Relevanz von Änderungen für Datenbestände mobiler Clients

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Diplom-Informatiker Hagen Höpfner

geb. am 25. Januar 1977 in Aschersleben

Gutachter:

Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Prof. Dr. Klaus Küspert

Magdeburg, den 28. Januar 2005

Hagen Höpfner
Relevanz von Änderungen für Datenbestände mobiler Clients
Dissertation, Otto-von-Guericke Universität
Magdeburg, 2005

Zusammenfassung

Aufgrund der Entwicklungen auf dem Sektor der Mobiletelefone ist davon auszugehen, dass die Anzahl kleiner, leichtgewichtiger mobiler Endgeräte, welche als Clients von Datenbanksystemen genutzt werden, weiter ansteigt. Ein wesentliches Problem in diesem Szenario ist die langsame und teure Datenübertragung in Funknetzwerken. Aus Sicht der Nutzer derartiger Technologien ist es somit nicht sinnvoll bzw. wünschenswert, Daten, welche keinen direkten Nutzen für sie bieten, auf das Mobilgerät zu übertragen. Des Weiteren werden einmal empfangene Daten lokal gespeichert und wiederverwendet (Caching). Ändern sich Daten auf dem Server des Informationssystems, müssen die mobilen Clients, welche von einer Änderung betroffen sind, ermittelt und darüber informiert werden.

Im Rahmen dieser Dissertation wird gezeigt, dass eine derartige exakte Relevanzprüfung nur mithilfe der Daten in der Datenbank realisiert werden kann. Andere Verfahren, welche das Problem auf semantischer Ebene adressieren, funktionieren nur, wenn die unterstützte Anfragesprache stark eingeschränkt wird und können selbst dann zur falschen Feststellung einer Relevanz führen. Insbesonderen in Informationssystemen mit einer großen Anzahl mobiler Clients kommt es dazu, dass verschiedene Nutzer gleiche oder zumindest ähnliche Anfragen stellen. Diese Eigenschaft wird hier genutzt, um – basierend auf einer speziellen Anfragenotation – syntaktisch gleiche Teile mehrerer Anfragen gemeinsam auf Relevanz zu testen. Hierzu werden die Anfragen in einem Anfragebaum gespeichert, welcher die IDs der Clients referenziert. Die Relevanzprüfung erfolgt durch Traversieren des Baumes.

In der vorliegenden Arbeit werden verschiedene Anfrageindexstrukturen diskutiert und die Menge der notwendigen Relevanztest formal auf der Basis der Relationenmodells hergeleitet. Abschließend werden die theoretisch erarbeiteten Techniken evaluiert und Vorschläge für ihre Optimierung diskutiert.

Es gibt eine Theorie, die besagt, wenn jemals irgendwer genau rausfindet, wozu das Universum da ist und warum es da ist, dann verschwindet es auf der Stelle und wird durch etwas noch Bizarreres und Unglaublicheres ersetzt.

Douglas Adams in [Ada82]

Vorwort

Mit der Geschichte der Menschheit ist stets das Streben nach mehr Wissen verbunden gewesen. Dies spiegelt sich auch in der Handhabung und der Verbreitung von Informationen wider. Durch die Erfindung des Papyrus ca. 2500 v. Chr. bestand erstmals die Möglichkeit, komplexes Wissen für einen längeren Zeitraum zu speichern und dieses gespeicherte Wissen auch zu transportieren. Sicherlich waren bereits die aus Mesopotamien bekannten Steinplatten in gewisser Weise transportabel, jedoch bedurfte dies eines erheblichen Mehraufwandes. Im Laufe der Zeit wurden weitere bahnbrechende Erfindungen gemacht, die dem Weitergeben und Speichern von Informationen dienlich waren. Beispielsweise ermöglichte das Buchdruckverfahren, welches durch Johannes Gutenberg zwischen 1448 und 1455 perfektioniert wurde, erstmals das Erzeugen einer Vielzahl inhaltlich gleicher Dokumente.

Mit dem Übergang des Industriezeitalters zum Informationszeitalter im Laufe des 20. Jahrhunderts wurde die Verfügbarkeit von Informationen zu einer wesentlichen Entwicklungsgrundlage. Parallel dazu führte der stetige Technologiefortschritt dazu, dass immer mehr Informationen maschinell erzeugt, verarbeitet, gespeichert und repräsentiert wurden. Aus traditionellen Forschungsgebieten wie der Mathematik und der Physik entstanden neue Forschungsgebiete wie die Informatik, die Elektrotechnik oder die Nachrichtentechnik, die sich mit den zentralen Fragestellungen dieser neuen und nunmehr digitalen Wissensrepräsentation und ihrer Handhabbarkeit beschäftigen. Die Ergebnisse dieser Bemühungen lassen sich wie folgt beschreiben: Die Hardware nahm und nimmt immer weniger Platz in Anspruch wobei die Leistungsfähigkeit steigt. Nach anfänglicher Zurückhaltung anderer Forschungs- und Industriegebiete wie dem Maschinenbau, der Biologie oder den Wirtschaftswissenschaften, fasste die Informationstechnologie recht schnell auch in diesen Bereichen Fuß und ist heute aus keinem Gebiet mehr wegzudenken. Durch den Einsatz von Computertechnik wuchsen natürlich auch die zu verarbeitenden Datenmengen und die Anforderungen an die eingesetzte Software. Neben zahlreichen anderen bahnbrechenden Entwicklungen im Bereich der Software war die Einführung und Standardisierung von Datenbankmanagementsystemen einer der Meilensteine bei der effizienten Handhabung von komplexen und großen Datenbeständen. Monolithische Systeme zur Lösung „einfacher“ Aufgaben wurden durch zunehmend leistungsfähiger werdende Netzwerkkomponenten ergänzt, wodurch ihre Ressourcen kombiniert wurden. Gleichzeitig wurden neue Kommunikationswege und -formen geschaffen, die zu einer schnellen Nutzerakzeptanz von Computern und Computernetzen führten. Insbesondere die Experimentierfreudigkeit der jungen Generation(en) brachte die Computertechnik ins Wohnzimmer und prägte ein neues Selbstverständnis im Umgang mit derartigen Technologien.

Der schlussendliche Schritt zur Einführung von tragbarer, autonom arbeitender und gleich-

zeitig vernetzbarer Hardware bildet die Grundlage für diese Dissertation. Die weite Verbreitung von mobilen Endgeräten wie Mobiltelefone, Laptops, PDAs oder Tablet PCs in Verbindung mit der Zunahme deren Leistungsfähigkeit ermöglicht theoretisch den Zugriff auf digitale Daten, die auf festnetzgebundenen Servern vorliegen, jederzeit an jedem Ort. In der Realität zeigt sich jedoch, dass unter Anderem die Portabilität mobiler Endgeräte und die Eigenarten von Funknetzwerken neue Fragestellungen aufwerfen. So kann beispielsweise keine permanente Netzverfügbarkeit garantiert werden. Darüberhinaus ist zu beobachten, dass flächendeckende Mobilfunknetze (GSM, GPRS) verhältnismäßig langsam und teuer sind, wohingegen kostengünstigere bzw. breitbandigere Netze (WLAN, UMTS) nicht überall verfügbar sind. Um daher die zu übertragende Datenmenge möglichst gering zu halten, werden Verfahren eingesetzt, welche einmal übertragene Daten auf den mobilen Endgeräten zwischenspeichern. Diese gewollte Redundanz führt bei Änderungen auf dem Server aber automatisch zu Inkonsistenzen, wenn die mobilen Clients nicht über diese Änderungen informiert werden, um entsprechend reagieren zu können. Die Tatsache, dass sich heutzutage selbst Mobiltelefone als Clients benutzen lassen, illustriert ein wesentliches Problem, das bei dieser Änderungsnotifikation auftritt:

- Wie kann serverseitig effizient festgestellt werden, welche Änderung für welchen Client relevant ist?

Die vorliegende Dissertation greift diese Fragestellung für relationale Datenbanksystemen mit mobilen Clients auf. Sie entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter in einem durch die Deutsche Forschungsgemeinschaft (DFG) unter den Förderkennziffern SA 732/3-1 und SA 732/3-2 unterstütztes Projekt am Institut für Technische und Betriebliche Informationssysteme der Fakultät für Informatik an der Otto-von-Guericke Universität Magdeburg.

Danksagung

An dieser Stelle sei all denen gedankt, die mich beim Erstellen dieser Dissertation gefördert, gedrängt oder einfach nur ertragen haben. Insbesondere meiner Frau Romy und meinen Kindern Jenny und Hannes sei versprochen, dass ich jetzt wieder mehr Zeit mit ihnen verbringen werde. Ein besonderes Dankeschön geht natürlich an Prof. Gunter Saake und Prof. Kai-Uwe Sattler für die wissenschaftlich Begleitung, konstruktive Kritik und zahlreiche Diskussionen. Darüberhinaus gilt mein Dank meinen Kollegen in der Arbeitsgruppe Datenbanken von denen ich Dr. Eike Schallehn und Herrn Ingolf Geist hervorheben möchte, die geduldig meinen Vorführungen „bahnbrechender“ Ideen gefolgt sind. Abschließend bedanke ich mich bei meinen Eltern dafür, dass sie mir das Studium der Informatik ermöglicht und mir immer mit Rat und Tat zur Seite gestanden haben.

Magdeburg, Oktober 2004 Hagen Höpfner

Inhaltsverzeichnis

Abbildungsverzeichnis **12**

Tabellenverzeichnis **13**

Verzeichnis der Abkürzungen **15**

1 Einleitung und Motivation **17**
1.1 Serverseitige vs. clientseitige Relevanzprüfung 19
1.2 Wahl des Datenmodells 20
1.3 Nomenklatur 21
1.4 Gliederung der Arbeit 22

2 Einordnung der Thematik **23**
2.1 Allgemeine Einordnung 23
2.2 Query Containment 28
2.2.1 Algorithmen zum Testen des Query Containments 28
2.2.2 Zusammenhang zwischen Query Containment und der Relevanzprüfung 32
2.3 Incremental View Update 34
2.3.1 Abgrenzung 37
2.4 Publish-Subscribe-Systeme 37

3 Anfragerepräsentation **39**
3.1 Formale Definition des Relationenmodells 39
3.2 Repräsentation von Anfragen 40
3.2.1 Vollständigkeit und Sicherheit 45
3.2.2 Umsetzung von PSQ nach SQL 45

4 Relevanzprüfung von Änderungsoperationen **47**
4.1 Notation von Änderungsoperationen 48
4.2 Prüfen der Relevanz von Einfügeoperationen 49

4.2.1 Relevanz von Einfügeoperationen für Verbundprädikate 50
4.2.2 Relevanz von Einfügeoperationen für Selektionsprädikate 51
4.2.3 Relevanz von Einfügeoperationen für Projektionsprädikate 52
4.3 Prüfen der Relevanz von Löschoperationen 53
4.3.1 Relevanz von Löschoperationen für Verbundprädikate 54
4.3.2 Relevanz von Löschoperationen für Selektionsprädikate 55
4.3.3 Relevanz von Löschoperationen für Projektionsprädikate 56
4.4 Prüfen der Relevanz von Update-Operationen 57
4.4.1 Relevanz von Update-Operationen für Verbundprädikate 59
4.4.2 Relevanz von Update-Operationen für Selektionsprädikate 63
4.4.3 Relevanz von Update-Operationen für Projektionsprädikate 64
4.5 Zusammenfassung 65

5 Client-Indexierung mithilfe von PSQ-Anfragen 67
5.1 Sequentielle Speicherung von Anfragen 67
5.2 Trie-basierte Indexierung 69
5.2.1 Praktische Umsetzung der Trie-basierten Indexierung 72
5.2.2 Algorithmen zur Verwaltung des Anfragebaums 76
5.2.3 Algorithmen zur Relevanzprüfung mithilfe des Anfragebaums 80
5.3 Optimierung des Anfragebaums 86

6 Evaluation 95
6.1 Beispielanwendung 95
6.2 Testumgebung 96
6.2.1 Rahmenbedingungen 96
6.2.2 Erzeugen der Testdaten 97
6.3 Evaluation der Speicherauslastung 98
6.4 Performanzevaluation 100
6.4.1 Einfügen von Anfragen 100
6.4.2 Relevanzprüfung 104
6.4.3 Optimierungsaspekte 109

7 Zusammenfassung und Ausblick 113
7.1 Ausblick auf Folgearbeiten 114

A Implementierung des Indexmanagers 119
A.1 Vorbereiten von PSQ-Anfragen 119
A.1.1 Sicherstellung der Großschreibung von Attribut- und Relationennamen 119

A.1.2 Bestimmen von Prädikattypen 120
A.1.3 Extrahieren von Relationennamen 120
A.2 Einfügen von vorverarbeiteten PSQ-Anfragen 121
A.2.1 Ermitteln der Attributnamen von Selektions- und Projektionsprädikaten 121
A.3 Relevanztests 121

B Details zum Anfragemix der Evaluation 123

C Beispiel: TJ-Anfrage bei Updates 125

Abbildungsverzeichnis

1.1 Gesamtarchitektur 18

2.1 Architektur von Infrastrukturnetzen 24
2.2 Einordnung von Techniken zur Cache-Invalidierung 27
2.3 Zusammenhang zwischen Q_u und Q_r 33
2.4 Kommunikationsmuster im Subskriptionsmodell [Leh02, Seite 31] 38

4.1 Eingefügtes Tupel wird in Selektion reflektiert 51
4.2 Durch Verbund- aber nicht durch Selektionsprädikate reflektiertes Einfügen . 52
4.3 Durch Verbund- aber nicht durch Selektionsprädikate reflektiertes Löschen . . 56
4.4 Gelöschtes Tupel wird in Selektion reflektiert (Q ohne Verbunde) 56
4.5 Relevanz bzw. Irrelevanz eines Updates für Verbundprädikate 61
4.6 Herleitung von TJ_{neu} mittels TJ_{alt}, J_{nu} und J_{vu} 61
4.7 Berechnung von TJ 62
4.8 Zusammenfassung der Relevanztests 66

5.1 ER-Schema zur sequentiellen Speicherung von Anfragen 68
5.2 Anfrage-Trie zur Indexierung von Client-IDs 71
5.3 Schematische Aufbau der Knotentypen 74
5.4 Anfragebaum zur Indexierung von Client-IDs 75
5.5 Hilfstruktur zur direkten Indexierung der Blätter 76
5.6 Beispiel für einen optimierten Anfragebaum 87
5.7 Naives Mischen von Q und $\mathcal{Q}$ 89
5.8 Einfügen in $\mathcal{AB}$ vs. Einfügen in $\mathcal{OAB}$ 90
5.9 Löschen einer Anfrage aus einem optimierten Anfragebaum 92

6.1 grundlegendes ER-Schema des Beispielszenarios 96
6.2 Anzahl von Prädikaten in den Anfragesets 98
6.3 Einfügen in den Anfragebaum 100
6.4 Betrachtungen zur Gesamtzeit beim Einfügen in den $\mathcal{OAB}$ 101

6.5 Einfügen in den optimierten Anfragebaum 102
6.6 Sortiertes Einfügen in den optimierten Anfragebaum 103
6.7 Durchschnittliche Laufzeit von Relevanzprüfungen 105
6.8 Einfluss von Ungleichheitsverbunden auf die Laufzeit (Änderungsoperationen mit der längsten Testzeit aus den vorherigen Tests) 107
6.9 Vergleich trie-basierter vs. sequentieller Relevanztest 108
6.10 Materialisierung von Zwischenergebnissen 110
6.11 Vergleich Relevanztest einer Einfügeoperation unter Mengen- und Multimengensemantik 111

Tabellenverzeichnis

6.1 Anzahl von Prädikaten in den Anfragesets 98
6.2 Anzahl von Prädikaten im Anfragebaum 99

B.1 Häufigkeit der Verwendung von Vergleichsoperatoren in den Anfragen 123
B.2 Häufigkeit der Verwendung einzelner Relationen in den Anfragesets 123
B.3 Zusammensetzung der Anfragesets 124

Verzeichnis der Abkürzungen

AID	Anfrageidentifikator
AMD	Advanced Micro Devices
CID	Client-Identifikator
DBMS	Datenbankmanagementsystem
DFG	Deutschen Forschungsgemeinschaft
BKI	Blattknotenindex
EDB	Extensional Data Base
ER	Entity-Relationship
FSK	Freiwillige Selbstkontrolle der Filmwirtschaft
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile Communication
HSCSD	High Speed Circuit Switched Data
HTML	Hypertext Mark-up Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
ID	Identifikator
IDB	Intensional Data Base
LDD	Location Dependent Data
MB	Megabyte
MD5	Message Digest 5
mDBIS	mobile Datenbanken und Informationssysteme
PC	Personal Computer
PDA	Personal Digital Assistant
PSQ	Predicate Sequence Query
QBE	Query-by-Example
RAM	Random Access Memory
SHA	Secure Hash Algorithm
SMoS	Scalable Mobility Server
SuSE	Gesellschaft für Software- und Systementwicklung AG
SPJ	Selection, Projection, Join
SQL	Structured Query Language
UMTS	Universal Mobile Telecommunications System

WAP	Wireless Application Protocol
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Kapitel 1

Einleitung und Motivation

> *The confluence of ever smaller and more powerful computing and communication devices, improved connectivity in both wired and wireless environments, and emerging and accepted standards for data transfer and presentation (e.g., HTML, XML, HTTP, WAP, etc.) are leading to a world in which computers are playing an ever increasing role in peoples's daily lives.*
>
> Michael J. Franklin in [Fra01]

Durch die fortschreitende Entwicklung im Umfeld der Mobilkommunikation entstehen für das Forschungsgebiet der Datenbanken und Informationssysteme zahlreiche neue Anforderungen, welche durch existierende Lösungen nur unzureichend unterstützt werden. Ein wesentlicher neuer Aspekt ist die Vielzahl von mobilen Clients, die auf eine Datenbank zugreifen. Heutzutage ist dies, fast an jedem Ort zu fast jeder Zeit möglich. Die einzigen Einschränkungen hierbei sind die Verfügbarkeit eines Mobilfunknetzes und von genügend Energie (meist Akkus), die Dauer der Datenübertragung und die mit ihr verbundenen Kosten. Um diese Einschränkungen zu minimieren, wurden Techniken entwickelt, die einmal empfangene Daten auf dem Mobilgerät zwischenspeichern und ein erneutes Übertragen derselben Daten vermeiden. Um die Kohärenz dieser Daten zu gewährleisten, müssen sie mit dem Server abgeglichen werden. Im Rahmen dieser Dissertation wird davon ausgegangen, dass Änderungen nur auf dem Server erfolgen. In der klassischen Datenbankforschung ist dieses Problem bekannt als Aktualisierung materialisierter Sichten wobei statt des vollständigen Neuaufbaus einer Sicht versucht wird, die Änderungen inkrementell einzupflegen. Der erste Schritt ist festzustellen, welche Änderungen für welche Sicht relevant sind. Das heißt, dass ermittelt werden muss, welche Sicht Daten enthält, die durch die Änderung entfernt, ergänzt oder verändert werden. Bisher wurde versucht, dieses Problem mithilfe semantischer Korrelationen zu lösen. Der Vorteil dieses Ansatzes ist, dass der „langsame“ Zugriff auf die Daten der Datenbank vermieden werden kann. Der wesentliche Nachteil ist jedoch, dass rein semantische Verfahren die Mächtigkeit der nutzbaren Anfragesprache stark einschränken. Unter Oracle können beispielsweise nur Sichten inkrementell aktualisiert werden, die über genau eine Master-Tabelle definiert wurden. Das

heißt, dass bei der Sichtdefinition keine Verbundoperationen erlaubt sind.

In mobilen Informationssystemen kann der Datenbestand der mobilen Clients als materialisierte Sicht über dem Datenbestand des Servers interpretiert werden. Für Informationssysteme mit sehr vielen mobilen Clients bedeutet dies aber, dass das Prüfen der Relevanz von Änderungen für sehr viele Sichten erfolgen muss. Daher ist das sequentielle Testen jeder Sicht bei jeder Änderung nicht sinnvoll. Im Rahmen dieser Dissertation werden daher Techniken entwickelt, die bezüglich der Anfragemächtigkeit weniger restriktiv als die rein semantischen Verfahren sind, jedoch gleichzeitig syntaktische Überlappungen von mehreren Anfragen nutzen, um diese gemeinsam zu testen.

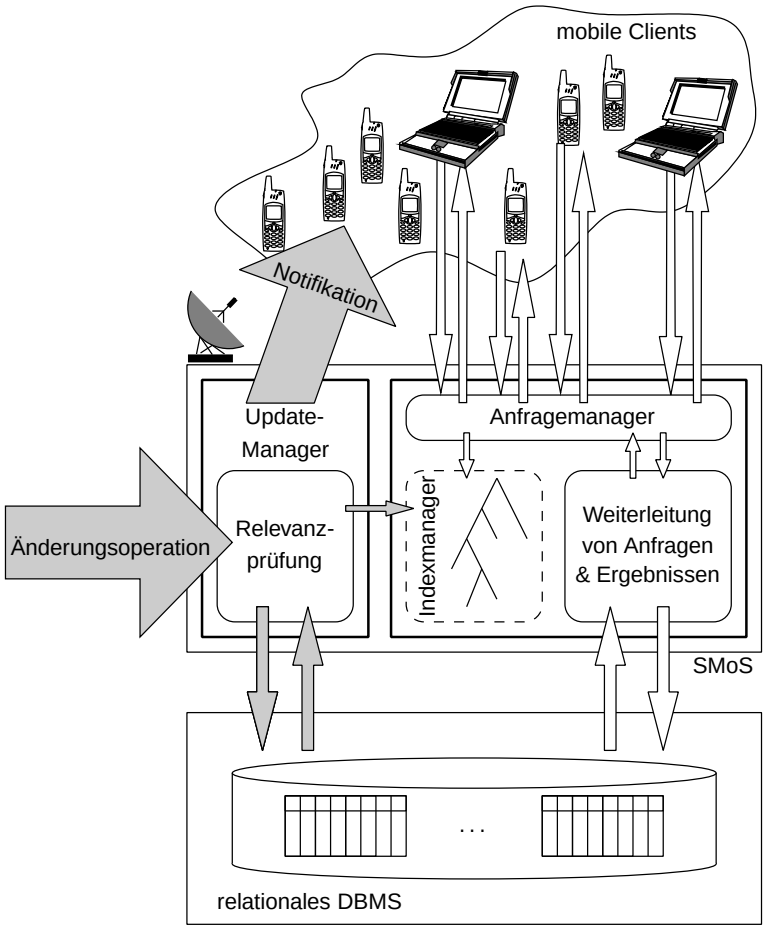


Abbildung 1.1: Gesamtarchitektur

Abbildung 1.1 illustriert schematisch den Aufbau des skalierbaren Mobilitätsservers (SMoS [HS03a]), welcher den Rahmen für diese Dissertation bildet. Mobile Clients stellen verbindungsorientiert Anfragen an SMoS und definieren damit ihre Sicht auf den Datenbestand. Diese Anfragen werden auf dem Server durch den Indexmanager zusammen mit der ID des Clients gespeichert, auf der Datenbank ausgeführt und das entsprechende Ergebnis an den Client zurückgeliefert. Dieser Vorgang ist in Abbildung 1.1 durch die weißen Pfeile

gekennzeichnet. Der mobile Client speichert das Ergebnis der Anfrage lokal und kann es gegebenenfalls wiederverwenden[1] (Caching). Die grauen Pfeile markieren nun die Behandlung von Änderungsoperationen bezüglich der Datenbank. Vor der Ausführung der Änderungen wird anhand des Anfrageindexes geprüft, welche Clientanfragen nach der Ausführung der Änderung ein anderes Ergebnis (als vorher) liefern werden. Die betroffenen Clients werden über die Änderung mittels Broadcast oder ebenfalls verbindungsorientiert informiert und die Änderung auf dem Datenbestand ausgeführt. Da nicht garantiert werden kann, dass alle relevanten mobilen Clients zum Zeitpunkt der Änderung erreichbar sind, müssen die entsprechenden Nachrichten für einen vordefinierten Zeitraum gepuffert und wiederholt gesendet werden. Kernpunkte der vorliegenden Dissertation sind der Indexmanager, und die Relevanzprüfung. Die anderen Komponenten spielen daher im Folgenden eine untergeordnete Rolle.

1.1 Serverseitige vs. clientseitige Relevanzprüfung

Die zentrale Fragestellung dieser Dissertation ist, wie serverseitig festgestellt werden kann, welche mobilen Clients von einer Änderung betroffen sind. Ein naives alternatives Vorgehen zur Entlastung des Servers besteht darin, den Clients die Entscheidung der Relevanz selbst zu überlassen. Hierbei würde es genügen, alle Änderungsoperationen an alle mobilen Clients zu propagieren. Dieses Vorgehen ist jedoch nur bei einer starken Einschränkung der Anfragesprache möglich. Beispiel 1 illustriert, warum ein solches Vorgehen nicht generell möglich ist. Offensichtlich resultiert dieses Problem hier in erster Linie aus der Projektion in der Anfrage. Würde der Client die Attribut `FID` und `KID` mit angefordert haben, wäre eine lokale Relevanzprüfung möglich. Da hierzu aber Daten, die aus Sicht des Nutzers nicht notwendig sind, mit übertragen werden müssen, ist eine solche Lösung ungenügend.

[1] Das eigentliche clientseitige Caching ist nicht Gegenstand dieser Dissertation.

Beispiel 1: *Clientseitige Relevanzprüfung nicht generell möglich*

Seien auf der Serverseite die beiden folgenden Relationen gegeben:

Kinos	KID	Name	Film_ID	Beginn
	99	CinemaxX	01	16:00
	99	CinemaxX	02	19:00

Filme	FID	Titel	Länge
	01	Matrix Reloaded	138 min
	02	Herr der Ringe III	210 min

Der mobile Client hat die folgende Anfrage gestellt: `SELECT Name, Titel, Anfang FROM Kinos, Filme WHERE Film_ID=FID AND Titel='Matrix Reloaded'`. Die entsprechende Ergebnisrelation enthält die folgenden Daten:

Name	Titel	Beginn
CinemaxX	Matrix Reloaded	16:00

Somit kann der Client bei der Änderungsoperation: `UPDATE Kinos SET Beginn='15:30' WHERE KID=99 and Film_ID=01` nicht lokal entscheiden, dass er von der Änderung betroffen ist.

Ein weiteres Problem, welches die clientseitige Relevanzprüfung ausschließt, ist der zusätzliche Berechnungsaufwand. Im angesprochenen Alternativszenario, bei dem alle Änderungen an alle mobilen Clients propagiert werden, muss jeder Client alle Änderungen lokal prüfen. Insbesondere im Hinblick auf die zum Teil stark eingeschränkte Hardware-Performanz mobiler Endgeräte (z. B. Java-Handys) bedeutet dies einen nicht unerheblichen Zusatzaufwand.

1.2 Wahl des Datenmodells

Eine zentrale Fragestellung bei der Realisierung eines Informationssystems auf der Basis eines Datenbankmanagementsystems ist die Wahl des zu verwendenden Datenmodells. Wie in der Einleitung erwähnt ist es in mobilen Informationssystemen notwendig, Daten auf dem mobilen Endgerät zwischenzuspeichern. Um deren Wiederverwendbarkeit zu vereinfachen, ist es sinnvoll, auf Server und Client das gleiche Datenmodell zu verwenden. Aufgrund der Verfügbarkeit entsprechender leichtgewichtiger DBMS für Mobilgeräte stehen hier das XML-Datenmodell [CRZ03] und das Relationenmodell [Cod91, Dat01] zur Diskussion. Ersteres wird durch die Tabino Mobile Suite [Rut01] der Software AG unterstützt. Leider ist zum Zeitpunkt des Schreibens dieser Arbeit unklar, ob bzw. wie die Entwicklung dieses Produktes weitergeführt wird. Für die clientseitige Verarbeitung von Daten im Relationenmodell existieren mehrere Lösungen[2] (z. B. Oracle Lite [Rad04b, Rad04a] oder IBM DB2 Everyplace [Mul00, FKL03]). Daher wurde im Rahmen dieser Arbeit das Relationenmodell gewählt. Prinzipiell sind aber die hier diskutierten Ansätze auch auf XML-Datenbanken übertragbar.

[2] Ein Überblick über mobile Datenbanksysteme kann in [MS04] gefunden werden.

1.3 Nomenklatur

Dieser Abschnitt dient der Festlegung einer einheitlichen Nomenklatur, der die vorliegende Dissertation folgt. Dies ist insbesondere daher notwendig, da selbst im Forschungsgebiet Datenbanken einige Begriffe in unterschiedlichen Kontexten verschieden genutzt werden. Des Weiteren ist eine Vielzahl der in dieser Arbeit zitierten und referenzierten Publikationen in englischer Sprache verfasst worden. Obwohl die Dissertation in Deutsch vorliegt, war es nicht sinnvoll alle Anglizismen ins Deutsche zu übertragen. Der Grund hierfür ist, dass teilweise zwei Worte, welche im Englischen verschieden sind, im Deutschen dieselbe oder eine sehr ähnliche Übersetzung haben. Um Missverständnisse zu vermeiden, wurde daher nicht jeder Fachbegriff übersetzt.

Update vs. Änderungsoperation

Im Rahmen dieser Arbeit wird untersucht, für welche Clients eine Änderungsoperation relevant ist. Der Begriff *Änderungsoperation* umfasst hierbei die drei Datenbankmanipulationsoperationen Einfügen (engl. insert), Löschen (engl. delete) und Aktualisieren (engl. update). Um deutlich zu unterscheiden, wann Änderung und wann Aktualisierung gemeint ist, wird anstelle von Aktualisierung der englische Begriff *Update* verwendet.

Relevanz

Der Begriff *Relevanz* wird in unterschiedlichen Spezialgebieten der Datenbankenforschung homonym verwendet. Beispielsweise beschreibt er im Umfeld von Ähnlichkeitsanfragen in Multimediadatenbanken[3] oftmals ein unscharfes Auswahlkriterium. Gleiches gilt für seine Verwendung im Umfeld der Wissensentdeckung[4]. In der vorliegenden Arbeit wird aber genau dann von der Relevanz einer Änderung gesprochen, wenn sie das Ergebnis einer Anfrage verändert. Es ist hier also ein boolesches Kriterium.

Logische Relevanz vs. Relevanz

In Kapitel 2 werden Algorithmen zum Testen der Relevanz von Änderungsoperationen auf semantischer Ebene vorgestellt. Die im Verlauf dieser Arbeit entwickelten Relevanztests betrachten im Gegensatz dazu nicht ausschließlich Anfragen sondern beziehen auch den aktuellen Datenbestand ein. Um eine Unterscheidung zwischen diesen beiden Ansätzen zu verdeutlichen, wird im Zusammenhang mit rein semantischen Verfahren der Begriff *logische Relevanz* verwendet.

[3] vgl. u. A. [SS03]
[4] vgl. u. A. [HV01]

1.4 Gliederung der Arbeit

Das oben eingeführte Szenario erweckt beim ersten Betrachten den Eindruck, dass es mithilfe des bekannten Publish-Subscribe-Paradigmas realisiert werden kann. Die Einordnung der Thematik sowie die Abgrenzung von verwandten Arbeiten in Kapitel 2 verdeutlicht, dass dies nicht der Fall ist (Abschnitt 2.4) und dass bisher existierenden Verfahren zum Bestimmen der Relevanz unzureichend sind. Hierzu wird in Abschnitt 2.2 das Query Containment Problem, welches die theoretische Grundlage dieser Verfahren (Abschnitt 2.3) bildet, erörtert. In der vorliegenden Arbeit wird die Relevanzprüfung nicht auf der semantischen Ebene, sondern auf Datenebene mithilfe des aktuellen Datenbankzustandes realisiert. Dazu werden aus der Änderungsoperation und der zu prüfenden Anfrage[5] Testanfragen generiert und auf der Datenbank ausgeführt. Im betrachteten Szenario müssen sehr viele Anfragen bezüglich der Relevanz jeder Änderung geprüft werden. Daher wird bei der formalen Herleitung der Testanfragen in Kapitel 4 ein prädikatweises Vorgehen gewählt. Die Idee hierbei ist, Anfragen in ihre Prädikate zu zerlegen und somit Teilanfragen zu betrachten. Der Vorteil dieses Vorgehens besteht nun darin, dass syntaktische Überlappungen zwischen unterschiedlichen Anfragen genutzt werden können, um die Relevanz mehrere Anfragen gleichzeitig zu prüfen. Diese Eigenschaft wird durch den Indexmanager genutzt um eine kompakte Repräsentation aller registrierter Anfragen zu garantieren. Kapitel 5 präsentiert die zu diesem Zweck entwickelten Indexstrukturen. Um die Realisierung dieser zu erleichtern, wird in Kapitel 3 eine spezielle Anfragesyntax eingeführt, welche die erlaubten Prädikate und deren Ordnung innerhalb von Anfragen festlegt. Die Arbeit endet mit einer Evaluation der entwickelten Techniken in Kapitel 6 und einer Zusammenfassung sowie dem Ausblick auf Folgearbeiten in Abschnitt 7.

[5] konjunktive Anfragen, die aus Selektionen, Verbunden und einer Projektion bestehen dürfen wobei sowohl Selbstverbund, als auch Ungleichheitsvergleiche erlaubt sind

Kapitel 2

Einordnung der Thematik

Erklärt Euch, eh Ihr weitergeht, was wählt Ihr für eine Fakultät? (Mephistopheles)

Johann Wolfgang Goethe in [Goe08]

Ziel dieses Kapitels ist es, nach einer groben Einordnung der Thematik dieser Dissertation, eine Abgrenzung von verwandten Forschungsgebieten vorzunehmen. Hierzu wird nach der allgemeinen Einordnung in Abschnitt 2.1 separat auf drei – für diese Dissertation besonders relevante – Forschungsschwerpunkte eingegangen. Abschnitt 2.2 diskutiert das Problem des Enthaltenseins von Anfragen. Das inkrementelle Ändern von materialisierten Sichten ist Gegenstand von Abschnitt 2.3. Abschließend wird in Abschnitt 2.4 die Abgrenzung zu Subskriptionssystemen erörtert.

2.1 Allgemeine Einordnung

Eine sehr grobe Einordnung der Thematik dieser Dissertation kann in das Forschungsgebiet „mobile Datenbanken und Informationssysteme“ (mDBIS) vorgenommen werden. Aus Sicht der Netzwerktechnologie kann dieser recht weit gefasste Rahmen aber auf Infrastrukturnetzwerke eingeschränkt werden. Ad-Hoc-Netzwerke mit den auf ihre Besonderheiten aufbauenden Verfahren wie dem Peer-to-Peer Computing spielen zwar ebenfalls im mDBIS-Umfeld eine wichtige Rolle, werden in dieser Dissertation aber nicht betrachtet. Infrastrukturnetzwerke sind dadurch gekennzeichnet, dass Mobilgeräte sich über eine Basisstation in ein statisches Netzwerk einwählen können. In diesem Netzwerk sind neben der Basisstation noch weitere statische Geräte (Computer) miteinander vernetzt. Abbildung 2.1 illustriert (basierend auf [PS97]) diese Architektur. Typischerweise formen Basisstationen Zellen. Diese beschreiben die geographische Region um eine Basisstation, die von dieser abgedeckt wird. Das heißt, dass sich Mobilgeräte, die sich in dieser Region befinden, potentiell[1] bei dieser Basisstation einwählen können. Beispiele für Infrastrukturnetze sind die Mobilfunknetze GSM [ETS93], GPRS [ETS98], HSCSD [ETS99a] und UMTS [ETS99b].

[1] die entsprechenden Rechte vorausgesetzt

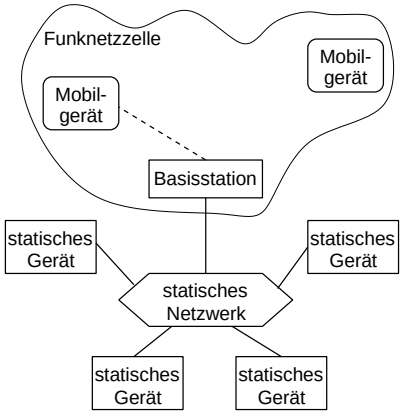


Abbildung 2.1: Architektur von Infrastrukturnetzen

Arten der Mobilität

Aus Sicht der Mobilkommunikation wird in [Rot02] (basierend auf [Pan00]) zwischen drei Arten von Mobilität unterschieden: *Endgerätemobilität* liegt vor, wenn ein Mobilgerät trotz einer Ortsveränderung vernetzt bleibt. Ein typisches Beispiel sind Handys in GSM-Netzen. Die *Benutzermobilität* fokussiert auf die Ortsveränderung der Nutzer wobei keine Endgerätemobilität vorliegen muss. Betrachtet man beispielsweise die Sprachkommunikation als Dienst, so kann dieser an unterschiedlichen Orten von einem Nutzer durch unterschiedliche Handys genutzt werden. Die dritte Art ist die *Dienstmobilität* bei der dem Nutzer personalisierte Dienste in der für ihn gewohnten Ausprägung zur Verfügung stehen. Die in dieser Dissertation vorgestellten Techniken unterstützen die Endgerätemobilität, da zur serverseitigen Identifikation eine Geräte-ID verwendet wird. Des Weiteren werden Anfrageergebnisse auf den Mobilgeräten zwischengespeichert. Nutzermobilität ist somit nicht sinnvoll realisierbar, zumal auf einem zweiten Mobilgerät im Allgemeinen ohnehin andere Daten gehalten werden. Damit kann auch eine Dienstmobilität nicht sinnvoll realisiert werden.

Eigenschaften mobiler Endgeräte

Eine weitere Verfeinerung der Eingrenzung führt zu Client-Server Datenbanksystemen, wobei die Clients mobil sind und den bekannte Ressourcenbeschränkungen unterliegen:

Energieversorgung: Typischerweise werden Mobilgeräte durch Batterien mit Energie versorgt womit ihre maximale Einsatzdauer beschränkt ist.

Konnektivität: Um einen hohen Grad an Mobilität zu realisieren, greifen mobile Clients über Funknetzwerke auf den Server zu. Die Nutzung derartige Netze ist, wie bereits in Kapitel 1 erwähnt, meist kostenintensiv und die Bandbreite, verglichen mit festnetzbasierter Kommunikation, gering. Darüberhinaus können verschiedene Umwelteinflüsse (Abschirmung durch Gebäude, etc.) dazu führen, dass ein Verbindungsaufbau durch den

Client nicht möglich ist.

Leichtgewichtigkeit: Aus Sicht der Hardware-Performanz muss festgestellt werden, dass Mobilgeräte im Vergleich zu stationären Computern weniger leistungsfähig sind. Insbesondere die Entwicklungen im Mobiltelefonsektor (Programmierbarkeit mit Java) führen dazu, dass zunehmend leichtgewichtige Geräte als Clients mobiler Informationssysteme genutzt werden.

In [Buc02] wird diskutiert, wie derartige Ressourcen substituiert werden können. Ein Beispiel hierbei ist, den Kommunikationsaufwand durch clientseitiges Caching zu reduzieren. Dieser Ansatz spiegelt sich auch in dieser Dissertation wider.

Semantisches Caching

Um einen hohen Grad an Wiederverwendbarkeit von lokal zwischengespeicherten Daten zu erreichen, werden häufig – insbesondere im relationalen Datenmodell – semantische Caches genutzt [KB96, GG99, RD03]. Im Gegensatz zu tupel- oder seitenweisem Caching werden hierbei Anfrageergebnisse im Cache gehalten und mittels der entsprechenden (Datenbank)anfrage indexiert. Bei neuen Anfragen wird dann geprüft, inwieweit die Daten im Cache wiederverwendet werden können. Ist ein äquivalentes Anfrageergebnis lokal vorhanden, so kann die neue Anfrage lokal beantwortet werden. Ist eine Obermenge (also das Ergebnis einer generelleren Anfrage) verfügbar, so kann dieses durch Filterung [HS03c] genutzt werden. Auch hierbei ist keine Kommunikation mit dem Server notwendig. Enthält der Cache eine Teilmenge des Ergebnisses der neuen Anfrage, so genügt es, eine Kompensationsanfrage an den Server zu schicken und das Gesamtergebnis lokal zusammenzufügen. Lediglich in dem Fall, dass der Cache nicht einmal eine Teilmenge des Ergebnisses der Anfrage enthält, muss diese vollständig an den Server geschickt werden. Da der Speicherplatz auf Mobilgeräten beschränkt ist, ist es sinnvoll, im Cache redundante Daten zu vermeiden. Bei semantischen Caches wird hierzu auf die Idee der semantischen Regionen [DFJ+96] zurückgegriffen. Es wird gefordert, dass die Vereinigung aller semantischer Regionen den Cache vollständig abdeckt wobei semantische Regionen nicht überlappend sein dürfen. Darauf aufbauend existieren Ansätze zur Clusterbildung [RD99], zur Cachepartitionierung [LLS99] und zum Zusammenfassen von überlappenden Anfragen [HS04]. Darüberhinaus wurde im Hinblick auf lokationsbasierte Informationssysteme in [RD00] mit dem LDD-Cache ein semantischer Cache für lokationsabhängige Daten vorgestellt.

Hoarding

Neben dem Caching können zur Reduktion der Kommunikationskosten auch Hoarding-Techniken eingesetzt werden. Das Horten von Daten ist hierbei als Erweiterung des reinen Cachings zu betrachten. Die Idee ist, frühzeitig festzustellen, welche Daten der Nutzer eines Mobilgerätes später benötigen wird, und diese dann provisorisch vom Server anzufordern. Ein bekannter Vertreter derartiger Techniken ist das Netzwerkdateisystem Coda [SER+00]. Dieses

ermöglicht es dem Nutzer manuell [KS92] festzulegen, welche Dateien er im lokalen Cache seines Mobilgerätes halten will. Dies ist insbesondere dann sinnvoll, wenn beispielsweise ein definierter Verbindungsabbruch bevorsteht. Wurden die richtigen Daten gehortet, so kann der Nutzer in der anschließenden Offline-Phase weiterarbeiten. Neben der manuellen Variante, besteht auch die Möglichkeit, dass das System selbst (beispielsweise durch Analyse von Dateizugriffsmustern in Kombination mit Clusterbildung) entscheidet, welche Daten in den Cache geholt und auch dort gehalten werden [KP97a, KP97b]. Im mobilen Umfeld kann darüberhinaus ortsbezogen gehortet werden. Dabei kann der aktuelle Aufenthaltsort [Pei04] bzw. ein vorhergesagter zukünftiger Aufenthaltsort [KR01b] des Mobilgerätes genutzt werden, um lokationsbezogen Daten anzufordern und lokal zwischenzuspeichern.

Replikation

Manuelle Hoarding-Techniken überlappen stark mit dem Themengebiet der Replikation [Höp02]. Auch hier definiert der Nutzer, welche Daten auf das Mobilgerät repliziert werden [Gol03]. Der Unterschied ist, dass bei der Replikation garantiert wird, dass diese Daten auch in einer Offline-Phase verfügbar sind[2]. Typischerweise erlaubt Replikation, dass auf dem Mobilgerät auch offline Änderungen an den replizierten Daten vorgenommen werden dürfen. Um diese bei der nächsten Online-Phase in den Serverdatenbestand zu integrieren, sind Synchronisationsverfahren [Höp01] notwendig. Im wesentlichen kommen hierbei optimistische Verfahren zum Einsatz, die eventuell auftretende Konflikte aufzulösen. Es kann zwischen zwei Klassen von optimistischen Synchronisationsverfahren unterschieden werden. Beim transaktionsorientierten [GHOS96] Vorgehen wird versucht, die Transaktionen, die provisorisch schon einmal auf dem Mobilgerät ausgeführt wurden, auch auf dem Server auszuführen. Eine Beispielimplementierung dieses Vorgehens ist das Bayou-System [TDP+94]. Der datenzentrierte Ansatz erkennt Konflikte durch Vergleichen der lokal geänderten Daten mit den aktuellen Daten auf dem Server. Bei Konflikten greifen dann Regeln [GMAB+84], die je nach Art des Konfliktes die Änderungen integrieren oder auf dem Mobilgerät zurücksetzen. Die Alternative zum optimistischen Vorgehen ist, dass bereits bei der Auswahl der zu replizierenden Daten, die zulässigen Änderungen eingeschränkt werden [BG04]. Somit kann es bei der Synchronisation zu keinen Konflikten kommen.

Kohärenzsicherung

Das lokale Vorhalten von Daten auf dem mobilen Endgerät sowohl durch Caching, Hoarding als auch durch Replikation führt dazu, dass Daten redundant im Informationssystem vorliegen. Ändert sich der Datenbestand auf dem Server, entstehen Inkonsitenzen und somit Konflikte. Bei der Replikation werden diese analog zu den Änderungen auf dem mobilen Client bei der Synchronisation behandelt[3]. Treten derartige Änderungen jedoch beim Caching oder Hoarding auf, kann die Kohärenz der lokalen Daten nicht garantiert werden. Beispielsweise könnte in

[2] Beim Hoarding kann dies durch eine ungünstige Entscheidung der Ersetzungsstrategie verhindert werden.

[3] Wobei kohärenzsichernde Maßnahmen während der Online-Phase auch hier sinnvoll sind.

diesem Fall eine neue Anfrage alte Daten aus dem Cache benutzen ohne dass dieser Fehler dem Nutzer ersichtlich ist. Um dies zu vermeiden, müssen die Clients vom Server über die Änderungen informiert bzw. die Änderungen auch auf dem Mobilgerät durchgeführt werden.

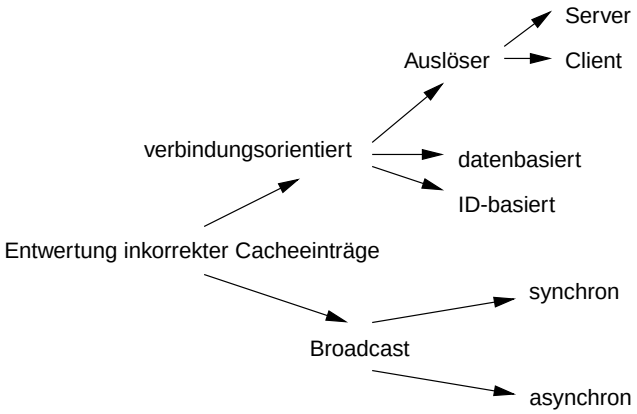


Abbildung 2.2: Einordnung von Techniken zur Cache-Invalidierung

Abbildung 2.2 illustriert, basierend auf der Taxonomie in [CTO97], die hierzu einsetzbaren Techniken. Die Wahl eines Verfahrens hängt hierbei in erster Linie von der Implementierung des Servers ab. Bei *zustandslosen Servern* werden keine Informationen über mobile Clients und deren Cache-Inhalte gespeichert. Somit kann die Entscheidung, für wen eine Änderung relevant ist, nicht serverseitig getroffen werden. Aus diesem Grund werden die Informationen über die Änderung an die Clients propagiert und diese prüfen die Relevanz lokal. Typischerweise werden hierzu Broadcast-Verfahren zur Datenübertragung genutzt. Des Weiteren kann das Senden einer solchen Invalidierungsnachricht sofort beim Auftreten der Änderung (synchron) oder nach einer bestimmten Anzahl von Änderungen (asynchron) erfolgen. Wie bereits in Beispiel 1 in Kapitel 1 illustriert, ist aber das lokale Prüfen nur bei einer starken Einschränkung der Anfragemächtigkeit möglich. Im Rahmen der vorliegenden Dissertation wird ein *zustandsbehafteter Server* realisiert, der die Anfragen der Clients speichert und somit feststellen kann, welche Clients von der Änderung betroffen sind. Auch hier können Broadcast-Techniken zur Invalidierung des Caches eingesetzt werden. Darüberhinaus besteht die Möglichkeit, die betroffenen Clients direkt (verbindungsorientiert) über die Änderung zu informieren, wobei abhängig von der Umsetzung des Invalidierungsprotokolls und des Caches entweder datenbasiert oder ID-basiert vorgegangen wird. ID-basiert bedeutet in diesem Zusammenhang, dass der Server nicht nur die Anfragen und IDs der Clients kennt, sondern auch deren internen Cacheaufbau. Somit können z.B. bei semantischen Caches den semantischen Regionen IDs zugewiesen werden. Die Invalidierung erfolgt dann gezielt für diejenige Region eines Clients, die von der Änderung betroffen ist. Stehen derartige Informationen nicht zur Verfügung, obliegt es dem Client entsprechende Maßnahmen zu ergreifen. Dazu muss er aber lokal festellen können, welcher Teil des Caches durch die Änderung betroffen ist. Eine weitere Unterscheidung muss zwischen *Push-* und *Pull-Techniken* gemacht werden. Löst wie bisher beschrieben der Server die Propagierung aus, so wird von *Push-Propagierung* gesprochen. Insbesondere

die Tatsache, dass das betroffene Mobilgerät zum Zeitpunkt der Nachricht nicht online sein muss, motiviert die Möglichkeit, vorliegende Änderungen durch den Client vom Server abzufragen. In diesem Fall wird von *Pull-Propagierung* gesprochen.

2.2 Query Containment

Die Frage, ob eine Anfrage Q_1 in einer zweiten Anfrage Q_2 enthalten ist, spielt in zahlreichen Bereichen der Datenbankforschung eine wesentliche Rolle. Bei dem oben bereits angesprochenen semantischen Caching muss anhand des Cache-Indexes, der aus Anfragen besteht, festgestellt werden, welche Teilmengen des Ergebnisses einer neuen Anfrage aus dem Cache gewonnen werden können und welche Teile vom Server anzufordern sind. Darüberhinaus spielt das Query Containment eine grundlegende Rolle bei der Beantwortung von Anfragen mithilfe von Sichten [Hal01] und bei der Datenintegration [MHF00].

Allgemein gilt für zwei Anfragen Q_1 und Q_2, dass Q_1 in Q_2 enthalten ist (geschrieben als $Q_1 \sqsubseteq Q_2$) wenn für alle Datenbankinstanzen D gilt, dass die Menge der durch Q_1 berechneten Objekte (im Relationenmodell Tupel) eine Teilmenge der durch Q_2 berechneten Objekte ist. Es muss also $Q_1(D) \subseteq Q_2(D)$ gelten. Darüberhinaus sind die beiden Anfragen äquivalent ($Q_1 \equiv Q_2$), wenn sowohl $Q_1 \sqsubseteq Q_2$ als auch $Q_2 \sqsubseteq Q_1$ gilt.

2.2.1 Algorithmen zum Testen des Query Containments

Nach [Sol79] ist das Query Containment Problem für beliebige relationale Kalküle und beliebige Anfragen in Relationenalgebra unentscheidbar. Gleiches gilt nach [Shm87] basierend auf [Pap85] auch für logische Anfragesprachen. Dennoch wurde bereits in [CM77] bewiesen, dass das Query Containment Problem für konjunktive Anfragen entscheidbar aber NP-vollständig ist. Durch geeignete Beschränkung der Anfragesprache (z.B. auf SPJ-Anfragen[4]) ist es möglich, Algorithmen mit polynomialer Zeitkomplexität anzugeben. Eine detaillierte Aufarbeitung der Komplexität des Query Containment Problems bezogen auf die Mächtigkeit konjunktiver Anfragen ist in [KMT98] zu finden.

Hypergraphen

Eine Möglichkeit des Prüfens ob eine SPJ-Anfrage in einer zweiten SPJ-Anfrage enthalten ist, basiert auf der Hypergraphdarstellung von Anfragen. Hierzu werden die Konstanten und Variablen der Anfragen als Knoten betrachtet. Die Hyperkanten beschreiben die Konjunkte[5] indem sie diejenigen Variablen und Konstanten „umfassen“, die zusammen in dem jeweiligen Konjunkt auftreten [Ull82]. Durch Beschränkung auf azyklische[6] Anfragen kann das Query

[4] SPJ-Anfragen sind auf Konstantenselektion ($A = c$), den natürlichen Verbund $\bowtie$ und Projektion beschränkt.
[5] In der Literatur wird hierbei häufi g von Teilzielen (subgoals) gesprochen.
[6] Eine Anfrage ist azyklisch, wenn ihre Hypergraphdarstellung azyklisch ist (vgl. [Qia96]).

Containment mit polynomialer Zeitkomplexität geprüft werden. Das Prüfen, ob ein Hypergraph azyklisch ist, kann mittels *GYO-Reduktion* [YO79] erfolgen. Die Idee hierbei ist, dass alle so genannten Ohren aus dem Hypergraph entfernt werden. Ohren sind Hyperkanten die nur Knoten enthalten, die in maximal einer anderen Hyperkante auftreten. Ist der somit entstandene Hypergraph leer, so ist die Anfrage azyklisch. Ein entsprechender Algorithmus mit linearer Zeitkomplexität, der testet, ob eine Anfrage Q_1 azyklisch ist, wurde in [TY84] präsentiert. Dieser Algorithmus berechnet auch den entsprechenden Verbundbaum[7] der zur Eingabe des Algorithmus' *AcyclicContainment* [CR97] benutzt wird. Dieser, auf den Ergebnissen aus [Qia96] basierende Algorithmus, bildet nun die Konjunkte einer Anfrage Q_2 knotenweise und bottom-up auf die Knoten des Verbundbaumes ab und berechnet anhand der Datenbank die jeweiligen Teilergebnisse. Diese werden in den Zwischenknoten mit dem natürlichen Semiverbund zusammengefasst und an die darüberliegenden Knoten weitergereicht. Q_2 ist damit in Q_1 enthalten, wenn die im Wurzelknoten entstandene Relation nicht leer ist. Es ist anzumerken, dass Q_2 für diesen Algorithmus nicht azyklisch sein muss. Ist dies nicht der Fall und soll getestet werden, ob $Q_2 \sqsubseteq Q_1$ gilt, so kann auf den Algorithmus *QueryContainment* zurückgegriffen werden, der ebenfalls in [CR97] publiziert wurde. Dieser benutzt statt des Verbundbaumes einen Zerlegungsbaum (engl. decomposition tree) funktioniert aber ansonsten analog[8] zur azyklischen Variante. Es wurde gezeigt, dass die Zeitkomplexität dieses Algorithmus' exponentiell von der Breite der Anfrage abhängt. Dieser Wert entspricht der maximalen Kardinalität der Knoten des Zerlegungsbaums (siehe hierzu auch [GLS01a]). In [GLS99] wird gezeigt, dass das Feststellen, ob eine konjunktive Anfrage höchstens die Breite 4 besitzt, NP-vollständig ist.

[GLS01b] diskutiert ausführlich die Komplexität des Query Containment Problems und verschiedener weiterer Probleme im Zusammenhang mit azyklischen Anfragen.

Tableaus

Eine weitere Möglichkeit zur Bestimmung des Query Containments besteht in der Nutzung der Tableau-Notation von Anfragen. Diese spielt u. A. auch bei der logischen Optimierung von Verbundanfragen (vgl. [SH99]) eine wesentliche Rolle. Tableau-Anfragen sind äquivalent zu den oben angesprochenen SPJ-Anfragen und zu QBE[9]-Anfragen [Zlo75]. Ein Tableau T ist eine Matrix deren Spalten die Attribute der Universalrelation bilden. Die erste Zeile, die als *summary* bezeichnet wird, enthält die ausgezeichneten Variablen (bzw. *blanks*) und repräsentiert somit die Projektion. Die weiteren Zeilen können nun *nichtausgezeichnete* Variablen, ausgezeichnete Variablen, blanks und Konstanten enthalten und beschreiben so, Verbunde und Selektionen. Eine derartig notierte Anfrage T_2 ist in einer weiteren Anfrage T_1 enthalten, wenn eine homomorphe Abbildung $\Phi : T_1 \rightarrow T_2$ angegeben werden kann, die T_1 auf T_2 abbildet und die folgenden Bedingungen erfüllt:

- Alle ausgezeichneten Variablen in einer Spalte A einer Zeile i werden auf eine ausgezeichnete Variable oder eine Konstante in Spalte A der Zeile $\Phi(i)$ in T_2 abgebildet.

[7] in [CR97] als „Elimination Tree“ bezeichnet
[8] bis auf die Initialisierungsphase
[9] Query-by-Example

- Wenn Zeile i in T_1 in der Spalte A eine Konstante c enthält, muss c auch in Zeile $\Phi(i)$ in der Spalte A stehen.

- Wenn die Zeilen i und j in T_1 in Spalte A die gleiche nichtausgezeichnete Variable enthalten, dann muss T_2 in den Zeilen $\Phi(i)$ und $\Phi(j)$ und der Spalte A das gleiche Symbol (Konstante, nichtausgezeichnete oder ausgezeichnete Variable) enthalten.

In [ASU79] wird als Algorithmus zum Finden einer solchen Abbildung angegeben, dass alle möglichen Abbildungen durchgetestet werden müssen, was zu einer exponentiellen Zeitkomplexität führt.

In [SY80] wird neben der Erweiterung des Tableau-Ansatzes zur relationalen Vollständigkeit gezeigt, dass für drei Teilklassen der Tableaus das Containment in polynomialer Zeit geprüft werden kann. Die erste Klasse sind die bereits in [ASU79] betrachteten einfachen Tableaus (engl. simple tableaux). Einfache Tableaus sind dadurch gekennzeichnet, dass gilt: wenn in einer Spalte i eine nichtausgezeichnete Variable in mehreren Zeilen auftritt, so darf in derselben Spalte keine ausgezeichnete Variable in mehreren Zeilen auftreten. Der Containment-Algorithmus hierbei basiert auf dem Algorithmus aus [ASU79], der die Äquivalenz von einfachen Tableaus in $O(n^3)$ mit der Größe n der beiden zu betrachtenden Tableaus berechnet. Damit gilt $T_2 \sqsubseteq T_1$ nur dann, wenn auch $T_1 \bowtie T_2 \equiv T_2$ gilt.

Die zweite Klasse erlaubt nur eine in mehreren Zeilen auftretende nichtausgezeichnete Variable pro Zeile. In diesem Fall gilt $T_2 \sqsubseteq T_1$ genau dann wenn gilt:

- Alle Zeilen in T_1, die die nichtausgezeichnete Variable a enthalten, können auf eine Menge von Zeilen in T_2 abgebildet werden. Die Zeilen in dieser Menge sind dadurch gekennzeichnet, dass sie in der Spalte von a das gleiche Symbol enthalten.

- Jede Zeile in T_1 die keine wiederholte, nichtausgezeichnete Variable enthält kann auf eine Zeile in T_2 abgebildet werden.

Mithilfe einer somit möglichen Linkliste, die für jede Variable a in T_1 und T_2 auf die Zeilen verweist in denen a vorkommt, können die beiden Bedingungen in $O(n^2)$ mit der Größe n der beiden zu betrachtenden Tableaus geprüft werden.

Die dritte Klasse von Tableaus, deren Containment-Prüfung in polynomialer Zeit durchgeführt werden können, bilden Paare von Tableaus T_1 und T_2 für die gilt, dass jede Zeile aus T_1 auf maximal zwei Zeilen in T_2 abgebildet werden kann.

In [Klu88] wird untersucht, wie das Query Containment mithilfe von Tableaus für konjunktive „Ungleichheits-Anfragen“ bestimmt werden kann. Die Idee hierbei ist, dass die Abbildungsfunktion nicht nur von einem Tableau T_1 auf ein zweites T_2 untersucht wird, sondern dass T_2 als Datenbankzustand interpretiert wird. Um nun Ungleichheits-Anfragen testen zu können, müssen mehrere (möglicherweise exponentiell viele) alternative Instanzen von T_2 betrachtet werden.

Überführung des Query Containment Problems auf eine logische Implikation

Neben den beiden bisher betrachteten Ansätzen besteht auch die Möglichkeit, die konjunktiven Anfragen als logische Formeln zu interpretieren. In der Literatur wird auf diese Weise das Query Containment Problem $Q_1 \sqsubseteq Q_2$ häufig durch die Implikation $Q_1 \rightarrow Q_2$ dargestellt. Es besteht also eine Enthaltenseinsbeziehung, wenn die Implikation $Q_1 \rightarrow Q_2$ erfüllt ist. Offensichtlich ist dies immer dann der Fall, wenn Q_1 unerfüllbar ist da auch $\emptyset \subseteq Q_2(D)$ immer gilt. Nach [GSW96b] impliziert die Anfrage Q_1 die Anfrage Q_2 darüberhinaus nur dann, wenn alle Variablenbelegungen, die Q_1 erfüllen, auch Q_2 erfüllen. Dieses Vorgehen ermöglicht das Zulassen von „Ungleichheits-Anfragen“ wobei jedoch die Wertebereiche der Variablen auf nummerische Werte beschränkt werden.

In [RH80] wurde ein Algorithmus vorgestellt, der die Erfüllbarkeit von konjunktiven, booleschen Anfragen[10] mit kubischer Zeitkomplexität prüft. Hierzu muss jedoch der Vergleichoperator $\neq$ verboten werden. Nach der Normalisierung der Anfrage der Art, dass die Konjunkte nur noch die Form $x \leq y+c$, $x \leq c$ oder $x \geq c$ (x und y sind Variablen, c ist eine Konstante) aufweisen, wird die Anfrage als gerichteter Graph mit gewichteten Kanten dargestellt. Dieser Graph besitzt einen Knoten für jede Variable und einen Knoten für die Konstante 0. Für jedes Konjunkt $x \leq y + c$ existiert eine Kante von Knoten x zu Knoten y mit dem Gewicht c. Konstantenvergleiche $x \leq c$ werden als Kante von Knoten x zum Knoten 0 abgebildet und erhalten das Gewicht c. Konstantenvergleiche der Form $x \geq c$ werden als mit $-c$ gewichtete Kante vom Knoten 0 zum Knoten x dargestellt. Wenn dieser Graph einen negativ gewichteten Zyklus enthält, ist die Anfrage unerfüllbar. Anderenfalls ist die Anfrage erfüllbar. Das Testen dieser Eigenschaft erfolgt analog zu [Pra77] mit dem Shortest Path Algorithmus [Flo62] in $O(n^3)$ (n ist die Anzahl der Vergleiche in der Anfrage). Ebenfalls in [RH80] wurde nachgewiesen, dass das Problem der Erfüllbarkeit im Fall der Zulassung von $\neq$-Vergleichen NP-hart ist. Dennoch, wird in [SKN89] ein auf diesem Ansatz aufbauender Algorithmus *Restricted-Implication-Check* angegeben, der entscheidet, ob eine Anfrage Q_1 eine zweite Anfrage Q_2 impliziert. Hierzu wird zuerst geprüft, ob in Q_1 Variablen auftreten, die nicht in der Graphdarstellung von Q_2 enthalten sind. Ist dies der Fall, so ist die Implikation nicht erfüllt und es besteht kein Containment der Form $Q_1 \sqsubseteq Q_2$. Anderenfalls, wird mithilfe des Shortest Path Algorithmus' die Erfüllbarkeit von Q_1 geprüft. Ist Q_1 unerfüllbar, gilt die Implikation als erfüllt. Im dritten Schritt werden die Konjunkte der Form $x \leq y + c$ aus Q_2 untersucht. Wenn der kürzeste Pfad von x nach y im Graphen von Q_1 mit c' gewichtet ist und $c' > c$ gilt, ist die Implikation nicht erfüllt. Existiert kein solcher Pfad, werden im vierten Schritt die Konjunkte der Form $x \neq y + c$ betrachtet. Hat der kürzeste Pfad von x nach y im Graphen von Q_1 ein Gewicht $c' > c-1$ und ist das Gewicht des kürzesten Pfades von y nach x größer als $-c-1$, dann ist die Implikation nicht erfüllt. Werden keine derartige Kanten gefunden, gilt die Implikation.

In [GSW96b] werden weitere Algorithmen angegeben, welche die Erfüllbarkeit einer Anfrage bzw. die Implikation $Q_1 \rightarrow Q_2$ prüfen. Hierbei werden die zugelassenen Vergleichstypen beschränkt und nur noch $x \phi y$ und $x \phi c$ mit den Variablen x, y und der Konstante c zugelassen. Mithilfe dieser Einschränkungen gelingt es sowohl für ganzzahlige Wertebereiche und

[10] Boolsche Anfragen sind Anfragen, bei denen alle Variablen existenzquantifiziert sind. [RH80]

$\phi \in \{<, \leq, =, \geq, >\}$ als auch für Gleitkommazahlen und $\phi \in \{<, \leq, =, \neq, \geq, >\}$ entsprechende Algorithmen anzugeben. In [GSW96a] wird diese Beschränkung aufgehoben und ein Algorithmus für Gleitkommawertebereiche; $x\phi y$, $x\phi c$ und $x\phi(y + c)$ mit $\phi \in \{<, \leq, =, \neq, \geq, >\}$ angegeben. Ein kompakter Überblick über die entsprechenden Komplexitäten dieser Algorithmen ist in [RD98] zu finden.

Query Containment bei Logik-Datenbankanfragen

Im Paradigma der Logik-Datenbankanfragen werden Anfragen als logische Programme aus Hornklauseln repräsentiert. Nach [Sag87a, Sag87b] ist ein derartiges Programm P_2 in einem weiteren Programm P_1 enthalten, wenn für alle extensionalen Datenbanken (EDB) $E_1, \ldots, E_m$ die Ausgabe von P_2 in der Ausgabe von P_1 enthalten ist. Es muss also, analog zu den bisher diskutierten Ansätzen $P_2(E_1, \ldots, E_m) \subseteq P_1(E_1, \ldots, E_m)$ gelten. Da Logik-Datenbankanfragen aber nicht nur auf Relationen, welche die Fakten enthalten und die EDB bilden, basieren, sondern auch abgeleitete Relationen durch IDB-Prädikate zulassen (vgl. [CGH98, EN02]), muss die Enthaltenseinsbeziehung angepasst werden. Nach [Sag87a, Sag87b] ist das uniforme Query Containment von P_2 in P_1, welches als $P_2 \sqsubseteq^u P_1$ geschrieben wird, gegeben, wenn die Menge alle Modelle von P_1 Teilmenge der Modelle von P_2 ist ($P_2 \sqsubseteq^u P_1 \leftrightarrow M(P_1) \subseteq M(P_2)$). Der Zusammenhang zwischen dem Containment und dem uniformen Containment ist der Folgende: Wenn P_2 uniform in P_1 enthalten ist, dann muss P_2 auch in P_1 enthalten sein. Der Umkehrschluss gilt aber nicht (vgl. [LS93]). Folglich ist $P_2 \sqsubseteq^u P_1 \rightarrow P_2 \sqsubseteq P_1$ gegeben. Im Gegensatz zum allgemeinen Containment, welches wie bereits erwähnt nach [Shm87] unentscheidbar ist, kann das uniforme Containment für Datalogprogramme ohne eingebaute Prädikate oder Negationen entschieden werden. In [Sag87a] wird hierzu der Chase-Prozess [MMS79] verwendet. Diese Idee basiert auf [CK86] wo gezeigt wurde, dass der Chase-Prozess für das Testen des uniformen Containments von Datalogprogrammen mit nur einem Prädikat benutzt werden kann. $M(P_1) \subseteq M(P_2)$ gilt genau dann, wenn für alle Regeln r in P_2 auch $M(P_1) \subseteq M(r)$ gilt. Damit gilt $P_2 \sqsubseteq^u P_1$ nur dann, wenn auch $\forall r \in P_2(r \sqsubseteq^u P_1)$ erfüllt ist. Es kann also jede Regel aus P_2 separat geprüft werden.

2.2.2 Zusammenhang zwischen Query Containment und der Relevanzprüfung

Der Zusammenhang zwischen Query Containment und dem Thema dieser Dissertation ist der folgende: die Änderungsoperationen sind für einen bestimmten Datenbereich definiert, welcher durch Anfragen Q_u bestimmt werden kann. Für Einfüge- und Löschoperationen kann Q_u aus der Anfrage recht einfach abgeleitet werden. Bei Updates muss zum Einen der Datenbereich, der von dem Update betroffen ist $Q_u^v(D)$ als auch der Datenbereich $Q_u^n(D)$, der durch das Update entsteht, betrachtet werden. Q_u ergibt sich in diesem Fall unter der Annahme, dass $Q_u^v(D) \neq Q_u^n(D)$ gilt, durch die Disjunktion $Q_u = Q_u^v \vee Q_u^n$. Die Annahme ist nun, dass eine Änderung für eine registrierte Anfrage Q_r relevant ist, wenn $Q_u \sqsubseteq Q_r \vee Q_r \sqsubseteq Q_u$ gilt. Es muss also der geänderte Datenbereich Teilmenge des durch Q_r abgedeckten Datenbereichs

sein oder diesen vollständig enthalten.

Wie jedoch Beispiel 2 illustriert, ist diese Bedingung maximal notwendig jedoch nicht hinreichend. In diesem Fall besteht keine Relevanz obwohl eine Containment-Beziehung besteht. Dieses Problem wurde bereits bei der Interpretation des Query Containment als Implikation erwähnt. Q_u ist für die gegebene Datenbank nicht erfüllbar. Damit gilt die Implikation $Q_u \rightarrow Q_r$ und somit auch $Q_u \sqsubseteq Q_r$.

Beispiel 2: *Query Containment nicht hinreichend für Relevanzprüfung*

Sei die folgende registrierte Anfrage Q_r (in SQL) gegeben: `SELECT * FROM TABELLE1 WHERE A<50` Die „Prüfanfrage“ der Löschoperation `DELETE FROM TABELLE1 WHERE A>25 AND A<50` ist gegeben als Q_u mit `SELECT * FROM TABELLE1 WHERE A>25 AND A<50`. Offensichtlich gilt hier $Q_u \sqsubseteq Q_r$. Für den Fall, dass $Q_u(D)$ nun aber die leere Menge ist, besteht keine Relevanz dieser Änderung für Q_u.

Abbildung 2.3 (vgl. auch [RD98]) illustriert ein weiteres Problem, das auftritt, wenn die Relevanz von Änderungen durch Query Containment festgestellt werden soll. Wenn der geänderte Wertebereich und der Wertebereich, der durch Q_r abgedeckt wird, überlappen ($Q_r(D) \cap Q_u(D) \neq \emptyset \wedge Q_u(D) - Q_r(D) \neq \emptyset \wedge Q_r(D) - Q_u(D) \neq \emptyset$), besteht zwar keine direkte Enthaltenseinsbeziehung, jedoch eine Relevanz.

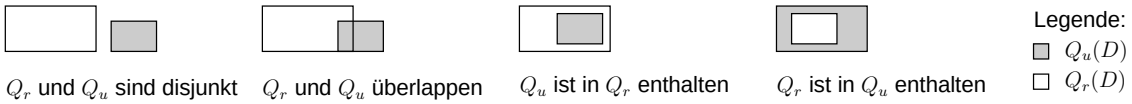


Abbildung 2.3: Zusammenhang zwischen Q_u und Q_r

Um auch den Fall der Überlappung korrekt als relevant erkennen zu können, muss auf Techniken des bereits oben angesprochenen semantischen Cachings zurückgegriffen werden. Mit den Mitteln des Query Containments ausgedrückt muss $(\mathcal{F} \sqsubseteq Q_r) \wedge (\mathcal{F} \sqsubseteq Q_u)$ erfüllt sein. Es muss also möglich sein, eine in Q_r als auch in Q_u enthaltene Anfragen $\mathcal{F}$ mit $\mathcal{F}(D) \neq \emptyset$ anzugeben, die den Schnitt berechnet. Algorithmen für das Finden einer derartigen Kompensationsanfrage basieren aber in der Regel auf zum Teil starken Einschränkungen der Anfragesprache. So werden in [DFJ+96] ausschließlich Selektionen zugelassen. In [LLS99, HS03c] besteht eine Beschränkung auf Selektionen und Projektion. Zusätzlich dazu wird in [RD98] noch der Gleichheitsverbund (EquiJoin) zugelassen.

Die im Rahmen dieser Dissertation verwendete Anfragerepräsentation (vgl. Kapitel 3) ermöglicht das Formulieren von zyklischen Anfragen und erlaubt die Verwendung des Vergleichsoperators $\neq$. Des Weiteren wird keine Beschränkung auf numerische Attribute vorgenommen. Insbesondere unter der Annahme, dass auf dem Server sehr viele registrierte Anfragen pro Änderung auf Relevanz zu prüfen sind, ist eine Relevanzprüfung über das Query Containment Problem nicht praktikabel. Der in dieser Dissertation vorgeschlagene Ansatz be-

trachtet die Anfragen nicht als ganzes, sondern schließt irrelevante Änderungen durch konjunktweises[11] Testen aus. Damit können Anfragen, die gleiche Konjunkte enthalten, gegebenenfalls frühzeitig gemeinsam ausgeschlossen werden.

2.3 Incremental View Update

Neben dem bereits in Abschnitt 2.1 angesprochenen Entwerten der Daten auf den mobilen Clients besteht auch die Möglichkeit, Änderungen direkt in diesen Datenbestand einzubringen. Die Datenbestände der Clients können als materialisierte Sichten über dem Serverdatenbestand interpretiert werden. Somit kann diese Aufgabe mithilfe von Techniken zur Sichtaktualisierung (engl. view maintanance) behandelt werden. Offensichtlich ist es insbesondere im mobilen Umfeld nicht sinnvoll, die materialisierten Sichten auf den Clients durch zyklisches Neuausführen der Anfragen neu zu berechnen. Stattdessen sollten Änderungen inkrementell in den Datenbestand eingepflegt werden. Nach [BCL89] kann das generelle Sichtaktualisierungsproblem in zwei Teilaufgaben unterteilt werden. Zuerst muss festgestellt werden, welche Sichten von einer Änderung betroffen sind. Anschließend muss die Änderung in die betroffenen Sichten integriert werden. Ein Überblick über Techniken zur Integration der Updates ist in [GMS93a] und in [Vis97] zu finden. Der Fall der Materialisierung der Sichten auf den mobilen Clients ist Gegenstand von [LLS00]. Für das Thema der Dissertation ist es notwendig, effizient festzustellen, *ob* eine Änderungsoperation eine materialisierte Sicht verändert. Dafür ist es aber nicht zwingend notwendig zu ermitteln, wie die Änderungen tatsächlich „aussehen“ und wie sie in die Sicht integriert werden können. Daher wird im Folgenden auf den ersten Schritt fokussiert.

Bestimmung logisch irrelevanter Updates

Um festzustellen, welche Sichten von einer Änderung betroffen sind, werden in der Literatur i. d. R. Techniken verwendet, die auf dem Query Containment Problem (vgl. 2.2) bzw. dem grundlegenden Problem der Erfüllbarkeit konjunktiver Anfragen [RH80, GSW96a, GSW96b] basieren.

In [MU83] werden ausschließlich Einfüge- und Löschoperation auf horizontalen Datenbankfragmenten betrachtet. Im Kontext materialisierter Sichten bedeutet dies, dass die Fragmente Sichten über einer Relation sind, wobei die Sichtdefinition nur Selektionen enthalten darf. Sei $\mathcal{C}$ die Selektionsbedingung einer Sichtdefinition E. Das Einfügen oder Löschen eines Tupels t ist somit für E nur relevant, wenn $\mathcal{C}(t) =$ *true* gilt.

In [BCL89, BCL98] wird zwischen den drei Änderungsoperationen INSERT, DELETE und MODIFY unterschieden, die wie folgt definiert sind:

INSERT(R_u**,** T**):** Die Menge von Tupeln T wird in die Relation $r(R_u)$ eingefügt.

[11] im Folgenden als prädikatweises bezeichnet

DELETE(R_u, $\mathcal{C}_D$): Die Tupel in $r(R_u)$, die die Bedingung $\mathcal{C}_D$ erfüllen, werden aus $r(R_u)$ gelöscht.

MODIFY(R_u, $\mathcal{C}_M$, $\mathbf{F}_M$): Alle Tupel in $r(R_u)$, die die Bedingung $\mathcal{C}_M$ erfüllen, werden anhand der Menge von Funktion $\mathbf{F}_M = \{f_{A_1}, \ldots, f_{A_n}\}$ verändert.

Des Weiteren sei die materialisierte Sicht $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$ mit der Attributmenge $\mathbf{A}$, der Relationenmenge $\mathbf{R}$ und der Menge der Bedingung $\mathcal{C}$ gegeben. $\mathcal{C}$ beschreibt sowohl die Selektionsbedingungen als auch die Verbundbedingungen, wobei Verbunde auf den Gleichheitsverbund beschränkt sind. Darüberhinaus werden Selbstverbunde ausgeschlossen. Ein INSERT ist somit für eine Sicht E logisch irrelevant, wenn $R_u \notin \mathbf{R}$ gilt oder alle Tupel $t \in T$ analog zu [MU83] die Bedingung $\mathcal{C}$ nicht erfüllen. Ein DELETE ist für E logisch irrelevant, wenn ebenfalls $R_u \notin \mathbf{R}$ gilt oder $\mathcal{C}_D \wedge \mathcal{C}$ unerfüllbar ist. Eine MODIFY-Operation ist für E logisch irrelevant, wenn eine der folgenden beiden Bedingungen [BCL98][12] erfüllt ist: (1) Die betroffenen Tupel sind weder vor der Modifikation noch danach in E repräsentiert. (2) Die betroffenen Tupel sind sowohl vor als auch nach der Modifikation in E repräsentiert, die Modifikation betrifft aber Attribute, die in E nicht sichtbar sind. Formal betrachtet ist eine Modifikation irrelevant, wenn $R_u \notin \mathbf{R}$ gilt oder

$$\forall[\mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \rightarrow \underbrace{(\neg\mathcal{C} \wedge \neg\mathcal{C}(\mathbf{F}_M))}_{\text{(D1)}} \vee \underbrace{(\overbrace{\mathcal{C}}^{\text{(K1)}} \wedge \overbrace{\mathcal{C}(\mathbf{F}_M)}^{\text{(K2)}} \wedge \overbrace{(\bigwedge_{A_i \in \mathbf{A} \cap \alpha(R_u)} (A_i = \rho(f_{A_i})))}^{\text{(K3)}})}_{\text{(D2)}}]$$

erfüllbar ist, wobei der Allquantor für alle auftretenden Variablen in der Formel, also alle betrachteten Attribute gilt. Um diese Formel zu analysieren, müssen noch einige zusätzlich verwendete Ausdrücke erläutert werden:

- $\alpha(R_u)$ ist die Menge der Attribute in R_u.
- $\rho(f_{A_i})$ ist der rechte Teil der Funktion f_{A_i}. Beispiel: Wenn f_{A_i} als $A_i = A_j + c$ gegeben ist, ist $\rho(f_{A_i}) = A_j + c$.
- $\mathcal{C}(\mathbf{F}_M)$ ist die Bedingung $\mathcal{C}$ die auf die geänderten Tupel angewendet wird. Beispiel: Seien $\mathcal{C} = (H > 30) \wedge (I = J)$ und die drei Abbildungen $f_H := (H = H + 20)$, $f_I := (I = 15)$ und $f_J := (J = J)$ gegeben. Dann ist $\mathcal{C}(\mathbf{F}_M) := (H + 20 > 30) \wedge (15 = J)$.
- $\mathcal{C}_B(\mathbf{F}_M)$ beschreibt die zulässigen Wertebereiche der jeweiligen Variablen. Beispiel: Sein die Wertebereiche $[0, 50]$, $[10, 100]$, $[10, 100]$ für die Variablen H, I und J gegeben. Damit ergibt sich $\mathcal{C}_B(\mathbf{F}_M)$ als $(H + 20 \geq 0) \wedge (H + 20 \leq 50) \wedge (15 \geq 10) \wedge (15 \leq 100) \wedge (J \geq 10) \wedge (J \leq 100)$

[12] Im ansonsten äquivalenten Papier in [BCL89] sind diese Bedingungen falsch formuliert, wurden jedoch in [BCL98] korrigiert.

Nach der Logik (vgl. [Wag03]) ist eine Implikation *true*, wenn die Prämisse nicht erfüllt ist oder sowohl Prämisse als auch Konklusion erfüllt sind. Hier ist die Prämisse nur erfüllt, wenn sowohl gilt, dass Tupel zu Änderung selektiert werden, als auch, dass die Änderung innerhalb der zugelassenen Wertebereiche der jeweilige Attribute liegen. Anderenfalls kann keine Änderung erfolgen, weshalb die Modifikation als irrelevant erkannt wird. Ist die Prämisse der Implikation erfüllt, muss deren Konklusion analysiert werden. Diese besteht hier aus den beiden Disjunkten (D1) und (D2). Ist eines dieser beiden Disjunkte erfüllt, so ist auch die Implikation erfüllt. (D1) ist erfüllt, wenn die Modifikation Wertebereiche der Attribute betrifft, die nicht in der Sicht enthalten sind ($\neg\mathcal{C}$) und wenn Tupel nach der Modifikation auch nicht in die Sicht einfließen ($\neg\mathcal{C}(\mathbf{F}_M)$). In diesem Fall ist die Änderung garantiert irrelevant. Das zweite Disjunkt (D2) setzt sich aus drei Konjunkten zusammen, die alle erfüllt sein müssen, damit auch (D2) erfüllt ist. (K1) und (K2) fordern, dass sowohl der modifizierte Wertebereich als auch der geänderte Wertebereich in E enthalten sind. In diesem Fall ist die Modifikation nur dann irrelevant, wenn die Änderung auf Attributen erfolgt, die nicht in E repräsentiert sind. Diese Eigenschaft prüft (K3).

Neben diesem Ansatz wird in [Elk90] die Irrelevanz von Änderungen für logische Datenbankanfragen diskutiert. Hierbei werden Selbstverbunde und Negationen ausgeschlossen und ausschließlich Einfüge- und Löschoperationen betrachtet. Es wird aber u. A. gezeigt, dass wenn zwei Teilanfragen Q_1^1 und Q_1^2 unabhängig von einer Änderung U sind, dann auch $Q_1^1 \wedge Q_1^2$ und $Q_1^1 \wedge \neg Q_1^2$ unabhängig sein müssen. Diese Eigenschaft wird in Kapitel 4 genutzt, indem die Anfragen in Teilanfragen (Prädikate) zerlegt werden und anschließend prädikatweise auf Relevanz geprüft werden. Die Prüfung der Irrelevanz einer Änderung für eine Anfrage wird hierbei erneut mittels der Unerfüllbarkeit eines logischen Ausdrucks durchgeführt. Im Gegensatz dazu wird in [LS93] die Äquivalenz von Datalogprogrammen benutzt, um die Irrelevanz nachzuweisen. Demnach ist ein Datalogprogramm zum Einfügen von Daten Q_u für ein Datalogprogramm zum Anfragen von Daten Q_r irrelevant (notiert als $In^+(Q_r, Q_u)$), wenn $Q \equiv Q^+$ gilt. Q^+ ist hierbei ein neu konstruiertes Datalogprogramm, welches sowohl Q_r als auch die neu eingefügten Daten „abdeckt“. Analog dazu ist das Vorgehen bei Löschoperationen wobei dann $In^-(Q_r, Q_u) \leftrightarrow Q_r \equiv Q^-$ gilt. Wie bereits in Abschnitt 2.2 diskutiert, ist aber das Testen des Enthaltenseins einer logischen Datenbankanfrage in einer anderen unentscheidbar. Somit ist auch die Äquivalenzprüfung im allgemeinen Fall unentscheidbar. In [LS93] werden daher zwei Einschränkungen diskutiert, die eine solche Äquivalenzprüfung zulassen:

1. $In^+(Q_r, Q_u)$ und $In^-(Q_r, Q_u)$ sind entscheidbar, wenn sowohl Q^+ bzw. Q^- als auch Q_r ausschließlich Vergleiche der Form $x\phi y$ und $x\phi c$ mit $\phi \in \{<, \leq, =, \neq, \geq, >\}$ benutzen und sich nicht in den intensionalen Prädikaten unterscheiden.

2. Sowohl $In^+(Q_r, Q_u)$ als auch $In^-(Q_r, Q_u)$ sind entscheidbar, wenn Q_r ausschließlich Vergleiche der Form $x\phi y$ und $x\phi c$ mit $\phi \in \{<, \leq, =, \neq, \geq, >\}$ und keine Projektionen benutzt und, wenn Q_u nur Regeln der Form $u(X_1, \ldots, X_k) :- e_1(X_1, \ldots, X_k), c$ enthält. Hierbei muss c eine Konjunktion von Vergleichen der Form $x\phi y$ oder $x\phi c$ sein.

Der Algorithmus, welcher nun zum Prüfen der Äquivalenz benutzt wird, basiert auf dem bereits in [Sag87a, Sag87b] publizierten Algorithmus und benutzt das uniforme Query Con-

tainment (vgl. Abschnitt 2.2) um die analog dazu definierte uniforme Äquivalenz zu testen. Eine Alternative hierzu ist das Prüfen der Anfrageerreichbarkeit (engl. query reachability). Nach [LS93] gelten sowohl $In^+(Q_r, Q_u)$ als auch $In^-(Q_r, Q_u)$, wenn weder Q_r noch Q_u Negationen enthalten und das Prädikat u nicht durch Q^+ erreichbar ist. Algorithmen um diese Eigenschaft zu testen, können u. A. in [LMSS93] und [LS92] gefunden werden. Die Idee hierbei ist, dass Problem der Anfrageerreichbarkeit auf das Erfüllbarkeitsproblem abzubilden. In [LMSS93] wird nachgewiesen, dass beide Probleme ineinander überführt werden können.

2.3.1 Abgrenzung

Der Hauptunterschied zwischen den diskutierten Verfahren zum Prüfen der Irrelevanz von Änderungen für Sichten und dem Thema dieser Dissertation liegt darin, dass hier eine große Anzahl von Sichten zu prüfen ist. Techniken, die jede Änderung sequentiell auf Relevanz für alle Sichten testen, sind in derartigen Szenarien analog zu Abschnitt 2.2.2 ineffizient. Erschwerend kommt hinzu, dass die in dieser Dissertation verwendete Anfragenotation (vgl. Kapitel 3) dazu führt, dass die meisten Annahmen, die in den bisherigen Techniken gemacht wurden, nicht erfüllt sind.

2.4 Publish-Subscribe-Systeme

Oberflächlich betrachtet ähnelt das in Abschnitt 1 eingeführte und dieser Arbeit zugrundeliegende Szenario einem typischen Einsatzgebiet von Subskriptionssystemen [Leh02]. Diensterbringer publizieren Nachrichten, welche diejenigen Dienstnutzer empfangen, die eine entsprechende Subskription registriert haben. Diensterbringer und Dienstnutzer werden hierbei logisch entkoppelt und durch eine Vermittlungskomponente ergänzt. Abbildung 2.4 illustriert das Kommunikationsmuster im Subskriptionsmodell wobei die folgenden fünf Dienstprimitive verwendet werden:

register: Bei der Registrierung eines Diensterbringers mittels *register* gib dieser sich und den durch ihn angebotenen Dienst der Vermittlungskomponente bekannt.

publish: Zustandsänderungen auf der Seite eine Diensterbringers werden durch diesen der Vermittlungskomponente mittels *publish* bekannt gegeben.

inquire: Aus Sicht der Dienstnutzer ist es notwendig, feststellen zu können, welche Dienterbringer, Dienste bzw. Nachrichtenschemata über die Vermittlungskomponente genutzt werden können. Hierzu dient *inquire*.

subscribe: Mittels *subscribe* registriert ein Dienstnutzer eine Subskription, die festlegt, welche Zustandsänderungen für den Dienstnutzer von Interesse sind.

notify: Die Bekanntgabe von Zustandsänderungen an die Diensterbringer erfolgt durch die Vermittlungskomponente durch *notify*.

Es kann festgehalten werden, dass die Primitive *register*, *inquire* und *notify* auf die Primitive *publish* und *subscribe* abgebildet werden können (vgl. [Leh02, Seiten 31–32]). Wird das hier

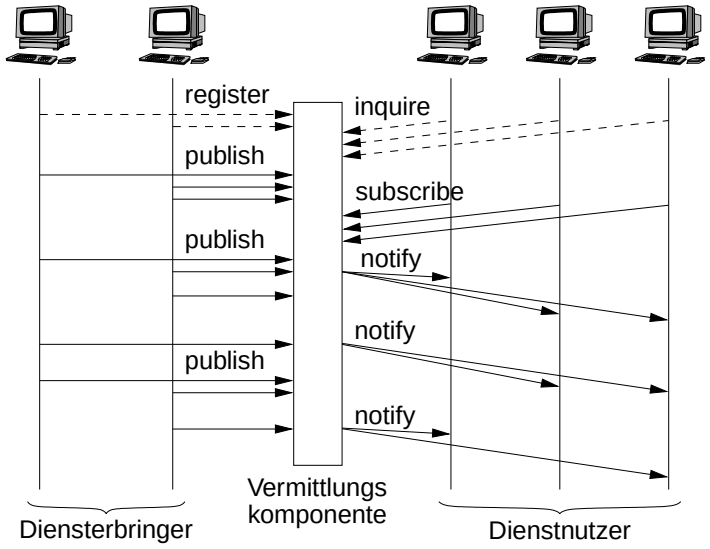


Abbildung 2.4: Kommunikationsmuster im Subskriptionsmodell [Leh02, Seite 31]

verwendete Szenario (vgl. Abbildung 1.1 auf Seite 18) auf ein derartiges Subskriptionssystem abgebildet, so kann das relationale DBMS als Diensterbringer, die mobilen Clients als Dienstnutzer und SMoS als Vermittlungskomponente interpretiert werden. Das Registrieren einer Anfrage entspricht demnach einem *subscribe* und das Benachrichtigen eines mobilen Clients über relevante Änderungen ist mithilfe von *publish/notify* realisierbar.

SMoS dient hier aber nicht nur als Vermittlungskomponente, sondern führt auch die Relevanzprüfung durch. Diese Aufgabe entspricht dem Feststellen einer Zustandsänderung bei einem Diensterbringer. Ferner greift SMoS bei der Relevanzprüfung auf den aktuellen Datenbankzustand zurück, wodurch die Trennung von Diensterbringer und Vermittlungskomponente hier nicht tragfähig ist. Würde dieses Szenario als Subskriptionssystem realisiert werden, so müsste der Update-Manager in den Diensterbringer transferiert werden. Um die Relevanzprüfung dann auf Diensterbringerseite durchführen zu können, müsste der Indexmanager sowohl im Diensterbringer als auch in der Verwaltungskomponente verfügbar sein. Eine Alternative bestünde darin, die Datenbank in die Verwaltungskomponente zu replizieren. Beide Ansätze bedeuten aber, dass künstlich Redundanzen eingefügt werden müssen. Eine dritte Alternative besteht darin, dass der Diensterbringer als Nachricht bei einer Zustandsänderung die von einer Änderung betroffenen Client-IDs liefert. Um in der Verwaltungskomponente nun die betroffenen Clients zu benachrichtigen, müssten diese nicht ihre Anfragen, sondern eine Bedingung der Form *if MyID∈Nachricht then notify_me* registriert haben. Ein derartiges Vorgehen setzt aber voraus, dass die eigentlichen Anfragen des Clients von der Verwaltungskomponente an den Diensterbringer weitergeleitet werden müssten, was der Entkopplung von Diensterbringer und Dienstnutzer widerspricht.

Kapitel 3

Anfragerepräsentation

A hundred years from now, I'm quite sure, database systems will still be based on Codd's relational foundations.
C. J. Date in [Dat01]

Anfragen in mobilen Informationssystemen werden in der Regel durch Applikationen erzeugt und nicht explizit durch den Nutzer eingegeben. Daher ist der Einsatz einer deskriptiven Anfragesprache wie SQL nicht zwingend notwendig. Stattdessen können Anfragen bereits bei ihrer Erzeugung derart repräsentiert werden, dass sie ohne großen Konvertierungsaufwand zur Indexierung auf dem Server benutzt werden können.

Die Betrachtungen in dieser Dissertation basieren auf dem Relationenmodell, welches in Abschnitt 3.1 kurz wiederholt wird. Anschließend wird in Abschnitt 3.2 die in dieser Arbeit verwendete kalkülähnliche Anfragenotation eingeführt.

3.1 Formale Definition des Relationenmodells

Im bereits 1969 von Codd [Cod69] eingeführten und ein Jahr später in [Cod70] publizierten Relationenmodell wird der Begriff *Relation* wie folgt definiert: Gegeben seien die Mengen von Wertebereichen S_1, S_2, $\ldots S_n$ mit $n \in \mathbb{N}$. Eine Relation R ist nun Teilmenge des Kartesischen Produkts $R \subseteq S_1 \times S_2 \times \cdots \times S_n$. In der gängigen Datenbankliteratur (z. B. [Ull82, Mai83, HS00, EN02]) wird eine abgewandelte, allgemeinere Definition des Relationenmodells gegeben. Es wird aus der nicht-leeren, endlichen Menge $\mathcal{U}$, dem *Universum* der Attribute, hergeleitet. Demnach ist ein *Attribut* $A \in \mathcal{U}$ Element des Universums. Der Wertebereich eines Attributs wird als die total definierte Funktion $\mathsf{dom} : \mathcal{U} \rightarrow \mathcal{D}$ festgelegt. Hierbei gilt, dass $\mathcal{D} = \{D_1, \ldots, D_m\}$ mit $m \in \mathbb{N}$ eine Menge von Wertebereichen D_i mit $1 \leq i \leq m$ ist. Der Wertebereich eines Attributes A wird als $\mathsf{dom}(A)$ angegeben womit für einen entsprechenden *Attributwert* w gelten muss, dass $w \in \mathsf{dom}(A)$. Ein *Relationenschema* ist eine Teilmenge $R \subseteq \mathcal{U}$ des Universums. Damit ergibt sich eine Relation r über einem Relationenschema $R = \{A_1, \ldots A_n\}$ mit $1 \leq n \leq m$ (kurz $r(R)$) als endliche Menge von Abbildungen der Form $t : R \rightarrow \bigcup_{i=1}^{m} D_i$. Eine derartige Abbildung t wird auch *Tupel* genannt. Dabei gilt

$t(A) \in \mathsf{dom}(A)$. Da t eine Abbildung ist, kann auch ihr Wertebereich eingeschränkt werden. Für die Teilmenge $X \subseteq R$ wird diese Einschränkung als $t_{|X}$ angegeben.

Im Gegensatz zur Definition von Codd wird somit vermieden, dass die Reihenfolge der Attribut-Attributwert-Kombinationen einer Relation festgeschrieben ist.

Für die Betrachtungen in den folgenden Abschnitten wird eine Erweiterung dieser formalen Definition des Relationenmodells benötigt, die den Zugriff auf den Namen einer Relation ermöglicht. Analog zu der Definition des Relationenmodells in [AHV96] wird daher die Konvention eingeführt, dass Relationenschemata eindeutig benannt werden müssen. Der Name einer Relation wird eindeutig durch `name` in $r(R_{\texttt{name}})$ festgelegt. Aus praktischer Sicht sei darauf hingewiesen, dass nicht gefordert wird, dass alle Relationschemata R_i mit $i \in \mathbb{N}$ und $1 \leq i \leq |\mathcal{U}|$ innerhalb des Universums $\mathcal{U}$ auch auf unterschiedlichen Attributen beruhen müssen. Folglich können mehrere Relationenschemata existieren, die zwar die gleiche Attributmenge, nicht jedoch den gleichen Namen verwenden.

Aktuelle relationale und objektrelationale Datenbankmanagementsysteme fassen Relationen nicht als Mengen im strengen mathematischen Sinne auf. Stattdessen können Relationen hier Duplikate enthalten. Im Rahmen dieser Dissertation wird auf die strikte, mengenorientierte Definition des Relationenmodells zurückgegriffen. Zwar können die im Folgenden verwendeten Algebraoperatoren auch für Multimengen definiert werden [DGK82], dennoch werden insbesondere bei den Relevanztests in Kapitel 4 Annahmen getroffen, welche bei zugelassenen Duplikaten nicht mehr gelten.

3.2 Repräsentation von Anfragen

Basierend auf dem im vorigen Abschnitt wiederholten Relationenmodell wird in diesem Abschnitt die in der vorliegenden Dissertation verwendete Notation von Anfragen vorgestellt. Diese enthält sowohl algebraische Elemente als auch Annahmen, die aus Kalkülen bekannt sind. Um diese zu verdeutlichen, wird die Semantik der jeweiligen Prädikate anhand der Relationenalgebra [Cod72, Cod91] und des Bereichskalküls [Cod72] erläutert.

Analog zum Bereichskalkül wird eine Anfrage als logischer Ausdruck dargestellt, der sich aus konjunktiv verknüpften Prädikaten zusammensetzt. Eingeführt wurde die verwendete Notation erstmals im Rahmen von Arbeiten zu semantischen Caches in [HS03c]. In [HSS04b] wurde sie um Verbundprädikate erweitert. Die dabei vorgenommene Beschränkung auf Gleichheitsverbunde wird hier aufgehoben, sodass typische SPJ-Ungleichheitsanfragen (engl. SPJ queries with inequalities) dargestellt werden können. Hierzu werden **S**elektionen durch Selektionsprädikate, **P**rojektionen durch Projektionsprädikate[1] und Verbunde (engl. **J**oin) durch Verbundprädikate repräsentiert. Syntax und Semantik dieser Prädikate sind wie folgt definiert:

Definition 1: *Selektionsprädikat*

Selektionsprädikate sind, analog zum Selektionsoperator σ der Relationenalgebra, Tu-

[1] in [HS03c] und [HSS04b] Relationsprädikate genannt

pelmengenbeschränkungen der Ergebnisrelation. Sei $\sigma_{\mathsf{scond}}(r(R_{\mathsf{name}}))$ eine Selektion mit der *konjunktiven* Selektionsbedingung scond, welche eine Menge von Konstantenselektionen der Form $s = A\ \gamma\ k$ oder Attribut-Selektionen der Form $s = A\ \gamma\ B$ mit $\gamma \in \{\leq, <, =, \neq, >, \geq\}$, $A, B \in R_{\mathsf{name}}$ und $k \in \mathsf{dom}(A)$ ist. Dann kann $\sigma_{\mathsf{scond}}(r(R_{\mathsf{name}}))$ als Menge von Selektionsprädikaten der Form $\{[\mathsf{name}.s_1], \ldots, [\mathsf{name}.s_n]\}$ mit $n \in \mathbb{N}$ und $n = |\mathsf{scond}|$ geschrieben werden wobei für alle s_i mit $1 \leq i \leq n$ gilt: $s_i \in \mathsf{scond}$. Die konjunktive Verknüpfung der einzelne Selektionen wird aufgrund der Kommutativität des logischen Und[2] implizit durch die konjunktive Verknüpfung der Selektionsprädikate in der Gesamtanfrage (siehe Definition 4) aufrecht erhalten.

Im Vergleich mit dem Bereichskalkül gilt für Selektionsprädikate die Konventionen, dass auch die Attribute der betroffenen Relation, die nicht zur Beschränkung des Ergebnisses herangezogen werden, als existenzquantifiziert angenommen werden.

Beispiel 3: *Selektionsprädikate*

Gegeben sei die Relation `Filme (FID, Name, FSK, Genre, Darsteller, Kurzinfo)`. Ein Beispiel für Anfragen mit einer Tupelmengenbeschränkung ist die Selektion aller Action-Filme mit einer Altersbeschränkung größer als 16 Jahre:

Relationenalgebra:	$\sigma_{\mathtt{FSK}>16 \wedge \mathtt{Genre}=\prime\mathtt{Action}\prime}(r(R_{\mathtt{Filme}}))$
Bereichskalkül:	$\{a, b, c, d, e, f \mid \mathtt{Filme}(a, b, c, d, e, f) \wedge c > 16 \wedge d =\prime \mathtt{Action}\prime\}$
Selektionsprädikat:	$\{[\mathtt{Filme.FSK} > 16], [\mathtt{Filme.Genre} =\prime \mathtt{Action}\prime]\}$

Definition 2: *Verbundprädikat*

Verbundprädikate repräsentieren, analog zum θ-Join der Relationenalgebra, das tupelweise Kombinieren zweier Relationen anhand von Verbundbedingungen. Sei $r(R_{\mathsf{name}_1}) \bowtie_{\mathsf{jcond}} r(R_{\mathsf{name}_2})$ ein θ-Join mit der *konjunktiven* Verbundbedingung jcond, welche eine Menge von Bedingungen der Form $j = A\ \theta\ B$ mit $\theta \in \{\leq, <, =, \neq, >, \geq\}$, $A \in R_{\mathsf{name}_1}$ und $B \in R_{\mathsf{name}_2}$ ist. Dies wird als Verbundprädikat der Form $[\mathsf{name}_1, \mathsf{name}_2, (j'_1, \ldots, j'_n)]$ mit $n \in \mathbb{N}$ und $1 \leq n \leq |\mathsf{jcond}|$ geschrieben. Dabei wird eine aufsteigende lexikographische Ordnung der Relationennamen gefordert. Des Weiteren werden die Bedingungen j'_i mit $i \in \mathbb{N}$ und $1 \leq i \leq n$ analog zur Relationenalgebra, aber mit der Konvention, dass die Attribute eindeutig durch Vorstellen des Namens des jeweiligen Relationenschemas identifiziert werden, gebildet. Eine Bedingung in der Relationenalgebra $j_i = A\ \theta\ B$ mit $\theta \in \{\leq, <, =, \neq, >, \geq\}$, $A \in R_{\mathsf{name}_1}$ und $B \in R_{\mathsf{name}_2}$ wird innerhalb des Verbundprädikates als $j'_i = \mathsf{name}_1.A\ \theta\ \mathsf{name}_2.B$ geschrieben.

Analog zu Selektionsprädikaten werden bei Verbundprädikaten nicht angegebene Attribute der betroffenen Relationen - im Vergleich zum Bereichskalkül - als existenzquantifiziert angenommen.

[2] siehe u. a. Optimierungsregeln der logischen Optimierung in [SH99]

Beispiel 4: *Verbundprädikate*

Gegeben seien die Relation aus Beispiel 3 und die Relation `laeuft_in(Datum, Uhrzeit, SID, FID)`. Ein Beispiel für den Verbund dieser beiden Relationen könnten offensichtlich über die gleichnamigen Attribute `FID` erfolgen:

Relationenalgebra	$r(R_{\texttt{Filme}}) \bowtie_{\texttt{FID=FID}} r(R_{\texttt{laeuft\_in}})$
Bereichskalkül:	$\{a, b, c, d, e, f, g, h, i, j \mid$
	$\texttt{Filme}(a, b, c, d, e, f) \wedge \texttt{laeuft\_in}(g, h, i, j) \wedge a = j\}$
Verbundprädikat:	$[\texttt{Filme}, \texttt{laeuft\_in}, (\texttt{Filme.FID} = \texttt{laeuft\_in.FID})]$

Im Bereichskalkül könnte der Gleichheitsverbund auch als $\{a, b, c, d, e, f, g, h, i, a \mid \texttt{Filme}(a, b, c, d, e, f) \wedge \texttt{laeuft\_in}(g, h, i, a)\}$ geschrieben werden. Da die Verbundprädikate aber anhand des θ-Joins definiert sind, wurden „kompatible“ Notationen angegeben.

Selbstverbund

Eine häufig vorkommende Datenbankoperation ist der Selbstverbund einer Relation. Da SQL[3] es nicht zulässt, dass gleiche Tabellennamen mehrfach im `FROM`-Teil vorkommen, muss eine Umbenennung von Tabellennamen möglich sein. Ein solcher Alias wird als `name@alias` notiert. Selbstverständlich ist die Verwendung von Umbenennungen in allen drei Prädikattypen zulässig.

Definition 3: *Projektionsprädikat*

Projektionsprädikate stellen, analog zum Projektionsoperator π der Relationenalgebra, eine Attributmengeneinschränkung der Ergebnisrelation dar. Eine Projektion $\pi_X(r(R_{\texttt{name}}))$ mit $X \subseteq R_{\texttt{name}}$ wird als Projektionsprädikat $[\texttt{name}(x_1, \ldots, x_n)]$ mit $n \in \mathbb{N}$, $1 \leq n \leq |X|$ und $\{x_1, \ldots, x_n\} = X$ dargestellt. Projektionen auf Verbundergebnissen der Form $\pi_{X_1, X_2, \ldots, X_i}(r(R_{\texttt{name}_1}) \bowtie_{\theta_1} r(R_{\texttt{name}_2}) \bowtie_{\theta_2} \cdots \bowtie_{\theta_{i-1}} r(R_{\texttt{name}_i}))$ mit $i, j \in \mathbb{N}, 1 \leq j \leq i$ und $X_j \subseteq R_{\texttt{name}_j}$, $1 \leq n_j \leq |X_j|$, $\{x_1^j, \ldots, x_{n_j}^j\} = X_j$ werden als Projektionsprädikat $[\texttt{name}_1(x_1^1, \ldots, x_{n_1}^1), \texttt{name}_2(x_1^2, \ldots, x_{n_2}^2), \ldots, \texttt{name}_i(x_1^i, \ldots, x_{n_i}^i)]$ notiert.

Im Bereichskalkül werden Attribute durch „_“ notiert, wenn sie als existierend angenommen werden, aber ihr Wert für das Ergebnis keine Bedeutung hat, da die Projektion sie „ausblendet“. In Projektionsprädikatschreibweise wird dies implizit für alle nicht enthaltenen Attribute der betroffenen Relation angenommen.

[3] Anfragen müssen zur Ausführung nach SQL umgesetzt werden.

Beispiel 5: *Projektionsprädikate*

Geben sei die Relation 4. Ein Beispiel für Anfragen mit einer Attributmengeneinschränkung ist die Projektion auf `FSK`, `Genre` und `Name`, die wie folgt notiert wird:

Relationenalgebra:	$\pi_{\texttt{Name,Genre,FSK}}(r(R_{\texttt{Filme}}))$
Bereichskalkül:	$\{x, y, z \mid \texttt{Filme}(\_, x, z, \_, \_, y)\}$
Projektionsprädikat:	$[\texttt{Filme}(\texttt{FSK}, \texttt{Genre}, \texttt{Name})]$

Ein Beispiel für eine Anfragen, die eine Attributmengeneinschränkung auf einem Verbundergebnis vornimmt, ist die Projektion von `Name`, `Datum` und `Uhrzeit` auf dem Verbundergebnis aus Beispiel 4:

Relationenalgebra:	$\pi_{\texttt{Name,Datum,Uhrzeit}}(r(R_{\texttt{Filme}}) \bowtie_{\texttt{FID=FID}} r(R_{\texttt{laeuft\_in}}))$
Bereichskalkül:	$\{b, g, h \mid \texttt{Filme}(a, b, c, d, e, f) \wedge \texttt{laeuft\_in}(g, h, i, j) \wedge a = j\}$
Verbundprädikat:	$[\texttt{Filme}, \texttt{laeuft\_in}, (\texttt{Filme.FID} = \texttt{laeuft\_in.FID})]$
Projektionsprädikat:	$[\texttt{Filme}(\texttt{Name}), \texttt{laeuft\_in}(\texttt{Datum}, \texttt{Uhrzeit})]$

Eine genaue Definition von Anfragen mithilfe von Prädikaten folgt in Definition 4.

Seien PP die Menge aller Projektionsprädikate, VP die Menge aller Verbundprädikate und SP der Verbund aller Selektionsprädikatmengen. Mithilfe der Kommutativität des logischen Und werden Datenbankanfragen nun durch konjunktive Verknüpfung der Elemente dieser Mengen repräsentiert, wobei eine lexikographische Ordnung gefordert wird.

Definition 4: *Prädikatensequenzanfrage (PSQ)*
Mit $V \subseteq VP$, $pp \in PP \cup \{\varepsilon\}$, $S \subseteq SP$ und $V \cup \{pp\} \cup S \neq \emptyset$ wird eine konjunktive Anfrage $Q = \bigwedge_{vp \in V} vp \wedge \bigwedge_{sp \in S} sp \wedge pp$ als Prädikatensequenzanfrage $Q' = \langle vp_1 \ \ldots \ vp_n \ sp_1 \ \ldots \ sp_o \ pp \rangle$ mit

- $\forall i, k \in \mathbb{N}, 1 \leq i < k \leq n; vp_i, vp_k \in V \cup \{\varepsilon\} \Rightarrow vp_i \triangleleft vp_k$ sowie
- $\forall i, k \in \mathbb{N}, 1 \leq i < k \leq o; sp_i, sp_k \in S \cup \{\varepsilon\} \Rightarrow sp_i \triangleleft sp_k$

dargestellt. Hierbei steht $\triangleleft$ für „lexikographisch kleiner“.Das Vorkommen von Prädikaten in einer derartigen Sequenz unterliegt den folgenden Restriktionen:

(1) Alle Verbundprädikate müssen über die Relationennamen verbunden sein und es dürfen keine zwei Verbundprädikate existieren, welche die selben beiden Relationen betreffen. Formal kann dieses wie folgt ausgedrückt werden:
Sei eine Anfrage mit genau einem Verbundprädikat $vp_1 = [\texttt{name}_1^1, \texttt{name}_2^1, (X_1)]$ eine korrekte PSQ-Anfrage. Für korrekte PSQ-Anfragen mit n Verbundprädikaten $vp_1 = [\texttt{name}_1^1, \texttt{name}_2^1, (X_1)], \ldots, vp_n = [\texttt{name}_1^n, \texttt{name}_2^n, (X_n)]$ muss dann gelten, dass für alle Verbundprädikate $vp_i = [\texttt{name}_1^i, \texttt{name}_2^i, (X_i)]$ mit $i \in \mathbb{N}$ und $1 < i \leq n$ mindestens ein Verbundprädikat $vp_j = [\texttt{name}_1^j, \texttt{name}_2^j, (X_j)]$

mit $j \in \mathbb{N}$ und $1 \leq j < i$ existiert, sodass $\mathtt{name}_1^i = \mathtt{name}_1^j \vee \mathtt{name}_1^i = \mathtt{name}_2^j \vee \mathtt{name}_2^i = \mathtt{name}_1^j \vee \mathtt{name}_2^i = \mathtt{name}_2^j$. Des Weiteren darf kein Verbundprädikat $vp_j = [\mathtt{name}_1^j, \mathtt{name}_2^j, (X_j)]$ existieren, sodass $\mathtt{name}_1^i = \mathtt{name}_1^j \wedge \mathtt{name}_2^i = \mathtt{name}_2^j$ gilt.

(2) Wenn Verbundprädikate existieren, so dürfen gegebenenfalls sowohl Selektionsprädikate als auch das Projektionsprädikat nur Relationennamen verwenden, die bereits in einem der Verbundprädikate vorkommen!

(3) Wenn keine Verbundprädikate existieren, so müssen alle Selektionsprädikate auf dem selben Relationennamen beruhen!

(4) Wenn keine Verbundprädikate und keine Selektionsprädikate in der Anfragesequenz enthalten sind, so muss ein Projektionsprädikat mit genau einem verwendeten Relationennamen existieren.

Diese Notation impliziert auch die Reihenfolge in der die angegebenen Prädikate ausgeführt werden müssen. Zuerst müssen Verbunde berechnet werden, da sowohl Selektionen als auch Projektionen auf dem Verbundergebnis basieren. Anschließend müssen Selektionen ausgeführt werden. Dies beruht auf der Tatsache, dass Projektionen Attribute aus der Ergebnisrelation ausblenden können, die für die Selektion benutzt werden. In der Relationenalgebra wird eine korrekte Ausführungsreihenfolge der einzelnen Operatoren ebenfalls durch die richtige Anordnung und Verschachtelung der einzelnen Operatoren gegeben. Im Bereichskalkül wird dies bei der Interpretation des Anfrageausdrucks und durch Klammerung garantiert.

Beispiel 6: *Prädikatensequenzanfrage*

Gegeben seien die Relationen aus den Beispielen 5 und 4. Ein Beispiel für eine Datenbankanfrage, die sowohl Selektionen, Projektionen und Verbunde umfasst liefert beispielsweise die Anfrangszeiten von Actionfilmen mit einer Altersbeschränkung größer als 16 Jahre.

Relationenalgebra:	$\pi_{\mathtt{Uhrzeit,Name}}(\sigma_{\mathtt{FSK}>16 \wedge \mathtt{Genre}='\mathtt{Action}'}$ $(r(R_{\mathtt{Filme}}) \bowtie_{\mathtt{FID}=\mathtt{FID}} r(R_{\mathtt{laeuft\_in}})))$
Bereichskalkül:	$\{e, b \mid \exists a \exists b \exists c \exists d \exists e \exists f \mathtt{Filme}(a, b, c, d, \_, \_) \wedge \mathtt{laeuft\_in}(\_, e, \_, f)$ $\wedge f = a \wedge d =' \mathtt{Action}' \wedge c > 16\}$
PSQ-Anfrage:	$\langle[\mathtt{Filme}, \mathtt{laeuft\_in}, (\mathtt{Filme.FID} = \mathtt{laeuft\_in.FID})]$ $[\mathtt{Filme.FSK} > 16][\mathtt{Filme.Genre} =' \mathtt{Action}']$ $[\mathtt{Filme}(\mathtt{Name}), \mathtt{laeuft\_in}(\mathtt{Uhrzeit})]\rangle$

Offensichtlich führt die Konvention, dass die einzelnen Prädikate innerhalb ihrer Klasse lexikographisch geordnet sein müssen dazu, dass auch die Attributname in der Ergebnismenge dieser Ordnung folgen. Wie in den folgenden Kapiteln aber gezeigt wird, kann diese Ordnung genutzt werden, um effizienter festzustellen, welche Anfragen im Index von einer Änderung betroffen sind. Daher wird hier der Kompromiss eingegangen, dass der mobile Client bei Bedarf lokal eine Umordnung der Attribute eines Ergebnisses vornimmt.

3.2.1 Vollständigkeit und Sicherheit

Zwei zentrale Eigenschaften von kalkülartigen Anfragesprachen bzw. Anfragerepäsentationen sind deren relationale Vollständigkeit und ihre Sicherheit. Nach [HS00] ist ein Kalkül streng relational vollständig, wenn es für jeden Term τ der Relationenalgebra einen äquivalenten (sicheren) Ausdruck η des Kalküls gibt. Offensichtlich ist die PSQ-Notation nicht streng relational vollständig, da weder die Mengenoperationen Vereinigung, Differenz und Durchschnitt, noch das Kreuzprodukt enthalten sind. PSQ deckt also nur eine Teilmenge des Bereichskalküls ab. Aufgrund der Definition der Prädikate ist aber die Sicherheit von PSQ garantiert. Alle in Anfragen verwendete Attribute (bzw. Variablen) sind entweder durch eine Bedingung oder per Definition (existenzquantifiziert) an einen endlichen Wertebereich gebunden.

Wie bereits erwähnt, ist das Ziel dieser Anfragerepräsentation eine einfache Nutzbarkeit der Anfragen zur Indexierung der mobilen Klienten. Relationale Vollständigkeit spielt daher hier eine untergeordnete Rolle. In vielen Anwendungsszenarien mobiler Informationssysteme, ist die Anfragemächtigkeit der PSQ-Notation aber hinreichend.

3.2.2 Umsetzung von PSQ nach SQL

Serverseitig müssen die Anfragen sowohl zur Indexierung benutzt, als auch ausgeführt werden. In dem dieser Arbeit zugrundeliegenden Szenario wird davon ausgegangen, dass die Komponente des Servers, welche die Anfrage beantwortet, ein relationales[4] Datenbankmanagementsystem ist. Daher kann hierbei die Standardanfragesprache SQL benutzt werden. Im SQL99-Standard (siehe z. B. [DD97]) werden Anfragen (ohne die durch PSQ nicht unterstützten Gruppierungen und Sortierungen) in der folgenden Notation gestellt:

```
SELECT [ALL | DISTINCT] select_list
FROM table_name1 [,..., table_nameN]
[WHERE search_and_join_condition]
```

Die Überführung einer Anfrage aus der PSQ-Notation nach SQL erfolgt anhand der folgenden Konventionen:

1. Verbundprädikate werden im `WHERE`-Teil einer SQL-Anfrage reflektiert, wobei die betroffenen Relationennamen in die `FROM`-Liste aufgenommen werden.

2. Selektionsprädikate werden ebenfalls im `WHERE`-Teil einer SQL-Anfrage repräsentiert, wobei die betroffenen Relationennamen in die `FROM`-Liste aufgenommen werden müssen.

3. Projektionsprädikate werden in die `select_liste` aufgenommen. Die betroffenen Relationennamen fließen wiederum in die `FROM`-Liste ein. Sind keine Projektionsprädikate angegeben, wird „`*`“ in die `select_liste` aufgenommen.

[4] Die meisten kommerziellen, auf dem Relationenmodell basierenden, DBMS sind mittlerweile objektrelational (siehe u. A. [Tür03]). Für die Realisierung der hier diskutierten Techniken, genügen aber die relationalen Bestandteile.

4. Tabellenumbenennungen der Form `name@alias` werden im `FROM`-Teil der SQL-Anfrage durch `name as alias` ausgedrückt. In `WHERE`-Teil und der `select_liste` werden sie durch den `alias` substituiert.

Relationennamen bzw. Relationenname-Alias-Kombinationen, die mehrfach in der `FROM`-Liste auftauchen würden, werden nur einmal in diese aufgenommen. Beispiel 7 illustriert die Umsetzung von PSQ nach SQL an zwei Beispielanfragen.

Beispiel 7: *PSQ nach SQL*

Die Anfrage ohne Aliase aus Beispiel 6 wird wie folgt nach SQL überführt:

PSQ: $\langle$[`Filme`, `laeuft_in`, (`Filme.FID` = `laeuft_in.FID`)]
[`Filme.FSK` $>$ 16][`Filme.Genre` $=$' `Action`']
[`Filme`(`Name`)][`laeuft_in`(`Uhrzeit`)]$\rangle$

Schritt 1:
```
SELECT DISTINCT      FROM Filme, laeuft_in
WHERE Filme.FID=laeuft_in.FID
```
Schritt 2:
```
SELECT DISTINCT      FROM Filme, laeuft_in
WHERE Filme.FID=laeuft_in.FID AND Filme.FSK>16
AND Filme.Genre='Action
```
Schritt 3:
```
SELECT DISTINCT Filme.Name, laeuft_in.Uhrzeit
FROM Filme, laeuft_in WHERE Filme.FID=laeuft_in.FID
AND Filme.FSK>16 AND Filme.Genre='Action'
```

Enthält jedoch die Anfrage Aliase, so ist Schritt 4 noch notwendig:

PSQ: $\langle$[`Filme@F1`, `Filme@F2`, (`Filme@F1.FID` = `Filme@F2.FID`)]
[`Filme@F1.Name!='a@b'`][`Filme@F1`(`Name`), `Filme@F2`(`FSK`)]$\rangle$

Schritt 1:
```
SELECT DISTINCT      FROM Filme@F1, Filme@F2
WHERE Filme@F1.FID=Filme@F2.FID
```
Schritt 2:
```
SELECT DISTINCT      FROM Filme@F1, Filme@F2
WHERE Filme@F1.FID=Filme@F2.FID
AND Filme@F1.Name!='a@b'
```
Schritt 3:
```
SELECT DISTINCT Filme@F1.Name, Filme@F2.FSK
FROM Filme@F1, Filme@F2
WHERE DISTINCT Filme@F1.FID=Filme@F2.FID
AND Filme@F1.Name!='a@b'
```
Schritt 4:
```
SELECT DISTINCT F1.Name, F2.FSK
FROM Filme AS F1, Filme AS F2
WHERE F1.FID=F2.FID AND F1.Name!='a@b'
```

Kapitel 4

Relevanzprüfung von Änderungsoperationen

Ein naiver Ansatz, serverseitig zu prüfen welche gespeicherten Anfragen von einer Datenmanipulationsoperation betroffen sind besteht darin, die registrierten Anfragen als materialisierte Sichten zu speichern und zu vergleichen, welche Sichten nach der Ausführung einer Aktualisierungsoperation geändert vorliegen. Offensichtlich bedeutet das aber einen enormen Speicheraufwand. Eine erste Verbesserung wäre die Verwendung von Prüfsummenverfahren wie beispielsweise MD5 [Riv92]. Hierbei könnten statt der materialisierten Sichten nur Hash-Werte dieser gespeichert werden. Nach einer Aktualisierung der Datenbank müssten dann die Hash-Werte neu berechnet und mit den alten (vor der Aktualisierung) verglichen werden. Damit wird zwar der Speicheraufwand reduziert, aber die Neuberechnung muss für alle gespeicherten Anfragen erfolgen, auch wenn nur wenige Anfragen tatsächlich betroffen sind. Offensichtlich bedeutet dies einen nicht unerheblichen Berechnungsaufwand. Des Weiteren ist eine zentrale Eigenschaft von Hash-Verfahren, dass Kollisionen auftreten können. Dadurch kann der sehr unwahrscheinliche Fall, dass sich zwar das Anfrageergebnis jedoch nicht der Hash-Wert geändert hat, nicht ausgeschlossen werden.

Nachdem im vorigen Kapitel die PSQ-Notation von Anfragen eingeführt wurde, kann diese verwendet werden, um die Relevanz von Datenmanipulationsoperationen für mobile Clients zu prüfen. Hierzu wird nicht nach dem Einbringen einer Änderung geprüft, welche Veränderungen diese bewirkt hat. Vielmehr wird – soweit möglich – vor der Ausführung einer Änderungsoperation auf der Datenbank anhand der Prädikate geprüft, ob die Änderungsoperation einen Datenbereich manipulieren wird, der durch die jeweilige Anfrage abgedeckt ist. Eine Ausnahme bildet hierbei die Relevanzprüfung von Update-Operationen, welche die Materialisierung der geänderten Relation als temporäre Tabelle voraussetzt.

In Abschnitt 4.1 wird auf Notationskonventionen bei den hier betrachteten Änderungsoperationenen Einfügen, Löschen und Updaten eingegangen. In den folgenden Abschnitten 4.2, 4.3 und 4.4 wird darauf aufbauend untersucht, wie die Relevanz einer Änderungsoperation abhängig von deren Typ für eine registrierte PSQ-Anfrage ermittelt werden kann.

4.1 Notation von Änderungsoperationen

Im Rahmen dieser Dissertation wird davon ausgegangen, dass Datenmanipulationsoperationen auf dem Server in SQL (vgl. z. B. [DD97]) notiert werden. Hierbei wird zwischen den folgenden drei zulässigen Änderungsoperationen unterschieden:

Einfügen von Tupeln: In SQL werden Tupel mittels einer `INSERT`-Anweisung in eine Relation oder Sicht eingefügt. Der SQL99-Standard sieht hierbei die folgende Syntax vor:

```
INSERT [INTO] [[database_name.]owner.] {table_name
| view_name} [(column_list)] {[DEFAULT] VALUES |
VALUES (value [,...]) | SELECT_statement}
```

Für die folgenden Betrachtungen wird diese Notation vereinfacht und davon ausgegangen, dass Einfügeoperationen wie folgt notiert werden: `INSERT INTO name (column_list) VALUES (value [,...])`. Des Weiteren wird – ohne Beschränkung der Allgemeinheit – gefordert, dass `column_list` alle Attribute der entsprechenden Relation enthält. Darüberhinaus gilt die Einschränkung, dass je Einfügeoperation nur ein Tupel eingefügt wird. Einige Datenbankmanagementsysteme wie Oracle, DB2 oder MySQL unterstützen das gleichzeitige Einfügen mehrerer Tupel. Ein derartiges Vorgehen muss hier durch mehrere hintereinander ausgeführte Einfügeoperationen realisiert werden.

Löschen von Tupeln: Das Löschen von Tupeln aus einer Relation erfolgt in SQL99 mittels der `DELETE`-Anweisung:

```
DELETE [FROM] [owner.]table_name [WHERE clause]
```

Diese wird ebenfalls vereinfacht als `DELETE FROM name WHERE clause` angenommen wobei die Selektionsbedingung `clause` auf Konstantenselektion beschränkt wird.

Aktualisieren von Tupeln: Aktualisieren von Tupeln einer Relation ist in SQL mithilfe einer `UPDATE`-Anweisung der Form:

```
UPDATE {table_name | view_name} SET {column_name
| variable_name} = {DEFAULT | expression} [,...n]
WHERE condition
```

möglich. Analog zum Einfügen und Löschen von Tupeln wird auch hier eine vereinfachte Notation benutzt: `UPDATE name SET column_name = expression [,...n] WHERE condition`.

Wie bereits erwähnt, ist es sinnvoll, die Relevanz derartiger Operationen vor deren eigentlicher Ausführung zu prüfen. Nach [BCL89] kann allgemein angenommen werden, dass Einfügeoperationen nur dann relevant sind, wenn dadurch neue Tupel zu dem entsprechenden Anfrageergebnis hinzukommen würden. Analog sind Löschoperationen nur dann relevant, wenn aus

dem Anfrageergebnis Tupel entfernt würden. Für Updates muss gelten, dass die zu verändernden Tupel oder die anschließend modifizierten Tupel im Anfrageergebnis enthalten sind. Darüberhinaus muss eine relevante Änderungsoperation auch tatsächlich Änderungen an den durch die der Anfrage abgedeckten Tupel vornehmen. Im Folgenden wird nun diskutiert, wie für PSQ-notierte Anfragen eine derartige Relevanzprüfung durchgeführt werden kann. Festzuhalten bleibt, dass eine Änderungsoperation nur dann für eine Anfrage relevant ist, wenn sie sich in dem entsprechenden Ergebnis dieser widerspiegelt. Das heißt, dass kein Prädikat der Anfrage den Effekt der Änderung „ausblenden“ darf. Anfragen werden folglich dahingehend analysiert, ob sie Prädikate enthalten, die eine derartige Reflexion verhindern. Aufgrund der Anfragestruktur werden zuerst die Verbundprädikate, dann die Selektionsprädikate und abschließend die Projektionsprädikate geprüft.

Behandlung von Integritätsbedingungen

In den folgenden Betrachtungen wird davon ausgegangen, dass die Änderungsoperationen keine Integritätsbedingungen verletzen und somit auch tatsächlich ausgeführt werden können. Dies schließt die Einhaltung der korrekten Wertebereiche von Attributen sowie die referenzielle Integrität ein. Darüberhinaus sind kaskadierende Lösch- und Update-Operationen (z. B. von PostgreSQL unterstützt) unzulässig.

4.2 Prüfen der Relevanz von Einfügeoperationen

Wie bereits erwähnt, erfolgt das Einfügen von Tupeln mithilfe der entsprechenden SQL-Anweisung. Um die Relevanz derartiger Änderungen für PSQ-Anfragen zu prüfen, wird von den folgenden Annahmen ausgegangen:

- Das Einfügen eines Tupels erfolgt in die Relation $r(R_{\texttt{name}})$, deren Name `name` in der SQL-Anweisung enthalten ist.
- $A = (a_1^i, \ldots, a_n^i)$ sind die Attribute des einzufügenden Tupels und sind in `(column_list)` aufgelistet.
- $X = (x_1^i, \ldots, x_n^i)$ sind die einzufügenden und mit A korrespondierenden Attributwerte, die `(value [,...])` entnommen werden können.

Die Einschränkung, dass nur Einfügeoperationen erlaubt sind, bei denen *alle* Attribute $a \in R_{\texttt{name}}$ angegeben sind, dient hier lediglich der Vereinfachung. Das Einfügen von nicht vollständig in der Einfügeoperation definierten Tupeln, würde einen zusätzlichen Schritt notwendig machen. In diesem Fall würden die Default-Werte der nicht angegebenen Attribute aus dem Datenbankkatalog abgefragt werden und die Einfügeoperation entsprechend komplettiert.

Da im hier verwendeten Datenmodell keine Duplikate erlaubt sind, kann eine Einfügeoperation nur dann relevant sein, wenn das einzufügende Tupel noch nicht in der betroffenen Relation vorhanden ist. Nur in diesem Fall müssen die Prädikate der registrierten Anfragen getestet werden.

4.2.1 Relevanz von Einfügeoperationen für Verbundprädikate

Das Einfügen eines Tupels ist für eine Anfrage Q bezüglich ihrer Menge von Verbundprädikaten $V \subseteq VP$ relevant, wenn zum Einen das Einfügen in eine der an den Verbundprädikaten beteiligten Relationen erfolgt und zum Anderen durch das Einfügen die Kardinalität des Gesamtergebnisses der Anfrage inkrementiert wird. Um die erste Eigenschaft zu testen genügt es, die Relationennamen, die in V vorkommen mit dem Relationennamen `name` der Insert-Anweisung zu vergleichen. Um die zweite Bedingung zu prüfen, muss auf die Datenbank zurückgegriffen werden. Im Wesentlichen muss hierbei der Gesamtverbund ausgeführt werden, wobei die Relation, in welche das Tupel eingefügt werden soll, durch dieses Tupel substituiert wird. Wenn dieser „Test-Verbund“ ein nicht leeres Ergebnis zurückliefert, ist die Insert-Operation für diese Anfrage aus Sicht der Verbundprädikate relevant.

Um dieses Vorgehen formal zu betrachten, wird im Folgenden auf die Relationenalgebra zurückgegriffen. Offensichtlich kann V als Verbundkette

$$r(R_1) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)$$

mit den in den Verbundprädikaten repräsentierten Relationen $r(R_1), \ldots, r(R_n)$ geschrieben werden. Sei $r(R_1) = r(R_{\mathtt{name}})$ die Relation, in die eingefügt werden soll. Mit der in Abschnitt 3.1 gegebenen Definition ist das einzufügende Tupel als Abbildung $t_i : A \rightarrow X$ zu betrachten. Für Anfragen ohne Selbstverbund ($\forall R_i, R_j | 1 \leq i < j \leq n \wedge r(R_i) \neq r(R_j)$) kann somit die Verbundkette als

$$TJ = \{t_i\} \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.1}$$

angenommen werden. Ist TJ leer, so ist die Einfügeoperation für Q nicht relevant.

In dem Fall, dass das Einfügen in eine Relation mit einem Relationsnamen erfolgt, der durch Aliase mehrfach in den Verbundprädikaten repräsentiert ist (Selbstverbund), muss die Substitution für alle Kombinationen der entsprechenden Relationenname-Alias-Kombinationen erfolgen. Anschließend werden alle Vorkommen dieser Relation in der Verbundkette entfernt und durch die Vereinigung der Ergebnisse des Selbstverbundtestes ersetzt. Seien erneut die Verbundprädikate als Verbundkette

$$\underbrace{r(R_1^1) \bowtie_{\mathsf{jcond}_{S_1}} r(R_1^2) \bowtie_{\mathsf{jcond}_{S_2}} \cdots \bowtie_{\mathsf{jcond}_{S_m}} r(R_1^{m+1})}_{\text{Selbstverbund}} \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.2}$$

repräsentiert. Um das kombinatorische Vorgehen zu verdeutlichen wird der Selbstverbund als Selektion über das m-malige[1] Kreuzprodukt von $r(R_1)$ dargestellt. Es gilt offensichtlich:

$$\begin{aligned} r(R_1^1) \bowtie_{\mathsf{jcond}_{S_1}} r(R_1^2) \bowtie_{\mathsf{jcond}_{S_2}} \cdots \bowtie_{\mathsf{jcond}_{S_m}} r(R_1^{m+1}) = \\ \sigma_{\mathsf{jcond}_{S_1} \wedge \mathsf{jcond}_{S_2} \wedge \cdots \wedge \mathsf{jcond}_{S_m}} (\underbrace{r(R_1^1) \times r(R_1^2) \times \cdots \times r(R_1^{m+1})}_{m \text{ mal } \times}) \end{aligned} \tag{4.3}$$

[1] Kreuzprodukt über $m+1$-Relationen

Damit wird deutlich, dass sich das Einfügen des Tupels t_i in $r(R_1)$ mehrfach (auf unterschiedliche Arten) im Selbstverbund widerspiegelt. Sei beispielsweise $vp = [\texttt{name@alias1}, \texttt{name@alias2}, (\Theta)]$ ein Verbundprädikat eines Selbstverbundes mit der Verbundbedingung Θ. Wird das Tupel t_i in die Relation $r(R_{\texttt{name}})$ eingefügt, so ist das für eine Anfrage, die nur vp umfasst relevant, wenn

$$(r(R_{\texttt{name}}^{\texttt{alias1}}) \bowtie_\Theta \{t_i\}) \cup (\{t_i\} \bowtie_\Theta r(R_{\texttt{name}}^{\texttt{alias2}})) \cup (\{t_i\} \bowtie_\Theta \{t_i\}) \neq \emptyset \tag{4.4}$$

bzw.

$$\sigma_\Theta(r(R_{\texttt{name}}^{\texttt{alias1}}) \times \{t_i\}) \cup (\{t_i\} \times r(R_{\texttt{name}}^{\texttt{alias2}})) \cup (\{t_i\} \times \{t_i\})) \neq \emptyset \tag{4.5}$$

gilt. Für mehrfache Selbstverbunde müssen, wie bereits erwähnt, alle alternativen Substitutionen durchgeführt werden. Wie aus der Kombinatorik bekannt ist, steigt die Anzahl dieser Alternativen exponentiell. Der Fall, dass keine Relation des Kreuzproduktes durch das einzufügende Tupel substituiert wird, kann an dieser Stelle ausgeschlossen werden. Damit existieren im Falle eines m-fachen Selbstverbundes $2^{m+1} - 1$ zu betrachtende Substitutionsmöglichkeiten.

Um gegebenenfalls noch vorhandene Selektionsbedingungen testen zu können, ist es auch im Fall des Selbstverbundes notwendig das Testergebnis TJ zwischenzuspeichern. TJ ergibt sich analog zu Formel 4.1, wobei jedoch $r(R_{\texttt{name}})$ durch die Vereinigung der Substitutionsalternativen (vgl. Formel 4.4) ersetzt wird.

4.2.2 Relevanz von Einfügeoperationen für Selektionsprädikate

Soll das Tupel t_i in die Relation $r(R_{\texttt{name}})$ eingefügt werden, ist dies für eine Anfrage Q *ohne* Verbundprädikate relevant, wenn keine Selektionsbedingung in Q existiert, die für das eingefügte Tupel nicht erfüllt ist (vgl. Abbildung 4.1).

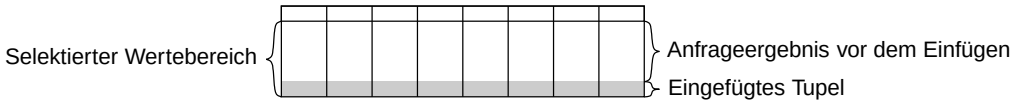


Abbildung 4.1: Eingefügtes Tupel wird in Selektion reflektiert

Demnach ist das Einfügen für Q aus Sicht der Selektionsprädikate $sp_n \in SP$ mit $n \in \mathbb{N}, 1 \leq n \leq |SP|$ und den entsprechenden Selektionsbedingungen scond_n relevant, wenn $TS \neq \emptyset$ für

$$TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(\{t_i\}) \tag{4.6}$$

gilt. In dem Fall, dass Q Verbundprädikate enthält, wurde bereits durch deren Relevanztest festgestellt, dass die vom Einfügen betroffene Relation im Verbundergebnis enthalten ist. In diesem Fall müssen alle Selektionsprädikate mit dem temporären Verbundergebnis TJ geprüft werden. Ist hierbei $TS \neq \emptyset$ für

$$TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(TJ) \tag{4.7}$$

erfüllt, so ist die Einfügeoperation für Q aus Sicht der Selektionsprädikate relevant. Abbildung 4.2 illustriert beispielhaft den Fall, dass eine Anfrage zwar seitens ihrer Verbundprädikate von einer Einfügeoperation betroffen würde, dieser Effekt aber durch die Selektionsprädikate aufgehoben wird.

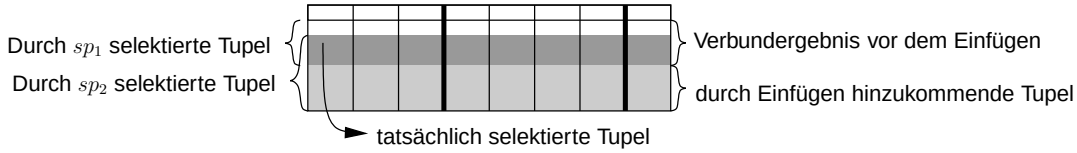


Abbildung 4.2: Durch Verbund- aber nicht durch Selektionsprädikate reflektiertes Einfügen

4.2.3 Relevanz von Einfügeoperationen für Projektionsprädikate

Das hier verwendete Datenmodell fordert die Duplikatfreiheit. Würde diese Eigenschaft nicht gefordert werden, so müssten Projektionsprädikate nur im Fall von Anfragen, die weder Selektions- noch Verbundprädikate enthalten, separat geprüft werden. Hierbei würde es auch genügen zu prüfen, ob das Einfügen die Relation betrifft, aus der die Anfrage Attribute projiziert. Bei Zulassung von Duplikaten führt ein erfolgreiches Einfügen zur Inkrementierung der Kardinalität des Ergebnisses. Dieser „Effekt“ kann durch eine Projektion nicht aus dem Anfrageergebnis „ausgeblendet“ werden.

Da hier aber ein duplikatfreies Datenmodell verwendet wird, muss die Projektion separat betrachtet werden. Beispiel 8 illustriert dieses Problem.

Beispiel 8: *Einfügen ändert Kardinalität des Ergebnisses nicht*

$r(R_{\texttt{name}})$	A	B
	1	X
	2	W

→`INSERT INTO name (A,B)`→
`VALUES (1,'H')`

$\pi_A(r(R_{\texttt{name}}))$	A
	1
	2

Sei QA die Menge der durch das Projektionsprädikat $pp \in PP$ von Q projizierten Attribute. Abhängig von Präfix Q' der Anfrage vor pp muss zwischen vier Fällen unterschieden werden:

(1) Q' ist leer: Enthält Q ausschließlich das Projektionsprädikat, so müssen im Falle der Relevanz einer Einfügeoperation in die Relation $r(R_{\texttt{name}})$ die folgenden Bedingungen gelten: (1) pp benutzt die Relation $r(R_{\texttt{name}})$ und (2) $|\pi_{QA}(r(R_{\texttt{name}}))| < |\pi_{QA}(r(R_{\texttt{name}}) \cup \{t_i\})|$.

(2) Q' enthält Selektions- aber keine Verbundprädikate: Benutzt Q sowohl Selektions- als auch Projektionsprädikate, wurde bereits beim Testen der Selektion geprüft, dass die richtige Relation verwendet wird. Daher muss geprüft werden, ob bereits ein Tupel in

$r(R_{\texttt{name}})$ existiert, sodass die projizierten Attributwerte dieses Tupels mit denen projizierten Attributwerten des eingefügten Tupels übereinstimmen. Aus der Einfügeoperation kann somit eine Bedingung $sc = \bigwedge_{\forall a_z^i \in QA} a_z^i = x_z^i$ hergeleitet werden. Bei dem einzufügenden Tupel handelt es sich bezüglich der projizierten Attribute um ein Duplikat, wenn $|\sigma_{sc}(\pi_{QA}(\sigma_{\bigwedge_{m=1}^{n} \texttt{scond}_m}(r(R_{\texttt{name}}))))| > 0$ gilt. In diesem Fall ist das Einfügen für Q irrelevant.

(3) Q' enthält Verbund- aber keine Selektionsprädikate: Bei Anfragen ohne Selektions- aber mit Verbundprädikaten und einem Projektionsprädikat, erfolgt das Testen des Projektionsprädikats nur, wenn der Verbund eine Relevanz angezeigt hat. TJ beinhaltet hierbei die Tupel, die durch das Einfügen zum Verbundergebnis hinzugekommen sind. Sei $\mathcal{B}$ die dem System zugrundeliegende Datenbank. Gilt nun $|\pi_{QA}(TJ) - Q(\mathcal{B})| > 0$, so ist in $\pi_{QA}(TJ)$ mindestens ein Tupel enthalten, das noch nicht in $Q(\mathcal{B})$ enthalten ist. Damit besteht eine Relevanz der Einfügeoperation bezüglich des Projektionsprädikats und somit auch bezüglich der Anfrage Q.

(4) Q' enthält sowohl Verbund- als auch Selektionsprädikate: Anfragen, die alle drei Prädikattypen verwenden können beim Relevanztest des Projektionsprädikats analog zu (3) behandelt werden. Hierbei muss im Falle einer Relevanz aufgrund der zusätzlichen Selektion $|\pi_{QA}(TS) - Q(\mathcal{B})| > 0$ gelten.

4.3 Prüfen der Relevanz von Löschoperationen

Allgemein kann den Betrachtungen zur Relevanz von Löschoperationen für PSQ-Anfragen vorweggenommen werden, dass das Löschen von Tupeln nur dann relevant ist, wenn dadurch die Kardinalität des entsprechenden Anfrageergebnisses dekrementiert wird. Analog zum Einfügen von Tupeln können anhand der in SQL-notierten Löschoperation die folgenden Annahmen getroffen werden:

- Das Löschen erfolgt aus der Relation $r(R_{\texttt{name}})$, deren Name `name` in der SQL-Anweisung enthalten ist.
- Die zu löschenden Tupel können durch die Selektion $\sigma_{\texttt{dcond}}(r(R_{\texttt{name}}))$ ermittelt werden. Die entsprechende Selektionsbedingung `dcond` entspricht hierbei der `WHERE`-Bedingung `clause` der SQL-notierten Löschoperation.

Im Folgenden wird davon ausgegangen, dass die Löschoperation tatsächlich Tupel aus $r(R_{\texttt{name}})$ entfernt. Wie Beispiel 9 illustriert, muss dies nicht zwangsläufig gegeben sein.

Beispiel 9: *Löschoperation ändert $r(R_{name})$ nicht*

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

→`DELETE FROM name WHERE A=4`→

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

Gilt $\sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}})) = \emptyset$, so hat das Löschen garantiert keine Änderungswirkung, sodass ein weiteres Prüfen nicht notwendig ist.

4.3.1 Relevanz von Löschoperationen für Verbundprädikate

Aus Sicht der Verbundprädikate ist eine Löschoperation für die Anfrage Q relevant, wenn die zu löschenden Tupel in das Verbundergebnis einfließen. Mit der Verbundkette

$$r(R_1) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)$$

muss also

$$\sigma_{\mathsf{dcond}}((R_1) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)) \neq \emptyset \tag{4.8}$$

gelten. Eine zwingende Bedingung hierzu ist natürlich, dass das Löschen aus einer an der Verbundkette beteiligten Relation erfolgen soll. Mithilfe der Kommutativität von Verbunden (vgl. [HS00, Seite 302]) sei erneut $r(R_{\mathtt{name}}) = r(R_1)$ angenommen. Somit kann eine einfache Optimierung dieses Relevanztests wie folgt erreicht werden: Das Löschen aus $r(R_1)$ kann nur dann für den Gesamtverbund relevant sein, wenn die zu löschenden Tupel in den Teilverbund $r(R_1) \bowtie_{\mathsf{jcond}_1} r(R_2)$ einfließen. Soll nun aus einer Relation gelöscht werden, die an *keinem* Selbstverbund beteiligt ist, ist die Löschoperation relevant, wenn $TJ \neq \emptyset$ mit

$$TJ = \sigma_{\mathsf{dcond}}(r(R_1)) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.9}$$

gilt. Im anderen Fall ($r(R_1)$ ist an einem Selbstverbund beteiligt):

$$r(R_1^1) \bowtie_{\mathsf{jcond}_{S_1}} r(R_1^2) \bowtie_{\mathsf{jcond}_{S_2}} \cdots \bowtie_{\mathsf{jcond}_{S_m}} r(R_1^{m+1}) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)$$

muss die Selektion alle alternativen Vorkommen von $r(R_1)$ in Betracht ziehen. Dies erfolgt analog zum Einfügen durch die Vereinigung der Substitutionsalternativen von $r(R_1)$ durch $\sigma_{\mathsf{dcond}}(r(R_1))$ in den Verbundprädikaten. Sei erneut $vp = [\mathtt{name@alias1}, \mathtt{name@alias2}, (\Theta)]$ ein Verbundprädikat eines Selbstverbundes mit der Verbundbedingung Θ. Werden nun Tupel, welche die Bedingung dcond erfüllen, aus $r(R_{\mathtt{name}})$ gelöscht, so ist das für eine Anfrage, die nur vp umfasst relevant, wenn

$$\begin{aligned} &(r(R_{\mathtt{name}}^{\mathtt{alias1}}) \bowtie_\Theta \sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}}^{\mathtt{alias2}})))\cup \\ &(\sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_\Theta r(R_{\mathtt{name}}^{\mathtt{alias2}}))\cup \\ &(\sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_\Theta \sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}}^{\mathtt{alias2}}))) \neq \emptyset \end{aligned} \tag{4.10}$$

gilt. Durch Adaption der Selektionsbedingung dcond an den jeweiligen Alias kann diese Bedingung vereinfacht werden. Sei $\mathsf{dcond}^{\mathtt{alias}}$ die Selektionsbedingung über der am Selbstverbund beteiligten Relation $r(R^{\mathtt{alias}})$ wobei alle Attribute in $\mathsf{dcond}^{\mathtt{alias}}$ durch das Voranstellen des Relationenaliases $\mathtt{alias}$ explizit auf diesen Teil des Selbstverbundes beschränkt sind. Die Bedingung aus Formel 4.10 wird damit zu:

$$\begin{aligned} &(r(R_{\mathtt{name}}^{\mathtt{alias1}}) \bowtie_\Theta \sigma_{\mathsf{dcond}^{\mathtt{alias2}}}(r(R_{\mathtt{name}}^{\mathtt{alias2}})))\cup \\ &(\sigma_{\mathsf{dcond}^{\mathtt{alias1}}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_\Theta r(R_{\mathtt{name}}^{\mathtt{alias2}}))\cup \\ &(\sigma_{\mathsf{dcond}^{\mathtt{alias1}}}(r(R_{\mathtt{name}}^{\mathtt{alias1}})) \bowtie_\Theta \sigma_{\mathsf{dcond}^{\mathtt{alias2}}}(r(R_{\mathtt{name}}^{\mathtt{alias2}}))) \neq \emptyset \end{aligned} \tag{4.11}$$

Aufgrund der expliziten Benennung[2] der Attributnamen in den Selektionsbedingungen kann diese Formel wie folgt umgestellt werden:

$$\begin{aligned}\sigma_{\mathsf{dcond}^{\mathtt{alias2}}}(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}}))\cup\\ \sigma_{\mathsf{dcond}^{\mathtt{alias1}}}(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}}))\cup\\ \sigma_{\mathsf{dcond}^{\mathtt{alias1}}\wedge\mathsf{dcond}^{\mathtt{alias2}}}(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}})) \neq \emptyset\end{aligned} \tag{4.12}$$

Nach [Mit95, Seite 57][3] gilt $\sigma_{pred1}(r(R)) \cup \sigma_{pred2}(r(R)) \equiv \sigma_{pred1\vee pred2}(r(R))$. Damit ist Formel 4.12 äquivalent zu

$$\begin{aligned}\sigma_{\mathsf{dcond}^{\mathtt{alias2}}\vee\mathsf{dcond}^{\mathtt{alias1}}\vee(\mathsf{dcond}^{\mathtt{alias1}}\wedge\mathsf{dcond}^{\mathtt{alias2}})}&(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}}))\cup\\ &(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}}))\cup\\ &(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}})) \neq \emptyset\end{aligned} \tag{4.13}$$

Offensichtlich können die Selektionsbedingung $\mathsf{dcond}^{\mathtt{alias2}} \vee \mathsf{dcond}^{\mathtt{alias1}} \vee (\mathsf{dcond}^{\mathtt{alias1}} \wedge \mathsf{dcond}^{\mathtt{alias2}})$ und die Vereinigung mithilfe einfacher Logik bzw. der „IdemUnion-Regel“ [SH99, Seite 361] noch weiter vereinfacht werden, sodass in diesem Beispiel eine Löschoperation für die Anfrage relevant ist, wenn:

$$\sigma_{\mathsf{dcond}^{\mathtt{alias2}}\vee\mathsf{dcond}^{\mathtt{alias1}}}(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_\Theta r(R^{\mathtt{alias2}}_{\mathtt{name}})) \neq \emptyset \tag{4.14}$$

gilt. Es kann gezeigt werden, dass für $m-$fache Selbstverbunde die Selektionsbedingung die Disjunktion der $m+1$ adaptierten Teil-Bedingungen (explizit definierte Attributnamen) ist, sodass das temporäre Verbundergebnis der zu löschen Tupel als:

$$\begin{aligned}TJ = \sigma_{\mathsf{dcond}^{\mathtt{alias1}}\vee\ldots\mathsf{dcond}^{\mathtt{alias}m+1}}(r(R^{\mathtt{alias1}}_{\mathtt{name}}) \bowtie_{\mathsf{jcond}_{S_1}} r(R^{\mathtt{alias2}}_{\mathtt{name}}) \cdots \bowtie_{\mathsf{jcond}_{S_m}} \cdots\\ \ldots r(R^{\mathtt{alias}m+1}_{\mathtt{name}})) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)\end{aligned} \tag{4.15}$$

gegeben ist.

4.3.2 Relevanz von Löschoperationen für Selektionsprädikate

Der Einfluss einer Löschoperation auf die Selektionsprädikate einer Anfrage Q hängt in erster Linie davon ab, ob das Löschen aus einer Relation erfolgt, die in das Anfrageergebnis einfließt. Enthält die Anfrage Verbundprädikate, so wurde dies bereits durch das Testen der Verbundprädikate geprüft. Darüber hinaus ist festzuhalten, dass das Prüfen der Selektionsprädikate nicht notwendig ist wenn das Löschen den Verbund nicht beeinflusst hat. Hat sich hingegen das Verbundergebnis geändert, besteht trotzdem die Möglichkeit, dass die Selektionsprädikate derart spezifiziert sind, dass sich das Gesamtergebnis nach dem Löschen nicht ändert. Wie in Abbildung 4.3 beispielhaft illustriert ist, tritt dieser Fall ein, wenn die Menge der selektierten Tupel und das temporäre Verbundergebnisses TJ disjunkt sind.

[2] in SQL `Relationenname.Attributname`

[3] In anderer Literatur zur logischen Optimierung (z. B. [SH99, EN02]) wird diese Regel nicht explizit angegeben, sie kann jedoch recht einfach aus der Definition der Vereinigung als $r(R_1) \cup r(R_1) := \{t|t \in r(R_1) \vee t \in r(R_2)\}$ (vgl. [HS00, Seite 304]) und der Definition der Selektion als $\sigma_F(r(R)) := \{t|t \in r(R) \wedge F(t) = \mathtt{true}\}$ (vgl. [HS00, Seite 299]) hergeleitet werden.

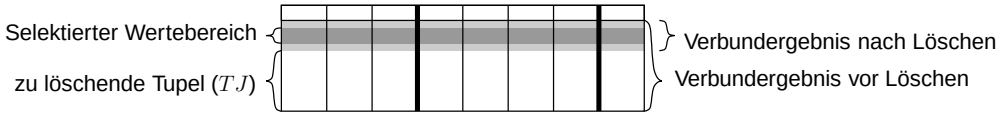


Abbildung 4.3: Durch Verbund- aber nicht durch Selektionsprädikate reflektiertes Löschen

Formal betrachtet ist eine Löschoperation für eine Anfrage mit Verbundprädikaten aus Sicht der Selektionsprädikate $sp_n \in SP$ mit $n \in \mathbb{N}, 1 \leq n \leq |SP|$ und den entsprechenden Selektionsbedingungen scond_n relevant, wenn $TS \neq \emptyset$ mit

$$TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(TJ) \tag{4.16}$$

gilt. Enthält Q hingegen keine Verbundprädikate, so muss im Falle der Relevanz der Löschoperation über $r(R_{\mathtt{name}})$ für

$$TS = \sigma_{\mathsf{dcond} \wedge \bigwedge_{m=1}^{n} \mathsf{scond}_m}(r(R_{\mathtt{name}})) \tag{4.17}$$

$TS \neq \emptyset$ gelten. Abbildung 4.4 illustriert diesen Fall.

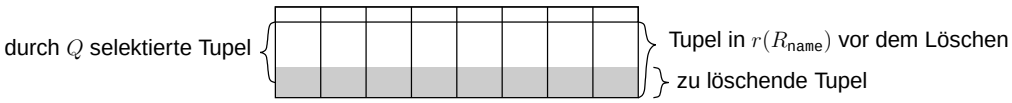


Abbildung 4.4: Gelöschtes Tupel wird in Selektion reflektiert (Q ohne Verbunde)

4.3.3 Relevanz von Löschoperationen für Projektionsprädikate

Für die Relevanz von Löschoperationen für Projektionsprädikate gilt analog zu Einfügeoperationen, dass sie aufgrund der geforderten Duplikatfreiheit getestet werden müssen. Beispiel 10 illustriert den Fall, dass zwar ein Tupel aus der Relation $r(R_{\mathtt{name}})$ gelöscht wird, sich aber das Anfrageergebnis aufgrund der Projektion nicht ändert.

Beispiel 10: *Delete ändert Kardinalität des Ergebnisses nicht*

$r(R_{\mathtt{name}})$	A	B
	1	X
	1	W

→`DELETE FROM name WHERE B='W'`→

$\pi_A(r(R_{\mathtt{name}}))$	A
	1

Sei erneut QA die Menge der durch das Projektionsprädikat $pp \in PP$ von Q projizierten Attribute. Für den Relevanztest muss analog zu Abschnitt 4.2.3 zwischen Alternativen unterschieden werden.

(1) Enthält Q keine weiteren Prädikate, muss eine relevante Löschoperation Tupel aus der Relation löschen, die durch Q verwendet wird. Ist dies der Fall kann dennoch eine Relevanz ausgeschlossen werden, wenn für jedes zu löschenden Tupel d mindestens ein Tupel t in $r(R)$ verbleibt, sodass $\pi_{QA}(\{d\}) = \pi_{QA}(\{t\})$ gilt. Diese Situation tritt ein, wenn $|\pi_{QA}(\sigma_{\mathsf{dcond}}(r(R_{\mathtt{name}}))) - \pi_{QA}(\sigma_{\neg\mathsf{dcond}}r(R_{\mathtt{name}}))| = 0$ gilt.

(2) Enthält Q neben dem Projektionsprädikat ausschließlich Selektionsprädikate, muss zum Testen die durch die Selektion voreingeschränkte Tupelmenge genutzt werden. Sei $sc = \bigwedge_{m=1}^{n} \mathsf{scond}_m$ die konjunktive Verknüpfung aller Selektionsbedingungen von Q. Damit ist das Delete für Q relevant, wenn $|\pi_{QA}(TS) - \pi_{QA}(\sigma_{sc\wedge\neg(\mathsf{dcond})}r(R_{\mathtt{name}}))| > 0$ gilt.

(3) Enthält Q neben dem Projektionsprädikat ausschließlich Verbundprädikate, muss zum Testen der Verbund virtuell ausgeführt werden, wobei aus der zu ändernden Relation diejenigen Tupel selektiert werden, welche nicht die dcond erfüllen und somit auch nach dem Löschen noch im Verbundergebnis enthalten sind. Sei $r(R_1) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)$ die aus den Verbundprädikaten von Q abgeleitete Verbundkette. Das Ergebnis dieser Anfrage nach dem Ausführen der Löschoperation in $r(R_1) = r(R_{\mathtt{name}})$ kann mithilfe von $VJ = \sigma_{\neg\mathsf{dcond}}(r(R_1)) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n)$ simuliert werden. Im Fall einer Anfrage mit Selbstverbund müssen alle Vorkommen von $r(R_1)$ durch $\sigma_{\neg\mathsf{dcond}}r(R_1)$ substituiert werden. Damit besteht eine Relevanz, wenn $|\pi_{QA}(TJ) - \pi_{QA}(VJ)| > 0$ gilt.

(4) Enthält Q alle drei Prädikattypen, so spiegelt sich dies auch im Relevanztest des Projektionsprädikats wieder. TS beschreibt in diesem Fall die Tupel, die durch die Löschoperation auf $r(R_{\mathtt{name}})$ aus dem Gesamtergebnis entfernt werden sollen. Dieser Fall ist eine Erweiterung von (3) auf Basis von (2), da VJ durch die Selektionprädikate weiter einzuschränken ist. Demnach ist die Löschoperation für Q relevant, wenn $|\pi_{QA}(TJ) - \pi_{QA}(\sigma_{sc}(VJ))| > 0$

4.4 Prüfen der Relevanz von Update-Operationen

Ein naiver Ansatz zu prüfen, ob eine Änderung für eine Anfrage relevant ist, besteht darin, die Änderungsoperation durch die Kombination von einer Lösch- und mehreren Einfügeoperationen zu behandeln. Im Gegensatz zu Einfüge- und Löschoperationen (ohne Projektionen) müssen sich Änderungsoperationen (ohne Projektionen) aber nicht in der Kardinalität des Ergebnisses einer Anfrage widerspiegeln. Des Weiteren kann ein `INSERT` den Effekt einer Löschoperation aufheben (vgl. Beispiel 11), wodurch sich das Ergebnis einer Anfrage nicht ändert. Dennoch sind beide Operationen separat betrachtet relevant. Es genügt daher nicht, die unter Abschnitt 4.3 bzw. Abschnitt 4.2 eingeführten Relevanzprüfungen hintereinander auszuführen.

Beispiel 11: *Update kann nicht als Delete-Insert-Kombination betrachtet werden*

$r(R_{\texttt{name}})$	A	B
	1	X
	2	X

→`SELECT * FROM A WHERE A=1`→

A	B
1	X

`UPDATE SET A=A-1 WHERE A<4`

$r(R_{\texttt{name}})$	A	B
	0	X
	1	X

→`SELECT * FROM A WHERE A=1`→

A	B
1	X

Beispiel 12: *Update modifiziert $r(R_{name})$ nicht*

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

→`UPDATE SET A=1 WHERE A=1`→
→`UPDATE SET A=5 WHERE A=4`→
→`UPDATE SET A=1 WHERE A<4`→

$r(R_{\texttt{name}})$	A	B
	1	X
	1	W

Sei ucond die Selektionsbedingung mit der die zu ändernden Tupel ausgewählt werden. Somit entspricht ucond der `condition` der SQL-notierten Update-Operation. Die Funktion uexpr, die beim Updaten auf die Tupel, welche ucond erfüllen, angewendet wird, kann ebenfalls direkt aus dem SQL-Ausdruck extrahiert werden. Dort ist sie als Liste von `column_name=expression`-Zuweisungen angegeben. In Anlehnung an die formale Betrachtung der relationalen Änderungsoperationen in [HS00] kann eine Update-Operation als Abbildung u : **DAT**$(\mathcal{S})$ → **DAT**$(\mathcal{S})$ betrachtet werden. **DAT**$(\mathcal{S})$, mit dem Datenbankschema $\mathcal{S} = (S, \Gamma)$, ist hierbei die Menge der korrekten Datenbankzustände, wobei $S = \{r(R_1), \ldots, r(R_p)\}$ die Menge der Relationen und Γ die Menge der Integritätsbedingungen ist. Wie eingangs erwähnt wird davon ausgegangen, dass Änderungsoperationen *keine* Integritätsbedingungen verletzen. Des Weiteren wird eine Update-Operation nur auf einer Relation zugelassen. Damit kann sie vereinfacht als Abbildung einer Menge von Tupeln auf eine Menge von Tupeln der Form $ut : r(R) \rightarrow r(R)$ betrachtet werden:

$$ut(t) = \begin{cases} \text{uexpr}(t) & \text{wenn } \sigma_{\text{ucond}}(\{t\}) \neq \emptyset \\ t & \text{sonst} \end{cases} \tag{4.18}$$

Auf der Ebene der Relationen ist ein Update somit eine Abbildung $ur : S \rightarrow S$ der Form

$$ur(r(R)) = \bigcup_{m=1}^{|r(R)|} ut(t_m) \tag{4.19}$$

mit $t_m \in r(R)$. Ein Update der Relation $r(R_{\texttt{name}})$ ändert diese Relation, wenn

$$r(R_{\texttt{name}}) \neq ur(r(R_{\texttt{name}})) \tag{4.20}$$

gilt. Wie in Beispiel 12 dargestellt ist, kann diese Bedingung auf zwei Arten verletzt werden:

- Die Unerfüllbarkeit der Bedingung ucond bezüglich $r(R_{\texttt{name}})$ führt dazu, dass keine Tupel zur Modifikation ausgewählt werden. Im Beispiel tritt dieser Fall bei der Update-Operation `UPDATE SET A=5 WHERE A=4` ein, da die Relation kein Tupel enthält, dessen Attribut A den Wert 4 hat. Gilt also $\sigma_{\texttt{ucond}}(r(R_{\texttt{name}})) = \emptyset$, so kann das Update für keine Anfrage relevant sein, womit eine weitere Prüfung unnötig ist.

- Durch das Ersetzen von Tupeln durch sich selbst ($\forall t((t \in r(R_{\texttt{name}})) \rightarrow (ut(t) = t))$), wie es im Beispiel durch `UPDATE SET A=1 WHERE A=1` getan wird, wird die Relation ebenfalls nicht verändert. Leider kann dieser Fall nicht generell direkt anhand der Anfrage festgestellt werden, wie das Update `UPDATE SET A=1 WHERE A<4` im Beispiel verdeutlicht. Allgemein gilt hier, dass ein Update nur dann relevant für eine Anfrage sein kann, wenn $ur(\sigma_{\texttt{ucond}}(r(R_{\texttt{name}})) \neq \sigma_{\texttt{ucond}}(r(R_{\texttt{name}}))$ gilt.

Im Folgenden wird davon ausgegangen, dass das Update tatsächlich zu Veränderungen an der Relation $r(R_{\texttt{name}})$ geführt hat.

4.4.1 Relevanz von Update-Operationen für Verbundprädikate

Primäre Voraussetzung für eine Relevanz von Update-Operationen für Verbundprädikate ist, dass die Änderung eine Relation $r(R_{\texttt{name}})$ modifiziert, die auch in den Verbundprädikaten verwendet wird. Darüberhinaus können aus Sicht der Kardinalität die folgenden Fälle eintreten:

- Das Update modifiziert $r(R_{\texttt{name}})$ derart, dass mehr Tupel die Verbundbedingung erfüllen. Dadurch wird die Kardinalität des Verbundergebnisses größer (Beispiel 13).

Beispiel 13: *Update inkrementiert Kardinalität des Verbundergebnisses*

$r(R_{\texttt{name}})$	A	B
	1	X
	2	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W

```
UPDATE name SET A=1 WHERE A=2
```

$r(R_{\texttt{name}})$	A	B
	1	X
	1	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	**Z**	**1**	**W**

- Das Update modifiziert $r(R_{\texttt{name}})$ derart, dass weniger Tupel die Verbundbedingung erfüllen, wodurch die Kardinalität des Verbundergebnisses kleiner wird (Beispiel 14).

Beispiel 14: *Update dekrementiert Kardinalität des Verbundergebnisses*

$r(R_{\texttt{name}})$	A	B
	1	X
	1	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	Z	1	W

```
UPDATE name SET A=2 WHERE B='Z'
```

$r(R_{\texttt{name}})$	A	B
	1	X
	2	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W

- Das Update modifiziert $r(R_{\texttt{name}})$ derart, dass sich die Tupelanzahl des Verbundergebnisses nicht ändert. Dennoch können auf der Ebene der Attributwerte Modifikationen in das Verbundergebnis einfließen (Beispiel 15).

Beispiel 15: *Update ändert Kardinalität des Verbundergebnisses nicht*

$r(R_{\texttt{name}})$	A	B
	1	X
	1	Z

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	Z	1	W

```
UPDATE name SET B='Y' WHERE B='Z'
```

$r(R_{\texttt{name}})$	A	B
	1	X
	1	**Y**

$r(R_2)$	C	D
	1	W
	3	U

$r(R_{\texttt{name}}) \bowtie_{A=C} r(R_2)$	A	B	C	D
	1	X	1	W
	1	**Y**	1	W

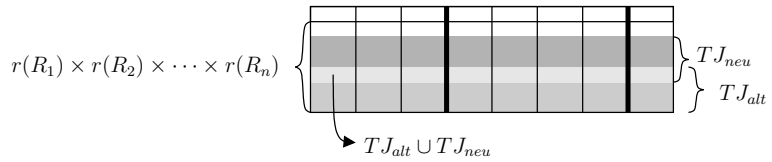
Wie bereits erwähnt kann somit die Kardinalität nicht als ausschlaggebendes Kriterium für eine Relevanz einer Update-Operation bezüglich der Verbundprädikate herangezogen werden. Vielmehr muss betrachtet werden, ob die Tupel auf Attributwert-Ebene von der Änderung betroffen sind. Sei hierzu TJ_{alt} das Verbundergebnis vor der Durchführung des Updates. Des Weiteren sei TJ_{neu} das Verbundergebnis nach dem Update. Offensichtlich ist ein Update aus Sicht der Verbunde relevant, wenn

$$TJ_{neu} \neq TJ_{alt} \tag{4.21}$$

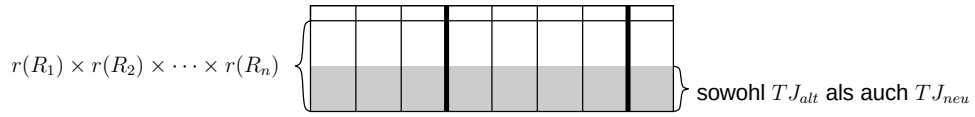
bzw. $TJ \neq \emptyset$ für

$$\begin{aligned} TJ &= (TJ_{neu} - TJ_{alt}) \cup (TJ_{alt} - TJ_{neu}) \\ &= (TJ_{alt} \cup TJ_{neu}) - (TJ_{alt} \cap TJ_{neu}) \end{aligned} \tag{4.22}$$

gilt (vgl. Abbildung 4.5). Dieser Ansatz bedeutet jedoch, dass sowohl TJ_{neu} als auch TJ_{alt}



(a) Update ist relevant für Verbundprädikate



(b) Update ist irrelevant für Verbundprädikate

Abbildung 4.5: Relevanz bzw. Irrelevanz eines Updates für Verbundprädikate

komplett berechnet werden müssen. Da TJ_{alt} durch das Update in TJ_{neu} überführt werden soll, kann TJ_{neu} formal wie folgt aus TJ_{alt} hergeleitet werden:

$$TJ_{neu} = (TJ_{alt} - J_{vu}) \cup J_{nu} \tag{4.23}$$

wobei J_{vu} die Tupel des Kreuzproduktes aller am Gesamtverbund beteiligten Relationen beschreibt, die geändert werden sollen (vgl. Abbildung 4.6). J_{nu} beschreibt hingegen die Tupel, welche durch die Änderung zum Kreuzprodukt hinzugefügt werden. Damit muss im Falle ei-

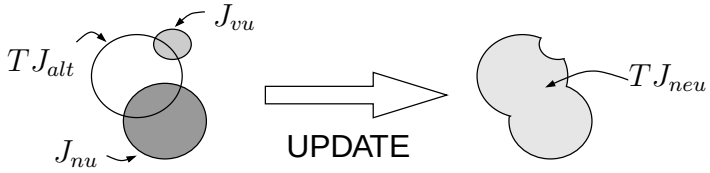


Abbildung 4.6: Herleitung von TJ_{neu} mittels TJ_{alt}, J_{nu} und J_{vu}

ner Relevanz des Updates $TJ \neq \emptyset$ für:

$$\begin{aligned}
TJ &= [\underbrace{TJ_{alt} \cup ((TJ_{alt} - J_{vu}) \cup J_{nu})}_{\text{Assoziativität}}] - [TJ_{alt} \cap ((TJ_{alt} - J_{vu}) \cup J_{nu})] \\
&= [\underbrace{TJ_{alt} \cup (TJ_{alt} - J_{vu})}_{(TJ_{alt} - J_{vu}) \subseteq TJ_{alt}} \cup J_{nu}] - [TJ_{alt} \cap ((TJ_{alt} - J_{vu}) \cup J_{nu})] \\
&= [TJ_{alt} \cup J_{nu}] - [TJ_{alt} \cap \underbrace{((TJ_{alt} - J_{vu}) \cup J_{nu})}_{\text{Distributivität}}] \\
&= [TJ_{alt} \cup J_{nu}] - [\underbrace{(TJ_{alt} \cap (TJ_{alt} - J_{vu}))}_{(TJ_{alt} - J_{vu}) \subseteq TJ_{alt}} \cup (TJ_{alt} \cap J_{nu})] \\
&= [TJ_{alt} \cup J_{nu}] - [(TJ_{alt} - J_{vu}) \cup (TJ_{alt} \cap J_{nu})]
\end{aligned} \tag{4.24}$$

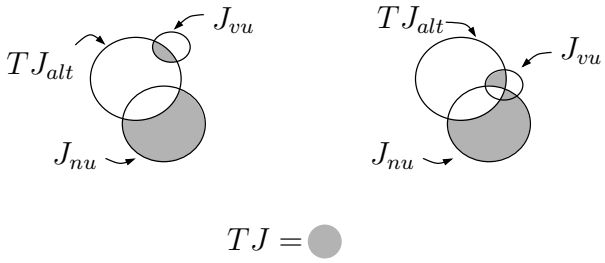


Abbildung 4.7: Berechnung von TJ

gelten (vgl. die beiden Beispiele in Abbildung 4.7).
Für die Bestimmung von J_{nu} muss das Update auf der Relation $r(R_{\mathsf{name}}) = r(R_1)$ virtuell ausgeführt werden. Es gilt also in dem Fall, dass die zu ändernde Relation an einem Selbstverbund beteiligt ist,

$$J_{nu} = \underbrace{ur(\sigma_{\mathsf{ucond}}(r(R_1^1))) \bowtie_{\mathsf{jcond}_{S_1}} ur(\sigma_{\mathsf{ucond}}(r(R_1^2))) \cdots \bowtie_{\mathsf{jcond}_{S_m}} ur(\sigma_{\mathsf{ucond}}(r(R_1^{m+1})))}_{\text{Selbstverbund}} \cdots \\ \cdots \bowtie_{\mathsf{jcond}_1} r(R_2) \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.25}$$

und im anderen Fall:

$$J_{nu} = ur(\sigma_{\mathsf{ucond}}(r(R_1))) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.26}$$

Im Gegensatz dazu ist es aber nicht notwendig, J_{vu} explizit zu bestimmen da – wie aus Formel 4.24 ersichtlich – lediglich diejenigen Tupel erfasst werden müssen, die auch in den aktuellen Verbund einfließen. Analog zum Löschen gilt für ein Update einer an *keinem* Selbstverbund beteiligten Relation:

$$(TJ_{alt} - J_{vu}) = \sigma_{\mathsf{ucond}}(r(R_1)) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.27}$$

und für den Fall des Selbstverbundes:

$$(TJ_{alt} - J_{vu}) = \sigma_{\mathsf{ucond}^{\mathtt{alias1}} \vee \ldots \mathsf{ucond}^{\mathtt{alias}m+1}}(r(R_{\mathsf{name}}^{\mathtt{alias1}}) \bowtie_{\mathsf{jcond}_{S_1}} r(R_{\mathsf{name}}^{\mathtt{alias2}}) \cdots \bowtie_{\mathsf{jcond}_{S_m}} \cdots \\ \ldots r(R_{\mathsf{name}}^{\mathtt{alias}m+1})) \bowtie_{\mathsf{jcond}_1} r(R_2) \bowtie_{\mathsf{jcond}_2} \cdots \bowtie_{\mathsf{jcond}_n} r(R_n) \tag{4.28}$$

Die Irrelevanz eines Updates für die Verbundprädikate tritt demnach auf, wenn die Differenz $[TJ_{alt} \cup J_{nu}] - [(TJ_{alt} - J_{vu}) \cup (TJ_{alt} \cap J_{nu})]$ die leere Menge ergibt. Hierzu müssen Minuent und Subtrahent gleich sein. In diesem Fall muss also $[(TJ_{alt} - J_{vu}) \cup (TJ_{alt} \cap J_{nu})] = [TJ_{alt} \cup J_{nu}]$ gelten. Dies geschieht in den folgenden Fällen:

1. Durch die Änderung kommen keine neuen Tupel zu dem Verbundergebnis hinzu ($J_{nu} = \emptyset$), und die zu ändernden Tupel sind nicht im ursprünglichen Verbundergebnis enthalten ($TJ_{alt} - J_{vu} = TJ_{alt}$). Der Sonderfall $J_{vu} = \emptyset$ impliziert $J_{nu} = \emptyset$.

Beweis.

$$\begin{aligned}[\underbrace{TJ_{alt} \cup J_{nu}}_{A \cup \emptyset = A}] &= [\underbrace{(TJ_{alt} - J_{vu})}_{\text{siehe Voraussetzung}} \cup \underbrace{(TJ_{alt} \cap J_{nu})}_{A \cap \emptyset = \emptyset}] \\ TJ_{alt} &= TJ_{alt}\end{aligned} \tag{4.29}$$

□

2. Das Verbundergebnis mit den durch die Änderung berechneten Tupel ist bereits vollständig im alten Verbundergebnis enthalten ($J_{nu} \subseteq TJ_{alt}$) und kompensiert das "Entfernen" von Tupeln durch das Update ($TJ_{alt} = (TJ_{alt} - J_{vu}) \cup ur(J_{vu})$)

   Beweis.

$$\begin{aligned}[\underbrace{TJ_{alt} \cup J_{nu}}_{A \cup B = A | \forall B \subseteq A}] &= [(TJ_{alt} - J_{vu}) \cup \underbrace{(TJ_{alt} \cap J_{nu})}_{A \cap B = B | \forall B \subseteq A}] \\ TJ_{alt} &= [\underbrace{(TJ_{alt} - J_{vu}) \cup J_{nu}}_{\text{siehe Voraussetzungen}}] \\ TJ_{alt} &= TJ_{alt}\end{aligned} \tag{4.30}$$

□

4.4.2 Relevanz von Update-Operationen für Selektionsprädikate

Aus Sicht der Selektionsprädikate $S \subseteq SP$ mit $S = \{sp_1, \ldots, sp_n\}$ ($n \in \mathbb{N}, 1 \leq n \leq |S|$) einer Anfrage Q und den entsprechenden Selektionsbedingungen scond_n besteht eine Relevanz eines Updates, wenn die Selektion Tupel erfasst, die durch das Update zum Verbundergebnis neu hinzukommen oder durch das Update aus dem Verbundergebnis entfernt werden. Enthält Q Verbundprädikate, muss also $TS \neq \emptyset$ für

$$TS = \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(TJ) \neq \emptyset \tag{4.31}$$

Sind in Q keine Verbundprädikate enthalten, so muss die Relevanzprüfung analog zum Verbund betrachtet werden. Es muss $TS \neq \emptyset$ für

$$\begin{aligned}TS = &[\sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(r(R_{\mathsf{name}})) \cup \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(ur(r(R_{\mathsf{name}})))] - \\ &[\sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(r(R_{\mathsf{name}})) \cap \sigma_{\bigwedge_{m=1}^{n} \mathsf{scond}_m}(ur(r(R_{\mathsf{name}})))]\end{aligned} \tag{4.32}$$

gelten. Da es sich hierbei bei $r(R_{\mathsf{name}})$ und $ur(r(R_{\mathsf{name}}))$ nicht um Teilmengen eines Kreuzproduktes mehrerer Relationen handelt, gilt hier offensichtlich $ur(\sigma_{\mathsf{ucond}}(r(R_{\mathsf{name}}))) = ur(r(R_{\mathsf{name}}))$. Damit ist eine Optimierung wie bei der Behandlung von Verbundprädikaten (vgl. Formel 4.24) nicht sinnvoll.

4.4.3 Relevanz von Update-Operationen für Projektionsprädikate

Da Update-Operationen nicht zwangsweise die Kardinalität eines Anfrageergebnisses beeinflussen, muss hier nicht nur aufgrund der geforderten Duplikatfreiheit untersucht werden, ob eine Relevanz bezüglich der Projektionsprädikate besteht. Allgemein trifft diese zu, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- Die zu verändernden Tupel können als gelöscht interpretiert werden. Es muss daher mindestens ein zu veränderndes Tupel existieren, dessen projizierte Attributwerte nicht in einem Tupel des Anfrageergebnisses nach dem Update vorkommen.
- Die geänderten Tupel können als eingefügte Tupel interpretiert werden. Daher muss zumindest ein neues Tupel existieren, dessen projizierten Attributwerte nicht in einem Tupel der Differenz zwischen dem alten Anfrageergebnis und den durch das Update daraus entfernten Tupeln vorkommen.

Analog zu den Relevanztests von Projektionsprädikaten bei Einfüge- und Löschoperationen, kann auch hier zwischen den folgenden vier Fällen unterschieden werden:

(1) Enthält die Anfrage Q ausschließlich das Projektionsprädikat, muss das Update die Relation modifizieren, welche durch Q verwendet wird, damit eine Relevanz bestehen kann. Eine weitere zwingende Bedingung für eine Relevanz ist, dass das Update zumindest ein Attribut modifiziert, das auch im Projektionsprädikat verwendet wird. Sei QA die Menge der von Q projizierten Attribute und UA die durch das Update veränderten Attribute[4]. Gilt also $UA \cap QA = \emptyset$, ist das Update irrelevant. Anderenfalls müssen die oben genannten Bedingungen geprüft werden. Es muss also

$$\pi_{QA}(\sigma_{\mathsf{ucond}}(r(R_{\mathsf{name}})) - \underbrace{\pi_{QA}(ur(r(R_{\mathsf{name}})))}_{Q(\{(ur(r(R_{\mathsf{name}})))\})} \neq \emptyset \tag{4.33}$$

oder

$$\underbrace{\pi_{QA}(ur(r(R_{\mathsf{name}})))}_{Q(\{(ur(r(R_{\mathsf{name}})))\})} - \pi_{QA}(\sigma_{\neg(\mathsf{ucond})}(r(R_{\mathsf{name}}))) \neq \emptyset \tag{4.34}$$

gelten.

(2) Wenn die Anfrage Q neben dem Projektionsprädikat ausschließlich n Selektionsprädikate enthält, wurde die Verwendung der richtigen Relation bereits bei den Selektionsprädikaten getestet. Dennoch gilt wenn $UA \cap QA = \emptyset$, dann ist das Update irrelevant. Ist dies nicht der Fall, müssen erneut die beiden oben genannten Bedingungen getestet werden. Somit ist das Update relevant, wenn

$$\pi_{QA}(\sigma_{\mathsf{ucond} \wedge \bigwedge_{m=1}^{n} \mathsf{scond}_m}(r(R_{\mathsf{name}}))) - Q(\{(ur(r(R_{\mathsf{name}})))\}) \neq \emptyset \tag{4.35}$$

[4] UA kann direkt aus der `column_name=expression`-Liste der SQL-notierten Update-Anweisung extrahiert werden.

oder

$$\underbrace{\pi_{QA}(\sigma_{\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}(ur(r(R_{\mathtt{name}})))}_{Q(\{(ur(r(R_{\mathtt{name}})))\})} - \pi_{QA}(\sigma_{\neg(\mathsf{ucond})\wedge\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}(r(R_{\mathtt{name}}))) \neq \emptyset \quad (4.36)$$

gilt.

(3) Besteht die Anfrage Q nur aus Verbundprädikaten und dem Projektionsprädikat, so wurde bereits beim Testen der Verbundprädikate die Verwendung der richtigen Relation geprüft. Im Gegensatz zu den beiden vorherigen Fällen kann außerdem die Beziehung zwischen QA und UA hier nicht als Irrelevanzkriterium herangezogen werden, da das Update Einfluss auf das Verbundergebnis hatte und sich somit dessen Kardinalität geändert hat. Beim Testen der beiden Bedingungen kann auf die in Abbildung 4.6 dargestellten Mengenbezeichner zurückgegriffen werden. Damit ist ein Update relevant, wenn

$$\pi_{QA}(J_{vu}) - \underbrace{\pi_{QA}(\overbrace{(TJ_{alt} - J_{vu}) \cup J_{nu}}^{TJ_{neu}(\text{siehe Formel 4.23})})}_{Q(\{(ur(r(R_{\mathtt{name}})))\})} \neq \emptyset \quad (4.37)$$

oder

$$\pi_{QA}(J_{nu}) - \pi_{QA}(TJ_{alt} - J_{vu}) \neq \emptyset \quad (4.38)$$

gilt.

(4) Sind alle drei Prädikattypen in der Anfrage Q vorhanden, muss Fall (3) dahingehend erweitert werden, dass die beiden Tests die n Selektionsprädikate reflektieren. In diesem Fall muss also:

$$\pi_{QA}(\sigma_{\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}(J_{vu})) - \underbrace{\pi_{QA}(\sigma_{\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}((TJ_{alt} - J_{vu}) \cup J_{nu}))}_{Q(\{(ur(r(R_{\mathtt{name}})))\})} \neq \emptyset \quad (4.39)$$

oder

$$\pi_{QA}(\sigma_{\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}(J_{nu})) - \pi_{QA}(\sigma_{\bigwedge_{m=1}^{n}\,\mathsf{scond}_m}(TJ_{alt} - J_{vu})) \neq \emptyset \quad (4.40)$$

gelten, damit das Update relevant ist.

4.5 Zusammenfassung

In den vorigen Abschnitten dieses Kapitels wurden Tests vorgestellt, mit denen die Relevanz von Änderungsoperationen für PSQ-notierte Anfragen ermöglicht wird. Hierzu wird prädikattypweise vorgegangen, wobei Teilergebnisse eines Tests im nächsten Test wiederverwendet werden. Das heißt, dass die Menge TJ, welche die Differenz zwischen dem Verbundergebnis vor der Änderung und nach der Änderung enthält, für das Testen der Selektionsprädikate

benutzt wird. Die daraus resultierende Menge TS wird wiederum zum Testen des Projektionsprädikates einer Anfrage benutzt. Eine Ausnahme bildet hierbei lediglich die Behandlung von Projektionsprädikaten bei Updates, da hier sowohl TJ als auch TS Mengen sind, die sowohl die durch das Update aus dem Anfrageergebnis zu entfernenden, als auch die durch das Update neu hinzukommenden Tupel umfassen. Dennoch können auch hier Teilergebnisse vorheriger Tests wiederverwendet werden. Die folgende Übersicht fasst die einzelnen, in diesem Kapitel besprochenen, Relevanztest zusammen.

- Relevanzprüfung
  - Test der Relationennamen
    - INSERT
      - Tupel bereits in $r(R_{\texttt{name}}) \Rightarrow$ irrelevant (Seite 49)
      - Test der Prädikate
        - $\bowtie_\theta$: $TJ = \emptyset \Rightarrow$ irrelevant (Formel 4.1 und Seite 51)
          - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formel 4.7)
            - π: Kardinalitätsänderung basierend auf TS (Seite 53)
          - π: Kardinalitätsänderung basierend auf TJ (Seite 53)
        - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formel 4.6)
          - π: Kardinalitätsänderung der Menge der sel. und proj. Tupel (S. 52)
        - π: Kardinalitätsänderung der Menge der projizierten Tupel (Seite 52)
    - DELETE
      - Zu löschende Tupel nicht in $r(R_{\texttt{name}}) \Rightarrow$ irrelevant (Seite 53)
      - Test der Prädikate
        - $\bowtie_\theta$: $TJ = \emptyset \Rightarrow$ irrelevant (Formeln 4.9 und 4.15)
          - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formel 4.17)
            - π: Kardinalitätsänderung basierend auf TS (Seite 57)
          - π: Kardinalitätsänderung basierend auf TJ (Seite 57)
        - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formel 4.16)
          - π: Kardinalitätsänderung der Menge der sel. und proj. Tupel (S. 57)
        - π: Kardinalitätsänderung der Menge der projizierten Tupel (Seite 57)
    - UPDATE
      - Update führt tatsächlich zu Veränderungen an $r(R_{\texttt{name}})$ (Seite 59)
      - Test der Prädikate
        - $\bowtie_\theta$: $TJ = \emptyset \Rightarrow$ irrelevant (Formel 4.24)
          - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formeln 4.32)
            - π: (Formel 4.39$\vee$ Formel 4.40)
          - π: „Duplikate“ nach Verbund? (Formel 4.37$\vee$ Formel 4.38)
        - σ: $TS = \emptyset \Rightarrow$ irrelevant (Formeln 4.31)
          - π: „Duplikate“ nach Selektion? (Formel 4.35$\vee$ Formel 4.36)
        - π: „Duplikateliminierung“ bei Projektion? (Formel 4.33$\vee$ Formel 4.34)

Abbildung 4.8: Zusammenfassung der Relevanztests

Kapitel 5

Client-Indexierung mithilfe von PSQ-Anfragen

Das folgende Kapitel diskutiert, wie die registrierten Anfragen auf dem Server gespeichert werden müssen, um die in Kapitel 4 diskutierten Relevanzprüfungen zu ermöglichen. Ziel ist es, die Anfragen als Index für die Clients zu verwenden. Dieser Index kann dann beim Auftreten von Änderungsoperationen genutzt werden, um die IDs der von der Änderung betroffenen Clients zu ermitteln. In Abschnitt 5.1 wird eine Listenrepräsentation vorgestellt, die eine sequentielle Relevanzprüfung ermöglicht. Die Relevanztests basieren aber zum Teil auf Datenbankoperationen, was aufgrund der Mächtigkeit der Anfragesprache und den Restriktionen der in Abschnitt 2 betrachteten Ansätze unumgänglich ist. Insbesondere müssen Verbunde zumindest partiell ausgeführt werden. In Abschnitt 5.2 wird daher untersucht, inwieweit ein Anfrageindex genutzt werden kann, um die Anzahl der Datenbankzugriffe bei der Relevanzprüfung von Änderungen zu minimieren. Dazu wird zuerst ausschließlich auf die Struktur (Syntax) der normalisierten Anfragen zurückgegriffen. Abschnitt 5.3 diskutiert eine erweiterte Variante des Anfrageindexes, welche die Symmetrie der Konjunktion beim Einfügen von Selektionsprädikaten nutzt.

5.1 Sequentielle Speicherung von Anfragen

Die einfachste Möglichkeit, um die Relevanz von registrierten Anfragen zu prüfen besteht in einer sequentiellen Einzelprüfung aller Anfragen. Dazu genügt es, die Anfragen in einer zweispaltigen Tabelle (siehe Beispiel 16) abzuspeichern. Die erste Spalte enthält die registrierten Anfragen. In der zweiten Spalte wird eine Liste der IDs der Clients gespeichert, die die jeweilige Anfrage gestellt haben. Tritt nun eine Änderungsoperation auf, so wird zeilenweise jede Anfrage getestet. Wird eine Relevanz der Änderung für die Anfrage in einer Zeile festgestellt, werden die entsprechenden Clients informiert.

Beispiel 16: *Naive sequentielle Speicherung von Anfragen*

Anfrage	Clients
$\langle$[Filme, laeuft_in, (Filme.FID = laeuft_in.FID)] [Filme.FSK > 16][Filme.Genre =′ Action′] [Filme(Name), laeuft_in(Uhrzeit)]$\rangle$	23,66,20
$\langle$[Filme.FSK >= 18][Filme(Name, FSK, Genre)]$\rangle$	200,11
$\langle$[Filme, laeuft_in, (Filme.FID = laeuft_in.FID)] [Filme(Name), laeuft_in(Datum,Uhrzeit)]$\rangle$	66,200
$\langle$[Filme.FSK >= 18][Filme(Name, FSK)]$\rangle$	45,24

Wie bereits in Kapitel 4 diskutiert, kann eine Änderung für eine Anfrage nur dann relevant sein, wenn die geänderte Relation in das Anfrageergebnis einfließt. Um diese Eigenschaft bei der Relevanzprüfung einfacher testen zu können, ist es sinnvoll, die in den Anfragen verwendeten Relationennamen in einer zusätzlichen Spalte (siehe Beispiel 17) zu speichern.

Beispiel 17: *Sequentielle Speicherung von Anfragen*

Relationen	Anfrage	Clients
Filme, laeuft_in	$\langle$[Filme, laeuft_in, (Filme.FID = laeuft_in.FID)] [Filme.FSK > 16][Filme.Genre =′ Action′] [Filme(Name), laeuft_in(Uhrzeit)]$\rangle$	23,66,20
Filme	$\langle$[Filme.FSK >= 18][Filme(Name, FSK, Genre)]$\rangle$	200,11
Filme, laeuft_in	$\langle$[Filme, laeuft_in, (Filme.FID = laeuft_in.FID)] [Filme(Name), laeuft_in(Datum,Uhrzeit)]$\rangle$	66,200
Filme	$\langle$[Filme.FSK >= 18][Filme(Name, FSK]$\rangle$	45,24

Der Vorteil eines derartigen tabellarischen Ansatzes besteht darin, dass die Datenstruktur direkt in eine relationale Datenbank abgebildet werden kann, wenn dies aus Speicherplatzgründen notwendig ist. Da hierbei aber keine mehrwertigen Attribute erlaubt sind, müssen die Spalten als Attribute vom Typ Zeichenkette betrachtet werden. Da dies nicht sinnvoll

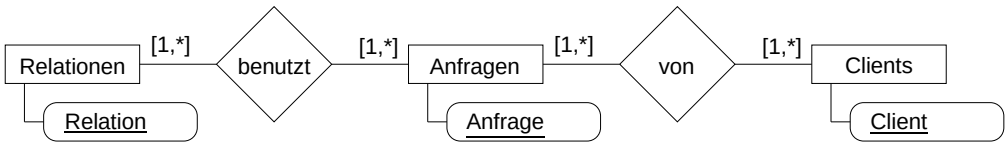


Abbildung 5.1: ER-Schema zur sequentiellen Speicherung von Anfragen

ist, muss das Schema neu modelliert[1] werden (vgl. Abbildung 5.1), wobei durch die beiden m:n-Beziehungen (Anfrage/Relationen, Anfrage/Clients) fünf Tabellen entstehen würden, die

[1] siehe hierzu u. a. [HS00, EN02, CB03, KBL04]

aber aufgrund der einattributigen Entitäten und der gegebenen Kardinalitäten auf eine Relation ClientIndex(Relation,Anfrage,Client) reduziert werden kann. Aus Speicherplatzgründen ist es effizienter zumindest die Entitäten Relationen und Anfragen mit Surrogat-Schlüsseln zu versehen. Damit müssen Anfragen und Relationen nicht für jeden betroffenen Client als Zeichenkette gespeichert werden. Eine andere Alternative besteht in der Verwendung von objektrelationalen Erweiterungen, welche mehrwertige Attribute beispielsweise in Form von Feldern unterstützen [Gep02, Tür03]. Im Rahmen dieser Arbeit wird davon ausgegangen, dass die registrierten Anfragen im Hauptspeicher gehalten werden können, sodass diese Umsetzung hier nicht detailliert diskutiert wird.

5.2 Trie-basierte Indexierung unter Nutzung der Anfrage-Syntax

Die in Abschnitt 5.1 diskutierte sequentielle Auswertung aller Anfragen ist unabhängig von der in Abschnitt 3 eingeführten Anfragerepräsentation. Im Gegensatz dazu wird im Folgenden diskutiert inwieweit die syntaktischen Eigenschaften von PSQ-Anfragen genutzt werden können, um die Relevanz von Anfragen, die gleiche Prädikate verwenden, gemeinsam zu testen. Die Idee ist, dass eine konjunktive Anfrage nur dann relevant ist, wenn kein Prädikat als irrelevant geprüft wird (vgl. Abschnitt 4).

PSQ-Anfragen sind Wörter Q_i über dem Alphabet der zulässigen Prädikate $\mathcal{P} = PP \cup SP \cup VP$. In der Literatur (z. B. [SH99, SS02]) werden für das Suchen in Texten (Mengen von Wörtern) digitale Bäume empfohlen. Ein Vertreter dieser Datenstrukturen ist der *Trie* [Fre59, Fre60]. Bei diesem sind die Kanten die Informationsträger und die Knoten beschreiben, über welches Element des Alphabets (Buchstaben), also über welche Kante, ein Wort fortgesetzt wird. Im Original-Trie enthalten die Knoten dazu ein Feld aller möglichen Buchstaben. Insbesondere bei großen Alphabeten führt dieser Ansatz häufig dazu, dass Knoten Buchstaben enthalten, deren Verweise nicht auf Kindknoten zeigen. Um diese „Verschwendung“ von Speicherplatz zu vermeiden, wird bei der Implementierung von Tries in der Regel eine Beschränkung auf die tatsächlich genutzten Buchstaben in den Knoten vorgenommen. Ein entsprechendes Verfahren wurde in [dlB59] vorgestellt. Darüberhinaus existieren Ansätze zur Reduktion der Tiefe von digitalen Bäumen. Beispiele hierfür sind Compact-Tries [Sus63], Patricia-Bäume [Mor68, Szp90] und Präfix-Bäume [SH99]. Die grundlegende Idee hierbei ist, die in der Baumstruktur enthaltenen Einträge soweit zu reduzieren, dass ihre Anzahl möglichst gering ist, wobei dennoch die Indexierungseigenschaften gewahrt bleiben. Beim Compact-Trie werden beispielsweise nicht verzweigende Teil-Pfade, die zu einem Blatt führen, weggeschnitten. Beim Patricia-Baum werden alle nicht verzweigenden Teil-Pfade durch die Anzahl ihrer Kanten repräsentiert. Präfix-Bäume sind Patricia-Bäume wobei die zusammengefassten Teilwörter (Teil-Pfade) zusätzlich zur Anzahl der gemeinsamen Buchstaben gespeichert werden und statt des ganzen Wortes nur der eindeutige Präfix des Wortes im jeweiligen Blatt enthalten ist. Festzuhalten bleibt, dass Kompressiontechniken wie beim Patricia-Baum oder bei Compact-Tries hier nicht angewendet werden können, da alle Prädikate einer Anfrage die Ir-

relevanz einer Änderungsoperation „entscheiden“ können. Darüberhinaus müssen hier nicht die Zeichenketten, die ein Pfad beschreibt durch diesen referenziert werden, sondern vielmehr die IDs der Clients, welche die Zeichenkette als Anfrage formuliert haben. Daher ist auch ein Präfixbaum nicht direkt verwendbar. Im Folgenden wird von einer angepassten Version des ursprünglichen Tries ausgegangen und eine Indexstruktur entwickelt, welche die syntaktischen Eigenschaften von PSQ-Anfragen ausnutzt.

Die Idee der trie-basierten Indexierung der Clients mithilfe der Anfragen wurde erstmals in [HS03a] publiziert. In [HSS04c, HSS04b][2] wurde ein erster[3] Algorithmus zum Ermitteln der von einem Update betroffenen Clients vorgestellt. Ein Anfrage-Trie basiert nun auf den drei Knotenmengen der Blattknoten B, der inneren Knoten I und des Wurzelknotens $\{root\}$. Jeder Blattknoten enthält eine ID-Liste derjenigen Clients, welche die Anfrage, die durch den Pfad vom Wurzelknoten zu dem Blattknoten repräsentiert wird, registriert haben. Der Wurzelknoten enthält eine nichtleere Liste von Verweisen auf innere Knoten. Diese verweisen auf mindestens einen weiteren inneren Knoten oder einen Blattknoten. Kanten, welche die Verweise auf innere Knoten repräsentieren, sind mit einem Prädikat markiert. Dagegen erfolgt die Markierung von Kanten, die auf Blattknoten verweisen, mithilfe des leeren Prädikats ε.

Definition 5: *Anfrage-Trie*

Ein Anfrage-Trie ist eine Menge $\mathcal{AT} = \{V, E\}$ mit der Knotenmenge $V = B \cup I \cup \{root\}$ und der Kantenmenge $E \subset V \times V \times (\mathcal{P} \cup \{\varepsilon\})$ für die gilt:

- Für alle Knoten $k_1 \in B$ existiert genau ein Knoten $k_2 \in I$, sodass genau eine Kante $k_2 k_1 \varepsilon \in E$ existiert.
- Für alle Knoten $k_1 \in I$ existiert genau ein Knoten $k_2 \in I \cup \{root\}$, sodass genau eine Kante $k_2 k_1 p$ mit $p \in \mathcal{P}$ existiert.
- Es existiert weder eine Kante $k_1 k_1 p$ noch eine Kante $k_1 k_1 \varepsilon$ mit $k_1 \in V$ in E.

Wenn $B = \emptyset$ gilt, muss auch $I = \emptyset$ und $V = \emptyset$ gelten, wodurch der leere Anfrage-Trie $AT = \{\{root\}, \emptyset\}$ definiert ist.

Abbildung 5.2 illustriert den Anfrage-Trie für die Anfragen und Client-IDs aus Beispiel 16. Hierbei wird auch deutlich, dass jede registrierte Anfrage $Q = \langle vp_1 \ \ldots \ vp_n \ sp_1 \ \ldots \ sp_o \ pp \rangle$ als Pfad $root \ k_1 vp_1 \ldots k_{n-1} k_n vp_n k_n k_{n+1} sp_1 \ldots k_{n+o-1} k_{n+o} sp_o k_{n+o} k_{n+o+1} pp$ repräsentiert ist.

Die Relevanzprüfung kann nun beim Traversieren des Baums erfolgen. Wird für ein Prädikat die Irrelevanz bezüglich einer Änderungsoperation festgestellt, so muss der gesamte „darunter-liegende“ Teilbaum nicht näher betrachtet werden. Dadurch und durch die „Zusammenfassung“ gleicher Anfangsprädikate von Anfragen wird die Anzahl der zu prüfenden Prädikate reduziert (vgl. Beispiel 18).

[2] Eine überarbeitete und gekürzte Version des Papiers [HSS04b] erscheint als [HSS04a].

[3] Hierbei wurde ein Update als Kombination aus Delete und Insert betrachtet, was aber, wie in Abschnitt 4.4 verdeutlicht, nicht alle irrelevanten Updates auch als solche erkennen lässt.

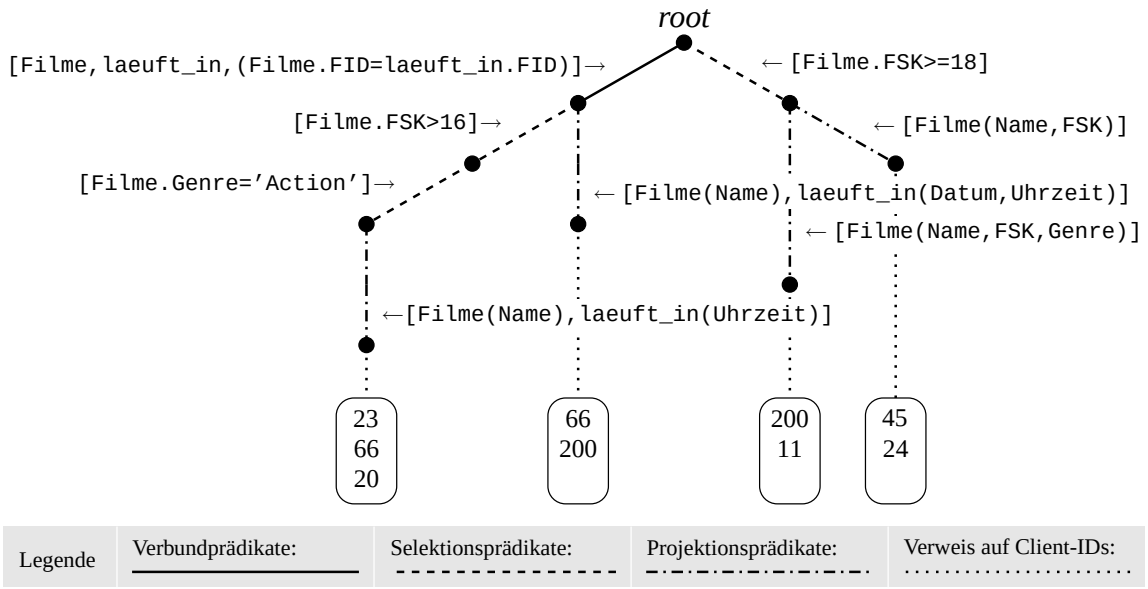


Abbildung 5.2: Anfrage-Trie zur Indexierung von Client-IDs

Beispiel 18: *Vergleich zwischen Trie-basierte und sequentielle Relevanzprüfung*

Seien die Anfragen und die Indexstruktur aus Beispiel 17 und der Anfrage-Trie aus Abbildung 5.2 gegen. Des Weiteren sei angenommen, dass eine Änderungsoperation einen neuen Film in die Relation `Filme` einfügt, der eine Freigabe ab 12 Jahren hat (FSK=12). Da dieses Einfügen eine Relation betrifft, die von jeder der vier Anfragen genutzt wird, müssen im sequentiellen Fall alle Anfragen getestet werden. Die Anzahl der Test ergibt sich wie folgt:

Anfrage 1: Testen des Verbundprädikates $[\texttt{Filme}, \texttt{laeuft\_in}, (\texttt{Filme.FID} = \texttt{laeuft\_in.FID})] \Rightarrow$ irrelevant, da der Film noch nicht in der Relation `laeuft_in` referenziert ist.

Anfrage 2: Testen des Selektionsprädikates $[\texttt{Filme.FSK} > 16] \Rightarrow$ Update ist irrelevant

Anfrage 3: Testen des Verbundprädikates $[\texttt{Filme}, \texttt{laeuft\_in}, (\texttt{Filme.FID} = \texttt{laeuft\_in.FID})] \Rightarrow$ irrelevant, (analog zu Anfrage 1)

Anfrage 4: Test (analog zu Anfrage 2)

Im sequentiellen Fall sind also 4 Prädikattests notwendig. Wird stattdessen der trie-basierte Ansatz gewählt, wird bereits beim Testen der Kanten, die vom Wurzelknoten ausgehen festgestellt, dass das Einfügen für keine der registrierten Anfragen relevant sein kann. Es sind also in diesem Beispiel nur 2 Test durchzuführen.

5.2.1 Praktische Umsetzung der Trie-basierten Indexierung

Wie bereits erwähnt, ist es bei großen Alphabeten nicht sinnvoll einen Trie derart umzusetzen, dass alle Elemente des Alphabets in jedem Knoten enthalten sind. Des Weiteren wurde bisher die im sequentiellen Fall betrachtete Optimierung mithilfe der separaten Speicherung der Relationennamen im Anfrage-Trie nicht betrachtet. Im Folgenden wird ein auf der Umsetzung von Tries nach [dlB59] basierender Anfragebaum $\mathcal{AB}$ hergeleitet. Darüberhinaus wird die Liste der Client-IDs erweitert und als $CA = \{(CID, AID)\}$ nicht in einem separaten Knoten gespeichert, sondern in den jeweilig letzten Prädikatknoten einer Anfrage aufgenommen. Hierbei repräsentiert CID die jeweilige Client-ID und AID ist eine Anfrage-ID (eindeutig für jede Client-ID), sodass die Kombination (CID, AID) global betrachtet eindeutig ist. Im Anfragebaum wird zwischen fünf Knotentypen unterschieden:

Wurzelknoten: Der Wurzelknoten $root = (0, K^c)$ mit der Knotentyp-ID 0 dient als Einstiegspunkt in den Baum und enthält eine Menge von Verweisen K^c auf Relationenknoten (vgl. Abbildung 5.3(a)).

Relationenknoten: Relationenknoten $k_r = (1, name, K^c, k^p)$ mit der Knotentyp-ID 1 repräsentieren je einen Relationennamen $name$ und enthalten eine Menge von Verweisen K^c auf weitere Relationenknoten, Verbund-, Selektions- oder Projektionsknoten (vgl. Abbildung 5.3(b)). Des Weiteren erfolgt eine Rückwärtsverknüpfung (k^p) mit dem Vaterknoten (entweder ein Relationenknoten oder der Wurzelknoten). Ausgehend vom Wurzelknoten unterliegen die Relationenknoten einer steigenden lexikographischen Ordnung.

Verbundknoten: Verbundknoten $k_v = (2, vp, A_v, K^c, CA, k^p)$ mit der Knotentyp-ID 2 repräsentieren je ein Verbundprädikat $vp \in VP$. Wie in Kapitel 4 diskutiert, müssen Verbunde zum Testen partiell ausgeführt werden. Um den Speicherverbrauch hierbei zu minimieren wird eine Projektion der Attribute, welche in den Prädikaten im Teilbaum ausgehend vom Verbundprädikat verwendet werden, vorgenommen. Um dies effizient realisieren zu können, enthält der jeweilig letzte (ausgehend von der Wurzel) Verbundknoten jedes Pfades die Menge A_v dieser Attribute. Darüberhinaus enthält jeder Verbundknoten einen Verweis k^p auf seinen Vaterknoten (Relationenknoten oder Verbundknoten) und eine Menge von Verweisen K^c auf Projektions-, Selektions- oder weitere Verbundknoten (vgl. Abbildung 5.3(c)).

Selektionsknoten: Selektionsknoten $k_s = (3, sp, K^c, CA, k^p, TR)$ mit der Knotentyp-ID 3 repräsentieren je ein Selektionsprädikat $sp \in SP$ (vgl. Abbildung 5.3(d)). Daneben enthalten sie eine Menge von Verweisen K^c auf Projektions- oder weitere Selektionsknoten sowie einen Rückverweis k^p auf ihren jeweiligen Vaterknoten (Selektions-, Verbund- oder Relationenknoten). Da zum Testen der Relevanz von Selektionsprädikaten ggf. auf das temporäre Ergebnis TJ des Testes von Verbundprädikaten zurückgegriffen werden muss, wird dieses in den Selektionsknoten als TR referenziert.

Projektionsknoten Projektionsknoten $k_p = (4, pp, k^p, CA, TR)$ mit der Knotentyp-ID 4 repräsentieren je einen Projektionsprädikat $pp \in PP$ und enthalten eine Rückverweis

(vgl. Abbildung 5.3(e)) auf den jeweiligen Vaterknoten (Selektions-, Verbund- oder Relationsknoten). Zum Testen der Relevanz von Projektionsprädikaten muss ggf. auf das Ergebnis TJ bzw. TS vorheriger Tests von Verbund- oder Selektionsprädikaten zurückgegriffen werden. Daher wird dieses in den Projektionsknoten als TR referenziert.

Die Rückwärtsverkettung ist für die eigentliche Relevanzprüfung nicht zwingend notwendig, vereinfacht aber die Verwaltung des Anfragebaums, die Gegenstand von Abschnitt 5.2.2 ist. Ein Anfragebaum ist formal wie folgt definiert:

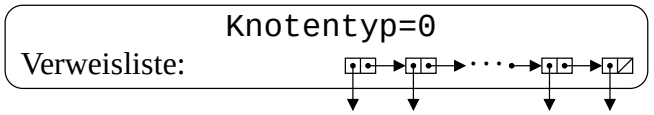
Definition 6: *Anfragebaum*

Sei der Wurzelknoten $root$ und die Knotenmengen K_r (Relationenknoten), K_v (Verbundknoten), K_s (Selektionsknoten) und K_p (Projektionsknoten) gegeben. Ein Graph $\mathcal{AB} = \{V, E\}$ repräsentiert einen Anfragebaum mit der Knotenmenge $V = \{root\} \cup K_r \cup K_v \cup K_s \cup K_p$ wenn für die Kantenmenge $E \subset V \times V$ gilt:

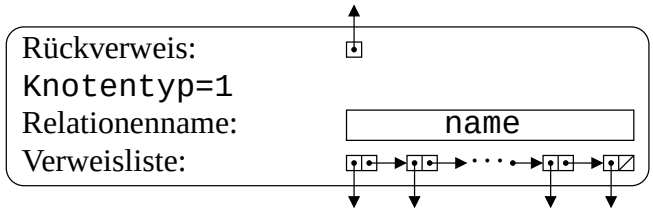
- Für alle Knoten $k_p \in K_p$ existiert genau ein Knoten $k \in K_r \cup K_v \cup K_s$, sodass die Kanten $k\ k_p \in E$ und $k_p\ k \in E$ existieren. Des Weiteren muss für alle $k_p \in K_p$ $CA \neq \emptyset$ gelten.
- Für alle Knoten $k_s \in K_s$ existiert genau ein Knoten $k \in K_s \cup K_v \cup K_r$ und eine Menge von Knoten $C \subseteq K_s \cup K_p$, sodass die Kanten $k_s\ k \in E$ und $k\ k_s \in E$ existieren. Darüberhinaus muss für alle Knoten $c \in C$ mit $C \neq \emptyset$ sowohl eine Kante $c\ k_s \in E$ als auch eine Kante $k_s\ c \in E$ existieren. Wenn hingegen $C = \emptyset$ gilt, dann muss $CA \neq \emptyset$ gelten.
- Für alle Knoten $k_v \in K_v$ existiert genau ein Knoten $k \in K_v \cup K_r$ und eine Menge von Knoten $C \subseteq K_v \cup K_s \cup K_p$, sodass die Kanten $k_v\ k \in E$ und $k\ k_v \in E$ existieren. Darüberhinaus muss für alle Knoten $c \in C$ mit $C \neq \emptyset$ sowohl eine Kante $c\ k_v \in E$ als auch eine Kante $k_v\ c \in E$ existieren. Wenn hingegen $C = \emptyset$ gilt, dann muss $CA \neq \emptyset$ gelten.
- Für alle Knoten $k_r \in K_r$ existiert genau ein Knoten $k \in K_r \cup \{root\}$ und eine Menge von Knoten $C \subseteq K_r \cup K_v \cup K_s \cup K_p$, sodass die Kanten $k_r\ k \in E$ und $k\ k_r \in E$ existieren. Darüberhinaus muss für alle Knoten $c \in C$ mit $C \neq \emptyset$ sowohl eine Kante $c\ k_r \in E$ als auch eine Kante $k_r\ c \in E$ existieren. Wenn hingegen $C = \emptyset$ gilt, dann muss $CA \neq \emptyset$ gelten.
- Für keine zwei Knoten $k_1, k_2 \in E$ mit $k_1 = k_2$ existiert eine Kante $k_1\ k_2 \in V$ oder $k_2\ k_1 \in V$.
- Für alle Pfade $root\ k_{r_1}\ \ldots k_{r_n}\ c$ mit $c \in K_v \cup K_s \cup K_p$ muss für alle $k_{r_i} = (0, name_i, K_i^C, k_i^p) \in K_r$ und $k_{r_j} = (0, name_j, K_j^C, k_j^p) \in K_r$ mit $1 \leq i < j \leq n$ die lexikographische Ordnung $name_i \triangleleft name_j$ gelten.

Der leere Anfragebaum ist definiert als $AB = \{\{root\}, \emptyset\}$.

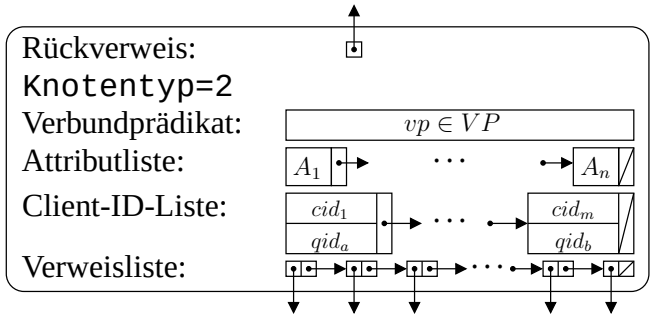
Abbildung 5.4 zeigt den Anfragebaum für die Anfragen aus Beispiel 16.



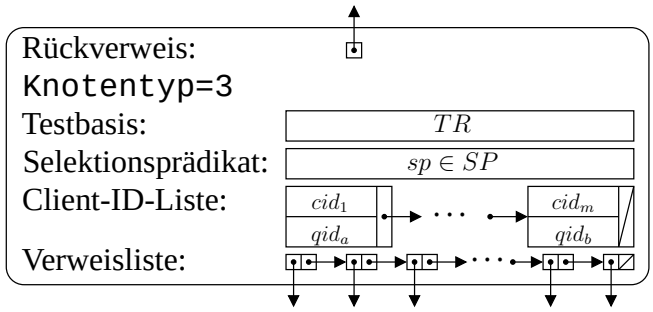
(a) Wurzelknoten



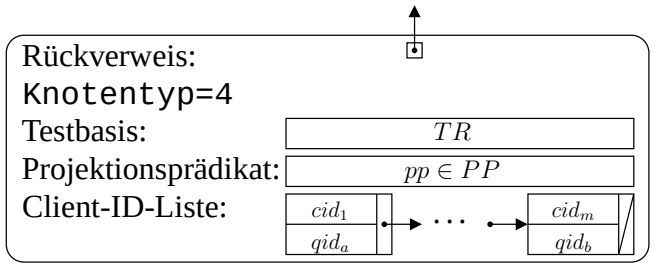
(b) Relationenknoten



(c) Verbundknoten

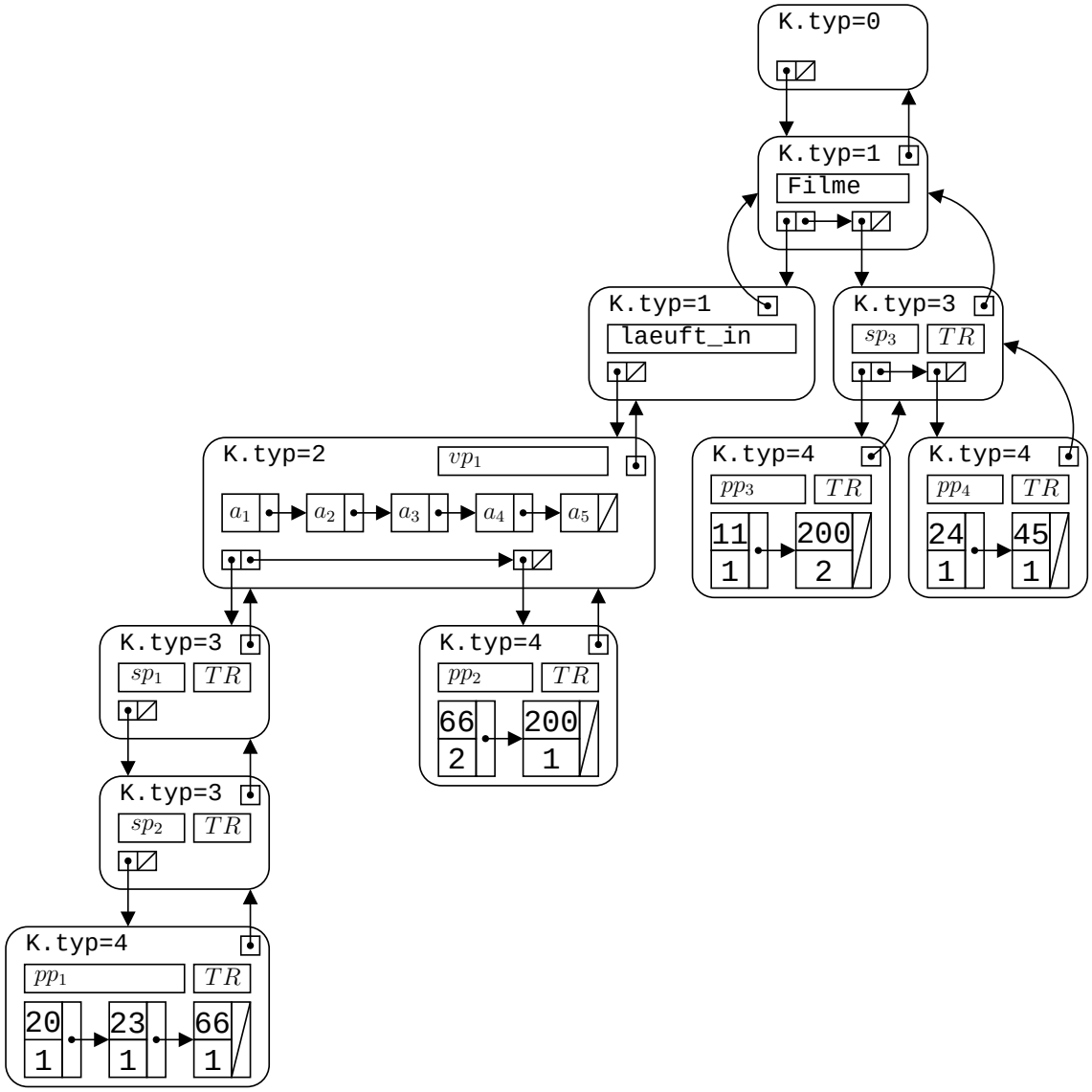


(d) Selektionsknoten



(e) Projektionsknoten

Abbildung 5.3: Schematische Aufbau der Knotentypen



vp_1 =[Filme,laeuft_in,(Filme.FID=laeuft_in.FID)]
sp_1 =[Filme.FSK>16]
sp_2 =[Filme.Genre='Action']
sp_3 =[Filme.FSK>=18]
pp_1 =[Filme(Name),laeuft_in(Uhrzeit)]
pp_2 =[Filme(Name),laeuft_in(Datum,Uhrzeit)]
pp_3 =[Filme(Name,FSK,Genre)]
pp_4 =[Filme(Name,FSK)]
a_1 =Filme.FSK a_2 =Filme.Genre a_3 =Filme.Name
a_4 =laeuft_in.Uhrzeit a_5 =laeuft_in.Datum

Abbildung 5.4: Anfragebaum zur Indexierung von Client-IDs

5.2.2 Algorithmen zur Verwaltung des Anfragebaums

In Kapitel 1 wurde das grundlegende Einsatzszenario der Relevanzprüfung diskutiert. Mobile Clients registrieren Anfragen, die in den Anfragebaum eingetragen werden müssen. Andererseits muss auch die Möglichkeit bestehen, die Registrierung von Anfragen aufzuheben. Im Gegensatz zur tabellarischen Speicherung bedarf die Speicherung der Anfragen in einem Anfragebaum spezieller Algorithmen, die sicherstellen, dass durch Einfügen und Löschen von Anfragen die unter Abschnitt 5.2.1 diskutierten Eigenschaften des Baumes gewahrt bleiben.

Hilfstruktur für das Bottom-up-Durchlaufen des Anfragebaumes

Da die Client-IDs und Anfrage-IDs in den Blättern des Anfragebaumes gespeichert sind, muss beim Löschen der gesamte Baum traversiert werden. Um dies zu vermeiden, wird im Folgenden eine Hilfstruktur definiert, um ein Bottom-up-Durchlaufen zu ermöglichen[4]. In [Höp04] wurde hierfür eine Liste vorgeschlagen, die nach den Clients-IDs sortiert vorliegt. Da aber das Auffinden eines Elementes in einer Liste im schlechtesten Fall bedeutet, dass die gesamte Liste durchlaufen werden muss, ist ein AVL-Baum [AVL62a] besser geeignet. Da es sich hierbei um einen ausgeglichenen Binärbaum handelt, kann in ihm mit binärer Suche [Kho90] effizient gesucht werden. Pro Client-ID existiert ein Knoten in diesem Blattknoten-Indexbaum (BKI). Diese enthalten wiederum einen AVL-Baum, der die Anfrage-IDs der durch diesen Client registrierten Anfragen speichert und auf die Knoten des Anfragebaums verweist, welche das Ende der entsprechenden Anfrage darstellen. Abbildung 5.5 illustriert den Zusammenhang zwischen

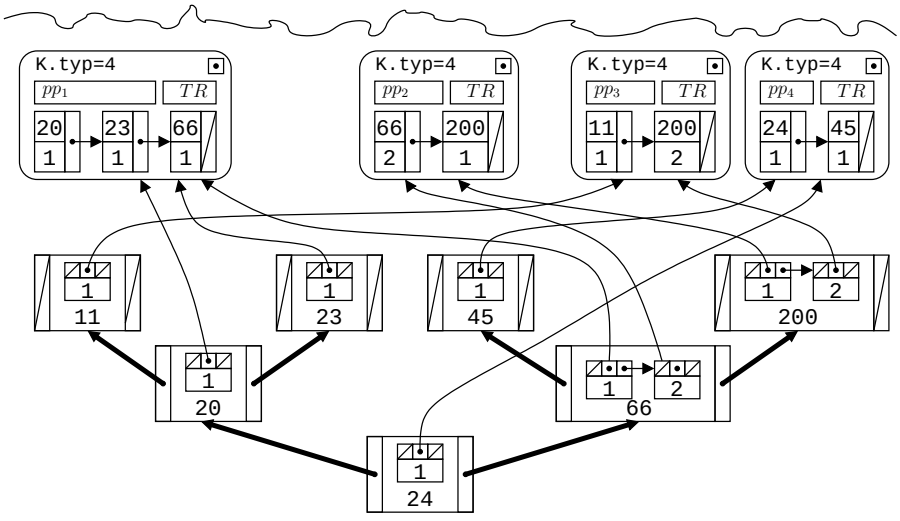


Abbildung 5.5: Hilfstruktur zur direkten Indexierung der Blätter

BKI und Anfragebaum. Da die Algorithmen zur Verwaltung eines AVL-Baumes in der gän-

[4]Ein ähnlicher Ansatz wurde in [LHJ+03] für das effiziente Aktualisieren von R-Bäumen [Gut84a] und R*-Bäumen [BKSS90a] verfolgt.

gigen Literatur zu Algorithmen und Datenstrukturen (z. B. [SS02]) nachgeschlagen werden können, wird hier nicht weiter auf sie eingegangen.

Registrieren neuer Anfragen

Um eine neue Anfrage zu registrieren, übermittelt der mobile Client die Anfrage und seine Client-ID an den Server. Wurde für einen Client noch keine ID vergeben, so wird diese serverseitig generiert und mit dem Anfrageergebnis an den Client übermittelt. Des Weiteren wird eine für den Client eindeutige Anfrage-ID generiert und diesem mitgeteilt.

Algorithmus 1: *Vorbereiten einer PSQ-Anfrage zum Einfügen*

Eingabe: Q // zu registrierende Anfrage
Ausgabe: EQ // erweiterte PSQ-Anfrage

```
01  RN = ⟨⟩        // Initialisierung der Sequenz, die die Relationennamen aufnimmt
02  EQ = ⟨⟩        // Initialisierung der Sequenz, die die vorbereitete Anfrage repräsentiert
03
04  def prepare_query(Q)
05      let Q' be Q mit Attribut- und Relationennamen in Großschrift
06      for each p ∈ Q' do // alle Prädikate der Anfrage durchlaufen
07          RN = RN ∪ RN^Q' // RN^Q' ist die Menge Relationennamen aus Q
08          if p ∈ VP then // p ist ein Verbundprädikat
09              EQ = EQ ∘ ⟨(p, 2)⟩
10          if p ∈ SP then // p ist ein Selektionsprädikat
11              EQ = EQ ∘ ⟨(p, 3)⟩
12          if p ∈ PP then // p ist ein Projektionsprädikat
13              EQ = EQ ∘ ⟨(p, 4)⟩
14      temp = ⟨⟩ // Initialisierung der temporären Relationennamen-Liste
15      for each rn ∈unique(sort(RN)) do // Relationennamen durchlaufen
16          temp = temp ∘ ⟨(rn, 1)⟩
17      EQ = temp ∘ EQ
18      return(EQ)
```

Um eine neue Anfrage in den Index einzutragen, wird diese im ersten Schritt vorverarbeitet. Hierbei wird eine Anfrage[5] mit r Relationennamen, n Verbund-, o Selektionsprädikaten und einem Projektionsprädikat in die Form $EQ = \langle(name_1, 1)(name_r, 1)(vp_1, 2)(vp_n, 2)(sp_1, 3)(sp_o, 3)(pp, 4)\rangle$ überführt. Algorithmus 1 illustriert das dazu notwendige Vorgehen wobei das Zeichen $\circ$ für die Konkatenation von zwei Zeichenketten steht. Der Algorithmus beinhaltet drei wesentliche Schritte:

[5] Das Vorgehen bei anderen Anfragen ist analog.

- Relationen- und Attributnamen werden einheitlich in Großbuchstaben angegeben (Zeile 05). Dadurch kann beim späteren Einfügen der Anfrage in den Baum sichergestellt werden, dass die unterschiedliche Benutzung von Groß- und Kleinschreibung nicht zu unnötigen Verzweigungen im Anfragebaum führt. (vgl. hierzu auch Anhang A.1.1)
- Die Prädikattypen werden ermittelt (Zeilen 08-13, vgl. hierzu auch Anhang A.1.2). Dadurch kann beim eigentlichen Einfügen der Anfrage in den Anfragebaum der Knotentyp festgelegt werden.
- Die Relationnamen werden extrahiert (Zeile 07, vgl. hierzu auch Anhang A.1.3) und den Prädikaten der Anfrage vorangestellt (Zeile 17). Dies ist notwendig, da der Anfragebaum separate Relationenknoten enthält.

Diese erweiterte Anfrage wird mithilfe von Algorithmus 2 in den Anfragebaum integriert. Hierzu wird zuerst durch die rekursiven Funktion *find_last_equal_node* (Zeilen 01-08) bestimmt, welche n Prädikate bzw. Relationennamen am Anfang der vorverarbeiteten Anfrage EQ bereits im Baum enthalten sind. Anschließend wird anhand des Blattknoten-Indexes eine neue Anfrage-ID (AID) ermittelt. Ist bereits die gesamte Anfrage im Baum enthalten, muss untersucht werden, ob diese durch den Client mit der Client-ID CID mehrfach gestellt wurde (Zeilen 17-18). Dies kann daran festgestellt werden, dass im BKI bereits ein Eintrag für die Client-ID vorhanden ist, der auf den Blattknoten k des Anfragepfades verweist. In diesem Fall kann das Einfügen abgebrochen werden (Zeile 19). War die Anfrage hingegen entweder nur teilweise (oder gar nicht) im Baum repräsentiert oder wurde sie durch andere Clients registriert, so muss (CID, AID) in den Blattknoten k aufgenommen werden (Zeile 20). Des Weiteren müssen CID, AID mit dem Verweis auf k im BKI eingetragen werden (Zeile 21). Enthält die Anfrage sowohl Verbundprädikate als auch Selektions- oder Projektionsprädikate (Zeile 22), so müssen im letzten Verbundknoten die jeweilig verwendeten Attributnamen (vgl. Anhang A.2.1) aufgelistet werden (Zeile 23).

Algorithmus 2: *Einfügen einer neuen Anfrage in den Anfragebaum*

Eingabe:	EQ	// vorverarbeitete Anfrage
	CID	// Client-ID
	$root$	// Wurzelknoten des Anfragebaums
Ausgabe:	AID	// Anfrage-ID

```
01 def find_last_equal_node(node, EQ, n)
02   if node.K^c ≠ ∅ ∧ n < |EQ|
03     for each child c ∈ node.K^c
04       let p be das n-te Element von EQ
05       if c.value == p.value ∧ c.type == p.type
06         return(find_last_equal_node(c, EQ, n + 1))
07     return(node,n)
```

```
08    return(node, n)
09
10  def insert_path(EQ, CID):
11    (node, pn)=find_last_equal_node(root, EQ, 0) // Teilbaum zum Einhängen finden
12    k = node // aktuell letzten Knoten merken
13    neue AID mithilfe des BKIs anfordern
14    if pn < |EQ| // Anfrage noch nicht komplett im Baum
15      Einfügen der Teilanfrage ab Prädikat Nummer pn + 1
16      let k be Blattknoten des Pfades der Anfrage
17    else // Anfrage ist bereits vollständig im Baum
18      if k in BKI für CID referenziert
19        break // Anfrage wurde mehrfach durch Client gestellt
20    k.CA = k.CA ∪ {(CID, AID)} // Eintragen von AID und CID in k
21    Eintragen der AID mit Referenz auf k im BKI für CID
22    if p1 in EQ, p2 ∈ EQ |p1 ∈ VP ∧ p2 ∈ SP ∪ PP
23      Eintragen der Attributnamen der Selektions- und Projektionsprädikaten
        in den letzten Verbundknoten
24    return(AID)
```

Abbestellen von Anfragen

Wie bereits erwähnt, überträgt ein Client zum Abbestellen von registrierten Anfragen seine Client-ID und die entsprechende Anfrage-ID. Mithilfe des Blattknoten-Indexes kann das Austragen einer Anfrage nun bottom-up erfolgen. Zuerst wird der entsprechende „Blattknoten“ der Anfrage ermittelt. Aus dessen Menge CA, wird nun der entsprechende Eintrage entfernt. Ist anschließend CA leer, so endet an diesem Knoten keine weitere Anfrage. Hat er darüberhinaus keine Kindknoten, so kann er gelöscht werden. Gleiches gilt für seinen Vaterknoten und dessen Vaterknoten solange keiner dieser Knoten weitere Kindknoten hat oder durch eine nicht leere Menge CA das Ende einer anderen Anfrage beschreibt. Abschließend muss noch der Eintrag für diese Anfrage im BKI entfernt werden.

Algorithmus 3: *Entfernen einer Anfrage aus dem Anfragebaum*

```
Eingabe:  CID   // Client-ID
          AID   // Anfrage-ID

01  def delete_path(CID, AID)
02    let node be „Blattknoten“ der Anfrage
03    node.CA = node.CA − {(CID, AID)} // Anfrageende aus Anfragebaum austragen
04    free_nodes(node)
```

```
05     {(CID, AID)} aus BKI entfernen
06
07  def free_nodes(node)
08     if node.CA == ∅ ∧ node.K^c == ∅
09        parent = node.k^p
10        parent.K^c = parent.K^c − node
11        free(node)
12        free_nodes(parent)
```

5.2.3 Algorithmen zur Relevanzprüfung mithilfe des Anfragebaums

Im Gegensatz zu den in den Abschnitten 2.2 und 2.3 diskutierten Ansätzen zur Bestimmung der Relevanz von Änderungsoperationen, wird in der vorliegenden Arbeit direkt das Datenbanksystem benutzt. Es ist also kein zusätzlicher Regelbeweiser, der die Erfüllbarkeit von logischen Ausdrücken testet, notwendig. Relevanztests erfolgen hier mithilfe spezieller, SQL-notierter Testanfragen, deren Ergebnisse Aufschluss über Relevanz oder Irrelevanz einer Änderung bezüglich einer Menge von Prädikaten geben. Durch Kombinieren mehrerer solcher Prädikattests ist es somit möglich, zu prüfen, ob das Ergebnis einer PSQ-notierte Anfrage von einer Änderung betroffen ist. Im Folgenden wird diskutiert, wie diese Testanfragen ausgewertet werden.

Den Rahmen bilden Algorithmus 4 und die drei Mengen CJ, CS und CP. Diese enthalten während des Testens der Relevanz einer Änderungsoperation jeweils diejenigen Verbund- (CJ), Selektions- (CS), und Projektionsknoten (CP), welche noch getestet werden müssen. Initialisiert werden die drei Mengen nach den globalen Relevanztests (Zeile 07) und dem Test der Relationenknoten (Zeilen 08,09 ; vgl. Algorithmus 5). Das heißt, dass alle (pfadweise) ersten „Nicht-Relationenknoten“, die sich in einem Teilbaum ab einem relevanten Relationenknoten befinden, in diese Mengen einsortiert werden (Zeile 12). Anschließend werden die Verbunde in CJ getestet (Zeilen 17, 18). Wird ein relevanter Verbund gefunden, so werden wiederum die ersten Kindknoten des jeweiligen Teilbaums in CS bzw. CP eingefügt. Abgearbeitete Verbundknoten werden aus CJ entfernt. Ist CJ leer, so wird CS abgearbeitet (Zeilen 19, 20) und gegebenenfalls CP erweitert. Ist auch CS leer, werden die Projektionsknoten in CP analysiert (Zeilen 21, 22). Wird bei der Abarbeitung der drei Mengen festgestellt, dass die Änderung für eine Anfrage relevant ist, so werden die entsprechenden Client-IDs und Anfrage-IDs zur Menge RC, der zu benachrichtigenden Clients hinzugefügt.

Die diesem Vorgehen zugrundeliegende Rekursion wird durch die in Algorithmus 4 enthaltene (Zeilen 16-22) und aufgerufene (Zeile 13) Funktion *check_predicates* gestartet.

Algorithmus 4: *Relevanztest mithilfe eines Anfragebaums*

Eingabe: UR // Änderungsoperation

	$root$	// Wurzelknoten des Anfragebaums
Ausgabe:	RC	// Menge CID-AID-Tupeln betroffener Anfragen

```
01 def check_relevance(root, UR)
02     RC = ∅ // Initialisierung der CID-AID-Tupelmenge
03     mt =get_modification_type(UR) // Art der Änderung ermitteln
04     name=extract_relation_from_modification(UR, mt) // Name der geänderte Relation
05     if mt == 3 // Änderung ist ein Update
06         r(R_name^u) = ur(r(R_name)) // Erzeugen der temporär aktualisierten Tabelle
07     if check_global_relevance(name,name^u, mt, UR) // globale Relevanztests
08         RN = ∅ // Initialisierung der Menge zu testender „Einstiegsprädikate“
09         check_relation_nodes(name, root, RN, false) // Aufruf von Algorithmus 5
10         if RN ≠ ∅ // geänderte Relation wird von mind. einer Anfrage benutzt
11             CS = ∅; CJ = ∅; CP = ∅ // Initialisierung der klassifizierten Knotenmengen
12             classify_collection(RN, CJ, CS, CP) // Klassifikation von RN
13             check_predicates(UR, CJ, CS, CP, mt, RC, name, name^u) // Präd. prüfen
14     return(RC)
15
16 def check_predicates(UR, CJ, CS, CP, mt, RC, name, name^u)
17     if CJ ≠ ∅ // zuerst Verbundknoten abarbeiten
18         check_joins(UR, CJ, CS, CP, mt, RC, name, name^u) // Algorithmus 6
19     else if CS ≠ ∅ // als zweites Selektionsknoten abarbeiten
20         check_selection(UR, CJ, CS, CP, mt, RC, name, name^u) // Algorithmus 7
21     else if CP ≠ ∅ // am Ende Projektionsknoten abarbeiten
22         check_projections(UR, CJ, CS, CP, mt, RC, name, name^u) // Algorithmus 8
```

Globale Tests

Unter den globalen Relevanztests ist das Prüfen von Eigenschaften einer Änderungsoperation, die mit einer speziellen Anfrage zusammenhängen, zu verstehen. Sie wurden bereits in Kapitel 4 vorgestellt und sind auch in Abbildung 4.8 auf Seite 66 aufgeführt. Demnach kann ein Einfügen nur dann relevant sein, wenn das einzufügende Tupel noch nicht in der ursprünglichen Relation enthalten ist. Eine Löschoperation kann nur dann relevant sein, wenn tatsächlich Tupel gelöscht werden und auch ein Update muss zu tatsächlichen Veränderungen führen. Erfüllt eine Änderungsoperation die entsprechende Eigenschaft nicht, so muss auch keine weitere Relevanzprüfung durchgeführt werden.

Testen der Relationennamen

Wird die zu ändernde Relation nicht durch eine Anfrage verwendet, so ist die Änderung für diese Anfrage garantiert irrelevant. Um dies zu testen, wird der Relationenname name aus

der SQL-Anweisung der Änderungsoperation extrahiert und durch Top-down-Traversierung mit den Werten der Relationenknoten verglichen. Wie in Algorithmus 5 verdeutlicht, ist das Ergebnis eine Menge RN von Zeigern auf den jeweilig ersten Prädikatknoten in Pfaden in denen name vorkommt. Hierbei wird die lexikographische Ordnung der Relationennamen innerhalb eines Pfades derart genutzt, dass die Traversierung eines Teilbaums abgebrochen werden kann, wenn ein Relationenknoten k_r erreicht wird, für den $k_r.value \triangleleft$ name gilt und wenn für den jeweiligen Pfad bisher keine Relevanz festgestellt wurde.

Algorithmus 5: *Testen der Relationennamen*

Eingabe: name // Name der geänderten Relation
$node$ // aktuell betrachteter Knoten des Anfragebaums
Ausgabe: RN // Menge von Zeigern auf Prädikatknoten

```
01 node = root   // Starten am Wurzelknoten
02 RN = ∅        // Zeigermenge initialisieren
03 fnd = false   // Initialisierung eines Merkers, ob Pfad relevant
04
05 def check_relation_nodes(name, node, RN, fnd)
06    for each c ∈ node.K^c
07       if ¬fnd ∧ c.value == name ∧ c ∈ K_r
08          check_relation_nodes(name, c, RN, true)
09       elif fnd ∧ c.type ≠ 1
10          RN = RN ∪ {c}
11       elif c ∉ K_r ∧ (fnd ∨ name ▷ c.value)
12          check_relation_nodes(name, c, RN, fnd)
```

Testen der Relevanz für Verbundprädikaten

Die grundlegende Idee beim Testen der Relevanz von Verbundprädikaten besteht darin, zu prüfen, ob $TJ \neq \emptyset$ gilt. Wie in Kapitel 4 eingeführt umfasst TJ diejenigen Tupel, die durch die Änderungsoperation zum Verbundergebnis hinzukommen oder aus diesem wegfallen. Um dies zu prüfen ist es zuerst notwendig die Verbundkette aufzubauen und anschließend die entsprechende Testanfrage zu generieren und auszuwerten (vgl. Algorithmus 6). Die Berechnung von TJ in Zeile 05 ist abhängig vom Typ der Änderung und beinhaltet im Wesentlichen die Umsetzung der in Kapitel 4 vorgestellten Algebraausdrücke mithilfe von SQL. Daher wird auf eine detaillierte Beschreibung hier verzichtet. Es sei aber darauf hingewiesen, dass im Falle eines Updates dieses temporär ausgeführt werden muss. Dazu wurde in Algorithmus 4 eine temporäre Relation $r(R_{\texttt{name}^u})$ angelegt, welche die Tupel der Relation $r(R_{\texttt{name}})$ enthält, und das Update auf $r(R_{\texttt{name}^u})$ ausgeführt. Des Weiteren wurde hier zur Vereinfachung die Funktion (Zeile 02) zum bestimmen aller Teilpfade von dem aktuell zu prüfenden Verbundknoten

zu den ersten „Nicht-Verbundknoten“ des Teilbaums nicht erläutert. Die Realisierung dieser Teilaufgabe ist in Anhang A.3 enthalten.

Durch die Rekursion, welche in Algorithmus 4 gestartet wurde und in Zeile 18 von Algorithmus 6 fortgesetzt wird, wird die Funktion *check_joins* solange aufgerufen, bis die Menge CJ leer ist.

Ein wesentliches Problem bei der Umsetzung dieses Algorithmus' mithilfe von SQL-Anfragen besteht darin, dass das Ergebnis eines Verbundes mehrere gleichnamige Attribute enthalten kann. Dies tritt dann auf, wenn die verbundenen Relationen gleichnamige Attribute umfassen. Um bei der späteren Prüfung von Selektions- und Projektionsprädikaten dennoch eine genaue Spezifikation der jeweils zu betrachtenden Attribute zu ermöglichen, wird eine interne Attributumbenennung vorgenommen. Aus `Relationenname.Attributname` der PSQ-Notierten Anfrage wird ein SHA-Hash-Wert berechnet und mit – einer vorgestellten Zeichenkette[6] – anstelle des jeweiligen Attributnamens verwendet. Beim späteren Prüfen weiterer Prädikate wird dann ebenfalls der jeweilige SHA-Hash-Wert [SHA95] der Attributname-Relationennamen-Kombination mit vorgestellter Zeichenkette benutzt. Im Falle von Selbstverbunden schließt die SHA-Berechnung auch den verwendeten Alias ein. Dadurch bleibt die in PSQ geforderte eindeutige Zuordnung der Attribute zu den Ursprungstabellen erhalten. Darüberhinaus ermöglicht dieses Vorgehen auch das Materialisieren des Verbundergebnisses (vgl. Kapitel 6).

Algorithmus 6: *Testen der Verbundprädikate*

Eingabe:	`name`	// Name der zu ändernden Relation
	`name`^{u}	// Name der temporär geänderten Relation (nur bei Update)
	UR	// Update-Operation
	CJ	// Menge der noch zu betrachtenden Verbundknoten
	CS	// Menge der noch zu betrachtenden Selektionsknoten
	CP	// Menge der noch zu betrachtenden Projektionsknoten
	RC	// bisher zu benachrichtigende Clients
	mt	// Typ der Änderungsoperation
Ausgabe:	RC	// ggf. erweiterte Menge von Client-IDs
	CJ, CS, CP	// angepasste Mengen noch zu testender Knoten

01 **def** *check_joins(*UR, CJ, CS, CP, mt, RC, `name`, `name`^{u}*)*
02 **let** $joins$ **be** Menge aller Verbundketten des Teilbaums
03 **for each** $join \in joins$ **do**
04 **let** $node$ **be** letzter Verbundknoten der aktuellen Verbundkette
05 $TJ =$*check_join_relevance(*UR,$join$,mt, `name`, `name`^{u}*)*
06 **if** $TJ \neq \emptyset$ // bisher keine Irrelevanz feststellbar

[6] Die hexadezimale Darstellung eines SHA-Hash-Wertes kann mit einer Ziffer beginnen. Um den Wert dennoch als Attributnamen verwenden zu können muss mindestens ein Buchstaben vorangestellt werden.

```
07        if node.K^c ≠ ∅ // Anfrage benutzt weitere „Nicht-Verbund“-Prädikate
08        for each child ∈ node.K^c
09            if child.type == 3 // Kind ist ein Selektionsknoten
10                CS = CS ∪ child
11                child.TR = TJ // TJ zum weiteren Testen merken
12            if child.type == 4 // Kind ist ein Projektionsknoten
13                CP = CP ∪ child
14                child.TR = TJ // TJ zum weiteren Testen merken
15        if node.CA ≠ ∅ // Aktueller Knoten enthält Client-IDS
16            RC = RC ∪ node.CS
17    CJ = CJ − joins // Aktueller Teilbaum wurde getestet
18    check_predicates(UR, CJ, CS, CP, mt, RC, name, name^u) // Rekursion
```

Testen der Relevanz für Selektionsprädikate

Das Testen der Selektionsprädikate erfolgt analog zum Testen der Verbundprädikate durch sukzessives Abarbeiten der Menge SJ, wobei geprüft wird, ob $TS \neq \emptyset$ gilt. Ein wesentlicher Unterschied ist aber der, dass „Ketten“ von Selektionsprädikaten im Gegensatz zu Verbundketten zerlegt werden können. Seien $sp_1, sp_2, \ldots, sp_n$ alle Selektionsprädikate einer Anfrage Q. Eine Änderung kann für diese Teilanfrage nur dann relevant sein, wenn sie für sp_1, für $sp_1 \wedge sp_2$, bis hin zu $sp_1 \wedge sp_2 \wedge \cdots \wedge sp_3$ relevant ist. Selbstverständlich kann beim Testen der Teilanfragen auf das Ergebnis der vorherigen Tests zurückgegriffen werden. Hat die Anfrage auch Verbundprädikate, so startet die Testserie mit $TS = sp_1(TJ)$. Der zweite Schritt ist dann $TS = sp_2(TS)$ usw.. Bei Anfragen ohne Verbundprädikate ist der Anfang abhängig vom Type der jeweiligen Änderungsoperation (vgl. Kapitel 4). Der Folgeablauf ist aber analog. Offensichtlich sorgt dieses Vorgehen dafür, das mehrere Anfragen mit dem gleichen Präfix solange zusammen betrachtet werden, bis sie sich in einem Selektionsprädikat unterscheiden. Um hierbei nicht alle Teilschritte berechnen zu müssen werden Selektionsprädikate, welche durch Knoten mit nur einem Nachfolger repräsentiert sind, solange zusammengefasst, bis ihr Knoten durch ein nicht leeres CA das Ende mindestens einer der gemeinsam betrachteten Anfragen signalisiert. Damit alle notwendigen Selektionsprädikate abgearbeitet werden, werden beim Bestehen einer Relevanz für eine Teilanfrage, alle Kindknoten des entsprechenden Pfades in CS eingetragen und das bisherige TS an die Kinder weitergereicht.

Im Fall von Einfüge- und Löschoperationen wird darüberhinaus das zuletzt berechnete TS an ggf. folgende Projektionsknoten weitergegeben. Bei Update-Operationen ist dies nicht sinnvoll, da das Testen der Projektionsprädikate hierbei nicht auf TS basiert (vgl. Abschnitt 4.4).

Algorithmus 7: *Testen der Selektionsprädikate*

Eingabe: `name` // Name der zu ändernden Relation

	$\texttt{name}^u$	// Name der temporär geänderten Relation (nur bei Update)
	UR	// Update-Operation
	CS	// Menge der noch zu betrachtenden Selektionsknoten
	CP	// Menge der noch zu betrachtenden Projektionsknoten
	RC	// bisher zu benachrichtigende Clients
	mt	// Typ der Änderungsoperation
Ausgabe:	RC	// ggf. erweiterte Menge von Client-IDs
	CS, CP	// angepasste Mengen noch zu testender Knoten

01 **def** *check_selections(*UR, CS, CP, mt, RC, $\texttt{name}$, $\texttt{name}^u$*)*
02 $node = cs$ mit $cs \in CS$ // ersten Selektionsknoten abarbeiten.
03 $SC = \{node.value\}$ // Selektionskette mit dem aktuellen Selektionsprädikat beginnen
04 **if** $node.TR \neq \emptyset$ // Es gab in diesem Pfad vorher schon Tests
05 $TR = node.TR$ // aktuelle Testgrundlage (TJ oder altes TS) merken
06 **else**
07 $TR = r(R_{\texttt{name}})$ // Selektionsprädikate müssen auf Basisrelation geprüft werden.
08 **while** $|node.K^c| == 1 \wedge node.CA = \emptyset \wedge (\forall nc(nc \in node.K^c) \rightarrow nc.type == 3)$
09 $SC = SC \cup nc.value$; $node = nc$
10 Berechnung von TS für SC
11 **if** $TS \neq \emptyset$ // bisher keine Irrelevanz feststellbar
12 $RC = RC \cup node.CA$ // Eventuell hier endende Anfragen sind relevant
13 **for each** $child \in node.K^c$
14 $child.TR = TS$ // TS and Kindknoten weitergeben
15 **if** $child.type == 3$ // Bisher wurde nur ein Teil der Selektion geprüft
16 $CS = CS \cup \{child\}$
17 **else** // Es muss noch wenigstens ein Projektionsknoten getestet werden
18 $CP = CP \cup \{child\}$
19 $CS = CS - \{node\}$ // abgearbeiteten Teilbaum aus CS entfernen
20 *check_predicates(*UR, CJ, CS, CP, mt, RC, $\texttt{name}$, $\texttt{name}^u$*)* // Rekursion

---

Testen der Relevanz für Projektionsprädikate

Wie bereits in Kapitel 4 erwähnt, müssen Projektionsprädikate bei Einfüge- und Löschoperationen nur deshalb getestet werden, weil das Datenmodell Duplikate verbietet. Da die entsprechenden Tests die mit Abstand am aufwendigsten sind (vgl. Kapitel 6), ist es sinnvoll, sie möglichst am Ende des Testablaufs durchzuführen. In diesem Fall sind CJ und CS abgearbeitet und CP ist nicht leer. Im Gegensatz zur Relevanzprüfung bei Verbund- und Selektionsprädikaten, kann keine generelle Relevanzbedingung (verglichen mit $TJ \neq \emptyset$ bzw. $TS \neq \emptyset$) angegeben werden. Der Grund hierfür ist, dass bei Einfügeoperationen und Anfragen ohne Verbunde geprüft werden muss, ob die projizierten Attribute des einzufügenden Tupels durch projizierte Attribute bereits enthaltender Tupel abgedeckt werden. In diesem Fall besteht nur eine Relevant, wenn $TP = \emptyset$ gilt (vgl. Abschnitt 4.2.3).

Algorithmus 8: *Testen der Projektionsprädikate*

Eingabe:	name	// Name der zu ändernden Relation
	name^u	// Name der temporär geänderten Relation (nur bei Update)
	UR	// Update-Operation
	CP	// Menge der noch zu betrachtenden Projektionsknoten
	RC	// bisher zu benachrichtigende Clients
	mt	// Typ der Änderungsoperation
Ausgabe:	RC	// ggf. erweiterte Menge von Client-IDs
	CP	// angepasste Menge noch zu testender Knoten

```
01 def check_projections(UR, CP, mt, RC, name, name^u)
02     if im Pfad vor node ∈ CP ist kein Verbundknoten ∧mt == 1 // Insert
03         if TP == ∅ // kein Duplikat
04             RC = RC ∪ node.CA
05     else
06         if TP ≠ ∅ // Relevanz
07             RC = RC ∪ node.CA
08     CP = CP − {node} // abgearbeiteten Teilbaum aus CS entfernen
09     check_predicates(UR, CJ, CS, CP, mt, RC, name, name^u) // Rekursion
```

5.3 Optimierung des Anfragebaums

Die Algorithmen in Abschnitt 5.2 nutzen die lexikographische Ordnung innerhalb von PSQ-Anfragen um beim Aufbau des Anfragebaumes gleiche Präfix von Anfragen durch gemeinsame Pfade darzustellen. Dadurch wird die Relevanz der entsprechenden Prädikate gemeinsam getestet und die Testanzahl reduziert. Im Folgenden wird eine weitere Optimierung der Speicherung von registrierten Anfragen vorgestellt. Dazu wird die lexikographische Ordnung innerhalb des Anfragebaumes durch eine Ordnung, basierend auf der Häufigkeit des Vorkommens von Prädikaten, ersetzen. Da die Relevanztests bei Verbundprädikaten den Verbund als Ganzes betrachten und Projektionsprädikate pro Anfrage maximal einmal auftreten, ist diese Optimierung nur für die Selektionsprädikate sinnvoll. Das Ziel des optimierten Anfragebaumes ist es, die Anzahl der notwendigen Tests weiter zu reduzieren. Hierzu werden nur aus Selektionsknoten bestehenden Teilbäume so gespeichert, dass diejenigen Selektionsknoten möglichst nah zur Wurzel stehen, welche in den Anfrage, die diesen Teilbaum bilden, am häufigsten vorkommen. Die Anzahl gleicher Anfragen spielt hierbei keine Rolle, da sie durch jeweils einen gemeinsamen Pfad im Baum repräsentiert werden. Abbildung 5.6 illustriert einen optimierten Anfragebaum, welcher die Anfragen aus Beispiel 19 repräsentiert.

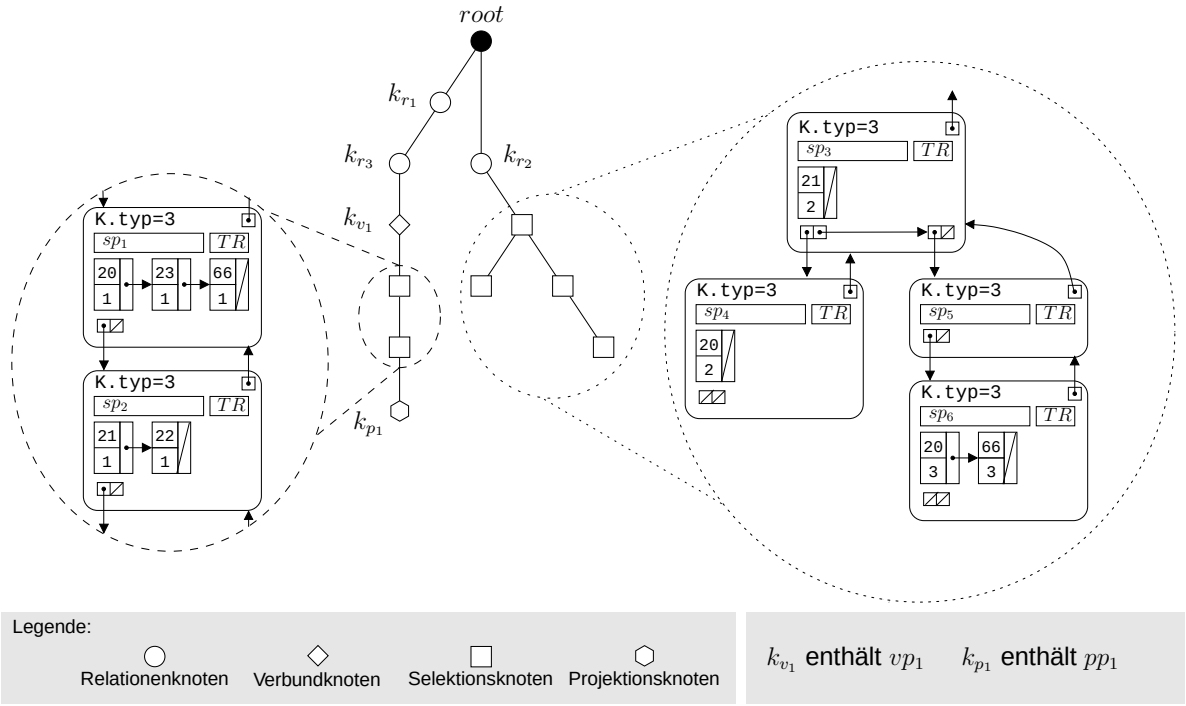


Abbildung 5.6: Beispiel für einen optimierten Anfragebaum

Beispiel 19: *Beispielanfragen für den $\mathcal{OAB}$ in Abbildung 5.6*

Anfrage	CID	AID	$\#sp_1$	$\#sp_2$	$\#sp_3$	$\#sp_4$	$\#sp_5$	$\#sp_6$
linker Teilbaum								
$[vp_1][sp_1]$	20	1	1					
	23	1						
	66	1						
$[vp_1][sp_1][sp_2]$	21	1	1	1				
	22	1						
$[vp_1][sp_1][sp_2][pp_1]$	??	?	1	1				
		Σ	3	2				
rechter Teilbaum								
$[sp_3]$	21	2			1			
$[sp_3][sp_4]$	20	2			1	1		
$[sp_3][sp_5][sp_6]$	20	3			1		1	1
	26	3						
		Σ			3	1	1	1

Im linken Teilbaum ist ein Projektionknoten enthalten, welcher der Endeknoten von mindestens einer weiteren Anfragen ist.

Aus Sicht der Relevanzprüfung können die in Abschnitt 5.2.3 vorgestellten Algorithmen unverändert benutzt werden. Offensichtlich müssen aber die Algorithmen zur Verwaltung des Anfragebaums für einen optimierten Anfragebaum modifiziert werden.

Registrieren neuer Anfragen Q im $\mathcal{OAB}$

Relations- und Verbundknoten werden analog zum Einfügen in einen Anfragebaum behandelt. Das heißt, dass solange kein neuer Knoten erzeugt werden muss, wie der Präfix der neuen Anfrage mit dem Präfix einer bereits registrierten Anfrage übereinstimmt. Anschließend sind die folgenden Fälle zu unterscheiden:

- Ist das erste Prädikat von Q nach dem gemeinsamen Präfix kein Selektionsprädikat, so werden die restlichen Prädikate der Anfrage analog zu Algorithmus 2 eingefügt.
- Enthält der letzte Knoten des bisher betrachteten Pfades im Baum ($subroot$) keine Selektionsknoten als Kindknoten, wird ebenfalls analog zu Algorithmus 2 vorgegangen.
- Gelten die ersten beiden Bedingungen nicht, muss der Teilbaum untersucht werden, welcher bei $subroot$ beginnt und alle Pfade enthält, die als Kind von $subroot$ einen Selektionsknoten haben. Enthält dieser Teilbaum keine Selektionsprädikat, welches in der neuen Anfrage benutzt wird, so kann analog zu Algorithmus 2 weiter vorgegangen werden. Anderenfalls werden Teilbaum und neue Anfrage gemischt.

Im Folgenden wird davon ausgegangen, dass Anfrage und Teilbaum gemischt werden. Wie bereits erwähnt ist das Ziel, dass Prädikatknoten, welche häufig vorkommen, möglichst nah an der Wurzel des Teilbaumes und somit auch an der Wurzel des optimierten Anfragebaumes gespeichert werden. Ein naiver Ansatz um dies zu erreichen (vgl. Abbildung 5.7) besteht darin, alle Teilanfragen des Teilbaumes zu extrahieren ((1) in Abbildung 5.7) und zusammen mit der neuen Anfrage den Teilbaum wie folgt neu aufzubauen ((2) in Abbildung 5.7):

Sei $\mathcal{Q}$ die Menge der Teilanfragen des Teilbaumes und Q' die noch einzufügenden Prädikate von Q. Zuerst werden die Anfragen aus $\mathcal{Q} \cup \{Q'\}$, welche nur ein Selektionsprädikat enthalten, in den Baum eingefügt und aus $\mathcal{Q} \cup \{Q'\}$ entfernt. Anschließend werden diejenigen Anfragen aus $\mathcal{Q} \cup \{Q'\}$ betrachtet, welche zwei Selektionsprädikate enthalten. Die Reihenfolge der Selektionsprädikate ist aufgrund der konjunktiven Verknüpfung irrelevant. Somit werden die Prädikate in diesen Anfragen so getauscht, dass sie die Prädikate der bereits eingefügten Anfragen[7] wiederverwenden können[8]. Sind beide Selektionen bereits im Teilbaum repräsentiert, so werden sie per Definition unverändert eingefügt. Wird stattdessen keine Überlappung gefunden, erfolgt die Ordnung der Prädikate in den Anfragen nach der Häufigkeit ihres Auftretens in den Anfragen im verbleibenden $\mathcal{Q} \cup \{Q'\}$. Nach dem Einfügen der Teilanfragen mit zwei Selektionsprädikaten, werden auch sie aus $\mathcal{Q} \cup \{Q'\}$ entfernt. Anschließend werden Anfragen mit drei $\ldots n$ Selektionsprädikaten eingefügt. Wobei jeweils die Permutation

[7] mit genau einem Selektionsprädikat

[8] vgl. $\sigma_1\sigma_2$ beim Einfügen von σ_2 (Abbildung 5.7)

der Selektionsprädikate gewählt wird, welche die meisten Selektionsknoten aus dem Teilbaum wiederverwendet.

Abbildung 5.7 verdeutlicht, dass die Knotenanzahl eines auf diese Weise erzeugten Teilbaums kleiner oder maximal gleich der Knotenanzahl eines mit Algorithmus 2 erzeugten Teilbaumes ist. In dieser Abbildung wurde aufgrund der Übersichtlichkeit auf die Darstellung von Projektionsknoten verzichtet. Wenn Teilanfragen Projektionsprädikate umfassen, werden diese immer als letztes Prädikat einer Permutation eingefügt. Darüberhinaus illustriert die unterschiedliche Darstellung der Selektionsknoten in Abbildung 5.7 das Ende von Anfragen. Hierzu wird zwischen zwei verschiedenen Knotentypen unterschieden. *Geschlossene Knoten* sind Selektionsknoten, welche entweder Endknoten einer Anfrage sind oder einen Projektionsknoten als Kindknoten haben. Sie werden als graues Rechteck dargestellt. *Offene Knoten* sind alle Selektionsknoten, die nicht geschlossen sind und werden als weißes Rechteck dargestellt.

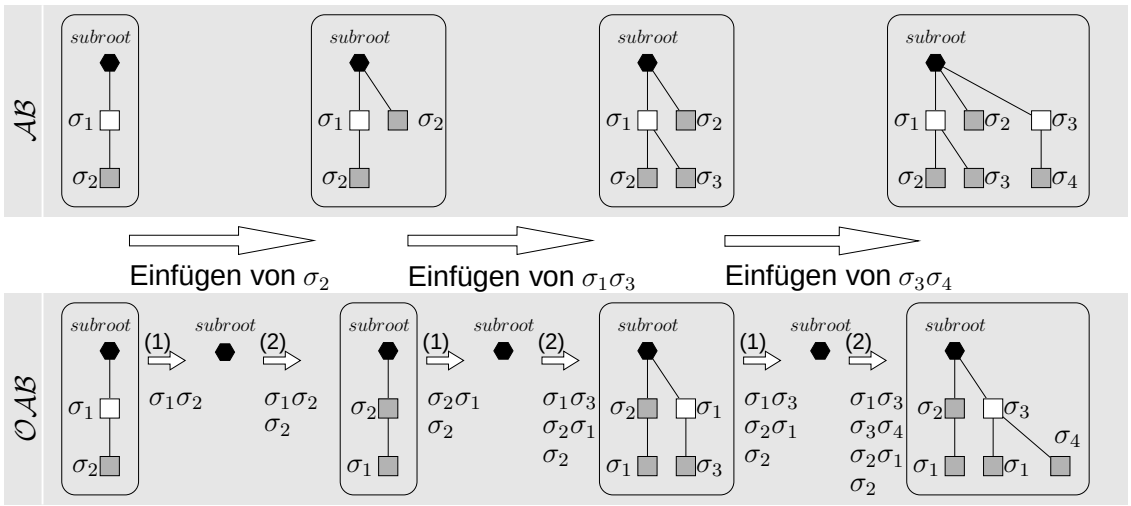


Abbildung 5.7: Naives Mischen von Q und $\mathcal{Q}$

Offensichtlich ist das vollständige Neuaufbauen des Teilbaums ineffizient, da beispielsweise einmal eingefügte Anfragen mit nur einem Selektionsprädikat bei jedem Neuaufbau an derselben Position eingefügt werden. Gleiches gilt für Anfragen mit zwei Selektionsprädikaten, die beim Einfügen einen Selektionsknoten wiederverwenden konnten. Verallgemeinert gilt, dass die Position von geschlossenen Knoten im Teilbaum fixiert ist, wenn sie keinen offenen Knoten als Vaterknoten haben. Daraus ist abzuleiten, dass Teilpfade (ausgehend von $subroot$) fixiert sind, wenn sie nur geschlossene Knoten umfassen. Somit kann ein vollständiger Neuaufbau des Teilbaums vermieden werden, indem Anfragen in fixierten Pfaden im ersten Schritt des Algorithmus nicht aus dem Teilbaum entfernt werden. Im Gegensatz dazu können die Selektionsprädikate von Anfragen, welche durch nicht fixierte Pfade beschrieben werden, frei permutiert werden. Es genügt also, die Anfragen der nicht fixierten Pfade aus dem Teilbaum zu entfernen und mit Q zusammen neu einzufügen.

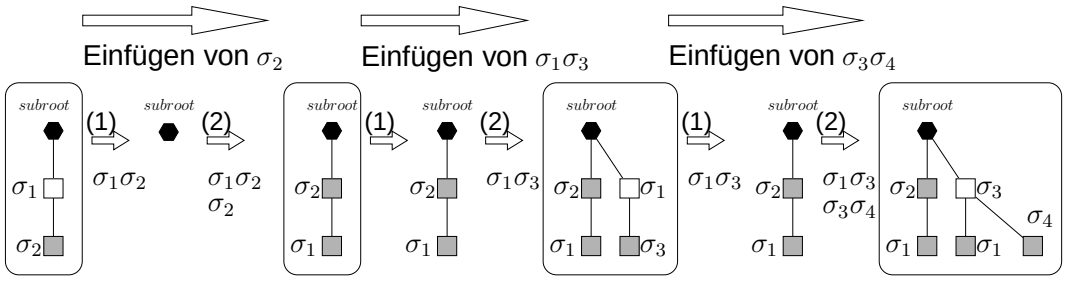


Abbildung 5.8: Einfügen in $\mathcal{AB}$ vs. Einfügen in $\mathcal{OAB}$

Diese in Abbildung 5.8 dargestellte Variante wird mithilfe von Algorithmus 9 realisiert. Das Vorgehen ist hierbei das Folgende: In Zeile 03 wird der letzte (ausgehend von $root$) Knoten $subroot$ desjenigen Pfades im Baum ermittelt, welcher dem längsten möglichen Präfix der vorverarbeiteten Anfrage EQ entspricht. Nach Abarbeitung der entsprechenden Funktion $find\_sub\_tree$ ist in EQ' der noch nicht im Baum repräsentierte Suffix von EQ enthalten. Analog zu den oben genannten Kriterien müssen nun drei Fälle unterschieden werden:

- Ist EQ' leer, so ist die Anfrage schon komplett im Pfad enthalten (Zeile 04). Daher genügt es, mithilfe von $add\_to\_leaf\_index$ (Zeile 05) den BKI zu aktualisieren. Der Rückgabewert dieser Funktion ist die neue AID, wenn der Client diese Anfrage noch nicht registriert hat. Anderenfalls wird ein negativer Wert zurückgegeben, wodurch das erneute Eintragen von CID und AID in der ID-Liste des Knotens $subroot$ verhindert wird.

- Ist die Anfrage noch nicht vollständig enthalten, muss untersucht werden, ob der Suffix der Anfrage nicht mit einem Selektionsprädikat beginnt oder ob $subroot$ keine Selektionsknoten als Kindknoten hat (Zeile 08). Trifft mindestens eine dieser Bedingungen zu, kann die Anfrage analog zu Algorithmus 2 erfolgen (Zeile 09).

- Sind die bisherigen Fälle nicht eingetreten, werden in Zeile 10 alle Teilanfragen, die durch nicht fixierte Pfade ab $subroot$ repräsentiert sind aus dem Teilbaum ermittelt. Wird hierbei festgestellt, dass keine nicht fixierten Teilpfade existieren (Zeile 25), werden die Selektionsprädikate der Suffixes der einzufügenden Anfrage so permutiert, dass möglichst viele Knoten im Baum wiederverwendet werden können (Zeile 26). In Zeile 27 wird anschließend der Suffix mithilfe der Funktion $insert\_suffix$ in den Baum eingefügt und die notwendigen Eintragungen von CID und AID realisiert. Wurden hingegen offene Pfade gefunden, so müssen für alle Permutationen der Selektionsprädikate des Suffixes geprüft werden, ob diese bereits vollständig im Baum enthalten sind (Zeile 13). Ist dies der Fall, genügt es, die CID in den BKI einzutragen wenn diese Anfrage durch diesen Client noch nicht registriert wurde (Zeile 14). Der hierfür notwendige Anfragendknoten aek wird durch die Funktion $is\_new\_suffix\_in\_open\_suffixes$ zurückgegeben. Ist keine dieser Permutation im Baum enthalten, so müssen die offenen

Teilanfragen[9] aus dem Baum ausgetragen werden (Zeile 16) und mit dem Suffix der einzufügenden Anfrage gemischt werden (Zeile 23). Das Wiedereinfügen erfolgt, wie oben beschrieben, in der Reihenfolge der Anzahl von Selektionsprädikaten wobei diese jeweils derart permutiert werden, dass möglichst viele bereits im Baum enthaltenen Knoten wiederverwendet werden können. Diejenigen Selektionsprädikate, deren Einfügen einen neuen Knoten erfordert, werden des Weiteren absteigend nach der Häufigkeit ihres Vorkommens in den offenen Anfragen sortiert.

Algorithmus 9: *Einfügen in den $\mathcal{OAB}$*

Eingabe: EQ // vorverarbeitete Anfrage
CID // Client-ID
$root$ // Wurzelknoten des Anfragebaums
Ausgabe: AID // Anfrage-ID

```
01 def insert_oab_path(EQ, CID)
02     EQ' = copy(EQ)
03     subroot = find_sub_tree(root, EQ') // letzten gleicher Nicht-Selektionsknoten
04     if EQ' = ∅ // Anfrage bereits vollständig enthalten
05         AID = add_to_leaf_index(subroot, CID)
06         if AID>-1
07             subroot.CA = subroot.CA ∪ {(CID, AID)}
08     elif (EQ' == ⟨p ...⟩ ∧ p ∉ SP) ∨ has_selection_childs(subroot) == False
09         Einfügen analog zu Algorithmus 2
10     else
11         open_suffixes = get_open_suffixes(subroot) // offene Teilanfragen ab subroot
12         if open_suffixes ≠ ∅ // Es gibt offene Teilpfade.
13             if aek = is_new_suffix_in_open_suffixes(open_suffixes, EQ')∧
               aek == True
14                 AID = add_cid_to_leaf_index(CID, allready_registered_as)
15             else
16                 temporäres Aushängen der offenen Pfade aus dem Baum und dem BKI
17                 zu jedem suffix^i ∈ open_suffixes CA^i-Menge merken
18                 msuffixes = open_suffixes+EQ' mit CA = {CID, −1}
19                 fq = Häufigkeiten des Auftretens der Selektionsprädikate in msuffixes
20                 msuffixes aufsteigend nach Anzahl Selektionsbedingungen sortieren
21                 for each suffix ∈ msuffixes
22                     suffix zweistufig Permutieren
                       ...(1) maximale Überlappung, (2) Häufigkeit anhand fq
23                     AID = insert_suffix(subroot, suffix, CA^i)
```

[9] Teilanfragen, die durch nicht fixierte Pfade repräsentiert sind.

24 entferne $suffix$ aus $msuffixes$
25 **else** // keine nicht fixierten Teilpfade
26 EQ einstufig nach maximaler Überlappung permutieren
27 $AID = insert\_suffix(subroot, EQ, \{(CID, -1)\}$
28 Eintragen der Attributnamen der Selektions- und Projektionsprädikate von EQ in $subroot$, wenn $subroot$ ein Verbundknoten ist
29 return(AID)

Austragen von Anfragen aus dem $\mathcal{OAB}$

Der Algorithmus zum Löschen aus einem optimierten Anfragebaum (siehe Algorithmus 10) startet analog zu Algorithmus 3. Zuerst wird der Endeknoten $node$ des Pfades der Anfrage mithilfe des Blattknoten-Indexes ermittelt (Zeile 02). Anschließend wird aus $node$ und dem BKI der CID-AID-Eintrag des anfragenden Clients entfernt. Wird dadurch $node.CA$ leer (vgl.(1) in Abbildung 5.9) und hat $node$ keine Kindknoten, so wird der Knoten entfernt. Anschließend wird sukzessiv geprüft, ob auch der Vaterknoten gelöscht werden kann (Zeile 06). Dies ist der Fall, wenn dieser keine weiteren Kindknoten hat und nicht Endeknoten einer anderen Anfrage ist. Ausgehend von dem ersten nichtgelöschten Knoten, wird dann bottom-up der erste Nicht-Selektionsknoten $subroot$ ermittelt (Zeile 11). Hat dieser weitere Selektionsknoten als Kindknoten (Zeile 12), so muss der Teilbaum ab $subroot$ gegebenenfalls neu gemischt werden (Zeile 13). Da dies analog zum Einfügen erfolgt, wird hier nicht nocheinmal detailliert darauf eingegangen (vgl.(2) und (3) in Abbildung 5.9).

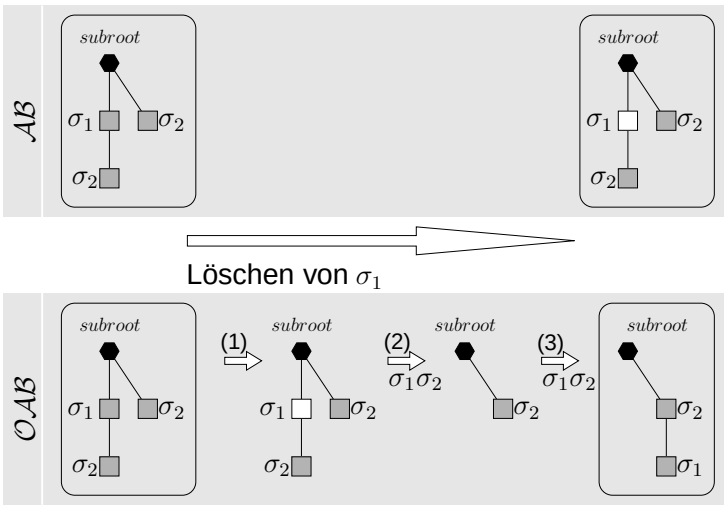


Abbildung 5.9: Löschen einer Anfrage aus einem optimierten Anfragebaum

Das Neumischen des Selektionsteilbaumes ist notwendig, da Anfragen, die ebenfalls $subroot$ benutzen gegebenenfalls auch andere Selektionsprädikate in offenen Knoten benut-

zen können. In diesem Fall muss sichergestellt werden, dass diese Anfragen optimal im Anfragebaum repräsentiert sind. Abbildung 5.9 illustriert, dass der Algorithmus zum Löschen in einem $\mathcal{AB}$ diese Eigenschaft nicht ausnutzt.

Algorithmus 10: *Löschen aus dem $\mathcal{OAB}$*

Eingabe: CID // Client-ID
AID // Anfrage-ID

```
01 def delete_path(CID, AID)
02    let node be „Blattknoten“ der Anfrage
03    node.CA = node.CA − {(CID, AID)} // Anfrageende aus Anfragebaum austragen
04    free_nodes(node)
05    {(CID, AID)} aus BKI entfernen
06    while node.CA == ∅ ∧ node.K^c == ∅
07       parent = node.k^p
08       parent.K^c = parent.K^c − node
09       free(node)
10       node = parent
11    while node.type == 3: node = node.k^p
12    if ∃child(child ∈ node.K^c → child.type == 3)
13       Teilbaum unter child analog zu Algorithmus 9 neu mischen
```

Kapitel 6

Evaluation

Jedes Denken wird dadurch gefördert, dass es in einem bestimmten Augenblick sich nicht mehr mit Erdachtem abgeben darf, sondern durch die Wirklichkeit hindurch muss.
Albert Einstein

6.1 Beispielanwendung

Im Rahmen dieser Dissertation wird zur Illustration der eingeführten Techniken ein einfaches mobiles Informationssystem verwendet, welches es dem Nutzer mobiler Endgeräte erlaubt, einen Kinobesuch zu planen. Das grundlegende Datenbankschema, ist in Abbildung 6.1 als ER-Diagramm abgebildet. Die gewählte Notation von Kardinalitäten basiert hier auf der zulässigen Anzahl von Instanzen eines Entity-Typs, welche an der jeweiligen Beziehung teilnehmen dürfen. Die Beziehung `gehört zu(Kino[1,1],Kette[1,*])` bedeutet also, dass jedes Kino genau zu einer Kette gehört, wohingegen eine Kette mehrere Kinos unterhalten kann. Darüberhinaus wurden Bezeichner von Beziehungstypen, deren „Leserichtung“ nicht eindeutig ist, durch Pfeile ergänzt. Es sei aber ausdrücklich darauf hingewiesen, dass dieses System lediglich zu Illustrations- und Erläuterungszwecken dient. Entwurfs- und Optimierungsaspekte des Datenbankentwurfs[1], wie z. B. die Normalisierung, spielen daher eine untergeordnete Rolle.

Das abgeleitete Datenbankschema gliedert sich in zwei über die Relation Adresse zusammenhängende Teilschemata. Das Erste (in Abbildung 6.1 hellgrau gekennzeichnet) umfasst die Relationen zur Speicherung der Informationen zu den Kinos, wohingegen das Zweite (in Abbildung 6.1 dunkelgrau gekennzeichnet) die aktuelle Verkehrssituation abbildet.

[1] siehe u. a. [HS00]

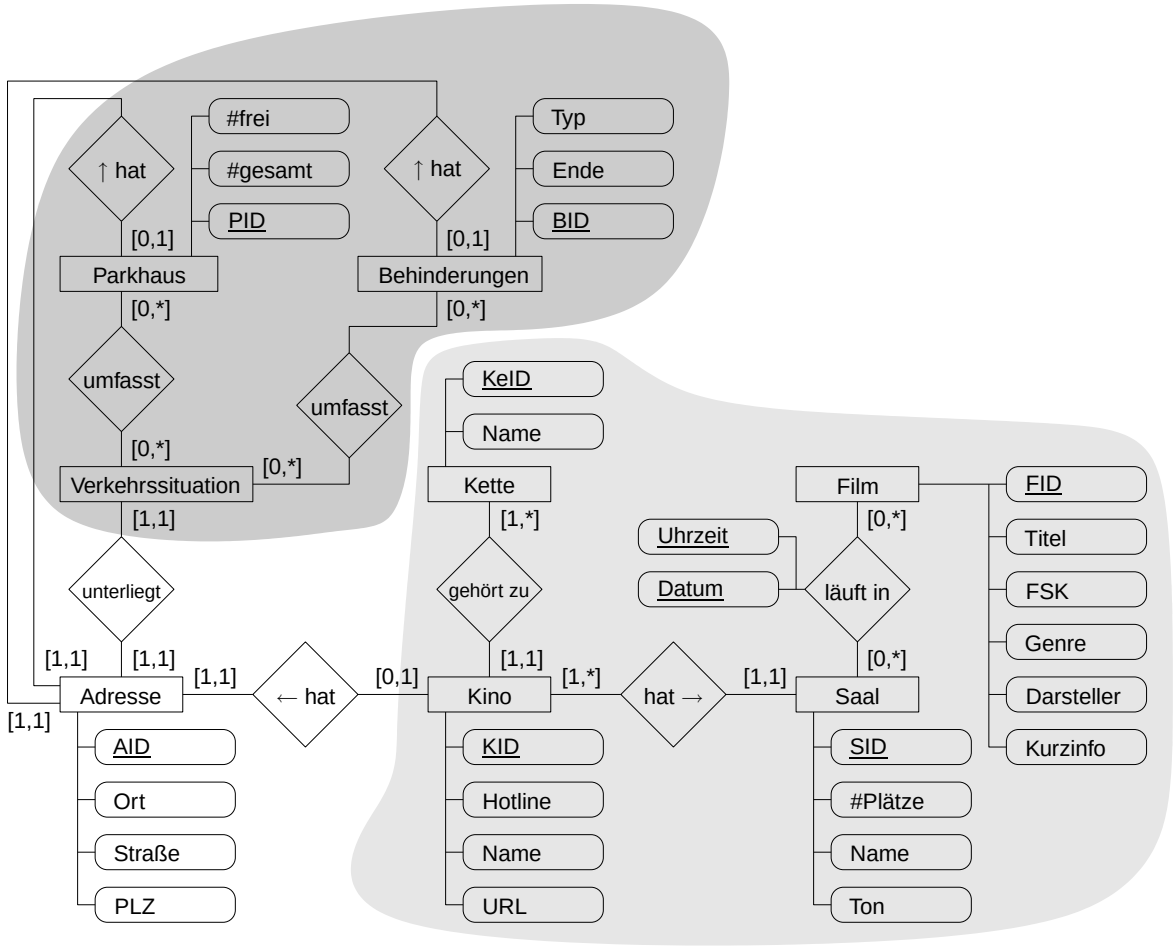


Abbildung 6.1: grundlegendes ER-Schema des Beispielszenarios

6.2 Testumgebung

6.2.1 Rahmenbedingungen

Wie bereits erwähnt, wurden die in dieser Arbeit entwickelten Techniken in der Programmiersprache Python[2] [Sch04] (Version 2.2.3) implementiert. Zur Evaluation wurde das DBMS PostgreSQL[3] [GH01b] (Version 7.4.2) verwendet und der entsprechende SQL-Dialekt [KK01, Boe03] genutzt. Die Anbindung von PostgreSQL an Python erfolgte mithilfe der PyGreSQL-Schnittstelle[4] (Version 3.5). Alle Messungen wurden mit einem Standard PC mit 512 MB RAM und einem AMD Athlon(tm) XP 2000+ Prozessor (1666.663 MHz) durchgeführt. Als Betriebssystem kam SuSE Linux 9.1 mit dem durch SuSE modifizierten Linux-

[2] `http://www.python.org`
[3] `http://www.postgresql.org`
[4] `http://www.pygresql.org/`

Kernel 2.6.5 zum Einsatz.

6.2.2 Erzeugen der Testdaten

Füllen der Datenbank

Die Daten der Relationen des Kino-Teils des Datenbankschemas wurden mit Hilfe von Wrappern aus den Internetseiten einer großen deutschen Kinokette importiert, wohingegen die Verkehrssitationsdaten synthetisch generiert wurden.

Erzeugen der Testanfragen

Das Erzeugen der Testanfragen erfolgte in zwei Schritten. Zuerst wurden mithilfe eines Skriptes 60000 Testanfragen generiert. Hierbei galten die folgenden Restriktionen, die einem natürlichen Benutzerverhalten nahe kommen:

- Verbunde erfolgen nur über die Schlüsselattribute der einzelnen Relationen.
- Selektionen und Projektionen werden *nicht* auf den Schlüsselattributen durchgeführt.

Die generierten Anfragen wurden auf der Datenbank ausgeführt und es wurden nur diejenigen Anfragen übernommen, die innerhalb von 10 Sekunden ein nicht-leeres Ergebnis geliefert haben. Im zweiten Schritt wurden aus diesem Set vier duplikatfreie Anfragesets mit den folgenden Eigenschaften erzeugt:

Set 1: Anfragen benutzen zwischen 1 und 3 Prädikate (kurze Anfragen)

Set 2: Anfragen benutzen zwischen 3 und 5 Prädikate (Anfragen mit mittlerer Länge)

Set 3: Anfragen benutzen zwischen 5 und 7 Prädikate (lange Anfragen)

Set 4: Anfragen benutzen zwischen 1 und 13 Prädikate.

Abbildung 6.2 und Tabelle 6.1 illustrieren die Verwendung der unterschiedlichen Prädikattypen innerhalb der Anfragesets. Hierbei wurde pro Set jeweils die Anzahl der Prädikate jedes Typs der Anzahl unterschiedlicher Prädikate dieses Typs (in Tabelle 6.1 in Klammern) gegenübergestellt. Dies verdeutlicht, dass innerhalb der Prädikatklassen Überlappungen zwischen den Anfragen der einzelnen Sets bestehen müssen. Des Weiteren fällt auf, dass sich die Anzahl verwendeter Projektionsprädikate in den Sets kaum unterscheidet. Die Gründe hierfür sind, dass eine Anfrage maximal einen Projektionsprädikat enthalten darf und die zur Entscheidung für oder gegen ein Projektionsprädikat genutzte Zufallszahl einer Gleichverteilung unterliegt.

Mehr Details zur Zusammensetzung der Anfragesets sind im Anhang B (ab Seite 123) in den Tabellen B.1, B.2 und B.3 enthalten.

	Set 1	Set 2	Set 3	Set 4
Verbundprädikate	1579 (26)	3792 (42)	6017 (45)	3948 (45)
Selektionsprädikate	14530 (2287)	24978 (2462)	41709 (2708)	27584 (2528)
Projektionsprädikate	8799 (390)	9343 (785)	9481 (1275)	9170 (896)

Tabelle 6.1: Anzahl von Prädikaten in den Anfragesets

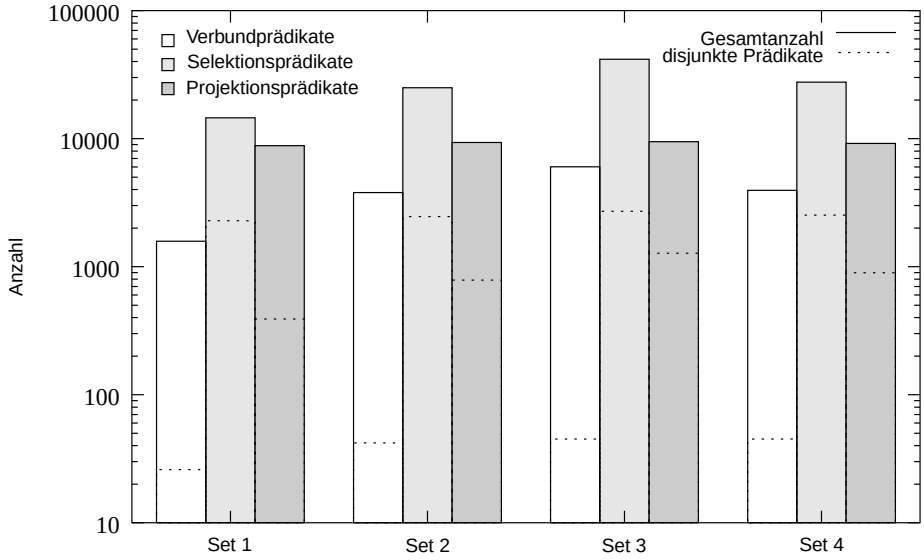


Abbildung 6.2: Anzahl von Prädikaten in den Anfragesets

Erzeugen der Test-Änderungsoperationen

Die Erzeugung der für die Evaluation genutzten Einfügeoperationen erfolgte mithilfe der Wrapper, welche auch zum Füllen der Datenbank genutzt wurden. Lösch- und Änderungsoperationen wurden manuell erstellt.

6.3 Evaluation der Speicherauslastung

Aufgrund der Eigenschaften der zur Implementierung verwendeten Programmiersprache ist es kaum möglich, exakte Angaben zum Speicherverbrauch der Indexstruktur zu treffen. Festzuhalten bleibt, dass alle vier Anfragesets problemlos im Hauptspeicher verwaltet werden können. Die Speichernutzung des Python-Prozesses beim Aufbauen des Anfragebaumes lag bei maximal 8%. Statt einer Angabe der Baumgröße in Kilobyte, kann zur Bewertung des Speicherverbrauches auch die Knotenanzahl, die eine Menge von Anfragen im Baum repräsentiert, genutzt werden. Tabelle 6.2 enthält die entsprechenden Messwerte für den Anfragebaum. Hierbei wird deutlich, dass schon der nicht optimierte Anfragebaum Überlappungen innerhalb

der Anfragen ausnutzt. Anfrageset 1 enthält 24908 Prädikate, welche durch 15662 Knoten im Baum repräsentiert sind. Bei Anfrageset 2 entsprechen 38113 Prädikate 25436 Knoten. Bei Anfrageset 3 ist das Verhältnis 57207 Prädikate zu 41277 Knoten und bei Anfrageset vier 40702 Prädikate zu 28840 Knoten. Bei der Betrachtung der prozentualen Einsparung (Set 1: 37,12%, Set 2: 33,26%, Set 3: 28%, Set 4: 29,14%) wird deutlich, dass kürzere Anfragen durch die Trie-Repräsentation ein höheres Einsparungspotential bieten. Der Grund dafür liegt darin, dass kurze Anfragen weniger Prädikate enthalten und somit auch weniger Kombinationsmöglichkeiten bieten. Dadurch ist die Wahrscheinlichkeit einer syntaktischen Überlappung bei kurzen Anfragen größer als bei Anfragen mit mehr Prädikaten.

	Set 1	Set 2	Set 3	Set 4
Gesamt:	15662	25436	41277	28840
Level 0 ($root$)	1	1	1	1
Level 1	8	7	7	8
Level 2	1936	1120	669	1505
Level 3	7641	5747	3768	5879
Level 4	5032	8166	6640	5987
Level 5	741	5621	8215	4704
Level 6	298	3237	9290	3698
Level 7	5	1075	6552	2639
Level 8		391	3867	1763
Level 9		64	1662	1108
Level 10		7	519	690
Level 11			75	400
Level 12			12	231
Level 13				113
Level 14				61
Level 15				36
Level 16				13
Level 17				5

Tabelle 6.2: Anzahl von Prädikaten im Anfragebaum

Durch den Einsatz des optimierten Anfragebaumes wird eine weitere Reduktion der Knotenanzahl erreicht. Für Set 1 werden hierbei nur 15517, für Set zwei 24171 , für Set drei 38276 und für Set vier 24742 Knoten benötigt um die Anfragen des jeweiligen Sets vollständig abzubilden. Im Vergleich zum nicht optimierten Anfragebaum bedeutet dies für Set 1 eine Einsparung von 1%, für Set 2 von 5%, für Set 3 von 7% und für Set 4 von 14%. Offensichtlich ist die Einsparung bei längeren Anfragen größer als bei kürzeren. Der Grund dafür ist, dass kurze Anfragen eher fixierte Pfade erzeugen. Dadurch ist hier eine Optimierung durch Umsortieren der offenen Knoten weniger wirkungsvoll.

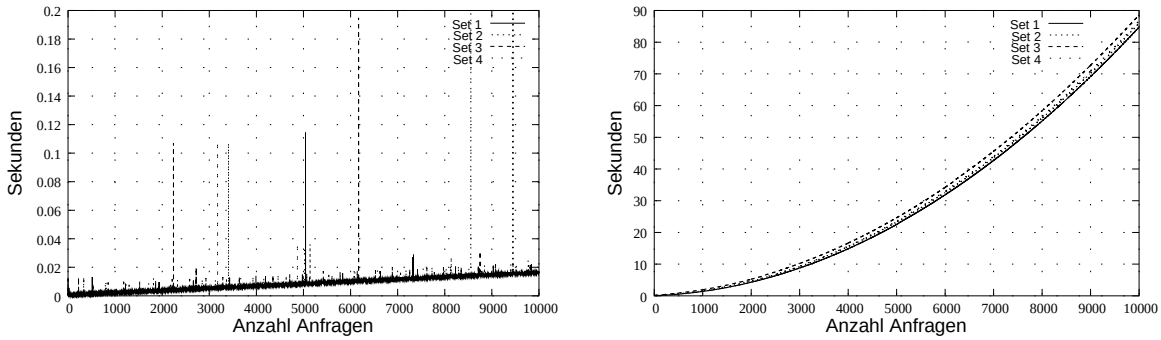
6.4 Performanzevaluation

Um den Einfluss von anderen Prozessen auf die Laufzeit der zu testenden Algorithmen zu minimieren, wurden die Testanfragen zuerst in eine Variable geladen. Erst anschließend begann die Zeitmessung. Hierzu wurde die Differenz aus der Systemzeit vor dem Aufruf einer Funktion und der Systemzeit nach der Abarbeitung der Funktion gemessen. Die Berechnung der Gesamtzeit erfolgte durch Aufsummieren der jeweiligen Differenzen. Um auch bei der Ausgabe der Messergebnisse möglichst wenig Festplattenzugriffe zu benötigen, wurden die Messdaten im Hauptspeicher gehalten und erst nach der kompletten Abarbeitung in eine Datei geschrieben.

6.4.1 Einfügen von Anfragen

Anfragebaum

Das Einfügen von Anfragen in einen Anfragebaum entspricht im Wesentlichen dem Einfügen in einen Trie. Aufgrund der Implementierung der Verkettung der Kindknoten innerhalb eines Vaterknotens kann keine – in der Theorie bei gleichlangen Zeichenketten mögliche – konstante Zeitkomplexität erreicht werden. Abhängig von der Länge dieser Verweisliste müssen beim Einfügen einer neuen Anfrage unterschiedlich viele Kindknoten verglichen werden. Abbildung 6.3(b) illustriert, dass die Länge der Anfragen einen geringen Einfluss auf die Einfügedauer hat. Wie Abbildung 6.3(a) zeigt, ist dies aus Sicht des Einfügens einer einzelnen Anfrage aber vernachlässigbar. Die Ausreißer in dieser Abbildung resultieren aus Hintergrundaktivitäten des Testrechners. Aufgrund der geringen Einfügezeit der einzelnen Anfrage von weniger als 0,02 Sekunden, fallen sie hier aber besonders auf.



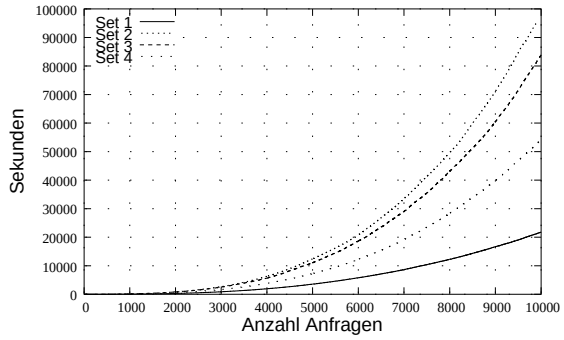
(a) Zeit pro Anfrage

(b) verbrauchte Gesamtzeit

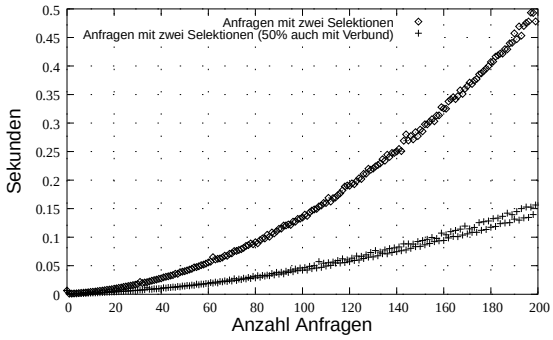
Abbildung 6.3: Einfügen in den Anfragebaum

Optimierter Anfragebaum

Der Algorithmus zum Einfügen in einen optimierten Anfragebaum ist wesentlich komplexer als der zum Einfügen in einen nicht optimierten Anfragebaum. Es ist daher zu erwarten, dass sich dies in der Dauer des Einfügens von Anfragen widerspiegelt. Wie die Abbildung 6.5 zeigt, ist dies tatsächlich der Fall. Interessant ist aber, dass die Länge der Anfragen nur bedingt Einfluss auf die Geschwindigkeit des Algorithmus' hat. Zwar ist die Geschwindigkeit bei den kürzesten Anfragen am höchsten, aber das Einfügen von Anfragen mittlerer Länge (Set 2) dauert länger als das Einfügen von langen Anfragen (Set 3). Der Grund dafür ist, dass längere Anfragen mehr unterschiedliche Kombinationen von Verbundprädikaten ermöglichen. Dadurch sind die umzuorganisierenden Teilbäume, welche per Definition nur Selektionsprädikate enthalten, kleiner. De facto enthält Set zwei 80 verschiedene Präfixe vor Selektionsprädikaten, wohingegen Set drei 143 verschiedene Präfixe enthält. Abbildung 6.4(b) illustriert das Problem anhand des Einfügens von 200 Anfragen mit zwei Selektionsprädikaten. Die untere Kurve ensteht, wenn 50% der Anfragen zusätzlich ein gemeinsames Verbundprädikat enthalten. Auf diese Weise entstehen zwei Teilbäume mit je 100 Pfaden anstatt eines einzelnen Teilbaumes mit 200 Anfragen.



(a) Einfügen in den $\mathcal{OAB}$ – Gesamtzeit

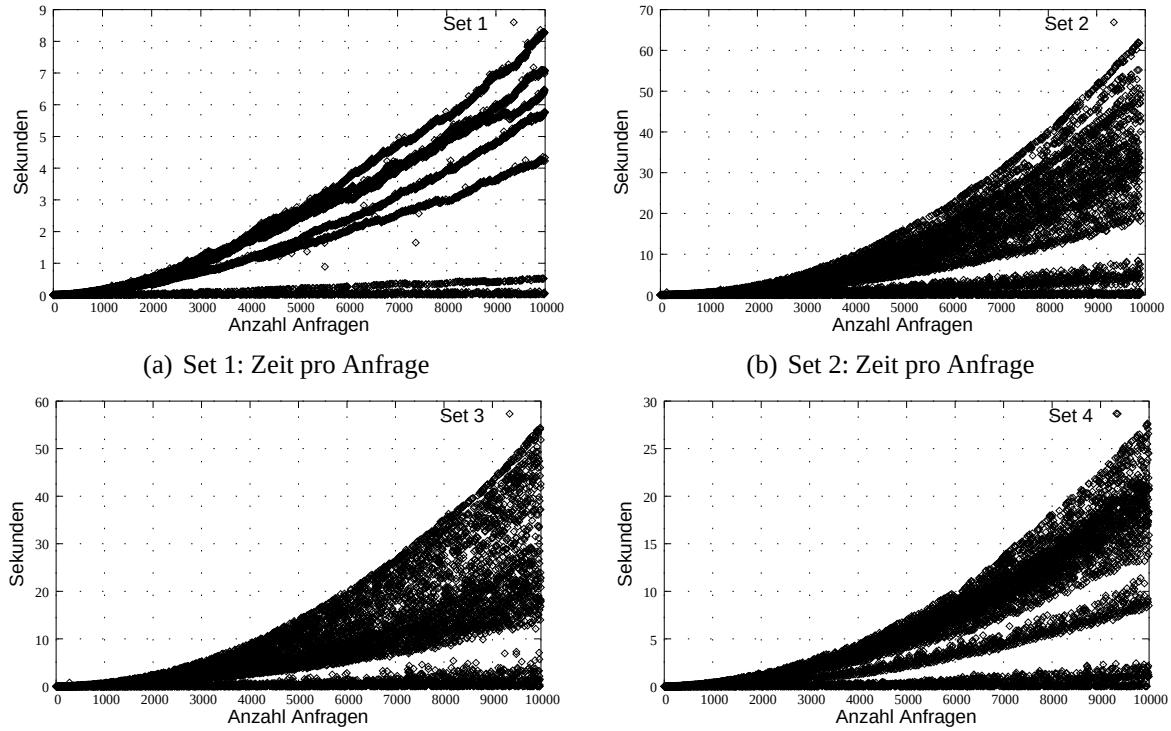


(b) Anfragenlänge vs. Größe offener Teilbäume beim Einfügen in den $\mathcal{OAB}$

Abbildung 6.4: Betrachtungen zur Gesamtzeit beim Einfügen in den $\mathcal{OAB}$

Auch aus der Betrachtung der Einzelzeiten pro Anfrage beim Einfügen (Abbildung 6.5) in einen optimierten Anfragebaum spiegelt sich dieser Zusammenhang wider. Hierbei sind über den Verlauf des Einfügens der Testanfragen Bündelungen erkennbar. Besonders deutlich sind diese beim Einfügen von kurzen Anfragen (Abbildung 6.5(a)) zu erkennen. Die oberen fünf Bündelungen reflektieren fünf separate Präfixe und somit Teilbäume. Je mehr nicht fixierte Pfade in einem Teilbaum enthalten sind und beim Einfügen einer neuen Anfrage umorganisiert werden müssen, desto länger dauert das Einfügen. Die beiden unteren Kurven reflektieren die in Algorithmus 9 betrachteten Sonderfälle bei denen (a) keine Überlappungen im Baum genutzt werden können oder, (b) die neue Anfrage bereits in einer längeren Anfrage im Baum enthalten ist.

Die Abbildungen 6.5(b), 6.5(c) und 6.5(d) zeigen, dass die scharfe Trennung der Bünde-



(a) Set 1: Zeit pro Anfrage

(b) Set 2: Zeit pro Anfrage

(c) Set 3: Zeit pro Anfrage

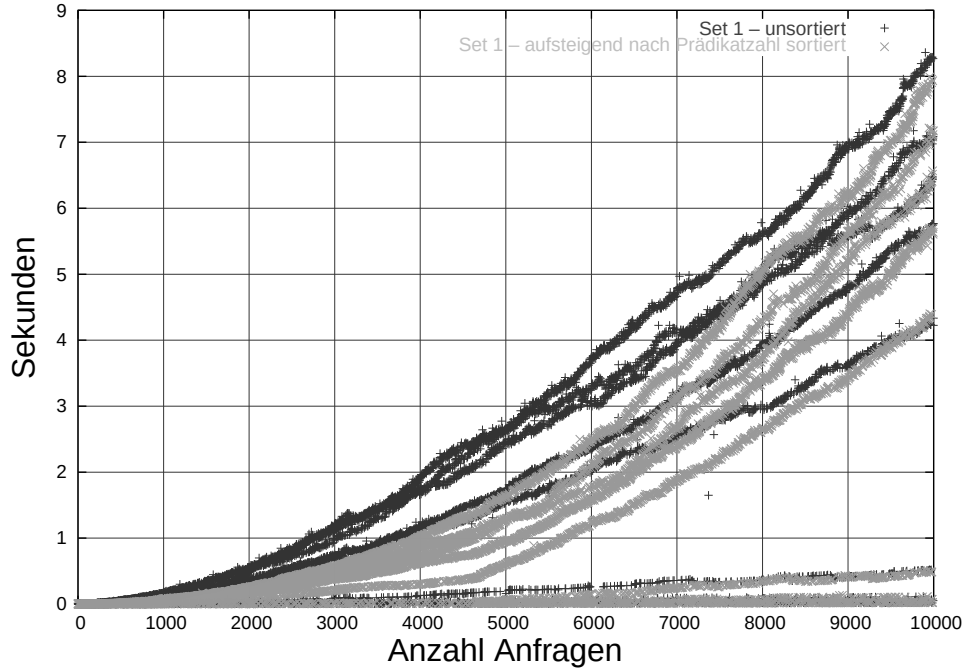
(d) Set 4: Zeit pro Anfrage

Abbildung 6.5: Einfügen in den optimierten Anfragebaum

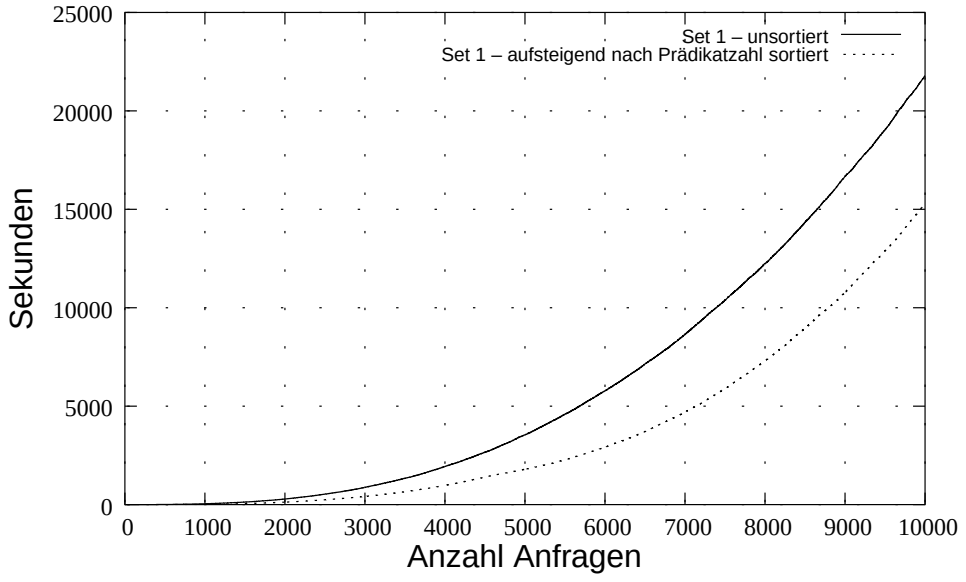
lungen bei längeren Anfragen aufgehoben wird. Vielmehr entstehen Bereiche, welche auf den unterschiedlichen Präfixen basieren. Der Grund, warum dennoch kurze Anfragen schneller in den optimierten Anfragebaum eingefügt werden als lange, ist der Folgende: Wie oben erwähnt, dürfen kurze Anfragen maximal drei Prädikate enthalten. Daraus folgt, dass die Selektions-Teilbäume deutlich mehr fixierte Pfade enthalten.

Einfluss der Sortierung auf die Einfügegeschwindigkeit

Bisher wurde untersucht, wie lange das Einfügen einer beliebigen Anfrage dauert. Eine genaue Betrachtung von Algorithmus 9 lässt vermuten, dass das Einfügen der 10000 Testanfragen schneller ist, wenn diese nach Länge aufsteigend sortiert eingefügt werden. Wie Abbildung 6.6(b) illustriert, trifft diese Vermutung zu. Der Grund dafür ist, dass die zuerst eingefügten kurzen Anfragen keine nicht fixierten Pfade erzeugen. Somit müssen sie beim Einfügen weiterer Anfragen nicht erneut permutiert werden. Wie aber Abbildung 6.6(a) verdeutlicht, gleichen sich die Einzelzeiten für das Einfügen mit Sortierung (hellgrau dargestellt) an die für das unsortierte Einfügen (dunkelgrau dargestellt) an. Daher ist für eine zunehmende Länge der Anfragen damit zu rechnen, dass die beiden Kurve in Abbildung 6.6(b) im Weiteren mit dem gleichen Anstieg verlaufen. Eine Sortierung, die im realen Einsatz ohnehin unwahrscheinlich ist, bringt somit nur Nutzen, wenn dadurch zuerst fixierte Pfade im Baum erzeugt werden.



(a) Set 1: Zeit pro Anfrage



(b) Set 1: verbrauchte Gesamtzeit

Abbildung 6.6: Sortiertes Einfügen in den optimierten Anfragebaum

6.4.2 Relevanzprüfung

Aufgrund des enormen Zeitunterschiedes beim Einfügen in einen Anfragebaum $\mathcal{AB}$ bzw. einen optimierten Anfragebaum $\mathcal{OAB}$ wurde zur Evaluation der Performanz auf den nicht optimierten Anfragebaum zurückgegriffen. Da der optimierte Anfragebaum per Definition maximal genausoviele Knoten enthält wie die nicht optimierte Variante, ist aber davon auszugehen, das die Relevanztests auf einem optimierten Anfragebaum weniger (maximal genauso viel) Zeit benötigen.

Auswahl der zu ändernden Relationen

Für die Evaluation wurden die zu ändernden Relationen anhand der Häufigkeit ihrer Nutzung in Anfragen ausgewählt. Wie Tabelle B.2 in Anhang B auf Seite 123 zu entnehmen ist, sind dies die Relationen `LAEUFT_IN`, `FILME` und `SAELE`. Die Begründung für diese Auswahl ist, dass bei der Relevanzprüfung von Änderungen auf diesen Relationen, eine große Anzahl von Anfragen getestet werden muss. Der schlimmste Fall wäre, wenn alle Anfragen die zu ändernde Relation benutzen. Daher sorgt die Auswahl der maximal genutzten Relationen hier dafür, dass die erreichten Messwerte die Obergrenze der Dauer von Relevanztests auf den Anfragesets entspricht.

Vorbemerkung

Die Ausführungsgeschwindigkeit der bei der Relevanzprüfung generierten Testanfragen steht in direktem Zusammenhang mit der Zeit, die zur Ausführung der eigentlichen Anfrage benötigt wird. Soll beispielsweise eine Verbundkette bezüglich einer Einfügeoperation geprüft werden, so wird die zu ändernde Relation, wie in Abschnitt 4.2 beschrieben, durch das eingefügte Tupel ersetzt. Die Konsequenzist also, dass das Testen der Relevanz länger dauert, wenn auch das Ausführen der jeweiligen Anfrage, welche mit dem betrachteten Prädikat bzw. der betrachteten Verbundkette korrespondiert, länger dauert. Bei der Auswahl der Testanfragen wurden nur Anfragen ausgewählt, deren Beantwortung maximal zehn Sekunden gedauert hat. Das heißt, dass auch das Relevanzprüfen entsprechende Zeit in Anspruch nimmt.

Zeitverbrauch des Prüfens von Änderungsoperationen

Zum Messen des Zeitverbrauchs einer Relevanzprüfung wurde wie folgt vorgegangen: Es wurden je zwei Änderungsoperationen pro betrachteter Relation (siehe oben) gegen das Anfrageset 4 getestet und der jeweilige Durchschnitt ermittelt. Das Testen erfolgte in Schritten von je 200 Anfragen, wodurch sich pro Änderung insgesamt 50 Messwerte ergaben. Wie Abbildung 6.7 illustriert, dauert – bei den gleichen zu prüfenden Anfragen – das Testen der Relevanz von Löschoperationen im Durchschnitt länger als von Einfügeoperationen. Am zeitaufwendigsten ist das Prüfen der Updates. Diese Eigenschaft hängt mit der Art der zum Prüfen erzeugten Testanfragen zusammen.

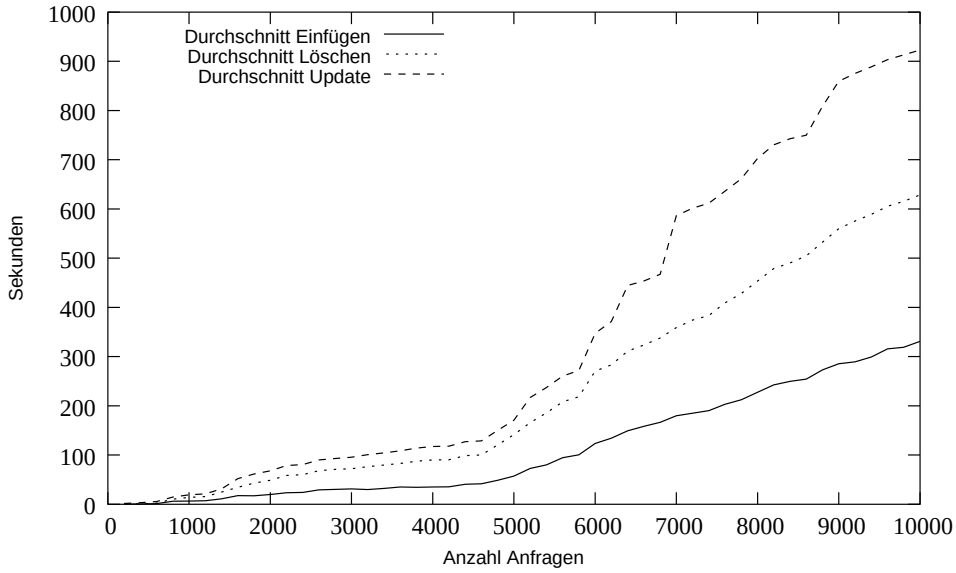


Abbildung 6.7: Durchschnittliche Laufzeit von Relevanzprüfungen

Beispiel 20 verdeutlicht diese Eigenschaft anhand des Relevanztests eines Verbundprädikates. Während beim Einfügen die zu ändernde Relation durch das neue Tupel substituiert wird, muss beim Löschen geprüft werden, ob zum Löschen ausgewählte Tupel in das Verbundergebnis einfließen. Dass die in den Anhang verschobene Testanfrage zum Prüfen der Relevanz eines Updates für Verbunde (siehe auch Formel 4.24 auf Seite 61) in der Ausführung am aufwendigsten ist, ergibt sich durch die in ihr enthaltenen Mengenoperationen.

Beispiel 20: *Relevanztestanfrage für Verbundprädikat*

Einfügen:
```
SELECT DISTINCT * FROM (SELECT DISTINCT 12 AS fsk,
    19233 AS fid, '...' AS kurzinfo, 'Clive Owen, Stephen
    Dillane, Keira Knightley' AS darsteller, 'Action' AS
    genre, 'KING ARTHUR' AS name) AS FILME, LAEUFT_IN WHERE
    FILME.FID=LAEUFT_IN.FID;
```

Löschen:
```
SELECT DISTINCT * FROM FILME, LAEUFT_IN WHERE
    FILME.FID=LAEUFT_IN.FID AND FILME.genre>'Action';
```

Update: aus Gründen der Übersichtlichkeit, siehe Abhang C auf Seite 125

Des Weiteren fällt in Abbildung 6.7 auf, dass trotz unterschiedlicher Änderungstypen die jeweiligen Kurven an vergleichbaren Stellen einen deutlichen Anstieg aufweisen. Durch Analyse des Anfragesets hat sich gezeigt, dass in den jeweilig neu hinzugekommenen 200 Anfragen verstärkt Ungleichheitsverbunde zwischen der durch das Update betroffenen Relation

und mindestens einer weiteren Relation auftraten. Aus dem Ausführungsplan einer solchen Testanfrage (siehe Beispiel 21) wird ersichtlich, dass derartige Verbundoperationen mithilfe einer geschachtelten Schleife ausgeführt werden, wohingegen Gleichheitsverbunde durch einen Hash-Verbund realisiert werden. Details zu den jeweiligen Implementierungsvarianten eines Verbundes sind der gängigen Literatur (z. B. [SH99, Seite 337–338]) zu Datenbank-Implementierungstechniken zu entnehmen. Es kann aber festgestellt werden, dass ein Hash Join i. allg. schneller ist als ein Nested Loop Join.

Beispiel 21: *Ausführungspläne bei Unterschiedlichen Verbundoperationen*

Gleichheitsverbund

Unique (cost=1143.84..1264.59 rows=4391 width=307)
→ Sort (cost=1143.84..1154.82 rows=4391 width=307)
Sort Key: filme.fid, filme.name, filme.fsk, filme.genre, filme.darsteller, filme.kurzinfo, laeuft_in.datum, laeuft_in.uhrzeit, laeuft_in.fid, laeuft_in.sid
→ **Hash Join** (cost=157.22..516.53 rows=4391 width=307)
Hash Cond: (”outer”.fid = ”inner”.fid)
→ Seq Scan on laeuft_in (cost=0.00..270.60 rows=8960 width=20)
→ Hash (cost=156.98..156.98 rows=98 width=287)
→ Seq Scan on filme (cost=0.00..156.98 rows=98 width=287)

Ungleichheitsverbund

Unique (cost=631538.44..655564.91 rows=873690 width=307)
→ Sort (cost=631538.44..633722.66 rows=873690 width=307)
Sort Key: filme.fid, filme.name, filme.fsk, filme.genre, filme.darsteller, filme.kurzinfo, laeuft_in.datum, laeuft_in.uhrzeit, laeuft_in.fid, laeuft_in.sid
→ **Nested Loop** (cost=156.98..20184.38 rows=873690 width=307)
Join Filter: (”inner”.fid <> ”outer”.fid)
→ Seq Scan on laeuft_in (cost=0.00..270.60 rows=8960 width=20)
→ Materialize (cost=156.98..157.96 rows=98 width=287)
→ Seq Scan on filme (cost=0.00..156.98 rows=98 width=287)

Durch das Aufeinanderaufbauen der einzelnen Tests beeinflusst die Zeit, die für das Ausführen dieses einen Tests notwendig ist, auch die Zeiten der Tests der Prädikate in den jeweiligen Kindknoten im Baum. Abbildung 6.8 verdeutlicht das Problem der Ungleichheitsverbunde indem die Änderungsoperationen, die in der Evaluation die längsten Testzeiten ergeben haben, nocheinmal gegen das selbe Anfrageset mit ausschließlich Gleichheitsverbunden getestet wurden. Werden keine Ungleichheitsverbunde verwendet, erfolgt die Relevanzprüfung deutlich schneller.

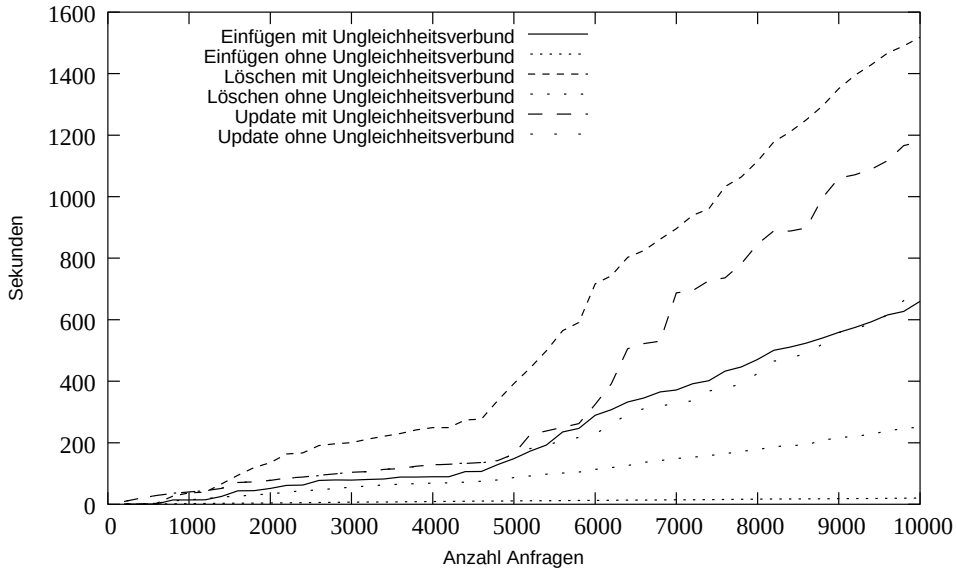
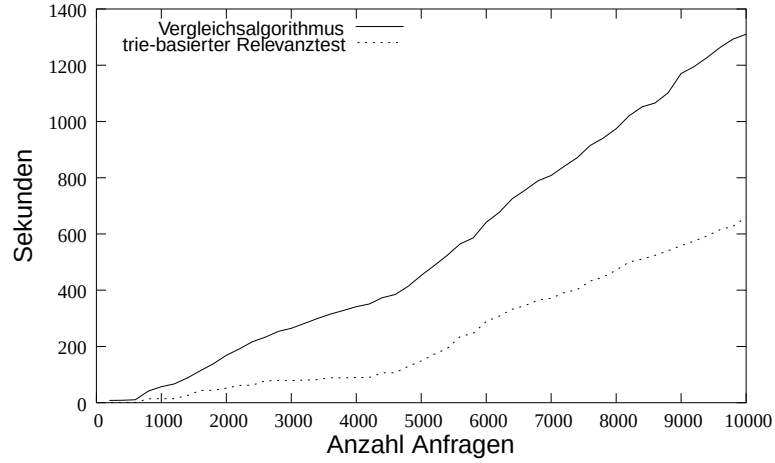


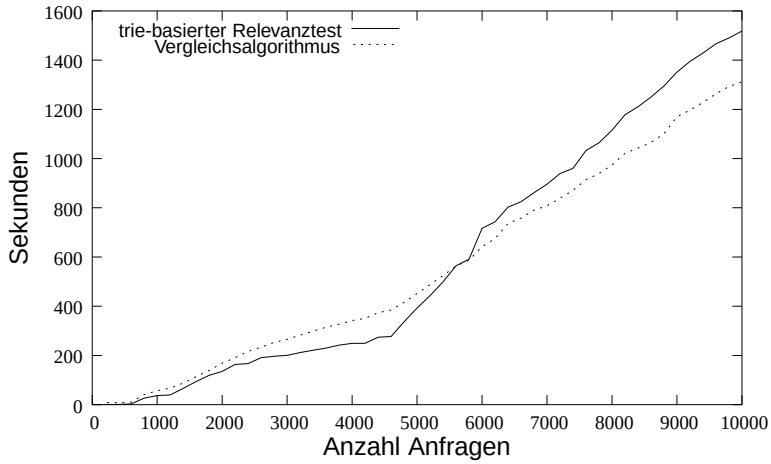
Abbildung 6.8: Einfluss von Ungleichheitsverbunden auf die Laufzeit (Änderungsoperationen mit der längsten Testzeit aus den vorherigen Tests)

Vergleichsalgorithmus

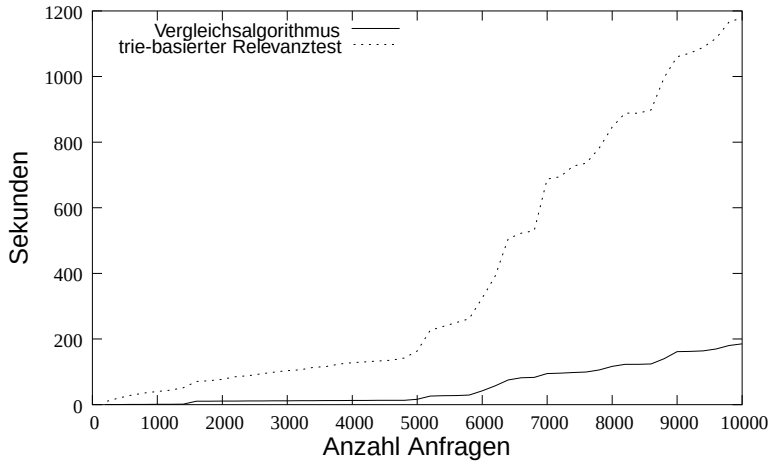
Im Folgenden wird der Trie-basierte Ansatz mit einer Variante des in Abschnitt 5.1 diskutierten sequentiellen Relevanztests verglichen. Hierzu werden die Anfragen in einer Liste repräsentiert, wobei die Relationennamen separat gespeichert werden. Das alte Anfrageergebnis wird als Hash-Wert in der Indexstruktur gespeichert. Das Prüfen der Relevanz einer Änderung erfolgt nach der Ausführung dieser durch Vergleichen des alten Hash-Wertes mit dem neuen Hash-Wert. Das Ergebnis dieses Vergleiches ist in Abbildung 6.9 dargestellt. Abbildung 6.9(a) verdeutlicht, dass das prädikatweise Testen der Relevanz von Einfügeoperationen effizienter ist als das nochmalige Ausführen der Anfrage. Im Gegensatz dazu ist der prädikatweise Ansatz bei Updates ineffizient. Der Grund dafür ist, dass die Beantwortung der Testanfragen in der Summe langsamer ist als das Neuausführen der Gesamtanfrage. Wie bereits oben erwähnt, ist ein Beispiel einer solchen Testanfrage in Anhang C enthalten. Interessant an Abbildung 6.9(b) (Löschoperation) ist, dass sich die beiden Kurven schneiden. Bis ca. zum Einfügen der 6000en Anfrage ist der trie-basierte Ansatz schneller als der Vergleichsalgorithmus. Wie bereits erwähnt sind die extremen Sprünge innerhalb der Kurve auf das verstärkte Auftreten von Ungleichheitsverbunden im Anfrageset zurückzuführen. Es ist also sinnvoll beide Ansätze zu kombinieren bzw. bei Updates automatisch eine Neuberechnung vorzunehmen. Um dies zu ermöglichen, kann auch der Anfragebaum genutzt werden. Der Hash-Wert des Anfrageergebnis kann in den Blattknotenindex integriert werden. Für das Extrahieren der Anfragen kann dann top-down vorgegangen werden.



(a) Einfügen



(b) Löschen



(c) Update

Abbildung 6.9: Vergleich trie-basierter vs. sequentieller Relevanztest

6.4.3 Optimierungsaspekte

Wie die bisherigen Performanzmessungen ergeben haben, ist der in dieser Arbeit vorgestellte Ansatz zwar zum Teil schneller als das sequentielle Prüfen der Anfragen, aus Sicht eines Informationssystems mit einer hohen Änderungsfrequenz aber nur bedingt einsetzbar. Aus diesem Grund, werden im Folgenden zwei Optimierungsansätze vorgestellt, welche den Zeitverbrauch von Relevanzprüfungen reduzieren. Hierzu wurden die aus Sicht der Relevanzprüfung langsamsten Änderungsoperationen der bisherigen Tests genutzt.

Materialisierung von Zwischenergebnissen

Ein bisher nicht explizit diskutierter Nachteil des prädikatweisen Testens der Relevanz einer Änderungsoperation ist, dass die Tests aufeinander aufbauen. Enthält beispielsweise eine Anfrage ein Verbund-, ein Selektions- und ein Projektionsprädikate, so wird zuerst das Verbundprädikat mithilfe von Testanfrage T_1 geprüft. Danach wird das Selektionsprädikat mit $T_2(T_1)$ geprüft. Abschließend wird dann das Projektionsprätikat mit $T_3(T_2(T_1))$ untersucht. Offensichtlich bedeutet das, dass Testanfragen mehrfach ausgeführt werden müssen. Um diesen Aufwand zu reduzieren, ist es möglich, das Ergebnis von Testanfragen bei der Relevanzprüfung zu materialisieren. Im Prototyp wurde für eine Entscheidung für bzw. gegen eine Materialisierung die Anzahl der Kindknoten des gerade zu testenden Knotens[5] herangezogen. Testanfragen eines Präfixes mehrere registrierter Anfragen werden als temporäre Tabelle in der Datenbank zwischengespeichert. Bei den Relevanztests der anderen Prädikate dieser Anfragen muss demnach das vorhergenende Testergebnis nicht erneut berechnet werden. Abbildung 6.10 illustriert, dass dieses Vorgehen zu einer erhöhten Geschwindigkeit der Relevanztests führt. Dabei fällt auf, dass eine generelle Materialisierung aller Zwischenergebnisse nicht sinnvoll ist. Der Grund hierfür liegt in dem Mehraufwand, der aus dem Anlegen der temporären Tabellen resultiert. Hat ein Teilbaum ab einem Prädikat, dessen Testergebnis materialisiert wurde, nur wenige Kindknoten, ist die Wiederverwendbarkeit des Ergebnisses eher gering.

Offensichtlich ist die Entscheidung anhand der Kindknotenzahl nur eine Möglichkeit, die in ungünstigen Fällen sogar zu Verschlechterungen führen kann. Daher muss in Folgearbeiten (siehe Anschnitt 7.1) ein detailliertes Kostenmodell erarbeitet werden.

Bemerkungen zu den Projektionsprädikaten

Im Rahmen dieser Dissertation wurde die Mengensemantik des Relationenmodells verwendet. Dadurch ist es notwendig, Projektionsprädikate auch bei Einfüge und Löschoperationen zu testen. In Vorarbeiten (vgl. [HSS04a]) wurde, neben einer eingeschränkteren Mächtigkeit der Anfragesprache, die Multimengensemantik verwendet. Hierbei war das Prüfen der Projektionspädikate bei Einfüge- und Löschoperationen nur notwendig, wenn die Anfrage weder Selektionen noch Verbunde benutzt. Der Grund dafür ist, dass erfolgreiche Einfügeoperationen in der Multimengensemantik garantiert eine Veränderung der Kardinalität bewirken, welche auch durch die Projektion nicht „ausgeblendet“ werden kann. Auch bei Updates muss

[5] bei Verbundketten, der letzte Knoten des Verbundes

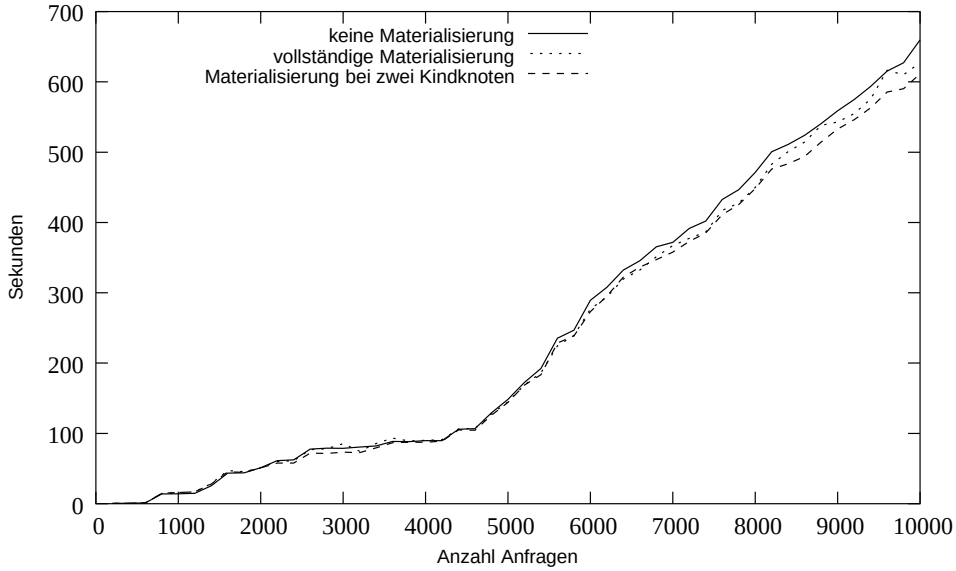


Abbildung 6.10: Materialisierung von Zwischenergebnissen

die unter Mengensemantik notwendige Projektionsprüfung nur dann erfolgen, wenn sich die Kardinalität beim prädikatweisen Testen der Verbund- und Selektionsprädikate nicht geändert hat. Wie Abbildung 6.11 illustriert führt das Nicht-testen-müssen der Projektionsprädikate zu einer deutlichen Beschleunigung der Relevanztests. Hierzu wurden die oben verwendet Anfragen derart modifiziert, dass sie nur Projektionsprädikate enthalten, wenn sie keine weiteren Prädikate umfassen. Die Relevanztests wurden jedoch nicht modifiziert. Da diese derart konzipiert sind, dass sie die Mengensemantik aufrecht erhalten, benutzen sie weiterhin `SELECT DISTINCT` anstelle des unter Multimengensemantik üblichen `SELECT`. Daraus ist zu schlussfolgern, dass eine reine, auf Multimengen basierende Umsetzung, einen weiteren Performanzgewinn ergibt.

In Abbildung 6.11 fällt darüberhinaus auf, dass die Kurve des Tests mit beschränkten Projektionsprädikaten zwischenzeitlich sinkt. Diese Eigenschaft ist durch das verwendete Datenbankmanagementsystems und die Durchführung der Evaluation begründet. Wurden in einem Testschritt viele Anfragen eingefügt, die bei der folgenden Relevanzprüfung nicht oder nur in geringem Maße geprüft werden müssen, so sind die Seiten der Daten des Anfrageergebnisse aus der vorherigen Relevanzprüfung noch im Cache des DBMS enthalten und werden entsprechend wiederverwendet. Durch die Komplexität der Tests bei den Projektionsprädikaten unter Mengensemantik ist dieser Effekt dort vernachlässigbar.

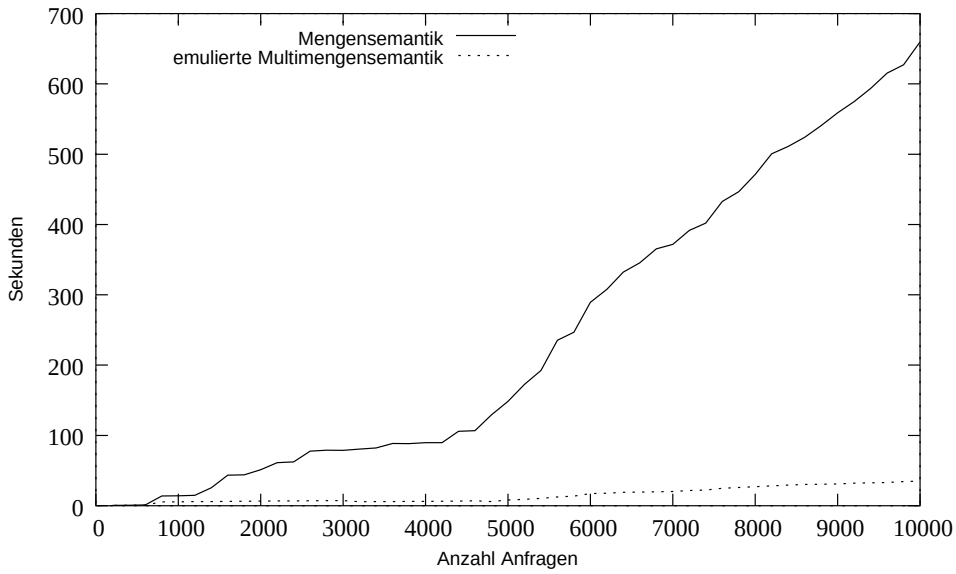


Abbildung 6.11: Vergleich Relevanztest einer Einfügeoperation unter Mengen- und Multimengensemantik

Kapitel 7

Zusammenfassung und Ausblick

Alles Wissen und alles Vermehren unseres Wissens endet nicht mit einem Schlusspunkt, sondern mit einem Fragezeichen.

Hermann Hesse

In der vorliegenden Arbeit wurde das Problem der Relevanzprüfung von Änderungsoperationen für mobile Clients eines relationalen Datenbanksystems bearbeitet. Es wurde gezeigt, dass existierende Ansätze, welche auf der Anfrageebene und semantischer Korrelationsbestimmung basieren unzureichend sind. Die beiden hauptsächlichen Nachteile solcher Verfahren sind die starke Beschränkung der Anfragesprache und die mangelnde Skalierbarkeit. Letzteres bedeutet, dass jedes Update mit jeder Anfrage geprüft werden muss. Darüberhinaus basieren derartige Ansätze auf dem Query Containment Problem. Mithilfe der Überführung dieses Problems in eine logische Implikation wurde gezeigt, dass der Fall eintreten kann, dass das Ergebnis einer Änderungsoperation zwar im Ergebnis der Anfrage eines Clients enthalten ist, jedoch keine Relevanz besteht. Dies ist der Fall, wenn die zur Änderungsoperation korrespondierende Anfrage die leere Menge ergibt.

Aus diesen Gründen wurde in der vorliegenden Dissertation ein Ansatz verfolgt, der nicht ausschließlich die Anfragen sondern auch die Daten der Datenbank zur Relevanzbestimmung heranzieht. Um den dazu notwendigen Berechnungsaufwand zu reduzieren wurden Indexstrukturen entwickelt, welche die Anfragen der Clients benutzen, um die Clients zu indexieren, wobei syntaktische Überlappungen von Anfragen zusammengefasst werden.

Es wurde eine spezielle Anfragenotation eingeführt, welche den Aufbau des Indexes erleichtert und gleichzeitig die Beschränkung der zugelassenen Anfrageoperationen darstellt. Erlaubt in PSQ sind θ-Verbunde, Selektionen (mit Ungleichheitsvergleichen), Projektionen und Tabellenumbenennungen, wodurch auch Selbstverbunde ermöglicht werden. Für den Aufbau des Anfrageindexes wurde auf das bekannte Konzept der Tries zurückgegriffen. Während dieses für die Repräsentation von Wörtern nur „gleichartige“ Knoten vorsieht, wurde es im Rahmen der Arbeit dahingehend erweitert, dass Knoten unterschiedliche Informationsmengen beinhalten können. Dazu wurden sechs verschiedene Knotentypen definiert.

Als Optimierung des entwickelten Anfragebaums wurden Teilpfade, die nur aus Selektionsknoten bestehen derart permutiert eingefügt, dass sie möglichst viele bereits vorhandene Knoten wiederverwenden können. Bei der Evaluation zeigte sich, dass dieses Vorgehen tatsächlich zu einer signifikanten Reduktion der Gesamtknotenzahl und somit der Anzahl von Vergleichen beim Relevanztest führt. Es wurde aber auch deutlich, dass das Einfügen einer Anfrage in diesen optimierte Anfragebaum wesentlich länger dauert, wobei die Geschwindigkeit nicht allein von der Länge der Anfragen abhängt. Vielmehr hängt sie von der Anzahl unterschiedlicher Präfixe bis zum jeweiligen $subroot$ im Baum ab.

Bei der Evaluation des Relevanztests wurde beim Vergleichen des prädikatweisen Vorgehens mit der Neuberechnung aller Anfragen festgestellt, dass der Trie-basierte Ansatz nur beim Einfügen einen echten Vorteil bringt. Durch die Analyse der Schwachpunkte dieses Ansatzes wurde festgestellt, dass die wesentlichen Performanzeinbuße durch Zulassung von Ungleichheitsverbunden entstehen. Im Rahmen der Evaluation wurde daher ebenfalls untersucht, ob das Materialisieren von Zwischenergebnissen zu einer Performanzsteigerung führt. Es wurde gezeigt, dass diese anfangs minimal ist, jedoch mit zunehmender Anfragezahl steigt. Eine wesentlich größere Performanzsteigerung ist durch die Ausnutzung von Eigenschaften der Multimengensemantik, welche durch alle kommerziell eingesetzten Datenbankmanagementsysteme unterstützt wird, zu erwarten.

Das Resümee dieser Dissertation kann wie folgt gezogen werden: Das prädikatweise Testen der Relevanz von Änderungsoperationen für durch mobile Clients registrierte Anfragen, welche nicht den Beschränkungen semantischer Verfahren unterliegen, ist auf Datenebene möglich. Durch eine Repräsentation der Anfragen als (optimierter) Anfragebaum kann eine Reduktion der Prädikatstest erreicht werden. Aus Sicht der Performanz ist es jedoch sinnvoll, die Tests nicht auf der formalen Mengensemantik des Relationenmodells, sondern unter Multimengensemantik umzusetzen, da hierbei Kardinalitätsänderung als Kriterium der Relevanz genutzt werden können. Darüberhinaus beeinflusst der Einsatz von Ungleichheitsverbunden in den Anfragen die Geschwindigkeit der Relevanztests negativ. Kann weder auf die Mengensemantik noch auf Ungleichheitsverbunde verzichtet werden, so muss der prädikatbasierte Ansatz durch ein Verfahren, welches auf einer Neuberechnung der Anfrageergebnisse basiert, ergänzt werden.

7.1 Ausblick auf Folgearbeiten

Die entwickelten Techniken bilden den Ausgangspunkt für zahlreiche mögliche Folgearbeiten, welche zum Teil schon detailliert angedacht wurden. Im Folgenden werden einige mögliche Erweiterungen vorgestellt.

Multimengensemantik

Klassische Datenbankmanagementsysteme benutzen das Relationenmodell nicht in seiner ursprünglichen mathematischen, mengenbasierten Definition, sondern interpretieren Relationen

als Multimengen. Im Rahmen der vorliegenden Dissertation wurde dennoch die Mengensemantik gewählt. Der Grund dafür ist, dass die Relevanztests bei Mengen, die Duplikateliminierung beachten müssen, wodurch die Ergebnisse dieser Dissertation als „worst case“ betrachtet werden können. Zumindest bei Lösch- und Einfügeoperationen ist das Testen von Projektionsprädikaten im Fall von Multimengen nicht notwendig. Beide Operationen führen zu einer Änderung der Kardinalität der betroffenen Relation. Eine derartige Veränderung kann aber nicht durch Projektionen ausgeblendet werden. Bei Update-Operationen muss das Projektionsprädikat auch unter Multimengensemantik geprüft werden. Der Grund hierfür ist, dass das Update die projizierten Attribute unverändert lassen kann und eine Änderung der Kardinalität nicht zwangsweise gegeben ist. Das Ausführen der entsprechenden Testanfrage kann hier jedoch auf den Fall beschränkt werden, dass die Anfrage Verbundprädikate enthält und dass das Update die Kardinalität des Verbundergebnissen und eventuell nachfolgender Ergebnisse der Tests von Selektionsprädikaten nicht beeinflusst.

Relationale Vollständigkeit

Zwar wurden in der vorliegenden Arbeit einige der wesentlichen Restriktionen der Anfragesprache aufgehoben, dennoch ist eine relational vollständige Anfragesprache nicht vollständig auf die hier verwendete PSQ-Notation überführbar. In Folgearbeiten muss nun untersucht werden, inwieweit die noch fehlenden Operatoren für das prädikatweise Relevanztesten nutzbar gemacht werden können. Insbesondere die fehlenden Mengenoperationen stellen hierfür eine Herausforderung dar. Hierzu ist die „Nichtkommutativität“ der Differenz sowie die „Disjunktivität“ der Vereinigung zu untersucht. Die Durchschnittsbildung wird fast implizit durch die verwendete Notation unterstützt. Durch Hinzufügen eines Umbenennungsoperators auf Attributebene, analog zu den Tabellen-Aliasen, kann er mit Hilfe von Verbund und Projektion ausgedrückt werden (vgl. hierzu [SH99, Seite 332]).

Ereignisgesteuerte Reorganisation des Anfragebaums

Wie die Evaluation in Kapitel 6 verdeutlicht hat, stellt die Aufrechterhaltung der Eigenschaften des optimierten Anfragebaums einen nicht unerheblichen Berechnungsaufwand dar. In Produktivszenarien sind Einfügezeiten von 3 Sekunden (und mehr) pro Anfrage nicht akzeptabel. Eine Alternative wäre das Einfügen zuerst analog zum normalen Anfragebaum durchzuführen und den Baum beim Eintreten bestimmter Ereignisse zu optimieren. Derartige Ereignisse könnten bestimmte Zeiten (z. B. immer 23.13 Uhr), festgelegte Lastprofile des Servers, maximal erlaubte Baumdimensionen oder ein Schwellwert der Dauer der Relevanzprüfung sein. Entsprechende Kostenmodelle müssen natürlich auch im Rahmen von Folgearbeiten entwickelt werden.

Automatische Optimierung der Relevanztests

Bei der Evaluation der Relevanztests wurde auch die Möglichkeit der Materialisierung von Zwischenergebnissen getestet. Im Prototyp wurden dazu manuell Parameter geändert. In Fol-

gearbeiten könnte anhand eines Kostenmodells entschieden werden, welche Zwischenergebnisse materialisiert werden. Faktoren, die in eine derartige Entscheidung einfließen können, sind z. B. die Anzahl von (direkten oder indirekten) Kindknoten eines Knotens, die Antwortzeit der jeweiligen Testanfrage sowie Speicherlimitationen.

Optimierung der Implementierung

Die zur Evaluation genutzte Implementierung erfolgte unter Aspekten des Prototypings und enthält einige Konstrukte, die in Produktivsystemen so nicht genutzt werden würden. Ein Beispiel hierfür ist der Einsatz von regulären Ausdrücken in Python (vgl. Angang A.1.2). Python bietet die Möglichkeit, reguläre Ausdrücke einmalig zu kompilieren und an unterschiedlichen Stellen im Code dann die kompilierte Variante zu verwenden. Beim Prototyp wurde hierauf aus Gründen der besseren Lesbarkeit des Quellcodes verzichtet. Eine weitere Fragestellung für Folgearbeiten ist die nach der optimalen Programmiersprache. Python eignet sich zwar aufgrund seiner Restriktionen bezüglich der Strukturierung des Quellcodes hervorragend, um gut lesbaren Code zu produzieren, bietet aber aufgrund der automatischen Freigabe von Speicher (engl. garbage collection) wenig Möglichkeiten, den Speicherverbrauch manuell zu optimieren. Aus diesem Grund sind in der Evaluation auch keine detaillierten Angaben zum Speicherverbrauch der Indexstruktur enthalten.

Anpassung an andere DBMS

Bei der Implementierung des Prototypen wurde PostgreSQL als DBMS verwendet. Um beispielsweise MySQL benutzen zu können sind neben der Änderung der Schnittstelle auch inhaltliche Änderungen notwendig. PostgreSQL ignoriert die Groß- und Kleinschreibung bei Attribut- und Tabellennamen. Datenbanksysteme, bei denen eine entsprechende Unterscheidung vorgenommen wird (z. B. MySQL) bedürfen daher der Anpassung der Algorithmen zur Anfragevorverarbeitung.

Nutzung von Kontextinformationen

Aktuelle Forschungsprojekte im Umfeld mobiler Informationssysteme versuchen, Anfragen im jeweils aktuellen Kontext der Benutzung des Informationssystems zu betrachten. Ein klassisches Beispiel hierfür sind lokationsbasierte Dienste, welche dem Nutzer Informationen abhängig von dessen Aufenthaltsort zur Verfügung stellen. Das hierzu notwendige Bestimmen von Positionen ist auf unterschiedlichen Wegen möglich. Wie es beispielsweise mithilfe von GPS realisiert werden kann, wurde in eigenen Vorarbeiten [ABF+03, ABF+04] untersucht. Im Rahmen des hier genutzten Kino-Szenarios könnte ein derartiger Ansatz die Information „Der Nutzer befindet sich gerade in Magdeburg.“ nutzen, um eine Vorauswahl der Daten durchzuführen. Das heißt, dass eine Anfrage zu aktuell laufenden Filmen automatisch nur die Kinos in Magdeburg einbezieht. Andere Einsatzgebiete hierfür sind kontextabhängige Bereichsanfragen, bei denen der Benutzer definieren kann, wie restriktiv eine solche Vorauswahl ist. So könnte er alle Kinos in das Ergebnis einfließen lassen, welche sich in einem Umkreis

von 50km um seine aktuelle Position befinden. Lokationsinformationen sind aber nur ein Teil des Nutzerkontextes. Andere Beispiele sind der Zeitbezug oder die Bündelung von zu erledigenden Aufgaben [PSPB04]. Wenn beispielsweise ein Nutzer einen Vortrag zu einem Thema vorbereiten muss ist i. allg. irrelevant, wo und wann dies geschieht. Das wesentliche Kontextelement ist die Aufgabe. Aus diesem Grund wurde in [HS03b] ein generischer Ansatz zur Modellierung von Kontexten vorgeschlagen, welcher auf der Idee basiert, Teile (Fragmente) einer Datenbank derart zu annotieren, dass sie beim Anfragen mithilfe einer Kontextbeschreibung ausgewählt werden können. Das Übertragen der Kontextbeschreibung vom Client erfolgt durch so genannte Client-Extensionen, welche ergänzt um eine zulässige Höchst- und eine geforderte Minimalabweichungen, Replikationskriterien bilden. Im Rahmen zukünftiger Forschung muss nun untersucht werden, ob derartige Replikationskriterien als Prädikate interpretiert werden können. Ist dies möglich, wäre es sinnvoll, die Anfragenotation derart zu erweitern, dass auch kontextbasierte Anfragen möglich sind.

Anhang A

Implementierung des Indexmanagers

Für das Implementieren der Serverkomponente wurde die Programmiersprache Python[1] in der Version 2.3.3 [vRD03a, vRD03b] verwendet. Im Folgenden werden einige Ausschnitte aus dem Quellcode vorgestellt. Diese Auswahl dient der Unterstützung der Algorithmenbeschreibungen in Kapitel 5.

A.1 Vorbereiten von PSQ-Anfragen

A.1.1 Sicherstellung der Großschreibung von Attribut- und Relationennamen

Beim Erzwingen von Großschrift in Relationen- und Attributnamen muss sichergestellt werden, dass Texte in Selektions- und Verbundbedingungen nicht verändert werden. Da diese in einfache Anführungszeichen gekapselt werden, ist das Vorgehen das Folgende: Die Anfrage wird zeichenweise durchlaufen. Solange kein einfaches Anführungszeichen auftritt, wird das aktuelle Zeichen in Großschrift konvertiert (up_query=up_query+string.upper(i)). Beim Auftreten des ersten einfachen Anführungszeichens wird die Variable noqm auf 1 gesetzt. Die darauf folgenden Zeichen werden unverändert übernommen. Beim nächsten einfachen Anführungszeichen wird die Variable zurück auf 0 gesetzt. Listing A.1 enthält den entsprechenden Quellcode dieses Algorithmus', wobei die Eingabeanfrage in der Variable query enthalten ist und die Ausgabeanfrage in up_query erzeugt wird.

Listing A.1: Anfrage in Großbuchstaben

```
up_query="";
noqm=0;
for i in query:
    if i=="'":
        up_query=up_query+"'"
```

[1] http://www.python.org

```
        if noqm==0:
            noqm=1;
        else:
            noqm=0;
    else:
        if noqm==0:
            up_query=up_query+string.upper(i)
        else:
            up_query=up_query+i
```

A.1.2 Bestimmen von Prädikattypen

Zum Bestimmen der Prädikattypen werden reguläre Ausdrücke (vgl. [Fri01]) verwendet. Bei einem Prädikat praed handelt es sich um ein Verbundprädikat, wenn die Bedingung **if** re.match('\A[\[][@a-zA-Z0-9_]*,',praed): erfüllt ist. Das Pädikat beginnt also mit einer öffnenden, eckigen Klammer, gefolgt vom ersten Relationennamen und einem Komma. Ein Selektionsprädikat kann durch **if** re.match('\A[\[][@a-zA-Z0-9_]*\.[@a-zA-Z0-9_]*[\>\<=\!]',praed): erkannt werden. Es beginnt ebenfalls mit einer öffnenden, eckigen Klammer und einem Relationennamen, wird dann aber durch einen Punkt, einen Attributnamen und einen Vergleichsoperator fortgesetzt. Im Gegensatz dazu sind Projektionsprädikate dadurch zu erkennen, dass sie nach einer öffnenden, eckigen Klammer und dem Relationennamen eine öffnende Klammer, gefolgt von einer Liste von Attributnamen, enthalten. Diese Eigenschaft kann mithilfe von **if** re.match('\A[\[][@a-zA-Z0-9_]*\([@a-zA-Z0-9,_]*\)',praed): getestet werden.

A.1.3 Extrahieren von Relationennamen

Die Extraktion der Relationennamen aus einem Prädikat praed erfolgt abhängig von dessen Typ. Verbundprädikate enthalten zwei Relationennamen, die durch rn=re.findall ('[@a-zA-Z0-9_]*,',praed) ermittelt werden können, wobei ein anschließendes rn=string.rstrip (rn,",") das noch enthaltene Komma entfernt. Selektionsprädikate enthalten genau einen Relationennamen, welcher durch rn=re.findall ('[\[][@a-zA-Z0-9_]*',praed) gefolgt von rn=string.lstrip (rn,"[") extrahiert wird. Projektionsprädikate können mehrere Relationennamen enthalten. Diese werden durch rn=re.findall ('[@a-zA-Z0-9_]*\(',praed) und r2=string.rstrip (r1,"(") ermittelt. Die extrahierten Relationennamen werden nur dann in eine Liste relationnames eingefügt, wenn sie noch nicht in dieser enthalten sind. Wurden alle Prädikate einer Anfrage betrachtet, so wird diese Liste mittels relationnames.sort () sortiert.

A.2 Einfügen von vorverarbeiteten PSQ-Anfragen

A.2.1 Ermitteln der Attributnamen von Selektions- und Projektionsprädikaten

Das Extrahieren der Attributnamen aus Selektions- und Projektionsprädikaten erfolgt erneut mithilfe von regulären Ausdrücken, wobei diese vom Typ des Prädikates abhängen. Listing A.2 enthält den Quellcode der Funktion get_attributs (p), welche als Eingabe ein Element der vorverarbeiteten Anfrage der Form p=[praed, type] erhält und als Ausgabe eine Liste der enthaltenen absoluten[2] Attributnamen zurückliefert.

Listing A.2: Extrahieren von Attributnamen

```
def get_attributs(p):
    rv=[]        # return value
    if p[1]==4: # Projektionspraedikat
        rn=re.findall('[@a-zA-Z0-9_]*\([a-zA-Z0-9_,]*\)',p[0])
        for rl in rn:
            name=re.findall('[@a-zA-Z0-9_]*\(',rl)
            name=string.rstrip(name[0],"(")
            att=re.findall('[\(|,][a-zA-Z0-9_]*', rl)
            for i in att:
                rv.append(name+"."+string.lstrip(i,"(,"))
    else:        # Selektionspraedikat
        att=re.findall('[\[][@a-zA-Z0-9]*\.[a-zA-Z0-9]*',p[0])
        rv.append(string.lstrip(attribut[0],"["))
    return (rv)
```

A.3 Relevanztests

Bestimmen aller Verbundketten eines Teilbaums

Das grundlegende Vorgehen zum Ermitteln aller Verbundketten eines Teilbaums kann auf mehrere unterschiedliche Arten realisiert werden. Prinzipiell ist es mit einer einzigen rekursiven Funktion umsetzbar, welche als Abbruchkriterium das Erreichen eines Selektions- oder Projektionsknotens oder eines Verbundknotens ohne Kindknoten hat. Bei der praktischen Umsetzung mit Python erwies es sich aber als problematisch, hierbei die einzelnen Pfade separat aufzubauen. Daher wurde auf ein Zweiteilung zurückgegriffen (vgl. Listing A.3). Die Funktion find_last_joins (node, ljn), welche durch die Funktion complete_join(node,completed_join) aufgerufen wird, liefert die Verbundknoten, wel-

[2] Unter absoluten Attributnamen ist die Kombination aus Relationenname und Attributname zu verstehen. Beispiel: `FILME.NAME`

che auf die oben angesprochenen Knoten zum Rekursionsabbruch verweisen. Diese werden in der Variable ljn gesammelt. Darauf aufbauend werden dann in der Funktion complete_join(node,completed_join) mithilfe der Rückwärtsverkettung durch Bottom-up-Traversierung die Verbundketten des Teilbaums erzeugt.

Listing A.3: Bestimmen von Verbundketten

```
def find_last_join_nodes(node,ljn):
    if len(node.next)>0:
        for child in node.next:
            if child.type>2:
                if ljn.count(node)==0: ljn.append(node)
            else:
                find_last_join_nodes(child,ljn)
    elif node.type==2:
        ljn.append(node)

def complete_join(node,completed_join):
    ljn=[]
    find_last_join_nodes(node,ljn)
    for vk in ljn:
        temp=[]
        while vk.type==2:
            temp.append(vk)
            vk=vk.previous
        completed_join.append(temp)
```

Anhang B

Details zum Anfragemix der Evaluation

	Set 1		Set 2		Set 3		Set 4	
Operator	$\bowtie_\theta$	σ	$\bowtie_\theta$	σ	$\bowtie_\theta$	σ	$\bowtie_\theta$	σ
$<$	52	730	134	1150	228	1930	152	1314
$\leq$	60	833	136	1417	247	2274	163	1643
$=$	1032	4592	2280	6793	3176	9865	2200	7124
$\neq$	321	6655	949	12966	1832	23290	1098	14616
$\geq$	57	910	147	1386	275	2306	172	1527
$>$	57	810	146	1266	259	2044	163	1360

Tabelle B.1: Häufigkeit der Verwendung von Vergleichsoperatoren in den Anfragen

	Set 1	Set 2	Set 3	Set 4
LAEUFT_IN	2323	3425	4781	3606
FILME	2952	3315	4176	3555
SAELE	2107	2345	2671	2427
ADRESSEN	1801	1730	1499	1612
KINOS	1595	1161	1055	1294
BEHINDERUNGEN	300	483	365	397
VERKEHRSITUATION	310	640	617	473
PARKHAEUSER	142	451	505	322
KETTEN	44	105	147	145

Tabelle B.2: Häufigkeit der Verwendung einzelner Relationen in den Anfragesets

Anfrageset 1				
#Anfragen	Relationen	Verbunde	Seletionen	Projektionen
4	2	1	0	0
5	3	3	0	0
5	4	3	0	0
11	3	2	0	0
21	3	2	1	0
39	2	1	1	0
39	2	1	2	0
73	1	0	0	1
192	2	1	0	1
267	3	2	0	1
303	1	0	3	0
382	1	0	1	0
392	1	0	2	0
677	2	1	1	1
3540	1	0	1	1
4050	1	0	2	1

Anfrageset 2				
#Anfragen	Relationen	Verbunde	Seletionen	Projektionen
2	3	3	0	0
2	3	3	2	0
2	4	3	2	0
5	4	3	0	0
6	3	3	1	0
6	4	3	1	0
7	5	4	0	1
11	3	2	3	0
16	3	2	1	0
20	2	1	4	0
21	3	2	2	0
23	2	1	3	0
28	2	1	2	0
45	3	3	1	1
57	4	3	1	1
101	1	0	5	0
171	1	0	4	0
193	3	2	2	1
210	3	2	0	1
237	3	2	1	1
243	1	0	3	0
453	2	1	3	1
503	2	1	2	1
533	2	1	1	1
1457	1	0	4	1
2299	1	0	3	1
3198	1	0	2	1

Anfrageset 3				
#Anfragen	Relationen	Verbunde	Seletionen	Projektionen
1	3	3	4	0
1	5	4	2	0
2	3	3	2	0
2	4	3	3	0
3	4	3	4	0
5	3	3	3	0
9	3	2	5	0
9	5	4	1	1
10	4	3	2	0
11	5	4	0	1
12	5	4	2	1
21	2	1	5	0
22	3	2	3	0
23	3	2	4	0
29	2	1	6	0
41	3	3	3	1
45	1	0	7	0
48	2	1	4	0
50	4	3	3	1
71	3	3	2	1
73	4	3	2	1
81	3	3	1	1
100	1	0	6	0
108	4	3	1	1
198	1	0	5	0
307	3	2	4	1
395	3	2	3	1
418	3	2	2	1
515	2	1	5	1
671	2	1	4	1
912	2	1	3	1
992	1	0	6	1
1828	1	0	5	1
2987	1	0	4	1

Anfrageset 4				
#Anfragen	Relationen	Verbunde	Seletionen	Projektionen
1	3	3	3	0
1	3	3	6	0
1	4	3	2	0
1	4	3	3	0
1	4	3	7	0

Anfrageset 4 (Vortsetzung)				
#Anfragen	Relationen	Verbunde	Seletionen	Projektionen
1	4	3	9	1
1	5	4	3	1
2	1	0	10	0
2	2	1	10	1
2	3	2	6	0
2	3	2	7	0
2	3	3	0	0
2	4	3	5	1
2	4	3	6	1
2	4	3	8	1
3	2	1	7	0
3	2	1	8	0
3	3	2	5	0
3	3	3	1	0
3	3	3	6	1
3	4	3	0	0
3	4	3	1	0
4	2	1	0	0
4	2	1	5	0
4	5	4	0	1
4	5	4	1	1
4	5	4	2	1
4	5	4	7	1
5	1	0	9	0
5	4	3	7	1
6	5	4	6	1
7	3	2	0	0
7	3	3	4	1
7	3	3	5	1
7	5	4	4	1
8	1	0	8	0
8	3	2	3	0
8	3	2	9	1
8	3	3	3	1
8	4	3	3	1
9	2	1	6	0
9	3	2	1	0
9	3	2	4	0
9	4	3	4	1
10	5	4	5	1
11	2	1	4	0
11	2	1	9	1
12	3	2	2	0
13	2	1	2	0
13	2	1	3	0
13	3	2	8	1
15	1	0	7	0
15	4	3	2	1
17	3	3	2	1
18	2	1	1	0
19	1	0	10	1
24	2	1	8	1
28	3	3	1	1
30	1	0	6	0
31	4	3	1	1
34	4	3	0	1
35	1	0	9	1
35	3	2	7	1
40	3	3	0	1
52	1	0	5	0
52	2	1	7	1
52	3	2	6	1
56	3	2	5	1
73	1	0	0	1
78	1	0	8	1
83	3	2	4	1
91	1	0	4	0
91	2	1	6	1
101	3	2	3	1
113	1	0	7	1
113	3	2	0	1
114	3	2	2	1
120	2	1	0	1
127	1	0	3	0
139	2	1	5	1
141	3	2	1	1
173	1	0	2	0
181	1	0	1	0
185	2	1	4	1
249	2	1	3	1
277	1	0	6	1
284	2	1	2	1
312	2	1	1	1
489	1	0	5	1
791	1	0	4	1
1270	1	0	3	1
1759	1	0	1	1
1822	1	0	2	1

Tabelle B.3: Zusammensetzung der Anfragesets

Anhang C

Beispiel: TJ-Anfrage bei Updates

Aus Platzgründen konnte im Fließtext nicht die Relevanztestanfragen für Verbunde bei Updates eingebunden werden. Daher ist dieses Beispiel hier aufgeführt. Auffällig ist hierbei, dass die Attributnamen umbenannt wurden. Dies ist notwendig, da Datenbankmanagementsysteme eine Zeichenkette der Form `Relationenname.Attributname` nicht als eigenständigen Attributnamen in einer anderen Relation verwenden können. Daher wurde stattdessen der SHA-Hash-Wert dieser Zeichenkette verwendet, wobei diesem noch `SMOS_` vorgestellt wurde um Attributnamen, die mit eine Ziffer beginnen zu vermeiden.

```
SELECT DISTINCT
  SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
  SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481 c,
  SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
  FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
  FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
  LAEUFT_IN.UHRZEIT AS SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
  LAEUFT_IN.DATUM AS SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
  FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
FROM LAEUFT_IN, SAELE, FILME
WHERE LAEUFT_IN.SID=SAELE.SID AND FILME.FID=LAEUFT_IN.FID
UNION
  SELECT DISTINCT
    SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
    SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481c,
    SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
    FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
    FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
    LAEUFT_IN.UHRZEIT AS SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
    LAEUFT_IN.DATUM AS SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
    FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
  FROM LAEUFT_IN, SMOS_0661198d24ebc3ff59945938273c55fece5d25ee AS SAELE,
    FILME
  WHERE LAEUFT_IN.SID=SAELE.SID AND FILME.FID=LAEUFT_IN.FID
  EXCEPT
    ((SELECT DISTINCT
      SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
```

```
  SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481c,
  SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
  FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
  FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
  LAEUFT_IN.UHRZEIT AS SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
  LAEUFT_IN.DATUM AS SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
  FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
FROM LAEUFT_IN, SAELE, FILME
WHERE LAEUFT_IN.SID=SAEL E.SID AND FILME.FID=LAEUFT_IN.FID
EXCEPT
  SELECT DISTINCT
    SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
    SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481c,
    SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
    FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
    FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
    LAEUFT_IN.UHRZEIT AS SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
    LAEUFT_IN.DATUM AS SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
    FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
  FROM LAEUFT_IN, SAELE, FILME
  WHERE LAEUFT_IN.SID=SAELE.SID AND FILME.FID=LAEUFT_IN.FID
    AND SAELE.plaetze>300 )
  UNION
    (SELECT DISTINCT
      SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
      SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481c,
      SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
      FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
      FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
      LAEUFT_IN.UHRZEIT AS
        SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
      LAEUFT_IN.DATUM AS SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
      FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
    FROM LAEUFT_IN, SAELE, FILME
    WHERE LAEUFT_IN.SID=SAELE.SID AND FILME.FID=LAEUFT_IN.FID
    INTERSECT
      SELECT DISTINCT
        SAELE.NAME AS SMOS_f0d6212d7f4b8f73275aa8f559ddfa5f3eb1c382,
        SAELE.PLAETZE AS SMOS_200ba11eb9ac6b83fb979cd4c6b7f94ba13a481c,
        SAELE.TON AS SMOS_4f0283af2ee25c597ab4f5d693340f03f804b1f7,
        FILME.NAME AS SMOS_887c633ec731e2b969c388d83919041b04ac514f,
        FILME.FSK AS SMOS_7acf1dca51c0ec61f98ab37aa9ba4c02bb1d9583,
        LAEUFT_IN.UHRZEIT AS
          SMOS_df873860bc7cc4f5fafe1aef07ad09dcebd6b344,
        LAEUFT_IN.DATUM AS
          SMOS_15532cf20f7ae55be0558bc952fd5515915d9c7c,
        FILME.GENRE AS SMOS_ea325b8f8b63ac564a93a8e28cabad38d0f12525
      FROM LAEUFT_IN, SMOS_0661198d24ebc3ff59945938273c55fece5d25eeAS
        SAELE, FILME
      WHERE LAEUFT_IN.SID=SAELE.SID AND FILME.FID=LAEUFT_IN.FID));
```

Abschließende Bemerkungen

Für die Erstellung dieser Dissertation und die im Rahmen der Evaluation durchgeführten Implementierungs- und Entwicklungsarbeiten wurden ausschließlich Softwareprodukte verwendet, welche durch diverse Personen zum Teil unentgeltlich entwickelt wurden. Ihnen möchte ich auf diesem Wege recht herzlich für ihre geleistete Arbeit danken. Die Dissertationsschrift wurde mit LaTeX erstellt, für die Bibliographie wurde BibTeX verwendet und die Diagramme wurden mithilfe von Gnuplot generiert. Beim Erstellen der Abbildungen kam Xfig zum Einsatz. Alle Implementierungs- und Schreibarbeiten erfolgten auf einem Linux PC mithilfe des Editors Emacs. Für die prototypische Umsetzung wurde die Programmiersprache Python genutzt. Als Datenbankmanagementsystem wurde Postgres verwendet und die Anbindung von Python an Postgres erfolgte mithilfe der PyGreSQL-Schnittstelle.

Literaturverzeichnis

[ABF+03] Alcala, F.; Beel, J.; Frenkel, A.; Gipp, B.; Lülf, J.; Höpfner, H.: Ortung von mobilen Geräten für die Realisierung lokationsbasierter Diensten. In Türker, C. (Hrsg.): *Mobilität und Informationssysteme - Workshop des GI-Arbeitskreises „Mobile Datenbanken und Informationssysteme“, 16. und 17. Oktober 2003, Zürich*, Nr. 422 der Reihe Technical Reports, S. 12–21. Eidgenössische Technische Hochschule Zürich, Zürich, Switzerland, Oktober 2003.

[ABF+04] Alcala, F.; Beel, J.; Frenkel, A.; Gipp, B.; Lülf, J.; Höpfner, H.: UbiLoc: A System for Locating Mobile Devices using Mobile Devices. In Kyamakya, K. (Hrsg.): *Proceedings of 1st Workshop on Positioning, Navigation and Communication 2004 (WPNC 04), University of Hanover, Germany, 26. March 2004*, Nr. 0.1 der Reihe Hannoversche Beiträge zur Nachrichtentechnik, S. 43–48. NICCIMON, IEEE, VDI, Shaker Verlag GmbH, Aachen, März 2004.

[Ada82] Adams, D.: *Das Restaurant am Ende des Universums*. Rogner & Bernhard GmbH & Co. Verlags KG, Hamburg, 1982.

[AHV96] Abiteboul, S.; Hull, R.; Vianu, V.: *Foundation of Databases*. Addison-Wesley Publishing Company, Inc., 1996.

[ASU79] Aho, A. V.; Sagiv, Y.; Ullman, J. D.: Equivalences among relational expressions. *SIAM Journal on Computing*, Band 8, Nr. 2, S. 218–246, Mai 1979.

[AVL62a] Adelson-Velskii, G. M.; Landis, E. M.: An algorithm for the organization of information. *Doklady Akademii Nauk*, Band 146, S. 263–266, 1962. in Russisch, übersetze Version erschienen als [AVL62b].

[AVL62b] Adelson-Velskii, G. M.; Landis, E. M.: An algorithm for the organization of information. *Soviet Math. Doklady*, Band 3, S. 1259–1263, 1962.

[BCL89] Blakeley, J. A.; Coburn, N.; Larson, P.-A.: Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems (TODS)*, Band 14, Nr. 3, S. 369–400, September 1989. ebenfalls publiziert als [BCL98].

[BCL98] Blakeley, J. A.; Coburn, N.; Larson, P.-A.: Updating derived relations: Detecting irrelevant and autonomously computable updates. In Gupta, A.; Mumick, I. S. (Hrsg.): *Materialized Views*, S. 295–322. MIT Press, London, England, 1998.

[BG04] Bittner, R.; Gollmick, C.: Implementierung von Konfliktverweidungstechniken für den mobilen Datenbankzugriff. In Höpfner, H.; Saake, G. (Hrsg.): *Beitragsband zum Workshop Grundlagen und Anwendungen mobiler Informationstechnologie, 23./24. März 2004 in Heidelberg*, Nr. 4/2004 der Reihe Preprints der Fakultät für Informatik, S. 13–22. Otto-von-Guerick-Universität Magdeburg, Magdeburg, Februar 2004.

[BKSS90a] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In Garcia-Molina, H. (Hrsg.): *Proceedings of the 1990 ACM SIGMOD international conference on Management of data, May 23-26, 1990, Atlantic City, New Jersey, United States*, S. 322–331. ACM Press, New York, NY, USA, 1990. ebenfalls publiziert als [BKSS90b].

[BKSS90b] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, Band 19, Nr. 2, S. 322–331, Juni 1990.

[Boe03] Boenigk, C.: *PostgreSQL: Grundlagen, Praxis, Anwendungsentwicklung mit PHP*. dpunkt.verlag GmbH, Heidelberg, 2003.

[Buc02] Buchmann, E.: Konzeption und Implementierung eines Speichermanagers für ein konfigurierbares, leichtgewichtiges DBMS. Diplomarbeit, Otto-von-Guericke Universität Magdeburg, Februar 2002.

[CB03] Connolly, T. M.; Begg, C. E.: *Database Solutions: A step-by-step guide to building databases*. Pearson Education Limited, Essex, England, 2. Auflage, Oktober 2003.

[CGH98] Cremers, A. B.; Griefahn, U.; Hinze, R.: *Deduktive Datenbanken*. Vieweg Verlagsgesellschaft, Wiesbaden, Dezember 1998.

[CK86] Cosmadakis, S.; Kanellakis, P.: Parallel evaluation of recursive rule queries. In Silberschatz, A. (Hrsg.): *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, March 24 - 26, 1986, Cambridge, Massachusetts, United States*, S. 280–293. ACM Press, New York, NY, USA, 1986.

[CM77] Chandra, A. K.; Merlin, P. M.: Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing, Boulder, Colorado, United States*, S. 77–90. New York, NY, USA, 1977.

[Cod69] Codd, E. F.: Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. *IBM Research Report, San Jose, California*, Band RJ599, 1969.

[Cod70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM (CACM)*, Band 13, Nr. 6, S. 377–387, 1970.

[Cod72] Codd, E. F.: Relational Completeness of Data Base Sublanguages. In Rustin, R. J. (Hrsg.): *Data Base Systems (Proceedings of the 6th Courant Computer Science Symposium, May 24-25, 1971, New York, N.Y.)*, Automatic Computation, S. 65–98. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[Cod91] Codd, E. F.: *The Relational Model for Database Management: Version 2*. Addison-Wesley Publishing Company, New York, USA, 2. Auflage, Juli 1991.

[CR97] Chekuri, C.; Rajaraman, A.: Conjunctive Query Containment Revisited. In Afrati, F. N.; Kolaitis, P. G. (Hrsg.): *Proceedings of the 6th International Conference on Database Theory - ICDT '97, Delphi, Greece, January 8-10, 1997*, Lecture Notes in Computer Science, Band 1186, S. 56–70. Springer-Verlag, Heidelberg, 1997. ebenfalls publiziert als [CR00].

[CR00] Chekuri, C.; Rajaraman, A.: Conjunctive Query Containment Revisited. *Theoretical Computer Science*, Band 239, Nr. 2, S. 211–229, Mai 2000.

[CRZ03] Chaudhri, A. B.; Rashid, A.; Zicari, R. (Hrsg.): *XML Data Management - Native XML and XML-Enabled Database Systems*. Addison-Wesley Publishing Company, Boston, USA, März 2003.

[CTO97] Cai, J.; Tan, K.-L.; Ooi, B. C.: On incremental cache coherency schemes in mobile computing environments. In Gray, A.; Larson, P.-Å. (Hrsg.): *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, S. 114–123. IEEE Computer Society, Los Altos, CA, USA, 1997.

[Dat01] Date, C. J.: *The Database Relational Model - A Retrospective Review and Analysis*. Addison Wesley Longman, Inc., 2001.

[DD97] Date, C. J.; Darwen, H.: *A guide to the SQL standard: a user's guide to the standard database language*. Addison-Wesley Longman, Inc., Reading, Massechusetts, USA, 4. Auflage, 1997.

[DFJ+96] Dar, S.; Franklin, M. J.; Jónsson, B. Þ.; Srivastava, D.; Tan, M.: Semantic Data Caching and Replacement. In Vijayaraman, T. M.; Buchmann, A. P.; Mohan, C.; Sarda, N. L. (Hrsg.): *Proceedings of 22th International Conference on Very Large Data Bases (VLDB'96), September 3-6, 1996, Mumbai (Bombay), India*, S. 330–341. Morgan Kaufmann, San Fransisco, CA, USA, September 1996.

[DGK82] Dayal, U.; Goodman, N.; Katz, R. H.: An extended relational algebra with control over duplicate elimination. In Ullman, J. D.; Aho, A. V. (Hrsg.): *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, March 29 - 31, 1982, Los Angeles, California*, S. 117–123. ACM Press, New York, NY, USA, 1982.

[dlB59] Briandais, R. d. l.: File Searching Using Variable Length Keys. In *Proceedings of the AFIPS Western Joint Computer Conference, San Francisco, California, USA*, AFIPS Press, Band 15, S. 295–298. Montvale, NJ, USA, 1959.

[Elk90] Elkan, C.: Independence of logic database queries and update. In Rosenkrantz, D. J.; Sagiv, Y. (Hrsg.): *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, April 02 - 04, 1990, Nashville, Tennessee, United States*, S. 154–160. ACM Press, New York, NY, USA, 1990.

[EN02] Elmasri, R.; Navathe, S. B.: *Grundlagen von Datenbanksystemen*. Pearson Education Deutschland GmbH, München, 3. Auflage, 2002.

[ETS93] ETSI: European digital cellular telecommunications system; Attachment requirements for Global System for Mobile communications (GSM) mobile stations. Technical basis for regulation, European Telecomunications Standards Institute, Sophia Antipolis Cedex, November 1993.

[ETS98] ETSI: Digital cellular telecommunications system (Phase 2+) (GSM); G eneral Packet Radio Service (GPRS); Service description; Stage 1 (GSM 02.60 version 6.1.1). European standard (communication series), European Telecomunications Standards Institute, Sophia Antipolis Cedex, November 1998.

[ETS99a] ETSI: Digital cellular telecommunications system (Phase 2+); Attachment requirements for Global System for Mobile communications (GSM); High Speed Circuit Switched Data (HSCSD) multislot mobile stations; Access (GSM 13.34 version 5.0.3). European standard (communication series), European Telecomunications Standards Institute, Sophia Antipolis Cedex, März 1999.

[ETS99b] ETSI: Telecommunications Management Network (TMN); Universal Mobile Telecommunications System (UMTS); Management architecture framework; Overview, processes and principles. Etsi standard, European Telecomunications Standards Institute, Sophia Antipolis Cedex, Januar 1999.

[FKL03] Fanghänel, T.; Karlsson, J. S.; Leung, C.: DB2 Everyplace Database Release 8.1: Architecture and Key Features. *Datenbank Spektrum*, Band 3, Nr. 5, S. 9–15, 2003.

[Flo62] Floyd, R. W.: Algorithm 97: Shortest path. *Communications of the ACM*, Band 5, Nr. 6, S. 345, Juni 1962.

[Fra01] Franklin, M. J.: Challenges in ubiquitous data management. In Wilhelm, R. (Hrsg.): *Informatics - 10 Years Back. 10 Years Ahead*, S. 24–33. Springer-Verlag, Heidelberg, 2001.

[Fre59] Fredkin, E.: *Trie memory*. Information Memorandum, Bolt Beranek and Newman Inc., Cambridge, MA, 1959.

[Fre60] Fredkin, E.: Trie memory. *Communications of the ACM*, Band 3, Nr. 9, S. 490–499, August 1960.

[Fri01] Friedl, J. E.: *Regulräre Ausdrücke*. O'Reilly Verlag, Köln, Juni 2001. 2., korrigierter Nachdruck.

[Gep02] Geppert, A.: *Objektrelationale und objektorientierte Datenbankkonzepte und -systeme*. dpunkt.verlag GmbH, Heidelberg, 2002.

[GG99] Godfrey, P.; Gryz, J.: Answering queries by semantic caches. In Bench-Capon, T.; Soda, G.; Tjoa, A. M. (Hrsg.): *Database and Expert Systems Applications: Proceedings of the 10th International Conference, DEXA'99, Florence, Italy, August/September 1999*, Lecture Notes in Computer Science, Band 1677, S. 485–498. Springer-Verlag, Heidelberg, September 1999.

[GH01a] Gomez, D.; Höpfner, H.: Postgres and MySQL in direct comparison. *Linux Magazine*, Band 8, S. 23–29, Mai 2001.

[GH01b] Gomez, D.; Höpfner, H.: Postgres und MySQL im Direktvergleich. *Linux Magazin*, Band 4, S. 44–53, April 2001. ebenfalls publiziert in Englisch als [GH01a].

[GHOS96] Gray, J.; Helland, P.; O'Neil, P. E.; Shasha, D.: The Dangers of Replication and a Solution. In Jagadish, H. V.; Mumick, I. S. (Hrsg.): *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, SIGMOD Record, Band 25, Nr. 2, S. 173–182. ACM Press, New York, NY, USA, Juni 1996.

[GLS99] Gottlob, G.; Leone, N.; Scarcello, F.: Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, United States*, S. 21–32. ACM Press, New York, NY, USA, 1999. ebenfalls publiziert als [GLS02].

[GLS01a] Gottlob, G.; Leone, N.; Scarcello, F.: Hypertree Decompositions: A Survey. In Goos, G.; Hartmanis, J.; Leeuwen, J. v. (Hrsg.): *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science August 27 - 31, 2001, Marianske Lazne, Czech Republic*, Lecture Notes in Computer Science (LNCS), Band 2136, S. 37–57. Springer-Verlag, Heidelberg, 2001.

[GLS01b] Gottlob, G.; Leone, N.; Scarcello, F.: The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, Band 48, Nr. 3, S. 431–498, Mai 2001.

[GLS02] Gottlob, G.; Leone, N.; Scarcello, F.: Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, Band 64, Nr. 2, S. 579–627, Mai 2002.

[GMAB+84] Garcia-Molina, H.; Allen, T.; Blaustein, B. T.; Chilenskas, R. M.; Ries, D. R.: Data-Patch: Integrating Inconsistent Copies of a Database After a Partition. In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems, October 17-19, Clearwater Beach, FL, USA*, S. 38–44. IEEE Computer Society Press, Los Altos, CA, USA, Januar 1984.

[GMS93a] Gupta, A.; Mumick, I. S.; Subrahmanian, V. S.: Maintaining views incrementally. In Buneman, P.; Jajodia, S. (Hrsg.): *Proceedings of the 1993 ACM SIGMOD international conference on Management of data, May 25 - 28, 1993, Washington, D.C., United States*, S. 157–166. ACM Press, New York, NY, USA, 1993. ebenfalls publiziert als [GMS93b] und [GMS98].

[GMS93b] Gupta, A.; Mumick, I. S.; Subrahmanian, V. S.: Maintaining views incrementally. *ACM SIGMOD Record*, Band 22, Nr. 2, S. 157–166, Juni 1993.

[GMS98] Gupta, A.; Mumick, I. S.; Subrahmanian, V. S.: Maintaining views incrementally. In Gupta, A.; Mumick, I. S. (Hrsg.): *Materialized Views*, S. 177–190. MIT Press, London, England, 1998.

[Goe08] Goethe, J. W.: *Faust – Der Tragödie erster Teil*. 1808.

[Gol03] Gollmick, C.: Nutzerdefinierte Replikation zur Realisierung neuer mobiler Datenbankanwendungen. In Weikum, G.; Schöning, H.; Rahm, E. (Hrsg.): *Tagungsband der 10. GI-Fachtagung Datenbanksysteme für Business, Technology und Web (BTW 2003), 26.-28. Februar 2003, Leipzig, Germany*, Lecture Notes in Informatics (LNI) - Proceedings, Band P-26, S. 453–462. Gesellschaft für Informatik e.V. (GI), Bonn, Februar 2003.

[GSW96a] Guo, S.; Sun, W.; Weiss, M. A.: On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems. *IEEE Transactions on Knowledge and Data Engineering*, Band 8, Nr. 4, S. 604–616, August 1996.

[GSW96b] Guo, S.; Sun, W.; Weiss, M. A.: Solving satisfiability and implication problems in database systems. *ACM Transactions on Database Systems (TODS)*, Band 21, Nr. 2, S. 270–293, Juni 1996.

[Gut84a] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In Smith, D.; Yormark, B. (Hrsg.): *Proceedings of the 1984 ACM SIGMOD international conference on Management of data, June 18-21, 1984, Boston, Massa-*

chusetts, S. 47–57. ACM Press, New York, NY, USA, 1984. ebenfalls publiziert als [Gut84b].

[Gut84b] Guttman, A.: R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, Band 14, Nr. 2, S. 47–57, Juni 1984.

[Hal01] Halevy, A. Y.: Answering queries using views: A survey. *The International Journal on Very Large Data Bases*, Band 10, Nr. 4, S. 270–294, Dezember 2001.

[Höp01] Höpfner, H.: Replikationstechniken für mobile Datenbanken. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, März 2001.

[Höp02] Höpfner, H.: Replication in Mobile Information Systems. In Schubert, S.; Reusch, B.; Jesse, N. (Hrsg.): *Informatik bewegt - Proceedings der 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 30. September - 3. Oktober, Dortmund, Germany*, Lecture Notes in Informatics (LNI) - Proceeding, Band P-19, S. 590–593. Gesellschaft für Informatik e.V. (GI), Bonn, September 2002.

[Höp04] Höpfner, H.: Serverseitige Auswertung von Indexen semantischer, clientseitiger Caches in mobilen Informationssystemen. In Dadam, P.; Reichert, M. (Hrsg.): *INFORMATIK 2004 - Informatik verbindet – Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI)), 20.-24. September 2004, Ulm, Germany*, Lecture Notes in Informatics (LNI) - Proceedings, Band P-50, Nr. 1, S. 298–302. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, Bonn, September 2004.

[HS00] Heuer, A.; Saake, G.: *Datenbanken: Konzepte und Sprachen*. MITP GmbH, Bonn, 2. Auflage, 2000.

[HS03a] Höpfner, H.; Sattler, K.-U.: SMoS: A Scalable Mobility Server. In James, A. E.; Younas, M. (Hrsg.): *Poster Proceedings of Twentieth British National Conference on Databases (BNCOD20), Coventry, UK 15th - 17th July, 2003*, S. 49–52. Coventry University, Juli 2003.

[HS03b] Höpfner, H.; Sattler, K.-U.: Semantic Replication in Mobile Federated Information Systems. In James, A.; Conrad, S.; Hasselbring, W. (Hrsg.): *Proceedings of the Fifth International Workshop on Engineering Federated Information Systems (EFIS2003), Coventry, UK 17th - 18th July, 2003*, S. 36–41. Ios Press Inc, Amsterdam, August 2003.

[HS03c] Höpfner, H.; Sattler, K.-U.: Towards Trie-Based Query Caching in Mobile DBS. In König-Ries, B.; Klein, M.; Obreiter, P. (Hrsg.): *Post-Proceedings of the Workshop "Persistence, Scalability, Transactions - Database Mechanisms for Mobile Applications", Karlsruhe, 10.-11. April 2003*, Lecture Notes in Informatics (LNI) - Proceedings, Band 43, S. 106–121. Gesellschaft für Informatik (GI), Bonn, 2003.

[HS04] Höpfner, H.; Schallehn, E.: Anfragegeneralisierung zur komprimierten Repräsentation von Indexen semantischer Caches auf mobilen Endgeräten. In Höpfner, H.; Saake, G. (Hrsg.): *Beitragsband zum Workshop Grundlagen und Anwendungen mobiler Informationstechnologie, 23./24. März 2004 in Heidelberg*, Nr. 4/2004 der Reihe Preprints der Fakultät für Informatik, S. 63–70. Otto-von-Guerick-Universität Magdeburg, Magdeburg, Februar 2004.

[HSS04a] Höpfner, H.; Schosser, S.; Sattler, K.-U.: An Indexing Scheme for Update Notification in Large Mobile Information Systems. In Lindner, W.; Mesiti, M.; Türker, C.; Tzikzikas, Y.; Vakali, A. (Hrsg.): *Current Trends in Database Technology - EDBT 2004 Workshops: PhD, DataX, PIM, P2PDB, ClustWeb, Heraklion, Greece, March 14-18, 2004, Revised Papers*, Lecture Notes in Computer Science (LNCS), Band 3268. Springer-Verlag, Berlin, 2004. to appear.

[HSS04b] Höpfner, H.; Schosser, S.; Sattler, K.-U.: An Indexing Scheme for Update Propagation in Large Mobile Information Systems. In Türker, C.; König-Ries, B. (Hrsg.): *Proceedings of the EDBT-Workshop on Pervasive Information Management, 18. March 2004, Heraklion - Crete, Greece*, S. 31–42, 2004.

[HSS04c] Höpfner, H.; Schosser, S.; Sattler, K.-U.: Toward Trie-based Indexing of Mobile Clients in Large Mobile Information Systems. Preprint Nr. 02, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Technische und Betriebliche Informationssysteme, Januar 2004.

[HV01] Herrera-Viedma, E.: An Information Retrieval System with Ordinal Linguistic Weighted Queries Based on Two Weighting Elements. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS)*, Band 9, S. 77–88, September 2001.

[KB96] Keller, A. M.; Basu, J.: A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, Band 5, Nr. 1, S. 35–47, Januar 1996.

[KBL04] Kifer, M.; Bernstein, A.; Lewis, P. M.: *Database Systems: An Application-Oriented Approach*. Addison Wesley, New York, USA, 2. Auflage, Mai 2004.

[Kho90] Khosraviyani, F.: Using binary search on a linked list. *ACM SIGCSE Bulletin*, Band 22, Nr. 3, September 1990.

[KK01] Kline, K.; Kline, D.: *SQL in a nutshell*. O'Reilly & Associates, Inc., Sebastopol, 1. Auflage, 2001.

[Klu88] Klug, A.: On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, Band 35, Nr. 1, S. 146–160, Januar 1988.

[KMT98] Kolaitis, P. G.; Martin, D. L.; Thakur, M. N.: On the Complexity of the Containment Problem for Conjunctive Queries with Built-in Predicates. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, S. 197–204. ACM Press, New York, NY, USA, 1998.

[KP97a] Kuenning, G. H.; Popek, G. J.: Automated Hoarding for Mobile Computers. In Waite, W. M. (Hrsg.): *Proceedings of the sixteenth ACM symposium on Operating systems principles, October 05-08, 1997, Saint Malo, France*, S. 264–275. ACM Press, New York, NY, USA, 1997.

[KP97b] Kuenning, G. H.; Popek, G. J.: Automated Hoarding for Mobile Computers. *ACM SIGOPS Operating Systems Review*, Band 31, Nr. 5, S. 264–275, Dezember 1997.

[KR01a] Kubach, U.; Rothermel, K.: Ein Hoarding-Verfahren für ortsbezogene Informationen. *Informatik Forschung und Entwicklung*, Band 16, Nr. 4, S. 189–199, November 2001.

[KR01b] Kubach, U.; Rothermel, K.: Exploiting location information for infostation-based hoarding. In Rose, C. (Hrsg.): *Proceedings of the 7th annual international conference on Mobile computing and networking, July 16-21, 2001, Rome, Italy Rome, Italy*, S. 15–27. ACM Press, New York, NY, USA, 2001. ebenfalls publiziert in Deutsch als [KR01a].

[KS92] Kistler, J. J.; Satyanarayanan, M.: Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, Band 10, Nr. 1, S. 3–25, Februar 1992.

[Leh02] Lehner, W.: *Subskriptionssysteme – Marktplatz für omnipräsente Informationssysteme*, TEUBNER-TEXTE zur Informatik, Band 36. B.G.Teubner GmbH, Stuttgart, Mai 2002.

[LHJ+03] Lee, M. L.; Hsu, W.; Jensen, C. S.; Cui, B.; Teo, K. L.: Supporting frequent updates in r-trees: A bottom-up approach. In Freytag, J.-C.; Lockemann, P. C.; Abiteboul, S.; Carey, M.; Selinger, P.; Heuer, A. (Hrsg.): *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003), 9-12 september 2003, Berlin, Germany*, S. 608–619. Morgan Kaufmann Publishers Inc., San Fransisco, CA, USA, Oktober 2003.

[LLS99] Lee, K. C. K.; Leong, H. V.; Si, A.: Semantic query caching in a mobile environment. *ACM SIGMOBILE Mobile Computing and Communications Review*, Band 3, Nr. 2, S. 28–36, April 1999.

[LLS00] Lee, K. C. K.; Leong, H. V.; Si, A.: Incremental View Maintenance for Mobile Databases. *Knowledge and Information Systems*, Band 2, Nr. 4, S. 413–437, November 2000.

[LMSS93] Levy, A.; Mumick, I. S.; Sagiv, Y.; Shmueli, O.: Equivalence, query-reachability and satisfiability in Datalog extensions. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, May 25 - 28, 1993, Washington, D.C., United States*, S. 109–122. ACM Press, New York, NY, USA, 1993.

[LS92] Levy, A.; Sagiv, Y.: Constraints and redundancy in datalog. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, June 02 - 05, 1992, San Diego, California, United States*, S. 67–80. ACM Press, New York, NY, USA, 1992.

[LS93] Levy, A. Y.; Sagiv, Y.: Queries Independent of Updates. In Agrawal, R.; Baker, S.; Bell, D. A. (Hrsg.): *Proceedings of th 19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland*, S. 171–181. Morgan Kaufmann, San Fransisco, CA, USA, Januar 1993.

[Mai83] Maier, D.: *The Theory of Relational Databases*. Computer Science Press, Inc., Rockville, Maryland, 1983.

[MHF00] Millstein, T.; Halevy, A.; Friedman, M.: Query containment for data integration systems. In Vianu, V.; Gottlob, G. (Hrsg.): *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, S. 67–75. ACM Press, New York, NY, USA, 2000. ebenfalls publiziert als [MHF03].

[MHF03] Millstein, T.; Halevy, A.; Friedman, M.: Query containment for data integration systems. *Journal of Computer and System Sciences*, Band 66, Nr. 1, S. 20–39, Februar 2003.

[Mit95] Mitschang, B.: *Anfrageverarbeitung in Datenbanksystemen: Entwurf- und Implementierungskonzepte*. Vieweg Verlag, Wiesbaden, 1995.

[MMS79] Maier, D.; Mendelzon, A. O.; Sagiv, Y.: Testing implications of data dependencies. *ACM Transactions on Database Systems (TODS)*, Band 4, Nr. 4, S. 455–469, Dezember 1979.

[Mor68] Morrison, D. R.: PATRICIA-Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, Band 15, Nr. 4, S. 514–534, Oktober 1968.

[MS04] Mutschler, B.; Specht, G.: *Mobile Datenbanksysteme*. Xpert.press. Springer-Verlag, Berlin, Juli 2004.

[MU83] Maier, D.; Ullman, J. D.: Fragments of relations. In Stonebraker, M. (Hrsg.): *Proceedings of the 1983 ACM SIGMOD international conference on Management of data, May 23 - 26, 1983, San Jose, California*. ACM Press, New York, NY, USA, 1983.

[Mul00] Mullins, C. S.: You Can Take it With You: MobileDatabases. *DB2 magazine*, Band 5, Nr. 3, S. 19–22, 2000.

[Pan00] Pandya, R.: *Mobile and Personal Communication Services and Systems*. IEEE Series on Digital & Mobile Communication. John Wiley & Sons, Inc., 2000.

[Pap85] Papadimitriou, C.: A note on the expressive power of prolog. *Bulletin of the EATCS*, Band 26, S. 21–23, Juni 1985.

[Pei04] Peissig, J.: guideport - an information and guidance system. In Kyamakya, K. (Hrsg.): *Proceedings of 1st Workshop on Positioning, Navigation and Communication 2004 (WPNC 04), University of Hanover, Germany, 26. March 200 4*, Nr. 0.1 der Reihe Hannoversche Beiträge zur Nachrichtentechnik, S. 1–17. NICCIMON, IEEE, VDI, Shaker Verlag GmbH, Aachen, März 2004.

[Pra77] Pratt, V. R.: Two Easy Theories Whose Combination is Hard. unpublished manuscript, Massachusetts Institute of Technology (M.I.T.), Cambridge, MA, USA, Januar 1977. http://boole.stanford.edu/pub/sefnp.pdf.

[PS97] Pitoura, E.; Samaras, G.: *Data Management for Mobile Computing*. Kluwer Academic Publishers, Oktober 1997.

[PSPB04] Prante, T.; Stenzel, R.; Petrovic, K.; Bayon, V.: Exploiting Context Histories: A Cross-Tool and Cross-Device Approach to Reduce Compartmentalization when Going Back. In Dadam, P.; Reichert, M. (Hrsg.): *INFORMATIK 2004 - Informatik verbindet – Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI)), 20.-24. September 2004, Ulm, Germany*, Lecture Notes in Informatics (LNI) - Proceedings, Band P-50, Nr. 1, S. 314–318. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, Bonn, September 2004.

[Qia96] Qian, X.: Query folding. In Su, S. Y. W. (Hrsg.): *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, S. 48–55. IEEE Computer Society, Washington, DC, USA, 1996.

[Rad04a] Radage, K.: *Oracle Database Lite 10g Business White Paper*. Oracle Corporation, Redwood Shores, USA, Januar 2004. `http://www.oracle.com/technology/products/lite/\ lite_10g_buspaper.pdf`.

[Rad04b] Radage, K.: *Oracle Database Lite Overview*. Oracle Corporation, Redwood Shores, USA, Januar 2004. `http://www.oracle.com/technology/products/lite/\ lite_datasheet_10g.pdf`.

[RD98] Ren, Q.; Dunham, M. H.: Semantic caching and query processing. Technischer Bericht Nr. 98-CSE-04, Southern Methodist University, Deptartement of Computer Science and Engineering, Dallas, TX, Mai 26 1998.

[RD99] Ren, Q.; Dunham, M. H.: Using Clustering for Effective Management of a Semantic Cache in Mobile Computing. In Banerjee, S.; Chrysanthis, P. K.; Pitoura, E. (Hrsg.): *Proceedings of the 1st ACM international workshop on Data engineering for wireless and mobile access, August 20, 1999, Seattle, Washington, United States*, S. 94–101. ACM Press, New York, NY, USA, 1999.

[RD00] Ren, Q.; Dunham, M. H.: Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In Pickholtz, R.; Das, S. K.; Caceres, R.; Garcia-Luna-Aceves, J. J. (Hrsg.): *Proceedings of the 6th annual international conference on Mobile computing and networking, August 06 - 11, 2000, Boston, Massachusetts, United States*, S. 210–221. ACM Press, New York, NY, USA, 2000.

[RD03] Ren, Q.; Dunham, M. H.: Semantic caching and query processing. *Transactions on Knowledge and Data Engineering*, Band 15, Nr. 1, S. 192–210, Januar 2003.

[RH80] Rosenkrantz, D. J.; Hunt III, H. B.: Processing Conjunctive Predicates and Queries. In Taylor, A. (Hrsg.): *Proceedings of the Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada*, S. 64–72. IEEE Computer Society Press, Los Altos, CA, USA, 1980.

[Riv92] Rivest, R. L.: The MD5 Message Digest Algorithm (Internet RFC 1320). Internetseite, April 1992. `http://theory.lcs.mit.edu/~rivest/Rivest-MD5.txt`.

[Rot02] Roth, J.: *Mobile Computing*. dpunkt.Verlag, Januar 2002.

[Rut01] Rutke, R.: Tamino Mobile. Vortragsfolien, November 2001. `http://www.minet.uni-jena.de/dbis/akmobil/\ gruendung/folien/rutke.pdf`.

[Sag87a] Sagiv, Y.: Optimizing datalog programs. In Vardi, M. Y. (Hrsg.): *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, March 23 - 25, 1987, San Diego, California, United States*, S. 349–362. ACM Press, New York, NY, USA, 1987. ebenfalls publiziert als [Sag87b].

[Sag87b] Sagiv, Y.: Optimizing Datalog Programs. In Minker, J. (Hrsg.): *Foundations of deductive databases and logic*, S. 659–698. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Dezember 1987.

[Sch04] Schwarzer, S.: Python – Vom einfachen Skript bis zur komplexen Anwendung strukturiert programmieren. *Linux Magazin*, Band 4, S. 60–74, 2004. Sonderheft - scripting edition.

[SER+00] Satyanarayanan, M.; Ebling, M. R.; Raiff, J.; Braam, P. J.; Harkes, J.: Coda file system user and system administrators manual. Internetseite, 2000. http://www.coda.cs.cmu.edu/doc/html/manual/book1.html.

[SH99] Saake, G.; Heuer, A.: *Datenbanken: Implementierungstechnicken*. MITP-Verlag GmbH, Bonn, 1. Auflage, 1999.

[SHA95] Secure Hash Standard. Federal Information Processing Standards Publications (FIPS) Nr. 180-1, Information Technology Laboratory of the National Institute of Standards and Technology (NIST ITL), Gaithersburg, MD, USA, April 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[Shm87] Shmueli, O.: Decidability and expressiveness aspects of logic queries. In Vardi, M. Y. (Hrsg.): *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, March 23 - 25, 1987, San Diego, California, United States*, S. 237–249. ACM Press, New York, NY, USA, 1987.

[SKN89] Sun, X.; Kamel, N.; Ni, L.: Processing implication on queries. *IEEE Transactions on Software Engineering*, Band 15, Nr. 10, S. 1168–1175, Oktober 1989.

[Sol79] Solomon, M. K.: Some properties of relational expressions. In *Proceedings of the 17th annual Southeast Regional Conference, Orlando, Florida, USA, April 09 - 11, 1979*, S. 111–116. ACM Press, New York, NY, USA, 1979.

[SS02] Saake, G.; Sattler, K.-U.: *Algorithmen & Datenstrukturen: Eine Einführung mit Java*. dpunkt.verlag GmbH, Heidelberg, 2002.

[SS03] Schulz, N.; Schmitt, I.: Relevanzgewichtung in komplexen Ähnlichkeitsanfragen. In Weikum, G.; Schöning, H.; Rahm, E. (Hrsg.): *BTW 2003 - Datenbanksysteme für Business, Technologie und Web – Tagungsband der 10. BTW-Konferenz, 26.-28. Februar 2003, Leipzig*, Lecture Notes in Informatics (LNI) - Proceedings, Band P-26, S. 187–196. Gesellschaft für Informatik e.V. (GI), Bonn, Februar 2003.

[Sus63] Sussenguth, Jr., E. H.: Use of tree structures for processing files. *Communications of the ACM*, Band 6, Nr. 5, S. 272–279, Mai 1963.

[SY80] Sagiv, Y.; Yannakakis, M.: Equivalences Among Relational Expressions with the Union and Difference Operators. *Journal of the ACM (JACM)*, Band 27, Nr. 4, S. 633–655, Oktober 1980.

[Szp90] Szpankowski, W.: Patricia tries again revisited. *Journal of the ACM (JACM)*, Band 37, Nr. 4, S. 691–711, Oktober 1990.

[TDP+94] Terry, D. B.; Demers, A. J.; Petersen, K.; Spreitzer, M. J.; Theimer, M. M.; Welch, B. B.: Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS), Austin, Texas, September 28-30, 1994*, S. 140–149. IEEE Computer Society, Los Altos, CA, USA, 1994.

[Tür03] Türker, C.: *SQL:1999 & SQL:2003*. dpunkt.verlag GmbH, Heidelberg, 1. Auflage, 2003.

[TY84] Tarjan, R. E.; Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, Band 13, Nr. 3, S. 566–579, August 1984.

[Ull82] Ullman, J. D.: *Principles of Database Systems*. Computer Science Press, Inc., Rockville, Maryland, 2. Auflage, 1982.

[Vis97] Vista, D.: *Optimizing Incremental View Maintenance Expressions in Relational Databases*. Dissertation, University of Toronto, 1997.

[vRD03a] Rossum, G. v.; Drake, Jr, F. L. (Hrsg.): *Python Library Reference Release 2.3.3*. PythonLabs, Dezember 2003. `http://www.python.org/doc/2.3.3/lib/lib.html`.

[vRD03b] Rossum, G. v.; Drake, Jr, F. L. (Hrsg.): *Python Reference Manual Release 2.3.3*. PythonLabs, Dezember 2003. `http://www.python.org/doc/2.3.3/ref/ref.html`.

[Wag03] Wagner, K. W.: *Theoretische Informatik - Eine kompakte Einführung*. Springer-Lehrbuch. Springer-Verlag, Heidelberg, 2. Auflage, 2003.

[YO79] Yu, C.; Ozsoyoglu, M.: An algorithm for tree-query membership of a distributed query. In *Proceedings of the IEEE Computer Society's Third International Computer Software and Applications Conference (COMPSAC 79), Chicago, Illinois, USA*, S. 306–312. IEEE Computer Society Press, Los Altos, CA, USA, 1979.

[Zlo75] Zloof, M. M.: Query-by-Example: the Invocation and Definition of Tables and Forms. In Kerr, D. S. (Hrsg.): *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, S. 1–24. ACM Press, New York, NY, USA, 1975.