbare Schedules erzeugt werden. Dies kann in zwei unterschiedlichen Modi erfolgen:

- **Pessimistisch:** Das Scheduling erfolgt unter der Annahme, dass die (nach Eingangszeit geordneten) Operationen nicht serialisierbar sind. Damit müss-en potentiell konfligierende Operationen geblockt bzw. verzögert werden, bevor sie ausgeführt werden können, um Serialisierbarkeit zu gewährleisten.
- **Optimistisch:** Das Scheduling erfolgt unter der Annahme, dass die (nach Eingangszeit geordneten) Operationen serialisierbar sind. Damit werden alle Zugriffe direkt ausgeführt, und Transaktionen werden erst *nach* ihrer Ausführung zurückgerollt, wenn festgestellt wird, dass sie keinen serialisierbaren Schedule zur Folge hatten.

Es lässt sich feststellen, dass „generell optimistische Scheduler besser [sind][...], wenn viele der Transaktionen nur lesen, da solche Transaktionen niemals nichtserialisierbares Verhalten verursachen können.“[29, S. 969]

III.3.3 Multiversion-Concurrency-Control

Multiversion-Concurrency-Control (MVCC) ist ein optimistischer Algorithmus zur Nebenläufigkeitskontrolle, der es erlaubt Serialisierbarkeit zu gewährleis-

[3] **Durchsatz:** Anzahl erfolgreich verarbeiteter Transaktionen je Zeiteinheit
Antwortzeit: Zeitspanne zwischen dem Eröffnen einer Transaktion und ihrer erfolgreichen Durchführung, gemessen durch die Anwendung

ten, ohne Lese- oder Schreibzugriffe zu blockieren. Dies wird dadurch ermöglicht, dass jeder Transaktion eine eigene Version der Daten bereitgestellt wird, so dass sie isoliert von anderen Transaktionen arbeiten kann. Zwei Gründe waren für die Auswahl von MVCC entscheidend. Einerseits verwendet die Programmiersprache Clojure[4] bereits erfolgreich MVCC zur Verwaltung geteilter Ressourcen, was ein ähnlicher Anwendungsfall zur SDB ist. Zum Anderen ermöglicht es das Vorhalten von Versionsdaten, sogenannte *Time-Travel-Querys*, also Analysen auf vergangenen Zuständen der Daten, auszuführen. Darauf wird in Kapitel IV genauer eingegangen. Es existieren mehrere Variationen des MVCC. Im Folgenden werden die Grundlagen anhand des Serial Validation Ansatzes [16] vorgestellt (weitere Ansätze können der Literatur, z.B. [4], entnommen werden).

Eine Transaktion wird dabei in 3 Phasen ausgeführt:

1. **Read:** In der Lesephase werden die Lese- und Schreibzugriffe der Transaktion auf einer lokalen Kopie der Daten ausgeführt.
2. **Validation:** In der Validierungsphase wird überprüft, ob das Veröffentlichen dieser Änderungen die Datenbank in einem konsistenten Zustand belässt.
3. **Write:** In der Schreibphase wird die lokale Version global verfügbar gemacht und als aktuelle Version der Daten gekennzeichnet.

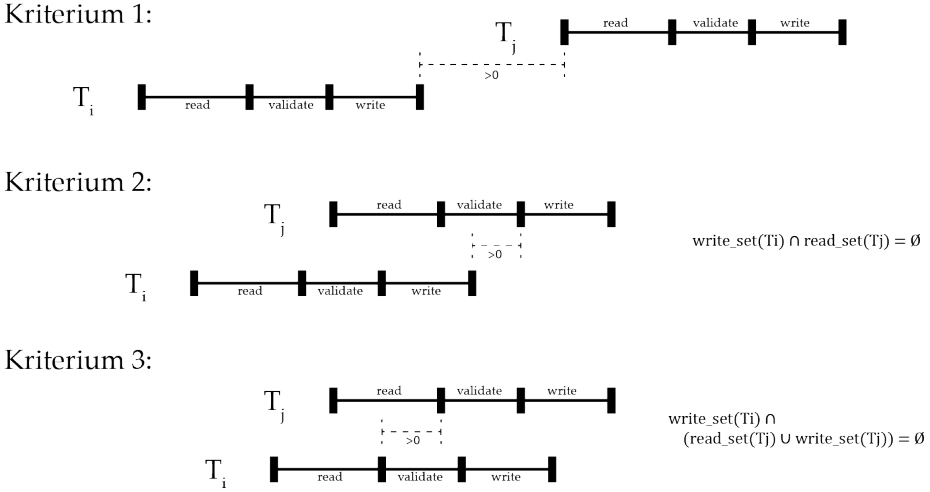


Abb. 23: *Mögliche Abläufe nebenläufiger Transaktionen und Bedingungen ihrer Serialisierbarkeit nach Serial Validation. (Darstellung angelehnt an [16])*

[4] `http://clojure.org/` (Stand: 10.06.2012)

Die `begin` und `commit` Befehle einer Transaktion stellen die Grenzen der Lesephase dar. Bevor die Änderungen jedoch commited werden können, muss in der Validierungsphase überprüft werden, ob die Daten, auf deren Basis die Transaktion gearbeitet hat, noch aktuell sind. Dies ist nicht a priori gewährleistet, da während der Abarbeitung einer Transaktion eine andere Transaktion commiten kann. Diese Änderungen bleiben der ersten Transaktion jedoch verborgen, da sie auf einer lokalen Kopie der Daten arbeitet. Wurde nun ein Datum, auf dessen Basis die Transaktion einen Schreibvorgang durchgeführt hat, verändert, so ist dieser Schreibvorgang nicht mehr konsistent zur aktuellen Version seines Ausgangsdatums, und die lokale Version der Transaktion darf nicht global verfügbar gemacht werden.

Dieser Beschreibung folgend ergeben sich nach [16] drei unabhängige Validierungskriterien, die jeweils ein unterschiedliches Maß an Nebenläufigkeit ermöglichen (siehe Abb. 23). Seien T_i und T_j Transaktionen, wobei i und j Transaktionsnummern sind. Dann muss für T_j und alle T_i mit $i < j$ eines der folgenden Kriterien gelten, damit der entstehende Schedule serialisierbar ist:

1. Die Schreibphase von T_i endet, bevor die Lesephase von T_j beginnt.
2. (a) Die Schreibphase von T_i endet, bevor die Schreibphase von T_j beginnt, und (b) die Änderungen, die T_i geschrieben hat, beeinflussen kein einziges Datum, das T_j gelesen hat.
3. (a) Die Lesephase von T_i endet, bevor die Lesephase von T_j endet, und (b) die Änderungen von T_i dürfen weder Daten beeinflussen, die T_j gelesen, noch geschrieben hat.

Kriterium eins beschreibt einen seriellen Schedule, der per Definition ohne weitere Bedingungen serialisierbar ist, damit aber keinerlei Nebenläufigkeit ermöglicht. Das zweite Kriterium formuliert den Fall, dass Schreibvorgänge der Transaktion T_i nicht dafür sorgen, dass T_j inkonsistent gewordene Daten schreibt (b) und verhindert, dass T_i ein Datum überschreibt, *nachdem* es von T_j geschrieben wurde (a), was auch einen inkonsistenten Endzustand zur Folge hätte. Damit ist Nebenläufigkeit während der Lese- und Validierungsphase möglich. Das letzte Kriterium stellt die härtesten Anforderungen, erlaubt allerdings Nebenläufigkeit während aller drei Phasen. Auf Grund von (b) ist garantiert, dass die Änderungen von T_i und T_j keinerlei Einfluss auf einander nehmen können, während (a) sicherstellt, das der Validierung von T_j alle Änderungen von T_i bekannt sind, so dass (b) überprüft werden kann.

Die Transaktionsnummern i und j müssen eine streng monotone Reihe bilden. Im vorgestellten Fall passiert die Vergabe mit Beginn der Validierungsphase, es sind aber auch andere Zeitpunkte möglich (siehe [16, Abschnitt 3.2]).

III.4 Nebenläufigkeitskontrolle für die SDB

Dieses Kapitel beschreibt den Entwurf des optimistischen Systems zur Nebenläufigkeitskontrolle, dass beim Prototyp der SDB zum Einsatz kommt. Die Zielstellung ist es, konsistente Schreib- und Lesezugriffe bei nebenläufigen Transaktionen zu ermöglichen, die z.B. dann entstehen können, wenn eine Anwendung der SDB mehrere Threads zur selben Zeit nutzt. Gleichzeitig soll der Mehraufwand in Zeit- und Raumkomplexität möglichst gering bleiben.

Das Regelverhalten wird in Abschnitt III.4.1 beschrieben. Es baut auf den Prinzipien der in Abschnitt III.3.3 vorgestellten Serial Validation auf und macht dabei Gebrauch von einigen in [4] beschriebenen Verbesserungsvorschlägen. Im Anschluss wird in Abschnitt III.4.2 der Spezialfall der `HBTM`–Relationen beschrieben und eine Ausnahme vom gängigen MVCC-Prinzip begründet.

III.4.1 Regelfall

Zum Verwalten von Transaktionen besteht threadlokal ein Stack, der gestartete und noch nicht beendete Transaktionen speichert. Das aktuelle System unterstützt zwar keine geschachtelten Transaktionen, ihre Implementierung ist mit dieser Architektur jedoch möglich und wird im Ausblick, Kapitel III.7, beschrieben.

Da die in Abschnitt III.3.1 vorgestellte Transaktionsschnittstelle eingehalten werden sollte, legt der `begin`-Befehl eine neue Transaktion auf den Stack. In ihrem Kontext finden alle Lese- und Schreiboperationen statt, bis entweder die Transaktion durch den `commit`– bzw. `abort`–Befehl abgeschlossen wird, oder bis eine andere Transaktion im selben Thread gestartet wird. Die Lese- und Schreibzugriffe auf Instanzen zu persistierender Klassen sind so modifiziert, dass sie den Index, an dem ihre Operation ausgeführt werden soll, von der aktuellen Transaktion[5] abfragen. Die Versionen unterschiedlicher Transaktionen liegen damit also genauso wie die aktuell gültige Version der Daten in den Arrays der SDB, allerdings an unterschiedlichen Indizes. Eine Transaktion hat also zwei Möglichkeiten den Index für eine Operation zu ermitteln:

1. Sie gibt den **global** aktuellen Index zurück, an dem die aktuell gültige Version der entsprechenden Instanz gespeichert ist. Dies ist der Index, den die letzte erfolgreich commitete Transaktion geschrieben hat, die die entsprechende Instanz modifiziert hat.

[5] Als aktuelle Transaktion wird hier die auf dem Transaktionsstack zuoberst liegende Transaktion bezeichnet.

2. Sie gibt den **lokal** aktuellen Index zurück, an dem eine neue, noch nicht commitete Version der entsprechenden Instanz steht. Dies ist der Index, den die aktuelle Transaktion für die Persistierung von Modifikationen der anfragenden Instanz reserviert hat.

Wird ein Lesezugriff auf einer Instanz ausgeführt, die nicht im Kontext der aktuellen Transaktion modifiziert worden ist, so findet der erste Fall Anwendung. Sobald auf einer Instanz im Kontext der aktuellen Transaktion das erste Mal ein Schreibzugriff ausgeführt wird, reserviert die Transaktion einen neuen Index, der jedoch als invalide gekennzeichnet wird. Alle Werte der Instanz werden vom alten Index an den Neuen kopiert und der neue Index wird zurück gegeben. Für alle weiteren Lese- und Schreibzugriffe macht die Transaktion von Fall zwei Gebrauch, und gibt den neuen Index zurück. Somit muss die Transaktion speichern, welche Instanzen in ihrem Kontext modifiziert wurden, sowie jeweils den neuen Index, an dem die lokale Version in der SDB persistiert ist. Wird der `commit`-Befehl gerufen, so startet die Transaktion ihre Validierungs-

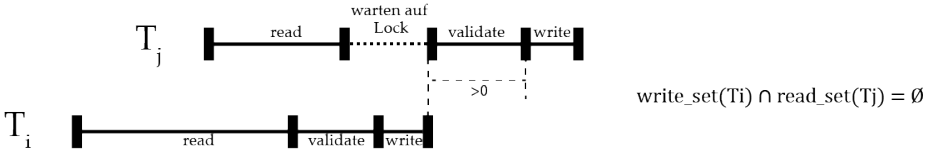


Abb. 24: *Dadurch, dass Validierung und Schreiben atomar ausgeführt werden, ist garantiert, dass eine Transaktion ihren Schreibvorgang abschließt, bevor eine Andere ihren beginnt (Strengeres Kriterium 2). Nebenläufiges Lesen ist trotz dieser Einschränkung möglich.*

phase. Wie in Abschnitt III.3.3 beschrieben, bestehen drei Möglichkeiten der Validierung. Variante 1 ermöglicht keine nebenläufige Ausführung von Transaktionen und ist aus diesem Grund nicht für die SDB geeignet. Variante 3 ermöglicht zwar Nebenläufigkeit in allen drei Phasen der Ausführung, erfordert zur Validierung jedoch die Überprüfung der lokalen Änderungen sowie der gelesenen Daten auf Konflikte mit den lokalen Versionen aller anderen Transaktionen. Dies führt dazu, dass Transaktionen nur in dem Fall valide sind, wenn sie auf Daten gearbeitet haben, die von keiner anderen Transaktion bearbeitet wurden. Mit zunehmender Anzahl an Transaktionen nimmt in diesem Fall also auch die Anzahl an invaliden Transaktionen, die potentiell wiederholt werden müssen, zu. Variante 2 erfordert lediglich die Überprüfung der gelesenen Daten auf Konflikte mit den lokalen Versionen aller anderen Transaktionen, schränkt dieses Problem also ein. Die Überprüfung auf Konflikte der Varianten 2 und 3 würde es erfordern, die Transaktionen unterschiedlicher Threads zentral vorzuhalten, was wiederum Synchronisationsaufwand birg. Außerdem würde es die Dauer der Validierung von der Anzahl der laufenden Transaktionen abhängig machen, was dazu führt, dass das System unter Last zusätzlich verlangsamt wird. Dies lässt sich bei Variante 2 jedoch umgehen, in dem die Neben-

läufigkeitsbedingung, dass Schreibphasen seriell ausgeführt werden müssen, strenger aufgefasst wird: Werden Validierung und Schreiben zusammen atomar ausgeführt, entfällt der Bedarf die lokalen Änderungen aller Threads zu untersuchen, und es muss nur mit der global gültigen Version der Daten verglichen werden. Dies gilt, da auf Grund dieser seriellen Ausführung bereits zu Anfang der Validierung fest steht, dass alle anderen aktiven Transaktionen per Definition ihre Schreibphase nicht abschließen können, bevor nicht die aktuell validierende Transaktion ihre Schreibphase abgeschlossen hat. Somit müssen lediglich die Änderungen der Transaktionen betrachtet werden, die ihr Schreiben im Verlauf der Lesephase der validierenden Transaktion abgeschlossen haben. Dies bedeutet aber zwangsläufig, dass sie nicht mehr aktiv sind, und das sich ihre Änderungen bereits in der global gültigen Version widerspiegeln. Die strengere Auslegung der Nebenläufigkeitsbedingung sorgt dafür, dass im Vergleich zu Variante 2 nur noch Lesephasen nebenläufig ausgeführt werden können (siehe Abb. 24). Dies ist jedoch akzeptabel, wenn die Annahme gilt, dass Lesephasen die längsten Phase einer Transaktion sind. Sie erscheint plausibel, wenn bedacht wird, dass eine Anwendungen zwischen Lese- und Schreiboperationen zumeist noch Berechnungen durchführt, und dass die Validierung mit diesem Ansatz lediglich darin besteht, zu überprüfen, ob die gelesenen Daten einer Transaktion global noch aktuell sind (Messungen, die diese Annahme werden in Abschnitt III.6.2 diskutiert). Konkret bedeutet dies, dass eine Transaktion zum Validieren lediglich einen globalen Commit-Lock anfordert, und die Aktualität der von ihr gelesenen Daten sicherstellt, in dem sie die Validität aller von ihre gelesenen Indizes prüft.

Ist eine Transaktion erfolgreich validiert, so setzt sie alle von ihr reservierten Indizes auf valide und invalidiert die veralteten Indizes. Danach gibt sie den Commit-Lock wieder frei. Konnte die Transaktion nicht validiert werden, so wird sie abgebrochen und der Lock auch freigegeben. Die geschriebenen Daten verbleiben invalidiert an ihrem Index[6]. Es besteht die Möglichkeit die Transaktion erneut aus zu führen, wobei die Operationen in ihrem Kontext in der korrekten Reihenfolge solange nach dem hier beschriebenen Prinzip ausgeführt werden, bis die Transaktion commiten kann. Um dabei das Verhungern von großen Transaktionen zu vermeiden ist es denkbar, die Anzahl der Wiederholungen zu speichern und bei Überschreiten einer bestimmten Grenze den Commit-Lock für die Dauer der gesamten Transaktion anzufordern.

[6] Diese invaliden Indizes können zum Beispiel von einer Freispeicherliste verwaltet und zum Überschreiben freigegeben werden. Der erhöhte Speicherverbrauch durch invalide Daten bleibt damit gering.

III.4.2 Ausnahme für `HBTM`–Assoziationen

Das in Abschnitt III.4.1 beschriebene Verhalten findet bei allen Instanzvariablen Anwendung, die primitive Datentypen oder `One`– bzw. `Many`–Assoziationen speichern. `HasAndBelongsToMany`–Assoziationen werden im Gegensatz zu diesen jedoch, wie in Abschnitt III.2.2 beschrieben, nicht in Form von (Fremdschlüssel-) Spalten, sondern als Join-Tabellen gespeichert. Nach dem MVCC-Prinzip aus Abschnitt III.3.3 würde jede Transaktion beim ersten Schreibzugriff auf den HBTM-Assoziationen die gesamten bestehenden ID-Paare des bearbeiteten Objekts kopieren und im Anschluss ihre Änderungen auf den Kopien durchführen. Dieses Verfahren hätte einen hohen Speicherbedarf zur Folge, was vermieden werden kann.

Im Gegensatz zu den anderen Attributen speichert eine `HBTM`–Variable eine Menge und keinen einzelnen Wert[7]. Ihre Einträge können nicht verändert werden, es können lediglich Elemente hinzugefügt oder gelöscht werden. Dadurch ist es nicht mehr notwendig für jede Transaktion eine Kopie der bestehenden Elemente zum Bearbeiten anzulegen. Das Hinzufügen eines Elements kann realisiert werden, in dem die Transaktion ein ID-Paar lokal erstellt, und in der Schreibphase validiert, womit es global sichtbar wird. Das Löschen eines Elements wird dadurch umgesetzt, dass die Transaktion in der Schreibphase dieses ID-Paar invalidiert. In der Validierungsphase muss wie im Regelfall überprüft werden, ob die Menge der Elemente einer `HBTM`-Assoziationen die selbe geblieben ist, wenn eine Lesezugriff auf sie ausgeführt worden ist.

III.5 Umsetzung

In diesem Kapitel wird die Implementierung des in Kapitel III.4 vorgestellten Entwurfs vorgestellt. Die Nebenläufigkeitskontrolle ist im Modul `transaction.py` umgesetzt. Zuerst wird in Abschnitt III.5.1 die Schnittstelle vorgestellt, die das Transaktionsmodul Anwendungsprogrammierern zur Verfügung stellt. Der Abschnitt III.5.2 stellt im Anschluss vor, wie diese Schnittstelle implementiert ist. In Abschnitt III.5.3 wird gezeigt, wie sich das Modul in die Gesamtarchitektur der SDB eingliedert. Die Anpassungen, die nötig waren um das Transaktions-Modul in die SDB einzubauen, werden in III.5.4 vorgestellt.

[7] Auch eine `Many`–Assoziation speichert eine Menge. Dies wird jedoch durch einen Fremdschlüssel der Zielobjekte umgesetzt (siehe III.2.1). Defacto ist die `HBTM`–Assoziation damit die einzige Instanzvariable der SDB, die eine Menge beinhaltet.

III.5.1 Schnittstelle

Den ersten Einstiegspunkt für einen Anwendungsprogrammierer bietet die Klasse `Transaction`. Sie stellt die Methoden der Transaktionsschnittstelle `Transactions.begin()`, `Transactions.commit()` und `Transactions.abort()` bereit, die eine neue Transaktion starten bzw. die aktuelle Transaktion beenden. Mit Hilfe der Methode `Transactions.active()` kann der Anwendungsprogrammierer die aktuelle Transaktion, eine Instanz der Klasse `Transaction`, erhalten. Diese bietet die Methoden `mark_as_abort()` und `mark_as_commit()` an, die die Implementierung eigener Validierungen ermöglichen, indem sie festlegen, ob eine Transaktion beim commit invalidiert wird.

Zur einfacheren Bedienung stellt das Transaktionsmodul außerdem die zwei Methodendekoratoren `@transactional` und `@retrying` bereit, die die dekorierte Methode mit einer Transaktion unschließen (siehe Quelltext 6). Erstere versucht die Transaktion durch zu führen und bricht ab, falls die Validierung fehlschlägt. Zweitere dagegen wiederholt die Transaktion solange, bis sie erfolgreich abgeschlossen werden kann.

Außerhalb eines Transaktionskontextes ist es nicht möglich, persistierte Attribute zu lesen oder zu schreiben, Attributaufrufe verursachen in diesem Fall eine `TransactionException`.

```
1  @column_layout
2  class Person(object):
3      age = int
4      car = One(Vehicle)
5
6      # method will be executed until it succeeds
7      @retrying
8      def mature(self, years):
9          self.age += years
10
11 class AgingCentre(object):
12     def treatPerson(self, person):
13         # transactional call of decorated method
14         person.mature(7)
```

Quelltext 6: *Die Methode mature wird transaktional ausgeführt, ohne dass `Transactions.begin()` und `Transactions.commit()` gerufen werden muss. Sie wird nach dem Aufrufen solange wiederholt, bis sie erfolgreich abgeschlossen werden kann.*

III.5.2 Transaktions-Modul

Die erste Aufgabe des Transaktionsmoduls ist es, unterschiedliche Transaktionen zu verwalten. Dazu wird die Liste `context.transactions` verwendet, die für jeden Thread den Stack der gerade aktiven Transaktionen bereithält. Die statische Methode `begin()` der Transaktions-Klasse fügt an das Ende der transactions-Liste neue Instanzen hinzu, während die statischen Methoden `commit()` und `abort()` das letzte Element der Liste entfernen. Nach dem Entfernen initiieren die Methoden den eigentlichen Commit- bzw. Abort-Vorgang dieser Transaktion. Initial enthält `transactions` nur eine Instanz der Klasse `NullTransaction`, die von `Transaction` erbt. Diese sorgt dafür, dass Operationen, die nur in einem transaktionalen Kontext erlaubt sind, nicht ausgeführt werden. Dazu wirft sie eine `TransactionException`, deren Fehlermeldung über die verbotene Operation aufklärt. Die `NullTransaction` bleibt immer in der Liste, da sie nicht durch commit oder abort entfernt werden kann.

Die zweite Aufgabe des Moduls ist es Lese- und Schreibzugriffe zu verwalten und Daten vorzuhalten, die das Validieren und Veröffentlichen dieser Operationen ermöglichen. Dazu enthält die Klasse `Transaction` drei Zwischenspeicher, in denen alle Operationen der Transaktion protokolliert werden. Der `read_cache` speichert alle Lesevorgänge. Dies ist nötig, weil die Transaktion dadurch validiert wird, dass überprüft wird, ob die von ihr gelesenen Daten sich verändert haben. Im `write_cache` werden alle Schreibvorgänge gespeichert, die Daten hinzufügen oder verändern, und im `delete_cache` werden diejenigen Schreibvorgänge festgehalten, die Daten löschen. Nötig ist dies, da die veränderten Daten bei einem erfolgreichen commit als valide gekennzeichnet werden müssen. Die Trennung in write und delete wurde gewählt, weil dies eine einfachere Bearbeitung von Anfragen ermöglicht, die auf Mengen operieren (genaueres siehe Abschnitt III.5.4).

Die Zwischenspeicher sind als Dictionaries implementiert, die jeweils von verarbeiteter Entität auf eine `CachedEntity` abbilden. Wie in Abschnitt III.4.2 ausgeführt, muss dabei auf die Unterschiede zwischen dem MVCC-Regel-verhalten, bei Operationen auf Instanzen, und dem Ausnahmefall, bei Operationen auf `HBTM`–Assoziationen, unterschieden werden. Entsprechend werden in den Caches Instanzen einer Unterklasse von `CachedEntity` gespeichert (siehe Abb. 25). Dadurch, dass jeder Entität eine entsprechende Unterklasse zugeordnet ist, kann das unterschiedliche Verhalten sowohl beim Validieren als auch beim Veröffentlichen so gekapselt werden.

Regelfall Im Regelfall bilden die Caches von Instanzen (die als ID-Klassen Tupel repräsentiert werden) auf `CachedObject`s ab. Diese speichern die ID, die Klasse und den Index der verarbeiteten Instanz. Bei Änderungen speichern

sie außerdem den neuen Index, an dem die lokale Kopie gespeichert ist. Um `CachedObject`s zu erzeugen, bietet eine Transaktion folgende Methoden:

- `add_object(self, cls, instance)`
- `delete_object(self, cls, id)`
- `read_index(self, cls, id)`
- `write_index(self, cls, id)`

Die Methode `add_object` dient dazu, das Anlegen neuer Instanzen fertig zu stellen. Dazu stattet sie die ihr übergebene Instanz mit einem freien Index aus und legt einen entsprechende Instanz der Klasse `TouchedObject` im `write_cache` ab. `Delete_object` ist dafür verantwortlich, Instanzen zu löschen (bzw. im Fall des SDB-Prototypen zu invalidieren). Dazu wird eine Instanz der Klasse `DeletedObject` im `delete_cache` abgelegt.

Soll ein Attribut einer bereits bestehenden Instanz gelesen werden, so gibt die Methode `read_index` den aktuellen Index der Instanz zurück. Dazu überprüft sie zuerst, ob die Instanz im aktuellen Kontext schon einmal verändert wurde, in dem sie nach einem passenden Eintrag im `write_cache` sucht. Wird sie fündig, gibt sie den neuen Index, der im `TouchedObject` gespeichert ist, zurück. Falls nicht, überprüft sie, ob die Instanz schon gelesen wurde, indem sie nach einem Eintrag im `read_cache` sucht. Bei Auffinden gibt sie den in diesem Eintrag gespeicherten Index zurück. Schlägt auch dies fehl, so gibt die Methode den Index der global aktuellen Version zurück und legt eine neue Instanz der Klasse `TouchedObject` im `read_cache` an. Analog dazu dient die Methode `write_index` dazu, den Index der lokalen Version zu ermitteln, an dem Modifikationen geschrieben werden. Dazu prüft sie, ob die zu ändernde Instanz im Kontext der aktuellen Transaktion schon einmal bearbeitet wurde, indem sie nach einem passenden Eintrag im `write_cache` sucht. Im positiven Fall gibt sie den neuen Index aus dem passenden `TouchedObject` zurück. Findet sie keinen Eintrag, reserviert sie einen neuen Index und kopiert die Daten aus den `Column`s, die zur Klasse der modifizierten Instanz gehören, vom Alten in den neuen Index. Im Anschluss legt sie ein neues `TouchedObject` im `write_cache` an, und gibt den neuen Index zurück.

Ausnahmefall Im Ausnahmefall bilden die Caches von Index-`DoubleColumn` Tupel auf `CachedHbtmEntry`s ab. Diese speichern den Index und die Double-Column, in der der Eintrag auf dem operiert wird, liegt. Zum Anlegen von `CachedHbtmEntry`s bietet eine Transaktion folgende Methoden:

- `add_hbtm(self, index, double_col)`
- `delete_hbtm(self, index, double_col)`
- `read_hbtm(self, index, double_col)`

Diese drei Methoden müssen dem Rufenden keine Indizes zurück geben, da im Ausnahmefall kein MVCC nötig ist. Sie dienen also lediglich dazu, Informationen zum Validieren von Lesevorgänge und zum Commiten von Schreibvorgänge festzuhalten. Dem entsprechend fügt `add_hbtm` ein `TouchedHbtmEntry` in den `write_cache` ein. Analog dazu fügt die Methode `delete_hbtm` ein `DeletedHbtmEntry` in den `delete_cache` ein. Bevor `read_hbtm` ein `TouchedHbtmEntry` in den `read_cache` einfügt, überprüft es ob ein passender Eintrag dort oder im `write_cache` bereits vorliegt. Letzteres sorgt dafür, dass bei der Validierung Lesevorgänge auf neu angelegten Einträgen nicht auf ihre Gültigkeit überprüft werden müssen. Da diese nur lokal vorliegen, können sie nicht von anderen Transaktionen verändert werden, weshalb eine Validierung in ihrem Fall nicht nötig ist.

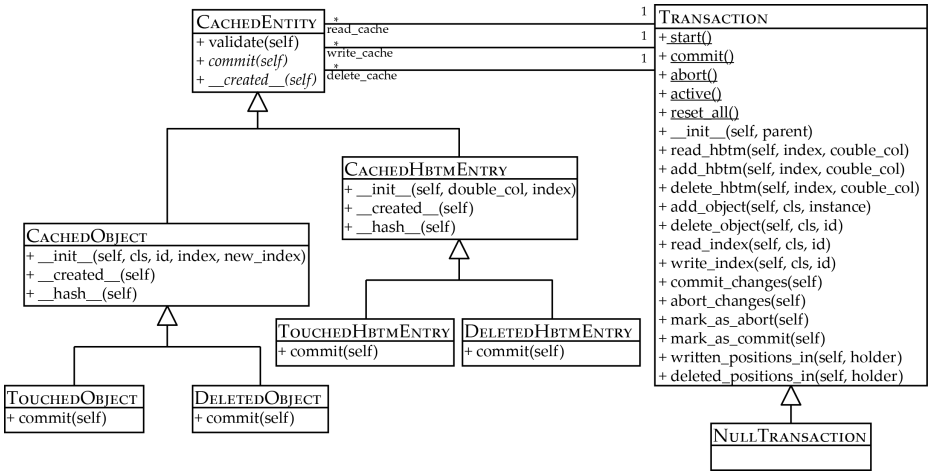


Abb. 25: *Klassendiagramm des Transaktions-Moduls.*

Validierung und Schreiben von Transaktionen Soll eine Transaktion commited werden, so fordert sie den `global_commit_lock` des Transaktionsmoduls an, und leitet die Validierungs- und Schreibphase ein, indem sie auf der aktiven Transaktion die Methode `commit_changes()` aufruft. Diese überprüft, ob die Transaktion vom Nutzer durch einen Aufruf von `mark_as_abort` invalidiert wurde, und wirft in diesem Fall eine entsprechende `InvalidTransactionException`. Ist dies nicht der Fall, so geht sie alle Einträge im `read_cache` durch, und lässt sie sich selbst validieren, in dem sie die Methode `validate` auf ihnen aufruft. Die Einträge überprüfen, ob sie noch aktuell sind und werfen eine `ConcurrentModificationException` wenn dem nicht so ist.

Ist die Validierung fehlerfrei abgeschlossen, so geht die `commit_changes` Methode zuerst den `write_cache` und dann den `delete_cache` durch, und ruft auf den Einträgen `commit` auf, damit diese sich selbst schreiben.

Dabei ist es wichtig, dass der `delete_cache` als Letztes durchlaufen wird, damit er Einträge die vom `write_cache` geschrieben wurden wieder löschen kann, falls sie innerhalb einer Transaktion sowohl verändert als auch gelöscht wurden.

In jedem Fall wird den Lock wieder freigegeben, nach dem die Methode ihre Arbeit abgeschlossen hat.

Methodendekoratoren Die Dekoratoren `@transactional` und `@retrying` sorgen dafür, dass die dekorierte Methode innerhalb einer Transaktion ausgeführt wird. Sie sind als Methoden implementiert, die ein Methoden-Objekt entgegen nehmen, eine neue Methode definieren, und diese zurück geben (siehe Quelltext 7). Der erste Dekorator führt in dieser Methode lediglich `Transaction.begin()`, die übergebene Methode, sowie `Transaction.commit()` aus, und gibt das Ergebnis der ausgeführten Methode zurück. Der Zweite verhält sich ähnlich, fängt jedoch eine eventuell von der erzeugten Transaktion geworfene `ConcurrentModificationException` ab, und führt in diesem Fall die neu definierte Methode rekursiv aus. Dies hat zur Folge, dass die dekorierte Methode solange ausgeführt wird, bis sie commiten kann.

```
1  def transactional(transaction):
2      def trans_func(*args, **kw):
3          Transaction.start()
4          value = transaction(*args, **kw)
5          Transaction.commit()
6          return value
7      return trans_func
8
9  def retrying(transaction):
10     def trans_func(*args, **kw):
11         Transaction.start()
12         value = transaction(*args, **kw)
13         try:
14             Transaction.commit()
15         except ConcurrentModificationException:
16             trans_func(*args, **kw)
17         return value
18     return trans_func
```

Quelltext 7: *Implementierung der Dekoratormethoden `@transactional` und `@retrying`.*

III.5.3 Architektur

Das Transaktionsmodul wird von mehreren anderen Modulen eingebunden und hat im Gegenzug selber eine Abhängigkeit. Benötigt wird es von den Modulen `primitives.py` und `associations.py`, die das Transaktionsmodul verwenden, um die Indizes zu erfragen, an denen sie Lese- und Schreibvorgänge durchführen können. Außerdem wird es von dem Modul `column-layout.py` eingebunden, das dem Endnutzer Methoden des Transaktionsmoduls zur Verfügung stellt. Die einzige Abhängigkeit des Transaktionsmoduls ist `clay_errors.py`, ein Modul, das alle neu definierten Exceptions der SDB enthält.

III.5.4 Anpassung der SDB

Der aktuelle Index von Objekten wird vom Transaktionsmodul verwaltet. Das bedeutet, dass Getter- und Settermethoden für Attribute derart angepasst werden mussten, dass sie die Methoden `read_index` und `write_index` verwenden, um den Index zu ermitteln, an dem ihre Operationen durchzuführen sind.

Dadurch, dass Änderungen von Daten-Attributen in den Caches einer Transaktion gespeichert werden, ist die Verwaltung der Speicherorte für aktuelle Versionen nicht mehr einheitlich. Den Index von Datensätze, die nicht im Kontext der aktiven Transaktion bearbeitet wurden hält eine Indexstruktur bereit, während der Index von veränderten Datensätzen von der Transaktion zu erfragen ist. Dies stellt kein Problem dar, wenn auf einem konkreten Datensatz operiert werden soll, da die Transaktionsklasse Methoden zur Verfügung stellt um den richtigen Index zu ermitteln. Es existieren allerdings Methoden in der SDB, die *alle* Einträge eine Art ermitteln müssen. So dient die Methode `all` einer persistierten Klasse zum Beispiel dazu, alle Instanzen dieser Klasse zurück zu geben, und der Getter einer `Many`–Assoziation muss alle Instanzen der referenzierten Klasse finden, deren Fremdschlüssel auf das referenzierende Objekt verweist. Diese Methoden mussten derart angepasst werden, dass sie zuerst (wie zuvor) den in der Datenbank global gültigen Zustand ermitteln, dann aber die Änderungen einbeziehen, die lokal in der aktiven Transaktion vorgenommen wurden. So muss die `all`–Methode die IDs aller Instanzen ihrer Klasse in dem global gültigen Zustand der SDB ermitteln, in dem sie über die Indexstruktur iteriert. Dann muss sie alle Instanzen finden, die in der aktiven Transaktion angelegt wurden, und deshalb im globalen Zustand noch nicht sichtbar sind. Dazu muss sie die Einträge im `write_cache` untersuchen. Im Anschluss muss sie die IDs derjenigen Instanzen finden, die in der aktuellen Transaktion gelöscht wurden, weil diese nicht mehr von `all` zurückgegeben werden dürfen. Dazu muss sie den `delete_cache` untersuchen (siehe Abb. 8). Analog dazu muss in allen Methoden verfahren werden, die eine Menge verwalten.

Um dies zu ermöglichen stellt die Transaktionsklasse die Methoden `written_positions_in(self, holder)` und `deleted_positions_in(self, holder)` bereit. Diese iterieren über den `write_` bzw. `delete_cache` und geben die Identifikatoren der Einträge zurück, die von `holder` verwaltet werden. Im Regelfall werden also die IDs der Objekte zurückgegeben, die Instanzen der Klasse holder sind und in der aktuellen Transaktion verändert bzw. gelöscht wurden. Im Ausnahmefall werden die Indizes der Einträge zurückgegeben, die in der `DoubleColumn` holder gespeichert sind und sich im entsprechenden Cache befinden.

```
1  def all(cls):
2      ids = set([])
3      for each in cls.__table__._id_index:
4          ids.add(cls.as_id(each))
5
6      for each_id in  Transaction.active(). \
7                          written_positions_for(cls):
8          ids.add(each_id)
9
10     for each_id in  Transaction.active(). \
11                         deleted_positions_for(cls):
12         ids -= set([each_id])
13
14     for each in ids:
15         yield cls.at(each)
```

Quelltext 8: *Die `all`–Methode gibt alle Instanzen einer Klasse zurück. Dazu ermittelt sie Instanz-IDs im globalen Zustand der DB, vereinigt diese Menge mit den IDs von Instanzen, die in der aktiven Transaktion neu angelegt wurden und zieht die Menge der IDs ab, die gelöscht wurden.*

III.6 Auswertung

Im folgenden Kapitel wird die Implementierung der Nebenläufigkeitkontrolle für die SDB aus drei Perspektiven betrachtet. Zuerst wird in Abschnitt III.6.1 anhand klassischer Nebenläufigkeitsprobleme überprüft, ob die implementierten Konzepte funktionieren. Um zu untersuchen, welchen Overhead die Nebenläufigkeitskontrolle verursacht, werden in Abschnitt III.6.2 für zwei Beispieltransaktionen die Laufzeit der SDB mit MVCC mit einer älteren Version ohne Nebenläufigkeitskontrolle verglichen. Zuletzt werden in Abschnitt III.6.3 die Änderungen diskutiert, die die Verwaltung von Nebenläufigkeit in die Syntax der SDB eingebracht hat.

III.6.1 Nebenläufigkeit

Um zu zeigen, dass der vorgestellte Algorithmus zur Kontrolle von Nebenläufigkeit in der Lage ist, wird argumentiert, dass die archetypischen Nebenläufigkeitsprobleme [30, S. 62–64] gelöst werden. Dies ist ein erster Schritt. Um die Argumentation zu stützen böte sich außerdem der Test im Produktivbetrieb sowie formale Betrachtungen an, die jedoch den Rahmen dieser Arbeit sprengen würden.

1. **Lost Update:** Das Lost Update Problem besteht darin, dass zwei Transaktionen zeitgleich den selben Wert einlesen und auf ihm operieren. Im Anschluss schreiben beide den veränderten Wert zurück, wobei die Transaktion die zuletzt schreibt die Änderung der Ersten überschreibt. Dies ist problematisch, da damit die Änderung der ersten Transaktion verloren geht, was das Durability-Kriterium verletzt. Im vorliegenden Prototypen tritt das Problem nicht auf, da die zweite Transaktion im Verlauf ihrer Validierung feststellen würde, dass der Wert den sie eingelesen hat nicht mehr mit dem global aktuellen Wert übereinstimmt, worauf hin das Schreiben nicht gestattet werden würde.
2. **Phantom Read:** Dieses Problem zeichnet sich dadurch aus, dass eine Transaktion einen temporär inkonsistenten Stand einließt und auf ihm operiert. Dieser Stand entsteht, weil ihr Lesevorgang zu einem Zeitpunkt statt findet, an dem eine andere Transaktion lediglich einen Teil ihrer Änderungen geschrieben hat. Dies verletzt das Consistency- und Atomicity-Kriterium. Der vorgestellte Algorithmus vermeidet dieses Problem, in dem jede Transaktion ihre Schreibvorgänge auf lokalen Versionen ausführt, und diese nur atomar global abrufbar macht.
3. **Dirty Read:** Ein Dirty Read Problem entsteht, wenn eine Transaktion Änderungen liest, die eine zweite aktive Transaktion geschrieben hat. Wird die zweite Transaktion zurück gerollt, so operiert die Erste auf Daten, die nicht mehr in der Datenbank gültig sind. Dies verletzt auch die Consistency- und Isolation-Kriterien. Die vorliegende Implementierung umgeht dieses Problem, da das globale Veröffentlichen der Daten erst nach der Validierung geschieht. Ein Rollback kann nie nach der Schreibphase auftreten, weshalb auch nie Daten gelesen werden können, die ein Rollback verändern würde.
4. **Non-repeatable Read** Führt eine Transaktion zwei mal den selben Lesevorgang aus und bekommt dabei unterschiedliche Ergebnisse, weil zwischenzeitlich das entsprechende Datum verändert wurde, so wird dies Non-reapeatable Read genannt. Dies ist problematisch, da die Transaktion nicht mehr auf konsistenten Daten operiert. Es verletzt sowohl das Isolation-, als auch das Consistency Kriterium. Der vorgestellte Prototyp löst das Problem dadurch, dass der Index eines Lesevorgangs beim ersten Mal zwischengespeichert, im Anschluss aus dem Zwischenspeicher reproduziert wird.

Damit ist ersichtlich, dass die durchgeführte MVCC-Implementierung zumindest grundsätzliche Probleme der Nebenläufigkeit verhindert.

III.6.2 Änderung der Laufzeiteigenschaften

Um die vorgestellte Implementierung beurteilen zu können ist es hilfreich einschätzen zu können, wie sie das Laufzeitverhalten der SDB beeinflusst. Dazu wurde ein stabiler Stand mit Nebenläufigkeitskontrolle mit einem stabilen Stand vor ihrer Implementierung verglichen. Zum Vergleich wurde die Dauer der Ausführung jeweils nur schreibender und nur lesender Anfragen gemessen. Bei der Implementierung mit MVCC wurden die Anfragen in jeweils einer Transaktion durchgeführt. Die Auswirkung einer nebenläufigen Ausführung der Anfragen in mehreren Transaktionen wurde also nicht untersucht, da der Fokus dieser Auswertung auf der Implementierung von Nebenläufigkeitskontrolle liegt, und nicht auf den potentiellen Vorteilen von Nebenläufigkeit. Es ist ersichtlich, dass die Kontrolle von Nebenläufigkeit einen Mehraufwand im Vergleich zum sequentiellen System bedeutet. Deshalb ist vor allem interessant, in welcher Größenordnung der zusätzliche Aufwand liegt, und wie er skaliert. Aus diesem Grund wurde die Auswertung auf Datenbasen unterschiedlicher Größe durchgeführt.

Für eine vollständige Auswertung müsste der Vergleich mit anderen Systemen zur Nebenläufigkeitskontrolle erfolgen, deren Implementierung jedoch zeitlich nicht im Rahmen des Projekts durchzuführen war.

Testsystem Die Auswertung erfolgte auf einem Intel Core i5 (4 Kerne, 2.67 GHz), mit 6 GB Arbeitsspeicher. Als Betriebssystem wurde ein 64-bit Windows 7 Enterprise SP1 und als Interpreter ein 32-bit PyPy 1.9[8] „JIT Compiler“ Version eingesetzt.

Anfragen Als Grundlage für die beiden Testanfragen wurde eine einfache Klassenstruktur aus der Businessdomäne verwendet. Das Modell enthält die vier Klassen Produkt, Bestellung, Bestellungsdetail und Kunde. Eine Bestellung besteht aus einem Datum und mehreren Bestellungsdetails. Ein Bestellungsdetaill enthält ein Produkt sowie die bestellte Anzahl, und das Produkt hat einen Preis und einen Namen. Ein Kunde speichert mehrere Bestellungen.

[8] Anlass für den Einsatz einer 32-bit Ausführung ist das Fehlen einer 64-bit Version von PyPy für Windows.

Die erste Anfrage erstellt eine bestimmte Anzahl zufällig gefüllter und verbundener Instanzen dieser Klassen. Die **Größe der Testdaten n** wird im Folgenden angegeben als die Anzahl der Bestellungsdetails. Es werden bei der Anfrage außerdem $n/10$ Bestellungen, $n/100$ Kunden und $n/1000$ Produkte angelegt. Die zweite Anfrage führt eine Analyse auf den entstandenen Daten aus und ermittelt den Umsatz je Produkt und Jahr, für Bestellungsdetails mit nur einem Produkt. Dazu gruppiert sie alle Bestellungsdetails, bei denen genau ein Produkt erworben wurde, nach dem Jahr ihrer Bestellung und dem Namen ihres Produkts. Für jede dieser Gruppen wird der Gesamtpreis der Produkte ermittelt.

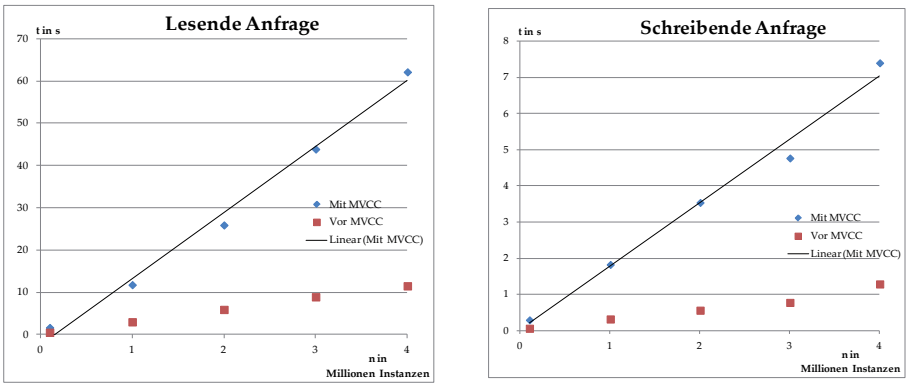


Abb. 26: *Dauer der Abarbeitung der lesenden (links) und schreibenden Anfrage (rechts) in Abhängigkeit von der Größe des Objektmodels.*

Dauer der Anfragen Abb. 26 zeigt die **Dauer t** der gesamten Abarbeitung der lesenden und schreibenden Anfragen in Sekunden, die im Fall mit Nebenläufigkeitskontrolle auch den Start und Commit der sie umfassenden Transaktion enthält. Die Messung erfolgte mit der Funktion `time` des Python Moduls `time`. Angegeben ist jeweils die durchschnittliche Dauer aus 10 Messungen je Datenpunkt. Die Begrenzung der Messung auf n = 4 Millionen ist damit begründet, das 5 Millionen Instanzen auf der Testmaschine die Hauptspeicherbegrenzung des PyPy-Interpreters überschreitet.

Aus den Messungen ist eindeutig ersichtlich, dass die Kontrolle von Nebenläufigkeit sowohl bei lese- als auch bei schreiblastigen Anfragen einen deutlichen Mehraufwand bedeutet. Außerdem lässt sich feststellen, dass die Dauer der Verarbeitung in beiden Fällen mit zunehmender Größe der Datenbasis bei der Version mit Nebenläufigkeitskontrolle schneller wächst als in der ohne. Aus den vorliegenden Messwerten lässt sich vermuten, jedoch nicht belegen, dass dieses Skalieren aber immer noch linear zur Größe der Datenbasis geschieht (siehe lineare Trendlinien in Abbildung 26).

Die Analyse der Ausführung mit einem Profiler ergab, dass der Mehraufwand bei den Leseanfragen zu einem großen Teil auf die Indirektion über das Transaktionsmodul und die Aufspaltung des aktuellen Zustands (wie z.B. bei der `all`–Methode) zurückzuführen ist. Insbesondere das Nachschlagen im `read_cache` verursachte zusätzliche Kosten. Bei der Schreibanfrage ergab sich der zusätzliche Aufwand zum Großteil durch das Invalidieren alter und das Validieren neuer Versionen.

Damit lässt sich formulieren, dass zumindest beim Lesen der Hauptaufwand in der Verwaltung der Speicherorte für aktuelle Versionen liegt. Die vorgestellte Implementierung schafft es also nicht, das Potential optimistischer Nebenläufigkeit ganz aus zu nutzen, das es ermöglichen würde, Transaktionen, die nur Lesevorgänge ausführen, kaum zu verlangsamen. Grund dafür ist, dass keine konsistente lokale Version der gesamten Daten vorgehalten wird und diese statt dessen beim Lesen nach und nach angelegt wird. Dies sorgt dafür, dass auch Transaktionen die nur Lesen validiert werden müssen, um eventuelle Inkonsistenzen zwischen gelesenen Versionen auf zu decken.

Dauer der Phasen In Abschnitt III.4.1 wurde die Annahme formuliert, dass Lesephasen die längsten Phasen eine Transaktion sind. Darauf aufbauend wurden die Validierungskriterien so gewählt, dass lediglich die Lesephase nebenläufig ausgeführt werden kann. Zur Untersuchung dieser grundlegenden Annahme wurden die beiden bereits vorgestellten Anfragen, sowie eine kombinierte Lese- und Schreibanfrage mit jeweils n = 1 Million verwendet. Eine Anfrage die nur liest muss alle Operationen validieren, hat also die längst-mögliche Validierungsphase. Eine Anfrage die nur schreibt muss alle Operationen schreiben, hat also die längst-mögliche Schreibphase. Für die kombinierte Anfrage wurde zuerst die Hälfte des Objekmodells angelegt. Dann wurde in einer einzigen Transaktion die andere Hälfte erstellt und die analytische Anfrage durchgeführt. Dieses Vorgehen dient dazu, das Verhalten eine bereits gefüllte Datenbank zu simulieren.

Die ersten beiden Anfragen stellen Extremfälle, und die Letzte einen durchschnittlichen Fall dar.

Aus den in Abb. 27 dargestellten Daten ist ersichtlich, dass in allen drei untersuchten Anfragen die Lesephase die längste war. Die Dauer der Validierung ist in allen Fällen verhältnismäßig kurz. Die Schreibphase nimmt im Extremfall der nur schreibenden Transaktion etwa einen Drittel der Gesamtdauer ein. Diese Messwerte stützen also die vorgestellte Annahme.

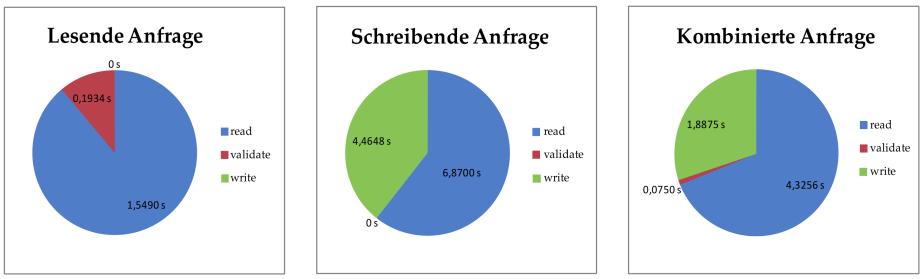


Abb. 27: *Dauer der Abarbeitung der drei Testanfragen aufgeschlüsselt nach den einzelnen Phasen, für* $n = 1000000$.

III.6.3 Änderungen in der Sprachnutzung

Für den Endanwender bedeutet das Einbringen von Nebenläufigkeitskontrolle effektiv drei Änderungen.

1. Für alle Operationen auf persistierten Attributen müssen Transaktionsgrenzen festgelegt werden.
2. Außerhalb dieser Transaktionsgrenzen dürfen weder Lese- noch Schreibzugriffe durchgeführt werden.
3. Fehlerbehandlung muss für den Fall scheiternder Transaktionen implementiert werden.

Die Definition der Transaktionsgrenzen kann mithilfe der Dekoratoren des Transaktions-Moduls an Methoden ausgerichtet werden. Dies ist eine leichtgewichtige Lösung, die lediglich den Gebrauch von einem Schlüsselwort erfordert. Dadurch, dass die Transaktion mit der Methode abschließt, ist keine explizite Überlegung darüber notwendig, wann sie zu starten und wann zu commiten ist. Allerdings bleibt dem Anwender die Entscheidung überlassen, welche Granularität für die Transaktionen gewählt wird. Wird zum Beispiel die Startmethode eines Threads annotiert, so bedeutet dies für den Nutzer den wenigsten Aufwand, da keine weiteren Annotationen nötig sind, und wie gewohnt programmiert werden kann. Dies hat jedoch negative Auswirkungen auf die Parallelisierbarkeit des Codes, da aufgrund der Länge und der Anzahl der Änderungen die innerhalb eines solchen Kontexts auftreten, die Wahrscheinlichkeit des Scheiterns einer solchen Transaktion hoch ist.

Der Fakt, dass auch Lesezugriffe auf persistierte Daten innerhalb einer Transaktion zu geschehen haben, ist für Anwendungsprogrammierer ungewöhnlich. Es muss jedoch unter allen Umständen garantiert werden, dass gelesene Daten konsistent sind. Der Umgang mit Fehlern bei nebenläufiger Ausführung ist ein Problem, das die meisten Systeme zur Nebenläufigkeitskontrolle dem Anwender überlassen. Die Möglichkeit Transaktionen mit Hilfe des `@retrying-`

Dekorators automatisiert mehrmals ausführen zu lassen, ist jedoch ein erster Ansatz, um den Endnutzer dabei zu entlasten.

III.7 Ausblick

In diesem Kapitel werden zwei Arten von Verbesserungsvorschlägen diskutiert, deren Anwendung vorteilhaft erscheint. Zum Einen werden in Abschnitt III.7.1 Ideen präsentiert, die die vorgestellte Lösung erweitern. Zum Anderen präsentiert Abschnitt III.7.2 einen Ansatz, der vor allem auf die in Kapitel III.6 angesprochenen Probleme abzielt und eine Änderung des Transaktions-Moduls und der ihm zugrundeliegenden Indexstruktur bedingt.

III.7.1 Erweiterung

Semantische Transaktionsserialisierung Die Serialisierbarkeit von Transaktionen ist abhängig von ihrer Semantik, wie ein einfaches Beispiel in Tabelle 1 verdeutlicht.

T_1	T_2	A	B
		25	25
A+=100			
$A_{T_1} = 125$			
	A *= 2		
	$A_{T_2} = 50$		
	B *= 2		
	$B_{T_2} = 50$		
	commit	50	50
B+=100			
$B_{T_1} = 150$			
commit - fail			

T_1	T_2	A	B
		25	25
A+=100			
$A_{T_1} = 125$			
	A *= 1		
	$A_{T_2} = 25$		
	B *= 1		
	$B_{T_2} = 25$		
	commit	25	25
B+=100			
$B_{T_1} = 125$			
commit		125	125

Tabelle 1: *Seien A und B Zahlen, und gelte die Konsistenzbedingung A==B. In beiden Tabellen ist jeweils der selbe Ablauf dargestellt, bei dem Transaktion T_2 später beginnt als T_1, aber eher commited. Im linken Fall führt dies dazu, dass T_1 auf inkonsistenten Werten arbeitet und letztendlich verworfen werden muss, weil sonst die Konsistenzbedingung verletzt ist. Im rechten Fall dagegen kann T_1 commiten, da die Konsistenz bewahrt bleibt. Obwohl der Ablauf in beiden Fällen gleich ist, konnte also im rechten Fall aufgrund des konkreten Verhaltens von T_2 ein effizienterer Schedule verwendet werden, als Links. (Beispiel nach [29, S. 922–923])*

Die üblichen Techniken zur Nebenläufigkeitskontrolle gehen von der Annahme aus, dass „es [für den Scheduler] nicht realistisch ist, sich mit Details der Berechnungen zu befassen, die von Transaktionen vorgenommen werden, [da sie] häufig Code einer Allzweck-Programmiersprache, sowie SQL Anfragen, umfassen [...].“ [29].

Da die SDB diese Schichten beseitigt hat, ist in ihrem Fall auch die Analyse der Semantik von Transaktionen denkbar. Da auch Kung in [16] in seinem Entwurf zur Serial Validation erwähnt, dass semantische Analysen für MVCC vorteilhaft sein können, erscheint es angebracht, diesen Ansatz zu untersuchen. Es bestehen zwei Möglichkeiten, dem Transaktionsmodul semantische Informationen zur Analyse bereit zu stellen. Anstatt lediglich zu speichern, auf welchem Index ein Lesevorgang basiert, kann die Transaktion den eigentlichen Wert speichern, der gelesen wurde. Während der Validierung muss dann nicht mehr überprüft werden, ob sich der Index verändert hat, sondern ob der aktuell globale Wert noch dem Gelesenen entspricht. Auf diese Weise können Beispiele wie das obige anhand der Semantik der (potentiell) konfligierenden Transaktionen validiert werden, anstatt bei Konflikten sofort ab zu brechen. Außerdem könnte das Cachen von gelesenen Werten in vielen Fällen die Geschwindigkeit von nachfolgenden Lesevorgängen erhöhen, da dadurch der wiederholte Lookup in der `Column` entfällt.

Eine andere Möglichkeit wäre es, der Transaktion bei ihrer Erzeugung die aufrufende Funktion zu übergeben. Dies ist möglich, da den `transactional`– und `retry`–Annotationen, die eine neue Transaktion starten, die auszuführende Methode als Methodenobjekt vorliegt. Ob diese zugrundeliegenden Berechnungen hilfreich dabei sein können, effizientere Schedules zu erzeugen, sollte untersucht werden.

Geschachtelte Transaktionen Datenbankmanagementsysteme wie z.B. Microsoft SQL-Server unterstützen die Schachtelung von Transaktionen. Dieses Modell ist mit der Struktur des Transaktions-Moduls auch in der SDB umsetzbar. Im Folgenden wird knapp das Vokabular definiert und im Anschluss eine Implementierungsstrategie umrissen.

Eine Transaktionen heißt *innere Transaktion*, wenn sie komplett von einer anderen Transaktion umschlossen ist, die *äußere Transaktion* genannt wird. Die Änderungen der äußeren Transaktion sind der Inneren sichtbar. Umgekehrt sieht die äußere Transaktion die Änderungen der Inneren erst, wenn diese erfolgreich commited. Global sichtbar werden alle Änderungen erst, wenn die äußerste Transaktion commited.

Eine innere Transaktion könnte gestartet werden, indem `Transaction.begin` innerhalb eines Transaktionskontexts gerufen wird. Des Weite-

ren müssten die Methoden `read_index` und `write_index` derart angepasst werden, dass sie im Falle eines in der aktiven Transaktion nicht veränderten Objektes den aktuellen Index von ihrer äußeren Transaktion erfragen. Das Verhalten einer ungeschachtelten Transaktion, die in diesem Fall den global gültigen Zustand in der Datenbank nachschlägt, kann erreicht werden, in dem die `NullTransaction` als äußerste Transaktion aufgefasst wird, deren Caches den globalen Zustand zurück geben. Analog müsste auch die Methode `commit_changes` umgebaut werden, damit sie in die Caches der jeweils äußeren Transaktion commited, wobei ein Commit in die `NullTransaction` dem globalen Zugängigmachen in der DB gleichkäme.

Eine solche Erweiterung erscheint aus theoretischer Perspektive nützlich. Die Möglichkeit Transaktionen zu schachteln erlaubt dem Anwendungsentwickler eine feinere Kontrolle über Ausführungseinheiten. Außerdem erleichtert sie modulare Entwicklung, da Methoden bei Bedarf einfach transaktional definiert werden können, ohne dass später Probleme entstehen, weil sie eventuell aus einem anderen transaktionalen Kontext gerufen werden müssen. Ob dies für den Anwendungsprogrammierer allerdings auch aus praktischer Sicht nützlich ist, muss noch bestimmt werden.

III.7.2 Änderung

Wie Abschnitt III.6.2 festgestellt wurde, stellt die Aufspaltung des aktuellen Zustands zwischen Transaktions-Caches und Indexstruktur ein Performanzproblem dar, weil Mengenoperationen zur Auflösung von Anfragen über die gesamte Datenbasis nötig sind. Um dies zu beseitigen scheint es logisch, die Verwaltung unterschiedlicher Versionen unterhalb der Transaktionsebene zu implementieren. Ein Ansatz dazu ist die *Trie* genannte Datenstruktur. Diese erlaubt die Übersetzung von IDs in Indizes zu versionieren, wodurch die Verwaltung des aktuellen Zustands nicht mehr durch unterschiedliche Module realisiert werden müsste. Im Folgenden wird kurz die Funktionsweise von Tries beschrieben, die sich an der ausführlichen Darstellung in Kapitel II orientiert. Im Anschluss wird aufgezeigt, wie dies genutzt werden könnte um MVCC zu ermöglichen.

Tries sind Präfix-Bäume, bei denen Schlüssel in kleinere Einheiten zerlegt werden, die jeweils die Position von Kindknoten darstellen. Beispielweise würde der Schlüssel 65052 in einem 256-ären Trie mit drei Ebenen nachgeschlagen werden, in dem er als dreistellige Zahl zur Basis 256 interpretiert wird: $65052 = (0, 254, 1)_{256}$. Der Eintrag wäre dementsprechend durch rekursives nachschlagen des 0-ten, 254-ten und 1-ten Eintrags erreichbar. Dabei bilden die ersten beiden Einträge wieder auf Knoten ab, während der letzte den gesuchten Wert enthält. Im Fall der SDB würden die Schlüssel Objekt-IDs sein, die auf Array-Indizes abbilden.

Soll nun Versionierung implementiert werden, so kann für jede startende Transaktion eine flache Kopie des Wurzelknotens angelegt werden, deren Einträge auf die Kindknoten ihrer „Elternversion“ verweisen. Lesevorgänge werden also auf den Daten ausgeführt die zum Zeitpunkt des Beginns der Transaktion global gültig waren. Wird ein Schreibvorgang ausgeführt, so findet ein Copy-on-Write des betroffenen Blattknotens statt. Die Änderungen werden also auf einer lokalen Kopie im Transaktionseigenen Baum durchgeführt. Außerdem müssen die Elternknoten des kopierten Blattes auch kopiert und derart angepasst werden, dass sie den neuen Eintrag referenzieren. Dies bedeutet, dass sich der Vorgang durch den Baum zur Wurzel propagiert, bis er auf einen bereits kopierten Knoten trifft (siehe Abb. 28). Lesevorgänge die im Anschluss statt finden werden durch den Trie also automatisch auf die neue Version übersetzt. Damit entfällt für das Transaktionsmodul der Bedarf lokale Versionen in einem `write_-` oder `delete_cache` zu speichern. Ein Zwischenspeicher für Lesevorgänge müsste zum Zweck der Validierung weiterhin geführt werden, könnte aber lediglich die IDs gelesener Objekte vorhalten.

Der Commit einer Transaktion bedeutet im Prinzip lediglich das Umsetzen einer globalen Instanzvariable, die den global gültigen Trie speichert, auf die lokale Version der abgeschlossenen Transaktion. Vorher muss allerdings während der Validierung überprüft werden werden, ob nicht einer der gelesenen Einträge in der Zwischenzeit verändert worden ist. Dies kann z.B dadurch geschehen, dass geprüft wird, dass die Eltern-Referenzen der entsprechenden Blattknoten immer noch auf den global aktuellen Wurzelknoten zeigen. Außerdem muss überprüft werden, ob nicht-kopierte Knoten noch aktuell sind, und diese im negativen Fall aus der aktuellsten Version übernommen werden. Für diesen Anwendungsfall können bestehende Algorithmen für einen Drei-Wege-Merge verwendet werden.

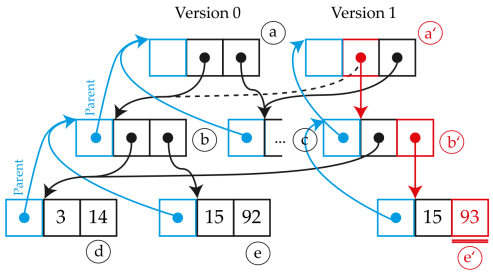


Abb. 28: *Ein Schreibvorgang auf e erzeugt den Blattknoten e' sowie rekursiv den Eintrag b'. Der Wurzelknoten a' bestand bereits, er wurde angelegt als Version 1 erzeugt wurde.*

Durch das Entfernen der Zwischenspeicher erscheint es plausibel anzunehmen, dass die Versionierung mithilfe eines Tries speichereffizienter ist als die aktuelle Lösung. Es müsste aber untersucht werden, ob der Geschwindigkeitsvorteil,

der durch die Beseitigung der Indirektion von Lese- und Schreibvorgängen über das Transaktionsmodul erreicht wird, die Kosten für den Merge-Vorgang aufwiegen kann. Ein davon unabhängiger Vorteil des vorgestellten Ansatzes ist jedoch, dass Transaktionen die nur Lese- und keine Schreibvorgänge ausführen nicht mehr validiert werden müssen, da sie immer nur die Daten der Elternversion lesen können, und dadurch niemals inkonsistent werden. Dies bedeutet, dass sie fast genauso schnell durchgeführt werden können, wie in einem System ohne Nebenläufigkeitskontrolle.

III.8 Fazit

Diese Arbeit demonstriert mit einem Prototypen, dass bei einer Verschmelzung der Adressräume von Hauptspeicherdatenbank und Laufzeitumgebung einer höheren Programmiersprache Nebenläufigkeitskontrolle derart implementiert werden kann, dass Anwendungen mit diesem Prototypen nebenläufig programmiert werden können. Dazu wurde der Prototyp der SDB um ein Modul zur transaktionalen Nebenläufigkeit nach dem Multiversion-Concurrency-Control-Prinzip erweitert, und entsprechend umgebaut um es zu nutzen.

Es wurde argumentiert, das damit typische Nebenläufigkeitsprobleme gelöst werden. Außerdem wurde festgestellt, dass die durch Nebenläufigkeitskontrolle entstehende Änderungen bei der Sprachnutzung leichtgewichtig sind, wobei der Synchronisationsaufwand für den Endnutzer in vielen Fällen darauf beschränkt werden konnte zu entscheiden, wie groß ein Transaktionskontext zu wählen ist.

Die gewählte Implementierung nutzt das Potential von MVCC nicht komplett aus, was in einem signifikanten Mehraufwand sowohl bei Lese- als auch bei Schreibintensiven Transaktionen im Vergleich zum ursprünglichen Prototypen resultiert. Es wurden allerdings mehrere Ansätze vorgestellt, mit denen die Implementierung weiter optimiert werden könnte. Insbesondere bei Transaktionen, die nur Leseoperationen durchführen, könnte dadurch größerer Mehraufwand im Vergleich zu einem System ohne Nebenläufigkeitskontrolle verhindert werden.

Teil IV

Embedded Query Language

IV.1 Einleitung

Einer der Schlüsselaspekte unserer sprachintegrierten Hauptspeicherdatenbank ist, dass sie die Trennung zwischen Datenbank und dem Anwendungscode vollständig auflöst. Dies erspart dem Programmierer den ständigen Wechsel zwischen verschiedenen Programmiersprachen. Infolgedessen kann er übersichtlicheren Code schreiben und Fehler schneller beheben, da jede Stelle des ausgeführten Codes vom Debugger erfasst werden kann. Die SHD ist also ähnlich wie Gemstone [7] oder Zope [12] in die Hostsprache selbst integriert. Bei der SHD werden allerdings die Objekte im Speicher nur durch ihre Attribute persistiert, anstatt komplett serialisiert abgespeichert zu werden. Dafür wird jedes Attribut eines Objektes in einer dedizierten Spalte innerhalb der Datenbank abgespeichert. Durch diese Art der Speicherung können Informationen gut komprimiert werden und die Menge von Daten, die unkomprimiert im Speicher vorliegen, minimal gehalten werden, wie in Kapitel II bereits erklärt wurde. Grundsätzlich ergeben sich so für den Programmierer keinerlei Einschränkungen beim Arbeiten mit der Datenbank. Jede Operation, die der Nutzer auf Einträgen in der Datenbank durchführen möchte, kann über die Interaktion mit den „Referenzobjekten“, wie in Kapitel II beschrieben, durchgeführt werden. Dazu gehören auch analytische Anfragen auf großen Datenmengen, welche auch als OLAP-Anfragen bezeichnet werden. Obwohl der Programmierer über die Referenzobjekte jede Form von OLAP-Anfrage imperativ definieren und ausführen könnte, ergeben sich für ihn mehrere Probleme: Einerseits muss er stets in einem transaktionalen Kontext arbeiten, der jedes Lesen eines Attributes aufzeichnet, um die Konsistenz des Lesevorgangs zu garantieren. Dies führt zu einem recht signifikanten Geschwindigkeitsverlust, der sich insbesondere beim Analysieren von sehr großen Datenmengen bemerkbar macht. Zudem ist die Ausführung von Anfragen im Python-Objektraum im Vergleich zu äquivalenten Anfragen auf gängigen Datenbanken[9] recht langsam, da der Attributdispatch stets über einen kostspieligen Dictionarylookup vollzogen wird, und Python parallel dazu den Speicher automatisch verwaltet. Letztlich gestaltet sich die imperative Formulierung von Anfragen an vielen Stellen als sehr umständlich. So sollte der Programmierer oft benutzte Operatoren wie Gruppierung oder Join generisch und optimiert benutzen können, ohne sich mit deren Implementierung zu befassen. Das bewahrt ihn davor, sich mit den Laufzeiteigenschaften der formulierten Anfrage auseinander zu setzen.

Die EQL ist als eine Schnittstelle für den Programmierer konzipiert, welche beim Arbeiten mit der SHD für genau die besprochenen Probleme eine Lösung anbietet. Die EQL ist eine deklarative Formulierung einer Anfrage, die sich direkt in Python integriert und es gleichzeitig erlaubt, den relativ langsamen Objektraum in Python zu umgehen, um analytische Anfragen direkt

[9] Dazu gehören zum Beispiel DB2, MySQL oder SQLite

auf den Spalteneinträgen auszuführen. Dabei verfügt die EQL über alle standardmäßigen Mengenoperationen wie Join, Gruppierung oder Aggregationen, welche hier insbesondere für das Arbeiten auf Spaltenbasis optimiert wurden. Zwischenergebnisse, die aus vorigen EQL-Anfragen generiert wurden, werden so platzsparend wie möglich gespeichert. Zudem werden alle Anfragen so spät wie möglich ausgeführt um unnötigen Mehraufwand zu vermeiden.

Die Arbeit gliedert sich in folgende Teile: Als Erstes werden wir die EQL-Schnittstelle aus Sicht des Endnutzers betrachten. Im Abschnitt IV.3 werden dann verschiedene Konzepte angerissen, die bei der Implementierung der EQL von großer Bedeutung sind. In Abschnitt IV.4 erfolgt eine detailliertere Beschreibung der Implementierung der EQL. Wir gehen außerdem auf die Probleme und Einschränkungen ein, die diese Implementierung mit sich bringt. In Abschnitt IV.5 beschäftigen wir uns mit der Frage, wie EQL-Anfragen durch einen Wechsel der Laufzeitumgebung beschleunigt werden können. Zudem haben wir Beispielanfragen ausgeführt, um den Geschwindigkeitszuwachs, den die EQL dem Anwender bringt, analysieren zu können (Abschnitt IV.6). Zudem werden wir die Aspekte der EQL ansprechen, die wir im Rahmen des Bachelorprojektes noch nicht implementiert haben (Abschnitt IV.7). Es folgt ein kurzer Vergleich zu LINQ, was eine in C# eingebettete Notation für Datenbankanfragen ist (Abschnitt IV.8) und anschließend ein Fazit.

IV.2 EQL: Notation und Ausführung

IV.2.1 Syntax und Semantik

Die EQL Schnittstelle basiert auf einer mengenorientierten, deklarativen Syntax. Mengenorientiert heißt, dass der Nutzer sein Ergebnis definiert, indem er mehrere Ausgangsmengen schrittweise einschränkt, miteinander kombiniert und anschließend projiziert. Dabei werden die Mengenoperationen in einer deklarativen Art und Weise konfiguriert. Das heißt, wir beschreiben für jede Zwischenmenge, wie das Ergebnis aussehen soll, und nicht wie wir das Ergebnis berechnen. Die Kernidee der EQL-Notation ist, dass die Ergebnismenge mithilfe einer Methodenkaskade definiert wird, die schrittweise die Ausgangsmenge einschränkt bzw. transformiert. Der Vorteil dieser Notation ist, dass sie sich nahtlos in die Hostsprache einfügt, und keine umständlichen Stringoperationen, wie etwa bei einer SQL-Anfrage, benötigt werden. Die EQL arbeitet nahtlos mit den aus der Datenbank extrahierten Objekten zusammen. So kann der Programmierer innerhalb der Operatorkonfigurationen auf alle Eigenschaften der Objekte zugreifen, die in der zu transformierenden Menge enthalten sind.

Dazu gehören Attribute, Assoziationen oder auch Methoden[10]. Durch diese Integration erreichen wir, dass für den Programmierer kein Bruch zwischen dem Arbeiten mit Referenzobjekten und Anfragen auf den Objektmengen entsteht. In Quelltext 35 ist ein Beispiel einer EQL-Anfrage zu sehen.

```
1  Transaction.start("24.1.2007")
2  query = OrderDetail.all_instances \
3          .where(lambda detail: detail.count == 1) \
4          .group_by(
5               year = lambda detail: detail.order.year,
6               name = lambda detail: detail.product.name
7           ) \
8           .select(
9               year = lambda group: group.key().year,
10              name = lambda group: group.key().name,
11              revenue = lambda group: group.sum(
12                  lambda detail: detail.product.price)
13         )
14 Transaction.commit()
```

Quelltext 9: *Beispielanfrage in EQL.*

Wir wollen nun die Wirkungsweise dieser Anfrage erläutern, indem wir die Semantik der einzelnen Methodenaufrufe beschreiben. Wir beginnen, wie jede Operation auf der SHD, mit dem Initialisieren einer Transaktion. Hier übergeben wir der Transaktion zusätzlich ein Datum, was dem Nutzer ermöglicht eine sogenannte *Timetravel Query* durchzuführen. Das heißt, dass die Anfrage auf einer Wertemenge arbeiten soll, die dem Zustand der Datenbank zu dem besagten Datum entspricht. Dieser Schritt ist allerdings rein optional, und hat auf die im Folgenden erklärten Eigenschaften keinen Einfluss. Ausgangspunkt für die Anfrage ist die Menge aller OrderDetails im System. Die Klassenvariable `all_instances` bildet dabei einen syntaktischen Einstieg für eine EQL-Anfrage. Es folgt die erste richtige Mengenoperation `where(lambda detail: detail.count == 1)`, welche die vorher referenzierte Menge der OrderDetails einschränkt. Dies geschieht per Auswertung eines Prädikates, dass als Argument in Form eines Lambdas (Block in Python), dem `where()` übergeben wird. Für jedes Lambda einer Mengenoperation gilt dabei: Das Argument, welches dem Lambda übergeben wird, ist stets ein Element der Menge, die transformiert wird. Im diesem Fall von `where()`, wäre das also ein `OrderDetail`-Objekt. Auf dieser eingeschränkten Menge wird nun mittels dem `group_by`-Statement eine Gruppierung der Objekte durchgeführt. Hier kann eine beliebige Menge von Lambdas übergeben werden, allerdings dienen sie nicht als ein Prädikat, sondern als Attributreferenz, nach der anschließend

[10] Leider gibt es aus technischen Gründen Einschränkungen beim Nutzen von Funktionen innerhalb von Lambdas. Mehr dazu im Abschnitt IV.4.

gruppiert werden kann. Zudem wird jedem Lambda ein Name zugewiesen (In diesem Fall `year` bzw. `name`), welcher in der daraus resultierenden Menge als Bezeichner für die Instanzvariable im Schlüssel fungiert.[11] Um über die gebildeten Gruppen anschließend eine Aggregationen durchzuführen, benutzen wir `select()`. Dabei werden wieder mithilfe der Lambda-Notation die gewünschten Attribute, die in den Ergebnis-Objekten enthalten sein sollen, selektiert und neue Attribute durch Aggregationen (In diesem Fall `sum`) erstellt. Jedes dieser Lambdas bekommt, wie auch schon bei der Gruppierung, einen Bezeichner, der dann als Instanzvariable für das entsprechende Ergebnisobjekt wieder auftaucht. Im konkreten Fall bestünde die Ergebnismenge aus einer Reihe von Objekten, mit den Instanzvariablen `year, name, revenue`. Die Ergebnisobjekte fungieren dabei als reine Datenhalter, ohne jegliches Verhalten, da Methoden aus Objekten der Ausgangsmenge offensichtlich nicht sinnvoll übernommen werden können. Haben wir die Ausgangsmenge lediglich gefiltert oder umsortiert, so sind im Ergebnis jedoch selbstverständlich Objekte der Quellmenge enthalten. Nachdem der besprochene Code ausgeführt wurde, ist in der Variable `query` eine vollständig ausführbare Anfrage gespeichert. Genau genommen wird in der Variable `query` dabei ein Generator gespeichert, welcher die Auswertung erst anstößt, sobald die ersten Elemente angefordert werden. Alternativ, kann die Anfrage auch als Subanfrage in eine andere eingesetzt werden. Ein solcher Anwendungsfall ist in Quelltext 10 zu sehen.

```
1 result = Order.all_instances \
2          .join(query,
3                lambda order: order.year,
4                lambda query_obj: query_obj.year.
5                lambda order, query_obj: \
6                                new(revenue = query_obj.revenue,
7                                    year = order.year))
```

Quelltext 10: *Einsetzen eines Zwischenergebnisses als Subanfrage.*

Hier verbinden wir die Menge aller Order-Objekte mit den Objekten, die aus unserer vorigen Anfrage entstanden sind. Dafür rufen wir auf der Menge der Order-Objekten den `join()`-Operator auf, der als erstes Argument die Menge der Joinpartner erwartet. Die nächsten beiden Argumente sind Lambdas, welche das jeweilige Joinattribut selektieren. Das erste Lambda ist dabei für das Attribut auf der Seite des Order-Objekts zuständig, während das zweite das Attribut auf den Objekten der `query`-Anfrage auswählt. Zu beachten ist, dass auch die in `query` gespeicherte Anfrage erst ausgewertet wird, sobald dies mit `result` geschehen ist.

[11] Man beachte dass in diesem Fall die Gruppierungsattribute über jeweils eine Assoziation erreicht werden, wir aber, konträr zu einer äquivalenten SQL-Anfrage, keinerlei Join durchführen müssen. Das ist einer der Vorteile, den wir durch das Formulieren einer Anfrage auf Objektbasis erhalten.

IV.2.2 Ausführung von EQL-Anfragen

Nachdem wir die Schnittstelle für den Endnutzer betrachtet haben, wollen wir nun einen groben Überblick über den Ablauf der Ausführung einer EQL-Anfrage erlangen. Diese gliedert sich in folgende drei Schritte:

Tracing-Phase Hier wird der eigentliche EQL-Code ausgewertet, allerdings nicht in der Art und Weise, wie es dem Nutzer suggeriert wird. Anstatt eine vollständige Ausführung der Anfrage anzustoßen, wird hier, den Methodenaufrufen von `where`, `group_by`, `join`, `etc.` entsprechend, ein Anfragenplan generiert. Zusammen mit dem Anfrageplan, werden auch die übergebenen Lambdas zu neuen Datenstrukturen umgewandelt, die direkt auf den Attributwerten in den Spalten und nicht auf den Eigenschaften von Laufzeitberechnungen Berechnungen ausführen. Diese neuen Datenstrukturen können dann indexbasiert ausgewertet werden. Es besteht zudem die Möglichkeit, verschiedene Arten von algebraischer Anfragenoptimierung durchzuführen. Diese können beispielsweise auf statistischen Werten von Entitätsmengen oder einfachem Zusammenführen von `where`-Bedingungen basieren.

Index-Abruf Hier werden von allen beteiligten Entitätsmengen die gültigen und konsistenten Indexmengen angefordert. Beteiligte Entitätsmengen sind dabei alle Mengen, die als Einstiegspunkt innerhalb der Anfrage (`all_instances`) genutzt wurden oder mit einer Assoziation innerhalb eines Lambdas traversiert werden. Ein Index steht dabei für einen gültigen Eintrag innerhalb der Spalten, in denen die jeweiligen Objekte des Entitätstypen gespeichert sind. Wird dem `Transaction.start()` ein Datum übergeben, um eine Timetravel Query zu initiieren, so muss die Indexmenge entsprechend angepasst werden, da die Menge der zum damaligen Zeitpunkt gültigen Indizes entsprechend anders war.[12] Um die Indexmenge effizient zu bestimmen, greifen wir auf die zur Entität gehörenden Primary-Key-Datenstruktur zurück, die in Form eines von uns konstruierten „Versioning-Tries“ Kapitel II vorliegt. Aus diesem können wir in effizienter Zeit die entsprechende Indexmenge extrahieren. Nebenläufige Änderungen auf der Entitätsmenge haben dabei, dank der Versioning-Trie-Datenstruktur, keinen Einfluss auf die konsistent gelesene Indexmenge.[13]

[12] Wie Änderungen auf der Datenbank die Indexmenge verändern, kann in Kapitel III nachgelesen werden.

[13] Es sei hier angemerkt, dass wir im aktuellen Stand den Versioning-Trie noch nicht in die Datenbank eingebunden haben. In unserem aktuellen Prototypen laufen die EQL-Anfragen also noch Gefahr inkonsistente Daten zu lesen. Auf die zu erwartende Ausführungsgeschwindigkeit sollte dies jedoch keinen Einfluss haben.

Indexbasierte Auswertung Es wird nun über die im vorigen Schritt erhaltene Indizes iteriert und die in der Tracing-Phase konstruierten Datenstrukturen pro Index ausgewertet. Bei der Auswertung der Einträge, werden die Spalten an der entsprechenden Stelle, auf den der Index verweist, dekomprimiert. Die gelesenen Werte werden anschließend weiter verarbeitet, bis ein Ergebniswert errechnet ist. Bei dieser Auswertung wird die Indirektion über Referenzobjekte vollständig umgegangen. Dies ist aufgrund der in der Tracing-Phase umgewandelten Lambdas möglich. Es muss darauf geachtete werden, dass beim Auswerten in einer möglichst sequentiellen Reihenfolge vorgegangen wird, um den Kompressionsaufwand gering zu halten. Der Anfrageplan führt anschließend weitere Operationen mit den Ergebnissen der indexbasierten Auswertung durch, wie Gruppierungen, Filter oder Join. Dabei werden für Gruppierungen und Joins rein hashbasierte Algorithmen verwendet. Hashalgorithmen eignen sich insbesondere, da sie im Gegensatz zu schleifen- oder sortierbasierten Algorithmen schlimmstenfalls eine lineare Ausführungskomplexität besitzen [21]. Die Ergebnisse einer analytischen Anfrage sind sogenannte *Benannte Tupel*, es sei denn, wir haben die Menge nur gefiltert oder umsortiert. Benannte Tupel sind Objekte, die als reine Datenhalter fungieren. Sie zeichnen sich lediglich durch eine Menge von Attributen aus, die mit entsprechenden Werten belegt sind. Diese Tupel können nun wieder wie Objekte einer neuen Entitätsmenge behandelt werden, sie haben nur keinerlei Verhalten.

IV.3 EQL-Konzepte

Um die im Abschnitt IV.2.2 definierte Abarbeitung von EQL-Anfragen zu ermöglichen, werden mehrere Konzepte verwendet, die im Folgenden beschrieben werden. Zum einen haben wir nach Lösungen gesucht, um die übergebenen Lambdas in eine von Objekten unabhängige Form zu konvertieren. Zum anderen muss ein Anfrageplan erstellt werden, um das gewünschte Anfrageergebnis zu berechnen. Außerdem haben wir nach Möglichkeiten gesucht, die definierten Anfragen in Umgebungen außerhalb von Python auszuführen um die Laufzeit der Ausführung zu minimieren.

IV.3.1 Meta-Tracing

Hinter dem Begriff „Meta-Tracing“ steckt die Idee ‚die Ausführung von Pythoncode aus Python heraus zu reflektieren. Genauer gesagt wird hierbei zur Laufzeit Pythoncode analysiert und anschließend ein Ausführungsplan erstellt, der darstellt, was der analysierte Pythoncode zur Laufzeit ausführen würde. In Abbildung 29 ist schematisch ein Beispiel für das Ergebnis des Meta-Tracing für ein Lambda zu sehen.

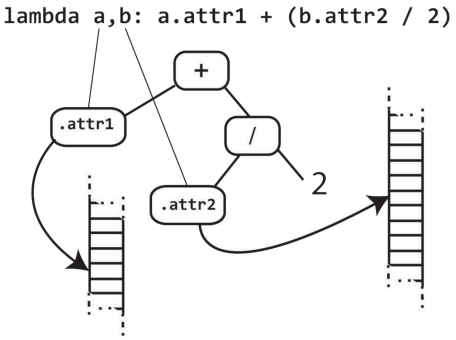


Abb. 29: *Objektstruktur nach erfolgreichen Meta-Tracing eines Lambdas.*

Das Resultat des Meta-Tracings bezeichnen wir als *Trace*. Ein Trace wird indexbasiert ausgewertet, was bedeutet, dass er als Argument nicht mehr die Objekte der Datenbank bekommt, sondern nur noch den Indexeintrag der Objekte innerhalb der jeweiligen Spalte. Dies ermöglicht es uns einerseits, die Objekterstellung zu umgehen, und andererseits, während der gesamten Auswertung der Anfrage an den Objekten vorbei zu arbeiten. Dabei sparen wir uns eine Reihe teurer Dictionarylookups und können die Attribute von Objekten mittels eines direkten Arrayzugriffes auswerten. Wie in Abschnitt IV.6 detaillierter gezeigt wird, konnten wir so die Geschwindigkeit von Anfragen innerhalb von Python verdoppeln. Zusätzlich kann der Trace in beliebiger Form abgespeichert, und entsprechend auch in anderen Laufzeitumgebungen verwendet werden. Ist die Ausführung von Anfragen beispielsweise in einem separaten C-Modul implementiert, bietet es sich an, den Trace in Form eines Bytecodes zu speichern und an eine dedizierte Funktion im C-Modul zu übermitteln, welche diesen dann interpretiert und ausführt.

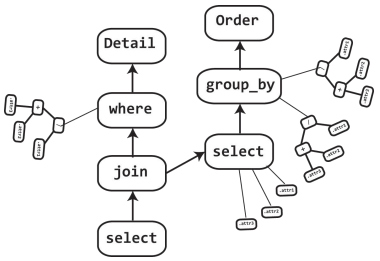


Abb. 30: *Objektstruktur eines generierten Anfrageplans. Die kleinen Bäume, sind die von manchen Mengenoperationen referenzierten Lambdas, die bereits das Meta-Tracing durchlaufen haben.*

IV.3.2 Generierung des Anfrageplans

Jeder Operator, der auf dem `all_instances`-Attribut des Entitätstyp, innerhalb der Kaskade aufgerufen wird, stellt einen Zwischenstand der gewünschten Ergebnismenge dar. Deshalb repräsentieren wir jeden Methodenaufruf durch ein eigenes Objekt, welches speichert, um welche Mengenoperation es sich aktuell handelt und wie diese zu berechnen ist, sowie alle benötigten Lambdas, die bereits das Tracing durchlaufen haben. Der Anfrageplan wird dann als eine verkettete Liste solcher Objekte abgespeichert. Subanfragen, die auch verkettete Listen von solchen Mengen-Objekten sind, können wiederum von Objekten innerhalb des Anfrageplans referenziert werden und somit auch ausgewertet werden. Dadurch erhalten wir Gabelungen innerhalb der Liste, wie man schematisch in Abbildung 30 erkennen kann.

IV.3.3 Minimale Laufzeitüberprüfungen der Transaktionen

Normalerweise muss für jedes Auslesen eines Attributes in einem transaktionalen Kontext aufgezeichnet werden, auf welcher Version des Attributwertes dieses Lesen stattfand. Dies ist nötig, da unsere Datenbank simultane Änderungen auf der Datenbank mittels *Multi-Version-Concurrency* (MVCC) durchführt. Werden Änderungen auf Basis von gelesenen Attributen gemacht, so muss zum Zeitpunkt des Commits geprüft werden, ob das gelesene Attribut noch aktuell ist, die Transaktion also einen konsistenten Stand der Daten übernehmen kann. Dazu wird bei jedem Commit die Aufzeichnung aller in der Transaktion stattgefundenen Leseoperationen überprüft. Näheres zum Ablauf des Lesens und Schreibens innerhalb einer Transaktion kann in Kapitel III nachgelesen werden. Da eine EQL-Anfrage ein rein lesender Zugriff ist, können wir einen großen Teil des Mehraufwands der Transaktionen weglassen. Wir haben hier nicht die Möglichkeit inkonsistente Daten zu schreiben, es besteht lediglich die Gefahr, inkonsistent zu lesen, auch wenn das in der Regel wenig Einfluss auf das Gesamtergebnis einer Anfrage hat. Um auch die Lesezugriffe nicht protokollieren zu müssen, holen wir uns direkt nach Beginn der Transaktion eine Referenz auf die aktuelle Indexstruktur jeder Entität. Dadurch erhalten wir in unserem transaktionalen Kontext einen konsistenten Schnappschuss der Indexmengen aller Entitäten. Dies ist dank der Version-Trie-Datenstruktur in konstanter Zeit möglich. Zudem garantiert uns der Versioning-Trie, dass parallele Änderungen die Indexmenge auf Basis deren wir die EQL-Anfrage ausführen, nicht beeinflussen. Anschließend führen wir die nötigen Auswertungen mithilfe der Traces indexbasiert durch. Dabei umgehen wir zudem die standardmäßigen Überprüfungen der Transaktionen, da wir nun nicht mehr mit den Referenzobjekten arbeiten.

IV.3.4 Beschleunigung der Laufzeitumgebung

Ein Möglichkeit, die Geschwindigkeit der EQL-Anfragen entscheidend zu erhöhen, ist es, diese in einer anderen Laufzeitumgebung als Python auszuführen. Dabei muss gewährleistet sein, dass die externe Ausführung für den Programmierer vollständig transparent geschieht. Dies ist eine Option, die ihm Rahmen des Bachelorprojektes noch nicht implementiert wurde. Wir werden an dieser Stelle aber aufgrund des großen Potentials der Idee, diese im Folgenden etwas detaillierter besprechen. Eine alternative Laufzeitumgebung kann beispielsweise eine SQL-Datenbank, oder auch ein C-Modul, welches man von Python aus ansteuern kann, sein. Die CPython Implementierung macht hierbei ausgiebig von der Möglichkeit Gebrauch, C-Code direkt aus Python heraus auszuführen. Dies wird beispielsweise beim `itertools`-Modul verwendet, welches eine weitläufige Auswahl von mathematischen Funktionen bereitstellt, um über Mengen zu iterieren. Hierbei wurden insbesondere die Schleifen in purem C implementiert, was zu einer sehr schnellen Ausführungsgeschwindigkeit des Codes führt. Um die Transparenz der externen Ausführung zu gewährleisten, müssen wir alle möglichen Fehler, die durch falsches Formulieren einer EQL-Anfrage auftreten können, noch innerhalb von Python abfangen. Dies können wir relativ einfach innerhalb der Tracingphase erledigen, bei der beispielsweise eine Typinferenz durchgeführt werden kann, die inkompatible Operationen vorzeitig erkennt und meldet. Für eine EQL-Anfrage bestünde nun die Möglichkeit, die Anfragepläne mitsamt der Traces an das C-Modul zu übermitteln, und die Operationen auf den Spalten innerhalb des C-Modules auszuführen. CPython bietet bei seinen Extension-Modulen die Möglichkeit an, innerhalb von C-Code noch direkt auf Python-Objekten zu arbeiten. Der Anfrageplan, sowie seine aufgezeichneten Lambdas, können hierbei im C-Code traversiert und dann entsprechend ausgewertet werden. Alternativ kann sich der Anfrageplan mitsamt seiner Lambdas zu einem Bytecode kompilieren, welcher dann an das C-Modul übergeben und ausgewertet werden kann. Der Vorteil dieser Lösung ist, dass hier auch der langsame Python-Objekt-Dispatch, der beim Arbeiten mit Proxyobjekten auftreten würde, umgangen wird.

IV.4 EQL Implementierung

Herzstück der Implementierung von EQL sind die Prozesse, welche innerhalb der Tracingphase stattfinden. Wir wollen uns deshalb im Folgenden mit der Implementierung der Anfrageplangenerierung und des Meta-Tracings beschäftigen.

IV.4.1 Lambda-Tracing

Das Lambda-Tracing geschieht technisch mithilfe von *Lambda Proxys*, die wir anstatt der echten Objekte in die Lambdas einspeisen. Der Proxy gibt sich dabei als das Objekt eines Entitätstypen aus und reagiert auf alle Arten von Operationen, die auf ihm ausgeführt werden, als ob sie auf dem Objekt direkt stattfinden. Dabei ist der entscheidende Unterschied zu einem normalen Objekt, dass nicht die eigentlichen Werte von Funktionsaufrufen oder Attributdereferenzierung zurückgegeben werden, sondern die Aktionen lediglich aufgezeichnet werden. Dieser Aufzeichnungsprozess ist als Aufbau eines Objektbaumes implementiert, bei dem die beteiligten Objekte genau die Proxys sind. Wird beispielsweise auf einem Proxy, der eine Entität von z.B. OrderItems emuliert, ein Attribut angefordert, so wird ein neuer Proxy (in diesem Falle ein AttributeProxy) angelegt und der OrderItem-Proxy als Datenquelle von diesem neuen Proxy referenziert. Jede Aktion, sei es ein Methodenaufruf, Attributaufruf oder die Anwendung eines mathematischen Operators, durchläuft in Python zur Laufzeit die `__getattr__()`-Methode, die den eigentlichen Dispatch zu der benötigten Information auf dem jeweiligen Objekt durchführt. Indem wir die `__getattr__()`-Methode innerhalb der Proxys entsprechend überschreiben, ermöglichen wir den Proxys, jede Operation, die auf ihnen ausgeführt wird, zu reflektieren. Das überschriebene `__getattr__()` stellt somit die Kernfunktion unseres Tracevorgangs dar. Der Tracevorgang ist ein relativ aufwändiger Prozess. Da das Tracing allerdings nur für ein Proxy-Objekt ausgeführt wird, hat das es somit keine messbare Auswirkung auf die Laufzeit einer EQL-Anfrage. Das ist möglich, da der Proxy als ein Repräsentant aller für einen Entitätstyp gespeicherten Objekte fungieren kann.

```
def __getattr__(self, attr):
    if attr.startswith('_'):
        if hasattr(self, attr):
            return getattr(self, attr)
        else:
            raise AttributeError, "%s not found!" \% attr
    if hasattr(self._data_source, attr + '_type'):
        association = getattr(self._data_source,
                              attr + '_type')
        if (isinstance(association, Many)):
            return ManyAssociationProxy(attr,
                                        association,
                                        self)
        elif (isinstance(association, One)):
            return OneAssociationProxy(attr,
                                       association,
                                       self)
        else:
            return HbtmAssociationProxy(attr, association)
    if callable(getattr(self._data_source, attr)):
```

```
21          self_proxy = type('Self_Proxy',
22                            (Proxy, self._data_source),
23                            dict(Proxy.__dict__))
24          self_proxy.__new__ = Proxy.__new__
25
26          for each in [key for key, value
27                       in self._data_source.__dict__.items()
28                       if isinstance(value, property)]:
29              setattr(self_proxy, each,
30              self_proxy(self._data_source).__getattr__(each))
31
32          return CallProxy(self_proxy(self._data_source),
33                           getattr(self._data_source, attr))
34
35      return AttributeProxy(self, attr)
```

Quelltext 11: *Der Tracing-Kernel der Lambda Proxys.*

Wie aus Quelltext 11 ersichtlich ist, gibt es genau vier verschiedene Fälle von Interaktionen, die auf dem Proxy ausgeführt werden können:

Aufruf proxyinterner Felder Proxys tätigen keine eigentliche Ausführung des in den Lambdas definierten Code, sondern zeichnen den entstehenden Kontrollfluss nur auf. Da aus diesem Grund jeder Proxy intern gänzlich andere Aktionen vollführt, als die vom Benutzer beschriebenen, muss garantiert werden, dass diese Ausführungen nicht mit aufgezeichnet werden. Dazu besitzt jeder Proxy private Instanzvariablen oder Methoden, die mit einem Unterstrich gepräfixt sind. Dies erlaubt dem Proxy, interne Operationen durchzuführen, ohne dass diese aufgezeichnet werden. Wird also ein Instanzvariablen, oder Methodenaufruf auf dem Proxy durchgeführt, wird erst geprüft, ob es sich um eine privates Attribut handelt, auf dem wir dann die nicht überschriebene Variante von `getattr()` aufrufen.

Assoziationsdereferenzierung Wird auf einem Objekt eines Entitätstyps ein Feld angefragt, das eine Assoziation referenziert, so kann diese eine One-, Many- oder Has-and-Belongs-to-Many-Assoziation sein. Für jeden dieser Fälle muss ein neuer Proxy erstellt werden, der das Auslesen einer solchen Assoziation auf einem Objekt entsprechend emuliert. Bei der indexbasierten Auswertung wird ein OneAssociationProxy beispielsweise nur eine Indexübersetzung durchführen, während ein ManyAssociationProxy eine komplett neue Indexmenge aufspannt. Zudem müssen Assoziations-Proxies in der Lage sein, das Verhalten des referenzierten Objektes zu emulieren.

Methodenaufruf Es werden auch Methodenaufrufe aufgezeichnet, die auf den Objekten ausgeführt werden. Dazu wird die Ausführung von der jeweiligen Methode aufgezeichnet. Dies machen wir, indem wir das `self`-Argument der Methode zur Laufzeit durch einen passenden Proxy ersetzen und die Methode dann ausführen. Aus Gründen, die in Abschnitt IV.4.2 beschrieben werden, kann es sein, dass man für manche Funktionsaufrufe, den Tracevorgang auslassen möchte. Wollen wir explizit verhindern, dass eine Methode aufgezeichnet wird, so kann die Methode in einem FunctionProxy verpackt werden. Dies geschieht, indem die Funktion mitsamt ihren Argumenten dem Funktionsruf `call_function` übergeben wird, wie Quelltext 12 zeigt.

```
1 Product.all_instances
2         .where(lambda p:
3                   call_function(external_f, p, 'verify')))
```

Quelltext 12: *Beispiel, wie man eine Methode ruft, ohne das sie aufgezeichnet wird.*

Attributdereferenzierung Trifft keine der oben aufgeführten Abfragen zu, nimmt der Proxy an, dass es sich bei dem angefragten Feld um ein Attribut der Entität handelt, also um ein Feld, dessen Werte direkt in einer Spalte abgelegt sind. Hierfür wird ein neuer AttributeProxy erstellt, der eine direkte Referenz auf die entsprechende Spalte hält, und diese indexbasiert auswerten kann. Zusätzlich speichern sich AttributProxys den Datentyp, der in der referenzierten Spalte gespeichert wird. Auf diese Weise kann eine vollständige Typinferenz über alle im Lambda genutzten Primitive und Operationen vollführt werden. Dies ist beispielsweise nötig, um die Anfrage in einer streng getypten Programmierumgebung auszuführen. Grundsätzlich kann man auch in einer schreibenden Transaktion Operationen per Lambda-Tracing aufzeichnen und dann über den jeweiligen Trace ausführen. Wird dabei aber auf Basis dieser Ergebnisse eine Änderung vorgenommen, kann die Konsistenz der geschriebenen Daten von den Transaktionen nicht mehr garantiert werden, weil wir beim indexbasierten Auswerten die Lesezugriffe nicht protokolliert haben. Um zu unterbinden, dass der Benutzer auf Basis von ausgewerteten Traces Daten in die Datenbank schreibt generiert das Lambda-Tracing eine Ausnahme, wenn es bemerkt, dass es außerhalb einer schreibgeschützten Transaktion ausgeführt wird. Dies garantiert im übrigen auch, dass alle EQL-Anfragen nur innerhalb von schreibgeschützten Transaktionen ausgeführt werden können.

IV.4.2 Grenzen des Lambda-Tracings

Der Ansatz des Lambda-Tracings, sowie die Tatsache, dass wir den Tracevorgang innerhalb von Python selbst durchführen, hat auch zu mehreren Proble-

men geführt, welche im Folgenden erläutert werden. Es ist hierbei wichtig anzumerken, dass diese Problematik vorwiegend der Tatsache entspringt, dass wir den Code der Lambdas aus Sicht der Argumente aufzeichnen. Alternativ kann man auch die gesamte Ausführung des Pythoncodes aufzeichnen, wofür man aber Teile des Interpreters austauschen müsste, was ein sehr viel größeren Aufwand darstellen würde.

Lazy-Logic-Operatoren Die erste Problematik hängt direkt mit der Implementierung von Python selbst zusammen: In Python besteht die Möglichkeit, Prädikate mittels postponiert auswertbaren Operatoren zu formulieren. Dazu gehören beispielsweise `and`, `or` oder `if`. Die Implementierung dieser Operatoren ist dabei identisch zur Formulierung von logischen Operatoren im Lambdakalkül. So wertet der `and`-Operator das erste Argument aus und wird das zweite Argument erst auswerten, wenn sich das erste als wahr herausgestellt hat. Diese Art und Weise der Auswertung ist nicht nur elegant zu implementieren, sie ermöglicht auch eine schnellere Ausführung, da unter Umständen aufwändige Auswertungen zum Berechnen des korrekten Ergebnisses wegfallen. Python implementiert diese Auswertung von Prädikaten, indem auf die Argumente eines logischen Operators die `__nonzero__()` Methode aufgerufen wird. Zudem wird verifiziert, dass der Rückgabewert ein `bool` oder `integer` ist. Diese Verifikation schlägt entsprechend fehl, wenn statt des Boolean oder Integers ein Proxy-Objekt zurückgegeben wird. Wir müssen aber bei jedem Methodenaufruf auf dem Proxy, wieder den Proxy selbst zurückgeben, da wir ansonsten die Möglichkeit verlieren, weiter aufzuzeichnen. Aus diesem Grunde wird der Trace bei einem Aufruf von `__nonzero__()` zwangsläufig abgebrochen, da die Proxy-Objekte hier nicht weiter propagiert werden können. Alle Versuche, Python dennoch dazu zu bewegen, das Proxyobjekt weiter zu reichen, schlugen bisher fehl. Die aktuelle Lösung verbietet dem Nutzer daher, die Lazy-Operatoren zu verwenden, indem bei einem Aufruf von `__nonzero__()` auf dem Proxy ein Fehler geworfen wird. Es muss stattdessen auf die vollständig auswertenden Operatoren wie & oder || zurückgegriffen werden.

Tracing von Methoden Da wir alle Argumente der Lambdas durch Proxys austauschen, und Lambdas in Python nur rein funktional definiert werden dürfen, können wir jeden Codeabschnitt innerhalb eines Lambdas aufzeichen. Beim Aufzeichnen von Methoden ergeben sich aus Sicht des Tracings allerdings mehrere Probleme: Zum einen übersieht das Tracing zwangsläufig alle Kontrollflüsse, die auf Variablen basieren, die nicht mit einem Proxy verpackt wurden. Da wir das gesamte Tracing auf Basis der von Proxys vertretenen Objekte durchführen, können wir keine sinnvolle Aussage über diese Variablen treffen. Ein Beispiel für eine solche Situation ist in Abbildung 31 zu sehen.

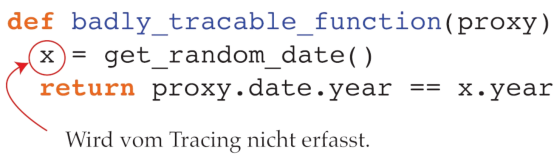


Abb. 31: *Informationsverlust beim Tracen von Methoden.*

Aus diesem Grund nehmen wir in so einem Fall stets an, dass es sich bei den Objekten um eine Konstante handelt. Sind solche vom Tracing übersehene Variablen mit von Proxys vertretenen Objekten in einem Term, so geschieht Folgendes: Wir verpacken die zur Zeit des Tracevorgangs vorhanden Werte in den übersehenen Variablen in sogenannten Constant-Proxys, die den Wert der jeweiligen Variable dann für jede Auswertung des Traces als konstant annehmen. Diese Annahme mag für einige Methoden funktionieren, sie schlägt aber freilich fehl, wenn es sich bei den nicht aufgezeichneten Variablen um ständig ändernde Werte, wie zum Beispiel die aktuelle Uhrzeit, handelt. Für solche Funktionen haben wir die Funktion `call_function` bereit gestellt, die den Tracevorgang an der Methodengrenze abbricht und das benutze Methodenobjekt in einem FunctionProxy verpackt. Bei der indexbasierten Auswertung wird diese Funktion dann als „Black-Box“ angesehen und die zum Tracevorgang identifizierten Argumente entsprechend übergeben. Da hierbei auch Objekte des Entitätstypen generiert werden müssen, kann die Verwendung von `call_function` zu großen Einbußen bei der Ausführungsgeschwindigkeit führen. Zudem ist man darauf angewiesen, dass der Code der vom Tracing ausgeschlossenen Funktion möglichst effizient ist. Dadurch kann die von `call_function` verpackte Funktion zu einem Flaschenhals für die gesamte Ausführung der Anfrage werden. Die zweite Gefahr ist, dass Methoden die aus den Lambdas heraus aufgerufen werden, Seiteneffekte haben, also Änderungen auf in der Datenbank befindlichen Objekten durchführen. Die reinen Änderungen sind dabei weniger ein Problem, da wir uns ja noch immer in einem transaktionalen Kontext befinden. Basieren diese Änderungen jedoch auf Werten, die durch indexbasierte Auswertung entstanden sind, so können inkonsistente Änderungen in die Datenbank gelangen, da das Auslesen der Spalten vollzogen wurde, ohne die Lesezugriffe zu protokollieren. An dieser Stelle kommen uns allerdings die bereits erwähnten schreibgeschützten Transaktionen zur Hilfe. Diese lassen die laufende Transaktion scheitern, sobald ein schreibender Zugriff auf die Datenbank stattfindet. Werden also in einer EQL-Anfrage Methoden aufgezeichnet oder ausgeführt, die einen Seiteneffekt haben, so scheitert die Transaktion, in der wir uns befinden.

IV.4.3 Collection-Proxys

Wie in Abschnitt IV.3 beschrieben, wird die formulierte EQL-Anfrage in eine verkettete Liste von Objekten umgewandelt, die jeweils den Zwischenstand

einer Anfrage repräsentieren. Werden andere EQl-Anfragen als Subanfragen eingeflochten, gibt es entsprechend Gabelungen in der Liste. Die einzelnen Objekte sind in unserer Implementierung sogenannte *Collection-Proxys*, da sie analog zu den Proxys, welche zum Tracen der Lambdas verwendet werden, als Stellvertreter für Mengen fungieren. Zum Zeitpuntk der Auswertung einer Anfrage wird dann jede Menge schrittweise materialisiert, wobei wir in jedem Collection-Proxy nur die nötigsten Informationen ab speichern, um den Speicheraufwand möglichst gering zu halten. Die Arten von Collection-Proxys lassen sich grob in folgende Kategorien einordnen: Einerseits solche Collection-Proxys, welche die Anordnung oder Menge von Objekten ändern. Dazu gehören beispielsweise der `WhereProxy`, `OrderByProxy` oder der `AllProxy`. Die Zwischenergebnisse werden dabei entweder als einfacher Wert (`bool, int, float ...`) oder als Liste von Indizes abgespeichert. Die andere Gruppe der Collection-Proxys erstellt neue Objekte zur Laufzeit, die dann als Anfrageergebnis fungieren oder weiter „verarbeitet“ werden können. Zu dieser Gruppe gehören `join`, `group_by` und `select`. Die Zwischenergebnisse werden hierbei als Kombination von alten und neuen Spalten gespeichert. Neue Spalten, welche beispielsweise Aggregationen enthalten, liegen als nicht komprimierte Liste vor. Dies erhöht einerseits die Ausführungsgeschwindigkeit von Anfragen, andererseits wäre eine Kompression auf solchen Spalten wenig ertragreich, da sie tendenziell klein sind und sehr selten Lücken (NULL-Werte) enthalten. Jeder Collection-Proxy besitzt Methoden wie `where(), order_by(), group_by()`, die jeweils den passenden Collection-Proxy mit den übergebenen Argumenten (Lambdas) initialisieren und zurückgeben. Auf diese Weise erreichen wir, dass eine wie in Quelltext 35 formulierte Methodenkaskade zum gewünschten Anfrageplan führt. Der Rückgabewert der Kaskade ist der jeweils zuletzt generierte Collection-Proxy. Jeder Collection-Proxy implementiert dabei die `__iter__`-Methode, also eine Generator-Schnittstelle. Wird auf dem Collection-Proxy angefangen zu iterieren, was beispielsweise mittels der `list()`-Funktion oder einer `for`–Schleife passieren kann, so beginnt sich der Collection-Proxy auszuwerten. Dazu lässt er seinen jeweiligen Eltern-Proxy die für ihn nötigen Auswertungen vornehmen. Nachdem die Auswertungskaskade durch den gesamten Anfragenbaum gelaufen ist, kann der unterste Collection-Proxy anfangen, die Ergebnistupel zu generieren.

IV.5 Laufzeitumgebungen

Es gibt grundsätzlich zwei verschiedene Möglichkeiten die formulierten EQL-Anfragen möglichst schnell auszuführen. Zum einen kann ein Wechsel der Ausführungsumgebung vollzogen werden. Da Collection- und Lambda-Proxies eine Repräsentation der Anfrage darstellen, die komplett unabhängig von der Python-Objektstruktur ist, können diese beispielsweise von in C geschriebenen Algorithmen interpretiert und ausgeführt werden. Dies wäre aus

CPython heraus zum Beispiel mittels eines Extension-Modules möglich. Die andere Möglichkeit ist, den ausführenden Interpreter durch einen schnelleren zu ersetzen. Dies hat aus Sicht der Implementierung den Vorteil, dass kein Wechsel der Programmiersprache nötig ist, und die Ausführung einer Anfrage direkt in Python vom Debugger erfasst werden kann.

IV.5.1 Ausführung in CPython-Extension-Modul

Die Ausführung in einem C-Extension-Modul wurde in unserem Prototypen nur sporadisch implementiert. Ein großer Teil des im Folgenden erläuterten basiert also auf theoretischen Konzepten, die aber ohne Probleme implementiert werden könnten.

Die Ausführung einer EQL-Anfrage innerhalb eines C-Modules bedingt, dass die Spalten, auf welche die Python -Objekte ihre Attribute abbilden, in C zur Laufzeit erreichbar sind. Hierfür haben wir ein Extension-Modul geschrieben, welches die in Kapitel II formulierte Abbildung von Attributen und Assoziation auf Spalten vornimmt. Dies hat zur Folge, dass alle persistierten Attribute in einer von C-Code zur Laufzeit erreichbaren Form im Extension Modul vorliegen. Generell ist die Ausführung von Code innerhalb eines Extension-Modules genau so schnell wie normaler C-Code, allerdings ist die Übersetzung von PythonObjekten zu C-Typen kostspielig und sollte so selten wie möglich verwendet werden. Es bietet sich entsprechend an, einen Großteil der Ausführung in das Modul zu verlagern.

Wir schlagen dazu folgende Implementierung vor: EQL-Operationen wie `where, group_by, join` oder `select` werden zu Bytecodes, die als Argumente wiederum Referenzen auf Bytecodes enthalten, die aus dem Tracing der Lambdas generiert wurden. Die EQL-Operatoren können durch Interpretieren des jeweiligen Bytecodes ausgeführt werden und laufen dabei in purem C-Code. Pro Iteration können die Operatoren die benötigten Daten dabei aus dem Bytecode berechnen, der aus den Lambdas generiert wurde.

Die zu erwartende Performance ist schwer einzuschätzen, sie dürfte sich aber in etwa an die von SQLite annähern, das auf dem gleichen Prinzip von Bytecodes arbeitet. Dort werden SQL-Anfragen, nachdem sie die Anfrageoptimierungen durchlaufen haben, an eine virtuelle Maschine übergeben, die Anfrage dann pro Opcode entsprechend auswertet. Ein Performancevergleich zu unserer Lösung in PyPy kann in Abschnitt IV.6 nachgelesen werden.

IV.5.2 Ausführung in PyPy

Für unseren Protypen der SHD haben wir den PyPy-Interpreter verwendet, was aktuell der schnellste allgemein verwendbare Python Interpreter ist. Auch die Ausführung der EQL-Anfrage erfolgt entsprechend in PyPy. Der PyPy-Interpreter basiert auf einem sogenannten Meta Tracing JIT-Compiler. Ein *JIT-Compiler* (Just in Time Compiler), ist ein Interpreter, der zur Laufzeit neuen Code generiert, kompiliert und dann ausführt. Dies wird meist dafür verwendet, die Ausführung von interpretierten Programmiersprachen, wie beispielsweise Python, zu beschleunigen. Bekannte Beispiele, bei denen die Verwendung von JITs zu deutlichen höheren Ausführungsgeschwindigkeiten führte, sind die JAVA-Hotspot VM oder die in Google Chrome verwendete Javascript-Engine V8. Die schnellere Ausführung von Programmen erreichen JITs vor allem, indem sie über den aktuell ausgeführten Code reflektieren. Der generierte Code kann so stark optimiert und bei mehrfacher Verwendung zwischengespeichert werden. Die Optimierungen können auch für das jeweilige ausführende Prozessormodell angepasst sein. Zudem können Optimierungen am generierten Code in Abhängigkeit von Statistiken gemacht werden, die zur Laufzeit vom interpretierten Programm gemacht wurden.

Bemerkenswert an PyPy ist dabei, dass der hier verwendete JIT-Compiler Meta-Tracing verwendet. Das heißt, dass zur Laufzeit die Ausführung des JIT-Compilers analysiert wird und entsprechend Optimierungen vorgenommen werden. Anschließend wird der JIT-Compiler zur Laufzeit neu kompiliert und löst die alte Version des JIT ab, in der Hoffnung nun eine noch schnellere Ausführung zu erlauben. Ein Vorteil eines Meta-Tracing-JIT ist, dass er unabhängig von der interpretierten Sprache optimieren kann. Dies ist insbesondere interessant für PyPy, was sich als Basis zur Entwicklung von Interpretern für jede beliebige Sprache versteht. So gibt es bereits PyPy-basierte Interpreter für Prolog oder Smalltalk. Die Heuristiken und Optimierungstechniken, die PyPy verwendet, sind komplex und würden den Rahmen dieser Arbeit bei weitem sprengen. Das Prinzip, nach dem PyPy den ausgeführten Code optimiert, lässt sich aber einfach in zwei grundsätzliche Annahmen[14] zusammen fassen:

1. Programme halten sich größtenteils innerhalb von Schleifen auf.
2. Die vom Programm durchlaufenen Schleifen werden zum Großteil auf die gleiche Art und Weise durchlaufen.

Diese beiden Annahmen schlagen sich innerhalb von PyPy in folgendem Optimierungsverhalten nieder: Bemerkt PyPy dass ein Code-Segment mehrere

[14] Näheres zum Optimierungsprozess des Meta-Tracing-JIT kann in [3] nachgelesen werden.

Male hintereinander ausgeführt wurde, der JIT also mehrmals den gleichen Code generiert hat, so nimmt es an, sich in einer Schleife zu befinden, bei der es Optimierungen vornehmen kann. Dies wird auch als das „Warmlaufen“ einer Schleife bezeichnet. Was nun passiert, ist, dass PyPy den nächsten Durchlauf der Schleife aufzeichnet und die Seiteneffekte protokolliert. Dies ist das gleiche Vorgehen, wie bei dem bereits beschriebenen Lambda-Tracing. Ist der Trace abgeschlossen, so werden die vom Kontrollfluss durchlaufen Abzweigungen identifiziert. Das können zum Beispiel Bedingungen innerhalb einer `while`-Schleife oder einem `if-else`-Konstrukt sein. An diesen Stellen positioniert PyPy nun sogenannte Guards, die prüfen, ob die Annahme für den aktuellen Durchlauf der Schleife noch stimmen. Gleichzeitig führt PyPy auf Basis des Traces eine Reihe von Optimierungen durch, wie das „Inlinen“ von Funktionen oder „Loopunrolling“.[15] Schlägt einer der aufgestellten Guards fehl, so fällt PyPy automatisch zu dem Standard-JIT-Compiler zurück und führt den weiteren Code dort aus, bevor es wieder anfängt, zu tracen und Optimierungen vorzunehmen. Da PyPy letztlich hochoptimierten Assembler generiert und ausführt, kann es vorkommen, dass in Python formulierte Programme schneller ausgeführt werden als äquivalenter C-Code, da normale C-Compiler keinerlei Laufzeitanalysen vornehmen oder den Code temporär an gewisse Begebenheiten anpassen. In einigen pathologischen Fällen kann allerdings auch das Gegenteil der Fall sein, und die Ausführung unter PyPy deutlich langsamer sein als innerhalb des CPython Interpreters. Es hängt also stets vom konkreten Code ab, wie stark der Geschwindigkeitsgewinn ist, der von PyPy erreicht werden kann. Unter der Annahme, dass PyPy für eine schnelle Ausführung der EQL-Anfrage innerhalb Python garantieren konnte, haben wir den gesamten Prototypen unserer SHD in Python programmiert. Auch die Ausführung der EQL-Anfragen, also die schrittweise erfolgende Auswertung der Ergebnismenge sowie die indexbasierte Auswertung erfolgen gänzlich in Python.

IV.6 Auswertung

Im Folgenden wollen wir analysieren, welchen Einfluss der PyPy-Interpreter auf die Ausführung der EQL-Anfrage hat. Insbesondere wollen wir dabei auf die hier aufgeführten Fragen eingehen:

[15] Unter *Inlining* von Funktion versteht man das Einfügen des gesamten Codes der Funktion an die Stelle, wo eigentlich der Funktionsruf stattfinden würde. Auf diese Weise spart man sich einen Sprung, bzw. den Dictionarylookup, der die Methode zur Laufzeit an eine konkrete Implementierung der Funktion bindet. Beim *Loopunrolling* werden Schleifen sukzessive ausgeschrieben, anstatt über sie zu iterieren.

1. Wie stark beschleunigt PyPy die Ausführung einer EQL-Anfrage gegenüber dem normalen CPython-Interpreter?
2. In wieweit ist das Tracen der Lambdas noch für die Ausführung von EQL-Anfragen relevant, wenn die Auswertung gänzlich in Python stattfindet?
3. Wie verhält sich die Laufzeit einer „imperativ“ formulierten Anfrage zu der einer EQL-Anfrage?
4. Wie verhält sich die Laufzeit der Anfragen im Verhältnis zu anderen Datenbanken?

IV.6.1 Testaufbau

Wir haben alle zur Auswertung angeführten Benchmarks auf einer Fedora 17 x64 Installation ausgeführt. Die Ausführung fand innerhalb einer Virtuellen Maschine mit 8 dedizierten Prozessorkernen und 11GB RAM statt. Die VM wurden auf einem System mit Intel Core i7-2720QM CPU à 4 x 3,30 GHz, Hyper-Threading und 16 GB DDR3-RAM ausgeführt. Da wir keine realistischen Firmendaten für die Auswertung zur Verfügung gestellt bekommen haben, haben wir die von der Anfrage analysierten Daten zufallsgeneriert. Das Domänenmodell, auf dem die Anfrage arbeitet, ist in Abbildung 32 zu sehen.

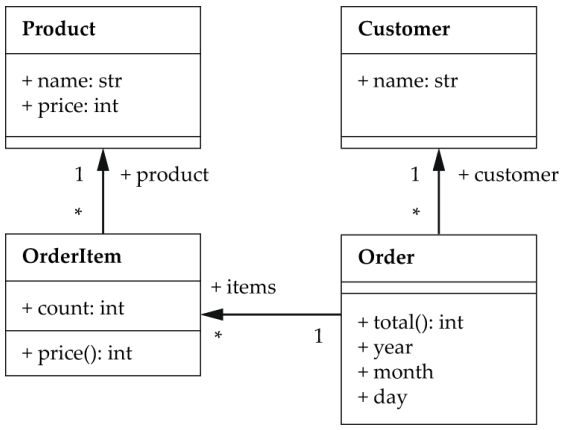


Abb. 32: *Domänenmodell, auf dem die Testanfragen ausgeführt werden.*

Für die Anfrage haben wir 1 mio. Posten, 100 000 Bestellungen, 10 000 Kunden und 1000 Produkten generiert. Die Anfrage berechnete dabei den Umsatz gruppiert nach Jahr und Produkt und entspricht der in Quelltext 13 gezeigten EQL-Anfrage. Alle Anfragen wurden, wie bereits beschrieben, in einem transaktionalen Kontext ausgeführt, der keine Schreiboperationen zulässt. Auf diese Weise werden überflüssige Überprüfungen vermieden, die die Laufzeit der Anfragen unnötig verlangsamen würden.

```
OrderDetail.all_instances \
    .where(lambda detail: detail.count > 5) \
    .group_by(
        year = lambda detail: detail.order.year,
        name = lambda detail: detail.product.name
    ) \
    .select(
         year = lambda group: group.key().year,
         name = lambda group: group.key().name,
         revenue = lambda group: group.sum(
             lambda detail: detail.product.price \
                            * detail.count)
    )
```

Quelltext 13: *Die für die Benchmarks verwendete EQL-Anfrage.*

Zusätzlich vergleichen wir die EQL-Anfrage mit einer äquivalenten imperativ formulierten Anfrage, die auf den einzelnen Objekten einer Entität arbeitet. Das Resultat ist hier zwar keine Menge von benannten Tupeln, sondern lediglich ein Dictionary, das die Schlüsselattribute auf die Aggregate abbildet. Für die Messung der Laufzeit hat dies aber keinen beachtenswerten Einfluss. Um zu messen, wie stark die Auswirkungen des Lambda-Tracings auf die Laufzeit bei einer Ausführung innerhalb von Python sind, haben wir eine zusätzliche Variante der imperativen Anfrage formuliert. Bei dieser sind alle Bedingungen und Attributauswertungen durch Lambdas ersetzt, die vorher das Lambda-Tracing durchlaufen haben und dann indexbasiert ausgewertet werden.

```
result = collections.defaultdict(int)

all_detail = list(OrderDetail.all())
for detail in all_detail:
    if(detail.count > 5):
        result[(detail.order.year, detail.product.name)] += \
                detail.product.price * detail.count
```

Quelltext 14: *Die zur EQL-Anfrage äquivalente imperativ formulierte Anfrage.*

Letztlich testen wir auch noch eine äquivalente SQL-Query, welche wir auf einer anderen Hauptspeicherdatenbank ausgeführt haben, in der das gleiche Schema abgespeichert ist. Hierfür haben wir die frei verfügbare Datenbank SQLite verwendet, die einen Betriebsmodus besitzt, bei der die gesamte Datenbank vollständig im Hauptspeicher gehalten wird.

```
SELECT o.year, p.name, SUM(p.price * d.count) AS revenue
FROM orders o, products p, order_details d
WHERE d.order_id = o.id
```

```
4         AND d.product_id = p.id
5         AND d.count > 5
6   GROUP BY p.name, o.year
```

Quelltext 15: *Die zur EQL-Anfrage äquivalente SQL-Anfrage.*

Wir haben die Ausführung der Anfrage in insgesamt sieben verschiedenen Varianten getestet: EQL-Anfrage in CPython 2.7 (EQL: Python), Imperative Anfrage in CPython 2.7 (Imp: Python), Imperative Anfrage mit Lambda-Tracing Unterstützung in CPython 2.7 (Imp-Proxy: Python), EQL-Anfrage in PyPy 1.9.2 (EQL: PyPy), Imperative Anfrage in PyPy 1.9.2 (Imp: PyPy), Imperative Anfrage mit Lambda-Tracing Unterstützung in PyPy 1.9.2 (Imp-Proxy: PyPy), SQL-Anfrage ausgeführt unter SQLite 3 (SQL: SQLite). Dabei haben wir die Zeit, die zum Einfügen von den Testdaten benötigt wird, nicht beachtet. Diese wird detaillierter in Kapitel II untersucht. Die Anfrage wurden nach dem Erstellen und Einfügen der Testdaten zehn Mal hintereinander auf dem System ausgeführt. Dabei wurden die kürzeste, längste und durchschnittliche Ausführungszeit der jeweiligen Ausführung gemessen. Um den Vergleich zwischen den imperativen Anfragen und der EQL möglichst fair zu gestalten, haben wir alle Anfragen in schreibgeschützten Transaktionen ausgeführt. Auf diese Weise ersparen wir insbesondere der rein objektorientierten Anfrage Laufzeitüberprüfungen der Transaktionen.

		durchsch. Zeit (s)	Min (s)	Max (s)
1	EQL: PyPy	0.9826	0.9380	1.1279
2	Imp: PyPy	1.8233	1.7908	1.9205
3	Imp-Proxy: PyPy	0.9519	0.8256	1.0314
4	EQL: Python	17.6563	17.5701	17.8115
5	Imp: Python	29.6542	29.1340	30.1918
6	Imp-Proxy: Python	22.1900	21.6395	22.6122
7	SQL: SQLite	2.8361	-	-

Tabelle 2: *Ergebnisse der beschriebenen Testanfragen.*

IV.6.2 Interpretation der Messungen

Beschleunigung durch PyPy Wie aus den Messungen erkennbar ist, erreicht PyPy einen Beschleunigung der Anfragen um durchschnittlich Faktor 19. Dabei wurde die EQL, wie auch die imperative Anfrage ähnlich stark beschleunigt. Interessant ist, dass PyPy die tracebasierte imperative Anfrage noch stärker optimieren konnte, sodass diese leicht schneller als die EQL-Anfrage ist.

Relevanz des Lambda-Tracing Vergleicht man die Laufzeit der imperativ formulierten Anfragen, so ist diejenige, welche die Auswertung auf Basis der Traces durchführt, knapp doppelt so schnell. Diese Beschleunigung ist zum Teil unabhängig von der Laufzeitumgebung, denn auch unter CPython ist die auf Traces basierende Anfrage schneller, obwohl der Algorithmus die Anfrage bei beiden Anfragen nach dem gleichen Prinzip abarbeitet. Der Grund für die schneller Auswertung der Traces, liegt im Umgehen der Objekte sowie deren Properties. Referenziert man im Objektraum beispielsweise eine Attribut über zwei Indirektionen (zwei One-Associations), so wird drei mal ein Dictonarylookup vollzogen, der jeweils die gewünschte Property zurückgibt, auf dem wiederum ein Methode zum Auslesen des jeweiligen Attributes aufgerufen wird. Innerhalb des Traces ist dieser Vorgang lediglich ein dreifach geschachtelter Methodenaufruf, der vom PyPy Interpreter zudem sehr einfach geinlined und entsprechend optimiert werden kann. Zudem wird zu keinem Zeitpunkt auf den Cache der Transaktionen zugegriffen.

Vergleich: Imperative und EQL-Anfragen In beiden Laufzeitumgebungen ist die EQL-Anfrage der rein objektorientierten imperativen Anfrage in Anbetracht der Geschwindigkeit überlegen. Vergleicht man die EQL-Anfrage mit der imperativen Anfrage, die auf Traces arbeitet, so wird deutlich, dass innerhalb von Python oder PyPy, sämtliche Geschwindigkeitsvorteile der EQL auf dem Lambda-Tracing beruhen. So ist die tracebasierte Anfrage unter PyPy sogar leicht schneller als die korrespondierende EQL-Anfrage. Dies ist aber darauf zurückzuführen, dass die imperative Anfrage den Aufbau von speziellen Ergebnistupeln auslässt und lediglich das entsprechende Dictionary zurückgibt. Durch konsequentes Inlining der `_evaluate`-Methoden der Lambda-Proxies, erreicht PyPy an dieser Stelle einen noch stärkeren Geschwindigkeitsgewinn als in CPython.

Vergleich zu SQLite Der Kern von SQLite ist ein in C geschriebener Bytecode Interpreter, der ähnlich arbeitet wie das von uns vorgeschlagenen Extensionmodul. SQLite optimiert nicht nur den Anfrageplan, sondern führt die einzelnen Bytecodes auch in hochoptimierten Algorithmen aus. Wenig überraschend ist deshalb jede Form von Anfrage innerhalb unserer SHD unter CPython der Laufzeit von SQLite mit bis zu Faktor 10 deutlich unterlegen. Interessant ist jedoch, dass PyPy alle Beispielanfragen auf der SHD schneller ausführt als SQLite. Die EQL-Anfrage ist dabei in etwa doppelt so schnell wie die äquivalente Anfrage in SQL. Über die genauen Gründe für diesen Unterschied kann nur gemutmaßt werden. Vermutlich spielen aber zwei Faktoren in das gemessene Ergebnis: Zum einen generiert PyPy zur Laufzeit optimierten Assembler, der in der Tat an einigen Stellen schneller sein kann als der von SQLite verwendete C-Code. Zum anderen ist die SQL Anfrage gezwungen, zwei Joins zu bilden, um die Assoziationen zwischen dem

OrderDetail und Product sowie der Order aufzulösen. Dies ist innerhalb der SHD nicht nötig, da wir Assoziation bereits vorher materialisiert abspeichern. Es ist deshalb lediglich ein Auffinden des Fremdschlüssels innerhalb der entsprechenden Spalte der Assoziation nötig.

IV.7 Ausblick

Um die Ausführung von EQL-Anfragen weiter zu beschleunigen gibt es weitere Möglichkeiten, die von uns im Rahmen dieses Bachelorpojektes allerdings nicht näher untersucht wurden. Dies ist zum einen die Optimierung von den generierten Anfragepläne, sowie die parallele Ausführung von Anfragen auf mehreren Kernen eines Prozessors. Auf diese Möglichkeiten soll im Folgenden kurz eingegangen werden.

IV.7.1 Anfragenoptimierung

Eine Grundidee der Anfrageoptimierung ist es, die einzelnen Zwischenergebnisse von Mengenoperationen so früh wie möglich so klein wie möglich zu halten. Die Optimierung eines Anfrageplanes geschieht dabei in der Regel auf Basis von zwei Methoden: Zum einen ist es möglich auf algebraischem Wege die Anfrage so umzuformulieren, dass die Zwischenergebnisse möglichst früh sehr klein werden. Dies geht beispielsweise durch das frühe Anwenden von Selektionen, oder Gruppierungen. Werden sehr viele Joins hintereinander durchgeführt, so bietet es sich auch an, diese in ihrer Reihenfolge so anzupassen, dass der Join mit dem kleinsten Zwischenergebnis am frühesten ausgeführt wird. Diese algebraischen Transformationen können anschließend auf den Anfrageplan angewendet werden, der ja aus Listen von Collection-Proxies besteht. Die andere Technik, anhand derer optimiert wird, sind Kostenabschätzungen. Dies ist insbesondere interessant, wenn wir keine generelle Regel angeben können, ob eine algebraische Transformation sinnvoll ist. Das ist beispielsweise bei Joins der Fall, denn im Gegensatz zu Selektionen, welche die Menge zuverlässig verkleinern, ist das Verhalten von Joins aus rein algebraischer Sicht nicht abschätzbar. Um Kosten abschätzen zu können, müssen Heuristiken über die einzelnen Ausgangsmengen oder Joinmengen gesammelt werden. Zu diesen gehören, Anzahl der Elemente, Anteil der Duplikate oder auch die Selektivität eines bestimmten Joins mit einer anderen Menge. Dazu könnten Statistiken in den Klassen der Entitäten abgespeichert werden und zusätzlich eine globale Liste von mehrfach benutzten Joins oder Gruppierungen zusammen mit den gesammelten Statistiken gesammelt werden.

IV.7.2 Parallele Ausführung auf Mehrkernsystemen

Neben dem Optimieren von Anfrageplänen setzen moderne Datenbanksysteme, wie zum Beispiel die von SAP entwickelte HANA [21], stark auf die parallele Ausführung von Anfragen auf Systemen mit mehreren Kernen. Python erlaubt das Programmieren mit mehreren Threads, allerdings lässt sich damit keine höhere Ausführungsgeschwindigkeit erzielen, da alle Threads innerhalb eines Interpreters aufgrund des GIL *(Global Interpreter Lock)* niemals echt parallel Code ausführen können. Der GIL entstammt dabei in erster Linie der Einsicht, dass parallele Programmierung in der Regel zu hochkomplizierten Abhängigkeiten führt, welche der Programmierer nur schwer durchschauen kann. Aus diesem Grunde ist parallele Ausführung in Python nur in mehreren Prozessen, nicht aber in mehreren Threads möglich. Hierfür kann das seit Python 2.5 bereitgestellt *multiprocessing*-Modul verwendet werden, mit dem sogenannte Prozessorpools erstellt werden können. Den Prozessorpools können dabei Arbeitsaufträge übermittelt werden, welche dann gleichmäßig auf alle verfügbaren Prozesse verteilt werden. Setzt man hierbei die Zahl der Prozesse gleich mit der Anzahl an Prozessorkernen, so kann jeder Prozess auf seinem eigenen Kern laufen und die Arbeit wird parallel ausgeführt. Die Verwendung des *multiprocessing*-Modul bei der Auswertung der EQL-Anfrage, führte in unseren Tests zu einer 3-4 mal schnelleren Ausführung unter CPython. Während man unter CPython deutliche Geschwindigkeitsgewinne erreicht, führt die Nutzung von Prozessorpools unter PyPy zu einer deutlichen Verlangsamung der Ausführung. Bis jetzt können wir uns dieses Phänomen nicht gänzlich erklären. Vermutlich stellt das multiprocessing Modul unter PyPy einen derartigen Mehraufwand dar, sodass sich weitere Geschwindigkeitsgewinne erst bei einer sehr viel höheren Zahl von Kernen bemerkbar machen. Fakt ist, dass PyPy in letzter Zeit einen ohnehin anderen Ansatz zur parallelen Ausführung verfolgt. Hierbei wird auf eine parallele Ausführung auf Basis von STM (Software Transactional Memory) gesetzt. Nebenläufige Threads werden dabei als Transaktionen betrachtet, die im Falle von Konflikten jeweils zurückgesetzt und abermals ausgeführt werden können. Die Annahme dieses Ansatzes ist es, dass Konflikte relativ selten auftreten, man dafür aber den GIL effektiv aus PyPy entfernen kann. Dies erlaubt seitens des Interpreters implizite Parallelisierung, aber auch der Programmierer kann konfliktfreie Funktionen programmieren und parallel ausführen. STM in PyPy ist zum aktuellen Stand (2012) noch in einer frühen Entwicklungsphase, weswegen wir keine aussagekräftigen Tests mit dieser Version von PyPy durchführen konnten. Es gilt entsprechend abzuwarten, ob STM auf Systemen mit einer ausreichenden Anzahl an Kernen zu einer schnelleren Ausführung von Python Code und letztlich auch EQL-Anfragen führt. Verfolgt man den alternativen Ansatz die EQL-Anfragen innerhalb eines Extensionmodules auszuführen so gestaltet sich die Parallelisierung von Anfragen einfacher. Findet die Abarbeitung einer EQL-Anfrage vollständig in C statt, so kann dort direkt auf Basis von Betriebssystem-Threads parallelisiert werden. Solange man dabei auf Strukturen arbeitet, die keine Pythonobjekte sind, ist die Ausfüh-

rung nicht durch den GIL beeinträchtigt und man erreicht damit die maximale vom Betriebssystem erlaubte parallele Ausführung.

IV.8 Verwandte Arbeiten

IV.8.1 LINQ

LINQ *(Language-Integrated Query)* ist ein Framework, das die Schnittstelle für Anfragen innerhalb von .NET vereinheitlicht. Das heißt, dass man über LINQ Anfragen formulieren kann, die unabhängig von der darunter liegenden Datenbank sind. Dies kann beispielsweise eine Datenbank mit SQL-Schnittstelle oder auch eine dokumentbasierte XML-Datenbank sein. Die Syntax von LINQ ist der von der EQL sehr ähnlich, denn wir haben diese praktisch direkt aus LINQ übernommen. Alternativ zur Funktionsschreibweise, wie in der EQL, bietet LINQ auch noch eine Blocknotation für Anfragen an (Quelltext 16).

```
1 Dim cs = From c In Customers
2          Where c.Address.City = 'Seattle'
3          Order By c.Age
4          Select c.Name, c.Phone
```

Quelltext 16: *Blockbasierte Notation von LINQ*

Anders als die EQL ist LINQ innerhalb von C# implementiert, was im Gegensatz zu Python eine streng getypte Sprache ist. Aus diesem Grunde ist LINQ in der Lage, die formulierten Anfragen zu Datenbankanfragen zu kompilieren. Dies setzt allerdings voraus, dass alle Typen von Objekten zur Kompilierzeit vorhanden sind oder bestimmte Objekte durch retrospektive Programmierkonstrukte wie `getattr()` ausgelesen werden. Kann LINQ hier nicht die Datentypen zuverlässig inferieren oder bestimme Programmierkonstrukte nicht nach SQL kompilieren, so wird der Kompiliervorgang abgebrochen. Problematisch wird dies, wenn man generische Anfragen formuliert, wie etwa in Kapitel VII beschrieben. Die EQL-Anfragen sind nicht auf Typinferenz zur Compilezeit angewiesen. Stattdessen wird anhand des Lambda-Tracing bestimmt, welche Operationen auf welchen Objekten innerhalb der Datenbank ausgeführt werden sollen. Dies kommt allerdings einher mit den bereits beschriebenen Einschränkungen, wie beispielsweise dem Tracing von Funktionen.

```
1 var results =
2      SomeCollection
3         .Where(c => c.SomeProperty < 10)
4         .Select(c => new {c.SomeProperty, c.OtherProperty});
```

Quelltext 17: *Funktionsbasierte Notation von LINQ*

IV.9 Fazit

In dieser Arbeit haben wir eine Lösung zur Formulierung und Ausführung von schnellen analytischen Anfragen innerhalb der SHD untersucht. Hauptziel war es dabei, eine schnelle Ausführung zu gewährleisten, obwohl der Nutzer alle Anfragen in Python formuliert. Gleichzeitig sollten sich die Anfragen nahtlos in Python einbetten. Unsere vorgestellte Lösung ist die EQL-Anfragensprache, welche eine deklarative, mengenorientierte Syntax besitzt. EQL-Anfragen generieren Anfragenpläne, die dann ausgewertet und ausgelesen werden können. Im Kern unseres Bestrebens, die Ausführung zu beschleunigen, liegt das Lambda-Tracing, welches uns ermöglicht, die Auswertung von Anfragen an den Referenzobjekten vorbei, direkt auf den Spalten der Datenbank auszuführen. Zudem haben wir die Möglichkeit, die Laufzeitumgebung, in der wir die Traces auswerten, zu wechseln. Der dritte Bestandteil unserer Lösung ist die Ausführung innerhalb des PyPy Interpreters, was sogar eine gänzlich in Python stattfindende Auswertung einer EQL-Anfrage schneller ausführt als gängige Hauptspeicherdatenbanken wie SQLite.

Teil V

Programmierkonzepte für Benutzeroberflächen von modellintensiven Anwendungen

V.1 Einleitung

Geschäftsanwendungen unterstützen die organisatorische Arbeit in einem Unternehmen. Sie helfen, allgemeine betriebliche Prozesse wie Personal- oder Warenwirtschaft und Kundenverwaltung zu vereinfachen. Da sie oft viele Aspekte der betrieblichen Abläufe umfassen, können sie ein umfangreiches Datenmodell besitzen und damit sind sie für uns modellintensiv. Da die Verwaltung der Daten im Vordergrund steht, macht das reine Bearbeiten von Datensätzen einen großen Teil der Funktionalität aus. Für viele Teile des Datenmodells wird also ähnliche Funktionalität benötigt. Entsprechend ähneln sich die Benutzeroberflächen sowohl im Aussehen, als auch in den möglichen Interaktionen. Doch selbst mit modernen Rahmenwerken zur Entwicklung von Webanwendungen ist die Beschreibung solcher, sich ähnelnden, Oberflächen aufwendig. Die Entwickler müssen Ansichten komplett neu umsetzen oder bestehende kopieren, nur um einen kleinen Unterschied auszudrücken. Dadurch entsteht für jedes Modell eine redundante Beschreibungen der gleichen Ansicht. Zum Beispiel existiert eine Umsetzung für die Übersichtstabelle der Kunden, der Produkte, der Aufträge und weitere. Redundante Beschreibungen erschweren die Wartung und Erweiterbarkeit der Anwendung. Diese Probleme gelten nicht nur für Geschäftsanwendungen, sondern für Anwendungen mit einem umfangreichen Datenmodell und einem großen Anteil an Standardfunktionalität im Allgemeinen.

Im Weiteren werden wir kurz die besonderen Anforderungen der Entwicklung von Oberflächen für Anwendungen mit vielen Modellen erläutern und grundlegende Begriffe klären. Anschließend untersuchen wir die prinzipiellen und praktischen Probleme bei der Entwicklung von Oberflächen mit vielen Modellen (Abschnitt 2). Unsere Lösung begegnet diesen Schwierigkeiten, indem sie statt technischer Abstraktionen, spezielle Abstraktionen für die Entwicklung solcher Oberflächen zur Verfügung stellt. Im Abschnitt 3 beschreiben wir die Grundideen unserer Lösung und deren Umsetzung. Wir bewerten unseren Ansatz anhand der festgestellten prinzipiellen und praktischen Probleme und schlagen vor, wie man die Entwicklung von modellintensiven Anwendungen weiter verbessern kann (Abschnitt 4). Im Abschnitt 5 ordnen wir unsere Lösung in verwandte Ansätze ein. Abschließend fassen wir die Ergebnisse dieser Arbeit zusammen (Abschnitt 6).

V.1.1 Architektur von Geschäftsanwendungen

Geschäftsanwendungen unterstützen die betrieblichen Abläufe in einem Unternehmen. Ein wichtiger Vorteil ist, dass viele Menschen direkt auf synchronisierten Daten arbeiten können. Darum sind Geschäftsanwendungen in der Regel mit verteilten Architekturen implementiert. Der Server synchronisiert und

verarbeitet dabei die Daten. Die Clients zeigen die Informationen an und bieten eine Schnittstelle, über die der Benutzer die Daten verändern kann. Mit dem Aufkommen mobiler Geräte müssen diese Anwendungen außerdem zunehmend mehrere Zielplattformen gleichzeitig unterstützen.

Im Mittelpunkt solcher Anwendungen steht weniger die Verarbeitung der Daten durch Menschen, als viel mehr die reine Verwaltung von Daten. Darum machen CRUD-Funktionen einen großen Teil der Funktionalität aus. CRUD bezeichnet die grundlegenden Operationen zum Verwalten von Daten: Erstellen (Create), Lesen (Read), Verändern (Update) und Löschen (Delete). [18]

V.1.2 Benutzeroberflächen von Geschäftsanwendungen

Der große Anteil an CRUD-Funktionen in der Funktionalität von Unternehmenssoftware führt zu sehr ähnlichen Ansichten in der Benutzeroberfläche. Diese Kontinuität ist aus Benutzersicht wünschenswert, da sie den Umgang mit dem gesamten System erleichtert. Einmal gelernt sind Strukturen und Interaktionen für das ganze System gültig. Bietet man als Entwickler der Software besondere Interaktionen konsistent über mehrere Softwareprodukte hinweg an, entsteht zudem ein Wiedererkennungswert.

Beispielsweise ähnelt die Übersicht aller angestellten Mitarbeiter der Übersicht über alle offenen Verträge in zweierlei Hinsicht. Zum Einen sind sie sich strukturell ähnlich. Die Übersicht könnte als Tabelle angezeigt werden, die in jeder Zeile eine Entität enthält. In den vorderen Spalten stehen die Attribute und in den hinteren Spalten stehen Symbole für die möglichen Aktionen pro Entität.

Zum anderen gleichen sie sich in den möglichen Interaktionen. Die Tabelle kann zum Beispiel gefiltert oder durchsucht werden. Außerdem bietet die Übersicht die Möglichkeit, eine neue Entität des angezeigten Modells anzulegen. In den konkret dargestellten Informationen unterscheiden sich die Darstellungen aber. Die Übersicht über Personen zeigt Name und Abteilung von Personen, die Vertragsübersicht jedoch Datum und Kunde. Diese konkreten Konfigurationen der Ansichten, wie die Menge der anzuzeigenden Attribute, sind pro Modell verschieden.

V.1.3 Anwendungsentwicklung mit aktuellen Rahmenwerken zur Webentwicklung

Aktuelle Rahmenwerke zur Entwicklung von Web-Anwendungen bieten größtenteils Abstraktionen für technische Konzepte. Beispiele für solche Rahmen-

werke sind Ruby On Rails, Java Spring oder ASP.NET MVC. Da diese Rahmenwerke eine große Bandbreite von Anwendungen unterstützen wollen, stellen sie allgemeine technische Abstraktionen zur Verfügung. Zum Beispiel besitzen alle drei Rahmenwerke einen Weg, den Aufruf einer URL auf einen Methodenaufruf abzubilden. Grafische Benutzeroberflächen werden unter Anderem mit sogenannten *Template Engines* beschrieben. Diese beschreiben Ansichten mit *Ansichtsvorlagen*, welche aus einer Mischung aus HTML und Quelltext in der Sprache der Anwendung bestehen.

V.1.4 Grundlegende Architektur: MVC

Der Architekturstil MVC ist verbreitet für komplexe Anwendungen. Im weiteren Teil der Arbeit beziehen wir uns auf Teile dieser Architektur. MVC ist eine Softwarearchitektur, die die Anwendung in drei Bereiche unterteilt: *Model*, *View* und *Controller* [13]. Diese Architektur eignet sich für Anwendungen, die ein gemeinsames Modell und eine gemeinsame Logik haben, aber unterschiedliche Ansichten auf das Modell anbieten.

Das ermöglicht die Architektur, indem sie die drei Teile weitestgehend von einander trennt. Jeder der drei Teile soll unabhängig von den anderen beschrieben werden können. Der *Controller* enthält die Logik, wie die Anwendung sich bei welchen Interaktionen verhalten soll. Das *Modell* enthält die *Domänenlogik*, die zum Beispiel festlegt, welche Attribute ein Produkt hat oder wie der Preis eines Produkts berechnet wird. Die *Views* definieren Ansichten, über die der Benutzer die Daten des Modells anzeigen und verändern kann. Diese Ansichten schicken Informationen über Interaktionen an den Controller. Dieser verändert daraufhin die Daten und das Modell informiert die Ansichten über Änderungen. Diese passen sich dann an die veränderten Daten an. Das Datenmodell bezeichnet in dieser Arbeit die Gesamtheit aller Modelle. Ein Modell ist die Beschreibung einer Art von Domänenobjekten. Technisch gesehen sind diese Modelle meist Klassen.

V.1.5 Strukturen und Interaktionen der Ansicht

Im Weiteren beziehen wir uns auf drei Bestandteile von Benutzeroberflächen: Ansichten, Strukturen und Interaktionen. Eine Ansicht ist eine Zusammenstellung von graphischen Bauteilen auf dem Bildschirm. Eine einzelne Webseite ist beispielsweise eine Ansicht, eine andere Webseite der selben Anwendung wäre ein weitere Ansicht. Ansichtsstrukturen, im Weiteren kurz Strukturen, beschreiben, wie Informationen dem Benutzer konkret angezeigt werden. Dazu gehören unter Anderem Listen, Formulare und auch Fließtext. Technisch gesehen beschreibt zum Beispiel HTML eben solche Strukturen.

Interaktionen beschreiben, wie man die Ansicht solcher Informationen und die Informationen selbst verändern kann. Eine Interaktion, die die Daten an sich verändert, ist zum Beispiel, dass der Anwender das Datum einer Bestellung ändert. Eine mögliche Interaktion, die die Ansicht verändert, wäre das Sortieren einer Tabelle nach einer Tabellenspalte.

V.2 Probleme bei der Entwicklung der Benutzerschnittstellen von modellintensiven Geschäftsanwendungen

Die Benutzerschnittstellen von Geschäftsanwendungen sind geprägt von ähnlichen Strukturen und Interaktionen über alle Modelle hinweg. Existieren in der Anwendung viele Modelle, entstehen mit den aktuellen Rahmenwerken Schwierigkeiten in der weiteren Entwicklung der Anwendung. Das zugrunde liegende Problem ist, dass das Verhalten der Anwendung in technischen Beschreibungen implizit formuliert ist. Das liegt an den Abstraktionen der Rahmenwerke, die größtenteils technisch sind. Aus diesem grundlegenden Problem erwachsen praktische Probleme in Wartung und Erweiterbarkeit. Sie entstehen hauptsächlich daraus, dass Entwickler Änderungen für jedes Modell separat durchführen müssen. Bei einer großen Anzahl von Modellen wird die Wartung bestehender Funktionalität aufwendiger und Inkonsistenzen können entstehen. Die Erweiterung des Systems führt dazu, dass bereits bestehende Strukturen und Interaktionen für ein neues Modell neu implementiert werden müssen. Soll dieselbe Anwendung auf andere Zugangswege portiert werden, zum Beispiel als mobile Variante, entsteht unnötiger Aufwand. Das liegt an Informationen die in plattformspezifischem Quelltext formuliert sind, aber eigentlich unabhängig vom Zugangsweg sind. Anhand eines Beispiels lassen sich die aufgeführten grundlegenden und praktischen Probleme erkennen.

V.2.1 Beispiel

Das Beispiel umfasst zwei Übersichtstabellen, jeweils eine für die Menge der Produkte (siehe Quelltext 18) und eine für die Menge der Kunden (siehe Quelltext 19). Beide zeigen nur wenige Attribute des Modells. Die Tabellen sollen beide filterbar sein. Die Tabelle der Produkte soll sortierbar sein, die Tabelle der Kunden aber nicht, da sie vorsortiert ist, nach Wichtigkeit des Kunden. Die Ausschnitte aus dem Quelltext zeigen nur die Tabellen selbst. Aus Gründen der Übersichtlichkeit fehlen die Überschrift und ein Button zum Erstellen von Entitäten. Die Produktübersicht beinhaltet eine Spalte, die Tags des Produkts anzeigt, die durch Kommata getrennt in einer Zeichenkette gespeichert sind.

```
<table class="sortable filterable" id="products_overview">
<thead>
  <tr>
    <th>Produktname</th>
    <th>Tags</th>
    <th>Preis</th>
    <th><!-- Spalte mit Anzeige-Button --></th>
  </tr>
</thead>
  % for product in products:
    <tr>
      <td>${product.product_name}</td>
      <td>${product.price}</td>
      <td><div class="tags">${product.tags}</div></td>
      <td><a href="/products/${product.id}">
        <img src="/contents/show.png" />
      </a></td>
    </tr>
  % endfor
</table>
```

Quelltext 18: *Übersichtstabelle für Produkte*

```
<table class="filterable" id="customers_overview">
<thead>
  <tr>
    <th>Kunde</th>
    <th>Mitarbeiterzahl</th>
    <th>Sitz (Land)</th>
    <th><!-- Spalte mit Anzeige-Button --></th>
  </tr>
</thead>
  % for customer in customers:
    <tr>
      <td>${customer.customer_name}</td>
      <td>${customer.number_of_employees}</td>
      <td>${customer.country}</td>
      <td><a href="/customers/${customer.id}">
        <img src="/contents/show.png" />
      </a></td>
    </tr>
  % endfor
</table>
```

Quelltext 19: *Übersichtstabelle für Kunden*

V.2.2 Implizite technische Beschreibung der Funktionalität

Aktuelle Rahmenwerke zur Web-Entwicklung ermöglichen Entwicklern zu formulieren, wie eine Anwendung funktioniert. Sie ermöglichen es nicht, explizit zu formulieren, welche Funktionalität die Anwendung zur Verfügung stellen soll. Die Funktionen der Anwendung werden implizit durch die technische Umsetzung beschrieben. Im Beispiel wird durch Ansichtsvorlagen unter Anderem implizit beschrieben, das die Anwendung Übersichten über Entitäten als Tabellen anzeigt. Diese implizit formulierten Informationen über Funktionalität finden sich innerhalb der Strukturen und Interaktionen, über Ansichten hinweg als auch zwischen Ansichten und dem Modell.

Implizite Definition von Strukturen und Interaktionen Die beiden Ausschnitte im Beispiel zeigen die gleiche Struktur. Es ist eine Tabelle, die eine Menge von Entitäten anzeigt. In den vorderen Spalten stehen Attribute und in der letzten ein Symbol, dass zu einer Detailseite für die entsprechende Entität führt. Der tatsächliche Unterschied liegt nur in der Menge der angezeigten Entitäten und den entsprechenden Attributen.

Dieses Beispiel zeigt, wie ein Struktur implizit durch technische Beschreibungen definiert wird. Beide Ausschnitte formulieren die Struktur der Übersichtstabelle, allerdings für das entsprechende Modell. Obwohl die Struktur allgemein für alle Übersichtstabellen gilt, ist sie nie explizit so definiert. Darum muss sie für jedes Modell neu definiert werden. Das beide Beschreibungen auf der allgemeinen Struktur einer Übersichtstabelle aufbauen zeigt sich am duplizierten Quelltext. Das selbe gilt auch für die Beschreibung von Interaktionen.

Implizite Informationen über Ansichten Ebenso implizit festgehalten ist, wie eine Ansicht allgemein aussehen soll und welche Funktionalitäten speziell für ein Modell aktiviert sind. Die Bedeutung der zwei Ansichten aus dem Beispiel ist: Zeige eine Übersicht aller Entitäten an. Da die Übersichtsansichten über die gesamte Anwendung hinweg gleich aussehen sollen, sind beide als Tabelle implementiert. Diese Information, dass Übersichten als Tabellen angezeigt werden sollen, ist aber wieder nur implizit durch alle Ansichtsvorlagen definiert. Genauso ist erst durch die Vorlagen definiert, dass alle Übersichtstabellen sortierbar und filterbar sein sollen. Die allgemeine Definition, wie eine Ansicht über die Anwendung hinweg aussehen soll, ist damit über alle Ansichtsvorlagen dieser Ansicht verteilt.

Die konkrete Konfiguration der Kundenübersicht ist zudem an deren technische Beschreibung gebunden. Das die Übersichtstabelle für Kunden nicht sortierbar sein soll, ist in einer HTML-Klasse festgehalten. Damit wird durch tech-

nische plattformspezifische Abstraktionen beschrieben, welche Funktionalität für welche Modellansicht aktiviert ist.

Implizite Informationen zwischen Ansichten und Modell Auch Informationen über das Datenmodell sind über die Anwendung verstreut. Zum Beispiel ist die Rolle eines Modellattributs in der Darstellung wieder erst durch HTML-Elemente in den Vorlagen formuliert, in denen das Attribut angezeigt werden soll. Im Beispiel sieht man das am `tag` Attribut der Produkte. Erst in der Ansichtsvorlage wird durch ein HTML-Element und eine entsprechende HTML-Klasse definiert, dass dieses Attribut mehrere Schlagwörter darstellt, die auf eine bestimmte Art dargestellt werden sollen. Diese Information muss in jeder Ansicht, die die Tags darstellt wieder formuliert werden. Außerdem muss jedes Modell mit einem `tag` Attribut in seinen Ansichten die Information enthalten, dass dieses Attribut auf eine bestimmte Art dargestellt werden soll.

V.2.3 Praktische Probleme in der Entwicklung

Das prinzipielle Problem der impliziten Formulierung von Informationen führt zu praktischen Problemen in der Entwicklung von Anwendungen. Unter anderem kommt es zu Problemen in der Wartbarkeit und Erweiterbarkeit. Häufig entsteht zusätzlicher Aufwand, da eine Änderung mehrfach von Hand durchgeführt werden muss. Außerdem können unvollständige Änderungen zu Inkonsistenzen in der Darstellung führen.

Wartbarkeit Softwarewartung ist definiert als „die Veränderung eines Softwareprodukts nach dessen Auslieferung, um Fehler zu beheben, Performanz oder andere Attribute zu verbessern oder Anpassungen an die veränderte Umgebung vorzunehmen.“ [15] Die Wartung der Benutzerschnittstelle ist durch die mehrfache implizite Formulierung von Informationen aufwändig. Ein Aspekt muss möglicherweise an mehreren Stellen im System geändert werden. Außerdem können durch verstreute und duplizierte Informationen Inkonsistenzen in der Benutzeroberfläche auftreten.

Möchte man zum Einen eine Struktur oder Interaktion konsistent über die ganze Anwendung hinweg ändern, so steigt der Aufwand proportional zur Anzahl der Modelle. Denn für jedes Modell muss dieselbe Änderung durchgeführt werden. Sollen die Übersichtstabelle im Beispiel etwa Zeilennummern vor jeder Zeile haben, müssten beide Übersichtstabellen von Hand angepasst werden. Bei zwei Tabellen hält sich der Aufwand in Grenzen, doch er steigt mit jedem weiteren Modell.

Wenn man zum Anderen das zugrunde liegende Datenmodell ändert, zieht das immer Änderungen in der Benutzerschnittstelle nach sich. Die Änderung eines Teils des Systems zieht somit die Anpassung eines anderen Teils nach sich. Wenn das Datenmodell für Kunden beispielsweise ein neues Attribut „Jahresumsatz“ bekommt, so müsste im Beispiel die Übersichtstabelle angepasst werden. Entsprechend müssten auch die Detailansichten und Formulare angepasst werden. Die MVC-Architektur entkoppelt zwar die Benutzerschnittstelle und das Datenmodell, dadurch ist es aber nur möglich, mehrere Ansichten für das gleiche Modell zu beschreiben. Diese Trennung ermöglicht nicht automatisch, dass eine Ansicht für mehrere Modelle gilt.

In beiden Fällen entsteht unnötiger Aufwand, da sehr ähnliche Änderungen mehrmals von Hand durchgeführt werden müssen. Außerdem könnten dem durchführenden Entwickler nicht alle Stellen bewusst sein, die er ändern muss. Wird die Änderung nicht vollständig durchgeführt, dann entstehen inkonsistente Strukturen und Interaktionen. Passt ein Entwickler die Übersichtstabelle der Produkte so an, dass die Symbole für Aktionen in den ersten Spalten stehen, muss er auch die Tabelle für Kunden anpassen. Vergisst er diese Änderung, so ist die Struktur der Übersichtstabellen im System nicht mehr konsistent, was zu Problemen in der Benutzbarkeit führen kann.

Erweiterbarkeit Die Erweiterbarkeit von Softwaresystemen ist „die Einfachheit, mit der ein System an neue Anforderungen angepasst werden kann“ [19]. Fehlende Abstraktionen verringern die Erweiterbarkeit durch zusätzlichen Aufwand, da Veränderungen mehrfach von Hand vorgenommen werden müssen. Das gilt sowohl für die Erweiterung einer bestehenden Anwendung als auch für ihre Portierung.

Die Erweiterung des Datenmodells um ein neues Modell führt dazu, dass Standardfunktionen für das neue Modell kopiert und angepasst werden müssen. Das bedeutet in MVC-Rahmenwerken, dass ein Controller mit mehreren Methoden neu angelegt und für jede Ansicht eine Ansichtsvorlagen-Datei erstellt werden muss. Der eigentliche Inhalt dieser Tätigkeit ist die Aktivierung von Standardfunktionen für das neue Modell. Zudem werden mit den Änderungen die Abweichungen für das neue Modell erfasst. Soll unser Beispielsystem auch noch Bestellungen umfassen, müssten die Übersichtstabelle ein weiteres Mal kopiert und für Bestellungen angepasst werden.

Die Formulierung von Funktionalität durch technische Abstraktionen behindert zudem die Portierung einer Anwendung. Die Informationen über Funktionalität sind direkt an eine technische Umsetzung gebunden. Zum Beispiel soll eine Anwendung zusätzlich zur Weboberfläche eine Desktop-Oberfläche besitzen, die sich in den Interaktionen so weit wie möglich gleichen. In diesem Fall müssen die Entwickler zu erst einmal die bestehenden Ansichtsvorlagen lesen,

um zu verstehen, welche Funktionalität eine Ansicht bieten soll. Beispielsweise könnten sie so herausfinden, dass die Tabelle der Kunden nicht sortiert werden darf. Dann müssen sie versuchen diese Funktionalität möglichst vollständig auf die neue Plattform zu übertragen.

V.3 Domänenspezifische Programmierkonzepte für Geschäftsanwendungen

Die praktischen Probleme in Wartbarkeit und Erweiterbarkeit entstehen aus der technischen Beschreibung von Funktionalität, die dadurch nur implizit erfasst wird. Wir gehen dieses Problem an, indem wir Abstraktionen für die Domäne der Benutzeroberflächen schaffen, mit denen man vormals implizite Informationen explizit formulieren kann. Dazu trennen wir die Informationen darüber was die Anwendung tun soll von der Umsetzung dieser Funktionalität.

CRUD-Anwendungen sollen mit unserer Lösung mit wenig Aufwand erstellt werden können. Gleichzeitig soll unsere Lösung aber nicht auf solche Ansichten eingeschränkt sein, sondern möglichst viele Ansichten beschreiben können. Darum war es unser Ziel, allgemeine Formulierungen zu schaffen mit denen man große Teile der Anwendung schnell beschreiben kann. Gleichzeitig wollten wir genug Flexibilität erhalten, dass die Entwickler auch Ansichten außerhalb der CRUD-Funktionalität beschreiben können.

Wir haben dazu ein Rahmenwerk als Prototypen entworfen und umgesetzt, der solche Beschreibungen ermöglicht. Da zur Definition einer Anwendung auch ihr Verhalten gehört, können Entwickler mithilfe des Rahmenwerks auch Abläufe allgemein beschreiben (Kapitel VI). In dieser Arbeit beschreiben wir, wie unser Prototyp die Beschreibung von Ansichten ermöglicht mithilfe von zusätzlichen Informationen über das zugrunde liegende Datenmodell, modellübergreifenden Konzepten für Strukturen und Interaktionen und Wege der Aktivierung dieser Konzepte.

Modell angereichert mit Darstellungsinformationen Die Grundlage des Rahmenwerks bilden Informationen zur Darstellung eines Modells. Diese werden direkt im Modell formuliert. Die Attribute werden um zusätzliche Informationen erweitert, wie zum Beispiel, ob es in Formularen schreibbar oder nur lesbar ist. Dabei beschreibt das Modell auch, welches Attribut zum Beispiel zur Identifikation einer Entität verwendet werden kann. Durch solche Zusatzinformationen entsteht eine Art Maske für das Modell. Diese bietet eine einheitliche Schnittstelle für Metainformationen zur Darstellung. Diese Schnittstelle können andere Komponenten des Rahmenwerks nutzen,

um beispielsweise zu erfahren welches Attribut eines Modells als Bezeichner fungieren kann.

Anzeigen von Attributen Auf Basis dieser Zusatzinformationen entscheidet der Prototyp, wie er ein Attribut darstellt. Er bestimmt sowohl die reine Darstellung als auch die Eingabeelemente zur Bearbeitung des Attributs. Der Entwickler kann zum Einen die Standarddarstellung des Attributs anpassen. Zum Anderen kann er für bestimmte Kontexte alternative Darstellungsarten angeben. Das kann unter Anderem nötig sein, wenn ein Attribut eines Modells auf eine ganz besondere Art dargestellt werden soll.

Struktur- und Interaktionskonzepte Ein Konzept im Allgemeinen vereint Gemeinsamkeiten einer Menge von Objekten. Die Struktur- und Interaktionskonzepte beschreiben entsprechend Gemeinsamkeiten mehrerer ähnlicher Strukturen oder Interaktionen. Unser Rahmenwerk bietet solche Konzepte und die Möglichkeit als Entwickler weitere zu definieren. Sie sollten so allgemein formuliert werden, dass sie für ein beliebiges Modell gelten. Sie greifen dafür auf die allgemeinen Metainformationen zur Darstellung des Modells zurück. Genauer gesagt, fixieren sie nur die Invariante der Lösung, die sie beschreiben. Zum Beispiel beschreibt eine Übersichtstabelle nur, dass die Entitäten darin in Tabellenform organisiert werden. Die angezeigten Attribute und deren Reihenfolge stammen ausschließlich aus dem Modell. Außerdem stellen sie die Unterschiede zwischen den verschiedenen Ausprägungen eines Konzepts als Variationspunkte zur Verfügung, deren Ausprägung der Entwickler bestimmen kann. Zum Beispiel können in einer Übersichtstabelle Zeilennummern aktiviert oder deaktiviert werden.

Beschreibung von Ansichten Mit diesen Bausteinen kann man dann die Anwendung definieren. Für alle CRUD-Aktionen können Standardansichten mithilfe der Struktur- und Interaktionskonzepte definiert werden. Anschließend reicht es, anzugeben, welche Modelle CRUD-Aktionen anbieten sollen. Möchte der Entwickler eine Funktion für ein bestimmtes Modell anpassen, reicht es, die Unterschiede zur Standardkonfiguration zu formulieren.

Entwicklung mit dem Prototypen Die Entwicklung mit unserem Prototypen teilt sich für uns auf zwei Rollen auf. Ein Entwickler nimmt die Rolle des Anwendungsentwicklers ein. Er übersetzt die Anforderungen an das System in Modelle und Konzepte und konfiguriert diese. Seine Variationspunkte sind: Strukturen, Interaktionen, Standardansichten, individuelle Ansichten und Lokalisierung. Fehlt ihm dabei ein Konzept oder ein Variationspunkt, so übernimmt der Konzeptentwickler. Dieser kennt die Strukturen im Rahmenwerk

und stellt neue Konzepte zur Verfügung. Dieser Prozess des Erkennens und Extrahierens von Konzepten ist iterativ. Während der gesamten Entwicklung sollte darauf geachtet werden, Gemeinsamkeiten in Konzepte zu fassen und Variationspunkte, wo nötig, hinzuzufügen.

Technische Grundlagen Unser Prototyp ist in Python geschrieben. Er basiert auf dem Pyramid Rahmenwerk für Webanwendungen[16]. Wir nutzen die vorhandene MVC-Struktur als Ausgangspunkt für die Implementierung. Das Herzstück bildet ein einziger Controller, der alle Anfragen behandelt. Er extrahiert aus einer angefragten URL, welche Aktion welches Modells ausgeführt werden soll. Dann ruft er zuerst die entsprechenden Funktionen auf und generiert anschließend aus der zugehörigen Ansichtsdefinitionen die nächste Ansicht. Unser Prototyp generiert keinen Quelltext sondern basiert auf einem Objektmodell das aus den oben beschriebenen Komponenten erstellt wird. Somit könnten auch zur Laufzeit Anpassungen an der Anwendung vorgenommen werden, ohne das Ressourcen neu geladen werden müssen. Graphische Elemente werden mit so genannten *Renderern* erstellt. Diese bekommen die Spezifikation eines graphischen Elements übergeben und erzeugen es entsprechend. Die Renderer für Webseiten erstellen zum Beispiel Zeichenketten, die HTML enthalten.

V.3.1 Modell angereichert mit Darstellungsinformationen

Das Datenmodell bietet neben den Daten und der Domänenlogik Informationen über die Darstellung. Viele dieser Informationen entstehen aus der vollständigen Beschreibung eines Attributs. Informationen wie Assoziationsart oder Wertemenge sind je nach Programmiersprache oder weiteren beteiligten Rahmenwerken bereits gegeben. Allerdings werden Informationen über die Rollen der Attribute in der Kommunikation mit dem Benutzer nicht erfasst.

Je nach Kontext hat ein Attribut eine andere Funktion. Diese Funktion kann über den reinen Inhalt hinaus gehen. In der Darstellung eines Produkts in einer Webansicht erfüllt sein Name zum Beispiel die Funktion eines Bezeichners, den ein Mensch als Identifikationsmerkmal für das Produkt versteht. Genauso kann eine ganze Gruppe von Attributen eine bestimmte Funktion erfüllen. Sie stellen eine Ansicht auf ein Objekt dar. Zum Beispiel werden in einer Übersicht andere Attribute angezeigt als auf einer Detailseite. Eine bestimmte Menge von Attributen erfüllt also beispielsweise den Zweck, einen schnellen Eindruck von einer Entität zu bekommen.

[16] `http://pyramid.readthedocs.org/` (Stand: 12.06.2012)

Außerdem sagt der technische Datentyp eines Attributs unter Umständen nichts über seine visuelle Kommunikation aus. Zum Beispiel könnte eine Zeichenkette durch Kommata getrennte Schlagwörter darstellen. Um die Aufnahme der Information zu erleichtern, soll die Anwendung Schlagwörter aber aufbereitet darstellen. Beispielsweise könnten Schlagwörter getrennt in kleinen Rechtecken dargestellt werden. Dazu muss die Anwendung aber wissen, dass dieses Attribut Schlagwörter beinhaltet.

```
@implicit_init
class Product(ImplicitModel):
  @classmethod
  def groups(cls, group, insert_group, attribute):
    super(cls, cls).groups(group, insert_group, attribute)

    group("overview",
          attribute("view_title", source = "name",
                    view_type = "view_title",
                    not_empty = True),
          attribute("price", type = float,
                    view_type = "currency",
                    not_empty = True),
          attribute("customers", type = Customer,
                    association = "many",
                    association_dependent = False),
          insert_group("standard_actions"),
          action("create_campaign"))

    group("details",
          insert_group("overview"),
          attribute("weight", type = int
                              view_type = "weight",
                              not_empty = True)
```

Quelltext 20: *Die Beschreibung des Produkt-Modells. Sie beschreibt alle für die Darstellung nötigen Informationen. Die Overview-Gruppe enthält die Definition des* `view_title`*. Auch Assoziationen werden hier bereits vollständig beschrieben. Die Overview-Gruppe enthält zudem die Standardaktionen. Die Details-Gruppe enthält die Overview-Attribute und fügt weitere Attribute hinzu. Die Sortierung der Attribute gibt ihre Position in der Darstellung an.*

Umsetzung: Attributgruppen und erweiterte Attribute Die komplette Formulierung der Darstellungsinformationen befindet sich in unserem Prototypen in der Definition des Modells, also in einer Klasse, wie in Quelltext 20 zu sehen. Attribute können im Kontext einer Gruppe deklariert werden. Gruppen können Namen tragen und andere Gruppen beinhalten. Es gibt zwei Standard-

gruppen: Overview und Details. Jedes Modell sollte Attribute in diesen Gruppen definieren, damit Standardkonzepte das Modell darstellen können. Außerdem können hier Darstellungsrollen definiert werden. Eine wichtige Darstellungsrolle eines Attributs ist der `view_title`, der als menschenlesbarer Bezeichner für eine Entität dient. Die Rolle eines Attributs ergibt sich direkt aus dem Namen. Zum Beispiel ist der `view_title` als normales Attribut definiert, dessen Wert aus einem anderen Attribut stammt. Außerdem können weitere Informationen zum Attribut angegeben werden: Wertemenge, Assoziationsart, Typ in der Darstellung, lesbar/schreibbar. Darstellungstypen sind zum Beispiel: `tags` oder `country`.

Neben Attributen können auch Aktionen zu Gruppen hinzugefügt werden. Dieser werden vorher definiert, wie in Kapitel VI beschrieben. Dann können sie über ihren Namen hinzugefügt werden. In Zeile 16 in Quelltext 20 wird beispielsweise die `create_campaign` Aktion hinzugefügt.

Technische Umsetzung Die zusätzlichen Informationen sind in unserem Prototyp in einem Metamodell für Attribute und Attributgruppen umgesetzt. Die Klassenannotation `@implicit_init` liest die Beschreibung der Attribute und legt entsprechende Attributgruppen- und Attribut-Objekte an. Diese sind erst einmal als Prototypen in Klassenvariablen gespeichert und fungieren als Vorlagen gemäß dem Prototyp-Entwurfsmuster [13]. Die Attribut-Objekte sind noch nicht an eine Instanz gebunden. Erst wenn man eine Instanz einer Modellklasse nach einem Attribut fragt, erhält man ein gebundenes Attribut-Objekt, das nun an die Instanz gebunden ist, von der man es erhalten hat. Dazu kopiert die Instanz den passenden Prototypen und bindet ihn an sich. Über ein solches gebundenes Attribut erhält man sowohl den gespeicherten Wert, als auch alle Metainformationen zum Attribut.

V.3.2 Anzeige von Attributen

Eine Sammlung von Attribut-Renderern bestimmt, wie ein Attribut dargestellt wird. Grundsätzlich ist für jeden Typ eine Art der Darstellung und ein Weg zur Bearbeitung definiert. Ändert man diese Definitionen, wird das Attribut global anders dargestellt. Es gibt in Anwendungen auch durchaus Sonderfälle in denen ein Attribut eines Modells auf eine bestimmte Art dargestellt werden soll. Zum Beispiel soll der `view_title` für alle Modelle als Link dargestellt werden, aber für Kunden soll er zusätzlich je nach Wichtigkeit eingefärbt werden. Durch ein abgestuftes Nachschlagen der konkreten Definition ist es darum möglich, die allgemeine Darstellung für mehrere Arten von Sonderfällen zu überschreiben.

```
# Allgemeine Standarddarstellung eines Attributs
@renders(primitive = True)
class PrimitivePropertyWidget:
  def render(self, property, localizer):
    return str(property.value())

# Stellt den view_title dar
@renders(view_type = 'view_title')
class ViewTitlePropertyWidget(PrimitivePropertyWidget):
  def render(self, property, localizer):
    return '<a href="' + URLHelper.url_to(property.owner)
                       + '">'
                       + str(property.value()) + '</a>'

# Stellt ein Datumsattribut dar
@renders(view_type = 'date', scope = "read")
class DateShowPropertyWidget(PrimitivePropertyWidget):
  def render(self, property, localizer):
    return localizer.l(datetime.datetime. \
                fromtimestamp(property.value()))
```

Quelltext 21: *Drei Attribut-Renderer mit unterschiedlichen Prioritäten. Das* `PrimitivePropertyWidget` *stellt alle Attribute dar, für die kein anderer Attribut-Renderer greift. Es wandelt den Wert in eine Zeichenkette um. Das* `ViewTitlePropertyWidget` *erzeugt einen Link auf den Besitzer des Attributs. Das* `DateShowPropertyWidget` *erzeugt ein Datumsobjekt aus einem gespeicherten Zeitstempel und erzeugt daraus eine lokalisierte Zeichenkette.*

Umsetzung Um eine Darstellung für ein Attribut zu definieren, beschreibt man eine Klasse mit der Methode `render`, die das Attribut übergeben bekommt. Quelltext 21 zeigt verschiedene Attribut-Renderer für die Darstellung auf Webseiten. Je nach Plattform muss die Methode `render` eine HTML-Zeichenkette oder ein GUI-Widget zurückgeben. Der Anwendungsentwickler kann die Definition mit einer Annotation registrieren. Er kann dabei mehrere Bedingungen angeben: Typ des Attributs, schreibbar/lesbar, Rolle des Attributs und, ob der Renderer das Attribut ausführlich oder nur für einen Überblick darstellt. Bei der Formulierung der Konzepte übergibt der Entwickler das Attribut an den Attribut-Renderer, der dann das entsprechende GUI-Element erzeugt.

Technische Umsetzung Der Attribut-Dispatcher nutzt die Metainformationen aus dem Attribut-Objekt, um den passenden Attribut-Renderer auszuwählen. Ein Attribut-Renderer bekommt ein Attribut-Objekt übergeben und erstellt daraus ein konkretes Darstellungselement. Findet er keine passende Darstellung wählt er die Standarddarstellung, der den im Attribut gespeicherten Wert

in eine Zeichenkette umwandelt und anzeigt. Die Priorität ist dabei absteigend wie folgt:

1. ViewType des Attributs
2. Attributname und Modell
3. Assoziationstyp falls das Attribut eine Assoziation ist
4. Datentyp des im Attribut enthaltenen Wertes
5. Standarddarstellung für alle Attribute

V.3.3 Struktur- und Interaktionskonzepte

Die Elemente von Benutzerschnittstellen wiederholen sich in ähnlicher Form. Zum Beispiel kann es viele verschiedene Übersichtstabellen in einem System geben. Jede kann unterschiedliche Funktionen und Ansichten zur Verfügung stellen. Sie haben aber alle gemeinsam, dass sie pro Zeile eine Entität darstellen und das jede Spalte ein Attribut oder eine Funktion einer Entität beinhaltet. Welche Attributgruppe eine solche Tabelle anzeigt, unterscheidet sich von Tabelle zu Tabelle. Diese Eigenschaft stellt also eine Variation dieser Tabellen dar.

Mit den Struktur- und Interaktionskonzepten kann der Anwendungsentwickler solche ähnlichen Elemente in einem Baustein für Ansichten zusammenfassen. Er beschreibt, wie Daten angezeigt werden sollen und wie diese Ansichten verändert werden können. Diese Struktur- und Interaktionskonzepte stellen Elemente der Ansichten als eigenständige Bausteine zur Verfügung. Um unabhängig von einem bestimmten Modell zu sein, nutzen sie die Metadaten im Modell. Damit sind sie weder an ein bestimmtes Modell gebunden, noch an eine Plattform. Die Funktionalität, die sie zur Verfügung stellen, ist die Invariante über alle möglichen Variationen ihrer selbst. Damit ähnelt die Art der Information in den Konzepten der Idee von Mustern, wie sie in [1] beschrieben sind.

Umsetzung: Konkrete Konzepte Die primäre Dekomposition der Ansichten sind in unserem Prototyp die strukturellen Konzepte. Die Interaktionskonzepte sind pro strukturellem Konzept definiert. Es ist aber auch denkbar, diese getrennt zu formulieren. Der Anwendungsentwickler kann im Kontext einer Seite ein Konzept als Objekt instantiieren und konfigurieren, wie in Quelltext 22 zu sehen ist. Eine solche konfigurierte Instanz eines Konzepts ist kurz gesagt eine Konfiguration.

Die Konzepte sind unabhängig von einem tatsächlichen Modell formuliert. Sie greifen auf Modellobjekte nur über die abstrakte Schnittstelle für Darstellungsinformationen zu. Zum Beispiel nutzen Detailansichten den `view_title` eines

```
# Konfiguration einer Uebersichtstabelle
table = TableCollectionComponent()
table.model = Product
table.collection = list(Product.all())
table.add_property_group("overview")
table.sortable = True
table.filterable = True
table.paginatable = True

# Konfiguration eines Instanzeditors
editor = EditInstanceComponent()
editor.mode = "new"
editor.model = Customer
editor.instance = Customer()
editor.add_property_group("details")
```

Quelltext 22: *Konfigurationen sind Instanzen der Konzepte mit konkreter Belegung der Variationspunkte. Die Variationspunkte sind Instanzvariablen oder Methoden der Konfigurationen. Die konfigurierte Übersichtstabelle zeigt zum Beispiel alle Produkte mit den Overview-Attributen an. Die Tabelle ist zudem sortierbar, filterbar und in Seiten aufgeteilt. Der Instanzeditor erlaubt die Bearbeitung einer neuen Kunden-Entität. Dabei sollen die Detail-Attribute eingetragen werden, die auch die Overview-Attribute einschließen.*

Modells als Inhalt der Überschrift. Entsprechend zeigen die Übersichtsansichten für Webanwendungen über der Übersicht aller Entitäten eine Reihe von Buttons für die Aktionen, die auf dem Modell definiert sind.

Die Menge der Strukturkonzepte in unserem Prototyp umfasst Strukturen unterschiedlichen Umfangs. Zum Einen beinhaltet sie das Konzept ganzer Seiten. Diese werden allerdings auf eine andere Art beschrieben (siehe Abschnitt V.3.4). Eine feinere Granularität haben die Strukturen, die Entitäten anzeigen. Dazu gehören sowohl Übersichtstabellen, Listen oder ähnliches, als auch Detailseiten und Formulare.

Zum Anderen gibt es eine Layoutstruktur, die als Container für Strukturen dient. Sie erlaubt es, mehrere andere Strukturen auf einer Seite zu kombinieren und anzuordnen. Damit schaffen wir die Möglichkeit beliebige Seiten zusammen zu setzen. Wir können damit Mischungen aus statischen Inhalten und verschiedenen CRUD-Ansichten unterschiedlicher Modelle erstellen. Die Variationsmöglichkeiten unterscheiden sich zwischen den verschiedenen Arten von Strukturen. Strukturen, die eine Menge von Entitäten anzeigen, haben zum Beispiel den Variationspunkt `highlight_instance`. Damit kann man in einer Übersicht Entitäten hervorheben, die eine Bedingung erfüllen. Dazu konfiguriert der Anwendungsentwickler den Variationspunkt mit einer anonymen

Funktion, die zu einer übergebenen Entität angibt, ob sie die Bedingung erfüllt oder nicht. Die Interaktionskonzepte sind spezielle Variationspunkte, die das Verhalten der Struktur ändern. Zum Beispiel ist es möglich, Übersichtsstrukturen filterbar, sortierbar oder durchsuchbar zu machen.

Technische Umsetzung Die Konzepte bilden eine eigene Zwischenschicht in der Anwendung. Ein Konzept besteht erst einmal nur aus einer Klasse, die für alle Variationspunkte entsprechend Methoden besitzt. Ein Objekt dieser Klasse stellt eine konkrete Konfiguration des Konzepts dar. Diese allgemeine Konfiguration wird dann später einem Konzept-Renderer übergeben, der aus der Konfiguration die entsprechende Ansicht generiert. Der Konzept-Renderer kann dabei die entsprechenden Variationspunkte plattformspezifisch umsetzen. Trotzdem bleibt die Funktionalität erhalten. Die Konzept-Renderer für Webansichten sind beispielsweise mit Ansichtsvorlagen umgesetzt. Zu jedem Konzept gehört eine Vorlage, die entsprechende Konfigurationen umsetzen kann.

V.3.4 Beschreibung von Ansichten

Mit den Darstellungsinformationen und Konzepten kann nun die Anwendung vollständig auf der Ebene von Funktionalitäten beschrieben werden. Die Funktionalitäten für die meisten Modelle sind in Geschäftsanwendungen allerdings gleich oder zumindest ähnlich. Zum Beispiel benötigen sehr viele Modelle eine Übersichtsansicht. Dazu muss man einmal die Standardansicht für Übersichten definieren und diese dann für alle entsprechenden Modelle verwenden. Das gilt für alle CRUD-Operationen.

Allerdings können CRUD-Ansichten bestimmter Modelle Abweichungen von der Standardansicht enthalten. In Quelltext 23 sind zum Beispiel in der Übersicht aller Kunden, die Kunden mit mehr als 150.000 Mitarbeitern hervorgehoben. Um die Anwendung dennoch vollständig zu beschreiben, reicht es aus, neben den Standardansichten nur die modellspezifischen Abweichungen zu beschreiben.

Umsetzung: Standardansichten und inkrementelle Definitionen In unserem Prototyp können Standardansichten und modellspezifische Abweichungen beschrieben werden, wie in Quelltext 23 zu sehen. Für alle CRUD-Operationen sind zum Beispiel Standardansichten für je ein Modell definiert. An einer zentralen Stelle kann der Anwendungsentwickler beschreiben, für welche Modelle CRUD-Operationen möglich sind. In der Webansicht entstehen dann zum Beispiel entsprechende Menüeinträge.

```
# Definition der CRUD-Modelle
set_crud_models([ Product, Order, Customer ])

# Definition der Standarduebersicht
@standard_view_for("index")
def standard_index(values):
    model = values["model"]
    table = TableCollectionComponent()
    table.model = model
    table.collection = list(model.all())
    table.add_property_group("overview")
    table.sortable = True
    table.filterable = True
    return table

# Definition der inkrementellen Uebersicht fuer Kunden
@view_for("index", Customer)
def customer_index(config, values):
    config.paginatable = False
    config.highlight_instance("important", \
        lambda x: x.number_of_employees > 150000)
    return config
```

Quelltext 23: *Die drei Möglichkeiten, Ansichten für Modelle zu aktivieren oder zu beschreiben: Aktivierung von CRUD-Funktionalität für eine Menge von Modellen, Definition einer Standardansicht, Definition von Abweichungen von einer Standardansicht für das Modell Kunden.*

Der Entwickler definiert die Ansichten in Funktionen. Innerhalb der Methode kann er dann entsprechende Konzepte erstellen und für ein Modell konfigurieren. Mit der Methodenannotation `@standard_view_for` registriert man die Methode als Standardansicht oder mit `@view_for` als abweichende Ansicht. Bei Beiden gibt man den Namen der Ansicht an. Zu der abweichenden Ansicht gibt man zusätzlich an, für welches Modell diese Abweichungen gelten sollen. Mit `@view_for` können zudem Einzelansichten definiert werden, die nur für das angegebene Modell gelten und auch nicht auf einer Standardansicht basieren müssen.

Technische Umsetzung Sowohl die Standardansichten als auch die Abweichungen werden global verwaltet. Für beide existiert ein `Dictionary`. Die Methodenannotationen registrieren eine Ansicht im entsprechenden `Dictionary`. Die Standardansichten werden unter ihrem Namen abgelegt, die abweichenden Ansichten mit ihrem Namen und dem zugehörigen Modell.

Soll nun eine bestimmte Ansicht eines Modells gerendert werden, wird erst nach einer entsprechenden Standardansicht gesucht. Wurde eine gefunden, so wird sie zuerst ausgeführt. Der Rückgabewert ist die Standardkonfiguration der Ansicht. Anschließend wird im zweiten `Dictionary` gesucht, ob Abweichungen definiert sind. Falls ja, so wird der abweichenden Ansicht das Ergebnis der Standardansicht übergeben. Die angepasste Konfiguration geht dann an den Konzept-Renderer für den konkreten Zugangsweg.

V.4 Evaluierung

Ziel der Lösung war es, implizit formulierte Funktionalität explizit zu machen. Dies ermöglicht unsere Lösung für Strukturen und Interaktionen, die Beschreibung von Ansichten und die Darstellung von Attributen. Das löst angesprochene Probleme der Wartbarkeit und Erweiterbarkeit, sorgt aber auch für neue Herausforderungen im Entwurf von Software und im Entwicklungsprozess. Unsere Lösung ist ein erster Schritt hin zu einer domänenspezifischen Beschreibung von Anwendungen. Ähnlich zu der Beschreibung der Ansichten können noch weitere Aspekte von Geschäftsanwendungen mit domänenspezifischen Mitteln beschrieben werden.

V.4.1 Implizite Informationen

Das grundlegende Problem in der Beschreibung von Geschäftsanwendungen ist die implizite Formulierung von Funktionalität. Diese entsteht durch weitestgehend technische Abstraktionen der aktuellen Rahmenwerke. Unsere Lösung stellt domänenspezifische Elemente zur Verfügung, mit denen einige Aspekte nun explizit formuliert werden können.

Strukturen und Interaktionen Das angereicherte Modell und die Konzepte ermöglichen die anwendungsweit konsistente Definition von Strukturen und Interaktionen. Da die Konzepte nur über die Metainformationen über die Darstellung des Modells auf das Modell zugreifen, sind sie unabhängig von einem bestimmten Modell. Die Schnittstelle der Konzepte über Variationspunkte macht den Unterschied deutlich zwischen dem essentiellen Teil des Konzepts und seinen veränderbaren Teilen.

Definition der Funktionalität für Modelle Die Aktivierung von Funktionalität für Ansichten ist in unserer Lösung in der Applikationsdefinition zusammengefasst. Der Anwendungsentwickler kann in mehreren Genauigkeiten festlegen, welches Modell welche Funktionalität durch welche Ansichten bieten

soll. Außerdem kann er Abweichungen in den Ansichten für dieses Modell beschreiben oder Ansichten, die nur für dieses Modell gelten. Damit ist diese Information an einer festen Stelle im System zu finden. Änderungen daran werden nur an dieser Stelle durchgeführt.

Darstellung von Attributen Bislang war die Darstellung eines Attributs verteilt in jeder Ansicht, in der es auftauchte, definiert. Die Attribut-Renderer erfassen diese Darstellung an einer Stelle. Die Methodenannotationen beschreiben zudem wann ein Attribut wie dargestellt werden soll.

V.4.2 Praktische Konsequenzen

Die explizite Formulierung von vorher impliziten Informationen über die Funktionalität der Anwendung verändert die Art und Weise, wie eine Anwendung entwickelt werden kann. Die Wartbarkeit und Erweiterbarkeit der Anwendung verbessern sich, da einige Anpassungen nur noch einmal durchgeführt werden müssen. Das Verhältnis zwischen Umfang der abstrakten Formulierung und Umfang der technischen Beschreibung von Funktionalität stellt eine neue Dimension dar, entlang derer Entwurfsentscheidungen getroffen werden müssen. Zudem hat die getrennte Formulierung der Funktionalität und ihrer Umsetzung Auswirkungen auf den Entwicklungsprozess.

Wartbarkeit Die Wartbarkeit verbessert sich mit unserer Lösung, da Anpassungen nur noch an einer Stelle durchgeführt werden müssen, wenn sie einen der oben genannten Teile der Anwendung betreffen. Eine solche Anpassung wirkt sich konsistent auf alle betroffenen Teile der Anwendung aus. Beispielsweise möchte ein Designer der Benutzerschnittstelle etwas an einer bestehenden Anwendung ändern: Der `view_title` von Entitäten soll nun ein Link auf die Entität sein. Dazu muss mit unserer Lösung nur der Attribut-Renderer für den `view_title` angepasst werden. Die Änderung wirkt sich sofort auf die gesamte Anwendung aus.

Erweiterbarkeit Die Verbesserung der Erweiterbarkeit hängt von der Art der Erweiterung ab. Ein neues Modell mit bestehenden Strukturen und Interaktionen anzulegen ist einfacher geworden. Das Domänenmodell muss zwar mit zusätzlichen Informationen ausgestattet werden, aber diese Informationen müssten andernfalls an einer anderen Stelle im System implizit und möglicherweise redundant definiert werden. Anschließend stehen alle bereits definierten Konzepte und Ansichten zur Verfügung, um das neue Modell dem Benutzer über

die Benutzerschnittstelle verfügbar zu machen. Der Anwendungsentwickler muss sie nur noch entsprechend konfigurieren und aktivieren.

Das Hinzufügen eines neuen Strukturkonzepts ist allerdings im gleichen Zug aufwendiger geworden. Der Konzeptentwickler muss die Umsetzung des Konzepts allgemein halten. Variationspunkte müssen aber nicht von vornherein vollständig abgedeckt sein. Sie können hinzugefügt werden, wenn sie gebraucht werden.

Eine Portierung der Anwendung auf einen anderen Zugangsweg haben wir nicht unternommen. Das allgemeine Vorgehen wäre, die Attribut-Renderer und die Umsetzung der Konzepte anzupassen. Sie müssten die passenden graphischen Bausteine für den angestrebten Zugangsweg erstellen. Die Variationspunkte der Konzepte müssten dann neu interpretiert und umgesetzt werden. Der Aufbau der Anwendung und die Verteilung von Funktionalität auf Modelle ändert sich jedoch nicht und bleibt über die verschiedenen Zugangswege hinweg erhalten.

Abwägung zwischen Abstraktion und Umsetzung Die Zusammenfassung von Umsetzungen in ein Konzept obliegt dem Konzeptentwickler. Dieser muss abwägen zwischen dem Abstraktionsniveau der Konzepte und dem Detailgrad der technischen Umsetzung. Bietet beispielsweise ein Konzept zur Darstellung einer Entität nur wenige Variationspunkte, so beschreibt es seine Funktionalität sehr allgemein. Bei der Umsetzung muss der Konzeptentwickler aber möglicherweise Aspekte festlegen, die eigentlich Variationspunkte sind. Zum Beispiel könnte das Konzept zur Darstellung einer Entität einfach immer alle Attributgruppen darstellen. Bieten die Konzepte wiederum zu viele Variationspunkte, sinkt die Ausdruckskraft des Konzepts selbst. Seine Funktionalität wird mehr durch die Variationspunkte, als durch seine inhärente Funktionalität beschrieben. Das Beispielkonzept, das eine Entität anzeigt, könnte zum Beispiel im Extremfall beschreiben, in welcher Reihenfolge die Attribute, mit welcher Bezeichnung, von welchem Attribut-Renderer dargestellt werden sollen.

Anwendungs- und Konzeptentwickler Die Trennung der Beschreibung der Funktionalität und ihrer Umsetzung verändert den Entwicklungsprozess. Der Quelltext, der die Umsetzung beschreibt, ist generischer als der Quelltext, der direkt Funktionalität und Umsetzung in Einem beschreibt. Das kann dazu führen, dass der Quelltext, der nur die Umsetzung beschreibt, schwieriger zu verstehen und anzupassen ist. Es ist Aufgabe des Konzeptentwicklers, diesen Teil des Quelltextes zu beherrschen. Dieses Expertenwissen bedeutet aber auch, dass Änderungen in den Umsetzungen einen längeren Weg gehen müssen. Möchte der Anwendungsentwickler eine Funktionalität ändern,

die er nicht durch seine Variationspunkte beeinflussen kann, so muss der Konzeptentwickler diese Änderung für ihn umsetzen. Das bedeutet auch, dass beispielsweise Designer der Benutzerschnittstelle schwieriger Änderungen an der Umsetzung der Benutzerschnittstelle umsetzen können, da HTML-Elemente in den Konzept-Renderern nicht mehr direkt auf Elemente der resultierenden Webseite abbilden.

V.4.3 Weitere Schritte

Unsere Lösung ist ein erster Schritt hin zu einer vollständig domänenspezifisch beschriebenen Anwendung. Eine vollständige Beschreibung, würde die komplette Anwendung erfassen und gleichzeitig wäre sie für verschiedene Plattformen umsetzbar. Dafür müssten aber noch weitere Aspekte als Konzepte zur Verfügung gestellt werden, unter Anderem Autorisierung. Diese ist ein wichtiger Aspekt moderner Anwendungen und zieht sich durch alle Schichten, ist aber in unserer Lösung nicht erfasst. Zum Beispiel könnte eine Benutzergruppe bestimmte Attribute vielleicht nicht einsehen. Wenn eine Aktion für einen Benutzer nicht erlaubt ist, dann soll der entsprechende Button auch nicht angezeigt werden. Weitere Aspekte sind unter Anderem: Weitere Layoutkonzepte für mehrere Strukturkonzepte in einer Ansicht, Konzepte zur Beschreibung von Menüstrukturen, Ansichten mit Unteransichten und Multi-Tenant-Anwendungen[17].

Umsetzung Auch die Umsetzung kann noch verbessert werden. Interaktionen sollten gelöst von Strukturen beschrieben werden können. Das Prinzip der Sortierbarkeit ist unabhängig von der Umsetzung für eine bestimmte Struktur. Einzige Bedingung ist, dass die Struktur eine Menge von Entitäten anzeigt. Hierfür könnte man Prinzipien wie die horizontale Komposition mit *Mixins* nutzen. Dann könnte man zum Beispiel beliebigen Strukturen, die bestimmte Bedingungen erfüllen, die Funktionalität der Sortierung von Einträgen hinzufügen.

Weiter könnte man die Beschreibung der Modelle, Konzepte und Konzeptaktivierungen in eine eigene Sprache fassen. Damit ist die Anwendungsbeschreibung weitestgehend von bestehenden Technologien losgelöst. Sie wird dadurch unabhängig von aktuellen Technologietrends, da sie auf beliebigen Plattformen umgesetzt werden kann.

[17] Anwendungen, deren Inhalte von mehreren getrennten Anbietern gleichzeitig gepflegt werden.

V.5 Verwandte Arbeiten

V.5.1 Magritte

Magritte ist ein Rahmenwerk für Squeak Smalltalk mithilfe dessen Teile von Ansichten auf Basis eines annotierten Modells erstellt werden. Dazu reichert der Entwickler das Modell mit Informationen an, die definieren, wie Attribute dargestellt werden sollen. Es bietet zudem Standardkomponenten, die solche ausführlich beschriebenen Modelle darstellen können. Auch Magritte ist unabhängig von einer konkreten Zielplattform. Das annotierte Modell kann als Grundlage für Web- oder auch GUI-Anwendungen genutzt werden. Die Ansicht des Modells wird als Komponente in den Rest der Benutzerschnittstelle eingebunden. Unsere Annotationen für Modellattribute sind stellenweise von Magritte inspiriert und davon ausgehend weiterentwickelt. Zum Beispiel hat Magritte kein Konzept für Attributgruppen.

Die Ansätze unterscheiden sich in der tatsächlichen Einbindung der Darstellung. Magritte bietet eine Übersicht über Entitäten des Modells als Komponente an, die man in den Rest der Ansicht einbinden kann. Unsere Lösung beschreibt die vollständige Benutzerschnittstelle. Dazu gehört die Struktur der Ansicht, ihre Interaktionen und die modellspezifischen Abweichungen vom Standard.

V.5.2 Datenbank Manipulierungs Anwendungen

Diese Anwendungen generieren aus den Tabellendefinitionen direkt Oberflächen für CRUD-Aktionen auf den Tabellen. Beispiele sind Catwalk und Sporx. Ihr Ziel ist die einfache Verwaltung von Rohdaten in den Tabellen und keine Oberfläche für den Endnutzer, wie es unsere Lösung anstrebt. Zudem nutzt unsere Lösung neben den Metainformationen aus der Definition des Datenbankschemas auch weitere annotierte Informationen.

V.5.3 Administrator-Benutzeroberfläche

Manche modernen Rahmenwerke zur Entwicklung von Web-Anwendungen bieten mittlerweile die Möglichkeit, Administrator-Benutzeroberflächen zu generieren. Dazu nutzen sie abermals Informationen aus dem Datenmodell und den Definitionen des Datenbankschemas. Das Ziel dieser Oberflächen ist es, Administratoren die einfache Verwaltung von Anwendungsdaten zu ermöglichen. Endnutzerschnittstellen sollen damit meist nicht erstellt werden.

Django Django ist ein Rahmenwerk in Python zur Entwicklung von Webanwendungen. Es bietet die Möglichkeit, eine Administrator-Benutzeroberfläche für ein beliebiges Modell zu definieren. Django besitzt auch Attribute-Renderer und ermöglicht es, Funktionalität für die Administrator Ansicht einzeln zu aktivieren, wie zum Beispiel Sortierbarkeit in Tabellen nur auf bestimmten Attributen. Für jedes Modell kann allerdings nur eine solche Ansicht definiert werden. Entsprechend gibt es auch keine Möglichkeit beliebige Ansichten abstrakt zu beschreiben. Es ist zum Beispiel nur mit Anpassungen des Administrator-Moduls möglich, zwei Übersichtstabellen in einer Ansicht anzuzeigen.

Python Camelot Dieses Python Rahmenwerk ähnelt dem Ansatz von Django, baut ihn aber noch aus. Es beinhaltet sowohl die Annotation des Modells als auch Attribut-Renderer und die Aktivierung von Funktionalität. Wie auch Django bleiben diese Beschreibungen aber auf eine einzige Administrations Ansicht beschränkt.

V.5.4 Evolutility

Dieses Rahmenwerk für Microsofts .Net Rahmenwerk baut den Ansatz der Administrator-Benutzeroberflächen aus. Es erlaubt die Beschreibung beliebiger Ansichten, die eine CRUD-Aktion ermöglichen. Evolutility wählt allerdings eine andere Aufteilung der Informationen über Datenmodell und Darstellung. Der Entwickler beschreibt das Modell in einer XML Datei. Dort legt er für jedes Attribut fest, wo und wie es in der Datenbank gespeichert wird. Außerdem definiert er die Darstellung des Attributs auf der Oberfläche. Zum Beispiel an welcher Position in der Übersichtsliste es angezeigt werden soll. Evolutility bietet dann eine Komponente, die man in seine Webseite einbinden kann und die alle CRUD-Operationen für das beschriebene Modell umsetzt.

Die Beschreibung des Modells und die Konfiguration der Ansichten sind in diesem Rahmenwerk verwoben. Dadurch können in der aktuellen Umsetzung keine Alternativen derselben Ansicht eines Modells erstellt werden. Zum Beispiel kann es nicht zwei unterschiedliche Übersichtsseiten des selben Datenmodells geben. Unsere Lösung trennt strikt zwischen Datenmodell und der Beschreibung der Nutzeroberfläche. Damit hoffen wir, dass die im Modell erfassten Informationen allgemeiner sind. Sie lassen sich möglicherweise einfacher für neue Ansichten wiederverwenden als die konkreten Beschreibungen im Evolutility Ansatz.

V.6 Fazit

Wir haben eine Lösung vorgestellt, die Probleme in der Entwicklung von Benutzeroberflächen von modellintensiven Anwendungen löst, indem sie die Möglichkeit bietet Funktionalität explizit zu formulieren.

Ausgehend von aktuellen Rahmenwerken für die Entwicklung von Webanwendungen haben wir Probleme in der Erweiterbarkeit und Wartbarkeit festgestellt. Diese resultieren aus der impliziten Beschreibung, welche Funktionalität die Anwendung zur Verfügung stellt. Technischen Abstraktionen definieren wie die Anwendung funktioniert, aber auch was sie können soll. Wir schlagen darum eine Lösung vor, mit der man die Information, welche Funktionalität eine Anwendung zur Verfügung stellen soll, explizit formulieren kann.

Der implementierte Prototyp basiert auf einem Datenmodell, das um Darstellungsinformationen erweitert wurde. Auf Basis dessen bietet er explizite Schnittstellen zur modellunabhängigen Beschreibung der Darstellung von Attributen, Strukturen, Interaktionen und ganzer Ansichten. Mit diesen Bausteinen kann eine Standardansicht für CRUD oder andere Aktionen definiert werden. Damit reicht es die Ansicht für ein Modell zu aktivieren um die Ansicht für den Benutzer zur Verfügung zu stellen. Abweichungen von der Standardansicht werden inkrementell pro Modell definiert.

Durch die explizite Formulierung der Funktionalität der Anwendung, lassen sich die gefundenen Probleme bezüglich Wartung und Erweiterbarkeit lösen. Für eine vollständige Beschreibung einer Anwendung müsste unser Prototyp aber noch weitere Aspekte von Anwendungen erfassen.

Teil VI

Programmierkonstrukte für die Beschreibung von Geschäftslogik

VI.1 Einleitung

Nicht nur durch die fortschreitende technologische Abhängigkeit, in die uns unsere Epoche mit Smartphones, Web 2.0 und IPv6, in den folgenden Jahren zu führen scheint, werden Webseiten immer mehr zu Anwendungen. Konzepte wie „Software as a Service“ zeigen uns, dass wir für einen Großteil der Endnutzer eine Webseite anstatt eines entsprechenden Programms substituieren könnten. In vielerlei Hinsicht wird in diese Richtung entwickelt: Die Frameworks für Webseiten werden ausgefeilter, Content Management Systeme gibt es wie Sand am Meer, HTML5 bringt eine neue Mächtigkeit für Webseiten und IPv6 hilft uns zu garantieren, dass wir auch in den nächsten Jahrzehnten noch die wachsende Zahl von Internetteilnehmern bewältigen können.

Daher ist gerade für Unternehmen der Umstieg von einer dedizierten Software zur Verwaltung der Kunden- und Unternehmensdaten zu einer Webseite mit denselben Fähigkeiten nicht nur erstrebenswert, sondern im großen Stil schon geschehen. Während die Möglichkeiten unserer über Jahre gewachsenen Beschreibungssprachen für Webseiten kontinuierlich zunehmen, halten sich mit ihnen verbundene, mittlerweile überholte oder unverhältnismäßig aufwendige Programmierkonzepte immer noch. Der kontinuierliche Fortschritt, mit dem wir in der Informatik umgehen lernen müssen, bringt uns dazu, dass wir uns oft länger an vertrauten Konzepten festhalten als gut für uns ist. Während sich die Möglichkeiten für Webseiten und ihr gewöhnliches Aussehen über die letzten Jahrzehnte radikal verändert hat, schreiben wir Webseiten zu großen Teilen immer noch so wie früher.

In Kapitel V haben wir bereits beschrieben, wie sich Webseiten über großen Modellmengen gut beschreiben lassen. Diese Arbeit wird in vielerlei Hinsicht an die vorherige anknüpfen und auf ihr aufbauen. Mit den von uns entwickelten Konzepten können wir Webseiten bereits effizient erzeugen, aber einen wichtigen Aspekt der Beschreibung von webbasierter Unternehmenssoftware haben wir noch nicht besprochen: Die Beschreibung von Abläufen.

Unternehmenssoftware sowie Webseiten im Allgemeinen enthalten sehr viele. Ob wir unser Passwort ändern, uns auf der Webseite registrieren, einen Einkauf tätigen oder eine Umfrage ausfüllen; überall stoßen wir auf Abläufe, die von der jeweiligen Domäne abhängen und aus diversen Schritten bestehen. Die verbreiteten Möglichkeiten, um solche Abläufe zu beschreiben, sind überraschend unausgereift und die Programmierung mühsam, fehleranfällig sowie nicht gut wartbar. Im nächsten Abschnitt werden wir das eben skizzierte Problem genauer fassen.

VI.2 Problembeschreibung

In diesem Abschnitt werden wir als erstes das Beispiel einführen, dass uns durch die restliche Arbeit geleiten wird. Außerdem werden wir einige entscheidende Konzepte einführen, bevor wir anhand einer skizzierten Implementierung des Beispiels, Probleme in bestehenden Lösungen aufzeigen werden.

VI.2.1 Beispiel

In Unternehmensanwendungen sind mehrschrittige Abläufe allgegenwärtig. Ein Beispiel ist der Abschluss eines Einkaufs in einem Online-Shop: Nachdem der Benutzer Waren in seinen Warenkorb gelegt hat, hat er die Möglichkeit, die Bestellung zu finalisieren. Dazu muss er zuerst seine Kontaktdaten eintragen, meist bestehend aus einer Liefer- und einer Rechnungsadresse. Dann wählt er eine Zahlungsmöglichkeit aus, von denen manche eventuell direkt im Browser abgeschlossen werden. Darüber hinaus muss er auch eine Versandart auswählen, wobei die Auswahl hier von der vorherigen Auswahl der Zahlungsart und insbesondere von der eingetragenen Lieferadresse abhängen kann. Am Ende werden dem Benutzer alle seine eingegebenen Daten präsentiert, sodass er auf fehlerhafte Eingaben prüfen kann. Danach gibt er ein verbindliches Vertragsangebot ein.

VI.2.2 Analyse des Beispiels

Wir können klar erkennen, dass alle Schritte Teil eines größeren Ablaufes sind. Diesen bezeichnen wir als einen Ablauf mit Geschäftslogik. Die Abläufe können dabei durchaus komplex werden, insbesondere können starke Abhängigkeiten zwischen den verschiedenen Teilschritten bestehen. Obwohl solche Szenarien alltäglich sind, finden sich in den gängigen modernen Frameworks keine adäquaten Möglichkeiten, um sie zu formulieren, wie wir zuerst am Beispiel von ASP.NET MVC [2] erörtern werden.

VI.2.3 Begriffsklärungen

Im Verlauf der dieser Arbeit wird auch einige Konzepte eingegangen und auf ihnen aufgebaut, im Folgenden werden die wichtigsten erklärt.

MVC Wie alle gängigen Webframeworks arbeitet auch ASP.NET nach dem **M**odel **V**iew **C**ontroller [27][26] Prinzip. Hierbei wird die Anwendungsbeschreibung in drei Teile aufgeteilt:

1. Model: Die Daten sind in Modellen gespeichert. Die Modelle sichern Einschränkungen wie Validität. Sie werden in der Datenbank persistiert.
2. View: Die Ansichten, insbesondere von Modellen, werden in der Regel von einem mit HTML-Elementen bestücken Dialekt der Hostsprache geschrieben (Templatesprache).
3. Controller: Der Controller steuert den Kontrollfluss zwischen verschiedenen Views und interpretiert, beispielsweise mit REST, die URLs[18] als Aktionen auf Modellen.

Zwar ist MVC eine sinnvolle Kapselung von Teilaspekten der Webentwicklung, doch so, wie Controller in den gängigen Frameworks umgesetzt sind, gibt es keine Konzepte, mit denen sich mehrschrittige Interaktionen mit dem Nutzer, in geeigneter Form, formulieren lassen. Die von uns vorgeschlagene Lösung gliedert sich in die Beschreibung von Controllern in MVC ein und dies auf eine ähnliche Art, wie in Kapitel V unserer Ansichtskonzepte in die Views.

REST **RE**presentational **S**tate **T**ransfer [10] bezeichnet eine Art der Verknüpfung von Internetadressen, mit Controller-Methoden. Ein Benutzer könnte so beispielsweise eine URL auf eine Instanz eines Modell haben: */Products/1*. Durch anfügen von */edit* könnte er die Editierungsfunktion des Produkt Controllers aufrufen, was in der Antwort eines Editierungsformulares resultieren würde. Im Bereich der Webframeworks hat sich REST vor allem wegen seiner Einfachheit durchgesetzt. Alle hier besprochenen Webframeworks unterstützen REST. REST unterscheidet dabei verschiedene Arten des Zugriffes auf URLs. Die drei wichtigsten sind:

1. GET: Lesender Zugriff auf eine URL. GET auf die Seite erwartet als Antwort die entsprechende Seite.
2. POST: Schreibender Zugriff auf eine URL. Wenn ein Formular ausgefüllt wurde und die Daten an den Server übermittelt werden sollen, werden diese mit POST übermittelt.
3. DELETE: Löschen eines Elements. */Products/1/delete* würde also die Instanz mit der Nummer 1 löschen.

Darüber hinaus gibt es noch PUT, PATCH, HEAD und OPTIONS, die für uns im Folgenden aber nicht relevant sind. Entscheidend an REST ist außerdem,

[18] Unique Resource Location

dass es zustandslos ist. Jede Nachricht an den Server enthält dabei alle nötigen Informationen, die zum Verständnis benötigt werden. Deshalb kann der Webserver, der REST versteht, zustandslos implementiert werden, was vor allem Lastverteilung vereinfachen kann.

CRUD CRUD [18] steht für vier, in Geschäftsanwendungen häufige, Aktionen.

1. Create: Anlegen eines neuen Datensatzes
2. Read: Auslesen eines Datensatzes
3. Update: Verändern eines Datensatzes
4. Delete: Löschen eines Datensatzes

Dies sind die wesentlichen Operationen, die auf Daten vorgenommen werden können. Eine Geschäfsanwendung muss in der Regel auf allen ihren Datenmodellen CRUD-Funktionalität bereitstellen. Diese vier Operationen lassen sich auf die drei weiter oben beschriebenen REST Befehle abbilden. So können über GET Daten gelesen, über POST erstellt und verändert sowie über DELETE gelöscht werden.

Von großem Interesse sind diese Aktionen für uns, weil wir mit einer allgemeingültigen Standardimplementierung dieser einen Großteil der Entwicklung von Geschäftsanwendungen beschleunigen könnten. Normalerweise müssen, um CRUD Funktionalität zu implementieren, für jedes Modell entsprechende Seiten angelegt und mit den Controllern verknüpft werden. Mit unserem Prototypen können alle Unterseiten wie */show*, */edit*, */create* und */delete* gerufen werden, ohne das wir eine einzige dieser Seite händisch erstellen mussten, da sie alle generiert werden können.

VI.2.4 Implementierung eines Ablaufes mit Geschäftslogin in ASP.NET MVC

In Anlehnung an unser vorheriges Beispiel in Abschnitt VI.2.1 werden wir nun eine mögliche Implementierung, in einem üblichen Framework betrachten. Es geht hierbei weniger um die konkreten Methoden, die im Hintergrung ausgeführt werden, sondern um den Entwicklungsfluss des Ablaufes in unserem Beispiel-Framework. Die folgenden Codebeispiele sind stark vereinfachter C#-Code. Damit einerseits die Übersichtlichkeit erhalten und andererseits die Aufmerksamkeit auf das Wesentliche konzentriert werden kann.

```
1  public ActionResult einkaufswagen_betrachten()
2  {
3      var warenliste = new List<String>();
4
5      var posten = getPosten();
6      foreach (var p in posten)
7      {
8          warenliste.Add(posten.Name);
9      }
10
11     ViewBag.warenliste = new SelectList(warenliste);
12     return View();
13 }
```

Quelltext 24: *Die C# GET-Methode, die Daten für den View des Warenkorbs bereit stellt*

Als Einstiegspunkt in den Ablauf betrachen wir hier das Ansehen des Warenkorbs. Hier können Artikel entfernt, in der Anzahl verändert und bestellt werden. Die Methode `einkaufswagen_betrachten` ist auf dem entsprechenden Controller definiert. Durch einen Klick auf das Einkaufswagensymbol gelangt man auf eine URL, die über REST auf unsere Methode geleitet wird.

Eine mögliche Aktion in der Methode könnte nun sein, dass der Inhalt des Warenkorbs als Auswahlliste angezeigt werden soll. Hierzu wird in Zeile 3 eine Liste von Strings angelegt. Über die Methode `getPosten()` werden die markierten Waren aus der Session gelesen. Im folgenden werden die Namen der ausgewählten Produkte in die Stringliste eingetragen und aus selbiger eine `SelectList` erstellt. Die `SelectList` ist eine vorformatierte Struktur, die später gegen einen HTML-Renderer gebunden und als `ListBox` gerendert werden kann. Anzumerken ist hier außerdem, dass der Bezeichner `warenliste` eindeutig sein muss, da im View genau dieser, bei der Konstruktion der Liste hinterlegt werden muss.

Der Einkaufswagen View Der in Quelltext 25 beschriebene View wird über seinen Namen an die Methode aus dem vorherigen Schritt gebunden. Er ist außerdem ein partieller View, was in diesem Kontext bedeutet, dass er in ein vorhandenes Template eingefügt wird. Die Inhaltsliste des Warenkorbs wird also in einem Standard-Rahmen im Firmendesign angezeigt werden.

```
1  @{
2      ViewBag.Title = "Einkaufswagen_betrachten";
3  }
```

```
4  <h2>Ihr Einkaufswagen</h2>
5  <div>
6  @using (Html.BeginForm())
7  {
8      <p><em>Artikel:</em> @Html.ListBox("warenliste") </p>
9      <input type="submit" name="Zurueck" value="Zur&uuml;ck" />
10     <input type="submit" name="Kasse" value="Zur Kasse gehen" />
11 }
12 </div>
```

Quelltext 25: *Der View des Einkaufswagens in CSHTML beschrieben*

In Zeile 6 wird ein Formular für den Benutzer erstellt. In diesem können von ihm Aktionen vorgenommen werden, die dann über entsprechende Buttons an den Server übermittelt werden. Auf der tatsächlichen Seite müssten für den Benutzer natürlich noch mehr Aktionen möglich sein, als „Zur Kasse gehen“, wie zum Beispiel, Artikel zu entfernen. In Zeile 8 wird eine Auswahlliste aus den im vorherigen Schritt erstellten Liste von Waren kreiert. Hierbei wird derselbe eindeutige Bezeichner verwendet, um Daten zwischen View und Controller auszutauschen. Genauso verhält es sich mit den Bezeichnern `Zurueck` und `Kasse`. Je nachdem welcher Button vom Benutzer gedrückt wird, wird der Wert zu diesen Bezeichnern verändert. So kann der Contoller abfragen, welche Aktion angestoßen wurde.

Die zugehörige POST-Methode Über die Methoden-Annotation wird die Methode in Quelltext 26 als POST-Gegenstück zu der vorherigen gekennzeichnet. Diese Methode nimmt außerdem noch einige Parameter, deren Namen durch den vorherigen View vorgegeben sind.

```
1  [HttpPost]
2  public ActionResult einkaufswagen_betrachten(
3                                          string Zurueck,
4                                          string Kasse,
5                                          string warenliste)
6  {
7      if (Kasse != null)
8          return RedirectToAction("bestellung_abschliessen");
9      else if (Zurueck != null)
10         return RedirectToAction("index");
11     return View();
12 }
```

Quelltext 26: *Die POST-Methode, die die Anfrage bearbeitet, in C#*

Um zu erfahren, ob der Benutzer zur Kasse gehen möchte, muss die entsprechende Variable ausgelesen werden. Ist der Wert von `Kasse` ungleich `null`, dann wurde der entsprechende Button gedrückt. Genauso verhält es sich mit dem Zurück-Button. Darüber hinaus wird noch die Warenlist als POST-Parameter angegeben. In dieser Beispiel Methode ist sie nicht notwendig, solange immer alle Artikel bestellt werden. Um aber die Logik zum Löschen, beziehungsweise Quantität ändern, zu implementieren, könnten aus diesem Parameter die ausgewählten Einträge extrahiert werden und Session-Variablen entsprechend modifiziert werden.

Die folgenden Schritte Für alle Schritte, die zu dem eingeleiteten Bestellvorgang gehören, müsste genauso vorgegangen werden. Es müsste eine neue Methode angelegt werden, auf deren Name durch die Post-Methode des vorherigen Schrittes, weitergeleitet werden würde. In unserem Beispiel hieße die nächste Methode also `bestellung_abschliessen`. Außerdem müsste ein gleichbenannter View erstellt werden, sowie eine Postmethode. Die zwei Methoden würden wiederrum gleich benannt sein und im selben Controller definiert sein. Der View hingegen wäre in einer anderen Datei. Die Abhänigkeiten zwischen dem View und den Controller-Methoden sind aber sehr stark. Für jeglichen Datenaustausch müssen in Methode und View dieselben Bezeichner verwendet werden. Falls es möglich sein soll, zurück zu navigieren, muss die neue Post-Methode auch den Namen der vorherigen Methode kennen. Diese muss nicht notwendigerweise im selben Controller definiert sein.

Wenn außerdem Daten zwischen verschiedenen Controller-Methoden weitergegeben werden sollen, zum Beispiel die Adresse an den Schritt zur Auswahl des Versandunternehmens, dann müssen diese wiederum über eindeutige Bezeichner identifiziert in der Session gespeichert werden. Wir können erkennen, das es starke, teils auch zirkuläre Abhängigkeiten zwischen den einzelnen Schritten und ihren Bestandteilen an voneinander getrennten Teilen des Codes gibt.

Während es beim initialen Schreiben des Ablaufes nur komplex und verschachtelt ist, wird der Umbau desselbigen um ein vielfaches fehleranfälliger. Angenommen, es würde sich um einen mehrschrittigen Ablauf handeln, in den wir in der Mitte einen neuen Schritt einfügen möchten. Dann müssten wir nicht nur den eigentlichen Schritt mit zwei Controller Methoden und einem View bauen. Wir müssten außerdem in der Post-Methode des vorherigen Schrittes den nächsten Schritt abändern. Genauso den vorherigen Schritt im nächsten Schritt des Ablaufes. Falls die Buttons in den Views entsprechend des nächsten oder vorherigen Schrittes benannt sind, müssten wir dies ebenso anpassen. Wenn in den Schritten eine Fortschrittsanzeige den Benutzer darüber informiert, in welchem Schritt er gerade ist und wieviele noch kommen werden, müssten wir

dies entsprechend verändern. Beim Umbenennen von Variablen oder Methodennamen verhält es sich nicht besser.

Abschließende Überlegungen Die soeben aufgezeigten Probleme treten nicht nur in ASP.NET MVC auf. Bei unserer Arbeit mit Spring und Ruby on Rails konnten wir genau die gleichen Probleme wiederfinden.

Ein offensichtliches Problem ist, dass Implizites explizit formuliert werden muss. Obwohl beispielsweise die Navigation zwischen Schritten innerhalb eines Ablaufes immer gleich funktioniert, muss sie in jedem Schritt erneut aufgeschrieben werden. In der Praxis werden die Buttons auch weniger genau benannt. In unserem Beispiel waren es *Kasse* und *Zurueck*. In der Realität blieben es wahrscheinlich über sämtliche Schritte *Next* und *Prev*. Die Controller Methoden sähen in diesem Punkt auch alle untereinander ähnlich. Ihre Form wird im Allgemeinen der, der Methode in Quelltext 27 gleichen.

```
1 [HttpPost]
2 public ActionResult methode(string Next, string Prev)
3 {
4     if (Next != null)
5         return RedirectToAction(...);
6     else if (Prev != null)
7         return RedirectToAction(...);
8     return View();
9 }
```

Quelltext 27: *Die grobe Struktur potentiell vieler POST-Methoden, die die Navigation zwischen Schritten beschreiben*

Während der Entwickelung und später während der Wartung, würden die Entwickler große Teile des Codes kopieren und nur ein paar Namen austauschen. Redundanz und Codeduplikation sind altbekannte Phänomene der Informatik. Zwar müssen sie nicht immer problematisch sein, aber in Code, der regelmäßiger Veränderung unterworfen ist, zeigt sich, dass mit der Zeit die Mehrkosten durch Fehler, die aus Redundanz und Duplikation entstehen, exorbitant sind. Wir wollen uns im Folgenden fragen: Ist es eine inhärente Eigenschaft eines Ablaufes, dass man ihn nur schwerlich aufschreiben kann, oder ist er vielleicht nur auf der falschen Abstraktionsebene beschrieben worden?

VI.2.5 Zusammenfassung

In diesem Abschnitt haben wir grundlegende Begriffe geklärt, sowie ein Beispiel für einen mehrschrittigen Ablauf mit Geschäftslogik eingeführt. In

der folgenden Implementierung desselben, haben wir Schwächen in den verbreiteten Ansätzen zur Programmierung gefunden und planen diese im Folgenden zu beheben.

VI.3 Lösungsidee

In diesem Abschnitt formulieren wir die Kernaussage unseres Beispielablaufes. Wir betrachten dabei geeignete Abstraktionsebenen und skizzieren die von uns gewünschte Lösung.

VI.3.1 Bestellungsablauf

Überlegen wir einmal, was die eigentliche Aussage des Bestellungs-Ablaufes ist und wie wir ihn möglichst kurz formulieren könnten. Der Inhalt des Warenkorbs soll verändert und bestellt werden können. Der Bestellung müssen Rechnungs- sowie Versandadresse zugeordnet werden. Eine Bezahlungsart muss ausgewählt werden. Die Versandart der Bestellung muss in Abhängigkeit von den enthaltenen Artikel, Versandadresse und eventuell Bezahlungsart ausgewählt werden können. In jedem Schritt soll es möglich sein, vor und zurück zu navigieren. Unter Umständen soll außerdem in jedem Schritt anzeigt werden, wieviel Prozent des Vorgangs schon abgeschlossen sind. Die einzelnen Schritte können wie folgt beschrieben werden.

1. Zeige dem Benutzer den Inhalt seines Warenkorbes an. Basierend auf seiner Auswahl und gewählten Aktion verändere den Inhalt des Warenkorbes und falls nötig, erstelle eine neue Bestellung mit den ausgewählten Waren.
2. Trage für die Bestellung die Versandadresse ein. Eine abweichende Rechnungsadresse kann eingetragen werden.
3. Lasse den Benutzer eine Zahlungsart für die Bestellung auswählen.
4. Berechne die möglichen Versandoptionen anhand der Zahlungsart und der Versandadresse. Lasse den Benutzer eine der Optionen auswählen.

VI.3.2 Die geeignete Abstraktionsebene

Mit dieser Art der Formulierung von Schritten, können wir das *Was* beschreiben und abstrahieren vom *Wie*. Wir beschreiben die wesentliche Aussage des Ablaufes und an anderer Stelle die wesentliche Aussage des einzelnen Schrittes. Wir können den Ablauf damit nach Wahl, in der für uns geeigneten Abstraktionsebene, betrachten. Ein Verstehen des Ablaufes ist auf der obersten Ebene einfach. Wenn Detailwissen eines Schrittes nötig wird, dann können wir

genau diesen Schritt genauer betrachten. Wenn Implementierungsdetails von beispielsweise der Berechnung der möglichen Versandoptionen benötigt werden, kann in die entsprechende, im Schritt Vier gerufene, Funktion abgetaucht werden. Wir können erkennen, dass wir für die Beschreibung eines Ablaufes mehrere relevante Abstraktionsebenen finden können, wobei in bestehenden Webframeworks Abläufe in der Regel auf der untersten beschrieben werden.

Es ist nun einfacher möglich, den Ablauf zu verstehen, da die Komplexität deutlich geringer ist und er im Ganzen formuliert ist, was den Zusammenhang zwischen verschiedenen Schritten bewahrt. Mit einer leichteren Verständlichkeit nimmt auch die Wartbarkeit zu, da neue Programmierer leichter Einblick in das System erhalten können.

VI.3.3 Ein geeignetes Programmiermodell

Eine natürlichsprachliche Beschreibung des Ablaufes wird aller Wahrscheinlichkeit nach noch nicht in naher Zukunft von einem Computer ausgeführt werden können. Im Folgenden wollen wir versuchen, die Kompaktheit dieser Beschreibung auf unsere Implementierungen zu übertragen.

Die Ablaufseinheit Einen Ablauf können wir auf eine Funktion abbilden. Eine Funktion ist immer ein sinnvoller Zusammenschluss von weiteren Funktionen und Primitiven. Die Schritte einer Funktion haben eine implizite Reihenfolge. Die Funktion ist dabei auch direkt unsere Transaktionseinheit. Wenn ein Schritt in der Funktion fehlschlägt, oder die Funktion abgebrochen wird, dann müssen alle vorherigen Änderungen der Funktion zurückgenommen werden.

Die Benutzer-Interaktion Die Interaktion mit dem Benutzer ist der schwierige Teil. Mit Sicherheit ist sie einer der Gründe, warum Abläufe in den gängigen Frameworks so umständlich beschrieben werden müssen. Wir wollen in unserer Beschreibung von den Implementierungsdetails der asychronen Interaktion mit dem Benutzer abstrahieren, indem wir Variablen durch Funktionen füllen lassen, die implizit eine Eingabe des Benutzers anfordern.

Die Viewgenerierung Die Konzepte dieser Arbeit kommen erst in Verbindung mit dem Rest unseres Webframeworks richtig zum Tragen, das in Kapitel V beschrieben ist. Das tolle ist, dass wir aus beliebigen Attributmengen beliebiger Modelle, entsprechende Eingabeformulare generieren können. Das heißt, dass wir für einen Ablauf keinen einzigen View anlegen müssen. Alles,

einschließlich der Vor- und Zurücknavigation sowie Fortschrittsbalken, wird generiert.

CRUD Was wir insbesondere vollständig generieren wollen, sind die Formulare, die wir für CRUD-Funktionalität benötigen. Eine mögliche Sicht auf die potentiellen Aktionen eines Benutzers könnte auch sein, dass alles was er tut, CRUD ist. Wir können nicht nur die offensichtlichen Seiten, wie *create* und *edit*, damit modellieren, sondern auch die Suche über Instanzen sowie Übersichts- und Detailseiten. Außerdem sind die meisten generischen CRUD-Funktionen einfach beschrieben, gerade wenn schlicht alle Attribute eines Modells befüllt und validiert werden müssen. Eine solche Funktion ist im Folgenden unser erstes Beispiel.

VI.3.4 Eine erste Funktion

```
1 def create():
2     instance = create(process, Model)
3     core_index(process, Model)
```

Die Funktion `order_create` kapselt hier das Anlegen einer Instanz, eines beliebigen Modells. In Zeile 2 wird eine Variable `instance` angelegt und durch die Funktion `create` mit Werten befüllt. Die `create`-Funktion ist hierbei eine implizite Interaktion mit dem Benutzer, da er Attribute wie Name oder ähnliches auswählen muss. Woher `create` weiß, welche Attribute ausgefüllt werden müssen, wird in Abschnitt VI.4.2 erklärt. In Zeile 3 wird der Benutzer auf die Übersichtsseite der jeweiligen Modellklasse umgeleitet und die Transaktion erfolgreich beendet. Die `create`-Funktion enthält abgesehen von der Interaktion auch implizit die Validierung der Eingaben des Benutzers und entsprechende Reaktionen. Sollte der Benutzer im ersten Schritt ungültige Eingaben tätigen, erhält er eine Fehlermeldung und verweilt solange bei Schritt 1, bis er Valides eingegeben oder den Vorgang abgebrochen hat.

VI.3.5 Zusammenfassung

In diesem Abschnitt haben wir die Kernaussagen unseres Beispielablaufes extrahiert und verschiedene Abstraktionsebenen entdeckt. Außerdem haben wir erste wichtige Ideen für die Implementierung gesammelt und eine mögliche Beschreibung, auf der obersten Abstraktionsebene, eines simplen Ablaufes gefunden.

VI.4 Implementierung

Dieser Abschnitt erklärt unsere Implementierung der Beschreibungskonzepte für Abläufe mit Geschäftslogik. Wir erklären die von uns eingeführten Schlüsselwörter und die Details der Ausführungsengine für die Abläufe und unsere Implementierung von Continuations in Python. Außerdem erklären wir noch die automatische Validierung der Eingaben des Benutzers.

VI.4.1 Die Programmiersprache

Bei der Implementierung der von uns entwickelten Konzepte zur Integration von Konzepten aus Hauptspeicherdatenbanken in die Programmiersprache (siehe Kapitel II), fiel die Wahl früh auf Python. Nachdem wir Teile unserer Konzepte für Webframeworks in Ruby on Rails integriert hatten, stiegen wir um die Projekte zusammenzuführen auf Python um. Dabei diente uns das Webframework Pyramid [22] als Grundlage. Zwar sind einige Implementierungsstrategien in unserem Prototypen von Python beeinflusst, doch die von uns entwickelten Konzepte sind sprachunabhängig.

VI.4.2 Ein Ablauf in einer Funktion

Wie im vorherigen Abschnitt dargestellt, wird ein Ablauf in einer Funktion gekapselt. Ein Teil des bisherigen beispielhaften Bestellvorgangs sähe in unserem Framework wie folgt aus.

```
1 @model_action(name="create", model=Order)
2 def order_create(process, model, instance = None):
3     order = create(process, Order,
4                    property_groups = ["overview",
5                                       "items",
6                                       "shipping_address"])
7     edit(process, order,
8          property_groups = ["shipping_agency"])
9     core_index(process, model)
```

Quelltext 28: *Der zentrale Ausschnitt aus der Beschreibung des Bestellvorgangs, der in einer Methode zusammen gefasst ist*

Die Annotation in Zeile 1 bindet die Funktion gegen die `create`-Aktion auf dem `Order`-Modell. Ansonsten würde eine Standardaktion, ähnlich der im

vorherigen Abschnitt, ausgeführt werden. Die `order_create` Funktion verfeinert sozusagen die `create`-Funktion auf dieser einen Klasse. Die Parameter der Funktion sind der aktuelle Ablauf, die Modellklasse sowie eine leere Instanz, da diese ja erst angelegt werden soll. Im ersten Schritt in Zeile 3 wird die `order`-Variable durch den Benutzer gefüllt. Allerdings nicht vollständig, da der Versanddienst erst nach Auswahl der Versandadresse ausgewählt werden kann. Stattdessen werden im ersten Schritt nur drei Attributgruppen ausgefüllt: Die allgemeinen[19], Artikel[20] und Versandadresse[21].

Nachdem der Benutzer die erforderlichen Eingaben getätigt hat, wird validiert, mehr dazu in Abschnitt VI.4.5, und bei Bestehen der nächste Schritt ausgeführt. Bevor die `edit`-Funktion gerufen wird, wird im Modell eine entsprechende Berechnung auf Basis der Versandadresse vorgenommen, um die in Frage kommenden Versanddienste, dem Benutzer zur Auswahl zu stellen. Der Benutzer wird über `edit` in Zeile 7 dazu aufgefordert, einen Versanddienst[22] auszuwählen.

Nachdem der Benutzer die Bestellung vollständig angelegt hat, wird die Funktion und damit der Ablauf und damit die Transaktion mit `core_index` erfolgreich beendet, der Benutzer auf die Übersichtsseite seiner Bestellungen umgeleitet und die Bestellung in der Datenbank persistiert.

VI.4.3 Ausführung eines Ablaufes

Die Ausführungsengine für die Abläufe ist relativ simpel gehalten. Im Grunde gibt es nur eine Schwierigkeit und das sind die asynchronen Interaktionen mit dem Benutzer. Wenn der Benutzer bei einem Schritt etwas eintragen muss, dann möchte er danach den Ablauf genau an der Stelle und im gleichen Zustand fortsetzen, in dem er vorher war. Für die Ausführungseinheit vergeht allerdings ungemein viel Zeit, in der auf den Benutzer gewartet werden muss. Diese Problem ist gut bekannt und eine allgemeingültige und viel verwendete Lösung sind Fortsetzungen, besser als Continuations bekannt.

Continuations erlauben es, eine Ausführung anzuhalten und zu einem späteren Zeitpunkt an genau derselben Stelle fortzusetzen. Sie garantieren dabei, dass sämtliche Variablen den gleichen Zustand wie zuvor haben. Continuations sind in der Computer-Technologie weit verbreitet. Mögliche Beispiele schließen

[19] overview in Kapitel V- u. a. Name
[20] `items` - enthalten die Posten der Bestellung
[21] `shipping_address`
[22] shipping_agency

PSW[23] oder Sicherung des Registersatzes in üblichen Betriebssystemen ein. Im Kontext von Webframework bedeutet eine Continuation, dass der Instruktionszeiger sowie der Funktionsstack gespeichert wird. Bei der Fortsetzung wird dann entsprechend der Funktionsstack wiederhergestellt und die Ausführung ab dem Instruktionszeiger fortgesetzt.

Continuations in Python Python bietet keine Continuations oder ein äquivalentes Feature, da es nicht möglich ist, Funktionsstacks zu kopieren und die Ausführung fortzusetzen. Ein verwandtes Konzept in Python sind Iteratoren, die sich aber nicht für unseren Einsatzzweck eignen.

VI.4.4 Ein lohnenswerter Workaround

Die Idee ist einfach: Anstatt die Ausführung an einem bestimmten Punkt anzuhalten, brechen wir sie ab und simulieren sie beim nächsten Mal bis zur selben Stelle. Zu bemerken ist erst einmal, dass sich ein Ablauf als Graph darstellen lässt. Die Berechnungen sind Knoten und die Eingaben des Benutzers sind Kanten.

Abb. 33: *Graph des Bestellprozesses*

Um also einen bestimmten Zustand im Graphen zu speichern, reicht es, alle passierten Kanten zu speichern, mit ihnen kann man den alten Zustand erneut herstellen. Genauso funktionieren unsere Continuations. Bei der Ausführung einer Funktion läuft alles normal, bis eine Funktion gerufen wird, die eine Nutzerinteraktion nach sich zieht. Diese Funktion versucht auf die entsprechenden Eingaben des Benutzers zuzugreifen. Falls diese noch nicht vorhanden sind, wirft sie eine Ausnahme, besser bekannt als Exception. Diese Exception wird den Stack hinaufgereicht und beendet die Funktion und damit die Transaktion, frühzeitig. Alle bisherigen Änderungen werden verworfen. Dann behandelt die Ausführungsumgebung die Exception, indem sie dem Benutzer ein entsprechendes Eingabeformular generiert. Wenn die Antwort des Benutzers eintrifft, wird sie gespeichert und der Ablauf neu angestoßen.

Der Ablauf startet also wieder von vorn und nimmt genau die gleichen Schritte, bis zu dem Moment, in dem die Funktion, die Eingaben des Benutzers erfordert

[23] Program Status Word**

und diesmal erfolgreich darauf zugreift, da die Daten ja schon vorliegen. Das folgende Bild erklärt die Ausführung genauer.

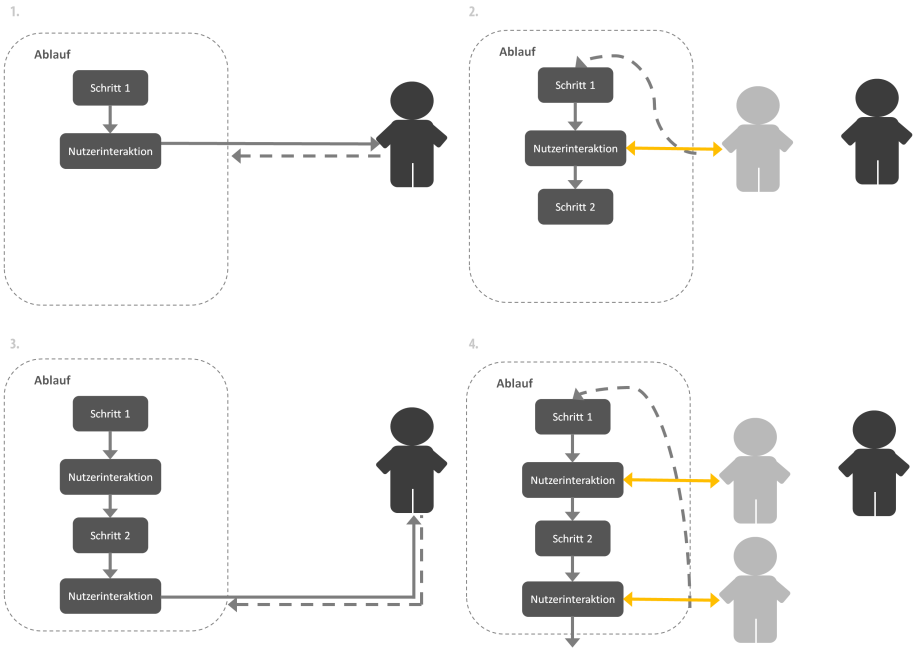


Abb. 34: *Ablauf eines Prozesses mit abstrahierter Nutzerinteraktion*

1. Nachdem der Benutzer den Ablauf angestoßen hat, wird eine interne Berechnung in Schritt 1 ausgeführt. Danach ist eine Nutzerinteraktion nötig. Es liegen noch keine Eingaben des Benutzers für diesen Schritt vor, daher wird eine Exception geworfen, die den Ablauf beendet. Der Nutzer wird über ein passendes Formular zur Eingabe der Daten aufgefordert.
2. Die Antwort des Benutzers, eine POST-Nachricht, enthält, nebst der eingegeben Daten, einen eindeutigen Bezeichner für den vorher angestoßenen Ablauf. Daher kann das Framework den richtigen Ablauf extrahieren, die Eingaben des Benutzers speichern und den Ablauf erneut anstoßen. Nach der Ausführung von Schritt 1 wird erneut, in Folge der notwendigen Nutzerinteraktion, auf die Daten des Benutzers zugegriffen, dieses Mal aber erfolgreich, da die Eingaben des Benutzers bereits bekannt sind. Nach dem Verarbeiten der Daten wird Schritt 2 angestoßen.
3. Nach Schritt 2 folgt erneut eine Nutzerinteraktion. Da noch keine Eingaben für diesen Schritt vorliegen, wird der Ablauf erneut abgebrochen und der Benutzer involviert. Seine Antwort startet den Ablauf erneut.
4. Es werden erneut alle Berechnungen der Schritte 1 und 2 ausgeführt und mit den vorliegenden Eingaben des Benutzers gefüttert.

An dieser Stelle könnten noch weitere Schritte folgen oder der Ablauf erfolgreich beendet werden. Festzustellen bleibt allerdings, dass der finale Ablauf synchron ausgeführt wurde, da alle Eingaben des Benutzers bereits vorlagen. Das Prinzip der Ausführung sollte hierdurch klar geworden sein. Eine wichtige Frage, nämlich die nach Seiteneffekten von Abläufen, wird in Abschnitt VI.5.3 erläutert.

Sollte der Benutzer in einen anderen Schritt navigieren wollen, so kann die Ausführungsumgebung dies anhand der Anfrage feststellen. Insbesondere kann sie feststellen, ob der Benutzer zurück oder vor navigiert. Falls er in den Schritten zurück geht, dann wird, entsprechend des angeforderten Schrittes, die Eingabe desselbigen nicht simuliert, sondern neu angefordert. Die Navigation nach vorn, ist nur solange möglich, wie schon Eingaben des Benutzers vorliegen, beziehungsweise, wo die Simulation erfolgreich ist.

Interessant bleibt die Frage, wie damit umgegangen wird, wenn der Benutzer zurück navigiert und eine Eingabe ändert. Diese Eingabe könnte schließlich Einfluss auf die folgenden Schritte haben. Eine Möglichkeit ist es, alle Folgeschritte zu löschen, doch dies ist nicht unbedingt notwendig. Viel eher scheint es sinnvoll, die Möglichkeit zu schaffen, spezielle Abhängigkeiten zu deklarieren, falls sie nicht sogar automatisch erkannt werden können. Mithilfe dieser Abhängigkeiten könnten spezielle Eingaben invalidiert werden, was nur dann zu einer erneuten Eingabe führen würde. Dies ist auch genau der Punkt, bei dem während der Navigation nach vorn, eine Validierung auf bereits Eingegebenem auftreten kann. In diesem Moment kann dann eine entsprechende Fehlermeldung angezeigt und der Benutzer dazu aufgefordert werden, genau dieses eine Feld neu auszufüllen.

Einige Implikationen Der Speicheraufwand für den Zustand des Prozesses eines Benutzers hängt nur noch von seinen Eingaben ab, die typischerweise sehr klein sowie gut komprimierbar sind. Im Gegensatz zu einem vollständigen Funktionsstack, der je nach Umgebung nicht triviale Größe haben kann, haben wir hier einen Vorteil. Außerdem müssen für die Navigation zwischen den Schritten auch nicht mehrere Continuations verwendet werden, also eine für jeden Schritt, sondern nur eine, da diese alle vorherigen Schritte enthält. Alternativ könnten auch mehrere Continuations gespeichert werden, so könnte der Benutzer auf alle seine vorherigen Eingaben, die er zwischenzeitlich geändert hat, zugreifen. Wir sind auch unabhänig davon, ob der Benutzer die Vor/Zurücktasten in seinem Browser benutzt, oder die entsprechenden Buttons in den Formularen.

Darüber hinaus können wir nun letztendlich von den asynchronen Interaktionen mit dem Benutzer abstrahieren, da bei der finalen Ausführung nichts Asynchrones vorkommt. Alle Eingaben des Benutzers liegen bereits vor, der

Ablauf kann mit einem Mal abgearbeitet werden. Die in Abschnitt VI.3.2 gewählte Abstraktionsebene hält stand.

Ein mögliches Feature Für Unternehmensanwendungen mit besonders langen und komplexen Abläufen könnte noch eine weitere Möglichkeit, die sich aus unserer Implementierung ergibt, von Interesse sein. Es handelt sich um das Teilen von Ablaufzuständen. Wenn ein Mitarbeiter an einer bestimmten Stelle im Ablauf nicht sicher ist, was er eintragen soll, oder Wissen aus einer anderen Abteilung benötigt, dann muss er den Ablauf nicht abbrechen, sondern er kann den entsprechenden eindeutigen Bezeichner an einen Kollegen übermitteln, worauf dieser den Ablauf an genau derselben Stelle aufnehmen und die fraglichen Felder ausfüllen kann. Dank der geringen Größe der zu persistierenden Daten, ist auch das langezeitge Speichern von Ablaufzuständen unproblematisch. Falls dieses Feature nicht gewünscht ist, beziehungsweise falls es Sicherheitsbedenken gibt, kann der entsprechende Bezeichner an die Session des Nutzers gebunden werden.

VI.4.5 Validierung

Wie bereits erwähnt, wird die Validierung unserer Modelle in den Schritten, in denen sie mit Daten befüllt werden, automatisch vorgenommen. Im Folgenden gehen wir genauer auf die Umsetzung ein.

Die Annotation der Modelle wurde bereits in Kapitel V erklärt. Allerdings können wir mit den Annotationen nicht nur Attribute in Gruppen zusammenfassen und je nach Typ richtig anzeigen, wir können sie auch validieren. Mehrere der Annotationen haben einen Einfluss auf die Validierung.

```
1 attribute("country",
2           type = str,
3           view_type = "translated_string",
4           domain = ["GER", "USA"],
5           not_empty = True)
```

Quelltext 29: *Beschreibung des Attributes* `country` *mit unserem Framework*

Dieser Codeausschnitt ist die genauere Beschreibung des Attributes `country` auf einem Modell. Während der `view_type` keinen Einfluss auf die Validierung hat, sind die drei anderen Definitionen relevant.

In Zeile 2 wir der Typ des Attributes definiert. Zwar ist Python eine ungetypte Sprache, jedoch war es für die Anbindung unserer Datenbank notwendig,

den Typ eines Attributs zu definieren, wie in Kapitel II beschrieben. Diese Definition setzt aber nicht nur den Typen des Feldes in der Datenbank, sondern zieht auch eine implizite Typprüfung nach sich. Während das Umwandeln in `string` immer möglich sein wird, könnte dort auch durchaus `float` stehen, womit wir bei falscher Eingabe, dieses sofort an einem Typfehler erkennen können. Zeile 4 definiert den Wertebereich der Attribute. In diesem Beispiel sind stehen nur zwei abgekürzte Ländernamen zur Verfügung. Dies kann auf der Ansichtsseite zum rendern einer entsprechenen Selektionsliste führen. Darüberhinaus kann aber auch unser Validator prüfen, ob die Eingabe in der definierten Menge enthalten ist und ansonsten einen Fehler anzeigen. Zeile 5 definiert die Ausfüllung des Feldes als verpflichtend. Eine Implikation ist, dass das Feld in der Ansicht mit einem Stern markiert wird. Darüberhinaus wird auch ein Validator involviert, der prüft, ob das Feld ausgefüllt wurde.

Schlüsselwörter wie `not_empty` können vom Programmierer selbst eingeführt werden. Dabei können sowohl Auswirkungen auf die Ansichten als auch spezielle Validatoren und deren Fehlermeldungen definiert werden.

Das Marshalling Unter Marschalling versteht man das Umwandlen von Daten in ein anderes Format. In unserem Fall sind es Eingaben aus einem Webformular in Attributwerte eines Datenmodells. Der Marshaller ist dabei die Ausführungseinheit. Wenn der Marshaller ein Attribut aus einem Formularfeld extrahiert und es versucht in das Modell einzufügen, geschieht der erste Validierungsschritt. Schließlich ist unsere Datenbank getypt. Wenn also der Marshaller eine Umwandlung in den richtigen Typ versucht, findet implizit eine erste Validierung des Typs statt. Ein Fehler an diesem Punkt kann dem Benutzer bereits entsprechend angezeigt werden.

Abb. 35: *Die Fehlermeldung wird gleich neben dem entsprechenden Feld angezeigt*

Der Validator Nachdem das Marshalling stattgefunden hat, können weitere Einschränkungen der Wertemenge auf den Attributen überprüft werden. Dafür wird der Validator auf der Instanz des Modells gerufen. Selbiger ermittelt auf Basis der Annotationen, ob beispielsweise der Wert unerlaubterweise leer ist oder eine nicht zulässige Option ausgewählt wurde. Die Einschränkungen können dabei in unserem Framework beliebig komplex formuliert werden.

VI.4.6 Zusammenfassung

In diesem Abschnitt haben wir die genauen Hindernisse bei der Implementierung unserer Lösung und, wie wir sie überwunden haben, erklärt. Wir haben eine alternative allgemeingültige Implementierung von Continuations gesehen, sowie unsere Möglichkeit der automatischen Validierung von Modellen, insbesondere unserer Konzepte für die Formulierung zu validierender Einschränkungen, vorgestellt.

VI.5 Evaluierung

Die vorgeschlagene Art der Ablaufsbeschreibung hat einige bereits genannte Vorteile, allerdings lassen sich auch Probleme erkennen. Diese sollen im Folgenden diskutiert werden. Außerdem möchten wir noch auf die Praxisrelevanz zu sprechen kommen und einen direkten Vergleich mit einem kleinen Beispiel wagen.

VI.5.1 Speicher- und Rechenaufwand

Während wir den Speicheraufwand verringert haben, haben wir den Rechenaufwand erhöht. Schließlich wird der Ablauf bei n Eingaben auch n-mal ausgeführt. Aus Implementierungsdetails ergibt sich sogar die (2*n) + 1 fache Ausführung . Inwiefern diese Einschränkung ein reales Problem ist bleibt fraglich. Zwar sind die einzelnen Schritte von Abläufen weniger komplex, allerdings kann es doch bei größeren Abläufen durchaus ins Gewicht fallen. Letztendlich ist es ein altbekannter Tradeoff [14] zwischen Hauptspeicher- und CPU-Belastung, beziehungsweise Raum gegen Zeit.

VI.5.2 Multibenutzersysteme

Eine weitere diskussionswürdige Frage ist die des gleichzeitigen Zugriffs. Mit unserer Lösung kann es vorkommen, dass ein Benutzer einen Ablauf beginnt und während er noch daran arbeitet, verändert ein anderer Benutzer etwas am selben Modell. Dies würde unter Umständen dazu führen, dass der Benutzer beim nächsten Schritt, in einem vorherigen Schritt landet, in dem Validierungsfehler aufgetreten sind. Der alternative Ansatz mit klassischen Continuations ist, dass das entsprechende Modell für die Zeit der Bearbeitung gesperrt wird. Somit kann kein anderer Benutzer daran arbeiten und den

Ausgangszustand verändern. Welche der beiden Lösungen vorzuziehen ist, ist Geschmacksfrage. Wir halten die erste Variante für durchaus hinnehmbar. Als Vorzug erachten wir außerdem, dass eine potentielle Änderung, die keinen Einfluss auf einen anderen bereits laufenden Ablauf hat, problemlos durchlaufen kann. Dies wäre bei Locking nicht möglich. Außerdem erscheint es gerade bei großen Systemen mit langwierigen Abläufen als risikobehaftet, Modelle langfristig zu sperren.

VI.5.3 Seiteneffekte

Eine unausgesprochene Prämisse unseres Modells ist, dass alle Seiteneffekte der transaktionalen Logik unterliegen. Ein Schritt eines Ablaufes kann theoretisch beliebig oft angestoßen werden. Alle seine Änderungen am System werden dabei jedes Mal verworfen. Daten eines Ablaufes werden nur dann persistiert, wenn er im letzten Schritt angekommen ist und erfolgreich terminiert.

Dieses System funktioniert nur, wenn alle Änderungen am System tatsächlich rückgängig gemacht werden können oder idempotent sind. Gerade aber übliche Aktionen eines Ablaufen in Webanwendungen, wie beispielsweise das Versenden einer Email, sind weder idempotent, noch können sie rückgänig gemacht werden. Wir benötigen für solche Aktionen also ebenfalls transaktionale Logik, um es uns zu ermöglichen, Schritte wie das Versenden einer Email garantiert nur einmal auszuführen. Realisiert werden könnte dies, mit einer Art transaktionalem Universal-Proxy, der sämtliche als extern deklarierten Funktionen erst dann ausführt, wenn der Ablauf abgeschlossen wird.

VI.5.4 Vergleich der Lösung

Eine Evaluierung der Ablaufbeschreibungssprache ist notwendigerweise auch eine Evaluierung unseres Webframeworks. Viele Metriken könnten nun angeführt werden, um unseren Ansatz mit anderen zu vergleichen. Da es sich jedoch bei unserer Lösung nur um einen Prototypen handelt, möchte ich mich auf ein kleines Beispiel beschränken, das möglichst viel von der Aussagekraft unserer Abläufe zeigen.

Seaside Counter Ein anschauliches Beispiel ist der „Seaside Counter“. Hier soll eine Seite geschrieben werden, auf der vier Elemente sichtbar sind. 3 Buttons: Vorwärts, Zurück, +1; und ein Eingabefeld für eine Zahl. Folgende Aktionen sind also auf der Seite möglich. Der +1-Button kann gedrückt werden, was den Wert im Eingabefeld erhöht. Mit dem Vorwärtsschritt wird diese Änderung

gespeichert und man wird auf die gleiche Seite weitergeleitet, allerdings mit der vorherigen Zahl um eins inkrementiert. Mit dem Zurück Button kann man sich wiederrum Schritte zurück bewegen und sieht dabei die vorherigen Stände. In einem Schritt kann die Zählvariable also um 1 oder mehr erhöht werden. Mit unserer Beschreibung kann man es wie folgt formulieren.

```
1 def standard_count_process(process, model,instance):
2       value = count_action(process, 0)
3       while True:
4           value = value + count_action(process, value)
5
6 def count_action(process, value, **kw):
7       values = show_view(process, "count", {"value" : value })
8       return int(values["count[add]"])
```

Quelltext 30: *Der Seaside Counter, beschrieben mit unserem Framework*

Standard CRUD Wie weiter oben beschrieben, können mit unserem Ansatz sämtliche *create*- und *edit*-Funktionen mit einfach gehaltenen Standardfunktionen behandelt werden, solange die Eingabe einschrittig ist. Insbesondere durch die implizite Validierung, ist so gut wie kein Aufwand mit der Implementierung von CRUD-Funktionalität verbunden, was ein starker Unterschied zu gängigen Frameworks wie ASP.NET MVC oder auch Ruby on Rails ist. Das Thema wurde in Abschnitt VI.3.3 und Abschnitt VI.3.4 besprochen.

VI.6 Fazit

Wir haben in dieser Arbeit eine neue Art vorgestellt, Abläufe mit Geschäftslogik zu beschreiben. Dieser neue Ansatz hat sich in einiger Hinsicht als vorteilhaft erwiesen. Die Beschreibung ist zentral in einer Funktion aufgeschrieben, wobei ein sinnvolles Abstraktionslevel verwendet wird. Der Ablauf ist dadurch leicht verständlich und wartbar.

Insbesondere in Verbindung mit der anderen Funktionalität unseres Frameworks, Ansichten zu generieren, ist die Implementierung von Standard-Funktionalität wie CRUD deutlich vereinfacht worden. Der Programmierer muss nicht mehr mehrere Ansichten und Controller-Funktionen anlegen oder modifizieren, sondern nur eine einzige Funktion. Während einige Frage noch zu beantworten sind, bevor man unseren Ansatz in Produktivsystemen einsetzen kann, hat unsere Prototyp sein Ziel erfüllt, indem er die Vorteile und Umsetzbarkeit unseres Ansatzes deutlich demonstriert.

Teil VII

Endnutzerdefinierbare analytische Anfragen

VII.1 Einleitung

Die analytische Verarbeitung von Geschäftsdaten ist ein weit verbreitetes Hilfsmittel, um Erkenntnisse über geschäftsrelevante Vorgänge zu erlangen. Das so erlangte Wissen kann genutzt werden, um Probleme zu erkennen, Maßnahmen abzuwägen und Entscheidungen zu treffen.

Diese Analyse geschieht üblicherweise durch statistische Auswertungen großer Datenmengen (zum Beispiel der Kunden-, Verkaufs- oder Mitarbeiterzahlen), welche den Entscheidungsträgern des Unternehmens durch eine Computeranwendung zur Verfügung gestellt werden. Genau wie auch die Entscheidungen, die diese Personen treffen müssen, unterscheiden sich auch die Anforderungen an diese Auswertungen. So kann man sich beispielsweise vorstellen, dass ein Betrieb, der bisher nur regional tätig war, nun jedoch auch auf nationaler Ebene involviert ist. Für diesen Betrieb wäre nun statt einer bloßen Unterteilung der statistischen Auswertungen nach Landkreis auch eine Aufteilung nach Bundesland nützlich.

Herkömmliche Analyse-Anwendungen mussten für jede solcher neuen Anforderungen von Entwicklern angepasst werden. Der Grund hierfür liegt darin, dass die Datenbankanfragesprachen, die für die Beschreibung der statistischen Auswertungen benutzt werden, nicht verständlich für den üblichen Endnutzer sind. Dieser Umweg über Entwicklungsabteilungen führt jedoch sowohl zu hohen Kosten, als auch zu großen Verzögerungen. Deshalb ist bei moderneren Anwendungen der Trend zu erkennen, dem Nutzer mehr und mehr Konfigurationsmöglichkeiten über eine Bedienoberfläche einzuräumen, um Anpassungen an diesen Auswertungen selbst vornehmen zu können. Meist sind die Auswertungen dann jedoch in ihrer Komplexität und/oder im Grad ihrer Anpassbarkeit beschränkt.

Wir werden in dieser Arbeit zunächst die Gründe für die schwere Verständlichkeit der Datenbankanfragesprachen, sowie die Einschränkungen vorhandener Analyse-Anwendungen untersuchen (Abschnitte VII.2 und VII.3). Anschließend werden wir ein für den Nutzer verständlicheres Anfragemodell vorstellen, welches die Beschreibung von statistischen Auswertungen über eine einfache Anwendungsoberfläche erlaubt, aber dennoch diesen Einschränkungen nicht unterliegt. Dafür formulieren wir in Abschnitt VII.4 die Anforderungen, die dieses Anfragemodell erfüllen soll und gehen in Abschnitt VII.5 genauer auf unsere Umsetzung dieser Anforderungen ein. In Abschnitt VII.6 folgt eine Auswertung der vorgestellten Lösung und in Abschnitt VII.7 fassen wir die Ergebnisse dieser Arbeit zusammen.

VII.2 Verständlichkeitprobleme der Anfragesprachen

Die Datenbankanfragesprachen für sich alleine sind ungeeignet, um vom Nutzer der Analyse-Anwendung für die Formulierung von statistischen Auswertungen verwendet zu werden. Wir werden in diesem Abschnitt die Gründe dafür am Beispiel der relationalen Anfragesprache SQL (Abschnitt VII.2.1) und der OLAP-Anfragesprache MDX (Abschnitt VII.2.2) untersuchen.

VII.2.1 SQL

SQL[24] ist eine strukturierte relationale Anfragesprache, die ursprünglich von D. Chamberlin und R. Boyce für die Firma IBM entwickelt wurde. In [5] beschreiben diese, dass SQL sowohl für Programmierer als auch für den gelegentlichen Datenbanknutzer konzipiert wurde.

Während sich SQL bei Entwicklern defacto als Standardanfragesprache für relationale Datenbanken durchgesetzt hat, ist die Verbreitung unter Endanwendern sehr gering. So zeigt zum Beispiel eine Umfrage unter den SQL-vertrauten Personen in Unternehmen, die relationale Datenbankmanagementsysteme einsetzen, dass weniger als 15% dieser Personen zu der Gruppe der Endanwender gehören [17]. Wir sehen die Hauptgründe hierfür in den folgenden zwei Eigenschaften von SQL.

Zum einen erfordert SQL die Einhaltung einer strikten Syntax und ist deshalb sehr anfällig für Fehler durch ungeübte Nutzer. In [8] vergleicht Davis die textuelle Beschreibung von Anfragen durch SQL mit einem visuellen menüorientierten Interface in einem Experiment unter SQL-unvertrauten Personen. Er beschreibt das Zusammensetzen einer syntaktisch korrekten Anfrage als eines der am häufigsten festgestellten Probleme bei der Anwendung von SQL. Fehlende Anführungszeichen und Klammern oder Tippfehler sind elementare Stolperfallen für ungeübte Nutzer.

Zum anderen entspricht das logische (relationale) Abstraktionsniveau der SQL-Anfragesprache nicht dem konzeptuellen Verständnis des Nutzers. In SQL werden Anfragen auf der Basis von Relationen (Tabellen) und Spalten formuliert. Fremdschlüsselbeziehungen werden genutzt, um Zeilen aus verschiedenen Relationen über eine Join-Operation miteinander zu verknüpfen. Der Nutzer denkt jedoch nicht in Relationen und Join-Operatoren. Üblicherweise hat er noch nicht einmal besondere Kenntnisse über das zugrunde liegende relationale Datenmodell. Er orientiert sich vielmehr an einem auf Entitäten, Zusammenhängen und Attributen basierten Weltverständnis. Chan u.a. zeigen dies in

[24] ehemals SEQUEL, Structured English Query Language

[6]: Ihre Untersuchungen ergaben, dass Nutzer Anfragen mithilfe eines Entity-Relationship-Modells 38% akurater und 65% schneller beschreiben können als mit einem relationalen Modell. Davis stellt in [8] ebenfalls fest, dass Nutzer bei der Formulierung von SQL-Anfragen Probleme damit haben, zu identifizieren, welche Relationen für eine Anfrage relevant sind.

VII.2.2 MDX

MDX[25] wurde von Microsoft für die Abfrage von Daten aus mehrdimensionalen OLAP[26]-Datenbanken eingeführt [20]. Diese Sprache lehnt in ihrer Syntax stark an SQL an, jedoch liegt ihr ein anderes Organisationskonzept der Daten zugrunde, denn diese werden in den Abfragen in der Form von OLAP-Würfeln (engl. *OLAP cubes*) genutzt. Ein OLAP-Würfel besteht aus einer Menge von Metriken und Dimensionen (*Measures* und *Dimensions*) [9]. Die Metriken beschreiben die Fakten, die abgefragt werden können (z.B. der Umsatz). Über die Dimensionen (z.B. die Zeit oder die Standorte des Unternehmens) können diese Metriken gruppiert werden.

Dieses Konzept der OLAP-Würfel ist aus unserer Sicht auch nicht für die Endnutzer von Analyse-Anwendungen intuitiv verständlich. Denn auch dieses Modell entspricht nicht dem oben beschriebenen konzeptuellen Verständnis des Nutzers von der Geschäftswelt. Dieses Anfragekonzept ist vielmehr ein logisch durch die Datenspeicherstrukturen bedingtes Modell. Denn der Zweck eines OLAP-Würfels ist es, die optimierte Speicherstruktur einer multidimensionalen Datenbank (MDDB) in ein Modell zu fassen.

Die OLAP-Würfel müssen außerdem vorher konfiguriert werden, d.h. die verfügbaren Metriken und Dimensionen müssen definiert werden. Deshalb stehen in den MDX-Anfragen dann nur bestimmte Metriken und Dimensionen zur Verfügung. Sie sind also in der Flexibilität eingeschränkt. Falls die Anforderungen an die zu beschreibende statistische Auswertung andere Metriken oder Dimensionen erfordern, muss der Würfel angepasst werden.

Da MDX in einer ähnlichen textuellen Weise wie SQL formuliert wird, entstehen dieselben syntaktischen Schwierigkeiten für den Nutzer, wie auch bei SQL. Die Syntax wird zusätzlich dadurch erschwert, dass die Auswahl der Metriken und Dimensionen über bestimmte Klammerungen geschieht.

Zusammenfassend zeigt sich also, dass sowohl SQL, als auch MDX, auf einem logischen Modell basieren, welches auf der Art der Speicherung der Daten

[25] Multidimensional Expressions
[26] Online Analytical Processing

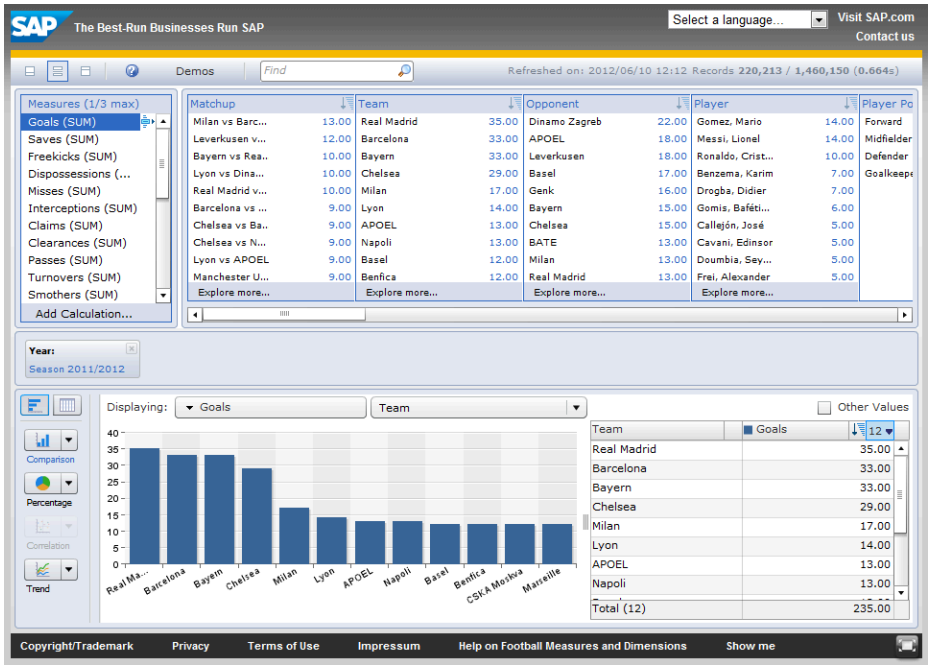


Abb. 36: *Nutzerschnittstelle des SAP BusinessObjects Explorers.*

beruht und nicht dem intuitiven Gedankenmodell des Nutzers entspricht. Sie besitzen außerdem durch ihre komplizierte Syntax keine für den Endnutzer geeignete Formulierungsweise. Aus diesen Gründen ist ihre Anwendung für den Endnutzer nicht gut geeignet.

VII.3 Einschränkungen vorhandener Analyse-Anwendungen

Aufgrund der Untauglichkeit der Anfragesprachen für den Endnutzer ist eine benutzerfreundliche Bedienoberfläche notwendig, um dem Nutzer die Anpassung von statistischen Auswertungen zu erlauben. Wir werden in diesem Abschnitt eine Anwendung der Firma *SAP*, den *BusinessObjects Explorer*, untersuchen und sowohl dessen Vorteile gegenüber der Anfragesprachen erläutern, als auch auf die Einschränkungen eingehen, denen diese Anwendung unterliegt.

VII.3.1 SAP BusinessObjects Explorer

Die Bedienoberfläche der Softwarelösung *SAP BusinessObjects Explorer* erlaubt dem Nutzer, individuelle Diagramme auf der Basis von Filtern und Metriken

darzustellen. In Abbildung 36 ist diese Oberfläche mit Beispieldaten aus dem europäischen Fußball dargestellt [27]. Die Abbildung zeigt eine Statistik der Anzahl der Tore der verschiedenen Vereine in der Saison 2011/2012.

Um eine solche Statistik zu erstellen, wählt der Nutzer zuerst die gewünschten Metriken aus einer Liste aus (linker oberer Bereich der Oberfläche). In der Abbildung ist die Summe der Tore als Metrik ausgewählt. Dann hat der Nutzer die Möglichkeit, im rechten oberen Bereich Filter auf den Datenbestand anzuwenden. Er kann für jedes filterbare Attribut aus einer Liste von Werten ein oder mehrere Werte auswählen, um die Ergebnisdaten einzuschränken. Die angewendeten Filter werden in der mittleren Reihe der Oberfläche angezeigt. Dabei handelt es sich in Abbildung 36 um einen Filter, der das Jahr auf die Fußballsaison 2011/2012 fixiert. Durch das Bestimmen einer Gruppierungsdimension aus einer Auswahlbox im unteren Bereich des *Explorers* (in der Abbildung ist dies der Verein) sind alle nötigen Daten für die Darstellung der Ergebnisse der Abfrage in Form einer Tabelle und eines Diagramms vorhanden.

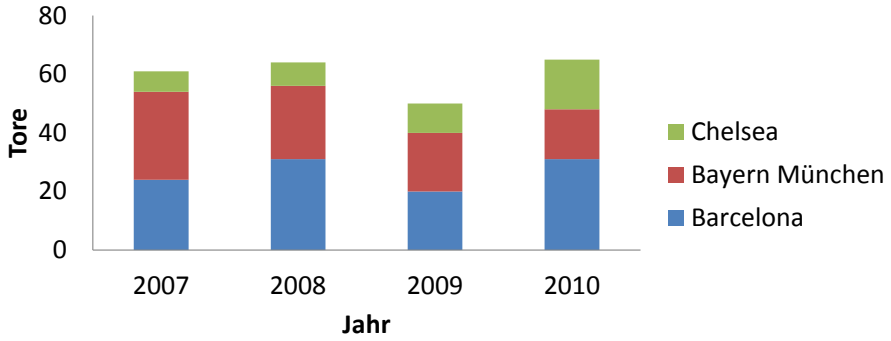
Vorteile gegenüber Anfragesprachen. Die grafische „Point-and-Click"-Nutzerschnittstelle des *Explorers* löst das Problem der komplizierten Syntax textueller Anfragesprachen durch einfache Bedienelemente. Der Nutzer muss keine Vorkenntnisse über Zeichensetzung oder ähnliche Dinge vorweisen, wodurch die Hürde für ungeübte Datenbanknutzer gegenüber Anfragesprachen geringer ist. Auch stellt der *Explorer* den Nutzer nicht vor ein relationales Datenbankschema oder OLAP-Würfel, sondern erlaubt ihm, die Auswertungen anhand eines vereinfachten Entitätsmodells zu beschreiben. Er geht sogar so weit, die Zusammenhänge zwischen den einzelnen Entitätsmengen (z.B. zwischen Toren, Vereinen und Begegnungen) auszublenden und hinter den Metriken zu verstecken. Der Nutzer muss diese Zusammenhänge also nicht selbst verfolgen.

Einschränkungen. Durch diese Vereinfachungen sind allerdings die Flexibilität und Mächtigkeit des *BusinessObjects Explorers* beschränkt. In Hinblick auf die Flexibilität resultieren diese zum Beispiel in der festen Vorgabe der zur Auswahl stehenden Metriken und Gruppierungsdimensionen. Wird eine zusätzliche Metrik notwendig, wie zum Beispiel eine Durchschnittsbildung statt einer Summierung oder eine Metrik über ein bisher überhaupt nicht vorgesehenes Attribut (z.B. die Spielergröße), so kann dies nur durch eine Anpassung des *Explorers* selbst umgesetzt werden. Auch erlaubt die Filterauswahl in der aktuellen Form nur eine Auswahl anhand diskreter Werte. Eine Selektion über

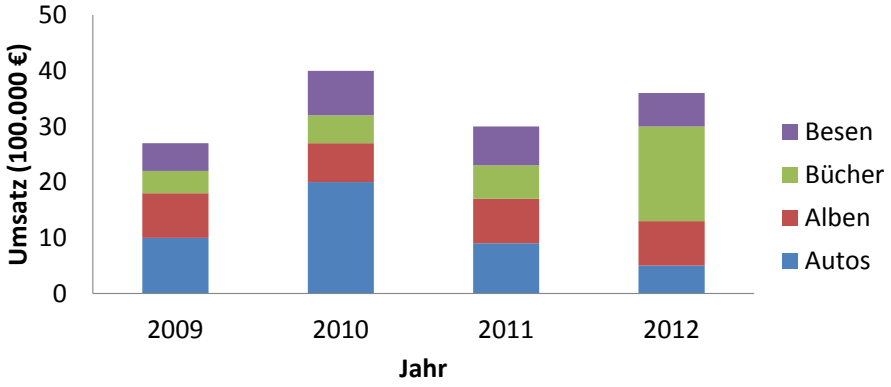
[27] Screenshot vom 10.06.2012 aus einer Beispielanwendung, verfügbar unter `http://coeportal98.saphosting.de/eurofootball/explorer.html`

einen kontinuierlichen Wertebereich zwischen einem Start- oder Endwert ist nicht möglich.

Die aus unserer Sicht größte Einschränkung liegt jedoch in der Mächtigkeit der Anfragen, die mithilfe des *Explorers* erstellt werden können, denn die resultierenden Statistiken besitzen nur zwei Dimensionen. Es gibt keine Möglichkeit, Statistiken mit drei oder mehr Dimensionen zu erstellen. Jedoch gibt es hierfür viele denkbare Anwendungsfälle. In der Domäne der Beispielanwendung wäre zum Beispiel eine Statistik über die zeitliche Entwicklung der Summe der Tore aller einzelnen Vereine denkbar, wie sie in Abbildung 37(a) dargestellt ist. Übertragen in die Geschäftswelt kann man sich als beispielhafte Statistik die zeitliche Entwicklung des Umsatzes durch die einzelnen Produktgruppen des Unternehmens vorstellen (Abbildung 37(b)).



(a) Summe der Tore pro Verein über die Zeit.



(b) Umsatz pro Produktgruppe über die Zeit.

Abb. 37: *Beispiele für dreidimensionale Statistiken.*

VII.4 Anforderungen an unsere Lösung

Aus unserer Sicht erfordern die in Abschnitt VII.3.1 besprochenen Einschränkungen die Suche nach einem alternativen Anfragemodell, welches ebenfalls die Beschreibung von statistischen Auswertungen über eine für den Nutzer verständliche Oberfläche erlaubt, jedoch die Probleme des *BusinessObjects Explorers* im Hinblick auf Flexibilität und Mächtigkeit überwindet.

Um die Anforderungen an unsere Lösung (Abschnitt VII.4.2) aufstellen zu können, haben wir uns zunächst damit beschäftigt, wie sich eine statistische Auswertung grob zusammensetzt (Abschnitt VII.4.1). Hierfür haben wir uns diverse Anwendungsfälle und Beispiele für Statistiken angesehen und dabei einige Gemeinsamkeiten festgestellt und in grundlegende Konzepte gefasst.

VII.4.1 Statistische Auswertungen

Die statistische Auswertung, die in Abbildung 37(b) dargestellt ist, kann natürlichsprachlich ausgedrückt werden als „Umsatz pro Produktgruppe pro Jahr“. In dieser Formulierung finden sich eine Metrik (Umsatz) und zwei Gruppierungen dieser Metrik (Produktgruppe und Jahr) wieder. Sowohl die Metrik als auch die Gruppierungen stellen Dimensionen in dem resultierenden Diagramm dar. Dieses Konzept trifft für alle Arten von statistischen Auswertungen zu. Eine oder mehrere Metriken werden anhand einer gewissen Anzahl von Gruppierungen aggregiert. Die aggregierten Werte für jede Kombination der Gruppierungen werden üblicherweise grafisch dargestellt.

Eine statistische Auswertung bezieht sich außerdem immer auf eine bestimmte Menge von Datensätzen. Dies können alle im System vorhandenen Datensätze sein, jedoch kann es sich auch nur um eine Teilmenge dieser handeln. Die Auswertung in Abbildung 37(b) bezieht sich auf alle Datensätze, jedoch könnte man sie beispielsweise auch auf die Bestellungen beschränken, die ins Ausland geliefert worden sind.

VII.4.2 Anforderungen

Die Anforderungen, welche unsere Lösung erfüllen soll, definieren wir wie folgt.

1. Dem Anfragemodell sollte statt einem logischen, technisch bedingten Datenmodell ein konzeptionelles Datenverständnis ähnlich dem Entity-Re-

lationship-Modell zugrunde liegen, um dem Endnutzer das Verständnis zu erleichtern.
2. Das Modell muss die Definition von unbegrenzt mehrdimensionalen statistischen Auswertungen unterstützen. Dabei muss eine Möglichkeit gefunden werden, zu definieren, nach welchen Attributen gruppiert bzw. aggregiert wird.
3. Alle theoretisch möglichen Metriken sollten im Anfragemodell selbst definier- und nutzbar sein. Auch sollte die Art und Weise der Aggregation (z.B. Summe oder Durchschnitt) einstellbar sein.
4. Es muss die Möglichkeit geben, Filter auf allen vorhandenen Attributen der Entitäten einzurichten, um die Datenmenge, auf der die Auswertung basiert, zu beschränken. Die Spezifikation des Wertebereichs sollte dabei sowohl diskret als auch kontinuierlich erfolgen können.
5. Das Modell muss die Konfiguration über eine einfache Nutzerschnittstelle erlauben. Auf die Eingabe von Text, wie beispielsweise Ausdrücken zur Definition einer Dimension, sollte dabei verzichtet werden können. Eine Außnahme kann die Eingabe von Attributwerten darstellen (z.B. für die Definition von Filtern).
6. Als nichtfunktionale Anforderungen führen wir an dieser Stelle sowohl nur geringe Geschwindigkeitseinbußen gegenüber statischen, nicht konfigurablen Anfragen (nicht länger als die doppelte Ausführungszeit) als auch die Möglichkeit einer schnellen und einfachen Prototypen-Implementierung des Anfrageprozessors und einer simplen Bedienoberfläche als Teilprojekt eines Bachelorprojektes (maximal zwei Mannwochen) auf.

VII.5 Umsetzung

Um die erste Anforderung zu erfüllen, also die Verständlichkeit für den Nutzer zu erleichtern, haben wir uns entschlossen, ein objektorientiertes Datenmodell als Grundlage für die Spezifikation von statistischen Auswertungen mithilfe unserer Lösung zu nutzen. Das objektorientierte Datenverständnis spiegelt, ähnlich wie ein Entity-Relationship-Modell, das konzeptionelle Verständnis des Nutzers wieder. Modellentitätsmengen werden durch Klassen repräsentiert, und Beziehungen können direkt in diesen als referenzierende Attribute modelliert werden. Hierdurch können wir von der tatsächlichen logischen Datenhaltung (wie z.B. Relationen und Fremdschlüsseln oder OLAP-Würfeln) abstrahieren und dem Nutzer die Beschreibung der Auswertungen direkt auf einem seiner Vorstellung nahem Niveau erlauben. (Ein Beispiel für ein solches Modell ist in Abbildung 39 abgebildet.)

In Abschnitt VII.5.1 werden wir das Anfragemodell vorstellen, das wir als Lösung der in den Abschnitten VII.2 und VII.3 beschriebenen Probleme vorschlagen. Dieses fungiert als Schnittstelle zwischen einem Anfrageprozessor und ei-

ner Nutzerschnittstelle, deren Implementierungen nicht direkt durch das Anfragemodell festgeschrieben sind. Der Anfrageprozessor bildet auf das unterliegende logische Datenmodell ab, die Nutzerschnittstelle kann eine einfache Bedienoberfläche für den Endnutzer darstellen. Um in Abschnitt VII.6 eine Auswertung des Anfragemodells durchführen zu können, haben wir diese beiden Komponenten prototypenhaft implementiert. Auf diese Implementierungen werden wir in den Abschnitten VII.5.2 und VII.5.3 eingehen.

VII.5.1 Anfragemodell

Das Anfragemodell (Abbildung 38) erfasst die Konzepte einer statistischen Auswertung in Form einer Anfrage an das Datenmodell. Wir haben uns bei der Definition des Anfragemodells ebenfalls für einen objektorientierten Ansatz entschieden. Jede Anfrage ist ein Objekt, welches dem Anfrageprozessor zur Evaluation übergeben werden kann. Die Nutzerschnittstelle bietet dem Anwender eine Oberfläche, die die Konfiguration dieser Anfrageobjekte erlaubt. Dabei ist die genaue Darstellungsweise nicht festgelegt, sie kann also unabhängig vom Aufbau des Anfragemodells gewählt werden.

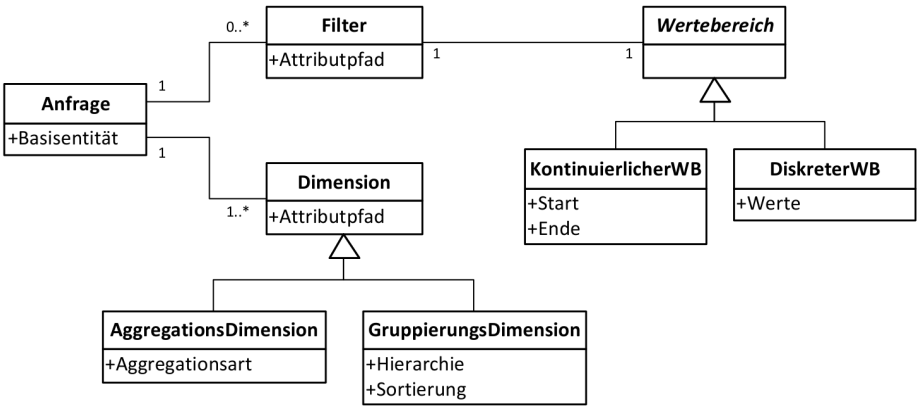


Abb. 38: *Aufbau des Anfragemodells (vereinfachtes Klassendiagramm).*

Anfragen. Eine Anfrage bezieht sich immer auf eine Basis-Datenentität, deren Entitätsmenge die Grundlage für alle Gruppierungen und Aggregationen bildet. Wenn zum Beispiel eine Statistik über den durchschnittlichen Produktpreis pro Produktgruppe erstellt werden soll, so wäre die Basisentität dieser Anfrage das Produkt. Von diesem aus kann nun definiert werden, dass als Aggregation eine Durchschnittsbildung über den Preis jedes Produktes und als Gruppierung die Produktgruppe des Produkts genutzt werden sollen. Hierfür haben wir in unserem Anfragemodell das Konzept von Dimensionen aufgegriffen.

Dimensionen. Jede Dimension, die in einer Anfrage definiert wird, spiegelt direkt eine Dimension der Statistik wieder, welche mithilfe dieser Anfrage erstellt wird. Eine Anfrage besteht aus mindestens einer Dimension, die die abzubildende Metrik beschreibt. Diese Dimension wird im einfachsten Fall durch ein Attribut der Basisentität und der Art, wie über dieses aggregiert werden soll, spezifiziert. Im Beispiel oben wäre dies das Attribut *Preis* und die Aggregationsart *Durchschnitt*. Zusätzlich zu dieser ersten Dimension können beliebig viele weitere Dimensionen in einer Anfrage beschrieben werden, die Gruppierungen der Aggregation einführen. Ohne diese würde die Aggregation der Metrik über alle Instanzen in der Entitätsmenge durchgeführt und es entstünde nur ein Wert (im Beispiel der Durchschnittspreis aller Produkte). Die Gruppierungsdimensionen werden im einfachsten Fall nur durch das Attribut beschrieben, über das die Gruppierung durchgeführt wird. Dies wäre für die Beispiel-Anfrage das Attribut *Produktgruppe*. Hierdurch wird die Aggregation separat für jeden Wert des Gruppierungsattributs durchgeführt. Übertragen auf das Beispiel wird also der Durchschnitt des Produktpreises separat für jede Produktgruppe gebildet. Natürlich ist es auch möglich, zusätzliche Aggregationsdimensionen anzugeben, sodass in der resultierenden Statistik zwei Metriken über dieselben Gruppierungen abgebildet werden. Für eine geordnete Zusammenstellung der Ergebnisdaten kann bei den Gruppierungsdimensionen zusätzlich die Sortierung (*aufsteigend/absteigend*) angegeben werden.

Attributpfade. Um komplexere Anfragen zu spezifizieren, reicht die Angabe eines einfachen Attributnamens für die Dimensionen nicht mehr aus. Hierfür betrachten wir die Statistik in Abbildung 37(b) (Umsatz pro Produktgruppe pro Jahr). Wenn wir davon ausgehen, dass mit einer Bestellung immer nur ein Produkt bestellt wird, fungiert die *Bestellung* als Basisentität dieser Anfrage. Die Aggregation geschieht über den Preis der Bestellung, die Gruppierungen zum einen über die Produktgruppe des Produktes, das in der Bestellung bestellt wurde, und zum anderen über das Jahr, in dem die Bestellung aufgegeben wurde. Wie in dieser Beschreibung zu erkennen ist, müssen für die Angabe der Dimensionen nun Referenzen verfolgt werden. Hierfür bietet sich als Notation für unser Modell die in der Objektorientierung übliche Punkt-Notation an, um den Pfad zum zu verwendenden Attribut anzugeben. Im betrachteten Beispiel kann dadurch die erste Gruppierungsdimension durch den Attributpfad *Produkt.Produktgruppe* beschrieben werden, da der Pfad von der Bestellung zunächst zum referenzierten Produkt und anschließend zur Produktgruppe dieses Produktes führt. Analog ist der Pfad zur zweiten Gruppierungsdimension *Bestelldatum.Jahr*. Für die Aggregationsdimension ist keine Verfolgung von Referenzen nötig, deshalb entspricht der Attributpfad dieser Dimension dem Attributnamen *Preis*.

Diese Notation erlaubt auch die Nutzung von Instanzmethoden der Modellentitäten, um berechnete Werte in der Statistik darzustellen. Wenn sich bei-

spielsweise der Gesamtpreis der Bestellung durch das mathematische Produkt aus dem Preis des bestellten Produktes und der bestellten Anzahl an Exemplaren dieses Produktes zusammensetzt, könnte es in der Klasse, die die Modellentität *Bestellung* repräsentiert, eine Instanzmethode mit dem Namen *Gesamtpreis* geben, die diesen Wert berechnet und zurückgibt. Im Attributpfad kann man diese Instanzmethoden benutzen, als wären es normale Attribute der Entitäten. In diesem Fall entspräche der Attributpfad der Aggregationsdimension im Beispiel dem Namen dieser Methode, er lautete also *Gesamtpreis*. Allerdings funktioniert dies nur, solange der berechnete Wert in einer Methode im Datenmodell erfasst ist. Um diese Einschränkung des Anfragemodells bezüglich der Flexibilität zu lösen, ist als Erweiterung dieser Notationsweise auch die Spezifikation einer Dimension durch einen mathematischen Ausdruck denkbar. Hierbei können die Attribute, die in diesem Ausdruck zur Berechnung verwendet werden sollen, weiterhin über ihre Attributpfade bestimmt werden und mit mathematischen Operatoren verknüpft werden. Um die Berechnung des Gesamtpreises der Bestellung auf diese Art zu formulieren, genügt der Ausdruck *(Produkt.Preis * Anzahl)*. Wir haben uns in unserer Prototyp-Implementierung jedoch aus Zeitgründen auf einfache Attributpfade mit Instanzmethodennutzung beschränkt.

Hierarchien. Für besondere Anfragen ist zudem ein weiteres Konzept für die Spezifikation von Gruppierungsdimensionen nötig. Würden wir beispielsweise den Umsatz nicht wie in Abbildung 37(b) pro Jahr, sondern pro Jahr und Monat in einer Dimension der resultierenden statistischen Auswertung gruppieren wollen (sodass beispielsweise in einem Säulendiagramm eine Säule für jeden Monat in jedem Jahr dargestellt werden würde), so reicht die Angabe eines Attributpfades alleine nicht mehr aus. Dieses Problem lösen wir durch die Angabe einer Hierarchie zum ausgewählten Attribut. Im beschriebenen Fall würde die Gruppierung über das *Bestelldatum* geführt werden, wobei die Attribute *Jahr* und *Monat* des Bestelldatums die Gruppe bestimmen. Die Hierarchie dieser Dimension wäre demzufolge das Tupel *(Jahr, Monat)*. Wenn nötig, kann die Hierarchie einer Dimension also über ein Tupel aus mindestens einem Attributpfad definiert werden. Die Attributpfade in diesem Tupel beziehen sich auf das Attribut, welches durch den Attributpfad der Dimension bestimmt wird. Eine alternative Lösung dieses Problemes wäre die Definition mehrerer Attributpfade für eine Dimension, also beispielsweise über das Tupel *(Bestelldatum.Jahr, Bestelldatum.Monat)*. Diese Variante erfordert jedoch, Teile des Attributpfades doppelt aufzuführen. Deshalb haben wir uns in unserer Prototypen-Implementierung für die zusätzliche Angabe der Hierarchie entschieden.

Filter. Die einer Anfrage zugrunde liegende Datenmenge kann durch die Angabe von Filtern beschränkt werden. Ein Filter bezieht sich dabei immer auf ein Attribut, welches über einen Attributpfad spezifiziert wird. Zusätzlich wird

ein Wertebereich definiert, in welchem sich der Wert dieses Attributs befinden soll. Dieser Wertebereich kann entweder diskret durch die Auflistung einer Menge von zugelassenen Werten oder kontinuierlich durch die Definition eines Start- und eines Endwertes angegeben werden.

Wir haben aus Zeitgründen in unserer Prototypen-Implementierung vorerst darauf verzichtet, die Komposition von Filtern durch logische Operatoren zu unterstützen. Hierfür kann jedoch eine gewöhnliche objektorientierte Baumstruktur für Ausdrücke verwendet werden, bei der die logischen Operatoren (zum Beispiel *UND* bzw. *ODER*) die Knoten repräsentieren und die Objekte der *Filter*-Klasse die Blätter.

Anfragebeispiel. In Abbildung 40 ist das Objektdiagramm der Anfrage dargestellt, die die Auswertung in Abbildung 37(b) beschreibt. Das zugrunde liegende Datenmodell ist in Abbildung 39 vereinfacht abgebildet. Im Vergleich zu den vorigen Beispielen ist anzumerken, dass dieses Datenmodell Bestellungen mit mehreren Posten vorsieht, welche unterschiedliche Produkte darstellen können. Aus diesem Grund ist die Basis-Entitätsmenge der Anfrage nicht die Menge der Bestellungen, sondern die der Posten auf den Bestellungen.

Abb. 39: *Datenmodell im Anfragebeispiel (vereinfachtes Klassendiagramm).*

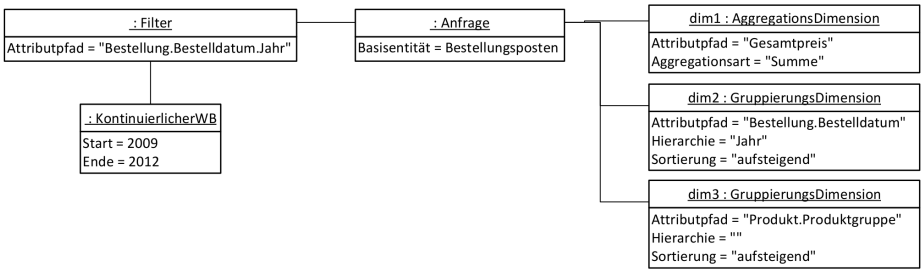


Abb. 40: *Anfragebeispiel (Objektdiagramm).*

VII.5.2 Anfrageprozessor

Für eine grafisch aufbereitete Darstellung einer mithilfe des Anfragemodells formulierten statistischen Auswertung müssen zunächst die Ergebnisse der Anfrage ermittelt werden. Hierfür ist der Anfrageprozessor zuständig. Er

erstellt aus einer gegebenen Anfrage eine Ergebnistabelle, welche für jede Werte-Kombination der in der Anfrage definierten Gruppierungen eine Zeile mit den aggregierten Werten der Metriken für diese Gruppe enthält.

Wir werden zunächst auf einen gescheiterten Versuch, den Anfrageprozessor auf Basis der *ASP.NET*[28]-Technologie umzusetzen, und die dafür ausschlaggebenden Probleme eingehen. Anschließend erläutern wir unsere prototypenhafte Implementierung basierend auf einer sprachintegrierten Datenbank in Python.

```
1  var ergebnis = database.Bestellungsposten
2    .Include("Bestellung")
3    .Include("Produkt")
4    .Where(posten =>
5      posten.Bestellung.Bestelldatum.Year >= 2009 &&
6      posten.Bestellung.Bestelldatum.Year <= 2012)
7    .GroupBy(posten => new {
8      Jahr = posten.Bestellung.Bestelldatum.Year,
9      Produktgruppe = posten.Produkt.Produktgruppe })
10   .Select(group => new {
11     Jahr = group.Key.Jahr,
12     Produktgruppe = group.Key.Produktgruppe,
13     Gesamtpreis = group.Sum(posten => posten.Gesamtpreis())
14     })
15   .OrderBy(zeile => zeile.Jahr)
16   .ThenBy(zeile => zeile.Produktgruppe)
17   .ToList();
```

Quelltext 31: *Beispielanfrage in LINQ. Die* `Include`*-Anweisungen sind erforderlich, um LINQ anzuweisen, die mit der Basisentität verknüpften Objekte mit in die Anfragedaten aufzunehmen. Mithilfe von Lambda-Ausdrücken wird in den anderen Anweisungen spezifiziert, welche Bedingungen geprüft oder welche Attribute selektiert werden sollen.*

Implementierungsversuch auf Basis von ASP.NET

Technischer Hintergrund. Die im ersten Implementierungsversuch erstellte C# und ASP.NET-Webanwendung nutzt eine MySQL[29]-Datenbank zur Speicherung der Daten. Um die dort relational abgelegten Daten in ein Objektmo-

[28] Webanwendungsrahmenwerk von Microsoft, `http://www.asp.net/` (Stand: 12.06.2012)

[29] Weit verbreitete SQL-Datenbank, `http://www.mysql.com/` (Stand: 12.06.2012)

```
1 x => new {
2   Item1 = (Object) AttributeValue(type, x,
3                                   dim0.AttributPfad),
4   Item2 = (Object) AttributeValue(type, x,
5                                   dim1.AttributPfad)
6 }
```

Quelltext 32: *Gruppierungslambda für zwei Gruppierungsdimensionen. Die Variable* `type` *spezifiziert den Typ der Basisentität, die Variablen* `dim0` *und* `dim1` *sind Dimensionsobjekte.*

dell zu überführen, kommt das Rahmenwerk *LINQ to Entites*[30] zur objektrelationalen Abbildung zum Einsatz, sodass jede Relation durch eine C#-Klasse repräsentiert wird. *LINQ*[31] stellt außerdem eine deklarative sprachintegrierte Anfragesprache zur Verfügung, die wir zur Übersetzung des Anfragemodells in tatsächliche Anfragen an die MySQL-Datenbank benutzen. Der syntaktische Aufbau dieser LINQ-Anfragen erinnert durch gleichlautende Anweisungen (wie `Select`, `Where` oder `OrderBy`) etwas an SQL. Jedoch werden die Parameter dieser Anweisungen durch C#-Lambda-Ausdrücke spezifiziert. Quelltext 31 zeigt eine LINQ-Anfrage, die genutzt werden kann, um die Daten für die in Abbildung 40 beschriebene statistische Auswertung zu ermitteln.

Aufbau der Implementierung. Der Anfrageprozessor hat nun die Aufgabe, eine wie in Quelltext 31 aufgeführte LINQ-Anfrage dynamisch aus einer Instanz unseres Anfragemodells (im Folgenden *Anfrageobjekt* genannt) zu erstellen. Die im Anfrageobjekt spezifizierten Filter werden dafür auf die `Where`-Anweisung der LINQ-Query abgebildet. Die `GroupBy`-, `Select`- und `OrderBy`-Anweisungen setzen sich aus den Dimensionen des Anfrageobjekts zusammen. Für jede Gruppierungsdimension wird ein Eintrag in dem dynamischen Objekt hinzugefügt, das in der `GroupBy`-Anweisung erstellt wird (*new-Struktur in Zeilen 7-9*). Die `Select`-Anweisung wählt dann die Gruppenschlüssel und aggregierten Werte der Aggregationsdimensionen als Spalten des Ergebnisdatensatzes aus. Falls die Gruppierungsdimensionen Sortierungen spezifiziert haben, werden diese mittels der `OrderBy/ThenBy`-Anweisungen umgesetzt.

In Quelltext 32 ist beispielhaft ein Lambda-Ausdruck dargestellt, der eine Gruppierung über zwei Dimensionen durchführt. Er kann also anstelle des Lambda-Ausdrucks in den Zeilen 7-9 in Quelltext 31 fungieren. Wie in diesem

[30] Technologie zur Abbildung von LINQ-Anfragen auf Datenbankentitäten, `http://msdn.microsoft.com/de-de/library/bb386964.aspx` (Stand: 12.06.2012)

[31] *Language Integrated Query*, `http://msdn.microsoft.com/en-us/library/bb397926` (Stand: 12.06.2012)

```
1  private Object AttributeValue(Type type,
2      Object instance, string path) {
3    string[] fieldNames = path.Split('.');
4
5    foreach (string field in fieldNames) {
6      PropertyInfo property = type.GetProperty(field);
7      instance = property.GetValue(instance, null);
8      type = property.PropertyType;
9    }
10
11   return instance;
12 }
```

Quelltext 33: *Auflösung des Attributpfads in C# (vereinfacht: ohne Methodenauflösung). Der Attribtpfad wird in die einzelnen Attributnamen zerlegt. Anschließend werden die Attribute in der `foreach`-Schleife nacheinander aufgelöst. Mithilfe der Introspektionsmethode `GetProperty()` werden die Informationen zum jeweiligen Attribut zur Laufzeit abgefragt und anhand dieser der Wert und Typ des Attributs ermittelt. Auf diesem Attributwert wird anschließend mit der Auflösung der restlichen Attribute aus dem Attributpfad fortgefahren.*

Lambda-Ausdruck zu sehen ist, findet die Auswahl der Attributwerte anhand des Attributpfads statt. Hierfür werden die Introspektionsfähigkeiten[32] von C# genutzt (Quelltext 33). Auf eine ähnliche Weise wie im dargestellten Gruppierungslambda werden auch die Lambda-Ausdrücke zur Selektion der Attributwerte der Aggregationsdimensionen erstellt. Auf diesen wird anschließend in Abhängigkeit der im Anfrageobjekt spezifizierten Aggregationsart eine Aggregationsfunktion ähnlich wie in Zeile 13 in Quelltext 31 ausgeführt.

Probleme bei der Implementierung. Mit LINQ ist es sehr aufwendig, Lambda-Ausdrücke ähnlich derer in Quelltext 32 für eine dynamische Anzahl von Attributen zu erstellen. Denn hierfür müsste nicht nur die Auflösung der Attributpfade über Introspektion geschehen, sondern auch die LINQ-interne Baumstruktur, die den jeweiligen Lambda-Ausdruck repräsentiert, manuell erstellt werden. In den vorigen Beispielen wurden diese durch die Lambda-Notationsweise automatisch erzeugt. Aufgrund dieses Problems haben wir die beschriebene ASP.NET-Implementierung auf Anfrageobjekte mit maximal zwei Gruppierungsdimensionen und einer Aggregationsdimension beschränkt.

Jedoch unterliegt unser Lösungsansatz noch tiefgreifenderen Problemen, denn unsere Lösung funktioniert in der vorliegenden Form nur mit lokal in der

[32] Introspektion: Auswertung von Typinformationen zur Laufzeit

Anwendung verfügbaren Daten. Die Anbindung an die MySQL-Datenbank kann nicht genutzt werden. Dies liegt daran, dass *LINQ to Entities* die von uns erstellten Anfragen nicht in SQL-Anfragen übersetzen kann. Dies wiederum hat den Ursprung darin, dass *LINQ to Entities* eine LINQ-Anfrage vollständig in eine SQL-Anfrage übersetzt. Daher ist es nicht möglich, innerhalb der Lambda-Ausdrücke die Introspektionsfunktionalität von C# zu nutzen (wie beispielsweise die `GetProperty`-Methode aufzurufen), denn diese Introspektion ist nicht in eine SQL-Anfrage übersetzbar. Wenn die LINQ-Anfrage auf C#-Objekten ausgeführt wird, die nicht in der Datenbank residieren, muss diese Übersetzung nicht vorgenommen werden, deshalb ist hier die Nutzung von Introspektion möglich. Die Lösung dieses Problems wäre es, die Auflösung des Attributpfades außerhalb des Lambda-Ausdrucks vorzunehmen und auch diesen Teil des Ausdruckbaums manuell zu erstellen, wie der Quelltext 34 skizziert.

```
1  public Expression BuildGet(Type type, string path){
2    string[] fieldNames = path.Split('.');
3    Expression arg = Expression.Parameter(type, "x");
4    Expression expr = arg;
5
6    foreach (string field in fieldNames) {
7      PropertyInfo property = type.GetProperty(field);
8      type = property.PropertyType;
9      expr = Expression.Property(expr, property);
10   }
11
12   return Expression.Lambda(expr, arg);
13 }
```

Quelltext 34: *Manuelle Erstellung des Lambda-Ausdruckbaums für den Attributwert (vereinfachte Typisierung der Ausdrücke). Die Informationen über die Attribute werden dazu genutzt, einen Lamda-Ausdruck zu erstellen, der auf die einzelnen Instanzen angewendet werden kann. Ausschlaggebend für die Komplexität einer Lösung auf diese Art und Weise sind die generischen `Expression`-Objekte von LINQ, die für die Erstellung des Ausdruckbaums benötigt werden.*

Ebenso problematisch für die Übersetzung in SQL-Anfragen ist die Nutzung von Typumwandlungen zu abstrakten Datentypen, wie `Object`, die wir für die generische Implementierung der Lambda-Ausdrücke in unserer Lösung benutzen müssen. *LINQ to Entities* bricht einen solchen Versuch mit der Fehlermeldung ab, dass der Datentyp nicht unterstützt wird. Auch dies kann dadurch gelöst werden, (typen-)spezialisierte Anfragen dynamisch mittels Introspektion zu generieren.

Wir sehen also, dass die zusätzliche Schicht der objektrelationalen Abbildung die Entwicklung dynamischer Anfragen erschwert, da die dynamischen As-

pekte nicht in der Anfrage selbst ausgewertet werden können, sondern für jedes Anfrageobjekt eine vollständig zugeschnittene Anfrage mittels Introspektion erstellt werden muss. Dies ist durch die Typisierung von C# jedoch sehr umständlich und zeitintensiv. Deshalb haben wir uns an diesem Punkt entschlossen, unseren Implementierungsversuch in C# abzubrechen.

Implementierung anhand einer sprachintegrierten Datenbank

Technischer Hintergrund. Unser zweiter Implementierungsversuch fußt auf einer von uns entwickelten sprachintegrierten Datenbank für die Programmiersprache Python, die in Kapitel II bis IV beschrieben wird. Diese Datenbank erlaubt es, die Datenobjekte der Laufzeitumgebung in einem für analytische Anfragen optimiertem Speicherlayout abzulegen und die Anfragen somit direkt auf diesen auszuführen. Eine separate objekt-relationale Abbildung ist deshalb nicht notwendig.

Für die Ausführung der Anfragen werden wir die eingebette Anfragesprache (*EQL, Embedded Query Language*) dieser Datenbank nutzen. Diese deklarative Anfragesprache lehnt in ihrem Aufbau an *LINQ* (siehe Abschnitt VII.5.2) an. Jedoch werden die Operationen nicht zu SQL-Anfragen umgewandelt, sondern werden direkt auf den Python-Datenobjekten angewandt. Hierfür nutzt die EQL für die Optimierung der Anfragebearbeitung eine Methode, die die Operationen auf den Objekten durch Emulation aufzeichnet und daraus einen Anfrageplan erstellt. Da eine Übersetzung zu SQL nicht nötig ist, ist die Nutzung von Introspektion auch innerhalb der Anfragen möglich, jedoch unterliegt die EQL durch die Emulation der Anfragen auch Einschränkungen, auf die wir später eingehen. Quelltext 35 zeigt die Umsetzung derselben Beispielanfrage mithilfe der EQL, wie Quelltext 31 für LINQ.

```
result = list(Bestellungsposten.all_instances \
  .where(lambda posten:
    (posten.bestellung.bestelldatum.jahr >= 2009) &
    (posten.bestellung.bestelldatum.jahr <= 2009)) \
  .group_by(
    Jahr = \
        lambda posten: posten.bestellung.bestelldatum.jahr,
    Produktgruppe = \
        lambda posten: posten.produkt.produktgruppe
    ) \
  .select(
    Jahr = lambda group: group.key().Jahr,
    Produktgruppe = \
        lambda group: group.key().Produktgruppe,
    Gesamtpreis = lambda group: group.sum(
                    lambda posten: posten.gesamtpreis())
```

```
17      ) \
18    .order_by(lambda result: result.Jahr) \
19    .then_by(lambda result: result.Produktgruppe))
```

Quelltext 35: *Beispielanfrage in der EQL.*

```
1  def method_predicate(attr, stack):
2    return lambda object: getattr(stack(object), attr)()
3
4  def attr_predicate(attr, stack):
5    return lambda object: getattr(stack(object), attr)
6
7  def property_path_predicate(base_entity, property_path):
8    predicate = lambda object: object
9    for property in split_path(base_entity, property_path):
10     if property.is_method():
11       predicate = \
12           method_predicate(property.source, predicate)
13     else:
14       predicate = \
15           attr_predicate(property.source, predicate)
16     base_entity = property.type
17   return predicate
```

Quelltext 36: *Auflösung des Attributpfads in Python (vereinfacht).*

Aufbau der Implementierung. Da der Aufbau der EQL-Anfragen ähnlich dem der LINQ-Anfragen ist, ähnelt auch die Implementierung des Anfrageprozessors in Python der oben beschriebenen C#-Lösung. Aus dem Anfrageobjekt wird eine EQL-Anfrage erstellt. Dabei werden die Filter und Dimensionen auf die gleiche Art wie in der C#-Lösung auf die `where`-, `select`-, `group_by`- und order_by-Anweisungen abgebildet.

In Quelltext 36 ist die Auflösung des Attributpfades in Python dargestellt. Die Methode `property_path_predicate` generiert einen Lambda-Ausdruck, der in den `select`-/`group_by`-Anweisungen benutzt werden kann. Wir benutzen hierfür das Konzept der Datenmodelleigenschaften (`properties`), das wir auch zur Erstellung der Nutzerschnittstelle verwenden (siehe Abschnitt VII.5.3). Im Quelltext zerteilt die Methode `split_path` den Attributpfad und gibt die Metaobjekte zurück, die die einzelnen Modelleigenschaften im Attributpfad beschreiben. Da diese Eigenschaften Methoden oder Attribute der Datenobjekte sein können, muss dies auch in der Attributpfadauflösung berücksichtigt werden. Bei der Generierung des Lambda-Ausdruckes werden die einzelnen Lambda-Ausdrücke für die Attribut- oder Methodenaufrufe ineinander geschachtelt.

Da die Parameter für die Anweisungen sehr leicht aus Python-Dictionaries entrollt werden können, ist die dynamische Konfiguration der EQL-Anfrage leicht umzusetzen, wie in Quelltext 37 skizziert.

```
dict = {}
for dimension in grouping_dimensions:
  dict[dimension.name] = property_path_predicate(base_entity,
                         dimension.property_path)
query.group_by(**dict)
```

Quelltext 37: *Dynamische Gruppierung über beliebig viele Dimensionen.*

Evaluierung der Implementierungsmethode nach Schwierigkeitsgrad. Durch die Nutzung der sprachintegrierten Datenbank in Python haben wir die Probleme, die aus der objektrelationalen Abbildung und aus der strikten Typisierung in dem vorherigen ASP.NET-Ansatz entstanden, vermieden. Jedoch ist uns bei der Nutzung der EQL aufgefallen, dass durch die Emulation der Anfragen, die für die Erstellung des Anfrageplans durchgeführt wird, einige Einschränkungen im Umgang mit der EQL hervorgerufen werden.

So konnte beispielsweise unsere anfängliche Idee für die Umsetzung der Attributpfadauflösung aufgrund dieser Einschränkungen nicht umgesetzt werden. Diese beinhaltete, eine instanzseitige Methode im Lambda-Ausdruck aufzurufen, die die Fallunterscheidung trifft, ob die Modelleigenschaft eine Methode oder ein Attribut ist. Sie ist in Quelltext 38 skizziert. Der Vorteil dieser Variante ist, dass ein Ineinanderschachteln von Lambda-Ausdrücken nicht nötig ist.

Für die Emulation des Lambda-Ausdruckes wird diesem statt einem tatsächlichen Objekt der Modellklasse ein Proxyobjekt übergeben. Die Operationen auf dem Proxyobjekt werden aufgezeichnet. Dafür wird bei jedem Attributaufruf auf dem Proxyobjekt wieder ein Proxyobjekt zurückgegeben (statt einem tatsächlichen Wert). Für Näheres zu dieser Emulation sei auf die Arbeit von Schreiber in Kapitel IV verwiesen. Sobald jedoch innerhalb des Lambda-Ausdrucks eine Entscheidung anhand des Wertes eines Attributs oder einer Eigenschaft der Instanz getroffen werden soll, kann diese Entscheidung nicht gefällt werden, weil der tatsächliche Wert während der Emulation nicht verfügbar ist. Außerdem ist die Klasse der übergebenen Proxyobjekte nicht die Modellklasse selbst. Deshalb kann auch die Klasse der übergebenen Instanz nicht bestimmt werden.

In unserem Beispiel versucht die Methode `get_property_value` eine Fallunterscheidung anhand einer Instanzeigenschaft zu treffen: Sie ermittelt den Wert des hinter dem übergebenen `property_name` stehenden Attributs. Falls

dieser Wert eine Methode ist, gibt sie die Rückgabe dieser Methode zurück, andernfalls wird der ermittelte Wert zurückgegeben (siehe Quelltext 39). Da der Aufruf der getattr-Methode jedoch während der Emulation nur ein Proxyobjekt zurückgibt, egal ob dahinter eine Methode oder ein Attribut steht, kann diese Unterscheidung nicht funktionieren. Alternativ hierzu könnte man sich vorstellen, diese Überprüfung auf der Klasse der übergebenen Instanz auszuführen, jedoch ist diese wie beschrieben nicht erreichbar. Aus diesen Gründen ist dieser intuitiv korrekte Ansatz so nicht umsetzbar, weshalb wir uns für die oben beschriebene Variante entschieden haben und diese Fallunterscheidung außerhalb der Generierung der Lambda-Ausdrücke vornehmen.

```
def property_path_predicate(property_path):
  return lambda instance: \
    property_path_value(instance, property_path)

def property_path_value(instance, property_path):
  for property_name in property_path.split('.'):
    instance = instance.get_property_value(property_name)
  return instance
```

Quelltext 38: *Attributpfadauflösung durch Instanzmethode.*

```
def get_property_value(instance, property_name):
  source = property_source(property_name)
  value = getattr(instance, property_name)
  if iscallable(value):
    return value(self)
  else:
    return value
```

Quelltext 39: *Instanzmethode mit Fallunterscheidung.*

Es muss also festgehalten werden, dass durch diese Emulation bei komplizierten Anwendungsfällen (gerade solche, die Introspektion erfordern) für den Programmierer ein Umdenken von der gewöhnlichen Programmausführung nötig ist, um funktionsfähige Lambda-Ausdrücke zu schreiben. Er muss sich bei der Formulierung der Lambda-Ausdrücke der Emulation dieser bewusst sein.

Nach unseren Eindrücken ist die Implementierung des Anfrageprozessors mithilfe der EQL dennoch einfacher umzusetzen als mithilfe von LINQ, da die Lambda-Ausdrücke nicht durch Spezifikation des Ausdrucksbaums und die Anfrage nicht mittels Introspektion erstellt werden müssen.

Evaluierung der Implementierungsmethode nach Geschwindigkeit. Die Motivation für die beschriebene Emulation der EQL-Anfragen ist die Optimierung der Ausführungsgeschwindigkeit der Datenbankanfragen. Für einen Geschwindigkeitsvergleich haben wir eine einfache Implementierung des Anfrageprozessors in purem Python ohne die Nutzung der EQL erstellt, um zu ermitteln, ob die EQL diesem Anspruch an die Ausführungsgeschwindigkeit gerecht wird. Hierbei muss gesagt werden, dass wir in dieser Implementierung wie auch bei der Implementierung mit der EQL keine besondere Rücksicht auf Geschwindigkeitsoptimierungen genommen haben. Die resultierenden Werte sind also darauf bezogen, welche Ausführungsgeschwindigkeiten mit geringem Implementierungsaufwand erreicht werden können.

Als vorteilhaft der puren Python-Implementierung empfanden wir, dass wir in der Umsetzung uneingeschränkt waren. Wir konnten also unter anderem Unterscheidungen anhand der Instanzeigenschaften der Datenobjekte treffen.

Die Ergebnisse der Geschwindigkeitstests[33] sind jedoch eindeutig. Diese Tests basieren auf dem in Abbildung 39 beschriebenen Datenmodell und der Anfragespezifikationen aus Abbildung 40. Das Datenmodell umfasste für unsere Tests 5000 Bestellungen mit jeweils einem bis fünf Bestellungsposten (gleichverteilt). Die durch die EQL optimierten Anfragen erreichten dabei Ausführungszeiten von 41,6ms (+/- 0,2ms), während die in purem Python implementierte Lösung 7490 ms (+/- 15ms) benötigte. Das heißt, dass die Optimierung der EQL-Anfragen in diesem Fall eine etwa 180fach schnellere Ausführung ermöglichte.

In Kapitel IV haben wir angemerkt, dass durch die Laufzeitoptimierungen der Pypy-Laufzeitumgebung unseres Prototypen theoretisch eine ähnlich hohe Geschwindigkeit durch die reine Python-Implementierung der Anfragen erreicht werden können müsste. Da unser Versuch jedoch einen großen Unterschied zwischen den Ausführungszeiten der reinen Python-Implementierung der Anfragen und der EQL-Anfragen aufzeigt, müssen wir feststellen, dass dies ohne weitere manuelle Optimierungen an der reinen Python-Implementierung in der Praxis nicht möglich ist.

Unser Fazit dieser Untersuchung ist, dass der Geschwindigkeitsvorteil, der durch die Nutzung des optimierten EQL-Anfrageplans entsteht, das oben beschriebene nötige Umdenken für den Programmierer rechtfertigt, da für eine ähnlich hohe Geschwindigkeit in einer reinen Python-Umsetzung ein sehr viel höherer manueller Optimierungsaufwand nötig wäre und dieser Optimierungsaufwand den durch das Umdenken entstehenden Aufwand übersteigt.

[33] Testsystem: Workstation, Windows 7 x64, Intel Core i5 4x2.6 GHz, 6 GB RAM

VII.5.3 Nutzerschnittstelle

Technischer Hintergrund. Um eine schnelle Implementierung der Nutzerschnittstelle zu ermöglichen, haben wir das in Kapitel V und VI beschriebene Rahmenwerk genutzt. Dieses erlaubt die automatische Erzeugung von CRUD-Oberflächen[34] für Modellentitäten, indem diesen durch zusätzliche Annotationen in den Modellklassen Informationen zur Darstellungsweise hinzugefügt werden.

Aktuelle Lösung. Durch die Anreicherung des Anfragemodells mit Darstellungsinformationen konnte auf einfache Weise das Anlegen und Editieren von statistischen Auswertungen erreicht werden. Zusätzlich wurde ein neues Widget für das Editieren des Attributpfades angelegt, sodass dieser über Drop-Down-Boxen ausgewählt werden kann. Hierbei haben wir die Eigenschaften (`properties`), die in den Datenmodellen annotiert wurden, ausnutzen können, um die zur Verfügung stehenden Attribute für die einzelnen Teile des Attributpfads übersetzt anzuzeigen. Auf diese Weise konnten wir erreichen, dass der Nutzer für die Spezifikation des Attributpfads keinen Text eintippen muss und aus allen verfügbaren Attributen auswählen kann, wodurch die Oberfläche also für den Nutzer einfach zu bedienen und die Validität der Eingaben gewährleistet ist.

Abb. 41: *Übersichtsseite mit Auflistung aller angelegten Auswertungen. Durch die Buttons auf der rechten Seite der Tabelle kann das Ergebnis der Auswertung abgerufen und grafisch angezeigt werden, sowie die Anfrage bearbeitet werden.*

VII.6 Auswertung

In diesem Abschnitt werden wir überprüfen, inwieweit die Anforderungen aus Abschnitt VII.4 durch unser Anfragemodell und dessen prototypenhafte Implementierung in Python (Abschnitt VII.5.2) gelöst wurden.

[34] *Create-*, *Read-*, *Update-* und *Delete*-Funktionalitäten

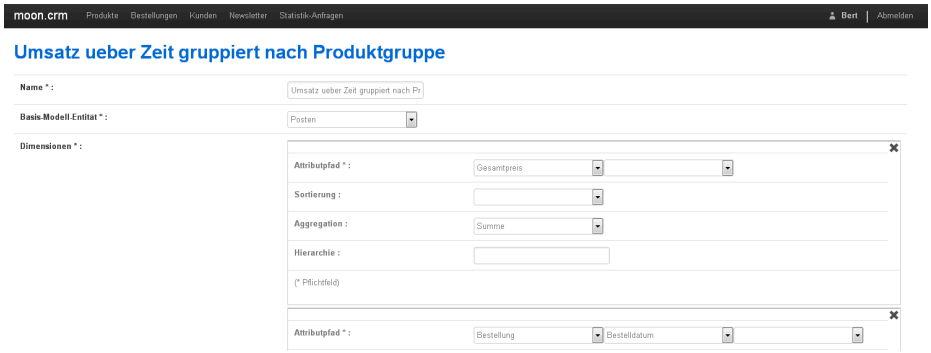


Abb. 42: *Bearbeitungsansicht einer Anfrage. Die Dimensionen können mit den Kreuzen in der oberen rechten Ecke entfernt werden und mit einem weiteren Button (außerhalb des Bildes) können zusätzliche Dimensionen hinzugefügt werden. Hier sind auch die Widgets zur Bearbeitung des Attributpfads zu sehen. Das Anlegen der Filter funktioniert auf ähnliche Weise wie das der Dimensionen.*

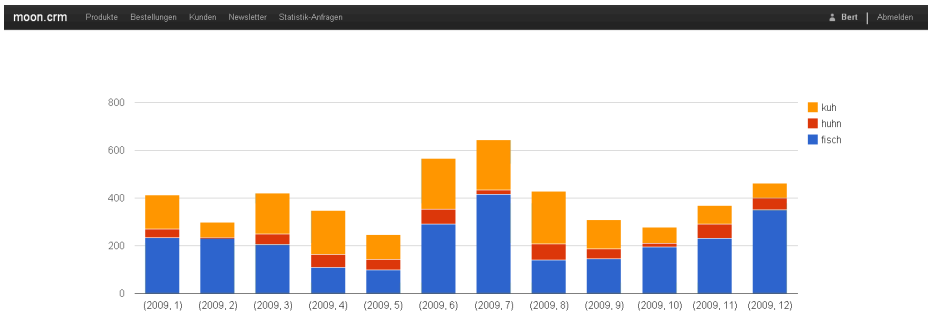


Abb. 43: *Grafische Ansicht des Ergebnisses der Auswertung. Hier zu sehen ist der Umsatz über die Zeit (hierarchisch unterteilt nach Jahr und Monat) pro Produktgruppe.*

VII.6.1 Anfragemodell

Verständlichkeit. Wir sind bei der Formulierung des Anfragemodells von den für den Endnutzer relevanten Eigenschaften einer statistischen Auswertung, also denen eines Diagramms, ausgegangen. Auf diesem Weg konnten wir eine direkte Beschreibungsweise für derartige Auswertungen erreichen, die für den Nutzer greifbarer ist als die Syntax und Semantik der in Abschnitt VII.2 betrachteten Anfragesprachen. Zusätzlich basiert unser Anfragemodell auf einem objektorientierten Datenmodell. Die Bestimmung der Metriken und Gruppierungen folgt dem intuitiven Prinzip, über die Beziehungen und Eigenschaften dieser Modellentitäten zu navigieren. Da der Endnutzer ebenfalls in Entitäten und Beziehungen denkt, beruht dieses Datenmodell auf dem konzeptuellen

Verständnis des Endnutzers. Aus diesem Grund ist es auch für diesen verständlich.

Flexibilität und Mächtigkeit. Das Anfragemodell erlaubt die Beschreibung von statistischen Auswertungen mit unbegrenzt vielen Aggregations- und Gruppierungsdimensionen. Hierbei wird die freie Wahl der Aggregationsarten und Metriken sowie Gruppierungsattributen unterstützt. Die in Abschnitt VII.3 beschriebenen Einschränkungen wurden dabei umgangen. So kann beispielsweise die dort in Abbildung 37(b) dargestellte Auswertung mithilfe des Anfragemodells spezifiziert werden, wie in Abbildung 40 illustriert. Eine zusätzliche Konfiguration von OLAP-Würfeln, wie dies bei MDX (siehe Abschnitt VII.2.2) der Fall war, ist ebenfalls nicht nötig. Auch ist eine Anpassung von bereits spezifizierten Anfragen sehr leicht möglich. Falls beispielsweise die Gruppierung der Anfrage in Abbildung 37(b) statt nach der Produktgruppe nach dem Kunden der Bestellung durchgeführt werden soll, muss nur der Attributpfad dieser Gruppierungsdimension (z.B. auf *Bestellung.Kunde.Name*) geändert werden. Die Definition von diskreten und kontinuierlichen Filtern wird auf eine ähnlich intuitive Art und Weise unterstützt.

Wir können also zusammenfassen, dass die Anforderungen in Punkto Verständlichkeit und Flexibilität des Anfragemodells erreicht wurden. Die Mächtigkeit unseres Anfragemodells konnten wir leider im Rahmen des Bachelorprojektes nicht genau analysieren. Jedoch können wir sagen, dass wir alle uns bisher denkbaren Metriken und Auswertungen mit dem Anfragemodell definieren konnten. Hierzu zählen unter anderem auch die in der Beispielanwendung des *BusinessObjects Explorers* (Abbildung 36) formulierbaren Auswertungen. Falls dennoch die Unterstützung von Auswertungen gewünscht ist, die unser Modell in der beschriebenen Version nicht erfassen kann, sollte diese Unterstützung durch einfache Erweiterungen des Modells hinzugefügt werden können. Ein Beispiel für eine solche Erweiterung ist die Unterstützung von Filterkompositionen durch logische Operatoren, wie in Abschnitt VII.5.1 beschrieben. Um eine genauere Analyse der Mächtigkeit vorzunehmen, könnte beispielsweise eine systematische Reduktion der Konzepte einer Referenz-Anfragesprache, von der eine bestimmte Mächtigkeit bekannt ist, auf die Konzepte unseres Anfragemodells vorgenommen werden.

VII.6.2 Verständlichkeit der Nutzerschnittstelle

In der aktuellen Form bildet die Bedienoberfläche das Anfragemodell strukturell so ab, wie dies durch die Objektkomposition des Anfragemodells vorgeschlagen ist. Durch die Unterstützung des Nutzers bei der Definition der Attributpfade ist die aktuelle Lösung nach einer Einführung des Nutzers aus unserer Sicht tauglich zur Verwendung durch diesen. Diese Einführung ist nötig, da

die Bedienoberfläche in der aktuellen Form nicht selbsterklärend auftritt. Der Nutzer benötigt für das Verständnis der Nutzerschnittstelle Hintergrundwissen über das Anfragemodell. Wir stellen deshalb im Folgenden noch weitere Ideen vor, wie diese Oberfläche alternativ gestaltet werden könnte. In dem uns vorgegebenen Zeitrahmen war die technische Realisierung dieser Ideen jedoch nicht möglich.

Um das Bearbeiten der statistischen Auswertungen einfacher zu gestalten, schlagen wir eine diagrammbasierte Oberfläche vor, die die Formularseiten der aktuellen Nutzerschnittstelle ersetzt. Ein Entwurf dieser Idee ist in Abbildung 44 abgebildet. Die Grundlage der Oberfläche bildet ein Säulendiagramm, an dessen Achsen die Konfiguration der statistischen Auswertung bzw. des Anfrageobjekts vorgenommen wird. Die linke vertikale Achse dient hierbei zur Spezifikation der Aggregationsdimensionen mit Attributpfaden und Aggregationsarten. Auf der unteren waagerechten Achse kann die primäre Gruppierungsdimension spezifiziert werden. Zu dieser Dimension kann eine Hierarchie oder Sortierung hinzugefügt werden, die leicht eingerückt unter dem Attributpfad erscheint. Auf die gleiche Art können Filter hinzugefügt werden, die relativ zum ausgewählten Attribut der Dimension angewendet werden. (Hierfür muss eine entsprechende Umwandlung in die Filterspezifikation des Anfragemodells vorgenommen werden.) Auf dieselbe Art können zusätzliche Gruppierungsdimensionen an der rechten vertikalen Achse definiert werden, die die Unterteilungen der Säulen beschreiben. Zusätzlich können unter der primären Gruppierungsdimension Filter (und andere Optionen der Auswertung) angelegt werden, die sich nicht auf die Gruppierungsdimensionen beziehen. Das Ergebnis der Anfrage kann in einer Live-Vorschau im Diagramm angezeigt werden. Mittels dieser Oberfläche wird dem Nutzer eine intuitivere Bearbeitung der statistischen Auswertungen erlaubt, da er die Dimensionen an den Achsen spezifiziert, an denen sie auch später erscheinen würden. Gleichzeitig erhält er Feedback zum Ergebnis der Anfrage. Natürlich ist auch eine andere Diagrammart für diese Oberfläche denkbar, genauso wie es möglich wäre, das Diagramm in der säulendiagrammbasierten Oberfläche zu bearbeiten und das Ergebnis später in einer anderen Diagrammart anzuzeigen.

Diese Idee für eine Oberfläche zum Editieren von Anfrageobjekten kann kombiniert werden mit einer menübasierten Attributpfadauswahl, wie sie in Abbildung 45 skizziert ist. Hierdurch entfällt der Platzbedarf für die Auswahlboxen der aktuellen Lösungen. Außerdem kann die Auswahl des Attributs schneller durchgeführt werden, da nicht immer von Auswahlbox zu Auswahlbox geklickt werden muss.

Zusätzlich könnte dem Nutzer der Einstieg erleichtert werden, indem ihm einige Beispielauswertungen oder Schablonen für Auswertungen zur Verfügung gestellt werden, die er kopieren und als Grundlage für ähnliche Auswertungen nutzen kann.

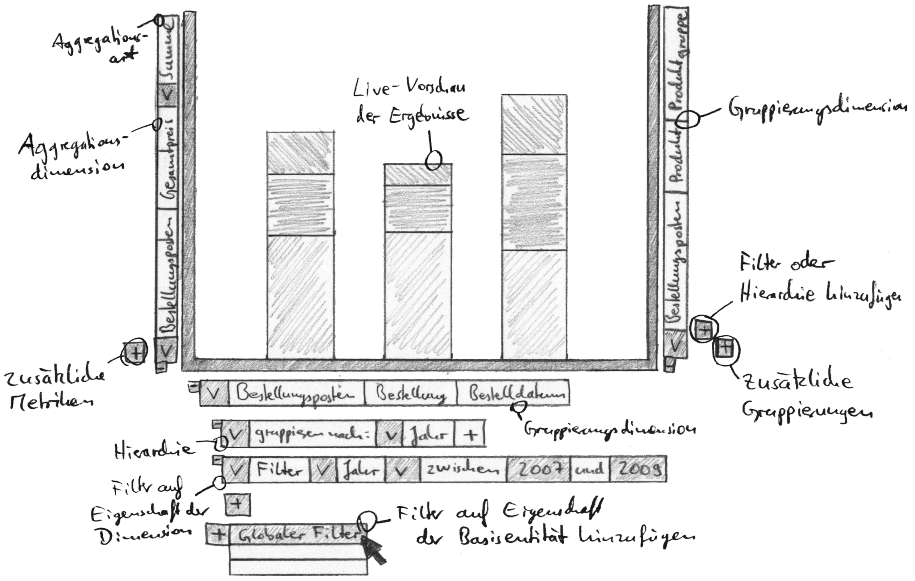


Abb. 44: *Ideenskizze: Säulendiagrammbasiertes Editieren eines Anfrageobjekts.*

Nach der Umsetzung dieser Ideen wäre der nächste Schritt, eine Nutzerstudie durchzuführen und die Verständlichkeit anhand bestimmter Faktoren wie Einarbeitungszeit, Fehlerrate und persönlichem Schwierigkeitsempfinden zu ermitteln.

VII.6.3 Implementierungstechnik

Die Wahl der sprachintegrierten Hauptspeicherdatenbank und des Rahmenwerks zur dynamischen Erzeugung der Bedienoberfläche hat sich bewährt. Durch die durchgängige Nutzung von Objektorientierung in der Formulierung der Datenbankanfragen auf der einen Seite und durch die simple Annotierung des Anfragemodells mit Darstellungsinformationen auf der anderen konnten wir das Grundgerüst des vorliegenden Prototypen innerhalb von zwei Manntagen erstellen. Im Vergleich zu dem ersten fehlgeschlagenen Ansatz basierend auf ASP.NET (siehe Abschnitt VII.5.2) ist dies ein Unterschied einer Größenordnung, denn dort haben wir etwa zwei Mannwochen auf die Erstellung des Grundgerüsts verwendet, jedoch keinen funktionstüchtigen Prototypen erstellen können.

Die Hauptspeicherdatenbank ist auch der Grund für die für einen Prototypen erstaunlich hohe Ausführungsgeschwindigkeit der Anfragen (siehe Abschnitt VII.5.2). Durch die Laufzeitoptimierungen bei der Ausführung der EQL-Anfragen ist auch der Geschwindigkeitsunterschied zwischen einer dynamisch

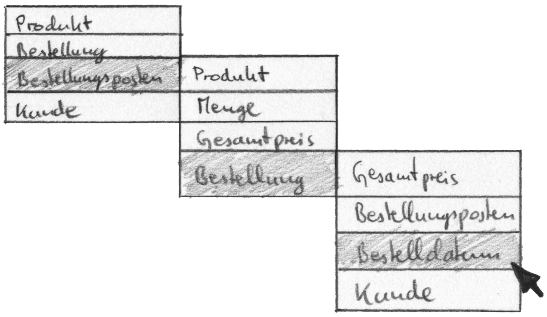


Abb. 45: *Ideenskizze: Attributpfadauswahl über Menüstruktur.*

und einer statisch erstellten Anfrage nur sehr gering. Bei dem durchgeführten Geschwindigkeitstest wurde auch dieser Unterschied ermittelt. Eine dynamisch erstellte Anfrage benötigte im Vergleich zu einer statisch auf eine Auswertung zugeschnittene Anfrage etwa 6% mehr Zeit (41,6 ms zu 39,3 ms bei +/- 0,2 ms).

VII.7 Fazit

Wir haben uns in dieser Arbeit angesehen, welche Probleme die Verständlichkeit, Mächtigkeit und Flexibilität aktueller Lösungen zur Erstellung von statistischen Auswertungen einschränken. Um diese Probleme zu lösen, haben wir ein neues Anfragemodell zur Beschreibung von statistischen Auswertungen vorgeschlagen, welches auf einem objektorientierten Modell der Daten basiert und somit auf dem für den Endnutzer intuitiven Verständnis des Datenmodells aufbaut.

Obwohl sich unsere Ideen für eine intuitiv für den Endanwender verständliche Umsetzung der Nutzerschnittstelle noch beweisen müssen, glauben wir, durch unsere prototypenhafte Umsetzung des Anfragemodells gezeigt zu haben, dass eine flexible und mächtige, aber trotzdem für den Endnutzer verständliche Lösung zur Spezifikation statistischer Auswertungen realisierbar ist.

Teil VIII

Literaturverzeichnis

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Series. Oxford University Press, 1977.

[2] *ASP.NET MVC*. `http://www.asp.net/mvc`. Stand: 10.06.2012.

[3] C.F. Bolz et al. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM. 2009, 18–25.

[4] Michael J. Carey and Waleed A. Muhanna. "The Performance of Multiversion Concurrency Control Algorithms". In: *ACM Transactions on Computer Systems* 4 (1986), 338–378.

[5] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A structured English query language". In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. SIGFIDET '74. Ann Arbor, Michigan: ACM, 1974, 249–264.

[6] Hock Chuan Chan, Kwok Kee Wei, and Keng Leng Siau. "User-database interface: the effect of abstraction levels on query performance*". In: *MIS Q.* 17.4 (Dec. 1993), 441–464.

[7] George Copeland and David Maier. "Making smalltalk a database system". In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. SIGMOD '84. Boston, Massachusetts: ACM, 1984, 316–325.

[8] J. Steve Davis. "Usability of SQL and menus for database query". In: *International Journal of Man-Machine Studies* 30.4 (1989), 447–455.

[9] Carl Dubler and Colin Wilcox. *Just What Are Cubes Anyway? (A Painless Introduction to OLAP Technology)*. `http://msdn.microsoft.com/en-us/library/aa140038.aspx`. Stand: 13.06.2012. Apr. 2002.

[10] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[11] Maciej Fijalkowski. *PyPy is faster than C, again: string formatting*. `http://morepypy.blogspot.de/2011/08/pypy-is-faster-than-c-again-string.html`. Stand: 10.06.2012.

[12] J. Fulton. "Introduction to the Zope Object Database". In: *Proceedings of the 8th International Python Conference*. 2000.

[13] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[14] M. Hellman. "A Cryptanalytic Time-Memory Tradeoff". In: *IEEE Transactions of Information Theory* (1980).

[15] *IEEE Standard 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[16] H. T. Kung and John T. Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems* 6 (1981), 213–226.

[17] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. "A survey on usage of SQL". In: *SIGMOD Rec.* 22.4 (Dec. 1993), 60–65.

[18] J. Martin. *Managing the data-base environment*. The James Martin books on computer systems and telecommunications. Prentice-Hall, 1983.

[19] B. Meyer. *Object-oriented software construction*. Prentice-Hall International Series in Computer Science. Prentice Hall PTR, 1997.

[20] Carl Nolan. "Manipulate and Query OLAP Data Using ADOMD and Multidimensional Expressions." In: *Microsoft Systems Journal* 63 (1999), 51–59.

[21] H. Plattner and A. Zeier. *In-Memory Data Management*. 2011.

[22] *Pylons Webframework*. `http://www.pylonsproject.org/`. Stand: 15.06.2012.

[23] *PyPy*. `http://pypy.org/`. Stand: 10.06.2012.

[24] *Python New-Style Classes*. `http://www.python.org/doc/newstyle/`. Stand: 13.06.2012.

[25] Jun Rao and Kenneth A. Ross. "Making B+-trees cache conscious in main memory". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. Dallas, Texas, United States: ACM, 2000, 475–486.

[26] Trygve Reenskaug. *Models-Views-Controllers*. Tech. rep. Xerox PARC, 1979.

[27] Trygve Reenskaug. *Thing-Model-View-Editor – an Example from a planning system*. Tech. rep. Xerox PARC, 1979.

[28] *SAP High-Performance Analytic Application 1.0*. `http://scn.sap.com/docs/DOC-13231`. Stand: 07.07.2012.

[29] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[30] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[31] Collin Winter. *PEP 3129: Class Decorators*. `http://www.python.org/dev/peps/pep-3129/`. Stand: 13.06.2012.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
70	978-3-86956-230-8	**HPI Future SOC Lab - Proceedings 2011**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Stromann, Ulrike Seibold, Doc D'Errico
69	978-3-86956-229-2	**Akzeptanz und Nutzerfreundlichkeit der AusweisApp: Eine qualitative Untersuchung**	Susanne Asheuer, Joy Belgassem, Wiete Eichorn, Rio Leipold, Lucas Licht, Christoph Meinel, Anne Schanz, Maxim Schnjakin
68	978-3-86956-225-4	**Fünfter Deutscher IPv6 Gipfel 2012**	Christoph Meinel, Harald Sack (Hrsg.)
67	978-3-86956-228-5	**Cache Conscious Column Organization in In-Memory Column Stores**	David Schalb, Jens Krüger, Hasso Plattner
66	978-3-86956-227-8	**Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software**	Thomas Vogel, Holger Giese
65	978-3-86956-226-1	**Scalable Compatibility for Embedded Real-Time components via Language Progressive Timed Automata**	Stefan Neumann, Holger Giese
64	978-3-86956-217-9	**Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants**	Basil Becker, Holger Giese
63	978-3-86956-204-9	**Theories and Intricacies of Information Security Problems**	Anne V. D. M. Kayem, Christoph Meinel (Eds.)
62	978-3-86956-212-4	**Covering or Complete? Discovering Conditional Inclusion Dependencies**	Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, Felix Naumann
61	978-3-86956-194-3	**Vierter Deutscher IPv6 Gipfel 2011**	Christoph Meinel, Harald Sack (Hrsg.)
60	978-3-86956-201-8	**Understanding Cryptic Schemata in Large Extract-Transform-Load Systems**	Alexander Albrecht, Felix Naumann
59	978-3-86956-193-6	**The JCop Language Specification**	Malte Appeltauer, Robert Hirschfeld
58	978-3-86956-192-9	**MDE Settings in SAP: A Descriptive Field Study**	Regina Hebig, Holger Giese
57	978-3-86956-191-2	**Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Stefan Neumann, Sebastian Wätzoldt
56	978-3-86956-171-4	**Quantitative Modeling and Analysis of Service-Oriented Real-Time Systems using Interval Probabilistic Timed Automata**	Christian Krause, Holger Giese
55	978-3-86956-169-1	**Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium**	Peter Tröger, Andreas Polze (Eds.)
54	978-3-86956-158-5	**An Abstraction for Version Control Systems**	Matthias Kleine, Robert Hirschfeld, Gilad Bracha

ISBN 978-3-86956-231-5
ISSN 1613-5652