Udo Kelter

Transaktionen

Transaktionen

Udo Kelter

2003-04

Inhaltsverzeichnis

Vorwort **10**

Lehrmodul 1: Transaktionen und die Integrität von Datenbanken **18**

1.1 Die Integrität von Datenbanken 19
1.1.1 Eine Hierarchie von Korrektheitsbegriffen für Datenbank-Zustände 19
1.1.2 Die Integritätsanforderung - 1. Ansatz 22
1.1.3 Transaktionen 23
1.1.3.1 Transaktionen in JDBC 24
1.1.3.2 Transaktionseigenschaften 24
1.1.4 Die Integritätsanforderung - 2. Ansatz 26
1.2 Maßnahmen des DBMS zur Integritätssicherung 28
1.2.1 Klassifikation der Schutzmechanismen 28
1.2.2 Semantische Integritätsprüfungen 28
1.2.3 Recovery 30
1.2.4 Concurrency Control 31
Glossar 34

Lehrmodul 2: Recovery **36**

2.1 Einführung 37
2.2 Quellen und Auswirkungen von Störungen 39
2.2.1 Systemarchitektur 40
2.2.2 Fehlerklassen 42
2.2.2.1 Medienfehler 43
2.2.2.2 Systemfehler 44
2.2.2.3 Transaktionsfehler 45

2.2.3 Ausgaben von Transaktionen 46
2.3 Anforderungen an das Recovery 48
2.3.1 Die Integrität der Datenbank aus Sicht des DBMS . . 48
2.3.2 Speicherteile der Datenbank 48
2.3.3 Qualitätsmerkmale von Recovery-Mechanismen 49
2.3.4 Zielzustand des Recoverys 50
2.4 Grundprinzipien des Recoverys 52
2.4.1 Recovery-Daten 52
2.4.2 Risikostreuung 53
2.4.3 Reparaturprinzipien 54
2.4.3.1 Arbeitsversion der Datenbank als Ausgangsbasis 54
2.4.3.2 Recovery-Daten als Ausgangsbasis 56
2.4.4 Präventionsprinzipien 56
2.5 Logging 58
2.5.1 Undo- vs. Redo-Logging 59
2.5.2 Archiv-Logs 60
2.5.3 Zustands- vs. Transitions-Logging 61
2.5.4 Logging auf verschiedenen Abstraktionsebenen 62
2.5.5 Logging auf Transaktionsebene 63
2.6 Recovery für Transaktionsfehler 63
2.7 Recovery für Systemfehler 64
2.7.1 Präventionsmaßnahmen 64
2.7.2 Globales Undo 66
2.7.3 Partielles Redo 68
2.7.4 Änderungsdateien 69
2.8 Recovery für Medienfehler 70
Glossar 71

Lehrmodul 3: Sperrverfahren **75**

3.1 Serialisierbarkeit 76
3.2 On-line-Scheduler 79
3.3 Grundlagen der Sperrverfahren 80
3.4 Protokolle 83
3.4.1 Wechselseitiger Ausschluß 85
3.4.2 Protokolle mit höherer Parallelität 87
3.4.3 Das Zwei-Phasen-Protokoll 88
3.4.3.1 Sperren bis EOT 90
3.4.3.2 Preclaiming 91

3.5 Isolationsstufen . . . 93
Glossar . . . 95

Lehrmodul 4: Sperrverfahren für Hierarchien von Sperreinheiten 97

4.1 Variable Granularität . . . 98
4.1.1 Gestaffelte Sperreinheiten . . . 99
4.1.2 Implizite Sperren . . . 101
4.2 Warnsperren . . . 102
4.3 Halbgeordnete Objektmengen . . . 106
Glossar . . . 110

Lehrmodul 5: Semantikgestützte Concurrency-Control-Verfahren 111

5.1 Einführung . . . 112
5.2 Modifikationen . . . 113
5.2.1 Definition . . . 113
5.2.2 Semantische Konfliktfreiheit von Modifikationen . . . 114
5.2.3 Sperrmodi für Modifikationen . . . 115
5.2.4 Undo von Modifikationen . . . 116
5.2.5 Atomarität von Modifikationen . . . 118
5.3 Parametrisierte Modifikationen . . . 119
5.3.1 Definition . . . 119
5.3.2 Vollständige Konfliktfreiheit . . . 120
5.4 Bereichsgrenzen . . . 121
5.4.1 Inkonsistente Zwischenzustände . . . 122
5.4.2 Unsichere Zwischenzustände und inverse Modifikationen 123
5.4.3 Überwachung von Unsicherheitsbereichen . . . 124
5.5 Konfliktfreiheit mit Parametereinschränkungen . . . 126
5.6 Konfliktfreiheit mit Objektzustandseinschränkungen . . . 128

Lehrmodul 6: Zeitstempelverfahren 131

6.1 Validierende Verfahren . . . 132
6.1.1 Verhinderung inkorrekter Verzahnungen durch Neustart 132
6.1.2 Vergleich mit Sperrverfahren . . . 133
6.1.3 Varianten validierender Verfahren . . . 135
6.2 Zeitstempel-Verfahren . . . 137
6.2.1 Die Grundform . . . 137

6.2.2 Zyklischer Neustart 140
6.2.3 Zeitstempel 141
6.2.4 Verwaltung der Zeitstempel 142
6.3 Kombinierte Sperr- und Zeitstempel-Verfahren 143

Lehrmodul 7: Optimistische Concurrency-Control-Verfahren 147

7.1 Motivation 148
7.2 Die Grundform 150
7.2.1 Die verbesserte Grundform 152
7.2.2 Verwaltung der Hilfsdaten 152
7.2.3 Eine alternative Realisierung des Validationstests . . . 154
7.2.4 Zyklischer Neustart 155
7.3 Präventive Validation 155

Lehrmodul 8: Concurrency-Control-Theorie 159

8.1 Einführung und Übersicht 160
8.2 Die Modelle 162
8.2.1 Struktur einer Datenbank mit parallelen Transaktionen 163
8.2.2 Logs 164
8.2.3 Andere Modelle für parallele Transaktionen 166
8.3 Konsistenzerhaltung 167
8.3.1 Konsistente Datenbank-Zustände 167
8.3.2 Serialisierbarkeit 170
8.3.2.1 Schwache Serialisierbarkeit 172
8.3.2.2 Strikte Serialisierbarkeit 173
8.3.3 Herbrand-Interpretationen 174
8.3.3.1 Definition 175
8.3.3.2 Eigenschaften von Herbrand-Interpretationen 176
8.3.4 Tote Transaktionen 178
8.3.4.1 Einführung 178
8.3.4.2 Verluste 180
8.3.4.3 Feststellung der Lebendigkeit 181
8.3.4.4 Tote Transaktionen in FS-äquivalenten Logs 182
8.3.4.5 Sichten von lebendigen Transaktionen 183
8.3.5 Temporäre Anomalien in serialisierbaren Logs 184
8.3.6 On-line-Scheduler 187
8.3.7 Zusammenfassung 189
8.4 Konsistente Sichten 189

8.5 Logische Atomarität 191
8.5.1 Sicht-Serialisierbarkeit 191
8.5.2 Konflikte 194
8.5.3 Serialisierungspunkte 198
8.5.4 Der Konfliktgraph eines Logs 201
8.5.5 Serialisierbare Logs ohne Verluste 207
8.5.6 2-Phasen-Sperren 208

Lehrmodul 9: Kooperation und Parallelität in SEU 210

9.1 Einleitung und Übersicht 211
9.2 Kooperationsmodelle 212
9.2.1 Kooperation und Koordination bei der Softwareentwicklung 213
9.2.2 Koordination der Bearbeitung von Dokumenten 214
9.2.3 Das “Bibliotheksmodell” 215
9.2.4 Das “Wandtafel-Modell” 216
9.2.5 Adaptierbarkeit der SEU 217
9.3 Zur Realisierung von Kooperationsmodellen nutzbare OMS-Dienste 218
9.3.1 Zugriffskontrollen 218
9.3.2 Transaktionen 219
9.3.3 Notifikation 220
9.4 Transaktionskonzepte für kooperative Umgebungen 220
9.4.1 Konventionelle Transaktionskonzepte 221
9.4.2 Arbeitseinheiten in Entwicklungsprozessen 222
9.4.3 Eine Hierarchie von Arbeitseinheiten in SEU 224
9.4.4 Langdauernde Arbeitseinheiten 226
9.4.5 Prozeßinterne Arbeitseinheiten und Kommandoprozeduren 229
9.5 Zusammenfassung 231
Glossar 232

Literatur 234

Index 236

Vorwort

Transaktionen sind eines der wichtigsten Konzepte der Informatik. Ohne das Konzept der Transaktion und die damit zusammenhängenden Technologien wäre der heutige massenhafte Einsatz von Datenbanken in allen Bereichen des Geschäftslebens nicht denkbar.

Transaktionen wurden ursprünglich im Kontext von Datenbanken entwickelt, es zeigte sich allerdings sehr bald, daß die grundlegenden Konzepte auch in anderen Systemen anwendbar sind, in denen parallele Prozesse auf gemeinsamen, u.U. sogar verteilten Datenbeständen arbeiten. Systeme in diesem Sinne sind Betriebssysteme, verteilte Systeme, manche parallele Algorithmen sowie diverse nichtkonventionelle Datenverwaltungssysteme.

Trotz ihrer Bedeutung sind Transaktionen – speziell im hier interessierenden technischen Sinne – wenig präsent, in der Allgemeinheit sowieso nicht[1], aber selbst ausgebildete Informatiker kennen diesen Begriff oft nur oberflächlich, und zwar sogar solche Entwickler, die Transaktionen bei der Realisierung von Informationssystemen praktisch einsetzen.

Letzteres ist eigentlich positiv zu sehen: Obwohl die Technologien zur Realisierung von Transaktionen sehr komplex sind, kann zumindest in erster Näherung die Wirkung von Transaktionen sehr einfach beschrieben werden. Es gibt auch nur sehr wenige Programmschnittstellen, über die ein Applikationsprogrammierer die Transaktionsverarbeitung

[1] Umgangssprachlich wird unter einer Transaktion ein Geschäft bzw. ein Geschäftsabschluß verstanden; Wortbildungen wie TAN (Transaktionsnummer) bei Banküberweisungen beziehen sich auf den so verstandenen Begriff.

 ©2003 Udo Kelter

beeinflussen kann oder muß. Die hier vorliegende enorme Abstraktionsleistung ist sicher einer der Gründe für die große Verbreitung des Konzepts.

Übersicht über dieses Buch. Einführende und oberflächliche Kenntnisse über Transaktionen reichen daher in vielen Fällen völlig aus. Den entsprechenden Kenntnisstand versucht das Lehrmodul "Transaktionen und die Integrität von Datenbanken" zu vermitteln; es motiviert und definiert die grundgelegenden Begriffe und gibt eine Übersicht über die wesentlichen Funktionsbereiche.

In vielen Fällen ist indes eine genauere Kenntnis der Transaktionskonzepte und der grundlegenden Techniken, wie Transaktionen realisiert werden, erforderlich:

1. Bei der Entwicklung von Informationssystemen, die eine größere Anzahl echt paralleler Nutzungen unterstützen müssen.

   Erforderlich ist hier vor allem eine genauere Kenntnis der Sperrprotokolle. Ohne diese Kenntnisse werden leicht Entwurfsfehler gemacht, die sich in "unerklärlich" schlechten Antwortzeiten im Betrieb des Systems äußern, d.h. obwohl die Hardware großzügig ausgelegt und offenbar schlecht ausgelastet ist, kommt es gelegentlich oder regelmäßig zu langen Wartezeiten oder schlechtem Durchsatz. Ursache des Problems sind unerkannte Sperrungen von Datenobjekten, die zu Wartebeziehungen zwischen Prozessen führen. Der hier erforderliche Kenntnisstand wird durch durch zwei Lehrmodule vermittelt:

   Das *Lehrmodul "Sperrverfahren"* stellt die Grundprinzipien von Sperrungen und die grundlegenden Sperrprotokolle dar; mit diesen Grundkenntnissen sollte verstanden werden, wann und wie Wartesituationen auftreten. Ferner werden die Isolationsstufen in SQL vorgestellt, mit deren Hilfe gezielt Kompromisse zugunsten der Performance eines Systems und, sofern tolerabel, zu Lasten der Konsistenz eingegangen werden können.

   Das *Lehrmodul "Sperrverfahren für Hierarchien von Sperreinheiten"* behandelt aufbauend hierauf das Problem, parallele Transaktio-

nen, die mit sehr unterschiedlich großen Datenmengen arbeiten – z.B. einzelne Objekte vs. ganze Relationen –, effizient zu unterstützen. Die technische Lösung des Problems sind sogenannte Warnsperren bzw. zugehörige Protokolle. Hierdurch können kleine, mittlere und sehr große Sperrgranulate gleichzeitig gesperrt werden. Ein mittleres, aus Anwendersicht unsichtbares Sperrgranulat ist z.B. die Menge der Tupel, die auf einer Speicherseite stehen.

Infolge der Warnsperren ist es effizienter, z.B. eine Relation komplett zu sperren, anstatt sehr viele Tupel dieser Relation einzeln zu sperren; man spricht hier von der Eskalation von Sperrungen. Dies kann zu besonders "unerklärlichen" Wartesituationen bzw. Performance-Problemen führen, weil hier intern Sperrungen eintreten, die vom Benutzer bzw. Programmierer einer Transaktion nicht angefordert wurden bzw. erkennbar sind.

2. Beim Installieren und Administrieren von Datenbanken: Für die Recovery-Funktionen müssen Hilfsdaten angelegt werden. Diese Hilfsdaten müssen administriert werden; im Extremfall eines Plattendefekts muß der Administrator mit diesen Hilfsdaten die Datenbank rekonstruieren. Je nach DBMS-Produkt kann können auch Umfang der Hilfsdaten, Schutzmaßnahmen für die Hilfsdaten und weitere Details konfiguriert werden. In allen Fällen ist ein grundlegendes Verständnis der Recovery-Funktionen erforderlich.

   Das *Lehrmodul "Recovery"* vermittelt ein an diesem Kenntnisstand orientiertes Grundwissen.

   Auch für beliebige andere Nutzer einer Datenbank ist dieses Verständnis nützlich und sollte zu der Erkenntnis führen, daß es keine perfekte Sicherheit gibt, daß größere Schäden an einer Datenbank nicht in fünf Minuten repariert sein können und daß der Umfang von Schäden durch geeignete Planungen und Vorsichtsmaßnahmen durchaus verkleinert werden kann.

Die weiteren in diesem Buch zusammengestellten Lehrmodule betreffen speziellere Themen und damit auch kleinere Interessentenkreise:

Lehrmodul "Semantikgestützte Concurrency-Control-Verfahren": Die grundlegenden Sperrverfahren unterscheiden nur lesende und schreibende Zugriffe zu Daten, die konkrete Semantik von Operationen auf den Daten bleibt außer Betracht. Das Lehrmodul "Semantikgestützte Concurrency-Control-Verfahren" stellt Erweiterungen der Sperrverfahren vor, durch die semantische Eigenschaften der Operationen – i.w. Kommutativität von Operationen – auf den Daten ausgenutzt werden sollen. Es zeigt sich allerdings, daß die Verfahren wegen Wechselwirkungen mit dem Recovery und wegen "kleinen Ausnahmen" von der Kommutativität von Operationen unerwartet kompliziert werden und ein echter Nutzen nur selten erreicht werden kann.

Lehrmodul "Zeitstempelverfahren": Wenn man Sperren einsetzt, können abgesehen von seltenen Ausnahmefällen auch Deadlocks auftreten. Die Erkennung und Auflösung von Deadlocks ist schon in zentralen Systemen sehr lästig und kostet Performance; in verteilten Systemen kommen Verzögerungen und Kosten für Kommunikationen hinzu, denn Deadlocks können sich über mehrere Rechner erstrecken. Dies motiviert die Suche nach deadlockfreien Concurrency-Control-Verfahren. Eine Klasse solcher Verfahren sind Zeitstempelverfahren. Zeitstempelverfahren basieren auf dem Prinzip, unzulässige Verzahnungen von Transaktionen zu verhindern, indem eine der beteiligten Transaktionen vom DBMS zurückgesetzt und neu gestartet wird. Zeitstempelverfahren sind daher nur dann anwendbar, wenn derartige Zurücksetzungen praktikabel sind.

Lehrmodul "Optimistische Concurrency-Control-Verfahren": Optimistische Concurrency-Control-Verfahren basieren wie Zeitstempelverfahren auf dem Prinzip, unzulässige Verzahnungen von Transaktionen durch Rücksetzungen zu verhindern, sind aber stärker an reinen Lesetransaktionen orientiert.

Lehrmodul "Concurrency-Control-Theorie": Alle "korrekten" Concurrency-Control-Verfahren garantieren, daß bei den entstehenden verzahnten Transaktionsausführungen keine "Anomalien" (Verlust

von Änderungen, Inkonsistenzen usw.) auftreten. Es ist ziemlich schwierig, aus der intuitiv eingängigen Forderung nach Abwesenheit von Anomalien einen präzise definierten Korrektheitsbegriff für verzahnte Transaktionsausführungen abzuleiten. In der Tat kristallisierte sich seinerzeit erst nach jahrelanger Forschung, die zu einer eigenen Concurrency-Control-Theorie führte, der Begriff Serialisierbarkeit als der gesuchte heraus. Es gibt mehrere Varianten dieses Begriffs, die theoretischen Analysen zeigen, daß in der Praxis mindestens die sogenannte cp- (*conflict preserving*) Serialisierbarkeit gefordert werden muß. Das Lehrmodul "Concurrency-Control-Theorie" präsentiert die wesentlichen Elemente dieser Theorie.

Lehrmodul "Kooperation und Parallelität in SEU": Transaktionen wurden ursprünglich im Kontext "konventioneller" betrieblicher Anwendungen und dafür gedachter DBMS entwickelt. Seit langem werden auch DBMS für nichtkonventionelle Anwendungen entwickelt, hierzu gehören insb. auch CAD-, CASE- und ähnliche Entwurfsumgebungen. An die Randbedingungen dieser Anwendungsgebiete müssen auch die Transaktionskonzepte angepaßt werden.

Ein Ergebnis in dieser Richtung sind sogenannte **Entwurfstransaktionen**. Die Grundidee, daß eine Transaktion bestimmte Arbeitsvorgänge kapselt, also serialisierbar ausführt und ganz oder gar nicht wirksam werden läßt, wird hier auf sehr umfangreiche Arbeitsvorgänge übertragen, nämlich tage- oder wochenlang dauernde Entwicklungsvorgänge in Entwicklerteams. Dieser Ansatz wirkt zunächst elegant, es zeigt sich allerdings, daß die Randbedingungen so stark verschieden sind, daß Entwurfstransaktionen und konventionelle Transaktionen weder konzeptionell noch implementierungstechnologisch viele Gemeinsamkeiten aufweisen.

Das Lehrmodul "Kooperation und Parallelität in SEU" analysiert zunächst parallele Arbeitsvorgänge in Entwicklerteams (am Beispiel von Software-Entwicklern). Während die Nutzer von klassischen Informationssystemen einander nicht kennen, kooperieren die Entwickler in einem Team mehr oder minder intensiv. Je nach den Umständen müssen verschiedene Kooperationsmodelle unterstützt werden, z.B.

das Bibliotheksmodell und Wandtafel-Modell. Anschließend wird analysiert, wie diese Modelle auf Basis von Transaktionsfunktionen und anderen Leistungen eines zugrundeliegende Objektmanagementsystems realisiert werden können. Es zeigt sich u.a., daß Entwurfstransaktionen nur eine Bündelung ohnehin erforderlicher Funktionen, nämlich Versionierung und gruppenorientierte Zugriffskontrollen, darstellen.

Zur Struktur dieses Buchs. Dieses Buch besteht im Gegensatz zu anderen Bücher nicht aus Kapiteln, sondern aus Lehrmodulen. Jedes Lehrmodul ist auch als selbständiger Text auf der Webseite `http://www.kltr.de` erreichbar.

Im Unterschied zu den Kapiteln eines Buchs sind Lehrmodule möglichst autark gestaltet. Durchgehende Beispiele oder häufige Verweise auf andere Kapitel, auf die man in Büchern besonders stolz ist, die aber Abhängigkeiten erzeugen und das separate Lesen einzelner Kapitel erschweren, werden hier ganz bewußt vermieden. Wenn ohne allzuviel Redundanz möglich, wird statt eines Verweises auf andere Stellen der Lerngegenstand, auf den Bezug genommen wird, direkt dargestellt, ggf. in skizzenhafter Form.

Während ein Buch fast immer sequentiell gelesen wird, ist die Lesereihenfolge der Lehrmodule wesentlich freier. Der Graph in Bild 1 zeigt die relativ wenigen noch vorhandenen Reihenfolgerestriktionen. Ein gepunkteter Pfeil stellt nur eine Empfehlung dar. Die Abkürzungen bedeuten folgendes:

TID	Transaktionen und die Integrität von Datenbanken
REC	Recovery
SPV	Sperrverfahren
SHS	Sperrverfahren für Hierarchien von Sperreinheiten
SCC	Semantikgestützte Concurrency-Control-Verfahren
ZSV	Zeitstempelverfahren
OCC	Optimistische Concurrency-Control-Verfahren
CCT	Concurrency-Control-Theorie
KPS	Kooperation und Parallelität in SEU

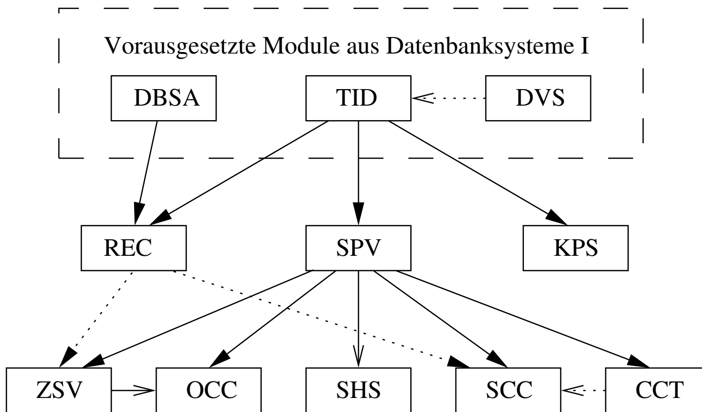


Abbildung 1: Lesereihenfolge der Lehrmodule

Einordnung und Vorkenntnisse. Transaktionen sind eher ein Thema für Spezialvorlesungen. Typischerweise wird man sich erst dann mit Transaktionen intensiver beschäftigen, wenn Grundkenntnisse über Datenbanksysteme vorhanden sind, wie z.B. in der Vorlesung Datenbanksysteme I vermittelt. Es zeigt sich allerdings, daß man erhebliche Anteile des Stoffs der Datenbanksysteme I (oder anderer Lehrbücher über Datenbanksysteme), insb. viele Details der Abfragesprachen und des Schemaentwurfs, nicht benötigt; relevant sind neben den Grundbegriffen vor allem Kenntnisse über die Architektur von DBMS[2].

Die in diesem Buch enthaltenen Lehrmodule behandeln das Thema Transaktionen keineswegs vollständig. Es gibt im Gegenteil eine ganze Reihe von Themen, die in diversen praktischen Problemstellungen auftreten und hier nicht abgedeckt sind:

- Transaktionsmonitore[3]

[2]Das Lehrmodul "Transaktionen und die Integrität von Datenbanken" ist auch im Buch Datenbanksysteme I enthalten; da es viele zentrale Begriffe einführt, erscheint es hier erneut, um dieses Buch selbständiger lesbar zu machen.

[3]Bei der Programmierung einer Transaktion im Sinne einer Funktion eines Informationssystems müssen viele technische Dinge erledigt werden, z.B. müssen Da-

- Transaktionen in verteilten Datenbanken
- Transaktionen in föderativen Systemen
- geschachtelte Transaktionen

Vortragsfolien. Für die einige Lehrmodule sind Vortragsfolien in Form von PostScript-Dateien auf Anfrage für Lehrende frei verfügbar. Die Folien sind derzeit noch nicht über die WWW-Seiten abrufbar.

Kommentare und Fehlermeldungen. Kein langer Text ist fehlerfrei, so auch dieser. Daher sind Hinweise auf Fehler und Verbesserungsmöglichkeiten sowie Kommentare aller Art sehr willkommen (am besten elektronisch an `uk@kltr.de`).

tenbankinhalte in Puffervariablen kopiert, zugehörige transiente Objekte verwaltet und ggf. Sitzungen für längere Dialoge verwaltet werden. Ein Transaktionsmonitor (oder Applikationsserver) ist eine Infrastruktur, die derartige Aufgaben übernimmt. Der Begriff Transaktion ist hier also wesentlich anwendungsnäher verstanden als bei Datenbank-Transaktionen. Insofern ordnen wir das Thema Transaktionsmonitore nicht im Gebiet Datenbanken, sondern im Gebiet Softwaretechnik, dort im Teilgebiet Standardarchitekturen, ein.

Lehrmodul 1:

Transaktionen und die Integrität von Datenbanken

Zusammenfassung dieses Lehrmoduls

Dieses Lehrmodul führt in die Problematik der Erhaltung der Integrität einer Datenbank ein. Es werden drei Arten von Gefährdungen der Integrität unterschieden: Fehler bei Dateneingaben oder in Anwendungsprogrammen, Betriebsstörungen wie z.B. Systemabstürze oder Mediendefekte und paralleler Zugriff mehrerer Anwendungen auf die gleichen Daten. Für alle drei Arten von Gefährdungen werden Gegenmaßnahmen im DBMS beschrieben; alle beruhen auf dem zentralen Konzept der Transaktion. Einleitend wird analysiert, welche Arten von Korrektheit unterschieden und sinnvollerweise gefordert werden können.

Vorausgesetzte Lehrmodule:

empfohlen: – Datenverwaltungssysteme

Stoffumfang in Vorlesungsdoppelstunden: 1.0

 ©2003 Udo Kelter

1.1 Die Integrität von Datenbanken

Die Korrektheit bzw. Integrität der Daten in einer Datenbank genießt sehr hohe Priorität; dementsprechend enthalten DBMS vielfältige Mechanismen, deren Zweck darin besteht, die Integrität der Daten sicherzustellen. Man unterscheidet drei Hauptklassen von Gefährdungen der Integrität:

1. Fehler bei Dateneingaben oder in Anwendungsprogrammen
2. Betriebsstörungen wie z.B. Systemabstürze oder Mediendefekte
3. paralleler Zugriff mehrerer Anwendungen auf die gleichen Daten

Die erste Klasse von Gefährdungen kann nur dadurch bekämpft werden, daß die korrekten Inhalte einer Datenbank dem DBMS möglichst genau beschrieben werden, so daß das DBMS Veränderungen, die zu einem inkorrekten Zustand führen, ablehnen kann. Entsprechende Konzepte können nur in engem Zusammenhang mit dem Datenbankmodell diskutiert werden.

Die zweite und dritte Klasse von Gefährdungen, auf die wir uns in diesem Lehrmodul konzentrieren werden, sind dagegen weitgehend unabhängig vom Datenbankmodell.

Bevor wir die Gefährdungen und die Maßnahmen dagegen diskutieren können, muß zunächst analysiert werden, was genau unter Korrektheit zu verstehen ist. Es zeigt sich, daß intuitiv naheliegende Interpretationen dieses Begriffs nicht brauchbar sind.

1.1.1 Eine Hierarchie von Korrektheitsbegriffen für Datenbank-Zustände

Integrität. Unter **Integrität** (Adjektiv: **integer**) sei verstanden, daß der Inhalt einer Datenbank einen interessierenden Teil bzw. eine Abstraktion der Realität genau (exakt), richtig und aktuell gültig wiedergibt. Beispiele für unerwünschte bzw. falsche Datenbankinhalte sind:

- unmögliche Werte (Kinderzahl = -3),

- Inkonsistenzen (z.B. eine Person ist verheiratet, hat aber Steuerklasse I)
- veraltete Angaben.

Es gibt somit verschiedene Aspekte der Korrektheit einer Datenbank. Unter Integrität wird diese Korrektheit in ihrem weitesten, umfassendsten Sinn verstanden. Oft wird diese generelle Korrektheit auch mit Konsistenz oder einfach nur mit Korrektheit bezeichnet.

Wir befassen uns hier in diesem Lehrmodul nur mit Maßnahmen, die ein DBMS durchführen kann, um die Korrektheit der Daten sicherzustellen. Nicht garantieren kann das DBMS die Aktualität der Daten, insoweit diese davon abhängt, daß rechtzeitig Daten eingegeben oder Änderungsprogramme gestartet werden.

Nicht befassen werden wir uns in diesem einführenden Lehrmodul auch mit Problemen, die spezifisch für einzelne Datenbankmodelle sind. Viele Probleme bei der Integritätssicherung sind in der Tat unabhängig vom Datenbankmodell. Wir werden daher i.f. vereinfachend annehmen, daß eine Datenbank eine Menge von **Datenbank-Objekten** enthält, die einen Zustand haben, und daß die Datenbank durch die Operationen des Datenbankmodells, die wir hier auch als **Aktionen** bezeichnen werden, verändert wird.

Statt vom Inhalt einer Datenbank, was vielleicht anschaulicher wäre, werden wir meist vom **Zustand der Datenbank** sprechen.

Erreichbarkeit. Unser Ziel ist die Beschreibung von integren Zuständen, also “völlig korrekten” Zuständen. Diesem Ziel nähern wir uns in mehreren Schritten. Zunächst können wir uns auf erreichbare Zustände beschränken. Ein Zustand heißt **erreichbar**, wenn er nach Initialisierung der Datenbank durch eine endliche Folge von Aktionen (mit zulässigen bzw. syntaktisch korrekten Parametern) konstruiert werden kann.

Logische Konsistenz. Ein erreichbarer Zustand ist nicht sinnvoll, wenn er etwas Unmögliches darstellt, also unrealistisch ist. Ein erreich-

barer Zustand heißt **logisch konsistent**, wenn eine Realität möglich ist, die durch ihn korrekt wiedergegeben wird.

Nicht jeder erreichbare Datenbank-Zustand ist realistisch. Beispielsweise können Werte einzelner Datenbank-Objekte schon für sich allein unrealistisch sein (Kinderzahl = -3), oder es kann bei redundanten Angaben in mehreren Objekten ein Widerspruch enthalten sein.

Unter Konsistenz versteht man oft einen speziellen Aspekt der logischen Konsistenz, wie sie oben definiert wurde, nämlich die gegenseitige Konsistenz zwischen mehreren, insgesamt redundanten Einzelangaben und ggf. Eigenschaften der Objektmenge. Beispiele für einzuhaltende Konsistenzbedingungen sind:

- die Summe aller Konten ist gleich 0,
- der Inhalt eines Felds “Zahl der Angestellten” ist gleich der Zahl der Angestelltensätze oder -tupel.

Wir werden hier jedoch unter **Konsistenz** stets die oben definierte allgemeinere Form von logischer Konsistenz verstehen.

Die vorstehenden Konsistenzbedingungen können i.a. nicht durch die elementaren Schemadefinitionskonzepte ausgedrückt werden; es kann also erreichbare, aber nicht logisch konsistente Zustände geben.

Physische Konsistenz. Mit **physischer Konsistenz** bezeichnet man den ordnungsgemäßen Zustand der internen Speicherungsstrukturen der Datenbank, vor allem der Zugriffspfade. Diese sind auf der Ebene von Datenbank-Objekten nicht sichtbar bzw. zugreifbar, sondern nur innerhalb des Laufzeitkerns des DBMS, in dem die Aktionen realisiert werden. Wir gehen davon aus, daß alle Aktionen die physische Konsistenz erhalten. Jeder erreichbare Zustand ist also physisch konsistent. Physisch konsistente, aber nicht erreichbare Zustände können z.B als Zwischenzustände innerhalb von Aktionsausführungen auftreten.

Damit verfügen wir bereits über eine kleine Hierarchie von Arten der Korrektheit eines Datenbank-Zustandes:

- Integrität
- logische Konsistenz

- Erreichbarkeit
- physische Konsistenz

Da wir das richtige Funktionieren der Aktionen voraussetzen, stellt die physische Konsistenz für uns kein Problem dar; wir werden uns nur mit erreichbaren Zuständen befassen.

Die logische Konsistenz ist eine notwendige, aber keine hinreichende Voraussetzung für die Integrität. Die logische Konsistenz der Datenbank ist eine minimale Form der Korrektheit, die fast immer unverzichtbar ist. Sie besagt noch nicht, daß der Zustand der Datenbank auch die Realität aktuell korrekt wiedergibt.

Auf die völlige Integrität der Datenbank kann hingegen bei manchen Anwendungen verzichtet werden; oft werden Abstriche an der Aktualität der Daten in Kauf genommen, um den Aufwand zu reduzieren. In diesem Fall liegt eine Zwischenform von Korrektheit vor. Wir werden hier solche anwendungsbezogenen Zwischenformen nicht betrachten.

1.1.2 Die Integritätsanforderung - 1. Ansatz

Da das Hauptziel die Sicherstellung der Korrektheit der Datenbank ist, könnte man auf die Idee kommen, folgende Korrektheitsanforderung aufzustellen: Die Datenbank soll *ständig* korrekt (integer oder konsistent) sein. Diese Anforderung ist jedoch nicht sinnvoll, denn im Verlauf mancher Veränderungen der Datenbank müssen zwangsläufig logisch inkonsistente Zwischenzustände auftreten. Das Standardbeispiel hierfür ist eine Umbuchung von X Euro von Konto A auf Konto B, die wir uns vereinfacht in 4 Zugriffen auf die Datenbank vorstellen:

Zugriff 1: alten Stand von Konto A lesen, X Euro abziehen
Zugriff 2: neuen Stand von Konto A schreiben
Zugriff 3: alten Stand von Konto B lesen, X Euro addieren
Zugriff 4: neuen Stand von Konto B schreiben

In unserem Beispiel könnte das Konsistenzkriterium gelten, daß die Summe aller Kontostände 0 sein muß. Nach dem zweiten obigen Zugriff ist dieses Kriterium offenbar verletzt. Erst nach dem vierten Schritt

ist die Konsistenz wiederhergestellt, die Verletzung war also nur *temporär*. Diese Beobachtung ist extrem wichtig und kann dahingehend verallgemeinert werden, daß bei allen komplexen Konsistenzbedingungen temporäre Verletzungen *unvermeidlich* sind, da die Datenbank i.a. nicht durch eine einzige Datenmanipulationsoperation vom bisherigen Zustand in einen neuen konsistenten Zustand überführt werden kann. Letzteres ist nur durch *mehrere* Zugriffe möglich.

Die obige Anforderung muß daher dahingehend modifiziert werden, daß die Datenbank immer nur dann korrekt sein soll, wenn sie **ruht**, d.h. wenn gerade keine komplexen Änderungen mehr stattfinden.

Da man einer Aktion nicht ansieht, ob sie die letzte in einer zusammengehörigen Folge ist, müssen Anfang und Ende von Änderungen explizit gekennzeichnet werden. Andernfalls wäre nie entscheidbar, ob die Datenbank gerade ruht oder nicht.

1.1.3 Transaktionen

Das Konzept zur Lösung dieses und anderer Probleme sind Transaktionen. Eine **Transaktion** ist eine vom Benutzer definierte Folge von Aktionen[4], sozusagen eine “elementare Arbeitseinheit”.

Transaktionen können nur von Applikationsprogrammen aus ausgenutzt werden (interaktive Schnittstellen zu Datenbanken sind hier selbst als Applikationen aufzufassen). Die Frage ist nun, wie eine Applikation, hier im Sinne eines laufenden Prozesses verstanden, eine Folge von Aktionen definieren kann, die eine Transaktion bilden soll. Im Detail hängt dies von der Gastsprache und dem Betriebssystem ab; wir schildern i.f. als Beispiel die Vorgehensweise bei JDBC.

---

[4] Üblicherweise wird unter einer Transaktion sowohl eine Folge ausgeführter Aktionen verstanden als auch ein Programm bzw. Programmstück, welches eine einzige bzw. eine Menge solcher Folgen definiert. Das ist zwar terminologisch etwas unsauber, aber bequem. Um nicht zu sehr vom üblichen Sprachgebrauch abzuweichen, werden wir hier ebenso verfahren. Aus dem Kontext ergibt sich stets, ob ein Programm (oder Programmteil) oder eine Ausführung gemeint ist.

1.1.3.1 Transaktionen in JDBC

Zunächst muß ein Applikationsprozeß, der auf einer (SQL-) Datenbank arbeiten will, eine Verbindung zum Datenbankserver aufbauen. Details übergehen wir hier, die Verbindung wird letztlich durch ein Objekt des Typs `Connection` repräsentiert und durch die Operation `getConnection` initialisiert, z.B.

```
Connection verbindung = DriverManager.getConnection (...)
```

Ein explizites Kommando zum Beginnen einer Transaktion ist nicht vorhanden; nach Aufbau der Verbindung und nach der Beendigung einer Transaktion wird immer implizit eine neue Transaktion begonnen.

Das (erfolgreiche) Ende einer Transaktion wird dem DBMS durch die Operation `commit` bekanntgegeben, z.B. durch folgenden Aufruf:

```
verbindung.commit();
```

Nach Aufbau der Verbindung befinden wir uns allerdings zunächst in einem Arbeitsmodus, bei dem sozusagen implizit nach jeder Aktion ein `commit` ausgeführt wird. Mit anderen Worten wird *jedes einzelne* SQL-Kommando als Transaktion behandelt, d.h. es werden nicht wirklich Gruppen von Aktionen gebildet. Das automatische `commit` kann man abschalten mit

```
verbindung.setAutoCommit(false);
```

1.1.3.2 Transaktionseigenschaften

Bei der Ausführung einer Transaktion werden die folgenden wichtigen Eigenschaften garantiert:

1. **Konsistenzerhaltung**: Eine Transaktion, die in einem logisch konsistenten Zustand startet, hinterläßt die Datenbank am Ende wieder in einem logisch konsistenten Zustand. Verantwortlich hierfür ist in erster Linie der Benutzer, der die Transaktion programmiert oder Daten eingibt.

2. **(Fehler-) Atomarität**: eine Transaktion wird entweder *ganz oder gar nicht* wirksam. Wenn also eine Transaktion, gleichgültig aus welchen Gründen, fehlschlägt und abgebrochen wird, verändert sie die Datenbank nicht. Dies bedeutet, daß alle bisher von ihr verursachten Änderungen rückgängig gemacht werden. Verantwortlich für die Atomarität ist das DBMS.
3. **Dauerhaftigkeit**: Sobald dem Benutzer die erfolgreiche Ausführung einer Transaktion gemeldet wurde, müssen ihre Wirkungen erhalten bleiben und dürfen nicht infolge von Störungen oder Beschädigungen der Datenbank, die der Benutzer nicht zu verantworten hat und von denen er ggf. nichts erfährt, verloren gehen. Verantwortlich hierfür ist das DBMS.
4. **Serialisierbarkeit**: Bei parallel ausgeführten Transaktionen entspricht der Gesamteffekt mehrerer überlappend ausgeführter Transaktionen dem Effekt, der bei einer denkbaren seriellen Ausführung der Transaktionen erreicht worden wäre. Jede Transaktion hat also den Eindruck, daß sie *isoliert* auf der Datenbank arbeiten würde. Verantwortlich hierfür ist das DBMS.
5. **Endliche Ausführungszeit**: Die Ausführung einer Transaktion darf nicht durch Synchronisationsvorgänge im DBMS immer wieder hinausgeschoben werden. Der Benutzer muß in endlicher Zeit nach dem Start einer Transaktion eine Rückmeldung über die erfolgreiche Ausführung oder den Abbruch erhalten. Verantwortlich hierfür ist das DBMS.

   Trivialerweise setzen wir hier voraus, daß jede Transaktion nur aus endlich vielen Aktionen besteht und daß das Transaktionsprogramm keine Endlosschleifen enthält (wofür der Benutzer die Verantwortung trägt).

In vielen älteren Quellen wird statt der Serialisierbarkeit als eine Transaktionseigenschaft die Isolation gefordert. Unter **Isolation** versteht man, daß alle Effekte innerhalb einer Transaktion, also vor ihrer Beendigung, für parallele Transaktionen unsichtbar sind. Diese Anforderung ist naheliegend, denn, wie oben gezeigt, können logisch inkonsistente Zwischenzustände auftreten. Bei "normalen" Transak-

tionen (insb. solchen, die keine Datenwerte blind überschreiben, ohne sie vorher zu lesen) impliziert die Isolation die Serialisierbarkeit, die Umkehrung gilt nicht.

In der Literatur werden die Transaktionseigenschaften oft mit der Abkürzung *ACID* für *atomicity, consistency preservation, isolation, durability* zusammengefaßt.

Transaktionen sind das Schlüsselkonzept schlechthin in der Architektur moderner (und der meisten älteren) DBMS; es ist nahezu obligatorisch. Änderungen und Abfragen in Datenbanken werden daher i.d.R. durch Transaktionen mit den oben erwähnten Eigenschaften durchgeführt[5]. Aus der Sicht des Benutzers ist ein DBMS somit ein Transaktionsverarbeitungssystem.

Obwohl das Stichwort Parallelität bereits mehrfach gefallen ist, werden wir im Rest dieses Lehrmoduls annehmen, daß Transaktionen *sequentiell* ausgeführt werden. Mit parallel ausgeführten Transaktionen befassen sich eigene Lehrmodule.

1.1.4 Die Integritätsanforderung - 2. Ansatz

Kommen wir nun auf die oben gestellte Anforderung zurück, daß der Datenbank-Zustand zumindest dann logisch konsistent sein soll, wenn die Datenbank ruht. Durch die Einführung des Transaktionskonzepts und die Eigenschaft "Konsistenzerhaltung" ist diese Anforderung offenbar bereits erfüllt. Wir setzen natürlich voraus, daß die Datenbank nach ihrer Initialisierung logisch konsistent ist.

Das Integritätsproblem wird durch die Transaktionen nicht gelöst. Ein logisch konsistenter Zustand beschreibt irgendeine denkbare Realität, der integre nur die aktuell vorhandene.

Die in der Datenbank darzustellende Realität ändert sich im Laufe der Zeit; jedenfalls nehmen wir dies an. Die Integrität der Datenbank

[5]Die im Prinzip angenehmen Transaktionseigenschaften haben ihren Preis, nämlich eine verschlechterte Performance des Systems. In performancekritischen Systemen muß daher bisweilen auf Transaktionseigenschaften verzichtet werden. In SQL kann man durch "Isolationsstufen" schrittweise Transaktionseigenschaften zugunsten besserer Performance opfern (s. [**?**])

kann über die Zeit hinweg nur dann erreicht werden, wenn nach jeder Änderung der Realität rechtzeitig der Datenbank-Inhalt entsprechend korrigiert wird. Rechtzeitig bedeutet: vor erneuter Benutzung der korrekturbedürftigen Daten. Nach den obigen Festlegungen sind Korrekturen in Form von Transaktionen durchzuführen. Zur Sicherstellung der Integrität muß also für folgendes gesorgt werden:

A) Korrigierende Transaktionen sind mit den richtigen Argumenten rechtzeitig zu starten.

B) Die Transaktionen müssen geeignet programmiert sein, um den Datenbank-Zustand richtig zu korrigieren, so daß die Änderung der Realität in der Datenbank nachvollzogen wird.

C) Die Transaktionen müssen korrekt ausgeführt werden (vgl. die obigen, vom DBMS zu garantierenden Ausführungseigenschaften).

Für A und B kann das DBMS wenig oder keine Verantwortung tragen: Ursache ist, daß es i.a. keine direkten "Wahrnehmungsorgane" für die Realität hat, die die Datenbank darstellen soll. Statt dessen nimmt es die Realität durch die "Brille" der Änderungen bzw. Transaktionen wahr, die von Benutzern oder automatischen Datenerfassungsgeräten veranlaßt werden. Das DBMS kann die Exaktheit von Angaben nicht prüfen, es kann auch die Rechtzeitigkeit von Änderungen nicht erzwingen. In den Bereichen A und B sind vorrangig die Benutzer bzw. die Systemumgebung verantwortlich. Das DBMS kann allenfalls gewisse grobe Fehler erkennen und Transaktionen abbrechen.

Der Bereich C steht hingegen voll in der Verantwortung des DBMS. Es muß dafür sorgen, daß der Zustand der Datenbank genau der ist, der sich gemäß den Transaktionen ergibt, deren Ausführung den Benutzern bestätigt worden ist. Diesen Zustand nennen wir **technisch korrekt**.

In diesem und den aufbauenden Lehrmodulen werden wir das Problem der Sicherstellung der Integrität auf das Problem der Sicherstellung der technischen Korrektheit reduzieren, denn es werden nur Maßnahmen behandelt, die *das DBMS* zur Sicherstellung der Integrität ergreifen kann.

1.2 Maßnahmen des DBMS zur Integritätssicherung

1.2.1 Klassifikation der Schutzmechanismen

Die Integrität bzw. Korrektheit der Datenbank wird durch viele verschiedene Ursachen gefährdet. Die Aufgabe an ein DBMS besteht also darin, durch geeignete Schutzmechanismen den Verlust der Integrität zu verhindern. Die Schutzmechanismen können in drei Kategorien eingeteilt werden:

1. **semantische Integritätsprüfungen**: Verhinderung von Änderungen, die zu einem inkorrekten Datenbank-Zustand führen; Ursache können z.B. versehentlich oder absichtlich falsch eingegebene Daten oder fehlerhafte Applikationsprogramme sein.
2. **Recovery**: Wiederherstellung der Datenbank nach einer Beschädigung infolge einer Störung; Ursachen können Fehler in der Hardware oder Software oder auch eine Fehlbedienung sein.
3. **Concurrency Control**: Verhinderung von Interferenzen zwischen parallelen Transaktionen oder Behebung unzulässiger Effekte eingetretener Interferenz.

Transaktionen bilden die konzeptionelle Grundlage bei allen oben erwähnten Arten von Schutzmaßnahmen, dies unterstreicht ihre Wichtigkeit.

Die einzelnen Arten von Maßnahmen decken im Prinzip unabhängige Problemstellungen ab, sie weisen aber, wie man bei näherer Analyse feststellt, signifikante Querbezüge untereinander auf. So wird z.B. für bestimmte Arten von semantischen Integritätsprüfungen auf Recovery-Mechanismen zurückgegriffen. Die folgenden Kurzdarstellungen der Einzelgebiete sollen u.a. diese Querbezüge verdeutlichen.

1.2.2 Semantische Integritätsprüfungen

Wir hatten schon oben erwähnt, daß im Prinzip der Programmierer von Transaktionen, die späteren Nutzer des Informationssystems und ggf.

weitere Personen dafür verantwortlich sind, daß Änderungen rechtzeitig durchgeführt, richtige Daten eingegeben und die richtigen Aktionen ausgeführt werden. Fehler passieren natürlich dennoch, und es stellt sich die Frage, inwieweit das DBMS solche Fehler erkennen und abwehren kann.

Das DBMS weiß von sich aus natürlich nicht, wann ein Fehler vorliegt, d.h. unser vorstehender Satz, daß “das DBMS Fehler erkennt bzw. abwehrt”, kann nur so verstanden werden, daß einem Programmierer bzw. DB-Administrator zusätzliche Funktionen angeboten werden, durch die Fehler beschrieben bzw. Reaktionen auf Fehler programmiert werden können. In vielen Fällen kann man solche Angaben als eine Erweiterung des Datenbankschemas oder als eine andere Art von Transaktionsprogrammierung ansehen.

Die einzelnen Datenbankmodelle (bzw. DBMS-Produkte) unterscheiden sich erheblich hinsichtlich der Funktionen, mit denen die Integrität der Daten überwacht werden kann. Diese Vielfalt wollen wir hier nicht auffächern, sondern direkt auf einen im Kontext dieses Lehrmoduls relevanten Aspekt zusteuern: In vielen Fällen können die Integritätsprüfungen erst durchgeführt werden, *nachdem* eine Transaktion schon Inhalte der Datenbank verändert hat. Sofern die Prüfung negativ ausgeht, muß die Transaktion abgebrochen werden. Hierzu können die ohnehin vorhandenen Recovery-Mechanismen, speziell die Funktion zum Zurücksetzen einer Transaktion (s.u.) verwendet werden, d.h. hier besteht ein Querbezug zwischen Integritätsprüfungen und Recovery.

Ein offener Punkt ist hier, von wo aus das Zurücksetzen einer Transaktion ausgelöst wird, infrage kommen das DBMS und die Applikation. Der erste Fall erfordert z.B. ein deskriptives Verfahren, durch das unzulässige, die Zurücksetzung auslösende Datenbankinhalte beschrieben werden können. Das Rücksetzen der Transaktion wäre dann ein “Nebeneffekt” eines Versuchs, die Datenbank in einen unzulässigen Zustand zu überführen. Dieser Ansatz ist aber wenig praxisgerecht, weil das Rücksetzen der laufenden Transaktion keine Nebensache ist, sondern z.B. die folgenden Ausgaben bzw. sonstigen Arbeitsschritte erheblich beeinflussen wird. Daher ist es sinnvoller, das Zurücksetzen einer Trans-

aktion durch das Transaktionsprogramm selbst steuern zu lassen. Um dies technisch zu ermöglichen, muß es eine API-Funktion geben, durch die die laufende Transaktion zurückgesetzt werden kann; in JDBC ist dies die Operation `rollback` .

Obwohl die Integritätsprüfungen nicht immer perfekt sind, bezeichnet man Transaktionen als "*Einheit logisch konsistenter Zustandsübergänge*".

1.2.3 Recovery

Maßnahmen zur Wiederherstellung (von Teilen) einer Datenbank nach einer Beschädigung werden unter der Bezeichnung Recovery zusammengefaßt. Die wichtigsten (allerdings nicht alle) hierbei angewandten Techniken beruhen im Prinzip darauf, zunächst in geschickter Weise redundante Hilfsdaten zu erzeugen und im Falle einer Beschädigung der Datenbank entweder

a. beim sogenannten **Rückwärts-Recovery** die beschädigten bzw. unsicheren Teile der Datenbank in einen früheren Zustand zurückzuversetzen, also sozusagen die Datenbank "zu reparieren", oder

b. beim sogenannten **Vorwärts-Recovery** die Datenbank völlig neu aufzubauen.

Die Alternative a wird dann gewählt werden, wenn nur kleine Teile der Datenbank beschädigt sind, insbesondere beim Zurücksetzen von Transaktionen; Alternative b hingegen, wenn sehr große Teile der Datenbank verfälscht worden sind oder wenn infolge eines Hardware-Fehlers des Speichermediums gar nicht bekannt ist, welche Teile (physisch) defekt sind. In der Praxis sollten beide Alternativen verfügbar sein.

Beide Arten benötigen spezielle Vorbereitungen und Hilfsdaten, sogenannte **Recovery-Daten**. Die meisten Systeme verfahren stark vereinfacht wie folgt:

1. Zu bestimmten Zeitpunkten (z.B. einmal wöchentlich) wird eine

Backup-Kopie der Datenbank auf einem anderen Medium angefertigt, in der Regel aus Kostengründen auf Bandkassetten.
2. Bei jeder Änderung eines Datenbank-Objekts werden die alten und neuen Werte in einem sogenannten **Log** (Logdatei) festgehalten. Beim Einfügen eines neuen Objekts existiert natürlich kein alter Wert, bei der Löschung kein neuer.

Tritt nun ein Fehler ein, so wird bei den beiden oben genannten Alternativen wie folgt verfahren:

a. Alle "verdächtigen" Änderungen werden rückgängig gemacht (Rollback von Transaktionen); die alten Werte werden dem Log entnommen.
b. Die letzte Kopie der Datenbank wird geladen und alle seit dem Zeitpunkt ihrer Anfertigung durchgeführten Änderungen werden mit Hilfe des Logs nachgeholt (Neustart mit "redo" von Transaktionen).

Die Recovery-Algorithmen sehen in der Praxis jedoch wesentlich komplizierter aus, da zur Steigerung der Effizienz bzw. Minimierung der Kosten eine Reihe von Detailänderungen vorgenommen werden; ferner spielen oft die Strukturen des zugrundeliegenden Betriebssystems eine wichtige Rolle in diesen Algorithmen, da in gewissen Fällen auch im Betriebssystem Reparaturen bzw. Neustart erforderlich sind. In allen Fällen sind Transaktionen ein wichtiges Konzept. Transaktionen werden auch als "*Einheit der Wiederherstellung*" ("unit of recovery") bezeichnet.

1.2.4 Concurrency Control

Es ist in wesentliches Ziel von Datenbanken, vielen Benutzern Zugriff zu gemeinsamen Daten zu ermöglichen. Bei vielen Anwendungen muß dies gleichzeitig möglich sein. Bei der parallelen Ausführung von Transaktionen, die auf gemeinsame Daten zugreifen, können einige typische unzulässige Interferenz-Effekte auftreten, sogenannte **Parallelitätsanomalien**, z.B. Verluste von Änderungen oder inkonsistente

Datenbank-Zustände. Durch diese geht die Korrektheit der Datenbank verloren, obwohl die Transaktionen, wenn sie alleine ausgeführt würden, die Korrektheit der Datenbank erhielten.

Die durch Parallelität verursachten Probleme können nach sehr unterschiedlichen Verfahren behandelt werden; die wichtigsten sind:

1. **Sperrverfahren**: Interferenzen werden durch geeignetes Blockieren von Zugriffen zu einzelnen Datenbank-Objekten vermieden. Hierdurch wird i.w. der wechselseitige Ausschluß realisiert, wie er in ähnlicher Form auch in anderen parallel arbeitenden Systemen bekannt ist, z.B. in Betriebssystemen. Die Transaktionen spielen hierbei die Rolle der parallelen Prozesse, die einzelnen Datenbank-Objekte sind die gemeinsam benutzten Ressourcen.

   Je nach der Technik des Sperrens können Deadlocks eintreten. Transaktionen sind somit auch "*Einheiten des Sperrens*".

2. **Zeitstempel-Verfahren**: Jede Transaktion erhält eine Zeitmarke (timestamp). Ziel ist, die Transaktionen zwar verzahnt, aber so auszuführen, als ob sie (aus der Sicht ihres Effekts) in der Reihenfolge ihrer Zeitmarken sequentiell ausgeführt worden wären. Stellt man fest, daß eine Transaktion eine von ihrer Zeitmarke abweichende Wirkung hat, wird sie abgebrochen und mit einer anderen Zeitmarke neu gestartet.

   Da kein Warten auftritt, sind Deadlocks ausgeschlossen. Es besteht allerdings die Gefahr des zyklischen Neustarts.

3. **Optimistische Verfahren**: Bei diesen geht man davon aus, daß Interferenzen nur sehr selten eintreten, was bei vielen Anwendungen tatsächlich der Fall ist. Vorkehrungen gegen Interferenzen werden überhaupt nicht getroffen. Statt dessen wird durch Überwachung der Datenflüsse festgestellt, ob eine unzulässige Interferenz eingetreten ist. Falls ja, werden die betroffenen Transaktionen abgebrochen und neu gestartet.

   Deadlocks können nicht auftreten, aber auch hier droht zyklischer Neustart.

Alle drei Grundtypen der Verfahren haben viele Varianten, alle wie-

derum mit speziellen Vor- und Nachteilen; ferner wurden Mischformen angeregt.

Praktische Bedeutung haben vor allem Sperrverfahren erlangt: in den heute vorhandenen DBMS werden fast ausschließlich Sperrverfahren angewandt.

Zeitstempel-Verfahren wurden speziell für verteilte Datenbanken entwickelt, sind aber auch in zentralen Datenbanken anwendbar. Auch bei verteilten Datenbanken sind die meisten Verfahren Sperrverfahren; die Brauchbarkeit von optimistischen Verfahren ist noch unklar.

Zeitstempel- und optimistische Verfahren wurden zwar aus verschiedenen Motivationen heraus entwickelt, weisen aber sehr viele Gemeinsamkeiten auf. Insbesondere arbeiten beide in der Grundform nach dem Prinzip, die Korrektheit von Abläufen durch Neustart von Transaktionen zu gewährleisten. Deshalb werden sie gemeinsam als "validierende Verfahren" bezeichnet.

Die oben aufgeführten Transaktionseigenschaften sind nur sinnvoll und technisch realisierbar, wenn die Transaktionen nur wenige Objekte verändern, Laufzeiten im Sekundenbereich haben und wiederholbar, also nicht interaktiv sind. Wenn diese Voraussetzungen nicht erfüllt sind, sind "nichtkonventionelle" Transaktionskonzepte erforderlich, oft gleichzeitig mit nichtkonventionellen DBMS. Einen Überblick über eine Vielzahl von nichtkonventionellen Transaktionskonzepten gibt [El92].

In verteilten Datenbanken, auf die wir hier nicht näher eingehen können, kommen zwei weitere Problemkomplexe hinzu:

1. bei replizierten Datenbanken die Erhaltung der Konsistenz der Replikate, insb. also die Propagation von Änderungen
2. die verteilte Ausführung von Transaktionen: Eine Transaktion kann Daten auf mehreren Rechnern ändern; Netzwerkausfälle und Systemabstürze einzelner Rechner bedrohen hier zusätzlich die Fehleratomarität.

Glossar

Aktion: im Kontext von Transaktionen: Ausführung einer der Operationen des Datenbankmodells unter Verwendung zulässiger bzw. korrekter Parameter

Concurrency Control: Verhinderung von Interferenzen zwischen parallelen Transaktionen oder Behebung unzulässiger Effekte eingetretener Interferenz

erreichbar: ein Inhalt bzw. Zustand einer Datenbank ist erreichbar, wenn nach Initialisierung der Datenbank durch eine endliche Folge von Aktionen (mit zulässigen bzw. syntaktisch korrekten Parametern) konstruiert werden kann

Fehleratomarität: Eigenschaft einer Transaktion, daß die Folge von Aktionen ganz oder gar nicht ausgeführt wird

Integrität: Korrektheit des Datenbankinhalts in einem umfassenden Sinne

Isolation: Eigenschaft einer Transaktionsausführung, demzufolge alle Effekte innerhalb einer Transaktion, also vor ihrer Beendigung, für parallele Transaktionen unsichtbar sind

Konsistenz: Synonym zu logische Konsistenz

logisch konsistent: ein erreichbarer Zustand einer Datenbank ist logisch konsistent, wenn eine Realität möglich ist, die durch ihn korrekt wiedergegeben wird

physisch konsistent: ein interner Zustand einer Datenbank ist physisch konsistent, wenn sich die internen Speicherungsstrukturen der Datenbank in einem ordnungsgemäßen Zustand befinden; auf einem physisch inkonsistenten Datenbankzustand kann der Laufzeitkern i.a. nicht mehr korrekt arbeiten

optimistische Concurrency-Control-Verfahren: validierende Verfahren, die von der Annahme, daß Konflikte sehr selten sind, ausgehen und die nur beim Commit einen Validationstest durchführen

Recovery: Wiederherstellung der Datenbank nach einer Beschädigung infolge einer Störung

Rollback: Aufheben der bisherigen Wirkungen einer Transaktion

Serialisierbarkeit: Eigenschaft parallel überlappend ausgeführter Transaktionen, daß deren Effekt der gleiche ist, der bei irgendeiner denkbaren seriellen Ausführung der Transaktionen erreicht worden wäre

Sperrverfahren: Concurrency-Control-Verfahren, in dem Sperren eingesetzt werden, um nichtserialisierbare Verzahnungen von Transaktionen zu verhindern

technisch korrekt: Zustand der Datenbank, der sich gemäß den Transaktionen ergibt, deren Ausführung den Benutzern bestätigt worden ist

Transaktion: Folge von Datenbankzugriffen (Aktionen), die einen konsistenten Datenbankzustand in einen neuen konsistenten Datenbankzustand überführt; Transaktionsausführungen haben folgende Eigenschaften: Fehler-Atomarität, Dauerhaftigkeit, Serialisierbarkeit, endliche Ausführungszeit

validierendes Concurrency-Control–Verfahren: Verfahren, in denen mit Hilfe eines Validationstests bestimmt wird, ob eine Verzahnung von Transaktionen als zulässig angesehen werden kann; im negativen Fall wird eine der involvierten Transaktionen zurückgesetzt und neu gestartet

Zeitstempel-Verfahren: validierendes Concurrency-Control-Verfahren, das Zeitstempel an Objekten und Transaktionen für die Validationstests benutzt und bei jedem Zugriff einen Validationstest durchführt

Lehrmodul 2:

Recovery

Zusammenfassung dieses Lehrmoduls

Unter dem Begriff Recovery faßt man diverse Maßnahmen zusammen, die eine Datenbank vor Schäden infolge von Störungen schützen oder die Schäden beheben. In diesem Lehrmodul untersuchen wir zunächst die Ursachen und Folgen von Störungen und bilden drei wichtige Fehlerklassen, nämlich Medienfehler, Systemfehler und Transaktionsfehler. Wir diskutieren die generellen Anforderungen an das Recovery und stellen dann allgemeinere Grundprinzipien des Recovery und für alle drei Fehlerklassen Datenstrukturen und Algorithmen vor, die Schäden vorbeugen oder sie beheben.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Architektur von DBMS

empfohlen:
- Datenverwaltungssysteme

Stoffumfang in Vorlesungsdoppelstunden: 2.5

Stand: 17.04.2003
©2003 Udo Kelter

2.1 Einführung

Dieses Lehrmodul erhebt nicht den Anspruch, eine auch nur annähernd vollständige Darstellung des Themenkomplexes Recovery in Datenbanken zu sein; eine solche würde leicht ein eigenes Buch füllen. Ursache ist die Vielfalt der Einzelaspekte des Recoverys; diese Vielfalt mag überraschen, da die Grundideen des Recoverys recht einfach sind: in Lehrmodul 1 wurden bereits die beiden wichtigsten Prinzipien vorgestellt, wie eine Datenbank nach einer Beschädigung mit Hilfe von sogenannten Recovery-Daten wiederhergestellt werden kann.

Von den Grundideen wird jedoch in vielerlei Weise abgewichen, was auch die Gliederung des Stoffes erschwert:

- Teilweise wird verhindert, daß überhaupt Schäden an der Datenbank eintreten, es handelt sich also nicht um Wiederherstellung, sondern um *Präventivmaßnahmen.* Die Präventiv-Prinzipien und -Techniken werden deswegen unter Recovery subsumiert, weil die Mechanismen fast die gleichen sind wie im Falle einer wirklichen Wiederherstellung des Zustands der Datenbank.
- Teilweise werden auch außerhalb der Datenbank liegende Beschädigungen mitbehoben, vor allem Schäden an laufenden Prozessen, deren Behebung genausogut *Aufgabe des Betriebssystems* sein könnte. Der Grund ist, daß die gemeinsame Behebung dieser Schäden effizienter ist. Diese Einbeziehung führt jedoch zu einer Aufblähung des Stoffes und Involvierung mit diversen Betriebssystemkonzepten; für detaillierte Algorithmen sind dann auch detaillierte Angaben über das Betriebssystem erforderlich. Dies macht auch eine Vorstellung realer Systeme sehr aufwendig.
- Teilweise ist die Trennung zwischen den Recovery-Daten und der *Arbeitsversion* der Datenbank schwierig; eine Unterscheidung ist dann eher willkürliche Interpretation anhand der Algorithmen auf den gesamten Daten. Dies gilt besonders bei Präventivtechniken.
- Teilweise ist das Recovery der Datenbank eng verzahnt mit dem *Recovery der Recovery-Daten.* Beschädigt werden können natürlich auch die Recovery-Daten.

Der Bereich Recovery hat eine Vielzahl von Berührungspunkten zu anderen Problembereichen, vor allem in Realisierungsfragen. Dies führt dazu, daß viele Konzepte aus diesen Nachbargebieten eindringen und zu einer zusätzlichen Erweiterung des Themenbereichs führen:

- Für das Recovery sind viele komplexe Funktionen zu erfüllen; diese sind letztlich in Programmen zu realisieren, wodurch sich eigene programmiertechnische Probleme ergeben. (Meist ist ein wesentlicher Teil des DBMS, etwa 10 - 30 %, den Recovery-Aufgaben gewidmet.)
- Für eine optimale Effizienz müssen Recovery-Techniken Eigenschaften der Implementierung der Datenbank und des Betriebssystems ausnutzen, z.T. sogar Eigenschaften der Hardware.
- Alle Recovery-Maßnahmen können auch dahingehend interpretiert werden, daß sie die *Zuverlässigkeit* des Datenbanksystems oder des gesamten DV-Systems erhöhen. Viele Maßnahmen sind allgemein verwendbar zur Zuverlässigkeitssteigerung beliebiger DV-Systeme, eventuell sind sie angepaßt an die speziellen Verhältnisse in Datenbanken. Daher ergibt sich in einer Vielzahl von Detailthemen eine Überschneidung mit dem Bereich “Zuverlässigkeit von DV-Systemen”.
- In Datenbanksystemen, die parallel benutzt werden, sind Recovery-Probleme konzeptionell und in der Realisation mehr oder weniger stark beeinflußt vom *Concurrency-Control-Problem* und den dort vorhandenen Konzepten.

Auf das Recovery in verteilten Datenbanken gehen wir hier überhaupt nicht ein; eine sehr ausführliche Behandlung findet sich in [BeHG87].

Bemerkungen zur Stoffgliederung. Eine geradlinige Gliederung des Stoffes sollte ausgehen von einer Analyse des Problems bzw. der Aufgabenstellung und aus dieser systematisch die zu ergreifenden Maßnahmen bzw. zu benutzenden Algorithmen herleiten. An diesem roten Faden orientiert sich die Gliederung des Stoffes in diesem Lehrmodul:

Zuerst wird untersucht, welche Störungen in DV-Systemen auftreten können, die die Integrität der Datenbank verletzen (Abschnitt 2.2).

Die Möglichkeiten für Störungen und resultierende Schäden an der Datenbank sind sehr vielfältig; es stellt sich allerdings heraus, daß die Schäden in ein einfacheres Schema gepreßt werden können, welches in etwa die reparierbaren Einheiten enthält (welche letztlich durch die verfügbaren Reparaturtechniken bestimmt werden). Als nächstes wird untersucht, was nach der Störung das Ziel und die Randbedingungen des Recoverys sind, genauer, welcher neue Zustand der Datenbank angestrebt wird und welche sonstigen Anforderungen zu beachten sind (Abschnitt 2.3). Danach werden die Grundprinzipien des Recoverys (Abschnitt 2.4) vorgestellt. In den Abschnitten 2.6, 2.7 und 2.8 werden besonders die Techniken des transaktionsbezogenen Recoverys genauer behandelt. Zuvor diskutieren wir in Abschnitt 2.5 Alternativen bei der Erzeugung von Logdaten.

Die schon oben erwähnten Abweichungen von der Grundidee des Recoverys führen zu gelegentlichen Abweichungen vom roten Faden durch den Stoff. Eine weitere Schwierigkeit, den roten Faden beizubehalten, liegt in der starken Rückkopplung von den Realisationsformen des Recoverys auf seine Ziele und damit auf seine Prinzipien:

- Teilweise wird das Ziel des Recoverys dadurch bestimmt, was in einer gegebenen Umgebung effektiv und effizient machbar ist.
- Durch die Notwendigkeit des Recoverys der Recovery-Daten werden die Ziele des Recoverys ebenfalls durch die Techniken bestimmt. Hierdurch sind gelegentlich Vorwärtsverweise im Text nötig.

2.2 Quellen und Auswirkungen von Störungen

Recovery-Maßnahmen richten sich gegen Störungen in irgendwelchen Teilen des gesamten Systems aus Hardware, Betriebssystem, DBMS und Benutzerprogrammen, die zum Verlust von Daten bzw. zum Verlust der Korrektheit der Daten führen können. Unter einer **Störung** verstehen wir ein Ereignis, bei dem irgendwelche Teile des Systems nicht erwartungsgemäß bzw. gemäß ihrer Spezifikation arbeiten; man kann dies auch einen Fehler oder ein Fehlverhalten des entsprechenden

Systemteils nennen. Der Effekt einer Störung besteht in einer unerwünschten Veränderung des Inhalts der Datenbank; dies nennen wir einen **Schaden** in der Datenbank. Die Fehlerquellen sollen in diesem Abschnitt genauer untersucht werden; die interessierenden Merkmale sind:

- Wo in der Systemarchitektur entstand die Störung und aus welchem Grund?
- Welche Schäden (in den Daten bzw. Aktivitäten) sind eingetreten?
- Wie werden die Störungen gemeldet, d.h. wie erhalten die Teile des DBMS, die sich mit Recovery befassen, Informationen über Art und Umfang von Schäden?

2.2.1 Systemarchitektur

Ein DBMS ist im engeren Sinne nur eine Komponente eines DV-Systems, welches auf einem Rechner ausgeführt wird. Nur wenige Arten von Störungen "entstehen" im DBMS. Um eine grobe Zuordnung der Arten zu ermöglichen, nehmen wir folgende Schichten einer Systemarchitektur an (eine ausführlichere Darstellung findet sich in [DBSA]):

Ein Benutzerprogramm kann on-line oder off-line ablaufen. Es startet i.d.R. mehrere Transaktionen; z.B. kann bei einer on-line Sitzung zur Datenerfassung jede einzelne Dateneingabe eine Transaktion auslösen.

Transaktionen werden in dieser Systemarchitektur als Programme (bzw. als spezielle Unterprogramme von Benutzerprogrammen) verstanden. Eine Transaktion kann mehrere Datenbankzugriffe ausführen[6]. Transaktionen werden zwar vom Benutzer programmiert, sind jedoch dem DBMS als solche bekannt.

Wir gehen i.f. davon aus, daß das DBMS nicht selbst direkt auf den Platten arbeitet, sondern das Betriebssystem beauftragt, Daten zwischen Arbeitsspeicher und permanenten Speichern zu transportieren.

Die Komponente des DBMS, die die Recovery-Maßnahmen durchführt, nennen wir **Recovery-Manager**.

[6]Den Fall geschachtelter Transaktionen betrachten wir hier nicht.

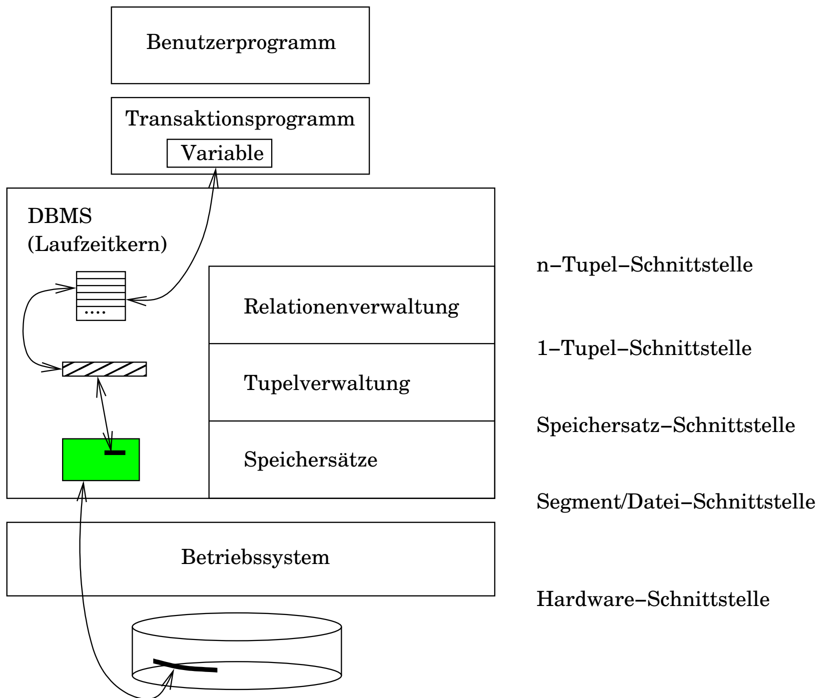


Abbildung 2.1: Eine Schichtenarchitektur für DBMS und Transaktionsprogramme

Die höheren Schichten in der Schichtenarchitektur veranlassen Transporte der auf der jeweiligen Schicht vorhandenen Datengranulate zwischen Arbeitsspeicher und Permanentspeichern. Hierbei werden (vor allem bei satzorientiertem Transfer) Pufferungstechniken angewandt, durch die dem DBMS der Abschluß eines Transports zum Permanentspeicher signalisiert wird, obwohl die Daten effektiv noch nicht dort vorhanden sind. Nehmen wir zur Illustration eine Operation `write(X,v)` an, die den Wert `v` im Objekt oder Tupel `X` speichert. Hierbei sind folgende Schritte auszuführen:

- Sofern die entsprechende Seite noch nicht geladen ist, muß sie erst

geladen werden.
- Der Wert `v` wird in die entsprechende Stelle der Seite kopiert.
- Anschließend muß die Seite noch auf Platte zurückgeschrieben werden. Dies ist aber u.U. nicht sofort möglich, weil z.B. noch andere, nicht beendete Transaktionen in anderen Teilen der Seite arbeiten.

Es kann also längere Zeit dauern, bis eine Änderung "materialisiert" wird. Wenn währenddessen ein Systemfehler eintritt, sind diese Änderungen verloren. Aufgrunddessen ist es sinnvoll, folgende Varianten des Begriffs Datenbank zu unterscheiden:

- Die **physische Datenbank** dies ist der *physische* Zustand, der sich allein aufgrund des Inhalts der persistenten Medien ergibt. Er ist relevant für Rettungsprogramme nach Medienfehlern
- Die **materialisierte Datenbank** ist der *logische* Zustand, der sich aufgrund des Inhalts der persistenten Medien ergibt. Er ist relevant für den Neustart nach einem Systemfehler.
- Die **aktuelle Datenbank** ist der logische Zustand, der sich aufgrund des Inhalts der persistenten *und* der flüchtigen Medien ergibt. Er ist relevant für den normalen Betrieb.

2.2.2 Fehlerklassen

Es gibt vielfältige Arten von Störungen und resultierenden Schäden. Es lohnt sich aber nicht, alle erdenklichen Arten separat zu behandeln, weil man letztlich wegen der verfügbaren Methoden zur Reparatur der Schäden i.w. nur drei Arten von Schäden (bzw. Fehlern) unterscheiden muß. Musterbeispiele von Störungen für die jeweiligen Klassen sind:

- Mediendefekte
- Systemabstürze
- ungeplante Transaktionsabbrüche

Im folgenden beschreiben wir die Fehler und die im Prinzip erforderlichen Schritte zur Behebung der Schäden für jede der drei Klassen. Diese Schritte bezeichnen wir als **Recovery-Grundfunktionen**.

2.2.2.1 Medienfehler

Störungen dieser Art sind recht selten; bei ihnen kann auf Speichermedien nicht mehr zugegriffen werden. Ursachen sind z.B.:

- Gerätedefekte;
- Defekte auf dem eigentlichen Datenträger (Kopfaufsetzer bei einer Magnetplatte, Bandriß, Zerstörung von Datenträgern usw.);
- Organisatorische Fehler (Verlust von Datenträgern, versehentliche Löschung, falsche Anordnung auf Geräten).

In einfachen Fällen kann die Störung durch mehrfache Zugriffsversuche oder andere Maßnahmen im Betriebssystem behoben werden, ohne daß sich die Störung auf das DBMS auswirkt. Die erforderlichen Mechanismen sollen hier nicht diskutiert werden.

In schwererwiegenden Fällen sind mehr oder weniger große Teile der Datenbank verloren oder ihre Zugriffspfade zerstört. Dies nennen wir einen **Medienfehler**. Die Datenbank wird i.d.R. hierdurch *physisch inkonsistent*. Die verlorenen Daten können im laufenden Betrieb nicht rekonstruiert werden. Daher treten in gewisser Weise Folgeschäden ein: Das DBMS muß alle Aktivitäten auf der Datenbank, also alle laufenden Transaktionen und Benutzerprogramme, abbrechen.

Die Reparatur der Schäden ist nur möglich, indem die Datenbank (ggf. nur die betroffenen Teile) rekonstruiert werden. Hierzu benötigt man die beiden folgenden Recovery-Grundfunktionen:

Neuladen einer früher erzeugten Sicherungskopie der Datenbank

globales Redo: Nachholen aller Änderungen, die seit dem Erzeugen der Sicherungskopie stattgefunden haben.

Bei sehr großen Datenmengen kann es durchaus Stunden dauern, eine Datenbank auf diese Weise zu rekonstruieren, d.h. bei unternehmenskritischen Anwendungen muß durch entsprechende Maßnahmen die Wahrscheinlichkeit eines Medienfehlers extrem klein gemacht werden.

2.2.2.2 Systemfehler

Ursache für Störungen im Betriebssystem, die zu Systemabstürzen führen, sind, wenn man so will, Programmierfehler des DBMS-Herstellers. Eine weitere Ursache für Systemfehler sind Stromausfälle.

Die Schäden, die derartige Störungen verursachen können, liegen in drei Bereichen:

- Aktivitäten auf der Datenbank, insbesondere Transaktionen, werden unvollständig ausgeführt; dies läuft auf Transaktionsfehler hinaus, die unten diskutiert werden.
- Der Inhalt der flüchtigen Speicher, also insb. der Puffer, ist verloren. Nach einem Neustart des Systems ist zunächst nur die materialisierte Datenbank verfügbar. Sogar der Effekt bereits abgeschlossener Transaktionen kann verloren gehen.
- Daten, die sich bereits auf Permanentspeichern befinden, werden bei Systemfehlern normalerweise nicht direkt beschädigt. Wenn allerdings das DBMS oder das unterliegende Betriebssystem innerhalb einer Aktion unterbrochen wurde, können Zugriffsstrukturen beschädigt werden und damit große Teile der Datenbank auf den Permanentspeichern physisch inkonsistent sein. Dies ist dann wie ein Medienfehler zu behandeln.

Sofern die Datenbank nur logisch inkonsistent wird, aber physisch konsistent bleibt, sprechen wir von einem **Systemfehler**.

Um die möglichen Schäden in der Datenbank genauer zu analysieren, betrachten wir die Graphik in Bild 2.2:

- Die Transaktionen T3 und T5 werden abgebrochen und hinterlassen möglicherweise die Datenbank in einem inkonsistenten Zustand.
- Die Transaktionen T1, T2 und T4 sind zwar schon abgeschlossen, aber ihre Effekte sind möglicherweise noch nicht materialisiert und gehen daher durch den Systemfehler verloren.

Zur Behebung dieser Schäden werden die folgenden Recovery-Grundfunktionen benötigt:

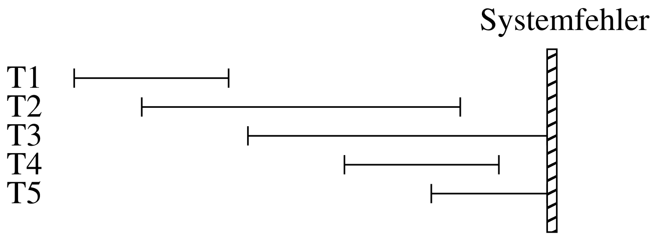


Abbildung 2.2: Abbruch von Transaktionen bei einem Systemfehler

globales Undo: macht die Wirkung aller unterbrochenen Transaktionen rückgängig; dies beinhaltet für jede Aktion, die eine dieser Transaktionen durchgeführt hat, ein ...

Undo einer Aktion: macht die Wirkung einer Aktion rückgängig

partielles Redo: holt die Änderungen der Transaktionen nach, die vor dem Systemfehler beendet wurden und deren Wirkung nicht in der materialisierten Datenbank enthalten ist; dies beinhaltet für jede betroffene Aktion ein ...

Redo einer Aktion: wiederholt die Wirkung einer Aktion

Ausgehend von der Annahme, daß ein Betriebssystemabsturz durchaus öfter vorkommt – über diese Annahme soll hier nicht diskutiert werden – kommt man zur Forderung, daß das globale Undo und partielle Redo sehr rasch, d.h. innerhalb von Minuten durchgeführt werden müssen.

Weiterhin ist der Fall denkbar, daß während des globalen Undos oder des partiellen Redos erneut ein Systemfehler auftritt, so daß diese Grundfunktionen nicht vollständig ausgeführt werden. Hierfür müssen entsprechende Vorsorgemaßnahmen getroffen werden.

2.2.2.3 Transaktionsfehler

Unter dem Begriff Transaktionsfehler faßt man alle Störungen zusammen, die zum Abbruch einer Transaktion führen, z.B.:

- Laufzeitfehler, z.B. eine Division durch Null, fehlende Autorisierung bei einem Zugriff auf die Datenbank
- Abbruch der Transaktion durch das DBMS, u.a.
  - wegen einer anderen Störung wie einem Medienfehler
  - zur Deadlockauflösung
  - zur Synchronisation; es gibt Concurrency-Control-Verfahren, bei denen Transaktionen mit Hilfe von Rücksetzungen synchronisiert werden
- programmierter Abbruch einer Transaktion (z.B. in JDBC mittels `rollback` -Operation), i.d.R. als Folge negativ ausgegangener Konsistenzprüfungen

Bei der Deadlockauflösung und beim Zurücksetzen zu Synchronisationszwecken darf das Benutzerprogramm nichts von dem Transaktionsfehler bemerken, die Transaktion muß hier vom DBMS automatisch neu gestartet werden. Hierzu muß die Transaktion analog zu gespeicherten Prozeduren serverseitig (!) als ausführbares Programm vorliegen, d.h. es muß hier eine deutlich andere Architektur vorliegen als bei den grundlegenden Formen der Nutzung von JDBC.

Der Schaden in der Datenbank besteht darin, daß die Wirkungen der bis zum Abbruch der Transaktion ausgeführten Aktionen i.a. zu einem logisch inkonsistenten Datenbankzustand führen.

Zur Behebung dieser Schäden wird die folgende Recovery-Grundfunktion benötigt:

Rollback einer Transaktion: macht die Wirkung der abgebrochenen Transaktion rückgängig; dies beinhaltet wie schon beim globalen Undo für jede Aktion, die diese Transaktionen durchgeführt hat, ein Undo dieser Aktion.

2.2.3 Ausgaben von Transaktionen

Die Wirkung einer Transaktion besteht i.a. nicht nur in der Veränderung des Inhalts von Datenbankobjekten, sondern auch in Ausgaben und Meldungen, die während der Ausführung der Transaktion auf den

Bildschirm des Benutzers oder in ein Batch-Protokoll ausgegeben werden. Da eine abgebrochene Transaktion keinerlei Wirkung haben soll, können Meldungen nicht unmittelbar ausgegeben werden, sondern müssen zunächst in einer Warteschlange gepuffert werden. Hierfür muß bei serverseitig ausgeführten Transaktionen eine eigene Verwaltungskomponente im Recovery-Manager oder im Rahmen der allgemeinen Kommunikationsmechanismen des Systems vorgesehen werden. Bei Erreichen des Commit-Punktes wird die Meldungsschlange ausgegeben, bei Abbruch der Transaktion gelöscht.

Dieses Verhalten ist jedoch bei programmierten Abbrüchen, die i.d.R. durch eine negativ ausgegangene Konsistenzprüfung ausgelöst werden nicht angemessen, weil hier trotz Rollback eine Beschreibung des Fehlers ausgegeben und ggf. eine Korrekturmöglichkeit angeboten werden sollte. Dies ist ein Sonderfall des allgemeineren Problems interaktiver Transaktionen.

Die Pufferung von Meldungen führt im Prinzip dazu, daß während der Laufzeit einer Transaktion keine Dialoge mit einem Benutzer möglich sind. Es bieten sich zwei Auswege an, beide sind nicht ganz befriedigend:

1. Der Dialog wird in Stücke aufgeteilt, die aus je einer Eingabe und den darauf folgenden Ausgaben bestehen. Jedes dieser Stücke bildet eine Transaktion. Commit oder Rollback der gesamten Änderungen während des Dialogs sind jedoch erst am Ende möglich; hierdurch kann es erforderlich werden, bereits abgeschlossene Transaktionen wieder rückgängig zu machen, was keine normale Aufgabe des Recoverys und technisch meist nicht möglich ist. Probleme bereiten auch Interferenzen mit parallel ablaufenden Transaktionen, die auf die gleichen Daten zugreifen.
2. Die Meldungen werden nicht gepuffert, sondern direkt ausgegeben. Bei einem Rollback ist dann eine Benachrichtigung des Benutzers erforderlich, welche der vergangenen Meldungen und Änderungen jetzt hinfällig sind. Dies ist in der Praxis höchstens dann akzeptabel, wenn die ersten Dialogschritte nur unkritische Sachverhalte betreffen.

2.3 Anforderungen an das Recovery

2.3.1 Die Integrität der Datenbank aus Sicht des DBMS

Das übergeordnete Ziel aller Recovery-Maßnahmen ist die Sicherstellung der Integrität der Datenbank. Dieses Ziel ist so noch recht grob formuliert. Ein DBMS kann nur für bestimmte Aspekte der Integrität sorgen und Verantwortung tragen:

Alle Änderungen am Inhalt der Datenbank sind durch das DBMS gemäß Spezifikation auszuführen. Eine Datenbank, die diese Bedingung erfüllt, nennen wir **technisch korrekt**.

Die bereits in Lehrmodul 1 gegebene Definition bezog sich allerdings auf eine sequentielle Ausführung der Transaktionen. Diese Annahme ist jetzt zu einschränkend. Da andererseits eine Verallgemeinerung des Begriffes technische Korrektheit für den Fall parallel ausgeführter Transaktionen nicht einfach ist, sei für dieses Lehrmodul folgende Arbeitsdefinition von technischer Korrektheit vereinbart:

Der technisch korrekte Zustand der Datenbank ist derjenige, der sich ergeben hätte, wenn alle von der Störung betroffenen Aktivitäten nicht stattgefunden hätten. Die Datenbank ruht somit gedanklich zum Zeitpunkt der Störung.

Ferner muß die Datenbank während der Zeiträume, zu denen das DBMS nicht aktiv ist, auf permanenten Speichermedien erhalten werden. Die Verhinderung von Störungen und Behebung von Schäden in dieser Zeit wird man eher als Aufgabe des Betriebssystems ansehen, wenngleich auch andere Mittel zur Behebung solcher Schäden benutzt werden können.

2.3.2 Speicherteile der Datenbank

In der Einleitung wurde bereits postuliert, daß man sich eine Datenbank so vorstellen kann, als ob sie aus einer Menge von Datenbankobjekten bestünde, die einzeln gelesen, verändert, erzeugt oder gelöscht werden können. Dies beschreibt das Bild eines Programmierers oder Benutzers von der Datenbank. Es besagt im Prinzip nichts darüber, in welchen Strukturen die Datenbank gespeichert wird; diese sind Geheimnis des

Zugriffssystems. Wenn wir annehmen, die Datenbank wäre zwecks Geheimhaltung verschlüsselt gespeichert, so wirkt die Datenbank wie eine einzige, unstrukturierte Variable mit einer völlig strukturlosen Form der Speicherung. In realen Systemen kann man sich die Datenbank jedoch sehr oft als aus getrennt gespeicherten Teilen zusammengesetzt vorstellen. Ein typischer Speicherteil einer Datenbank ist eine Datei, wie sie durch das Betriebssystem verwaltet wird. Wir nehmen an, daß jedes Datenbankobjekt der Benutzersicht in einem Speicherteil der Datenbank von der jeweils kleinsten Einheit enthalten ist.

Speicherteile der Datenbank sind insofern für das Recovery relevant, als sie unabhängig voneinander beschädigt und wiederhergestellt werden können.

Statt auf die gesamte Datenbank kann man stets die Recovery-Mechanismen auf Teile der Datenbank anwenden, sofern sich der Aufwand lohnt.

2.3.3 Qualitätsmerkmale von Recovery-Mechanismen

Nach einer Störung muß es das Ziel des DBMS sein, die Datenbank wieder in einen Zustand zu versetzen, der für die Benutzer akzeptabel ist; dies soll

- möglichst schnell geschehen,
- unter Verlust von möglichst wenig bereits geleisteter Arbeit,
- möglichst automatisch, d.h. ohne zusätzliche Handarbeit,
- mit möglichst geringen Kosten.

Im einzelnen ergeben sich folgende "Qualitätsanforderungen" für Recovery-Mechanismen:

1. Es soll möglichst wenig Arbeit der Benutzer verloren gehen. Anders gesagt soll das Recovery möglichst vollständig sein. Bei interaktivem Zugriff zur Datenbank sollen z.B. möglichst wenig Dateneingaben verloren gehen, bei Batch-Verarbeitung sollen aufwendige Programme nicht von vorne laufen müssen.

2. Die beschädigten Teile der Datenbank stehen den Benutzern nicht zur Verfügung. Das Recovery soll also schnell sein. Die Recovery-Maßnahmen sollen nicht unnötig aufwendig sein. (Vgl. unten den Kostenaspekt; die Größe des Schadens und der Aufwand für das Recovery sollen in einem vernünftigen Verhältnis zueinander stehen.)
3. Die Wiederherstellung beschädigter Teile der Datenbank soll lokal erfolgen, d.h. die Recovery-Maßnahmen sollen sich möglichst nur auf die beschädigten Teile der Datenbank auswirken. Während der Wiederherstellung soll der Zugriff auf die unbeschädigten Teile nicht behindert werden. Insbesondere soll transaktionsbezogenes Recovery möglich sein.
4. Recovery-Maßnahmen sollten je nach Bedarf automatisch durchgeführt werden. Die Benutzer bzw. Administratoren sollen so weit wie möglich von manueller Arbeit entlastet werden.
5. Die Hilfsdaten für das Recovery können natürlich ebenfalls beschädigt werden. Für ihre Sicherheit muß ebenfalls gesorgt werden.
6. Die Kosten des Recoverys, letztlich Qualität und Quantität des Aufwands, sollen möglichst gering sein. Grob gesagt sind die Kosten umso höher, je besser die vorstehenden Qualitätsanforderungen erfüllt werden. In der Praxis müssen daher oft Kompromisse eingegangen werden; mit fallenden Kosten für die Hardware werden jedoch tendenziell leistungsfähigere Recovery-Mechanismen realisierbar.

2.3.4 Zielzustand des Recoverys

Das Recovery soll zu einem für die Benutzer akzeptablen Zustand der Datenbank führen. Was akzeptabel ist, hängt von vielen Umständen ab, vor allem von

- Art und Umfang des Schadens,
- erforderlicher Geschwindigkeit und den übrigen oben erwähnten Qualitätsmerkmalen.

Je nach den Eigenschaften des durch das Recovery hergestellten Zustands der Datenbank können wir einige *Arten des Zielzustandes des Recoverys* unterscheiden (nach [Ve77]). Das Wort “Ziel” ist leider mit

zwei Ausnahmen nicht so zu verstehen, daß der Benutzer an den Eigenschaften dieser Zielzustände besonders interessiert wäre, sondern daß Zustände mit diesen Eigenschaften durch die Recovery-Techniken effizient hergestellt werden können.

1. Der (technisch) *korrekte Zustand*: alle Effekte der Störung in der Datenbank (und ggf. in den Aktivitäten auf der Datenbank) werden vollständig behoben. Dies setzt voraus, daß die Datenbank während des Recoverys nicht verändert wird bzw. daß das Recovery schneller als die laufenden Änderungen ist.
2. Ein *früherer korrekter Zustand*: die Datenbank wird in einen früher korrekten, jetzt aber nicht mehr aktuellen Zustand versetzt. Die Änderungen in der Zwischenzeit sind verloren.
3. Ein *hypothetischer früherer korrekter Zustand*: die Datenbank wird in einen Zustand versetzt, der früher hätte eintreten können, sofern gewisse Änderungen ihres Inhalts in anderer Reihenfolge stattgefunden hätten. Dies ist dann der Fall, wenn einzelne Teile der Datenbank in einen früheren Zustand zurückversetzt werden, der nie gleichzeitig mit den derzeitigen Zuständen anderer Teile existierte, z.B. beim Rollback von Transaktionen.
4. Ein *früherer korrekter Teilzustand*: Die Datenbank wird in einen Zustand versetzt, der zum Teil mit einem früher oder jetzt noch korrekten Zustand übereinstimmt. Der Inhalt der restlichen Teile ist verloren bzw. wird neu initialisiert.
5. Ein *logisch konsistenter Zustand*: die Datenbank wird in irgendeinen Zustand versetzt, der die statischen Integritätsbedingungen erfüllt, und der dem korrekten Zustand möglichst nahe kommt. Dieser Zustand kann "schlimmer" inkorrekt sein als infolge des Fehlens von früheren Änderungen oder des Verlustes von Teilen, d.h. nicht infolge von derartigen Verfälschungen entstanden sein.

Die o.g. Arten von Zielzuständen beschreiben Eigenschaften des Zustands der Datenbank nach Beendigung der *vom DBMS* angebotenen Recovery-Maßnahmen, die entweder automatisch oder unter Kontrolle des Benutzers oder Operateurs stattfinden. Darüber hinaus bleibt es

bei den Fällen 2 bis 5 dem Benutzer natürlich unbenommen, weitere Wiederherstellungsmaßnahmen *von Hand* durchzuführen.

Ein ungefähres Maß für die Korrektheit eines Datenbankzustandes ist die Zahl der Änderungen einzelner Datenbankobjekte von Hand, die erforderlich wären, um einen völlig korrekten Zustand herzustellen. Ganz grob gesagt wird dann von Fall 1 bis 5 gehend der Zustand, der durch das Recovery erreicht wird, immer inkorrekter, die Vollständigkeit des Recoverys ist immer geringer; exakt kann das Maß der Korrektheit bzw. die Vollständigkeit des Recoverys bei den Alternativen 2 - 5 weder absolut noch relativ zueinander angegeben werden.

Es sei noch explizit darauf hingewiesen, daß zwischen der Größe des Schadens, den man ebenfalls in der erforderlichen Handarbeit zu seiner Reparatur messen könnte, und der Inkorrektheit des Zielzustands des Recoverys kein direkter Zusammenhang bestehen muß. So kann eine Datenbank, die wegen eines einzigen Records inkonsistent geworden ist, durch Recovery der 2. Art auf den Stand von vor einer Woche zurückgesetzt werden, obwohl dann 1000 Änderungen (in anderen Teilen) der Datenbank verlorengehen. Intuitiv wird man hier jedoch ein Mißverhältnis zwischen der Größe des Schadens und der Größe der durch das Recovery betroffenen Teile der Datenbank sowie der Inkorrektheit des Datenbankzustandes nach dem Recovery empfinden.

2.4 Grundprinzipien des Recoverys

Im letzten Abschnitt wurde eine Auswahl von Zuständen der Datenbank angegeben, die nach einer Störung angestrebt werden können. Nunmehr erhebt sich die Frage, wie dies effektiv erreicht werden soll.

2.4.1 Recovery-Daten

Recovery ist letzten Endes immer nur deshalb möglich, weil neben der Arbeitsversion der Datenbank zusätzlich andere Daten für Recovery-Zwecke, sogenannte **Recovery-Daten**, gespeichert werden. Sofern die enthaltenen Informationen identisch sind, liegt Redundanz vor, in vie-

len Fällen enthalten die Recovery-Daten jedoch andere Informationen (z.B. für Zielzustandstypen 2 und 4).

Viele der Störungen, die die Arbeitsversion der Datenbank beschädigen, können ebenso die Recovery-Daten beschädigen. Man muß somit, je nach der Wichtigkeit der Recovery-Daten, auch für diese ein Recovery vorsehen, sozusagen eine Sekundär-Recovery-Maßnahme mit neuen Sekundär-Recovery-Daten. Diese Daten können natürlich auch wieder beschädigt werden, usw.; letztlich kann nie eine völlige Sicherheit gegen alle denkbaren Fälle erreicht werden, denn irgendwann muß dieser Kreis in der Praxis abgebrochen werden. Die Sicherheit ist immer nur endlich.

2.4.2 Risikostreuung

Recovery-Daten und die Mechanismen, die sie benutzen, helfen nie bei allen denkbaren Arten von Störungen, sondern immer nur bei bestimmten. Hieraus folgt:

1. Man soll möglichst *verschiedene Arten von Recovery-Daten und -Mechanismen* vorsehen, die komplementäre Risiken abdecken, um die verschiedenen Arten von Störungen, mit denen gerechnet werden muß (eventuell sogar mehrfach) abzudecken. Durch die Redundanz in den Recovery-Daten wird ebenfalls die *Zuverlässigkeit des Recoverys* erhöht.
2. Die Recovery-Daten dürfen nicht so gespeichert werden, daß sie bei Störungen, bei denen sie helfen sollen, selbst verletzt wären. So sollten beispielsweise zum Schutz gegen physische Schäden die Recovery-Dateien auf anderen Geräten oder Medien gespeichert werden als die Arbeitsversion der Datenbank.

Ein wichtiges Mittel zur Risikostreuung ist die Verwendung verschiedener Speichermedien. Bei den heute verfügbaren Technologien kommen in Großrechnern für die Massendatenhaltung nur Platten oder Bänder / Bandkassetten in Betracht. Typischerweise sind beide Medien gleichzeitig verfügbar, so daß es sich anbietet,

- die Arbeitsversion der Datenbank auf Platte zu speichern und
- die Recovery-Daten auf Band, zumindest längerfristig; bei manchen Arten von Recovery-Daten ist es erforderlich, sie zumindest zeitweise auf Medien mit schnellem Zugriff zu halten, also Platte.

Wir werden *Platte* und *Band* im folgenden als Synonym benutzen für *irgendwelche* permanenten Speichermedien, die die erforderliche Zugriffsgeschwindigkeit haben und die auf unterschiedlichen Geräten montiert sind, also eine Risikostreuung gegen Gerätefehler ermöglichen. Das Band kann also auch eine andere Platte sein.

2.4.3 Reparaturprinzipien

Im Sinne von Wiederherstellung bedeutet Recovery, daß man von einem defekten Zustand der Datenbank ausgeht und daß dieser Zustand anschließend in einen akzeptablen anderen Zustand verändert wird. Dieses Denkschema ist allerdings bei Präventivmaßnahmen, die man ja ebenfalls zum Recovery zählt, nicht anwendbar. Im Falle einer Störung, gegen die die Präventivmaßnahme schützt, ist die Datenbank gerade *nicht* defekt und diesbezügliche Reparaturaktivitäten sind nicht erforderlich. Wir trennen daher zwischen *Reparaturprinzipien* und *Präventionsprinzipien*.

Wir können die Reparaturprinzipien grob danach unterscheiden, ob sie von der Arbeitsversion der Datenbank oder von den Recovery-Daten aus versuchen, den gewünschten neuen Zustand zu konstruieren.

2.4.3.1 Arbeitsversion der Datenbank als Ausgangsbasis

Diese Prinzipien sind immer nur dann anwendbar, wenn die Arbeitsversion der Datenbank nach der Störung überwiegend erhalten geblieben ist. Es wird versucht, durch lokale Änderungen den erwünschten Zustand zu konstruieren. Hierbei wird nach folgenden Prinzipien verfahren.

Rückwärts-Recovery (**backward recovery**): Die beschädigten Teile der Datenbank werden in einen früheren Zustand zurückversetzt, möglichst in den unmittelbar vor der Störung. In den meisten

Fällen bedeutet dies, den Effekt von vorzeitig abgebrochenen Transaktionen rückgängig zu machen (Rollback). Vorbereitend müssen die früheren Inhalte der durch Transaktionen veränderten Datenbankobjekte in den Recovery-Daten gespeichert werden. Je nach den Umständen wird die Datenbank in Zustände des Typs 1, 2 oder 3 gebracht.

Fehlerkompensation: Dieses Prinzip ist nur dann anwendbar, wenn die Störung so interpretierbar ist, daß sie den eigentlich gewünschten, korrekten Zustand der Datenbankobjekte *invertierbar funktional* in den nun vorhandenen, inkorrekten Zustand umformt. Der korrekte Zustand kann nun dadurch herbeigeführt werden, daß eine *inverse* Funktion auf die Zustände der betroffenen Datenbankobjekte angewendet wird. Beispiel: Eine Zahl wurde um 1000 erhöht, sollte aber nur um 100 erhöht werden. Die Abweichung beträgt 900, sie kann durch Subtraktion kompensiert werden.

Dieses Recovery-Prinzip ist offensichtlich nur in wenigen Fehlerklassen anwendbar. Es hat allerdings den Vorteil, auch bei schon abgeschlossenen Transaktionen anwendbar zu sein. Unter Umständen ist es auch effizienter als ein Rollback der Transaktion.

Bei richtiger Anwendung wird die Datenbank durch die Fehlerkompensation in den korrekten Zustand (Typ 1) gebracht.

Rettung: In manchen Fällen sind die üblichen Recovery-Prinzipien nicht mehr anwendbar, weil entweder die Recovery-Daten ebenfalls zerstört sind oder gar keine vorgesehen sind, insbesondere für die (Primär-) Recovery-Daten. In dem Fall muß man "retten, was zu retten ist". Dies ist zwar Flickschusterei, aber besser als ein völliger Verlust der Datenbank. Eine Reihe von realisierten Systemen (vgl. [Ve77]) bietet Programme an, die die Datenbank selbständig oder in Interaktion mit dem Benutzer in einen Zustand versetzen, der zumindest physisch und logisch konsistent ist (Zustandstyp 3 oder 5). Denkbare Maßnahmen sind:

- Die beschädigte Arbeitsversion der Datenbank (oder die beschädigten Recovery-Daten) werden daraufhin untersucht, welche Teile

unbeschädigt geblieben sind und weiterverwendet werden können.

- Bei beschädigten Zugriffspfaden und ausreichender Redundanz in diesen kann der *wahrscheinliche* Zustand der Datenbank vor der Störung rekonstruiert werden.

Fehlerkompensation und Rettung sollten sehr zurückhaltend angewandt werden. Sie sind mit relativ viel Handarbeit verbunden und die Gefahr ist groß, daß neue Fehler eingeführt werden.

2.4.3.2 Recovery-Daten als Ausgangsbasis

Bei diesem Prinzip benötigt man in jedem Fall sogenannte Dumps als Teil der Recovery-Daten. Ein (Backup-) **Dump** ist eine Kopie des Zustands der Datenbank oder eines Teils von ihr in der Vergangenheit, der in der Regel zum damaligen Zeitpunkt korrekt war und für Recovery-Zwecke aufgezeichnet wurde.

Die Benutzung des Dumps als neue Arbeitsversion der Datenbank entspricht Zielzustandstyp 2 oder 3, je nachdem, ob die gesamte Datenbank oder nur Teile durch die alte Version ersetzt wurden.

Zusätzlich können nach dem Laden des Dumps Änderungen an der Datenbank, die seit dem Zeitpunkt der Erstellung des Dumps durchgeführt wurden, nachgeholt werden. Hierzu ist es erforderlich, daß vorbereitend geeignete Informationen über diese Veränderungen gespeichert werden. Je nach der Vollständigkeit dieser Informationen kommt Zielzustandstyp 1, 2 oder 3 zustande.

Die wichtigste Technik in diesem Zusammenhang ist das *Wiederholen von Transaktionen* auf dem Dump. Da hier im wesentlichen der zeitliche Ablauf der Ereignisse in der Datenbank wiederholt wird, spricht man von **Vorwärts-Recovery** (*forward recovery*), im Gegensatz zum Rückwärts-Recovery, wo die Zeit "ein Stück zurückgedreht" wird.

2.4.4 Präventionsprinzipien

Merkmal von Maßnahmen nach dem Präventionsprinzip ist, daß bei gewissen Störungen gar kein echter Schaden in der Arbeitsversion der

Datenbank eintritt. Meist sind nach der Störung überhaupt keine Reparaturaktivitäten in der Datenbank erforderlich oder nur ein "Aufräumen", welches jedoch keine weiteren Hilfsdaten erfordert.

Bei den Präventivmaßnahmen ist oft keine klare Trennung zwischen Arbeitsversion der Datenbank und Recovery-Daten möglich.

Verzögertes Schreiben (*deferred update*): Dieses Prinzip ist nur in bestimmten Situationen bei Transaktions- und Systemfehlern wirksam. Alle Schreibaktionen (Änderungen) einer Transaktion werden möglichst erst unmittelbar vor dem Commit-Zeitpunkt ausgeführt. Die zu schreibenden Werte müssen bis dahin aufgehoben werden. Praktisch müssen Kopien der betroffenen Objekte im Arbeitsspeicher angelegt werden, auf denen die Änderungen zunächst stattfinden.

Solange noch keine Schreibaktion ausgeführt wurde, ist das Rollback einer Transaktion trivial: lediglich die Kopien der Datenbankobjekte sind zu löschen und die Transaktion ist beim Recovery- Manager abzumelden.

Zur Vermeidung von gewissen Parallelitätsanomalien empfiehlt es sich sogar, alle Schreibaktionen im Rahmen der Abarbeitung des Commit-Befehls auszuführen. Da man das Commit als atomares Ereignis ansieht, ist die Zeitdauer vom ersten Schreiben bis zum Commit Null, jedenfalls in dieser idealisierenden Annahme.

Es ergibt sich folgender Vorteil: Bei einem Systemfehler ist die Arbeitsversion der Datenbank auf Platte nach einem Neustart des Systems sofort in einem früher korrekten Zustand (Typ 2), lediglich die vorher noch aktiven und abgebrochenen Transaktionen sind nachzuholen, um den korrekten Zustand der Datenbank herzustellen. Insbesondere sind keine Maßnahmen erforderlich, um einen logisch und physisch konsistenten Zustand der Datenbank herzustellen. Für viele Zwecke kann die Datenbank unmittelbar weiterverwendet werden.

Einziger Nachteil des verzögerten Schreibens ist der Aufwand für die Kopien. Bei "kleinen" Transaktionen ist dieser Aufwand erträglich, so daß das verzögerte Schreiben (bzw. das Schreiben erst bei Commit) unbedingt zu empfehlen ist.

Vorsichtiges Ändern (*careful replacement*): Dieses Prinzip wird vor allem in der Verbindung mit dem verzögerten Schreiben eingesetzt und behebt weitgehend dessen Schwachpunkt, denn es schützt gegen physische Inkonsistenzen infolge von Systemfehlern beim Ändern der Datenbank (also beim Rückschreiben der Kopien). Es kann nur durch eine spezielle Programmiertechnik bei den verändernden Aktionen realisiert werden.

Die zu verändernden Einheiten können Sätze, Speicherseiten o.ä. sein. Die Gefahr bei konventionellem Ändern besteht darin, daß stets eine Zeitspanne lang der alte Zustand vor der Änderung schon zerstört, der neue Zustand jedoch noch nicht vollständig hergestellt ist, vor allem wenn mehrere Einheiten von einer Änderung betroffen sind.

Beim vorsichtigen Ändern werden zunächst alle Änderungen auf Kopien gemacht, die Kopien in die Datenbank eingefügt (auf Platte!), so daß während des Änderns zwei Versionen (sogenannte "virtuelle Kopien") derselben Einheiten nebeneinander existieren können. Dann muß noch zur neuen Version "umgeschaltet" werden, und die alten Versionen werden gelöscht. Während des Umschaltens besteht kein Schutz gegen Systemfehler, die Dauer des Umschaltens ist jedoch um Größenordnungen kleiner als die der gesamten Änderung.

2.5 Logging

Die in Abschnitt 2.2.2 beschriebenen Grundfunktionen sind nicht ohne vorbereitende Maßnahmen denkbar; konkret müssen alle relevanten Daten in einem Log protokolliert werden.

Unter einem (Recovery-) **Log** versteht man eine Aufzeichnung der Änderungen in der Datenbank für Recovery-Zwecke, insb. für die Grundfunktionen, aber ggf. auch diverse andere Zwecke. Andere Bezeichnungen sind *audit trail*, *system log* und *journal*. Der Log enthält für jede ändernde Aktion einen Eintrag (für lesende Aktionen sind i.a. keine Einträge nötig) mit folgenden Angaben:

- Identifikation der Transaktion
- Identifikation des Objekts

- Art der Aktion
- Undo- und/oder Redo-Daten

sowie ggf. zusätzliche Angaben wie Datum und Uhrzeit, Identifikation betroffener Seiten der Datenbank oder Verweis auf den vorhergehenden Logeintrag der gleichen Transaktion.

Neben den Einträgen für ändernde Aktionen gibt es zusätzlich Einträge für Beginn, Commit und Rücksetzung von Transaktionen (BOT-, EOT- und Abort-Einträge) und weitere interne Zwecke.

Ferner erhält jeder Log-Eintrag eine laufende Nummer (*log sequence number*), die verschiedenen Zwecken dient.

Aus Sicherheitsgründen kann ein Log doppelt geführt werden, z.B. parallel auf Platte und einem Band.

2.5.1 Undo- vs. Redo-Logging

Für das Undo bzw. Redo einer Aktion werden andere Daten benötigt; daher kann man reine **Undo-Logs** bzw. **Redo-Logs** benutzen, die keine Daten für den jeweils anderen Zweck enthalten. Man kann natürlich auch einen gemeinsamen Log für beide Zwecke benutzen; wegen der gemeinsamen Daten ist dies zwar platzsparend, aber dennoch nicht immer sinnvoll, weil die Logs für unterschiedliche Zwecke benutzt werden.

Unter **Vorwärts-Logging** bzw. **Redo-Logging** versteht man die Erzeugung von Logdaten für die Redo-Funktionen. Beim globalen bzw. partiellen Redo werden vollständige Transaktionen in ihrer ursprünglichen Reihenfolge nachgeholt; für diese Zwecke reicht es aus, den Log auf einem sequentiellen Medium wie einem Band zu speichern. Für die Erzeugung der Redo-Logdaten gilt folgende Regel:

> *Die Redo-Logdaten müssen vor dem Abschluß der Commit-Anweisung materialisiert sein.*

Andernfalls wäre einerseits der Applikation und damit dem Benutzer der erfolgreiche Abschluß der Transaktion angezeigt worden, andererseits

könnten bei einem sofort folgenden Systemabsturz solche Änderungen der Transaktion, die noch nicht in der materialisierten Datenbank enthalten sind, nicht wiederhergestellt werden.

Unter **Rückwärts-Logging** bzw. **Undo-Logging** versteht man die Erzeugung von Logdaten für die Undo-Funktionen. Da das Rollback von Transaktionen sehr schnell sein muß, kommt als Speicherungsmedium für einen Rückwärts-Log nur die Platte in Frage. Für die Erzeugung der Undo-Logdaten gilt die folgende (sogenannte *write ahead log-*) Regel:

> *Die Undo-Logdaten müssen vor der Änderungen der materialisierten Datenbank im Log materialisiert sein.*

Andernfalls ist nach einem Systemfehler kein Undo mehr möglich.

2.5.2 Archiv-Logs

Bei großen Systemen und hohem Änderungsaufkommen können die Logs sehr groß werden. An dieser Stelle hilft die Beobachtung weiter, daß die Undo-Logdaten sehr schnell überflüssig werden: diese werden nur für das Rollback von Transaktionen benötigt, von denen wir annehmen, daß sie relativ kurz sind. Sobald eine Transaktion ihr Commit erreicht, können im Prinzip alle zu dieser Transaktion gehörigen Undo-Daten gelöscht werden. Bei einem sequentiell organisierten Log ist das aber nicht ohne weiteres möglich. Eine Lösung besteht darin, den gesamten Log zu teilen in

- einen **aktiven Log**, der auf Platte steht, der die neuen Einträge enthält und der sowohl Undo- als auch Redo-Daten enthält, und in
- einen oder mehrere **Archiv-Logs**, die alle alten Einträge enthalten, die nur noch Redo-Daten enthalten und die auf anderen Medien stehen können (z.B. CD-ROM oder Band).

In bestimmten Zeitabständen oder nach Erreichen einer bestimmten Größe wird ein neuer aktiver Log eingerichtet und der bisherige aktive Log zum nächsten Archiv-Log. Da alle Logeinträge noch nicht abgeschlossener Transaktionen auf Platte vorhanden sein müssen, muß der

alte aktive Log solange auf Platte bleiben, wie er noch Einträge aktiver Transaktionen enthält. (Alternativ könnte man diese Transaktionen abbrechen, rücksetzen und neu starten.) In den alten aktiven Log werden auch alle noch folgenden Einträge dieser Transaktionen eingeschrieben. Alle Einträge von neu gestarteten Transaktionen kommen in den neuen aktiven Log.

Nachdem alle Transaktionen, die noch dem alten aktiven Log zugeordnet sind, beendet sind, kann dieser komprimiert und ggf. schon parallel für eine eventuelle Verwendung in einem globalen Redo präpariert werden:

- Falls es ein gemeinsamer Undo- und Redo-Log ist, können die Undo-Daten entfernt werden.
- Mehrfache Wertzuweisungen für das gleiche Datenelement können mit Ausnahme der letzten gelöscht werden.
- Einträge von zurückgesetzten Transaktionen können entfernt werden.
- Die Änderungen können ggf. schon so vorsortiert werden, daß alle Änderungen, die eine Seite betreffen, hintereinander liegen, also beim globalen Redo eine mehrfache Übertragung der gleichen Seite vermieden wird.

2.5.3 Zustands- vs. Transitions-Logging

Wir hatten bisher völlig offengelassen, woraus die Undo- bzw. Redo-Daten überhaupt bestehen. Hier sind zwei Ansätze denkbar:

Beim **Zustands-Logging** werden *Zustände* protokolliert:

- für das Redo: der Zustand nach der Aktion
- für das Undo: der Zustand vor der Aktion

Die Implementierung des Redo bzw. Undo ist hier sehr einfach: der vorhandene Wert wird mit dem neuen bzw. alten Wert überschrieben.

Beim **Transitions-Logging** werden *Zustandsübergänge* protokolliert:

- für das Redo: die beobachtete Zustandsveränderungsfunktion

- für das Undo: die invertierende Funktion der beobachteten Zustandsveränderungsfunktion

Die Implementierung des Redo bzw. Undo besteht hier darin, die Zustandsveränderungsfunktion bzw. ihre invertierende Funktion erneut auszuführen.

2.5.4 Logging auf verschiedenen Abstraktionsebenen

Logging ist unabhängig von der Wahl zwischen Zustands- und Transitions-Logging auf beliebigen Abstraktionsebenen (s. Schichtenarchitektur in Bild 2.1 in Abschnitt 2.2.1) möglich. Bei der Entscheidung sind folgende Aspekte zu berücksichtigen:

- der Umfang der Logdaten; dieser sollte möglichst klein sein.
- die Bestimmbarkeit der Undo-Daten: zunächst kennt das System nur die Aktion
- der Aufwand zur Implementierung der Grundfunktionen

Das Logging auf Seitenebene ist sehr einfach zu realisieren, die Undo-Daten sind leicht bestimmbar, nachteilig ist aber das hohe Datenvolumen: selbst wenn nur eine einzige Zahl, die nur wenige Byte beansprucht, geändert wird, muß eine ganze Seite von z.B. 1 kB in den Log aufgenommen werden. Dieses Problem kann durch Transitions-Logging vermieden werden, da hier Veränderung viel kompakter beschrieben werden kann. Beim Logging auf Seitenebene muß ferner verhindert werden, daß mehrere Transaktionen parallel auf der gleichen Seite arbeiten, es kommt also zu logisch nicht notwendigen Verzögerungen.

Das Logging auf Speichersatzebene ist ebenfalls einfach zu realisieren, die Undo-Daten sind leicht bestimmbar, das Problem des Datenvolumens entfällt hier, wenn die Sätze kurz sind.

Wenn man auf der Ebene der Tupel/Objekte relativ feingranulare Operationen wie "setze Attribut A an Objekt O" loggt, läßt sich, sofern die Applikationen häufig einzelne Attribute ändern, das Datenvolumen deutlich gegenüber dem Logging auf Speichersatzebene reduzieren.

Auf der n-Tupel-Ebene kann das Datenvolumen noch weiter reduziert werden, da hier durch ein einzelnes Kommando Veränderungen an

sehr vielen Objekten kompakt beschrieben werden können. Dennoch ist das Logging auf dieser Ebene i.a. aus mehreren Gründen nicht zu empfehlen: (a) Diese Aktionen können mehrere Seiten betreffen und (ohne besondere Maßnahmen) ist die Fehler-Atomarität dieser Aktionen nicht gewährleistet: bei einem Systemfehler ist nicht sichergestellt, daß diese Aktionen ganz oder gar nicht materialisiert sind. Anders gesagt muß die Datenbank immer “aktionskonsistent” gehalten werden; dies ist nicht immer möglich. (b) Die Undo-Daten können i.a. nicht in gleicher Weise kompakt dargestellt werden. (c) Bei objektorientierten oder navigierenden Datenbankmodellen kann es sehr schwierig sein, die Undo-Daten überhaupt auf dieser Ebene mit vertretbarem Aufwand zu bestimmen.

2.5.5 Logging auf Transaktionsebene

Eine weitere Alternative besteht darin, komplette Transaktionsaufrufe zu loggen und beim Redo zu wiederholen. Der Log enthält hier den Namen der Transaktion und die Aufrufparameter. Voraussetzung ist hier, daß die Transaktionsprogramme komplett innerhalb des DBS verwaltet werden, denn sie müssen bei einem Neustart ja vom DBMS aufgerufen werden können.

Äußerst problematisch ist hier die Gewinnung der Undo-Daten, vgl. die Bemerkungen zum Logging auf der n-Tupel-Ebene. Insgesamt ist das Logging auf Transaktionsebene nur in seltenen Ausnahmefällen sinnvoll.

2.6 Recovery für Transaktionsfehler

Rückwärts-Recovery. Für das **Rollback** einer Transaktion wird ein Rückwärts-Log benötigt. Dieser Log wird von hinten aus rückwärts nach Einträgen für diese Transaktion durchlaufen, bis der BOT-Eintrag gefunden wird. Bei jedem gefundenen Eintrag wird das Aktions-Undo ausgeführt. Durch eine Verkettung der Einträge kann die Suche nach den Einträgen stark beschleunigt werden.

Präventionsmaßnahmen. Um die Wahrscheinlichkeit bzw. den Umfang von Undo-Maßnahmen klein zu machen, kann man auf allen Ebenen Schreibaktionen möglichst lange aufschieben; man nennt dies **verzögertes Schreiben**. Statt also Schreibaktionen immer sofort auf die nächsttiefere Ebene weiterzugeben, arbeitet man zunächst auf Puffern und propagiert diese Änderungen erst möglichst spät, am besten erst unmittelbar vor dem Commit, "nach unten". Vor der ersten Schreibaktion ist ein Rollback daher trivial, und die materialisierte DB ist bei einem Systemfehler nicht logisch inkonsistent.

Das verzögerte Schreiben ist generell sehr zu empfehlen, lediglich bei sehr großen Mengen geänderter Daten kann es wegen des Platzaufwands sinnvoller sein, auf das verzögerte Schreiben zu verzichten.

2.7 Recovery für Systemfehler

2.7.1 Präventionsmaßnahmen

Ziel ist hier natürlich, den Aufwand für das globale Undo und partielle Redo zu optimieren. Dieser Aufwand hängt sehr vom Zustand der materialisierten DB nach einem Medien- oder Systemfehler ab, dieser wiederum davon, ob bestimmte Präventionsmaßnahmen durchgeführt werden. Wir besprechen zunächst derartige Präventionsmaßnahmen und Einflußfaktoren. Bei diesen Maßnahmen muß man abwägen, welchen Gewinn sie ggf. bei einem (relativ seltenen) Systemfehler bringen und welchen Aufwand sie im Normalbetrieb verursachen.

Direkte vs. indirekte Seitenadressierung: Hier geht es um die Frage, ob einer Seite immer genau ein Block zugeordnet ist (direkte Seitenallokation) oder mehrere Blöcke alternativ zugeordnet sein können (indirekte Seitenallokation).

Eine indirekte Seitenallokation ist Voraussetzung für das vorsichtige Ändern von Blöcken, denn dann müssen während einer Änderung die alte und neue Version der Seite zugleich in der physischen Datenbank existieren.

Bei einer direkten Seitenallokation sind die Änderungen überschreibend (*update in place*), es besteht hier kein Unterschied zwischen materialisierter und physischer Datenbank.

Nachteil der indirekten Seitenadressierung ist der erhöhte Aufwand; für die Umschaltung auf die neue Version ist ein zusätzlicher Plattenzugriff erforderlich.

Atomares Ändern mehrerer Seiten: Sofern eine Aktion (z.B. das Löschen eines Tupels) mehrere Seiten verändert, stellt sich die Frage, ob diese Änderungen atomar materialisiert werden.

Wenn solche Änderungen nicht atomar erfolgen, ist der Zustand der materialisierten Datenbank nach einem Systemfehler unvorhersehbar, es können physische Inkonsistenzen auftreten.

Atomares Ändern ist leider praktisch nur bei einer indirekten Seitenadressierung implementierbar, die hohen Aufwand verursachen kann.

Materialisieren unsicherer Änderungen: Das Problem ist hier, daß mehrere Transaktionen u.U. mehr Seiten verändern, als Puffer vorhanden sind. In diesem Fall müssen veränderte Blöcke schon vor EOT in die physische Datenbank zurückgeschrieben werden (analog zum Paging in Betriebssystemen). Beim *update in place* wird dann sogar die materialisierte Datenbank verändert.

Nachteil des Materialisierens unsicherer Änderungen sind hohe Undo-Kosten. Vermeiden kann man dies durch größere Puffer, was heute meist kein Problem ist.

Materialisieren aller Änderungen bei Commit: Die Frage ist hier, ob alle veränderten Seiten im Rahmen der Commit-Verarbeitung materialisiert werden oder nicht.

Falls ja, kann das globale Redo nach einem Systemfehler komplett entfallen! Leider verursacht das sofortige Materialisieren aller veränderten Seiten einen hohen Aufwand im laufenden Betrieb und ist daher nicht immer möglich.

Zusammenfassend läßt sich sagen, daß das Materialisieren aller Änderungen bei Commit und das atomare Ändern mehrerer Seiten zwar sehr attraktiv bzgl. des erreichten DB-Zustands sind, aber oft an zu hohem Aufwand scheitern.

2.7.2 Globales Undo

Das globale Undo findet beim erneuten Hochfahren des System nach einem Systemfehler statt; wir gehen davon aus, daß der normale Betrieb zunächst noch gesperrt ist, das globale Undo also die materialisierte Datenbank exklusiv verfügbar hat.

Ziel ist im Prinzip ein Undo aller abgebrochenen Transaktionen. Man könnte dies dadurch erreichen, daß man die Menge der abgebrochenen Transaktionen bestimmt und für jede einzeln ein Undo durchführt. Dies wäre aber ungeschickt, weil man dann für jede Transaktion den Log durchsuchen müßte. Stattdessen ist ein gemeinsames Rollback aller abgebrochenen Transaktionen effizienter. Analog zum Transaktion-Rollback durchläuft man den Log von hinten aus rückwärts und führt ein Aktions-Undo für jede Aktion einer abgebrochenen Transaktionen durch.

Die Menge der abgebrochenen Transaktionen ist anhand des Logs bestimmbar: es handelt sich um die Transaktionen, für die ein BOT-, aber kein Commit- oder Abort-Eintrag im Log vorhanden ist. Wenn man den Log von hinten durchsucht, findet man einen Aktionseintrag, ohne vorher einen Commit-Eintrag gefunden zu haben.

Ohne vorbereitende Maßnahmen muß man den ganzen (!) Log durchlaufen, denn es könnte ganz vorne noch eine Transaktion stehen, die aus irgendwelchen Gründen nicht beendet worden ist. Dies ist natürlich unerwünscht, denn dies dauert lange und es ist sehr unwahrscheinlich, daß sich vorne noch “liegengebliebene” Transaktionen befinden. Abhilfe schaffen hier Undo-Checkpoints.

Ein **Undo-Checkpoint** ist ein spezieller Log-Eintrag, der die Identifikationen der derzeit aktiven Transaktionen enthält. Sobald beim Rückwärtsdurchlauf ein Undo-Checkpoint gefunden wird, ist die Restmenge der abgebrochenen Transaktionen bekannt. Der Rückwärts-

durchlauf kann beendet werden, sobald alle zugehörigen BOT-Einträge gefunden worden sind.

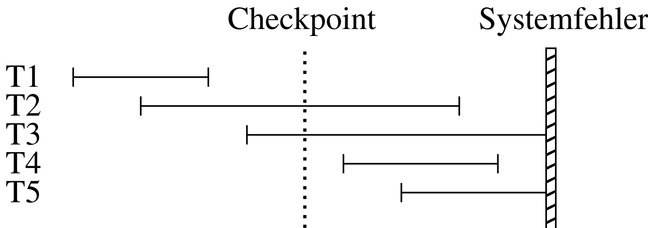


Abbildung 2.3: Beispiel eines Checkpoints

Erzeugt werden sollten Undo-Checkpoints etwa im Abstand von einigen Minuten oder alternativ nach jeweils 100 gestarteten Transaktionen.

Bei dem Beispiel in Bild 2.3 würde der Checkpoint festhalten, daß zu diesem Zeitpunkt die Transaktionen T2 und T3 nicht beendet waren.

Systemfehler beim globalen Undo. Während der Ausführung eines globalen Undo kann natürlich erneut ein Systemfehler eintreten, so daß das globale Undo unterbrochen würde und in der materialisierte Datenbank nur ein undefinierter Teil der Änderungen rückgängig gemacht wird.

Beim Zustands-Logging kann nun einfach das globale Undo komplett wiederholt werden; dies liegt daran, daß das Undo einer Aktion hier durch Schreiben des früheren Werts implementiert wird und daß bei dieser Implementierung des Aktions-Undo eine mehrfache Ausführung letztlich den gleichen Effekt erzeugt wie eine einmalige Ausführung. Dies bezeichnet man auch als **Idempotenzeigenschaft**.

Beim Transitions-Logging hat das Aktions-Undo die Idempotenzeigenschaft nicht, deshalb darf das globale Undo nach einer Unterbrechung nicht einfach wiederholt werden. Stattdessen ist eine aufwendigere Implementierung des globalen Undo erforderlich, bei der sichergestellt wird, daß jedes Aktions-Undo nur genau einmal ausgeführt wird.

2.7.3 Partielles Redo

Das partielle Redo findet nach dem globalen Undo statt. Es müssen noch die Änderungen nachgeholt werden, die nicht in der materialisierten Datenbank enthalten sind und die von Transaktionen stammen, die vor dem Systemfehler beendet worden sind. Durch vorbeugende Maßnahmen (Materialisieren aller Änderungen bei Commit) können solche Änderungen ganz vermieden werden, aber diese vorbeugenden Maßnahmen sind nicht immer möglich.

Der Grund, warum man nicht alle Änderungen sofort materialisiert, liegt im Zeitverlust für die Plattenzugriffe, d.h. manche veränderten Seiten können erst einige Zeit nach dem Commit auf Platte zurückgeschrieben werden. Zu einem bestimmten Zeitpunkt kann eine Menge von Seiten darauf warten, zurückgeschrieben zu werden, und es ist zu entscheiden, welche davon als nächste zurückgeschrieben werden soll. Diese Frage stellt sich übrigens auch bei der Pufferverwaltung von Dateien; die in Betriebssystemen üblichen Strategien – die angewandt werden, wenn man die Segmente als Dateien realisiert und die Segmentverwaltung so dem Betriebssystem überläßt – laufen darauf hinaus, stark frequentierte Seiten nicht zurückzuschreiben, weil deren Inhalt ja doch bald wieder verändert wird. Im Sinne des partiellen Redo sind derartige Strategien gerade falsch: derartige Seiten enthalten Änderungen von vielen, u.U. lange zurückliegenden Transaktionen. Sinnvoller ist es hier, mit erster Priorität Seiten zurückzuschreiben, die Änderungen abgeschlossener Transaktionen enthalten.

Nach einem Systemfehler hat man ohne vorbereitende Maßnahmen keinerlei Daten darüber, welche Änderungen nicht materialisiert worden sind, d.h. man müßte auf Verdacht *alle* vollständigen Transaktionen, die im Log vorkommen, wiederholen. Aus Aufwandsgründen kommen auch keine Verfahren in Frage, bei denen man die komplette Datenbank oder den kompletten Log durchsuchen müßte. Abhilfe schaffen hier Redo-Checkpoints. Bei einem **Redo-Checkpoint** werden alle veränderten Seiten zurückgeschrieben, und im Log wird ein entsprechender Eintrag geschrieben. Der komplette vor dem Redo-Checkpoint liegende Teil des Logs braucht daher beim partiellen Redo nicht mehr

berücksichtigt zu werden.

Wenn wir in Bild 2.3 annehmen, daß der eingezeichnete Checkpoint zugleich ein Redo-Checkpoint ist, dann würden im Rahmen des Redo-Checkpoints die bisherigen Änderungen von T2 und T3 materialisiert werden. Infolge des Rückwärtsdurchlaufs durch den Log im Rahmen des vorherigen globalen Undo ist schon bekannt, daß

- T2 und T4 erfolgreich beendet worden sind; diese Transaktionen werden also beim partiellen Redo berücksichtigt;
- T3 und T5 nicht erfolgreich beendet worden sind; diese Transaktionen werden beim partiellen Redo nicht berücksichtigt.

Systemfehler beim partiellen Redo. Während der Ausführung eines partiellen Redo kann erneut ein Systemfehler eintreten, so daß das partielle Redo unterbrochen würde und in der materialisierten Datenbank nur ein undefinierter Teil der Änderungen nachgeholt sind. Hier gelten im Prinzip die gleichen Überlegungen wie oben schon im Zusammenhang mit Systemfehlern beim globalen Undo. Sofern die Implementierung des Redo einer Aktion idempotent ist, kann das partielle Redo einfach wiederholt werden.

2.7.4 Änderungsdateien

Diese Technik ist eine Präventiv-Technik gegen Systemfehler. Nach Neuladen des Systems ist die Datenbank *sofort* und ohne Recovery-Maßnahmen benutzbar, allerdings nur in einem früher korrekten Zustand (Zielzustandstyp 2, vgl. Abschnitt 2.3.4). Eine saubere Trennung zwischen Arbeitsversion der Datenbank und Recovery-Daten ist bei dieser Technik nicht möglich.

Die Datenbank wird aufgeteilt in eine Hauptdatei und eine Änderungsdatei.

Die **Hauptdatei** enthält einen früher korrekten Datenbankzustand. Insofern ähnelt sie stark einem Backup-Dump. Im Unterschied zu diesem ist sie jedoch direkt zugreifbar auf Platte gespeichert, und es wird auch laufend auf sie zugegriffen, allerdings nur lesend.

Die **Änderungsdatei** hat im wesentlichen den gleichen Inhalt wie ein Vorwärts-Log. Die enthaltenen Änderungen werden jedoch nicht in der Hauptdatei ausgeführt, d.h. es gibt keine Version der Datenbank auf dem aktuellen Stand.

Bei Zugriffen auf die Datenbank muß zunächst der aktuelle Stand für das betreffende Datenbankobjekt rekonstruiert werden. Zuerst wird die Änderungsdatei nach zutreffenden Einträgen durchsucht, danach erst zur Hauptdatei zugegriffen. Der Mehraufwand gegenüber einem Zugriff zu einer Datenbank auf aktuellem Stand kann durch geschickte Wahl der Algorithmen in erträglichem Rahmen gehalten werden (vgl. [SeL76]).

Die Änderungsdatei wird regelmäßig und meist im laufenden Betrieb in die Hauptdatei gemischt. Zeitpunkt und Häufigkeit hängen von einigen Umständen ab, jedoch sind die Mischungen i.d.R. so häufig, daß die Änderungsdatei erheblich kleiner ist als ein Log.

Während des Mischens besteht kein Schutz gegen Systemfehler mehr. Aus diesem Grund werden zwei Kopien der Hauptdatei empfohlen, von denen eine als Backup-Kopie dient. Ferner empfiehlt sich das Prinzip des vorsichtigen Änderns.

2.8 Recovery für Medienfehler

Grundlage aller Recovery-Maßnahmen für Medienfehler sind Dumps. Ein **Dump** (auch *backup copy* oder *image copy*) ist eine Kopie (eines Teils) der Datenbank. Die Erzeugung eines Dumps ist bei sehr großen Datenbanken aufwendig, selbst bei schnellen Platten und Rechnern kann die Dauer für die Erzeugung in der Größenordnung einer Stunde liegen. Ein Dump sollte daher möglichst außerhalb der normalen Betriebsszeiten erzeugt werden, u.a. um den Normalbetrieb nicht zu beeinträchtigen. Bei Systemen, die rund um die Uhr verfügbar sein müssen, kann für die Erzeugung des Dumps nicht einfach die ganze Datenbank gesperrt werden; hier muß durch spezielle Maßnahmen dafür gesorgt werden, daß der Dump überhaupt einen konsistenten Datenbankzustand enthält.

Ein **inkrementeller Dump** ist eine Kopie der seit einem bestimmten Datum veränderten Teile der Datenbank, i.d.R. auf dem Niveau von Seiten. Bezugszeitpunkt ist der Zeitpunkt der Anfertigung des letzten vollständigen oder inkrementellen Dumps.

Rücksetzen, der DB Für das **Rücksetzen der DB** benötigt man nun einen vollständigen Dump und beliebig viele inkrementelle Dumps und geht wie folgt vor:

1. löschen der vorhandenen Arbeitversion
2. laden des vollständigen Dumps
3. einmischen der inkrementellen Dumps in der gleichen Reihenfolge, in der sie erzeugt worden sind

Für das anschließende **globale Redo** benötigt man einen Vorwärts-Log (also ggf. mehrere Archiv-Logs und einen aktiven Log). Zuerst werden die Archiv-Logs in Reihenfolge ihrer Entstehung eingespielt, dann der aktive Log. Beim Einspielen des aktiven Log ist das Vorgehen analog zum partiellen Redo, d.h. es werden nur die vollständigen Transaktionen wiederholt.

Glossar

Änderungsdatei: Aufteilung der Datenbank in eine (nahezu statische) Hauptdatei und eine Änderungsdatei, die Differenzen zwischen aktuellem Stand und Stand gemäß der Hauptdatei enthält

aktiver Log: auf Platte stehender Teil des gesamten Logs, in dem ältere Einträge, die in Archiv-Logs ausgelagert worden sind, fehlen

aktuelle Datenbank: logischer Zustand, der sich aufgrund des Inhalts der persistenten *und* der flüchtigen Medien ergibt; relevant für den normalen Betrieb

Archiv-Log: komprimierter Redo-Log, der ggf. für ein globales Redo benutzt wird

Dump: früher angefertigte Kopie des Zustands der Datenbank oder eines Teils von ihr

Fehler-Atomarität: Eigenschaft einer Transaktion, daß die Folge von Aktionen ganz oder gar nicht ausgeführt wird

Fehlerkompensation: Behebung von Schäden durch Anwendung einer kompensierenden Operation auf die betroffenen Objekte

globales Redo: Recovery-Grundfunktion, die alle Änderungen in der Datenbank, die seit dem Erzeugen der letzten Sicherungskopie stattgefunden haben, nachholt

globales Undo: Recovery-Grundfunktion, die die Wirkung aller durch einen Systemfehler unterbrochenen Transaktionen rückgängig macht

Idempotenz: Eigenschaft von Operationen (hier speziell Aktions-Undos oder -Redos), bei mehrfacher Ausführung den gleichen Effekt zu erzeugen wie einmaliger Ausführung

inkrementeller Dump: Dump, der nur die seit einem bestimmten Datum veränderten Teile der Datenbank enthält

Log: Aufzeichnung der Änderungen in der Datenbank für Recovery-Zwecke

materialisierte Datenbank: logischer Zustand, der sich aus dem Inhalt der persistenten Medien ergibt (die physisch konsistent sein müssen); relevant für den Neustart nach einem Systemfehler

Medienfehler: Gruppe von Schäden, bei denen ein Speichermedium so stark beschädigt ist, daß der Inhalt komplett unbrauchbar geworden und vollständig aus den Recovery-Daten rekonstruiert werden muß

Neuladen: Recovery-Grundfunktion, die eine früher erzeugte Sicherungskopie als Datenbankinhalt installiert

partielles Redo: Recovery-Grundfunktion, die die Änderungen der Transaktionen nachholt, die vor dem Systemfehler beendet wurden und deren Wirkung nicht in der materialisierten Datenbank enthalten ist

physisch konsistent: eine Datenbank ist physisch konsistent, wenn sich die internen Speicherungsstrukturen in einem ordnungsgemäßen Zustand befinden; auf einem physisch inkonsistenten Zustand kann der Laufzeitkern i.a. nicht mehr korrekt arbeiten

physische Datenbank: physischer Zustand, der sich allein aufgrund des Inhalts der persistenten Medien ergibt; relevant für Rettungsprogramme nach Medienfehlern

Recovery: Wiederherstellung der Datenbank nach einer Beschädigung infolge einer Störung

Redo einer Aktion: Recovery-Grundfunktion, die die Wirkung einer Aktion nachholt

Redo-Checkpoint: spezieller Log-Eintrag, der anzeigt, daß zu diesem Zeitpunkt alle veränderten Seiten zurückgeschrieben worden sind; erlaubt eine effizientere Implementierung des partiellen Redos

Redo-Log: Log, der nur Daten für das Vorwärts-Recovery enthält

Rollback: Recovery-Grundfunktion, die die bisherigen Wirkungen einer Transaktion aufhebt

Rückwärts-Recovery (*backward recovery*): Recovery-Verfahren, bei denen die beschädigten Teile der Datenbank in einen früheren Zustand zurückversetzt werden, möglichst in den unmittelbar vor der Störung

Schaden: Veränderung des Inhalts der Datenbank infolge einer Störung

Störung: Ereignis, bei dem irgendwelche Teile des Systems nicht erwartungsgemäß bzw. gemäß ihrer Spezifikation arbeiten

Systemfehler: Absturz des DBMS-Kerns bzw. Gruppe von Schäden, die dadurch verursacht werden; die Schäden bestehen insb. im Verlust der Daten in den flüchtigen Speichern (Puffern), aber nicht darin, daß die Datenbank physisch inkonsistent wird

Transaktionsfehler: Abbruch einer Transaktion bzw. Gruppe von Schäden, die hierdurch verursacht werden; die Schäden bestehen in den veränderten Objekten, die potentiell einen logisch inkonsistenten Zustand bilden

Transitions-Logging: Logging-Verfahren, bei dem in den Logeinträgen Zustandsübergänge protokolliert werden

Undo einer Aktion: Recovery-Grundfunktion, die die Wirkung einer Aktion rückgängig macht

Undo-Checkpoint: spezieller Log-Eintrag, der die Identifikationen der zur Zeit aktiven Transaktionen enthält; erlaubt eine effizientere Implementierung des globalen Undos

Undo-Log: Log, der nur Daten für das Rückwärts-Recovery enthält

verzögertes Schreiben (*deferred update*): alle Schreibaktionen (Änderungen) einer Transaktion werden erst im Rahmen des Commits durchgeführt

vorsichtiges Ändern (*careful replacement*): Änderungen (in der physischen Datenbank) werden nicht durchgeführt, indem Sektoren überschrieben werden, sondern indem zunächst eine neue Version aller veränderten Seiten eingefügt und dann auf einmal auf die neuen Versionen umgeschaltet wird

Vorwärts-Recovery (*forward recovery*): Recovery-Verfahren, bei denen Änderungen in der Datenbank nachgeholt werden, z.B. nach Neuladen der Datenbank

Zustands-Logging: Logging-Verfahren, bei dem in den Logeinträgen Zustände vor bzw. nach einer Aktion gespeichert werden

Lehrmodul 3:

Sperrverfahren

Zusammenfassung dieses Lehrmoduls

Sperrverfahren sind die am meisten verbreiteten Concurrency-Control-Verfahren. Wir motivieren zunächst die parallele Ausführung von Transaktionen und den Begriff Serialisierbarkeit. Das simpelste Sperrverfahren ist der wechselseitige Ausschluß. An diesem erläutern wir die Grundprinzipien der Funktion von Sperren. Wir stellen einige weitere Protokolle vor, von denen das allgemeinste das 2-Phasen-Protokoll ist. Von diesem betrachten wir noch die Sonderfälle Sperren bis EOT und Preclaiming. Weiter werden die Isolationsstufen von SQL skizziert.

Vorausgesetzte Lehrmodule:

obligatorisch: – Transaktionen und die Integrität von Datenbanken

Stoffumfang in Vorlesungsdoppelstunden: 1.2

©2006 Udo Kelter

3.1 Serialisierbarkeit

Betriebliche Informationssysteme müssen i.d.R. eine große Anzahl an Benutzern gleichzeitig bedienen. Die Benutzer rufen Transaktionsprogramme auf und führen so einzelne Transaktionen auf der Datenbank durch.

Wenn man diese Transaktionsprogramme ungeschützt auf den gleichen Daten arbeiten läßt, kommt es zu Interferenzen und diversen Fehlern. Das bekannteste Beispiel ist eine verlorene Änderung: zwei Transaktionen lesen das gleiche Datenelement, berechnen einen neuen Wert und schreiben ihren jeweiligen neuen Wert in die Datenbank zurück. Der zuerst geschriebene Wert wird dann überschrieben, d.h. der Effekt der zugehörigen Transaktion geht verloren.

In manchen Fällen kann man das Problem lösen, indem man die parallele Ausführung von Transaktionen komplett vermeidet und sie strikt hintereinander ausführt. Dieses Vorgehen ist indessen bei größeren Systemen unannehmbar. Wenn eine Transaktionsausführung durchschnittlich 0.5 Sekunden dauert, kann man pro Minute maximal 120 Transaktionen ausführen; dieser Durchsatz ist oft zu wenig. Noch gravierender sind die entstehenden Antwortzeiten; wenn viele Benutzer gleichzeitig Transaktionen starten, können leicht Wartezeiten im Bereich von Minuten auftreten; üblicherweise wird eine Obergrenze von ca. 2 Sekunden verlangt.

Es muß also möglich sein, Transaktionen parallel auszuführen, ohne daß Interferenzen auftreten. Bild 3.1 zeigt ein Szenario, in dem zwei Benutzer je eine Transaktion aufrufen. Aus Sicht der Benutzer überlappen die Zeiträume, in denen die Transaktionen ausgeführt werden. Innerhalb der Transaktionen werden einzelne Aktionen aufgerufen. Unter einer **Aktion** verstehen wir hier irgendeinen lesenden oder schreibenden Zugriff auf die Datenbank. Im Beispiel rufen die beiden Transaktionen jeweils eine Folge von Aktionen auf: a11-a12-a13 bzw. a21-a22-a23-a24. Wir nehmen hier zur Vereinfachung an, daß die Aktionen innerhalb des Kerns strikt seriell ausgeführt werden. Aus Sicht des Kerns werden die Aktionsfolgen der einzelnen Transaktionen verzahnt ausgeführt, im Beispiel ist es die Verzahnung a11-a21-a12-a22-a13-a23-a24.

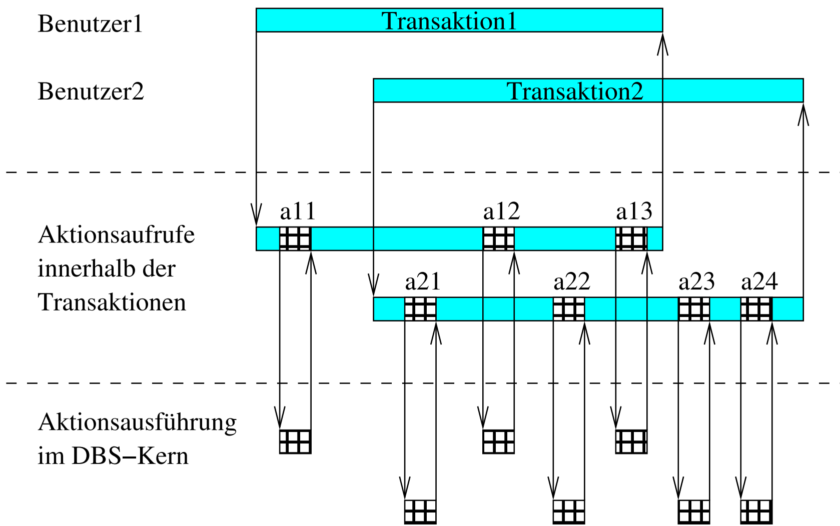


Abbildung 3.1: Parallelität von Transaktionen

Das DBMS steht nun vor der Aufgabe, nur solche Verzahnungen zuzulassen, bei denen keine Interferenzen auftreten. Man spricht hier von der **Nebenläufigkeitssteuerung|seeConcurrency Control** (*Concurrency Control*, Abk.: **CC**) im DBMS.

Informell kann man “Abwesenheit von Interferenzen” so definieren, daß die Transaktionen *scheinbar* eine nach der anderen jeweils auf einen Schlag, sozusagen atomar, ausgeführt worden sind. Eine exakte Definition der “Abwesenheit von Interferenzen” ist alles andere als trivial und Gegenstand eines eigenen Zweigs der Theoretischen Informatik, der Concurrency-Control-Theorie. Das letztliche Resultat dieser Theorie kann wie folgt zusammengefaßt werden: eine Verzahnung verursacht keine Interferenzen, wenn es in ihr für jede Transaktion einen Serialisierungspunkt gibt; dann nennt man die Verzahnung **serialisierbar**. Ein **Serialisierungspunkt** einer Transaktion ist ein Zeitpunkt, zu dem alle Aktionen der Transaktion “konfliktfrei” hingeschoben werden können. Aus Sicht der Benutzer findet die ganze Transaktion scheinbar atomar

auf einen Schlag am Serialisierungspunkt statt.

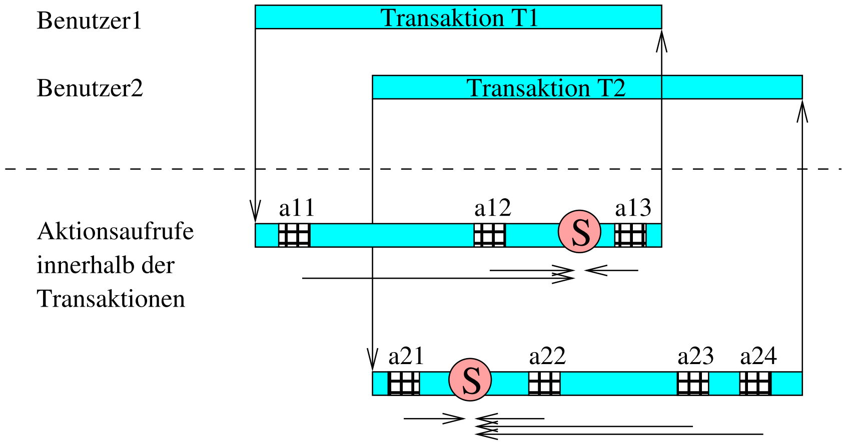


Abbildung 3.2: Serialisierungspunkte von Transaktionen

Bild 3.2 zeigt dies an einem Beispiel: die Serialisierungspunkte der beiden Transaktionen sind durch ein eingerahmtes S gekennzeichnet. Aus Sicht der Benutzer liegt der Serialisierungspunkt der Transaktion T1 hinter dem von T2, d.h. die Wirkung von T2 tritt logisch gesehen zuerst ein. Daß T1 früher angefangen hat und vielleicht sogar als erstes eine Aktion ausgeführt hat, sagt nichts über die logische Reihung aus.

Wenn die Aktionen einer Transaktion zum Serialisierungspunkt verschoben werden müssen, dann kann der Fall auftreten, daß die Reihenfolge von zwei Aktionen, die zu verschiedenen Transaktionen gehören, vertauscht werden muß. In unserem Beispiel muß u.a. die Reihenfolge von a11 und a22 vertauscht werden. Eine solche Reihenfolgevertauschung ist nur zulässig, wenn sie *keinen Einfluß auf die Wirkung der Transaktionen* hat.

Wir gehen i.f. von Aktionen der Form `read(X)` und `write(X)` aus, worin X ein unabhängig les- bzw. schreibbares Datengranulat (Tupel, Objekt, Satz o.ä.) ist. Unter dieser Annahme kann die Reihenfolge von zwei Aktionen ohne Auswirkungen vertauscht werden, wenn

- sie verschiedene Datengranulate betreffen oder
- beide Aktionen lesende Aktionen sind.

In diesem Fall nennen wir die beiden Aktionen **konfliktfrei**.

3.2 On-line-Scheduler

Die Serialisierbarkeit ist zunächst ein Korrektheitskriterium für die verzahnte Ausführung *vollständiger* Transaktionen. Wir können also erst, nachdem alle laufenden Transaktion beendet sind, entscheiden, ob die aufgetretene Verzahnung zulässig war oder nicht. Bei den meisten CC-Verfahren müssen wir aber sofort entscheiden, ob eine Aktion ausgeführt werden kann, wir können nicht warten, bis alle laufenden Transaktion beendet sind. Betrachten wir hierzu Bild 3.3. Jede einzelne Transaktion verursacht eine Sequenz von Aktionsaufrufen.

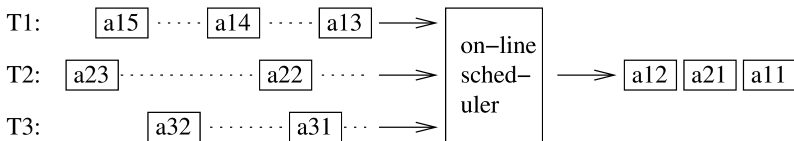


Abbildung 3.3: Parallelität von Transaktionen

Immer dann, wenn eine Transaktion erneut eine Aktion aufruft, muß eine DBMS-Komponente, die wir **Scheduler** nennen, entscheiden, ob diese Aktion sofort oder erst später ausgeführt wird. Diese Entscheidungen bestimmen, wie die einzelnen Sequenzen von Aktionsaufrufen zu einer einzigen Sequenz von effektiven Aktionsausführungen zusammengesetzt werden. Die Entscheidungen können nur auf Informationen über die neuen Aktionen und über die bisher schon durchgeführten Aktionen der Transaktionen basieren, nicht auf zukünftig vielleicht folgenden Aktionsaufrufen, denn der Scheduler kann nicht hellsehen. Wir sprechen daher von einem **On-line-Scheduler**.

Da jede Transaktion jederzeit ein Commit verlangen kann, folgt hieraus, daß die Serialisierbarkeit auch für beliebige Anfangsstücke der

effektiven Verzahnung gelten muß. Die Aufgabe lautet also, für jede Transaktion ständig einen Serialisierungspunkt zu garantieren.

Wenn man annimmt, daß zu einem Zeitpunkt immer nur ein Aktionsaufruf eintrifft und dieser sofort ausgeführt wird, definieren die parallelen Transaktionen eine gemeinsame **Aufrufsequenz**. Die Aufgabe des Schedulers kann darin gesehen werden, diese Aufrufsequenz in eine effektive **Ausführungssequenz** umzuformen.

3.3 Grundlagen der Sperrverfahren

Serialisierungspunkte können durch eine sehr einfache (und relativ grobe) Technik garantiert werden. Alle Objekte, die eine Transaktion benutzt (aber nicht die ganze DB), werden während ihrer gesamten Laufzeit *exklusiv* für sie reserviert. Die Objekte werden somit von den parallelen Transaktionen unter **wechselseitigem Ausschluß** benutzt.

Bei den so entstehenden Ausführungssequenzen ist jeder Zeitpunkt zwischen Beginn und Ende einer Transaktion ein gültiger Serialisierungspunkt für diese Transaktion.

Der wechselseitige Ausschluß kann dadurch erreicht werden, daß jede Transaktion die von ihr benutzten Objekte gleich zu Beginn *sperrt*. Eine andere Transaktion, die zu einem gesperrten Objekt zugreifen will, muß *warten*, bis dieses Objekt wieder freigegeben ist. Damit haben wir bereits ein sehr einfaches Sperrverfahren, bei dem garantiert ist, daß die entstehenden Ausführungssequenzen serialisierbar sind.

Grundprinzip des Sperrens. Zu einem gesperrten Objekt sollen andere Transaktionen als die, die es gesperrt hat, nicht zugreifen. Doch wer hindert sie daran? Um die Beachtung von Sperren durchzusetzen, führen wir folgendes *Grundprinzip* ein:

> *Nur Inhaber von Sperren dürfen zu Datenbankobjekten zugreifen.*

Verantwortlich für die Durchsetzung dieses Grundprinzips ist das DBMS. Das DBMS führt nur dann Zugriffe zu einem Objekt durch, wenn die Transaktion eine Sperre hält.

Sperrmodi. Eine Sperre verleiht dem Inhaber sozusagen das temporäre Recht, Zugriffe durchzuführen. Man klassifiziert die Zugriffe üblicherweise in lesende und schreibende und unterscheidet dazu passend Sperren. Die Art einer Sperre nennt man auch **Sperrmodus**. Die Sperrmodi unterscheiden sich durch folgende Merkmale:

- durch die Zugriffsrechte eines Inhabers und
- durch die Verträglichkeit (Kompatibilität) mit anderen Sperren.

Die Sperren, die in dem oben vorgestellten wechselseitigen Ausschluß benutzt wurden, waren **Schreibsperren** (bzw. **exklusive Sperren**, **X-Sperren**, *exclusive locks*). Ihre Merkmale sind:

- ein Inhaber darf beliebig zu einem Objekt zugreifen.
- Schreibsperren sind mit keiner anderen Sperre verträglich.

Eine andere sehr häufige Art von Sperren sind **Lesesperren** (bzw. *shared locks* oder **S-Sperren**). Ihre Merkmale sind:

- Ein Inhaber darf nur lesend zum gesperrten Objekt zugreifen.
- Eine Lesesperre ist verträglich mit anderen Lesesperren, nicht jedoch mit Schreibsperren.

Die Verträglichkeit verschiedener Arten von Sperren kann übersichtlich durch eine **Verträglichkeitsmatrix** dargestellt werden. Ein + bzw. - zeigt an, ob die beantragte Sperre mit der vorhandenen Sperre verträglich ist oder nicht.

vorhandene	beantragte Sperre:	
Sperre:	S-Sperre	X-Sperre
keine	+	+
S-Sperre	+	-
X-Sperre	-	-

Anforderung und Freigabe von Sperren. Das DBMS prüft bei jedem Zugriffsversuch, ob eine Sperre vorhanden ist, die für diesen Zugriff ausreichende Rechte verleiht. Falls nicht, wird normalerweise *implizit* eine Sperre angefordert, d.h. die Applikation bemerkt – außer einer eventuellen Verzögerung – nichts von dieser Sperrenanforderung.

Am Ende einer Transaktion werden stets automatisch alle Sperren freigegeben, die eine Transaktion noch hält. Diese Automatik ist besonders für den Fall erforderlich, wo eine Transaktion durch Rollback beendet wird.

Die implizite Anforderung und Freigabe von Sperren ist sehr häufig und für die Entwickler von Applikationen am bequemsten, weil man sich überhaupt nicht explizit um Sperren kümmern muß.

Sperren können aber auch *explizit* angefordert oder freigegeben werden. Hierzu muß das DBMS geeignete Kommandos zur Verfügung stellen. Deren Einzelheiten hängen von den Besonderheiten des Datenbankmodells und der Schnittstellen des DBMS ab. In unseren Beispielen benutzen wir die folgenden einfachen Befehle:

- `XLOCK(o)`: Aufforderung zur Einrichtung einer exklusiven Sperre für das Objekt o.
- `SLOCK(o)`: Aufforderung zur Einrichtung einer Lesesperre für o.
- `REL(o)`: Freigabe der Sperre auf o. (Wir können davon ausgehen, daß die Art der Sperre dem DBMS bekannt ist, da eine Transaktion zu einem Zeitpunkt höchstens eine Sperre für o innehat.)

Statt o kann auch eine Liste von Identifikationen von Objekten angegeben werden, die alle gesperrt werden sollen.

In den folgenden Beispielen gehen wir immer davon aus, daß beantragte Sperren bei Verträglichkeit ohne Verzug eingerichtet werden. (Dies ist nicht immer sinnvoll, wie wir später sehen werden.) Bei Unverträglichkeit muß die anfordernde Transaktion auf die Freigabe der vorhandenen Sperren *warten*. Das Warten findet innerhalb der LOCK-Operation statt, d.h. eine Transaktion bemerkt selbst gar nicht, ob und wie lange sie bis zur Zuteilung der Sperre warten muß. Sie kann

sich darauf verlassen, daß nach Beendigung der LOCK-Operation die gewünschten Sperren für sie eingerichtet sind.

Statt beliebig lange zu warten, kann man bei vielen DBMS eine maximale Wartezeit vorgeben. Wird die Sperre nicht innerhalb dieser Frist eingerichtet, wird die explizite oder implizite Sperrenanforderung abgebrochen, und die auslösende Aktion endet mit einem entsprechenden Fehlercode. Das Problem des Wartens wird so letztlich auf die Applikation übertragen.

3.4 Protokolle

Nachdem wir im letzten Abschnitt die Wirkung von Sperren definiert haben, ist noch offen, nach welchen Regeln Sperren von einer Transaktion beantragt und freigegeben werden. Solche Regeln nennt man ein **Protokoll**[7].

Da jedes Protokoll die Grundregel beachten muß, daß Objekte vor ihrer Benutzung gesperrt und am Ende freigegeben werden müssen, unterscheiden sich Protokolle vor allem dadurch, mit welchen Arten von Sperren sie arbeiten und wann Sperren angefordert und freigegeben werden.

Protokolle können auf zwei Arten eingehalten werden:

- “von Hand“: Der Programmierer der Transaktion sorgt selbst dafür, daß Sperren durch explizite Kommandos richtig angefordert und freigegeben werden.
- automatisch: Das DBMS und ggf. der Compiler, mit dem die Transaktionsprogramme übersetzt werden, sorgt automatisch für die richtige Anforderung und Freigabe von Sperren.

Die automatische Realisierung der Protokolle ist bequemer und sicherer. Sofern ein DBMS bzw. Compiler nur ein einziges Protokoll unterstützt, braucht sich der Programmierer überhaupt nicht um

[7] Diese Benennung ist nicht sehr anschaulich, aber allgemein üblich. Aus der Sicht eines Objekts regelt ein Protokoll die Reihenfolge von Sperrungen, Zugriffen verschiedener Art und Freigaben.

das Protokoll zu kümmern, er muß nicht einmal seine Funktionsweise kennen. Sofern mehrere Protokolle unterstützt werden, muß bei der Übersetzung einer Transaktion oder auf andere Weise eines ausgewählt werden. Auf die Verträglichkeit verschiedener Protokolle muß der Programmierer selbst achten.

Die oben vorgestellten Kommandos zur Handhabung von Sperren sind nur bei der "von-Hand"-Realisierung der Protokolle sinnvoll. Bei automatischer Realisierung der Protokolle benötigt ein Programmierer entweder gar keine Sprachmittel oder nur Anweisungen zur Auswahl eines Protokolls.

In den nachfolgenden Beispielen soll vor allem die Wirkung der Protokolle gezeigt werden. Ob die Sperren automatisch oder von Hand angefordert und freigegeben werden, ist dabei unerheblich. Wir gehen davon aus, daß das jeweils diskutierte Protokoll befolgt wurde.

In den folgenden Beispielen benutzen wir die folgenden Notationen:

`r(X)` Lesen des Objekts X

`w(X)` Schreiben des Objekts X

`?...` Sperrenanforderung und nachfolgende Wartezeit

Anforderungen von Sperren werden nicht dargestellt, wenn sie sofort erfüllt werden können.

Wir nehmen i.f. immer die folgende Aufrufsequenz an:

```
T1    r(X)------------------------w(X)
T2          r(Y)-----------w(X)
T3                r(Y)--------------------w(Z)
```

Sofern alle Zugriffe sofort ausgeführt werden, ist die resultierende Ausführungssequenz nicht serialisierbar. T2 müßte nämlich entweder logisch vor oder nach T1 liegen, dazu muß die Aktion `w(X)` von T2 entweder nach vorne verschoben, also mit `r(X)` von T1 vertauscht werden oder nach hinten verschoben, also mit `w(X)` von T1 vertauscht werden. Beides ist nicht konfliktfrei.

3.4.1 Wechselseitiger Ausschluß

Das Protokoll XPE. Das Protokoll, welches dem wechselseitigen Ausschluß zugrunde lag, war:

> **Protokoll XPE**: Zu Beginn jeder Transaktion werden alle Objekte, zu denen die Transaktion zugreifen wird, exklusiv gesperrt. (Am Ende der Transaktion werden alle Objekte automatisch freigegeben.)

In der Abkürzung XPE bedeuten die Zeichen:

X Das Protokoll arbeitet nur mit exklusiven Sperren.

P Alle Sperren werden zu Beginn der Transaktion sozusagen im voraus angefordert (*preclaiming*).

E Alle Sperren werden erst bei EOT (*end of transaction*) freigegeben.

Die Wirkung des Protokolls XPE sei an unserem Standardbeispiel erläutert. Nehmen wir an, alle Transaktionen befolgen das Protokoll XPE. T2 beantragt dann anfangs eine Schreibsperre für Y *und* X. Wir gehen davon aus, daß Y dann sofort schreibgesperrt wird, obwohl die Sperre für X noch nicht eingerichtet werden kann. T3 beantragt anfangs eine Schreibsperre für Y und Z. Das folgende Bild zeigt die entstehende Ausführungssequenz:

```
T1    r(X)---------w(X)
T2         ?...........r(Y)----w(X)
T3             ?...................r(Y)----w(Z)
```

Die entstandene Ausführungssequenz ist seriell, also mit Sicherheit korrekt. In der Tat ist die Aufrufsequenz unnötig stark umgeformt worden, denn Y wird von T2 und T3 nur gelesen.

Das Protokoll SPE. Wir erweitern das Protokoll XPE daher auf Lesesperren und erhalten das

> **Protokoll SPE**: Alle Objekte, zu denen im Laufe der Transaktion zugegriffen wird, werden zu Beginn gesperrt und bei EOT freigegeben. Alle Objekte, die nur gelesen werden, werden lesegesperrt, alle anderen schreibgesperrt.

Die obenstehende Aufrufsequenz führt bei Protokoll SPE zu folgender Ausführungssequenz:

```
T1    r(X)------------w(X)
T2          ?..............r(Y)--------w(X)
T3              r(Y)-------------w(Z)
```

T2 fordert zu Beginn eine Lesesperre für Y und eine Schreibsperre für X an; letztere kann nicht sofort zugeteilt werden.

Die Ausführungssequenz ist wieder serialisierbar, aber nicht mehr seriell. Im Vergleich zum Protokoll XPE muß T3 nicht mehr warten. Das Protokoll SPE erlaubt daher ein höheres Maß an Parallelität.

Endlosblockierungen. Wir hatten oben festgelegt, daß beantragte Sperren bei Verträglichkeit mit den vorhandenen Sperren ohne Verzug eingerichtet werden. Nehmen wir nun an, eine Transaktion T beantragt eine Schreibsperre für ein lesegesperrtes Objekt X, muß also warten. Während der Wartezeit beantragt eine andere Transaktion wieder eine Lesesperre für X und erhält sie, da X lesegesperrt ist. Die Zuteilung der beantragten Schreibsperre wird unendlich lange hinausgezögert, wenn immer wieder neue Transaktionen X in überlappenden Zeiträumen lesesperren. Dies nennt man eine **Endlosblockierung**. T ist endlos blockiert, obwohl das System weiterhin arbeitet, T "verhungert" sozusagen. Blockierungen dieser Art können nur vermieden werden, wenn irgendwann eine beantragte Lesesperre nicht eingerichtet wird, obwohl es sofort möglich wäre.

3.4.2 Protokolle mit höherer Parallelität

Das Protokoll SE. Eine genauere Inspektion der Ausführungssequenz, die bei SPE entstanden war, zeigt, daß auch T1 und T2 zumindest teilweise parallel ausgeführt werden könnten: Der Zugriff von T2 zu Y wird unnötig verzögert, da noch auf die Freigabe von X durch T1 gewartet werden muß. X wird von T2 schon zu Beginn gesperrt, obwohl erst am Ende zugegriffen wird. Der Zugriff zu Y würde nicht unnötig verzögert, wenn X erst unmittelbar vor seiner Benutzung gesperrt würde. Wir erhalten dann das neue Protokoll:

> **Protokoll SE**: wie Protokoll SPE, abweichend von diesem werden jedoch alle Objekte erst unmittelbar vor ihrer ersten Benutzung gesperrt.

Unsere Beispielaufrufsequenz führt bei Protokoll SE zu folgender Ausführungssequenz. T2 wird, nachdem es bereits lief, während der Anforderung der Schreibsperre für X in einen Wartezustand versetzt, der beendet wird, sobald T1 seine Sperre auf X freigibt.

```
T1     r(X)------------------w(X)
T2           r(Y)-------?........w(X)
T3                 r(Y)----------------w(Z)
```

Beim Protokoll SE wurde gegenüber SPE die Parallelität erhöht und die Wartezeit (von T2) verkürzt. Dieser Vorteil ist allerdings mit einem erheblichen Nachteil verbunden: Beim Protokoll SE sind Deadlocks möglich. Die folgende Aufrufsequenz führt beim Protokoll SE zu einem Deadlock:

```
T1     r(X)---------w(Y)
T2           r(Y)---------w(X)
```

Nachdem T1 und T2 jeweils ihren ersten Zugriff ohne Verzögerung ausführen konnten, warten sie später beide darauf, daß der andere ein Objekt freigibt. Auf das Deadlock-Problem und seine Lösungen kommen wir später zurück. Festzuhalten ist, daß beim Protokoll SE spezielle Vorsorgemaßnahmen gegen Deadlocks nötig sind. Der Aufwand hierfür kann durchaus den Leistungsgewinn durch höhere Parallelität gegenüber dem Protokoll SPE übersteigen.

Der Deadlock im vorstehenden Beispiel würde nicht eintreten, wenn die Sperren nicht “unnötig lange“, d.h. bis zum Commit, gehalten würden, also nach der letzten Benutzung eines Objekts freigegeben würden. Das so entstehende Protokoll würde allerdings die Korrektheit der entstehenden Verzahnungen nicht mehr garantieren! So würde die obige Aufrufsequenz ohne Verzögerungen ausgeführt; die identische Ausführungssequenz ist aber nicht serialisierbar.

3.4.3 Das Zwei-Phasen-Protokoll

Die Grundform. Die Zeitdauer, während der ein Objekt gesperrt ist, soll natürlich so kurz wie möglich sein. Das letzte Beispiel zeigt, daß es allerdings nicht ausreicht, ein Objekt nur für die Dauer seiner Benutzung zu sperren. Bei den anderen bisher vorgestellten Protokollen wurde die Sperrzeit bis zum Commit oder zum Beginn der Transaktion verlängert, was u.U. zu lang ist. Das folgende Protokoll vermeidet eine solche starre Festlegung und garantiert dennoch die Serialisierbarkeit:

> **2-Phasen-Protokoll (2PL, *2-phase locking*)**: Wenn eine Sperre freigegeben worden ist, werden keine neuen Sperren mehr angefordert.

Die Grundregel des Sperrens legt ohnehin fest, daß ein Objekt vor seiner ersten Benutzung gesperrt und nach seiner letzten Benutzung und spätestens bei Commit freigegeben werden muß. Ferner nehmen wir an, daß Objekte, die nur gelesen werden, lesegesperrt werden, und daß alle anderen Objekte schreibgesperrt werden.

Eine Transaktion, die ihre Sperren unter Beachtung des 2-Phasen-Protokolls anfordert und freigibt, heißt auch **2-Phasen-gesperrt** (**two-phase-locked**).

Die beiden typischen Phasen, durch die die Bezeichnung für dieses Protokoll motiviert ist, sind:

1. **Wachstumsphase**: Die Zeit vom Beginn der Transaktion bis zur letzten Anforderung einer Sperre. In dieser Zeit werden keine Sperren freigegeben.
2. **Schrumpfungsphase**: Die Zeit von der ersten Freigabe einer Sperre bis Commit. In dieser Zeit werden keine Sperren mehr angefordert.

In der Zeit zwischen den beiden Phasen werden Sperren weder angefordert noch freigegeben; sofern sie überhaupt interessiert, kann man sie **Verarbeitungsphase** nennen.

Die Zahl der gehaltenen Sperren hat beim 2-Phasen-Protokoll den in Bild 3.4 gezeigten typischen Verlauf. Dieses Bild soll natürlich *nicht* andeuten, daß Sperren in umgekehrter Reihenfolge angefordert und freigegeben werden müssen.

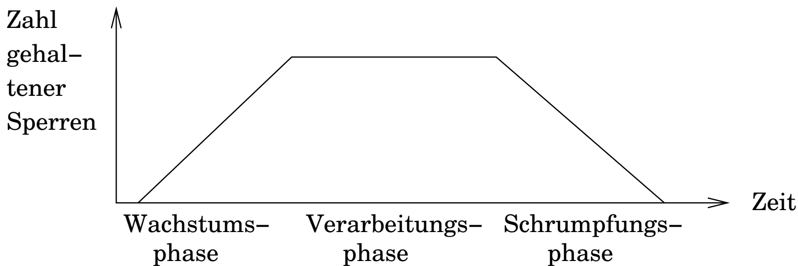


Abbildung 3.4: Zwei-Phasen-Protokoll

Korrektheit des 2-Phasen-Protokolls. Bei 2-Phasen-gesperrten Transaktionen sind alle Ausführungssequenzen serialisierbar. Jeder Zeitpunkt in der Verarbeitungsphase ist nämlich ein Serialisierungspunkt für diese Transaktion. Dies ergibt sich daraus, daß jedes Objekt

von seiner ersten bis zu seiner letzten Benutzung und während der gesamten Verarbeitungsphase ununterbrochen gesperrt ist. Jeder Zugriff kann also konfliktfrei in die Verarbeitungsphase verschoben werden.

Die Korrektheit des 2-Phasen-Protokolls ist insofern wichtig, als alle bisher vorgestellten Protokolle Sonderfälle des 2-Phasen-Protokolls sind und damit auch ihre Korrektheit gezeigt ist.

Bei den bisherigen Protokollen wurde im Vergleich zum 2-Phasen-Protokoll die Sperrzeit der Objekte in eine oder beide Richtungen maximal verlängert. Betrachten wir beide Optionen genauer:

3.4.3.1 Sperren bis EOT

Alle Sperren werden bis EOT gehalten. Explizit freigegeben wird keine Sperre, da bei Commit bzw. Rollback automatisch alle vorhandenen Sperren freigegeben werden. (Durch die Regel: "keine Sperre freigeben" ist das 2-Phasen-Protokoll bereits erfüllt!) Die oben gezeigte Phasenkurve bekommt eine senkrecht absteigende Flanke:

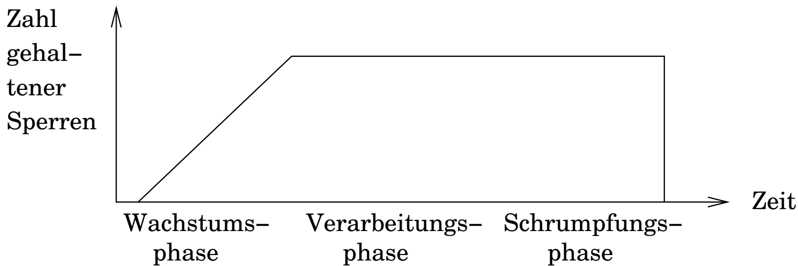


Abbildung 3.5: Sperren bis EOT beim Zwei-Phasen-Protokoll

Nachteil des Sperrens bis EOT ist die gegenüber der Grundform des 2-Phasen-Protokolls verlängerte Sperrzeit, die im Endeffekt zu einer geringfügigen Verminderung der Parallelität und damit zu einer Verlängerung der mittleren Wartezeit einer Transaktion führt. Außer bei sehr langen Transaktionen ist dieser Nachteil jedoch unbedeutend.

Ein Vorteil ist eine Verringerung des Verwaltungsaufwandes: Die Abarbeitung vieler einzelner Freigaben von Sperren ist i.a. aufwendiger

als die automatische Freigabe aller Sperren auf einen Schlag.

Ein entscheidender Vorteil liegt darin, daß keine Fortpflanzung von Rollback auftreten kann. Wenn man eine Schreibsperre vor EOT freigibt, dann enthält das entsprechende Objekt einen **unsicheren Wert**: möglicherweise wird die Transaktion später zurückgesetzt, und dann wird dieser Wert wieder zurückgesetzt. Wenn eine andere Transaktion den unsicheren Wert liest, muß sie bei einem Rollback der ersten Transaktion auch zurückgesetzt werden; dies nennt man eine **Rollback-Fortpflanzung**. Es sind sogar Ketten von Rollback-Fortpflanzungen denkbar. Rollback-Fortpflanzungen sind normalerweise aus vielen Gründen völlig inakzeptabel, d.h. zumindest Schreibsperren müssen schon deshalb bis EOT gehalten werden.

Insgesamt ist das Sperren bis EOT sehr zu empfehlen. Wenn überhaupt, dann sollten höchstens Lesesperren vor EOT freigegeben werden.

3.4.3.2 Preclaiming

Bei dieser Technik werden alle Objekte schon zu Beginn der Transaktion gesperrt. Entscheidend ist, daß alle Objekte auf einmal gesperrt werden. Die Phasenkurve bekommt eine senkrechte "steigende" Flanke:

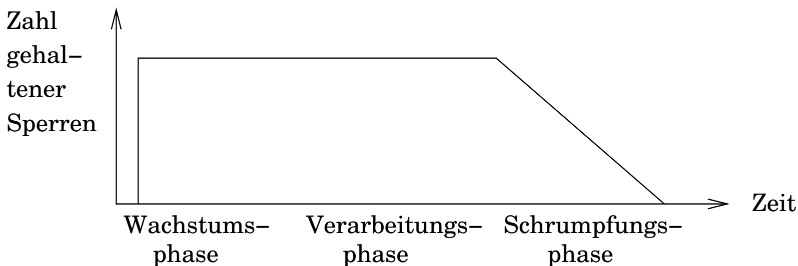


Abbildung 3.6: Preclaiming beim Zwei-Phasen-Protokoll

Nachteil des Preclaiming ist (wie beim Sperren bis EOT) die Verlängerung der Sperrzeit, die aber höchstens bei langen Transaktionen ins Gewicht fällt.

Sehr vorteilhaft beim Preclaiming ist, daß kein Deadlock auftre-

ten kann. Deadlocks sind im Prinzip zyklische Wartesituationen, die durch das DBMS mittels Rollback und Neustart einer der beteiligten Transaktionen aufgelöst werden müssen. Um zu erkennen, ob ein Deadlock vorliegt, müssen aufwendige Zyklustests durchgeführt werden. Der Aufwand hierfür ist so hoch, daß in vielen DBMS eine Transaktion einfach nach Überschreiten einer maximalen Wartezeit auf Verdacht zurückgesetzt und neu gestartet wird.

Für das Preclaiming muß spätestens beim ersten Zugriff in einer Transaktion bekannt sein, zu welchen Objekten während der gesamten Transaktion zugegriffen werden wird. Die Objektmenge muß von der Transaktion selbst herausgefunden und durch eine explizite Sperr-Anweisung dem DBMS bekanntgegeben werden. Leider ist diese Vorausbestimmung der Objektmenge nicht immer möglich. Dies kann verschiedene Ursachen haben:

- Zugriffe können datenabhängig sein, d.h. die bei früheren Zugriffen gelesenen Werte bestimmen, zu welchen Objekten später zugegriffen werden wird.
- Die Objekte werden sukzessive durch Navigieren erreicht, wobei Sperren auf den vorher besuchten Objekten angefordert werden.

In diesen Fällen muß man, sofern man am Preclaiming festhalten will, eine Obermenge der tatsächlich benötigten Objekte sperren, z.B. eine ganze Relation. (Diese Sperrung läßt sich mit Hilfe besonderer Techniken effizient durchführen.) Die Zahl der gesperrten Objekte wird hierdurch oft um Größenordnungen erhöht. Deshalb und wegen der zusätzlichen Verlängerung der Sperrzeit kann die Parallelität erheblich reduziert werden, so daß das Preclaiming eher nachteilig wird.

Insgesamt führen die Probleme dazu, daß man das Preclaiming nicht generell voraussetzen kann. Mechanismen zur Deadlock-Behandlung müssen daher im DBMS auf jeden Fall vorhanden sein.

3.5 Isolationsstufen

Die Serialisierbarkeit ist sozusagen der perfekte Schutz paralleler Transaktionen untereinander. Diese Perfektion hat ihren Preis, nämlich Wartezeiten und schlechteren Durchsatz. Wenn beispielsweise in einem Informationssystem regelmäßig komplexe Abfragen auftreten, für die ganze Relationen gelesen werden müssen, und wenn diese Abfragen serialisierbar ausgeführt werden sollen, können währenddessen keine Daten eingegeben oder verändert werden. Wenn die Abfragen länger dauern (z.B. 30 Sekunden), ist das fast so schlimm wie eine Betriebsstörung und kann je nach Umständen inakzeptabel sein. In diesem Fall muß die Serialisierbarkeit zugunsten einer besseren Performanz des Gesamtsystems geopfert werden. SQL bietet hierzu mehrere Isolationsstufen an, die pro Transaktion individuell gewählt werden kann; im einzelnen:

`read uncommitted`: Transaktionen mit dieser Isolationsstufe müssen Lese-Transaktionen sein[8]; sie werden überhaupt nicht vor parallelen Transaktionen geschützt. Daher brauchen keine Sperren gesetzt zu werden, es werden auch keine anderen Transaktionen durch die (Lese-) Sperren gebremst, und es entsteht kein Aufwand zur Verwaltung der Sperren. Der Name der Isolationsstufe rührt daher, daß sogar unsichere Werte gelesen werden.

Im Prinzip können hier beliebige Fehler auftreten, in der Praxis können je nach Umständen die Fehler aber selten und ihre Effekte vernachlässigbar sein, z.B. bei unkritischen statistischen Auswertungen.

`read committed`: Solche Transaktionen können keine unsicheren Werte lesen, d.h. ein entsprechender Leseversuch würde zum Warten auf das Commit der anderen Transaktion führen.

Es werden aber keine Lesesperren für gelesene Werte angefordert. Wenn der gleiche Datenwert zweimal innerhalb der Transaktion gelesen wird, kann er zwischendurch von einer ande-

---

[8] SQL erlaubt es, eine Transaktion als `read only` anzugeben; sie kann dann nicht mehr schreiben.

ren Transaktion verändert worden sein. Dies bezeichnet man als nichtwiederholbares Lesen[9].

`repeatable read`: Auf dieser Stufe wird eine Lesesperre für gelesene Tupel angefordert und bis EOT gehalten, das Problem des nicht wiederholbaren Lesens kann nicht mehr auftreten. Es können aber Phantome auftreten (Detail s.u.)

`serializable`: Diese Stufe garantiert die Serialisierbarkeit. Phantome können hier nicht auftreten.

In allen Fällen werden für erzeugte oder veränderte Tupel Schreibsperren angefordert und bis EOT gehalten.

Die Isolationsstufe `repeatable read` garantiert überraschenderweise nicht, daß eine SQL-Abfrage, die zweimal unverändert ausgeführt wird, jedesmal das gleiche Ergebnis liefert. Die bei der ersten Ausführung selektierten Tupel werden zwar alle einzeln lesegesperrt und sind daher unverändert im Ergebnis der zweiten Ausführung enthalten[10], eine andere Transaktion könnte aber inzwischen weitere Tupel erzeugt haben, die die Selektionskriterien der Abfrage erfüllen. Derartige Tupel nennt man **Phantome**, weil sie bei der ersten Ausführung noch nicht existierten und bei der zweiten Ausführung sozusagen aus dem Nichts auftauchen. Phantome können ebenfalls zu Parallelitätsanomalien führen.

Phantome bzgl. einer bestimmten Abfrage in einer Transaktion zu verhindern bedeutet, für alle Relationen, die in der Abfrage vorkommen, die Bereiche gegen Veränderungen zu sperren, die für das Abfrageergebnis relevant sind. Änderungs- und Löschoperationen sind kein Problem, denn sie betreffen existierende Tupel und werden schon durch die Lesesperren an diesen Tupeln verhindert. Zusätzlich verhindert werden müssen nur Einfügungen.

Die betroffenen Bereiche sind gerade durch die Selektionsprädikate der Abfrage gegeben und können sehr groß sein. Die resultierenden Blockierungen können daher sehr störend sein.

[9] Der Ausdruck ist nicht wörtlich zu nehmen. Die Leseoperation kann natürlich wiederholt werden. Allerdings wird nicht wiederholt der gleiche Wert zurückgeliefert.

[10] Wir unterstellen hier eine Abfrage ohne Aggregation.

Glossar

2-Phasen-Protokoll (*2-phase locking protocol*): Sperrprotokoll mit der Hauptregel, daß keine neuen Sperren mehr angefordert werden, sobald irgendeine Sperre freigegeben worden ist

Concurrency Control: Verhinderung von Interferenzen zwischen parallelen Transaktionen oder Behebung unzulässiger Effekte eingetretener Interferenz

Deadlock (bzw. Verklemmung): Wartezyklus bestehend aus einer Menge von Prozessen, von denen jeder wenigstens eine Sperre hält, auf deren Freigabe der "nächste" Prozeß wartet, und wenigstens eine zweite Sperre anfordert, die nicht zugeteilt werden kann, weil das betroffene Objekt für den "vorigen" Prozeß gesperrt ist

Isolationsstufen: Ausführungsvarianten von Folgen von Aktionen, bei denen mehr oder weniger umfangreich auf die Transaktionseigenschaften verzichtet wird (zugunsten besserer Performance)

kompatibel: Synonym zu verträglich

konfliktfrei: zwei Aktionen sind konfliktfrei, wenn ihre Reihenfolge ohne äußerlich sichtbare Auswirkungen vertauscht werden kann, also wenn sie verschiedene Datengranulate betreffen oder beide Aktionen nur lesen

Lesesperre: Sperre mit Sperrmodus "Lesen", d.h. dem Inhaber sind nur lesende Zugriffe erlaubt

On-line-Scheduler: Komponente des DBMS-Kerns, der ankommende Aktionsaufrufe entgegennimmt und entscheidet, ob sie sofort oder später ausgeführt werden; on-line drückt aus, daß sofort entschieden wird und nicht auf das Ende der involvierten Transaktionen gewartet wird und, wenn möglich, aufgerufene Aktionen sofort ausgeführt werden

Preclaiming: Anforderung *aller* Sperren, die im Verlauf einer Transaktion benötigt werden, zu Beginn der Transaktion auf einen Schlag

Protokoll: im Kontext von Sperrverfahren: Menge von Regeln, wann Sperren in welchem Modus angefordert bzw. freigegeben werden

Rollback-Fortpflanzung: tritt ein, wenn eine Transaktion T1 eine Schreibsperre vor Transaktionsende freigibt, eine Transaktion T2 einen unsicheren Wert aus dem betroffenen Objekt liest, T1 zurückgesetzt wird und deshalb auch T2 zurückgesetzt werden muß

Schreibsperre: Sperre mit Sperrmodus "Schreiben", d.h. dem Inhaber sind beliebige Zugriffe erlaubt

serialisierbar: eine verzahnte Ausführung mehrere Transaktionen ist serialisierbar, wenn es für jede Transaktion einen Serialisierungspunkt gibt, zu dem alle Aktionen der Transaktion "konfliktfrei" hingeschoben werden können

Sperre: eine Sperre bezieht sich auf ein sperrbares Objekt und hat einen Prozeß als Besitzer; beruht auf dem Konzept, daß ein Prozeß nur dann zu einem zu sperrbaren Objekt zugreifen darf, wenn er Inhaber einer Sperre für dieses Objekt ist; wenn ein Objekt gesperrt ist, können keine weiteren Sperren eingerichtet werden, die mit den vorhandenen Sperren inkompatibel sind

Sperrmodus: definiert die mit einer Sperre verbundenen Rechte des Inhabers (d.h. die Menge der erlaubten Operationen auf dem Objekt)

Transaktion: Folge von Datenbankzugriffen (Aktionen), die einen konsistenten Datenbankzustand in einen neuen konsistenten Datenbankzustand überführt; Transaktionsausführungen haben folgende Eigenschaften (sofern nicht eine reduzierte Isolationsstufe gewählt wird): Fehler-Atomarität, Dauerhaftigkeit, Serialisierbarkeit, endliche Ausführungszeit

Transaktion: Folge von Datenbankzugriffen (Aktionen), die mit den Transaktionseigenschaften ausgeführt wird (oder Programm, das eine solche Folge von Aktionen auslöst) bei denen mehr oder weniger umfangreich auf die Transaktionseigenschaften verzichtet wird (zugunsten besserer Performance)

unsicherer Wert: Datenwert, der von einer nicht beendeten Transaktion geschrieben worden ist und bei dem jederzeit der Fall eintreten kann, daß er auf einen früheren Zustand zurückgesetzt wird, weil die Transaktion zurückgesetzt worden ist

verträglich: Beziehung zwischen Sperrmodi; wenn die Sperrmodi A bzw. B verträglich sind, können zwei Sperren mit diesen Modi gleichzeitig an einem Objekt vorhanden sein

Lehrmodul 4:

Sperrverfahren für Hierarchien von Sperreinheiten

Zusammenfassung dieses Lehrmoduls

Die grundlegenden Sperrverfahren, insb. das 2-Phasen-Protokoll sind nur anwendbar auf Sperreinheiten, die unabhängig voneinander sind. Vielfach will man mit Sperreinheiten gleichzeitig arbeiten, die verschiedenen Größenklassen angehören und die nicht unabhängig voneinander sind, weil sie Teil voneinander sind. Das Warnsperrenprotokoll erweitert das 2-Phasen-Protokoll dahingehend und bewirkt, daß baumartige oder halbgeordnete Mengen von Sperreinheiten korrekt behandelt werden.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Sperrverfahren
- Transaktionen und die Integrität von Datenbanken

Stoffumfang in Vorlesungsdoppelstunden: 0.8

©2004 Udo Kelter Stand: 11.07.2004

4.1 Variable Granularität

Die grundlegenden Sperrverfahren, insb. das 2-Phasen-Protokoll (s. Lehrmodul 3) sind auf beliebige “Objekte” bzw. sperrbare Einheiten anwendbar. Die Verfahren basieren auf zwei Annahmen bzgl. der Menge der sperrbaren Objekte:

- Jedes Objekt ist während seiner Lebensdauer als eine einfache, unstrukturierte Variable anzusehen. Eine eventuell vorhandene interne Struktur bleibt außer Betracht.
- Die einzelnen Einheiten sind unabhängig voneinander sperrbar bzw. zugreifbar.

Typischerweise verstehen wir unter einem Objekt in einer Datenbank einen Satz, ein Feld oder ein Tupel. Neben diesen kleinen Einheiten, die sich aus dem Datenmodell ergeben, sind auch andere sperrbare Einheiten denkbar und üblich:

- größere Einheiten im Datenmodell, z.B. Relationen;
- größere organisatorische Einheiten, insbesondere sogenannte Areas oder Partitionen (dies sind disjunkte Bereiche, in die eine Datenbank oft unterteilt wird und die mehrere Relationen enthalten können);
- physische Einheiten, insbesondere Speicherseiten bzw. die auf einer Speicherseite befindlichen Tupel.

Statt von der Größe der sperrbaren Einheiten spricht man von deren **Granularität**, also der Feinheit, mit der sie die gesamte Datenbank zerlegen. Die Wahl der Granularität ist wichtig für die Leistung des DBS. Folgende Zusammenhänge sind zu beachten:

- Je größer die Einheiten sind, desto stärker muß beim Sperren auf Einheitsgröße “aufgerundet” werden und desto mehr unbenötigte Teile werden unnütz mitgesperrt. So werden Sperrkonflikte wahrscheinlicher, und die Parallelität der Transaktionsausführungen wird reduziert.

- Je kleiner die Einheiten sind, desto mehr Sperren müssen verwaltet werden.

Bei der Entscheidung sind die speziellen Verhältnisse der vorhandenen Anwendungen zu berücksichtigen. Als Faustregel gilt, daß eine Transaktion nur "wenige" Einheiten sperren sollte, d.h. möglichst deutlich unter 10 Einheiten.

Wenn also eine Transaktion einige Tupel einer relationalen Datenbank liest oder verändert, dann sind Tupel die richtigen Sperreinheiten. Für eine Transaktion, die ganze Relationen z.B. für einen Verbund verarbeitet, sind hingegen Relationen die richtigen Sperreinheiten.

4.1.1 Gestaffelte Sperreinheiten

Die Menge der elementaren Objekte, zu denen eine Transaktion zugreift, kann sogar bei gleichzeitig aktiven Transaktionen sehr stark variieren: Bei einer Dateneingabe ist vielleicht nur ein einziges Feld betroffen, bei einer Abfrage können mehr oder weniger große Teile der Datenbank gelesen werden, bei einem Tages- oder Wochenabschluß wird die gesamte Datenbank gelesen und ggf. verändert.

Bei so verschiedenen Transaktionen ist eine feste Einheit insgesamt nie zufriedenstellend. Vielmehr müssen Einheiten verschiedener Größe vorgesehen werden.

Die Lösung des Problems besteht darin, eine Folge aufeinander aufbauender **Sperreinheiten** bzw. **Typen von Objekten** zu benutzen, z.B. die Folge Tupel, Relation und Datenbank. Jedes Objekt eines Typs ist in *genau einem* Objekt des nächstgrößeren Typs enthalten. Die Menge der sperrbaren Objekte ist daher baumartig strukturiert. Bild 4.1 zeigt ein Beispiel.

Die ganze Datenbank besteht aus den Objekten der zweitgrößten Einheit (im Beispiel: Relationen), diese wiederum aus Objekten des nächstkleineren Typs (im Beispiel: Tupel) usw. bis zur kleinsten Einheit.

Wenn ein Objekt in einem anderen enthalten ist, nennen wir es ein **Teilobjekt** und das andere Objekt **übergeordnetes Objekt**. Jeder Zugriff zu einem Objekt ist gleichzeitig ein (potentieller) Zugriff zu allen

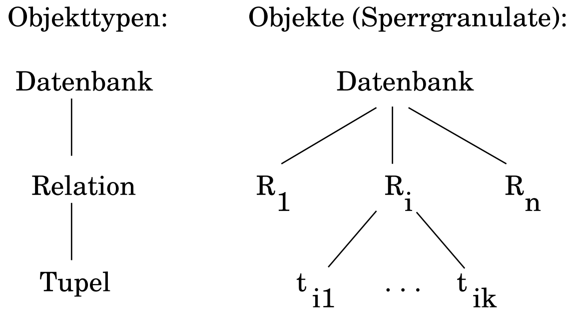


Abbildung 4.1: Eine Hierarchie von Sperreinheiten

seinen Teilobjekten. Wenn bspw. eine Relation gelöscht wird, werden auch alle ihre Tupel gelöscht. Ebenso ist ein Zugriff zu einem Objekt gleichzeitig ein Zugriff zu seinen übergeordneten Objekten. Wenn ein Tupel geändert wird, so wird dadurch auch der Gesamtinhalt der Relation verändert.

Die einzelnen Sperreinheiten sind somit nicht mehr unabhängig voneinander. Die einleitend erwähnte Voraussetzung dafür, die grundlegenden Sperrverfahren anwenden zu können, sind hier nicht mehr gegeben. In diesem Lehrmodul werden Erweiterungen des 2-Phasen-Protokolls vorgestellt, durch die auch Hierarchien von Sperreinheiten korrekt behandelt werden.

Sperreinheiten vs. Typen von Einheiten. Die Menge aller Objekte ist durch die Beziehung "Teilobjekt" baumartig strukturiert. Wurzel ist immer die Datenbank. Die Blätter des Baumes nennen wir **elementare Objekte**.

Die baumartige Enthaltenseinsstruktur der Sperreinheiten ist entscheidend für die folgenden Protokolle[11]; daß die Sperreinheiten in Ebenen angeordnet werden können und man jede Ebene durch einen

[11] Am Ende dieses Abschnitts werden wir das Sperrverfahren auf den Fall verallgemeinern, daß die Objektmenge nur halbgeordnet (also kein Baum) ist.

Sperreinheitentyp charakterisieren kann, erhöht die Anschaulichkeit, ist für die Protokolle aber nicht entscheidend. Man kann die Baumstruktur auch als "besteht-aus"-Struktur interpretieren. Ein Zugriff zu einem nichtelementaren Objekt wird dann durch Zugriffe zu (potentiell) allen seinen Teilobjekten realisiert.

4.1.2 Implizite Sperren

Unser Ziel ist es, ein Sperrverfahren zu entwickeln, das es erlaubt, gleichzeitig Objekte verschiedener Größe (also auf verschiedenen Ebenen) zu sperren. Die impliziten Zugriffe zu Teilobjekten begründen das folgende *Grundprinzip* für Sperren bei variabler Granularität:

> *Wenn ein Objekt gesperrt ist, dann sind implizit alle seine Teilobjekte im gleichen Modus gesperrt.*

Anders gesagt existieren an den Teilobjekten **implizite Sperren** im gleichen Modus.

Dieses Grundprinzip ist vor allem für die Einrichtung von Sperren relevant. Beispiel: Eine Relation sei lesegesperrt, damit auch implizit alle Tupel der Relation. Ein einzelnes Tupel kann jetzt für eine andere Transaktion zwar lesegesperrt, aber nicht mehr schreibgesperrt werden. Umgekehrt kann dann, wenn ein einzelnes Tupel lesegesperrt ist, die gesamte Relation nicht mehr schreibgesperrt werden.

Die impliziten Sperren müssen bei der Prüfung, ob eine beantragte Sperre zugeteilt werden kann, genauso wie explizite Sperren berücksichtigt werden. Eine beantragte Sperre auf einem Objekt o kann nur dann eingerichtet werden, wenn gilt:

a. Sie ist verträglich mit allen Sperren anderer Transaktionen auf o. Diese Sperren sind explizit.
b. Sie ist verträglich mit allen impliziten Sperren anderer Transaktionen auf o, d.h. mit allen expliziten Sperren anderer Transaktionen auf übergeordneten Objekten von o.
c. Sie ist verträglich mit allen expliziten Sperren auf allen Teilobjekten von o (wegen der impliziten Sperren auf den Teilobjekten).

Die Überprüfung der beiden ersten Bedingungen ist ohne größeren Aufwand möglich, da die Zahl der Ebenen und damit auch die Zahl der übergeordneten Objekte gering ist. Die Überprüfung der dritten Bedingung ist hingegen ohne spezielle Vorkehrungen sehr aufwendig: Alle Teilobjekte müssen durchlaufen und praktisch einzeln gesperrt werden. Dieses Vorgehen ist viel zu ineffizient.

4.2 Warnsperren

Die Lösung des Aufwandsproblems ergibt sich aus folgender Überlegung: Für die Entscheidung, ob eine Sperre eingerichtet werden kann, reicht es völlig aus zu wissen, ob überhaupt auf irgendeinem Teilobjekt eine Lese- oder Schreibsperre vorhanden ist. Damit diese Information sofort verfügbar ist, muß beim Sperren eines Teilobjektes eine entsprechende Warnung bei allen übergeordneten Objekten hinterlassen werden. Dies kann in einem mit den Prüfungen zu b erfolgen. Diese Warnungen werden als spezielle Sperren behandelt, sogenannte **Warnsperren** oder *intention locks*. Warnsperren haben demgemäß folgende Besonderheiten:

- Sie sind nur für Objekte mit Teilobjekten zulässig.
- Sie implizieren keinerlei Zugriffsrechte auf das Objekt selbst, sondern zeigen nur an, daß ein Teilobjekt gesperrt wird oder ist.

Der Sperrmodus ist **IS** bzw. **IX**, wenn Teilobjekte lese- bzw. lese- und schreibgesperrt sind. Zusammen mit den vorhandenen Sperrmodi gilt folgende Verträglichkeitsmatrix:

Modus der vorhandenen Sperre:	Modus der beantragten Sperre: S	X	IS	IX
S	+	–	+	–
X	–	–	–	–
IS	+	–	+	+
IX	–	–	+	+

Warnsperrenprotokoll für baumstrukturierte Objektmengen: Für Warnsperren muß das 2-Phasen-Protokoll um folgende Regeln erweitert werden:

- Sperren müssen immer in der Reihenfolge auf dem Pfad von der Wurzel zum Teilbaum hin angefordert werden.
- Freigaben von Sperren sind nur in der umgekehrten Reihenfolge zulässig.
- Bevor ein (Teil-) Objekt gesperrt wird, müssen vorher alle übergeordneten Objekte gemäß der folgenden Tabelle gesperrt werden (sofern sie es nicht schon sind):

Sperrmodus für Teilobjekt	erforderlicher Sperrmodus für übergeordnete Objekte:
S (oder IS)	IS (oder IX)
X (oder IX)	IX

Beispiel: Die Folge der Sperreinheitstypen sei Tupel, Relation, Datenbank. T1 will in Relation R ein Tupel t1 lesen, und T2 will in Relation R ein Tupel t2 schreiben. Dann sind folgende Sperren in dieser Reihenfolge anzufordern:

T1:	sperre Datenbank	im Modus IS
	sperre Relation R	im Modus IS
	sperre Tupel t1	im Modus S
T2:	sperre Datenbank	im Modus IX
	sperre Relation R	im Modus IX
	sperre Tupel t2	im Modus X

Wenn nun t1 $\neq$ t2, so sind beide Transaktionen parallel ausführbar, denn die IX- und die IS-Sperre auf der Datenbank und der Relation R sind verträglich.

Ein gewisser Nachteil von variabel großen Sperreinheiten ist die Vervielfachung der Zahl der Sperren, besonders wenn viele einzelne elementare Objekte gesperrt werden müssen. Warnsperren auf übergeordneten Objekten dürfen analog zu Lesesperren nicht in der Sperrtabelle "zusammengelegt" werden, wenn sie von verschiedenen Transaktionen stammen (wegen der Korrektheit der Freigabe). Stammen sie von der gleichen Transaktion, so kann diese mit einer einzigen Warnsperre für mehrere gesperrte Teilobjekte auskommen, muß aber bei der Freigabe von Sperren vor EOT über diese Objekte Buch führen, so daß letztlich nur wenig gewonnen ist.

SIX-Sperren. Die beiden bisher vorhandenen Warn-Sperrmodi sind bei einer häufigen Form von Transaktionen noch nicht zufriedenstellend. Diese Transaktionen

- lesen typischerweise ein ganzes Objekt (bzw. einen Teilbaum), z.B. eine Relation, und
- schreiben nur einzelne Teilobjekte (bzw. Teile des Teilbaums), z.B. einzelne Tupel.

Mit den vorhandenen Sperrmodi können Sperren auf zwei Arten gesetzt werden:

a. Das Gesamtobjekt wird exklusiv gesperrt (und damit implizit alle seine Teilobjekte). Dies führt zu unnötigen Blockierungen.
b. Das Gesamtobjekt wird nur im IX-Modus gesperrt, die geschriebenen Objekte im X-Modus, die anderen im S-Modus. Diese Vorgehensweise ist wegen der Vielzahl der S-Sperren zu aufwendig.

Wünschenswert wäre eigentlich eine S-Sperre auf dem Gesamtobjekt und X-Sperren auf den geschriebenen Teilobjekten. Für letztere würden zwar zusätzliche implizite S-Sperren existieren, diese würden aber nicht weiter stören. Das Gesamtobjekt müßte S- *und* IX-gesperrt werden. Eine Transaktion kann aber nur eine einzige Sperre auf einem Objekt halten.

Die Lösung des Problems ist der **SIX**-Sperrmodus. Dieser vereinigt genau die Zugriffsrechte des S- und IX-Modus.

Der SIX-Modus ist nur mit dem IS-Modus verträglich. Wir erhalten die folgende erweiterte Kompatibilitätsmatrix:

Modus der vorhandenen Sperre:	Modus der beantragten Sperre:				
	S	X	IS	IX	SIX
S	+	–	+	–	–
X	–	–	–	–	–
IS	+	–	+	+	+
IX	–	–	+	+	–
SIX	–	–	+	–	–

Das Warnsperrenprotokoll muß um folgende Regel erweitert werden: Für eine SIX-Sperre müssen alle übergeordneten Objekte mindestens im IX-Modus gesperrt sein. Da der SIX-Modus alle Zugriffsrechte des IX-Modus umfaßt, reicht er für die übergeordneten Objekte ebenfalls aus.

Die X-Sperren auf den Teilobjekten eines SIX-gesperrten Objektes müssen natürlich zusätzlich einzeln angefordert werden.

Als Beispiel betrachten wir eine Transaktion T3, die die Relation R ganz liest und das Tupel t in R schreibt. Folgende Sperren sind anzufordern:

T3:	sperre Datenbank	im Modus IX
	sperre Relation R	im Modus SIX
	sperre Tupel t	im Modus X

Wir haben nun insgesamt fünf verschiedene Sperrmodi kennengelernt, die verschiedene Zugriffsrechte für Inhaber gewähren. Ein Vergleich ergibt die in Bild 4.2 gezeigte Struktur. Darin bedeutet jede Linie, daß die Zugriffsrechte des unteren Sperrmodus in denen des oberen enthalten sind. S- und IX-Sperren haben Zugriffsrechte, die einander nicht implizieren.

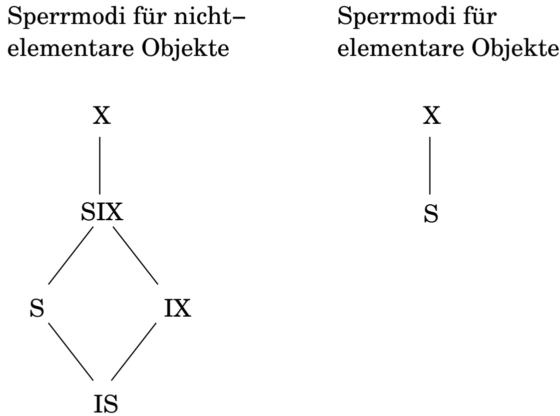


Abbildung 4.2: Eine Hierarchie von Sperrmodi

Die oberen Sperren sind "**strenger**" bzw. "**exklusiver**", die unteren sind "**schwächer**". Man kann immer eine strengere Sperre verwenden, als unbedingt notwendig, ohne inkorrekte Verzahnungen zu riskieren. Die Parallelität wird dadurch natürlich unnötig eingeschränkt.

4.3 Halbgeordnete Objektmengen

Die variable Granularität, wie sie bisher behandelt wurde, basierte auf einer baumartigen Enthaltenseinsstruktur. Jedes Objekt (außer der Datenbank) ist in genau einem umgebenden Objekt als Teilobjekt enthalten.

Diese Annahmen sind in vielen Fällen nicht zulässig. Nehmen wir z.B. folgendes an: Eine Datenbank wird in mehreren Dateien (oder Partitionen o.ä.) gespeichert. Eine Datei enthält mehrere Relationen oder nur Teile von Relationen, eine Relation kann sich über mehrere Dateien erstrecken. Ein Speichersatz enthält genau ein Tupel. Sowohl Dateien als auch Relationen sollen sperrbare Einheiten sein. Eine Datei ist weder eine größere noch eine kleinere Einheit als eine Relation. Die Menge der Objekte ist daher nur halbgeordnet (gleiches gilt für die

beteiligten Objekttypen):

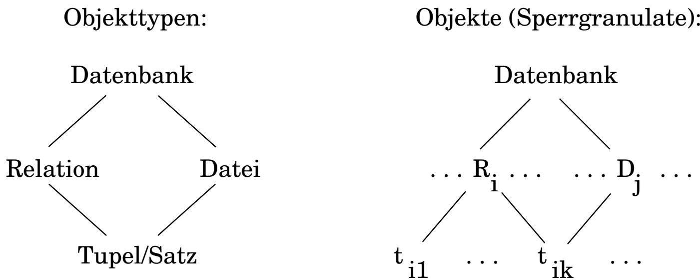


Abbildung 4.3: Halbgeordnete Sperreinheiten

Die Menge der Objekte ist kein Baum mehr, aber immer noch halbgeordnet mit der Datenbank als Wurzel; die Halbordnung ist wieder die Relation "Teilobjekt" zwischen Objekten.

Das oben aufgestellte Grundprinzip des Sperrens bei variabler Granularität lautete: Wenn ein Objekt gesperrt ist, dann sind alle seine Teilobjekte implizit im gleichen Modus gesperrt. Ziel war, implizite Sperren nicht verwalten zu müssen. Das Protokoll erreichte dies: implizite Sperren werden respektiert, ohne bei Sperrenanforderungen je explizit "abgefragt" zu werden. Abgefragt werden nur die expliziten Sperren bzw. die Warnsperren bei übergeordneten Objekten. *Implizite Sperren können nicht abgefragt werden.*

Das obige Grundprinzip und zugehörige Protokoll kann nicht direkt auf halbgeordnete Objekttypen verallgemeinert werden. Nehmen wir wieder die Objekttypen wie im letzten Beispiel an, ferner zwei Transaktionen, die folgende Sperren gemäß obigem Protokoll anfordern (und auch erhalten):

T1:	sperre Datenbank	im Modus IX
	sperre Datei D	im Modus IX
	sperre Satz S	im Modus X

T2:	sperre Datenbank	im Modus IS
	sperre Relation R	im Modus S

Für Satz S, der ein Tupel der Relation R enthalten möge, hält T1 eine explizite X-Sperre und T2 eine implizite S-Sperre. Ursache dieses Fehlverhaltens ist, daß T1 nicht bei allen übergeordneten Objekten von Satz S Warnsperren hinterlassen hat. Wir müssen also das Protokoll um folgende Regel erweitern:

Zusatzregel 1: Eine explizite Sperre darf erst dann angefordert werden, wenn passende Warnsperren *auf allen Pfaden* von der Datenbank zu dem betreffenden Objekt eingerichtet wurden.

Mit dieser Änderung hätte T1 im obigen Beispiel zusätzlich eine IX-Sperre auf der Relation R angefordert. Diese hätte die S-Sperre auf R von T2 verhindert. Wenn umgekehrt zuerst T2 seine Sperren verlangt hätte und dann erst T1, hätte die S-Sperre von T2 verhindert, daß die IX-Sperre auf R für T1 eingerichtet worden wäre.

Dennoch sind wir nicht fertig. Wir ändern T1 wie folgt:

T1:	sperre Datenbank	im Modus IX
	sperre Datei D	im Modus X

Der Satz S wird nun von T1 und T2 implizit gesperrt, aber in unverträglichen Sperrmodi.

Datei D und Relation R sind zwei Objekte, von denen keines Teilobjekt des anderen ist, die aber gemeinsame Teilobjekte haben. Solche Objektpaare nennen wir **Nachbarn**. Explizite Sperren auf Nachbarn sind insofern problematisch, als die impliziten Sperren von gemeinsamen Teilobjekten selbst nach der Änderung des Protokolls unverträglich sein können. Daher benötigen wir die

Zusatzregel 2: Bei explizitem Sperren (von nichtelementaren Objekten) müssen vorher alle Nachbarn geeignet gewarnt werden.

In unserem Beispiel muß, bevor die Datei D gesperrt wird, auf jeder Relation, die ganz oder teilweise in D enthalten ist, eine Warnsperre im passenden Modus angefordert werden. T1 muß also konkret die Relation R mindestens IX-sperren; hierdurch wird verhindert, daß T2 sie S-sperrt.

Die vorgeschlagene Zusatzregel 2 ist allerdings problematisch, wenn D genau R enthält (was häufig der Fall ist), also wenn die Nachbarn (hier: D und R) die gleichen Teilobjekte haben. Wenn T1 einen der Nachbarn X-sperrt, können auf dem anderen Nachbarn keine expliziten Sperren für T2 mehr zugeteilt werden, denn vorher müßte T2 eine IS- oder IX-Sperre auf dem X-gesperrten Nachbarn erhalten. Genausogut könnte man alle Nachbarn für T1 im X-Modus sperren. Aus diesem Grunde wählen wir eine andere

Zusatzregel 2a: Wenn ein Objekt X-gesperrt wird, dann werden auch alle Nachbarn X-gesperrt.

Im Vergleich zur ersten Version schränkt dies die Parallelität meist nicht ein, liefert aber dem Inhaber der Sperren bessere Zugriffsmöglichkeiten.

Zusatzregel 2a erlaubt es übrigens, Zusatzregel 1 für S-Sperren zu vereinfachen: es müssen nur noch auf irgend*einem* Pfad von der Datenbank zum Objekt IS-Sperren eingerichtet werden. Bei S-Sperren sind keine Warnungen bei den Nachbarn erforderlich: Andere S-Sperren stören nicht, eine X-Sperre auf einem Nachbarn kann bei beiden Versionen der Zusatzregel 2 nicht eingerichtet werden.

Anders gesehen erzeugt eine X-Sperre auf einem Objekt nicht mehr implizite X-Sperren auf seinen Teilobjekten. Ein Objekt ist nur dann implizit X-gesperrt, wenn *alle* seine direkt übergeordneten Objekte (implizit oder explizit) X-gesperrt sind.

Dagegen ist ein Objekt implizit S-gesperrt, wenn *eines* seiner übergeordneten Objekte explizit S-gesperrt ist (oder X-gesperrt, denn X-Sperren implizieren S-Sperren, oder SIX-gesperrt).

Damit haben wir ein auf halbgeordnete Objektmengen *verallgemeinertes Grundprinzip* der variablen Granularität erhalten; die frü-

here Form ist ein Spezialfall der neuen Form. Mit diesem veränderten Grundprinzip können wir die erforderlichen Ergänzungen zum 2-Phasen-Protokoll so zusammenfassen:

Warnsperrenprotokoll für halbgeordnete Objektmengen: Neben der 2-Phasen-Regel gelten folgende Regeln:

- Ein Objekt kann erst dann S- (oder IS-) gesperrt werden, wenn alle Objekte auf *irgendeinem* Pfad bis zur Wurzel des Objektbaumes, also der Datenbank, im IS-Modus oder strenger gesperrt sind.
- Ein Objekt kann erst dann IX-, SIX- oder X-gesperrt werden, nachdem alle Objekte auf *allen* Pfaden zur Wurzel im IX-Modus oder strenger gesperrt sind.
- Freigaben von Sperren sind nur möglich, wenn keine (expliziten) Sperren auf Teilobjekten mehr gehalten werden.

Eine X-Sperrung eines Objekts ist nur sinnvoll, wenn alle seine Nachbarn ebenfalls X-gesperrt sind. Man könnte dies daher als weitere Regel zum Protokoll hinzunehmen.

Eine Verallgemeinerung von Warnsperren für nichtelementare Objekte, unter anderem auf sogenannte Update-Sperren, sowie weitere Varianten zugehöriger Protokolle finden sich in [Ko83].

Glossar

Granularität (*granularity*): Feinheit, mit der die gesamte Datenbank in sperrbare Einheiten zerlegt ist

implizite Sperre: sind an Sperreinheiten in einer Hierarchie von Sperreinheiten impliziert durch eine explizite Sperre an einer äußeren Sperreinheit

Nachbarn: (in einer Hierarchie von Sperreinheiten) Sperreinheiten, von denen keines Teil des anderen ist, die aber gemeinsame Untereinheiten haben

Warnsperre (*intention lock*): Anzeige, daß (potentiell) ein Teilobjekt des Objekts gesperrt ist; impliziert keine Zugriffsrechte auf dem ganzen Objekt

Lehrmodul 5:

Semantikgestützte Concurrency-Control-Verfahren

Zusammenfassung dieses Lehrmoduls

Die gängigen Concurrency-Control-Verfahren basieren auf einem "syntaktischen" Konfliktbegriff: Es werden nur lesende und schreibende Operationen unterschieden. In manchen Fällen liegt es nahe, semantische Eigenschaften, insb. Kommutativitätseigenschaften der Operationen auszunutzen und zu einem semantischen Konfliktbegriff überzugehen. Typischerweise handelt es sich um numerische Datenfelder, auf denen Beträge addiert und subtrahiert werden. Es zeigt sich allerdings, daß eine Reihe nicht offensichtlicher Probleme gelöst werden müssen, die durch Rollback, Bereichsgrenzen und "Ausnahmen" von der Kommutativität entstehen und die teilweise sehr aufwendige Gegenmaßnahmen erforderlich machen.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Sperrverfahren

empfohlen:
- Recovery
- Concurrency-Control-Theorie

Stoffumfang in Vorlesungsdoppelstunden: 1.2

©2001 Udo Kelter

5.1 Einführung

Die gängigen Concurrency-Control- (CC-) Verfahren basieren auf einem "syntaktischen" Begriff Konflikt: Es wird nur unterschieden, ob Objekte geschrieben oder nur gelesen werden, von weitergehenden Details der Zugriffsoperationen wird abstrahiert. Bei den Sperrverfahren (s. Lehrmodul 3) äußert sich dies darin, daß nur Lese- und Schreibsperren unterschieden werden. Auch theoretische Analysen des Serialisierbarkeitsproblems führen zu der Erkenntnis, daß man i.a. Korrektheitsbegriffe auf Basis des syntaktischen Konfliktbegriffs verwenden muß (s. Lehrmodul 8), konkret muß ein CC-Verfahren wenigstens die cp-Serialisierbarkeit realisieren. Ein Ablauf ist cp-serialisierbar, wenn beim Serialisieren keine in Konflikt stehenden Ereignisse vertauscht werden. Würde man in Konflikt stehende Ereignisse vertauschen, würde die Sicht der betroffenen Transaktionen oder der Endzustand der betroffenen Objekte verändert werden.

Manchmal kennt man die Semantik von Aktionen recht genau und möchte diese Kenntnisse dahingehend ausnutzen, die Parallelität der Transaktionsausführungen zu erhöhen. Die Anwendungsbeispiele kommen nicht nur aus klassischen Datenbankanwendungen, sondern auch aus verteilten Betriebssystemen bzw. allgemeiner verteilten, parallel benutzbaren Objektsystemen, wo z.B. Datentypen wie Listen oder Mengen auftreten.

In vielen Fällen scheint die Ausnutzung semantischer Eigenschaften (zumindest auf den ersten Blick) relativ einfach möglich zu sein. Ein Beispiel: Wir nehmen an, zwei Umbuchungen seien wie in Bild 5.1 angegeben verzahnt. Wir benutzen eine tabellenförmige Notation für Abläufe; X und Y seien lokalisierte Objekte, "X:=X+a" steht als Abkürzung für "u:=X; u:=u+a; X:=u", worin u eine lokale Variable sei.

Bei diesem Ablauf liest T2 einen ungesicherten Wert von T1, beide Transaktionen erhalten eine inkonsistente Sicht der Datenbank, und dennoch wird man ihn als "korrekt" empfinden, denn die Umbuchungen werden korrekt durchgeführt. Im nächsten Abschnitt wollen wir zunächst klären, in welchem Sinne dieser Ablauf korrekt ist, um dann

T1	T2	Werte von X	Werte von Y
		1000	500
X:=X-500		500	
	Y:=Y-300		200
Y:=Y+500			700
	X:=X+300	800	

Abbildung 5.1: Verzahnte Ausführung zweier Umbuchungen

im übernächsten Abschnitt konkrete Sperrverfahren zu entwickeln.

5.2 Modifikationen

5.2.1 Definition

Daß man den oben gezeigten Ablauf als “korrekt” empfindet, liegt u.a. an der Art, wie die Objekte gelesen und zurückgeschrieben werden. Wesentlich sind folgende Merkmale:

1. Die Veränderungen an den Datenbank-Objekten, also das Lesen des alten Inhalts, Berechnen und zurückschreiben des neuen Inhalts (“X:=X+a”) sind atomar, d.h. das veränderte Datenbank-Objekt wird zwischenzeitlich nicht von anderen Transaktionen gelesen oder verändert. Wir nennen eine solche Veränderung eine **Modifikation**.

   Wir fassen Modifikationen als eine dritte Art von Aktionen auf (neben Lese- und Schreibaktionen).

   Im Gegensatz zu konventionellen Aktionen sind Modifikationen ggf. *keine* Elementaroperation des jeweiligen Datenbankmodells, d.h. sie sind applikationsspezifisch und oberhalb des APIs der Datenbank individuell zu programmieren, genauso wie Transaktionen. Man kann sie daher auch “**Subtransaktionen**” nennen.
2. Gewisse Integritätstests müssen innerhalb der Modifikationen durchgeführt werden. In unserem obigen Beispiel könnte es einen unteren

Grenzwert für den Kontostand geben. Da der Wert des Objekts bei der Rückkehr zur aufrufenden Transaktion schon verändert ist, kommt ein anschließender Test innerhalb der Transaktion schon zu spät. Das Ergebnis von Integritätstests wird durch einen Fehlercode an die aufrufende Transaktion mitgeteilt.

3. Die innerhalb einer Modifikation gelesenen Werte zählen *nicht* zur Sicht einer Transaktion. Typischerweise, so auch im obigen Beispiel, werden sie nur innerhalb dieser Modifikation benutzt, also nicht direkt oder in transformierter Form an den Benutzer weitergegeben oder in ein anderes Objekt geschrieben. In vielen Fällen liefern Modifikationen dennoch einen Wert an die aufrufende Transaktion ab, z.B. einen Fehlercode. Dieser zählt (neben den Werten, die bei normalen Lese-Aktionen gelesen werden) zur Sicht der Transaktion.

5.2.2 Semantische Konfliktfreiheit von Modifikationen

Das Modifikationskonzept allein erklärt noch nicht, warum der obige Ablauf "korrekt" sein soll. Er ist nicht cp-serialisierbar, sogar nicht einmal Sicht-serialisierbar (vgl. 8.5.1). Diese Korrektheitsbegriffe sind hier unnötig streng[12], da sie die vollständigen Sichten berücksichtigen, während tatsächlich nur die reduzierten Sichten relevant sind.

Sofern man die Semantik der Rechenoperationen außer Betracht läßt, ist der obige Ablauf auch nicht Endzustands-serialisierbar (s. Abschnitt 8.3.2). Man findet leicht Interpretationen (also Semantiken der Datentypen und Operationen), bei denen der Endzustand von X und Y im obigen Beispiel mit keinem der Endzustände übereinstimmt, die bei den beiden seriellen Abläufen entstehen. Bei der Endzustands-Serialisierbarkeit werden beliebig "ungünstige" Interpretationen berücksichtigt, während wir oben wissen, daß eine "günstige" Interpretation vorliegt.

Günstig ist hier, daß die Modifikationen in einem gewissen Sinn kommutieren. Um diesen Begriff zu präzisieren, fassen wir eine Modifikation, die oben als *Algorithmus* eingeführt wurde, stattdessen als eine

[12] Wobei wir sie in kanonischer Weise auf das n-Schritt-Modell für Transaktionen erweitern könnten.

Funktion auf dem Wertebereich des modifizierten Datenbank-Objektes auf. Der Hintereinanderausführung von Modifikationen entspricht die Komposition der Funktionen.

So wird im obigen Beispiel zunächst die Modifikation “Erniedrige um 500” auf X ausgeführt, danach “Erhöhe um 300“. Die entsprechenden Funktionen sind $x \mapsto x - 500$ und $x \mapsto x + 300$; ihre Komposition ist $x \mapsto x - 200$. Diese Funktion läßt sich sogar direkt durch eine Modifikation erzeugen, nämlich “Erniedrige um 200”; notwendig ist dies aber nicht.

Das entscheidende Merkmal des Konfliktbegriffs ist folgendes: Wenn zwei Ereignisse nicht in Konflikt stehen, darf ihre Reihenfolge vertauscht werden, ohne daß die Vertauschung Einfluß hat auf

1. den Endzustand der betroffenen Objekte
2. die Sichten der beiden beteiligten Transaktionen.

Zwei derartige Ereignisse nennen wir **semantisch konfliktfrei**.

Die bisherige Definition des Konfliktbegriffs war insofern *syntaktisch*, als sie sich nur auf die Identität von Objekten und die Unterscheidung in verändernde und nicht verändernde Zugriffe bezog, also Merkmale, die auf dem syntaktischen Niveau angesiedelt sind. Die syntaktische Konfliktfreiheit ist hinreichend, aber nicht notwendig für die semantische Konfliktfreiheit.

Offensichtlich stehen Modifikationen i.d.R. in Konflikt mit Lese- und Schreibereignissen.

5.2.3 Sperrmodi für Modifikationen

Wir können semantisch konfliktfreie Aktionen genauso behandeln wie herkömmlich konfliktfreie Aktionen, sowohl bei den Korrektheitsbegriffen für Logs wie auch bei Sperrverfahren.

Wenn eine Transaktion eine Sperre für ein Objekt hält und somit das Recht hat, gewisse Aktionen auszuführen, können wir dies so interpretieren, daß während der Sperrzeiten nur konfliktfreie Aktionen anderer Transaktionen auf diesem Objekt eintreten dürfen. Für

Modifikationen gilt dies analog. Im einfachsten Fall gehört zu einer Modifikation M ein eigener Sperrmodus. Dieser ist kompatibel mit solchen Sperrmodi, die nur das Recht zu Ausführung solcher Aktionen implizieren, die mit M nicht in Konflikt stehen.

Allgemeiner definieren wir zwei Sperrmodi als **kompatibel**, wenn jede der beim ersten Sperrmodus zulässigen Aktionen mit jeder beim zweiten Sperrmodus zulässigen Aktionen (auf dem gleichen Objekt) semantisch konfliktfrei ist. Die bisherige Definition der Verträglichkeit von Lese- und Schreibsperren ist ein Sonderfall dieser allgemeineren Definition[13].

Im obigen Beispiel ergeben sich folgende Verträglichkeiten, wobei die üblichen Modi S und X hinzugenommen wurden:

vorhandene	beantragte Sperre:					
Sperre:	S	X	"+300"	"-300"	"+500"	"-500"
S	+	-	-	-	-	-
X	-	-	-	-	-	-
"+300"	-	-	+	+	+	+
"-300"	-	-	+	+	+	+
"+500"	-	-	+	+	+	+
"-500"	-	-	+	+	+	+

5.2.4 Undo von Modifikationen

Das Undo einer Aktion wird üblicherweise durch Rückschreiben des vorher vorhandenen Werts realisiert. Da ein Objekt von mehreren gleichzeitig aktiven Transaktionen verändert werden kann, würden beim Rückschreiben des alten Werts anläßlich des Rollbacks einer Transaktion auch zwischenzeitliche Änderungen anderer, eventuell bereits abgeschlossener Transaktionen verloren gehen.

Konventionelle Sperrprotokolle halten die Isolation ein: es werden keine ungesicherten Werte gelesen. Die Isolation muß bei Modifikationen

[13] Die Konstruktion von Update- und Warnsperrmodi kann ebenfalls auf beliebige Basis-Sperrmodi verallgemeinert werden, s. [Ko83]).

aufgegeben werden, wenn man mit ihnen überhaupt einen Parallelitätsgewinn erzielen will. Hierdurch entsteht wieder das Problem der Rollbackfortpflanzung.

Zur Lösung dieses Problems benutzt man ein anderes (Rückwärts-) Recovery-Prinzip, nämlich Kompensation. Zu jeder Modifikation wird eine **inverse Modifikation** (kurz: **Invertierung**) vorgesehen, welche vom Programmierer der Modifikation zu liefern ist. Hieraus resultiert, daß für Modifikationen spezielle Einträge im Log vorgesehen werden müssen und daß die inversen Modifikationen jederzeit dem Recovery-Manager in einer Bibliothek zur Verfügung stehen müssen.

Sofern ein Objekt innerhalb einer Transaktion mehrfach modifiziert wurde, müssen die Invertierungen in umgekehrter Reihenfolge durchgeführt werden, sofern sie nicht semantisch konfliktfrei miteinander sind.

Eine zurückgesetzte Transaktion wird durch die Invertierungen gedanklich fortgesetzt, bis sie am Ende den Effekt Null hat. Hieraus folgt:

- Sperren für Modifikationen dürfen *nicht* vor Commit freigegeben werden[14]. Damit die entstehenden Logs korrekt sind, müssen die Invertierungen nämlich konfliktfrei in die Verarbeitungsphase der Transaktion verschoben werden können.
- Die Invertierung zu einer Modifikation M muß mit allen Modifikationen konfliktfrei sein, mit denen M konfliktfrei ist bzw. die beim zu M gehörigen Sperrmodus für andere Transaktionen aufgerufen werden dürfen.

Wir nennen daher zwei Modifikationen nur dann **konfliktfrei**, wenn zusätzlich jede Modifikation mit der Invertierung der anderen und die beiden Invertierungen zueinander konfliktfrei sind.

[14]Bei Sperren für konventionelle Aktionen ist dies im Rahmen des 2-Phasen-Protokolls im Prinzip erlaubt, wegen der Fortpflanzung von Rollback ist jedoch generell davon abzuraten.

Die Verwendung des Kompensationsprinzips führt zu weiteren Problemen beim Neustart nach Systemfehlern, die wir hier nicht näher diskutieren.

5.2.5 Atomarität von Modifikationen

Wir hatten oben vereinbart, Modifikationen als eine dritte Art von Aktionen aufzufassen. Aktionen müssen die gleichen Atomaritätseigenschaften erfüllen wie Transaktionen, nämlich Fehler-Atomarität und Serialisierbarkeit (bzw. funktionale Atomarität).

Ein Unterschied von Modifikationen zu Lese- oder Schreibaktionen ist, daß letztere durch das DBMS implementiert sind, während Modifikationen ggf. vom Anwender zu implementieren sind.

Bei der Realisierung der funktionalen Atomarität ist zu bedenken, daß Modifikationen typischerweise kurz sind und nur ein einziges Objekt betreffen. Durch komplizierte Protokolle ist daher nur wenig zu gewinnen; wir gehen davon aus, daß das modifizierte Objekt wechselseitig ausgeschlossen benutzt wird.

Bezüglich der Fehler-Atomarität gelten im Prinzip die gleichen Überlegungen wie für Transaktionen. Ursachen für den Abbruch einer Modifikation können sein:

- unbeabsichtigte Laufzeitfehler
- programmiertes Rollback nach einem Integritätstest mit negativem Ausgang. Dies kann, muß aber nicht bedeuten, daß auch die zugehörige Transaktion abbricht. Dies wird innerhalb der Transaktion anhand des von der Modifikation zurückgegebenen Fehlercodes entschieden.
- die zugehörige Transaktion wird aus äußeren Gründen zurückgesetzt.

Diese Fehler treten innerhalb der Modifikation auf, d.h. zwischen dem Lesen des Objekts und dem Zurückschreiben des neuen Inhalts. Für das Rollback einer Modifikation können im Prinzip die gleichen

Techniken angewandt werden wie für Transaktionen[15].

5.3 Parametrisierte Modifikationen

5.3.1 Definition

Bei den Modifikationen in den obigen Beispielen wurde stets um einen festen Wert erhöht oder erniedrigt. Programmtechnisch wird man diesen Wert natürlich als Parameter übergeben, d.h. es handelt sich um **parametrisierte Modifikationen**. Diese entsprechen Funktionalen, durch Einsetzen eines zulässigen Parameters ergibt sich eine Modifikation bzw. eine Funktion. Es können auch mehrere Parameter vorhanden sein, so daß man ein zulässiges Tupel von Parametern einsetzen müßte. Die Menge aller zulässigen Parameterbelegungen definiert **die zu einer parametrisierten Modifikation gehörige Menge von Modifikationen**.

Beispiele von parametrisierten Modifikationen, die oben auftraten, sind "Erhöhe um ..." und "Erniedrige um ...", kurz **incr()** und **decr()**.

Diesen Funktionalen entsprechen numerische Operatoren. Deren Kommutativität impliziert sofort, daß Modifikationen bei *beliebigen* Parameterwerten semantisch konfliktfrei sind (Probleme, die bzgl. Kommutativität durch Bereichsüberlauf verursacht werden, behandeln wir später), denn für alle Parameterwerte a, b gilt (* steht für Komposition):

$$\begin{aligned} \mathrm{incr(a)} * \mathrm{incr(b)} &= \mathrm{x} \mapsto (\mathrm{x} + \mathrm{a}) + \mathrm{b} \\ &= \mathrm{x} \mapsto (\mathrm{x} + \mathrm{b}) + \mathrm{a} = \mathrm{incr(b)} * \mathrm{incr(a)} \end{aligned}$$

"Einfache" Modifikationen fassen wir i.f. als Sonderfall von parametrisierten auf. Die "Menge von zugehörigen Modifikationen" besteht nur aus ihr selbst.

[15] Für die Invertierung von Modifikationen müssen hingegen spezielle Recovery-Mechanismen vorgesehen werden.

5.3.2 Vollständige Konfliktfreiheit

Durch eine parametrisierte Modifikation wird i.a. eine so große Menge von zugehörigen Modifikationen definiert, man man nicht mehr sinnvoll nach der oben vorgestellten Methode verfahren kann, jeder Modifikation einen eigenen Sperrmodus zuzuordnen. Wie eben am Beispiel "incr()" gezeigt wurde, sind sowieso alle zugehörigen Modifikationen paarweise semantisch konfliktfrei, so daß man offensichtlich mit einem einzigen Sperrmodus auskäme. Dies gilt sogar dann noch, wenn man zusätzlich alle Modifikationen hinzunimmt, die zu "decr()" gehören. Diese Gegebenheit läßt sich wie folgt formaler beschreiben:

Sei **PM** eine Menge von parametrisierten (und "einfachen") Modifikationen. Wir bilden die Gesamtmenge aller zugehörigen Modifikationen. Wenn in dieser Gesamtmenge alle Modifikationen paarweise konfliktfrei sind, dann heißt PM **vollständig (semantisch) konfliktfrei**.

Offensichtlich reicht für eine Menge vollständig konfliktfreier parametrisierter Modifikationen ein einziger Sperrmodus aus. Eine Sperre in diesem Modus berechtigt dazu, auf dem jeweiligen Objekt beliebige Modifikationen aus dieser Menge auszuführen. Der Modus ist mit sich selbst verträglich.

Im obigen Beispiel können wir also für "incr()" und "decr()" einen einzigen Sperrmodus "incr" vergeben. Wir fügen noch einen weiteren Sperrmodus "mult" hinzu, der für Multiplikationen oder Divisionen steht. Die Verträglichkeitsmatrix ist:

vorhandene	beantragte Sperre:			
Sperre:	S	X	incr	mult
S	+	-	-	-
X	-	-	-	-
incr	-	-	+	-
mult	-	-	-	+

Eine Sperre im Modus "incr" berechtigt nun dazu, beliebig viele Beträge zu einem Objekt hinzuzuaddieren.

In dem Fall, daß man innerhalb der gleichen Transaktion dieses Objekt außerdem mit einem Faktor multiplizieren will, benötigt man einen Sperrmodus, der die Zugriffsrechte von “incr” und “mult” vereinigt. Ein Verfahren zur Konstruktion derartiger Kombinations-Sperrmodi ist in [Ko83] für die normalen Sperrmodi vorgestellt worden; es kann im Prinzip auf beliebige Arten von Sperrmodi verallgemeinert werden. Es stellt sich allerdings die Frage, unter welchen Randbedingungen so komplexe Mengen von Sperrmodi und zugehörigen Kompatibilitätstest sowie Höherstufungsregeln noch sinnvoll sind.

5.4 Bereichsgrenzen

Bei der Feststellung, daß die beiden Modifikationen im obigen Beispiel semantisch konfliktfrei sind, haben wir ein Problem ausgeklammert: nach der ersten Modifikation könnte ein Bereichsüberlauf eintreten.

Hierzu ein Beispiel: das veränderte Objekt stellt ein Konto, einem Lagerbestand o.ä. dar, der Stand darf nicht unter 0 fallen, anfangs sei der Stand 300. Nun wollen zwei Transaktionen jeweils 200 Einheiten abbuchen. Bei der zweiten Abbuchung wird die Bereichsgrenze unterschritten, was durch einen Integritätstest innerhalb der Modifikation festgestellt wird. Die Modifikation wird abgebrochen und zurückgesetzt. Die aufrufende Transaktion erhält einen entsprechenden Fehlercode als Ergebnis. Dieser Fehlercode zählt zur Sicht der Transaktionen; er wird benutzt, um über das weitere Vorgehen in der Transaktion zu entscheiden, z.B., ob die Transaktion ebenfalls abgebrochen werden soll. *Die Sicht der Transaktion* und der Endzustand des modifizierten Objekts sind also durch die Reihenfolgevertauschung *verändert worden*!

In vielen Fällen wird nach einer Bereichsüberschreitung die Transaktion ebenfalls abgebrochen werden; man kann sich dann auf den Standpunkt stellen, daß die Sicht der Transaktion dann ohnehin eine Rolle gespielt hat, das Risiko des Abbruchs der Transaktion bestand auf jeden Fall und der Abbruch war somit ein “korrekter” Ausgang der Transaktion.

5.4.1 Inkonsistente Zwischenzustände

Das folgende Beispiel zeigt allerdings, daß die logische Atomarität dennoch verloren gehen kann: zwei Transaktionen soll zwei Teilbeträge von zwei Konten abbuchen und die Summe einem dritten Konto gutschreiben. Transaktion T1 würde die Sequenz X:=X-200; Y:=Y-200; Z:=Z+400 ausführen, T2 die Sequenz Y:=Y-200; X:=X-200; U:=U+400. Innerhalb jeder Modifikation "-200" wird getestet, ob der Wert negativ werden würde; falls ja, wird die Modifikation abgebrochen. Die Transaktion löst daraufhin ein Rollback aus.

Nehmen wir an, die Konten X und Y enthalten beide anfangs 300. Dann würde bei serieller Ausführung genau eine der beiden Transaktionen erfolgreich ausgeführt werden, die andere nicht mehr. Im folgenden Ablauf wird aber keine Transaktion erfolgreich ausgeführt:

T1	T2	Werte von		Fehler
		X	Y	
		300	300	
X:=X-200		100		
	Y:=Y-200		100	
Y:=Y-200			(-100??) 100	Y < 0
	X:=X-200	(-100??) 100		X < 0
Rollback (X:=X+200)		300		
	Rollback (Y:=Y+200)		300	

Die Ursache des Problems im vorigen Beispiel liegt in folgendem: nach den beiden ersten Rechenschritten haben die Daten einen *temporären* Zustand erreicht, den man als inkonsistent bezeichnen kann. Er würde bei einer seriellen Ausführung nicht erreicht und liegt sozusagen "zu nahe" an den Bereichsgrenzen, er manifestiert sich in Form

von überflüssigen Fehlermeldungen bzw. Rollbacks. Man kann auf das Problem unterschiedlich reagieren:

1. Man kann solche Abläufe verhindern: Dann sind allerdings komplizierte Algorithmen zu deren Erkennung erforderlich (s.u.), die u.U. die Absicht, durch Modifikationen die Performance des DBMS zu verbessern, durchkreuzen.
2. Man kann solche Abläufe dulden: Dann verzichtet man auf die vollständige Atomarität, man toleriert die (seltenen) Abweichungen, sofern nur geringe Folgeschäden auftreten. Die Vorstellung ist typischerweise, daß aus Sicht jeder einzelnen Transaktion der erfolglose Ausgang ein im Prinzip denkbares, also korrektes Ergebnis war und daß der Benutzer eine erfolglose Transaktion später und einmal wiederholen kann.

5.4.2 Unsichere Zwischenzustände und inverse Modifikationen

Bereichsgrenzen werfen zusätzliche Probleme bei den inversen Modifikationen auf, die im Rahmen eines Rollbacks fällig werden. Betrachten wir hierzu folgendes Beispiel: Ein Konto X hat einen Stand von 200 und darf nicht negativ werden; Transaktion T1 erhöht das Konto um 500 Einheiten und führt danach weitere Aktionen auf anderen Objekten aus, die zu einem Rollback führen; Transaktion T2 vermindert das Konto um 400 Einheiten. Unterstellt sei folgender Ablauf:

T1	T2	Werte von X	Fehler
		200	
X:=X+500		700	
	X:=X-400	300	
Rollback (X:=X-500)		-200??	X<0 !!

Die im Rahmen des Rollbacks von T1 erforderliche inverse Modifikation X:=X-500 würde zu einer Überschreitung der Bereichsgrenze

führen! Verursacht wird das Problem dadurch, daß T2 auf einem unsicheren Wert gearbeitet hat. T2 hätte an dieser Stelle gar nicht ausgeführt werden dürfen, denn T2 ist im gegebenen Zustand nicht semantisch konfliktfrei mit der inversen Modifikation zu X:=X+500. Anders gesehen hatte bei der Ausführung von T2 X zwar den Wert 700, und bei diesem Wert war die Bereichsgrenzen noch weit genug entfernt, aber es war ein Rollback von T1 möglich, wonach X=200 gewesen wäre, und in diesem Zustand war T2 nicht mehr erfolgreich ausführbar.

Allgemeiner gesehen stellt sich das Problem folgendermaßen dar: Ein Wert kann unsicher sein, weil mehrere nicht abgeschlossene Transaktionen Modifikationen auf ihm ausgeführt haben. Jede der Transaktionen kann unabhängig von den anderen zurückgesetzt werden; bei n Transaktionen bestehen somit 2^n Möglichkeiten, daß eine Teilmenge der Transaktionen zurückgesetzt wird. Jede Teilmenge entspricht einem bestimmten Wert des Objekts, der bei Rücksetzung dieser Transaktionen entstehen würde. Eine Transaktion darf nur dann ausgeführt (bzw. die entsprechende beantragte Sperre zugeteilt) werden, wenn sie bei *allen* Werten erfolgreich ausgeführt werden kann[16].

Wegen der kombinatorischen Explosion der Zahl der Teilmengen ist es praktisch nahezu ausgeschlossen, ab einem Parallelitätsgrad von ca. 5 alle Werte tatsächlich einzeln zu berechnen. Ein noch halbwegs effizient realisierbares Verfahren ist die nachfolgend beschriebene Überwachung von Unsicherheitsbereichen.

5.4.3 Überwachung von Unsicherheitsbereichen

Das folgende Verfahren ist nur bei linear geordneten Wertebereichen anwendbar, bei denen sich die Bereichsgrenzen einfach als Intervall (minimaler und maximaler Wert des Objekts) ausdrücken lassen. In der Praxis kommen wohl nur numerische Wertebereiche und Modifikationen wie incr() und decr() infrage, von denen wir i.f. auch ausgehen.

Die Idee des Verfahrens besteht darin, daß man gar nicht alle Teil-

[16] Analog kann man dies auch für nicht erfolgreiche Ausführungen definieren, an diesen ist man i.a. aber nicht interessiert, so daß wir diesen Fall nicht weiter betrachten.

mengen von Rücksetzungen zu berechnen braucht, sondern sich wegen der linearen Ordnung auf den ungünstigsten Fall beschränken kann. Dieser ungünstigste Fall sieht wie folgt aus:

- bei einer Inkrementierung:
  für die obere Grenze: kein Rollback
  für die untere Grenze: Rollback
- bei einer Dekrementierung:
  für die obere Grenze: Rollback
  für die untere Grenze: kein Rollback

Zu einem numerischen Objekt X seien **deltaMin** und **deltaMax** die Differenzen zu den Werten von X, die infolge von Rücksetzungen im ungünstigsten Fall eintreten können. Die beiden Werte stellen den Unsicherheitsbereichen nach oben bzw. unten dar und werden wie folgt berechnet:

- bei einer Inkrementierung incr(a) ($a \geq 0$):
  deltaMax bleibt unverändert
  deltaMin := deltaMin + a
- bei einer Dekrementierung decr(a) ($a \geq 0$)::
  deltaMax := deltaMax + a
  deltaMin bleibt unverändert

Sobald eine Transaktion endet (Commit oder Rollback), werden für alle von dieser Transaktion durchgeführten Modifizierungen die vorstehenden Änderungen von deltaMax bzw. deltaMin wieder rückgängig gemacht.

Seien XMin und XMax die minimal bzw. maximal für X zulässigen Werte. Zugelassen wird incr(a) nur noch dann, wenn

$$X + \text{deltaMax} + a \leq \text{XMax}$$

ist. Analog dazu wird decr(a) nur noch dann, wenn

$$X - \text{deltaMin} - a \geq \text{XMin}$$

ist.

5.5 Konfliktfreiheit mit Parametereinschränkungen

Das folgende Beispiel zeigt zwei parametrisierte Modifikationen, die nicht für alle Parameterwerte semantisch konfliktfreie Modifikationen ergeben: Wertebereich seien Mengen über einer Basismenge (z.B. set of char), Funktionale sind das Hinzufügen eines Elements (oder einer Menge) zu einer Menge und das Wegnehmen im Sinne der unsymmetrischen Differenz. Unter der Bedingung, daß als Parameter verschiedene Elemente (bzw. disjunkte Mengen) verwendet werden, sind die beiden entstehenden Modifikationen semantisch konfliktfrei, sonst nicht.

Für eine genauere Definition dieses Sachverhalts bilden wir wieder zu einer gegebenen Menge PM von (parametrisierten) Modifikationen die Gesamtmenge der zugehörigen Modifikationen. Wenn sich eine Menge von Paaren von konfliktfreier Modifikationen aus dieser Gesamtmenge durch eine Bedingung an die Parameter, die in den beteiligten Modifikationen gelten, angeben läßt, dann heißt PM **(semantisch) konfliktfrei mit Parametereinschränkungen**.

Im obigen Beispiel bestand PM aus "einfügen(x)" und "ausfügen(y)", die Parametereinschränkung war $x \neq y$.

Die Menge der Paare in der obigen Definition sollte eine sinnvolle Größe haben, also in der gleichen Größenordnung wie das Quadrat der Größe der Gesamtmenge der Modifikationen liegen; letztlich ist diese Bewertung etwas subjektiv und auch von der Häufigkeit des Auftretens einzelner Parameter abhängig. Als Negativbeispiel sei genannt: incr(x) und mult(y) sind mit der Parametereinschränkung $x=0$ oder $y=1$ konfliktfrei.

Einheitliche Sperrmodi für die Gesamtmenge von Modifikationen sind nun leider nicht mehr anwendbar. Wir müssen daher zu modifikationsbezogenen Sperrmodi zurückkehren. (In gewissen Fällen kann die Zahl der Sperrmodi aber verringert werden, indem man gleichwertige Modi geschickt zusammenfaßt.) In der Kompatibilitätsmatrix notieren wir anstelle von + oder - die Bedingung, die die Parameter der Modifikationen erfüllen müssen.

vorhandene Sperre:	beantragte Sperre: S	X	insert(a)	remove(b)
S	+	-	-	-
X	-	-	-	-
insert(c)	-	-	+	b≠c
remove(d)	-	-	a≠d	+

Beispielsweise darf eine Sperre in Modus "remove(x)" nicht zugeteilt werden, solange eine Sperre im Modus insert(x) für diese Menge besteht.

Die Realisierung solcher Sperren erfordert flexiblere Sperroperationen als bisher; genauer müssen neben den bisherigen Sperrmodi (S, X, IS, ...) Darstellungen für alle Modifikationen incl. Parameterwerte verarbeitet werden können. Hierzu müssen die Datenstrukturen in einer Sperrtabelle erweitert werden. Die Kompatibilität von Sperren muß durch spezielle Algorithmen festgestellt werden, die vom *Benutzer* (bzw. dem Programmierer der Modifikationen) zu liefern sind. Dieser Algorithmus benötigt als Eingabedaten die Namen der Modifikationen und ggf. deren Parameter. Daher bietet es sich an, diese Angaben direkt in der Sperrtabelle zu speichern. Die Sperroperation hätte dann drei Parameter:

1. Objektidentifikation
2. Identifikation der Modifikation
3. ggf. Parameter der Modifikation

Derartige äußere Eingriffe in die Sperrenverwaltung werden in den meisten Fällen völlig undenkbar sein; selbst wenn sie in einem speziellen Fall zulässig sind, bleibt das Effizienzproblem: Die Sperroperation darf trotz der deutlich höheren Flexibilität nicht wesentlich ineffizienter werden, sonst wird je nach den Umständen insgesamt kein Gewinn an Performance durch diese Sperrmodi erzielt.

5.6 Konfliktfreiheit mit Objektzustandseinschränkungen

Beim letzten Beispiel hing die semantische Konfliktfreiheit zweier parametrisierter Modifikationen von den Parametern ab; im nächsten Beispiel wird sie statt dessen vom Zustand des modifizierten Objekts abhängen. Der Typ des Objekts sei hier eine (Warte-) Schlange über irgendeinem Basistyp; wir betrachten nur die Operationen:

append(x)
: ein Element mit Inhalt x hinten an die Schlange anhängen

remove():x
: eine Element vorne entnehmen; x ist Rückgabewert. Ist die Schlange leer, wird ein Fehlercode zurückgegeben.

Zwei append-Operationen sind nicht konfliktfrei, da die Reihenfolge der Anfügungen relevant ist. Zwei remove-Operationen sind ebenfalls nicht konfliktfrei, da i.a. verschiedene Werte in den Elementen der Schlange enthalten sind und der gelesene Wert bei remove zur Sicht dieser Operation zählt!

Je eine append- und remove-Operation sind offensichtlich genau dann konfliktfrei, wenn die Schlange nicht leer ist. Hierbei handelt es sich um eine neue Bedingung an die semantische Konfliktfreiheit, die unabhängig von Bedingungen an die Parameter von parametrisierten Modifikationen auftreten kann und die auch für nichtparametrisierte Modifikationen sinnvoll ist. Wir definieren daher:

- Zwei (nichtparametrisierte) Modifikationen heißen **(semantisch) konfliktfrei mit Objektzustandseinschränkung P**, wenn ihre beiden seriellen Ausführungen dieselbe Funktion ergeben, vorausgesetzt der Zustand des modifizierten Objekts erfüllt zu Beginn ein Prädikat P.
- Zwei parametrisierte Modifikationen heißen **(semantisch) konfliktfrei mit Objektzustandseinschränkung P**, wenn alle zugehörigen Paare von Modifikationen konfliktfrei mit Objektzustandseinschränkung P sind.

- Zwei parametrisierte Modifikationen heißen **(semantisch) konfliktfrei mit Parametereinschränkung P1 und Objektzustandseinschränkung P2**, wenn für alle Paare von Modifikationen, die unter Einhaltung der Bedingung P1 an die Parameter abgeleitet werden können, gilt, daß deren beide Hintereinanderausführungen dieselbe Funktion ergeben, vorausgesetzt der Zustand des modifizierten Objekts erfüllt zu Beginn ein Prädikat P2.

Man kann nun analog zum letzten Beispiel eine Verträglichkeitsmatrix, z.B. für append und remove, konstruieren, in der Bedingungen auftreten, die sich auf den Zustand des Objekts beziehen. Die obigen Bemerkungen zu solchen Einträgen gelten hier verstärkt. Hinzu kommen allerdings weitere Probleme mit derartigen Modifikationen und ihren Invertierungen, die es zweifelhaft erscheinen lassen, ob sie wirklich praktisch verwertbar sind:

1. Die erforderlichen inversen Modifikationen, z.B. das Zurückstellen eines Eintrags “vorne” in eine Schlange, können i.d.R. nicht durch schon vorhandene Modifikationen realisiert werden; stattdessen sind zusätzliche Operationen zu realisieren, die ggf. eine völlige Neuspezifikation und Neuimplementierung des Typs erforderlich machen.
2. Es können unsichere Teilobjekte entstehen. So kann eine Transaktion T1 ein Element in die Schlange einfügen, Transaktion T2 will dieses Element entnehmen, bevor T1 beendet ist. Wenn nun T1 zurückgesetzt wird, muß auch T2 zurückgesetzt werden. Wie schon in Lehrmodul 1 erwähnt ist die Fortpflanzung von Rollback äußerst problematisch.

   Anders gesagt ist bei unsicheren Teilobjekten das Ausfügen nicht mit der Invertierung des Einfügens konfliktfrei[17].

[17]Hierbei handelt es sich übrigens um ein allgemeineres Bereichsüberwachungsproblem: Der Wert des Objekts ist durch eine erste Modifikation verändert worden, anschließend durch eine konfliktfreie Modifikation, und hat einen Wertebereich erreicht, in dem die Invertierung der ersten Modifikation nicht mehr anwendbar ist. In unserem Beispiel kann das unsicher eingefügte Element nicht mehr ausgefügt werden, weil es schon von der anderen Transaktion ausgefügt ist.

3. Alle bisherigen Sperrmodi berechtigten dazu, die zulässigen Operationen beliebig oft auszuführen, die Atomarität der Transaktion ist hierdurch nicht gefährdet. Bei konfliktfreien Modifikationen galt dies deshalb, weil der Objektzustand keine Rolle spielte. Im Gegensatz dazu kann eine Bedingung an den Objektzustand vor einer zulässigen Modifikation erfüllt, danach aber verletzt sein. Z.B. ist die Bedingung "Schlange nicht leer" erfüllt, wenn die Schlange genau ein Element enthält; nach Ausführung von remove ist sie es nicht mehr.

Lehrmodul 6:

Zeitstempelverfahren

Zusammenfassung dieses Lehrmoduls

Zeitstempelverfahren sind sog. validierende Concurrency-Control-Verfahren, bei denen inkorrekte Verzahungen von Transaktionsausführungen verhindert werden, indem abhängig von Validationstests einzelne Transaktionen zurückgesetzt und neugestartet werden. Die Validationstests werten die Zeitpunkte von Zugriffen zu Objekten aus. Dieses Lehrmodul erläutert zunächst das Validationsprinzip und seine Vor- und Nachteile. Weiter wird die Grundform der Zeitstempelverfahren vorgestellt, ferner Techniken zur Erzeugung und Verwaltung von Zeitstempeln. Eine Vereinigung der Vorteile von Sperr- und Zeitstempelverfahren ist in einigen kombinierten Verfahren gelungen, die anschließend vorgestellt werden.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Sperrverfahren

empfohlen:
- Recovery

Stoffumfang in Vorlesungsdoppelstunden: 1.0

©2001 Udo Kelter

6.1 Validierende Verfahren

6.1.1 Verhinderung inkorrekter Verzahnungen durch Neustart

Bei Sperrverfahren werden inkorrekte Verzahnungen dadurch verhindert, daß einzelne Zugriffe von Transaktionen verzögert werden. Technisches Mittel zum Verzögern waren Sperren, auf deren Freigabe eine Transaktion ggf. warten muß. Infolge des Wartens können Deadlocks auftreten. Deadlocks können i.a. nicht durch präventive Maßnahmen verhindert, sondern nur entdeckt und aufgelöst werden (s. Lehrmodul 3). Für diesen Zweck sind zum einen Software-Komponenten innerhalb der Concurrency-Control-Komponente vorzusehen, ferner Datenstrukturen, die eine effiziente Deadlock-Erkennung ermöglichen (Wartegraph). Außerdem müssen laufend vorbereitende Maßnahmen zur Deadlock-Erkennung durchgeführt und im Falle eines Deadlocks gewisse Transaktionen zurückgesetzt werden.

Die Maßnahmen zur Deadlock-Behandlung sind schon bei zentralen Datenbanken aufwendig und natürlich unerwünscht. Bei verteilten Datenbanken ist schon die Deadlock-Erkennung äußerst aufwendig: Der Wartezyklus kann sich über verschiedene Rechner erstrecken, so daß zu seiner Erkennung systemweite (und sehr teure) Kommunikation erforderlich wird. Gesucht sind also deadlockfreie Verfahren, bei denen die Transaktionen nicht vorab deklarieren müssen, zu welchen Objekten sie insgesamt zugreifen werden. Dies ist die Hauptmotivation für sog. **validierende CC-Verfahren**.

Die Grundidee dieser Verfahren besteht darin, überhaupt nicht zu warten, sondern inkorrekte Verzahnungen durch Rücksetzen einzelner Transaktionen zu verhindern. Diese Verfahren arbeiten nach folgendem Prinzip:

- Alle Zugriffe werden, wenn überhaupt, dann sofort ausgeführt.
- Bei bestimmten Gelegenheiten, z.B. bei jedem Zugriff oder am Ende jeder Transaktion, werden **Validationstests** durchgeführt: es wird überprüft, ob die bisherige effektiv eingetretene Verzahnung korrekt

(also mindestens cp-serialisierbar) ist. Falls nicht, wird die Transaktion zurückgesetzt und automatisch neugestartet. Dies bezeichnen wir in diesem Kapitel kurz mit **Neustart**. Der letzte (und oft einzige) Validationstest findet im Rahmen des Commits statt. Führt er nicht zum Neustart, ist die Transaktion insgesamt **validiert**. (Üblich ist auch die Bezeichnung "zertifiziert".)

Durch dieses Prinzip wird natürlich die Häufigkeit von Rollback merklich erhöht. In der Konsequenz ist es dringend geboten, Fortpflanzung von Rollback zu verhindern. Daher nehmen wir bei allen folgenden Verfahren das verzögerte Schreiben (*deferred update*) an[18], wobei veränderte Objekte erst bei Commit in die Datenbank geschrieben und vorher gepuffert werden. Aus diesem Grund gibt es keine ungesicherten Werte in der Datenbank, alle geschriebenen Werte sind von validierten Transaktionen geschrieben worden.

6.1.2 Vergleich mit Sperrverfahren

Im Vergleich zum Sperren hat das Validieren folgende Vor- und Nachteile:

- Die validierenden Verfahren sind deadlockfrei. Weder Erkennungsmechanismen (Software) noch vorbereitende Maßnahmen zur Deadlock-Erkennung sind erforderlich. Ebenso entfallen Kosten der Deadlock-Auflösung.
- Die Vermehrung des Rollbacks von Transaktionen ist – unabhängig von der vereinfachten Realisierung des Rollbacks durch das verzögerte Schreiben – ein Nachteil.
- Zur Realisierung des verzögerten Schreibens müssen lokale Kopien der Objekte vorgesehen werden, bei der Realisierung durch Sperren ist dies nicht erforderlich. Validierende Verfahren sind daher wenig für Anwendungsfälle geeignet, bei denen Transaktionen viele Objekte schreiben.

[18] S. auch Ausführungen zum verzögerten Schreiben in Abschnitt 2.4.4 in Lehrmodul 2.

- Wenn eine Transaktion abgebrochen und neugestartet worden ist, kann ihr dieses “Pech” beim nächsten Ausführungsversuch erneut widerfahren. Frühere Neustarts einer Transaktion verbessern die Wahrscheinlichkeit nicht, daß diese Transaktion bei ihrem nächsten Ausführungsversuch nicht noch einmal neugestartet wird. Bei den Grundformen der validierenden Verfahren kann (zumindest theoretisch) ein zyklischer Neustart eintreten, d.h. eine Transaktion wird endlos neugestartet. Betroffen sind vor allem längere Transaktionen, denn die Wahrscheinlichkeit eines Neustarts steigt mit der Länge einer Transaktion. Alle Maßnahmen zur Lösung des Neustartproblems beruhen in irgendeiner Weise auf Sperren, allerdings so, daß keine Deadlocks möglich sind. Die Verfahren sind dann allerdings wieder in Sperren involviert, es müssen Mechanismen zur Realisation der Sperren vorgesehen werden, und die Verfahren werden relativ kompliziert.
- Bezüglich des Aufwands für die Hilfsdaten ist keine pauschale Aussage zugunsten von Neustarts oder Sperrungen möglich. Bei validierenden Verfahren entfällt zwar die Sperrtabelle (sofern Sperren zur Lösung des Neustart-Problems nicht doch wieder eingeführt werden), dafür sind aber andere Hilfsdaten für die Validationstests erforderlich, die bzgl. Speicherplatzbedarf und Handhabungsaufwand durchaus mit der Sperrtabelle vergleichbar sein können.

Eine quantitative Abwägung der Vor- und Nachteile einzelner Verfahren (z.B. mit stochastischen Modellen) ist wegen der hohen Komplexität und Vielfalt der Einflußfaktoren sehr schwierig und gleichzeitig problematisch. Das CC-Verfahren ist nur einer von mehreren wesentlichen Einflußfaktoren, die die Gesamtleistung eines DBS bestimmen. Wir beschränken uns deshalb im folgenden auf einen qualitativen Vergleich wichtiger Einzelaspekte der Verfahren; es sei noch einmal davor gewarnt, von der Überlegenheit eines Verfahrens in Einzelaspekten auf eine generelle Überlegenheit unter beliebigen Einsatzbedingungen zu schließen (einige triviale Fälle einmal ausgenommen). Für eine Auswahl eines Verfahrens in einem konkreten Fall sind darüber hinaus noch andere Aspekte wesentlich, z.B. die Komplexität bzw. der

Realisierungsaufwand für die Software.

6.1.3 Varianten validierender Verfahren

Das Grundprinzip des Validierens läßt Raum für viele Varianten. Die beiden wichtigsten Arten von validierenden CC-Verfahren sind:

- Zeitstempel-Verfahren und
- optimistische Verfahren.

Die Verfahren unterscheiden sich vor allem in folgenden Punkten:

***Zeitpunkt der Validationstests*:** Wann werden Validationstests durchgeführt?

***Korrektheitsbegriff*:** Welche eingetretenen Verzahnungen werden als korrekt erachtet und welche nicht?

***Hilfsdaten*:** Welche Hilfsdaten werden für die Entscheidung benutzt?

Alle drei Aspekte hängen natürlich zusammen; wir wollen jedoch schon vorab einige Alternativen isoliert besprechen.

Zum Zeitpunkt der Validationstests: Sinnvolle Gelegenheiten für Validationstests sind neben dem Commit, das alle Schreib-Aktionen enthält, nur einzelne frühere Lese-Aktionen. Zeitstempel-Verfahren führen bei jedem Zugriff einen Test durch, optimistische Verfahren nur bei Commit.

Für die frühen Tests bei Leseaktionen spricht, daß inkorrekte effektive Verzahnungen früher erkannt werden können. Durch das Rücksetzen der Transaktion geht dann weniger geleistete Arbeit verloren, die Tests sind relativ einfach, allerdings häufig.

Bei späten Tests kann theoretisch das gesamte Geschehen im Verlauf der Transaktionsausführung berücksichtigt werden, es können also unnötige Neustarts und viele Einzeltests vermieden werden. Jedoch steigt die Komplexität der Tests dann ganz erheblich, so daß der Testaufwand eher größer ist als bei frühen Tests.

Zum Korrektheitsbegriff und den Hilfsdaten: In Lehrmodul 3 hatten wir eine Verzahnung als serialisierbar definiert, wenn es zu jeder enthaltenen Transaktion einen Serialisierungspunkt gibt, d.h. alle Ereignisse einer Transaktion können konfliktfrei zu ihrem Serialisierungspunkt verschoben werden. Dieser Korrektheitsbegriff ist allerdings für die Verhältnisse, die bei validierenden Verfahren unterstellt werden, aus Aufwandsgründen weniger geeignet.

Hierzu müssen wir zunächst die Korrektheitsbegriffe für Verzahnungen etwas näher betrachten. Serialisierungspunkte waren so motiviert, daß hier scheinbar die gesamte Transaktion stattfinden kann. Wenn wir alle Aktionen dementsprechend verschieben, erhalten wir einen **seriellen** Ablauf. Die aufgetretene Verzahnung und dieser serielle Ablauf sind dann **äquivalent** in dem Sinne, daß sie beide die gleiche Menge von Aktionen umfassen und daß, dann, wenn zwei Aktionen in Konflikt miteinander stehen, diese beiden Aktionen in beiden Abläufen in der gleichen Reihenfolge auftreten, also nicht vertauscht werden. Die vorstehende Definition der Äquivalenz zweier Abläufe wird auch **cp-Äquivalenz** (*conflict-preserving equivalence*) genannt. Es gibt noch andere Definitionen der Äquivalenz von Abläufen, auf die wir hier aber nicht eingehen.

Unserer bisherige Definition von Serialisierbarkeit ist mit diesen Begriffen äquivalent zu der folgenden: Eine Verzahnung ist **serialisierbar**, wenn es einem cp-äquivalenten seriellen Ablauf gibt.

Theoretisch kann für jeden Korrektheitsbegriff ein validierendes Verfahren entwickelt werden: Die bis zum Testzeitpunkt abgelaufene Verzahnung muß in ihren relevanten Details bekannt sein, d.h. entsprechende Daten sind laufend zu speichern; getestet wird bei jeder Aktion oder am Ende einer Transaktion, ob die entstandene Verzahnung Präfix einer korrekten Verzahnung ist, wobei mit dem Rücksetzen aller noch nicht beendeten Transaktionen gerechnet werden muß. Fällt der Test negativ aus, wird die Transaktion, die den Test verursachte, zurückgesetzt und automatisch neugestartet.

Wegen der Häufigkeit, mit der die Tests durchgeführt werden, ist

deren Effizienz sehr kritisch[19]. Wenn man nun entscheiden will, ob ein gegebener Ablauf serialisierbar ist, muß man prüfen, ob ein cp-äquivalenter serieller Ablauf existiert oder nicht. Hierfür sind keine ausreichend effizienten Verfahren verfügbar.

Bei den praktisch brauchbaren Verfahren wird ein Trick angewandt, durch den die Korrektheitsüberprüfung stark vereinfacht wird. Für die Serialisierbarkeit reicht es aus, wenn es *irgendeinen* seriellen Ablauf gibt, der äquivalent zur vorhandenen Verzahnung ist. Die Vereinfachung besteht darin, *einen ganz bestimmten* seriellen Ablauf vorzugeben. Vorgegeben wird natürlich ein serieller Ablauf, der mit möglichst geringer Wahrscheinlichkeit zu einem Neustart führt. In etwa wird dieser serielle Ablauf gemäß den Ankunftszeitpunkten der Transaktionen gebildet.

Scheduling. Eine Umformung einer Aufrufsequenz von Aktionen in eine effektive Ausführungssequenz (s. Abschnitt 3.2) findet bei validierenden Verfahren nicht in dem Sinne wie bei Sperrverfahren statt: Die Aufrufsequenz wird durch das Rücksetzen selbst verändert. Sofern keine Transaktion zurückgesetzt wird, ist die Aufrufsequenz gleichzeitig effektive Ausführungssequenz. Unter der **effektiven Ausführungssequenz** verstehen wir daher in diesem Lehrmodul die letztlich wirksame Aufrufsequenz. Die Umformung der "ursprünglichen" Aufrufsequenz in die effektive Ausführungssequenz ist nicht reproduzibel, denn die Zeit bis zum Neustart einer Transaktion ist zufällig.

6.2 Zeitstempel-Verfahren

6.2.1 Die Grundform

Bei der Grundform der Zeitstempel-Verfahren wird bei jedem Zugriff ein Validationstest durchgeführt. Getestet wird, ob die bisherige effektive Ausführungssequenz cp-äquivalent zu dem seriellen Ablauf ist, der sich aus der Ankunftszeit der Transaktionen ergibt. Für zurückgesetzte Transaktionen ist der Zeitpunkt ihres Neustarts relevant.

[19]Anmerkung: bei Sperrverfahren werden solche Tests überhaupt nicht durchgeführt, die entstehenden Verzahnungen sind automatisch serialisierbar.

Das wichtigste technische Hilfsmittel sind **Zeitstempel**. Jede Transaktion T_i erhält von der CC-Komponente einen eigenen, eindeutigen Zeitstempel, **Z(T_i)**. Den Zeitstempel erhält die Transaktion bei ihrer Ankunft oder später, spätestens vor ihrem ersten lesenden Zugriff. Einen Zeitstempel stellen wir uns vorerst am besten als die aktuelle Uhrzeit, ggf. verbunden mit dem Datum, vor, und zwar in einer Genauigkeit, daß keine zwei Transaktionen den gleichen Zeitstempel und eine spätere Transaktion einen "größeren" Zeitstempel erhalten. Wir werden "Zeitstempel" und "Zeitpunkt" als Synonyme benutzen.

Der Zeitstempel einer Transaktion ist der vorgegebene Serialisierungspunkt dieser Transaktion in der entstehenden effektiven Ausführungssequenz. Alle Ereignisse in der effektiven Ausführungssequenz müssen also konfliktfrei zu den jeweiligen Zeitstempeln der Transaktionen verschoben werden können. Diese Verschiebung ist nur dann nicht möglich, wenn folgende typische Situation vorliegt:

```
Ti Z------------ei
Tj          Z-----ej
```

In diesem und den folgenden Beispielen benutzen wir eine graphische Notation für Abläufe, in der das Symbol 'Z' den Zeitstempel darstellt, `r(x)` und `w(x)` das Lesen bzw. Schreiben eines Objekts x.

Im obigen Beispiel seien e_i und e_j zwei Ereignisse, die in Konflikt stehen. e_i kann nicht konfliktfrei zum Zeitstempel von T_i verschoben werden, denn e_j müßte hierfür vor den Zeitstempel von T_j verschoben werden.

Zwischen e_i und e_j können drei verschiedene Arten von Konflikten bzgl. eines Objekts x bestehen:

Lese-Schreib-Konflikt:

```
Ti Z--------r(x) ....
Tj          Z--w(x)
```

Schreib-Lese-Konflikt:

```
Ti Z--------w(x)
Tj          Z--r(x)
```

Schreib-Schreib-Konflikt:

```
Ti Z--------w(x)
Tj          Z--w(x)
```

Da wir verzögertes Schreiben annehmen, kann nach einem Schreibereignis einer Transaktion kein weiteres Ereignis von dieser Transaktion mehr stattfinden. Bei Lese-Ereignissen können jedoch noch weitere Ereignisse geplant sein, was durch "..." dargestellt ist.

Bei jedem Zugriff e_i einer Transaktion T_i muß folgender Validationstest durchgeführt werden: T_i ist abzubrechen, wenn es eine *jüngere* Transaktion T_j (d.h. der Zeitstempel von T_j ist größerer als der von T_i) gibt und wenn (a) T_j schon *vor* e_i einen Zugriff e_j durchgeführt hat und (b) e_j in *Konflikt* mit e_i steht. Andernfalls kann der Zugriff ausgeführt werden.

Zur Vereinfachung der Suche nach einem derartigen T_j sehen wir bei jedem Objekt zwei Zeitstempel vor:

`ZR(x)`: Zeitstempel der jüngsten Transaktion, die x *gelesen* hat (m.a.W. gibt es ein Leseereignis, das nicht vor diesen Zeitpunkt verschoben werden kann).

`ZW(x)`: Zeitstempel der jüngsten Transaktion, die x *geschrieben* hat (m.a.W. gibt es ein Schreibereignis, das nicht vor diesen Zeitpunkt verschoben werden kann).

Bei jedem erfolgreichen Zugriff müssen die Hilfsvariablen `ZR` und `ZW` auf den neuen Stand gebracht werden. Diese Aktualisierung führen wir zusammen mit dem Validationstest aus. Insgesamt erhalten wir folgende Testalgorithmen:

- T_i will x lesen (Test auf Lese-Schreib-Konflikt; vor Commit):

```
IF Z(Ti) < ZW(x)
   THEN NEUSTART
   ELSE ZR(x) := max ( ZR(x), Z(Ti) )   (* und Zugriff *)
```

- T_i will x schreiben (Test auf Schreib-Lese- oder Schreib-Schreib-Konflikt; nur während der Commit-Verarbeitung):

```
IF Z(Ti) < ZR(x)                      (* Schreib-Lese-Konflikt *)
OR Z(Ti) < ZW(x)                   (* Schreib-Schreib-Konflikt *)
```

```
    THEN NEUSTART
    ELSE ZW(x) := Z(T_i)                    (* und Zugriff *)
```

Die Zeitstempel an den Objekten ermöglichen einen sehr effizienten Validationstest, der bei dem ursprünglichen Korrektheitsbegriff Serialisierbarkeit nicht möglich wäre.

Das obige Kommando `NEUSTART` ist so zu interpretieren, daß T_i zurückgesetzt und anschließend automatisch neugestartet wird. Dabei erhält T_i einen *neuen* Zeitstempel.

Wir nehmen hier an, daß alle Validationstests automatisch innerhalb der Aktionen durchgeführt werden und nicht etwa von Hand durch den Programmierer. Alle Validationstests müssen unter wechselseitigem Ausschluß auf den betroffenen Daten durchgeführt werden.

6.2.2 Zyklischer Neustart

Eine Transaktion, die zurückgesetzt und mit neuem Zeitstempel neugestartet wurde, fängt völlig gleichberechtigt mit allen anderen noch vorhandenen Transaktionen von vorne an. Sie kann daher wieder Opfer eines Validationstests werden. In der Grundform des Zeitstempel-Verfahrens gibt es keine Möglichkeit, die Zahl der Neustarts zu begrenzen. Im schlimmsten Fall wird eine Transaktion immer wieder wegen neu hinzukommender Transaktionen abgebrochen[20]. Vor allem lange Transaktionen, die viele Lesezugriffe durchführen, können leicht das Opfer von kurzen schreibenden Transaktionen werden.

Wenn man die Wahrscheinlichkeit eines einzelnen Neustarts einer Transaktion als fest annimmt, dann werden häufige Neustarts zwar immer unwahrscheinlicher. Bei langen Transaktionen ist jedoch die Wahrscheinlichkeit eines einzelnen Neustarts relativ hoch. In jedem Fall muß mit unvertretbaren Ausreißern bei der Zahl der Neustarts gerechnet werden.

Unter gewissen Umständen können zwei Transaktionen immer wieder gegenseitig ihren Neustart verursachen. Beispiel:

[20] Ein ähnliches Problem besteht bei der Deadlock-Auflösung in Sperrverfahren.

```
T1  Z-r(x)----w(y)N
T2          Z-r(y)-------w(x)N
T1_1                          Z-r(x)-------w(y)N
T2_1                                        Z-r(y)-------w(x)N
T1_2                                                      Z-r(x)--...
```

In diesem Beispiel wurde ein Zugriff, dessen Validationstest negativ ausging und zu einem Neustart führte, durch "N" dargestellt. Die neu gestarteten Transaktionen erhielten den Index 1, 2 usw. Das Beispiel läßt sich endlos fortführen, hier liegt ein zyklischer Neustart vor, bei dem nicht ständig neue Transaktionen hinzukommen müssen.

Wiederholter oder zyklischer Neustart sind ein Problem aller validierenden Verfahren. Bei Sperrverfahren existierte dieses Problem praktisch nicht. Daher wurden Sperrverfahren und validierende Verfahren kombiniert, um sowohl Deadlock-Freiheit wie Begrenzung der Neustarts zu erreichen. Einige Ansätze werden wir später behandeln.

6.2.3 Zeitstempel

Zeitstempel können einer Transaktion irgendwann zwischen ihrem Start durch ein Benutzerprogramm und ihrer ersten Aktion zugewiesen werden. Für unsere Betrachtungen ist die Zeit vor der Zuteilung des Zeitstempels irrelevant, d.h. für uns beginnt jede Transaktion mit der Zeitstempelzuteilung.

Zeitstempel waren oben zunächst eingeführt worden als Angabe einer Uhrzeit mit ausreichender Genauigkeit. Für Zeitstempel sind folgende Eigenschaften wesentlich:

1. Zwei Transaktionen haben nie den gleichen Zeitstempel.
2. Nichtüberlappende Transaktionen erhalten Zeitstempel in aufsteigender Reihenfolge.
3. Nachdem eine Transaktion ihren Zeitstempel erhalten hat, erhalten anschließend höchstens endlich viele andere Transaktionen einen kleineren ("älteren") Zeitstempel.

Die dritte Eigenschaft ist bei der Methode "Uhrzeit" in einer zentralen Datenbank trivialerweise erfüllt: Es gibt überhaupt keine Transakti-

on, die zukünftig einen kleineren Zeitstempel erhält. Die Abschwächung auf "endlich viele" ist wichtig für mehrere Prozessoren mit eigener Uhr, besonders also für verteilte Datenbanken, für die Zeitstempel-Verfahren ursprünglich entwickelt wurden und in denen u.U. nicht alle lokalen Uhren ausreichend synchronisiert werden können.

In zentralen Datenbanken bietet sich eine andere Lösung als die Uhrzeit an: Transaktionsnummern. Hierzu ist ein zentraler Transaktionszähler zu installieren, der bei jeder neuen Transaktion um eins erhöht wird. Transaktionsnummern haben gegenüber Uhrzeitangaben den Vorteil, daß zu ihrer Speicherung weniger Platz gebraucht wird.

6.2.4 Verwaltung der Zeitstempel

Gleichgültig, ob Uhrzeiten oder Transaktionsnummern als Zeitstempel verwandt werden, ist es nicht sinnvoll, bei jedem Objekt zwei Hilfsvariablen passender Größe zur Speicherung der Zeitstempel `ZR` und `ZW` fest einzurichten: Der Platzbedarf wäre sehr hoch, während andererseits nur ein Bruchteil dieser Zeitstempel wirklich benötigt wird. Benötigt werden nur noch solche Zeitstempel, die größer sind als der Zeitstempel der derzeit ältesten Transaktion, denn nur solche Zeitstempel können noch zum Rücksetzen einer Transaktion führen.

Betrachten wir zunächst die Lese-Zeitstempel getrennt. Die Lösung besteht in einer separaten Tabelle, die Einträge der Form (x, `ZR(x)`) enthält, allerdings nur für diejenigen Objekte x, deren `ZR` -Zeitstempel noch benötigt wird. Nicht mehr benötigte Einträge der Tabelle werden jeweils gelöscht. Zu der Tabelle gehört eine weitere Variable R_{min}, deren Wert kleiner als alle noch vorhandenen Zeitstempel in der Tabelle und größer als alle schon gelöschten Zeitstempel ist. Abfragen bzw. Änderungen von `ZR(x)` werden mit Hilfe der Tabelle wie folgt durchgeführt:

- Abfrage von `ZR(x)` : Die Tabelle wird nach einem Eintrag (x,z) durchsucht. Wird ein solcher Eintrag gefunden, so wird z zurückgegeben, andernfalls R_{min}.
- Änderung von `ZR(x)` : Die Tabelle wird nach einem Eintrag (x,z)

durchsucht. Wird er gefunden, so wird z durch den neuen Wert ersetzt, andernfalls wird ein neuer Eintrag eingefügt.

Bei bestimmten Gelegenheiten, z.B. bei Überlauf der Tabelle, wird ein neuer Wert für R_{min} festgelegt. Dieser Wert muß auf jeden Fall kleiner als der kleinste Zeitstempel aller noch aktiven Transaktionen sein. In der Tabelle werden dann alle Einträge (x,z) mit $z < R_{min}$ gelöscht.

Für die `ZW` -Zeitstempel muß eine weitere Tabelle mit einer zugehörigen Variablen W_{max} eingerichtet werden. Beide Tabellen können auch zusammengelegt werden. Diese Zeitstempel-Tabellen sind in einigen Details der Sperrtabelle sehr ähnlich.

6.3 Kombinierte Sperr- und Zeitstempel-Verfahren

Einige Verfahren benutzen sowohl Sperren wie auch Zeitstempel-gesteuertes Rollback mit Neustart. Man kann sie als Sperrverfahren auffassen, in denen Zeitstempel-gesteuertes Rücksetzen zur Verhinderung von Deadlocks benutzt wird.

In [StLR76] und [RoSL78] wurden Verfahren vorgeschlagen, die das Deadlock- bzw. Neustart-Problem lösen, indem älteren Transaktionen höhere Priorität eingeräumt wird als jüngeren. Zunächst müssen Transaktionen das 2-Phasen-Protokoll befolgen. Jede Transaktion erhält bei ihrer Ankunft im System eine Zeitmarke. Sie behält diese Zeitmarke auch im Falle eines Neustarts (im Gegensatz zur Grundform der Zeitstempelverfahren)!

Die von den Zeitstempelverfahren her bekannten Lese- und Schreib-Zeitstempel an Objekten (`ZW(x)` und `ZR(x)`) werden nicht mehr direkt in dieser Form benutzt, jedoch indirekt durch Zeitstempel von Transaktionen, die in der Sperrtabelle verwaltet werden müssen.

Abweichend vom 2-Phasen-Protokoll gelten in dem Falle, daß T_1 eine Sperre beantragt, die nicht verträglich ist mit einer schon vorhandenen Sperre auf diesem Objekt, welche von T_2 gehalten wird, folgende Regeln:

wait-die-Verfahren:

- Wenn T_1 älter als T_2 ist, dann wartet T_1 auf die Freigabe der vorhandenen Sperre (“*wait*”).
- Wenn T_1 jünger als T_2 ist, dann wird T_1 neu gestartet (“*die*”).

wound-wait-Verfahren:

- Wenn T_1 älter als T_2 ist, dann wird T_2 neu gestartet; T_2 wird sozusagen von T_1 tödlich verwundet (“*wound*”).
- Wenn T_1 jünger als T_2 ist, dann wartet T_1 auf die Freigabe der vorhandenen Sperre (“*wait*”).

Das erste Stichwort in wait-die und wound-wait deutet an, was passiert, wenn eine ältere Transaktion auf eine jüngere warten soll, das zweite Stichwort den anderen Fall.

Deadlockfreiheit. Der Hauptunterschied zwischen beiden Verfahren ist die zulässige Wartebeziehung zwischen jüngeren und älteren Transaktionen:

- Beim wait-die-Verfahren wartet immer eine ältere Transaktion auf eine jüngere, nie umgekehrt.
- Beim wound-wait-Verfahren wartet immer eine jüngere Transaktion auf eine ältere, nie umgekehrt.

In beiden Fällen geben die Zeitstempel der Transaktionen eine lineare Ordnung vor (aufsteigende oder fallende Reihenfolge der Zeitstempel), in der sich die Wartebeziehungen aller Transaktionen bewegen können. Ein Wartezyklus ist somit unmöglich, die Verfahren sind deadlockfrei.

Neustarts. Gemeinsam ist beiden Verfahren, daß immer die *jüngere* Transaktion zurückgesetzt wird, wenn eine unzulässige Wartebeziehung droht, nie die ältere. Zyklische Neustarts sind damit ebenfalls unmöglich. Wesentlich hierbei ist, daß – im Gegensatz zur Grundform der Zeitstempel-Verfahren – eine Transaktion beim Neustart ihren ursprünglichen Zeitstempel behält. Sie kann also höchstens durch die

endlich vielen älteren Transaktionen zurückgesetzt werden. Diese wiederum werden in endlicher Zeit beendet, da kein Deadlock eintreten kann. (Auf eine spezielle Möglichkeit zur Endlosblockierung und ihre Verhinderung gehen wir anschließend ein.)

Beim wait-die-Verfahren ist die jüngere Transaktion die aktive Instanz, die einen Neustart auslöst, nämlich ihren eigenen. (Beim wound-wait-Verfahren ist dies die ältere Transaktion.) Hieraus resultiert ein Vorteil des wait-die-Verfahrens: Sobald eine Transaktion keine Sperren mehr anfordert, also alle benötigten Objekte besitzt, wird sie nicht mehr neugestartet. Beim wound-wait-Verfahren kann eine Transaktion dagegen jederzeit zurückgesetzt werden. Aus diesem Grund dürfte auch die Häufigkeit von Neustarts beim wound-wait-Verfahren höher sein.

Beim wait-die-Verfahren kann eine Transaktion, die zurückgesetzt wurde, wenn sie rasch neugestartet wird, in den gleichen Konflikt wie vorher mit einer älteren Transaktion kommen, die nach wie vor die gleichen Sperren hält. Besonders bei stark frequentierten, exklusiv zu sperrenden Objekten droht, daß sehr viele Transaktionen immer wieder wegen dieser Objekte scheitern. Hieraus leitet sich die Empfehlung ab, nicht zu schnell neu zu starten. Beim wound-wait-Verfahren besteht dieses Problem nicht.

Sperrenzuteilung. Ein weiteres Problem beim wait-die-Verfahren sind Endlosblockierungen[21]. So kann eine ältere Transaktion, die ein Objekt schreibsperren möchte und auf die Freigabe von vorhandenen Lesesperren wartet, von neu gewährten Lesesperren für dieses Objekt endlos blockiert werden. Für die Aspekte, die bei der Wahl einer Sperrenzuteilungsstrategie zu beachten sind, sei auf Lehrmodul 3 verwiesen.

Bei beiden Verfahren muß in dem Fall, daß mehrere Transaktionen auf Freigabe der gleichen Sperre warten, die "Warterichtung" berücksichtigt werden, sonst können Deadlocks auftreten. Hierzu betrachten wir als Beispiel folgende Wartesituation nach dem Präfix eines Ablaufs beim wait-die-Verfahren (XLOCK(..) fordert Schreibsperren für die angegebenen Objekte an, ... steht für eine anschließende Wartezeit):

[21] vgl. Abschnitt 3.4.1 in Lehrmodul 3.

```
T1 Z--------XLOCK(x,y).............
T2   Z-----r(y)-------XLOCK(x)...
T3     Z--r(x)--------------w(z)
```

T_1 und T_2 warten auf die Freigabe der Lesesperre auf x, die T_3 hält. T_1 wartet außerdem auf die Freigabe der Lesesperre auf y, die T_2 hält. Angenommen, T_3 endet und gibt die Sperre auf x frei. Nun kann entweder T_1 oder T_2 eine Schreibsperre für x gewährt werden. Im ersten Fall tritt ein Deadlock ein, die Sperre muß T_2, der jüngeren Transaktion gewährt werden.

Sofern bei der Zuteilung einer Sperre zwischen zwei Transaktionen zu entscheiden ist, muß bei beiden Verfahren die Transaktion bevorzugt werden, auf die die andere potentiell wartet, also beim wait-die-Verfahren die jüngere ("älter" wartet auf "jünger") und beim wound-wait-Verfahren die ältere.

Abschließend sei noch bemerkt, daß die Zeitstempel bei beiden Verfahren – im Gegensatz zur Grundform der Zeitstempel-Verfahren – keine Serialisierungspunkte sind.

Lehrmodul 7:

Optimistische Concurrency-Control-Verfahren

Zusammenfassung dieses Lehrmoduls

Optimistische Concurrency-Control-Verfahren gehören zur Gruppe der Verfahren, die auf der Validierung von Abläufen und dem Neustart von Transaktionen basieren. Sie sind vor allem an Szenarien mit wenig schreibenden Transaktionen orientiert. Es findet nur ein einziger Validationstest bei Commit statt. Dessen Grundform prüft, ob die früher gelesen Werte inzwischen nicht überschrieben wird. Ein alternativer, präventiver Ansatz vermeidet es, von anderen Transaktionen gelesene Werte zu überschreiben. Die einzelne Alternativen weisen im Detail einige überraschende Ähnlichkeiten zu Sperr- und Zeitstempel-Verfahren auf.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Sperrverfahren
- Zeitstempelverfahren

Stoffumfang in Vorlesungsdoppelstunden: 0.8

©2001 Udo Kelter

7.1 Motivation

Optimistische Concurrency-Control-Verfahren gehören neben den Zeitstempel-Verfahren zur Gruppe der Verfahren, die auf der Validierung von Abläufen und dem Neustart von Transaktionen basieren. Die Motivation zur Entwicklung beider Arten war (soweit sich das in einer "historischen" Rückschau überhaupt sagen läßt) völlig verschieden: Zeitstempel-Verfahren wurden entwickelt, um das (in verteilten Datenbanken besonders große) Problem der Deadlock-Vermeidung zu lösen. Optimistische Verfahren hingegen beruhen auf der Beobachtung, daß in vielen Fällen Sperren "eigentlich überflüssig" sind (vgl. [KuR81]):

- bei sequentiell ausgeführten Transaktionen (d.h. bei geringer Belastung des Systems)
- bei Lesetransaktionen
- bei konfliktfreien schreibenden Transaktionen.

So deuten einige Untersuchungen darauf hin, daß Konflikte zwischen Transaktionen selten sind. Ferner sind in vielen Systemen die meisten Transaktionen reine Abfragen. Sperrverfahren sind insofern "pessimistisch", als sie stets Sperren einrichten, obwohl speziell in den obigen Fällen die Sperren nie oder fast nie zu Wartevorgängen führen. Der Aufwand zur Handhabung der Sperren und zur Bewältigung der Folgeprobleme des Sperrens wirkt daher unangemessen hoch.

Die Kernidee zur Lösung des Problems besteht darin, optimistisch zu sein und darauf zu vertrauen, daß "schon nichts passiert". Notfalls muß der Schaden repariert werden, was kostengünstiger als die präventiven Maßnahmen bei Sperrverfahren erscheint. Die Reparatur besteht wieder im Rücksetzen und Neustarten einer Transaktion. Deshalb wird verzögertes Schreiben vorgeschrieben, was durch lokale Kopien der zu schreibenden Objekte zu realisieren ist (vgl. Abschnitt 2.4.4).

Im Gegensatz zu Zeitstempel-Verfahren wird nur ein einziger Validationstest durchgeführt, nämlich bei Commit. Dieser bezieht sich auf die gesamte Transaktion und ist vergleichsweise komplex. Man unterscheidet daher in der Commit-Abarbeitung eine **Validationsphase** und eine **Schreibphase**. Bei Lesetransaktionen fehlt die Schreibphase.

Validations- und ggf. Schreibphase müssen *atomar* ausgeführt werden. Der Hauptteil der Transaktion wird **Lesephase** genannt: In ihr werden alle benötigten Objekte gelesen und alle Berechnungen auf den lokalen Kopien zu schreibender Objekte durchgeführt.

Die optimistischen Verfahren unterscheiden sich im Validationstest. Da nur ein einziger Test am Ende der Transaktion durchgeführt wird, kann theoretisch das gesamte Geschehen in der Datenbank während der Laufzeit der Transaktion in die Validation mit einbezogen werden. Im Extremfall könnte genau entschieden werden, ob durch das geplante Schreiben die anschließend vorhandene effektive Ausführungssequenz serialisierbar bleibt oder nicht. Hierzu wäre i.w. der sog. Konfliktgraph des bisherigen Ablaufs laufend mitzuführen und auf Zyklen zu untersuchen. Dieses Verfahren wäre allerdings viel zu aufwendig. Generell müssen sehr viel gröbere Tests verwendet werden, sonst wäre der Aufwand für die Speicherung und Handhabung der erforderlichen Hilfsdaten zu hoch.

Eine Vereinfachung der Tests bedeutet gleichzeitig, daß die effektiven Ausführungssequenzen ein strengeres Korrektheitskriterium als Serialisierbarkeit erfüllen. Tatsächlich ist der Commit-Zeitpunkt bei allen Varianten der Tests zugleich Serialisierungspunkt[22].

Beim Serialisieren, also beim Verschieben aller Ereignisse einer Transaktion zu ihrem Serialisierungspunkt, brauchen die Schreibereignisse gar nicht verschoben zu werden, denn sie finden ja innerhalb des atomaren Commits statt. (Hierbei sehen wir einmal davon ab, daß wegen der Komplexität der Validation vielfach die Validations- und Schreibphasen verschiedener Transaktionen überlappend ausgeführt werden müssen.) Die einzige Form von Konflikt, die noch auftreten kann, ist ein Lese-Schreib-Konflikt folgenden Musters:

```
T1 r(x)----- ....
T2           ...--w(x)
```

Das Leseereignis von T_1 kann wegen des Schreibereignisses von T_2 nicht konfliktfrei zum Ende von T_1 verschoben werden. Der Startzeitpunkt

[22]Zum Vergleich: Bei der Grundform der Zeitstempel-Verfahren war es der Startzeitpunkt einer Transaktion.

von T_2 kann auch vor T_1 liegen.

7.2 Die Grundform

Die Grundform (nach [KuR81]) beruht auf einem recht groben Test: Während der gesamten Lesephase einer Transaktion dürfen keine Schreibereignisse (also Commits) anderer Transaktionen stattfinden, bei denen eines der gelesenen Objekte verändert wird. Folgende Hilfsdaten werden vorgesehen:

`tc`: `tc` ist ein globaler Transaktionszähler. Bei jeder Validation einer Transaktion wird er um 1 erhöht, der neue Wert ist eine Nummer, die der soeben validierten Transaktion zugewiesen wird.

Die Nummer einer Transaktion ist vergleichbar mit einem Zeitstempel. Im Gegensatz zu Zeitstempel-Verfahren erhält eine Transaktion hier ihren "Zeitstempel" erst beim erfolgreichen Commit.

`tc` gibt somit die Nummer der letzten erfolgreich beendeten Transaktion an.

`t_start`: Dies ist eine transaktionslokale Kopie des Wertes, den `tc` beim Start dieser Transaktion hatte.

Bei der Validation einer Transaktion müssen nur solche Transaktionen berücksichtigt werden, deren Nummer größer als `t_start` ist.

`readset`: ist eine transaktionslokale Variable, die die Menge der Identifikationen der (bisher) gelesenen Objekte angibt.

`writeset`: ist eine transaktionslokale Variable, die die Identifikationen der zu schreibenden Objekte angibt.

`writeset(t)`: ist eine globale Variable, die die Menge der Identifikationen der Objekte angibt, die die Transaktion mit Nummer t bei ihrem erfolgreichen Commit geschrieben hat.

Der Validationsalgorithmus lautet:

```
BEGIN
   VAR valid :  BOOLEAN;
       t     :  INTEGER;
   valid := TRUE;
   FOR t := t_start TO tc DO
      IF ( writeset(t) ∩ readset ≠ ∅ )
      THEN valid := FALSE;

   IF valid
   THEN BEGIN
           tc := tc + 1;
           writeset(tc) := writeset
        END
   ELSE NEUSTART
END
```

Anschließend folgt die Schreibphase.

In [KuR81] werden noch einige Erweiterungen des Validationsalgorithmus vorgestellt, die die überlappende Ausführung von Schreib- und Validationsphasen ermöglichen. Hierauf wollen wir jedoch aus Platzgründen nicht weiter eingehen.

Man erkennt, daß alle gelesenen Objekte während der gesamten Dauer der Transaktion mit einer gedachten Lesesperre versehen sind; eine Verletzung dieser Lesesperre führt zum Abbruch der Transaktion. Es reicht, eine einzige Verletzung zu finden, d.h. die for-Schleife im Algorithmus könnte abgebrochen werden, sobald `valid = FALSE` ist.

Wenn das Schreibereignis vor dem ersten Lesen von x stattfindet, ist der Konflikt offenbar harmlos. Dies ist im folgenden Beispiel dargestellt:

```
Tj ....--------r(x)--Commit
Ti          ..--w(x)
```

Im Validationstest wird dieser Schein-Konflikt nicht berücksichtigt, es wird ein eigentlich überflüssiger Neustart von T_j veranlaßt.

7.2.1 Die verbesserte Grundform

Die gedachte Sperrung eines Objektes während der gesamten Transaktion führt somit zu unnötig vielen Neustarts, eine gedachte Sperrung vom ersten Lesen bis zum Commit reicht völlig aus, um zu gewährleisten, daß der Commit-Zeitpunkt Serialisierungspunkt ist. Der Validationstest ist wie folgt zu ändern:

- Für jedes gelesene Objekt x wird eine lokale Variable `tcr(x)` vorgesehen, welche beim ersten Lesen den aktuellen Wert von `tc` notiert.
- Die for-Schleife im Validationstest wird ersetzt durch:

```
FOR x IN readset DO
   FOR t := tcr(x) + 1 TO tc DO
      IF x IN writeset(t) THEN valid := FALSE;
```

Für alle gelesenen Objekte wird überprüft, ob sie seit dem ersten Lesen bis zum Commit von einer anderen Transaktion geschrieben worden sind. Diese muß eine Nummer zwischen `tcr(x)` $+ 1$ und `tc` haben.

Die lokalen Transaktionsnummern an den Objekten `tcr(x)` vermeiden im Vergleich zur Grundform zwar gewisse unnötige Neustarts, verursachen andererseits jedoch Mehraufwand zu ihrer Speicherung und bei Zugriffen zu der globalen Variablen `tc` ; ferner wird der Validationstest etwas komplizierter.

7.2.2 Verwaltung der Hilfsdaten

Problematisch sind nur die Mengenvariablen, also die transaktionslokalen Variablen `readset` und `writeset` und die globalen Variablen `writeset(t)` . Letztere sollten natürlich dynamisch verwaltet werden, d.h. nicht mehr benötigte sollten gelöscht werden. Sei `t_min` der kleinste Wert `t_start` von allen aktiven Transaktionen. Dann müssen bei beiden Versionen der Grundform alle Variablen `writeset(t)` für `t_min` $\leq$ t $\leq$ `tc` zugreifbar sein, denn die Transaktion mit `t_start` $=$ `t_min` benötigt diese Variablen alle bei ihrer Validation.

Bei einzelnen, sehr langen Transaktionen kann die Speicherung dieser Variablen ein ernsthaftes Problem werden.

Die Variablen `readset` brauchen nicht über die Laufzeit der Transaktion hinaus gespeichert zu werden: sobald eine Transaktion validiert ist, interessiert die Menge der von ihr gelesenen Objekte nicht mehr.

Insgesamt müssen folgende Mengenvariablen verwaltet werden:

- `readset` und `writeset` lokal in allen aktiven Transaktionen;
- `writeset(t)` für `t_min` $\leq$ t $\leq$ `tc` .

Vergleicht man diese Variablen mit dem Inhalt einer Sperrtabelle unter der Annahme, daß keine Konflikte zwischen Transaktionen auftreten (also den Verhältnissen, für die die optimistischen Verfahren gerade prädestiniert sind), so stellt man überraschenderweise fest, daß die lokalen Variablen `readset` und `writeset` bereits den gesamten Inhalt der Sperrtabelle (bis auf eventuelle zusätzliche Hilfsdaten) darstellen! Dargestellt wird in beiden Fällen eine dreistellige Relation (Transaktionsidentifikation, Objektidentifikation, (gedachter) Sperrmodus).

Unterschiedlich sind die Organisation und die Zugriffsmöglichkeit: In der Sperrtabelle erhält man zu einer gegebenen Objektidentifikation die zugehörigen Sperren, also Transaktionsidentifikationen und Sperrmodi. Beim optimistischen Ansatz erhält man zu einer Transaktionsidentifikation und zu einem gedachten Sperrmodus (S oder X) die entsprechend gesperrten Objekte.

Die globalen Variablen `writeset(t)` haben überhaupt keine Entsprechung in Sperrverfahren. Man kann sie so interpretieren, daß bei optimistischen Verfahren die gedachten Schreibsperren noch eine Zeitlang über die Transaktion hinaus verwaltet werden müssen. Je nach der internen Struktur des DBMS kann ein Vorteil der optimistischen Verfahren darin bestehen, daß die Variablen `readset` und `writeset` nur lokal innerhalb der Transaktionen verwaltet werden müssen.

Zusammenfassend muß gesagt werden, daß die optimistischen Verfahren ihr Hauptziel i.a. nicht erreichen: Hauptziel war, bei sehr seltenen

Konflikten den "überflüssigen" Aufwand zu vermeiden, der durch die unbenutzten Sperren entsteht. Um jedoch über die Validation einer Transaktion entscheiden zu können, werden in etwa die gleichen Informationen benötigt wie für die Entscheidung, ob auf die Freigabe einer Sperre gewartet werden muß (letztlich, weil ein ähnlicher Korrektheitsbegriff zugrunde liegt). In beiden Fällen müssen i.w. die gleichen Hilfsdaten verwaltet werden. Der Aufwand für die Verwaltung der Hilfsdaten für die Validation liegt daher in der gleichen Größenordnung wie der Aufwand für die Sperren, bei nur wenigen langen Transaktionen ist er sogar deutlich höher.

Unbeschadet des Aspekts Verwaltungsaufwand scheinen optimistische Verfahren auf den ersten Blick einen weiteren Vorteil zu bieten: Es können nicht, wie bei Sperrverfahren, lange Ketten von aufeinander wartenden Transaktionen auftreten, die zu immer höherer Konfliktwahrscheinlichkeit führen. In solchen Konstellationen muß allerdings auch mit häufigem Neustart gerechnet werden.

7.2.3 Eine alternative Realisierung des Validationstests

Im Validationstest der Grundform muß der Durchschnitt zweier Mengen von Objektidentifikationen gebildet werden. Bei der verbesserten Grundform muß geprüft werden, ob ein Objekt in einer Menge enthalten ist. Die Mengenvariablen sollten so implementiert werden, daß diese Tests effizient durchführbar sind. So sollten bei einer Implementation mit Listen die Listenelemente aufsteigend sortiert werden, z.B. nach dem Primärschlüssel der Objekte.

Bei beiden Verfahren ist ein deutlich einfacherer Validationstest möglich, wenn zu jedem Objekt bekannt ist, von welcher Transaktion es zuletzt geschrieben worden ist. Nehmen wir an, daß die globale Variable `tcw(x)` die Nummer dieser Transaktion enthält. Die beiden Validationstests könnten dann durch folgende Schleifen ersetzt werden:

Grundform:

```
FOR x IN readset DO
   IF tcw(x) > t_start THEN valid := FALSE
```

Verbesserte Grundform:

```
FOR x IN readset DO
   IF tcw(x) > tcr(x) THEN valid := FALSE
```

Ein Vergleich mit der Grundform der Zeitstempel-Verfahren offenbart überraschende Ähnlichkeiten: Die Variablen `tcw(x)` und `tcr(x)` entsprechen in etwa den Zeitstempeln an Objekten bei der Grundform der Zeitstempel-Verfahren. Unterschiede bestehen in der Verwendung und der "Lebensdauer" der Variablen.

7.2.4 Zyklischer Neustart

Bei der Grundform der optimistischen Verfahren ist die Zahl der Neustarts einer Transaktion nicht begrenzt. Dieses Problem bestand ebenfalls bei der Grundform der Zeitstempel-Verfahren.

Besonders betroffen sind lange Transaktionen, sie können nämlich leicht "Opfer" von kurzen schreibenden Transaktionen werden. In [KuR81] wird vorgeschlagen, nach Überschreiten einer gewissen Zahl von Neustarts die gesamte Datenbank für die betroffene Transaktion exklusiv zu sperren. Wir werden anschließend ein weiteres optimistisches Verfahren kennenlernen, das nicht so "rohe Gewalt" anwendet.

7.3 Präventive Validation

Die Grundform befriedigt in zwei Punkten nicht, nämlich in Bezug auf zyklische Neustarts (s.o.) und die Behandlung von Lesetransaktionen.

Lesetransaktionen verursachen untereinander nie Konflikte. Dennoch müssen sie eine Validationsphase ausführen. Dies stört besonders deshalb, weil optimistische Verfahren gerade auf solche Anwendungen abzielen, in denen Lesetransaktionen überwiegen. Konflikte werden nur durch schreibende Transaktionen direkt verursacht. Konsequenterweise sollten sie allein für die Lösung des Concurrency-Control-Problems sorgen. Umgekehrt sollten die Lesetransaktionen nicht damit befaßt sein, also gar nicht validieren. (Ihre gelesenen Objekte müssen sie

natürlich auf jeden Fall "bekanntgeben".) Lesetransaktionen können somit jederzeit "ohne Vorwarnung" enden, ihr COMMIT-Zeitpunkt muß automatisch Serialisierungspunkt sein. Somit darf keine schreibende Transaktion einen Konflikt mit einer noch aktiven Lesetransaktion verursachen. Sofern von aktiven Transaktionen nicht bekannt ist, ob sie Lesetransaktionen sind oder nicht, müssen sie vorsichtshalber als Lesetransaktionen behandelt werden. Dann darf eine schreibende Transaktion überhaupt keinen Konflikt mit einer anderen aktiven Transaktion verursachen. Auch ansonsten erscheint es sinnvoll, den schreibenden Transaktionen ebenfalls eine "ungestörte Lesephase" zu garantieren.

Der Validationstest (einer schreibenden Transaktion) ändert seinen Charakter gegenüber der Grundform völlig: dort wird überprüft, ob die von einer Transaktion in der Vergangenheit gelesenen Werte noch vorhanden sind. Für die Prüfung muß die Menge der `writeset` s früherer Transaktionen bekannt sein. Dieses Problem existiert nun nach Voraussetzung nicht mehr, eine schreibende Transaktion muß nun durch *präventive* Maßnahmen bewirken, daß diese Voraussetzung stets erfüllt ist. Bei einer präventiven Validation muß somit geprüft werden, ob die Objekte, die geschrieben werden sollen, von anderen Transaktionen gelesen wurden; hierfür müssen die `readset` s der derzeit aktiven Transaktionen bekannt sein. Abgeschlossene Transaktionen spielen keine Rolle mehr.

Für die Organisation der Hilfsdaten nehmen wir an, daß die aktiven Transaktionen von 1 an durchnumeriert sind, ihre Anzahl steht in der globalen Variablen `tca` . Die Menge der bisher von der Transaktion mit der Nummer t gelesenen Objekte wird in einer globalen Variablen `readset(t)` angezeigt.

Der Validationsalgorithmus lautet nun:

```
BEGIN
   VAR valid :  BOOLEAN;
       t     :  INTEGER;
   valid := TRUE;
   FOR t := 1 TO tca DO
      IF (readset(t) ∩ writeset ≠ ∅)
```

```
      THEN valid := FALSE;
   IF valid THEN (* Schreibphase *)
            ELSE löse_Konflikt_auf;
END
```

Die präventive Validation hat gegenüber der Grundform den großen Vorteil, daß sie sehr flexible Reaktionen auf Konflikte erlaubt, weil noch keine der Transaktionen, die in den Konflikt verwickelt sind, abgeschlossen ist; jede kann neugestartet werden. Die Möglichkeiten zur Konfliktauflösung sollen hier nur skizziert werden:

- eigener Neustart oder
- Warten auf Beendigung oder
- Neustart der aktiven Transaktionen, die den Konflikt verursachen

oder Kombinationen hiervon. Letztlich können fast alle Konzepte und Ideen integriert werden, die sich schon bei der Behandlung von Konflikten in Sperr- oder Zeitstempel-Verfahren bewährt haben. Auf diese Weise kann das Problem zyklischer Neustarts gelöst werden, ebenso können Prioritäten oder sonstige strategische Ziele realisiert werden.

Je nach Verwendung von Zeitstempeln oder Sperren ergäbe sich ein kombiniertes optimistisches Sperr- und/oder Zeitstempel-Verfahren. Die Frage ist allerdings, welchen Anteil das Konzept "Optimismus" in solchen Kombinationen noch hat oder ob nicht Spezialfälle von Sperrverfahren oder kombinierten Sperr- und Zeitstempel-Verfahren entstehen; der Spezialfall ist gerade die Vereinfachung der Verfahren, die infolge des verzögerten Schreibens und seiner Realisierung durch Puffer (anstelle von Sperren) möglich ist.

Nachteilig gegenüber der Grundform ist, daß alle `readset` s global sind und ständig auf dem neuesten Stand gehalten werden müssen. (In der Realisierung unterscheiden sich die `readset` s praktisch nicht von Lesesperren, vgl. oben.) Andererseits dürfen sie während der Validation einer anderen Transaktion nicht verändert werden, so daß während dieser Zeit entweder alle `readset` s zu sperren sind oder relativ komplizierte Algorithmen verwendet werden müssen, die diese Sperrung vermeiden.

Vorteilhaft gegenüber der Grundform ist die reduzierte Zahl von Validationstests und der Wegfall der globalen `writeset` s. Insgesamt ist die präventive Validation bei einer geeigneten Konfliktbehandlung der Grundform fast immer überlegen.

Das Verhältnis zu Sperrverfahren und insbesondere zu kombinierten Sperr- und Zeitstempelverfahren ist schwierig zu beurteilen. Die Gründe für diese Schwierigkeiten wurden bereits genannt.

Selbst im Falle seltener Konflikte, also dem Fall, für den optimistischen Verfahren in erster Linie gedacht sind, ist eine eindeutige Überlegenheit nicht nachweisbar. Im Gegenteil kommt Härder in einer ausführlichen Diskussion [Ha84] zu dem Ergebnis, daß nur unter weiteren einschränkenden Voraussetzungen optimistische Verfahren den Sperrverfahren überlegen sind.

Bei hoher Konfliktwahrscheinlichkeit sind optimistische Konzepte generell nicht vorteilhaft. (Dies gilt für das Validationsprinzip ganz allgemein, also auch für die Grundform der Zeitstempelverfahren; bei den kombinierten Sperr- und Zeitstempelverfahren kann man sich darüber streiten, wie groß der “Anteil” des Validationskonzeptes ist.) Im Extremfall sinkt die Wahrscheinlichkeit einer erfolgreichen Validation gegen Null.

Für universell anwendbare CC-Verfahren kommen daher beim heutigen Stand der Erkenntnis “reine” oder überwiegend optimistische Verfahren nicht in Betracht, sondern nur als lastabhängige Alternative zu Sperr- oder kombinierten Sperr- und Zeitstempelverfahren, ggf. mit automatischer Umschaltung der Verfahren durch die CC-Komponente.

Lehrmodul 8:

Concurrency-Control-Theorie

Zusammenfassung dieses Lehrmoduls

Concurrency-Control- (CC-) Verfahren müssen garantieren, daß bei der verzahnten Ausführung mehrerer paralleler Transaktionen keine störenden Interferenzen auftreten. Was diese Anforderung konkret bedeutet, ist intuitiv nicht offensichtlich und auch nicht trivial zu beantworten. Die Concurrency-Control-Theorie löst dieses Problem, indem formale Modelle parallel ausgeführter Transaktionen definiert und darin Korrektheitskriterien für Abläufe definiert werden. Zentral ist hier der Begriff Serialisierbarkeit; von diesem können mehrere Varianten gebildet werden. Die wichtigste ist die konflikterhaltende (cp-) Serialisierbarkeit; diese wird von allen üblichen CC-Verfahren garantiert. Man kann ferner zeigen, daß das 2-Phasen-Sperren nur geringfügig restriktiver als unbedingt notwendig ist, d.h. daß es keine Verfahren geben kann, die wesentlich mehr Parallelität ermöglichen.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Sperrverfahren

Stoffumfang in Vorlesungsdoppelstunden: 3.5

©2003 Udo Kelter

8.1 Einführung und Übersicht

Die üblichen Concurrency-Control- (CC-) Protokolle garantieren, daß verzahnte Ausführungen mehrerer paralleler Transaktionen serialisierbar sind. Serialisierbarkeit ist ein Korrektheitskriterium für verzahnte Ausführungen von Transaktionen, i.f. als **Log** bezeichnet, und besagt, daß keine störenden Interferenzen auftreten. Störenden Interferenzen können sich auf zwei Arten manifestieren:

- Der Datenbankinhalt wird inkorrekt (hier verstanden im Sinne der logischen Konsistenz, s. Lehrmodul 1.
- Die von den einzelnen Transaktionen gelesenen Daten, i.f. als ihre **Sicht** bezeichnet, sind inkorrekt.

Man kann eine verzahnte Ausführung von Transaktionen alternativ dazu als korrekt ansehen, wenn die einzelnen Transaktionen scheinbar atomar ausgeführt werden. Die so definierte **logische Atomarität** ist zunächst ein informeller Begriff; es ist intuitiv nicht klar, wie sie mit der Abwesenheit von Interferenzen zusammenhängt.

Die informellen Definitionen kann man in verschiedenen Varianten durch formale Modelle präzisieren. In diesem Lehrmodul werden nach einigen einleitenden Bemerkungen zur historischen Entwicklung Modelle im allgemeinen und Modelle von parallelen Transaktionen in Datenbanken im besonderen diskutiert. Abschnitt 8.2 enthält die grundlegenden Definitionen.

In Abschnitt 8.3 untersuchen wir Merkmale von Logs, die die Erhaltung der Konsistenz der Datenbank implizieren, in Abschnitt 8.4 solche Merkmale, die konsistente Sichten von Transaktionen implizieren. In Abschnitt 8.5 wird untersucht, welche Logs die Einhaltung der logischen Atomarität gewährleisten. In allen Abschnitten werden Varianten der Serialisierbarkeit untersucht und in Verbindung gesetzt. Eine Reihe von Zusammenhängen zwischen den Varianten der Serialisierbarkeit wird nachgewiesen, zum Teil unter einigen zusätzlichen Randbedingungen.

Mit Hilfe der formalen Serialisierbarkeitsbegriffe kann die Korrektheit von CC-Verfahren beurteilt werden. Ein zentrales Resultat ist,

daß das 2-Phasen-Sperren nur geringfügig restriktiver als unbedingt notwendig ist, d.h. daß es keine alternativen (allgemeinen) Verfahren geben kann, die wesentlich mehr Parallelität ermöglichen.

Zur historischen Entwicklung. Man verlangt von CC-Verfahren, daß bei den entstehenden Logs "keine Interferenzen auftreten". Bei einem solchen informellen Umgang mit Begriffen bleiben oft Feinheiten unklar und außer acht, deren Behandlung mit informellen Mitteln sehr umständlich wäre und die Diskussion sehr zähflüssig machen würde. Verzahnungen von Transaktionen können eine sehr komplizierte Struktur haben, es sind viele Sonderfälle zu beachten. Ferner sind Entscheidungsverfahren anzugeben, also Algorithmen, die bei einem gegebenen Ablauf eine Antwort liefern, ob dieser Ablauf die Restriktion erfüllt oder nicht. Ferner können sich bei detaillierterer Betrachtung noch Varianten der Begriffe identifizieren lassen.

Seit Mitte der 70er Jahre wurden die genannten Probleme vielfach in formalen Modellen untersucht. Es bildete sich eine eigene, sehr reichhaltige Theorie heraus, die meist *Concurrency-Control-Theorie* oder Serialisierbarkeits-Theorie genannt wird. Kernthemen sind die formale Beschreibung korrekter Abläufe und die Erforschung von Zusammenhängen zwischen den verschiedenen Korrektheitsbegriffen für Abläufe, die Realisierbarkeit bzw. Komplexität von Mechanismen, die gewisse Formen korrekter Abläufe ermöglichen, sowie die maximal erreichte Parallelität bei bestimmten Voraussetzungen[23].

Wir stellen hier nur die Grundzüge dieser Theorie vor, dies auch nur für zentrale Datenbanken. Bei verteilten Datenbanken ergeben sich einige zusätzliche Problemfaktoren (z.B. Zuverlässigkeit beim Verlust von Nachrichten).

[23] Diese Theorie ist - vor allem in einigen Grundlagen - eng verwandt mit der Theorie paralleler Prozesse, die im Laufe der 60er Jahre im Hinblick auf Probleme in Betriebssystemen und der parallelen Programmierung entstand; man kann die Concurrency-Control-Theorie als einen Ableger der Theorie paralleler Prozesse bezeichnen. Sie hat jedoch ein eigenständiges Profil gewonnen, vor allem deshalb, weil einige Randbedingungen in Datenbanken wesentlich anders als in Betriebssystemen sind.

Wie behandeln hier ferner nur Korrektheitskriterien, die für **universelle** CC-Verfahren gefordert werden müssen. In der Praxis duldet man manchmal “unkritische” Interferenzen, weil dadurch die Parallelität erhöht werden kann. CC-Verfahren, die solche weicheren Korrektheitskriterien realisieren, sind nicht universell einsetzbar, sondern müssen i.d.R. fallspezifisch entworfen bzw. adaptiert werden.

8.2 Die Modelle

Exakte Beschreibungen von Systemen erhält man durch formale Modelle. *Modelle* sind ein in allen Naturwissenschaften übliches Mittel zur Beschreibung und Untersuchung von Systemen. Ein Modell M ist ein System, welches ein anderes reales oder gedachtes (geplantes) System S modelliert, wenn gilt:

- M und S ähneln sich in den interessierenden Eigenschaften; in der Regel sind dies Eigenschaften der Systemstruktur. Wegen der Ähnlichkeit sind Schlüsse von M auf S möglich.
- M ist “einfacher” (kleiner, billiger, schneller verfügbar, u.s.w.) als S; in der Regel ist M auf die interessierenden Teilstrukturen von S, auf ”das Wesentliche” von S, reduziert. Von den unwichtigen und lästigen Details von S wird in M abstrahiert.

Wir werden hier in doppelter Hinsicht von Details abstrahieren:

- Die Struktur der Datenbanken wird gegenüber der Realität stark vereinfacht.
- Bei der graphischen Darstellung von Transaktionen und Abläufen werden diese auf das Zugriffsverhalten reduziert.

LS-Modelle von parallelen Transaktionen. Nach den Bedingungen, die für universelle CC-Verfahren gelten, sind allein Zugriffe zu Datenbank-Objekten wesentlich, nicht hingegen die Verarbeitungslogik innerhalb eines Transaktionsprogramms.

In unserem Modell für parallele Transaktionen in Datenbanken werden wir noch einen Schritt weiter gehen: wir werden annehmen, daß jede Transaktionsausführung nur aus zwei Ereignissen besteht, die wir mit $\mathbf{R}_i$ und $\mathbf{W}_i$ bezeichnen; i ist die Nummer der Transaktion.

$\mathbf{R}_i$ ist das Ereignis, bei dem die Eingabeoperanden der Transaktion gelesen werden. Die Menge dieser Datenbank-Objekte bezeichnen wir mit $\mathbf{readset}_i$.

$\mathbf{W}_i$ ist analog das Ereignis, bei dem die Datenbank-Objekte in $\mathbf{writeset}_i$ geschrieben werden.

Beide Ereignisse sind in unserem Modell atomar. W_i findet auf jeden Fall nach R_i statt.

So geformte Transaktionen nennen wir **Lese-Schreib–Transaktionen**, die Modelle für Abläufe dementsprechend Lese-Schreib-Transaktions-Modelle (kurz: **LS-Modelle**).

In der Realität haben zwar viele Transaktionen eine ähnlich simple Struktur, es können aber auch wesentlich komplexere Strukturen mit vielen abwechselnden Lese- und Schreibzugriffen auftreten. Formale Modelle, in denen sich ein so komplexes Zugriffsverhalten nachbilden läßt, müssen deutlich komplizierter als die LS-Modelle sein und führen zu einem höheren Notationsaufwand. Es stellt sich indessen heraus, daß bereits in LS-Modellen alle wesentlichen Probleme auftreten und Begriffsvarianten unterschieden werden können, so daß der Notationsaufwand für Mehrschritt-Transaktionsmodelle nicht gerechtfertigt ist.

In unseren Modellen gehen wir weiter davon aus, daß in einem beobachteten Ablauf eine beliebige, aber feste Menge von Transaktionen genau einmal ausgeführt wird. Die folgende Definition enthält die gesamte statische Struktur der Modelle:

8.2.1 Struktur einer Datenbank mit parallelen Transaktionen

Definition: Die Struktur einer Datenbank mit parallelen Transaktionen wird durch folgende Mengen modelliert:

- Datenbank (= Menge von Datenbank-Objekten):
  DB := { o_1, ..., $o_{|DB|}$ }
- Transaktionen:
  TR := { $\mathbf{T}_1$, ..., $\mathbf{T}_n$ }; n = | TR |
- Ein-/Ausgabeoperandenmengen der Transaktionen:
  für alle i, $1 \leq i \leq n$, ist $\mathbf{readset}_i \subseteq$ DB die Menge der gelesenen Objekte, $\mathbf{writeset}_i \subseteq$ DB die Menge der geschriebenen Objekte
  Es mußt gelten $\text{readset}_i \cup \text{writeset}_i \neq \emptyset$

Alle Mengen sind endlich. Anmerkungen:

- Es wird nicht verlangt, daß $\text{writeset}_i \subseteq \text{readset}_i$ ist, d.h. eine Transaktion kann ein Datenbank-Objekt "blind" überschreiben, ohne den alten Inhalt gelesen und weiterverarbeitet zu haben.
- $\text{writeset}_i = \emptyset$ ist zulässig. In diesem Fall ist T_i eine **Lesetransaktion**.
- Eine Transaktion mit leerer Ein- und Ausgabeoperandenmenge wäre in unserem Modell sinnlos und wird deshalb aus geschlossen.

8.2.2 Logs

Abläufe in unserem Modell nennen wir Logs. Wegen der Atomarität der Lese- bzw. Schreibereignisse ist ein **Log** eine *Folge* von Ereignissen, die wir als Wort notieren werden. Da wir annehmen, daß in einem Log jede der Transaktionen aus TR genau einmal ausgeführt wird, enthält ein Log genau 2n Ereignisse.

In der Literatur werden Abläufe in formalen Modellen häufig anders bezeichnet, z.B. mit *history*, *computation*, *schedule* u.a.m.; die Bezeichnung Log ist am häufigsten, obwohl bei dieser Bezeichnung eine gewisse Verwechslungsgefahr mit Recovery-Logs besteht. Beide sind in gewisser Weise Aufzeichnungen von Abläufen, allerdings einmal in einer realen Datei und das andere Mal in einem gedachten formalen Modell.

Definition (Logs zu TR): $\mathbf{L} := \{ R_1W_1 \} \between \ldots \between \{ R_nW_n \}$

In dieser Definition ist

- $\{R_iW_i\}$ die Menge, die aus dem einen Wort mit den beiden Zeichen R_i und W_i besteht;
- $\between$ der sog. Mischoperator (shuffle-Operator), der Wortpaare aus zwei Mengen ordnungserhaltend mischt. Bei endlichen Worten können wir ihn wie folgt definieren:

Definition: Sei A ein Alphabet und A^* die Menge aller Worte über A. Seien M, M' Teilmengen von A^*.

$$M \between M' := \{ u_1v_1 \ldots u_kv_k \mid u_i, v_i \in A^* \text{ für } 1 \leq i \leq k, \; u_1 \ldots u_k \in M, \; v_1 \ldots v_k \in M' \}$$

Die Menge L enthält alle Abläufe, die potentiell eintreten könnten, wenn alle Transaktionen aus TR parallel gestartet würden und jede Transaktion das sequentielle "Programm" `begin` R_i; W_i `end` ausführte.

Logs sind als Folgen von effektiv ausgeführten Zugriffen zu verstehen, nicht als Folgen von Anforderungen für Zugriffe, die an einen CC-Mechanismus gerichtet werden und die von diesem ggf. umsortiert werden.

Wir werden für Logs zwei verschiedenartige, aber äquivalente *Notationen* benutzen:

1. die halbgraphische Notation, die schon in Lehrmodul 3 eingeführt wurde. Beispiel:

   l1:

   Darin bedeuten:

   `r(X)` Lesen des Objekts bzw. der Objektmenge X

   `w(X)` Schreiben des Objekts bzw. der Objektmenge X

2. die textuelle Notation: Die Folge der Ereignisse wird als Wort notiert. Zur besseren Übersicht wird hinter dem Zeichen für das Ereignis (R_i oder W_i) die Menge der gelesenen bzw. geschriebenen Datenbanken-Objekte in Klammern angegeben. Beispiel:
   $l_1 = R_1(x)R_2(x)W_2(x)W_1(x)$

Die formale Definition eines Logs bzw. der Menge L ist fest verknüpft mit der Menge TR, jeder Log hat 2n Ereignisse. Gelegentlich werden wir auch mit Anfangsstücken von Logs und mit Logs, die sich auf eine Teilmenge von TR beziehen, arbeiten. Hierzu definieren wir:

Definition: Sei TR' eine Teilmenge von TR, $l \in L$. Der **Teillog zu TR' aus l** ist die Folge von Ereignissen, die aus l entsteht, wenn man alle Ereignisse von Transaktionen aus TR - TR' entfernt.

Definition: Sei w ein Wort der Länge k. Für $0 \leq i \leq k$ ist **präfix(w,i)** das Wort, welches aus den i ersten Zeichen von w besteht.

Definition: Sei M eine Wortmenge. **präfixe(M)** ist die Menge aller Präfixe von Worten aus M, also:
präfixe(M) $:= \{ u \mid \exists$ v mit $uv \in M \}$

Da jedes Wort (unechtes) Präfix von sich selbst ist, ist M Teilmenge von präfixe(M). Ebenso ist das leere Wort in präfixe(M) enthalten, sofern $M \neq \emptyset$ ist. präfixe(L) ist somit die Menge aller Präfixe von Logs.

Die zeitliche Reihenfolge von Ereignissen in einem Log notieren wir mit einem Pfeil, also $R_i \rightarrow W_i$.

8.2.3 Andere Modelle für parallele Transaktionen

Der hier vorgestellte Typ von Modellen ist keineswegs der einzig mögliche und vernünftige. Er ist aber durchaus bedeutend, da er in vielen wichtigen Quellen benutzt wurde. Ein anderer, ebenso bedeutender Modelltyp modelliert Transaktionen als feste, aber beliebig lange Folge von Ereignissen, bei denen jeweils genau eine Variable gelesen *und* geschrieben wird (vgl. [EsC75]). Es wird oft n-Schritt-Modell genannt.

Daneben existieren auch Modelle, die die Struktur der Datenbanken berücksichtigen, z.B. hierarchische Datenbanken-Strukturen in [Si82].

Bei LS-Transaktionen können gewisse Parallelitätsanomalien gar nicht auftreten, z.B. das nichtwiederholbare Lesen. Anders gesagt können diese real durchaus erzeugbaren Erscheinungen in unseren Modellen nicht nachvollzogen bzw. dargestellt werden. Die Alternative wären allgemeinere (n-Schritt-) Modelle gewesen, bei denen das Zugriffsverhalten der Transaktionen keinen Beschränkungen unterworfen gewesen wäre. Der Vorteil der allgemeineren Modelle ist ihre höhere Modellierungsfähigkeit. Die Kehrseite der Medaille ist der deutlich höhere notationelle Aufwand und die kompliziertere Formulierung und Behandlung von Zusammenhängen.

Ein wichtiger Aspekt beim Vergleich der beiden Modelltypen ist natürlich, wie wichtig die Erkenntnisse sind, die nur durch sie ermöglicht werden. In dieser Hinsicht haben die allgemeineren Modelle nur unwesentliche Vorteile gegenüber den Modellen mit LS-Transaktionen. Alle wichtigen Begriffe und Zusammenhänge, die in diesem Lehrmodul vorgestellt werden, können direkt auf die allgemeineren Modelle verallgemeinert werden. Weiter wird normalerweise nicht ohne Grund auf Preclaiming und Sperren bis EOT (bzw. Schreiben aller Objekte erst bei Commit) verzichtet, da sonst u.a. Fortpflanzung von Rollback auftreten könnte; dann aber werden die höheren Modellierungsfähigkeiten der allgemeinen Modelle nicht ausgenutzt.

Eine Abwägung von Aufwand und Nutzen spricht daher gegen eine Verwendung allgemeinerer Modelle im Rahmen dieses Lehrmoduls.

8.3 Konsistenzerhaltung

8.3.1 Konsistente Datenbank-Zustände

In diesem Abschnitt suchen wir nach Merkmalen von Logs, die garantieren, daß die Datenbanken am Ende in einem konsistenten Zustand ist. Zunächst ist zu klären, wann ein Zustand unter den Voraussetzungen und Annahmen unseres Modells konsistent ist. Voraussetzungen sind:

- Der Anfangszustand der Datenbanken ist konsistent.
- Jede Transaktion formt, allein ausgeführt, jeden konsistenten Zustand in einen neuen konsistenten Zustand um. Das gleiche gilt für jede *Folge* nichtüberlappend ausgeführter Transaktionen.
- Wir haben keine Kenntnis von der Bedeutung der Datenbanken-Inhalte, Verarbeitungsfunktion der Transaktionen, den Konsistenzbedingungen oder sonstigen semantischen Eigenschaften des Datenbanksystems. Wir können auch nicht davon ausgehen, daß es parallele, sich gegenseitig beeinflussende Ausführungen von Transaktionen gibt, die ebenfalls zu konsistenten Zuständen führen, welche nicht schon durch nichtüberlappende Ausführungen erzeugt werden können.

Wir müssen daher einen pessimistischen Standpunkt einnehmen dergestalt, daß jeder Datenbanken-Zustand als inkonsistent angesehen wird, dessen Konsistenz mit den vorhandenen Mitteln nicht explizit nachweisbar ist.

Die Menge der *konstruierbaren* konsistenten Datenbanken-Zustände in unserem Modell ist daher der Anfangszustand sowie jeder Zustand, der durch eine nichtüberlappende Ausführung einer beliebigen Folge von Transaktionen aus TR entsteht. Aus Gründen, die erst später klar werden, sind nur solche Folgen für unsere Zwecke verwendbar, die höchstens eine Ausführung jeder Transaktion aus TR enthalten. Ein Zustand ist für uns also nur dann konsistent, wenn er aus dem Anfangszustand der Datenbanken durch nichtüberlappende Ausführung der Transaktionen einer (eventuell leeren) Teilmenge von TR entsteht.

Serielle Logs. Statt des etwas holprigen Wortes "nichtüberlappend" ist die Bezeichnung **seriell** üblich. Die seriellen Logs sind so wichtig, daß wir eine eigene Abkürzung für diese Teilmenge von L einführen.

Bei seriellen Logs werden wir gelegentlich die *Notationsform* $\mathrm{l} = \mathrm{T}_{i1} \ldots \mathrm{T}_{in}$ benutzen, sofern die Transaktionen schon bekannt sind.

Definition: Für $1 \leq \mathrm{i} \leq \mathrm{n}$ sei (das Wort) $\mathbf{T}_i := \mathrm{R}_i\mathrm{W}_i$. Die **Konkatenation** von Worten notieren wir durch Hintereinanderschreibung.

$\mathbf{S} := \{ T_{i1}....T_{in} \mid \{i_1,.. ..,i_n\}$ ist eine Permutation von $\{1,...,n\} \}$

Ein serieller Log erhält offensichtlich die Konsistenz der Datenbanken. Die seriellen Logs sind aber keineswegs die einzigen Logs, die dies tun. Beispiel:

```
l2:
T1   r(x,y)-------w(x)
T2           r(y)------w(y)
T3                              r(x,y)--w(x,y)

l3:
T1   r(x,y)--w(x)
T2                r(y)--w(y)
T3                               r(x,y)--w(x,y)
```

Der Log l_2 erhält, obwohl nicht seriell, die Konsistenz der Datenbanken, da es *unter allen "Umständen"* den gleichen Endzustand der Datenbanken erzeugt wie l_3.

Interpretationen. Die konkreten Inhalte von Datenbank-Objekten hängen von folgenden "Umständen" ab:

- von den Wertemengen, die einzelnen Typen der Datenbank-Objekte bzw. den Operandenstellen der Transaktionen zugeordnet werden;
- vom Anfangszustand der Datenbank;
- von der funktionalen Wirkung der Transaktionen.

Unter einer **Interpretation** versteht man eine Festlegung all dieser Einzelheiten. Wir könnten Interpretationen auch formal definieren. Wir müßten jedoch für diese Einzelheiten relativ viele formale Bezeichnungen einführen, was zu aufwendig wäre. Interpretationen sollten auch so ausreichend zu verstehen sein.

Mit Hilfe einer Interpretation kann der Anfangszustand der Datenbank und der Zustand nach jedem Ereignis eines Logs konstruiert werden, d.h. eine Interpretation definiert zu jedem Log eine Folge von Zuständen der Datenbank.

Wir können nun die obige Ausdrucksweise, daß zwei Logs unter allen Umständen den gleichen Endzustand der Datenbanken erzeugen, mit Hilfe von Interpretationen präzisieren.

Definition: Seien l, l' $\in$ präfixe(L). l und l' heißen **fs-äquivalent** (bzw. **Endzustands-äquivalent**) $\Longleftrightarrow$ bei allen Interpretationen ist der Datenbanken-Zustand nach l und l' der gleiche.

Die Abkürzung fs steht für 'final state', also Endzustand. Ein Beispiel für ein Paar fs-äquivalenter Logs ist l_2 und l_3. Wenn man in einem konkreten Fall die fs-Äquivalenz von zwei Logs anhand der gegebenen Definition prüfen wollte, würde man nie fertig, denn es gibt i.a. unendlich viele Interpretationen. Dennoch ist die fs-Äquivalenz sehr einfach entscheidbar; wir werden aber erst später ein Verfahren angeben.

8.3.2 Serialisierbarkeit

Ein Log, der fs-äquivalent zu irgendeinem seriellen Log ist, erhält somit garantiert die Konsistenz der Datenbank. Solche Logs nennen wir serialisierbar:

Definition: l $\in$ L heißt **fs-serialisierbar** $\Longleftrightarrow$ $\exists$ l' $\in$ S und l und l' sind fs-äquivalent.
l' heißt auch (fs-) **Serialisierung** von l.
SR ist die Menge der serialisierbaren Logs.

In der Literatur ist Serialisierbarkeit auf viele verschiedene Arten definiert worden, teilweise informell, teilweise in unterschiedlichen formalen Modellen. Alle Definitionen sind insofern gleichartig, als sie von einem serialisierbaren Log stets verlangen, daß er das gleiche bewirkt wie irgendein serieller Log. Die Definitionen beziehen sich stets auf eine Äquivalenz zwischen Logs. Die Unterschiede entstehen durch Bezug auf verschiedene Äquivalenzbegriffe.

Die hier angegebene Definition bezieht sich auf die Endzustands-Äquivalenz, sie wurde von zwei “Klassikern” der einschlägigen Literatur ([Pa79] und [Be+79]) übernommen. Wir werden später auch andere

Varianten der Serialisierbarkeit bzw. andere Äquivalenzen von Logs betrachten.

Das Prädikat serialisierbar wird z.T. auch auf CC-Mechanismen angewendet. Gemeint ist dann, daß alle effektiven Folgen von Zugriffen (in diesem Sinne verstehen wir Logs ja stets), die der CC-Mechanismus verursachen kann, serialisierbar sind. Wir werden die Bezeichnung serialisierbar hier nicht in diesem Sinne benutzen.

Bei einigen Gelegenheiten werden wir auch Datenbanken-Zustände im Verlauf von Logs betrachten, also Zustände, die von einem Präfix eines Logs erzeugt werden, welches ggf. sogar mehr Lese- als Schreiberéignisse enthält. Auch für diese Teil-Logs ist der Begriff Serialisierbarkeit sinnvoll. Die obige Definition deckt diesen Fall nicht ab, so daß eine eigene Definition erforderlich wird.

Definition: Sei l $\in$ präfixe(L). l heißt **Präfix-fs–serialisierbar** $\iff$ $\exists$ ein l' $\in$ präfixe(S) mit der gleichen Menge von Schreiberéignissen wie in l, und l und l' sind fs-äquivalent.

Die hier gegebene Definition von Serialisierbarkeit enthält die oben gegebene als Sonderfall. Sie ist aber etwas umständlich und daher wenig gebräuchlich. Wir werden alle späteren Definitionen von Varianten der Serialisierbarkeit formal stets auf L beziehen, im Bedarfsfall jedoch die auf präfixe(L) bezogene Definition darunter verstehen.

Ein serialisierbarer Log hinterläßt die Datenbank in einem konsistenten Zustand. Dies war auch der Ausgangspunkt unserer Überlegungen am Anfang dieses Abschnitts. Tatsächlich leistet die Serialisierbarkeit jedoch mehr: die Datenbank befindet sich *am Ende* des Logs sogar in einem integren Zustand, sofern alle Transaktionen von den Benutzern rechtzeitig gestartet wurden. Die Serialisierung des Logs gibt nämlich die logische Reihenfolge der Transaktionen an. Die Serialisierbarkeit schließt auch den Verlust von Änderungen aus, soweit diese Änderungen nicht ohnehin verloren wären, weil sie später gelöscht werden. Diese Ausnahme ist nicht ganz einfach zu behandeln, sie betrifft sogenannte tote Werte. Wir werden sie später genauer untersuchen.

8.3.2.1 Schwache Serialisierbarkeit

Die Serialisierbarkeit ist hinreichend für die Erhaltung der Konsistenz der Datenbank, sie ist aber nicht notwendig, d.h. sie ist eigentlich etwas zu restriktiv. Als Beispiel betrachte man den oben definierten Log l_1. l_1 ist nicht serialisierbar (Beweis: Übung!), erhält aber die Konsistenz, denn er ist fs-äquivalent zum Log, der nur aus T_1 besteht. Dieser Log ist auch seriell, enthält aber nicht alle Transaktionen, ist also auf eine andere Menge TR zu beziehen.

Nach unseren früheren Bemerkungen über die Menge der konstruierbaren konsistenten Datenbanken-Zustände erhält ein Log genau dann die Konsistenz der Datenbanken, wenn er fs-äquivalent zu der seriellen Ausführung einer Teilmenge aller Transaktionen ist. Einen solchen Log nennen wir **schwach serialisierbar**. Die schwache Serialisierbarkeit ist im Rahmen unserer Randbedingungen hinreichend und notwendig für die Erhaltung der Konsistenz der Datenbanken.

Definition: l $\in$ L heißt **schwach serialisierbar** $\iff$ $\exists$ l' $\in$ präfixe(S) und l und l' sind fs-äquivalent.
WSR ist die Menge der schwach serialisierbaren Logs.

Der Buchstabe W steht für weak(ly).

Ein serialisierbarer Log ist auch schwach serialisierbar, denn S ist Teilmenge von präfixe(S). Die Umkehrung gilt nicht, z.B. ist l_1 schwach serialisierbar, aber nicht serialisierbar. SR ist also i.a. eine echte Teilmenge von WSR. Die echte Teilmengenbeziehung zwischen Mengen von Logs entspricht dem "echt restriktiver" zwischen Restriktionen. Serialisierbarkeit ist also echt restriktiver als schwache Serialisierbarkeit.

Die schwache Serialisierbarkeit schließt, im Gegensatz zur Serialisierbarkeit, den Verlust von Änderungen nicht aus, bspw. in l_1. Deshalb ist sie i.a. ein unzureichendes Korrektheitskriterium für Logs. Ein gewisser Nutzen in diesem Kriterium liegt darin, daß es genau die Konsistenzerhaltung ausdrückt, d.h. ein Datenbanken-Zustand ist inkonsistent, wenn er von einem nicht schwach serialisierbarem Log erzeugt wurde. Ein Zustand, der von einem nicht serialisierbaren Log erzeugt wurde, kann hingegen konsistent sein, weil "nur" eine Änderung verlorenging.

8.3.2.2 Strikte Serialisierbarkeit

Mit "serialisieren" bezeichnen wir die Umformung eines gegebenen Logs in einen fs-äquivalenten seriellen Log, was man sich so vorstellen kann, daß die im Log enthaltenen Transaktionen gegeneinander verschoben werden.

Beim Serialisieren kann ein merkwürdiger Effekt zu beobachten sein: Es ist manchmal zwingend erforderlich, zwei nicht überlappende Transaktionen in ihrer Reihenfolge zu vertauschen. Beispiel:

```
l4:
T1   r(x,y)--------w(y)
T2          r(y,z)-------------w(z)
T3                      r(z,u)------w(u)
```

Sei $l_5 = T_3T_2T_1$

l_4 ist serialisierbar, denn l_4 ist fs-äquivalent zu l_5 und l_5 ist seriell. l_5 ist zugleich der einzige serielle Log, der fs-äquivalent zu l_4 ist. T_1 und T_3 überlappen in l_4 nicht und sind in l_5 in der Reihenfolge vertauscht.

Definition: Eine **Umordnung** ist eine Vertauschung der Reihenfolge zweier nichtüberlappender Transaktionen in zwei Logs.

Eine Umordnung beim Serialisieren kann unter gewissen Umständen von einem Benutzer bemerkt und als Fehlverhalten bewertet werden. Die logische Reihenfolge der Transaktionen (bzw. Änderungen der Realität) ist nämlich diejenige, die in dem seriellen Log angegeben wird. Ein Benutzer hingegen, der im obigen Beispiel die Transaktionen T_1 und T_3 gestartet hat, hat die umgekehrte Reihenfolge geplant.

Wir sind daher besonders interessiert an solchen Logs, die ohne Umordnungen serialisierbar sind. Solche Logs nennen wir strikt serialisierbar.

Definition: $l \in L$ heißt **strikt serialisierbar** $\iff \exists\, l' \in S$, l, l' sind fs-äquivalent und ohne Umordnungen.
SSR ist die Menge der strikt serialisierbaren Logs.

Offensichtlich ist SSR i.a. eine echte Teilmenge von SR.

Umstände, unter denen ein Benutzer Umordnungen überhaupt bemerken kann und als fehlerhaft empfindet, sind recht selten. Es ist daher nicht zwingend notwendig zu fordern, daß ein CC-Mechanismus nur strikt serialisierbare Logs zuläßt. Andererseits sind jedoch CC-Mechanismen, die serialisierbare, aber nicht strikt serialisierbare Logs erzeugen, relativ kompliziert, so daß die obige Forderung meist "freiwillig" erfüllt wird.

8.3.3 Herbrand-Interpretationen

Wir hatten oben die Frage, wie denn die fs-Äquivalenz zweier Logs effektiv entschieden werden kann, zurückgestellt und kommen jetzt darauf zurück.

Das Problem besteht darin, daß wir eine Aussage folgender Art beweisen wollen: Ein Objekt enthält unabhängig von der Wahl der Interpretation bei verschiedenen Gelegenheiten (hier: am Ende zweier Logs) stets den gleichen Wert. Wir können diese Aussage aber nicht einfach für alle Interpretationen ausprobieren, denn es gibt mit Ausnahme von uninteressanten Sonderfällen stets unendlich viele Interpretationen.

Zur Lösung dieses und einiger verwandter Probleme führen wir eine spezielle Art von Interpretationen ein, sogenannte Herbrand-Interpretationen. Diese haben die folgende sehr nützliche Eigenschaft: Wenn zwei Objektinhalte bei einer Herbrand-Interpretation gleich sind, dann sind sie bei *jeder* Interpretation gleich. Dank dieser Eigenschaft reduziert sich das o.g. Problem darauf, die Gleichheit der Objektinhalte bei einer Herbrand-Interpretation zu überprüfen. Die fs-Äquivalenz wird dadurch in linearer Zeit entscheidbar.

Herbrand-Interpretationen sind eines der wichtigsten technischen Hilfsmittel in der Theorie paralleler Systeme. Gelegentlich werden sie auch **freie** oder **symbolische** Interpretationen genannt. Die letzte Bezeichnung ist insofern treffend, als man interpretierte Abläufe als symbolische Berechnungen ansehen kann, bei denen nur mit Formeln und Symbolen manipuliert wird, die Formeln aber nicht "ausgerechnet" werden. Herbrand-Interpretationen sind übrigens sehr eng verwandt

einer Term-Algebra.

8.3.3.1 Definition

Kommen wir nun zur Definition. Zu jedem Paar TR/DB gibt es genau eine Herbrand-Interpretation. Anzugeben sind die Wertemengen, der Anfangszustand der Datenbanken und die Verarbeitungsfunktionen der Transaktionen.

(a) Wertemengen: Es werden keine sortenspezifischen Wertemengen unterschieden, sondern es wird nur eine einheitliche Wertemenge verwandt, das sogenannte **Herbrand-Universum**. Dieses enthält Terme über folgenden Konstanten- und Funktionssymbolen:

- Symbole für Konstanten: für jedes $o \in DB$ ist 'o' ein Konstantensymbol.
- Symbole für Funktionen: für jedes i mit $1 \leq i \leq n$ und jedes j mit $1 \leq j \leq |writeset_i|$ ist T_{ij} ein Funktionssymbol. j ist die Nummer eines Ausgabeoperanden von T_i. Hierzu stellen wir uns vor, alle Ausgabeoperanden von T_i seien durchnumeriert, und zwar von links nach rechts in der Parameterliste.

Das Herbrand-Universum enthält nun folgende Elemente, die wir kurz **Terme** nennen:

1. Alle Konstantensymbole sind Terme.
2. Wenn T_{ij} ein Funktionssymbol ist, $t_1,...,t_k$ Terme sind und $k = |readset_i|$, dann ist '$T_{ij}(t_1,...,t_k)$' ebenfalls ein Term (wobei die Terme t_i textuell eingesetzt wurden).
3. Alle Terme des Herbrand-Universums sind durch die Regeln 1 und 2 in endlich vielen Schritten konstruierbar.

(b) Anfangszustand der Datenbank: Das Datenbank-Objekt o enthält den Term 'o' als Anfangswert.

(c) Verarbeitungsfunktion der Transaktionen: Sei T_i aus TR, $k = |readset_i|$. Wenn die Terme $t_1,...,t_k$ als Eingabewerte gelesen wurden,

dann schreibt T_i in seinen j-ten Ausgabeoperanden ($1 \leq j \leq |writeset_i|$) den Term '$T_{ij}(t_1,...,t_k)$'.

Terme kann man auch als Formeln bzw. Ausdrücke, ähnlich wie in Programmiersprachen, ansehen. Diese Ausdrücke müssen aber als Text verstanden werden, was auch durch die Schreibweise in Hochkommata angedeutet wird. Keinesfalls dürfen die Ausdrücke als Ergebnis ihrer Auswertung verstanden werden, Terme sind *uninterpretierte* Ausdrücke.

Betrachten wir nun an einem konkreten Beispiel, welche Inhalte der Objekte sich bei einer Herbrand-Interpretation ergeben. Wir verwenden den oben definierten Log l_3. Wir notieren die Folge der Datenbanken-Zustände in Form einer Tabelle:

Zustand	Inhalt von x	Inhalt von y
Anfangszustand	'x'	'y'
nach R_1(x,y)	'x'	'y'
nach W_1(x)	'T_{11}(x,y)'	'y'
nach R_2(y)	'T_{11}(x,y)'	'y'
nach W_2(y)	'T_{11}(x,y)'	'T_{21}(y)'
nach R_3(x,y)	'T_{11}(x,y)'	'T_{21}(y)'
nach W_3(x,y)	'T_{31}(T_{11}(x,y),T_{21}(y))'	'T_{32}(T_{11}(x,y),T_{21}(y))'

Der Inhalt der Datenbanken ändert sich natürlich nur bei Schreibereignissen.

8.3.3.2 Eigenschaften von Herbrand-Interpretationen

Im folgenden Lemma listen wir einige der nützlichen Eigenschaften von Herbrand-Interpretationen auf, deren Beweis trivial ist.

Lemma 1: Bei einer Herbrand-Interpretation gilt für jeden Log l und jeden im Laufe von l auftretenden Objektinhalt (bzw. Term) t:

a) t hat entweder die Form 'v', v aus Datenbanken, oder die Form 'T_{ij}(...)'.

b) Man kann an t erkennen, ob es sich um den Anfangswert des Objekts oder um einen später geschriebenen Wert handelt. Im zweiten Fall kann man weiter erkennen,

- von welcher Transaktion t geschrieben wurde,
- in den wievielten Ausgabeoperanden dieser Transaktion (also in welches Objekt) t geschrieben wurde,
- welche Eingabewerte (-terme) von dieser Transaktion bei ihrem Leseereignis gelesen wurden.

c) Man kann an t erkennen, in welchem Objekt dieser Wert stehen muß, d.h. es gibt keinen Wert, der in zwei verschiedenen Objekten auftreten kann.

d) Jeder Wert wird im Laufe von l höchstens einmal geschrieben.

e) Man kann aus t die Baumstruktur der Rechenschritte rekonstruieren, die zur Bildung von t führten.

Aus e können wir nun sofort den folgenden Satz ableiten:

Satz 2: Sei t der Inhalt eines Datenbank-Objekts nach einem Präfix eines Logs l bei einer Herbrand-Interpretation. Aus t können wir den Inhalt dieses Objekts bei jeder anderen Interpretation I ableiten.

Beweis: Wir fassen t als Ausdruck auf und werten ihn gemäß I aus. Der resultierende Wert ist der gesuchte Objektinhalt. Aus Lemma 1e folgt, daß sich dieser Wert auch bei I ergeben hätte. q.e.d.

Diesen Satz wollen wir am obigen Beispiel illustrieren. Als spezielle Interpretation nehmen wir an:

- Alle Wertemengen: integer
- Anfangswert von x: 10; von y: 20
- Verarbeitungsfunktionen:
  T_1: x := x + y;
  T_2: y := y + 100;
  T_3: (x,y) := (x-y,0);

Zustand	Inhalt von x	Inhalt von y
Anfangszustand	10	20
nach R_1(x,y)	10	20
nach W_1(x)	30	20
nach R_2(y)	30	20
nach W_2(y)	30	120
nach R_3(x,y)	30	120
nach W_3(x,y)	-90	0

Die Auswertung von Termen sei anhand des Endinhalts von x exemplarisch vorgeführt:
$T_{31}(T_{11}(x, y), T_{21}(y)) = T_{31}(T_{11}(10,20), T_{21}(20)) = T_{31}(30, 120) = -90$

Man beachte, daß sich aus den Termen die zeitliche Reihenfolge mancher Ereignisse nicht exakt rekonstruieren läßt, die zu diesem Term führten. Beispielsweise hätten W_1 und R_2 in umgekehrter Reihenfolge auftreten können.

Mit Hilfe von Satz 2 folgen sofort die beiden folgenden Korollare:

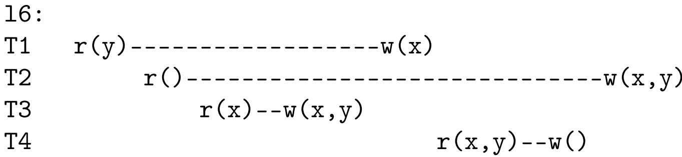
Korollar 3: l,l' $\in$ präfixe(L) sind fs-äquivalent $\iff$ l und l' erzeugen unter einer Herbrand-Interpretation den gleichen Endzustand der Datenbanken.

Korollar 4: Die fs-Äquivalenz zweier Logs ist in einer Zeit entscheidbar, die linear in der Summe ihrer Längen und |DB| ist.

8.3.4 Tote Transaktionen

8.3.4.1 Einführung

Ein (strikt) serialisierbarer Log hinterläßt die Datenbank in einem konsistenten Zustand. Dies impliziert aber nicht, daß nicht *zwischenzeitlich* ein inkonsistenter Datenbanken-Zustand aufgetreten und von einer anderen Transaktion gelesen worden sein könnte. Hierzu ein Beispiel:



Die Datenbank ist nach dem Ereignis W_1 offenbar inkonsistent, denn das Präfix von l_6 bis einschließlich W_1 ist nicht schwach serialisierbar.

T_4 liest inkonsistente Daten und gibt sie potentiell an den Benutzer weiter. Da T_4 eine Lesetransaktion ist, kann ihre inkonsistente Sicht nicht zu einer weiteren Verletzung der Konsistenz der Datenbanken führen.

Eine Variante dieses Beispiels kann gewonnen werden, indem man $\text{writeset}_4 = \{ x \}$ wählt. T_4 ist jetzt zwar keine Lesetransaktion mehr, ihre Ausgabe in x wird jedoch überschrieben, ohne vorher gelesen worden zu sein.

In beiden Varianten hat T_4 keinen Einfluß auf den Endzustand der Datenbanken. Solche Transaktionen nennen wir **tot**.

In diesem Abschnitt werden wir i.w. den Begriff tot und sein Gegenteil, **lebendig**, exakt beschreiben. Wir werden zeigen, daß in serialisierbaren Logs nur tote Transaktionen inkonsistente Daten lesen können. Im folgenden Abschnitt werden wir dann etwas allgemeiner die Frage diskutieren, ob überhaupt temporäre Inkonsistenzen zugelassen werden dürfen.

Definition: Eine **Aktion** ist (im Kontext der LS-Modelle!) das *Schreiben* eines Datenbank-Objekts anläßlich eines Schreibereignisses W_i in einem Log l[24]. Für alle $o \in \text{writeset}_i$ findet eine eigene Aktion statt, die wir mit dem Symbol A_{ij} identifizieren, wenn o der j-te Ausgabeoperand von T_i ist.

Definition: Sei $l \in L$. Ein Wert (bzw. die Aktion, die ihn schreibt) ist **lebendig** in l, wenn er entweder

[24] In diesem Abschnitt benötigen wir keinen Begriff für das Lesen eines Objekts, Aktionen werden daher automatisch als Schreib-Aktionen verstanden.

- bis zum Ende von l in seinem Datenbank-Objekt unverändert bleibt, oder
- von einer anderen Transaktion gelesen wird, die wenigstens einen lebendigen Wert schreibt.

Eine Transaktion ist **lebendig** in l, wenn sie wenigstens eine in l lebendige Aktion enthält.

Lesetransaktionen sind stets tot. Bei allen anderen Transaktionen bzw. bei allen Aktionen hängt die Lebendigkeit immer von der Position der Ereignisse in dem Log l ab.

Eine Transaktion ist schon lebendig, wenn sie eine einzige lebendige Aktion enthält. Wenn keine Lesetransaktionen vorhanden sind, impliziert die Abwesenheit toter Aktionen in einem Log die Abwesenheit toter Transaktionen.

8.3.4.2 Verluste

Ein toter Wert in einem Log entspricht in der Realität dem sofortigen oder späteren Verlust von Information in der Datenbank. Wenn ein toter Wert allerdings gelesen wurde, so kann er an die Umwelt zurückgegeben worden sein und hat dann ggf. seinen Zweck erfüllt; wenn er anschließend absichtlich gelöscht wird, kann man dies schlecht eine Parallelitätsanomalie nennen. Es bietet sich daher an, zwei Arten von Verlusten zu unterscheiden:

1. **serielle Verluste**: Dies sind Verluste bzw. tote Aktionen in seriellen Logs (oder seriellen Teilen von Logs), z.B.:

   ```
   T1   r(x)---w(x)
   T2                 r()---w(x)
   ```

   A_{11} ist tot, denn der Inhalt von x wird ungelesen gelöscht. T_2 schreibt x, ohne es vorher zu lesen.
2. **Verluste durch Parallelität**: Dies sind Verluste bzw. tote Aktionen in nichtseriellen Logs, bei denen eine Transaktion in einem Objekt einen *anderen* Wert überschreibt als den, den sie gelesen hat. Ein Beispiel ist der Log l_1.

Verluste durch Parallelität können durch einen CC-Mechanismus vermieden werden, indem die betroffenen Transaktionen eben nicht parallel ausgeführt werden, serielle Verluste hingegen nicht: Ursache sind Transaktionen, bei denen nicht $\text{writeset}_i \subseteq \text{readset}_i$ gilt. Auf diese Mengen hat ein CC-Mechanismus jedoch keinen Einfluß, er kann keine seriellen Verluste verhindern.

Am Ende dieses Abschnitts werden wir zeigen, daß die Abwesenheit von beiden Arten von Verlusten in serialisierbaren Logs eng zusammenhängt.

8.3.4.3 Feststellung der Lebendigkeit

Die Definition von Lebendigkeit ist rekursiv, an einer Stelle erscheint ein Existenz-Quantor. Die effektive Entscheidung, ob ein gegebener Wert tot ist oder nicht, scheint daher nicht ganz einfach. Dieses Problem können wir wieder mit Hilfe von Herbrand-Interpretationen lösen.

Satz 5: Sei $l \in L$, A_{ij} eine Aktion in l, die bei einer Herbrand-Interpretation den Term t_{ij} schreibt. A_{ij} bzw. t_{ij} ist lebendig in l $\Longleftrightarrow$ $\exists$ ein Objekt o, welches nach l bei einer Herbrand-Interpretation einen Term t enthält, in dem das Funktionssymbol T_{ij} auftritt (d.h. t_{ij} ist in t als Unterterm enthalten oder ist gleich t).

Beweis: "$\Rightarrow$": Klar, wenn der von A_{ij} geschriebene Wert in seinem Objekt nicht mehr in l verändert wird ("unechter" Unterterm von t). Andernfalls wird dieser Wert von einer anderen lebendigen Transaktion T' gelesen und erscheint als Unterterm in den Termen, die diese Transaktion schreibt. Für die lebendigen Aktionen von T' gilt die gleiche Argumentation. Da l endlich lang ist, kann t_{ij} nur endlich oft weitertransformiert werden, d.h. irgendwann ist ein Term erreicht, der Endzustand seines Datenbank-Objekts ist.

"$\Leftarrow$": Klar, falls $t = T_{ij}(...)$. Andernfalls können wir aus der Struktur des Terms t gemäß Lemma 1e eine Kette von Transaktionen ableiten, die t_{ij} zu t weiterverarbeitet hat. Die letzte dieser Transaktionen ist lebendig, da sie den Endinhalt eines Datenbank-Objekts schreibt, alle

früheren sind lebendig, da sie einen Wert schreiben, der von einer lebendigen Transaktion gelesen wird, nämlich der jeweils nächsten in der Kette. t_{ij} wird also von einer lebendigen Transaktion gelesen. q.e.d.

Die Lebendigkeit einer Aktion in l kann somit dadurch entschieden werden, daß der Endzustand der Datenbanken nach l bei einer Herbrand-Interpretation gebildet und nach dem Funktionssymbol T_{ij} durchsucht wird. Dies ist mit einem Zeitaufwand möglich, der linear in der Größe von |DB|, TR und der Länge von l ist.

8.3.4.4 Tote Transaktionen in FS-äquivalenten Logs

Eine weitere wichtige Anwendung von Herbrand-Interpretationen ist die folgende Charakterisierung von fs-äquivalenten Logs:

Lemma 6: Seien l, l' aus L fs-äquivalent. Dann gilt:

a) l und l' haben die gleiche Menge lebendiger Aktionen.

b) l und l' haben die gleiche Menge lebendiger Transaktionen.

c) Bei jeder Interpretation liest jede lebendige Transaktion in l die gleiche Eingabe wie in l'.

Beweis: a folgt direkt aus Satz 5, b aus a, c aus b und Lemma 1b. q.e.d.

Die Umkehrung von Lemma 6 gilt nicht. Als Gegenbeispiel betrachte man den Log

```
T1   r(x)--w(y)
T2             r(y)--w(z)
T3                       r(u)--w(y)
T4                                 r(y)--w(v)
```

und mit den gleichen Transaktionen den Log $T_3T_4T_1T_2$. In beiden Logs sind alle Aktionen und Transaktionen lebendig und lesen die gleichen Daten. Trotzdem sind sie nicht fs-äquivalent.

8.3.4.5 Sichten von lebendigen Transaktionen

Mit Hilfe von Satz 5 und Lemma 6 können wir nun, wie schon eingangs angekündigt, folgendes beweisen:

Satz 7: Sei l ∈ SR, T_i ∈ TR, T_i lebendig in l. Dann ist die Sicht von T_i in l konsistent.

Beweis: Sei l∈ S, l, l' fs-äquivalent. Lemma 6 impliziert, daß T_i auch in l' lebendig ist und die gleiche Sicht wie in l hat. Da in serialisierbaren Logs alle Sichten konsistent sind, folgt die Behauptung. q.e.d.

Wenn nun alle Transaktionen in SR lebendig sind, dann folgt daraus mit Satz 7, daß alle Sichten konsistent sind. (Lesetransaktionen dürfen dann allerdings nicht auftreten.)

Das folgende Lemma liefert uns eine einfach zu formulierende Bedingung, die die Abwesenheit toter Transaktionen in SR garantiert: Für alle i, $1 \leq i \leq n$, muß gelten: $\emptyset \neq \text{writeset}_i \subseteq \text{readset}_i$.

Lemma 8: Falls für alle i, $1 \leq i \leq n$, gilt: $\text{writeset}_i \subseteq \text{readset}_i$, so enthält ein serialisierbarer Log l keine toten Aktionen.

Beweis: Sei zunächst l ∈ S. Jeder Wert, der bei einer Herbrand-Interpretation im Laufe von l in ein Objekt x geschrieben wird, bleibt dort unverändert oder wird in x weiter transformiert bis zum Endzustand von x. Alle Aktionen sind daher lebendig in l. Für ein l ∈ SR - S folgt die Behauptung direkt mit Lemma 6b. q.e.d.

Lemma 8 und Satz 7 beweisen die obige Behauptung, daß $\emptyset \neq \text{writeset}_i \subseteq \text{readset}_i$ für alle i impliziert, daß alle Sichten in serialisierbaren Logs konsistent sind. Tatsächlich impliziert diese Voraussetzung noch erheblich mehr, dies wird in Abschnitt 4.4.5 gezeigt werden.

Die Umkehrung von Lemma 8 gilt i.a. natürlich nicht. Sie gilt aber "fast", denn sie gilt außer für einige uninteressante Randfälle immer:

Lemma 9: Angenommen, alle Logs in SR (oder S) enthalten keine toten Aktionen und für ein i, $1 \leq i \leq n$, gilt: $\text{writeset}_i \not\subseteq \text{readset}_i$. Dann gilt für alle j, $1 \leq j \leq n$, $i \neq j$, $\text{writeset}_i \cap \text{writeset}_j = \emptyset$.

Beweis: Angenommen, die Behauptung gilt nicht. Man wähle ein T_j mit $\text{writeset}_j \cap \text{writeset}_i \neq \emptyset$ und einen beliebigen seriellen Log l, der mit $R_j W_j R_i W_i$ beginnt. Dann überschreibt W_i ein Objekt von T_j, d.h. l enthält eine tote Aktion, Widerspruch. q.e.d.

Lemma 9 hat folgende Konsequenz: Bei Abwesenheit toter Aktionen in SR oder S kann kein im Laufe eines Logs geschriebener Wert seriell verlorengehen, sondern nur Anfangswerte. Da die Anfangswerte letztlich aber auch irgendwann in früherer Zeit, die im Log nicht mehr dargestellt wird, erzeugt wurden (oder ersatzweise durch eine initialisierende Transaktion T_0 mit $\text{writeset}_0 = \text{DB}$ erzeugt wurden), können überhaupt keine seriellen Verluste auftreten (da $\text{writeset}_0 = \text{DB}$, könnte die Bedingung $\text{writeset}_i \cap \text{writeset}_0 = \emptyset$ nicht erfüllt werden ($\text{writeset}_i \neq \emptyset$!)).

8.3.5 Temporäre Anomalien in serialisierbaren Logs

Im Beispiel l_6 aus dem vorigen Abschnitt hatte eine Transaktion in einem strikt serialisierbaren Log eine inkonsistente Sicht. Ursache war eine temporäre Inkonsistenz der Datenbank.

Eine temporäre Inkonsistenz tritt auch im Standardbeispiel der Umbuchung eines Beldbetrags zwischen zwei Konten auf. Die Ursache für die Inkonsistenz liegt hier darin, daß die Transaktion ihre Ergebnisse in mehreren Schritten schreibt. Derartige Inkonsistenzen sind vergleichsweise harmlos, denn sie werden von der gleichen Transaktion behoben, die sie verursachte. Wenn alle Schreiboperationen auf einmal beim Commit durchgeführt werden, können sie gar nicht auftreten. Da wir in unseren Modellen das atomare Schreiben beim Commit voraussetzen, können wir temporäre Inkonsistenzen vom Typ "Zwischenergebnisse" in den Modellen nicht nachbilden, d.h. im Beispiel l_6 liegt eine ganz andere Ursache vor.

Die eigentliche Ursache von temporären Inkonsistenzen, die in unserem Modell auftreten können, sind *nichtserialisierbare Präfixe von Logs*. In l_6 ist das Präfix bis W_1 nicht serialisierbar, und es liegt ein **teilweiser Verlust** vor, d.h. die Wirkung einer Transaktion geht dadurch

teilweise verloren, daß eine echte, nichtleere Teilmenge ihrer Ausgabeoperanden in einer Art gelöscht wird, die in einem seriellen Log nicht auftreten könnte.

Teilweise Verluste sind jedoch nicht die einzige denkbare Ursache für temporäre Inkonsistenzen (z.B. wähle man in l_6 $writeset_3 = \{ y \}$), entscheidend ist die Nichtserialisierbarkeit.

Daneben unterscheiden sich die temporären Inkonsistenzen in unserem Modell von denen vom Typ "Zwischenergebnisse" in einem weiteren wichtigen Merkmal: Die Transaktionen, die die Inkonsistenz verursachten, können sie nicht selbst beheben, da sie sie bei ihrem einzigen Schreibereignis verursachen und damit beendet sind. Die Reparatur kann nur von einer oder mehreren anderen, eventuell schon gestarteten Transaktion zu einem späteren Zeitpunkt bewirkt werden. In l_6 bspw. repariert T_2 die Inkonsistenz.

Die Reparatur besteht i.w. darin, die inkonsistenten Teile der Datenbanken zu löschen. Dies muß nicht sofort geschehen, die inkonsistenten Werte können sogar noch einige Male weiterverarbeitet werden. Entscheidend ist, daß sie den Endzustand nicht beeinflussen.

Eine reparierende Transaktion ist eigentlich in ihrem Verhalten völlig autonom. Insbesondere kann sie zurückgesetzt werden, sofern es sich nicht um ein DBS ohne Rollback-Möglichkeit handelt, was aber sehr ungewöhnlich ist[25]. Da es keine Garantie gibt, daß die reparierende Transaktion tatsächlich zu Ende geführt wird, sind die temporären Inkonsistenzen vom Typ "Fremdreparatur" i.a. nicht akzeptabel.

Oft werden zurückgesetzte Transaktionen sofort automatisch neu gestartet, z.B. nach einem Rollback zur Deadlock-Auflösung. Durch den automatischen, eventuell wiederholten Neustart wird letztlich doch garantiert, daß der inkonsistente Teilzustand gelöscht wird. Die Zeitdauer bis zur Löschung kann jedoch erheblich länger sein als die Ausführungsdauer der löschenden Transaktion. Während dieser Wartezeit müßten die inkonsistenten Objekte für neue Transaktionen gesperrt

[25] Transaktionen werden auch in parallelen Programmen, dort insb. bei parallelen abstrakten Datentypen eingesetzt. In diesem Kontext ist die Abwesenheit von Rollback durchaus denkbar.

werden. Deswegen sind temporäre Inkonsistenzen selbst im Falle des automatischen Neustarts nicht akzeptabel.

Der später gelöschte Teilzustand, der die temporäre Inkonsistenz verursacht, ist meist tot, so auch im Log l_6. Zwingend notwendig ist dies jedoch nicht, denn die betreffenden Werte können auch einzeln weiter transformiert werden, ohne eine permanente Inkonsistenz der Datenbanken zu verursachen. Beispiel:

```
l7:
T1  r(u,x)---------------------w(x,z)
T2        r()--w(x,y)
T3                   r(x)--w(u)
T4                                   r(x)--------w(v)
T5                                       r(x,y,z)----w()
T6           r()----------------------------------------w(x)
```

l_7 ist fs-äquivalent zu $T_1T_4T_2T_3T_6$ und T_5 an beliebiger Stelle, also serialisierbar. Der Zustand nach W_1 ist inkonsistent, aber kein Wert ist tot. Das Objekt x wird zunächst für einen “Datenfluß” von T_2 nach T_3 benutzt, danach für einen Datenfluß” von T_1 nach T_4, und schließlich von T_6 blind überschrieben. Die Reihenfolge der beiden Datenflußnutzungen von x kann ohne Auswirkung auf den Endzustand vertauscht werden.

l_7 ist nicht strikt serialisierbar, denn in allen fs-äquivalenten Logs sind T_2 und T_4 umgeordnet. Man kann zeigen, daß in strikt serialisierbaren Logs ohne tote Aktionen keine temporären Inkonsistenzen auftreten können.

Statt eines teilweisen Verlustes mit anschließender Inkonsistenz der Datenbanken kann natürlich auch ein völliger Verlust einer Transaktion eintreten. Im Beispiel l_6 wäre hierzu $\text{writeset}_3 = \{ x \}$ zu wählen. T_3 wäre dann tot. T_4 liest jetzt zwar konsistente Daten, aber nicht mehr die aktuell gültigen, da ja T_3 verlorenging.

Diese Erscheinung können wir analog einen **temporären Verlust** nennen. Der Verlust ist insofern temporär, als er später gegenstandslos wird, weil die betroffenen Teile der Datenbanken gelöscht bzw.

überschrieben werden. Der Zeitraum, in dem er zu nicht aktuellen Sichten führen kann, ist somit beschränkt. Im abgewandelten Beispiel l_6 "repariert" wieder T_2 den temporären Verlust.

Auch bei dieser temporären Anomalie sind reparierende Transaktionen erforderlich, welche möglicherweise zurückgesetzt werden können. Daher sind auch temporäre Verluste i.a. nicht akzeptabel.

Beide vorgestellten temporären Anomalien können in strikt serialisierbaren Logs auftreten. Hieraus folgt, daß die *Serialisierbarkeit* wie auch die strikte Serialisierbarkeit i.a. *kein hinreichendes Korrektheitskriterium für Logs* sind.

8.3.6 On-line-Scheduler

Die Serialisierbarkeit ist zunächst ein Korrektheitskriterium für die verzahnte Ausführung *vollständiger* Transaktionen. Wir können also erst, nachdem alle laufenden Transaktion beendet sind, entscheiden, ob die aufgetretene Verzahnung zulässig war oder nicht. Bei den meisten CC-Verfahren müssen wir aber sofort entscheiden, ob eine Aktion ausgeführt werden kann, wir können nicht warten, bis alle laufenden Transaktion ihre restlichen Zugriffe durchgeführt haben und beendet sind. Betrachten wir hierzu Bild 8.1. Jede einzelne Transaktion verursacht eine Sequenz von Aktionsaufrufen.

Abbildung 8.1: On-line-Scheduler

Immer dann, wenn eine Transaktion erneut eine Aktion aufruft, muß eine DBMS-Komponente, die wir **Scheduler** nennen, entscheiden, ob diese Aktion sofort oder erst später ausgeführt wird. Diese Entscheidungen bestimmen, wie die einzelnen Sequenzen von Aktionsaufrufen zu

einer einzigen Sequenz von effektiven Aktionsausführungen zusammengesetzt werden. Die Entscheidungen können nur auf Informationen über die neuen Aktionen und über die bisher schon durchgeführten Aktionen der Transaktionen basieren, nicht auf zukünftig vielleicht folgenden Aktionsaufrufen (diese müßten nämlich irgendwie vorab deklariert werden). Wir sprechen daher von einem **On-line-Scheduler**.

Den Ankunftsstrom von Aktionsaufrufen können wir ebenfalls durch einen Log darstellen[26], den wir i.f. als Aufruflog bezeichnen. Die Aufgabe des Schedulers besteht u.a. darin, Aktionen in anderer Reihenfolge als im Aufruflog auszuführen, wenn sonst Parallelitätsanomalien eintreten würden. Anders gesagt formt ein Scheduler den Aufruflog in einen effektiv wirksamen Log um. Da wir den Aufruflog ebenfalls als Log darstellen, bildet ein Scheduler die Menge L auf sich selbst ab. Genauer sollen natürlich nicht alle Logs als Bild auftreten, sondern nur die korrekten Logs. Wenn **K** eine Menge von korrekten Logs ist, dann ist ein Scheduler für K eine Funktion **SCH**: L $\rightarrow$ K. Kandidaten für K sind z.B. SR und SSR. SCH formt einen Log i.a. in einen nicht-fs-äquivalenten Log um. Für l aus K sollte natürlich gelten: SCH(l) = l.

Bei einem Aufruflog l können die ersten i Aktionen genau dann sofort ausgeführt werden, wenn präfix(l,i) aus präfixe(K) ist. Dieser Test muß effizient durchführbar sein, d.h. der Zeitbedarf sollte höchstens polynomial in der Länge von l sein. Nun ist das Entscheidungsproblem für SR NP-vollständig (vgl. [Pa79]). NP-vollständig bedeutet: die Menge kann nur dann mit polynomialem Zeitaufwand erkannt werden, wenn das NP-Problem positiv gelöst wird. Das NP-Problem ist die Frage: gilt P = NP? P bzw. NP ist die Menge der Sprachen, die durch deterministische bzw. nichtdeterministische Turingmaschinen in polynomialer Zeit erkennbar ist. Das NP-Problem ist eines der wichtigsten ungelösten Probleme der theoretischen Informatik. Die Gleichheit von P und NP ist allerdings äußerst unwahrscheinlich. Ohne den Nachweis dieser Gleichheit kann kein Algorithmus gefunden werden, der SR (oder je-

[26] Zur Erinnerung: Wir haben Logs als Folgen effektiv wirksamer Aktionen auf der Datenbanken definiert.

de andere NP-vollständige Sprache) in polynomialer Zeit erkennt. Alle bekannten Algorithmen haben exponentiellen Zeitbedarf.

Mit SR ist auch präfixe(SR) NP-vollständig. Es ist daher fast sicher, daß es keinen effizienten Scheduler für SR gibt.

SSR ist i.a. ebenfalls NP-vollständig. Unter speziellen Voraussetzungen (keine toten Aktionen) ist SSR jedoch in polynomialer Zeit erkennbar (s. [Se82]).

8.3.7 Zusammenfassung

Wir haben in diesem Abschnitt die Forderung, daß die Datenbank nach der verzahnten Ausführung mehrerer Transaktionen korrekt bleiben soll, präzisiert und formalisiert. Die Forderung wird erfüllt, wenn der Log fs-serialisierbar ist. Die fs-Serialisierbarkeit ist somit ein Minimalkriterium, andererseits aber nicht ausreichend bzw. praktikabel:

- Wegen der schlechten Entscheidbarkeit muß ein einfacheres, effizienter entscheidbares Kriterium verwendet werden.
- Weil jederzeit mit dem Abbruch von löschenden Transaktionen gerechnet werden muß, die die bei fs-serialisierbaren Logs möglichen temporären Inkonsistenzen nicht tolerabel, d.h. man muß die Präfix-fs-Serialisierbarkeit fordern.

8.4 Konsistente Sichten

Wie wir an vorstehenden Beispielen gesehen haben, ist die fs-Serialisierbarkeit keine hinreichende Voraussetzung für konsistente Sichten. In diesem Abschnitt wollen wir eine Eigenschaft von Logs formulieren, die - ähnlich wie WSR exakt der Konsistenzerhaltung entsprach - exakt konsistenten Sichten entspricht.

Eine Sicht ist offenbar bei den Voraussetzungen in unserem Modell genau dann konsistent, wenn sie *Teil* eines konsistenten Datenbanken-Zustandes ist. Dies bedeutet nicht, daß die Datenbanken in dem Moment, wo das Leseereignis ausgeführt wurde, konsistent sein muß. Beispiel:

```
l8:
T1   r(x)----------------w(y,z)
T2        r(x)----w(x,z)
T3                                 r(x,y)--w(u)
```

Der Datenbanken-Zustand bei R_3 ist inkonsistent. T_3 sieht indes nur einen Teil der Datenbanken und diesen Teil in einem Zustand, wie er nach dem seriellen Log T_1T_2 entstanden wäre. T_3 hat somit eine konsistente Sicht.

Definition: $l \in L$ heißt **schwach Sicht-serialisierbar**, wenn für alle i, $1 \leq i \leq n$, gilt: wenn l_i das Präfix von l bis einschließlich R_i ist, dann gibt es ein l_i' aus präfixe(S), so daß l_i und l_i' in allen Datenbank-Objekten aus readset_i bei der Herbrand-Interpretation (also bei allen Interpretationen) den gleichen Inhalt erzeugen.
WISR ist die Menge der schwach Sicht-serialisierbaren Logs.

Der Buchstabe I in WISR steht als Kürzel für *i*nput. In der Literatur (z.B. [Ca81]) finden sich einige weitere ähnliche Korrektheitsbegriffe.

Die schwache Sicht-Serialisierbarkeit impliziert nicht die Konsistenzerhaltung; l_8 ist ein Gegenbeispiel.

Die Frage liegt nun nahe, ob denn die Kombination von (strikter) Serialisierbarkeit und schwacher Sicht-Serialisierbarkeit ein ausreichendes Korrektheitskriterium ist. Das nächste Beispiel zeigt, daß diese Frage negativ beantwortet werden muß:

```
l9:
T1   r(x)--------------w(y)
T2        r(y)----w(x)
T3                            r(z)--w(x,y)
```

Das Präfix bis W_1 ist nicht schwach serialisierbar, der entstandene Datenbanken-Zustand ist nicht konsistent. Die Inkonsistenz wird von T_3 repariert, sie ist also nur temporär. Wie oben schon diskutiert, sind temporäre Anomalien jedoch i.a. nicht tolerierbar.

Eine zusammenfassende Beurteilung aller bisher vorgestellten Korrektheitskriterien für Logs (mit Ausnahme der Präfix-fs-Serialisierbarkeit, die wir hier noch nicht näher analysiert haben) muß damit negativ ausfallen: Selbst in ihrer Vereinigung schließen sie temporäre Anomalien nicht aus und implizieren nicht die logische Atomarität der Transaktionen. Daneben ist die Definition von SR, SSR und WISR und die Prüfung der Zugehörigkeit eines Logs zu diesen Mengen zu komplex. Daß diese Begriffe trotz dieses negativen Gesamtbildes vorgestellt wurden, hat folgende Gründe:

- Einige Begriffe, bspw. die fs-Serialisierbarkeit, sind intuitiv sehr naheliegend, ihre Unzulänglichkeit im allgemeinen Fall ist nicht offensichtlich.
- Die Begriffe werden in der einschlägigen Literatur häufig benutzt, und es existiert eine inzwischen recht reichhaltige Theorie über sie (speziell Komplexitätstheorie).
- Einige Begriffe sind unter gewissen Randbedingungen (vgl. Lemma 6 und Satz 7) dennoch hinreichend. Sie sind dann nämlich äquivalent zu schärferen Korrektheitsbedingungen, die der logischen Atomarität entsprechen.

8.5 Logische Atomarität

8.5.1 Sicht-Serialisierbarkeit

Wir suchen in diesem Abschnitt nach Korrektheitskriterien für Logs, die direkt der logischen Atomarität entsprechen. Nach unseren bisherigen Überlegungen kann die Menge dieser Logs nur eine echte Teilmenge von SR bzw. SSR und WISR sein.

Die logische Atomarität bedeutet: Es gibt eine gedachte Folge von atomaren Zustandsübergängen der Realität und eine dementsprechende Folge von gedachten Datenbanken-Zuständen. Real vorhanden müssen die Zustände nicht sein, sondern nur als Teilzustände (der Realität bzw. der Datenbanken), wie sie die Benutzer wahrnehmen.

Im Kontext unserer Modelle ist somit in einem Log l die logische Atomarität gewährleistet, wenn es einen seriellen Log l' gibt, so daß alle Transaktionen in l die gleiche Sicht wie in l' haben und der Endzustand nach l und l' gleich ist. Dies nennen wir Sicht-Serialisierbarkeit.

Definition: l, l' $\in$ präfixe(L) heißen **Sicht-äquivalent** $\Longleftrightarrow$ für alle i, $1 \leq i \leq n$, gilt:

1. R_i in l $\Longleftrightarrow$ R_i in l'
2. R_i liest in l und l', sofern vorhanden, die gleichen Werte unter allen Interpretationen.

Definition: l $\in$ L heißt **(strikt) Sicht-serialisierbar** $\Longleftrightarrow$ $\exists$ ein l' $\in$ S und l, l' sind Sicht- *und* fs-äquivalent (und ohne Umordnungen). **ISR** (**SISR**) ist die Menge der (strikt) Sicht-serialisierbaren Logs.

Einige Beziehungen von ISR und SISR zu bereits vorgestellten Teilmengen von L ergeben sich unmittelbar aus den Definitionen:

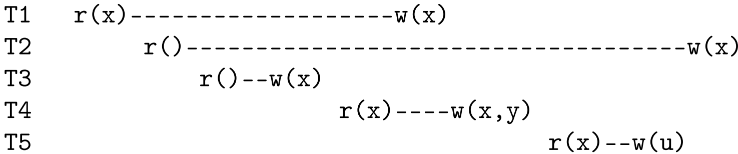
- ISR $\subseteq$ SR
- SISR $\subseteq$ SSR
- SISR $\subseteq$ ISR
- ISR $\subseteq$ WISR

Hieraus folgt weiter:

- ISR $\subseteq$ SR $\cap$ WISR
- SISR $\subseteq$ SSR $\cap$ WISR

Die beiden letzten Teilmengenbeziehungen sind im allgemeinen echt, d.h. die (strikte) Sicht-Serialisierbarkeit ist nicht einfach die Konjunktion von der (strikten) Serialisierbarkeit und der schwachen Sicht-Serialisierbarkeit. Die Echtheit der Inklusionen kann mit dem Log l_9 gezeigt werden.

Zu klären bleibt noch die Frage, ob die Sicht-Serialisierbarkeit eines Logs temporäre Anomalien ausschließt. Dies ist leider nicht der Fall, wie durch eine leichte Abwandlung des Beispiels l_6 gezeigt werden kann:



```
l10:
T1   r(x)------------------w(x)
T2        r()------------------------------------w(x)
T3            r()--w(x)
T4                      r(x)----w(x,y)
T5                                     r(x)--w(u)
```

l_{10} und $T_1T_3T_4T_5T_2$ sind Sicht- und fs-äquivalent und ohne Umordnungen. Nach W_4 ist die Datenbanken wegen der Objekte x und y jedoch temporär inkonsistent, T_2 repariert die Inkonsistenz. Ändert man das Beispiel ab, indem man y aus writeset_4 entfernt, so tritt nach W_1 ein temporärer Verlust ein.

In Logs aus SISR sind also temporäre Anomalien möglich. Sie können jedoch, im Gegensatz zu Logs aus SR, nicht zu inkonsistenten Sichten führen. Einzelne Objekte eines "anomalen" Zustandes können dennoch gelesen werden. Bspw. liest in l_{10} T_5 einen solchen Zustand.

Festzuhalten ist, daß die logische Atomarität der Sicht-Serialisierbarkeit entspricht und daß sie temporäre Inkonsistenzen nicht ausschließt, wenngleich diese von keinem Benutzer als solche erkannt werden können.

Da wir temporäre Anomalien nicht tolerieren können, müssen wir nach Korrektheitsbegriffen suchen, die noch strenger als die Sicht-Serialisierbarkeit sind. Man beachte, daß die Gründe, derentwegen temporäre Anomalien abgelehnt werden, eher technischer Natur sind und nicht aus den informellen Integritätsforderungen, die der Benutzersicht entsprachen, abgeleitet werden können.

Berücksichtigt man die Möglichkeit des Rollbacks ohne Neustart, so muß man fordern, daß auch *alle Teillogs* Sicht-serialisierbar sind. Bei l_{10} ist dies offensichtlich nicht der Fall: man entferne die reparierende Transaktion T_2. Dieses neue strengere Korrektheitskriterium werden wir später genauer beschreiben.

8.5.2 Konflikte

Wenn man die bisherigen Beispiele von Logs, in denen irgendwelche unerwünschten Effekte zu beobachten waren, nach gemeinsamen Merkmalen durchsucht, so fällt wahrscheinlich eines sofort auf: beim Serialisieren werden stets gewisse Ereignisse vertauscht, durch die entweder der Datenfluß zwischen Transaktionen oder die Folge der Werte, die in ein Objekt geschrieben wird, verändert wird. Die Wirkung solcher Veränderungen kann lokal beschränkt bleiben, z.B. bei temporären Anomalien. Zwischen den vertauschten Ereignissen besteht ein sogenannter Konflikt.

Definition: Sei $1 \leq i, j \leq n, i \neq j$.
R_i und W_j stehen in **Konflikt**, wenn $\text{readset}_i \cap \text{writeset}_j \neq \emptyset$.
W_i und W_j stehen in **Konflikt**, wenn $\text{writeset}_i \cap \text{writeset}_j \neq \emptyset$.

Entscheidend dafür, daß zwei Ereignisse nicht in Konflikt stehen, ist folgendes: sie **kommutieren**, d.h. in beiden möglichen Reihenfolgen werden die gleichen Eingaben gelesen und die gleichen Endinhalte in die Ausgabevariablen geschrieben. Wenn man nur Lese- und Schreiboperationen berücksichtigt, ist "kommutieren" äquivalent zu "nicht in Konflikt stehen" gemäß obiger Definition. Wenn man die Semantik der Transaktionen genauer kennt, können auch verändernde Operationen kommutativ sein, z.B. das Aufaddieren auf eine numerische Variable[27]. Wirksam sind diese Techniken nur, wenn man auf das ausschließliche Schreiben beim Commit verzichtet und kontrollierte Interferenzen zwischen Transaktionen zuläßt. In LS-Modellen lassen sich diese Effekte natürlich nicht nachvollziehen, man müßte n-Schritt-Modelle verwenden.

---

[27] Man kann dies zur Erhöhung der Parallelität ausnutzen; Ansätze hierfür werden für numerische bzw. strukturierte Datentypen u.a. in [Ga83] und [Re82] beschrieben. Es treten allerdings erhebliche Komplikationen infolge von Rollback auf.

Um die Fortpflanzung von Rollback zu vermeiden, wendet man teilweise ein anderes Prinzip der Fehlerkorrektur an, die Fehlerkompensation: für jeden Schritt ist ein inverser Schritt vom Programmierer der Transaktion zu liefern. Für derartige Transaktionen müssen also völlig andere Recovery-Mechanismen vorhanden sein.

Die "Liest-von"-Struktur. Wenn in zwei Logs zwei Ereignisse R_i und W_j, die in Konflikt stehen, vertauscht sind, so führt dies zumindest potentiell bei R_i zu einer anderen Sicht. Die Sicht von R_i bleibt jedoch unverändert, wenn R_i gar keine Daten von W_j liest, weil diese von einem anderen W_k überschrieben wurden, welches zwischen W_j und R_i liegt ($W_j \rightarrow W_k \rightarrow R_i$). Für den *Datenfluß* ist entscheidend, welches Schreibereignis zuletzt vor einem Leseereignis ein gelesenes Objekt geschrieben hat.

Definition: Sei $x \in DB$, T_i, $T_j \in TR$, $l \in L$. **T_i liest x von T_j in l** $\iff$ $x \in readset_i$ und $x \in writeset_j$, $W_j \rightarrow R_i$ in l, und es gibt kein W_k mit $x \in writeset_k$ und $W_j \rightarrow W_k \rightarrow R_i$ in l.

Sofern T_i einen Wert in x liest, der noch aus dem Anfangszustand der Datenbank stammt, gibt es keine Transaktion, von der T_i liest. In diesem Fall stellen wir uns eine zusätzliche **Initialisierungstransaktion T_0** vor, die vor dem Log alle Objekte der Datenbank initialisiert, also $readset_0 = \emptyset$ und $writeset_0 = DB$.

Die "liest-von"-Struktur ist gerade der Datenfluß in einem Log. Zwei Logs mit gleichem Datenfluß sind Sicht-äquivalent.

Lemma 10: l, l' aus L sind Sicht-äquivalent $\iff$ für alle $x \in DB$, $T_i,T_j \in TR$ gilt: T_i liest x von T_j in l $\iff$ T_i liest x von T_j in l'.

Beweis: "$\Rightarrow$": Wir betrachten l und l' unter der Herbrand-Interpretation. Jeder Term wird in l und l' höchstens einmal von einer aus dem Term ableitbaren Transaktion geschrieben. Da l und l' Sicht-äquivalent sind, kann jede Transaktion T_i, die einen Wert in x liest, diesen nur von einer eindeutig bestimmten Transaktion T_j lesen.

"$\Leftarrow$": trivial. q.e.d.

Wenn also l und l' Sicht-äquivalent sind und in beiden liest T_i x von T_j, dann müssen W_j und R_i in beiden in der gleichen Reihenfolge auftreten. Anders gesehen darf beim (Sicht-) Serialisieren die Reihenfolge solcher Ereignispaare nicht vertauscht werden.

Die "Überschreibt"-Struktur. Die Gleichheit des Datenflusses impliziert nicht die Gleichheit des Endzustandes, vgl. l_1 und l_{11}):

```
l1:                                 l11:
T1  r(X)--------------w(X)   T1       r(X)------w(X)
T2       r(X)----w(X)         T2  r(X)------w(X)
```

Bei diesen beiden Logs ist die "überschreibt"-Struktur verschieden.

Definition: Sei x $\in$ DB, T_i, T_j $\in$ TR, l $\in$ L. **T_i überschreibt x von T_j in l** $\iff$ x $\in$ writeset_i, x $\in$ writeset_j, $W_j \rightarrow W_i$ und es gibt kein W_k mit x $\in$ writeset_k und $W_j \rightarrow W_k \rightarrow W_i$ in l.

Die temporären oder permanenten Anomalien in den früheren Beispielen entstanden oft dadurch, daß Werte in einer Art überschrieben wurden, die in einem seriellen Log nicht möglich gewesen wäre. Beim Serialisieren sind in diesen Fällen Ereignisse W_i und W_j vertauscht worden, wobei T_i irgendein Objekt von T_j überschrieb.

CP-Serialisierbarkeit. Wir haben nunmehr Beispiele für Vertauschungen spezieller Paare von in Konflikt stehenden Ereignissen gefunden, die zu unerwünschten Effekten führen. Naheliegend ist, die Vertauschung von in Konflikt stehenden Ereignissen beim Serialisieren generell zu verbieten. Diese Idee führt zu der für die Praxis wichtigsten Form der (strikten) Serialisierbarkeit, die wiederum auf einem speziellen Äquivalenzbegriff beruht:

Definition: l,l' $\in$ präfixe(L) heißen **cp-äquivalent** $\iff$ l und l' enthalten die gleiche Menge von Ereignissen, und jedes Paar von in Konflikt stehenden Ereignissen tritt in l und l' in der gleichen Reihenfolge auf.

Die Abkürzung cp steht für *c*onflict *p*reserving, also Konflikt-(Reihenfolge-) erhaltend.

Definition: l ∈ L heißt (**strikt**) **cp-serialisierbar** $\iff$ ∃ ein l' ∈ S und l und l' sind cp-äquivalent (und ohne Umordnungen).
CPSR (**SCPSR**) ist die Menge der (strikt) cp-serialisierbaren Logs.

Das Verhältnis von (S)CPSR zu den bisher bekannten Teilmengen von L ergibt sich i.w. aus dem folgenden Lemma.

Lemma 11: Seien l, l' ∈ L cp-äquivalent. Hieraus folgt:

a) l und l' haben die gleiche "liest-von"- Struktur.
b) l und l' haben die gleiche "überschreibt"-Struktur.
c) l und l' sind Sicht-äquivalent.
d) l und l' sind fs-äquivalent.

Beweis: Die Annahme des Gegenteils von a oder b führt sofort zum Widerspruch zur cp-Äquivalenz. c folgt mit Lemma 10 direkt aus a. d folgt aus b und c: ein beliebiges x ∈ DB wird in l und l' wegen b von der gleichen Transaktion zum letzten Mal geschrieben, die wegen a die gleichen Eingabewerte las. Daher wird in beiden Fällen derselbe Wert geschrieben. q.e.d.

Satz 12: CPSR ⊆ ISR. SCPSR ⊆ SISR.

Beweis: direkt aus Lemma 11. q.e.d.

Die Sicht- und fs-Äquivalenz implizieren zusammen nicht die cp-Äquivalenz. Ein Gegenbeispiel ist l_{10} und seine Serialisierung. Dies zeigt auch, daß es sich in Satz 12 im allgemeinen um echte Inklusionen handelt.

Zusammenfassend erhalten wir die in Bild 8.2 gezeigte Struktur von Beziehungen zwischen den wichtigsten Korrektheitsbegriffen für Logs bzw. Teilmengen von L.

Die Pfeile können gelesen werden als "ist Teilmenge von" oder als "impliziert", wenn man in den Korrektheitsbegriffen denkt. Alle Inklusionen sind im allgemeinen echt, das heißt, es lassen sich Mengen in Datenbanken und TR finden, bei denen sie echt sind. An jedem Pfeil ist ein Beispiel für einen Log angegeben, der die Echtheit der Inklusion zeigt.

Abbildung 8.2: Varianten der Serialisierbarkeit

8.5.3 Serialisierungspunkte

Das Hauptmotiv bei der Bildung der cp-Serialisierbarkeit war der Wunsch, beim Serialisieren keine in Konflikt stehenden Ereignisse vertauschen zu müssen. Das Vertauschen von Ereignissen entspricht einer Änderung der Zeitpunkte, an denen sie stattfinden.

Wir haben bisher für die Zeit eine Ordinalskala verwendet. Zeitpunkte waren 1, ..., 2n. Der Zeitpunkt eines Ereignisses wurde durch seine Stellung in einem Log angezeigt. Im folgenden gehen wir zu einer kontinuierlichen Zeitskala über und sehen alle realen Zahlen zwischen 1 und 2n als mögliche Zeitpunkte an. Auch für diese Zeitpunkte benutzen wir die Notation "$\rightarrow$" um "später" auszudrücken.

Wir können uns das Serialisieren nun so vorstellen, daß die beiden Ereignisse, die zu jeder Transaktion gehören, zu einem Zielzeitpunkt verschoben werden. Die Reihenfolge der Zielzeitpunkte ist genau die Reihenfolge der Transaktionen in dem angestrebten seriellen Log. Falls dieses Verschieben ohne Vertauschung von in Konflikt stehenden Ereignissen möglich ist, nennen wir die Zielzeitpunkte Serialisierungspunkte.

Definition: Sei $l \in L$. Eine Menge von paarweise verschiedenen Zeitpunkten $S_1,...,S_n$ heißt **Menge von Serialisierungspunkten** $\Longleftrightarrow$ für alle i,j, $1 \leq i,j \leq n$ gilt:
$S_i \rightarrow S_j \Rightarrow$
a) falls $W_j \rightarrow R_i$ in l $\Rightarrow$ kein Konflikt zwischen W_j und R_i.
b) falls $R_j \rightarrow W_i$ in l $\Rightarrow$ kein Konflikt zwischen R_j und W_i.
c) falls $W_j \rightarrow W_i$ in l $\Rightarrow$ kein Konflikt zwischen W_j und W_i.

Wenn zusätzlich für alle i, $1 \leq i \leq n$, gilt: $R_i \rightarrow S_i \rightarrow W_i$, dann heißen die S_i **innere** Serialisierungspunkte.

Die Serialisierungspunkte müssen nicht nach ihren Indizes zeitlich geordnet sein.

Die Bedingungen a, b und c besagen, daß immer dann, wenn aufgrund der Lage der Serialisierungspunkte eine Vertauschung von zwei Ereignissen beim Serialisieren erforderlich ist, diese Ereignisse nicht in Konflikt stehen dürfen. Die Vertauschung von zwei Leseereignissen tritt oben nicht als eigener Fall auf, denn Leseereignisse stehen nie miteinander in Konflikt.

Es gibt Logs, die zwar Serialisierungspunkte, aber keine inneren Serialisierungspunkte haben, z.B. l_4. Manche Quellen bezeichnen sogar nur innere Serialisierungspunkte als Serialisierungspunkte. Logs mit inneren Serialisierungspunkten werden auch "schwach 2-Phasen-gesperrt" genannt. (vgl. [Be+79])

Man beachte, daß $S_i \rightarrow S_j$ bei inneren Serialisierungspunkten $R_i \rightarrow W_j$ impliziert. Die Bedingung a ist daher überflüssig.

Der folgende Satz zeigt, daß wir mit Serialisierungspunkten eine andere Charakterisierung der cp-Serialisierbarkeit gewonnen haben:

Satz 13: $l \in$ (S)CPSR $\Longleftrightarrow$ $\exists$ eine Menge von (inneren) Serialisierungspunkten für l.

Beweis: "$\Leftarrow$": Gegeben seien die Serialisierungspunkte S_i. Gesucht ist ein l' aus S, welches cp-äquivalent zu l ist. Wir wählen l' so, daß alle Transaktionen gemäß der Reihenfolge der S_i seriell ausgeführt werden. Nehmen wir nun an, es gäbe zwei Transaktionen T_i, T_j, $i \neq j$, und in l bzw. l' sind zwei zugehörige Ereignisse, die in Konflikt stehen, vertauscht. Wir nehmen oBdA. an, daß $S_i \rightarrow S_j$ (andernfalls vertausche man die Nummern). In l' gilt demnach:

$R_i \rightarrow W_i \rightarrow R_j \rightarrow W_j$

Da in l ein in Konflikt stehendes Paar von Ereignissen vertauscht sein soll, muß in l gelten:

$W_j \rightarrow R_i$ oder $R_j \rightarrow W_i$ oder $W_j \rightarrow W_i$

In allen Fällen widerspricht dies jedoch den Eigenschaften der Serialisierungspunkte.

Sind die S_i zusätzlich innere Serialisierungspunkte, so muß die Reihenfolge von nichtüberlappenden Transaktionen in l und l' gleich sein, d.h. l und l' sind cp-äquivalent und ohne Umordnungen, daher ist l aus SCPSR.

"$\Rightarrow$": Gegeben ist l' $\in$ S; l und l' sind cp-äquivalent. Gesucht sind die S_i. Wir wählen die S_i beliebig gemäß l'. Zu zeigen ist nun: für alle i,j, $1 \leq$ i,j $\leq$ n gilt: $S_i \rightarrow S_j \Rightarrow$

a) $W_j \rightarrow R_i \Rightarrow$ kein Konflikt zwischen W_j und R_i.

b) $R_j \rightarrow W_i \Rightarrow$ kein Konflikt zwischen R_j und W_i.

c) $W_j \rightarrow W_i \Rightarrow$ kein Konflikt zwischen W_j und W_i.

Falls in a, b oder c die Prämisse gilt, so sind in l und l' die beiden Ereignisse in der Reihenfolge vertauscht. Die cp-Äquivalenz liefert dann sofort ihre Konfliktfreiheit.

Nehmen wir zusätzlich an, l und l' seien ohne Umordnungen. Behauptung: Wir können die vorhandenen S_i nach "innen", d.h. zwischen R_i und W_i, verschieben, ohne die zeitliche Reihenfolge der S_i zu verändern. Hierzu werden beim Verschieben alle in der Zielrichtung liegenden S_i, soweit erforderlich, mitverschoben. Wir verschieben zuerst den frühesten Serialisierungspunkt, dann den nächsten usw. (Die Reihenfolge des Verschiebens ist allerdings unerheblich.)

Angenommen S_i und S_j können nicht nach innen verschoben werden, wobei $S_i \rightarrow S_j$ bzw. $T_i \rightarrow T_j$ in l'. Dann muß in l z.B. folgende Situation vorliegen:

$R_j \rightarrow W_j \rightarrow S_i \rightarrow S_j \rightarrow R_i \rightarrow W_i$

S_i und S_j können auch beliebig anders liegen. Entscheidend ist die Lage von R_i und W_j: Falls R_i in l vor W_j läge, könnten S_i und S_j dazwischen geschoben werden und lägen innen. Bei der angenommenen Reihenfolge $W_j \rightarrow R_i$ in l sind T_i und T_j aber in l und l' umgeordnet. Dies widerspricht den Voraussetzungen, die Verschiebungen sind also doch möglich.

Da nach Konstruktion bei allen Verschiebungen die Ordnung der S_i erhalten bleibt, haben wir am Ende immer noch Serialisierungspunkte, die nunmehr innen liegen. q.e.d.

Satz 13 zeigt, daß man die Serialisierungspunkte als gedachte Zeitpunkte ansehen kann, bei denen die Transaktionen *atomar* ausgeführt wurden. Bei inneren Serialisierungspunkten liegt dieser Zeitpunkt innerhalb des Zeitraums vom Start der Transaktion durch den Benutzer und der Rückkehr zum Benutzer.

8.5.4 Der Konfliktgraph eines Logs

Wir hatten bereits oben erwähnt, daß nur dann ein effizienter CC-Mechanismus zu einem Korrektheitsbegriff gefunden werden kann, wenn die zugehörige Menge von Logs leicht erkennbar ist. Bei SR war dies bspw. nicht der Fall. Anders jedoch bei CPSR: Es gibt effiziente Verfahren,

- die zu einem gegebenen l aus CPSR einen cp-äquivalenten seriellen Log konstruieren
- bzw. die zu einem l aus L die Existenz eines cp-äquivalenten seriellen Logs entscheiden, was für den Test, ob l ∈ CPSR, ausreicht.

Alle diese Verfahren arbeiten konzeptionell mit dem sogenannten Konfliktgraphen eines Logs. In diesem Graphen werden Konflikte zwischen Ereignissen von Transaktionen dargestellt.

Definition: Sei l ∈ präfixe(L). Der **Konfliktgraph D(l)** ist folgender gerichtete Graph: Knotenmenge von D(l) ist die Menge der Transaktionen, von denen in l ein Ereignis enthalten ist. Eine Kante von T_i nach T_j ist in D(l) genau dann vorhanden, wenn a oder b oder c gilt:
a) $R_i \rightarrow W_j$ in l und R_i und W_j stehen in Konflikt
b) $W_i \rightarrow R_j$ in l und W_i und R_j stehen in Konflikt
c) $W_i \rightarrow W_j$ in l und W_i und W_j stehen in Konflikt

Die Konfliktgraphen von l_1 und l_4 sehen bspw. wie folgt aus:

Eine Kante von T_i nach T_j in D(l) stellt dar, daß zwei Ereignisse, die zu T_i und T_j gehören und die in Konflikt zueinander stehen, gemäß der Richtung der Kante zeitlich in l geordnet sind. In jedem zu

Abbildung 8.3: Konfliktgraph von l_1 und l_4

l cp-äquivalenten seriellen Log muß daher T_i vor T_j liegen. Man erkennt bereits intuitiv, daß ein Log l nicht cp-serialisierbar sein kann, wenn D(l) einen Zyklus enthält (z.B. D(l_1)), denn die Anforderungen an die Reihenfolge der Transaktionen in der cp-Serialisierung von l sind widersprüchlich. Umgekehrt ist bei zyklusfreiem D(l) eine Halbordnung der Transaktionen gegeben, die beim Serialisieren zu einer linearen Ordnung ergänzt werden kann. Dies werden wir i.f. beweisen.

Lemma 14: Seien l,l' $\in$ präfixe(L) cp-äquivalent. Dann ist D(l) $=$ D(l').

Beweis: Da die Menge der Ereignisse in l und l' gleich ist, ist auch die Knotenmenge von D(l) und D(l') gleich. Eine Kante von T_i nach T_j ist in D(l) genau dann vorhanden, wenn a oder b oder c gemäß der Definition von D(l) gilt. Die beiden Ereignisse, die die Kante verursachen, stehen auch bei l' in Konflikt und müssen wegen der cp-Äquivalenz in l' in der gleichen Reihenfolge stehen. Somit enthält D(l') eine Kante von T_i nach T_j. Da die Argumentation in beiden Richtungen gilt, müssen die Kantenmengen von D(l) und D(l') gleich sein. q.e.d.

Die Umkehrung von Lemma 14 gilt nicht generell; l_1 und l_{11} sind ein Gegenbeispiel. Die Umkehrung gilt aber bei einer zusätzlichen Bedingung:

Lemma 15: Seien l,l' $\in$ präfixe(L), D(l) $=$ D(l'), beide Graphen ohne Zyklen der Länge 2. Dann sind l und l' cp-äquivalent.

Beweis: Wir konstruieren eine Folge von Logs $l_0,\ldots,l_{2n}$ mit $l_0 = l$ und $l_{2n} = l'$ und mit:

a) l und l_i sind cp-äquivalent;

b) präfix(l_i,i) = präfix(l',i)

für alle i, $0 \leq i \leq 2n$. Mit anderen Worten wird $l = l_0$ schrittweise umgebaut zu l'. Zuerst wandert das Ereignis, welches in l' an erster Stelle steht, nach vorne in l, der Rest von l bleibt unverändert: so erhalten wir l_1. Dann verschieben wir in l_1 das gemäß l' zweite Ereignis an Position 2, usw. Im i-1-ten Schritt liegt folgende Situation vor:

$$\begin{array}{lll} l_{i-1}: & e_1.....e_{i-1} & e_i \; .... \; e_j \; .... \; e_{2n} \\ l_i: & e_1.....e_{i-1} & e_j \; e_i \ldots e_{j-1} \; e_{j+1} \ldots e_{2n} \end{array}$$

e_j ist das Ereignis, das in l' an i-ter Stelle steht. Es wandert in l_{i-1} nach vorne, der Rest bleibt unverändert, und wir erhalten l_i. Die obige Bedingung b ist somit nach Konstruktion erfüllt. Zu zeigen bleibt Bedingung a.

Dies beweisen wir durch Induktion über i. Der Fall i = 0 ist trivial. Im Fall i müssen wir nur noch die cp-Äquivalenz von l_i und l_{i-1} zeigen.

l_i und l_{i-1} sind genau dann cp-äquivalent, wenn e_j nicht mit e_i,...,e_{j-1} in Konflikt steht. Nehmen wir das Gegenteil an, also ein k mit $i \leq k < j$ und e_j und e_k stehen in Konflikt. Zunächst können e_j und e_k nicht zur selben Transaktion gehören, denn sonst fände entweder in l oder in l' das Schreibereignis vor dem Leseereignis statt.

Nehmen wir also an, daß e_k zu Transaktion T_a gehört (also $e_k \in \{R_a, W_a\}$) und e_j zu T_b. D(l_{i-1}) enthält somit eine Kante von T_a nach T_b und nach Induktionsvoraussetzung und Lemma 14 auch D(l). e_j und e_k stehen in l' in anderer Reihenfolge, deshalb enthält D(l') eine Kante von T_j nach T_i. Da wir D(l) = D(l') voraussetzen, müssen beide Graphen einen Zyklus zwischen T_i und T_j enthalten. Dies widerspricht aber den Voraussetzungen. q.e.d.

Man kann Logs mit Konfliktgraphen finden, die nur Zyklen größer 2 enthalten. Diese erfüllen dann die Voraussetzungen von Lemma 15. Wir werden das Lemma aber nur für den Fall zyklusfreier Graphen benutzen.

Satz 16: $l \in CPSR \iff$ D(l) zyklusfrei.

Beweis: "$\Rightarrow$": Sei l $\in$ CPSR, also gibt es l' $\in$ S, l,l' cp-äquivalent. Da l' seriell ist, enthält D(l') keinen Zyklus. Nach Lemma 14 ist D(l) = D(l'); somit enthält auch D(l) keinen Zyklus.

"$\Leftarrow$": Sei D(l) zyklusfrei, also eine Halbordnung auf der Menge der Transaktionen. Man wähle eine beliebige Linearisierung dieser Halbordnung und bilde dementsprechend einen seriellen Log l'. Offensichtlich gilt: D(l) = D(l'). Nach Lemma 15 sind dann l und l' cp-äquivalent. q.e.d.

Satz 16 zeigt, daß man bei cp-serialisierbaren Logs die Knoten von D(l) auch als "graphische Serialisierungspunkte" auffassen kann, die Kanten als "später". Die in D(l) gegebene Halbordnung der Zeitpunkte kann beliebig zu einer linearen Ordnung ergänzt werden, man erhält immer eine Menge von Serialisierungspunkten zu l.

Eine sehr wichtige Konsequenz aus Satz 16 ist, daß die cp-Serialisierbarkeit effizient entscheidbar ist, und zwar in einer Zeit, die linear in der Größe von Datenbanken und quadratisch in n ist. Dies ermöglicht die Konstruktion von effizienten CC-Mechanismen. Tatsächlich lassen alle universellen CC-Mechanismen, die für die Praxis relevant sind, nur cp-serialisierbare Logs zu, meist sogar nur strikt cp-serialisierbare. Die Logs, die im Rahmen unseres Modells dem 2-Phasen-Sperren entsprechen, sind eine echte Teilmenge von SCPSR. Es gibt auch einige Mechanismen, die Umordnungen zulassen. Meist arbeiten sie in irgendeiner Weise mit Zeitstempeln (z.B. in SDD-1). Die Menge der zugelassenen Logs ist eine Teilmenge von CPSR, aber nicht von SCPSR.

Verallgemeinert man den Begriff cp-Serialisierbarkeit auf Präfixe von Logs (wie in Abschnitt 4.2.1 angegeben), so gilt Satz 16 auch für solche Präfixe. Der Konfliktgraph eines Präfixes eines Logs ist ein Teilgraph des Konfliktgraphs des gesamten Logs. Wenn der gesamte Graph keinen Zyklus enthält, kann auch der Teilgraph keinen Zyklus enthalten. Das gleiche gilt, wenn man in einem cp-serialisierbaren Log beliebige Transaktionen entfernt. In diesem Fall entsteht ein Teillog des ursprünglichen Logs. Wir erhalten so die folgenden wichtigen Eigenschaften der cp-serialisierbaren Logs:

Korollar 17: Sei l $\in$ CPSR, l' ein Präfix oder ein Teillog von l. Dann ist l' cp-serialisierbar (also insbesondere Sicht-serialisierbar).

In cp-serialisierbaren Logs sind also keine temporären Anomalien möglich!

Korollar 17 ist in gewisser Weise umkehrbar: Wenn alle Teillogs eines Logs Sicht-serialisierbar sind, dann ist der Log cp-serialisierbar. Konsequenz ist, das alle CC-Mechanismen, die die Sicht-Serialisierbarkeit der von ihnen zugelassenen Logs auch bei Rollback ohne Neustart garantieren wollen (vgl. Abschnitt 8.4), die cp-Serialisierbarkeit garantieren müssen. Dies zeigt erneut, daß für praktische Anwendungen die cp-Serialisierbarkeit das minimale Korrektheitskriterium sein muß.

Zum Beweis der Umkehrung von Korollar 17 benötigen wir das folgende Lemma:

Lemma 18: Sei l $\in$ L. Angenommen, D(l) enthält eine Kante von T_1 nach T_2, x ist das Objekt, welches diese Kante verursacht, und keine Transaktion außer T_1 und T_2 schreibt x. Sei l' ein zu l Sicht- und fs-äquivalenter, serieller Log. Dann liegt in l' T_1 vor T_2.

Beweis: Der Konflikt von T_1 und T_2 kann durch 3 verschiedene Kombinationen von Lese- bzw. Schreibereignissen verursacht werden: Wir betrachten die Fälle getrennt.

1. W_1 und W_2: Nach Voraussetzung liegt in l W_1 vor W_2. Läge in l' W_2 vor W_1, so würde x in beiden Logs von verschiedenen Transaktionen zum letzten Mal beschrieben, da außer T_1 und T_2 kein anderer Log x schreibt. Sie wären dann nicht fs-äquivalent.
2. W_1 und R_2: Nach Voraussetzung liest T_2 x von T_1. Läge in l' T_2 vor T_1, würde in l' T_2 den Anfangswert von x lesen. Dann wären l und l' nicht Sicht-äquivalent.
3. R_1 und W_2: führt analog zu Fall 2 zu verschiedenen Sichten von T_2 in l und l'.

In allen 3 Fällen führt die Annahme, daß in l' T_2 vor T_1 liegt zu einem Widerspruch. q.e.d.

Satz 19: Sei l ∈ ISR. Wenn alle Teillogs aus l Sicht-serialisierbar sind, dann ist l ∈ CPSR.

Beweis: Wir zeigen, daß D(l) keinen Zyklus haben kann.

Nehmen wir das Gegenteil an, also einen Zyklus T_1,T_2,...,T_k,T_1 in D(l). k soll minimal sein, d.h. wenn es einen kürzeren Zyklus in D(l) gibt, wählen wir diesen. Sei l' der Teillog aus l zu T_1,...T_k. Nach Voraussetzung ist l' Sicht-serialisierbar.

Falls k = 2, können wir Lemma 18 anwenden, denn in l' gibt es keine Transaktionen außer T_1 und T_2. Die beiden Kanten des Zyklus implizieren, daß in einem Log, der Sicht- und fs-äquivalent zu l' ist, T_1 vor T_2 und gleichzeitig T_2 vor T_1 liegen müßte. Dies ist unmöglich, l' ist also nicht Sicht-serialisierbar.

Angenommen, k = 3. Sei x ein Objekt, welches die Kante von T_1 nach T_2 verursacht.

Behauptung: Dann schreibt T_3 x nicht. Nehmen wir das Gegenteil an. Wegen der Kante von T_2 nach T_3 in D(l') müssen alle Ereignisse von T_2, die x betreffen, *vor* W_3 liegen. Andernfalls gäbe es eine Kante von T_3 nach T_2 und somit einen Zyklus der Länge 2. Wegen der Kante von T_3 nach T_1 müssen analog alle Ereignisse von T_1, die x betreffen, *nach* W_3 liegen. Dann könnte x aber nicht die Kante von T_1 nach T_2 verursachen.

Da T_3 x nicht schreibt, ist Lemma 18 auf l' anwendbar. Die Kanten des Zyklus liefern insgesamt eine unerfüllbare Bedingung, die ein serieller Log, der Sicht- und fs-äquivalent zu l' ist, erfüllen müßte. l' ist also nicht Sicht-serialisierbar.

Angenommen, k > 3. Sei x ein Objekt, das eine Kante in dem Zyklus verursacht, z.B. von T_1 nach T_2. Sei T_i eine andere Transaktion. In D(l') ist T_i mit T_{i-1} und T_{i+1} verbunden, beide Indizes modulo k, verbunden, ansonsten mit keiner anderen Transaktion des Zyklus, sonst gäbe es einen kürzeren Zyklus. Da k > 3, kann T_i entweder mit T_1 *oder* T_2 verbunden sein, nicht aber mit beiden. Würde nun T_i x schreiben, so existierten Kanten zwischen T_i und sowohl T_1 wie T_2, Widerspruch. Somit ist Lemma 18 auf jedes Paar von Transaktionen, die im Zyklus durch eine Kante verbunden sind, anwendbar. Hierdurch folgt wieder eine unerfüllbare Bedingung, die ein serieller, zu l' Sicht-

und fs-äquivalenter Log erfüllen müßte. q.e.d.

Die Voraussetzungen von Satz 19 können nicht abgeschwächt werden. Insbesondere ist ein Log, dessen Teillogs alle (strikt) serialisierbar sind, deswegen nicht cp-serialisierbar. Ein Gegenbeispiel ist:

```
l12:
T1   r(x,y)------------------------w(y)
T2          r(y,z)---w(x,z)
T3                          r(y,z)------w()
```

8.5.5 Serialisierbare Logs ohne Verluste

Wir hatten bereits mehrfach festgestellt, daß die fs-Serialisierbarkeit i.a. kein ausreichendes Korrektheitskriterium ist, weil temporäre Anomalien auftreten können. Diese standen meist wiederum im Zusammenhang mit toten Transaktionen, also teilweisen oder ganzen Verlusten der Wirkung von Transaktionen. In diesem Abschnitt wollen wir die Frage untersuchen, ob bei Abwesenheit von Verlusten die fs-Serialisierbarkeit nicht doch ausreichend ist. Im Kontext unserer Modelle ist also nach Bedingungen zu suchen, unter denen SR und CPSR bzw. SSR und SCPSR identisch sind.

Zunächst wollen wir klären, was "Abwesenheit von Verlusten" überhaupt bedeutet. Wir hatten schon früher zwischen seriellen (von einem CC-Mechanismus nicht vermeidbaren) Verlusten und Verlusten durch Parallelität unterschieden. In Lemma 8 wurde gezeigt, daß bei Abwesenheit serieller Verluste in SR keine toten Aktionen auftreten können, also überhaupt keine Verluste und damit auch keine Verluste durch Parallelität. Jedoch impliziert selbst unter dieser Voraussetzung die Serialisierbarkeit nicht die cp-Serialisierbarkeit. l_{12} ist ein Gegenbeispiel.

Wenn zusätzlich Lesetransaktionen ausgeschlossen sind, gilt die gesuchte Implikation. Die Sicht-Serialisierbarkeit benötigt den zusätzlichen Ausschluß von Lesetransaktionen nicht, um die cp-Serialisierbarkeit zu implizieren. Die Beweise für diese Zusammenhänge sind relativ aufwendig und können hier aus Platzgründen nicht aufgeführt werden. Sie

können in [Ke86] und teilweise in [Be+79] und [Pa79] nachgelesen werden. Die Gleichheit der Varianten der Serialisierbarkeit wird in allen Fällen dadurch nachgewiesen, daß die zugehörigen Äquivalenzbegriffe einander implizieren. Wir zitieren hier nur die wichtigsten Zwischen- und Endresultate:

Satz 20: Sei $\text{writeset}_i \subseteq \text{readset}_i$ für alle i, $1 \leq i \leq n$, l,l' $\in$ L. Dann gilt:

1. l, l' Sicht- und fs-äquivalent $\Rightarrow$ l, l' cp-äquivalent.
2. SISR = SCPSR, ISR = CPSR.
   Falls zusätzlich $\text{writeset}_i \neq \emptyset$ für alle i, so gilt:
3. l, l' fs-äquivalent $\Rightarrow$ l, l' cp-äquivalent.
4. SSR = SISR = SCPSR, SR = ISR = CPSR.

Zusammenfassend kann gesagt werden, daß die Serialisierbarkeit in gewissen Fällen, die keineswegs unrealistisch sind, mit der cp-Serialisierbarkeit übereinstimmt und dann ein ausreichendes Korrektheitskriterium ist.

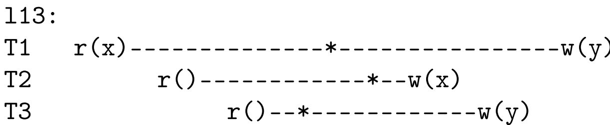
8.5.6 2-Phasen-Sperren

Das **2-Phasen-Protokoll** (2PL-Protokoll) ist das gängste CC-Verfahren (s. Lehrmodul 3). Es besagt, daß eine Transaktion in einer Anforderungsphase Sperren anfordern und anschließend in einer Freigabephase wieder freigeben kann; beide Phasen liegen strikt hintereinander, die Zeit dazwischen kann man als Verarbeitungsphase bezeichnen.

Wenn alle Transaktionen das 2PL-Protokoll befolgen, gilt für jeden entstehenden Log folgendes: Jedes von einer Transaktion benutzte Objekt ist durchgehend von seiner ersten bis zur letzten Benutzung gesperrt, es können daher in dieser Zeit keine Zugriffe anderer Transaktionen eintreten, die mit der Sperre in Konflikt stehen. Hieraus folgt, daß alle Ereignisse einer Transaktion in die Verarbeitungsphase verschoben werden können, ohne den Konfliktgraphen des Logs zu verändern.

Somit ist jeder beliebige Zeitpunkt innerhalb der Verarbeitungsphase ein zulässiger Serialisierungspunkt für diese Transaktion, sogar ein innerer. Somit ist der Log strikt cp-serialisierbar.

Umgekehrt kann nicht jeder strikt cp-serialisierbare Log durch 2-Phasen-gesperrte Transaktionen erzeugt werden. Ein Gegenbeispiel ist:



Der einzige Log, der zu l_{13} cp-äquivalent ist, ist $T_3T_1T_2$. Die Serialisierungspunkte der Transaktionen können daher nur in dieser Reihenfolge liegen. Sie sind als * eingezeichnet. l_{13} liegt also in SCPSR. Nehmen wir an, die Transaktionen seien alle 2- Phasen-gesperrt. Ihre Verarbeitungsphasen müssen in der gleichen Reihenfolge liegen wie die Serialisierungspunkte. Dies ist aber nicht möglich: Die Zeiträume, in denen y von T_1 und T_3 gesperrt sein müßte, überlappen nämlich. Anders gesehen müßte, damit W_2 und W_3 wie angegeben stattfinden können, T_1 x schon freigegeben, y aber noch nicht gesperrt haben.

Das 2PL-Protokoll schränkt somit die Parallelität etwas mehr ein, als es im Sinne der strikten cp-Serialisierbarkeit notwendig wäre. Man beachte, daß in diesem Beispiel Objekte blind überschrieben (bzw. gelöscht) werden; ohne dieses Verhalten könnte man kein Beispiel konstruieren. Die Parallelitätsreduktion betrifft somit eher seltene Fälle und kann in der Praxis vernachlässigt werden.

Lehrmodul 9:

Kooperation und Parallelität in SEU

Zusammenfassung dieses Lehrmoduls

Software wird typischerweise in Teams entwickelt. Eine Software-Entwicklungsumgebung (SEU) muß daher Teamarbeit unterstützen. Dies heißt zunächst, daß die SEU es überhaupt mehreren Entwicklern erlaubt, an den gleichen Dokumenten zu arbeiten. Abhängig von der Art der Dokumente und der Entwicklungsmethode sind verschiedene Formen der Kooperation denkbar; hierzu führen wir den Begriff Kooperationsmodell ein und skizzieren mehrere Kooperationsmodelle.

Kooperationsmodelle kann man als Anforderungen an eine SEU auffassen; im zweiten Teil des Lehrmoduls untersuchen wir, wie diese Anforderungen unter Ausnutzung von Diensten eines Objektmanagementsystems, insb. Zugriffskontrollen, Transaktionen, Versionen und Notifizierung, realisiert werden können.

Vorausgesetzte Lehrmodule:

obligatorisch: – Transaktionen und die Integrität von Datenbanken
empfohlen: – Software-Entwicklungsumgebungen

Stoffumfang in Vorlesungsdoppelstunden: 1.4

 ©2003 Udo Kelter

9.1 Einleitung und Übersicht

Software wird, abgesehen von sehr kleinen Projekten, in Teams entwickelt. Eine Software-Entwicklungsumgebung (SEU) muß daher von mehreren Benutzern gleichzeitig benutzt werden können. "SEU" ist im letzten Satz im Sinne einer SEU-Installation zu verstehen, d.h. die Entwickler arbeiten auf gleichen oder zusammenhängenden Dokumenten und die SEU verwaltet diese Dokumente. Die SEU im Sinne von Software (also Compiler, Editoren usw.) verwaltet tatsächlich keine Dokumente; für die Dokumentverwaltung wird stattdessen ein unterliegendes Datenverwaltungssystem eingesetzt, beispielsweise

- ein Dateisystem, oft in Verbindung mit einem Konfigurationsmanagementsystem, oder
- ein Objektmanagementsystem (**OMS**), also ein dediziertes Datenverwaltungssystem für SEU

Manche Datenverwaltungssysteme unterstützen den parallelen Zugriff von Anwendungen bzw. Benutzern durch integrierte Transaktionskonzepte. Derartige Transaktionskonzepte sind allerdings nur Mechanismen, die bestimmte technische Ziele erreichen: typischerweise wird Anwendungen der Zugriff auf Daten zeitweise verwehrt, wenn durch parallele Zugriffe die Konsistenz der Daten (vermutlich) beschädigt werden würde. Durch derartige automatisch arbeitende Mechanismen wird jedoch aus Einbenutzersystemen, die zufällig auf den gleichen Daten arbeiten, keineswegs automatisch ein brauchbares Mehrbenutzersystem.

Vielmehr bildet bei parallel benutzbaren Systemen das Thema Parallelität einen völlig eigenständigen Problemkomplex. Dieser sollte idealerweise unbeeinflußt von Implementierungsrestriktionen analysiert werden. Erst nach Feststellung der Anforderungen sollten diese unter möglichst weitgehender Ausnutzung verfügbarer Basismechanismen realisiert werden. Tatsächlich muß man natürlich darauf achten, daß die Anforderungen auch implementierbar sind, und so haben die verfügbaren und den Entwicklern vertrauten Implementierungstechnologien durchaus Einfluß auf die Festlegung der Anforderungen. Relevante Technologien bzw. Funktionsgruppen der Datenverwaltungssysteme – und speziell von OMS, von denen wir i.f. ausgehen werden – sind:

- Transaktionsmechanismen
- Notifizierung (entweder als selbständige Systeme oder innerhalb des OMS)
- Zugriffskontrollen
- Interprozeßkommunikationsmechanismen, z.B. RPC

In diesem Lehrmodul werden wir zuerst die Anforderungsseite betrachten und hier einige Alternativen aufzeigen. Anschließend diskutieren wir die oben erwähnten Basismechanismen und diskutieren ihre Nützlichkeit im Hinblick auf die unterschiedlichen denkbaren Anforderungen.

9.2 Kooperationsmodelle

Wir betrachten eine SEU hier in erster Linie als interaktives System; es gibt zwar auch viele nichtinteraktive Werkzeuge (Compiler, Prüfwerkzeuge usw.), diese reduzieren aber die Problematik nicht. Die folgenden Überlegungen sind auch auf viele andere Umgebungen übertragbar, in denen irgendwelche Systeme entwickelt werden.

Wenn wir sagen, ein interaktives System sei parallel benutzbar, dann kann es von einer Gruppe von Personen gleichzeitig benutzt werden[28]. Eine zentrale Frage ist nun, ob und wie diese Personen miteinander kooperieren. Eine Beschreibung bzw. Festlegung der Kooperation nennen wir ein **Kooperationsmodell**. Wir werden mehrere Kooperationsmodelle verbal beschreiben, formale Notationen sind in diesem Bereich nicht verfügbar und vermutlich auch nicht hilfreich.

Ein häufiges und sehr einfaches Kooperationsmodell besteht darin, daß sich die Personen überhaupt nicht kennen ("anonyme Benutzer") und daher auch nicht kooperieren; wenn überhaupt, dann *konkurrieren* die Personen um die gleichen Ressourcen und müssen, eben weil sie nicht kooperieren, voreinander geschützt werden. Das Kooperationsmodell

[28] Den Fall, daß der gleiche Benutzer mehrere Anwendungen parallel startet, betrachten wir hier nicht.

"nichtkooperierende Benutzer" wird bei klassischen Transaktionskonzepten (mit ACID-Eigenschaften[29]) unterstellt.

Bei allen anderen Kooperationsmodellen kennen die Agierenden einander mehr oder minder gut ("*group awareness*") und verfolgen i.d.R. bestimmte *gemeinsame* Ziele. Man redet hier auch von rechnergestützter Gruppenarbeit (*computer-supported cooperative work*; *CSCW*). Die Rechnerunterstützung kann sehr unterschiedliche Aspekte betreffen: elementare Voraussetzung sind zunächst Kommunikationsdienste, über die asynchron oder synchron Daten zwischen den Beteiligten ausgetauscht werden können. Darüber hinaus sind vielfältige Funktionen denkbar, die die Koordination der Gruppe und die Arbeitsprozesse der Gruppe unterstützen: elektronische Post, verteilte Zeit- und Aufgabenplanung, verteilte Editoren, Geschäftsprozeßunterstützung usw.; diese Dienste hängen stark von der Problemstellung ab. Der Begriff CSCW ist daher sehr vielschichtig und wird sehr unterschiedlich interpretiert. Für eine ausführliche Behandlung sei auf die Monographie [BoS95] verwiesen.

9.2.1 Kooperation und Koordination bei der Softwareentwicklung

Im konkreten Kontext der Softwareentwicklung ist es naheliegenderweise das Ziel der Arbeitsgruppe, ein Softwaresystem fertigzustellen. Der Gruppenarbeitsprozeß umfaßt insb. die folgenden Bereiche:

1. Diskussionen über die Gestaltung des Systems
2. Austausch oder gemeinsame Bearbeitung von Dokumenten
3. Terminkoordination

Der erste Bereich wird üblicherweise durch SEU überhaupt nicht unterstützt; diskutiert wird auf Sitzungen oder allenfalls (bei verteilten Gruppen) mit Unterstützung durch elektronische Post.

Für den dritten Bereich sind diverse Projektmanagementwerkzeuge verfügbar, in der Praxis werden aber überwiegend informelle Metho-

[29] Die Abkürzung *ACID* steht für *atomicity, consistency preservation, isolation, durability*. Lehrmodul 1 enthält eine Einführung in dieses Thema.

den eingesetzt. Die Koordination bei der Bearbeitung von Dokumenten wird i.d.R. durch SEU nicht unterstützt.

Wir betrachten i.f. nur den zweiten Bereich näher, obwohl alle Bereiche wichtig sind und ganzheitlich betrachtet werden sollten.

9.2.2 Koordination der Bearbeitung von Dokumenten

Ein völlig regelloser, unkoordinierter Zugriff auf Dokumente ist (unabhängig von ihrer Speicherungsform) fast immer schon aus rein sachlogischen Aspekten nicht möglich. Wenn zwei Autoren an einem Buch schreiben und an verschiedenen Stellen des Buchs unkontrolliert herumändern, paßt hinterher an dem Buch nichts zusammen. Gleiches gilt für Programme, Spezifikationen und beliebige andere Dokumenttypen. Ob und wie ein Dokument in Teile zerlegt werden kann, die gefahrlos parallel bearbeitet werden können, ist zuallererst eine Frage der Sachlogik des Dokuments und eventuell der Entwicklungsmethode, die zum Dokumenttyp gehört, und nicht primär bestimmt durch die Speicherungsform in ggf. mehreren Dateien oder Objekten.

Eine wichtige Rolle spielen hier die logischen Konsistenzkriterien für einen Dokumenttyp. Bei der Bearbeitung von Dokumenten entstehen inkonsistente Zwischenzustände. Inkonsistente Dokumente können i.d.R. nicht sinnvoll zwischen verschiedenen Entwicklern ausgetauscht werden.

Aus der sachlogischen Struktur der Dokumente ergeben sich maximal mögliche Formen der parallelen unkoordinierten Arbeit. Beispielsweise können bei den üblichen Modularisierungsmethoden für Programme die Rümpfe verschiedener Module, nachdem die Schnittstellen festliegen, völlig unkoordiniert entwickelt werden. Es gibt aber auch viele Fälle, bei denen eine unkoordinierte Parallelität zu Fehlern führt und trotzdem parallel gearbeitet werden muß. In solchen Fällen müssen sich die Agierenden auf irgendeinem Weg koordinieren. Diese Koordination erfolgt sehr oft verbal (Entwickler A ruft B quer durchs Zimmer zu: “Du, ich muß in types.h was ändern, kannst Du mal für 10 Minuten alles liegenlassen, ich übersetze dann auch das ganze System neu...”). Diese Koordination kann durch die SEU in Form verschiedener

Dienstleistungen unterstützt werden, beispielsweise durch:

- allgemeine Auskünfte, wer gerade im System eingeloggt ist und wofür diese Personen zuständig sind bzw. welche Rolle sie spielen
- Auskünfte, wer ein bestimmtes Dokument gerade bearbeitet, wie lange die Bearbeitung voraussichtlich dauern wird u.ä.
- Propagation von Änderungen, die ein Entwickler veranlaßt, auf den Arbeitsplatz anderer Entwickler
- Sperrmechanismen, die den parallelen Zugriff zu Dokumenten verhindern
- allgemeine Kommunikationsfunktionen (elektronische Post, Konferenzsysteme u.ä.)

Welche Funktionen sinnvoll sind, hängt stark von der Größenordnung der Projekte, der Art der Dokumente und von den subjektiven Präferenzen der Entwickler ab. Das Kooperationsmodell, in dem die oben aufgelisteten Details festgelegt werden, muß also individuell für jeden Anwendungsfall entwickelt werden. Dies schließt natürlich nicht aus, daß es vorkonfektionierte Kooperationsmodelle gibt, die mit geringfügigen Anpassungen in einem konkreten Anwendungsfall übernommen werden.

9.2.3 Das "Bibliotheksmodell"

In diesem und im nächsten Abschnitt beschreiben wir die Grundzüge zweier typischer Kooperationsmodelle für Dokumente in einer SEU. Konkrete Werkzeuge und SEU würden diese Grundzüge noch mit vielen Details anreichern, man kann aber bereits auf dieser ganz abstrakten Ebene Vor- und Nachteile erkennen.

Beim "Bibliotheksmodell" wird ein Dokument wie in ein Buch in einer Bibliothek behandelt. Das Dokument lagert normalerweise in der Bibliothek; dort kann es *ein* Benutzer entnehmen und dann exklusiv damit arbeiten; später stellt er es wieder in die Bibliothek zurück. Während der Ausleihe können andere Benutzer nicht auf das Dokument

zugreifen, allenfalls auf eine Kopie, können aber herausfinden, wer im Moment das Original hat und ggf. diese Person bitten, das Dokument zurückzustellen.

Das Bibliotheksmodell hat den Nachteil, daß Änderungen verschiedener Benutzer an einem Dokument sequentialisiert werden müssen; dies kostet Zeit. Dieser Nachteil kann entschärft werden, wenn man ein Dokument in eine Grobstruktur ("Inhaltsverzeichnis") und möglichst kleine Dokumentteile ("Abschnitte") gliedern kann und wenn Änderungen primär innerhalb der Dokumentteile stattfinden.

Konfigurationsmanagementsysteme realisieren i.d.R. das Bibliotheksmodell entweder wie vorstehend beschrieben oder aber in einer Variante, die mehrfaches Ausleihen eines Dokuments – mit realen Büchern schlecht vorstellbar – erlaubt (z.B. CVS, s. [KM, CVS]).

9.2.4 Das "Wandtafel-Modell"

Beim "Wandtafel-Modell" steht das Dokument – gedanklich – auf einer großen Tafel, und verschiedene Benutzer ändern an verschiedenen Stellen der Tafel. Jeder sieht sofort alle Änderungen aller Benutzer. Das Dokument wird parallel bearbeitet, es ist kein Warten auf seine Freigabe nötig.

Ein Problem beim Wandtafel-Modell ist, daß man nicht ungestört an einem Dokument arbeiten kann. Da Änderungen anderer Benutzer sofort sichtbar sind, erzeugen sie Unruhe, lenken ab und führen zu inkonsistenten Zwischenzuständen. Quellprogramme in üblichen Programmiersprachen sind in dieser Hinsicht sehr empfindlich: wenn durch ein paralleles Bearbeiten an verschiedenen Teilen des System auch nur geringfügige Inkonsistenzen erzeugt werden, ist das System nicht mehr übersetzbar und die Entwickler können praktisch nicht mehr arbeiten.

Verallgemeinert kann man sagen, daß das Wandtafel-Modell ungeeignet ist, wenn Dokumente komplexe Konsistenzbedingungen haben und Arbeitsprozesse "empfindlich" auf Inkonsistenzen reagieren, wenn also die Konsistenzbedingungen ständig erfüllt sein müssen. In solchen Fällen muß ein Entwickler einen Arbeitsbereich haben, in dem er vor äußeren Einflüssen geschützt ist und ungestört einen Entwicklungsschritt

zu Ende führen kann; es muß also letztlich anhand des Bibliotheksmodells verfahren werden. Wenn dennoch parallel am gleichen Dokument gearbeitet werden muß, müssen die einzelnen Ergebnisse i.d.R. von Hand in eine Hauptlinie eingemischt werden; dies ist zwar aufwendig, aber immer noch praktikabler als ständige Störungen.

Das Wandtafel-Modell ist auch für größere Gruppen nicht mehr geeignet. Ab ca. 10 - 20 aktiven Teilnehmern an einem Gruppenprozeß entsteht zuviel Unruhe. Hier ist noch anzumerken, daß Änderungen in Dokumenten meist begleitet werden müssen von Erklärungen, warum diese Änderungen erfolgen und in welche generellen Pläne sie einzuordnen sind. Die hierzu erforderliche Kommunikation findet i.d.R. verbal statt. Bei größeren Gruppen ist diese Kommunikation nicht mehr möglich.

Neben den beiden vorgestellten Kooperationsmodellen sind vielfältige Zwischenformen und Varianten denkbar.

Die beiden Kooperationsmodelle führen zu erheblichen Unterschieden in der Bedienschnittstelle von Editoren und den ausnutzbaren Diensten eines OMS. Beim Bibliotheksmodell können die Editoren übliche Ein-Benutzer-Systeme sein, die Editoren dürfen gerade kein Gruppenbewußtsein produzieren. Für die Realisierung des Ausleiheffekts können Zugriffsrechte oder Sperren (allerdings keine klassischen, s.u.) verwendet werden. Beim Wandtafel-Modell sind hingegen Notifizierung und Interprozeßkommunikation relevant.

9.2.5 Adaptierbarkeit der SEU

Die Vielfalt möglicher Kooperationsmodelle führt zu der Frage, ob bzw. inwieweit ein bestimmtes Kooperationsmodell in einer SEU hart verdrahtet sein soll. Die Forderung liegt nahe, daß die SEU möglichst flexibel an die Benutzerwünsche anpaßbar sein soll, m.a.W. daß möglichst viele Aspekte der Kooperation einstellbar sind. Letzteres führt dazu, daß eine SEU i.w. nur Mechanismen zur Kooperationsunterstützung anbieten würde und der Einsatz dieser Mechanismen durch entsprechende Einstellungen (bei der Installation und Konfiguration der SEU oder durch Ressource-Dateien oder wie auch immer) gesteuert

werden würde. Unterschiede in den Kooperationsmodellen können allerdings leicht zu erheblichen Unterschieden in der softwaretechnischen Realisierung führen; in solchen Fällen ist es fraglich, ob die Einstellbarkeit des Kooperationsmodells wirklich praktikabel ist. Es muß ein sinnvoller Kompromiß zwischen Aufwand und Ausmaß der Einstellbarkeit gefunden werden. Anders gesagt wird eine SEU fast immer bestimmte Aspekte der Kooperation hart verdrahten; hierüber sollte ein Entwickler von Werkzeugen beim Entwurf bewußt entscheiden, und die Entscheidungen sollten dokumentiert werden. Analog gilt dies bei der Bewertung und Auswahl einer SEU.

9.3 Zur Realisierung von Kooperationsmodellen nutzbare OMS-Dienste

Wir wollen in diesem Abschnitt analysieren, wie Dienste eines OMS bei der Realisierung von Kooperationsmodellen ausgenutzt werden können. Relevant sind vor allem Transaktionen (genauer gesagt Concurrency Control), Zugriffskontrollen und Notifikation. Wir beginnen mit einer Übersicht und Kurzbeschreibung dieser OMS-Dienste.

9.3.1 Zugriffskontrollen

Die Mitglieder einer Entwicklergruppe haben i.d.R. unterschiedliche Zugriffsrechte. Diese Rechte können sehr gut durch die diskretionären Zugriffskontrollen eines OMS überwacht werden.

Bei diskretionären Zugriffskontrollen hat jedes Objekt (bzw. allgemeiner jede Einheit der Rechteverwaltung) wenigstens einen Besitzer, der festlegen kann, wie andere Subjekte (einzelne Nutzer oder Gruppen) mit diesem Objekt arbeiten dürfen. Die Rechtefestlegungen werden in einer **Zugriffskontrolliste** an dem Objekt verwaltet.

Um die Rechte in Arbeitsgruppen, die mit komplexen Dokumenten arbeiten, effizient verwalten zu können, sollten die Zugriffskontrollen eines OMS folgende Leistungsmerkmale aufweisen: erstens sollten zusammengesetzte Objekte Einheit der Rechteverwaltung sein, zweitens sollten für beliebig viele Subjekte an einem Objekt Rechte vergebbar

sein[30], und die Gruppenstruktur sollte bei der Administration bzw. Auswertung der Rechte ausnutzbar sein. Ein Beispiel für derartige gruppenorientierte Zugriffskontrollen findet sich in [DAC].

9.3.2 Transaktionen

Transaktionen adressieren drei Problemkomplexe[31]:

- automatisierte Konsistenzprüfungen
- den parallelen Zugriff mehrerer Nutzer auf die gleichen Daten durch Concurrency-Control- (CC-) Mechanismen, meist Sperrverfahren
- die Reparatur der Datenbank nach Störfällen durch Recovery-Mechanismen

Diese Problemkomplexe wirken auf den ersten Blick unzusammenhängend; die Verbindung liegt darin, daß es in allen drei Bereichen darum geht, die Konsistenz der Daten zu schützen. Gefährdet wird die Konsistenz (a) durch fehlerhafte Programme oder Dateneingaben, (b) durch die verzahnte Ausführung von Applikationen und (c) durch Beschädigungen infolge von Störfällen wie z.B. Rechnerabstürzen. Die gemeinsame Behandlung all dieser Gefährdungen der Konsistenz ist technisch durchaus sinnvoll. Leider wird dadurch die Diskussion des Problemkomplexes Parallelität bzw. Kooperation zusätzlich aufgebläht.

Der Effekt konventioneller CC-Verfahren besteht darin, einer Applikation eine exklusiv verfügbare Datenbank zu simulieren. Kooperation zwischen verschiedenen Anwendungen ist daher nicht intendiert und nicht möglich, d.h. es wird hier im Prinzip das Kooperationsmodell "anonyme Benutzer" unterstellt.

Sperrverfahren wirken im Prinzip so, daß dem Inhaber einer Sperre temporäre Zugriffsrechte erteilt und allen anderen Anwendungen temporär entzogen werden. Sieht man von vielen Begleitumständen

[30] In normalen UNIX-Systemen kann man nur für den Besitzer, eine Gruppe und den Rest der Benutzer Rechte festlegen; dies ist zu restriktiv.

[31] Eine ausführlichere Einführung in das Thema Transaktionen findet sich in Lehrmodul 1.

ab, so liegt hier eine Ähnlichkeit zu diskretionären Zugriffskontrollen vor. Eine detaillierte Analyse in den folgenden Abschnitten wird untersuchen, unter welchen Begleitumständen welche Dienste sinnvoll einsetzbar sind.

9.3.3 Notifikation

Die grundlegende Leistung von Notifikationsmechanismen liegt darin, eine Applikation über bestimmte interessierende Ereignisse, namentlich Änderungen an Daten, zu informieren. Diese Ereignisse sind normalerweise von parallel laufenden Applikationen verursacht. Notifikationsmechanismen sind daher zur Realisierung kooperativer Funktionen in Werkzeugen sinnvoll oder genereller gesagt zur Realisierung von Kooperationsmodellen ähnlich dem Wandtafel-Modell. Voraussetzung ist allerdings, daß die Notifikationsmechanismen verteilt arbeiten[32].

Auf ein bestimmtes Datengranulat gleichzeitig Notifikation und Sperrungen anzuwenden, ist nicht sinnvoll, da die Intentionen gerade entgegengesetzt sind.

Im folgenden betrachten wir vor allem Transaktionen detaillierter. In [PlK97] diskutieren wir Anforderungen an und Gestaltungsvarianten von Notifikationsmechanismen im Detail.

9.4 Transaktionskonzepte für kooperative Umgebungen

In diesem Abschnitt analysieren wir Anforderungen an Transaktionskonzepte in Datenverwaltungssystemen, die als Basis von SEU (oder anderer kooperativer Entwicklungsumgebungen) dienen sollen. Es wird sich zeigen, daß konventionelle Transaktionskonzepte hier in vieler Hinsicht ungeeignet sind. Wir beginnen daher mit einer Zusammenfassung wesentlicher Merkmale konventioneller Transaktionskonzepte.

[32] Diese Bedingung wird von Notifikationsmechanismen innerhalb von GUI-Frameworks nicht erfüllt.

9.4.1 Konventionelle Transaktionskonzepte

In konventionellen Anwendungen wird der Problemkomplex der parallelen Benutzung eines Systems durch mehrere Benutzer primär durch Transaktionen gelöst. Die konventionellen Transaktionskonzepte gehen von folgenden Annahmen und Randbedingungen aus:

1. Die Anwendungsprogramme unterstützen Kooperation i.a. nicht[33], d.h. das Kooperationsmodell “anonyme Benutzer” wird unterstellt.
2. Transaktionen sind unabhängig voneinander, sie sind anonym und bilden eine “flache” Menge.
3. Transaktionen sind i.d.R. nicht interaktiv, das System kann sie also bei Bedarf sogar neu starten, wenn z.B. wegen eines Deadlocks eine Transaktion zurückgesetzt werden muß.
4. Eine zentrale Annahme ist, daß nur komplett ausgeführte Transaktionen einen konsistenzerhaltenden Zustandsübergang der Datenbank realisieren. Zwischenzustände, die durch partiell ausgeführte Transaktionen entstehen, werden generell als inkonsistent angesehen, auch wenn sie es im Einzelfall aus Anwendersicht nicht wirklich sind.
5. Die meisten Transaktionen sind reine Lesetransaktionen (Größenordnung des Anteils: 90 %).
6. Bei den schreibenden Transaktionen geht man von Buchungen, Umbuchungen, einfachen Dateneingaben und ähnlichem als den typischen konsistenzerhaltenden Zustandsübergängen aus. Solche Transaktionen sind “kurz”, d.h. die Laufzeit liegt im Sekundenbereich, nur wenige Objekte werden benutzt.
7. Technisch sind Transaktionen so gestaltet, daß während der Dauer der Transaktion ununterbrochen eine Kommunikationsverbindung zwischen Applikationsprozeß und Datenbankserver existieren muß.

[33] Workflow-Management-Systeme realisieren Kooperation, aber nicht mit Hilfe von Transaktionen, sondern unter Benutzung eigener Daten.

9.4.2 Arbeitseinheiten in Entwicklungsprozessen

Was sind nun konsistenzerhaltende Zustandsübergänge in Entwicklungsprozessen? Denkbar sind einzelne Editierkommandos, ein Compilerlauf, ganze Sitzungen, ein ganzes Projekt. Diese Beispiele sollen zeigen, daß man diese Übergänge hier besser als **Arbeitseinheiten**[34] auffaßt.

Merkmale großer Arbeitseinheiten. Kleine Arbeitseinheiten wie die Übersetzung eines Programmoduls haben sehr ähnliche Merkmale wie konventionelle Anwendungen. Die großen Arbeitseinheiten in Entwicklungsumgebungen weisen dagegen völlig andere Merkmale[35] auf:

1. Die Arbeitseinheiten können sehr lang und umfangreich sein (d.h. Laufzeit im Wochenbereich, sehr viele benutzte Objekte).
2. Die Arbeitseinheiten sind ineinander geschachtelt, d.h. die größeren Arbeitseinheiten setzen sich aus kleineren zusammen.
3. Viele Arbeitseinheiten sind interaktiv, d.h. die Arbeit kann allenfalls manuell wiederholt werden, woraus folgt, daß das System auf keinen Fall interaktive Arbeitseinheiten von sich aus zurücksetzen darf.
4. Die Benutzer kennen einander und kooperieren.
5. Es gibt keine global eindeutigen Konsistenzkriterien, sondern die Anforderungen an die Konsistenz der Daten hängen vom Bearbeitungszustand ab. Es kann durchaus sinnvoll sein, ein nur teilweise getestetes Modul eines Programms oder ein noch nicht korrekturgelesenes Kapitel eines Buchs an bestimmte andere Entwickler weiterzugeben, obwohl die entsprechende Arbeitseinheit noch nicht beendet worden ist.

[34] Für die großen Arbeitseinheiten verwenden wir auch den Begriff Vorgang, insb. im Kontext der Netzplantechnik.

[35] Diese Merkmale treffen übrigens auch vielfach auf konventionelle Anwendungen zu, wenn man einen kompletten Geschäftsvorfall (z.B. die Bearbeitung einer Bestellung eines Kunden) als Arbeitseinheit betrachtet. Dies führte zur Bildung einer Vielzahl von Konzepten für "lange Transaktionen". Auf diese gehen wir hier nicht näher ein, weil sie zu sehr von den Merkmalen betrieblicher Geschäftsprozesse, der Architektur der Anwendungssysteme und der Struktur der Datenbestände abhängen.

Eignung konventioneller Transaktionskonzepte. Es kann nicht überraschen, daß die konventionellen Transaktionskonzepte für Entwicklungsumgebungen nicht geeignet sind:

- Ein konsistenzerhaltender Zustandsübergang eines Programms beispielsweise kann Stunden oder Tage dauern, nämlich so lange, bis das Programm wieder korrekt (in welchem Sinne auch immer...) ist. Bei den üblichen Sperrverfahren wäre in der ganzen Zeit, in der an einem Dokument gearbeitet wird, niemand anders imstande, es auch nur zu lesen; dies ist völlig inakzeptabel. Wenigstens die alte Version des Dokuments muß lesbar bleiben.
- Ziel der konventionellen Transaktionskonzepte ist es, nur Verzahnungen von parallelen Zugriffen mehrerer Transaktionen auf die gleichen Daten zuzulassen, die serialisierbar sind. **Serialisierbarkeit** bedeutet, daß es zu der tatsächlich aufgetretenen Verzahnung der Transaktionen eine äquivalente gibt, in der alle Transaktionen seriell nacheinander, also anders gesehen ganz allein auf der Datenbank ausgeführt worden sind. Bei einer seriellen Ausführung sind natürlich keinerlei Interferenzen zwischen parallel ausgeführten Transaktionen möglich. In konventionellen Anwendungen sind Interferenzen immer ein Fehler, denn die einzelnen Transaktionen kennen einander nicht, sie *konkurrieren* lediglich um die gemeinsam benutzten Daten.

  Zwei Entwickler hingegen, die im gleichen Projekt arbeiten und die zwei aufeinander aufbauende Module A und B realisieren, kennen einander durchaus und sprechen sich miteinander ab, sofern ihre Arbeit voneinander abhängt, d.h. sie *kooperieren.* Sie müssen daher typischerweise schon vor Ende ihrer Arbeit Zwischenergebnisse untereinander austauschen, z.B. die Spezifikation der Module A und B. Mit der Serialisierbarkeit ist dies nicht vereinbar.
- Konventionelle Transaktionen werden ganz oder gar nicht ausgeführt. Wenn eine Transaktion aus irgendeinem Grund (Rechnerabsturz, Programmabsturz, ...) nicht beendet werden kann, werden alle Effekte von ihr wieder rückgängig gemacht. Dies wäre völlig unangemessen für Entwicklungsvorgänge: wenn nach einigen Stunden

Editierarbeit der Rechner abstürzt, darf diese Arbeit auf keinen Fall verloren sein.

Selbst wenn z.B. ein Compiler ganz bewußt ein Rollback auslöst, weil das zu übersetzende Programm nicht korrekt war und bereits gewisse Ausgaben in die Objektbank erzeugt worden sind, muß die Liste der Fehlermeldungen das Rollback überstehen, es wäre also falsch, *alle* produzierten Daten zu vernichten. Es muß möglich sein, Ausnahmen zu machen.

9.4.3 Eine Hierarchie von Arbeitseinheiten in SEU

Die vorstehende Analyse zeigt, daß bei der Softwareentwicklung signifikant verschiedene Typen von Arbeitseinheiten auftreten, die man mit eigenen Konzepten abdecken und deren Besonderheiten man genauer analysieren muß. Sinnvoll sind bei unserer derzeitigen Betrachtung nur solche Arbeitseinheiten, die Gegenstand von Rücksetzungen sein und/oder parallel ausgeführt und voneinander isoliert werden müssen. Kandidaten für zu unterscheidende Arbeitseinheiten sind (angefangen von den kleinsten):

1. elementare Editierkommandos in Editoren
2. Ausführungen von Compilern oder anderen nicht-interaktiven Werkzeugen
3. Ausführungen von Kommandoprozeduren (shell-Skripten), in denen mehrere Werkzeuge aufgerufen werden
4. Arbeitsaufträge an einen Entwickler ("Kümmern Sie sich doch mal um diese Fehlermeldung; schaffen Sie das heute nachmittag?")
5. Arbeitsaufträge an eine Gruppe ("Die Gruppe von Herrn Kiesel soll in den nächsten beiden Wochen das Administrationshandbuch komplett neu schreiben, das stimmt derzeit ja vorne und hinten nicht.")
6. ein komplettes Projekt

Der Begriff "Sitzung" oder "Ausführung eines Editors" kommt in dieser Liste nicht vor. Dies ist kein Zufall, denn wenn ein Entwick-

ler von 14 bis 17 Uhr am Rechner sitzt und während dieser Zeit seine SEU läuft, dann kann er oder sie in dieser Zeit diverseste, logisch nicht zusammengehörige Einzelarbeitsschritte durchführen, die in ihrer Gesamtmenge nicht als Gegenstand von Sperrungen oder Rücksetzungen sinnvoll sind.

Gleiches gilt für Editoren (wie `emacs`, `jove`, ...): ein laufender Editor (im Sinne eines Betriebssystemprozesses) kann mehrere Dokumente gleichzeitig bearbeiten (in verschiedenen "Puffern"); innerhalb der Ausführungszeit des Editorprozesses können mehrere ganze Arbeitsaufträge vom Bediener erledigt werden oder aber vielleicht auch nur ein Teil eines Arbeitsauftrags. Die Ausführung eines solchen Editors ist daher nicht unbedingt eine sinnvolle Arbeitseinheit im Sinne von Transaktionen.

Technische Randbedingungen. Wenn man die oben genannten Arbeitseinheiten praktisch unterstützen will, müssen auch die technischen Randbedingungen berücksichtigt werden. In dieser Hinsicht können wir drei Gruppen von Arbeitseinheiten unterscheiden:

1. **prozeßinterne Arbeitseinheiten**: diese Arbeitseinheiten werden *innerhalb eines Betriebssystemprozesses*[36] ausgeführt. Dies trifft auf die beiden kleinsten o.g. Arbeitseinheiten zu.
2. **Kommandoprozeduren**: Eine Kommandoprozedur startet mehrere Programme, die jeweils als selbständige Betriebssystemprozesse ausgeführt werden. Die Ausführung einer Kommandoprozedur kann man als **geschachtelte Transaktion** behandeln, d.h. innerhalb einer äußeren Transaktion werden **Subtransaktionen** ausgeführt. Subtransaktionen können ihrerseits Subtransaktionen haben, i.a. kann ein beliebig tiefer Subtransaktionsbaum entstehen.

   Eine äußere Transaktion entspricht hier dem Prozeß, in dem der Kommandointerpreter ausgeführt wird, der die Kommandoprozedur interpretiert. Ein vor dort aus aufgerufenes Werkzeug wird als

[36] Ein Prozeß entsteht beim Start eines Programms und kann danach als die "Ausführung" dieses Programms angesehen werden. Mit Beendigung des Programms wird der Prozeß wieder aufgelöst.

Subtransaktion ausgeführt[37]. Subtransaktionen führen zunächst ein lokales Commit durch; die Wirkungen der Subtransaktion werden aber zunächst nur innerhalb der nächsten umgebenden Transaktion sichtbar; wenn diese scheitert, werden auch die Wirkungen bereits abgeschlossener innerer Subtransaktionen wieder aufgehoben. Die Vorstellung, daß eine Transaktion ganz oder gar nicht ausgeführt wird, wird hier aufrechterhalten.

3. **langdauernde Arbeitseinheiten**: diese Arbeitseinheiten laufen *über Rechnerabschaltungen hinweg.* Sowohl der Arbeitsplatzrechner wie auch der Server, auf dem das OMS installiert ist, können zwischendurch abgeschaltet werden. Dies trifft auf die letzten drei o.g. Typen von Arbeitseinheiten zu.

9.4.4 Langdauernde Arbeitseinheiten

Behandlung paralleler Vorgänge. Bei langdauernden Arbeitseinheiten sind Sperren in ihrer ursprünglichen Form nicht mehr realisierbar: Inhaber von Sperren sind *Prozesse*, Prozesse überdauern Systemabschaltungen aber nicht, Sperren sind normalerweise nicht persistent. Es ist ohnehin unklar, welchem Prozeß man "persistente" Sperren zuordnen sollte; wie schon oben erwähnt sind Sitzungen (bzw. die zugehörigen Kommandointerpreterausführungen) dazu wenig geeignet. Es zeigt sich hier, daß die Inhaber von langfristigen, persistenten Sperren eher die Entwickler persönlich oder Arbeitsgruppen sind.

Im Gegensatz zu normalen, transienten Sperren benötigt man bei persistenten Sperren diverse Administrationsfunktionen, durch die

- die Menge der vorhandenen Sperren abgefragt,
- die Rechte aus einer Sperre explizit aktiviert oder deaktiviert,
- eine Sperre gelöscht oder auf einen anderen Benutzer übertragen,

[37] Die Subtransaktionen *innerhalb einer* Transaktion können ggf. auch parallel ausgeführt werden; sie können dabei konkurrierend auf die gleichen Ressourcen zugreifen und müssen daher wie normale "flache" Transaktionen voneinander isoliert werden. Zusammen mit dem verzögerten Commit entsteht ein ziemlich komplexes Transaktionsmodell; ein solches ist z.B. im ISO-Standard PCTE definiert.

- die Berechtigung zur Löschung und Übertragung von Sperren kontrolliert

werden kann; alle diese Funktionen sind mehr oder minder äquivalent zu Funktionen, die man von der Administration von diskretionären Zugriffsrechten kennt.

An dieser Stelle ist es nützlich, sich daran zu erinnern, daß der Effekt einer Sperre i.w. darin besteht, dem Inhaber gewisse Zugriffsrechte einzuräumen und allen "Konkurrenten" Zugriffe zu verbieten. Derartige Effekte kann man aber, wenn Entwickler (also in der OMS-Installation angemeldete Benutzer) Inhaber der Rechte sind, auch hinreichend gut mit Zugriffskontrollen realisieren. Hieraus folgt, daß man den Aspekt der Zugriffsbeschränkungen bei langdauernden Arbeitseinheiten mit Hilfe von Zugriffsrechten behandeln sollte, um keine Funktionalitäten (auch hinsichtlich der Administrationsfunktionen) zu duplizieren.

Analysiert man vor diesem Hintergrund die in der Literatur vorgeschlagenen Transaktionskonzepte für langdauernde Arbeitseinheiten (sog. **Entwurfstransaktionen** oder **Gruppentransaktionen**) und die zugehörigen Operationen `checkout` und `checkin` , so kann etwas vereinfachend die Wirkung dieser Operationen wie folgt beschrieben werden:

- `checkout` :
  1. erzeugen einer Kopie des betroffenen Dokuments als Nachfolgeversion der bisherigen Version des Dokuments; ggf. Transfer der Kopie auf einen Arbeitsplatzrechner, um sie lokal effizient zugreifbar zu machen
  2. Entzug der Schreibrechte an der bisherigen Version des Dokuments für andere Benutzer; Leserechte an der bisherigen Version bleiben bestehen
  3. einräumen von Lese- und Schreibrechten an der Kopie für den Nutzer, der das `checkout` veranlaßt hat; andere Benutzer haben keine oder allenfalls Leserechte an der Kopie, sofern sie überhaupt technisch zugreifbar ist

- `checkin` :

1. ggf. Rücktransfer der inzwischen modifizierten Kopie des Dokuments auf den Server
2. einfügen der Kopie als offen sichtbare Nachfolgeversion, hierdurch Einfrieren der Vorgängerversion (in Ausnahmefällen überschreiben der bisherigen Version)
3. einräumen von Lese- und Schreibrechten an der neuen Version für alle berechtigten Nutzer

Recovery. Bei langdauernden Arbeitseinheiten ist auch die übliche "alles-oder-nichts"-Semantik von Transaktionen unangemessen. Langdauernde Arbeitseinheiten werden nie vom System, sondern höchstens von Benutzern explizit zurückgesetzt, wobei im Einzelfall entschieden wird, welche Zwischenergebnisse vernichtet und welche verwahrt werden.

Dieser Problembereich sollte mit Hilfe von *Versionen* behandelt werden: Zu Beginn einer solchen Arbeitseinheit werden neue Versionen der benötigten Dokumente angelegt. Bei einem Abbruch der Arbeit können diese problemlos wieder gelöscht werden, bei einer erfolgreichen Beendigung kann man die inzwischen überflüssige Vorgängerversion löschen; diese wird aber oft trotzdem zwecks Dokumentation der Entwicklung noch verwahrt werden, ggf. in einem Archiv auf einem off-line-Datenträger. Diese Vorgehensweise ist die gleiche, wie man sie schon aus Versionsverwaltungssystemen wie SCCS oder CVS kennt, die Operationen wie `checkout` und `checkin` anbieten.

Die explizite Verwendung von Versionen löst auch das Problem, daß man gleichzeitig auf die letzte freigegebene Version eines Dokuments, aber auch auf die neue, gerade entstehende Version des Dokuments zugreifen können möchte.

Gesamtbewertung. Zusammenfassend können wir festhalten, daß langdauernde Arbeitseinheiten auf der Ebene der Objektverwaltung durch Zugriffskontrollen und Versionsmechanismen unterstützt werden sollten und daß die Wirkung von `checkout` und `checkin` weitgehendst durch Funktionen aus diesen Bereichen erzielt werden kann. Es

ist daher nicht sinnvoll, hierfür ein spezielles Transaktionskonzept zu entwickeln, weil es nur andere, ohnehin erforderliche Funktionalitäten duplizieren würde. Sinnvoll hingegen ist es allerdings, Operationen wie `checkout` und `checkin` *oberhalb* der Ebene der Objektverwaltung zu realisieren und insb. ein komfortables Benutzerinterface zu schaffen.

9.4.5 Prozeßinterne Arbeitseinheiten und Kommandoprozeduren

Behandlung paralleler Vorgänge. Wir hatten oben bereits einzelne Editierkommandos als Arbeitseinheiten identifiziert, die ganz innerhalb von Prozessen ausgeführt werden. Echt parallele Vorgänge sind hier eigentlich nicht möglich, weil nur ein Eingabemedium (Tastatur und Maus) vorhanden ist und weil i.d.R. die Reaktion auf eine Eingabe zuerst komplett verarbeitet wird, bevor die nächste Eingabe möglich ist.

Sofern die Umgebung mit parallel offenen Fenstern arbeitet, kann aus Benutzersicht allerdings jeweils ein Fenster durchaus einem separaten Arbeitsvorgang entsprechen, d.h. es lägen auf dieser Ebene parallele Arbeitsvorgänge vor. Diese Arbeitsvorgänge können sowohl kooperativ als auch konkurrierend miteinander sein. Eine Abdeckung aller Optionen führt zu sehr komplexen Transaktionskonzepten (s. z.B. [Pl99]), auf die wir hier nicht näher eingehen.

In Kommandoprozeduren sind parallele Vorgänge unüblich. Die dann erforderliche parallele Programmierung ist relativ umständlich und aufwendig, ferner läuft eine SEU fast nie auf einem Mehrprozessor-Rechner, und nur auf einem solchen kann durch parallele Ausführung von Kommandos die Antwortzeit verbessert werden. Daher betrachten wir diese Art paralleler Vorgänge nicht näher.

Recovery. Wir betrachten zunächst das Rücksetzen einzelner Editierkommandos. Viele Editoren bieten eine Undo-Funktion an, die den letzten Arbeitsschritt rückgängig macht. Oft kann eine solche Undo-Funktion wiederholt ausgeführt werden, wobei zwei alternative Bedeutungen auftreten:

(a) die zweite Ausführung macht das erste Undo rückgängig, ist also eher ein Redo, die dritte Ausführung wiederholt das Undo usw.

(b) man läuft mehrere Schritte in der Historie rückwärts. In diesem Fall kann man aber auch zu weit gelaufen sein, d.h. man braucht hier noch dringender als beim Fall (a) ein Mittel, um wieder vorwärts laufen zu können (also eine Redo-Funktion).

Es bietet sich an, das Undo mit Hilfe des Rollback-Mechanismus zu realisieren, insoweit als Effekte eines Kommandos in der Objektbank rückgängig zu machen sind[38]. Hierzu muß man für die Version (a) vor Beginn der Ausführung eines Kommandos eine Transaktion starten, die unmittelbar vor Ausführung des nächsten Kommandos mit Commit beendet wird, wenn das nächste Kommando nicht gerade das Undo-Kommando war.

Für Version (b) muß man Transaktionen schachteln können; man müßte die Transaktionen linear schachteln, d.h. man fängt für jedes Kommando eine neue Transaktion an und hätte nach 100 erfolgreichen Kommandos 100 angefangene Transaktionen, die alle noch irgendwann mit commit beendet werden müssen. Eventuelle Rollback-Kommandos beziehen sich immer auf die letzte gestartete Transaktion, wenn man also 30 Kommandos rückwärts laufen will, muß man 30 Rollback-Kommandos abgeben. Diese lineare Schachtelung von Transaktionen wirkt äußerst ineffizient und unbequem zu benutzen, ferner ist nach einem Undo kein Redo mehr möglich.

Daher kennen viele Transaktionsmechanismen für OMS das Konzept der **Sicherungspunkte**. Ein Sicherungspunkt ist ein identifizierbarer Zeitpunkt innerhalb einer Transaktion; mit einer speziellen Operation, einer Art partiellem Rollback, kann man alle Änderungen seit diesem Zeitpunkt aufheben, die Objektbank also bzgl. der betroffenen Objektmenge auf den Zustand zum Zeitpunkt der Erzeugung des Sicherungspunkts zurücksetzen.

---

[38] Weitere Effekte sind natürlich Veränderungen der Anzeige, z.B. Inhalte eines oder mehrerer Fenster.

Eine generelles Problem ist, daß Rollback-Konzepte in normalen DBMS, aber auch in fast allen OMS, nur ein Rückgängigmachen von Änderungen unterstützen, nicht hingegen eine Wiederholung.

Es besteht natürlich immer die Möglichkeit, innerhalb des Werkzeugs ein eigenes Protokoll aller Änderungsoperationen (analog zu einem Vorwärtsrecovery-Log, allerdings auf einer höheren Ebene) zu führen und ein Redo mit Hilfe dieses Protokolls zu realisieren. Dies würde allerdings bedeuten, daß sowohl innerhalb des Werkzeugs wie im OMS Veränderungen protokolliert werden, also Daten und Funktionalitäten dupliziert werden, was eher unelegant erscheint. Man kann hier auch auf die Nutzung der Dienste des OMS komplett verzichten, die Behandlung der Undo- und Redo-Funktionen ist dann letztlich einfacher komplett innerhalb der Werkzeuge zu realisieren.

Prinzipiell ist auch ein Redo zwischen Sicherungspunkten im OMS möglich, solange die Ausgangsdaten nicht modifiziert sind. Die zugehörigen Algorithmen sind aber sehr komplex und können hier nicht behandelt werden.

9.5 Zusammenfassung

Wir haben das Problem der parallelen Benutzung einer SEU durch mehrere Entwickler von zwei Seiten betrachtet: Von der Benutzerseite her ist festzustellen, daß im Rahmen der Anforderungsfestlegung ein Kooperationsmodell entwickeln muß, welches abhängig von den Entwicklungsmethoden und Wünschen der Benutzer ist. Unterschiedliche Kooperationsmodelle führen zu ganz verschiedenen Anforderungen an die Dienste eines OMS.

Von der Seite eines OMS aus gesehen haben wir verschiedene Mechanismen auf ihre Eignung untersucht, bestimmte Kooperationsmodelle zu realisieren. Gruppenorientierte Zugriffskontrollen sind praktisch immer erforderlich, um die Rechte unterschiedlicher Gruppenmitglieder zu kontrollieren. Bzgl. der Transaktionskonzepte ist festzustellen, daß die Größe der auftretenden Arbeitseinheiten um mehrere Größenordnungen differiert und daß je nach Größenordnung andere technische Konzepte benötigt werden. Für kleinere Arbeitseinheiten benötigt man pro-

zeßgebundene Transaktionen; diese sollten zusätzlich Rücksetzpunkte anbieten und geschachtelt werden können. Langandauernde Arbeitseinheiten werden besser durch eine Kombination von Versionierung und Rechtevergabe unterstützt. Notifizierungsmechanismen können offensichtlich direkt in Werkzeugen, die die Kooperation unterstützen, ausgenutzt werden.

Glossar

Bibliotheksmodell: Kooperationsmodell, bei dem ein Dokument zunächst in einer Bibliothek lagert, dort von genau einem Benutzer entnommen werden kann, anschließend bearbeitet wird und später wieder in die Bibliothek zurückgestellt wird

Entwurfstransaktion: Transaktionskonzept zur Unterstützung langdauernder Arbeitseinheiten; Hauptoperationen sind `checkout` und `checkin` , deren Bedeutung ähnlich wie in Versionsmanagementsystemen definiert ist und i.w. aus dem Anlegen von Versionen und Einstellen von Zugriffsrechten besteht

geschachtelte Transaktionen: Transaktion mit Subtransaktionen; Subtransaktionen können ihrerseits Subtransaktionen haben, i.a. kann ein beliebig tiefer Subtransaktionsbaum entstehen

Kommandoprozedur: Programm, das in der Kommandosprache eines Kommandointerpreters geschrieben ist; einzelne Kommandos werden ggf. als Kindprozesse des Prozesses, in dem der Kommandointerpreter ausgeführt wird, ausgeführt

Kooperationsmodell: Beschreibung bzw. Festlegung, wie Personen in einer Gruppe, die auf gemeinsamen Dokumenten arbeiten, kooperieren

langdauernde Arbeitseinheit: Arbeitseinheit, die über Rechnerabschaltungen hinweg verläuft

Objektmanagementsystem: dediziertes Datenverwaltungssystem zur Verwaltung von Entwicklungsdokumenten (CASE, CAD usw.)

prozeßinterne Arbeitseinheit: Arbeitseinheit, die innerhalb eines Betriebssystemprozesses ausgeführt wird

Serialisierbarkeit: Eigenschaft parallel überlappend ausgeführter Transaktionen, daß deren Effekt der gleiche ist, der bei irgendeiner denkbaren seriellen Ausführung der Transaktionen erreicht worden wäre

Sicherungspunkt: Zeitpunkt in einer Transaktion, bis zu dem alle Aktionen in einem partiellen Rollback rückgängig gemacht werden können

Wandtafel-Modell: Kooperationsmodell, bei dem alle Beteiligten alle Änderungen am Dokument durch alle anderen Beteiligten sofort sehen

Zugriffskontrolliste: Datenstruktur, in der die Rechtefestlegungen zu einem Objekt verwaltet werden

Literaturverzeichnis

[Be+79] Bernstein, P.A.; Shipman, D.W.; Wong, W.S.: Formal aspects of serializability in database concurrency control; IEEE-TSE SE-5:3, p.203-216; 1979/05

[BeHG87] Bernstein, P.A.; Hadzilacos, V.; Goodman, N.: Concurrency control and recovery in database systems; Addison-Wesley Publishing Company; 1987

[BoS95] Borghoff, Uwe M.; Schlichter, Johann H.: Rechnergestützte Gruppenarbeit; eine Einführung in Verteilte Anwendungen; Springer; 1995

[Ca81] Casanova, M.A.: The concurrency control problem for database systems; Springer, LNiCS 116, 175p; 1981

[CVS] Kelter, U.: Lehrmodul "Einführung in CVS"; 2003

[DAC] Kelter, U.: Lehrmodul "Diskretionäre Zugriffskontrollen"; 1999

[DBSA] Kelter, U.: Lehrmodul "Architektur von DBMS"; 2001

[DVS] Kelter, U.: Lehrmodul "Datenverwaltungssysteme"; 2002

[El92] Elmagarmid, A. (ed.): Database transaction models for advanced applications; Morgan Kaufmann; 1992

[EsC75] Eswaran, K.P.; Chamberlin, D.D.: Functional specifications of a subsystem for data base integrity; p.48-68 in Proc. VLDB75; 1975/09

[Ga83] Garcia-Molina, H.: Using semantic knowledge for transaction processing in a distributed database; ACM-TDS 8:2, p.186-213; 1983/06

[Ha84] Härder, T.: Observations on optimistic concurrency control schemes; Information Systems 9:2, p.111-120; 1984

[Ke86] Kelter, U.: Strictness and serializability; p.252-261 in: Proc. STACS 86, Paris; Springer LNiCS 210; 1986/01

[KM] Kelter, U.: Lehrmodul "Einführung in das Konfigurationsmanagement"; 2003

[Ko83] Korth, H.F.: Locking primitives in a database system; JACM 30:1, p.55-79; 1983/01

[KuR81] Kung, H.T.; Robinson, J.: On optimistic methods for concurrency control; ACM-TDS 6:2, p.213-226, 1981/06

[Pa79] Papadimitriou, C.H.: The serializability of concurrent database updates; JACM 26:4, p.631-653; 1979/10

[Pl99] Platz, Dirk: Ein Werkzeugtransaktionskonzept für Objektmanagementsysteme als Basis von Software-Entwicklungsumgebungen; Dissertation, FG PI, FB12, Univ. Siegen; 1999

[PlK97] Platz, D.; Kelter, U.: Konsistenzerhaltung von Fensterinhalten in Software-Entwicklungsumgebungen; Informatik – Forschung und Entwicklung 12:4, p.196-205; 1997/12 (ISSN 0178-3564)

[Re82] Reuter, A.: Concurrency control on high-traffic data elements; p.83-92 in: Proc. Symp. Principles of Database Systems; 1982

[RoSL78] Rosenkrantz, D.J.; Stearns, R.; Lewis, P.M.: System level concurrency control for distributed database systems; ACM ToDS 3:2, p.178-198; 1978/06

[Se82] Sethi, R.: Useless actions make a difference: strict serializability of database updates; JACM 29:2, p.394-403; 1982/04

[SeL76] Severance, D.G.; Lohman, G.M.: Differential files: their application to the maintenance of large databases; ACM-TODS 1:3, p.256-267; 1976/09

[Si82] Silberschatz, A.: A multi-version concurrency scheme with no rollbacks; p.216-223 in Proc. SPDC82; 1982/08

[StLR76] Stearns, R.E.; Lewis, P.M.; Rosenkrantz, L.J.: Concurrency control for database systems; p.19-32 in: Annual Symposium on Foundations of Computer Science; 1976

[Ve77] Verhofstad, J.S.M.: Recovery and crash resistance in a filing system; p.158-167 in: SIGMOD77; 1977/08

Stichwortverzeichnis

2-Phasen-Protokoll, 88, 94, 116, 142, 207
2PL, 88

Änderungsdatei, 70, 72
Äquivalenz
    cp-∼, 135, 195
    Endzustands-∼, 169
    fs-∼, 169
    Sicht-∼, 191
    von Abläufen, 135
Aktion, 20, 33, 76, 112, 178
Arbeitseinheit, 221, 223
    interaktive, 221
    langdauernde, 225, 231
    prozeßinterne, 224, 228, 232
    Zwischenergebnisse, 222
Arbeitsversion der Datenbank, 37
Architektur, 40
Archiv-Log, 60, 61, 72
Atomarität, *siehe Transaktion*
    Ändern mehrerer Seiten, 65
audit trail, 59
Aufrufsequenz, 80
Ausführungssequenz, 80

Backup-Kopie, 30
Bereichsgrenze, 120, 122
Betriebssystem, 40
Bibliotheksmodell, *siehe Kooperationsmodell*
BOT, 59

careful replacement, 58
CC-Verfahren
    Hilfsdaten, 133, 134, 151, 155
    kombinierte, 142, 156
    optimistische, *siehe optimistische CC-Verfahren*
    Sperrverfahren, *siehe Sperrverfahren*
    universelles, 161
    validierendes, 131, 134, 135
    Wait-Die-, 142
    Wound-Wait-, 142
    Zeitstempel-∼, *siehe Zeitstempel-Verfahren*
`checkin`, 227
`checkout`, 226
Concurrency Control, 28, 31, 34, 38, 77, 94, 218, 225, 228
Concurrency-Control-, *siehe CC-*
CPSR, 196
CVS, 227

D(l), 200
Datenbank
    aktuelle, 42, 72
    Arbeitsversion, 54
    Dump, *siehe Dump*
    materialisierte, 42, 68, 72
    physische, 42, 73
    Speicherteile, 49
Datenbankmodell, 19
Dauerhaftigkeit, 24, 25
DB, 162
Deadlock, 13, 32, 87, 91, 94, 131
Deadlockauflösung, 46
Deadlockfreiheit, 131, 132, 143

deferred update, 57
Dump, 56, 71, 72
    inkrementeller, 71, 72
Durchsatz, 76

Editor, 212
    Sitzung, 224
Einbenutzersystem, 210
Elementaroperation, 112
Endlosblockierung, 86, 144
Entwicklungsmethode, 213
Entwurfstransaktion, 14, 226, 231
EOT, 59, 66, 85
Erreichbarkeit, 20, 34
exclusive lock, 81

Fehler-Atomarität, *siehe Transaktion*
Fehlerklasse, 42
Fehlerkompensation, 55, 72
forward recovery, 57
freie Interpretation, 173

Granularität, 97, 109
    variable, 97, 105
group awareness, 212
Gruppenarbeit, 212
Gruppentransaktion, 226

Hauptdatei, 70
Herbrand-Universum, 174

Idempotenz, 68, 72
integer, 19
Integrität, 19, 34, 48
    ∼sanforderung, 22, 26
    ∼sprüfungen, 28, 112
    Sicherung, 28
intention lock, 101
Interferenz, 76, 77, 159, 222
Interpretation, 113, 168
Invertierung, 116
Isolation, 25, 34
Isolationsstufe, 92, 94
ISR, 191

JDBC, 23
    `commit`, 24
    `rollback`, 30
    `setAutoCommit`, 24
journal, 59

Kommandoprozedur, 224, 231
Kommunikation, 212
Kommutativität, 13, 113, 118
kompatibel, *siehe Sperrmodus*
Kompensation, 116
Konflikt, 111, 137, 147, 157, 193
    Schein-, 150
konfliktfrei, 79, 94, 116, 119
    mit Objektzustandseinschränkungen, 127
    mit Parametereinschränkungen, 125
    semantisch, 114
Konfliktgraph, 200
Konkurrenz, 222
Konsistenz, 24, 34, 213, 218, 220, 221
    Dokument∼, 215
    logische, 20, 34, 44
    physische, 21, 34, 43, 44, 73
    temporäre In∼, 22
Kooperation, 14, 212
Kooperationsmodell, 211, 231
    Adaptierbarkeit, 216
    Anforderungsdefinition, 214
    anonyme Benutzer, 211, 218
    Bibliotheksmodell, 214, 231
    Dokumentkonsistenz, 215
    Wandtafel-Modell, 215, 219, 232
Koordination, 212
    SEU-Dienste, 214
Korrektheit, 19
    technische, 27, 35
    von Abläufen, 134

lebendig, 179
Lese-Schreib–Transaktion, 162
Lesephase, 148
Lesesperre, *siehe Sperre*

Lesetransaktion, *siehe Transaktion*
liest-von-Struktur, 194
Log, 31, 59, 66, 72, 159
  aktiver, 61, 72
  serieller, 167
log sequence number, 59
Logging, 59
  Abstraktionsebene, 62
  auf Seitenebene, 62
  auf Speichersatzebene, 63
  auf Transaktionsebene, 63
  Redo-∼, 60
  Rückwärts-∼, 60
  Transitions-∼, 62, 68, 74
  Undo-∼, 59
  Vorwärts-∼, 60
  Zustands-∼, 62, 68, 74
logische Atomarität, *siehe Serialisierbarkeit*

Materialisieren, 42, 69
  bei Commit, 66
  unsicherer Änderungen, 65
materialisierte Datenbank, 42
Medienfehler, 43, 71, 72
Modifikation, 112
  Atomarität, 117
  inverse, 116, 122
  kommutierende, 113
  parametrisierte, 118
    Konfliktfreiheit, 119
  Undo, 115
  Zwischenzustände, 122

Nachbarn, 107, 109
Nebenläufigkeitssteuerung, *siehe Concurrency Control*
Neuladen der Datenbank, 43, 72
Neustart, 31, 46, 132, 142, 143, 147
  zyklischer, 132, 139, 154
Notifikation, 219

Objekt
  elementares, 99
  übergeordnetes, 98
Objektmanagementsystem, 232
On-line-Scheduler, 79, 94, 186, 187
optimistische CC-Verfahren, 13, 32, 34, 134, 147, 153
  Grundform, 149, 151
  präventive Validation, 154

Permanentspeicher, 44
physische Datenbank, 42
Präfix, 165
präfixe(M), 165
Präventionsmaßnahme, 37, 64
Preclaiming, 91, 94
Protokoll, 83, 94
  2PL, 88
  SE, 86
  SPE, 85
  XPE, 85
Pufferung, 41

R_i, 162
`readset`, 149, 150, 151, 152, 153, 155, 156
readset_i, 162, 163
Recovery, 28, 30, 34, 73, 218, 227, 229
  ∼-Daten, 30, 37, 50, 53, 56
  ∼-Grundfunktion, 43, 45, 46
  ∼-Manager, 41
  Grundprinzipien, 52
  Qualitätsanforderungen, 49
  Rückwärts-∼, 30, 55, 64, 73
  Vorwärts-∼, 30, 57, 74
  Zielzustand, 50
Redo, 229, 230
  ∼-Checkpoint, 69, 73
  ∼-Log, 59, 73
  ∼-Logging, 60
  einer Aktion, 45, 73
  globales, 43, 71, 72
  partielles, 45, 68, 69, 73
Reparaturprinzipien, 54

Rettung, 55
Risikostreuung, 53
Rollback, 31, 34, 46, 64, 73, 82, 116, 121, 131, 132, 229, 230
  Fortpflanzung, 90, 95, 115
  partielles, 230
Rücksetzen, *siehe Rollback*
  der DB, 71

S, 168
Schaden, 12, 40, 73
Scheduler, 79, 186
Scheduling, 136
Schreibphase, 148
Schreibsperre, *siehe Sperre*
Schrumpfungsphase, 89
SCPSR, 196
Seitenadressierung, 65
semantische Integritätsprüfungen, 28
Serialisierbarkeit, 14, 25, 34, 76, 77, 94, 95, 111, 113, 131, 135, 222, 232
  cp-∼, 111, 196
  fs-∼, 169
  logische Atomarität, 159
  Präfix-fs-∼, 170
  schwache fs-∼, 171
  schwache Sicht-∼, 189
  Sicht-∼, 191
  strikte, 173
  strikte Sicht-∼, 191
Serialisierung, 169
Serialisierungspunkt, 77, 80, 89, 137, 198
  innerer, 198
serieller Ablauf, 135
shared lock, 81
shuffle-Operator, 164
Sicherungspunkt, 230, 232
Sicht, 111, 159
  Fehlercode, 113, 120
SISR, 191
Sitzung, 224
Sperre, 11, 80, 95, 216
  Anforderung, 81
    Endlosblockierung, 86
    explizite, 82
    implizite, 82
  exklusive, 81
  Freigabe, 81, 116
  implizite, 100, 103, 106, 107, 108, 109
  Kompatibilität, *siehe Sperrmodus*
  Lesesperre, 81, 85, 93, 94
  persistente, 225
  Schreibsperre, 81, 95
  Verträglichkeit, *siehe Sperrmodus*
  Warnsperre, 12, 101, 102, 106, 108, 109
  Zuteilung, 123, 124, 126, 144
Sperreinheit, 11, 97
  Strukturierung, 98
Sperren bis EOT, 90
Sperrmodus, 81, 95, 114, 125
  IS/IX, 101, 102, 103, 108
  Kompatibilität, *siehe Verträglichkeit*
  Rechte, 114, 128
  SIX, 103, 104, 108
  Vergleich, 104
  Verträglichkeit, 81, 95, 115, 119
  Verträglichkeitsmatrix, 81, 103, 115, 119, 125, 128
Sperrtabelle, 152
Sperrverfahren, 32, 34, 147, 156, 218, 222
SR, 169
SSR, 173
Störung, 28, 39, 73
Subtransaktion, 225, 231
symbolische Interpretation, 173
system log, 59
Systemfehler, 44, 58, 64, 69, 73
  beim globalen Undo, 67
  beim partiellen Redo, 69

`tcr(x)`, 151
`tcw(x)`, 153
Teillog, 165

TR, 163
Transaktion, 23, 35, 40, 161, 210, 218
  Ausgaben, 47
  Eigenschaften, 24
  Fehleratomarität, 24
  geschachtelte, 17, 224, 229, 231
  interaktive, 47, 223
  konventionelle, 220
  kurze, 220
  lange, 221
  Lese∼, 163, 220
  parallele, 28
  Rollback, *siehe Rollback*
  Serialisierbarkeit, *siehe Serialisierbarkeit*
  tote, 178
Transaktionsfehler, 46, 64, 73
Transaktionsmonitor, 17
Transaktionsprogramm, 23

überschreibt-Struktur, 195
Umordnung, 172
Undo, 229, 230
  ∼-Checkpoint, 67, 74
  ∼-Log, 59, 74
  ∼-Logging, 60
  einer Aktion, 45, 74
  einer Modifikation, 115
  globales, 45, 66, 72
unsicher, 90, 93, 95, 115, 123
  Teilobjekt, 128
Unsicherheitsbereich, 123

Validation, 35, 131, 147
  präventive, 154
Validationsphase, 147
Validationstest, 131, 134, 135, 136, 138, 139, 147, 148, 153, 155
  Algorithmus, 138, 149, 155
Version, 227
verträglich, *siehe Sperrmodus*
Verzahnung, 77
verzögertes Schreiben, 57, 64, 74, 132, 147
vorsichtiges Ändern, 58, 65, 74

W_i, 162
Wachstumsphase, 88
Wait-Die-Verfahren, 142, 143, 144
Wandtafel-Modell, *siehe Kooperationsmodell*
Warnsperre, *siehe Sperre*
Warnsperrenprotokoll, 101, 104, 108
wechselseitiger Ausschluß, 32, 80
WISR, 189
Wound-Wait-Verfahren, 142, 143
`writeset`, 149, 151, 152, 155, 156
writeset_i, 162, 163
WSR, 171

Zeitstempel, 136, 140, 149
  Verwaltung, 141
Zeitstempel-Verfahren, 13, 32, 35, 134, 136, 154
`ZR(x)`, 138, 141
Zugriffskontrollen, 15, 217
Zugriffskontrolliste, 217, 232
Zugriffsrecht, 226
Zustand, *siehe Konsistenz*
  erreichbarer, 20
  logisch konsistenter, 51
  technisch korrekter, 48, 51
Zuverlässigkeit, 38
`ZW(x)`, 138, 142
zyklischer Neustart, 139