Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Qualitative und quantitative Evaluation von Varianten zur performanten Verwaltung von Prozessdaten in einem Hochleistungs-PMS

Diplomarbeit

vorgelegt von
Bastian Philipp Glöckle
am 23. Dezember 2009

1. Gutachter: Prof. Dr. Manfred Reichert
2. Gutachter: PD Dr. Stefanie Rinderle-Ma
Betreuer: Dipl.-Inf. (U) Ulrich Kreher

Kurzfassung

Der stetig anwachsende Grad der Globalisierung ließ Prozess-Management-Systeme (PMS) in den letzten Jahren stark an Bedeutung gewinnen. Diese bilden Prozesse ab, welche dann computergestützt durchgeführt werden können. Hierzu müssen die Prozesse dem PMS bekannt sein. Ein Modell, welches dies ermöglicht ist das ADEPT2-Metamodell [Rei00]. Dieses muss in einer geeigneten Speicherrepräsentation dem PMS zur Verfügung gestellt werden. Ziel dieser Arbeit ist, durch qualitative und quantitative Untersuchungen eine optimale Repräsentation zu ermitteln.

Das ADEPT2-Metamodell gliedert die Daten in Schema- und Instanzdaten. Schemata bilden hierbei den Bauplan eines Prozesses, Instanzen dessen Ausführung. Neben der Modellierung und der Ausführung sieht das Metamodell auch die flexible Änderung von Prozessen vor. Hierbei sind Änderungen sowohl auf Instanz- als auch auf Schemaebene möglich. Dies muss als zentrales Merkmal in die Repräsentation der Daten einfließen.

In einem ersten Schritt werden verschiedene Konzepte vorgestellt. Dies umfasst die Partitionierung von Schemata und die konzeptionelle Umsetzung dieser. Es stellt sich heraus, dass neben verschiedenen Repräsentationen für Instanzen auch unterschiedliche Repräsentationen für Schemata existieren. Zusätzlich wird die grobe Clusterung vorgestellt, welche eine Einschätzung des Laufzeitzustands von Instanzen ermöglicht. Hierdurch lässt sich die Laufzeit der so genannten Schemaevolution [Rin04] verbessern.

Diese Konzepte werden anschließend umgesetzt und implementiert. Hierbei werden sowohl Repräsentationen im Primär- als auch im Sekundärspeicher vorgestellt. Es kommen verschiedene Sekundärspeichervarianten zum Einsatz, wie zum Beispiel eine Speicherung in XML-Dateien oder ausgewählten relationalen Datenbanksystemen. Mithilfe dieser Implementierung werden auf Basis von Anwendungsszenarien Messungen der verschiedenen Repräsentationsvarianten durchgeführt. Hierbei werden Laufzeiten von Anfragen an die Primärpseicherrepräsentation untersucht, ebenso wie das Ein- und Auslagern der verschiedenen Sekundärspeicherrepräsentationen und Anfragen an Kollektionen. Die Messergebnisse werden anschließend interpretiert und verglichen. Ein abschließendes Fazit stellt als Folgerung aller Untersuchungen die optimale Repräsentation der Daten vor.

Die Vorstellung alternativer Konzepte zur Modellierung von Prozessen, welche anstelle des ADEPT2-Metamodells verwendet werden können, rundet diese Arbeit ab.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Aufgabenstellung . . . 1
1.3 Aufbau . . . 2

2 Grundlagen **3**
2.1 Das ADEPT2-Metamodell . . . 3
2.1.1 Grundlegende Begrifflichkeiten . . . 3
2.1.2 Schemata . . . 4
2.1.3 Instanzen . . . 6
2.1.4 Abstraktion . . . 7
2.1.5 Zusammenfassung . . . 8
2.2 Repräsentation . . . 8
2.2.1 Schemata . . . 8
2.2.2 Instanzen . . . 9
2.2.3 Instanzbasierte Änderungen . . . 10
2.2.4 Zusammenfassung . . . 13
2.3 ADEPT2-Architektur . . . 13
2.4 Anwendungsszenarien . . . 15
2.4.1 Weiterschalten . . . 15
2.4.2 Schemaevolution . . . 15
2.4.3 Anfragen auf Kollektionen . . . 18
2.4.4 Zusammenfassung . . . 19
2.5 Allgemeine Speicherbetrachtungen . . . 19
2.6 Zusammenfassung . . . 20

3 Konzepte **21**
3.1 Ansätze zur Partitionierung von Schemata . . . 21
3.1.1 Partitionierung anhand Bearbeiterzuordnungen . . . 23
3.1.2 Blockstruktur . . . 24
3.1.3 Fazit . . . 25
3.2 Schemata . . . 25
3.2.1 Speicherbetrachtungen . . . 25
3.2.2 Relation zwischen Knoten . . . 26
3.2.3 Blockrepräsentation . . . 28
3.3 Grobe Clusterung . . . 33
3.3.1 Schemaevolution . . . 34
3.3.2 Fazit . . . 40
3.4 Instanzbasierte Änderungen . . . 41
3.4.1 Erweiterung des Deltaschicht-Modells . . . 41
3.4.2 Speicherbetrachtungen . . . 42

3.4.3 Repräsentation . . . 43
3.4.4 Clusterung und Schemaevolution . . . 46
3.5 Fazit . . . 48

4 Umsetzung 49
4.1 Einführung . . . 49
4.1.1 Relevante Aspekte von Java™ . . . 49
4.1.2 Architektur . . . 50
4.2 Primärspeicher . . . 52
4.2.1 Standard-Datenstrukturen von Java™ . . . 52
4.2.2 Schemata . . . 53
4.2.3 Deltaschicht . . . 56
4.2.4 Instanzen . . . 57
4.2.5 Cluster . . . 58
4.2.6 Zusammenfassung . . . 59
4.3 Sekundärspeicher . . . 59
4.3.1 Realisierungsalternativen . . . 60
4.3.2 Schnittstellen zwischen Primär- und Sekundärspeicher . . . 60
4.3.3 Schemata . . . 61
4.3.4 Instanzen . . . 68
4.3.5 Zwischenspeicher . . . 72
4.3.6 Flüchtiger Speicher . . . 73
4.3.7 Zusammenfassung . . . 73
4.4 Anfragen auf Kollektionen . . . 73
4.4.1 Instanzen eines Schemas . . . 74
4.4.2 Instanzen mit bestimmten Knotenzustand . . . 74
4.4.3 Zusammenfassung . . . 75
4.5 Zusammenfassung . . . 76

5 Messungen 77
5.1 Rahmenbedingungen . . . 77
5.2 Sekundärspeichermessungen . . . 79
5.2.1 Einführung . . . 80
5.2.2 XML . . . 81
5.2.3 PostgreSQL . . . 86
5.2.4 Oracle . . . 91
5.2.5 DB2 . . . 95
5.2.6 JPA . . . 100
5.2.7 Vergleich . . . 101
5.2.8 Zusammenfassung . . . 107
5.3 Primärspeichermessungen . . . 107
5.3.1 Schemata und Instanzen . . . 108
5.3.2 Clusterung . . . 111
5.4 Fazit . . . 114

6 Verwandte Konzepte 117

7 Zusammenfassung und Ausblick 119
7.1 Zusammenfassung 119
7.2 Ausblick 120

A Abbildungsverzeichnis vii

B Tabellenverzeichnis xi

C Literaturverzeichnis xiii

D Algorithmen xix
D.1 getNodeRelation xix
D.2 findPositions xix
D.3 preEvolveSelection xxiv

1 Einleitung

1.1 Motivation

Die Entwicklung des Menschen zum dominanten Wesen auf diesem Planeten liegt auch in der Fähigkeit begründet, prozessorientiert zu denken. Um ein Ziel zu erreichen, müssen zuerst andere Tätigkeiten als Vorbereitung ausgeführt werden. Zum Beispiel war es nötig, ein geeignetes Tier zu finden und zu erlegen, bevor man dessen Haut zu einem Kleidungsstück verarbeiten konnte. Im Laufe der Zeit wurde diese Prozessorientierung weiter verfeinert. So führte Henry Fords Erfolg des Fließbands am Anfang des 20. Jahrhunderts zu einer Art zu arbeiten, bei der jeder Arbeiter für einen Arbeitsschritt spezialisiert ist und die Fertigung eines Erzeugnisses nicht mehr von Anfang bis Ende begleitet. Die einsetzende Globalisierung ist bislang einer der letzten Schritte in dieser Entwicklung. Diese erfordert eine Unterstützung der Prozesse, welche in weltumspannenden Firmen ablaufen. Hierbei helfen Prozess-Management-Systeme (PMS), welche sich die elektronische Datenverarbeitung hierfür zu Nutze machen. So wird ein anstehender Prozessschritt zur richtigen Zeit dem richtigen Mitarbeiter vorgelegt, egal wo sich dieser auf der Welt befindet. Nachdem die anstehende Aufgabe bearbeitet wurde, wird der Prozess automatisch dem für den nächsten Prozessschritt verantwortlichen Mitarbeiter vorgelegt, welcher die Bearbeitung des Prozesses fortsetzt.

PMS müssen die Prozesse kennen, welche mit ihrer Hilfe durchgeführt werden sollen. Hierfür ist ein Modell dieser nötig, welches in Form einer geeigneten Repräsentation dem PMS zur Verfügung gestellt werden kann. Es bietet sich ein Prozessgraph an, mithilfe dessen ein Prozess vollständig beschrieben wird. Dieser Graph besteht hierbei aus einer Menge durch Kanten aneinander gereihten Prozessschritten, welche je einen Arbeitsschritt abbilden. Dieser Prozessgraph muss im PMS durch geeignete Strukturen beschrieben werden. Hierbei existieren verschiedene Varianten. Zu beachten ist dabei, dass für die Akzeptanz eines PMS die performante Verarbeitung von Anfragen unabdingbar ist. So ist es beispielsweise denkbar, das 10^5 Prozesse gleichzeitig ablaufen. Für diese ist eine effiziente Verwaltung und Repräsentation nötig.

1.2 Aufgabenstellung

In dieser Arbeit werden verschiedene Repräsentationen von Prozessen verglichen. Dies wird am Beispiel des ADEPT2-Metamodells [Rei00] untersucht, welches an der Universität Ulm entwickelt wurde. Die angestellten Untersuchungen sind allerdings allgemein gültig und auf andere PMS [vv04] übertragbar.

Die Prozessdaten gliedern sich in Schemata und Instanzen. Schemata werden einmalig modelliert und bilden die Vorlage, den „Bauplan“ eines Prozesses. Die Durchführung von diesem findet mithilfe von Instanzen statt. Diese basieren auf einem Schema und stellen die Abarbeitung der dort definierten Prozessschritte dar. ADEPT2 bietet hierbei große Flexibilität in Form von instanzbasierten Änderungen [RD98]. Dies bedeutet, dass der

Prozess-Bauplan einer Instanz geändert werden kann, ohne dass andere Instanzen, welche auf dem selben Schema basieren, von der Änderung betroffen sind. Ein weiteres Merkmal der Flexibilität ist die Schemaevolution [Rin04]. Mithilfe dieser kann ein Schema geändert und die darauf basierenden Instanzen zu Laufzeit automatisch auf das neue Schema, den neuen Bauplan des Prozesses, übertragen werden, sofern deren Zustand dies erlaubt. Die Herausforderung hierbei ist, auch instanzbasiert geänderte Instanzen automatisch zu übertragen.

Diese Flexibilität muss als ein Merkmal bei der Erarbeitung von Repräsentationen von Prozessdaten beachtet werden. Diese Arbeit stellt verschiedene Varianten vor, Schemata und Instanzen zu repräsentieren. Diese werden nach einer qualitativen Untersuchung des Speicherverbrauchs und der zu erwartenden Laufzeiten realisiert. Mithilfe dieser Implementierung werden anschließend Messungen durchgeführt, um die Repräsentationen auch quantitativ vergleichen zu können. Bei diesen Untersuchungen wird einerseits der Speicherplatzverbrauch der verschiedenen Varianten ermittelt und verglichen. Andererseits wird die Laufzeit von verschiedenen Anfragen, wie sie in PMS typisch sind, auf Basis der verschiedenen Repräsentationen untersucht. Das Ziel besteht darin, eine optimale Repräsentationen bezüglich der Anforderungen zu ermitteln.

1.3 Aufbau

Die Arbeit gliedert sich wie folgt: Die Grundlagen werden in Kapitel 2 vorgestellt. Hier werden neben dem ADEPT2-Metamodell und der ADEPT2-Architektur auch vorhandene Untersuchungen zur Repräsentation der Prozessdaten betrachtet. Ebenso werden typische Anfragen an das PMS beschrieben. Die hierauf aufbauenden Konzepte werden in Kapitel 3 entwickelt. Diese betreffen Repräsentationen von Schemata und die Verwaltung von Instanzen. Kapitel 4 stellt die Umsetzung der Grundlagen und Konzepte vor. Nach der Wahl einer Programmierumgebung wird die hierfür spezifische Umsetzung beschrieben. Aufbauend darauf werden Messungen durchgeführt, welche in Kapitel 5 betrachtet werden. Hierbei werden nach der Entwicklung von Testszenarien die Ergebnisse präsentiert und interpretiert und ein Fazit aus diesen gezogen. Kapitel 6 stellt verwandte Konzepte zur Repräsentation von Prozessen vor. Abschließend wird diese Arbeit in Kapitel 7 zusammengefasst und ein Ausblick auf Verbesserungen und Erweiterungen gegeben.

2 Grundlagen

Dieses Kapitel erläutert die Grundlagen auf denen diese Arbeit beruht. Nach der Vorstellung des ADEPT2-Metamodells werden aktuelle Forschungsergebnisse zur Repräsentation der Prozessdaten betrachtet. Es folgt eine Vorstellung der ADEPT2-Architektur, in welche sich die in dieser Arbeit entwickelten Konzepte einbetten. Die für Laufzeitmessungen notwendigen Anwendungsszenarien werden danach im Detail untersucht. Um den Speicherbedarf betrachten zu können, werden abschließend grundlegende Speicherarten rekapituliert.

2.1 Das ADEPT2-Metamodell

Das ADEPT2-Metamodell definiert Strukturen zur Darstellung von Prozessgraphen. Für das Verständnis ist zuerst eine Begriffsbildung nötig. Danach werden grundlegende Strukturen des ADEPT2-Metamodells vorgestellt. Da im Rahmen dieser Arbeit nicht das vollständige Metamodell benötigt wird, wird dieses abschließend vereinfacht.

2.1.1 Grundlegende Begrifflichkeiten

Ein *Prozess-Management-System* (PMS) verwaltet *Prozesse*, wobei diese auf *Schemata* abgebildet werden. Ein Schema ist dabei die Vorlage, der „Bauplan" eines Prozesses. Hierauf basieren *Instanzen*, welche die Durchführung eines Prozesses abbilden. Diese Trennung zwischen Modellierung und Ausführung ist sinnvoll, um einen häufig ablaufenden Prozess nicht mehrmals definieren zu müssen. Dieser kann stattdessen *einmal* modelliert und *häufig* durchgeführt werden. In Abbildung 1 ist ein Schema mit zwei darauf basierenden Instanzen dargestellt.

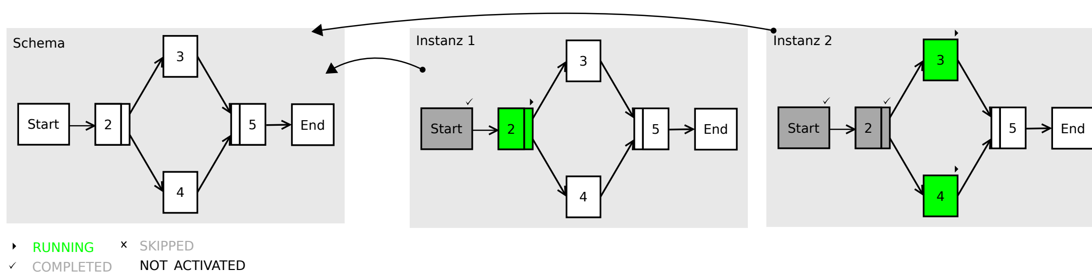


Abbildung 1: Ein Schema mit zwei Instanzen

Das ADEPT2-Metamodell [Rei00] definiert ein Schema als Graph [bpm09, Ley01, LA94, MWW+98]. Dieser ist azyklisch und gerichtet [DD90, CLRS09], wobei jeder Knoten einen Arbeitsschritt, einen so genannten *Prozessschritt* abbildet. Zusätzlich beinhaltet es einen Start- und einen End-Knoten, welche den Start- bzw. Endpunkt des Prozesses darstellen.

Durch die Verknüpfung der Knoten mit gerichteten Kanten entsteht eine Abfolge von Prozessschritten, welche in der durch die Kanten angegebenen Reihenfolge durchlaufen werden. Diese Kanten heißen *Kontrollkanten* und bilden den so genannten *Kontrollfluss*.

Jeder Knoten besitzt eine *Aktivität*, welche bei der Durchführung dieses Prozessschrittes ausgeführt wird. Hierbei kann es sich zum Beispiel um die Anzeige oder Abfrage von Daten mithilfe von Formularen [Mic09] handeln. Da ADEPT2 als Mehrbenutzersystem konzipiert ist, muss für jede Aktivität eine *Bearbeiterzuordnung* definiert werden. Diese legt fest, welcher Mitarbeiter den Prozessschritt zur Ausführung vorgelegt bekommt. Hierfür existiert für jeden Benutzer eine *Arbeitsliste*, in welcher die anstehenden Prozessschritte aufgeführt sind.

Bei der Durchführung der Aktivitäten fallen Daten an, welche erzeugt, geändert, gelesen [Rei00] oder konsumiert [For09] werden können. Dies wird durch den *Datenfluss* [RD98] abgebildet, welcher implizit Teil des Schemas ist. Er besteht ebenso aus Knoten, den so genannten *Datenelementen*, welche über *Datenkanten* mit den Prozessschritten verbunden sind. Die Datenkanten sind hierbei ebenfalls gerichtet und bestimmen, ob der Prozessschritt lesend oder schreibend auf das Datenelement zugreift.

2.1.2 Schemata

Schemata sind Prozessgraphen, welche die Vorlage von Prozessen bilden. Sie werden in dieser Arbeit mithilfe des ADEPT2-Metamodells dargestellt, welches verschiedene Strukturen hierfür vorsieht. So existieren neben verschiedenen Kantenarten auch verschiedene Typen von Knoten. Normale Knoten besitzen eine Aktivität, Verzweigungs- und Schleifenknoten dienen zur Steuerung des Kontrollflusses und Start- und End-Knoten bilden Einstiegs- bzw. Endpunkt des Prozesses. Für diese Strukturen existiert eine Notation, welche in Abbildung 2 dargestellt ist.

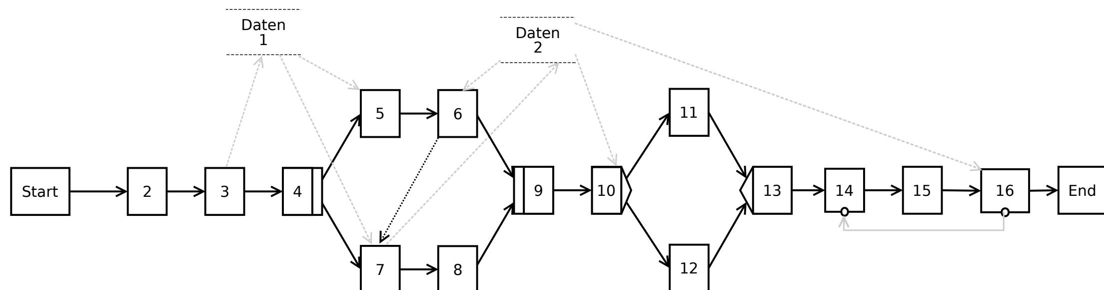


Abbildung 2: Ein ADEPT2-Schema

In Abbildung 2 bilden die Knoten 2 und 3 eine *Sequenz*. Der nachfolgende sogenannte *AND-Split*-Knoten 4 leitet eine *Parallelverzweigung* ein. Es folgen verschiedene Zweige, welche alle parallel abgearbeitet werden. Die Zusammenführung dieser bewerkstelligt Knoten 9, welcher ein *AND-Join*-Knoten ist. Knoten 10 ist ein *XOR-Split*-Knoten, welcher eine *Entweder-Oder-Verzweigung* einleitet. Diesem folgen wiederum mehrere Zweige,

wobei hier allerdings zur Laufzeit nur einer der Zweige ausgeführt wird und die anderen ignoriert werden. Die Entscheidung, welcher Zweig ausgeführt wird, wird durch Auswerten des Datenelements 2 getroffen. Die Zusammenführung der Zweige erfolgt durch Knoten 13, einem so genannten *XOR-Join*-Knoten. Der Knoten 14 ist ein *Loop-Start*-Knoten. Die Schleife wird durch den zugehörigen *Loop-End*-Knoten 16 begrenzt. Die Kante von Knoten 16 zu Knoten 14 ist hierbei eine spezielle *Schleifenkante*. Diese wird im ADEPT2-Metamodell gesondert behandelt, um die azyklische Eigenschaft des Prozessgraphen zu erhalten. Diese Eigenschaft gilt bezüglich der Kontrollkanten (nicht der Schleifen- und Datenkanten) und ist für Korrektheitsuntersuchungen notwendig.

Die Kanten eines Schemas können attributiert sein. Hierbei ist zum Beispiel der Kantentyp zu nennen. Es gibt Kontroll-, Schleifen-, Daten- und so genannte *Synchronisationskanten*. Letztere dienen dazu, die Parallelausführung von Knoten zeitlich aufeinander abzustimmen, so dass ein Prozessschritt zwingend nach dem anderen ausgeführt wird. Dies ist zum Beispiel sinnvoll, wenn ein Prozessschritt eines parallelen Zweigs Daten ändert, die von einem anderen Prozessschritt benötigt werden, welcher auf einem anderen Zweig der selben parallelen Verzweigung liegt. Solch eine Kante ist in Abbildung 2 zwischen Knoten 6 und 7 zu sehen. Hierbei kann der Prozessschritt 6 erst gestartet werden, wenn Knoten 7 abgeschlossen ist. Dies wird benötigt, da die Daten von Prozessschritt 7 zuerst in das Datenelement 2 geschrieben werden müssen, bevor Prozessschritt 6 diese lesen kann.

Betrachtet man die Knoten des Schemas ausschließlich unter Beachtung der Kontrollkanten, so entsteht eine streng hierarchische Blockstruktur [Rei00] wie die Rechtecke verschiedener Grautöne in Abbildung 3 verdeutlichen.

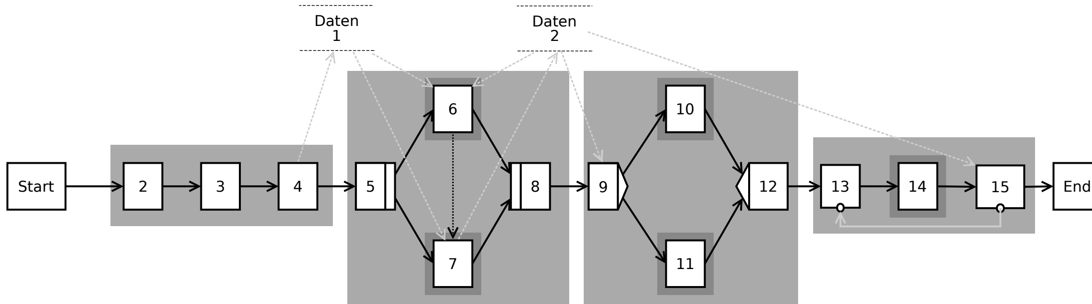


Abbildung 3: Die Blockstruktur eines ADEPT2-Schemas

Diese Blockstruktur zeigt, dass Kontrollstrukturen wie Sequenzen, Schleifen, Parallel- und Entweder-Oder-Verzweigungen eindeutige Start- und Endknoten besitzen. Hierdurch ist es zum Beispiel möglich einem Split-Knoten einen Join-Knoten eindeutig zuzuordnen. Diese Blöcke dürfen sich nicht überlagern, können aber beliebig ineinander geschachtelt werden. Durch diese strenge Struktur werden Prozesse übersichtlicher und viele Fehler, welche beim Modellieren unterlaufen könnten (zum Beispiel falsches Zusammenführen von Zweigen), sind dadurch ausgeschlossen. Eine korrekte Blockstruktur ist Teil der De-

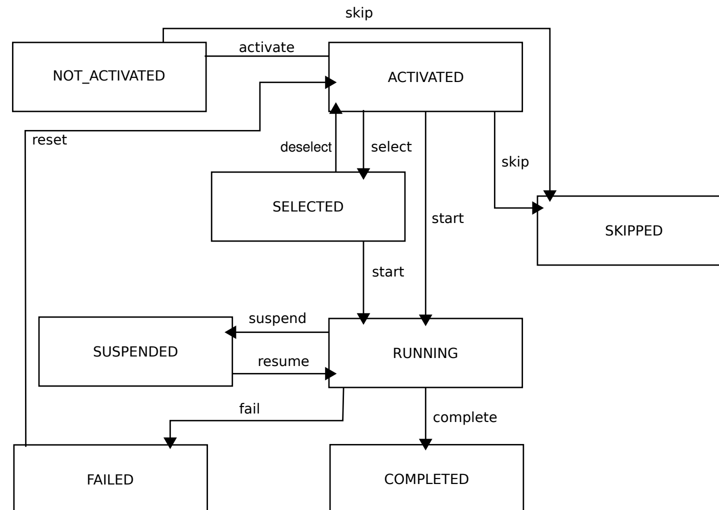


Abbildung 4: Knotenzustände und -übergänge (in Anlehnung an [Rei00])

finition von *Korrektheit* des ADEPT2-Metamodells. Hierzu zählt auch der Ausschluss von Verklemmungen, wie sie durch Synchronisationskanten zustande kommen können [Jur06]. Ebenso beachtet werden Datenelemente. Es ist sichergestellt, dass ein Datenelement nicht gelesen werden kann, bevor es geschrieben wurde und dass kein geschriebener Wert ungelesen überschrieben wird („lost update"). Hinzu kommt, dass Synchronisationskanten in oder aus Schleifenkonstrukten verboten sind. Durch die Bereitstellung von *Änderungsoperationen* [Jur06] ist diese Korrektheit sichergestellt. Diese dienen zum Ändern von Schemata. Dabei gelten für die Anwendung einer Änderungsoperation Vor- und Nachbedingungen, mithilfe derer eine Änderung hin zu einem inkorrekten Schema festgestellt und unterbunden wird.

Bei der Erzeugung eines Schemas wird initial ein Prozessgraph angelegt, welcher ausschließlich aus Start- und Endknoten besteht. Dieser kann im Folgenden vom Modellierer mithilfe der Änderungsoperationen so geändert werden, dass der gewünschte Prozess abgebildet wird. Somit ist das Korrektheitskriterium für Schemata immer erfüllt.

2.1.3 Instanzen

Aufbauend auf einem Schema, kann der hierin modellierte Prozess mithilfe einer *Instanz* ausgeführt werden. Diese repräsentiert die Ausführung des Prozesses und reichert das Schema hierfür mit einer *Instanzmarkierung* an, welche jedem Knoten einen der aus Abbildung 4 zu entnehmenden Zustände zuweist. Außerdem beinhaltet die Instanz auf logischer Ebene die Werte der Datenelemente des Schemas (vergleiche Abbildung 5).

Zu Beginn besitzen alle Knoten den Zustand NOT_ACTIVATED. Als Erstes wird der Start-Knoten gestartet und somit in den Zustand RUNNING versetzt. Nach der Aus-

führung dessen geht der folgende Knoten in den Zustand ACTIVATED über, welcher dem zugeordneten Benutzer dann in dessen Arbeitsliste angezeigt wird. Markiert der Benutzer den Prozessschritt zur Bearbeitung, bekommt der Knoten den Zustand SELECTED. Sobald dieser ausgeführt wird, folgt die Markierung des Knotens als RUNNING: Die beim Knoten hinterlegte Aktivität wird gestartet. Der Prozessschritt kann während seiner Ausführung pausiert werden, was durch einen Übergang in den Zustand SUSPENDED widergespiegelt wird. Hierbei wird die laufende Aktivität ebenfalls unterbrochen. Die Durchführung des Prozessschritts kann dann später fortgesetzt werden. Schlägt die Durchführung der Aktivität fehl, so wird der Knoten in den Zustand FAILED versetzt. Bei einer erfolgreichen Abarbeitung der Aktivität bekommt der Knoten den Zustand COMPLETED. Der Zustand SKIPPED markiert Knoten, welche auf dem Zweig eines XOR-Split-Knotens liegen, der nicht zur Bearbeitung ausgewählt wurde.

2.1.4 Abstraktion

In dieser Arbeit wird von gewissen Aspekten des ADEPT2-Metamodells abstrahiert. So werden Datenelemente inklusive ihrer Datenkanten nicht beachtet, da die Betrachtung dieser den Rahmen dieser Arbeit sprengen würde. Die gesonderte Behandlung von Schleifen ist hier überflüssig, da der Schleifenkörper als Sequenz angesehen werden kann. Hieraus folgt, dass auch Schleifenkanten nicht betrachtet werden. Auch Synchronisationskanten sind nicht relevant für diese Arbeit, somit werden nur Kontrollkanten betrachtet. Diese werden im Folgenden einfach *Kanten* genannt.

Auch die Menge an Knotenzuständen lässt sich verringern. Im Rahmen dieser Arbeit sind nur die Zustände NOT_ACTIVATED, RUNNING, COMPLETED und SKIPPED interessant. ACTIVATED ist überflüssig, da sich dieser Zustand für einen Knoten berechnen lässt, in dem man überprüft, ob der Vorgängerknoten im Zustand COMPLETED ist. SELECTED ist ebenfalls vernachlässigbar, da dies nur das „Reservieren“ eines Prozessschrittes für einen Benutzer des Systems ist und für die folgenden Betrachtungen keine Auswirkungen hat. Diese würden sich allerdings auch leicht anpassen lassen. SUSPENDED und FAILED werden im Rahmen dieser Arbeit äquivalent zu RUNNING angesehen. SUSPENDED ist ein temporärer Zustand eines Knotens, welcher RUNNING folgt und

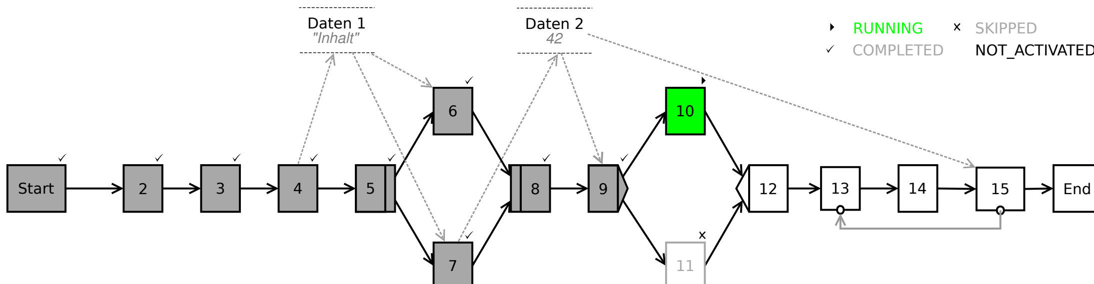


Abbildung 5: Eine ADEPT2-Instanz während Knoten 10 ausgeführt wird

auch von RUNNING gefolgt wird. FAILED kann in diesem Zusammenhang ebenfalls wie RUNNING angesehen werden, da Knoten den Zustand FAILED nur über RUNNING erreichen können und im Rahmen dieser Arbeit keine Differenzierung zwischen FAILED und RUNNING notwendig ist.

2.1.5 Zusammenfassung

In diesem Abschnitt wurde das ADEPT2-Metamodell vorgestellt. Hierzu wurden nach einer grundlegenden Begriffsbildung Schemata und Instanzen betrachtet. Da nicht alle Aspekte des ADEPT2-Metamodells in dieser Arbeit betrachtet werden müssen, wurde abschließend eine Abstraktion dessen durchgeführt.

2.2 Repräsentation

Das vorgestellte ADEPT2-Metamodell modelliert Schemata als gerichtete, azyklische Graphen. Diese lassen sich in verschiedenen Varianten im Speicher repräsentieren [KRRD09, KR10], welche in diesem Abschnitt vorgestellt werden. Ebenso betrachtet werden verschiedene Realisierungsmöglichkeiten von Instanzen und, aufbauend darauf, instanzbasierte Änderungen. Diese sind, wie sich herausstellen wird, für die geforderte Flexibilität des ADEPT2-Metamodells nötig.

2.2.1 Schemata

Schemata, welche aus Knoten und Kanten bestehen, lassen sich in verschiedenen Varianten im Speicher repräsentieren [KRRD09]. Da Knoten eine große Anzahl an Attributen besitzen, empfiehlt sich die Realisierung durch eigenständige Knoten-Objekte. Diese besitzen jeweils eine ID, die ID des direkt übergeordneten Split-Knotens und des Zweigs auf dem sich der Knoten befindet, sowie eine topologische ID [JMSW04]. Die IDs sind aus Performanzgründen vorhanden und beschleunigen zum Beispiel Schleifeniterationen, wenn das vollständige ADEPT2-Metamodell eingesetzt wird. Die topologische ID bildet

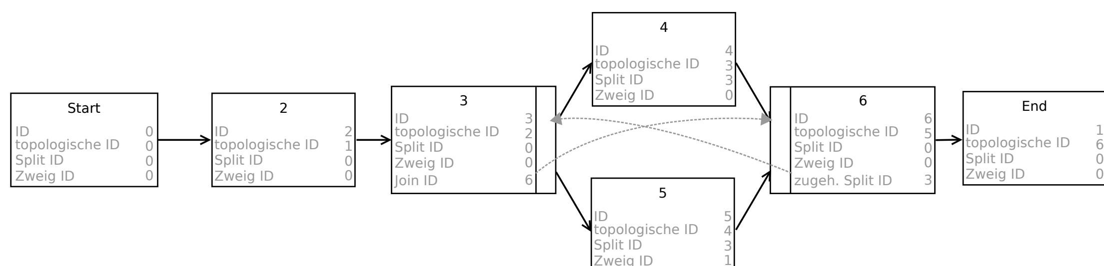


Abbildung 6: Die Knoten eines Schemas und deren Knotenattribute

eine topologische Sortierung [Sch01a] der Knoten innerhalb des Schemas ab, welche aufgrund der Zyklenfreiheit des ADEPT2-Metamodells verfügbar ist. Mithilfe der Zweig-ID lassen sich Nachfolgerknoten von Split-Knoten zu den Zweigen des Split-Knotens zuordnen. Split-Knoten besitzen zusätzlich einen Verweis auf den zugehörigen Join-Knoten und Join-Knoten einen Verweis auf den zugehörigen Split-Knoten [JMSW04]. Dies wird ebenfalls in Form von Knoten-IDs repräsentiert. Abbildung 6 illustriert das.

Für die Realisierung der Kanten zwischen den Knoten-Objekten ergibt sich eine Adjazenzliste [CLRS09] als optimale Alternative [KRRD09]. Dies bedeutet, dass das Schema eine Menge von Kanten verwaltet, welche aus einem Paar aus Quell-Knoten-ID und Ziel-Knoten-ID bestehen. Zusätzlich können die Kanten bei ADEPT2 attributiert sein. Auch dies ist mithilfe einer Adjazenzliste leicht umsetzbar, da hier ein weiterer Wert zu jedem Quell-/Ziel-Knoten-ID-Paar hinzugefügt wird, mithilfe dessen die Attribute der Kante verwaltet werden.

2.2.2 Instanzen

Da eine Instanz die Ausführung eines Schemas abbildet, muss diese Beziehung in der Repräsentation von Instanzen modelliert werden. Hierfür bieten sich verschiedene Möglichkeiten an [KRRD09]. Als beste Variante hat sich die Schemareferenz herausgestellt, welche eine Referenz auf die Schemadaten besitzt. Im Gegensatz zur Schemakopie werden

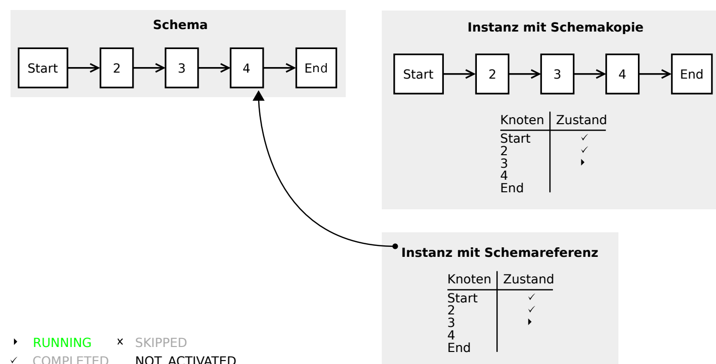


Abbildung 7: Schemakopie vs. Schemareferenz

bei dieser Variante die Graphstrukturen nicht von der Instanz redundant gehalten (siehe Abbildung 7). Im Folgenden werden Instanzen zur besseren Lesbarkeit allerdings aus der logischen Sicht inklusive der Graphstrukturen wie in Abbildung 5 dargestellt.

Eine Instanz reichert ein Schema mit einer Zustandsmarkierung an, welche für jeden Knoten einen Knotenzustand liefern können muss. [KRRD09] nennt hier verschiedene Möglichkeiten, wobei in dieser Arbeit die implizite und die explizite Markierung sowie die Clusterung von Interesse sind. Die *implizite Markierung* speichert nur Zustände für die Knoten, welche im Zustand RUNNING oder SKIPPED sind. Aufgrund der

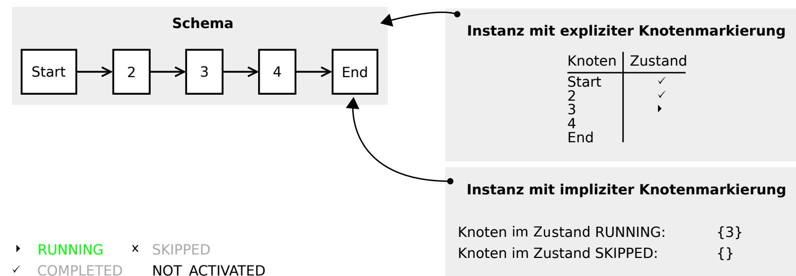


Abbildung 8: Instanz mit impliziter und expliziter Markierung

Vorgänger- und Nachfolgerbeziehungen zwischen den Knoten des Schemas lassen sich die übrigen Zustände zur Laufzeit errechnen. So befinden sich alle (transitiven) Vorgänger eines RUNNING-Knotens im Zustand COMPLETED und die (transitiven) Nachfolger im Zustand NOT_ACTIVATED. Die *explizite Markierung* speichert hingegen für jeden Knoten den zugehörigen Zustand. Dies bedeutet mehr Speicheraufwand, aber die Ersparnis von Berechnungen der Knotenzustände NOT_ACTIVATED und COMPLETED zur Laufzeit. In Abbildung 8 sind implizite und explizite Markierung illustriert.

Die *Clusterung* von Instanzen [KRRD09] geht hierbei einen anderen Weg: Die Instanzen werden nicht direkt markiert, sondern in einem Clusterbaum gehalten. Dieser besteht aus Clustern, welche jeweils genau einen Zustand der Instanz abbilden, wie Abbildung 9 zeigt. Hierbei existiert für jeden Zustand der Instanz genau ein Cluster. Ein Cluster verweist auf eine Menge an Instanzen, welche in diesem Zustand sind. Da alle Zustände, die eine Instanz annehmen kann durch das Schema definiert sind, kann der Clusterbaum zur Modellierzeit des Schemas vorberechnet werden.

Eine weitere Möglichkeit, den Zustand eines Knotens zu bestimmen, ist die *Ausführungshistorie* zu untersuchen. Diese beinhaltet alle Markierungsänderungen der Instanz in einer zeitlich sortierten Liste. Die aktuelle Knotenmarkierung ist hierbei die zeitlich zuletzt eingetragene Markierungsänderung des Knotens.

2.2.3 Instanzbasierte Änderungen

Das ADEPT2-Metamodell unterstützt das Ändern des Prozessgraphen einer bestimmten Instanz. Dieser wird durch das Schema abgebildet, auf das die Instanz eine Referenz hält. Da allerdings mehrere Instanzen auf diesem Schema basieren, welche nicht geändert werden sollen, kann nicht das zugrunde liegende Schema geändert werden.

Die Lösung bietet die *Deltaschicht*, welche Änderungen gegenüber einem Schema abbildet [LRR04, JMSW04, Jur06]. Bei einer instanzbasierten Änderung wird eine Deltaschicht erzeugt, welche von der Instanz referenziert wird.

Es existieren verschiedene Ansätze, die Deltaschicht im Speicher zu repräsentieren [KR10]. Auf der einen Seite ist dabei die Art des Zugriffs der Instanz auf Schemadaten

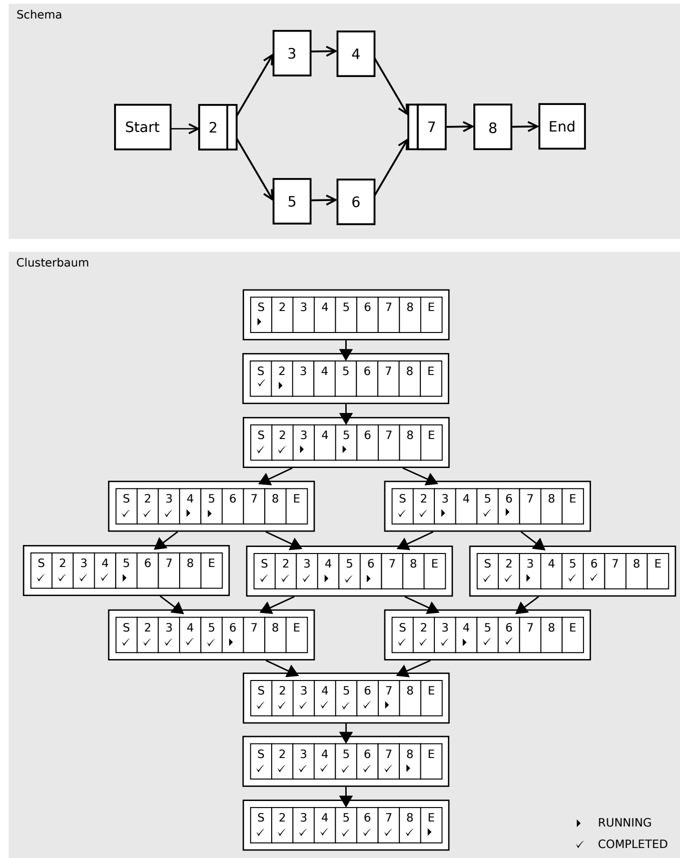


Abbildung 9: Clusterung

zu nennen. Hierbei hat sich die *Kapselung* der Schemadaten als vorteilhaft herausgestellt [KR10]. Dabei bildet die Deltaschicht die gleichen Zugriffsmöglichkeiten wie ein vollwer-

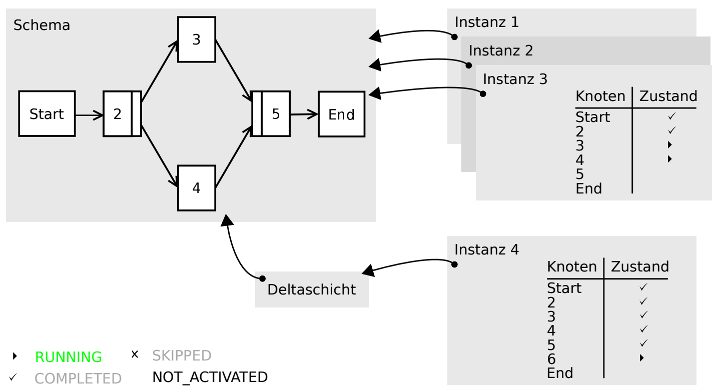


Abbildung 10: Instanzbasierte Änderungen mit Deltaschicht

tiges Schema. Wird eine Anfrage an die Deltaschicht gestellt, so wird überprüft, ob diese einen geänderten Bereich des Schemas betrifft oder nicht. Dementsprechend werden die überlagerten Daten zurückgeliefert, welche in der Deltaschicht gehalten werden, oder die Daten vom zugrunde liegenden Schema abgerufen. Hierzu benötigt die Deltaschicht eine Referenz auf dieses. Durch Einsatz der Kapselung stellt es sich für eine Instanz als transparent dar, ob diese auf einem vollwertigen Schema oder auf einer Deltaschicht beruht.

Auf der anderen Seite ist die Repräsentation der Änderungen von Bedeutung. Diese können als *operationale* oder *strukturelle* Deltas dargestellt werden. Operationale Deltas speichern die bei der Änderung durchgeführten Änderungsoperationen, strukturelle

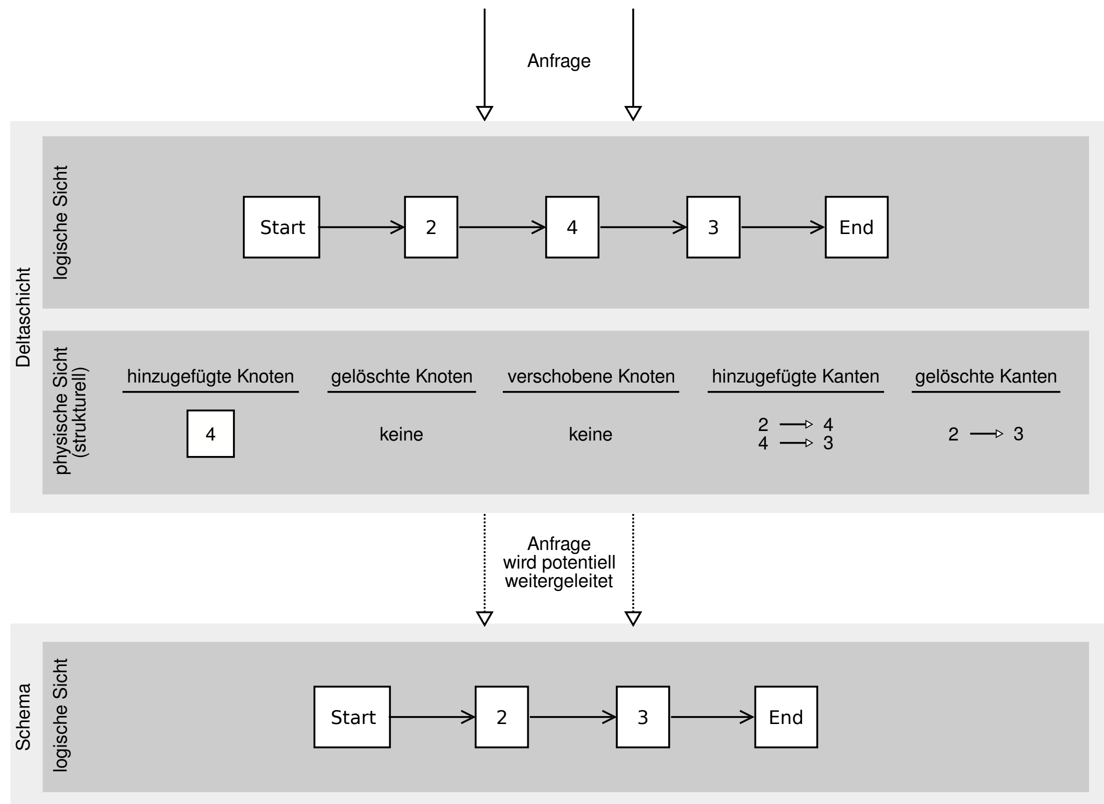


Abbildung 11: Kapselung bei Einsatz einer strukturellen Deltaschicht

Instanzdeltas halten hingegen realisierte Graphstrukturen der geänderten Bereiche nach der Anwendung der Änderungsoperationen. Hierbei stellen sich strukturelle Instanzdeltas als besser geeignet heraus [KR10]. Diese werden durch Mengen repräsentiert, welche hinzugefügte, gelöschte und verschobene Knoten und hinzugefügte und gelöschte Kanten beinhalten [Jur06, KR10]. Dabei können die gelöschten und verschobenen Knoten ausschließlich mit ihrer ID vorgehalten, während von hinzugefügten Knoten alle Daten (vergleiche Abbildung 6) benötigt werden [Jur06]. In Abbildung 11 ist die Überlagerung der Daten bei einer Anfrage (zum Beispiel von einer Instanz) mithilfe einer gekapsel-

ten, strukturell aufgebauten Deltaschicht dargestellt. Hierbei wird Knoten 4 zwischen die Knoten 2 und 3 hinzugefügt.

2.2.4 Zusammenfassung

Es wurden die Repräsentationen des ADEPT2-Metamodells vorgestellt. Ein Schema besteht aus mehreren Knotenobjekten und einer Adjazenzliste von Kanten. Für Instanzen, welche mithilfe von Schemareferenzen dargestellt werden, bieten sich mehrere Alternativen zur Repräsentation der Knotenzustände: Eine implizite Markierung, welche nur bestimmte Knotenzustände speichert und die restlichen zur Laufzeit errechnet, eine explizite Markierung, die alle Knotenzustände vorhält und als orthogonales Konzept die Clusterung von Instanzen. Um instanzbasierte Änderungen zu unterstützen, wird eine Deltaschicht benutzt, welche als gekapseltes Schema die Daten intern als realisierte Graphstrukturen vorhält.

2.3 ADEPT2-Architektur

Das ADEPT2-System implementiert das ADEPT2-Metamodell. Hierbei kommt eine schichtartige Architektur zum Einsatz, wobei jede Schicht aus verschiedenen Komponenten aufgebaut ist [RDR+09]. Jede Schicht versteckt so viel Komplexität wie möglich, ohne dabei die Möglichkeit zur flexiblen Konfiguration zu verlieren. Diese Arbeit bewegt sich in dem in Abbildung 12 rot eingefärbten Bereich.

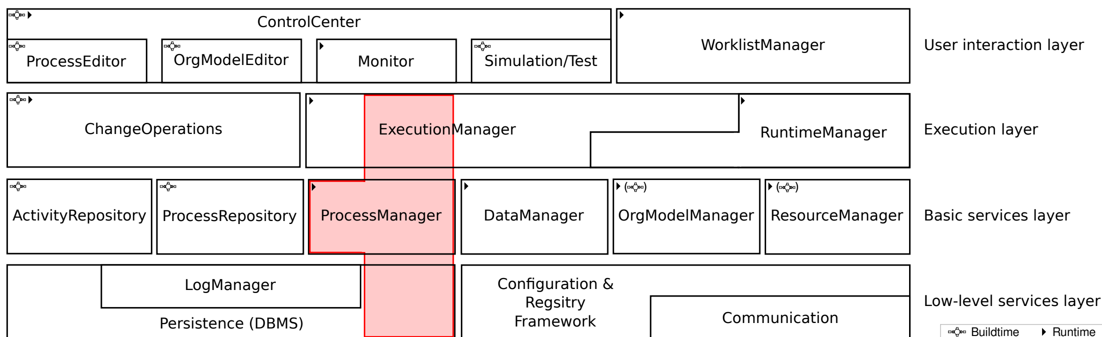


Abbildung 12: Die ADEPT2-Architektur [RDR+09]

Die unterste Schicht (*Low-level services layer*) stellt die persistente Datenspeicherung zur Verfügung. Dadurch sind übergeordnete Schichten unabhängig von der eingesetzten Variante zur Persistierung. Hierbei sind mehrere Möglichkeiten denkbar, wobei relationale Datenbanksysteme oder eine Speicherung im Dateisystem möglich sind. Da diese Arbeit Schemata und Instanzen persistiert, ist ein Teil der Komponente zur Persistenz rot eingefärbt. Der *Low-level services layer* bietet ebenfalls die Möglichkeit, verschiedene

Nachrichten aus dem System zu protokollieren (*LogManager*) und bietet den Komponenten eine Abstraktion zur Kommunikation untereinander und damit die Möglichkeit zur Verteilung der Komponenten auf unterschiedlichen Rechnern. Dies wird durch die Kommunikationskomponente bereitgestellt und läuft für die übrigen Komponenten transparent ab. Die unterste Schicht stellt auch Zugriff auf die Konfiguration des Systems bereit.

Der *Basic services layer* stellt grundlegende Dienste zur Verfügung. Das *ActivityRepository* verwaltet die Aktivitäten, welche auf Knoten von Schemata platziert sind. Aufgrund der Flexibilität des ADEPT2-Metamodells und speziell der Unterstützung der Schemaevolution, können Schemata sich mit der Zeit entwickeln. Dies erfordert eine Versionierung der Schemata, welche das *ProcessRepository* übernimmt. Die einzelnen Schemata werden ohne die Versionierungsinformationen im *ProcessManager* zusammen mit den Instanzen verwaltet. Dies beinhaltet auch die Wahl der Repräsentation der Prozessdaten. Diese Komponente ist auch für die Migration der Instanzen bei instanzbasierten Änderungen und der Schemaevolution zuständig. Diese Arbeit umfasst den Aufgabenbereich des *ProcessManager*s komplett. Der *DataManager* versorgt andere Komponenten mit Daten, welche in Prozessen angefallen sind. Hierzu speichert und verwaltet er die Werte der einzelnen Datenelemente aller Instanzen. Im *OrgModelManager* ist ein Organisationsmodell der Benutzer des Systems abgebildet, mithilfe dessen die Bearbeiterzuordnungen der Prozessschritte ausgewertet werden. Der *ResourceManager* verwaltet zusätzliche Ressourcen, welche in einem Unternehmen vorhanden sind (zum Beispiel Räume oder Maschinen).

Der *Execution layer* sorgt für die Ausführung und Änderung der Prozesse. Es wird zurückgegriffen auf die im *ProcessManager* gespeicherten Prozesse, welche mithilfe des *ExecutionManager*s und des *RuntimeManager*s ausgeführt und deren Aktivitäten gestartet werden. Hierbei umfasst diese Arbeit Teile des *ExecutionManager*s, da Funktionen auf Schemata und Instanzen durchgeführt werden sollen. Die Komponente *ChangeOperations* beinhaltet die Änderungsoperationen, mithilfe denen das Anlegen oder Ändern eines Schemas möglich ist.

Die oberste Schicht, der *User interaction layer* sorgt schließlich mithilfe von Endbenutzer-Software dafür, dass das System korrekt angesteuert und benutzt wird, es bildet die Schnittstelle zum Benutzer. Hierzu gehört der *ProcessEditor*, welcher Schemata anlegen und ändern kann. Der *OrgModelEditor* bildet die grafische Oberfläche zum Steuern des *OrgModelManager*s. Mithilfe des *Monitor*s können Instanzen untersucht werden und es existiert die Möglichkeit zur Simulation und zum Test von Prozessen. Hinzu kommt der *WorklistManager* welcher die Arbeitslisten der Benutzer verwaltet. Dieser besitzt keine eigene grafische Oberfläche, wird allerdings von den anderen Komponenten dieser Schicht benutzt.

2.4 Anwendungsszenarien

In diesem Abschnitt werden Funktionen auf Basis von Schemata und Instanzen vorgestellt. Dies sind Funktionen, welche intern von PMS durchgeführt werden und später in dieser Arbeit zu Messungen herangezogen werden.

In dieser Arbeit werden die Funktionen *Weiterschalten einer Instanz*, *Schemaevolution* und *Anfragen auf Kollektionen* betrachtet.

2.4.1 Weiterschalten

Das Weiterschalten [Rei00] ist die Änderung der Markierung einer Instanz gemäß den Regeln des Metamodells. Wird die Bearbeitung einer Aktivität abgeschlossen, muss der nachfolgende Prozessschritt dem zuständigen Benutzer oder die nachfolgenden Prozessschritte den zuständigen Benutzern vorgelegt werden. Diese zu aktivierenden Prozessschritte müssen zuerst identifiziert und danach die Knotenzustände angepasst werden.

Um Weiterschalten zu können, werden zwei Informationen benötigt: Welcher Knoten weitergeschaltet und, im Fall einer Entweder-Oder-Verzweigung, welcher Zweig aktiviert werden soll. Der weiterzuschaltende Knoten muss dabei im Zustand RUNNING sein. Ihm wird beim Weiterschalten der Zustand COMPLETED zugewiesen.

Ist der weiterzuschaltende Knoten ein normaler Knoten, ein AND-Split-Knoten oder ein Join-Knoten, so werden alle direkten Nachfolgerknoten ermittelt und diesen der Zustand RUNNING zugewiesen, wie die ersten vier Beispiele in Abbildung 13 veranschaulichen. Ist dieser hingegen ein XOR-Split-Knoten, so muss der auf diesem Zweig folgende Knoten gefunden und auf den Zustand RUNNING gesetzt werden. Zusätzlich müssen alle anderen Zweige durchlaufen und die Zustände dieser Knoten auf SKIPPED gesetzt werden. Dies ist im letzten Beispiel der Abbildung 13 dargestellt.

Ist der Knoten, der durch das Weiterschalten auf RUNNING gesetzt werden soll, ein AND-Join-Knoten, so müssen die anderen Zweige untersucht werden, bevor dies geschieht. Ein AND-Join-Knoten darf erst aktiviert werden, wenn alle Zweige abgearbeitet, also die jeweils topologisch letzten Knoten auf den Zweigen den Zustand COMPLETED haben. Sollte dies nicht zutreffen, darf der AND-Join-Knoten nicht auf RUNNING gesetzt werden, sondern muss im Zustand NOT_ACTIVATED verbleiben. Dies ist im dritten Beispiel in Abbildung 13 zu sehen: Nachdem Knoten 3 weitergeschaltet wurde, darf Knoten 5 nicht auf RUNNING gesetzt werden, da sich Knoten 4 noch in der Ausführung befindet.

2.4.2 Schemaevolution

Durch gesetzliche Änderungen oder aufgrund von Anpassungen des Betriebsablaufs, kann es nötig werden, die im PMS vorhandenen Prozesse zu ändern. Hierbei reicht eine Än-

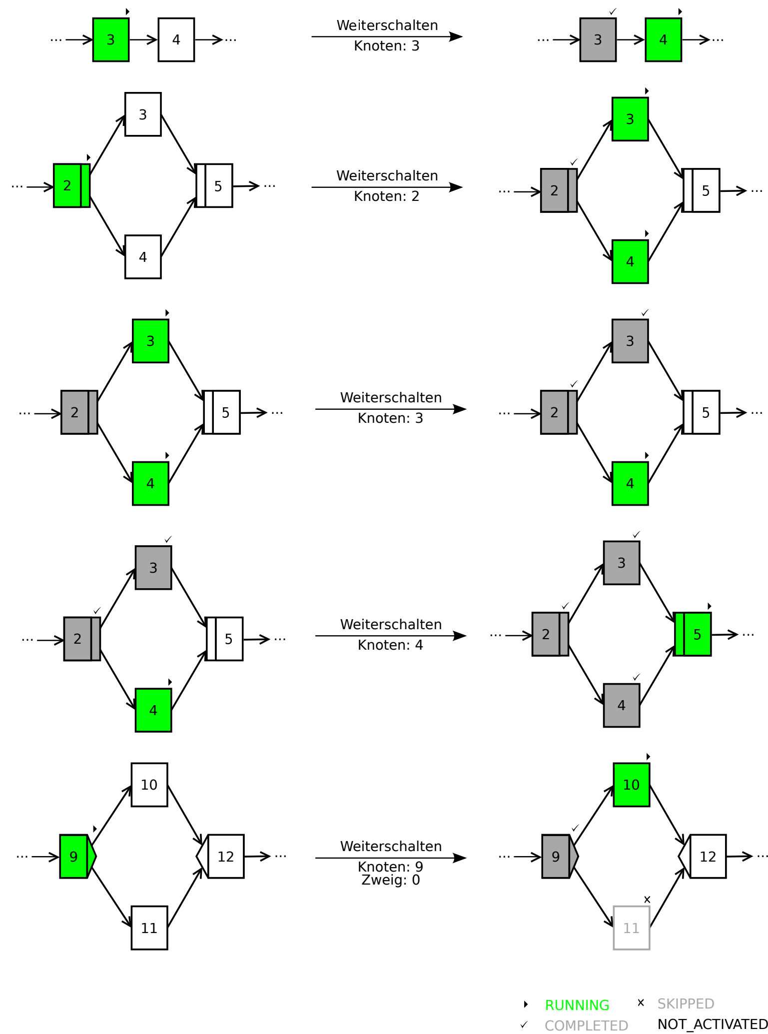


Abbildung 13: Die verschiedenen Fälle des Weiterschaltens

derung des entsprechenden Schemas nicht aus. Es ist davon auszugehen, dass Instanzen auf dem zu ändernden Schema beruhen, welche sich in der Ausführung befinden. Auf der einen Seite wäre es möglich, die vorhandenen Instanzen weiterhin auf Basis des „alten" Schemas durchzuführen und neue Instanzen mit dem neuen Schema zu starten. Dies ist allerdings nicht immer durchführbar. So besteht die Möglichkeit, dass der Gesetzgeber eine sofortige Umstellung auch der laufenden Prozesse fordert oder sonstige Umstände eine Migration der Instanzen notwendig machen. Dies ist die Grundlage der Schemaevolution [Rin04].

Der Schemaevolution stehen neben dem „alten" Schema und den Instanzen, welche auf diesem basieren, die durchgeführten Änderungen in Form einer Deltaschicht zur Verfügung [Jur06] (vergleiche Abschnitt 2.2.3). Ziel der Schemaevolution ist die Untersuchung

der Instanzen auf ihre Verträglichkeit mit dem neuen, geänderten Schema. Diese Verträglichkeit gliedert sich in eine *zustandsbasierte Verträglichkeit* und eine *strukturelle Verträglichkeit*. Abbildung 14 zeigt eine Schemaevolution und illustriert Verträglichkeitsprobleme.

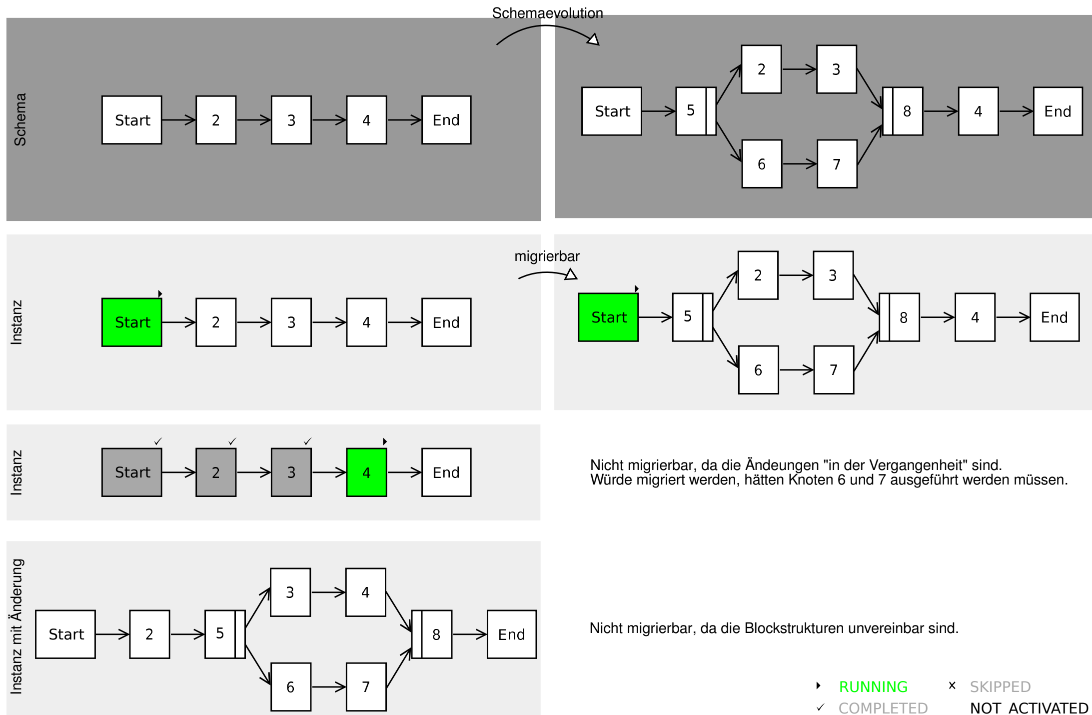


Abbildung 14: Beispiel einer Schemaevolution

Die zustandsbasierte Verträglichkeit überprüft, ob geänderte Stellen bei einer Instanz bereits abgearbeitet wurden, die Änderungen somit sozusagen „in der Vergangenheit“ der Instanz liegen. Ist dies der Fall, so ist die Instanz nicht verträglich. Formal wird definiert, dass eine Instanz zustandsbasiert verträglich ist, wenn ihre Ausführungshistorie auch auf dem neuen Schema nachvollzogen werden kann [Rin04].

Die Überprüfung der strukturellen Verträglichkeit ist dafür zuständig, Prozesse zu identifizieren, welche dem Korrektheitskriterium nicht entsprechen (vergleiche Abschnitt 2.1.2): Ist eine zu untersuchende Instanz instanzbasiert geändert, so müssen die instanzbasierten Änderungen mit den Schemaänderungen vereint werden. Hierbei ist es möglich, dass das resultierende Schema nicht länger korrekt ist, da z.B. Verklemmungen auftreten können oder die Blockstruktur nicht länger korrekt ist [Rin04, Jur06]. Ist dies der Fall, ist eine solche Instanz nicht verträglich. Für Instanzen, welche nicht instanzbasiert geändert wurden, ist die strukturelle Verträglichkeit immer gegeben, da hier keine zwei Änderungsmengen vereint werden müssen und somit die Korrektheit durch die Änderungsoperationen sichergestellt ist.

Migrierbare Instanzen müssen sowohl zustandsbasiert als auch strukturell verträglich sein und können auf Basis des neuen Schemas fortgesetzt werden. Hierbei sind unter Umständen noch strukturelle Anpassungen oder Modifikationen der Zustandsmenge notwendig [Rin04, Jur06]. Alle Instanzen, welche nicht zustandsbasiert oder strukturell verträglich sind, sind nicht kompatibel mit dem neuen Schema und sind deshalb *nicht migrierbar*.

2.4.3 Anfragen auf Kollektionen

Wie einleitend erläutert, werden mithilfe von PMS Geschäftsprozesse ausgeführt. Das bedeutet, dass diese in Unternehmen eingesetzt werden, welche eine Organisation ihrer Mitarbeiter in Abteilungen haben. Eine Abteilung ist hierbei meist an der Durchführung verschiedener Prozesse beteiligt. In Abbildung 15 sind exemplarisch die Abteilungen *Beschaffung*, *Produktion* und *Verkauf* dargestellt. Mitarbeiter jeder dieser Abteilungen sind an den Prozessen A, B und C beteiligt.

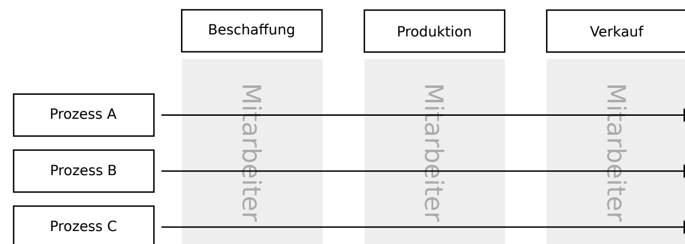


Abbildung 15: Exemplarische Organisation eines Unternehmens

Um die Auslastung einer Abteilung einschätzen zu können, muss das PMS Instanzen abfragen können, welche *auf einem bestimmten Schema beruhen und sich ein bestimmter Knoten hierin in einem bestimmten Zustand* befindet. Diese Anfrage kann dann für die verschiedenen Prozesse und für die verschiedenen Knoten durchgeführt werden, welche von der Abteilung bearbeitet werden. Dadurch kann die Menge der Instanzen ermittelt werden, welche aktuell in der Abteilung abgearbeitet werden. Hiermit kann die Auslastung der Abteilung abgeschätzt werden.

Ebenfalls interessant ist die Auslastung der einzelnen „Prozesslinien“. Hierfür ist eine Abfrage im PMS nötig, welche *alle Instanzen zu einem gegebenen Schema* abrufen kann.

Im Rahmen dieser Arbeit werden diese beiden Anfragen in ihrer Laufzeit genauer untersucht. Diese werden hierbei beispielhaft für andere Anfragen auf Kollektionen herangezogen, welche auf der Datenbasis eines PMS definiert werden können.

2.4.4 Zusammenfassung

In diesem Abschnitt wurden Anwendungsszenarien von PMS und von PMS' zu unterstützende Anfragen untersucht. Zuerst wurde das häufig benötigte Weiterschalten betrachtet. Anschließend wurde die Schemaevolution vorgestellt, gefolgt von zusätzlichen Anfragen auf Kollektionen.

2.5 Allgemeine Speicherbetrachtungen

In dieser Arbeit wird neben der Laufzeit von PMS-typischen Anfragen auch der Speicherverbrauch von verschiedenen Repräsentationen betrachtet und verglichen. In diesem Abschnitt wird zuerst die Speicherhierarchie beschrieben und anschließend werden Vereinbarungen für diese Arbeit gültige Vereinbarungen zum Speicherverbrauch atomarer Größen vorgestellt.

Der Speicher eines Rechners ist in eine Speicherhierarchie [Cle06] aufgeteilt, welche in Abbildung 16 dargestellt ist.

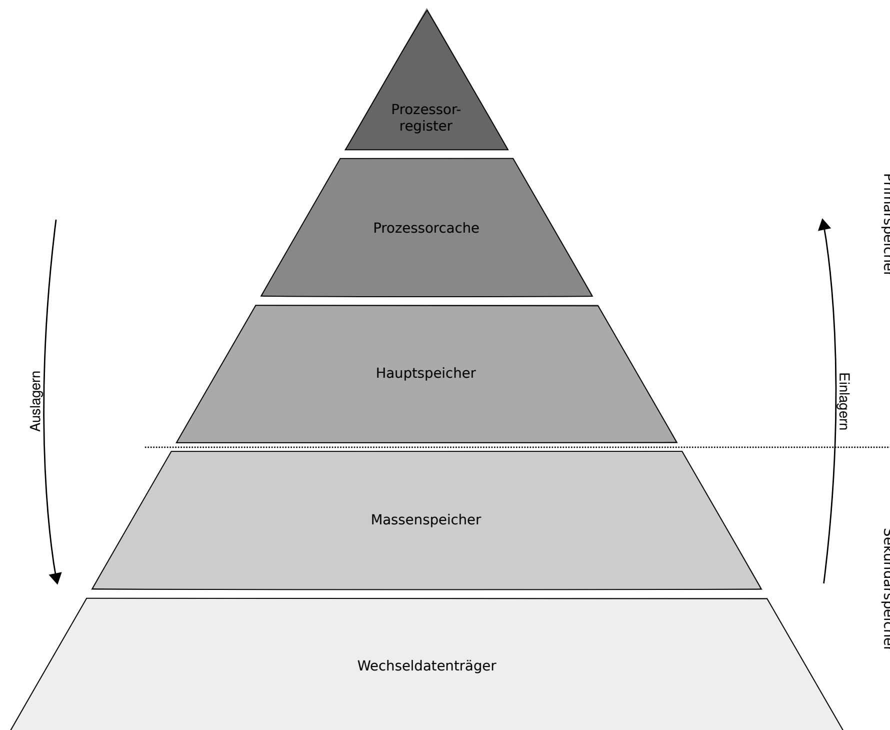


Abbildung 16: Speicherhierarchie

Hierbei bilden die Register des Prozessors die Spitze der Hierarchie, da sie zwar am schnellsten zugreifbar, aber aufgrund ihrer Kosten nur in geringem Umfang in einem Rechner vertreten sind. Es folgen die Prozessorcaches, welche Daten, die vom Prozessor

aus dem Hauptspeicher geladen werden, zwischenspeichern. Der Hauptspeicher bildet die dritte Schicht. Auf Daten, welche in einer dieser Speicherarten gehalten werden, kann gegenüber den restlichen Speicherarten relativ schnell zugegriffen werden. Sie sind allerdings alle flüchtige Speicher, das bedeutet, dass die Daten nach einem Neustart des Rechners nicht mehr vorhanden sind. Die darunter liegenden Schichten von Massenspeicher und Wechseldatenträgern halten die Daten dagegen persistent, wobei allerdings eine wesentlich höhere Zugriffszeit benötigt wird. Wie in Abbildung 16 dargestellt, werden die Schichten der Prozessorregister, -caches und des Hauptspeichers *Primärspeicher* genannt, da diese Speicher vorrangig während der Laufzeit des Rechners genutzt werden. Die beiden unteren Schichten werden *Sekundärspeicher* genannt, da diese während der Laufzeit nur zur Persistierung von Daten benutzt werden. Der Transfer von Daten aus dem Primärspeicher in den Sekundärspeicher nennt man *Auslagern*, die Gegenrichtung *Einlagern*.

Betrachtet man die Daten, welche in diesen Speichern gespeichert werden, so fällt auf, dass im Rahmen dieser Arbeit Vereinbarungen für atomar anzusehende Speicherstrukturen getroffen werden müssen. Die Größen dieser Daten sind grundlegend abhängig von der Architektur des eingesetzten Rechners. Um im Rahmen dieser Arbeit keine unveränderlichen, absoluten Werte für diese Größen angeben zu müssen, werden die Variablen $s_{int}, s_{Objektheader}$ und $s_{Referenz}$ eingeführt. s_{int} stellt dabei den Speicherverbrauch einer Variablen dar, welche einen ganzzahligen Wert (*Integer*) hält. Bei objektorientierten Programmierumgebungen haben Objekte im Allgemeinen einen Objektheader im Speicher. Dies ist zusätzlicher Speicherplatz, welcher zur Organisation des Objekts benötigt wird. Hierin gespeichert ist zum Beispiel die Klasse des Objekts. Die Größe dieses zusätzlichen Speicherverbrauchs hält die Variable $s_{Objektheader}$. $s_{Referenz}$ speichert die Größe, welche eine Referenz auf ein Objekt im Speicher einnimmt. Hierbei ist grundlegend zwischen den heute üblichen 32-Bit- und 64-Bit-Rechnern zu unterscheiden. Bei ersteren haben Referenzen 32 Bit, also 4 Byte, bei letzteren 8 Byte. Um nicht bei jedem Aufkommen eine Fallunterscheidung machen zu müssen und um zukünftige Entwicklungen auf diesem Gebiet in diese Arbeit einzugliedern wird die Variable $s_{Referenz}$ hierfür verwendet.

2.6 Zusammenfassung

Nach der Betrachtung des ADEPT2-Metamodells wurden Repräsentationen der dort definierten Strukturen mithilfe von aktuellen Forschungsergebnissen vorgestellt. Daraufhin wurde die Architektur des ADEPT2-Systems betrachtet und die Konzepte dieser Arbeit darin eingegliedert. Es wurden die Funktionen Weiterschalten, Schemaevolution und Anfragen auf Kollektionen vorgestellt. Abgeschlossen wurde dieses Kapitel durch allgemeine Speicherbetrachtungen, welche dieser Arbeit zugrunde liegen.

3 Konzepte

Aufbauend auf den in Kapitel 2 vorgestellten Grundlagen zeigt dieses Kapitel weiterführende Konzepte der Repräsentation und Verwaltung von Prozessdaten auf. Hierbei wird zuerst eine geeignete Partitionierung von Schemata erarbeitet. Danach wird diese in die bereits vorgestellte Repräsentation von Schemata eingegliedert. Wie zuvor angemerkt, ist die Durchführung einer Schemaevolution sehr zeitintensiv. Dies kann durch die in diesem Kapitel vorgestellte grobe Clusterung vermindert werden. Abschließend werden instanzbasierte Änderungen in Bezug auf die in diesem Kapitel vorgestellten Konzepte untersucht.

3.1 Ansätze zur Partitionierung von Schemata

Die in Abschnitt 2.5 vorgestellte Speicherhierarchie teilt den Speicher eines Rechners in Primär- und Sekundärspeicher. Schemadaten müssen hierbei in beiden repräsentiert werden: Um Funktionen auf den Daten auszuführen werden diese im Primärspeicher benötigt, es ist aber gleichzeitig eine Persistierung der Daten notwendig. Der Primärspeicher ist hierbei in geringeren Größen vorhanden als der Sekundärspeicher. Aufgrund dessen ist es anzustreben, nur benötigte Teile im Primärspeicher zu halten. Es gilt somit eine geeignete Aufteilung der Daten zu finden. Diese ist einerseits für die Menge der Schemata möglich. Wird ein Schema aktuell nicht von Instanzen im Primärspeicher referenziert und auch nicht anderweitig für Funktionen des Systems benötigt, ist es nicht sinnvoll, dieses im Primärspeicher zu halten. Auf der anderen Seite lässt sich eine Aufteilung der Daten der aktuell benötigten Schemata treffen. Hierbei können Teile abgespaltet werden, da nicht immer alle Informationen eines Schemas benötigt werden.

Dies kann mit einer geeigneten Partitionierung erreicht werden. So lassen sich die Knoten- und Kantenmengen *horizontal* partitionieren, in dem man nur die aktuell benötigten Knoten und Kanten einlagert. Eine andere Möglichkeit besteht darin, diese Mengen vollständig im Primärspeicher zu halten, allerdings Daten von den Knoten abzuspalten und diese bei Bedarf nachzuladen. Dies ist die sogenannte *vertikale Partitionierung*.

Bei der *vertikalen Partitionierung*, wie in Abbildung 17 dargestellt, werden alle Knoten und Kanten im Primärspeicher gehalten. Da Knoten einige Attribute besitzen (in der Abbildung sind dies die ID, topologische ID, Split ID und Zweig ID) können diese vertikal partitioniert werden, also die genannten Attribute abgespaltet werden. Diese sind nicht initial im Primärspeicher vorhanden, sondern werden bei Bedarf aus dem Sekundärspeicher nachgeladen. Dies ist sinnvoll, da diese Daten nicht zwangsweise von allen Knoten des Schemas benötigt werden.

Es muss allerdings beachtet werden, dass zusätzlicher Speicher benötigt wird, um die Möglichkeit zur Abspaltung zu schaffen. So benötigt man zum Beispiel bei einer objektorientierten Implementierung der Entitäten zusätzlichen Speicherplatz für eine Referenz auf ein Objekt, welches die eingelagerten Daten hält. Ist dieses Objekt dann instantiiert,

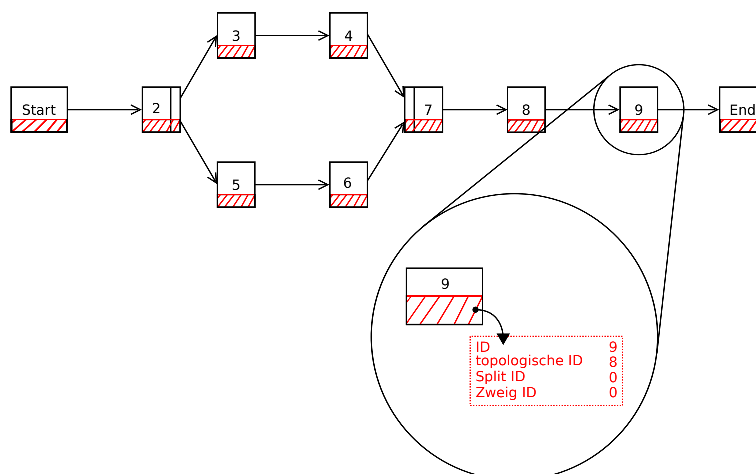


Abbildung 17: Vertikale Partitionierung eines Schemas

sprich die Daten eingelagert, so fällt noch einmal zusätzlicher Speicher an, da ein Objekt einen Objektheader besitzt, in dem unter anderem der Typ des Objekts gespeichert ist. Dieser Aufwand lohnt sich erst bei einer größeren Menge an abzuspaltenden Daten, welche hier allerdings nicht gegeben ist. Die einzelnen Attribute eines Knoten beschränken sich auf vier bis fünf ganzzahlige Werte, wie in Abschnitt 2.2.1 angegeben. Weiterhin ist an eine Verschlechterung der Zugriffszeit auf Daten zu denken, da vor jedem Zugriff überprüft werden muss, ob diese bereits eingelagert sind. Da der hierfür nötige Zeitaufwand allerdings konstant ist, kann er vernachlässigt werden. Sollten die Daten noch nicht eingelagert sein, so muss dies vor dem Zugriff geschehen. Vergleicht man den Gesamtaufwand für das Einlagern aller Daten eines vertikal partitionierten Schemas mit dem eines nicht partitionierten Schemas, stellt sich heraus, dass diese äquivalent sind. Beim nicht partitionierten Schema fällt der gesamte Aufwand beim initialen Einlagern an, während er beim vertikal partitionierten zeitlich verteilt auftritt.

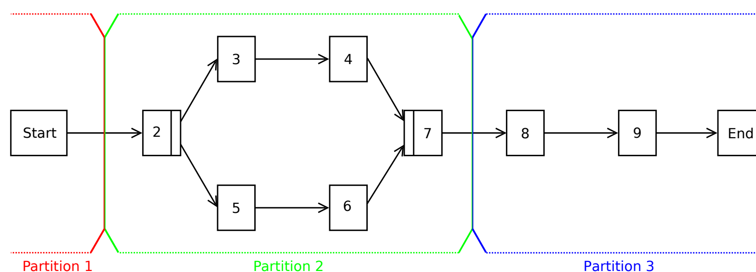


Abbildung 18: Horizontale Partitionierung eines Schemas

Im Gegensatz zur vertikalen Partitionierung werden bei der *horizontalen Partitionierung* keine Daten von den Knoten und Kanten abgespalten. Es werden verschiedene Knoten und Kanten zu Partitionen zusammengefasst, welche dann ein- und ausgelagert werden,

wie in Abbildung 18 illustriert. Dies hat den Vorteil, dass gegenüber einer Realisierung ohne Partitionierung kein zusätzlicher Speicherplatz benötigt wird, sofern die Partitionszugehörigkeit der Kanten und Knoten ohne zusätzliche Strukturen verfügbar ist. Ebenso gilt, dass die Umsetzung der horizontalen Partitionierung im Allgemeinen leichter ist als die der vertikalen: Es muss sichergestellt werden, dass die Partition (als Ganzes) eingelagert ist und nicht, wie bei der vertikalen, vor jedem Zugriff auf Daten des Knoten geprüft werden muss, ob diese eingelagert sind. Die horizontale Partitionierung benötigt allerdings zusätzliche Funktionen des Sekundärspeichers: Sind bei der vertikalen Partitionierung alle Knoten und Kanten im Primärspeicher vorhanden und die Menge dieser somit bekannt, benötigt man bei einer horizontalen Partitionierung eine Abfrage an den Sekundärspeicher, welche Knoten und Kanten verfügbar sind. Dies stellt aber keine Hürde dar, da die als Sekundärspeicher eingesetzten Systeme wie zum Beispiel relationale Datenbanksysteme diese Aufgabe ohne zusätzlichen Aufwand erfüllen können.

Im Rahmen dieser Arbeit wird nur die horizontale Partitionierung weiter betrachtet, da sich die vertikale aufgrund der geringen Datenmengen nicht eignet. Hierbei gibt es verschiedene Ansätze, welche im Folgenden näher untersucht werden. Zuerst wird die horizontale Partitionierung anhand der Bearbeiterzuordnungen der Aktivitäten der Knoten untersucht. Anschließend betrachten wir eine horizontale Partitionierung, welche die hierarchische Blockstruktur des ADEPT2-Metamodells ausnutzt.

3.1.1 Partitionierung anhand Bearbeiterzuordnungen

Ein Ansatz, horizontal zu partitionieren, besteht im Auswerten der Bearbeiterzuordnungen der Aktivitäten von Knoten. Dies basiert auf der Untersuchung der Verteilung ganzer Prozesse auf verschiedenen Workflow-Servern [Bau00]: Sind mehrere Workflow-Server vorhanden, so kann ein ablaufender Prozess auf demjenigen ausgeführt werden, welcher dem Bearbeiter am nächsten ist. Hierdurch werden unter anderem die Kosten von anfallendem Datentransfer minimiert.

Überträgt man diese Idee auf die Partitionierung von Schemata, so ist es sinnvoll, direkt aneinander gereihte Knoten als Partition zusammenzufassen, welche vom selben Mitarbeiter bearbeitet werden. Dies geschieht unter der Annahme, dass der Mitarbeiter mehrere Prozessschritte der selben Instanz direkt hintereinander bearbeitet, sofern er für deren Bearbeitung zuständig ist.

Die Umsetzung gestaltet sich allerdings schwierig, da die Partitionierung erst zur Laufzeit des Prozesses vollzogen werden kann. Dies folgt aus der Möglichkeit, voneinander abhängige Bearbeiterzuordnungen zu definieren, die erst zur Laufzeit auflösbar sind. Beispiele hierfür sind: „diese Aktivität muss vom Bearbeiter des Vorgängerknotens ausgeführt werden“ oder „ein Mitglied der Organisationseinheit x muss diese Aktivität ausführen“. Hieraus folgt, dass nicht nur die Partitionierung erst zur Laufzeit bekannt ist, sondern auch, dass sich die Partitionierungen von Instanzen des gleichen Schemas unterscheiden können. Dies wird in Abbildung 19 illustriert. Hierbei ist zur Modellierzeit nicht zu ent-

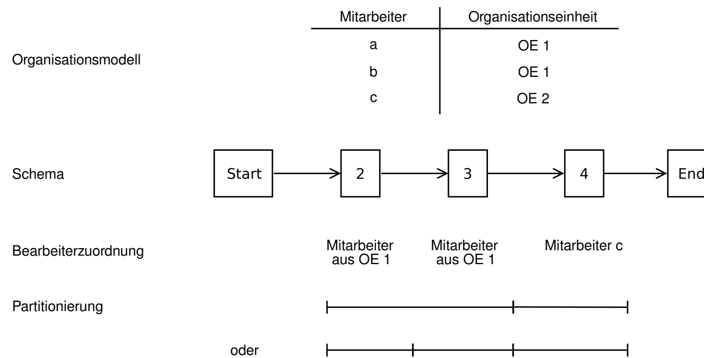


Abbildung 19: Partitionierungsschwierigkeiten bei Bearbeiterzuordnungen

scheiden, ob Knoten 2 und 3 zur selben Partition gehören, da sie jeweils von Mitarbeiter `a` und `b` bearbeitet werden können. Ebenfalls zu beachten ist, dass die Annahme, dass der Bearbeiter direkt folgende Prozessschritte hintereinander ausführt, nicht zwingend erfüllt ist und somit die Möglichkeit besteht, dass die Partitionierung nicht benötigte Knoten einlagert. Dies hängt von weiteren „weichen" Faktoren wie dem Zeitaufwand für das Bearbeiten eines Prozessschritts ab.

Aufgrund dieser Nachteile wird die Partitionierung durch Auswerten von Bearbeiterzuordnungen nicht weiter betrachtet.

3.1.2 Blockstruktur

Eine weitere Möglichkeit zur horizontalen Partitionierung besteht im Ausnutzen der streng hierarchischen Blockstruktur des ADEPT2 Metamodells. Hierbei bildet jeder Block eine Partition. Die hierfür benötigten Informationen sind bereits zur Modellierzeit bekannt. Dadurch entsteht ein Vorteil dieser Partitionierung gegenüber dem Auswerten von Bearbeiterzuordnungen: Bei Ausnutzung der Blockstruktur müssen keine Berechnungen zur Laufzeit durchgeführt werden und die Partitionierung ist für alle Instanzen eines Schemas gleich.

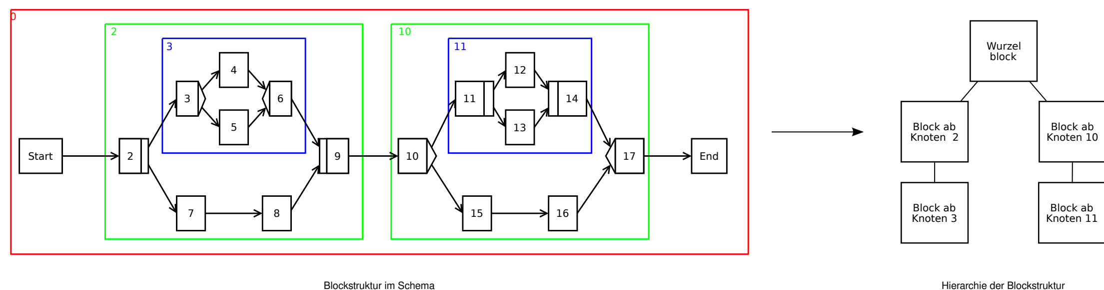


Abbildung 20: Blockstruktur mit Hierarchie

Die in Abschnitt 2.1.2 vorgestellte Blockstruktur wird leicht vereinfacht, um die Anzahl der Partitionen zu verringern. Ein Block beginnt vor einem Split-Knoten, welcher den Block *aufspannt*. Zusätzlich enthält er vorgestellten Blockstruktur alle Knoten und Kanten der untergeordneten Zweige. Diese waren bisher eigenen Blöcken zugeordnet, wie Abbildung 3 gezeigt hat. Diese werden hier zusammengefügt um eine übersichtlichere Anzahl an Partitionen zu erhalten. Der Block wird nach dem Join-Knoten geschlossen, welcher dem Split-Knoten zugeordnet ist. Hierbei kann ein Block Unterblöcke enthalten, sofern sich auf den Zweigen des Split-Knoten weitere Split-Knoten befinden. Jeder Knoten gehört zu genau einem Block. Deshalb gehören Knoten aus untergeordneten Blöcken nicht zu übergeordneten (Knoten 4 in Abbildung 20 gehört nur zu Block 3, nicht zu 2). Für die Knoten auf der Ebene des Start- und Endknotens des Prozesses, welche sich ja auf keinem Zweig eines Split-Knotens befinden, wird ein künstlicher Block eingeführt, der *Wurzelblock*. Dieser wird benötigt, um sicherzustellen, dass jeder Knoten genau einem Block zugeordnet ist und somit die Knoten besser gehandhabt werden können.

3.1.3 Fazit

Nach der Betrachtung verschiedener Ansätze zur Partitionierung von Schemata wurde festgestellt, dass eine vertikale Partitionierung nicht geeignet ist. Das dazu orthogonale Konzept der horizontalen Partitionierung wurde anhand von zwei Varianten untersucht. Hierbei wurde die Partitionierung anhand der Bearbeiterzuordnungen von Knoten verworfen, da diese erst zur Laufzeit bestimmt werden kann und somit zusätzlichen Rechenaufwand zur Laufzeit des Prozesses erfordert. Hieraus folgt auch, dass sich die Partitionierungen von Instanzen des selben Schemas voneinander unterscheiden können. Das Ausnutzen der Blockstruktur für die Partitionierung besitzt hingegen keine Nachteile und ist somit bei Schemata anwendbar.

3.2 Schemata

Die zuvor erarbeiteten Ergebnisse zur Partitionierbarkeit werden in diesem Abschnitt auf die Repräsentation von Schemata angewandt. Dabei existieren verschiedene Varianten, welche ausgearbeitet und qualitativ verglichen werden. Hierfür ist ein qualitativer Vergleich des Speicheraufwands der einzelnen Varianten nötig, wofür zur einheitlichen Handhabung einleitend Vereinbarungen getroffen werden. Ebenso wird die Erweiterung des Algorithmus vorgestellt, welcher die Relation zwischen zwei Knoten ermittelt und häufig benötigt wird [JMSW04].

3.2.1 Speicherbetrachtungen

Dieser Abschnitt leitet in die qualitative Berechnung des Speicherplatzverbrauchs eines Schemas S ein. Hierzu werden zusätzlich zu den in Abschnitt 2.5 definierten Größen s_{int}, $s_{Objektheader}$ und $s_{Referenz}$ folgende Größen herangezogen:

- $n_{Kanten}(S)$ Anzahl der Kanten von S
- $n_{Knoten}(S)$ Anzahl der Knoten von S
- $n_{Split}(S)$ Anzahl der Split-Knoten von S
- Die mit Δ bezeichneten Größen bezeichnen den Speicherplatz, welcher von einer Implementierung zusätzlich benötigt wird. Ein Beispiel hierfür ist der zusätzliche Speicherplatz, der für eine lineare Liste benötigt wird, um eine Menge an Werten zu halten.

Hieraus folgt die Berechnung des Speicherverbrauchs für die Kanten eines Schemas mit der Formel

$$s_{Kanten}(S) = n_{Kanten}(S) \cdot 2 \cdot s_{int} + s_{Kantenattribute}(S) + \Delta_{Kanten}(S)$$

Eine Kante besteht aus zwei ganzzahligen Werten: Der ID des Ursprungknotens und der des Zielknotens. Da ohne zusätzliche Information nicht einzuschätzen ist, wie viele Kantenattribute gespeichert werden, wird die Summe der Größen aller Kantenattribute durch $s_{Kantenattribute}(S)$ repräsentiert.

Für die Knoten eines Schemas S ergibt sich:

$$s_{Knoten}(S) = n_{Knoten}(S) \cdot (4 \cdot s_{int} + s_{Objektheader}) + 2 \cdot n_{Split}(S) \cdot s_{int} + \Delta_{Knoten}(S)$$

Ein Knoten hat nach Abschnitt 2.2 vier ganzzahlige Werte: Die ID und den Typ des Knotens, sowie die Zweig- und Split-Knoten-ID. Split-Knoten besitzen hierbei einen weiteren ganzzahligen Wert: Die ID des zugehörigen Join-Knotens. Für Join-Knoten kommt ebenfalls ein zusätzlicher ganzzahliger Wert hinzu: Die ID des zugehörigen Split-Knotens. Da ein Schema gleich viele Split-Knoten wie Join-Knoten besitzt, kann $2 \cdot n_{Split}(S) \cdot s_{int}$ addiert werden. Hinzu kommt die Objektheader der Knoten-Objekte.

Der Speicherverbrauch eines Schemas S kann mit folgender Formel berechnet werden:

$$s_{Schema}(S) = s_{int} + s_{Kanten}(S) + s_{Knoten}(S) + n_{Knoten}(S) \cdot s_{Referenz} + \Delta_{Schema}(S)$$

Dieses besteht neben der ID des Schemas aus den Kanten und Knoten. Zusätzlich werden Referenzen auf die einzelnen Knotenobjekte benötigt.

3.2.2 Relation zwischen Knoten

Eine häufig für ein Schema benötigte Information ist die Relation zwischen zwei Knoten n_1 und n_2 bezüglich der Kontrollkanten, also die Berechnung der Vorgänger-/Nachfolgerbeziehung von n_1 und n_2. Der Algorithmus `isCPredecessorOf` [JMSW04] berechnet für zwei gegebene Knoten, ob der eine Vorgänger des anderen ist. Dies ist allerdings nicht ausreichend. Es existieren nicht nur die Relationen „Knoten n_1 ist Vorgänger von n_2“ und „Knoten n_1 ist nicht Vorgänger von n_2“, sondern es existieren deren drei:

- n_1 ist Vorgänger von n_2
- n_1 ist Nachfolger von n_2
- n_1 und n_2 stehen in keiner Vorgänger-/Nachfolgerbeziehung. Dies ist beispielhaft in Abbildung 21 dargestellt. Hierbei stehen die Knoten 3 und 6 in keiner Vorgänger-/Nachfolgerbeziehung, da sie sich auf parallelen Zweigen befinden.

Der Algorithmus `getNodeRelation` aus Anhang D.1 berechnet die korrekte Relation zwischen zwei Knoten eines Schemas auf Basis dieser drei Kategorien.

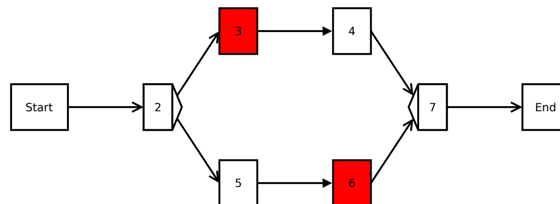


Abbildung 21: Knoten ohne Vorgänger-/Nachfolgerbeziehung

Der vorgestellte Algorithmus arbeitet auf der topologischen Sortierung der Knoten. Diese Ordnung wird abgebildet durch die *topologische ID*. Besitzen die beiden Knoten n_1 und n_2 die selbe topologische ID, so handelt es sich um den Trivialfall: $n_1 = n_2$. Um die restlichen Fälle abzudecken, genügt es nicht, die topologischen IDs zu vergleichen.

Es gibt mindestens einen Weg [DD90] vom Start-Knoten zu einem Knoten n des Schemas, welcher durch Traversieren des Schemagraphen vom Start-Knoten aus errechnet werden kann. Hierbei können mehrere existieren, was durch vorgelagerte Split-Knoten verursacht wird (vergleiche Abbildung 22).

Durch Betrachten dieser Wege für die Knoten n_1 und n_2 kann der Algorithmus die restlichen Fälle abdecken. Sollte der Knoten n_1 auf einem der Wege von Knoten n_2 liegen, ist n_1 Vorgänger von n_2. Ist n_2 auf einem der Wege von n_1, so ist n_1 Nachfolger von n_2. Befindet sich keiner der Knoten auf einem der Wege des anderen, so befinden sich die Knoten auf verschiedenen Zweigen und stehen somit in keiner Vorgänger-/Nachfolgerbeziehung.

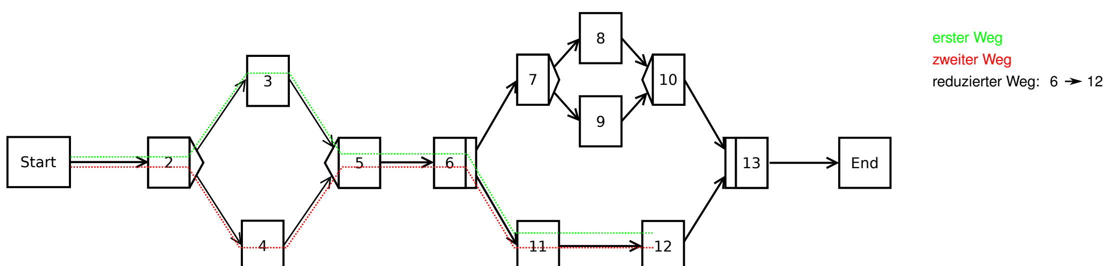


Abbildung 22: Verschiedene Wege vom Start-Knoten zu Knoten 12

Der Algorithmus nutzt aus, dass die Wege ohne Informationsverlust auf die dem Knoten übergeordneten Split-Knoten reduziert werden können [JMSW04]. Der reduzierte Weg für Knoten 12 aus Abbildung 22 ist beispielsweise 6 → 12, da der Split-Knoten 6 der einzige, dem Knoten 12 übergeordnete Split-Knoten ist. Hierbei können die Knoten 2 bis 5 unbeachtet bleiben. Wichtig ist, dass der Split-Knoten, bei welchem „abgezweigt" werden muss, Knoten 6 ist. Hierdurch existiert für einen Knoten nur noch *ein* Weg vom Startknoten aus, welcher *reduzierter Weg* genannt wird. Diese reduzierten Wege der Knoten werden außerdem rückwärts betrachtet (also ausgehend vom Knoten n, nicht vom Start-Knoten), da hier eine weitere Optimierung möglich wird: Es muss nicht der komplette Weg betrachtet werden, sondern nur der bis zum ersten gemeinsamen Split-Knoten [JMSW04]. Im worst-case ist dies der Start-Knoten (welcher hier als Split-Knoten betrachtet wird, da er den äußersten Block aufspannt). Hiernach muss überprüft werden, von welchem Zweig die einzelnen Wege der Knoten n_1 und n_2 kommen. Sind diese nicht identisch, befinden sich die Knoten auf unterschiedlichen Zweigen und stehen somit in keiner Vorgänger-/Nachfolgerbeziehung. Sind sie identisch, so gibt ein Vergleich der topologische IDs die Vorgänger-/Nachfolgerbeziehung an.

Für ein Schema S besitzt der Algorithmus somit eine Laufzeitkomplexität von $\mathcal{O}(n_{Split}(S))$.

3.2.3 Blockrepräsentation

Um die horizontale Partitionierung anhand der Blockstruktur wie in Abschnitt 3.1.2 zu unterstützen, muss die Blockinformation im Schema vorhanden sein. In diesem Abschnitt wird erörtert, welche allgemeinen Vorgehensweisen zu empfehlen und welche speziellen Realisierungen der Blockrepräsentation möglich sind.

Für jeden Block ist ein eindeutiges Identifizierungsmerkmal nötig. Hierfür wäre eine einfach Durchnummerierung der Blöcke denkbar. Dies würde bei einer Umsetzung allerdings weitere Strukturen erfordern, welche die Zugehörigkeit eines Knotens zu seinem Block regeln. Besser ist es, die ID des den Block aufspannenden Split-Knotens als Blockidentifikation zu benutzen. Somit sind keine zusätzlichen Strukturen nötig. Es folgt, dass die Anfrage, zu welchem Block ein Knoten gehört, in konstanter Laufzeit beantwortet werden kann, da die ID des direkt übergeordneten Split-Knotens bereits im Knoten mitgeführt wird.

Die Vorgänger-/Nachfolgerbeziehung zwischen zwei Blöcken b_1 und b_2 kann mithilfe des in Abschnitt 3.2.2 vorgestellten Algorithmus `getNodeRelation` berechnet werden. Die möglichen Relationen zwischen Blöcken sind:

- b_1 ist Vorgänger von b_2
- b_1 ist Nachfolger von b_2
- b_1 und b_2 stehen in keiner Vorgänger-/Nachfolgerrelation
- b_1 ist b_2 untergeordnet
- b_1 ist b_2 übergeordnet

Aufgrund der zwei zusätzlichen Relationen gegenüber denen zwischen Knoten, kann durch *einen* Aufruf von `getNodeRelation` somit kein Ergebnis errechnet werden. Hat man allerdings die letzten beiden Relationen ausgeschlossen, reicht zur Berechnung ein Aufruf von `getNodeRelation` mit den beiden Split-Knoten, welche die Blöcke aufspannen. Somit müssen diese beiden Fälle zuerst überprüft werden. Ist b_1 untergeordnet zu b_2, so befindet sich der aufspannende Split-Knoten von b_2 auf dem reduzierten Weg (vergleiche Abschnitt 3.2.2) vom Start-Knoten zu b_1. Dies gilt analog für b_1.

Ein blockorientiertes Schema muss zusätzliche Anfragen zu den in Abschnitt 2.2 definierten beantworten können. So muss neben der angesprochenen Auflösung eines Knotens zum zugehörigen Block auch die Information verfügbar sein, welche Blöcke im Schema und welche Knoten und Kanten in einem bestimmten Block vorhanden sind. In den Laufzeiten dieser Anfragen unterscheiden sich die verschiedenen Möglichkeiten, die Blockstruktur zu repräsentieren.

Es gibt zwei Varianten zur Darstellung der Blockinformation: Zunächst wird die *explizite Blockrepräsentation* untersucht. Im Anschluss wird die *implizite* betrachtet und ein Vergleich der beiden Varianten gezogen. Hierbei werden, neben dem Speicheraufwand auch die Laufzeiten für die Berechnung der folgenden Mengen untersucht:

- alle Knoten eines Blocks
- alle Kanten eines Blocks
- alle Blöcke des Schemas
- alle Knoten und Kanten des Schemas

3.2.3.1 Explizite Blöcke Die Blockinformation kann explizit dargestellt werden. Dies bedeutet, dass ein Schema aus einer Menge an Blöcken besteht, welche wiederum die Knoten und Kanten beinhalten. Die blockbasierte Partitionierung ist in dieser Repräsentation ohne Anwendung weiterer Algorithmen verfügbar, wie aus Abbildung 23 ersichtlich ist.

[KRRD09] schlägt vor, die *Hierarchie* der Blockstruktur mithilfe leerer Knoten abzubilden. Diese Knoten verweisen auf einen Unterblock, was auf der Behandlung von Subprozessen basiert. Hierbei lässt sich die Hierarchie allerdings nur von oben nach unten, also vom Wurzelblock zu den untergeordneten Blöcken traversieren. Dies ist zum Beispiel dann problematisch, wenn der Nachfolger eines Knotens bestimmt werden soll. Wird diese Anfrage für Knoten 6 aus Abbildung 24 gestellt, so muss zuerst der Block des Knotens bestimmt werden. Dieser wird daraufhin untersucht, ob der Nachfolger des Knotens durch Untersuchen der Kantenmenge bestimmt werden kann. Dies ist für Knoten 6 nicht möglich, da dies der topologisch letzte Knoten im Block ist. Daraufhin muss der übergeordnete Block und darin der Nachfolger des „leeren Knotens“ bestimmt werden. Dies ist sehr aufwändig und benötigt zur Speicherung der leeren Knoten unnötig Speicherplatz. Abhilfe schafft die zweite in Abbildung 24 dargestellte Möglichkeit zur Repräsentation. Es werden innerhalb von Blöcken keine leeren Knoten zur Abbildung der Hierarchie benutzt. Vielmehr werden in einem Block auch diejenigen Kanten gespeichert, welche entweder als

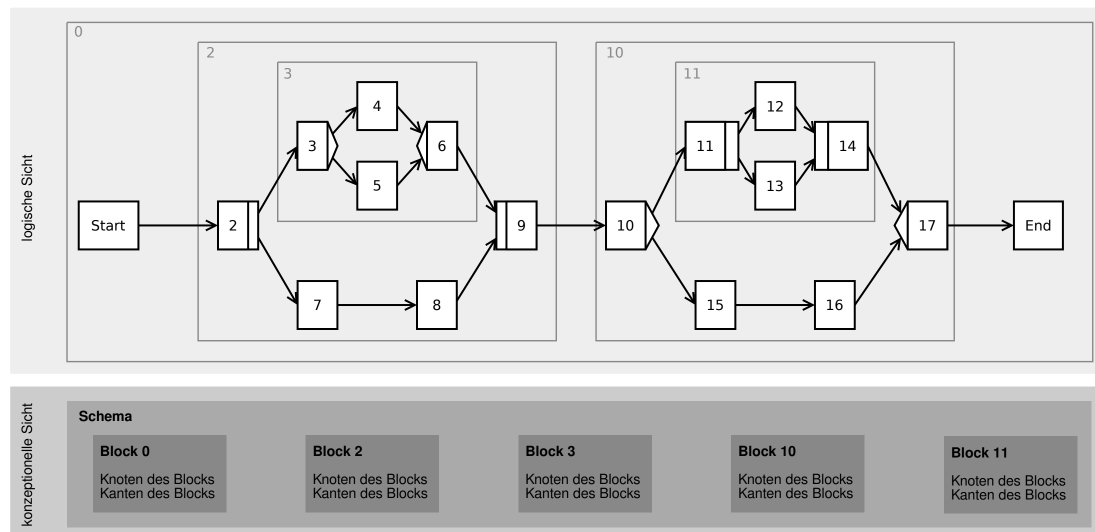


Abbildung 23: Schemarepräsentation mit expliziter Blockstruktur

Quell- oder Ziel-Knoten-ID einen Knoten referenzieren, der außerhalb des Blocks liegt. So kann der Vorgänger und Nachfolger eines Knotens jeweils direkt bestimmt werden. Um die Hierarchie abzubilden, wird eine weitere Methode benötigt, welche zu einem gegebenen Schema und einem Knoten von diesem die Block-ID berechnet, zu der dieser Knoten gehört. Ist also das Ziel einer Kante ein Knoten, welcher nicht im Block gespeichert ist, so kann der nachfolgende Block mithilfe dieser Methode bestimmt werden. Die Hierarchie der Blockstruktur ergibt sich somit implizit.

Die Menge der Knoten, welche sich in einem Block befinden, lässt sich bei einer expliziten Blockrepräsentation in konstanter Zeit beantworten, da diese Menge bereits innerhalb des Blocks vorliegt. Gleiches gilt für die Menge an Kanten eines Blocks.

Die Information, welche Blöcke in einem Schema vorhanden sind, kann man bei der expliziten Blockrepräsentation ebenfalls in konstanter Laufzeit bestimmen, da die Blockmenge direkt im Schema verfügbar ist.

Um alle Knoten oder Kanten des Schemas zu berechnen, fällt linearer Aufwand auf der Anzahl der Knoten an, da die einzelnen Blöcke durchlaufen und die jeweiligen Knoten- bzw. Kantenmengen vereinigt werden müssen. Hierbei liegt die Anzahl der Kanten ebenfalls in $\mathcal{O}(n_{Knoten}(S))$, deshalb ist auch hierbei der Aufwand linear zur Anzahl der Knoten.

Den Speicherplatzaufwand für ein Schema, welches durch explizite Blöcke dargestellt ist, erhält man durch die Formel

$$\begin{aligned} s^{explizit}_{Schema}(S) &= \sum_{Block\ b} [s_{Referenz} + s_{Objektheader} + s_{int} + s_{Kanten}(b) + s_{Knoten}(b)] \\ &\quad + s_{int} + \Delta^{explizit}_{Schema}(S) \end{aligned}$$

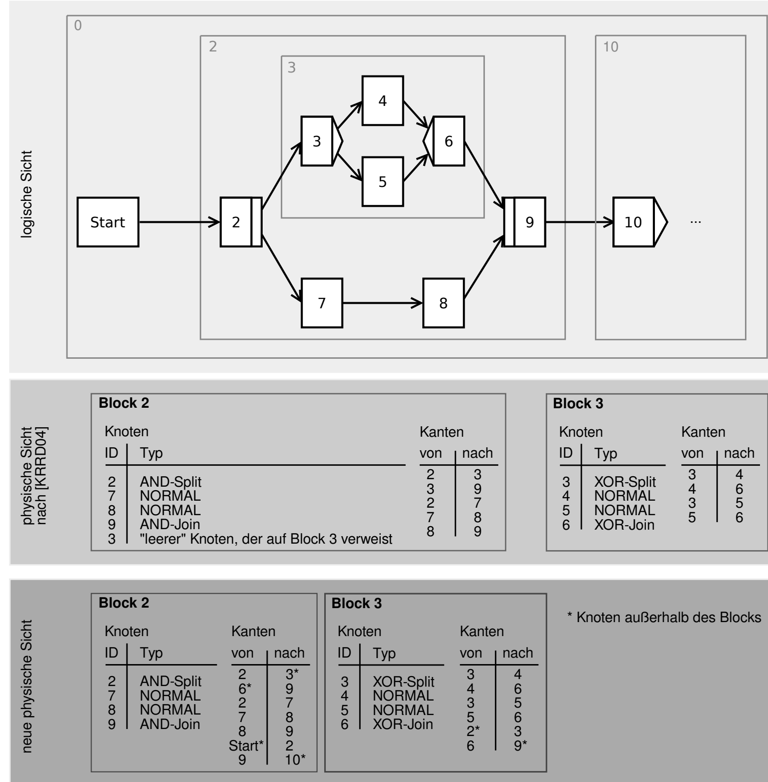


Abbildung 24: Hierarchiedarstellung bei expliziter Blockstruktur

Für jeden Block wird hierbei eine Referenz auf das Block-Objekt benötigt. Dieses besteht aus einem Objektheader, der Block-ID und den Kanten und Knoten. Hinzu kommt die ID des Schemas und implementierungsabhängige Speicherstrukturen. $s_{Knoten}(Block)$ wird auf die gleiche Weise berechnet wird wie $s_{Knoten}(Schema)$, wobei nur die Knoten, welche zum angegebenen Block gehören in die Berechnung einbezogen werden.

$s_{Kanten}(Block)$ ist folgendermaßen definiert:

$$\begin{aligned} s_{Kanten}(B) &= (n_{Kanten}(B) + n_{eingehende\ Kanten}(B) + n_{ausgehende\ Kanten}(B)) \\ &\quad \cdot 2 \cdot s_{int} + \Delta_{Kanten}(B) \end{aligned}$$

Hierbei müssen wie bereits erwähnt auch die ein- und ausgehenden Kanten des Blocks betrachtet werden.

Im Vergleich zu einem nicht blockorientiert gespeicherten Schema (vergleiche Abschnitt 2.2) ergibt sich somit einen Speicherplatzmehraufwand von

$$\Delta^{explizit}_{Schema}(S) + \sum_{Block\ b} [s_{Referenz} + s_{Objektheader} + s_{int} + 2 \cdot s_{int} \cdot n_{ausgehende\ Kanten}(b)]$$

Dieser befindet sich in der Komplexitätsklasse $\mathcal{O}(n_{Split}(S))$, da die Anzahl der Blöcke direkt von der Anzahl der Split-Knoten abhängt.

3.2.3.2 Implizite Blöcke Neben der expliziten Blockrepräsentation lassen sich diese auch implizit darstellen, da bereits jeder Knoten die Information beinhaltet, zu welchem Block er gehört (Split-Knoten-ID, siehe Abbildung 6). Aufgrund dessen ist es nicht notwendig, die Blockstruktur explizit zu realisieren, sie kann implizit vorgehalten werden. Hierdurch entsteht kein zusätzlicher Speicherplatzaufwand, da nur eine flache Knotenmenge gehalten wird.

Es ist allerdings ein linearer Zeitaufwand nötig, um die Menge der Knoten zu berechnen, welche zu einem bestimmten Block gehören, da alle Knoten auf ihre Blockzugehörigkeit untersucht werden müssen. Gleiches gilt für die Information, welche Blöcke im Schema vorhanden sind. Um herauszufinden, welche Kanten zu einem Block gehören, müssen alle Kanten untersucht werden. Es fällt somit ein linearer Aufwand an. Da die Menge aller Knoten und aller Kanten des Schemas bereits vorliegt, fällt hierfür konstanter Zeitaufwand an.

Der Speicherplatzaufwand für ein implizites Schema S wird berechnet mit

$$s_{Schema}^{implizit}(S) = s_{int} + s_{Knoten}(S) + s_{Kanten}(S) + n_{Knoten}(S) \cdot s_{Referenz} + \Delta_{Schema}^{implizit}(S)$$

Da $\Delta_{Schema}^{implizit}(S) = \Delta_{Schema}(S)$ gilt, fällt kein zusätzlicher Speicherplatz gegenüber einem nicht blockorientiert gespeicherten Schema an.

$$s_{Schema}^{implizit}(S) = s_{Schema}(S)$$

3.2.3.3 Fazit Es ist zu erkennen, dass beide Realisierungsmöglichkeiten Vorteile bieten, welche in Tabelle 1 zusammengefasst sind. Es bleibt zu untersuchen, welche quantitativen Unterschiede sich in Speicherverbrauch und in der Laufzeit zeigen.

	Explizite Blöcke	*Implizite Blöcke*
zusätzlicher Speicherplatz	$\mathcal{O}(n_{Split}(S))$	0
Blockmenge	$\mathcal{O}(1)$	$\mathcal{O}(n_{Knoten}(S))$
Knoten → Block	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Block → Knotenmenge	$\mathcal{O}(1)$	$\mathcal{O}(n_{Knoten}(S))$
Block → Kantenmenge	$\mathcal{O}(1)$	$\mathcal{O}(n_{Kanten}(S))$
Knoten-/Kantenmenge	$\mathcal{O}(n_{Knoten}(S))$	$\mathcal{O}(1)$

Tabelle 1: Explizite und implizite Realisierung von Blöcken

3.3 Grobe Clusterung

Neben der expliziten und impliziten Markierung besteht auch die Möglichkeit, einen Instanzzustand mithilfe der Clusterung abzubilden (vergleiche Abschnitt 2.1.3). Diese ist für eine Schemaevolution vorteilhaft, da die zustandsbasierte Verträglichkeit nur einmal überprüft werden muss und für ganze Cluster bestimmt werden kann, ob die referenzierten Instanzen migrierbar oder nicht migrierbar sind. Es muss nicht jede Instanz separat untersucht werden. In [KRRD09] wurde allerdings festgestellt, dass die Clusterung aufgrund des hohen Speicherbedarfs und der großen Anzahl an Clustern bei Parallelverzweigungen nicht einsetzbar ist. In diesem Abschnitt wird die Idee der Clusterung hin zu einer speicherplatzeffizienten Optimierung der Schemaevolution entwickelt, die so genannte *grobe Clusterung*.

Hierbei wird die in Abschnitt 3.1.2 vorgestellte Partitionierung anhand der Blockstruktur eines Schemas ausgenutzt. Für jeden Block des Schemas existiert ein Cluster. Ein Cluster referenziert diejenigen Instanzen, welche Knoten mit dem Zustand RUNNING des entsprechenden Blocks beinhalten. Der Cluster ist somit ein grober Anhaltspunkt, wo eine Instanz in ihrer Ausführung steht. Die grobe Clusterung ist im Vergleich zur (genauen) Clusterung aus Abschnitt 2.1.3 somit ebenfalls zustandsbasiert, repräsentiert allerdings keine vollständige Instanzmarkierung.

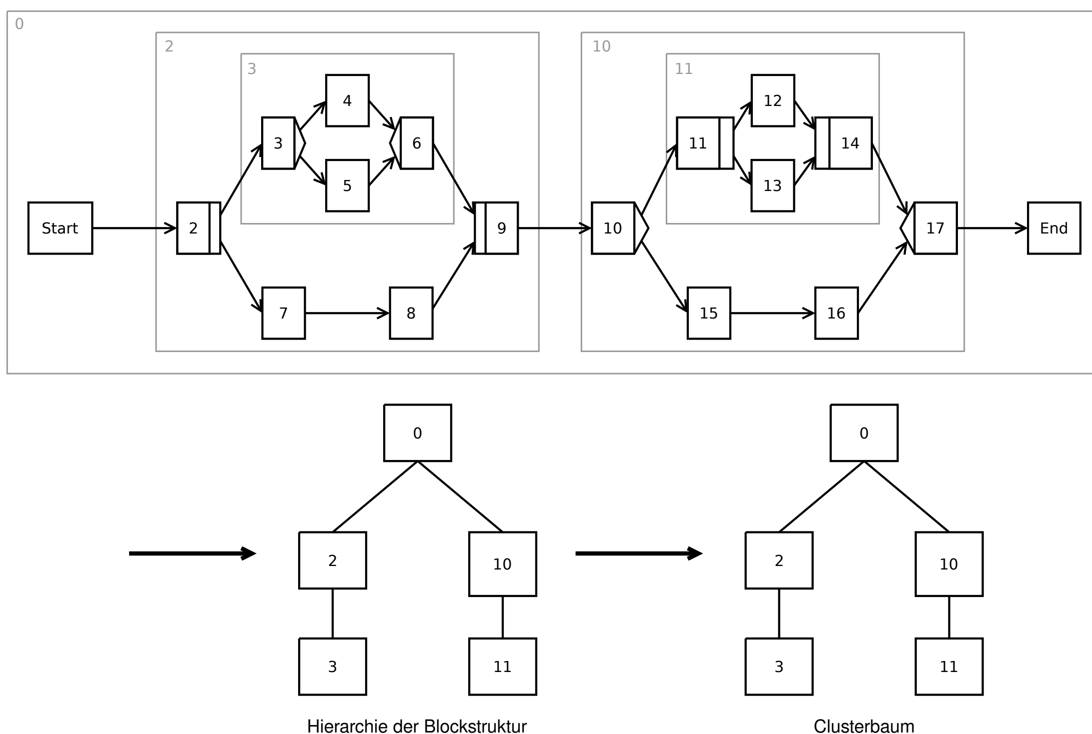


Abbildung 25: Hierarchie der Blockstruktur und Clusterbaum

Es entsteht analog zur Hierarchie der Blockstruktur ein Clusterbaum, welcher aus der hierarchischen Anordnung der Cluster besteht (vergleiche Abbildung 25). Dieser Clusterbaum ist eindeutig für ein Schema und lässt sich zur Modellierzeit berechnen. Die exponentielle Anzahl an Clustern, welche durch Parallelverzweigungen entstehen und welche in [KRRD09] zur Verwerfung des Ansatzes geführt haben, treten hierbei nicht auf. Dies ist in der Blockstruktur des Schemas begründet: Es gibt nicht für jeden Zweig eines Split-Knotens einen eigenen Cluster, sondern für den Split-Knoten mitsamt seiner Zweige genau einen. Hierdurch potenziert sich die Anzahl der Cluster bei Parallelverzweigungen nicht mehr, sie reduziert sich auf $n_{Split}(S) + 1$. Aufgrund dessen ist es möglich, den Clusterbaum eines Schemas im Primärspeicher zu halten.

Es lässt sich nicht nur im Clusterbaum eine Analogie zu den für Blöcke angestrengten Betrachtungen ziehen. Auch die Vorgänger-/Nachfolgerbeziehung zwischen Clustern lässt sich analog zu den der Blöcke definieren. Ein Cluster c_1 ist also Vorgänger eines Clusters c_2, sofern der Block von c_1 Vorgänger des Blocks von c_2 ist. Dies gilt analog für die übrigen Relationen zwischen Blöcken.

Das Beispiel aus Abbildung 26 veranschaulicht, wie eine Instanz durch diesen Clusterbaum „wandert". Bei der Instanz befindet sich als erstes der Start-Knoten im Zustand RUNNING. Da dieser Knoten zum Wurzelblock gehört, wird diese Instanz vom Wurzelcluster referenziert. Durch Weiterschalten wird der Start-Knoten COMPLETED und Knoten 2 RUNNING. Da sich nun nur im Block 2 Knoten im Zustand RUNNING befinden, wandert die Instanz im Clusterbaum nach unten: Nur der Cluster 2 referenziert die Instanz. Nach dem Weiterschalten des AND-Split-Knotens befinden sich nun in zwei verschiedenen Blöcken Knoten im Zustand RUNNING, somit wird auch die Instanz von den Clustern 2 und 3 referenziert. Die Zugehörigkeit einer Instanz zu mehreren Clustern ist somit möglich. Zum nächsten Schritt im Beispiel wurde mehrfach weitergeschaltet. Der untere Zweig des AND-Splits wurde vollständig abgearbeitet. Der AND-Join-Knoten ist allerdings noch nicht aktiviert, da sich ein Knoten auf dem oberen Zweig noch in Ausführung befindet. Dieser ist in Block 3, somit wird die Instanz nur noch von Cluster 3 referenziert. Durch weiteres, mehrmaliges Weiterschalten wird schließlich der AND-Join-Knoten in den Zustand RUNNING versetzt. Hierdurch wandert die Instanz im Clusterbaum wieder nach oben in den Cluster 2.

Die Änderung der Clusterzugehörigkeit kann hierbei durch den Algorithmus zum Weiterschalten berechnet werden. Dieser kennt den Block des aktiven Knotens vor dem Weiterschalten und weiß, zu welchem Knoten weitergeschaltet wird. Diese Information kann zur effizienten Berechnung der (neuen) Clusterzugehörigkeit benutzt werden.

3.3.1 Schemaevolution

Bei einer Schemaevolution wird das Schema S an einer bestimmten Stelle x geändert. Nach [Rin04, Jur06] müssen alle Instanzen von S in den Primärspeicher eingelagert und einzeln untersucht werden (vergleiche Abschnitt 2.4). Dies ist sehr aufwändig. Durch den

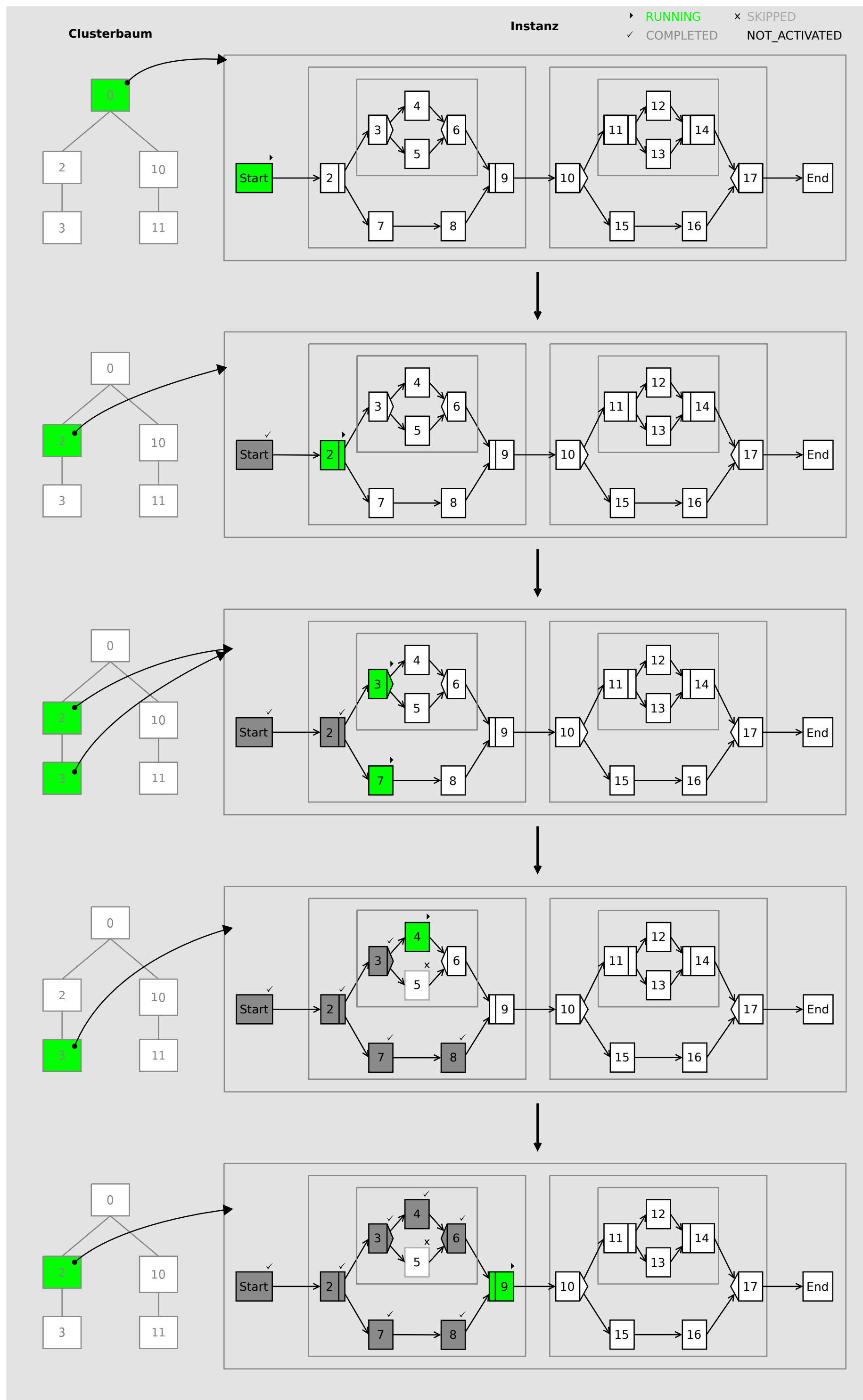


Abbildung 26: Beispiel für die grobe Clusterung

Einsatz der groben Clusterung lässt sich dies allerdings optimieren. Mit ihr ist es möglich, die zu untersuchende (und damit einzulagernde) Menge an Instanzen zu verringern, da leicht festgestellt werden kann, welche Instanzen sich in ihrer Ausführung zwingend *vor* der geänderten Stelle x und welche sich zwingend *nach* x befinden. Erstere sind migrierbare Instanzen, letztere können nicht migriert werden (vergleiche Abbildung 27). Hierbei ist es nicht nötig, diese Instanzen aus dem Sekundärspeicher einzulagern, es genügt, die Clusterzugehörigkeit zu untersuchen. Übrig bleibt noch eine Menge an Instanzen, welche sich in der Ausführung nahe an der Stelle x befinden. Diese werden dann mit den in [Rin04, Jur06] vorgestellten Methoden untersucht. Danach kann eine Migration der Instanzen wie in Abschnitt 2.4 beschrieben durchgeführt werden.

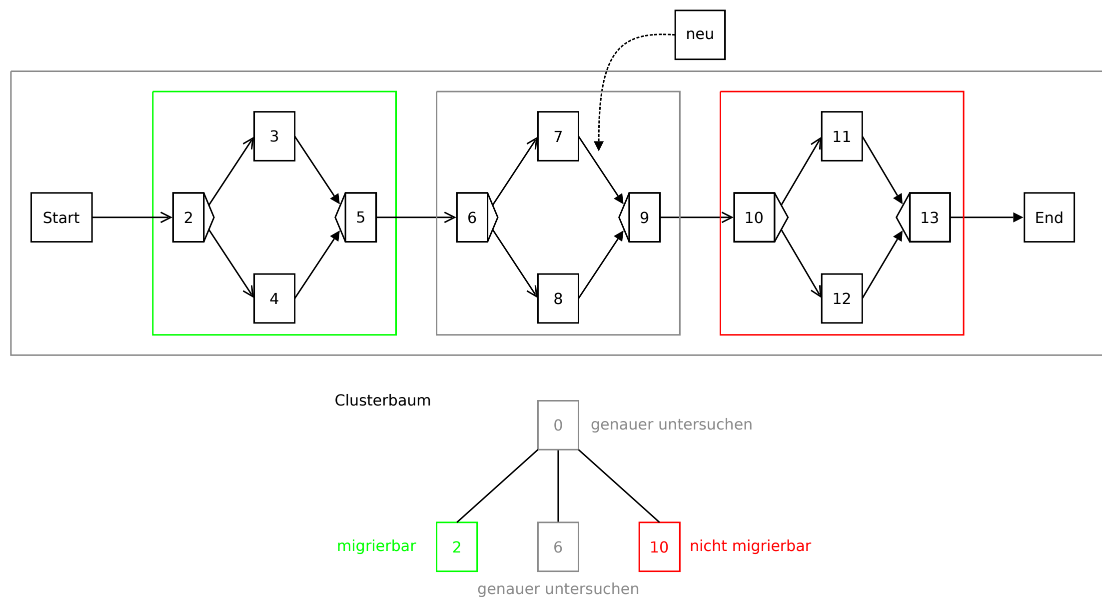


Abbildung 27: Beispiel einer Schemaevolution bei grober Clusterung

Der im Folgenden vorgestellte Algorithmus `preEvolveSelection` teilt die Instanzen durch Auswerten des Clusterbaums in folgende drei Mengen ein:

- Die Instanz ist migrierbar
- Die Instanz ist nicht migrierbar
- Die Instanz muss mithilfe der Methoden aus [Rin04, Jur06] genauer untersucht werden

Hierfür ist ein weiterer Algorithmus, `findPositions`, nötig, der die im Schema geänderten *Stellen* findet. Nach dessen Vorstellung folgt `preEvolveSelection`.

3.3.1.1 Stellen Die Änderungen eines Schemas können zu *Stellen* zusammengefasst werden. Dies berechnet der hier vorgestellte Algorithmus `findPositions` (siehe An-

hang D.2). Hierbei bildet jeder Block, der dem Wurzelblock direkt untergeordnet ist eine potentielle Stelle. Diese existiert dann, wenn der Block selbst oder einer seiner direkten oder indirekten Unterblöcke Änderungen enthält. Diese Änderungen sind dann dieser Stelle zugeordnet, wie es in Abbildung 28 dargestellt ist. In der Abbildung ist auch der in Block 3 hinzugefügte Knoten der Stelle 2 zugeordnet. Für die Berechnung hiervon genügt ein Verfolgen des reduzierten Weges (vergleiche Abschnitt 3.2.2) vom Start-Knoten zum geänderten Knoten. Der zuerst besuchte Split-Knoten spannt den direkt dem Wurzelblock untergeordneten Block der Stelle auf. Für Änderungen im Wurzelblock existiert keine Stelle. Das ist möglich, da diese für `preEvolveSelection` einen Spezialfall darstellen. Sie werden in `findPositions` ignoriert.

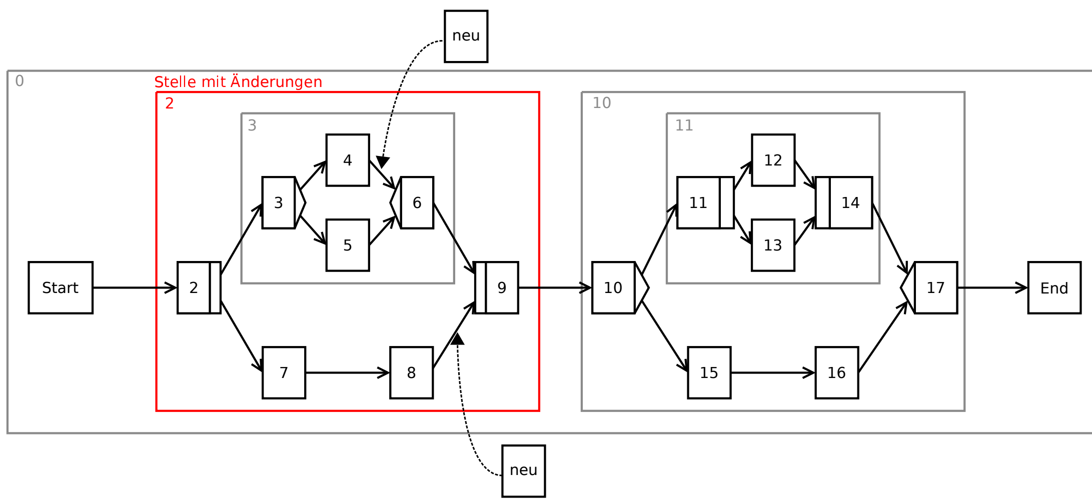


Abbildung 28: Zuordnung von Änderungen zu einer Stelle

Ziel des Algorithmus ist nicht nur die Bestimmung der Stellen, sondern auch festzustellen, ob eine Instanz diese Stelle in der Ausführung passiert haben kann, *ohne* einen geänderten Knoten ausgeführt zu haben. Ein *geänderter Knoten* ist hierbei das Ziel einer gelöschten oder hinzugefügten Kante [Jur06].

Dies wird über den so genannten *Stellentyp* abgebildet, welcher jeder Stelle zugeordnet ist. Dieser ist entweder *LEAKY*, wenn eine Instanz diese Stelle passiert haben kann, *ohne* einen geänderten Knoten ausgeführt zu haben, oder im gegensätzlichen Fall *COMPACT*. Somit ist die markierte Stelle der Instanz des Schemas 1 in Abbildung 29 *COMPACT*. Die Stelle der anderen dargestellten Instanz ist allerdings *LEAKY*. Durch die grobe Clusterung stehen nur Informationen zu den Knoten zu Verfügung, welche aktuell RUNNING sind. Überträgt man dies auf die Instanz des Schemas 2, so ist nur bekannt, dass Knoten 6 RUNNING ist. Auf Basis dieser Information ist nicht entscheidbar, ob die Instanz den geänderten Zweig des Split-Knotens ausgeführt hat. Dies wird durch den Stellentyp *LEAKY* verdeutlicht.

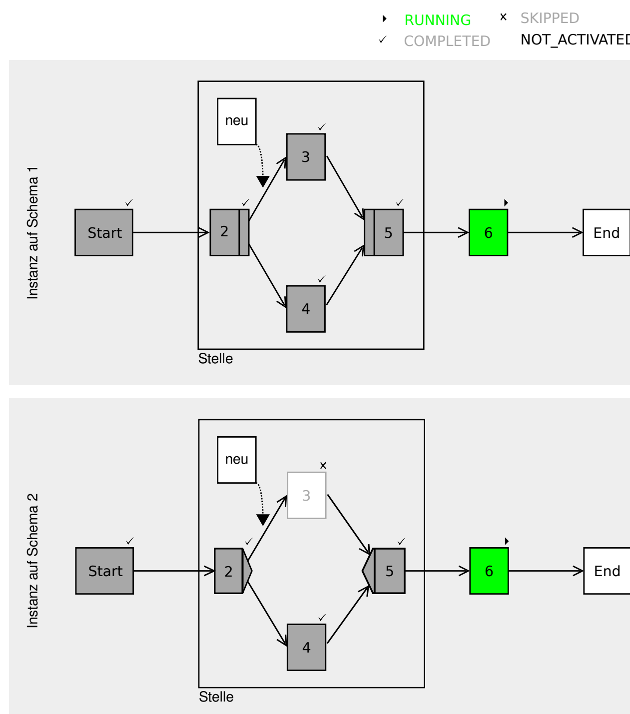


Abbildung 29: COMPACT und LEAKY

Für die Unterscheidung ob eine Stelle *LEAKY* oder *COMPACT* ist, müssen die reduzierten Wege aller Änderungen einer Stelle zum Start-Knoten betrachtet werden. Diese werden durch Einträge in der Menge `splitBranches` repräsentiert. Die Einträge sind Tripel der Form $\{s, b, n\}$. Hierbei ist s ein Split-Knoten auf einem der reduzierten Wege. Verfolgt man von diesem ausgehend den Zweig b, so ist der nächste Knoten auf dem reduzierten Weg der Knoten n. Dieser ist entweder wiederum ein Split-Knoten (für den ein eigener Eintrag in `splitBranches` existiert) oder es ist ein geänderter Knoten. Hierdurch sind die reduzierten Wege aller Änderungen verfügbar. Die rekursive Methode `isFull` überprüft auf Basis dieser Menge und einer gegebenen ID eines Split-Knotens, ob der von diesem aufgespannte Block durchlaufen werden kann, ohne einen geänderten Knoten ausgeführt zu haben. Hierfür untersucht `isFull` zuerst den Typ des Split-Knotens. Ist dies ein AND-Split-Knoten so genügt es, wenn sich auf *einem* Zweig des Split-Knotens eine unpassierbare Änderung befindet (also `isFull` für diese Änderung „wahr" als Ergebnis liefert). Ist der von `isFull` zu untersuchende Knoten ein XOR-Split-Knoten, so müssen auf *allen* Zweigen unpassierbare Änderungen sein. Im gegensätzlichen Fall ist der Block passierbar, ohne eine Änderung ausgeführt haben zu müssen. Folgerichtig liefert `isFull` in diesem Fall „falsch" zurück.

Um den Typ einer Stelle zu berechnen, genügt es, `isFull` mit dem Split-Knoten aufzurufen, welcher den Block der Stelle aufspannt. Wird „wahr" zurückgeliefert, so ist die Stelle *COMPACT*, andernfalls *LEAKY*.

3.3.1.2 Kategorisierung der Instanzen Aufbauend auf dem vorgestellten Algorithmus `findPositions` ist es möglich, Instanzen anhand ihrer Clusterzugehörigkeit in die oben genannten Kategorien einzuteilen. Der Algorithmus hierfür, `preEvolveSelection` (siehe Anhang D.3) wird in diesem Abschnitt vorgestellt.

Die Zuordnung der Änderung zu Stellen wird von `findPositions` vorgenommen. Hierbei werden die Änderungen im Wurzelblock ignoriert, da diese keiner Stelle zuzuordnen sind. Diese werden als Spezialfall behandelt. Hierbei gilt, dass bei einer Änderung im Wurzelblock alle Instanzen als nicht-migrierbar einzuordnen sind, welche von Clustern referenziert werden, die der Änderung nachgelagert sind. In Abbildung 30 ist dies illustriert. Hierbei müssen Instanzen aus dem Cluster 7 als nicht migrierbar eingestuft werden, da sie der Änderung nachgelagert sind. Für Instanzen, welche sich im Wurzelcluster befinden gilt in jedem Fall, dass diese mit den Methoden aus [Rin04, Jur06] genauer untersucht werden müssen.

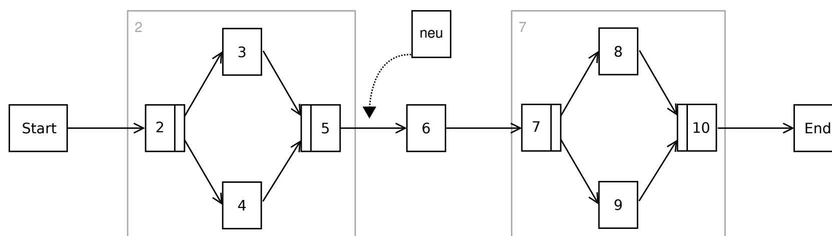


Abbildung 30: Änderung im Wurzelblock

Die Änderungen aus allen anderen Blöcken wurden von `findPositions` zu Stellen zugeordnet. Diese Stellen befinden sich ebenfalls in einer topologischen Sortierung, die durch die topologischen IDs der Split-Knoten gegeben ist, welche die Blöcke der Stellen aufspannen. Somit kann die erste von Instanzen durchlaufene Stelle vom Typ *COMPACT* mithilfe des Algorithmus `getNodeRelation` berechnet werden. Alle Instanzen, welche von Clustern referenziert werden, die dieser Stelle folgen, müssen als nicht migrierbar eingestuft werden. Diese haben zwangsweise einen geänderten Knoten der *COMPACT* Stelle durchlaufen.

Instanzen, welche von einem Cluster referenziert werden, der einer *LEAKY* Stelle folgt, haben nicht zwangsweise einen geänderten Knoten der Stelle ausgeführt. Die Entscheidung, ob diese Instanzen migrierbar sind oder nicht, kann nicht mithilfe der groben Clusterung getroffen werden. Somit müssen diese Instanzen mit den Methoden aus [Rin04, Jur06] genauer untersucht werden. Hierbei gibt es keinen Sinn, eine bereits als nicht migrierbar eingestufte Instanz als genauer zu untersuchen einzustufen. Sollte der topologisch ersten *LEAKY* Stelle also eine *COMPACT* Stelle folgen, müssen Instanzen aus Clustern, welche der *COMPACT* Stelle nachgeordnet sind, als nicht migrierbar eingestuft werden.

Bisher wurden Instanzen eingestuft, welche von Clustern referenziert werden, die einer Stelle *folgen*. Instanzen, die sich in ihrer Ausführung *vor* der topologisch ersten Stelle

befinden, sind migrierbar. Es ist sichergestellt, dass diese noch keinen geänderten Knoten ausgeführt haben.

Nicht betrachtet wurden bisher Instanzen, welche sich in der Ausführung in einem Block befinden, welcher der (topologisch) ersten Stelle zugeordnet ist. Diese Instanzen könnten alle mithilfe von [Rin04, Jur06] genauer untersucht werden. Es ist allerdings eine genauere Unterscheidung aufgrund der groben Clusterung möglich. Es werden alle geänderten Knoten der (topologisch) ersten Stelle untersucht. Instanzen, welche sich im *Cluster eines geänderten Knotens* befinden, müssen genauer untersucht werden, da sich nicht herausfinden lässt, ob diese den geänderten Knoten bereits ausgeführt haben. Für *untergeordnete Cluster* kann untersucht werden, ob der Block der Änderung nachgelagert ist. Sollte dies so sein, sind alle Instanzen des entsprechenden Clusters und dessen Untercluster nicht migrierbar. Befindet sich der untergeordnete Cluster auf einem anderen Zweig als die Änderung und wird der Cluster von einem AND-Split-Knoten aufgespannt, müssen die Instanzen des Clusters (und dessen untergeordneten) mit den Methoden aus [Rin04, Jur06] genauer untersucht werden. Sollte der Cluster von einem XOR-Split aufgespannt werden, ist dies nicht notwendig, da eine Instanz, welche von diesem Untercluster referenziert wird, den Zweig des Unterclusters und somit nicht den Zweig mit der Änderung ausgeführt hat. Instanzen aus einem Untercluster, welcher der Änderung vorgelagert ist, sind ebenfalls weiterhin potentiell migrierbar. Zusätzlich müssen *alle anderen Cluster* der Stelle untersucht werden. Ist einer der Cluster der Änderung nicht vorgelagert, müssen die Instanzen hierin mithilfe [Rin04, Jur06] genauer untersucht werden.

Da diese Berechnungen für jede Änderung der (topologisch) ersten Stelle durchgeführt werden, ist es möglich, dass Instanzen mehrfach bewertet wurden. Aufgrund dessen ist es notwendig, die Mengen anschließend zu bereinigen. Hierbei erfolgt eine Priorisierung der Kategorien:

1. „nicht migrierbar“
2. „genauer untersuchen“
3. „migrierbar“

Das heißt, ist eine Instanz in der ersten Kategorie, muss sie aus den Instanzmengen der anderen Kategorien entfernt werden. Wurde eine Instanz in die zweite Kategorie eingeordnet, muss sie aus der Instanzmenge der dritten entfernt werden. Nach der Überprüfung der Instanzen der zweiten Kategorie nach [Rin04, Jur06] kann die Migration wie in Abschnitt 2.4 fortgesetzt werden.

3.3.2 Fazit

In diesem Abschnitt wurde die grobe Clusterung vorgestellt, welche zur Optimierung des Laufzeitaufwands bei einer Schemaevolution herangezogen werden kann. Mithilfe der groben Clusterung ist es möglich, eine Einschätzung zu treffen, wo eine Instanz in ihrem Ablauf in etwa steht. Es wurden zwei Algorithmen vorgestellt, welche in Kombination eine Einteilung der Instanzen vor einer Schemaevolution in migrierbare und nicht

migrierbare Instanzen, sowie in Instanzen vornehmen, welche nach den Methoden aus [Rin04, Jur06] genauer untersucht werden müssen. Dadurch müssen nicht mehr alle Instanzen eines Schemas bei einer Schemaevolution einzeln untersucht und hierfür aus dem Sekundärspeicher eingelagert werden.

3.4 Instanzbasierte Änderungen

Wie in Abschnitt 2.2 bereits erwähnt, unterstützt das ADEPT2-Metamodell instanzbasierte Änderungen. In diesem Abschnitt wird allerdings aufgezeigt, dass die Realisierung des Konzepts in der bisherigen Form [KR10] nicht ausreichend ist. Dies wird daraufhin entsprechend erweitert. Anschließend werden Definitionen zur Speicherbetrachtung der Deltaschicht aufgezeigt und mithilfe dieser die verschiedenen Repräsentationen erläutert. Abschließend werden instanzbasierte Änderungen in Bezug auf die grobe Clusterung untersucht.

3.4.1 Erweiterung des Deltaschicht-Modells

Um eine praktische Anwendung der Deltaschicht zu ermöglichen, muss diese angepasst werden. So wurde bei bisherigen Betrachtungen (vergleiche Abschnitt 2.2.3) davon ausgegangen, dass verschobene Knoten mithilfe ihrer ID repräsentiert werden können. Dies reicht allerdings nicht aus, da sich beim Verschieben unter Umständen die Split-Knoten-ID und die Zweig-ID des Knotens ändern, was dazu führt, dass diese Werte ebenso überlagert werden müssen. Hierbei lohnt sich die Überlagerung des ganzen Knoten-Objekts, da dieses nicht viele weitere Eigenschaften hat. Für eine Überlagerung der Split-Knoten-ID und Zweig-ID in separaten Mengen würde zusätzlicher Speicherbedarf zur Verwaltung dieser anfallen.

Ebenso beachtet werden muss, dass sich bei einer instanzbasierten Änderung die topologischen IDs nicht nur von den überlagerten Knoten ändern. Wird ein Knoten durch die Deltaschicht eingefügt oder gelöscht, so ändern sich auch die topologischen IDs aller nachgelagerten Knoten. Gleiches gilt beim Verschieben eines Knotens, hier ändern sich die topologischen IDs der Knoten, welche sich (topologisch) zwischen der alten und neuen Position im Schema befinden. Dies ist mit dem bisherigen Ansatz, die topologische ID im Knoten vorzuhalten (vergleiche Abbildung 6), nicht effizient umsetzbar: Die Deltaschicht müsste schon bei kleinen Änderungen eine Vielzahl an Knoten überlagern, nur, weil deren topologische IDs nicht mehr korrekt sind. Dieses Problem wird dadurch behoben, dass die topologischen IDs fortan nicht mehr innerhalb der Knoten sondern im Schema beziehungsweise in der Deltaschicht verwaltet werden. Dies illustriert Abbildung 31.

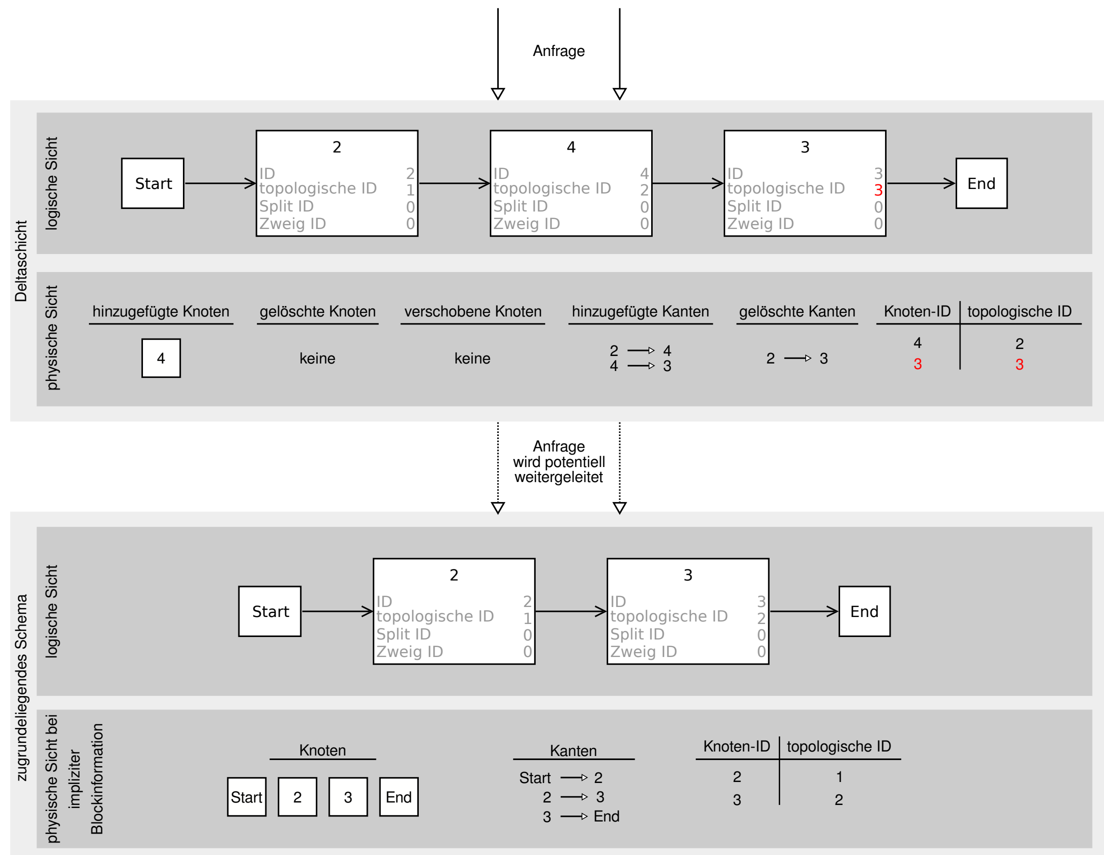


Abbildung 31: Kapselung bei Einsatz der erweiterten Deltaschicht

3.4.2 Speicherbetrachtungen

Für eine Deltaschicht D, welche auf einem Schema S beruht, sei $S + D$ das Schema inklusive den Änderungen der Deltaschicht (*resultierendes Schema*). Analog zu den Betrachtungen für Schemata gilt hierfür:

- $n_{Kanten}(D)$ Anzahl der Kanten von $S + D$
- $n_{Knoten}(D)$ Anzahl der Knoten von $S + D$
- $n_{Split}(D)$ Anzahl der Split-Knoten von $S + D$
- $n_{HKanten}(D)$ Anzahl der hinzugefügten Kanten in D
- $n_{GKanten}(D)$ Anzahl der gelöschten Kanten in D
- $n_{HKnoten}(D)$ Anzahl der hinzugefügten Knoten in D
- $n_{GKnoten}(D)$ Anzahl der gelöschten Knoten in D
- $n_{VKnoten}(D)$ Anzahl der verschobenen Knoten in D
- $n_{HBlock}(D)$ Anzahl der hinzugefügten Blöcke in D
- $n_{GBlock}(D)$ Anzahl der gelöschten Blöcke in D

- $n_{VBlock}(D)$ Anzahl der verschobenen Blöcke in D
- $n_{TopIDs}(D)$ Anzahl der überlagerten topologischen IDs in D

Es gilt weiterhin:

$$\begin{aligned} s_{Kanten}(D) &= (n_{HKanten}(D) + n_{GKanten}(D)) \cdot 2 \cdot s_{int} + s_{Kantenattribute}(D) \\ &\quad + \Delta_{Kanten}(D) \end{aligned}$$

und

$$s_{TopIDs}(D) = n_{TopIDs}(D) \cdot 2 \cdot s_{int} + \Delta_{TopIDs}(D)$$

Hierbei ist zu beachten, dass die topologischen IDs eine Zuordnung von einer Knoten-ID zu einer (neuen) topologischen ID sind und somit zwei ganzzahlige Werte anfallen (vergleiche Abbildung 31).

$$\begin{aligned} s_{Knoten}(D) &= n_{GKnoten}(D) \cdot s_{int} \\ &\quad + (n_{HKnoten}(D) + n_{VKnoten}(D)) \cdot (4 \cdot s_{int} + s_{Objektheader}) \\ &\quad + 2 \cdot n_{HBlock}(D) \cdot s_{int} \\ &\quad + \Delta_{Knoten}(D) \end{aligned}$$

Für gelöschte Knoten reicht die Speicherung dessen ID. Der zweite Summand berechnet den benötigten Speicherplatz für überlagerte, also hinzugefügte und verschobene Knoten. Jeder Knoten besteht aus vier ganzzahligen Werten, wobei für Split- und Join-Knoten jeweils ein weiterer hinzukommt. Den hierfür nötigen Speicheraufwand stellt der dritte Summand dar. Dieser lässt sich aus der Anzahl der hinzugefügten Blöcke berechnen, da jeder hinzugefügte Block einem hinzugefügten Split- und einem hinzugefügten Join-Knoten entspricht.

Für die gesamte Deltaschicht ergibt sich

$$\begin{aligned} s_{Delta}(D) &= s_{Kanten}(D) + s_{Knoten}(D) + s_{TopIDs}(D) \\ &\quad + (n_{HKnoten}(D) + n_{VKnoten}(D)) \cdot s_{Referenz} \\ &\quad + \Delta_{Delta}(D) \end{aligned}$$

3.4.3 Repräsentation

Nach der Erweiterung von Schemata um Blockinformationen muss auch die Deltaschicht diese abbilden. Dies folgt aus der Anforderung, dass eine Deltaschicht die gleichen Zugriffsmöglichkeiten wie ein vollwertiges Schema bieten muss (vergleiche Abschnitt 2.2.3). Hierbei ergeben sich analog zu Schemata die Möglichkeiten, die Blockinformation *explizit* oder *implizit* darzustellen. Diese Varianten werden im Folgenden vorgestellt.

3.4.3.1 Explizite Blockinformation Die explizite Darstellung der Blockinformation in der Deltaschicht geschieht analog zur entsprechenden Repräsentation von Schemata: Der Deltaschicht sind *Deltablöcke* zugeordnet, welche wiederum die Änderungen einzelner Blöcke repräsentieren. Hierbei besitzen die Deltablöcke Mengen, welche hinzugefügte und gelöschte Knoten und Kanten, verschobene Knoten und Änderungen der topologischen IDs darstellen. Analog zu „normalen“ Blöcken, werden auch von den Deltablöcken eingehende und ausgehende Kanten der einzelnen Blöcke abgebildet. Die Deltablöcke müssen nur vorgehalten werden, sofern Änderungen im entsprechenden Block getätigt wurden. Hierunter fallen auch jegliche Änderungen auf Blockebene, also das Hinzufügen von neuen Blöcken und das Verschieben und Löschen von vorhandenen Blöcken. Zusätzlich hält die Deltaschicht eine Menge, welche Knoten-IDs zu Block-IDs auflöst. Hierbei existieren für alle Knoten Einträge in der Menge, für die sich sie Blockzuordnung geändert hat. Die explizite Repräsentation der Deltaschicht ist in Abbildung 32 dargestellt: In Block 3 wird Knoten 10 hinzugefügt. Hieraus folgt, dass auch für die Blöcke 0 und 2 Deltablöcke erzeugt werden müssen, da sich innerhalb dieser Blöcke topologische IDs ändern.

Für Anfragen an die Deltaschicht nach den hinzugefügten oder gelöschten Knoten oder Kanten oder nach verschobenen Knoten müssen die Informationen der einzelnen Deltablöcke zusammengefasst und somit linearer Zeitaufwand getrieben werden, also $\mathcal{O}(n_{HBlock}(D) + n_{VBlock}(D) + n_{GBlock}(D))$.

Bei Schemata wurde die Komplexität für die Berechnung folgender Mengen untersucht:

- Auflösung Knoten-ID zu Block-ID
- Blockmenge des Schemas
- Knotenmenge eines Blocks
- Kantenmenge eines Blocks
- Knoten- und Kantenmenge des gesamten Schemas

Diese Anfragen können aufgrund der Kapselung von Schemata auch an eine Deltaschicht gestellt werden. Für die Auflösung einer Knoten-ID zur Block-ID wird hierzu zuerst die in der Deltaschicht vorhandene Menge untersucht, ob zum entsprechenden Knoten ein Eintrag vorhanden ist. Ist dies der Fall, wird der korrespondierende Wert zurückgeliefert. Ansonsten wird die Anfrage an das Schema weitergeleitet. Es entsteht ein zusätzlicher konstanter Zeitaufwand gegenüber der Anfrage an ein Schema. Die Blockmenge ist durch Abrufen der Blockmenge des Schemas und Anpassen dieser mit den in der Deltaschicht vorhanden Deltablöcken verfügbar. Es entsteht zusätzlicher linearer Aufwand auf der Anzahl der Deltablöcke. Die Knoten- und Kantenmenge eines bestimmten Blocks kann durch Überlagerung des Blocks des Schemas mit den Informationen des entsprechenden Deltablocks errechnet werden, sofern dieser vorhanden ist. Hierzu muss die Knoten-/Kantenmenge des zugrunde liegenden Blocks abgerufen werden und diese durch die im Deltablock gespeicherten Mengen der hinzugefügten und gelöschten Knoten beziehungsweise Kanten mit linearem Zeitaufwand angepasst werden. Die Anfrage nach allen im resultierenden Schema vorhandenen Knoten und Kanten benötigt analog zur entsprechenden Repräsentation von Schemata linearen Aufwand, da die Informationen für die einzelnen Blöcke gesammelt und zusammengefügt werden müssen.

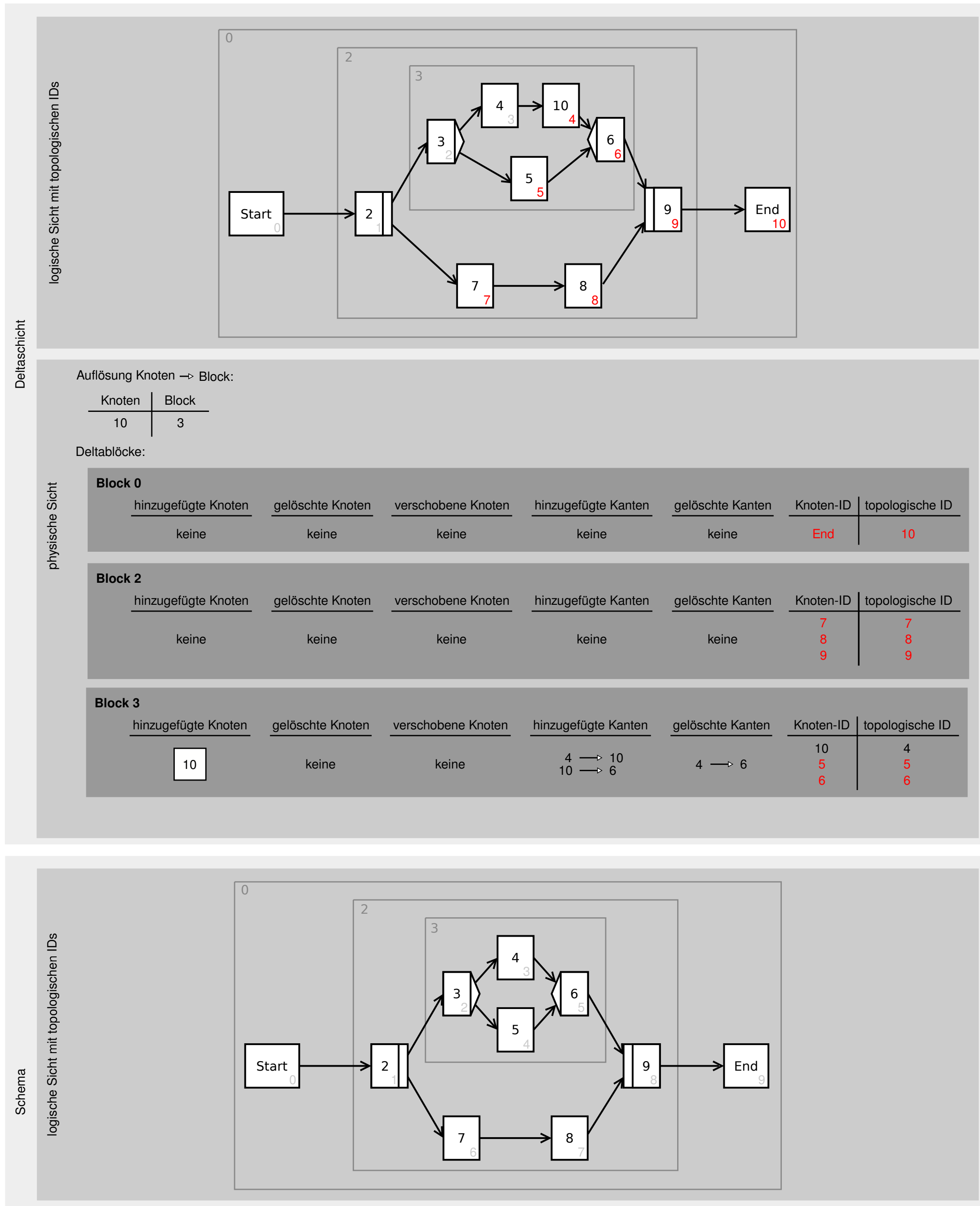


Abbildung 32: Deltaschicht mit expliziter Blockinformation

3.4.3.2 Implizite Blockinformation Bei der Darstellung mit impliziter Blockinformation werden für hinzugefügte, gelöschte und verschobene Knoten und für hinzugefügte und gelöschte Kanten Mengen vorgehalten. Dies gleicht der Repräsentation aus Abbildung 31.

Wird die Menge der hinzugefügten oder gelöschten Knoten oder Kanten, oder der verschobenen Knoten angefragt, so können diese in konstanter Zeit beantwortet werden, da diese Mengen bereits vorliegen.

Die im vorigen Abschnitt rekapitulierten Anfragen an Schemata können auch an eine Deltaschicht mit impliziter Blockinformation gestellt werden. So kann die Knoten-ID zur korrespondierenden Block-ID ebenfalls mithilfe der im Knoten-Objekt vorhandenen Split-Knoten-ID aufgelöst werden. Dieses ist entweder direkt in der Deltaschicht verfügbar (wenn der Knoten verschoben oder hinzugefügt wurde) oder kann vom zugrunde liegenden Schema abgerufen werden. Es fällt gegenüber einer Anfrage an ein Schema zusätzlicher, konstanter Zeitaufwand an. Die Blockmenge kann durch Berechnung der Blockmenge des Schemas errechnet werden. Hierbei müssen für in der Deltaschicht hinzugefügte Knoten mit linearem Zeitaufwand überprüft werden, ob diese einen neuen Block aufspannen und somit ein weiterer Block existiert. Ebenso muss die Menge der gelöschten Knoten linear untersucht werden, ob ein Block komplett entfernt wurde. Gleiches gilt für die Bestimmung der Knoten- und Kantenmengen eines bestimmten Blocks: Nachdem die ursprünglichen Daten vom Schema abgerufen sind, müssen diese mit linearem Zeitaufwand durch Untersuchen der in der Deltaschicht vorhandenen Mengen angepasst werden. Das Ergebnis der Anfrage nach allen im resultierenden Schema vorhandenen Knoten oder Kanten wird errechnet, indem die Daten des zugrunde liegenden Schemas abgerufen und durch die Mengen der hinzugefügten und gelöschten Knoten und Kanten angepasst wird. Hierbei fällt ebenfalls linearer Zeitaufwand an.

3.4.3.3 Vergleich Betrachtet man den Speicheraufwand, der für die explizite Repräsentation der Deltaschicht getrieben werden muss, so erkennt man, dass dieser unverhältnismäßig ist. Wird nur ein Knoten verschoben, müssen möglicherweise mehrere Deltablöcke vorgehalten werden. Dieser Speichermehraufwand ist mit den leicht besseren Laufzeiten bei Anfragen nicht auszugleichen. Deshalb wird der Ansatz der expliziten Speicherung der Blockinformationen in der Deltaschicht verworfen. Dies schließt allerdings nicht aus, dass die Repräsentation mit expliziter Blockstruktur bei Schemata eingesetzt wird. Es ist somit möglich, dass eine Deltaschicht mit impliziter Blockstruktur ein Schema überlagert, welches in expliziter Blockstruktur vorliegt.

3.4.4 Clusterung und Schemaevolution

Die in Abschnitt 3.3 vorgestellte grobe Clusterung optimiert eine Schemaevolution dahingehend, dass einige Instanzen bereits vor deren Einlagerung aus dem Sekundärspeicher als migrierbar und nicht migrierbar eingestuft werden können. Hierbei müssen allerdings auch instanzbasiert geänderte Instanzen betrachtet werden. Ziel ist ebenfalls eine Einstufung dieser in migrierbare, nicht migrierbare und genauer zu untersuchende Instanzen.

Bei einer Schemaevolution werden für instanzbasiert geänderte Instanzen die Änderungen am ursprünglichen Schema und die instanzbasierten miteinander vereinigt. Diese

strukturellen Anpassungen sind nach der Überprüfung der strukturellen Konfliktfreiheit zusätzlich zur Überprüfung der zustandsbasierten Verträglichkeit zu tätigen. Die grobe Clusterung bildet allerdings ausschließlich den Zustand von Instanzen ab und besitzt keine Informationen über deren Struktur. Es ist somit offensichtlich, dass alleine mithilfe der groben Clusterung die Entscheidung nicht getroffen werden kann, ob eine instanzbasiert geänderte Instanz migrierbar oder nicht migrierbar ist. Die Instanz muss also eingelagert werden, wenn die zustandsbasierte Verträglichkeit gegeben ist, um die strukturellen Überprüfungen und Anpassungen durchzuführen. Sollte die zustandsbasierte Verträglichkeit allerdings ergeben, dass die Instanz nicht migrierbar ist, so muss die Struktur auch nicht untersucht werden. Dies führt zu der Annahme, dass in diesem Fall die instanzbasiert geänderte Instanz auch nicht eingelagert werden muss und die grobe Clusterung zur Bestimmung der zustandsbasierten Verträglichkeit herangezogen werden kann.

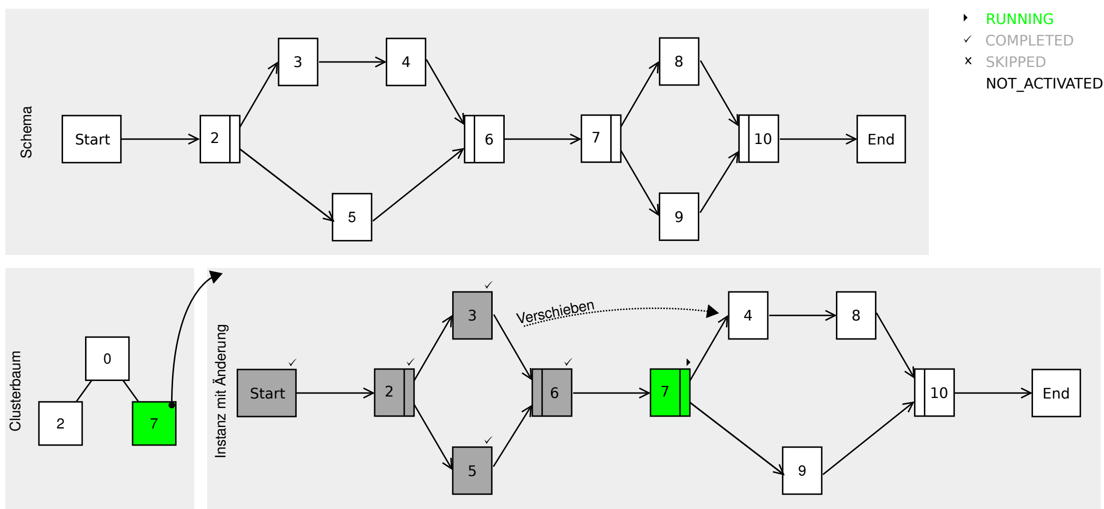


Abbildung 33: Schemaevolution mit instanzbasierten Änderungen

Abbildung 33 stellt allerdings ein Gegenbeispiel dar. Auf dem dargestellten Schema basiert eine Instanz mit einer instanzbasierten Änderung. Es wird davon ausgegangen, dass die Instanz im Clusterbaum des ursprünglichen Schemas gehalten werden kann. Angenommen es wird die selbe Änderung auf dem Schema durchgeführt, welche bereits für die Instanz durchgeführt wurde. Es wird also Knoten 4 zwischen Knoten 7 und 8 verschoben. Hierbei wird die Kante von Knoten 3 nach Knoten 6 gelöscht und somit Knoten 6 als geändert markiert. Dies bedeutet, dass Cluster 2 vom Algorithmus `findPositions` korrekterweise als Stelle vom Typ *COMPACT* erkannt wird. Somit werden alle Instanzen aus Clustern als nicht migrierbar eingestuft, welche Cluster 2 nachgeordnet sind. Dies trifft auch auf die gezeigte instanzbasiert geänderte Instanz zu, obwohl diese migrierbar ist, da die instanzbasierten Änderungen und die Änderungen am Schema übereinstimmen. Hieraus folgt, dass die Annahme fehlerhaft war, auch instanzbasiert geänderte Instanzen im Clusterbaum des ursprünglichen Schemas halten zu können.

Somit stellt sich heraus, dass die grobe Clusterung keine Vorteile für instanzbasiert geänderte Instanzen bietet. Diese müssen bei einer Schemaevolution alle eingelagert und nach den bisherigen Methoden [Rin04, Jur06] untersucht werden.

3.5 Fazit

In diesem Kapitel wurde eine geeignete Partitionierung für Schemata gefunden und die Repräsentation von Schemata aus Abschnitt 2.2 entsprechend angepasst. Aufbauend darauf wurde die grobe Clusterung eingeführt, welche auf Basis einer impliziten oder expliziten Markierung einer Instanz für eine Laufzeitoptimierung bei einer Schemaevolution herangezogen werden kann. Es wurden Algorithmen beschrieben, mit deren Hilfe eine Einordnung der Instanzen vor einer Schemaevolution in migrierbare und nicht migrierbare Instanzen möglich ist, sowie Instanzen bestimmt werden können, welche nach den Methoden aus [Rin04, Jur06] genauer untersucht werden müssen. Danach wurden instanzbasierte Änderungen und deren Repräsentation der Blockinformation betrachtet. Abschließend wurde festgestellt, dass die grobe Clusterung keine Möglichkeit der Optimierung für instanzbasiert geänderte Instanzen bereit hält.

4 Umsetzung

In diesem Kapitel wird die Realisierung einer Workflow-Testumgebung beschrieben, welche auf den in Kapitel 2 vorgestellten Grundlagen beruht und die Konzepte aus Kapitel 3 beinhaltet. Ziel ist es, mithilfe dieser Umsetzung quantitative Untersuchungen der verschiedenen vorgestellten Varianten zur Realisierung durchzuführen.

Schemata und Instanzen müssen sowohl im Primär- als auch im Sekundärspeicher repräsentiert werden. Dies ist nötig, da Anfragen an die Workflow-Testumgebung im Primärspeicher stattfinden, die Daten allerdings auch persistiert werden müssen. Nach einführenden Betrachtungen werden zuerst die Primärspeicherrepräsentationen untersucht. Hierauf aufbauend folgen die Repräsentationen im Sekundärspeicher. Dieses Kapitel schließt eine Betrachtung von Anfragen auf Kollektionen von Schemata und Instanzen ab.

4.1 Einführung

Für eine Realisierung der Konzepte ist die Entscheidung für eine Programmiersprache und -umgebung nötig. Hier bietet sich Java™ [1][GJSB05] an, da es weit verbreitet, einfach anzuwenden und auf verschiedenen Softwareplattformen lauffähig ist.

In diesem Abschnitt werden zuerst für diese Arbeit relevante Aspekte von Java™ betrachtet. Anschließend wird ein Überblick über die Architektur der Workflow-Testumgebung gegeben.

4.1.1 Relevante Aspekte von Java™

Die Java™ Virtual Machine (JVM) ist eine Ausführungsumgebung für objektorientierte Programme, welche in so genanntem Bytecode vorliegen. Dieser entsteht durch Übersetzen von Quelltext, welcher üblicherweise in der Programmiersprache Java™ geschrieben ist. Durch diese Abstraktionsebene ist es möglich, ein in Bytecode vorliegendes Programm auf unterschiedlichen Wirtssystemen auszuführen, sofern für dieses eine JVM existiert. Außerdem wird in der JVM ein Garbage Collector [JL96, Knu97] eingesetzt, welcher nicht weiter benötigte Java™ -Objekte aus dem Primärspeicher automatisch entfernt. Die JVM setzt außerdem die HotSpot-Optimierung [Sun09a] ein. Dies ist ein sogenannter Just-In-Time-Übersetzer, welcher den vorhandenen Bytecode bei Bedarf in nativen Programmcode übersetzt. Dieser kann auf dem Wirtsrechner ohne weitere Interpretation der JVM ausgeführt werden.

Diese Optimierungen beeinflussen die Messungen, die Ziel der Umsetzung sind. Es ist zum Beispiel nicht vorhersagbar, *wann* der Speicher bereinigt und Rechenzeit für das Auffinden und Beseitigen der nicht länger benötigten Objekte benötigt wird. Um die hieraus resultierenden Messfehler zu minimieren, werden die Messungen mehrmals durchgeführt

[1] Java™ ist eine eingetragene Marke der Firma Sun Microsystems Inc.

und ein Mittelwert errechnet. Außerdem wird der Garbage Collector während der Messungen an geeigneten Stellen aufgerufen [Rou02], um eine automatische Bereinigung des Speichers möglichst zu unterbinden.

Auch die HotSpot-Optimierung erzeugt Messfehler. So ist nicht abzuschätzen, *wann* der Bytecode in nativen Code übersetzt wird. Hier hilft ein Kommandozeilenparameter für die JVM, welcher erzwingt, dass bereits beim ersten Aufruf einer Methode diese in nativen Programmcode übersetzt werden soll: `-XX:CompileThreshold=1` [Sun09c]. Hierfür ist es allerdings nötig, dass vor der ersten Messung bereits alle benötigten Methoden einmal aufgerufen wurden. Deshalb wird bei den Messungen ein so genannter *First Shot* durchgeführt, welcher das gleiche Verhalten wie ein normaler Testlauf hat, jedoch nicht zum Ergebnis gerechnet wird.

Die Einführung des First Shot behebt auch das Problem, dass die JVM bei der ersten Benutzung einer Klasse die zugehörigen Daten aus dem Dateisystem lädt. Dies verfälscht den Testlauf ebenfalls.

Um die Größe von Java™ -Objekten zu bestimmen, werden zuerst mithilfe des Garbage Collectors alle Daten aus dem Primärspeicher entfernt, die nicht mehr referenziert werden. Dann wird der verbrauchte Speicherplatz gemessen und 100.000 Objekte der zu untersuchenden Klasse erzeugt. Hiernach folgt eine erneute Messung des verbrauchten Speicherplatzes. Dadurch ist es möglich, den Speicherplatz eines Objektes zu errechnen [Rou02, Kre02]. Messfehler, die durch einen nicht wie erwartetet funktionierenden Garbage Collector entstehen, werden durch die große Anzahl an Objekten minimiert.

In dieser Arbeit kommt die 32-Bit-Variante der Sun Java™ VM in der Version 1.6.0_16 unter Ubuntu Linux 9.04 zum Einsatz.

4.1.2 Architektur

Die Anforderung an die mithilfe von Java™ realisierte Testumgebung besteht in der Durchführung von quantitativen Messungen des Speicherverbrauchs und der Laufzeit. Diese Messungen werden durch so genannte *Tests* abgedeckt. Das Hauptprogramm bekommt als Kommandozeilenparameter die Namen der Tests übergeben, startet diese hiernach und gibt die Ergebnisse auf der Konsole und, falls gewünscht, in eine Textdatei aus.

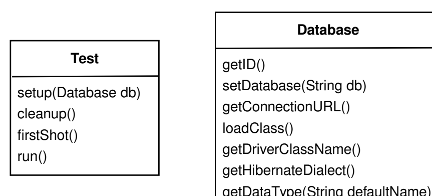


Abbildung 34: `Test`- und `Database`-Schnittstelle

Die Tests implementieren die `Test`-Schnittstelle (*Interface*), welche in Abbildung 34 dargestellt ist. Diese bietet die Möglichkeit, den Test vorzubereiten, einen First Shot, die eigentliche Messung und Aufräumarbeiten nach dem Testlauf durchzuführen. Die Tests sind unabhängig vom eingesetzten Sekundärspeicher, dessen Daten zur Ansteuerung von einer Implementierung der `Database`-Schnittstelle geliefert werden, welche ebenfalls in Abbildung 34 dargestellt ist. Mithilfe dieser Schnittstelle ist es möglich, den Namen der zu benutzenden Datenbank des RDBMS zu wählen und die für JDBC notwendige `ConnectionURL` abzurufen. Diese beinhaltet alle zum Verbindungsaufbau nötigen Eigenschaften. Die Methode `loadClass()` lädt den JDBC-Treiber. Um das RDBMS auch für JPA (Hibernate) benutzen zu können, werden die beiden weiteren Informationen der Treiberklasse und des einzusetzenden Hibernate-Dialekts [Red09] benötigt. Abschließend stellt die Schnittstelle eine Möglichkeit zur Anpassung von Datentyp-Namen zur Verfügung. Bei PostgreSQL heißt der Standard-Datentyp `CLOB` zum Beispiel `TEXT`. Das Hauptprogramm wählt eine Implementierung der `Database`-Schnittstelle anhand der ID aus, welche als Kommandozeilenparameter übergeben wird. Diese wird dann der Implementierung der `Test`-Schnittstelle zur Verfügung gestellt.

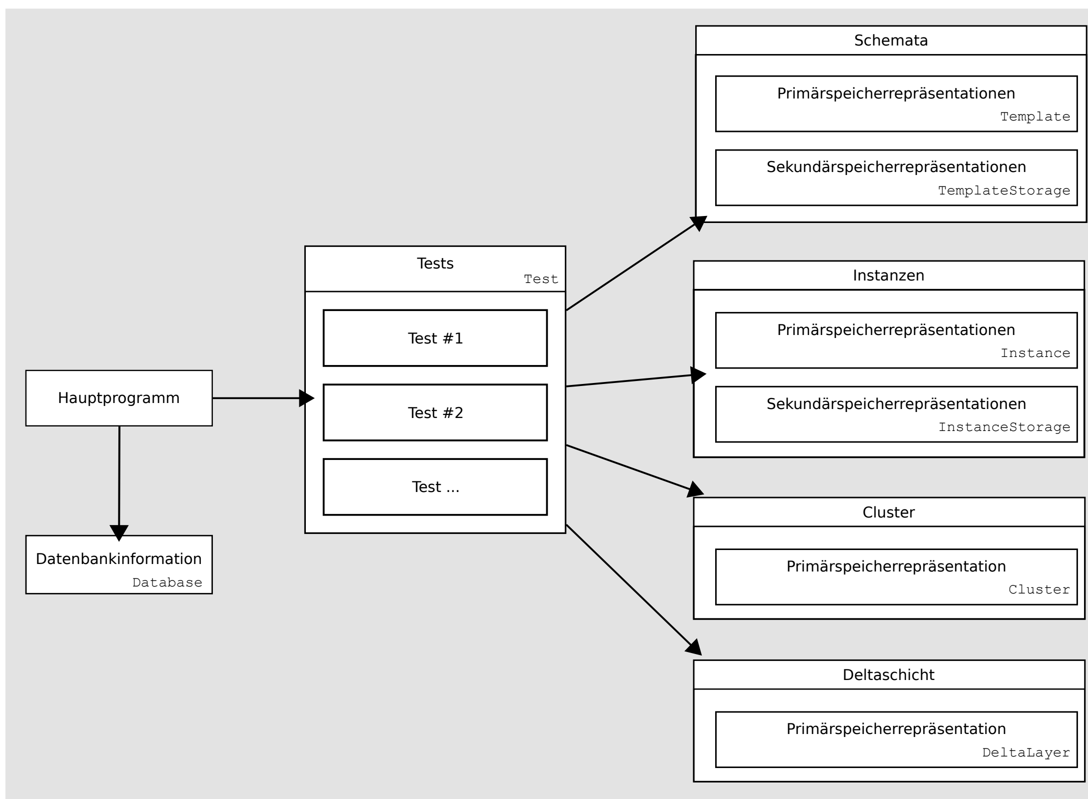


Abbildung 35: Architektur der Workflow-Testumgebung mit Schnittstellen

Der Rest der Workflow-Testumgebung teilt sich auf in die Bereiche Schemata, Instanzen, instanzbasierte Änderungen (Deltaschicht) und Cluster. Die Tests realisieren Zugriffe hierauf und führen die spezifischen Messungen durch. Abbildung 35 illustriert dies. Hierbei sind die Namen der jeweils eingesetzten Schnittstellen den einzelnen Komponenten zugeordnet.

4.2 Primärspeicher

Die Entitäten werden im Primärspeicher durch Java™ -Objekten dargestellt. Es existieren sowohl für Schemata als auch für Instanzen verschiedene Repräsentationen, welche in Kapitel 2 und 3 konzeptionell vorgestellt wurden. Deren Umsetzung wird zusammen mit den Repräsentationen einer Deltaschicht und der groben Clusterung im Folgenden nach der Betrachtung der Standard-Datenstrukturen von Java vorgestellt.

4.2.1 Standard-Datenstrukturen von Java™

Bei der Realisierung der Entitäten durch Primärspeicherobjekte ist eine Nutzung von Standard-Datenstrukturen von Java™ sinnvoll. Die Eigenschaften der verschiedenen bereitgestellten Varianten werden hier erläutert.

Eine häufig benötigte Datenstruktur ist eine `Map`. Diese stellt eine Zuweisung eines Wertes zu einem Schlüssel dar. Es gibt verschiedene Implementierungsvarianten einer `Map` in der Standardbibliothek, welche sich in der Verwaltung der Schlüssel unterscheiden: `HashMap` benutzt Hashing [CLRS09], während `TreeMap` eine Baumstruktur benutzt. Somit hat eine `HashMap` erwartet konstanten Zugriff, eine `TreeMap` logarithmischen. Der Speicherplatzverbrauch der Implementierungen ist in Abbildung 36 gegenüber gestellt.

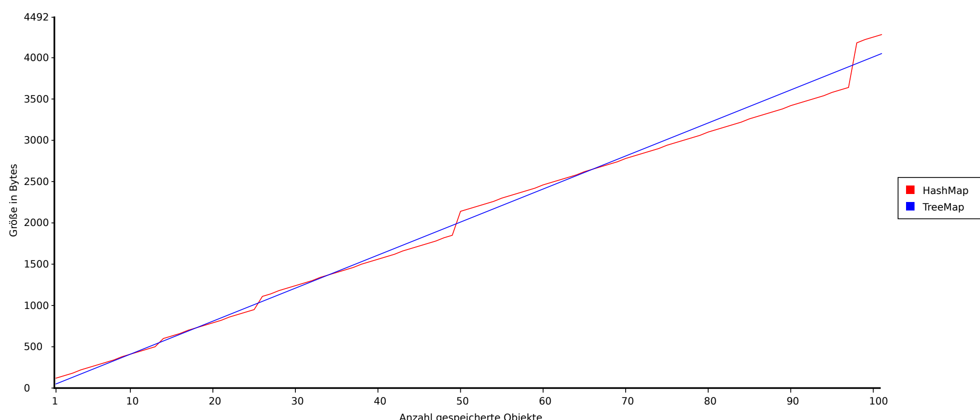


Abbildung 36: Speicherverbrauch von `Maps`

Hierbei ist zu erkennen, dass die `HashMap` nahezu einen linearen Speicheraufwand zu der Anzahl der in ihr gespeicherten Objekte besitzt, genau wie die `TreeMap`. Die treppenartigen Abweichungen wie zum Beispiel bei 50 gespeicherten Objekten sind in einer Vergrößerung der Hashtabelle begründet, welche die `HashMap` verwaltet. Aufgrund der konstanten Zugriffszeit auf die Daten ist die Benutzung der `HashMap` im Allgemeinen vorzuziehen. Wird eine Sortierung der Schlüssel-Wert-Paare der `Map` benötigt, muss allerdings `TreeMap` benutzt werden, da `HashMap` die Daten nicht sortiert liefern kann.

Eine weitere häufig benötigte Datenstruktur ist das `Set`. Dieses stellt eine Menge von beliebigen Java™ -Objekten dar. Auch hier existieren analog zu `Map`s die beiden grundlegenden Implementierungen `HashSet` und `TreeSet`. Diese greifen auf die Implementierungen von `Map` zurück, wobei jeweils nur Schlüssel gespeichert werden. Aufgrund dessen gelten für `HashSet` und `TreeSet` die gleichen Betrachtungen wie für `HashMap` und `TreeMap`.

4.2.2 Schemata

Auf Basis der vorgestellten Standard-Datenstrukturen von Java™ ist die Ausarbeitung von Primärspeicherrepräsentationen von Schemata möglich, welche die `Template`-Schnittstelle implementieren. Diese ist in Abbildung 37 dargestellt. In dieser, wie in folgenden Abbildungen, wird eine Semantik für Pfeile benutzt, welche sich an der von Klassendiagrammen der *Unified Modeling Language* [Obj06] anlehnt. Hierbei bilden Pfeile mit ausgefüllter Spitze eine Assoziation mit Angabe der Navigierbarkeit. Bei denjenigen mit unausgefüllten Spitzen wird weiter zwischen gestrichelten und durchgezogenen Pfeilen unterschieden. Die gestrichelten stellen eine Schnittstellenrealisierungsbeziehung dar, die durchgezogenen eine Generalisierung.

Die `Template`-Schnittstelle bietet die Möglichkeit, neben der Blockstruktur auch einen Knoten anhand seiner ID abzurufen, und die Vorgänger, Nachfolger und die topologische ID eines Knotens zu bestimmen. Außerdem lässt sich die Relation zwischen zwei Knoten mithilfe des Algorithmus `getNodeRelation` aus Kapitel 3.2.2 berechnen. Für Schemata existieren zwei Repräsentationen, wie in Abschnitt 3.2.3 erläutert: `ExplicitTemplate` bietet eine Repräsentation mit expliziter Blockstruktur, während `ImplicitTemplate` eine mit impliziter Blockstruktur darstellt. Auf diese wird nach der Vorstellung der Knoten-Objekte näher eingegangen.

Den beiden Repräsentationen ist gemein, dass Kantenmengen durch eine `Map` dargestellt werden. Diese besitzt als Schlüssel die ID des Knotens bei dem Kanten beginnen. Als Wert wird ein `Set` benutzt, welches die IDs der Zielknoten der Kanten beinhaltet. Eine Menge an topologischen IDs von Knoten wird ebenfalls von einer `Map` repräsentiert. Der Schlüssel ist hierbei die ID des Knotens und der Wert die zugeordnete topologische ID.

4.2.2.1 Knoten Ein Knoten wird durch ein Objekt der Klasse `Node` repräsentiert. Diese Klasse hält alle Attribute eines Knoten. Hierbei wird die topologische ID des Knotens nach Abschnitt 3.4.1 vom Schema bzw. von der Deltaschicht verwaltet. Es ist allerdings

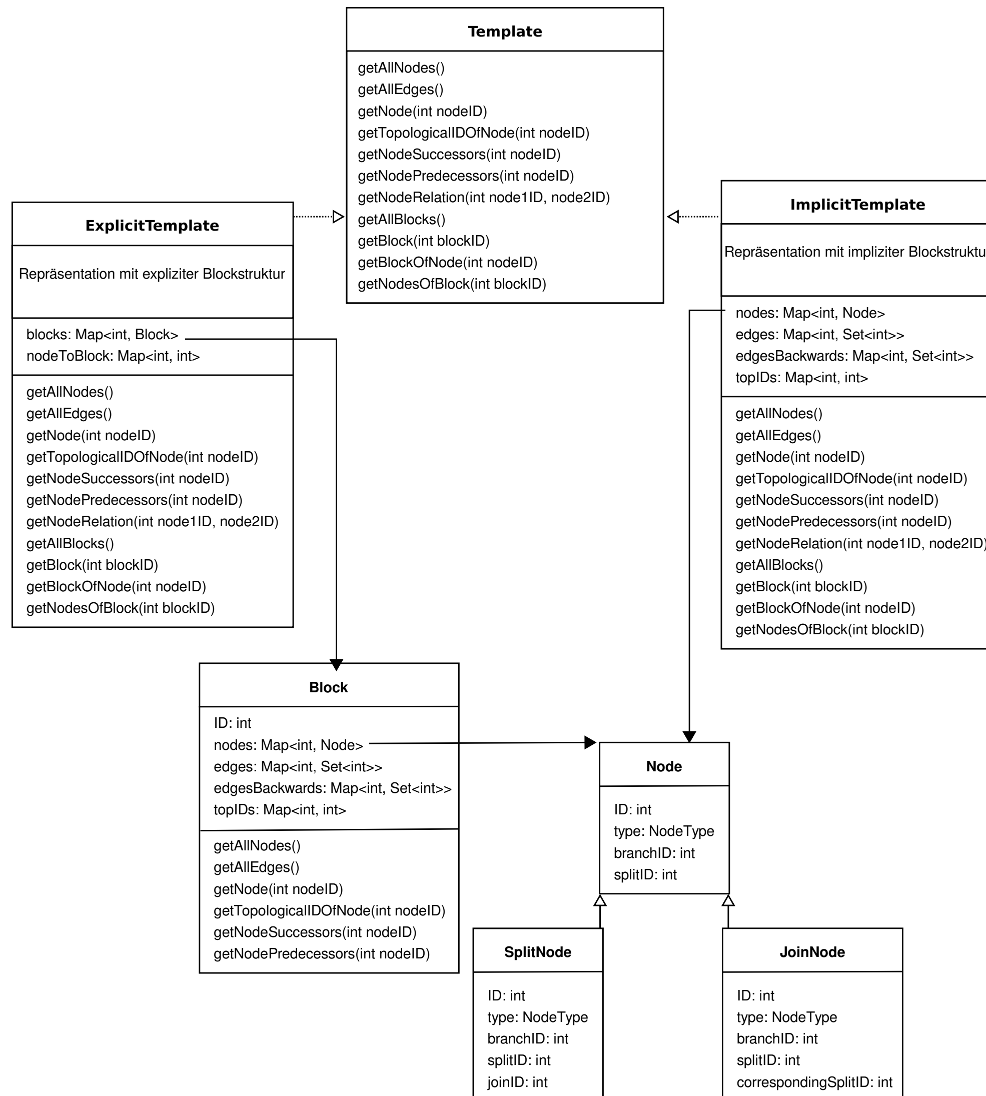


Abbildung 37: `Template`-Schnittstelle

zu beachten, dass es zwei Knotentypen gibt, für die mehr Daten existieren. Ein Split-Knoten benötigt die ID des zugehörigen Join-Knotens und ein Join-Knoten die ID des korrespondierenden Split-Knotens. Dies ist über Vererbung [DN67] gelöst: `SplitNode` leitet von `Node` ab und enthält ein weiteres Feld, das die nötigen Informationen bereit hält. Entsprechendes gilt für `JoinNode`.

Eine Menge von Knoten wird durch eine `Map` repräsentiert. Diese benutzt als Schlüssel die ID des Knoten und als zugeordneten Wert das `Node`-Objekt. Dies wird der Verwendung eines `Sets` vorgezogen um einen konstanten Laufzeitfaktor beim Zugriff auf einen Knoten aufgrund seiner ID zu erreichen. Bei einem `Set` wäre es nötig, die ganze Menge der Knoten

auf den gesuchten hin zu untersuchen, während dieser bei einer `Map` direkt abgerufen werden kann.

`Node`-Objekte werden von beiden folgenden Schema-Repräsentationen zur Darstellung von Knoten benutzt.

4.2.2.2 ExplicitTemplate Ein `ExplicitTemplate` repräsentiert ein Schema, welches die Blockinformation explizit speichert (vergleiche Abbildungen 23 und 24). Hierbei wird ein Block mithilfe eines Objekts der Klasse `Block` repräsentiert. Das Schema referenziert mehrere dieser `Block`-Objekte. Diese beinhalten jeweils die Knoten und Kanten des entsprechenden Blocks. Da die Kanten zum Beispiel für den Algorithmus `getNodeRelation` rückwärts gerichtet benötigt werden, wird zusätzlich eine Menge dieser gehalten. Um die geforderte konstante Laufzeit beim Auflösen einer Knoten-ID zu einer Block-ID zu gewährleisten, wird eine weitere Menge innerhalb von `ExplicitTemplate` benötigt, welche dies abbildet. Es ist zu bedenken, dass Abschnitt 3.2.3.1 davon ausgeht, dass die Block-ID eines Knoten durch dessen Split-Knoten-ID verfügbar ist und somit von keinem zusätzlichen Speicherbedarf ausgeht. Hier wird allerdings eine Knoten-*ID* zu einer Block-ID aufgelöst. Das Knoten-*Objekt* ist allerdings nur innerhalb des `Block`-Objekts vorhanden. Um dieses abzurufen muss allerdings die gesuchte Block-ID vorhanden sein, es entsteht somit ein zirkulärer Bezug. Aufgrund dessen wird eine weitere Menge benötigt, welche Knoten-ID auf Block-ID abbildet. Dies geschieht mit einer `Map`. Es entsteht ein zusätzlicher Speicherplatzaufwand von $\mathcal{O}(n_{Knoten}(S))$.

Aufgrund der expliziten Partitionierung ergibt sich eine geeignete Möglichkeit zur Aufteilung der Knoten- und Kantenmengen in Primär- und Sekundärspeicher. So werden initial nicht alle `Block`-Objekte im Primärspeicher gehalten. Wird ein nicht eingelagerter Block oder ein Knoten dessen angefragt, kann dieser beim Sekundärspeicher angefragt und nachgeladen werden.

4.2.2.3 ImplicitTemplate Diese Klasse repräsentiert ein Schema mit impliziter Blockrepräsentation (vergleiche Abschnitt 3.2.3.2). Analog zu `Block` beinhaltet auch diese Klasse die Kanten zusätzlich rückwärts gerichtet. Zusätzlich zu den Kantenmengen werden die topologischen IDs und eine Knotenmenge mit Objekten der Klasse `Node` gehalten.

Ein dynamisches Nachladen der Blöcke, wie `ExplicitTemplate` es unterstützt, ist bei `ImplicitTemplate` nicht sinnvoll, da flache Knoten- und Kantenmengen gehalten werden. Es wäre denkbar, diese Mengen zu partitionieren und die Knotenobjekte erst einzulagern, wenn der Block angefragt wird. Hier entsteht allerdings ein zirkulärer Bezug. Soll ein Knoten anhand seiner ID abgerufen werden, so muss überprüft werden, ob dieser eingelagert ist. Ist dies nicht der Fall, so muss dessen Block bestimmt und dieser eingelagert werden. Um allerdings die Auflösung von Knoten-ID nach Block-ID zu bewerkstelligen, wird die Split-Knoten-ID des Knotens benötigt. Dieser soll aber gerade nachgeladen werden und ist deshalb nicht im Primärspeicher vorhanden. Somit müsste hierfür analog zu `ExplicitTemplate` eine weitere Menge geführt werden, welche diese

Auflösung übernimmt. Dies erfordert allerdings zusätzlichen Speicher mit Aufwand von $\mathcal{O}(n_{Knoten}(S))$, was unverhältnismäßig hoch ist, da sonst der Speichervorteil gegenüber einem Schema mit expliziten Blockinformationen verloren geht. Deshalb unterstützt die Realisierung mit impliziten Blockinformationen kein dynamisches Nachladen.

4.2.3 Deltaschicht

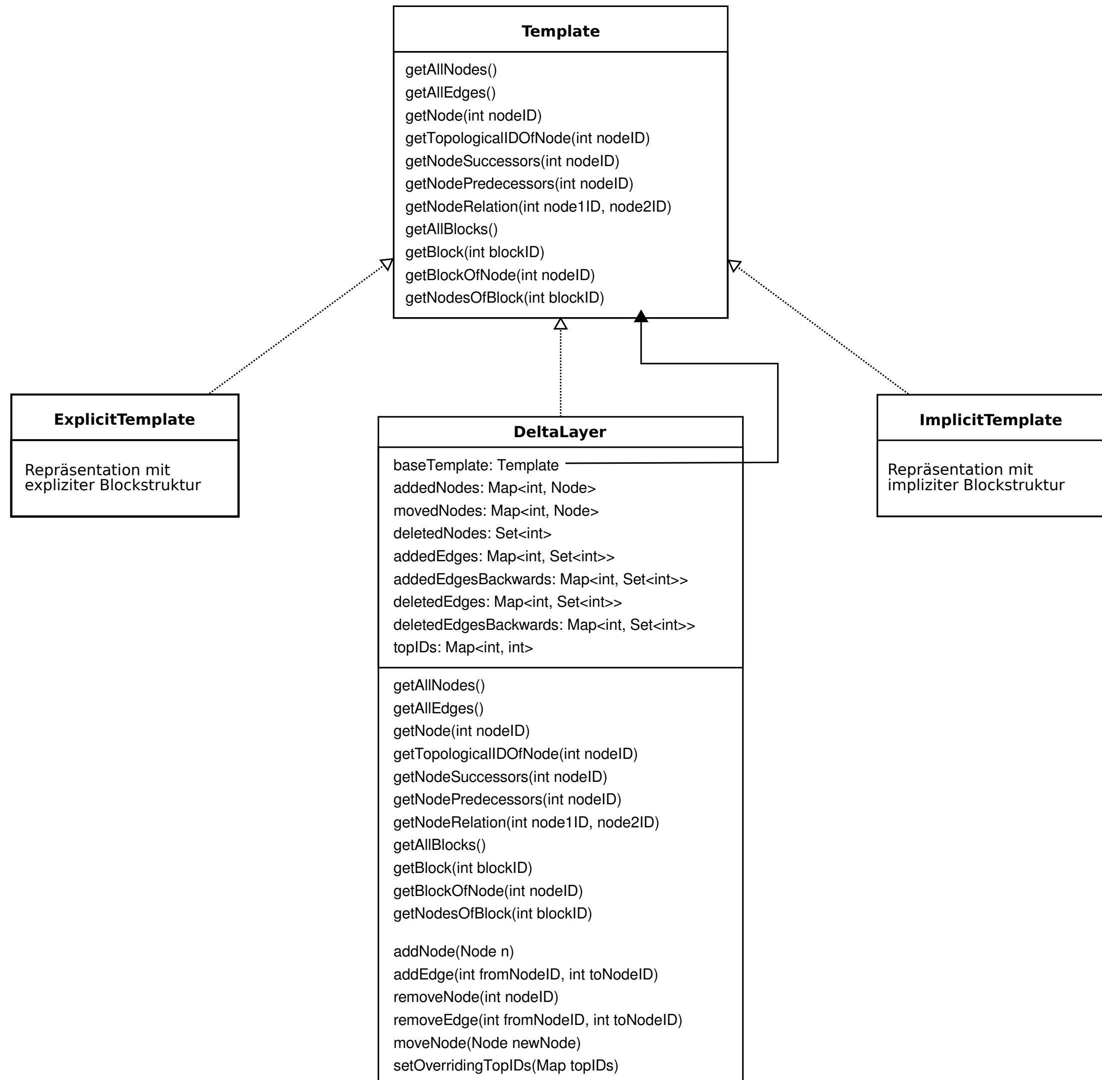


Abbildung 38: Die Klasse `DeltaLayer`

Aufbauend auf den vorgestellten Repräsentationen von Schemata muss zur Unterstützung von instanzbasierten Änderungen eine Deltaschicht definiert werden. In Abschnitt 3.4.3 wurde festgestellt, dass hierbei nur die Repräsentation mit impliziten Blockstrukturen

sinnvoll ist. Dessen Repräsentation wird mithilfe der Klasse `DeltaLayer` realisiert. Diese besitzt die Möglichkeit, zu einem zugrunde liegenden Schema Knoten und Kanten hinzuzufügen, daraus zu löschen oder als verschoben zu markieren. Ebenso wird eine Referenz auf das zugrunde liegende Schema gehalten und die überlagerten topologischen IDs verwaltet. Es werden hier keine Korrektheitsuntersuchungen durchgeführt, da diese von Änderungsoperationen [Jur06] durchgeführt werden und im Rahmen dieser Arbeit nicht relevant sind.

Wie in Abbildung 38 dargestellt, implementiert `DeltaLayer` die Schnittstelle `Template` um wie ein Schema („gekapselte Vorlage“ [KR10]) benutzt werden zu können. Hierbei werden die Methoden von `Template` innerhalb von `DeltaLayer` neu implementiert. Diese nehmen die Überlagerung der instanzbasierten Daten über die Schemadaten vor.

4.2.4 Instanzen

Wie für Schemata gibt es auch für Instanzen zwei verschiedene Repräsentationen, welche in Abschnitt 2.2 vorgestellt wurden. Hierzu zählt eine mit impliziten Knotenzuständen, `ImplicitInstance`, und eine mit expliziten, `ExplicitInstance`. Beide implementieren die Schnittstelle `Instance`, um von der Workflow-Testumgebung einheitlich behandelt werden zu können.

Von einer Instanz kann man die Zustände der einzelnen Knoten abfragen und diese neu setzen. Außerdem kann die Instanz weitergeschaltet und instanzbasiert geändert werden, wie in Abbildung 39 dargestellt.

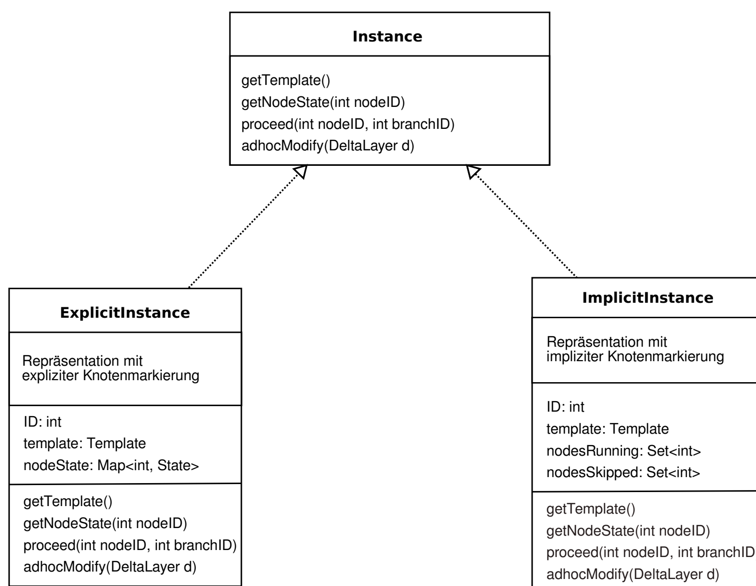


Abbildung 39: `Instance`-Schnittstelle

4.2.4.1 ExplicitInstance Diese Klasse hält die Knotenzustände für jeden Knoten des Schemas explizit vor: Es wird eine `Map` vorgehalten, welche jeder Knoten-ID einen Knotenzustand zuordnet.

Hinzu kommen ein Feld für die ID der Instanz und eine Referenz auf das zugrunde liegende Schema.

4.2.4.2 ImplicitInstance `ImplicitInstance` repräsentiert eine Instanz, welche die Knotenzustände implizit speichert, wie in Abschnitt 2.2 erwähnt. Es werden zwei Mengen in Form von `Set`s gehalten: Die IDs der Knoten, welche RUNNING respektive SKIPPED sind. Die Knotenzustände COMPLETED und NOT_ACTIVATED werden aus diesen Mengen zur Laufzeit berechnet.

Auch hier kommen Felder für die ID der Instanz und eine Referenz auf das zugrunde liegende Schema hinzu.

4.2.5 Cluster

Für die Realisierung der groben Clusterung aus Abschnitt 3.3 existiert die Klasse `Cluster` (vergleiche Abbildung 40). Diese bildet einen Clusterbaum ab, welcher für ein bestimmtes Schema bei dessen Erzeugung angelegt wird. Die `Map` **`instances`** repräsentiert die Cluster. Schlüssel ist hierbei die ID des Clusters, Wert ist ein `Set` von Instanz-IDs, welche sich in diesem Cluster befinden. Die Hierarchie der Cluster wird über zwei weitere `Map`s abgebildet: **`topCluster`** und **`subClusters`**. **`topCluster`** besitzt als Schlüssel die Cluster-IDs und als Wert die ID des jeweils übergeordneten Clusters. **`subClusters`** bildet die Gegenrichtung ab, speichert also zu einer Cluster-ID als Schlüssel ein `Set` der untergeordneten Cluster. Instanzen, welche auf dem Schema beruhen, aber instanzbasiert geändert sind, werden in der separaten Menge **`adhocInstances`** gehalten, welche mithilfe eines `Set`s umgesetzt wurde.

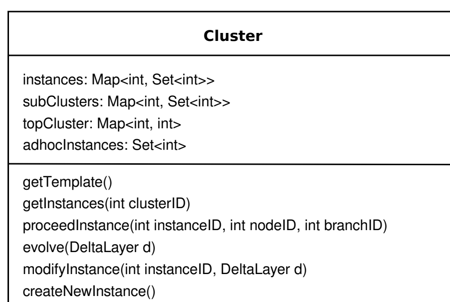


Abbildung 40: Die Klasse `Cluster`

Die `Cluster`-Klasse bietet die Möglichkeit, die Instanzen eines bestimmten Clusters abzurufen, instanzbasiert zu ändern, oder eine Schemaevolution auf dem zugrunde liegenden

Schema durchzuführen. Hierfür bekommt die entsprechende Methode einen `DeltaLayer` übergeben, welcher die Änderungen auf dem ursprünglichen Schema repräsentiert. Es werden dann alle Instanzen des Schemas mithilfe des Algorithmus `preEvolveSelection` aus Anhang D.3 klassifiziert. Die genauere Untersuchung der Instanzen, welche durch den Algorithmus in `preciseCheck` eingeordnet wurden, wird allerdings nicht realisiert, da dies im Rahmen dieser Arbeit nicht relevant ist.

Außerdem bietet `Cluster` die Möglichkeit, eine Instanz weiterzuschalten. Dies darf bei Einsatz der groben Clusterung nicht von der Instanz behandelt werden, da nach dem Weiterschalten eine Neubewertung der Clusterzugehörigkeit der Instanz notwendig ist. Dies kann nur `Cluster` vornehmen.

4.2.6 Zusammenfassung

In diesem Abschnitt wurden die Realisierungsmöglichkeiten der Entitäten der Workflow-Testumgebung im Primärspeicher aufgezeigt. Diese basieren auf Java™ -Objekten. Aufbauend auf der Betrachtung der Standard-Datenstrukturen von Java™ wurden die verschiedenen Varianten erläutert, ein Schema darzustellen, ebenso wie die verschiedenen Möglichkeiten eine Instanz zu repräsentieren. Danach wurde die Umsetzung der groben Clusterung aus Abschnitt 3.3 betrachtet und die Repräsentation der Deltaschicht vorgestellt.

Mithilfe dieser Primärspeicher-Umsetzung ist bereits eine funktionsfähige Testumgebung verfügbar. Es können Schemata angelegt und Instanzen darauf basierend definiert werden. Diese können weitergeschaltet werden und bieten somit die grundlegende Funktionalität eines Workflow-Systems. Durch die Möglichkeit, Instanzen instanzbasiert zu ändern, und dank den Überlegungen bezüglich der Schemaevolution sind die grundlegenden Flexibilitätseigenschaften von ADEPT2 verfügbar.

4.3 Sekundärspeicher

Eine Workflow-Umgebung benötigt neben einer Primärspeicherrepräsentation der Entitäten des Systems auch eine Sekundärspeicherrepräsentation, um realistische Verhältnisse abzubilden. Da Rechner in zeitlich unbestimmten Abständen neu gestartet werden oder Sicherungskopien der Daten angelegt werden sollen, ist eine persistente Speicherung dieser in der Realität unerlässlich.

Hierzu werden verschiedene Realisierungsalternativen vorgestellt. Nachdem die Begriffe *Primärspeicher* und *Sekundärspeicher* präziser definiert wurden, folgt die Untersuchung der verschiedenen Sekundärspeicherrepräsentationen von Schemata und Instanzen. Anschließend werden Zwischenspeicher, so genannte *Caches* [Cle06], zur Optimierung des Zugriffs auf die Entitäten eingeführt. Um Messungen auch ausschließlich unter Verwendung des Hauptspeichers ohne aufwändige Änderungen an der Architektur der Workflow-

Testumgebung realisieren zu können, wird abschließend die Speicherung der Entitäten in flüchtigen Speicher vorgestellt.

4.3.1 Realisierungsalternativen

Es existieren verschiedene Ansätze, die Entitäten im Sekundärspeicher zu speichern. Hierzu zählt die Persistierung im Dateisystem, mithilfe eines relationalen Datenbanksystems und die Verwendung der Java™ Persistence API [Sun06]. Diese werden im Folgenden vorgestellt.

Für die Speicherung im Dateisystem ist die Verwendung von XML [W3C06] aufgrund dessen Verbreitung und einfachen Handhabung zu empfehlen. Darauf basierend werden Schema- und Instanz-Repräsentationen definiert. Die Erstellung der XML-Repräsentation nennt man *Serialisierung*, die Rücktransformation *Deserialisierung*. Die Schemata und Instanzen werden in das XML-Format serialisiert, im Dateisystem gespeichert und können hieraus wieder geladen und deserialisiert werden.

Eine weitere Möglichkeit besteht in der Anwendung von relationalen Datenbanksystemen (RDBMS) [Cod70]. Hier wird eine Auswahl der am Markt erhältlichen Systeme getroffen:

- PostgreSQL [Pos09a], Version 8.4.0
- Oracle [Ora09a] XE, Version 10.2.0.1
- DB2 [IBM09a] Express-C, Version 9.70

Die Anbindung dieser Systeme an die JVM wird mit der Java™ Database Connectivity (JDBC) [Sun09b] realisiert. Nachdem ein geeignetes Datenbanklayout gefunden wurde, können die Daten aus Schemata und Instanzen über die JDBC-Verbindung in den Tabellen des RDBMS gespeichert werden. Beim Laden werden die Daten über die JDBC-Verbindung abgerufen und zu Java™-Objekten zusammengefasst.

Als letzte hier behandelte Möglichkeit, bietet sich noch die Java™ Persistence API (JPA) zur Persistierung von Daten an. Dies ist die Definition einer Schnittstelle, welche die Möglichkeit bietet, Java™-Objekte als Ganzes in relationale Datenbanksysteme zu speichern und daraus zu laden. Hierzu sind Annotationen [GJSB05] an die Primärspeicherklassen notwendig, aus welchen die JPA-Implementierung ein Datenbanklayout generiert. Das Speichern und Laden geschieht mithilfe von Reflection [Sun09d]. Als Implementierung von JPA wurde im Rahmen dieser Arbeit Hibernate [Red09] gewählt. Hierbei wurden folgende Versionen benutzt: Hibernate 3.3.2GA, Hibernate Annotations 3.4.0GA und Hibernate EntityManager 3.4.0GA.

4.3.2 Schnittstellen zwischen Primär- und Sekundärspeicher

Bei den vorgestellten Varianten zur Speicherung im Sekundärspeicher entsteht das Problem, dass die Grenzen zwischen Primär- und Sekundärspeicher verschwimmen. Dies

liegt daran, dass die Objekte, welche die Speicherung implementieren, ebenfalls im Primärspeicher liegen und hier gegebenenfalls Daten zwischenspeichern: RDBMS führen im Allgemeinen Primärspeichercaches und die Daten werden von JDBC im Primärspeicher transportiert. Das gleiche Bild ergibt sich bei JPA: Bei einer solchen Implementierung fällt zusätzlicher Aufwand im Primärspeicher an, um Anfragen beantworten zu können. Trotzdem kann die Workflow-Testumgebung nicht auf diese Daten zugreifen.

Um dieses Problem handhaben zu können, werden die Begriffe *Primärspeicher* und *Sekundärspeicher* für die folgenden Betrachtungen präzisiert:

- Daten im *Primärspeicher* sind diejenigen Daten, auf welche von der Workflow-Testumgebung zugegriffen werden kann
- Zum *Sekundärspeicher* gehören alle Daten, welche in einem RDBMS, im Dateisystem oder per JPA gespeichert sind und Daten, die im Hauptspeicher zur Verwaltung dieser Daten dienen

Hierbei haben die Begriffe Primärspeicher und Sekundärspeicher fortan nicht länger eine Bindung an die physischen Speicherschichten aus Abbildung 16.

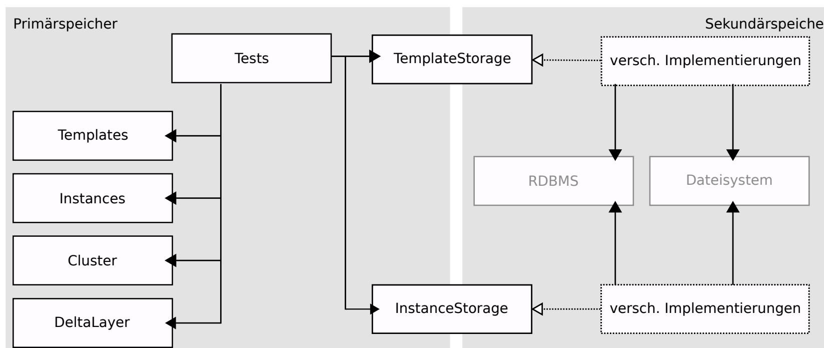


Abbildung 41: Speichertrennung anhand von Storage-Schnittstellen

Um diese Grenze zu verdeutlichen, werden so genannte Storage-Schnittstellen eingeführt. Diese bilden die Trennung zwischen Workflow-Testumgebung und der persistenten Datenhaltung. Es gibt für die Speicherung von Schemata und Instanzen unterschiedliche Storage-Schnittstellen (vergleiche Abbildung 41): `TemplateStorage`, respektive `InstanceStorage`. Diese werden mehrmals implementiert, um die aufgezeigten Realisierungsalternativen abzubilden.

4.3.3 Schemata

Schemata können im Sekundärspeicher auf verschieden Arten repräsentiert werden. Es werden nacheinander eine Speicherung als XML-Dateien und in RDBMS mit expliziter und impliziter Blockstruktur betrachtet. Diesen Abschnitt schließt die Betrachtung der

Speicherung mithilfe von JPA ab. Jede dieser Varianten besitzt eine Implementierung der `TemplateStorage`-Schnittstelle.

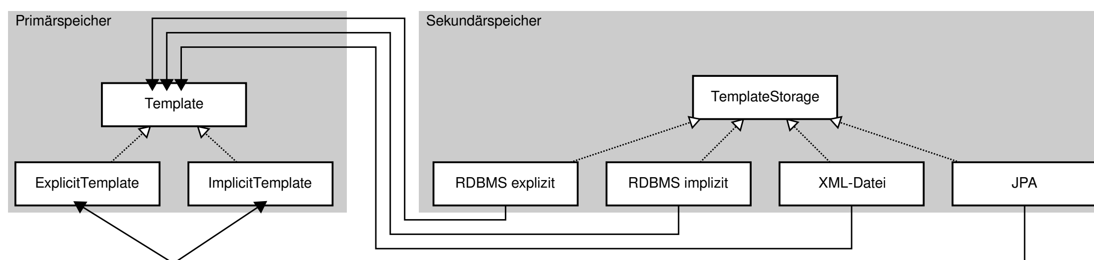


Abbildung 42: Schemarepräsentationen im Primär- und Sekundärspeicher

Durch die Einführung der `Template`-Schnittstelle ist der Zugriff auf die Primärspeicherdaten unabhängig von der Sekundärspeicherrepräsentation (vergleiche Abbildung 42). Es ist also möglich, dass ein Schema im Primärspeicher mit impliziter Blockstruktur vorliegt, der Sekundärspeicher die Daten allerdings mit einer expliziten Blockstruktur speichert. Die Implementierung des Sekundärspeichers ruft hierbei die einzelnen Blöcke von der Primärspeicherrepräsentation ab und speichert diese. Gleiches gilt im gegensätzlichen Fall: Es ist möglich, dass die Daten im Primärspeicher mit expliziter Blockinformation gehalten werden, im Sekundärspeicher allerdings implizit gespeichert werden. Hierbei ruft die Sekundärspeicherimplementierung die Knoten- und Kantenmenge sowie die Menge der topologischen IDs als flache Mengen von der Primärspeicherrepräsentation ab und speichert diese.

Dies gilt nicht beim Einsatz von JPA als Sekundärspeichervariante. Da JPA die Primärspeicherobjekte als Ganzes speichert, befinden sich die Daten im Sekundärspeicher auch in der gleichen Repräsentation wie im Primärspeicher.

Die `TemplateStorage`-Schnittstelle (siehe Abbildung 43) muss neben dem Laden, Speichern und Erzeugen von Schemata auch die Abfrage von einzelnen Blöcken eines Schemas unterstützen. Dies wird benötigt, um das dynamische Nachladen zu ermöglichen, welches die Primärspeicherrepräsentation `ExplicitTemplate` benutzt. Ebenfalls ist es nötig, die Zuordnung von Knoten-IDs zu Block-IDs abrufen zu können (`getBlockInformation`), da dies ebenfalls von der Primärspeicherrepräsentation benötigt wird, um den zirkulären Bezug beim Auflösen einer Knoten-ID zu einer Block-ID zu beheben (siehe Abschnitt 4.2.2.2). Um die Implementierung zu initialisieren ist die Methode `init(Database db)` und zur Bereinigung `cleanup` vorhanden. Ebenso benötigt werden Methoden, die einzelne Knoten, alle Knoten, alle Kanten und alle topologischen IDs abrufen. Für die in diesem Abschnitt folgenden Varianten zur Sekundärspeicherrepräsentation wird jeweils untersucht, in wie fern sie das dynamische Nachladen von Blöcken unterstützen.

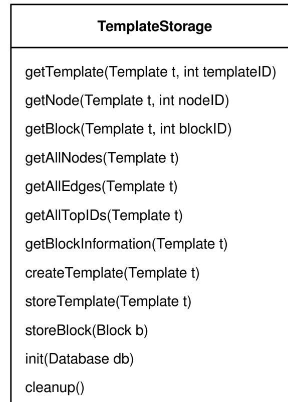


Abbildung 43: `TemplateStorage`-Schnittstelle

In Abschnitt 3.4.1 wurde festgestellt, dass die topologischen IDs der Knoten vom Schema verwaltet werden müssen, da sich diese bei Einsatz einer Deltaschicht auch bei Knoten ändern, die selbst nicht geändert wurden. Bei der Implementierung des Sekundärspeichers für Schemata kann hiervon allerdings abgesehen werden, da diese nur Schemata und keine Deltaschichten speichert und somit keine Überlagerung stattfindet. Hier können die topologischen IDs bei den Knoten gespeichert werden.

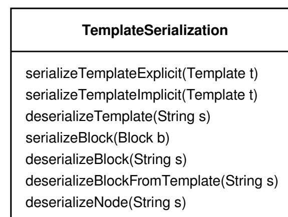


Abbildung 44: `TemplateSerialization`-Schnittstelle

Die in Abbildung 44 dargestellte Schnittstelle `TemplateSerialization` erlaubt das Serialisieren und Deserialisieren von einzelnen Blöcken und ganzen Schemata. Ebenso können einzelne Knoten, welche aus einer serialisierten Zeichenkette extrahiert wurden, deserialisiert werden. Für diese Schnittstelle existiert `XMLTemplateSerialization` als implementierende Klasse, welche XML zur De-/Serialisierung benutzt. Hierbei gibt es eine XML-Repräsentation mit expliziter und eine mit impliziter Blockstruktur. Die explizite besitzt hierbei einen `<template>`-Tag, welcher neben der ID des Schemas (`<id>`-Tag) mehrere `<block>`-Tags beinhaltet. Diese sind aus den Knoten (`<node>`-Tags) und Kanten (`<edge>`-Tags) und der jeweiligen ID (`<id>`-Tag) der einzelnen Blöcke aufgebaut. Die implizite Repräsentation beinhaltet keine `<block>`-Tags, sondern nur `<node>`- und `<edge>`-Tags. Die Unterscheidung zwischen expliziter und impliziter Repräsentation wird durch ein Attribut des umschließenden `<template>`-Tags getroffen. Dieser besitzt das

Attribut `type`, welches entweder `explicit` oder `implicit` ist. Dies ist in Abbildung 45 beispielhaft illustriert.

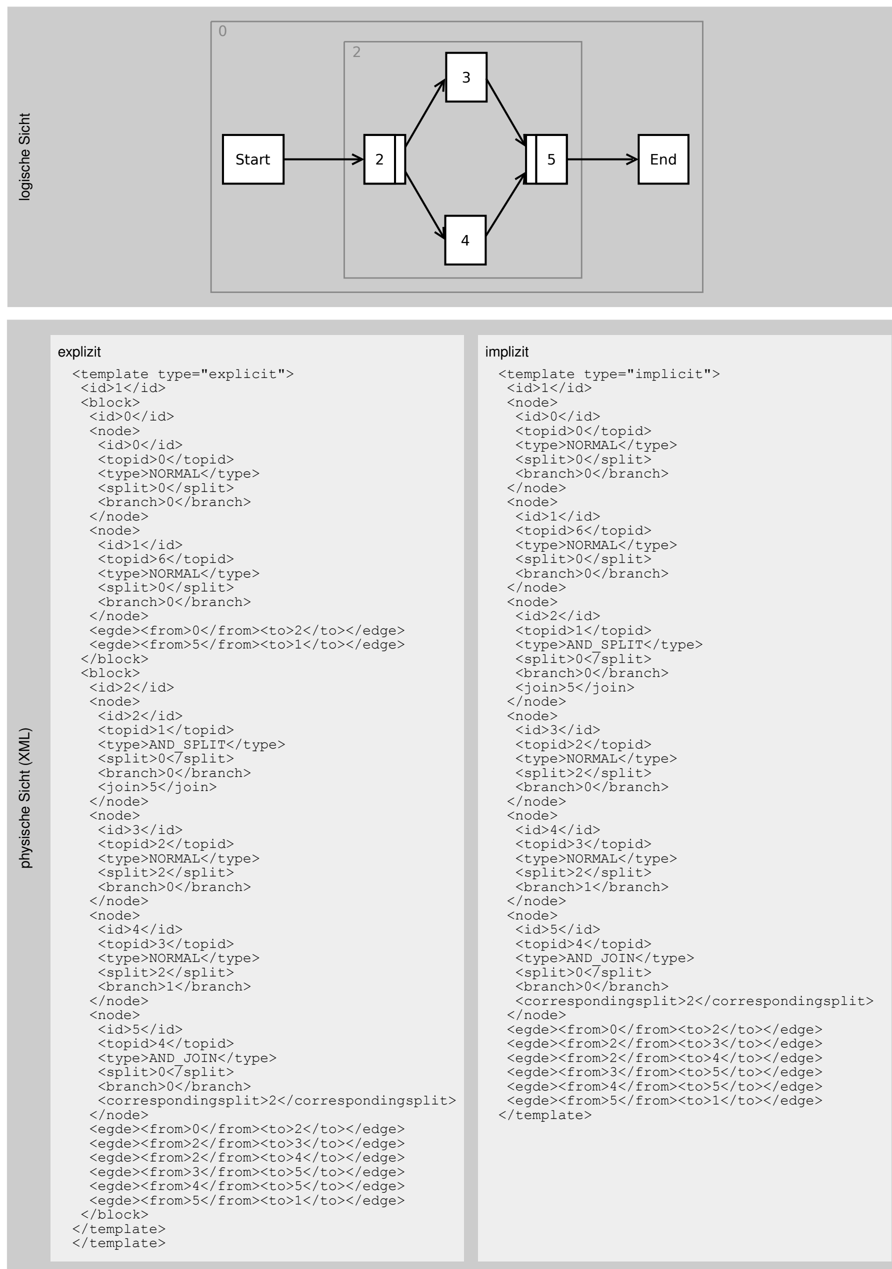


Abbildung 45: XML-Repräsentationen von Schemata

4.3.3.1 XML mit expliziter Blockstruktur

Die vorgestellte Serialisierung von Schemata als XML mit expliziter Blockstruktur lässt sich zur Persistierung der Daten benutzen.

Hierzu wird diese in einer Datei gespeichert, wobei der Dateiname aus der ID des Schemas gebildet wird. Dies ist sinnvoll, da für eine Deserialisierung die ID des Schemas vorhanden ist (vergleiche `TemplateStorage`-Schnittstelle) und der zugehörige Dateiname somit in konstanter Zeit bestimmt werden kann.

Dynamisches Nachladen einzelner Blöcke aus Schemata wird von dieser Sekundärspeicherrepräsentation nicht effizient unterstützt, da die Informationen zusammen mit allen anderen Daten des Schemas in einer einzigen Datei vorliegen. Hierbei muss zumindest ein Teil der nicht benötigten Daten geparst werden: Kommt ein SAX-Parser [MB02] zum Einsatz, so kann nach dem Auffinden des zu ladenden Blocks abgebrochen werden, nachdem dieser deserialisiert wurde. Für die Anfrage nach der Zuordnung der Knoten-IDs zu den Block-IDs muss allerdings das ganze Schema geparst werden, da alle Knoten einzeln untersucht werden müssen.

4.3.3.2 XML mit impliziter Blockstruktur Analog zur Speicherung mit einer expliziten Blockstruktur, wird auch bei der Speicherung mit einer impliziten Blockstruktur ein Schema von `XMLTemplateSerialization` serialisiert und anschließend in eine Datei gespeichert. Diese hat ebenfalls einen Dateinamen, welcher aus der ID des Schemas besteht.

Das dynamische Nachladen von Blöcken wird hierbei nicht effizient unterstützt: Es muss jeweils das ganze Schema deserialisiert werden und der angefragte Block identifiziert und zurückgeliefert werden. Gleiches gilt für die Anfrage nach der Auflösung von Knoten-ID nach Block-ID.

4.3.3.3 RDBMS mit expliziter Blockstruktur Die explizite Blockstruktur aus Abschnitt 3.2.3.1 lässt sich im Sekundärspeicher auch mithilfe von RDBMS umsetzen. Ziel ist es, alle Informationen zusammen zu speichern, welche zu einem Block gehören. Hierbei lassen sich die Daten serialisiert speichern und die von der `TemplateSerialization`-Schnittstelle definierten Methoden wiederverwenden: Es existiert eine Tabelle `Block`, welche die ID des Blocks und den ganzen Block in serialisierter Form vorhält. Die Serialisierung wird in einer Spalte gespeichert, welche zeichenbasierte große Objekte („Character Large Object“, „CLOB“ [IBM09b, Ora09b] bzw. „TEXT“ [Pos09b]) halten kann.

Zusätzlich zur Tabelle `Block` existiert eine Tabelle `Template`, welche in einer 1:n-Beziehung zu `Block` steht. Diese bildet auf logischer Ebene ein vollständiges Schema ab und wird für die Vollständigkeit der Repräsentation benötigt.

Diese Repräsentation von Schemata im Sekundärspeicher unterstützt das dynamische Nachladen von Blöcken effizient, da die gewünschten Informationen direkt zugänglich vorliegen. Um der Primärspeicherrepräsentation allerdings für die Knoten die zugeordneten Block-IDs liefern zu können, wird eine weitere Tabelle (`BlockInfo`) benötigt. Wäre diese nicht vorhanden, so müsste bei einer entsprechenden Anfrage das ganze Schema

geladen und somit alle Blöcke deserialisiert werden. Da dies unverhältnismäßig und eine effizientere Lösung mit wenig Aufwand realisierbar ist, hält die Tabelle `BlockInfo` die benötigten Informationen.

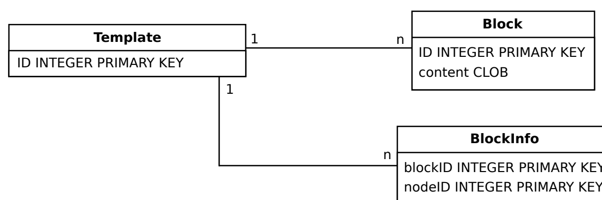


Abbildung 46: Schemata mit expliziter Blockstruktur im RDBMS

Neben der vorgestellten Repräsentation ist eine zweite möglich. Die eingesetzten RDBMS unterstützen das Speichern von XML-Daten mit einem gesonderten XML-Datentyp, also nicht CLOB. Dies hat den Vorteil, dass an das RDBMS gestellte Anfragen auch auf Daten zugreifen können, welche in einer Spalte mit serialisierten Daten persistiert sind. Somit wird die Tabelle `BlockInfo` überflüssig. Diese zweite Repräsentation mit expliziter Blockstruktur ist in Abbildung 47 dargestellt.

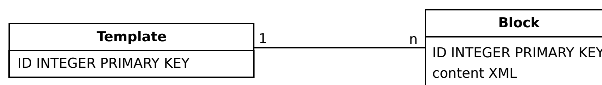


Abbildung 47: Schemata mit expliziter Blockstruktur in XML im RDBMS

4.3.3.4 RDBMS mit impliziter Blockstruktur Bei einer impliziten Realisierung der Blockstruktur werden analog zur Primärspeicherrepräsentation aus Abschnitt 4.2.2 alle Knoten in einer Tabelle `Node` gehalten. Diese stehen von einer Tabelle `Template` ausgehend in einer 1:n-Beziehung. Gleiches gilt für die Tabelle `Edges`, welche die Kanten eines Schemas beinhaltet. Dies ist in Abbildung 48 dargestellt.

Das dynamische Nachladen von Blöcken wird von dieser Sekundärspeicherrepräsentation unterstützt. Hierfür werden bei einer Anfrage nach einem Block die zugehörigen Knoten aus der Tabelle `Node` und die benötigten Kanten aus der Tabelle `Edges` geladen und zu einem `Block`-Objekt zusammengefasst. Die Auflösung von Knoten-IDs zu Block-IDs ist durch eine Abfrage der Spalten `ID` und `splitNodeID` der Tabelle `Node` möglich.

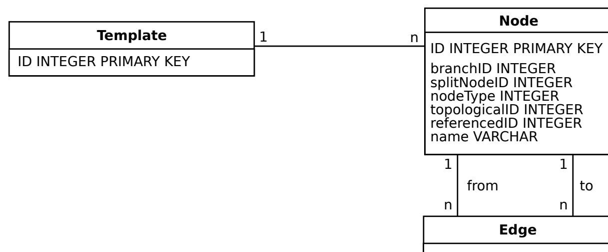


Abbildung 48: Schemata mit impliziter Blockstruktur im RDBMS

4.3.3.5 JPA Bei einer Speicherung der Daten mithilfe von JPA werden die im Primärspeicher vorhandenen, zu persistierenden Klassen mit Annotationen versehen. Aus diesen generiert die Implementierung von JPA, ein Datenbanklayout. Da im Primärspeicher zwei Repräsentationen von Schemata verfügbar sind (vergleiche Abschnitt 4.2.2), existieren auch für JPA diese beiden Repräsentationen: Schemata mit expliziter Blockstruktur und mit impliziter.

Die *explizite Repräsentation* aus Abbildung 49 ist vergleichbar mit der Speicherung in einem RDBMS mit expliziter Blockstruktur: Ein Schema enthält mehrere Blöcke und diese enthalten die Knoten, Kanten und topologischen IDs in serialisierter Form als zeichenbasierte große Objekte. Hierbei geschieht die Serialisierung über die `Externalizable`-Schnittstelle von Java™ . Es kann keine Implementierung von `TemplateSerialization` zur Serialisierung herangezogen werden, da JPA die Serialisierung intern steuert und die in Java™ mitgelieferte Serialization-API anspricht, welche durch `Externalizable` gesteuert werden kann. Dies hat zur Folge, dass im RDBMS, nicht pures XML gespeichert wird, sondern dieses noch einen Serialisations-Header besitzt. Dieser beinhaltet eine Beschreibung der serialisierten Klasse.

Das dynamische Nachladen von Blöcken wird bei Einsatz der expliziten Blockstruktur unterstützt. Hierbei kommt die von JPA definierte *Query Language* [Sun06] zum Einsatz, welche ähnlich zu SQL [ISO92] ist. Mithilfe dieser Sprache ist es möglich, einzelne `Block` -Objekte zu laden. Hierbei ist die Auflösung von Knoten-IDs zu Block-IDs allerdings nur durch Laden aller Blöcke zu erreichen, da die Knoteninformationen in der Tabelle `Block` serialisiert gespeichert werden.

Abbildung 49: Schemata mit expliziter Blockstruktur mit JPA

Wie bei der Speicherung in einem RDBMS mit impliziter Blockstruktur, repräsentiert auch die JPA-Variante mit *impliziter Blockstruktur* ein Schema durch eine Knotenmenge, welche in einer separaten Tabelle gespeichert wird (vergleiche Abbildung 50). Die

Kantenmenge und die Menge der topologischen IDs hingegen werden hier serialisiert gespeichert, da JPA keine Unterstützung bietet, eine `Map`, welche als Wert ein `Set` oder nur eine Zahl hat, in einer eigenen Tabelle darzustellen.

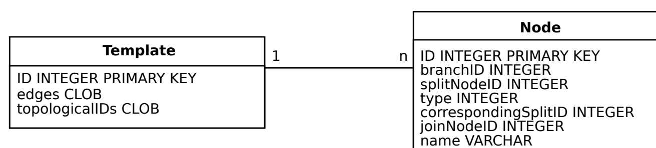


Abbildung 50: Schemata mit impliziter Blockstruktur mit JPA

Durch Einsatz der *Query Language* von JPA lassen sich bei der impliziten Realisierung die Knoten leicht bestimmen, welche zu einem bestimmten Block gehören, da die Split-Knoten-ID als Feld in der Tabelle `Node` vorhanden ist. Die Zuordnung der Kanten und der topologischen IDs gestaltet sich allerdings schwierig, da diese als zeichenbasiert große Objekte in der Tabelle `Template` vorliegen. Hierfür muss somit das ganze Schema geladen werden. Für die Auflösung von Knoten-ID zu Block-ID muss das entsprechende Knoten-Objekt eingelagert werden, was mit der JPA Query Language möglich ist. Die Block-ID liegt dann in Form der Split-Knoten-ID des `Node`-Objekts vor.

4.3.4 Instanzen

Wie bei Schemata ergeben sich für Instanzen verschiedene Repräsentationen im Sekundärspeicher. Implementierungen zur Speicherung der Daten als XML-Dateien, in einem RDBMS und eine Realisierung mithilfe von JPA existieren als Implementierungen der `InstanceStorage`-Schnittstelle, welche in Abbildung 51 dargestellt ist.

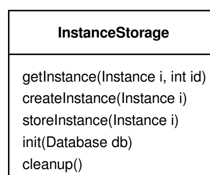


Abbildung 51: `InstanceStorage`-Schnittstelle

Alle Implementierungen von `InstanceStorage` benötigen hierzu eine `TemplateStorage`. Von einer Instanz kann direkt das zugehörige Schema abgerufen werden. Deshalb wird zum Beispiel beim Laden einer Instanz das entsprechende Objekt benötigt, welches von der zur Verfügung stehenden `TemplateStorage` geladen wird. Eine weitere Möglichkeit besteht darin, das korrekte Schema beim Laden einer Instanz der Methode zu übergeben, womit ein erneutes Laden umgangen wird. Die `InstanceStorage`-Schnittstelle stellt Methoden zur Verfügung um Instanzen zu laden, speichern oder neu anzulegen.

Analog zu Schemata ist es aufgrund der Einführung der `Instance`-Schnittstelle auch bei Instanzen möglich, diese in einer Repräsentation mit expliziten Knotenzuständen im Primärspeicher zu halten, während im Sekundärspeicher eine Repräsentation mit impliziten Knotenzuständen vorliegt. Ebenso ist es möglich, die Instanz im Primärspeicher implizit und im Sekundärspeicher explizit zu halten. Wie auch schon bei Schemata gilt auch für Instanzen bei Einsatz von JPA als Sekundärspeichervariante, dass die Sekundärspeicherrepräsentation immer der des Primärspeichers entspricht. Abbildung 52 illustriert dies.

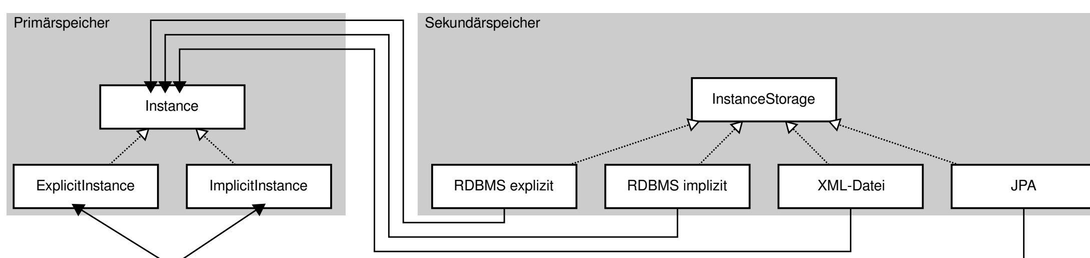


Abbildung 52: Instanzrepräsentation im Primär- und Sekundärspeicher

Für die Serialisierung von Instanzen sorgt, analog zu `TemplateSerialization` für Schemata, die Schnittstelle `InstanceSerialization` (vergleiche Abbildung 53). Diese bietet die Möglichkeit, eine Instanz oder eine Menge von Knotenzuständen zu de-/serialisieren. Es existiert mit `XMLInstanceSerialization` eine Implementierung, welche XML zur De-/Serialisierung benutzt. Hierbei stehen wiederum Methoden für eine explizite und eine implizite XML-Repräsentation zur Verfügung. Die explizite Repräsentation beinhaltet in einem `<instance>`-Tag die ID der Instanz (`<id>`-Tag) und die des zugrunde liegenden Schemas (`<templateid>`-Tag). Zusätzlich werden die Knotenzustände aller Knoten innerhalb von `<nodestate>`-Tags gehalten. Diese beinhalten die Knoten-ID (`<node>`-Tag) und den zugehörigen Knotenzustand (`<state>`-Tag).

InstanceSerialization
serializeInstanceExplicit(Instance i) serializeInstanceImplicit(Instance i) deserializeInstance(String s) serializeNodeStatesExplicit(Map<int, State> nodeStates) serializeNodeStatesImplicit(Set<int> nodesRunning, Set<int> nodesSkipped) deserializeNodeStatesExplicit(String s) deserializeNodeStatesImplicit(String s)

Abbildung 53: `InstanceSerialization`-Schnittstelle

Die implizite XML-Repräsentation gleicht der expliziten, nur dass nicht für alle Knoten ein Knotenzustand vorhanden ist. Es werden nur RUNNING und SKIPPED Zustände gespeichert. Die Unterscheidung der beiden Repräsentationen findet durch ein Attribut

des umschließenden `<instance>`-Tags statt: Dieser besitzt ein Attribut `type` welches als Wert entweder `explicit` oder `implicit` besitzt. Dies ist in Abbildung 54 dargestellt.

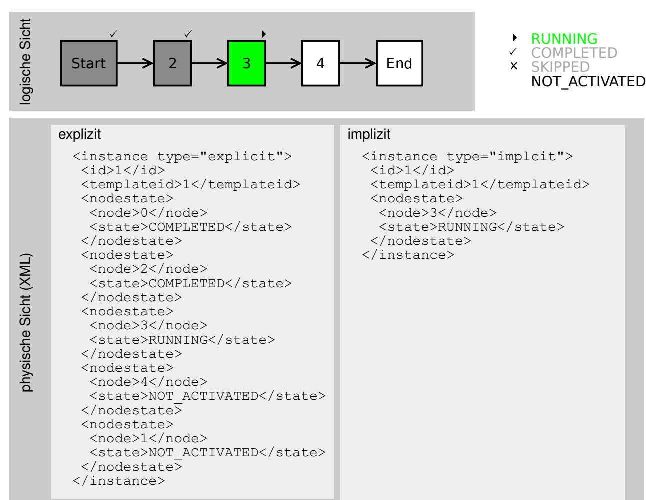


Abbildung 54: XML-Repräsentationen von Instanzen

4.3.4.1 XML mit expliziten Knotenzuständen Analog zu den XML-Repräsentationen von Schemata wird auch die explizite XML-Repräsentation für eine Instanz von `XMLInstanceSerialization` erzeugt. Diese Serialisierung wird dann in einer Datei mit der Instanz-ID als Dateinamen gespeichert. Dies ist sinnvoll, da hiermit Anfragen, eine Instanz auf Basis ihrer ID zurück zu liefern, am schnellsten beantwortet werden können.

4.3.4.2 XML mit impliziten Knotenzuständen Für die implizite XML-Repräsentation von Instanzen gilt ebenso, dass die Instanzen von `XMLInstanceSerialization` in eine Repräsentation mit impliziten Knotenzuständen de-/serialisiert und in Dateien gespeichert werden. Hierbei kommt ebenfalls die Instanz-ID als Dateiname zum Einsatz.

4.3.4.3 RDBMS mit expliziten Knotenzuständen Bei der Speicherung einer Instanz mit expliziten Knotenzuständen in einem RDBMS ist es sinnvoll, die Knotenzustände in serialisierter Form zu halten, da dies für jeden Knoten nur ein zugeordneter Wert ist. Eine zusätzliche Tabelle hierfür ist nicht sinnvoll. Somit werden bei einer expliziten Repräsentation alle Knotenzustände von `XMLInstanceSerialization` serialisiert und in der Spalte für zeichenbasierte große Objekte der Tabelle `Instance` gespeichert. Hinzu kommen Felder für die ID der Instanz und die ID des zugrunde liegenden Schemas.

Instance
ID INTEGER PRIMARY KEY templateID INTEGER nodeStates CLOB

Abbildung 55: Instanzen mit expliziten Knotenzuständen im RDBMS

Auch hier ist es, analog zu in RDBMS gespeicherten Schemata mit expliziter Blockstruktur, möglich, die serialisierten Daten in einer Spalte des XML-Datentyps zu speichern. Hierdurch sind Abfragen auch auf die serialisierten Daten möglich.

Instance
ID INTEGER PRIMARY KEY templateID INTEGER nodeStates XML

Abbildung 56: Instanzen mit expl. Knotenzuständen in XML im RDBMS

4.3.4.4 RDBMS mit impliziten Knotenzuständen Wie schon für die Speicherung in einem RDBMS mit expliziten Knotenzuständen ist es auch bei dieser Variante zu aufwändig, eine eigene Tabelle für die impliziten Knotenzustände zu erstellen. Diese werden auch in serialisierter XML-Form gehalten, deren Umwandlung von `XMLInstanceSerialization` zur Verfügung gestellt wird. Hinzu kommen Felder für die ID der Instanz und die ID des zugrunde liegenden Schemas, wie Abbildung 57 zeigt.

Instance
ID INTEGER PRIMARY KEY templateID INTEGER nodesStatesImplicit CLOB

Abbildung 57: Instanzen mit impliziten Knotenzuständen im RDBMS

4.3.4.5 JPA Analog zu den Betrachtungen aus Abschnitt 4.3.4.3 und 4.3.4.4 wurden die Annotationen für eine Speicherung mithilfe von JPA so gewählt, dass die beiden zur Verfügung stehenden Primärspeicherrepräsentationen aus Abschnitt 4.2.4 im Sekundärspeicher mithilfe von serialisierten Knotenzustandsmengen repräsentiert werden. Hierbei ist, analog zur JPA-Repräsentation von Schemata, zu beachten, dass die Serialisierung durch die Implementierung der `Externalizable`-Schnittstelle realisiert wird, da JPA die von Java™ zur Verfügung gestellte Serialization-API verwendet und eine Implementierung von `InstanceSerialization` nicht benutzt werden kann.

Instance
ID INTEGER PRIMARY KEY templateID INTEGER nodeStates CLOB

Abbildung 58: Instanzen mit expliziten Knotenzuständen mit JPA

Instance
ID INTEGER PRIMARY KEY templateID INTEGER nodesRunning CLOB nodesSkipped CLOB

Abbildung 59: Instanzen mit impliziten Knotenzuständen mit JPA

4.3.5 Zwischenspeicher

Es ist unter Umständen sinnvoll, die Implementierungen der Sekundärspeicher-Schnittstellen mit Zwischenspeichern (*Caches*) im Hauptspeicher zu versehen. Für die restliche Workflow-Testumgebung soll dies transparent geschehen, deshalb bieten sich Implementierungen von `TemplateStorage` und `InstanceStroage` an. Diese, als *Caching Storage* bezeichneten Klassen, greifen auf eine der bereits vorgestellten Implementierungen zurück, um die Entitäten persistent zu speichern. Bei einem Abruf der Entität wird diese aus der zugrunde liegenden Implementierung abgerufen und zwischengespeichert. Bei einer erneuten Anfrage nach dieser Entität wird diejenige aus dem Zwischenspeicher zurückgeliefert und keine neue von der zugrunde liegenden Implementierung angefragt. Dies spart den Zugriff auf das RDBMS oder das Dateisystem und beschleunigt somit den Vorgang. Abbildung 60 illustriert dies.

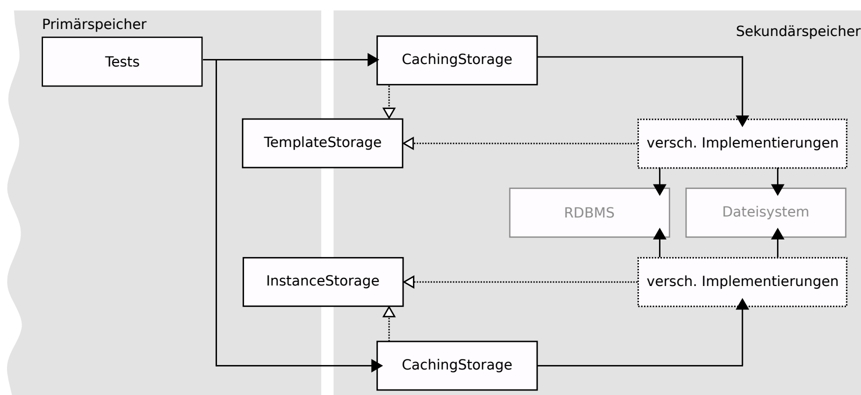


Abbildung 60: Caching Storage

Es ist sinnvoll, die Größe des Zwischenspeichers zu begrenzen, da der Hauptspeicher (vergleiche Abbildung 16) im Extremfall sonst vollständig mit Schema- und Instanzdaten gefüllt ist. Dieser wird allerdings zum Beispiel auch für die Programmausführung benötigt und sollte deshalb nicht vollständig mit diesen Daten gefüllt werden. Aufgrund dessen bietet sich ein so genannter Least-Recently-Used-Cache (*LRU-Cache*) [OOW93] an. Hat dieser seinen maximalen Füllstand erreicht, wird die am längsten nicht mehr benötigte Entität aus dem Zwischenspeicher gelöscht. Sollte diese später wieder angefragt werden, so wird diese erneut durch die der Caching Storage zugrunde liegende Implementierung geladen und wieder in den Zwischenspeicher eingefügt.

Wird die Caching Storage angewiesen, eine Entität zu speichern, muss diese im Zwischenspeicher invalidiert und die zugrunde liegende Implementierung angewiesen werden, die neue Entität zu persistieren.

Aufgrund der Definitionen aus Abschnitt 4.3.2 zählt auch eine Caching Storage korrekterweise zum Sekundärspeicher.

4.3.6 Flüchtiger Speicher

Um auch Messungen durchführen zu können, welche ausschließlich auf den Hauptspeicher zugreifen, bieten sich weitere Implementierungen der Schnittstelle `TemplateStorage` und `InstanceStorage` an. Hierdurch kann auch der flüchtige Speicher transparent von der Workflow-Testumgebung angesprochen werden. Die Implementierungen der Storage-Schnittstellen, welche *Transient Storage* genannt werden, benutzen Standard-Datenstrukturen von Java™ zur Verwaltung der Daten im Hauptspeicher. Hierbei ist es nicht sinnvoll, eine Transient Storage mit einer Caching Storage zu verbinden, da beide die Daten im Hauptspeicher halten. Hier reicht eine Transient Storage aus.

Auch eine Transient Storage gehört nach Abschnitt 4.3.2 korrekterweise zum Sekundärspeicher.

4.3.7 Zusammenfassung

Es wurden verschiedene Realisierungsvarianten zur Speicherung von Schemata und Instanzen im Sekundärspeicher vorgestellt. Nachdem die Begriffe Primär- und Sekundärspeicher aufgrund von Ungenauigkeiten präzisiert wurden, folgte die Erläuterung der verschiedenen Repräsentationen. Hierbei wurden XML-Serialisierungen vorgestellt und Möglichkeiten aufgezeigt, die Entitäten in RDBMS und mit Hilfe von JPA zu persistieren. Aufbauend hierauf wurden Caching Storages und Transient Storages eingeführt.

Mithilfe dieser Ergebnisse ist es nun möglich, einen Prozess vollständig in Primär- und Sekundärspeicher zu beschreiben. Dies umfasst die grundlegenden Betrachtungen aus Kapitel 2 und die hierauf basierenden Konzepte aus Kapitel 3.

4.4 Anfragen auf Kollektionen

Zum Abschluss der Umsetzung fehlt noch die Eingliederung der Anfragen auf Kollektionen in die zuvor ausgearbeitete Architektur der Workflow-Testumgebung. Dies wird in diesem Abschnitt erläutert.

4.4.1 Instanzen eines Schemas

Um die Auslastung der Prozesslinien (vergleiche Abbildung 15) abschätzen zu können, ist es hilfreich, die Instanzen eines bestimmten Schemas abrufen zu können. Hierfür wird eine Methode benötigt, welche die entsprechenden Instanzen finden kann. Diese muss Bestandteil der `InstanceStorage`-Schnittstelle werden, da die Implementierungen von diesem über alle nötigen Informationen verfügen.

Um diese Anfrage zu beantworten, benötigen die Implementierungen, welche XML benutzen lineare Laufzeit auf der Anzahl der Instanzen. Es müssen *alle* verfügbaren Dateien im Dateisystem untersucht werden, ob die jeweilige Instanz zum angegebenen Schema gehört. Dies ist sehr aufwändig.

Die Implementierungen auf Basis von RDBMS können diese Anfrage jedoch effizient beantworten. Hier steht die Anfragesprache SQL zur Verfügung, mit der Abfragen auf die `templateID`-Spalte der Tabelle `Instance` möglich ist. Diese Spalte ist in allen RDBMS-Repräsentationen von Instanzen vorhanden.

Auch die Implementierung mit JPA bietet eine performante Beantwortung der Anfrage. Die JPA *Query Language* bietet die Möglichkeit, die Instanzen aufgrund der ID eines Schemas zu laden.

4.4.2 Instanzen mit bestimmten Knotenzustand

Um die Auslastung einzelner Abteilungen eines Unternehmens einschätzen zu können, ist die Abfrage nach Instanzen eines Schemas mit einem bestimmten Knoten in einem bestimmten Zustand hilfreich (vergleiche Abschnitt 2.4.3). Auch diese Anfrage muss von `InstanceStorage` abgedeckt werden.

Wie bei der Abfrage nach den Instanzen eines Schemas können die XML-Implementierungen auch diese Anfrage nicht performant beantworten. Es müssen ebenso *alle* Instanzen untersucht werden.

Bei Instanzen, welche in einem RDBMS gespeichert sind, werden die Knotenzustände als ein zeichenbasiert großes Objekt gespeichert. Dies hat zur Folge, dass hier nicht effizient mithilfe von SQL selektiert werden kann. Es müssen ebenfalls *alle* Instanzen geladen und untersucht werden.

Hier schafft der Datentyp XML der eingesetzten RDBMS Abhilfe. Mithilfe von diesem können Anfragen auch auf diejenigen Spalten gestellt werden, in denen Daten in serialisierter Form vorliegen. Die zu stellenden Anfragen sind hierbei abhängig vom RDBMS, da diese keinen einheitlichen Standard implementieren. Der Einsatz eines XPath-Ausdrucks [W3C99] zur Selektion der Daten aus der gespeicherten XML-Zeichenkette ist allen untersuchten RDBMS allerdings gemein. Eine beispielhafte Anfrage wird in Listing 1 vorgestellt. Diese wird an DB2 gerichtet und basiert auf der Annahme, dass die Instanzen mit expliziten Knotenzuständen vorgehalten werden (siehe Abbildung 56). Als Parameter

werden die ID des Schemas und des Knotens sowie der gesuchte Knotenzustand übergeben. Die Anfrage erzeugt über die in `i.nodeStates` gespeicherte XML-Zeichenkette eine virtuelle Tabelle. Diese besitzt die Spalten `nodeid` und `nodestate`, welche durch Parsen aller Ergebnisse der XPath Anfrage `/nodestates` erzeugt werden. Diese virtuelle Tabelle kann dann im `WHERE`-Teil der Anfrage benutzt werden und die Selektion der Instanzen auf die beschränkt werden, welche den entsprechenden Knoten im korrekten Zustand haben.

```
FROM
  Instance AS i,
  XMLTABLE('$d/nodestate'
    passing i.nodeStates as "d"
    COLUMNS
      nodeid INTEGER PATH 'node',
      nodestate VARCHAR(20) PATH 'state') AS x
WHERE
  i.templateID = ? AND
  x.nodeid = ? AND
  x.nodestate = ?
```

Listing 1: Anfrage an DB2 mit Auswertung der XML-Daten

Durch den Einsatz des XML-Datentyps und der Möglichkeit, die hiermit gespeicherten Daten in Abfragen einzubinden, kann die Anfrage effizient beantwortet werden. Sollte die RDBMS-Repräsentation die Knotenzustände allerdings implizit vorhalten und nicht nach dem Knotenzustand RUNNING oder SKIPPED gesucht werden, ist eine effiziente Beantwortung auch hier nicht möglich. Der Sekundärspeicher kann die Knotenzustände bei einer impliziten Speicherung nicht berechnen, da der hierfür notwendige Algorithmus nur im Primärspeicher vorliegt. Hierbei ist die Umsetzung des Algorithmus in *Stored Procedures* denkbar. Dies wurde allerdings im Rahmen dieser Arbeit nicht umgesetzt, da dies den Rahmen sprengen würde.

Bei einer Umsetzung mittels JPA lässt sich diese Anfrage ebenfalls nicht performant beantworten, da hier die Knotenzustände ebenfalls in serialisierter Form gespeichert werden. Hierbei besteht allerdings kein Zugriff auf den Datentyp der entsprechenden Spalte im zugrunde liegenden RDBMS.

4.4.3 Zusammenfassung

Es wurde die Kompatibilität der einzelnen Sekundärspeicherrepräsentationen zu den Anfragen auf Kollektionen untersucht, wie sie in Abschnitt 2.4.3 definiert wurden. Hierbei stellte sich heraus, dass die Anfrage nach Instanzen mit einem bestimmten Knoten in einem bestimmten Zustand besser beantwortet werden kann, sofern die Knotenzustände in einer Spalte mit XML-Datentyp gespeichert werden, wenn mithilfe eines RDBMS gespeichert wird. Außerdem wurde festgestellt, dass die Repräsentation von Instanzen im Dateisystem mithilfe von XML keine performante Beantwortung der Anfragen zulässt.

4.5 Zusammenfassung

In diesem Kapitel wurde die Umsetzung einer Workflow-Testumgebung auf Basis der Grundlagen aus Kapitel 2 erläutert. Diese beinhaltet die Konzepte aus Kapitel 3. Es wurden verschiedene Repräsentationen der Entitäten sowohl für den Primär- als auch den Sekundärspeicher vorgestellt. Als Sekundärspeicher kommt hierbei eine Speicherung in serialisierter Form im Dateisystem, eine Speicherung in einem RDBMS und eine Speicherung mithilfe von JPA in Frage. Abschließend wurden Anfragen auf Kollektionen und deren Umsetzung auf Basis der verschiedenen Sekundärspeicherrepräsentationen betrachtet. Dies rundet die Umsetzung der Workflow-Testumgebung ab. Es ist eine grundlegende Funktionalität gegeben: Es können Schemata und darauf basierende Instanzen definiert werden. Es besteht die Möglichkeit, diese instanzbasiert zu ändern und weiterzuschalten. Hierbei sind jeweils verschiedene Speicherrepräsentationen vorgestellt worden, welche im Folgenden nach der qualitativen Betrachtung aus Kapitel 3 auch quantitativ untersucht werden können.

5 Messungen

Nachdem die Umsetzung der Grundlagen und Konzepte sowohl in Primär- als auch in Sekundärspeicherstrukturen vorgestellt wurde, schließt sich in diesem Kapitel die Vorstellung und Interpretation der damit durchgeführten Zeit- und Speichermessungen an. Hierbei wird zuerst auf die Rahmenbedingungen und dann auf die Messergebnisse der verschiedenen Sekundärspeicherrepräsentationen eingegangen. Im Anschluss folgt die Auswertung der Messungen zur Primärspeicherrepräsentation. Das Fazit dieser Arbeit schließt dieses Kapitel ab.

5.1 Rahmenbedingungen

In diesem Abschnitt werden die Rahmenbedingungen der Messungen vorgestellt. Zuerst wird der Rechner beschrieben, auf dem diese durchgeführt wurden. Um die Vergleichbarkeit zu wahren, müssen die ermittelten Zeitwerte allerdings unabhängig vom Rechner sein. Aufgrund dessen wird anschließend der in dieser Arbeit gewählte Weg beschrieben, die Eigenschaften des Rechners aus den Messergebnissen „herauszurechnen“. Dies ist für die gemessenen Werte bezüglich des Speicherverbrauchs nicht notwendig, da diese unabhängig vom eingesetzten Rechner sind, wenn man von der Bitbreite der Architektur absieht (32 Bit und 64 Bit).

Der Rechner, auf dem die Messungen durchgeführt wurden, besitzt einen 32 Bit AMD Athlon™ XP 2800+ Prozessor [Adv09]. Dieser ist mit 2083 Megahertz getaktet und greift auf einen Hauptspeicher von 1 Gigabyte Größe zu. Um die Messungen nicht zu verfälschen wurde kein Auslagerungsspeicher eingerichtet, der zur Erweiterung des Hauptspeichers durch Einsatz der Festplatte dienen würde. Als Betriebssystem kommt Ubuntu Linux [Can09] in der Version 9.04 zum Einsatz.

Um die Messergebnisse vergleichen zu können, müssen die im Folgenden vorgestellten Zeitwerte allerdings unabhängig vom eingesetzten Rechner sein. Hierzu wird die Performanz von diesem gemessen und aus den absoluten Ergebnissen der Messungen durch Division herausgerechnet. Dieser *Grundwert* des Rechners wird im Rahmen dieser Arbeit dadurch bestimmt, dass die Dauer eines bestimmten Schleifendurchlaufs gemessen wird. Der hierfür eingesetzte Java™ -Quelltext ist in Listing 2 vorgestellt.

```
1 int i = 0;
2 long after;
3 long before = System.nanoTime();
4 while (i < 690000) i++;
5 after = System.nanoTime();
```

Listing 2: Quelltext zur Performanzbestimmung

Es wird die Zeit gemessen, welche für eine 690.000 malige Erhöhung einer Zählvariable benötigt wird. Hierfür kommt die Methode `System.nanoTime()` zum Einsatz, welche die

aktuelle Systemzeit in 10^{-9} Sekunden liefert. Der resultierende Grundwert ist durch die Subtraktion `after-before` verfügbar.

Diese Messung wurde auf dem zur Verfügung stehenden Rechner 1.000 mal direkt hintereinander durchgeführt. Es fiel auf, dass sich der resultierende Wert bei den ersten sieben aufeinander folgenden Messungen um mehr als das 200-fache von den folgenden Werten unterscheidet. Die Ergebnisse wurden um diese ersten sieben Werte und um „Ausreißer“ bereinigt und der Durchschnitt der restlichen Werte berechnet. Hierbei stellte sich für den eingesetzten Rechner ein Grundwert von 1.0 Millisekunden heraus, wobei dieser auf die erste Nachkommastelle kaufmännisch gerundet ist.

Die im Folgenden präsentierten Zeitwerte sind die absolut gemessenen Werte in Millisekunden geteilt durch den Grundwert 1.0 des eingesetzten Rechners, sofern dies nicht anders angegeben ist.

Die kaufmännische Rundung auf die erste Nachkommastelle wurde im Allgemeinen auch bei den absoluten Werten aller Messungen im Folgenden durchgeführt. Bei Sekundärspeichermessungen ist es meist nicht sinnvoll, die Messungen genauer anzugeben, da in diesem Bereich „natürliche“ Schwankungen der Messergebnisse auftreten. Dies ist in der Komplexität der Rechnersysteme begründet. Hierbei befinden sich auf der eingesetzten Maschine gleichzeitig zur Messung auch noch andere Prozesse in der Ausführung (zum Beispiel Kernel- und Shell-Prozesse aber vor allem auch Prozesse des eingesetzten RDBMS). Dabei kann die „Umschaltung“ zwischen den Prozessen nicht beeinflusst oder gemessen – und somit aus dem Ergebnis herausgerechnet – werden. Die Werte sind ebenso abhängig von eingesetzten Zwischenspeichern, welche ebenfalls nicht beeinflusst werden können. Diese befinden sich zum Beispiel auf Software-Ebene im Kernel oder aber auch auf Hardware-Ebene (wie zum Beispiel Prozessor-Caches). Bei den Sekundärspeichermessungen wird nur in ausgewählten Fällen eine genauere Betrachtung vorgestellt. Bei Primärspeichermessungen sind die Werte in Tausendstel des Grundwertes angegeben, da hierbei kein weiterer zeitintensiver Prozess die Rechenzeit des Rechners benötigt, wie das durch die Prozesse des eingesetzten RDBMS bei Sekundärspeichertests der Fall ist. Es sei an dieser Stelle darauf hingewiesen, dass Werte, welche im Bereich von Tausendstel des Grundwertes liegen, zum Teil starken (relativen) Schwankungen unterliegen. Die Messergebnisse wurden dabei nach bestem Wissen und Gewissen ausgewertet, sollten allerdings dennoch eher als grobe Richtwerte angesehen werden.

Für die Messungen wurden drei unterschiedliche Schemata eingesetzt. Zum Einen das „einfache“ Schema, welches wenige Knoten und nur zwei Blöcke besitzt (vergleiche Abbildung 61). Auf der anderen Seite wurden zwei Schemata mit mehreren Knoten und Blöcken eingesetzt: Im Normalfall wurde das in Abbildung 62 vorgestellte „komplexe“ Schema benutzt. Da dieses relativ unübersichtlich ist, wurde für Tests, welche auf einer Menge von Instanzen arbeiten das „evolve“-Schema aus Abbildung 63 benutzt. Auf diesem können Instanzen leicht gleich auf die Knoten verteilt werden. Dies bedeutet, dass für jeden Knoten des Schemas genau eine Instanz existiert, welche diesen Knoten im Zustand RUNNING hat.

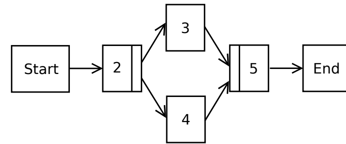


Abbildung 61: Das einfache Schema für die Messungen

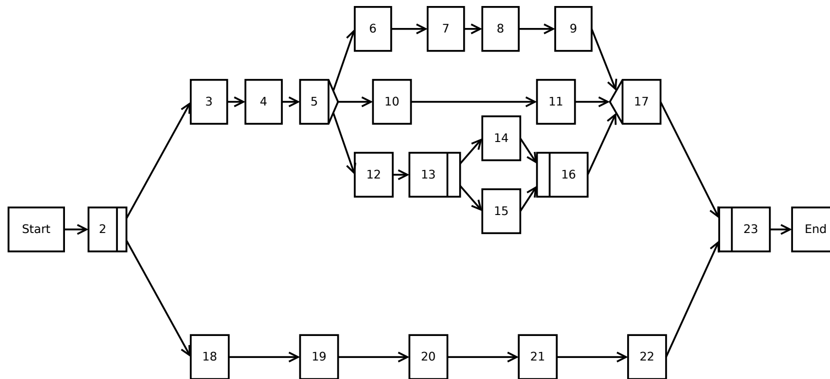


Abbildung 62: Das komplexe Schema für die Messungen

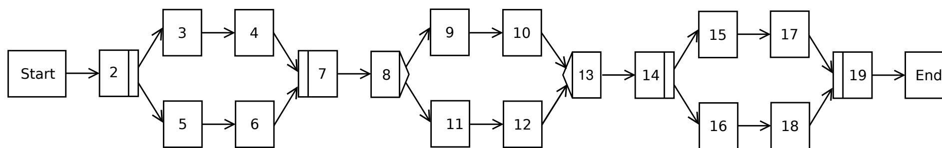


Abbildung 63: Das „evolve“-Schema für die Messungen

5.2 Sekundärspeichermessungen

Auf Basis der vorgestellten Rahmenbedingungen werden in diesem Abschnitt die Ergebnisse von Sekundärspeichermessungen vorgestellt. Diese gliedern sich in Messungen des Ein- und Auslagerns der Primärspeicherdaten und Anfragen auf Kollektionen. Auf diese Aspekte wird nach einer einführenden Beschreibung der durchgeführten Messungen auf Basis der einzelnen eingesetzten Sekundärspeichervarianten (XML, RDBMS und JPA) eingegangen. Ziel ist es, am Ende dieses Abschnittes sowohl eine optimale Sekundärspeicherrepräsentation als auch eine optimale Speichervariante durch einen Vergleich der einzelnen Varianten zu bestimmen.

5.2.1 Einführung

Dieser Abschnitt beschreibt die Testfälle, mit welchen die Ergebnisse im Folgenden gemessen wurden. Hierbei werden die Testfälle des Ein- und Auslagerns von Instanzen und Schemata betrachtet, sowie die einzelnen eingesetzten Anfragen auf Kollektionen vorgestellt.

Das Einlagern von Instanzen erfolgt auf direktem Weg: Es wird ein Objekt der Klasse `ExplicitInstance` oder `ImplicitInstance` erzeugt und die eingesetzte Implementierung der `InstanceStorage`-Schnittstelle angewiesen, dieses mit den entsprechenden Daten zu füllen. Für Schemata gilt folgende Vorgehensweise: Zuerst wird ein Objekt der Klasse `ExplicitTemplate` oder `ImplicitTemplate` erzeugt, welches dann durch die eingesetzte Implementierung von `TemplateStorage` initialisiert wird. Anschließend wird `ExplicitTemplate` oder `ImplicitTemplate` angewiesen, alle Daten zu laden. Hierbei werden verschiedene Methoden aus `TemplateStorage` aufgerufen, je nachdem ob die Daten im Sekundärspeicher explizit oder implizit vorgehalten werden.

Das Auslagern von Schemata und Instanzen ist trivial: Das Primärspeicherobjekt wird an die Implementierung von `TemplateStorage` bzw. `InstanceStorage` übergeben. Danach führt diese die Persistierung durch.

Bei der Auswertung der Ergebnisse des Ein- und Auslagerns muss zwischen Schemata und Instanzen unterschieden werden: Schemata werden im Normalfall nur ein mal gespeichert, dafür sehr häufig geladen. Dies liegt daran, dass Schemata nicht geändert werden. Sollte eine Änderung an einem Schema nötig werden, muss eine Schemaevolution durchgeführt werden und dadurch neben dem alten ein neues Schema angelegt werden. Somit ist die Zeit zum Einlagern von Schemata bei einer Interpretation stärker zu gewichten als die zum Auslagern. Bei Instanzen zeigt sich ein anderes Bild: Diese müssen in etwa gleich oft geladen wie gespeichert werden. Soll eine Instanz weitergeschaltet werden, wird diese aus dem Sekundärspeicher geladen, im Primärspeicher weitergeschaltet und wieder ausgelagert.

Neben dem Ein- und Auslagern von Schemata und Instanzen wurden auch Messungen von Anfragen auf Kollektionen durchgeführt. In Abschnitt 2.4.3 wurden zwei solcher Anfragen vorgestellt. Zum Einen müssen alle Instanzen eines gegebenen Schemas gefunden werden können. Diese Anfrage wird im Folgenden aus Platzgründen in den entsprechenden Diagrammen mit „Instanzen“ abgekürzt. Die zweite definierte Anfrage an Kollektionen sucht nach Instanzen, welche einen bestimmten Knoten in einem bestimmten Zustand haben. Diese Anfrage wurde für alle Sekundärspeichervarianten vier mal durchgeführt. Zuerst wurde nach dem Start-Knoten gesucht. Dieser sollte einmal im Zustand RUNNING und einmal im Zustand COMPLETED sein. Danach wurde nach dem End-Knoten gesucht und die gleichen Zustände untersucht. Die Ergebnisse sind in den folgenden Diagrammen jeweils passend betitelt („StartRunning“, „StartComplete“, „EndRunning“ und „EndComplete“).

Als Initialisierung dieser Anfragen auf Kollektionen wurden zwei Schemata erzeugt. Ein einfaches und eines vom Typ „evolve“ (vergleiche Abschnitt 5.1). Danach wurden für beide Schemata so viele Instanzen erzeugt, wie diese Knoten haben. Abschließend wurden die Instanzen mehrfach weitergeschaltet und gleich über die Knoten verteilt. Es existierte somit für jedes Schema und jeden Knoten genau eine Instanz, welche diesen Knoten im Zustand RUNNING hatte.

5.2.2 XML

Dieser Abschnitt stellt die Ergebnisse von Messungen vor, welche Schemata und Instanzen in XML-Dateien ein- und auslagern. Hierbei ist die Serialisierung der Daten von zentraler Bedeutung. Deshalb wird zunächst der Zeitaufwand betrachtet, welcher für die De-/Serialisierung, also die Transformation der Primärspeicherobjekte in XML und zurück, von Schemata und Instanzen nötig ist. Danach werden die Ergebnisse der Speicherung dieser Daten im Dateisystem aufgeschlüsselt nach Schemata und Instanzen vorgestellt. Abschließend wird auf die Messergebnisse bei Anfragen auf Kollektionen eingegangen.

5.2.2.1 Serialisierung Um die Persistierung mithilfe von XML-Dateien zu ermöglichen, müssen sowohl Schemata als auch Instanzen serialisiert werden (vergleiche Abschnitt 4.3).

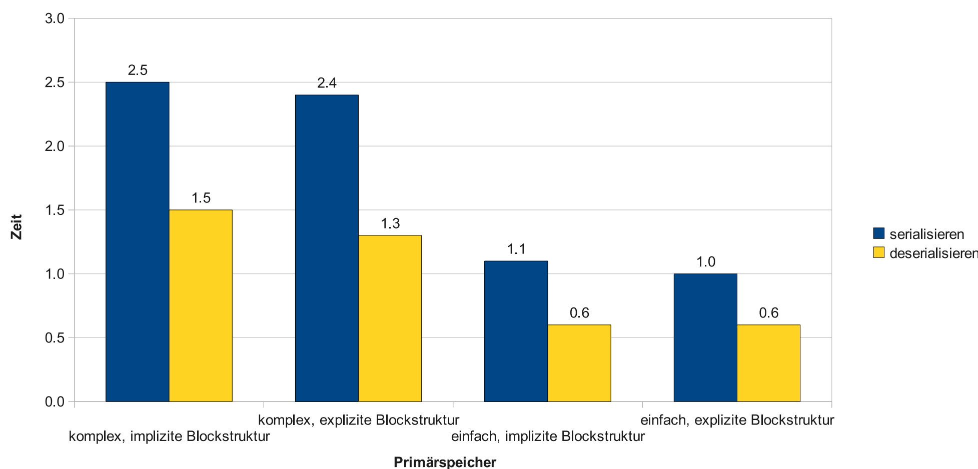


Abbildung 64: De-/Serialisieren eines Blocks

Hierzu wurde auch die Laufzeit der De-/Serialisierung von einzelnen Blöcken eines Schemas betrachtet. Die Ergebnisse sind in Abbildung 64 dargestellt. Die explizite Primärspeicherrepräsentation hat hierbei beim Serialisieren Vorteile, da die Blockrepräsentation bereits vorliegt und nicht berechnet werden muss. Auch beim Deserialisieren ist die ex-

plizite Primärspeicherrepräsentation schneller, da die implizite die (expliziten) Daten umformen muss.

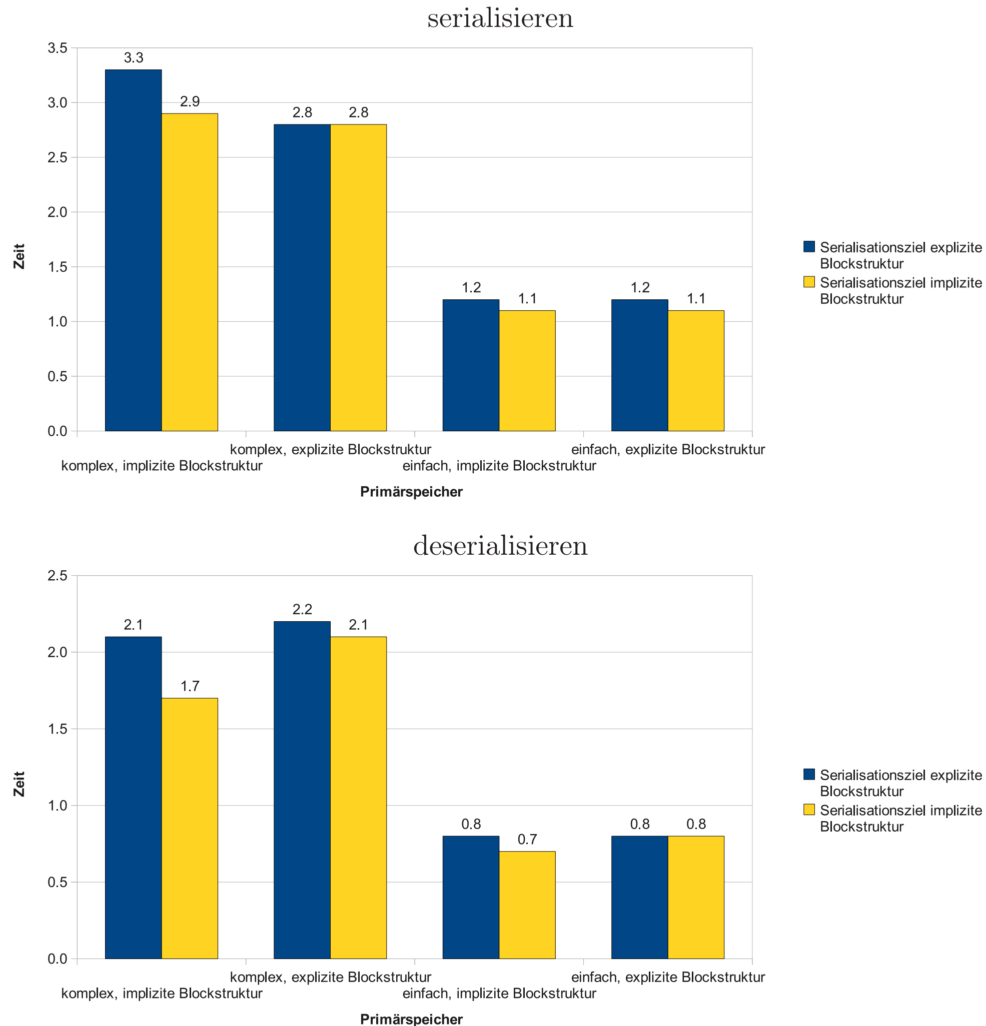


Abbildung 65: De-/Serialisieren von Schemata

Auf Basis der Messwerte von Schemata aus Abbildung 65 ist zu erkennen, dass das De-/Serialisieren von Schemata in eine XML-Repräsentation mit impliziter Blockstruktur im Allgemeinen schneller ist als die Serialisierung in eine mit expliziter Blockstruktur. Dies ist unabhängig von der Primärspeicherrepräsentation in der das Schema vorliegt, wobei der Vorteil einer impliziten XML-Repräsentation bei einer impliziten Primärspeicherrepräsentation größer ist. Dies liegt daran, dass die explizite Primärspeicherrepräsentation hierbei in eine implizite umgewandelt werden muss.

Die Messergebnisse des De-/Serialisierens von Instanzen ist in Abbildung 66 dargestellt. Hierbei fällt auf, dass es am Schnellsten ist, wenn man als XML-Repräsentation die gleiche

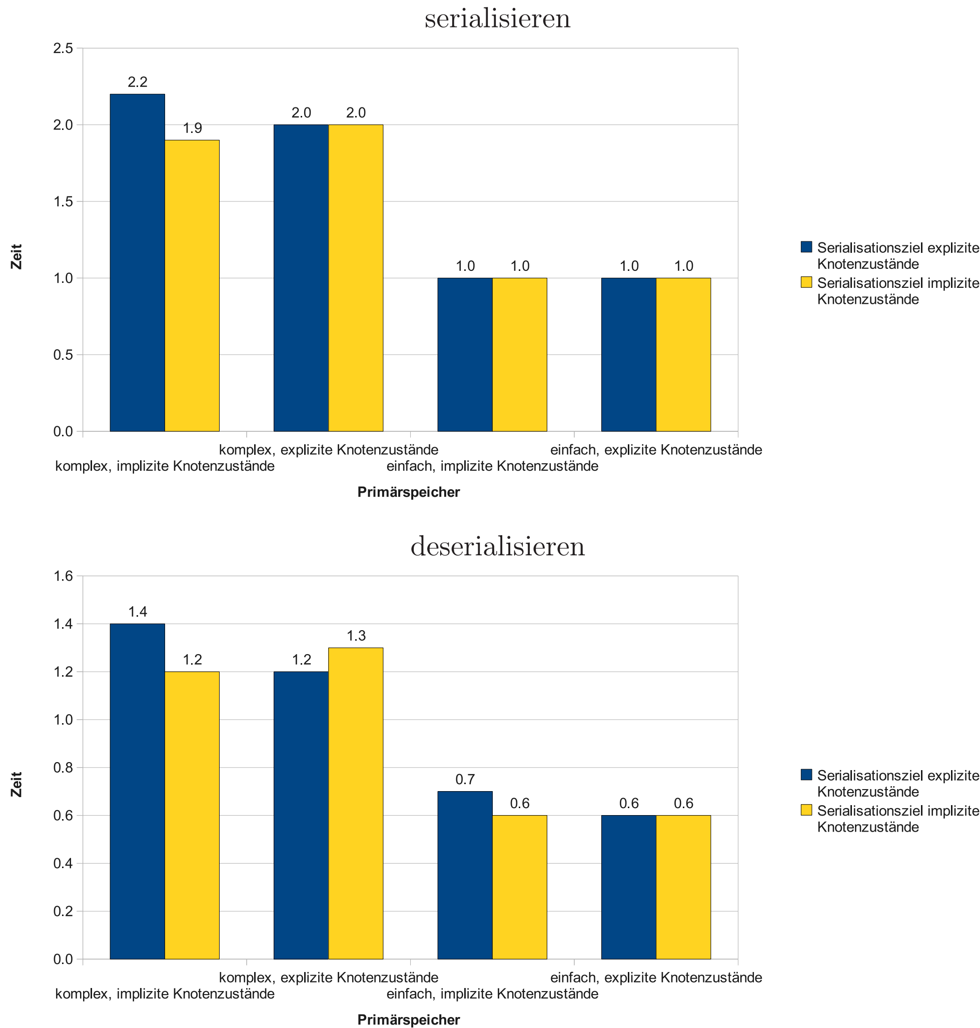


Abbildung 66: De-/Serialisieren von Instanzen

Repräsentation wählt, wie sie im Primärspeicher vorliegt. Ist eine Instanz mit impliziten Knotenzuständen vorhanden, ist es vergleichsweise langsam, diese in eine explizite Repräsentation zu serialisieren. Für eine explizite Primärspeicherrepräsentation ist hingegen die Deserialisierung aus einer impliziten Repräsentation vergleichsweise langsam. In beiden Fällen müssen die fehlenden Knotenzustände COMPLETED und NOT_ACTIVATED errechnet werden.

5.2.2.2 Schemata Diese Serialisierung ist Basis der Speicherung von Schemata in XML-Dateien. Die in Abbildung 67 dargestellten Messergebnisse umfassen sowohl die De-/Serialisierung als auch das Laden und Speichern dieser in eine Datei. Es wird erwartet, dass Schemata, die im Primärspeicher in expliziter Blockrepräsentation vorliegen

auch am schnellsten in eine XML-Datei gespeichert werden, welche explizit serialisiert wird. Die Messungen zeigen allerdings das Gegenteil. So zeigen die Werte für das Speichern eines komplexen Schemas, dass ein in impliziter Blockrepräsentation im Primärspeicher vorliegendes Schema schneller in eine XML-Datei mit expliziter Blockrepräsentation gespeichert werden kann als ein Schema, welches in expliziter Blockrepräsentation im Primärspeicher vorliegt. Ebenso gilt die Gegenrichtung, was aus den anderen, paarweise markierten Zahlen in den Diagrammen ablesbar ist. Dieses Verhalten ist nicht erklärbar, wurde aber durch mehrfache Ausführung der Messungen bestätigt.

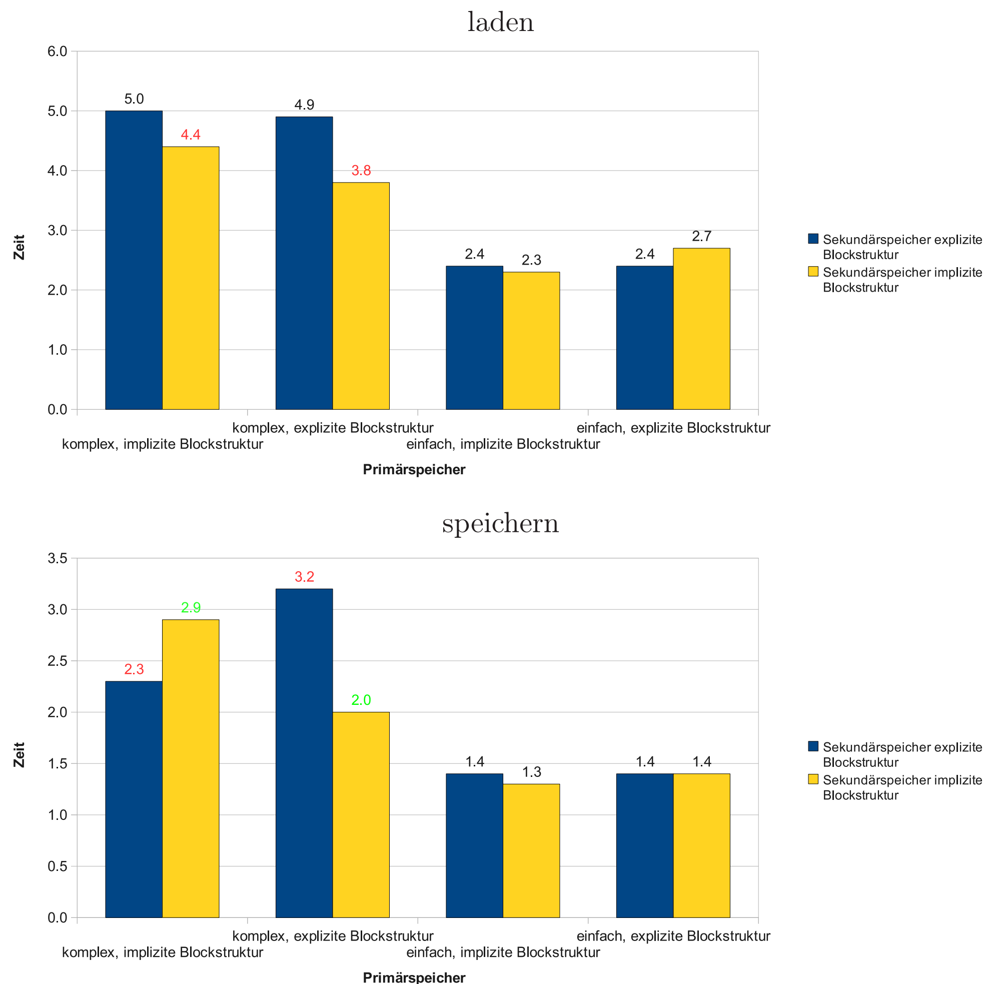


Abbildung 67: Messergebnisse für Schemata bei einer XML-Datei-Speicherung

Vergleicht man die implizite Sekundärspeicherrepräsentation mit der expliziten, so stellt man fest, dass die implizite unabhängig von der eingesetzten Primärspeicherrepräsentation Vorteile bietet. Hierbei ist zu beachten, dass Schemata häufig geladen und selten gespeichert werden. Somit ist die Ladezeit höher zu gewichten. Es fällt auf, dass für ein

einfaches, mit expliziten Blockinformationen vorliegendes Schema die explizite Sekundärspeicherrepräsentation schneller ist als die implizite. In allen anderen Fällen ist eine implizite Sekundärspeicherrepräsentation vorteilhaft.

5.2.2.3 Instanzen Für Instanzen existieren die beiden Sekundärspeicherrepräsentationen mit expliziten und impliziten Knotenzuständen. Diese müssen ebenfalls zuerst serialisiert und dann in eine Datei gespeichert werden.

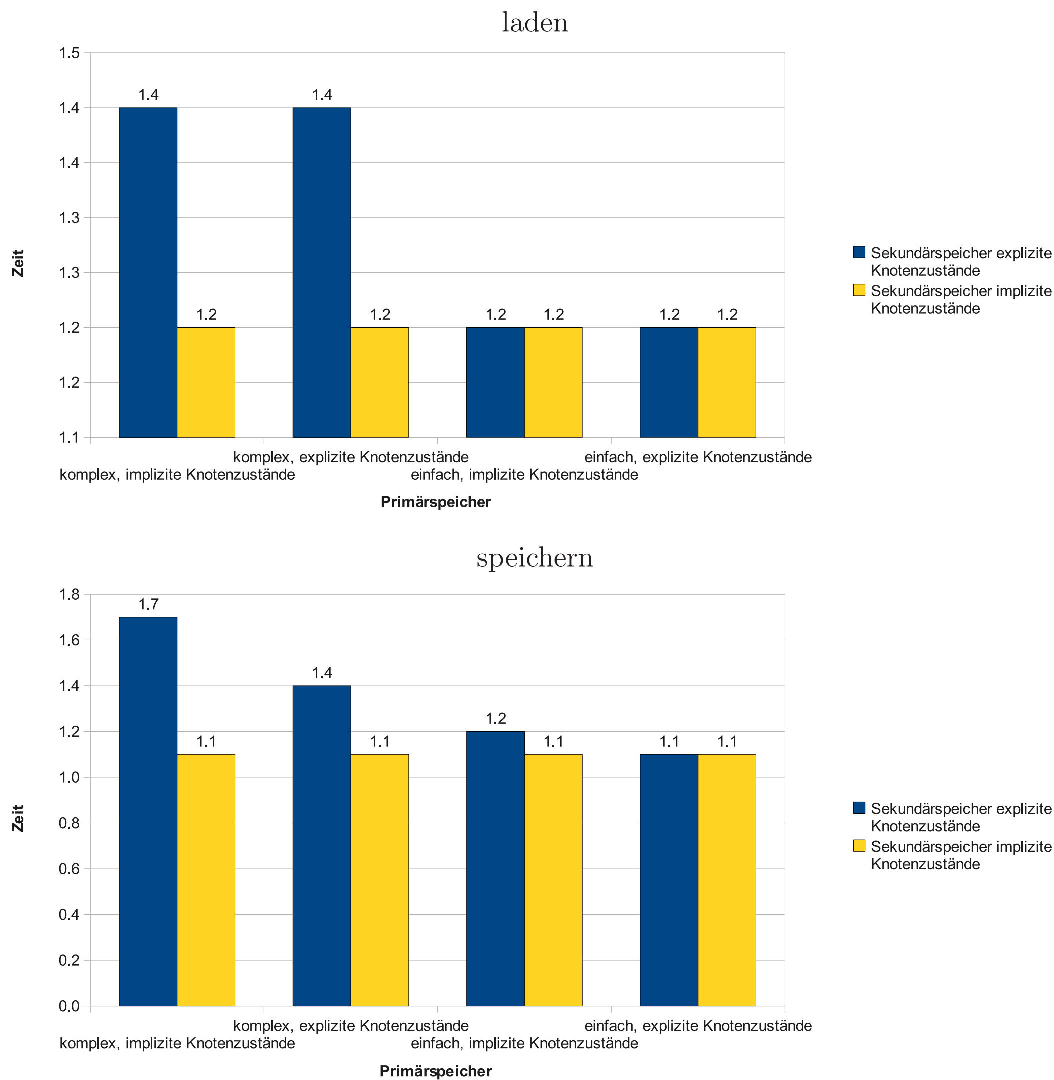


Abbildung 68: Messergebnisse für Instanzen bei einer XML-Datei-Speicherung

In Abbildung 68 sind die Messergebnisse dargestellt. Es fällt auf, dass keine merklichen Unterschiede in den Ergebnissen zwischen impliziter und expliziter Primärspeicher-

repräsentation bestehen, wenn eine implizite XML-Repräsentation gewählt wird. Dies ist erstaunlich, da beim Laden erwartet wird, dass die explizite Primärspeicherrepräsentation langsamer ist. Hierbei müssen die fehlenden Zustände COMPLETED und NOT_ACTIVATED errechnet werden, da sie nicht in der XML-Datei vorliegen. Anschaulich wird dies bei Betrachtung der expliziten XML-Repräsentation. Hier fällt auf, dass beim Speichern die implizite Primärspeicherrepräsentation langsamer ist, da nicht alle Zustände, die zur Serialisierung benötigt werden, direkt vorliegen.

Zusammenfassend ist die implizite Sekundärspeicherrepräsentation von Instanzen zu empfehlen. Sie ist in allen Fällen bezüglich des Zeitaufwands bei Ein- und Auslagerung der expliziten mindestens gleichwertig.

5.2.2.4 Anfragen auf Kollektionen Die Messergebnisse von Anfragen auf Kollektionen bei der XML-Datei-Speicherung sind in Abbildung 69 dargestellt. Für die implizite XML-Repräsentation ist erkennbar, dass die beiden Anfragen nach Knoten im Zustand RUNNING in etwa den gleichen Zeitaufwand benötigen. Dieser ist geringer als der Aufwand bei Anfragen an den Zustand COMPLETED, da dieser bei der impliziten Repräsentation nicht direkt vorhanden ist. Deshalb müssen bei einer Anfrage nach dem Zustand COMPLETED die Zustände aller Knoten geladen und anschließend im Primärspeicher ausgewertet werden. Bei Anfragen nach dem Zustand RUNNING können die XML-Daten dagegen direkt untersucht werden.

Die implizite XML-Repräsentation ist allerdings durchgängig schneller als die explizite. Dies liegt daran, dass bei der impliziten weniger Daten vorhanden und somit weniger Zeit dafür aufgewendet werden muss, diese zu untersuchen. Selbst in den Fällen, in denen bei der impliziten Repräsentation alle Knotenzustände geladen und im Primärspeicher untersucht werden müssen, ist die explizite Repräsentation langsamer. Hieraus folgt, dass der Großteil der Zeit beim Laden des Datenstroms von der Festplatte und nicht beim Deserialisieren und Berechnen der fehlenden Knotenzustände anfällt.

5.2.2.5 Zusammenfassung Bei der Betrachtung der Messungen der Ein-/Auslagerung von Schemata wurde festgestellt, dass die explizite Sekundärspeicherrepräsentation ausschließlich im Fall von einer expliziten Primärspeicherrepräsentation und einfachen Schemata vorzuziehen ist. Für die Sekundärspeicherrepräsentation von Instanzen wurde sowohl für das Ein- und Auslagern als auch für Anfragen auf Kollektionen festgestellt, dass die Repräsentation mit impliziten Knotenzuständen der mit expliziten überlegen ist.

5.2.3 PostgreSQL

Neben einer Speicherung in XML-Dateien ist der Einsatz eines RDBMS eine weitere Möglichkeit zur Persistierung. In diesem Abschnitt werden die Messergebnisse beim Einsatz von PostgreSQL vorgestellt. Hierbei wird ebenfalls zunächst auf das Ein- und Auslagern

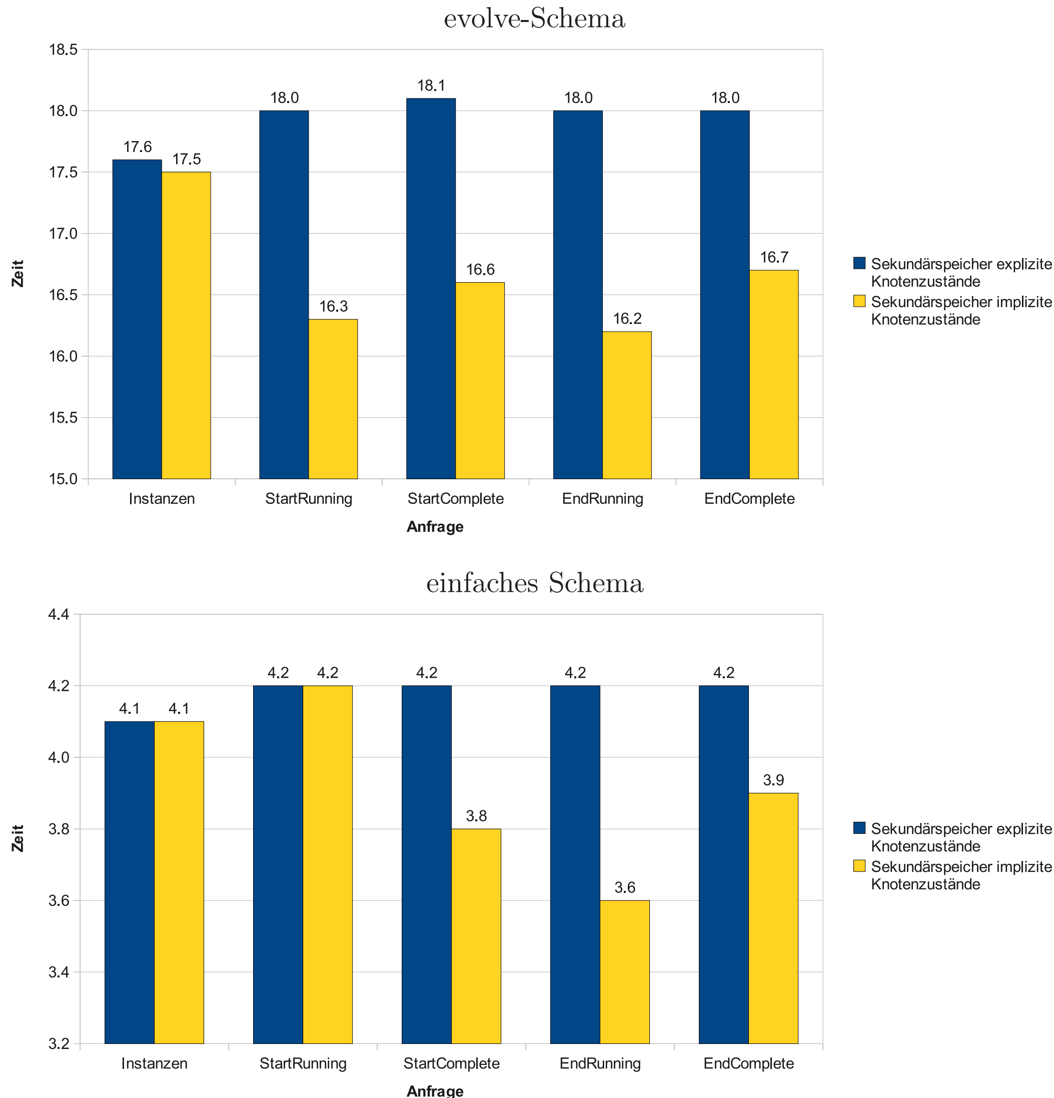


Abbildung 69: Anfragen auf Kollektionen bei XML-Datei-Speicherung

von Schemata und Instanzen und im Anschluss auf Anfragen auf Kollektionen eingegangen.

5.2.3.1 Schemata Die Messergebnisse für Schemata sind in Abbildung 70 dargestellt. Hierbei wird unterschieden zwischen den drei Sekundärspeicherrepräsentationen, welche in Abschnitt 4.3.3 vorgestellt wurden: Eine Repräsentation mit impliziten Blockinformationen, eine mit expliziten, eine mit expliziten unter Einsatz des XML-Datentyps und eine mit impliziten Blockinformationen.

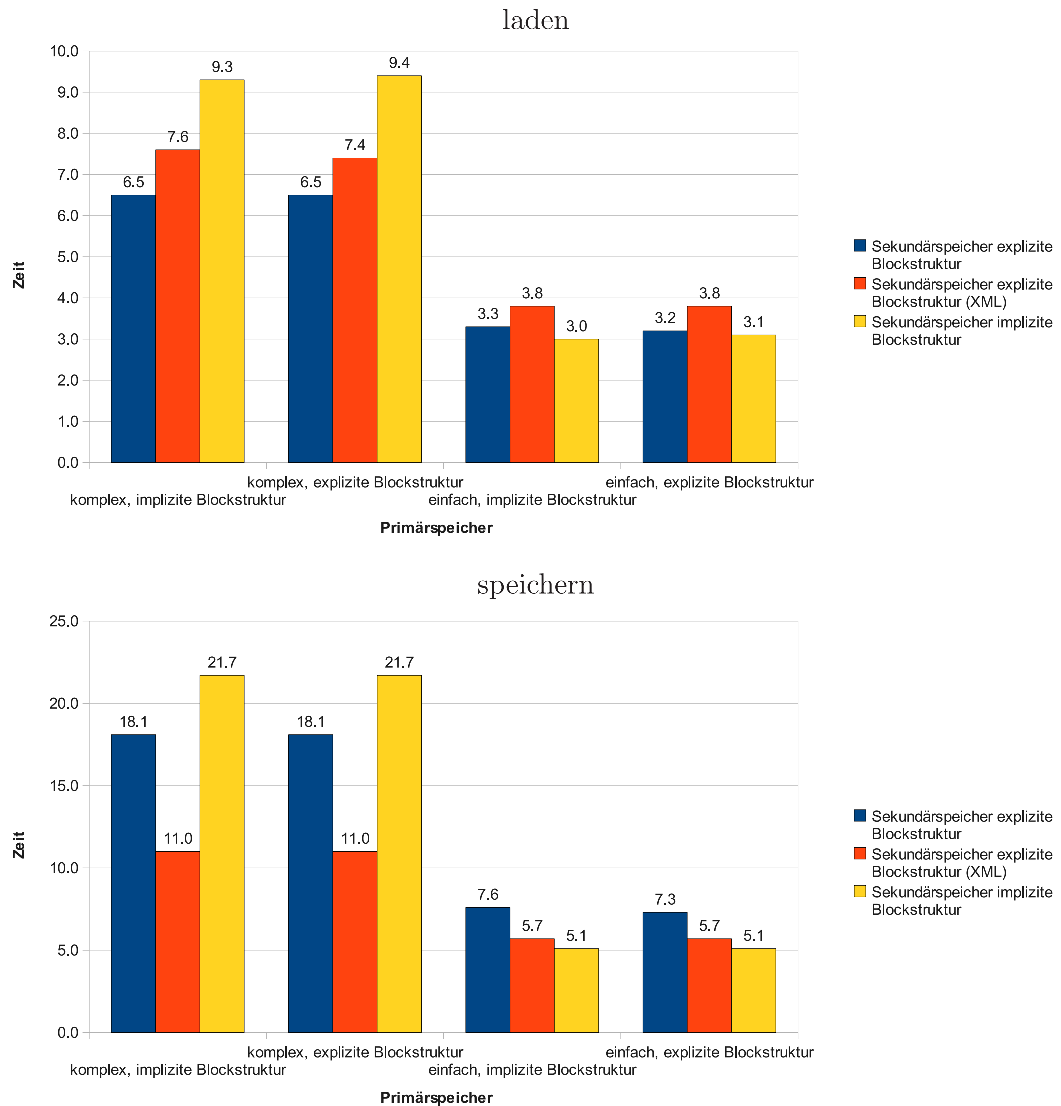


Abbildung 70: Messergebnisse von Schemata bei Einsatz von PostgreSQL

Man erkennt, dass die implizite Sekundärspeicherrepräsentation der expliziten bei dem eingesetzten einfachen Schema vorteilhaft, beim komplexen Schema hingegen wesentlich langsamer ist. Dies wurde weitergehend untersucht und herausgefunden, dass diese Grenze von der prozentualen Anzahl der Blöcke zur Anzahl der Knoten abhängig ist. Hierbei wurde festgestellt, dass die implizite Sekundärspeicherrepräsentation etwa bis zu einem Grad von 27% Blockanteil schneller ist. Dies wurde anhand von Ladezeiten von Schemata untersucht, welche in die selbe Primärspeicherrepräsentation geladen wurden, wie sie im Sekundärspeicher vorlagen. Die Knoten wurden hierbei auf die Blöcke gleich verteilt. Überprüft man dieses Ergebnis anhand der durchgeführten Messung aus Abbildung 70,

so stellt sich für das eingesetzte komplexe Schema ein Blockanteil etwa 13.6% heraus. Hierbei ist allerdings die explizite Sekundärspeicherrepräsentation bereits schneller als die implizite, was dem vorigen Ergebnis von 27% widerspricht. Der Grund hierfür liegt darin, dass im benutzten komplexen Schema die Knoten nicht gleich auf die Blöcke verteilt sind. Somit hängt der Blockanteil, ab dem die explizite Repräsentation der impliziten vorzuziehen ist, auch von der Verteilung der Knoten auf die Blöcke ab. Dies konnte im Rahmen dieser Arbeit nicht weiter untersucht werden.

Beim Vergleich der expliziten Sekundärspeicherrepräsentation mit und ohne XML-Datentyp fällt auf, dass die Daten zwar etwa zwischen 22% und 40% schneller gespeichert, allerdings 13% bis 16% langsamer geladen werden. Da ein Schema im Normalfall nur einmal gespeichert, aber oft geladen wird, ist die Verwendung des XML-Datentyps beim Ein- und Auslagern der Daten nicht zu empfehlen.

5.2.3.2 Instanzen Abbildung 71 fasst die Messergebnisse vom Ein- und Auslagern von Instanzen bei Einsatz von PostgreSQL zusammen. Hierbei wurde eine Repräsentation mit expliziten und eine mit impliziten Knotenzuständen untersucht, jeweils einmal mit und einmal ohne Einsatz des XML-Datentyps.

Auffällig hierbei ist der hohe Aufwand, der zur impliziten Speicherung von Instanzen getrieben werden muss. Der Einsatz des XML-Datentyps beschleunigt diesen Vorgang erheblich. Das Laden aus einer impliziten Sekundärspeicherrepräsentation benötigt, analog zur Betrachtung für XML-Dateien, selbst für explizite Primärspeicherrepräsentationen keinen zusätzlichen Zeitaufwand. Auch hier ist ersichtlich, dass eine implizite Sekundärspeicherrepräsentation aufgrund der geringen zu ladenden/zu speichernden Datenmenge der expliziten vorzuziehen ist.

5.2.3.3 Anfragen auf Kollektionen Abbildung 72 fasst die Messergebnisse von Anfragen auf Kollektionen zusammen. Auffällig hierbei ist, dass die implizite Sekundärspeicherrepräsentation in allen Fällen schneller ist. Dieser Vorteil kann durch Einsatz des XML-Datentyps weiter ausgebaut werden. Dadurch kann bei Anfragen nach dem Knotenzustand RUNNING der gespeicherte XML-Datenstrom direkt vom RDBMS ausgewertet werden (vergleiche Abschnitt 4.4.2). Wird der XML-Datentyp nicht eingesetzt oder nach dem Zustand COMPLETED gesucht, müssen alle Instanzen eingelagert und im Primärspeicher untersucht werden.

Wird bei der expliziten Repräsentation der XML-Datentyp eingesetzt, so sind Anfragen an Knoten im Zustand COMPLETED in der impliziten Repräsentation mit XML-Datentyp etwa um das 1,7-fache langsamer. Bei Anfragen nach dem Knotenzustand RUNNING hingegen besitzt die explizite Repräsentation etwa die 2,2-fache Laufzeit wie die implizite. Somit ist der Einsatz einer impliziten Sekundärspeicherrepräsentation unter Verwendung des XML-Datentyps in Bezug auf Anfragen auf Kollektionen zu empfehlen.

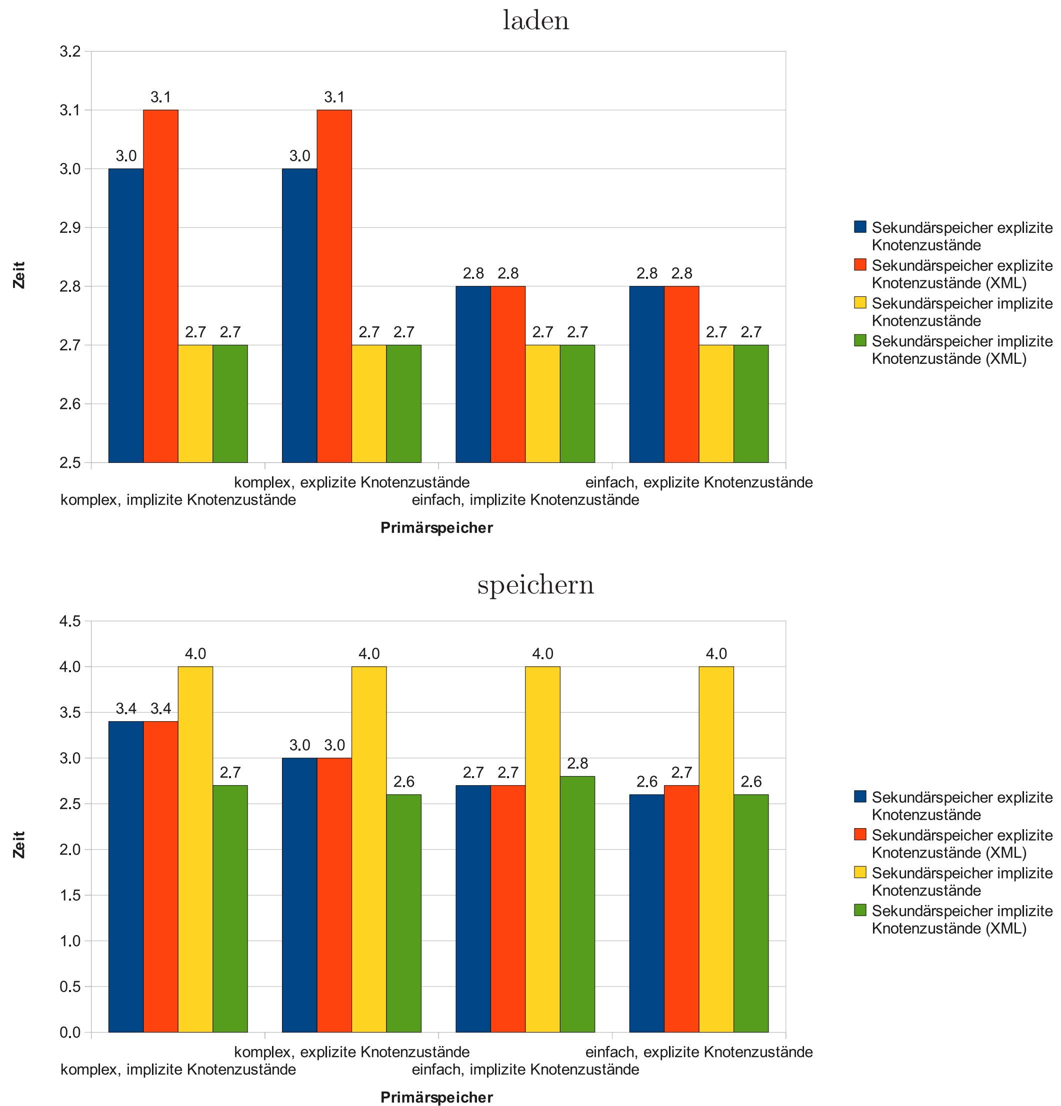


Abbildung 71: Messergebnisse von Instanzen bei Einsatz von PostgreSQL

5.2.3.4 Zusammenfassung Bei Einsatz von PostgreSQL ist es sinnvoll, für einfache Schemata die implizite, für komplexe Schemata die explizite Sekundärspeicherrepräsentation zu benutzen. Der Einsatz des XML-Datentyps ist nicht zu empfehlen. Bei Instanzen hat sich gezeigt, dass es am schnellsten ist, diese im Sekundärspeicher mit impliziten Knotenzuständen unter Einsatz des XML-Datentyps zu repräsentieren. Dies ist sowohl für die Ein-/Auslagerung als auch für Anfragen auf Kollektionen vorteilhaft.

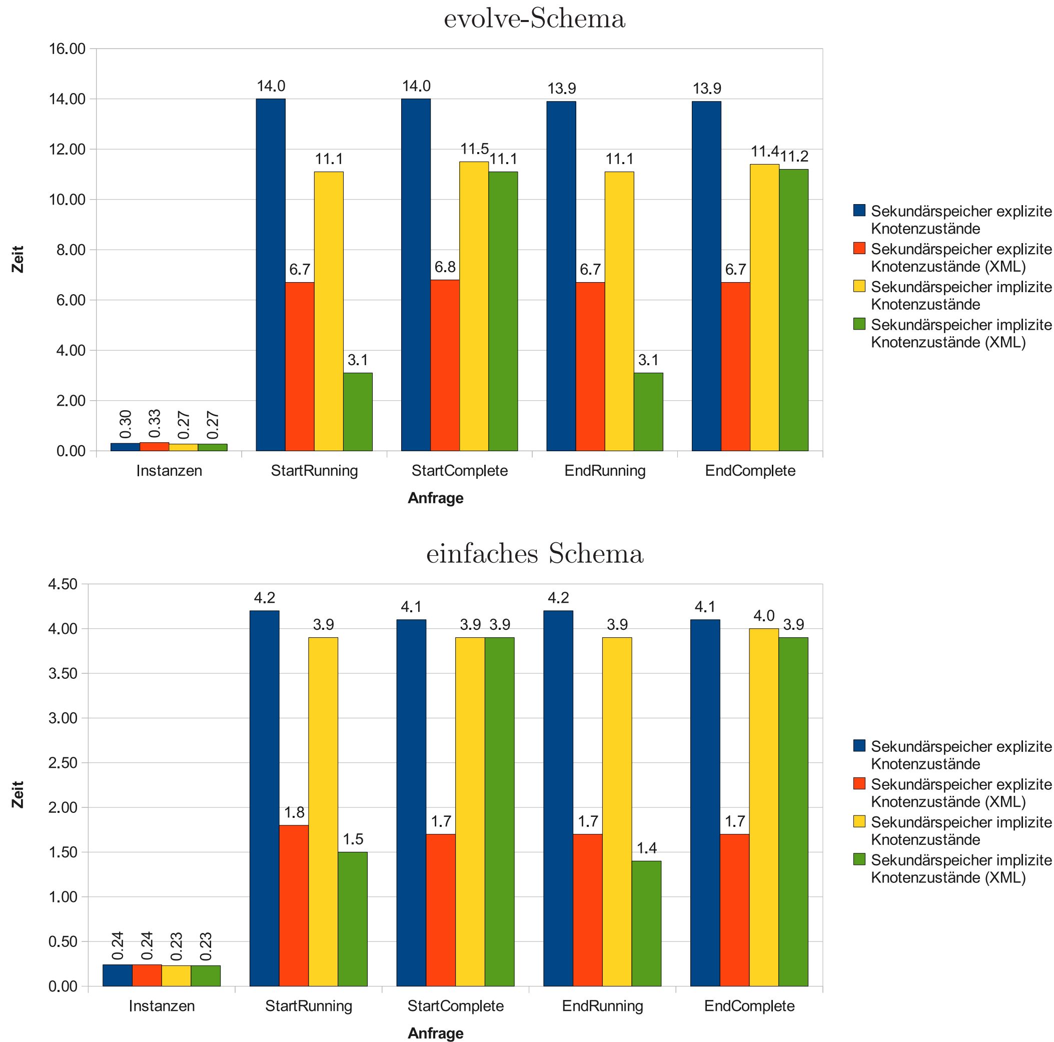


Abbildung 72: Anfragen auf Kollektionen bei Einsatz von PostgreSQL

5.2.4 Oracle

Neben PostgreSQL wurde auch die Speicherung in einem RDBMS von Oracle untersucht. Die hierbei festgestellten Messergebnisse werden im Folgenden vorgestellt. Es wird wiederum zuerst auf das Ein- und Auslagern von Schemata, dann von Instanzen und schließlich auf Anfragen auf Kollektionen eingegangen.

5.2.4.1 Schemata Die Messergebnisse des Ein-/Auslagerns von Schemata in verschiedenen Repräsentationen sind in Abbildung 73 dargestellt.

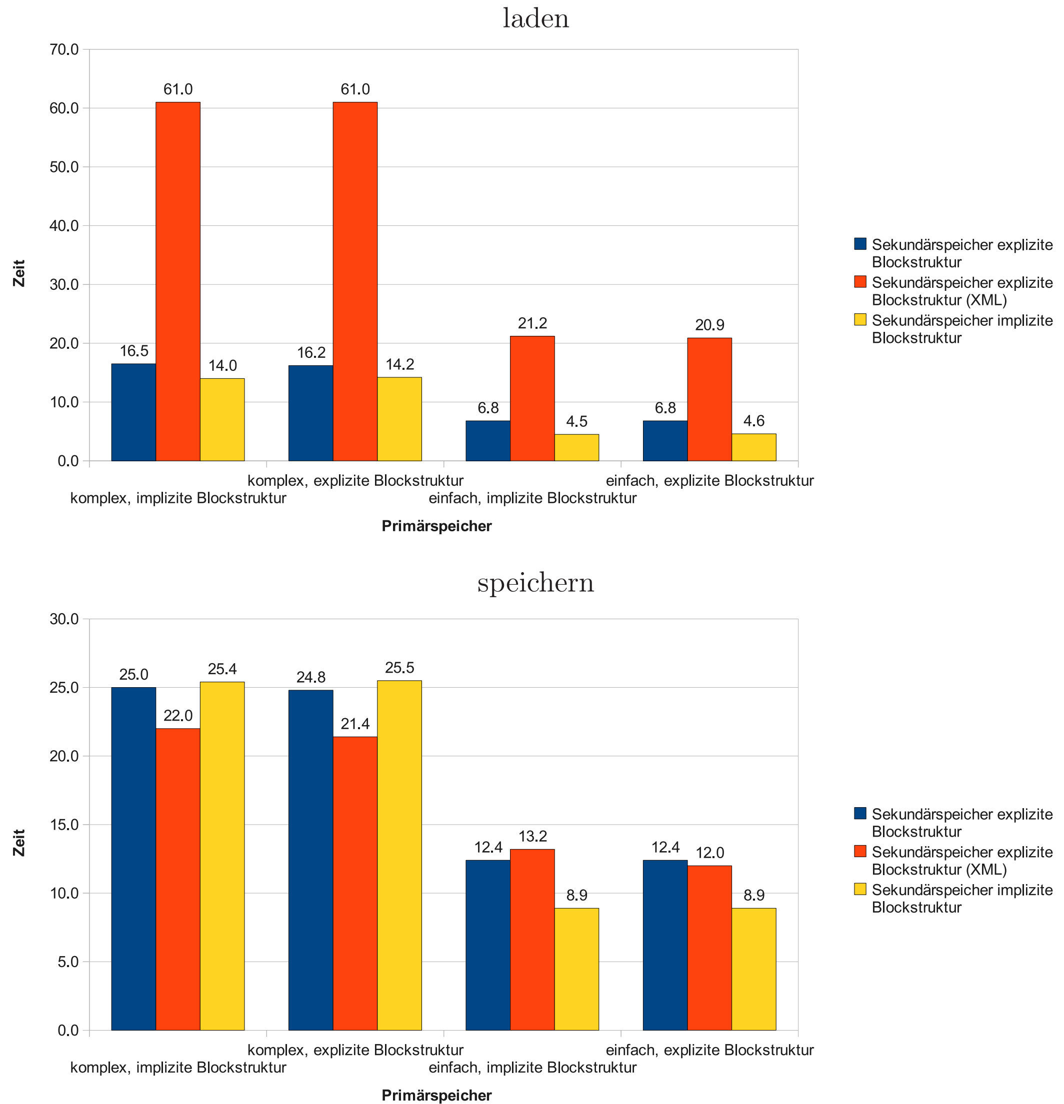


Abbildung 73: Messergebnisse von Schemata bei Einsatz von Oracle

Die implizite Sekundärspeicherrepräsentation ist bei den Ladezeiten jeweils am schnellsten und deren Einsatz somit zu empfehlen. Das Speichern ist im komplexen Fall zwar langsamer als bei einer expliziten Realisierung, da ein Schema aber einmal gespeichert und oft geladen wird, fällt diese Beobachtung nicht ins Gewicht. Auffallend ist außerdem, dass das Laden eines explizit im Sekundärspeicher vorliegenden Schemas bei Einsatz des XML-Datentyps sehr langsam ist im Vergleich zur Variante ohne Einsatz des XML-Datentyps. Hierbei ist beim komplexen Schema ein Faktor von bis zu 3.77 und beim einfachen Schema einer bis zu 3.12 zu erkennen.

5.2.4.2 Instanzen Aus den in Abbildung 74 dargestellten Ergebnissen ist abzulesen, dass es beim Einsatz von Oracle vorteilhaft ist, einfache Instanzen mithilfe einer expliziten Sekundärspeicherrepräsentation zu speichern. Dies liegt daran, dass eine Instanz im Gegensatz zu Schemata auch mehrfach gespeichert wird. Somit gleicht sich die leicht schlechtere Ladezeit gegenüber der impliziten Repräsentation durch die wesentlich bessere Laufzeit zum Speichern aus. Für komplexe Instanzen ist allerdings eine implizite Repräsentation mit Einsatz des XML-Datentyps von Vorteil.

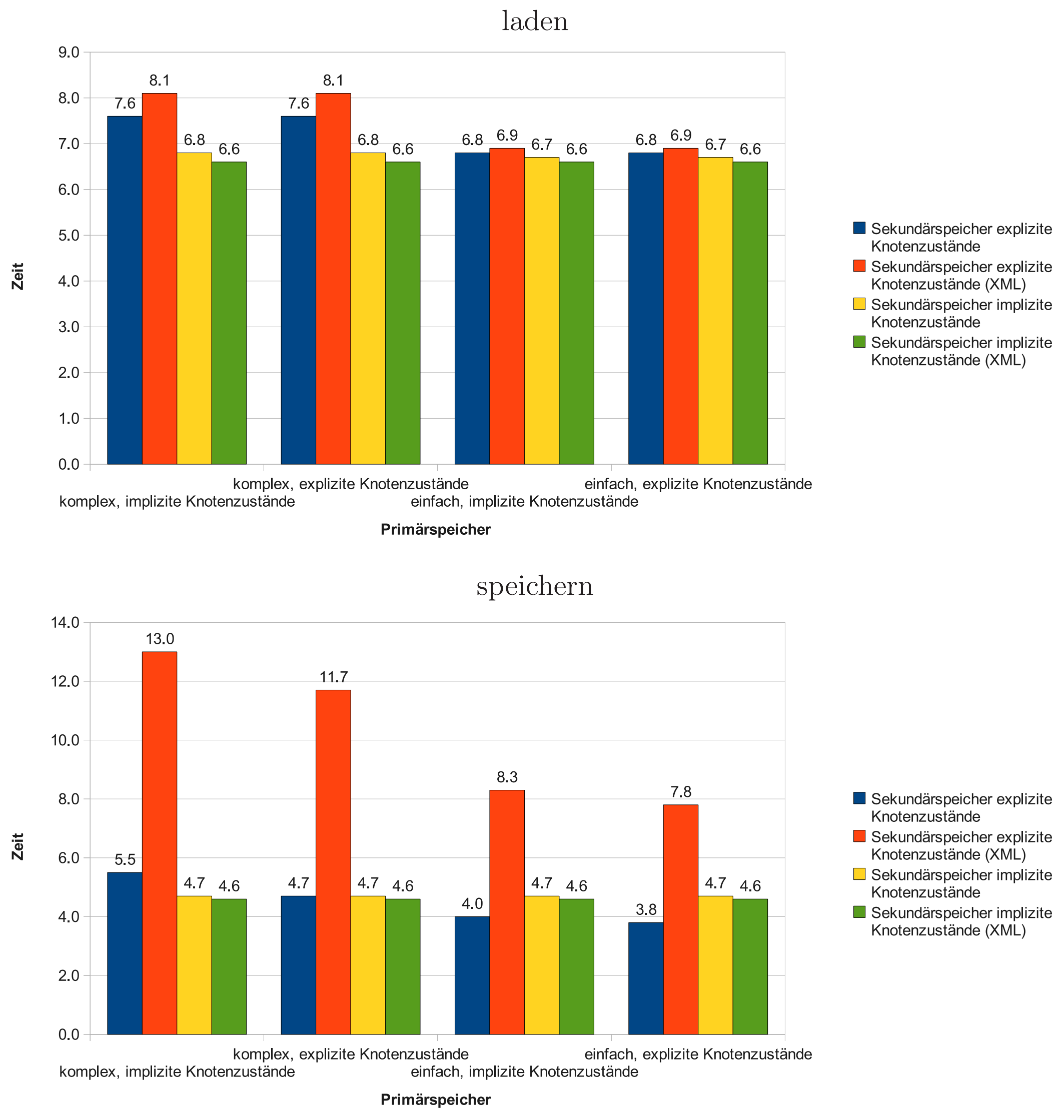


Abbildung 74: Messergebnisse von Instanzen bei Einsatz von Oracle

5.2.4.3 Anfragen auf Kollektionen Bei der Untersuchung der Messergebnisse von Anfragen auf Kollektionen aus Abbildung 75 fallen als erstes die sehr schlechten Zahlen beim Einsatz des XML-Datentyps auf. Diese liegen bis zu 11,6 mal höher als beim Einsatz des CLOB-Datentyps. Die einzigen Ausnahmen bilden hierbei die beiden Anfragen nach dem Zustand COMPLETED bei der impliziten Sekundärspeicherrepräsentation, diese sind schneller als bei der Variante ohne Einsatz des XML-Datentyps. Dies liegt

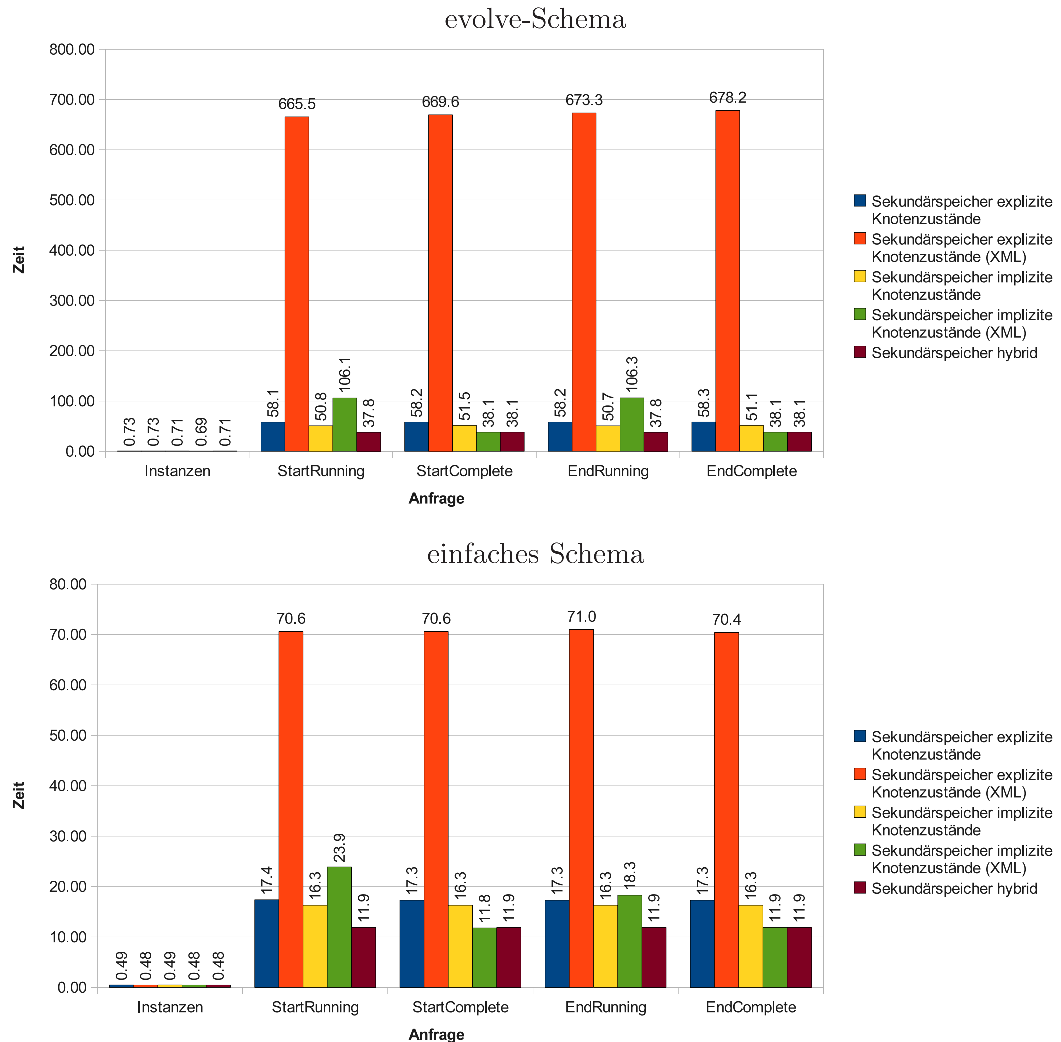


Abbildung 75: Anfragen auf Kollektionen bei Einsatz von Oracle

daran, dass bei diesen Anfragen die kompletten serialisierten Daten von der Implementierung von `InstanceStorage` abgerufen, im Primärspeicher deserialisiert und untersucht werden. Die im RDBMS vorliegenden XML-Daten werden hierbei nicht, wie bei den Anfragen nach dem Zustand RUNNING, direkt vom RDBMS untersucht. Hieraus kann man ableiten, dass Oracle die Unterstützung zum Untersuchen von XML-Daten bei Abfragen

zwar anbietet, diese allerdings nur in unbefriedigender Laufzeit beantworten kann. Wird allerdings die gesamte XML-Serialisierung abgerufen (wie zum Beispiel bei den Anfragen zum Knotenzustand COMPLETED und impliziter Realisierung), zeigt sich, dass diese bei Einsatz des XML-Datentyps schneller geliefert werden, als wenn der CLOB-Datentyp eingesetzt wird. Es liegt somit nahe, für Oracle eine hybride Lösung aus den bisher vorgestellten einzusetzen. Diese basiert auf der impliziten Repräsentation mit XML-Datentyp, führt allerdings keine direkten Datenbankanfragen auf den XML-Daten durch, sondern lädt jeweils die ganzen Daten, deserialisiert im Primärspeicher und wertet diese dann aus. Die Ergebnisse einer solchen Realisierung sind ebenfalls in den Diagrammen in Abbildung 75 dargestellt. Diese Ergebnisse sind in Bezug auf die Anfrage nach Instanzen eines Schemas und Instanzen mit Knoten im Zustand RUNNING vergleichbar mit den Werten der impliziten Repräsentation unter Einsatz des XML-Datentyps. Bei Anfragen nach Knoten im Zustand COMPLETED ist die hybride Repräsentation allerdings wesentlich schneller und deren Einsatz ist somit zu empfehlen.

5.2.4.4 Zusammenfassung Bei Einsatz von Oracle ist die Repräsentation mit impliziten Blockinformation für Schemata vorteilhaft. In Bezug auf Instanzen wurde festgestellt, dass die explizite Repräsentation zu empfehlen ist, wenn die Instanz wenige Knoten hat. Hat sie viele Knoten, ist die implizite Repräsentation unter Einsatz des XML-Datentyps vorteilhaft. Bei der Betrachtung der Messergebnisse zu Anfragen auf Kollektionen fällt allerdings auf, dass Abfragen der XML-Daten von Oracle nur mit einer hohen Laufzeit beantwortet werden können. Aufgrund dessen wurde bei den Anfragen auf Kollektionen ein Hybridverfahren empfohlen, welches auf der impliziten Repräsentation mit Einsatz des XML-Datentyps beruht, aber keine Abfragen auf diesen Daten durchführt. Somit ist die hybride Variante zum Einsatz für Instanzen empfehlenswert.

5.2.5 DB2

Als letztes in dieser Arbeit untersuchtes RDBMS werden in diesem Abschnitt die Ergebnisse von DB2 vorgestellt. Hierbei wird wiederum zuerst auf Schemata, dann auf Instanzen und schließlich auf Anfragen auf Kollektionen eingegangen.

5.2.5.1 Schemata Abbildung 76 stellt die Messergebnisse von Schemata bei Einsatz von DB2 dar. Hierbei sind die markierten Paare verwunderlich, da diese in der falschen Relation zueinander stehen. So wird davon ausgegangen, dass ein einfaches, im Primärspeicher implizit vorliegendes Schema langsamer in eine explizite Sekundärspeicherrepräsentation mit Einsatz des XML-Datentyps gespeichert werden kann, als das selbe Schema, welches allerdings im Primärspeicher bereits explizit vorliegt. Diese Abweichungen können zum jetzigen Zeitpunkt nicht erklärt werden.

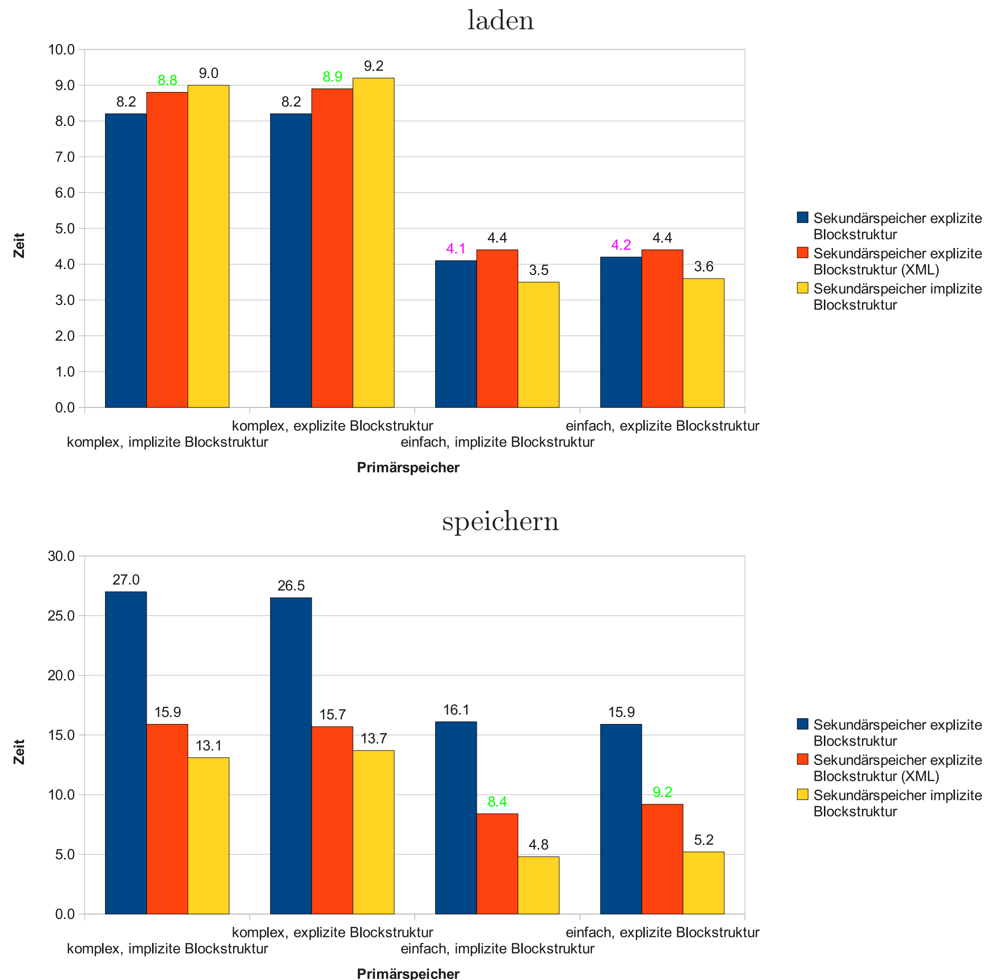


Abbildung 76: Messergebnisse von Schemata bei Einsatz von DB2

Vergleicht man die Werte, ist ersichtlich, dass sich für das einfache Schema eine implizite Sekundärspeicherrepräsentation und für das komplexe eine explizite eignet. Dies wurde bereits bei PostgreSQL beobachtet und wurde auch hier tiefergehend untersucht: Es wurde ebenfalls ein Test durchgeführt, der die Ladezeiten von Schemata mit variierendem Blockanteil misst. Hier waren die Knoten ebenfalls gleich auf die Blöcke verteilt. Bei einem Blockanteil von 25% war die implizite Sekundärspeicherrepräsentation ab etwa 4 bis 5 Knoten gegenüber der expliziten vorteilhaft, bei 22.5% Blockanteil ab etwa 15 Knoten und bei einem Anteil von 20% noch ab etwa 50 Knoten. Ist der Blockanteil geringer, ist stets die explizite Sekundärspeicherrepräsentation schneller als die implizite, bei einem Blockanteil über 25% ist die implizite schneller als die explizite. Dies bestätigt die Ergebnisse aus den Diagrammen aus Abbildung 76. Das komplexe Schema liegt mit 13.6% im Bereich, in dem die explizite Sekundärspeicherrepräsentation vorteilhaft ist,

das einfache Schema mit einem Blockanteil von 25% und 4 Knoten ist im Bereich, in dem die implizite Repräsentation schneller ist.

Bei der Betrachtung der Daten aus Abbildung 76 stellt man ebenso fest, dass ein Einsatz des XML-Datentyps nicht sinnvoll ist, da dieser beim Laden des Schemas langsamer ist.

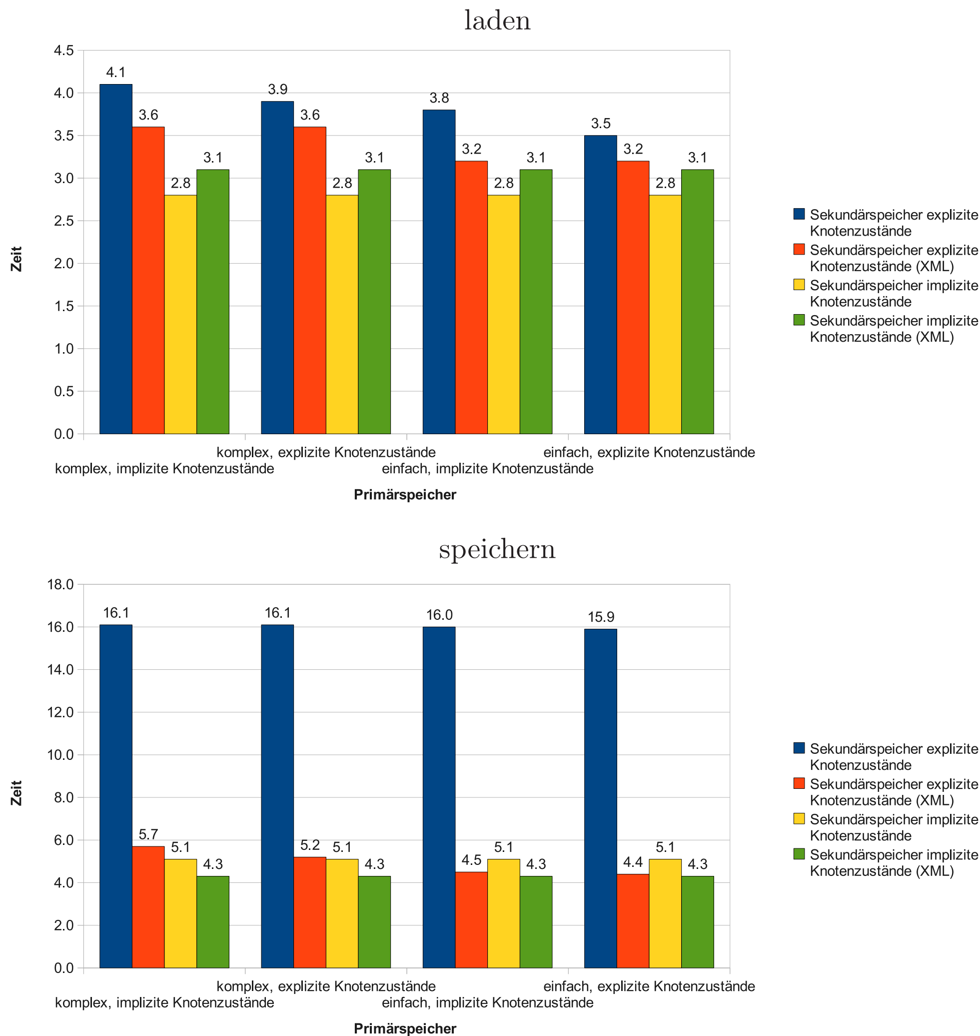


Abbildung 77: Messergebnisse von Instanzen bei Einsatz von DB2

5.2.5.2 Instanzen Die implizite Sekundärspeicherrepräsentation unter Einsatz des XML-Datentyps ist der expliziten überlegen. Dies kann an den Messergebnissen aus Abbildung 77 direkt abgelesen werden. Betrachtet man die Werte der impliziten Repräsentation mit und ohne XML-Datentyp, erkennt man, dass die implizite Repräsentation ohne XML-Datentyp beim Speichern ein Vorteil von 15.6% hat, während das Laden 9.7% lang-

samer ist als die Repräsentation mit XML-Datentyp. Da eine Instanz allerdings häufig gespeichert wird, ist die Repräsentation mit XML-Datentyp vorzuziehen.

5.2.5.3 Anfragen auf Kollektionen Abbildung 78 zeigt die Messergebnisse von Anfragen auf Kollektionen bei Einsatz von DB2. Es ist abzulesen, dass die implizite Repräsentation unter Einsatz des XML-Datentyps in den meisten Fällen am schnellsten ist. Eine

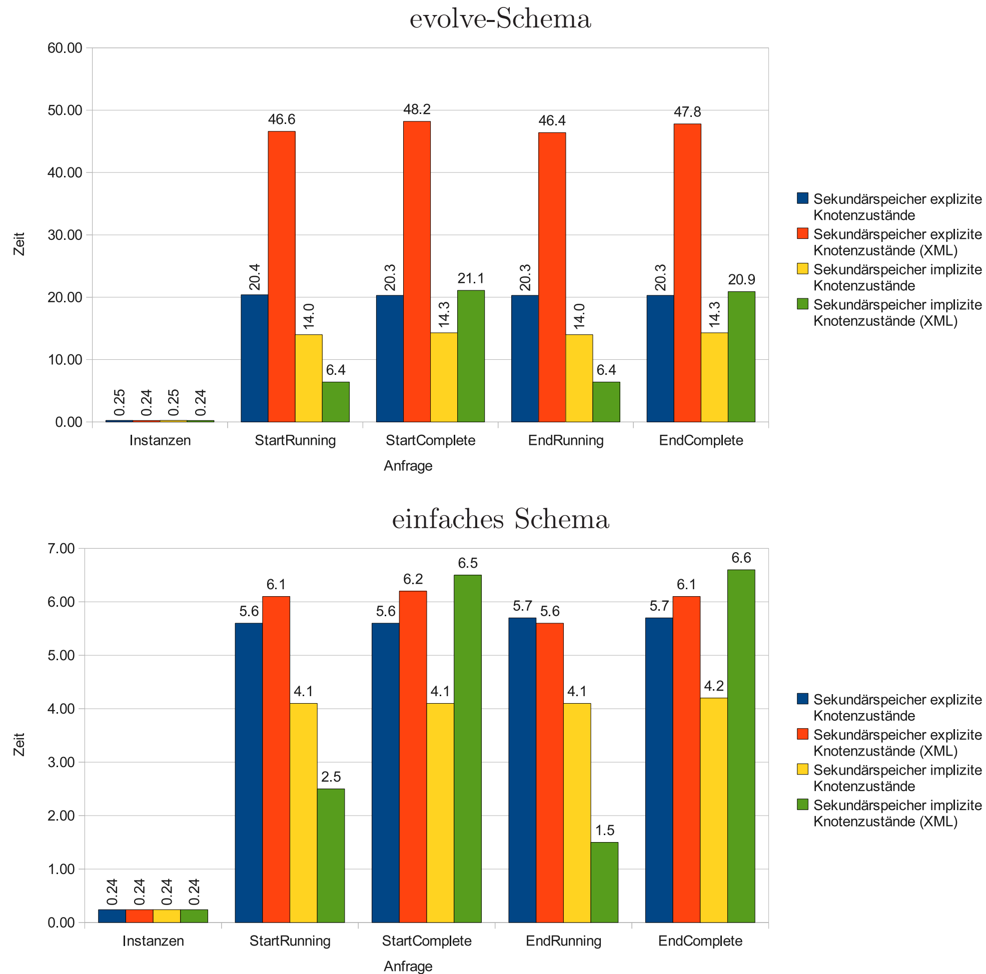


Abbildung 78: Anfragen auf Kollektionen bei Einsatz von DB2

Ausnahme bilden die Abfragen nach COMPLETED Knoten, bei denen durch eine explizite Repräsentation eine bis zu 13.8% schnellere Antwortzeit erreicht werden kann. Auf der anderen Seite benötigt die explizite Repräsentation mehr als drei mal so lange wie die implizite, eine Anfrage nach einem RUNNING Knoten zu beantworten. Ein nicht ganz so krasses, aber dennoch deutliches Bild entsteht bei der Gegenüberstellung der impliziten Repräsentation mit und ohne Einsatz des XML-Datentyps. Hierbei wird in etwa die

1,5-fache Zeit benötigt, um einen COMPLETED Knoten aus einer XML-Spalte zu laden gegenüber einer CLOB-Spalte. Dafür benötigt die Repräsentation ohne XML-Datentyp mehr als doppelt so lange, eine Abfrage auf RUNNING Knoten zu beantworten. Somit ist die implizite Repräsentation mit Einsatz des XML-Datentyps zu empfehlen.

5.2.5.4 Zusammenfassung Es ist wie bereits bei PostgreSQL vorteilhaft, Schemata implizit zu repräsentieren, sofern es einfache Schemata sind. Handelt es sich um komplexe, so eignet sich eine explizite Repräsentation besser. Für Instanzen ist es sinnvoll, diese mit impliziten Knotenzuständen unter Einsatz des XML-Datentyps zu repräsentieren. Dies ist sowohl für die Ein- und Auslagerung als auch für Anfragen auf Kollektionen von Vorteil.

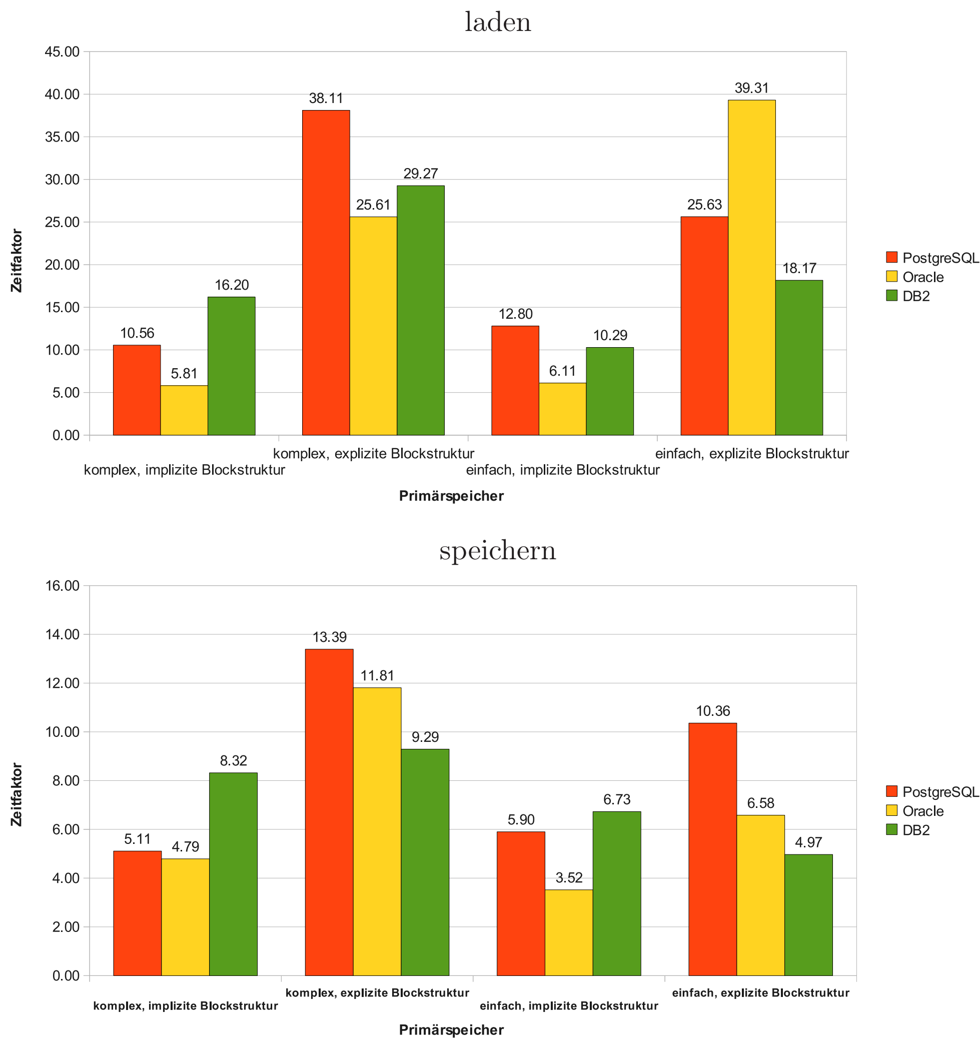


Abbildung 79: Geschwindigkeitsfaktoren von Schemata bei Einsatz von JPA

5.2.6 JPA

Nach der Vorstellung der Messergebnisse bei direkter Speicherung von Schemata und Instanzen in RDBMS geht dieser Abschnitt auf die Speicherung mithilfe von JPA ein. Hierbei werden die Daten ebenfalls in einem der vorgestellten RDBMS gespeichert. Eine Persistierung mit JPA ist im Allgemeinen wesentlich langsamer als eine direkte Persistierung. Deshalb wurde in den Abbildungen 79 und 80 eine andere Darstellungsform als bisher gewählt. Die angegebenen Werte sind die Faktoren, um die die Persistierung mit JPA langsamer ist als eine vergleichbare direkte. Es ist erkennbar, dass das Laden von Schemata das 5,81-fache bis zum 39,31-fachen der Zeit benötigt, welche bei einer direkten Speicherung anfällt. Das Speichern ist mit dem 3,52-fachen bis 13,39-fachen ebenfalls auffällig langsamer.

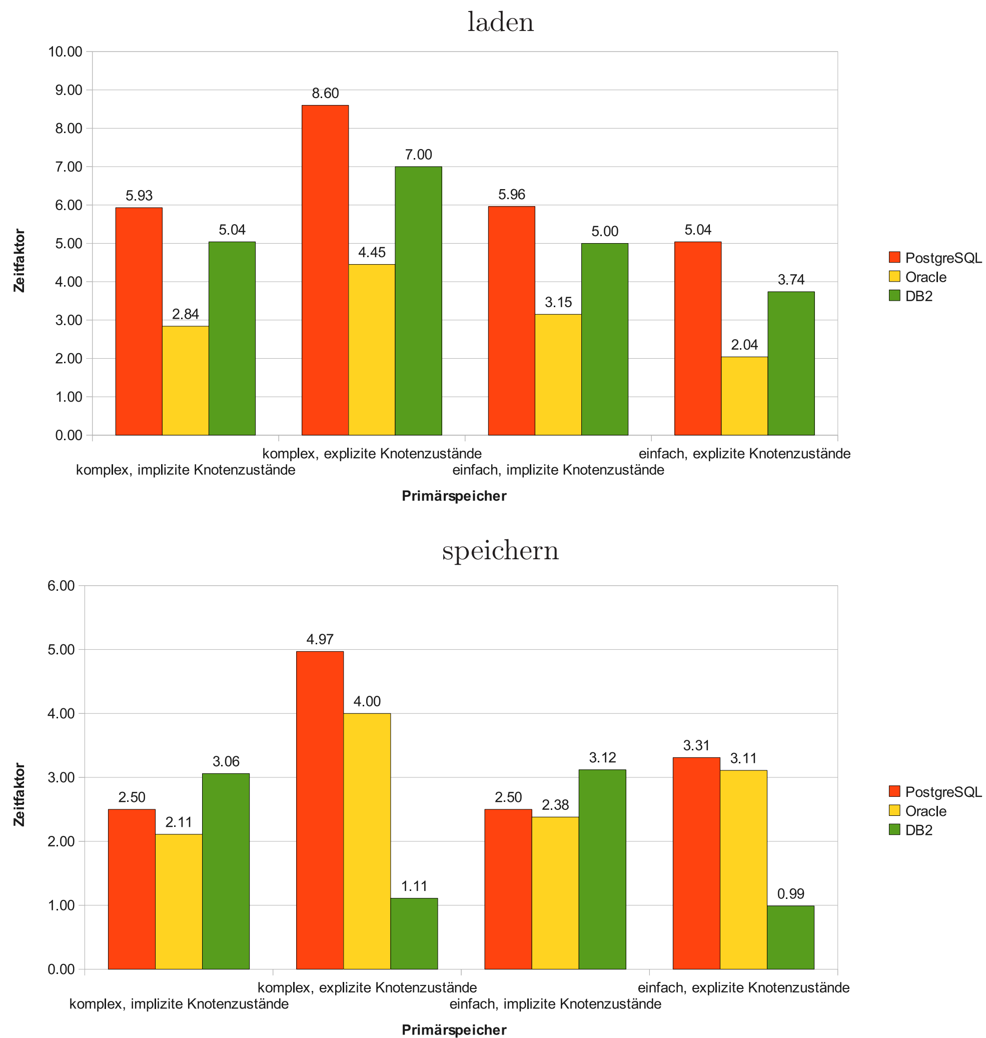


Abbildung 80: Geschwindigkeitsfaktoren von Instanzen bei Einsatz von JPA

Bei Instanzen zeigt sich ein nicht ganz so krasses Bild wie bei Schemata, dennoch werden beim Laden bis zum 8,6-fachen und beim Speichern bis zum 4,97-fachen der Zeit von einer direkten Speicherung benötigt.

Aufgrund dieser Werte wird JPA fortan nicht weiter betrachtet.

5.2.7 Vergleich

Es wurden sowohl für eine XML-Datei-Speicherung als auch einer Persistierung mit den verschiedenen RDBMS Messwerte vorgestellt. In diesem Abschnitt werden die jeweils empfohlenen Repräsentationen in Relation zueinander gesetzt und verglichen. Hierbei werden wie bisher zuerst Schemata, dann Instanzen und schließlich Anfragen auf Kollektionen betrachtet.

5.2.7.1 Schemata Abbildung 81 zeigt den Vergleich der Werte der jeweils empfohlenen Repräsentationen beim Ein-/Auslagern von Schemata. Hierbei zeigt sich, dass die XML-Datei-Speicherung am schnellsten ist.

Um einen direkten Vergleich ziehen zu können, wird angenommen, dass auf eine Speicherung eines Schemas 100 Ladevorgänge kommen. Außerdem gebe es gleich viele komplexe wie einfache Schemata. Dann lässt sich für jede der Varianten ein Performanzkoeffizient berechnen, wobei l die Ladezeiten und s die Zeiten zum Speichern sind:

$$c_{Schema} = \left(\frac{100 \cdot l_{einfach} + s_{einfach}}{101} + \frac{100 \cdot l_{komplex} + s_{komplex}}{101}\right) \cdot \frac{1}{2}$$

Diese Koeffizienten repräsentieren den durchschnittlichen Zeitaufwand zum Laden oder Speichern und sind in Abbildung 82 für die einzelnen Speichervarianten in Abhängigkeit der Primärspeicherrepräsentation angegeben.

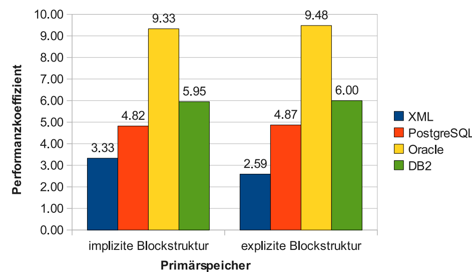
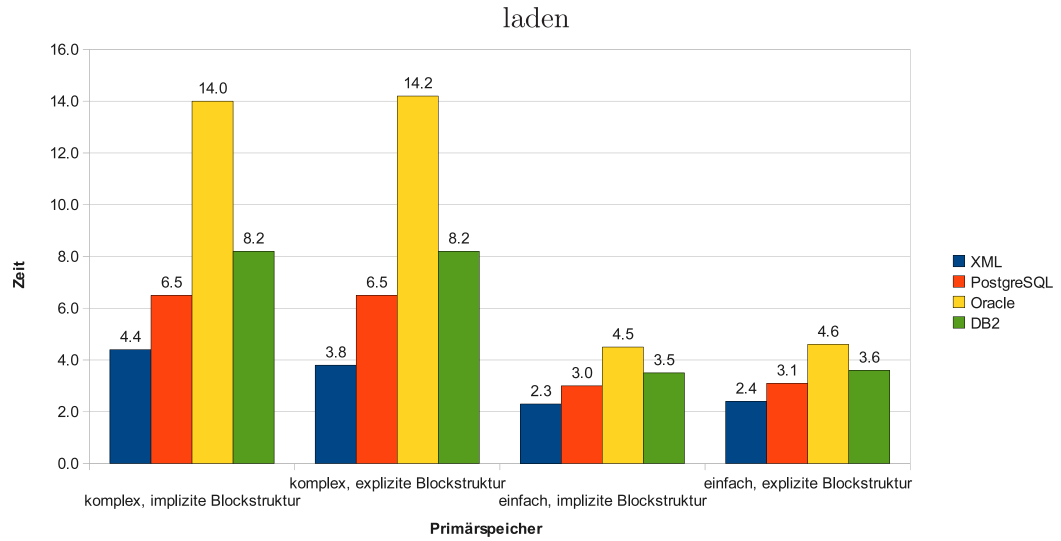
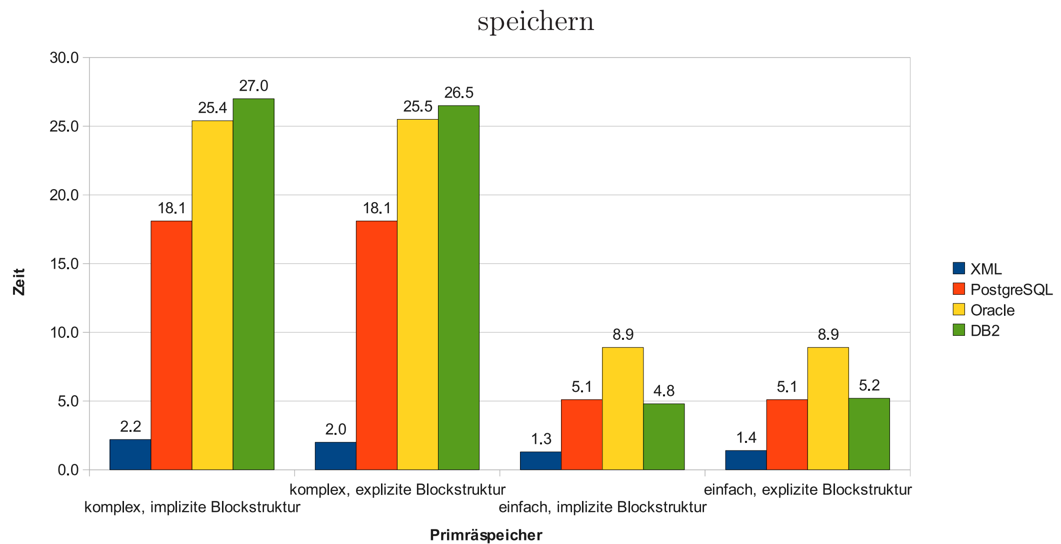


Abbildung 82: Performanzkoeffizienten c_{Schema} der einzelnen Varianten

Es ist hierbei ersichtlich, dass die XML-Datei-Speicherung wesentlich schneller ist als eine Speicherung in einem RDBMS. Hierbei muss angemerkt werden, dass die XML-Datei-Speicherung nicht so viele Möglichkeiten bietet wie eine Persistierung in einem





		Primärspeicher			
		implizit		explizit	
Sekundärspeicher	Schema	laden	speichern	laden	speichern
XML	komplex	4.4[2]	2.2[2]	3.8[2]	2[2]
	einfach	2.3[2]	1.3[2]	2.4[1]	1.4[1]
PostgreSQL	komplex	6.5[1]	18.1[1]	6.5[1]	18.1[1]
	einfach	3[2]	5.1[2]	3.1[2]	5.1[2]
Oracle	komplex	14[2]	25.4[2]	14.2[2]	25.5[2]
	einfach	4.5[2]	8.9[2]	4.6[2]	8.9[2]
DB2	komplex	8.2[1]	27[1]	8.2[1]	26.5[1]
	einfach	3.5[2]	4.8[2]	3.6[2]	5.2[2]

[1] explizite Sekundärspeicherrepräsentation
[2] implizite Sekundärspeicherrepräsentation

Abbildung 81: Vergleich Ein-/Auslagern von Schemata der versch. Varianten

RDBMS. Diese kann ausschließlich auf Basis der Schema-ID Ein- und Auslagern, während bei RDBMS (effiziente) Abfragen auf weitere, in dieser Arbeit nicht betrachtete Eigenschaften eines Schemas denkbar sind. Da diese Eigenschaften von RDBMS für Schemata allerdings nicht benötigt werden, wird hier aufgrund des Geschwindigkeitsvorteils die Speicherung in XML-Dateien empfohlen.

5.2.7.2 Instanzen In Abbildung 83 sind die Messwerte der einzelnen Speicherarten beim Ein- und Auslagern von Instanzen zusammengefasst. Auch hier wurden, wie schon bei Schemata, die jeweils empfohlenen Werte in die Diagramme aufgenommen.

Analog zur Betrachtung bei Schemata, lassen sich auch hier Performanzkoeffizienten berechnen. Es wird davon ausgegangen, dass Instanzen gleich oft gespeichert wie geladen werden, da diese zum Beispiel nach jedem Weiterschalten gespeichert werden müssen. Somit wird für den Vergleich der folgende Koeffizient herangezogen:

$$c_{Instanz} = (l_{einfach} + s_{einfach} + l_{komplex} + s_{komplex}) \cdot \frac{1}{2}$$

Er gibt an, wie lange es im Durchschnitt dauert, eine Instanz bei der gegebenen Sekundärspeichervariante und unter Abhängigkeit von der Primärspeicherrepräsentation zu laden und zu speichern. Die Werte sind in Abbildung 84 dargestellt.

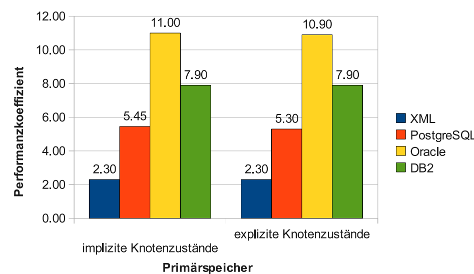
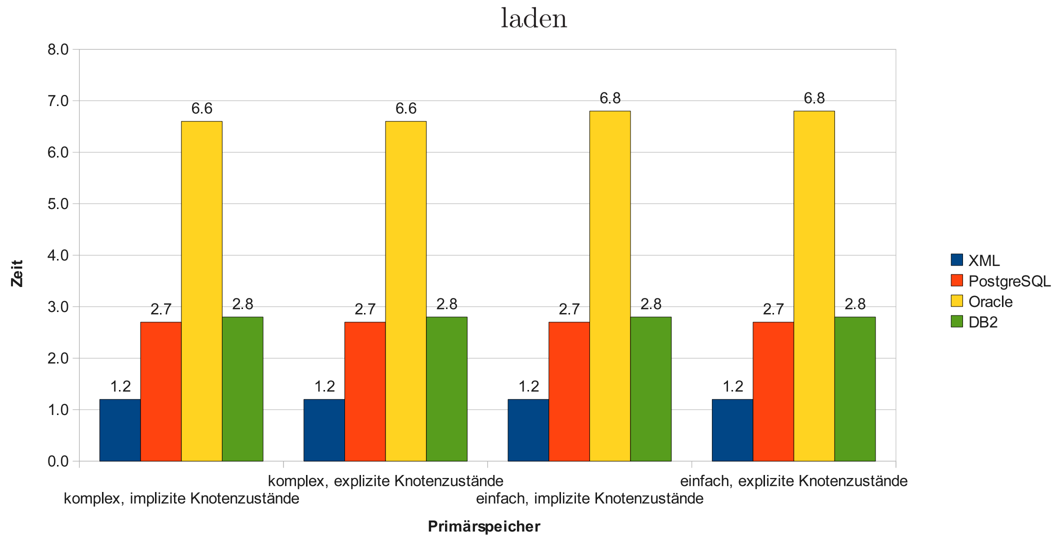
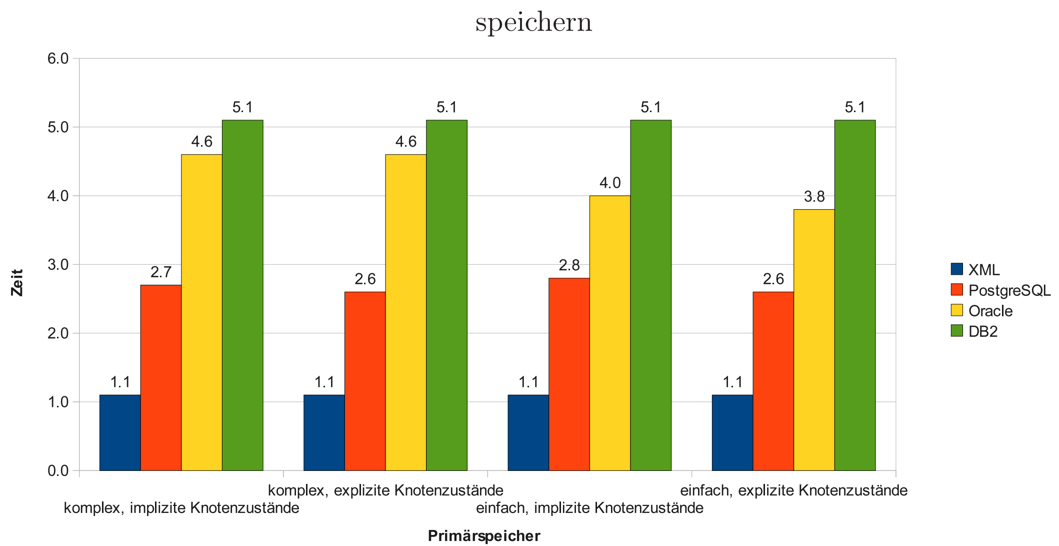


Abbildung 84: Performanzkoeffizienten $c_{Instanz}$ der einzelnen Varianten

Analog zur Betrachtung für Schemata ist auch bei der Betrachtung der Ein- und Auslagerung von Instanzen die XML-Datei-Speicherung zu empfehlen. Die Variante, welche am zweit schnellsten ist, die Speicherung mit PostgreSQL, braucht mehr als 2,3 mal so lange zum Laden und Speichern einer Instanz.

5.2.7.3 Anfragen auf Kollektionen Zusätzlich zum Ein- und Auslagern von Instanzen wurden auch Messungen anhand der Anfragen auf Kollektionen durchgeführt. Es fällt auf, dass bei allen untersuchten Sekundärspeichervarianten die implizite Repräsentation von Instanzen empfohlen wurde. Hierbei wurde bei allen RDBMS der XML-Datentyp eingesetzt. Bei Oracle kam die Hybridvariante zum Einsatz, welche keine Abfragen auf





		Primärspeicher			
		implizit		explizit	
Sekundärspeicher	Schema	laden	speichern	laden	speichern
XML	komplex	1.2[2]	1.1[2]	1.2[2]	1.1[2]
	einfach	1.2[2]	1.1[2]	1.2[2]	1.1[2]
PostgreSQL	komplex	2.7[3]	2.7[3]	2.7[3]	2.6[3]
	einfach	2.7[3]	2.8[3]	2.7[3]	2.6[3]
Oracle	komplex	6.6[3]	4.6[3]	6.6[3]	4.6[3]
	einfach	6.8[1]	4[1]	6.8[1]	3.8[1]
DB2	komplex	2.8[3]	5.1[3]	2.8[3]	5.1[3]
	einfach	2.8[3]	5.1[3]	2.8[3]	5.1[3]

[1] explizite Sekundärspeicherrepräsentation
[2] implizite Sekundärspeicherrepräsentation
[3] implizite Sekundärspeicherrepräsentation mit XML

Abbildung 83: Vergleich Ein-/Auslagern von Instanzen der versch. Varianten

die in XML vorliegenden Daten innerhalb des RDBMS tätigt. Die in Abbildung 85 zusammengestellten Werte relativieren die bisherigen Einschätzungen der Performanz der XML-Datei-Speicherung. Diese benötigt zum Beispiel bei der Anfrage nach allen Instanzen eines Schemas mehr als die 64-fache Zeit, die PostgreSQL benötigt. Dies liegt daran, dass die Repräsentationen auf Basis von RDBMS diese Information in einer Tabelle direkt vorliegen haben, die XML-Datei-Speicherung aber alle vorhandenen Dateien auf die Zugehörigkeit untersuchen muss.

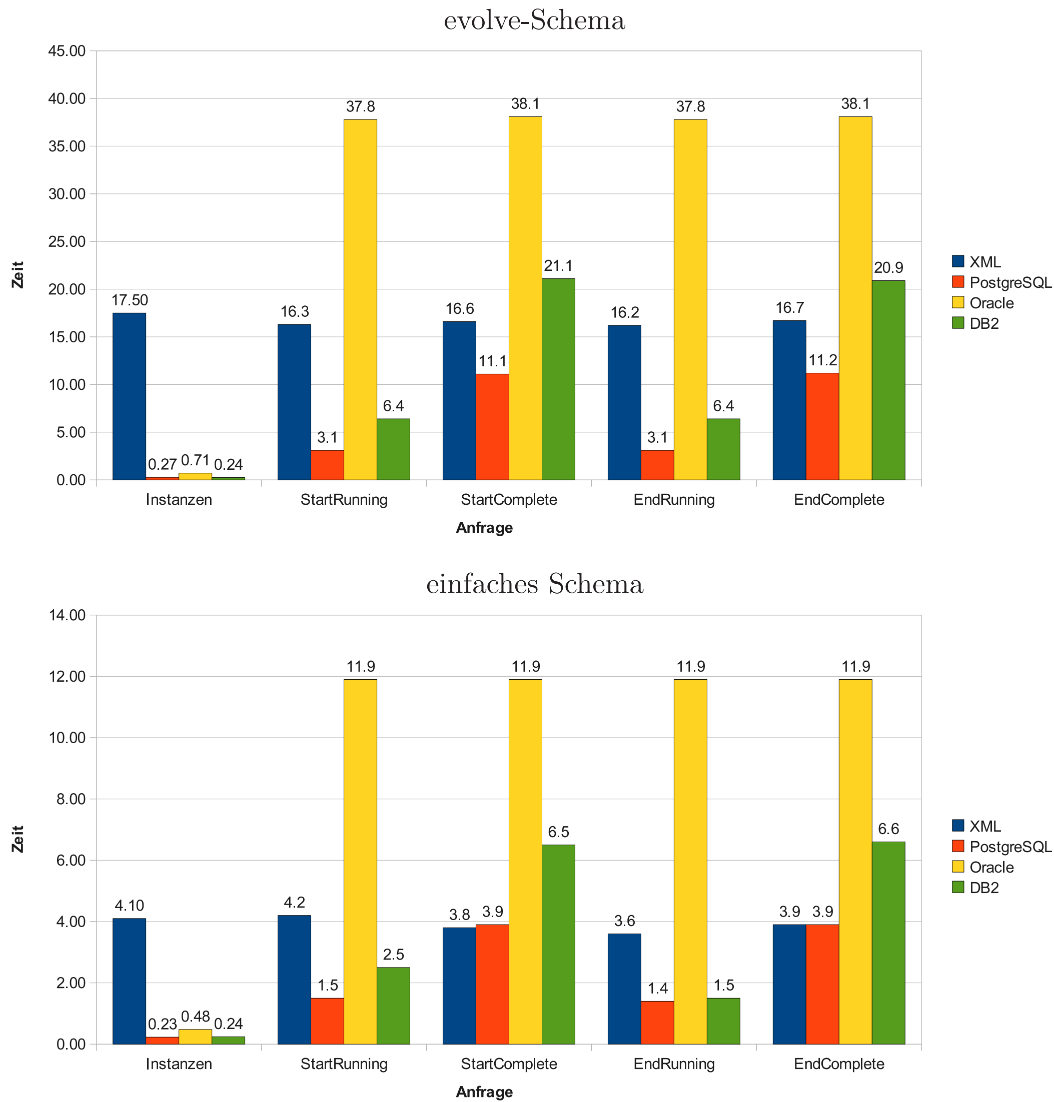


Abbildung 85: Vergleich der Anfragen auf Kollektionen der versch. Varianten

Für einen direkten Vergleich wird auch hier ein Performanzkoeffizient berechnet. Es wird davon ausgegangen, dass die Anfrage nach Instanzen eines Schemas wesentlich häufiger durchgeführt wird als die Anfrage nach einem bestimmten Knotenzustand. Ebenso wird angenommen, dass gleich viele einfache Schemata und Schemata vom Typ „evolve" vorhanden sind. Seien k_S^T die Werte aus Abbildung 85 mit $S = \{\text{einfach, evolve}\}$ und $T = \{\text{Instanzen, StartRunning, StartComplete, EndRunning, EndComplete}\}$, dann gilt für den Koeffizienten:

$$c_{Kollektion} = \left(\frac{50 \cdot k_{evolve}^{Instanzen} + k_{evolve}^{Zustand}}{54} + \frac{50 \cdot k_{einfach}^{Instanzen} + k_{einfach}^{Zustand}}{54}\right) \cdot \frac{1}{2}$$

mit

$$k_S^{Zustand} = k_S^{StartRunning} + k_S^{StartComplete} + k_S^{EndRunning} + k_S^{EndComplete}$$

Der Koeffizient repräsentiert also die durchschnittliche Laufzeit einer Anfrage an eine Kollektion, wenn man davon ausgeht, dass die Instanzen eines Schemas 50 mal häufiger angefragt werden als die Zustände von Knoten. Die in Abbildung 86 vorgestellten Koeffizienten zeigen, dass die XML-Datei-Speicherung sehr langsam ist. Um einen endgültigen Vergleich zu ziehen, müssen diese Werte mit denen des Ein- und Auslagerns von Instanzen aus Abbildung 84 vereint werden.

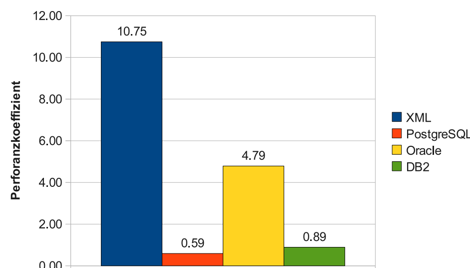


Abbildung 86: Performanzkoeffizienten $c_{Kollektion}$ der einzelnen Varianten

Diese Vereinigung stellt Abbildung 87 dar. Die Werte werden durch Addition der Werte aus den Abbildungen 84 und 86 berechnet. Dies ist anschaulich gesehen die Zeit, die im Durchschnitt benötigt wird, eine Instanz zu suchen (anhand der Zugehörigkeit zu einem bestimmten Schema oder auf Basis der Knotenzustände), diese zu laden und danach wieder zu speichern. Der Koeffizient stellt also einen für die Performanz der Sekundärspeichervariante von Instanzen aussagekräftigen Wert dar.

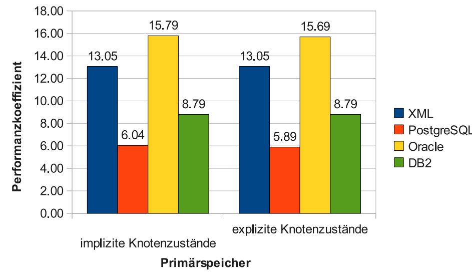


Abbildung 87: Vereinigte Performanzkoeffizienten der einzelnen Varianten

Es ist erkennbar, dass mit Einbeziehung der Anfragen auf Kollektionen nicht länger die XML-Datei-Speicherung die favorisierte Sekundärspeichervariante ist. PostgreSQL kann Anfragen dieser Art in 45-46% der Zeit beantworten und ist somit für den Einsatz zur Speicherung von Instanzen zu empfehlen.

5.2.8 Zusammenfassung

In diesem Abschnitt wurden die Sekundärspeicherrepräsentationen und die Sekundärspeichervarianten untersucht. Es zeigte sich, dass es am schnellsten ist, Schemata in XML-Dateien mit impliziter Blockstruktur zu speichern. Werden im Primärspeicher Schemata mit expliziter Blockstruktur eingesetzt, so ist es vorteilhaft, die Dateien für einfache Schemata allerdings in einer expliziten XML-Repräsentation zu persistieren. In Bezug auf Instanzen wurde festgestellt, dass es vorteilhaft ist, diese mithilfe von PostgreSQL zu speichern. Wird hierbei die Repräsentation mit impliziten Knotenzuständen gewählt und der XML-Datentyp für die Spalte der Knotenzustände eingesetzt, so ist diese Sekundärspeicherrepräsentation unabhängig von der eingesetzten im Primärspeicher nach Abschnitt 5.2.7 optimal.

5.3 Primärspeichermessungen

Nach der Betrachtung des Sekundärspeichers, werden in diesem Abschnitt Messergebnisse bezüglich des Primärspeichers vorgestellt. Hierbei wurde für Messungen, bei denen aufgrund der Architektur des Testsystems (vergleiche Abschnitt 4.1.2) eine `TemplateStorage` oder `InstanceStorage` benötigt wird, diejenige Implementierung gewählt, welche auf flüchtigem Speicher beruht (vergleiche Abschnitt 4.3.6). Nachdem die verschiedenen Primärspeicherrepräsentationen von Schemata und Instanzen betrachtet wurden, wird in Bezug auf die grobe Clusterung überprüft, in wie weit sich die Optimierungen durch Messungen bestätigen lassen, welche sich durch Einsatz der groben Clusterung nach den konzeptionellen Betrachtungen aus Abschnitt 3.3 ergeben.

5.3.1 Schemata und Instanzen

Zur Messung des Primärspeicherverbrauchs von Schemata wurden Schemata mit unterschiedlicher Anzahl Knoten generiert und deren Speicherverbrauch gemessen [Rou02]. Ebenso wurde der prozentuale Anteil an Blöcken variiert. Die Ergebnisse stellt Abbildung 88 dar.

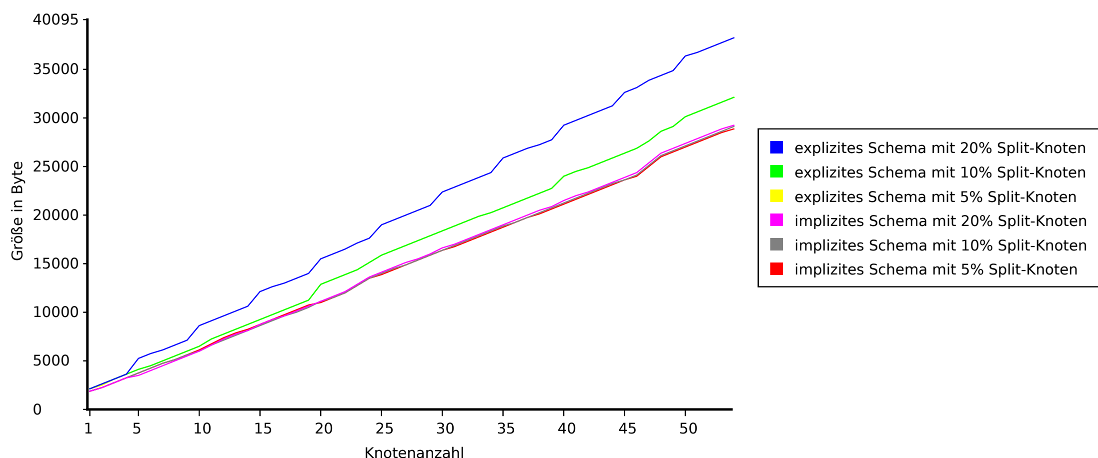
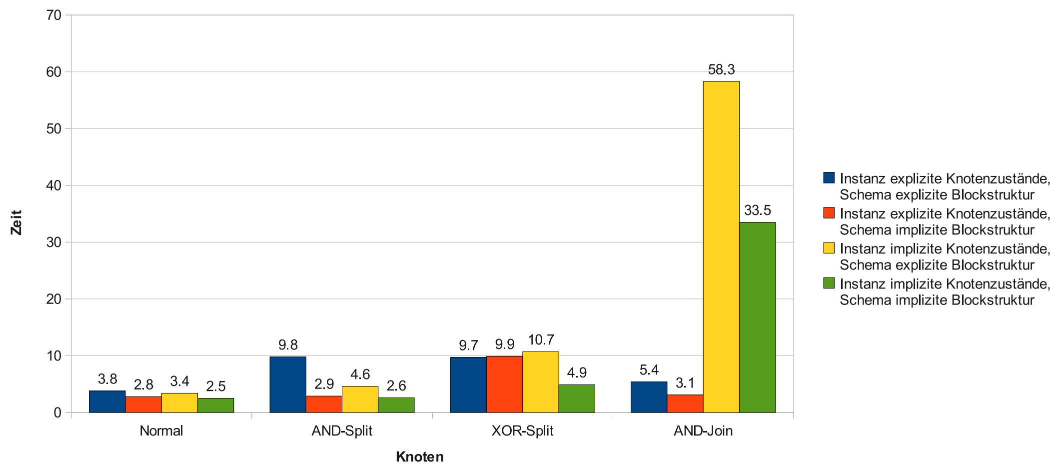


Abbildung 88: Primärspeicherverbrauch von Schemata

Für explizite Schemata mit 10% und 20% Split-Knoten sind deutlich die Aufkommen von neuen Blöcken an den treppenartigen Abweichungen zu erkennen. Die explizite Repräsentation mit 5% Split-Knoten gleicht sich in ihrem Speicherverbrauch dem der impliziten Repräsentationen an. Auch hierbei sind treppenartige Abweichungen zu erkennen. Diese repräsentieren bei der expliziten Repräsentation die Aufkommen neuer Blöcke, bei den impliziten Repräsentationen ist dieser Speichermehrverbrauch in der Vergrößerung der Hashtabelle der eingesetzten `HashMap` begründet. Allgemein ist die Repräsentation mit impliziter Blockstruktur im Primärspeicher zu empfehlen, da diese unabhängig vom Blockanteil wenig Speicher benötigt. Dieses Ergebnis bestätigt die in Abschnitt 3.2.3 erarbeiteten qualitativen Untersuchungen des Speicherverbrauchs von Schemata.

In Abschnitt 3.2.3 wurde allerdings eine längere Laufzeit bei Anfragen an implizite gegenüber expliziten Schemata aufgezeigt. Um dies zu messen, wurden Instanzen weitergeschaltet, welche auf Schemata der unterschiedlichen Repräsentationen beruhen, und die Laufzeit gemessen. Hierbei kam das komplexe Schema zum Einsatz. Das Ergebnis ist in Abbildung 89 zu sehen. Zu beachten ist, dass diese Werte im Bereich von Tausendstel des Grundwertes liegen (siehe Abschnitt 5.1). Bei den in der Tabelle markierten Werten traten bei der Wiederholung der Messung starke, periodische Schwankungen im Ergebnis auf. Die dargestellten Werte bilden den Mittelwert und sollten aufgrund dessen mit Vorsicht betrachtet werden. Die jeweiligen Extremwerte sind angegeben. Warum die Er-

gebnisse bei diesen Werten so stark schwanken und warum diese Schwankungen nahezu exakt periodisch sind, konnte im Rahmen dieser Arbeit nicht geklärt werden.



Knoten	Schema	Instanz explizit	Instanz implizit
NORMAL	explizit	3.8	3.4
	implizit	2.8	2.5
AND-Split	explizit	9.8* (4.1-18.5)	4.6
	implizit	2.9	2.6
XOR-Split	explizit	9.7* (7.2-18.4)	10.7* (8.0-19.3)
	implizit	9.9* (4.4-18.3)	4.9
AND-Join	explizit	5.4	58.3* (49.5-60.7)
	implizit	3.1	33.5* (22.1-38.8)

* sehr starke periodische Schwankungen im Ergebnis

Abbildung 89: Laufzeit beim Weiterschalten (in Tausendstel des Grundwertes)

Es ist zu erkennen, dass Instanzen, welche auf impliziten Schemata beruhen entgegen der qualitativen Untersuchungen aus Abschnitt 3.2.3 durchgehend schneller weitergeschaltet werden können als explizite Schemata. Dies liegt daran, dass das Weiterschalten keine der Methoden von Schemata aufruft, welche die Blockstruktur von Schemata liefern und für welche die explizite Primärspeicherrepräsentation schneller ist. Aufgrund dessen und aufgrund des geringeren Speicheraufwands für implizite Schemata, sind diese für den Einsatz im Primärspeicher zu empfehlen.

Bei den in Abbildung 89 vorgestellten Messergebnissen wurden zuerst ein normaler Knoten (Knoten 3 des komplexen Schemas), ein AND-Split- (Knoten 2) und ein XOR-Split-Knoten (Knoten 5) weitergeschaltet. Es stellt sich heraus, dass der normale Knoten am schnellsten weitergeschaltet wird. Der AND-Split-Knoten muss mehrere nachfolgende

Knoten als RUNNING markieren und benötigt deshalb mehr Zeit. Beim Weiterschalten des XOR-Split-Knoten, bei dem der Zweig von Knoten 10 ausgewählt wurde, fällt zusätzlicher Aufwand an, da der zu aktivierende Zweig zuerst gefunden werden muss. Hierfür müssen alle nachfolgenden Knoten untersucht werden, wofür zusätzliche Zeit benötigt wird. Die Knoten der nicht gewählten Zweige werden hierbei zusätzlich als SKIPPED markiert, was ebenfalls Zeit benötigt. Es ist offensichtlich, dass das Weiterschalten mehr Zeit benötigt, je mehr Knoten als SKIPPED markiert werden müssen. Somit hängt dieser Wert direkt mit dem zugrunde liegenden Schema zusammen. Im vorliegenden Fall wurden neun Knoten als SKIPPED markiert. Neben diesen Knoten wurde außerdem zu einen AND-Join-Knoten (zu Knoten 23) weitergeschaltet. Bevor dieser aktiviert werden kann, muss überprüft werden, ob alle Zweige abgearbeitet sind. Dies ist bei einer Instanz mit impliziter Knotenmarkierung sehr zeitintensiv, da die vorgelagerten Zweige viele Knoten beinhalten und somit die Rekonstruktion der Knotenzustände aus den impliziten Informationen viel Zeit in Anspruch nimmt. Obwohl bei einer Repräsentation mit impliziten Knotenzuständen alle anderen Knotentypen schneller weitergeschaltet werden können, wird dennoch die explizite Repräsentation empfohlen, da diese zum Weiterschalten eines AND-Join-Knotens im vorliegenden Fall nur etwa ein Elftel der Zeit benötigt und keine allgemeinen Aussagen zu der Anzahl der Zweige vor einem AND-Join-Knoten oder der Anzahl der sich darauf befindenden Knoten getroffen werden können.

Neben der Laufzeit des Weiterschaltens wurde auch der Speicherverbrauch der unterschiedlichen Instanzrepräsentationen untersucht. Dieser ist in Abbildung 90 dargestellt. Bei der expliziten Repräsentation sind die treppenartigen Anstiege bei der Vergrößerung der Hashtabelle der eingesetzten `HashMap` zu erkennen. Erwartungsgemäß ist der Speicherverbrauch einer mit impliziten Knotenzuständen gespeicherten Instanz konstant. Die in Abbildung 90 untersuchten Instanzen hatten jeweils ausschließlich den Start-Knoten im Zustand RUNNING.

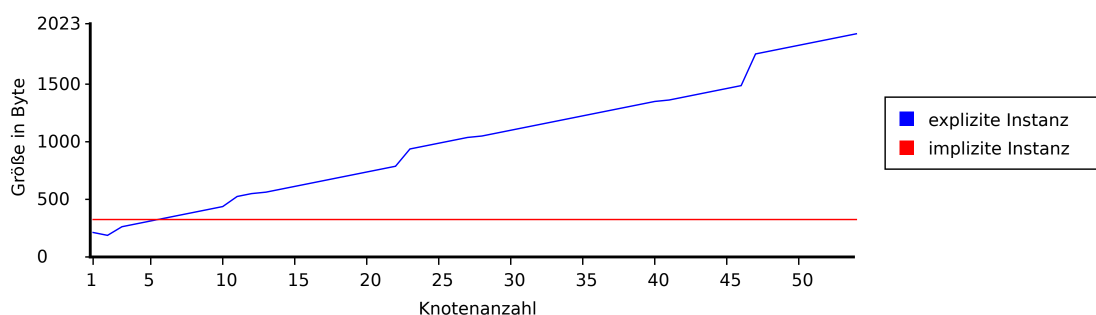


Abbildung 90: Primärspeicherverbrauch von Instanzen

Betrachtet man die Laufzeit des Weiterschaltens und den Speicherverbrauch für die beiden Repräsentationen von Instanzen, so stellt sich für Instanzen mit weniger oder genau fünf Knoten heraus, dass die explizite Repräsentation zu empfehlen ist, da diese im Durchschnitt sowohl schneller weitergeschaltet als auch weniger Speicherplatz benötigt. Sollte die Instanz mehr als fünf Knoten haben, so ist keine eindeutige Empfehlung

zu geben. Dies ist abhängig von dem zur Verfügung stehenden Speicher. Ist genügend Speicher für die Instanzen vorhanden, ist es ratsam, die explizite Repräsentation einzusetzen. Im gegensätzlichen Fall die implizite. Es ist auch denkbar, die Repräsentation während der Laufzeit des PMS zu wechseln. Ist ein bestimmter Füllgrad des Speichers nicht erreicht, werden Instanzen mit expliziten Knotenmarkierungen benutzt. Sobald der Füllstand überschritten ist, werden Instanzen mit impliziten Knotenzuständen repräsentiert. Dies ist in der Praxis zum Beispiel durch Einsatz einer Schnittstelle ähnlich der `Instance`-Schnittstelle (siehe Abbildung 39) leicht umsetzbar, da es für die restlichen Teile des PMS transparent ist, wie die Knotenzustände repräsentiert werden. Das „Umschalten“ der Repräsentation kann hierbei auf die Weise geschehen, dass nach Erreichen des Füllstandes (oder der Unterschreitung des Füllstandes) die Persistenzschicht, also bei der hier vorgestellten Architektur die `InstanceStorage`-Implementierung, die neu angefragten Instanzen in der entsprechenden Repräsentation liefert. Ebenso ist mithilfe von entsprechend zu wählenden Architekturaspekten eine Propagierung der „neuen“ Repräsentation auf alle im Speicher vorhandenen Instanzen denkbar.

5.3.2 Clusterung

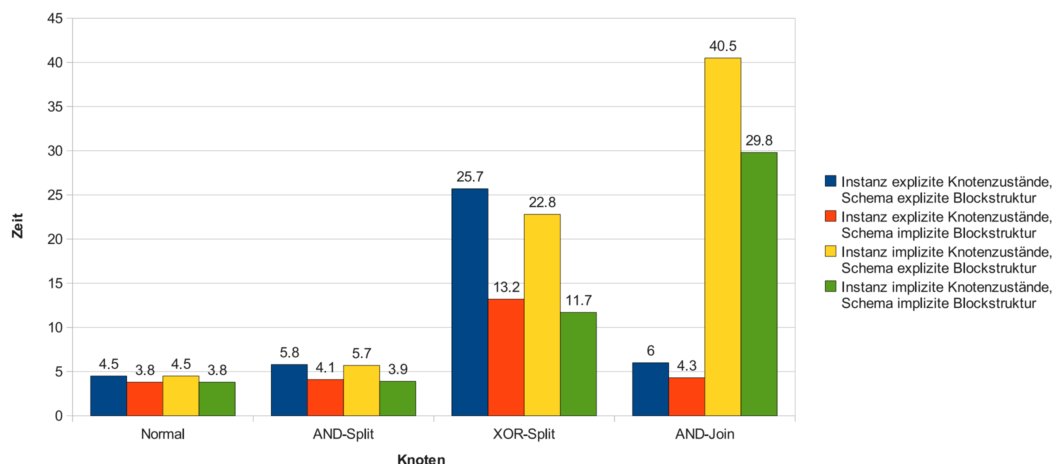
Neben Schemata und Instanzen wurde auch die grobe Clusterung im Primärspeicher untersucht. Hierbei sind einerseits die Messwerte des Weiterschaltens zu beachten. Nachdem diese vorgestellt wurden, werden die Messergebnisse der Laufzeit des Algorithmus `preEvolveSelection` betrachtet.

Die Tabelle in Abbildung 91 stellt die Messwerte des Weiterschaltens mit Neuberechnung der Clusterzugehörigkeit der ohne Neuberechnung aus Abbildung 89 gegenüber.

Lässt man die aufgrund von Schwankungen markierten Werte außer Acht, so bestätigt sich, dass das Weiterschalten mit Neuberechnung der Clusterzugehörigkeit langsamer ist als ohne Neuberechnung. Geht man davon aus, dass, wie bereits empfohlen, Schemata mit impliziter Blockstruktur und Instanzen mit expliziten Knotenmarkierungen zum Einsatz kommen, ist durch die Neuberechnung der Clusterzugehörigkeit eine Verschlechterung der Laufzeit des Weiterschaltens zwischen 35.7% und 41.4% zu erwarten.

Neben dem Weiterschalten ist bei der Messung der groben Clusterung die Laufzeit des Algorithmus `preEvolveSelection` zu beachten, welche in Tabelle 2 dargestellt ist. Hierbei wurde ein Schema vom Typ „evolve“ (siehe Abbildung 63) eingesetzt und 20 Instanzen auf Basis dessen erstellt. Für jeden Knoten des Schemas existiert genau eine Instanz, welche diesen Knoten im Zustand RUNNING hat. Es wurden die Laufzeiten des Algorithmus auf Basis der Änderungen gemessen, welche aus Abbildung 92 ersichtlich sind.

Es ist zu erkennen, dass durch geringen Zeitaufwand meist der Großteil der Instanzen als „migrierbar“ und „nicht migrierbar“ kategorisiert werden kann und nur ein Teil der Instanzen nach den Methoden aus [Rin04, Jur06] genauer untersucht und somit eingelagert werden muss. Geht man davon aus, dass die in Abschnitt 5.2 empfohlene Sekundärspeicherrepräsentation von Instanzen mithilfe von PostgreSQL eingesetzt wird, hat das



		mit Neuberechnung Instanz				ohne Neuberechnung Instanz	
Knoten	Schema	explizit		implizit		explizit	implizit
NORMAL	explizit	4.5	+18.4%	4.5	+32.4%	3.8	3.4
	implizit	3.8	+35.7%	3.8	+52.0%	2.8	2.5
AND-Split	explizit	5.8		5.7	+23.9%	9.8*	4.6
	implizit	4.1	+41.4%	3.9	+50%	2.9	2.6
XOR-Split	explizit	25.7		22.8		9.7*	10.7*
	implizit	13.2		11.7	+138.8%	9.9*	4.9
AND-Join	explizit	6.0	+11.1%	40.5		5.4	58.3*
	implizit	4.3	+38.7%	29.8		3.1	33.5*

* sehr starke periodische Schwankungen im Ergebnis

Abbildung 91: Weiterschalten mit & ohne Clusterung (in Tausendstel)

Einlagern einer Instanz etwa einen Zeitaufwand von 2.7 Zeiteinheiten (vergleiche Abbildung 83). Somit lassen sich durch den Einsatz der groben Clusterung 16.2 bis 32.4 Zeiteinheiten sparen, da 6 bis 12 Instanzen weniger eingelagert werden müssen. Ohne die grobe Clusterung fallen 54 Zeiteinheiten an, es entsteht somit eine Zeitersparnis von 30% bis 60%. Hierbei ist die Zeit nicht mit eingerechnet, die zur Untersuchung der Instanzen nach [Rin04, Jur06] nötig ist und somit durch Einsatz der groben Clusterung für 6 bis 12 Instanzen gespart werden kann.

Diese Werte können allerdings nur als Beispiel dienen. Es lässt sich keine allgemein gültige Aussage daraus ableiten, da die Ergebnisse sehr stark vom ursprünglichen Schema und den Änderungen darauf abhängen. So wird der Algorithmus zum Beispiel bei einem Schema, welches keinen Split-Knoten besitzt, also der Wurzelblock der einzige Block

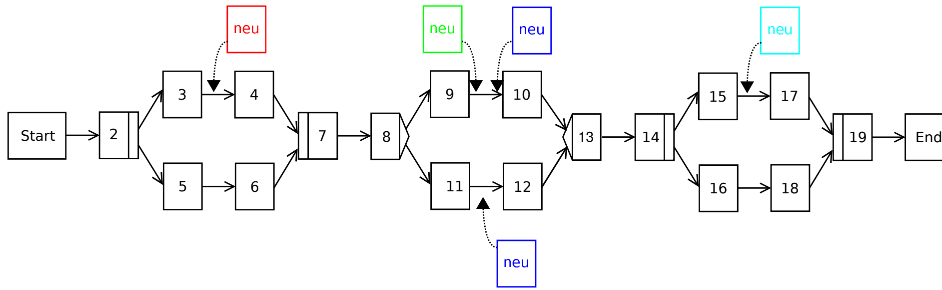


Abbildung 92: Die untersuchten Änderungen am evolve-Schema

Knoten einfügen zwischen	Laufzeit	Ergebnis
3 → 4	77.4	(0 - 12 - 8)
9 → 10	84.1	(6 - 0 - 14)
9 → 10 und 11 → 12	90.2	(6 - 6 - 8)
15 → 16	76.7	(12 - 0 - 8)

Ergebnis: (migrierbar - nicht migrierbar - genauer untersuchen)

Tabelle 2: Laufzeit von `preEvolveSelection` (in Tausendstel)

ist, keinen Vorteil bieten, da alle Instanzen als „genauer zu untersuchen" kategorisiert werden, weil alle vom Wurzelcluster (dem einzigen Cluster) referenziert werden (vergleiche Abschnitt 3.3.1.2). Allgemein kann die Aussage getroffen werden, dass die grobe Clusterung mehr Zeitersparnis bei einer Schemaevolution bieten kann, je mehr Split-Knoten im zugrunde liegenden Schema vorhanden sind. Dies folgt, weil jeder zusätzliche Split-Knoten einen zusätzlichen Cluster bedeutet, anhand dessen Instanzen durch `preEvolveSelection` vor dem Einlagern kategorisiert werden können. Außerdem ist es vorteilhaft, wenn möglichst viele dieser Split-Knoten einen Block aufspannen, der dem Wurzelblock direkt untergeordnet ist, da diese potentiell Stellen aufspannen. Je mehr (potentielle) Stellen in einem Schema vorhanden sind, desto genauer kann die Position einer Änderung mithilfe der groben Clusterung bestimmt und damit mehr Instanzen aus anderen Clustern kategorisiert werden.

Aufgrund der starken Verschlechterung der Laufzeit des Weiterschaltens durch die Neuberechnung der Clusterzugehörigkeit und der starken Abhängigkeit der Zeitersparnis bei einer Schemaevolution vom zugrunde liegenden Schema und den vorgenommenen Änderungen, kann keine allgemeine Empfehlung zum Einsatz der groben Clusterung in der Praxis gegeben werden. Es ist denkbar, dass diese von einem Benutzer des PMS beim Modellieren eines Schemas manuell eingeschaltet wird, wenn sich das Schema für die grobe Clusterung eignet und/oder erwartet wird, dass Schemaevolutionen auf diesem Schema durchgeführt werden.

5.4 Fazit

Nachdem die Grundlagen und Konzepte und deren Umsetzung in den vorigen Kapiteln beschrieben wurden, runden die Messungen aus diesem Kapitel die Arbeit ab. Es wurden verschiedene Repräsentationen sowohl von Schemata als auch von Instanzen untersucht. Bei Schemata wurde durch die Messungen festgestellt, dass die Vorteile, welche eine Repräsentation mit expliziter Blockstruktur bei Anfragen bezüglich dieser bietet, in der Praxis nicht zum Tragen kommen. Schemata mit impliziter Blockstruktur sind sowohl im Primär- als auch im Sekundärspeicher vorzuziehen, sofern die empfohlene XML-Datei-Speicherung eingesetzt wird. Dies wurde durch Messungen des Ein- und Auslagerns der Daten, als auch durch Messungen des Weiterschaltens von Instanzen bestimmt.

Für Instanzen wurden ebenfalls die beiden Repräsentationen untersucht. Hierbei stellte sich heraus, dass diese im Sekundärspeicher mithilfe von PostgreSQL und einer Repräsentation mit impliziten Knotenzuständen gespeichert werden sollten. Hierbei ist der Einsatz des XML-Datentyps für die Spalte der serialisierten Knotenzustände empfehlenswert. Diese Variante bietet die performantesten Zugriffe auf Instanzen, sofern sowohl das Ein- und Auslagern als auch Anfragen auf Kollektionen beachtet werden. Im Primärspeicher sollten Instanzen hingegen in einer Repräsentation mit expliziten Knotenzuständen dargestellt werden. Diese bietet eine bessere Performanz beim Weiterschalten, da die implizite Repräsentation hier bei AND-Join-Knoten sehr schlecht abschneidet.

Nicht empfehlenswert ist der Einsatz von JPA zur Persistierung von Daten, da das Ein- und Auslagern dabei wesentlich mehr Zeit benötigt als bei einer direkten Speicherung. Ebenfalls nicht eingesetzt werden sollte Oracle, da die Performanz hierbei nicht ausreichend ist. Auffallend war dies besonders bei Anfragen, welche auf serialisierte XML-Daten getätigt wurden. Dies ist aber auch beim Betrachten der Abbildungen 82 und 87, welche die Ergebnisse der einzelnen Varianten zusammenfassen, leicht ersichtlich.

Mit den in Abschnitt 4.3.5 vorgestellten Zwischenspeicher-Implementierungen von `TemplateStorage` und `InstanceStorage` wurden keine Messungen durchgeführt. Dies liegt daran, dass der Vorteil, den ein „Zwischenschalten“ dieser Klassen bringt, stark vom Zugriffsverhalten auf Schemata und Instanzen abhängig ist und somit keine allgemeinen Messungen durch die Testumgebung möglich sind. Der Einsatz solcher Zwischenspeicher ist allerdings generell zu empfehlen, sofern genug Hauptspeicher vorhanden ist, da hierdurch „teure“ Zugriffe auf den Sekundärspeicher beim Laden von Daten unter Umständen verhindert werden.

Es kann keine allgemeine Empfehlung für den Einsatz der groben Clusterung aus Abschnitt 3.3 abgegeben werden. Für deren Einsatz ist es notwendig, die Clusterzugehörigkeit beim Weiterschalten neu zu berechnen, was diesen Vorgang stark verlangsamt. Außerdem kann keine allgemeine Einschätzung über die Zeiteinsparungen getroffen werden, welche die grobe Clusterung bei einer Schemaevolution bringt. Diese hängt stark vom Schema und von den Änderungen daran ab. Somit ist denkbar, die grobe Clusterung durch manuelles Einschalten während dem Modellieren von Schemata einzusetzen.

Dadurch kann eine manuelle Einschätzung getroffen werden, ob die grobe Clusterung bei einer Schemaevolution dieses Schemas einen Vorteil bringt. Allgemein gilt die Aussage, dass die grobe Clusterung mehr Vorteile bietet, je mehr Split-Knoten im Schema vorhanden sind. Ebenso kann die grobe Clusterung manuell eingeschaltet werden, wenn bereits beim Modellieren des Schemas bekannt ist, dass Schemaevolutionen auf diesem durchgeführt werden.

6 Verwandte Konzepte

Im Rahmen dieser Arbeit wurde bisher von interpretierten Graphstrukturen als interne Repräsentation von Prozessen ausgegangen. Dieses Kapitel rekapituliert die Eigenschaften einer solche Repräsentation und stellt andere mit deren Eigenschaften vor.

Bei einer Repräsentation mit *interpretierten Graphstrukturen* modelliert der Benutzer einen Prozess als Graph, welcher intern vom PMS auch als solcher repräsentiert wird [Rei00, HJKW96]. Dieses interpretiert den Graph und führt auf dieser Basis Aktivitäten aus, welche den Knoten zugeordnet sind. Anfallende Daten werden ebenfalls durch interpretieren einer geeigneten Graphmodellierung abgebildet. Diese Funktionen führt das PMS auf einem zentralen Server aus, welcher gleichzeitig Mitarbeitern Überprüfungsmöglichkeiten, wie zum Beispiel die Anfragen auf Kollektionen aus Abschnitt 2.4.3 anbieten kann. Neben der im Laufe dieser Arbeit betrachteten Anwendung der Schemareferenz von Instanzen, ist es auch möglich, die Schemadaten beim Erzeugen einer Instanz zu kopieren. Dieses Vorgehen beinhaltet eine einfachere Anwendung von instanzbasierten Änderungen. Hierbei ist keine Deltaschicht notwendig, welche die Änderungen gegenüber einem zugrunde liegendem Schema abbildet, sondern die Änderungen können direkt im instanzeigenen Graph realisiert werden. Nachteil dieser Repräsentation ist, dass diese sehr viel Speicher benötigt, da jede Instanz die volle Repräsentation der (redundante) Schemadaten hält.

Neben interpretierten Graphstrukturen können Prozesse auch als *verteilte Prozesspartikel* dargestellt werden [DKM+97, WWWK96]. Dieser Ansatz verfolgt die Idee, jeden Prozessschritt in einen eigenständigen Automaten [Sch01b] zu übersetzen. Diese Automaten kommunizieren unabhängig von einem zentralen Server über (Netzwerk-)Nachrichten miteinander und bilden so den Kontroll- und Datenfluss ab. Jeder dieser Automaten besitzt neben der Logik zur Ausführung des eigentlichen Prozessschrittes Vorbedingungen und nachgelagerte Aktivitäten. Sind alle Vorbedingungen erfüllt, also alle Nachrichten von vorgelagerten Automaten korrekt eingetroffen, wird die Ausführung des Prozessschrittes aktiviert. Danach werden alle nachgelagerten Aktivitäten abgearbeitet. Dies bedeutet, dass allen nachgelagerten Automaten die entsprechenden Nachrichten gesendet werden. Der Vorteil dieser Variante besteht darin, dass sie ohne einen zentralen Server auskommt und die eigenständigen Automaten somit beliebig verteilt werden können: Es ist denkbar, diese auf beliebigen Rechnern an beliebigen Orten auszuführen, was zum Beispiel in Hinblick auf datenschutzrechtliche Bestimmungen von Vorteil sein kann. Hieraus folgt allerdings, dass keine einfache globale Steuerungsmöglichkeit des Prozesses besteht. Ebenso sind instanzbasierte Änderungen nicht möglich, da das Verhalten der einzelnen verteilten Automaten nicht auf Instanzbasis verändert werden kann.

Als weitere Möglichkeit lassen sich Prozesse als *Regelmenge* repräsentieren [DHL90, BMR96, KPRR95, Joe99]. Hierbei wird der Prozess ebenfalls als Graph modelliert, zur Repräsentation im PMS allerdings zu einer Regelmenge übersetzt. Diese Regeln werden durch Eintreten von Ereignissen getriggert. Hierbei kann eine Regel zusätzliche Bedingungen enthalten, wie zum Beispiel die Überprüfung auf eine bestimmte Tages-

zeit. Sind alle Bedingungen erfüllt, wird eine Aktion ausgeführt, welche nach Abarbeitung wiederum Ereignisse auslösen kann. Abgebildet werden können solche regelbasierten Systeme zum Beispiel durch (erweitere) RDBMS. Hierbei bilden Datenbank-Trigger [Pos09b, Ora09b, IBM09b] die Regeln ab. Innerhalb dieser können verschiedene Bedingungen geprüft und Aktionen ausgeführt werden. Der Datenfluss wird bei diesem Konzept ebenfalls durch die Regeln definiert, so kann in Datenbank-Triggern zum Beispiel auf bestimmte Felder bestimmter Tabellen zugegriffen werden. Regelbasierte PMS benötigen wenig zusätzlichen Entwicklungsaufwand, da viele RDBMS die Möglichkeit von Datenbank-Triggern bereits bieten. Die Prozessmodellierung ist allerdings aufgrund der großen Zahl an Regeln, welche zur Modellierung benötigt werden, sehr komplex. Eine weitere Auswirkung der Größe der Regelmenge ist, dass diese schnell unübersichtlich und somit schlecht wartbar wird. Ebenso werden instanzbasierte Änderungen nur sehr bedingt unterstützt.

Ebenso lässt sich eine weitere Repräsentation *direkt auf RDBMS* definieren. Hierbei besteht eine Instanz aus einem Eintrag in einer Tabelle. Diese besitzt außer den Daten des Prozesses an sich noch eine weitere Spalte, welche den Zustand angibt. Auf Basis dieses Zustandsfelds kann eine Selektion zur Ansicht für den Benutzer generiert werden. Dies ist zum Beispiel durch Einsatz von Views möglich [ISO92]. Hierbei existiert für jeden Benutzer eine View, welche die Einträge in der Tabelle auf Basis des Zustandsfelds filtert. Somit bekommt der Benutzer diejenigen Instanzen in genau den Zuständen angezeigt, für die er als Bearbeiter vorgesehen ist. Nachteil ist, dass auch hier instanzbasierte Änderungen nur sehr schwierig umsetzbar sind.

Prozesse lassen sich intern auch als Menge von Operationen repräsentieren, wobei die einzelnen *(Prozess-)Instanzen als Datenobjekte* von Operation zu Operation weitergereicht werden [KRW90, BMR96]. Hierbei ist allerdings die Realisierung von Parallelverzweigungen sehr schwierig, da die Instanz nur mithilfe eines Datenobjekts repräsentiert wird. Somit kann nicht das gleiche Datenobjekt an zwei unterschiedliche Operationen geleitet werden. Dies wäre durch eine Duplizierung des Datenobjekts zwar möglich, diese Datenobjekte müssten beim Zusammenführen der Parallelverzweigung allerdings aufwändig vereint werden. Auch die Unterstützung von instanzbasierten Änderungen ist nur sehr eingeschränkt möglich.

In diesem Kapitel wurden verschiedene Konzepte vorgestellt, Prozesse zu repräsentieren. Hierzu zählen die in dieser Arbeit untersuchten interpretierten Graphstrukturen, aber auch eine Repräsentation als verteilte Prozesspartikel oder als Regelmenge. Ebenso lassen sich Prozesse direkt auf RDBMS mithilfe von Views repräsentieren. Als letzte Möglichkeit wurde eine Repräsentation mithilfe von Datenobjekten vorgestellt, welche zwischen Operationen einer festen Menge weitergereicht werden.

7 Zusammenfassung und Ausblick

PMS erlangen aufgrund der anwachsenden Globalisierung in den letzten Jahren immer mehr Aufmerksamkeit. Die Systeme müssen in Zukunft auch von weltumspannenden Firmen eingesetzt werden können. Hierzu ist eine performante Verarbeitung von Anfragen, welche an die Systeme gestellt werden, von zentraler Bedeutung. Dies muss auch gewährleistet sein, wenn viele Prozesse gleichzeitig ablaufen. Neben der eingesetzten Softwarearchitektur spielt hierbei die Repräsentation der Prozessdaten eine zentrale Rolle.

In dieser Arbeit wurden verschiedene Varianten der Repräsentation von Prozessdaten untersucht. Durch die Untersuchungen wurden optimal geeignete Repräsentationen der Prozessdaten, welche sich in Schema- und Instanzdaten gliedern, sowohl im Primär- als auch im Sekundärspeicher ermittelt. Dieses Kapitel fasst die Arbeit zusammen und stellt Ideen zu weiteren Untersuchungen des Themengebiets vor.

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden Varianten zur Repräsentation von Prozessdaten untersucht. Diese gliedern sich in Schemata und Instanzen. Hierbei existieren für diese beiden jeweils zwei Repräsentationen, welche sowohl qualitativ als auch quantitativ untersucht wurden.

Nach der Vorstellung der Grundlagen wurden darauf aufbauende Konzepte erläutert. Hierzu zählt die Partitionierung von Schemata. Mithilfe dieser ist es möglich, nur vom System benötigte Daten im Primärspeicher zu halten. Momentan nicht benötigte Daten können dynamisch aus dem Sekundärspeicher nachgeladen und in die vorhandenen Primärspeicherdaten eingegliedert werden. Hierfür wurde zunächst eine konzeptionelle Umsetzung beschrieben. Im Anschluss wurde die grobe Clusterung vorgestellt, mit deren Hilfe die Laufzeit der Schemaevolution verringert werden kann. Hierbei werden Instanzen in einem Clusterbaum gehalten, welcher die grobe Position angibt, an der die Instanz in ihrer Ausführung steht. Es wurden Algorithmen vorgestellt, mit deren Hilfe die nötige zustandsbasierte Verträglichkeit allein anhand der Einordnung einer Instanz im Clusterbaum überprüft werden kann. Da die grobe Clusterung den Zustand einer Instanz nicht genau angibt, können auf dieser Basis nicht alle Instanzen bewertet werden und ein Teil muss weiterhin mit den bekannten Methoden [Rin04, Jur06] untersucht werden. Dafür wird bei Einsatz der groben Clusterung nur wenig Speicherplatz benötigt, was das Verfahren praxistauglich macht.

Der konzeptionellen Untersuchung schloss sich eine Umsetzung an, welche die vorgestellten Grundlagen und Konzepte implementiert. Hierbei wurden die verschiedenen ermittelten Repräsentationen mithilfe von Java™ im Primärspeicher umgesetzt. Da auch eine persistente Speicherung der Prozessdaten in der Praxis notwendig ist, wurden die verschiedenen Repräsentationen auch als Sekundärspeicherstrukturen umgesetzt. Hierbei

wurde auf eine XML-Datei-Speicherung und die Persistierung mithilfe verschiedener, am Markt erhältlicher relationalen Datenbanksysteme zurückgegriffen.

Mithilfe dieser Implementierung wurden Messungen anhand von verschiedenen Anwendungsszenarien durchgeführt. Hierbei wurden neben dem Ein- und Auslagern der Prozessdaten auch Anfragen auf Kollektionen betrachtet. Diese Sekundärspeichermessungen wurden für die verschiedenen Sekundärspeichervarianten durchgeführt. Deren Ergebnisse wurden anschließend verglichen und sowohl die optimale Sekundärspeicherrepräsentation, als auch die optimale Sekundärspeichervariante ermittelt. Es wurden ebenfalls Messungen der verschiedenen Primärspeicherrepräsentationen durchgeführt. Hierbei wurde ebenfalls eine optimale Repräsentation anhand eines Anwendungsszenarios sowohl für Schemata als auch Instanzen ermittelt. Abschließend wurde die Performanz der groben Clusterung gemessen und überprüft, ob diese den konzeptionellen Erwartungen gerecht wird.

Aufgrund der qualitativen und quantitativen Untersuchungen wurden optimale Repräsentationen der einzelnen Datenstrukturen ermittelt. Mithilfe dieser Repräsentationen ist eine Implementierung eines PMS' mit einer optimalen internen Darstellung der Prozessdaten möglich.

7.2 Ausblick

Dieser Abschnitt gibt abschließend einen Ausblick auf Themen, welche in zukünftigen Arbeiten untersucht werden können.

Es kann die Realisierung benutzerdefinierter Blöcke untersucht werden. Blöcke wurden im Rahmen dieser Arbeit anhand von Split-Knoten und deren zugehörigen Join-Knoten definiert. Auf dieser Idee basiert sowohl die Partitionierung von Schemata als auch die grobe Clusterung von Instanzen. Es ist denkbar, dass Blöcke manuell beim Modellieren des Schemas definiert werden. Setzt man dann auf Basis dieser Blöcke die grobe Clusterung um, so sind unter Umständen wesentlich bessere Ergebnisse von der groben Clusterung bei einer Schemaevolution zu erwarten, da der Modellierer die Zusammenhänge von Knoten besser feststellen kann. Bedacht werden muss hierbei allerdings, dass durch Einführung benutzerdefinierter Blöcke weitere Strukturen im Schema notwendig werden, die diese Blockstruktur abbilden. Eine implizite Darstellung der Blockstruktur ist unter Umständen nicht mehr möglich.

Ebenfalls untersucht werden kann, ob instanzbasiert geänderte Instanzen trotz des Widerspruchs in Abschnitt 3.4.4 im Clusterbaum gehalten werden können. Hierbei könnte eine Art „Clusterdelta“ eingesetzt werden, welches die strukturellen Änderungen der instanzbasiert geänderten Instanz abbildet und zusätzlich zur Instanz vom Clusterbaum referenziert wird. Hiermit ist es unter Umständen möglich, die Verträglichkeit dieser instanzbasiert geänderten Instanzen bei einer Schemaevolution auch ohne Einlagerung festzustellen.

A Abbildungsverzeichnis

1 Ein Schema mit zwei Instanzen . . . 3
2 Ein ADEPT2-Schema . . . 4
3 Die Blockstruktur eines ADEPT2-Schemas . . . 5
4 Knotenzustände und -übergänge (in Anlehnung an [Rei00]) . . . 6
5 Eine ADEPT2-Instanz während Knoten 10 ausgeführt wird . . . 7
6 Die Knoten eines Schemas und deren Knotenattribute . . . 8
7 Schemakopie vs. Schemareferenz . . . 9
8 Instanz mit impliziter und expliziter Markierung . . . 10
9 Clusterung . . . 11
10 Instanzbasierte Änderungen mit Deltaschicht . . . 11
11 Kapselung bei Einsatz einer strukturellen Deltaschicht . . . 12
12 Die ADEPT2-Architektur [RDR+09] . . . 13
13 Die verschiedenen Fälle des Weiterschaltens . . . 16
14 Beispiel einer Schemaevolution . . . 17
15 Exemplarische Organisation eines Unternehmens . . . 18
16 Speicherhierarchie . . . 19
17 Vertikale Partitionierung eines Schemas . . . 22
18 Horizontale Partitionierung eines Schemas . . . 22
19 Partitionierungsschwierigkeiten bei Bearbeiterzuordnungen . . . 24
20 Blockstruktur mit Hierarchie . . . 24
21 Knoten ohne Vorgänger-/Nachfolgerbeziehung . . . 27
22 Verschiedene Wege vom Start-Knoten zu Knoten 12 . . . 27
23 Schemarepräsentation mit expliziter Blockstruktur . . . 30
24 Hierarchiedarstellung bei expliziter Blockstruktur . . . 31
25 Hierarchie der Blockstruktur und Clusterbaum . . . 33
26 Beispiel für die grobe Clusterung . . . 35
27 Beispiel einer Schemaevolution bei grober Clusterung . . . 36
28 Zuordnung von Änderungen zu einer Stelle . . . 37
29 COMPACT und LEAKY . . . 38
30 Änderung im Wurzelblock . . . 39
31 Kapselung bei Einsatz der erweiterten Deltaschicht . . . 42
32 Deltaschicht mit expliziter Blockinformation . . . 45
33 Schemaevolution mit instanzbasierten Änderungen . . . 47
34 `Test`- und `Database`-Schnittstelle . . . 50
35 Architektur der Workflow-Testumgebung mit Schnittstellen . . . 51
36 Speicherverbrauch von `Maps` . . . 52
37 `Template`-Schnittstelle . . . 54
38 Die Klasse `DeltaLayer` . . . 56
39 `Instance`-Schnittstelle . . . 57
40 Die Klasse `Cluster` . . . 58
41 Speichertrennung anhand von Storage-Schnittstellen . . . 61
42 Schemarepräsentationen im Primär- und Sekundärspeicher . . . 62

43 TemplateStorage-Schnittstelle . . . 63
44 TemplateSerialization-Schnittstelle . . . 63
45 XML-Repräsentationen von Schemata . . . 64
46 Schemata mit expliziter Blockstruktur im RDBMS . . . 66
47 Schemata mit expliziter Blockstruktur in XML im RDBMS . . . 66
48 Schemata mit impliziter Blockstruktur im RDBMS . . . 67
49 Schemata mit expliziter Blockstruktur mit JPA . . . 67
50 Schemata mit impliziter Blockstruktur mit JPA . . . 68
51 InstanceStorage-Schnittstelle . . . 68
52 Instanzrepräsentation im Primär- und Sekundärspeicher . . . 69
53 InstanceSerialization-Schnittstelle . . . 69
54 XML-Repräsentationen von Instanzen . . . 70
55 Instanzen mit expliziten Knotenzuständen im RDBMS . . . 71
56 Instanzen mit expl. Knotenzuständen in XML im RDBMS . . . 71
57 Instanzen mit impliziten Knotenzuständen im RDBMS . . . 71
58 Instanzen mit expliziten Knotenzuständen mit JPA . . . 71
59 Instanzen mit impliziten Knotenzuständen mit JPA . . . 72
60 Caching Storage . . . 72
61 Das einfache Schema für die Messungen . . . 79
62 Das komplexe Schema für die Messungen . . . 79
63 Das „evolve“-Schema für die Messungen . . . 79
64 De-/Serialisieren eines Blocks . . . 81
65 De-/Serialisieren von Schemata . . . 82
66 De-/Serialisieren von Instanzen . . . 83
67 Messergebnisse für Schemata bei einer XML-Datei-Speicherung . . . 84
68 Messergebnisse für Instanzen bei einer XML-Datei-Speicherung . . . 85
69 Anfragen auf Kollektionen bei XML-Datei-Speicherung . . . 87
70 Messergebnisse von Schemata bei Einsatz von PostgreSQL . . . 88
71 Messergebnisse von Instanzen bei Einsatz von PostgreSQL . . . 90
72 Anfragen auf Kollektionen bei Einsatz von PostgreSQL . . . 91
73 Messergebnisse von Schemata bei Einsatz von Oracle . . . 92
74 Messergebnisse von Instanzen bei Einsatz von Oracle . . . 93
75 Anfragen auf Kollektionen bei Einsatz von Oracle . . . 94
76 Messergebnisse von Schemata bei Einsatz von DB2 . . . 96
77 Messergebnisse von Instanzen bei Einsatz von DB2 . . . 97
78 Anfragen auf Kollektionen bei Einsatz von DB2 . . . 98
79 Geschwindigkeitsfaktoren von Schemata bei Einsatz von JPA . . . 99
80 Geschwindigkeitsfaktoren von Instanzen bei Einsatz von JPA . . . 100
82 Performanzkoeffizienten c_{Schema} der einzelnen Varianten . . . 101
81 Vergleich Ein-/Auslagern von Schemata der versch. Varianten . . . 102
84 Performanzkoeffizienten $c_{Instanz}$ der einzelnen Varianten . . . 103
83 Vergleich Ein-/Auslagern von Instanzen der versch. Varianten . . . 104
85 Vergleich der Anfragen auf Kollektionen der versch. Varianten . . . 105
86 Performanzkoeffizienten $c_{Kollektion}$ der einzelnen Varianten . . . 106

87 Vereinigte Performanzkoeffizienten der einzelnen Varianten 107
88 Primärspeicherverbrauch von Schemata 108
89 Laufzeit beim Weiterschalten (in Tausendstel des Grundwertes) 109
90 Primärspeicherverbrauch von Instanzen 110
91 Weiterschalten mit & ohne Clusterung (in Tausendstel) 112
92 Die untersuchten Änderungen am evolve-Schema 113

B Tabellenverzeichnis

1 Explizite und implizite Realisierung von Blöcken 32
2 Laufzeit von `preEvolveSelection` (in Tausendstel) 113

C Literaturverzeichnis

[Adv09] Advanced Micro Devices, Inc.: *AMD Athlon™ XP Product Information*. Webseite, 2009. http://www.amd.com/us-en/Processors/ProductInformation/ 0,,30_118_3734_11672,00.html, abgerufen am 14. Dezember 2009.

[Bau00] Bauer, T.: *Effiziente Realisierung unternehmensweiter Workflow-Management-Systeme*. Dissertation, Universität Ulm, Oktober 2000.

[BMR96] Barbara, D., Mehrotra, S. und Rusinkiewicz, M.: *INCAs: Managing Dynamic Workflows in Distributed Environments*. Journal of Database Management, Special Issue on Miltidatabases, 7(1):5–15, 1996.

[bpm09] *Business Process Modeling Notation (BPMN)*. Technischer Bericht Version 1.2, OMG, Januar 2009.

[Can09] Canocical Ltd.: *Ubuntu*. Webseite, 2009. http://www.ubuntu.com, abgerufen am 14. Dezember 2009.

[Cle06] Clements, A.: *Principles of Computer Hardware*. Oxford University Press, 4 Auflage, März 2006.

[CLRS09] Cormen, T.H., Leiserson, C.E., Rivest, R.L. und Stein, C.: *Introduction to Algorithms*. MIT Press, 3. Auflage, September 2009.

[Cod70] Codd, E. F.: *A relational model of data for large shared data banks*. Commun. ACM, 13(6):377–387, 1970.

[DD90] Domschke, W. und Drexl, A.: *Einführung in Operations Research*. Springer, 6. Auflage, Oktober 1990.

[DHL90] Dayal, U., Hsu, M. und Ladin, R.: *Organizing Long-Running Activities with Triggers and Transactions*. In: *Proc. ACM SIGMOND Int'l Conf. on the Management of Data*, Seiten 204–214, 1990.

[DKM+97] Das, S., Kochut, K., Miller, J., Sheth, A. und Worah, D.: *ORB-Work: A Reliable Distributed CORBA-based Worflow Enactment System for METEOR$_2$*. Technischer Report #UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, 1997.

[DN67] Dahl, Ole-Johan und Nygaard, Kristen: *Class and Subclass Declarations*. In: Buxton, J. N. (Herausgeber): *Simulation Programming Languages*, Seiten 158–174, 1967.

[For09] Forschner, A.: *Fortschrittliche Datenflusskonzepte für flexible Prozessmodelle*. Diplomarbeit, Universität Ulm, Juni 2009.

[GJSB05] Gosling, J., Joy, B., Steele, G. und Bracha, G.: *Java™ Language Specification, The (3rd Edition)*. Addison Wesley, 3. Auflage, Juni 2005.

[HJKW96] Heimann, P., Joeris, G., Krapp, C.-H. und Westfechtel, B.: *DYNAMITE: Dynamic Task Nets for Software Process Management.* In: *18. Int'l Conf. Software Engineering (ICSE'96)*, Seiten 331–341, März 1996.

[IBM09a] IBM: *DB2.* Webseite, 2009. http://www.ibm.com/db2, abgerufen am 4. September 2009.

[IBM09b] IBM: *IBM DB2 Database for Linux, UNIX and Windows Information Center.* Webseite, 2009. http://publib.boulder.ibm.com/infocenter/db2luw/v9/, abgerufen am 8. September 2009.

[ISO92] ISO/IEC 9075:1992: *Database Language SQL.* International Organization for Standardization, 1992.

[JL96] Jones, R. und Lins, R. D.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, September 1996.

[JMSW04] Jurisch, M., Mihalca, T., Sauter, P. und Waimer, M.: *Verwaltung von Prozessinstanzen in Workflow Systemen.* Praktikum ADEPT2, Institut für Datenbanken und Informationssysteme, Universität Ulm, April 2004.

[Joe99] Joeris, G.: *Defining Flexible Workflow Execution Behaviors.* In: *Proc. Workshop on Enterprise-Wide and Cross-Enterprise Workflow-Management: Concepts, Systems, Applications, CEUR Workshop*, Band 24, Seiten 49–55, 1999.

[Jur06] Jurisch, M.: *Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und -instanzen.* Diplomarbeit, Universität Ulm, März 2006.

[Knu97] Knuth, D. E.: *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition).* Addison-Wesley Professional, 3. Auflage, Juli 1997.

[KPRR95] Kappel, G., Pröll, B., Rausch-Schott, S. und Retschitzegger, W.: *Workflow Manahement Based on Objects, Rules and Roles.* IEEE Data Engineering Bulletin, Special Issue on Worflow Systems, 18(1), März 1995.

[KR10] Kreher, U. und Reichert, M.: *Speichereffiziente Repräsentation instanzspezifischer Änderungen in Prozess-Management-Systemen.* Ulmer Informatik Berichte (erscheint Januar 2010), Universität Ulm, Ulm, 2010.

[Kre02] Kreher, U.: *Performanzaspekte bei der Programmierung mit Java™ - Eine eingehende Untersuchung von Threads, Garbage Collection und Objekt-Pools.* Diplomarbeit, Universität Ulm, Dezember 2002.

[KRRD09] Kreher, U., Reichert, M., Rinderle-Ma, S. und Dadam, P.: *Effiziente Repräsentation von Vorlagen- und Instanzdaten in Prozess-Management-*

Systemen. Ulmer Informatik Berichte UIB-2009-08, Universität Ulm, Ulm, 2009.

[KRW90] Karbe, B., Ramsperger, N. und Weiss, P.: *Support of Cooperative Work by Electronic Circulation Folders*. ACM SIGOIS Bulletin, 11(2+3):109–117, April 1990.

[LA94] Leymann, F. und Altenhuber, W.: *Managing Business Processes as an Information Resource*. IBM Systems Journal, 33(2):326–348, 1994.

[Ley01] Leymann, F.: *Web Services Flow Language (WSFL 1.0)*. IBM Technical White Paper, IBM, 2001.

[LRR04] Lauer, M., Rinderle, S. und Reichert, M.: *Repräsentation von Schema- und Instanzobjekten in adaptiven Prozess–Management–Systemen*. In: *Workshop Geschätfsprozessorientierte Architekturen (Informatik '04)*, Band LNI-P51, Seiten 555–560, 2004.

[MB02] Means, Scott und Bodie, Michael: *The Book of SAX: The Simple API for XML*. No Starch Press, 1 Auflage, Juni 2002.

[Mic09] Micheler, F.: *Konzeption, Implementierung und Integration einer Komponente für die Erstellung intelligenter Formulare*. Diplomarbeit, Universität Ulm, Juni 2009.

[MWW+98] Muth, P., Wodtke, D., Weissenfels, J., Weikum, G. und Kotz Dittrich, A.: *Enterprise-Wide Workflow Management Based on State and Activity Charts*. NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences, 164:281–303, 1998.

[Obj06] Object Management Group: *Unified Modeling Language: Infrastructure, Version 2.0*, März 2006.

[OOW93] O'Neil, Elizabeth J., O'Neil, Patrick E. und Weikum, Gerhard: *The LRU-K page replacement algorithm for database disk buffering*. SIGMOD Rec., 22(2):297–306, 1993.

[Ora09a] Oracle: *Database*. Webseite, 2009. http://www.oracle.com/database, abgerufen am 4. September 2009.

[Ora09b] Oracle: *Oracle Database Documentation Library - 10g Release 2*. Webseite, 2009. http://download.oracle.com/docs/cd/B19306_01/index.htm, abgerufen am 8. September 2009.

[Pos09a] PostgreSQL. Webseite, 2009. http://www.postgresql.org, abgerufen am 4. September 2009.

[Pos09b] PostgreSQL: *PostgreSQL 8.4.0 Documentation*. Webseite, 2009. http://www.postgresql.org/docs/8.4/static, abgerufen am 8. September 2009.

[RD98] Reichert, M. und Dadam, P.: *ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control*. Journal of Intelligent Information Systems, 10(2):93–129, 1998.

[RDR+09] Reichert, M., Dadam, P., Rinderle-Ma, S., Jurisch, M., Kreher, U. und Göser, K.: *Architectural Principles and Components of Adaptive Process Management Technology*. In: *PRIMIUM Process Inoovation for Enterprise Software*, Band P-151 der Reihe *Lecture Notes in Informatics - Proceedings*, Seiten 81–97. Gesellschaft für Informatik, April 2009.

[Red09] RedHat: *Hibernate*. Webseite, 2009. http://www.hibernate.org, abgerufen am 4. September 2009.

[Rei00] Reichert, M.: *Dynamische Ablaufänderungen in Workflow-Managment-Systemen*. Dissertation, Universität Ulm, Mai 2000.

[Rin04] Rinderle, S.: *Schema Evolution in Process Management Systems*. Dissertation, Universität Ulm, Oktober 2004.

[Rou02] Roubtsov, V.: *Java Tip 130: Do you know your data size?* Webseite, 2002. http://www.javaworld.com/javaworld/javatips/jw-javatip130.html, abgerufen am 6. September 2009.

[Sch01a] Schöning, U.: *Algorithmik*. Spektrum Akademischer Verlag, 1. Auflage, September 2001.

[Sch01b] Schöning, U.: *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag, 4. Aufl. Auflage, März 2001.

[Sun06] Sun Microsystems Inc.: *JSR 220: Enterprise JavaBeans, Version 3.0 - Java Persistence API*, Mai 2006.

[Sun09a] Sun Microsystems Inc.: *HotSpot Group*. Webseite, 2009. http://openjdk.java.net/groups/hotspot, abgerufen am 4. September 2009.

[Sun09b] Sun Microsystems Inc.: *Java Database Connectivity (JDBC)*. Webseite, 2009. http://java.sun.com/products/jdbc, abgerufen am 4. September 2009.

[Sun09c] Sun Microsystems Inc.: *Java HotSpot VM Options*. Webseite, 2009. http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp, abgerufen am 6. September 2009.

[Sun09d] Sun Microsystems Inc.: *JDK™ 6 Documentation - Reflection*, 2009. http://java.sun.com/javase/6/docs/technotes/guides/reflection, abgerufen am 8. September 2009.

[vv04] van der Aalst, W. und van Hee, K.: *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, März 2004.

[W3C99] W3C - World Wide Web Consortium: *XML Path Language (XPath) 1.0*. Webseite, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116, abgerufen am 9. November 2009.

[W3C06] W3C - World Wide Web Consortium: *Extensible Markup Language (XML) 1.1 (Second Edition)*. Webseite, September 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/, abgerufen am 4. September 2009.

[WWWK96] Wodtke, D., Weissenfels, J., Weikum, G. und Kotz-Dittrich, A.: *The Mentor Project: Steps Towards Enterprise-Wide Workflow-Management*. In: *Proc. of the 12. IEEE International Conference on Data Engineering*, Seiten 556–565, März 1996.

D Algorithmen

D.1 getNodeRelation

```
getNodeRelation(Node n1, Node n2) {

  if (n1.topologicalID == n2.topologicalID)
    return EQUAL;

  // Untersuchen der reduzierten Wege von n1 und n2, um den ersten
  // gemeinsamen Split-Knoten zu finden:
  // Wird merken uns den aktuellen Split-Knoten auf dem Weg von n1 und n2
  // in den jeweiligen Variablen

  Node splitNodeN1 = n1.splitNode;
  Node splitNodeN2 = n2.splitNode;

  // u.U. muessen spaeter die Zweige untersucht werden, somit merken wir uns
  // diese ebenfalls
  int lastBranchIDN1 = n1.branch;
  int lastBranchIDN2 = n2.branch;

  while (splitNodeN1.topologicalID != splitNodeN2.topologicalID) {
    // in jedem Durchlauf gehen wir einen Schritt "zurueck" auf dem
    // reduzierten Weg desjenigen Knotens, welcher topologisch nachgelagert
    // ist
    if (splitNodeN1.topologicalID > splitNodeN2.topologicalID) {
      lastBranchIDN1 = splitNodeN1.branch;
      splitNodeN1 = splitNodeN1.splitNode;
    } else {
      lastBranchIDN2 = splitNodeN2.branch;
      splitNodeN2 = splitNodeN2.splitNode;
    }
  }

  if (lastBranchIDN1 != lastBranchIDN2)
    // wir haben den ersten gemeinsamen Split-Knoten gefunden, kommen
    // allerdings von unterschiedlichen Zweigen
    return DIFFERENT_BRANCH;

  if (n1.topologicalID > n2.topologicalID)
    return SUCCESSOR;
  else
    return PREDECESSOR;
}
```

D.2 findPositions

```
/*
 * Stellentyp
 */
enum PositionType {
  // die Stelle ist nicht passierbar ohne einen geaenderten Knoten
  // ausgefuehrt zu haben
  COMPACT,
  // die Stelle kann auch ohne Ausfuehrung eines geaenderten Knotens
  // passiert werden
```

```
10    LEAKY
11  }
12
13  /*
14   * Sucht die Stellen der geaenderten Knoten
15   *
16   * Liefert eine Map zurueck, die der Stellen ID ein Paar aus Stellentyp und
17   * Knoten-IDs zuweist. Diese Knoten-IDs sind diejenigen von geaenderten
18   * Knoten, welche der entsprechenden Stelle zugeordnet sind.
19   */
20  Map<Integer, Pair<PositionType, Set<Integer>>> findPositions(DeltaLayer delta) {
21
22    // Initialisierung des Rueckgabeobjekts
23    Map<Integer, Pair<PositionType, Set<Integer>>> positions =
24      new HashMap<Integer, Pair<PositionType, Set<Integer>>>();
25
26    // Eine Menge an Knoten, welche geaendert wurden in der gegebenen
27    // Deltaschicht.
28    // Diese Knoten sind die Ziele von hinzugefuegten und geloeschten Kanten
29    // [Jur06]
30    Set<Integer> changedNodes = new HashSet<Integer>();
31    changedNodes.addAll(delta.getAddedEdgesBackwards().keySet());
32    changedNodes.addAll(delta.getDeletedEdgesBackwards().keySet());
33
34    // diejenigen Knoten, die im urspruenglichen Schema nicht vorhanden sind,
35    // koennen nicht "geaendert" sein und muessen somit aus der Menge entfernt
36    // werden
37    for (int nID : new HashSet<Integer>(changedNodes)) {
38      if (delta.getAddedNodes().contains(nID))
39        changedNodes.remove(nID);
40
41      Node n = template.getNode(nID);
42      // sicher gehen, dass dies keine Aenderung im Wurzelblock ist
43      if (n.getType() == Type.NORMAL &&
44        n.getSplitNodeID() == Template.START_NODE_ID)
45        continue;
46    }
47
48    // Die Stellen der einzelnen geaenderten Knoten werden nun gesucht
49    // positionsOfChange ist das Ergebnis dieser Berechnung:
50    // Einer Stellen-ID sind Knoten-IDs von geaenderten Knoten zugeordnet
51    Map<Integer, Set<Integer>> positionsOfChanges =
52      new HashMap<Integer, Set<Integer>>();
53
54
55    for (int changedNodeID : changedNodes) {
56      // wir suchen die Stellen ID von changedNodeID
57
58      // splitNodeID ist die ID des zuletzt besuchten Split-Knotens
59      // der Wert wird auf den Split-Knoten, der dem geaenderten Knoten
60      // direkt uebergeordnet ist, initialisiert
61      int splitNodeID = template.getNode(changedNodeID).getSplitNodeID();
62
63      // das Ergebnis unserer Suche: der zuletzt besuchte Split-Knoten
64      int topLevelSplitNode;
65
66      // hole das Objekt des geaenderten Knotens vom Schema
67      Node changedNode = template.getNode(changedNodeID);
68
69      // Ueberpruefung, ob der geaenderte Knoten selbst ein Split- Knoten ist,
70      // welcher einen Block aufspannt, der dem Wurzelblock
71      // direkt untergeordnet ist. In diesem Fall ist die Block-ID des
```

```
72      // Knotens (und somit die Stellen-ID) nicht ueber
73      // changedNode.getSplitNodeID() verfuegbar (dieser Wert ist gleich
74      // Template.START_NODE_ID), sondern die Block-ID ist aequivalent
75      // zur ID des Split-Knotens selbst
76      if (changedNode.getSplitNodeID() == Template.START_NODE_ID &&
77          changedNode instanceof SplitNode) {
78        splitNodeID = changedNodeID;
79        topLevelSplitNode = changedNodeID;
80
81      } else if (changedNode.getSplitNodeID() == Template.START_NODE_ID &&
82          changedNode instanceof JoinNode) {
83        // fuer einen Join-Knoten, der dem Wurzelblock direkt unter-
84        // geordnet ist, ist die Block-ID wie folgt verfuegbar:
85        splitNodeID =
86          ((JoinNode)changedNode).getCorrespondingSplitNodeID();
87        topLevelSplitNode =
88          ((JoinNode)changedNode).getCorrespondingSplitNodeID();
89
90      } else {
91        // Standardfall:
92        // merke in splitNodeID jeweils die ID des naechst "hoeher"
93        // gelegenen Split-Knotens
94        while (splitNodeID != Template.START_NODE_ID) {
95          topLevelSplitNode = splitNodeID;
96          splitNodeID = template.getNode(splitNodeID).getSplitNodeID();
97        }
98      }
99
100     // in topLevelSplitNode steht nun die ID desjenigen Split-Knoten,
101     // welcher einen Block aufspannt, der dem Wurzelblock direkt
102     // untergeordnet ist. Somit ist dies unsere Stellen ID
103
104     // Die Map muss u.U. initialisiert werden
105     if (!positionsOfChanges.containsKey(topLevelSplitNode))
106       positionsOfChanges.put(topLevelSplitNode, new HashSet<Integer>());
107
108     positionsOfChanges.get(topLevelSplitNode).add(changedNodeID);
109   }
110
111
112
113   // Die Stellentypen der einzelnen Stellen muessen bestimmt werden
114
115   // fuer alle gefundenen Stellen...
116   for (int positionID : positionsOfChanges.keySet()) {
117
118     // wir suchen fuer jeden geaenderten Knoten der Stelle den reduzierten
119     // Weg vom Split-Knoten, der die Stelle aufspannt
120     // splitBranches haelt alle diese Wege
121     // Inhalt in splitBranches sind Tripel: {s, b, Set<n>}
122     // s = Split-Knoten, b = Zweig-ID, n = naechster Knoten
123     // Das bedeutet: Wenn man bei s ist, kann man durch Verfolgen des
124     // Zweigs b zu den Knoten n kommen. Diese sind wieder Split-Knoten
125     // fuer welche man splitBranches untersuchen kann oder ein Knoten,
126     // der geaendert wurde. Fuer diese geaenderte Knoten g existiert ebenso
127     // ein Eintrag in splitBranches: {g, -1, null}. Dies wird fuer die
128     // Abbruchbedingung von isFull benoetigt.
129     Map<Node, Map<Integer, Set<Node>>> splitBranches =
130       new HashMap<Node, Map<Integer, Set<Node>>>();
131
132     // gehe alle geaenderten Knoten der Stelle durch... Die reduzierten
133     // Wege werden rueckwaerts (vom geaenderten Knoten aus) zu
```

```
134      // splitBranches hinzugefuegt
135      for (int nID : positionsOfChanges.get(positionID)) {
136
137        // n = Knotenobjekt des geaenderten Knoten
138        Node n = template.getNode(nID);
139
140        // Fuege Tripel {g, -1, null} hinzu (siehe oben)
141        if (!splitBranches.containsKey(nID))
142          splitBranches.put(n, new HashMap<Integer, Set<Node>>());
143        splitBranches.get(n).put(-1, null);
144
145        // wir Verfolgen die uebergeordneten Split-Knoten bis zu
146        // demjenigen der die Stelle aufspannt (positionID ist ja
147        // gleichzeitig auch die ID des die Stelle aufspannenden Split-
148        // Knoten!)
149        while (n.getID() != positionID) {
150          // der uebergeordnete Split-Knoten
151          Node splitNode = template.getNode(n.getSplitNodeID());
152          // der Zweig, auf dem sich n befindet. Dieser Zweig beginnt
153          // bei splitNode
154          int branchID = n.getBranchID();
155
156          // u.U. muss splitBranches initialisiert werden
157          if (!splitBranches.containsKey(splitNode))
158            splitBranches.put(splitNode,
159                new HashMap<Integer, Set<Node>>());
160          if (!splitBranches.get(splitNode).containsKey(branchID))
161            splitBranches.get(splitNode).put(branchID,
162                new HashSet<Node>());
163
164          // merke diesen Schritt des reduzierten Weges
165          splitBranches.get(splitNode).get(branchID).add(n);
166
167          // fahre in der naechsten Iteration mit dem Split-Knoten fort
168          n = splitNode;
169        }
170      }
171
172      // Knoten-Objekt des Split-Knotens, der die Stelle aufspannt
173      // (positionID ist gleichzeitig die ID des aufspannenden
174      // Split-Knotens!)
175      Node splitNode = template.getNode(positionID);
176
177      // berechne durch Aufruf von isFull, welchen Typ die Stelle hat
178      PositionType positionType =
179        isFull(splitBranches, splitNode)?
180            PositionType.COMPACT:
181            PositionType.LEAKY;
182
183      // merke Stelle in der Ergebnismenge
184      positions.put(positionID,
185          new Pair<PositionType, Set<Integer>>(
186              positionType, positionsOfChanges.get(positionID)));
187    }
188
189    return positions;
190  }
191
192  /*
193   * Berechnet fuer die gegebene Menge an reduzierten Wegen (splitBranches,
194   * siehe findPositions), ob der von splitNode aufgespannte Block von einer
195   * Instanz nicht durchlaufen werden kann, ohne einen geaenderten Knoten aus-
```

```
196    * gefuehrt zu haben.
197    */
198  boolean isFull(Map<Node, Map<Integer, Set<Node>>> splitBranches, Node splitNode) {
199    // Abbruchbedingung. Ist splitNode ein geaenderter Knoten, existiert in
200    // splitBranches ein Eintrag mit der Zweig-ID -1 (dies ist keine gueltige
201    // Zweig-ID und wird somit normalerweise nicht benutzt). Somit sind wir
202    // bei einem geaenderten Knoten auf dem Weg angekommen. Dieser kann nicht
203    // von einer Instanz passiert werden ohne ausgefuehrt zu werden.
204    if (!splitBranches.get(splitNode).containsKey(-1))
205      return true;
206
207    if (splitNode.getType() == Type.AND_SPLIT) {
208
209      // Trivialfall: keine Aenderungen auf den Zweigen des Split-Knotens
210      if (splitBranches.get(splitNode).isEmpty())
211        return true;
212
213      // durchsuche alle Zweige des Split-Knotens, auf denen Aenderungen
214      // sind
215      for (int branchID : splitBranches.get(splitNode).keySet()) {
216
217        // Untersuche alle Aenderungen auf dem Zweig branchID von
218        // splitNode. Um "true" zurueck liefern zu koennen genuegt es, wenn
219        // eine dieser Aenderungen unpassierbar fuer eine Instanz ist.
220        for (Node nextNode : splitBranches.get(splitNode).get(branchID))
221          if (isFull(splitBranches, nextNode))
222            return true;
223      }
224      return false;
225    } else {
226      // splitNode ist ein XOR-Split!
227
228      // bei einem XOR-Split muessen alle Zweige Aenderungen besitzen, fuer
229      // die isFull(.) = true gilt!
230
231      // Trivialfall: Nicht alle Zweige haben Aenderungen
232      if (splitBranches.get(splitNode).size() !=
233          template.getNodeSuccessors(splitNode.getID()).size())
234        return false;
235
236      // untersuche alle Zweige...
237      for (int branchID : splitBranches.get(splitNode).keySet()) {
238
239        boolean branchIsFull = false;
240        // untersuche alle Aenderungen auf dem aktuellen Zweig branchID
241        for (Node nextNode : splitBranches.get(splitNode).get(branchID)) {
242          branchIsFull = isFull(splitBranches, nextNode);
243
244          // wenn diese Aenderung unpassierbar ist, muessen die
245          // restlichen Aenderungen des Zweiges nicht untersucht werden
246          if (branchIsFull)
247            break;
248        }
249        if (!branchIsFull)
250          // es wurde keine unpassierbare Aenderung auf dem Zweig
251          // gefunden
252          return false;
253      }
254
255      return true;
256    }
257  }
```

D.3 preEvolveSelection

```
/*
 * Die Kategorien, in die preEvoleSelection die Instanzen einteilt
 */
enum EVOLVE_CATEGORY {
  MIGRATABLE,
  NOT_MIGRATABLE,
  PRECISE_CHECK
};

// Weist einer Cluster-ID eine Menge an Instanz-IDs zu, welche sich in
// diesem Cluster befinden
Map<Integer, Set<Integer>> instances;

// weist einer Cluster-ID die IDs der direkt untergeordneten Cluster zu
Map<Integer, Set<Integer>> subClusters;

// weist einer Cluster-ID die ID des direkt uebergeordneten Cluster zu
Map<Integer, Integer> topCluster;



/*
 * Kategorisiert die Instanzen anhand der Aenderungen in delta ein.
 */
Map<EVOLVE_CATEGORY, Set<Integer>> preEvolveSelection(DeltaLayer delta) {
  // IDs von Instanzen, die nicht migriert werden koennen
  Set<Integer> notMigratable = new HashSet<Integer>();
  // IDs von Instantzen, die migriert werden
  Set<Integer> migratable = new HashSet<Integer>();
  // IDs von Instanzen, die genauer untersucht werden muessen
  Set<Integer> preciseCheck = new HashSet<Integer>();

  // Zuerst werden alle Aenderungen im Wurzelblock untersucht

  // Eine Menge an Knoten, welche geaendert wurden in der gegebenen
  // Deltaschicht.
  // Diese Knoten sind die Ziele von hinzugefuegten und geloeschten Kanten
  // [Jur06]
  Set<Integer> changedNodesInRootCluster = new HashSet<Integer>();
  changedNodesInRootCluster.addAll(delta.getAddedEdgesBackwards().keySet());
  changedNodesInRootCluster.addAll(delta.getDeletedEdgesBackwards().keySet());

  // diejenigen Knoten, die sich nicht im Wurzelblock befinden, muessen
  // wieder aus der Menge entfernt werden
  for (int nID : new HashSet<Integer>(changedNodesInRootCluster)) {
    Node n = template.getNode(nID);
    if (n.getType() == Type.NORMAL &&
      n.getSplitNodeID() == Template.START_NODE_ID)
      // Dieser Knoten befindet sich im Wurzelblock
      continue;

    changedNodesInRootCluster.remove(nID);
  }


  for (int changedNodeID : changedNodesInRootCluster) {
    // untersuche alle direkt untergeordneten Cluster des Wurzelclusters
    for (int subClusterID : subClusters.get(Template.START_NODE_ID)) {
      // subClusterID ist auch ID des den Cluster aufspannenden Split-
      // Knotens!
```

```
61        if (template.getNodeRelation(subClusterID, changedNodeID) ==
62          NodeRelation.SUCCESSOR)
63          // der Untercluster ist der Aenderung nachgelagert
64          // somit sind alle Instanzen nicht vertraeglich!
65          addInstancesToSet(subClusterID, notMigratable);
66      }
67    }
68
69    // alle Instanzen aus dem Wurzelcluster muessen genauer untersucht werden
70    addInstancesToSet(Template.START_NODE_ID, preciseCheck);
71
72    // instances bereinigen
73    for (int cID : instances.keySet()) {
74      instances.get(cID).removeAll(notMigratable);
75      instances.get(cID).removeAll(preciseCheck);
76    }
77
78
79    // Finde Stellen.
80    // Mapping von Stellen-ID zu Stellentyp und Menge an Knoten IDs von
81    // geaenderten Knoten dieser Stelle
82    Map<Integer, Pair<PositionType, Set<Integer>>> positions =
83      findPositions(delta);
84
85    // finde topologisch erste LEAKY und erste COMPACT Stelle
86    int smallestLEAKYtopID = Integer.MAX_VALUE;
87    int smallestLEAKYID = -1;
88    int smallestCOMPACTtopID = Integer.MAX_VALUE;
89    int smallestCOMPACTID = -1;
90    // gehe alle Stellen durch (Stellen-ID = ID des die Stelle aufspannenden
91    // Split-Knotens!)
92    for (int splitID : positions.keySet()) {
93
94      int topID = template.getTopologicalIDOfNode(splitID);
95
96      if (positions.get(splitID).getKey() == PositionType.COMPACT) {
97        if (topID < smallestCOMPACTtopID) {
98          smallestCOMPACTtopID = topID;
99          smallestCOMPACTID = splitID;
100       }
101     } else {
102       if (topID < smallestLEAKYtopID) {
103         smallestLEAKYtopID = topID;
104         smallestLEAKYID = splitID;
105       }
106     }
107   }
108
109   // topologisch erste Stelle
110   int smallestChangePositionID =
111     (smallestLEAKYtopID < smallestCOMPACTtopID)?
112         smallestLEAKYID:
113         smallestCOMPACTID;
114
115   int smallestChangePositionTopID =
116     (smallestLEAKYtopID < smallestCOMPACTtopID)?
117         smallestLEAKYtopID:
118           smallestCOMPACTtopID;
119
120
121   // untersuche alle geaenderte Knoten der ersten Stelle
122   for (int nID : positions.get(smallestChangePositionID).getValue()) {
```

```
123      Node n = template.getNode(nID);
124      int cID; // ID des Blocks/Clusters von n
125      if (n.getType() == Type.NORMAL)
126        cID = n.getSplitNodeID();
127      else if (n instanceof SplitNode)
128        cID = n.getID();
129      else if (n instanceof JoinNode)
130        cID = ((JoinNode)n).getCorrespondingSplitNodeID();
131
132      // alle Instanzen aus dem Cluster des geaenderten Blocks genauer
133      // untersuchen!
134      addInstancesToSet(cID, preciseCheck);
135
136      // ID des Split Knotens, der Cluster aufspannt = cID!
137      Node splitNode = template.getNode(cID);
138
139      // untersuche direkt untergeordnete Cluster
140      for (int subClusterID : subClusters.get(cID)) {
141        // subClusterID = ID des den untergeordneten Cluster
142        // aufspannenden Split-Knoten!
143        NodeRelation relation =
144          template.getNodeRelation(subClusterID, nID);
145        if (relation == NodeRelation.SUCCESSOR)
146          // untergeordneter Cluster ist Aenderung nachgelagert
147          addInstancesToSet(subClusterID, notMigratable);
148
149        if (relation == NodeRelation.DIFFERENT_BRANCH &&
150            splitNode.getType() == Type.AND_SPLIT)
151          // untergeordneter Cluster liegt auf anderem Zweig und
152          // Split-Knoten ist ein AND-Split -> preciseCheck
153          addInstancesToSet(subClusterID, preciseCheck);
154      }
155
156
157      // untersuche alle uebergeordneten Cluster
158      // Dies wird mithilfe einer Breitensuche [CLRS09] geloest
159      Set<Integer> clustersChecked = new HashSet<Integer>();
160      clustersChecked.add(cID);
161      Queue<Integer> clustersToCheck = new LinkedList<Integer>();
162      clustersToCheck.add(topCluster.get(cID));
163
164      while (!clustersToCheck.isEmpty()) {
165        int checkID = clustersToCheck.poll();
166
167        if (clustersChecked.contains(checkID))
168          continue;
169        clustersChecked.add(checkID);
170
171        // fuege unter- und uebergeordnete Cluster als "zu pruefen" hinzu
172        clustersToCheck.addAll(subClusters.get(checkID));
173        clustersToCheck.add(topCluster.get(checkID));
174
175        if (template.getNodeRelation(checkID, nID) !=
176          NodeRelation.PREDECESSOR)
177          // Cluster ist nicht Vorgaenger der Aenderung -> preciseCheck
178          preciseCheck.addAll(instances.get(checkID));
179      }
180    }
181
182
183    // Bereinigen von preciseCheck
184    preciseCheck.removeAll(notMigratable);
```

```
185
186   // instances bereinigen
187   for (int cID : instances.keySet()) {
188     instances.get(cID).removeAll(notMigratable);
189     instances.get(cID).removeAll(preciseCheck);
190   }
191
192
193   // Instanzen, welche sich immer noch in Clustern der ersten Stelle
194   // befinden sind der Aenderung vorgelagert und somit migrierbar!
195   // Wieder per Breitensuche
196   Queue<Integer> subClustersToCheck = new LinkedList<Integer>();
197   subClustersToCheck.add(smallestChangePositionID);
198
199   while (!subClustersToCheck.isEmpty()) {
200     int subClusterID = subClustersToCheck.poll();
201
202     subClustersToCheck.addAll(subClusters.get(subClusterID));
203
204     migratable.addAll(instances.get(subClusterID));
205
206     // instances bereinigen
207     instances.get(subClusterID).clear();
208   }
209
210
211   if (smallestChangePositionID == smallestLEAKYID &&
212       smallestCOMPACTID != -1) {
213     // die erste Stelle ist LEAKY und es folgt eine COMPACT Stelle
214     // es muessen alle Instanzen der COMPACT Stelle als genauer zu
215     // untersuchen markiert werden
216     // wieder per Breitensuche
217     subClustersToCheck = new LinkedList<Integer>();
218     subClustersToCheck.add(smallestCOMPACTID);
219
220     while (!subClustersToCheck.isEmpty()) {
221       int subClusterID = subClustersToCheck.poll();
222
223       subClustersToCheck.addAll(subClusters.get(subClusterID));
224
225       preciseCheck.addAll(instances.get(subClusterID));
226
227       // instances bereinigen
228       instances.get(subClusterID).clear();
229     }
230   }
231
232
233
234   // resltiche Instanzen einteilen
235   for (int cID : instances.keySet()) {
236
237     if (instances.get(cID).isEmpty())
238       // wenn keine Instanzen im Cluster sind kann dieser uebersprungen
239       // werden
240       continue;
241
242     // cID = ID des Clusters = ID des den Cluster aufspannenden
243     // Split-Knotens
244
245     if (template.getNodeRelation(cID, smallestChangePositionID) ==
246       NodeRelation.PREDECESSOR)
```

```
247        // Cluster ist erster Aenderung vorgelagert
248        migratable.addAll(instances.get(cID));
249      else if (smallestCOMPACTID != -1 &&
250          template.getNodeRelation(cID, smallestCOMPACTID) ==
251            NodeRelation.SUCCESSOR)
252        // Es existiert eine COMPACT Stelle und Cluster ist dieser
253        // nachgelagert -> nicht migrierbar!
254        notMigratable.addAll(instances.get(cID));
255      else {
256        // der Cluster ist nach der ersten Stelle (welche LEAKY ist)
257        // aber vor der ersten COMPACT Stelle, falls eine solche
258        // existiert -> genauer untersuchen!
259        preciseCheck.addAll(instances.get(cID));
260      }
261      // instances bereinigen
262      instances.get(cID).clear();
263    }
264
265    // Ergebnis aufbereiten
266    Map<EVOLVE_CATEGORY, Set<Integer>> res =
267      new HashMap<EVOLVE_CATEGORY, Set<Integer>>();
268    res.put(EVOLVE_CATEGORY.MIGRATABLE,
269        new HashSet<Integer>(migratable));
270    res.put(EVOLVE_CATEGORY.NOT_MIGRATABLE,
271        new HashSet<Integer>(notMigratable));
272    res.put(EVOLVE_CATEGORY.PRECISE_CHECK,
273        new HashSet<Integer>(preciseCheck));
274
275    return res;
276  }
277
278  /*
279   * fuegt alle Instanzen des Clusters oder einer seiner direkt oder indirekt
280   * untergeordneten Cluster zum angegebenen Set hinzu
281   */
282  void addInstancesToSet(int clusterID, Set<Integer> set) {
283    set.addAll(instances.get(clusterID));
284    for (int subClusterID : subClusters.get(clusterID))
285      addInstancesToSet(subClusterID, set);
286  }
```

Danksagung

Zuerst möchte ich mich bei meiner Familie ganz herzlich bedanken. Ihr habt mich immer unterstützt und es mir durch eure Kraft und euren Zuspruch ermöglicht, diesen, meinen Weg zu finden und zu gehen. Wann immer ich ein Problem habe, kann ich mich vertrauensvoll an euch wenden. Danke!

Besonders bedanken möchte ich mich auch bei meinem Betreuer Ulrich Kreher. Du hast mir beim Erstellen und der Korrektur dieser Arbeit sehr geholfen. Deine hohen Ansprüche haben mich angespornt und deine konstruktive Kritik hat mich immer *noch* ein Stückchen weiter gebracht. Danke!

Ebenfalls großer Dank geht an meine beiden Korrekturleser Florian Micheler und Wolfgang Holoch. Vielen Dank für eure Hilfe, meine ungefilterten Worte zu verstehen und mir zu helfen, diese in eine lesbare Form zu bringen. Danke!

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, 23. Dezember 2009

Bastian Philipp Glöckle, Matrikel-Nr. 541471