Transaktionen in föderierten Datenbanksystemen unter eingeschränkten Isolation Levels

Ralf Schenkel

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
an der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Dekan der Naturwissenschaftlich-Technischen Fakultät I:	Prof. Dr. Rainer Schulze-Pillot-Ziemen
Vorsitzender der Prüfungskommission:	Prof. Dr. Reinhard Wilhelm
Erstgutachter:	Prof. Dr. Gerhard Weikum
Zweitgutachter:	Prof. Dr. Wolfgang Paul
Tag des Promotionskolloquiums:	19. Dezember 2001

Inhaltsverzeichnis

Kurzfassung

Atomarität und Isolation von Transaktionen sind Schlüsseleigenschaften fortgeschrittener Anwendungen in föderierten Systemen, die aus verteilten, heterogenen Komponenten bestehen. Während Atomarität von praktisch allen realen Systemen durch das Zweiphasen-Commitprotokoll gewährleistet wird, unterstützt kein System eine explizite föderierte Concurrency Control. In der Literatur wurden zwar zahlreiche Lösungsansätze vorgeschlagen, doch sie haben wenig Einfluss auf Produkte genommen, weil sie die weitverbreiteten Isolation Levels nicht berücksichtigen, die Applikationen Optimierungsmöglichkeiten auf Kosten einer eingeschränkten Kontrolle über die Konsistenz der Daten erlauben.

Diese Arbeit vergleicht zunächst existierende Definitionen für Isolation Levels und entwickelt eine neuartige, formale Charakterisierung für Snapshot Isolation, dem Isolation Level des Marktführers Oracle. Anschließend werden Algorithmen zur föderierten Concurrency Control vorgestellt, die beweisbar auch unter lokaler Snapshot Isolation die korrekte Ausführung föderierter Transaktionen gewährleisten, und Isolation Levels für föderierte Transaktionen diskutiert. Die Algorithmen sind in ein prototypisches föderiertes System integriert. Performancemessungen an diesem Prototyp zeigen ihre praktische Einsetzbarkeit.

Abstract

Atomicity and isolation of transactions are key requirements of advanced applications in federated systems consisting of distributed and heterogeneous components. While all existing federated systems support atomicity using the two-phase commit protocol, they lack support for federated concurrency control. Many possible solutions have been proposed in the literature, but they failed to make impact on real systems because they completely ignored the widely used concept of isolation levels, which offer optimization options to applications at the cost of less rigorous control over data consistency.

This thesis compares existing definitions for isolation levels and develops a new characterization for Snaphot Isolation, an isolation level provided by Oracle, the market leader in the database field. We present algorithms for federated concurrency control that provably guarantee the correct execution of federated transactions even under local Snapshot Isolation, and discuss isolation levels for federated transactions. The algorithms are integrated into a federated system prototype. Performance measurements with this prototype show the practical viability of the developed methods.

Zusammenfassung

Eine Schlüsseleigenschaft fortgeschrittener Anwendungen in föderierten Systemen wie dem Internet, die aus verteilten und oft hochgradig heterogenen Komponenten bestehen, ist die Gewährleistung von *Atomarität* und *Isolation* föderierter Transaktionen. Die immer wichtiger werdenden E-Services wie E-Commerce und elektronische Auktionen, aber auch der Zusammenschluss von Unternehmensdatenbanken wegen Kooperationen und Fusionen, sind ohne sie nicht denkbar. Atomarität und Isolation sind in föderierten Systemen inhärent schwieriger sicherzustellen als in homogenen, zentral administrierten verteilten Datenbanksystemen. Das liegt vor allem daran, dass die einzelnen Komponentensysteme unterschiedliche Verfahren zur lokalen Concurrency Control einsetzen und der föderierten Ebene keinen Einblick in die Einzelheiten der internen Abläufe bieten.

In den letzten Jahren wurden in der Literatur viele Ansätze zur Transaktionsverwaltung in föderierten Systemen vorgeschlagen, die sich aber nur im Bereich der Atomaritätssicherung durchgesetzt haben. Praktisch kein existierendes föderiertes System setzt dagegen Mechanismen zur föderierten Concurrency Control ein. Dies liegt vor allem daran, dass die vorhandenen Algorithmen zu strenge Voraussetzungen an die beteiligten Komponentensysteme machen. Insbesondere ignorieren sie die Tatsache, dass nahezu jedes Datenbanksystem eingeschränkte Korrektheitskriterien, die sogenannten *Isolation Levels*, einsetzt, um die erzielbare Performance auf Kosten gewisser Abschwächungen der Konsistenzgarantien zu erhöhen. Obwohl solche Mechanismen weit verbreitet sind, gibt es nur wenige Forschungsarbeiten, die sich damit beschäftigt haben.

Wir stellen in dieser Arbeit Algorithmen zur föderierten Concurrency Control vor, die beweisbar auch dann die korrekte Ausführung föderierter Transaktionen gewährleisten, wenn einige der beteiligten Datenbanksysteme nur eingeschränkte Isolation Levels unterstützen. Einen besonderen Schwerpunkt bilden dabei Systeme, die *Snapshot Isolation* garantieren; dieser Isolation Level ist deshalb sehr praxisrelevant, weil er der höchste Korrektheitsgrad ist, den der Marktführer Oracle unterstützt. Wir vergleichen in dieser Arbeit verschiedene Definitionen, die in der Literatur für Isolation Levels vorgeschlagen wurden, und entwickeln anschließend eine neuartige formale Charakterisierung für Snapshot Isolation, die insbesondere erlaubt, auch in solchen Datenbanken serialisierbare Schedules zu gewährleisten, die nur Snapshot Isolation sicherstellen. Auf dieser Basis untersuchen wir, welche etablierten Algorithmen unter lokaler Snapshot Isolation noch einsetzbar sind, und diskutieren mögliche Variationen, um die globale Korrektheit sicherstellen zu können. Wir konzentrieren uns dabei auf die verschiedenen Varianten der Tickettechnik von Georgakopoulos et al. und auf Mehrschichtentransaktionen von Weikum et al. Zusätzlich leiten wir aus unserer Charakterisierung für Snapshot Isolation einen neuartigen Algorithmus zur föderierten Concurrency Control ab.

Ähnlich wie in den Komponentensystemen möchte man auch auf der föderierten Ebene Isolation Levels einsetzen, um die potentiell erzielbare Performance zu erhöhen, wenn die Anwendungen mit den möglicherweise auftretenden Anomalien leben können. Wir zeigen, dass sich die Isolation Levels aus dem ANSI-SQL-Standard unterhalb von Serialisierbarkeit einfach auf föderierte Transaktionen übertragen lassen, ohne dass dazu besondere Aktionen notwendig wären. Föderierte Snapshot Isolation, die besonders bei Föderationen aus Oracle-Datenbanken interessant ist, kann dagegen nur mit zusätzli-

chen Maßnahmen sichergestellt werden. Wir entwickeln zwei Algorithmen, die verschiedene Ansätze verfolgen, um für föderierte Transaktionen Snapshot Isolation sicherzustellen.

Alle Algorithmen, die wir in dieser Arbeit entwickelt haben, wurden in ein Prototypsystem integriert, das auf der Basis des existierenden föderierten Datenbanksystems *VHDBS* entwickelt wurde. VHDBS wurde vom Fraunhofer-Institut für Software- und Systemtechnik (ISST) in Dortmund entwickelt; es integriert auf der Basis des Wrapper-Mediator-Ansatzes von Wiederhold heterogene Datenbanksysteme, unter anderem das objektrelationale System Oracle und das objektorientierte System O_2. Sowohl zur internen Kommunikation als auch clientseitig verwendet VHDBS *CORBA* in Gestalt des Objekt Request Brokers Orbix. Wir haben VHDBS um *TraFIC*, eine Komponente zur Transaktionsverwaltung, erweitert, die verschiedene Strategien zur Gewährleistung von Atomarität und Isolation föderierter Transaktionen unterstützt, und zwar selbst dann, wenn manche der beteiligten Datenbanksysteme lediglich Snapshot Isolation garantieren. Ein Systemadministrator kann die Strategie auswählen, die für die jeweilige Anwendungsumgebung die beste Performance verspricht; zusätzlich gibt es Kombinationsstrategien, die automatisch die vermutlich beste Strategie auswählen. Die erweiterbare Architektur von TraFIC erlaubt es, neue Strategien auf einfache Weise in das System einzubinden. Analog zu VHDBS ist auch TraFIC CORBA-basiert, es verwendet zusätzlich Dienste des CORBA Object Transaction Service in der Implementierung Orbix, um die Atomarität verteilter Transaktionen sicherzustellen.

Anhand dieses Prototyps untersuchen wir die Leistungsfähigkeit der Algorithmen, die wir entwickelt haben. Wir entwerfen dazu zunächst eine Suite von Benchmarkprogrammen, die quantitative Urteile über die Performancepotentiale der einzelnen Strategien erlauben. Umfassende Messungen zeigen, dass die neu entwickelten Strategien für den praktischen Einsatz ausreichend leistungsfähig sind.

Summary

Data consistency and transaction atomicity are key requirements of advanced applications in federated systems like the Internet that consist of distributed and often highly heterogeneous components. Existing and upcoming E-service applications such as electronic auctions and mobile commerce, but also the federation of business databases when two companies collaborate or merge, are not conceivable without the underlying transactional mechanisms. Providing atomicity and isolation is much harder in federated systems than in a homogeneous, centrally administered distributed database system. Among the reasons for this are the different concurrency control protocols used by the component databases and the invisibility of internal processing details to software at the federated level.

Recently many algorithms for federated transaction management have been proposed in the literature. However, only techniques for federated atomicity have been adopted by real products. Virtually no existing federated database system includes algorithms for federated concurrency control. The reason is that the solutions proposed in the literature impose hard constraints on the databases in the federation that most real database products do not meet. In particular, they completely ignore the frequent use of relaxed correctness criteria, the so-called *Isolation levels*. Isolation Levels allow increasing the performance of many applications at the cost of less rigorous guarantees for data consistency. Even though such mechanisms are widely used in existing systems, the scientific literature has mostly ignored them.

In this thesis we present algorithms for federated concurrency control that provably guarantee the correct execution of federated transactions, even when some of the involved database systems guarantee only restricted isolation levels. We focus on systems supporting Snapshot Isolation. This level is highly relevant for real-world applications as it is the highest degree that Oracle, the market leader, can guarantee for transactions. We compare several proposed definitions of Isolation Levels. For Snapshot Isolation, we develop a new graph-based characterization that can be applied to guarantee a serializable execution on top of Snapshot Isolation. Based on these formal definitions we reconsider existing algorithms for federated concurrency control and extend them so that they can guarantee globally correct execution even when some component systems support only Snapshot Isolation. We focus on the family of ticket techniques by Georgakopoulos et. al. and multilevel transactions by Weikum et. al. From the characterization of Snapshot Isolation we also derive a new algorithm for federated concurrency control.

Analogously to the local component systems one may use Isolation Levels for federated transactions to increase the potential performance for applications, if they can cope with the effects of the lower degree of consistency. We prove that the Isolation levels from ANSI SQL weaker than serializability automatically hold for federated transactions if they are guaranteed in all component systems. To ensure federated Snapshot Isolation, which is most interesting in Oracle-only federations, additional actions at the federated layer are necessary. We propose two different algorithms to guarantee Snapshot Isolation for federated transactions.

The algorithms presented in this thesis are integrated into a prototype system which was developed based on the federated database system VHDBS. VHDBS was designed and

implemented by Fraunhofer Institute for Software and Systems Engineering in Dortmund, it integrates heterogeneous database systems based on Wiederhold's wrapper-mediator paradigm, including the object-relational system Oracle and the object-oriented system O_2. For both internal and external communication, VHDBS makes use of Orbix, Iona's implementation of the CORBA standard. We have integrated a transaction manager, coined *TraFIC*, into the VHDBS core. TraFIC offers a suite of strategies to guarantee atomicity and serializability for federated transactions even if some component systems support only Snapshot Isolation. The administrator of the VHDBS system may choose the strategy that promises the best performance for a given application environment; additionally, there are strategies that automatically aims to choose the (performance-wise) best strategy among a subset of the available strategies. TraFIC has an extensible architecture that allows easy integration of newly developed strategies. Like VHDBS, TraFIC is based on CORBA and uses OrbixOTS, Iona's implementation of CORBA's Object Transaction Service, to guarantee that federated transactions are executed atomically.

The developed prototype serves as a test bed for the experimental evaluation of the proposed algorithms. We develop a suite of benchmark programs for systematic performance comparison of the different strategies. The experimental results show the practical viability of the newly developed techniques.

1 Einleitung

1.1 Föderierte Datenbanksysteme

Die weltweite Vernetzung über das Internet oder private Netze hat die Informationstechnik entscheidend verändert. War man früher auf eigene, lokale Rechnersysteme und Daten angewiesen, kann man heute mit Informationen aus der ganzen Welt arbeiten. Moderne Handelsformen wie E-Commerce, M-Commerce und Business-to-Business-Anwendungen sind dadurch erst möglich geworden [DE00]. Üblicherweise werden solche Softwaresysteme auf der Basis der bestehenden Software entwickelt, so dass die Anwendungen mit vielen verschiedenartigen Datenquellen arbeiten müssen; unter anderem können dies Webserver, Datenbanken verschiedener Hersteller und Applikationsserver sein. Viele Anwendungsgebiete erfordern eine einheitliche Sicht auf die Daten aus allen diesen Quellen. Typische Beispiele dafür sind etwa Portale, die Informationen aus verschiedenen Quellen aufbereiten und zentral präsentieren, oder Shop-Systeme, die Produkte verschiedener Händler unter einer einheitlichen Oberfläche anbieten. Online-Shops müssen darüber hinaus auch auf die Datenbanken von Kreditkartenanbietern zugreifen, um Zahlungen für Lieferungen anzufordern; außerdem muss natürlich auch die Datenbank des Shops selbst integriert werden, um Bestellungen und Lieferungen zu verbuchen. Ein weiteres Beispiel sind Kooperationen oder Zusammenschlüsse von Firmen, die es erfordern, dass die (sehr wahrscheinlich heterogenen) Daten der Partnerfirmen zu einer einzigen, gegebenenfalls virtuellen, Datenbank integriert werden.

Aber auch firmenintern spielen solche Anwendungen eine große Rolle, zum Beispiel wenn Informationen aus den einzelnen Abteilungen und Geschäftsbereichen einer Firma in der Geschäftsführung gesammelt und ausgewertet werden müssen. Es kann sich dabei etwa um die Kunden der Geschäftsbereiche handeln, um Umsatz- und Kosteninformationen oder um Personaldaten. Neben den Zugriffen auf die integrierten Daten in der Geschäftsführung müssen auch die bestehenden Vorgänge in den einzelnen Abteilungen weiter laufen können, die auf die vorhandenen Systeme abgestimmt sind.

Anwendungen wie die genannten integrieren mehrere, bestehende Datenbanken, die unabhängig voneinander entwickelt wurden [Ston98, CSS99, Klee99, HSC99]; man spricht dann von einem *Multidatenbanksystem* [LMR90, SL90, HBP93, BE96, Conr97, MKRZ99]. Der wesentliche Unterschied von Multidatenbanksystemen gegenüber homogenen, verteilten Datenbanksystemen ist die *Heterogenität* der beteiligten Datenbanksysteme, der sogenannten *Komponentendatenbanksysteme* (oder kurz *Komponentensysteme*). Während verteilte Datenbanksysteme von Datenbanksoftware eines einzigen Herstellers verwaltet werden, bilden Multidatenbanksysteme einen Verbund aus heterogenen Datenbanksystemen verschiedener Hersteller. Die einzelnen Systeme können sich unter anderem in ihren Interfaces, in ihrer Anfragesprache, in ihrem Datenmodell (relational, objektrelational oder objektorientiert) oder in ihren Mechanismen zur Transaktionsverwaltung unterscheiden. Zusätzlich spielt auch die *semantische Heterogenität* der Komponentensysteme eine wichtige Rolle. Man meint damit, dass in den Komponentensystemen gleiche oder zusammenhängende Daten verschieden interpretiert werden, so dass bei der Integration dieser Daten Schwierigkeiten auftreten können. Preise von Produkten können etwa in den einzelnen Datenbanken in verschiedenen Währungen sowie mit oder ohne Steueranteilen abgelegt sein. Es ist die Aufgabe des Multidatenbanksystems, solche Heterogenitäten zu erkennen und gegenüber den An-

wendungen, die es verwenden, zu maskieren. Während es für die systemspezifischen Unterschiede oft akzeptable Lösungen gibt, kann das Problem der semantischen Heterogenität bisher nur für konkrete Einzelfälle gelöst werden.

Ein Multidatenbanksystem wird *föderiertes Datenbanksystem* genannt, wenn seine Komponentensysteme weitgehend selbständig bleiben, etwa weil sie weiter dezentral administriert werden und lokale Anwendungen ausführen [SL90, Conr97, ÖV98]. Man spricht auch von der Autonomie der Komponentensysteme; dabei unterscheidet man drei Arten von Autonomie. *Entwurfsautonomie* fordert, dass die lokalen Schemata der Komponentendatenbanken unverändert erhalten bleiben und auch vom föderierten System nicht geändert werden können. Die *Kommunikationsautonomie* eines Systems bedeutet, dass das System (bzw. sein Administrator) selbst entscheidet, ob und wann es an einer Föderation teilnimmt und welche Daten es in die Föderation einbringt. Die *Ausführungsautonomie* schließlich garantiert, das jedes Komponentensystem selbst bestimmt, welche Anfragen und Änderungen es ausführt und in welcher Reihenfolge es dies tut. Insbesondere bedeutet dies, dass die lokalen Systeme über eigene, potentiell heterogene Transaktionsverwaltungen verfügen. Wir werden später sehen, dass dies zu Schwierigkeiten führen kann, sobald föderierte Transaktionen datenbankübergreifend arbeiten. Die Gewährleistung der Autonomie der Komponentensysteme spielt im praktischen Einsatz föderierter Systeme eine große Rolle, weil häufig Datenbanken integriert werden, für die es bereits eine Menge lokaler Anwendungen gibt, die nicht verändert werden können, aber weiter funktionsfähig bleiben müssen und auch sonst (z.B. in ihrer Leistung) nicht eingeschränkt werden sollen.

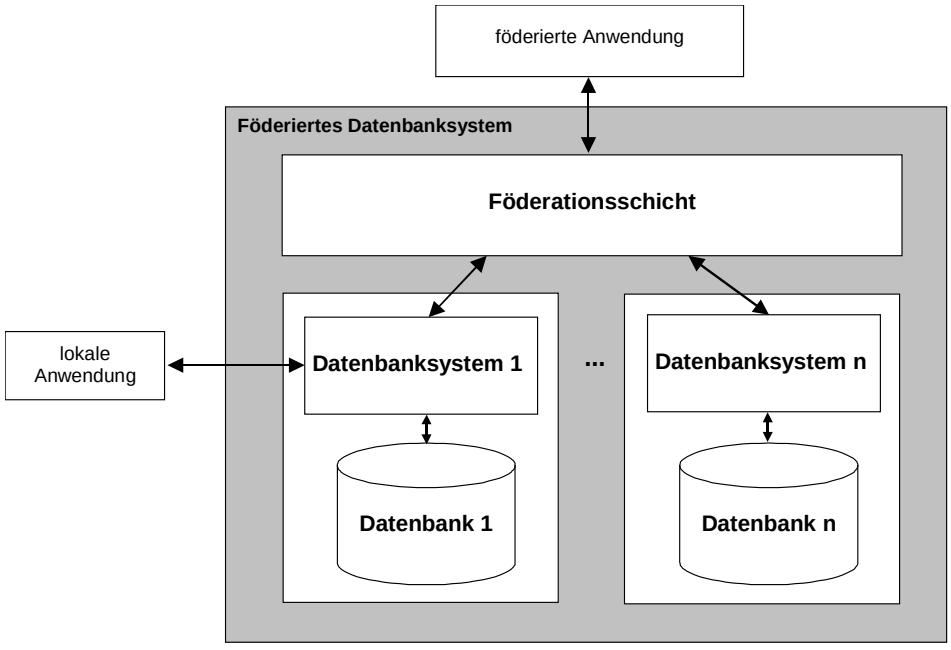

Abbildung 1.1 - allgemeine Architektur eines föderierten Datenbanksystems

Abbildung 1.1 zeigt die allgemeine Architektur eines föderierten Datenbanksystems. Die Kernkomponente bildet dabei die *Föderationsschicht*, die Daten aus dem Komponentensystem unter einem einheitlichen Interface und in einem einheitlichen Datenmodell zur Verfügung stellt. Föderierte Anwendungen können Anfragen auf diesen Daten stellen und Änderungen vornehmen; es ist Aufgabe der Föderationsschicht, daraus entsprechende Operationen in den Komponentensystemen abzuleiten. Daneben gibt es

auch Anwendungen, die unmittelbar auf die Komponentensysteme zugreifen, ohne die Föderationsschicht zu benutzen.

1.2 Transaktionen in föderierten Datenbanksystemen

Ein wichtiges Problem, das in föderierten Datenbanken gelöst werden muss, ist es, die Konsistenz der verteilten Daten über die einzelnen Komponentensysteme hinweg sicherzustellen. Analog zu zentralisierten Datenbanken kapselt man dazu zusammengehörende Zugriffe auf die föderierte Datenbank zu einer *föderierten* oder *globalen Transaktion* [BGS92, BGS95], die für sich genommen die föderierte Datenbank von einem konsistenten Zustand in einen anderen überführt. Für solche Transaktionen garantiert das System gewisse Korrektheitskriterien, die sogenannten *ACID-Eigenschaften*. ACID steht dabei für Atomarität, Konsistenzerhaltung, Isolation und Dauerhaftigkeit von Transaktionen:

- *Atomarität*: Wenn eine föderierte Transaktion erfolgreich beendet wurde, werden alle Änderungen, die sie in den Komponentendatenbanken gemacht hat, persistent gemacht, oder überhaupt keine. Wird eine Transaktion abgebrochen, hat sie in keiner Komponentendatenbank einen Effekt.
- *Konsistenzerhaltung*: Föderierte Transaktionen verletzen keine Konsistenzbedingungen, die auf den Daten einer Komponentendatenbank oder auch zwischen Daten verschiedener Komponentendatenbanken definiert sind. Es ist Aufgabe des föderierten Datenbanksystems, die Einhaltung dieser Konsistenzbedingungen sicherstellen, indem föderierte Transaktionen, die eine Bedingung verletzen, zurückgesetzt werden.
- *Isolation*: Nebenläufige Transaktionen verhalten sich so, als wären sie alleine im System; insbesondere sehen sie keine partiellen Effekte der anderen Transaktionen. In föderierten Datenbanken fordert man zusätzlich, dass die Teiltransaktionen zweier nebenläufiger Transaktionen so ausgeführt werden, als seien sie in allen Komponentensystemen in der gleichen Reihenfolge abgelaufen.
- *Dauerhaftigkeit*: Änderungen von erfolgreich beendeten föderierten Transaktionen überleben Fehlerzustände des föderierten Datenbanksystems und der Komponentensysteme, sowohl Softwarefehler als auch Hardwareschäden.

In dieser Arbeit setzen wir voraus, dass die beteiligten Komponentensysteme selbst die Dauerhaftigkeit von Transaktionen garantieren, so dass das föderierte Datenbanksystem keine besonderen Maßnahmen dazu treffen muss. Wir beschäftigen uns ebenfalls nicht mit den Maßnahmen zum Erhalt der Konsistenz der Datenbank, sondern konzentrieren uns auf Atomarität und Isolation föderierter Transaktionen. Die Gewährleistung dieser beiden Eigenschaften ist in föderierten Systemen inhärent schwieriger als in homogenen, zentral administrierten verteilten Datenbanksystemen. Das liegt unter anderem daran, dass die Komponentensysteme der Föderation unterschiedliche Algorithmen und Protokolle zur lokalen Transaktionsverwaltung einsetzen, und dass zusätzlich lokale Transaktionen an der föderierten Ebene vorbei direkt auf den lokalen Datenbanken arbeiten. Wegen der Autonomie der Komponentensysteme kann man die lokalen Verfahren weder beeinflussen noch voraussetzen, dass man Informationen über die internen Abläufe erhält. Verfahren zur Gewährleistung von Atomarität und Isolation müssen

daher ausschließlich mit Informationen arbeiten, die an der Schnittstelle der Komponentensysteme verfügbar sind.

Atomarität und Isolation von föderierten Transaktionen sind für moderne Anwendungen essentiell. Betrachten wir zum Beispiel eine Anwendung aus dem E-Commerce, die zur Abwicklung eines Online-Kaufs im Kontext einer Transaktion

- den Verkauf in der Datenbank eines Händlers einträgt, damit die Waren geliefert werden,
- die Belastung der Kreditkarte des Kunden in der Datenbank des Kartenunternehmens vermerkt, und
- einem Vermittler die Provision für den Verkauf gutschreibt.

Wenn die Atomarität einer solchen Transaktion nicht sichergestellt wird, kann es passieren, dass der Kunde seine Ware erhält, ohne dass seine Kreditkarte dafür belastet wird, weil zeitweilig die Datenbank des Kartenunternehmens nicht erreichbar war. Lässt man die Serialisierbarkeit außer Acht, kann zum Beispiel das letzte Exemplar eines Artikels doppelt verkauft werden, so dass einem Kunden nichts geliefert werden kann; dazu prüfen zwei nebenläufige Transaktionen zunächst, ob noch eine ausreichende Menge vorhanden ist, und verbuchen dann den Verkauf (und die Belastung der Kreditkarte). War der erste Fall noch vorteilhaft für den Kunden, wird er sich spätestens bei Eintreten dieses Problems mit einer Beschwerde an den Online-Shop wenden und zukünftig eventuell einen anderen Anbieter bevorzugen, der korrekte Anwendungen einsetzt.

In der Literatur wurden in den letzten Jahren zahlreiche Verfahren vorgeschlagen, um Atomarität und Isolation föderierter Transaktionen sicherzustellen. Das Zweiphasen-Commit-Protokoll, das in Form des XA-Interfaces der X/Open-Gruppe [XOpen96] standardisiert wurde, hat sich dabei als Methode der Wahl zur Gewährleistung föderierter Atomarität durchgesetzt. Auf der Seite der föderierten Concurrency-Control wurde eine große Menge von Verfahren vorgeschlagen, die von der Einschränkung der zugelassenen Protokolle in den Komponentendatenbanken bis zu komplexen Transaktionsmanagern auf der föderierten Ebene reichen [Weihl89, BGRS91, Raz91, BS92, Raz92, GRS94]. In existierenden Produkten zur Föderation unabhängiger Datenbanksysteme, wie IBM's Datajoiner, der in DB2 integriert wurde [IBM98], Oracle's Heterogeneous Services [Orac99a, Orac99b] oder Implementierungen des CORBA Object Transaction Service OTS [OMG97], findet sich allerdings lediglich das Zweiphasen-Commitprotokoll wieder, das für die Atomarität föderierter Transaktionen sorgt. Kein Produkt unterstützt explizite Algorithmen zur föderierten Concurrency Control; stattdessen verlassen sich alle auf die Concurrency-Control-Komponenten der beteiligten Datenbanksysteme. Datajoiner zum Beispiel hat zwar einen eigenen Sperrmechanismus, der aber ausschließlich für seine eigene, nur zu internen Zwecken verwendete Datenbank benutzt wird [IBM98]. Oracle warnt sogar davor, dass die Konsistenz der verteilten Daten in Föderationen, an denen der Microsoft SQL Server beteiligt ist, nicht mehr garantiert werden kann [Orac96], während dies in verteilten Datenbanken möglich ist, die ausschließlich aus Oracle-Instanzen bestehen [Orac99a].

Einer der Gründe für die geringe Verbreitung der Algorithmen zur Sicherstellung globaler Serialisierbarkeit ist, dass sie starke Anforderungen an die Komponentensysteme stellen, die an der Föderation teilnehmen dürfen. Alle bekannten Verfahren erfordern mindestens, dass die lokalen Systeme serialisierbare Schedules erzeugen; viele erlauben

sogar nur Schedules mit zusätzlichen Eigenschaften, die nur wenige praktische Systeme haben. Im Gegensatz dazu verwenden reale Anwendungsprogrammen häufig nicht Serialisierbarkeit, sondern nutzen sogenannte *Isolation Levels* wie das verbreitete "Read Committed", das bei allen in der Praxis relevanten Datenbanksystemen der Defaultmodus ist. Solche Optimierungsmöglichkeiten erlauben für viele praktische Anwendungen eine größere Leistung, können aber keine serialisierbaren Schedules mehr garantieren, so dass ein Teil der Konsistenz der Datenbank verloren gehen kann. Für viele Anwendungen spielt dies aber keine Rolle, weil sie entweder ohnehin nur approximative Daten berechnen oder weil man bereit ist, für den möglichen Performancegewinn gewisse Inkonsistenzen in Kauf zu nehmen. In der Literatur hat man die praktische Bedeutung von Isolation Levels erst in den letzten Jahren entdeckt; erst seit dieser Zeit gibt es Arbeiten, die formale Definitionen für Isolation Levels aufstellen und Eigenschaften daraus ableiten [ALO00, ABJ97, BLL00, FLO+99, SWW+99]. Auf dieser Basis kann man dann Aussagen über die Anwendbarkeit von Isolation Levels für konkrete Problemstellungen machen [OOSF00].

Darüber hinaus gibt es Datenbanksysteme, etwa das weitverbreitete System Oracle [Orac99c], die auch im stärksten verfügbaren Isolation Level keine serialisierbaren Schedules sicherstellen können. Oracle verwendet den Isolation Level *Snapshot Isolation*, der zwar für Transaktionen, die nur lesend auf die Datenbank zugreifen, Serialisierbarkeit gewährleistet, nicht aber für Transaktionen, die die Datenbank auch ändern. Die bekannten Verfahren für globale Serialisierbarkeit können nicht ohne weiteres angewandt werden, wenn ein solches System an der Föderation teilnimmt. Um die Konsistenz der föderierten Daten unter allen Umständen sicherzustellen, müssen daher neue Verfahren entwickelt werden, die auch mit lokalen Isolation Levels umgehen können.

1.3 Beitrag und Überblick über die Arbeit

Im folgenden Kapitel 2 diskutieren wir Atomarität und Serialisierbarkeit von Transaktionen in zentralisierten Datenbanksystemen. Nach einer kurzen Übersicht über die Grundlagen von Einversions-, Mehrversions- und Mehrschichtenserialisierbarkeit in Abschnitt 2.2 vergleichen wir in Abschnitt 2.3 verschiedene Definitionen, die in der Literatur für Isolation Levels vorgeschlagen wurden. Wir entwickeln anschließend in Abschnitt 2.4 eine neuartige formale Charakterisierung für Snapshot Isolation, die insbesondere erlaubt, auch in solchen Datenbanken serialisierbare Schedules zu gewährleisten, die nur Snapshot Isolation sicherstellen. In Abschnitt 2.5 stellen wir zunächst etablierte Kriterien für die Rücksetzbarkeit von Schedules vor, anschließend betrachten wir erstmals auch die Rücksetzbarkeit von Mehrversionenschedules und den Einfluss von Isolation Levels auf die Rücksetzbarkeit von Schedules. Den Abschluss dieses Kapitels bildet eine kurze Übersicht über die Verfahren zur Gewährleistung von Atomarität und Serialisierbarkeit in den wichtigsten kommerziellen Datenbanksystemen in Abschnitt 2.6.

In Kapitel 3 befassen wir uns mit Atomarität und Serialisierbarkeit von föderierten Transaktionen. Nach einer kurzen Einführung in die Problematik stellen wir in Abschnitt 3.3 etablierte Verfahren zur Gewährleistung von Atomarität vor und vergleichen ihre Einsetzbarkeit. Im folgenden Abschnitt 3.4 kategorisieren wir Techniken, die in der Literatur vorgeschlagen wurden, und bewerten sie im Hinblick auf ihre Leistungsfähigkeit und Einsetzbarkeit. In Abschnitt 3.5 stellen wir die Unterstützung für Transaktionen in zwei kommerziellen Systemen zur Föderation heterogener Datenquellen vor. Auf der

Basis unserer Ergebnisse aus Kapitel 2 untersuchen wir dann in Abschnitt 3.6, welche etablierten Algorithmen unter lokaler Snapshot Isolation noch einsetzbar sind, und diskutieren mögliche Variationen, um die globale Korrektheit sicherstellen zu können. Wir konzentrieren uns dabei auf die verschiedenen Varianten der Tickettechnik von Georgakopoulos et al. und auf Mehrschichtentransaktionen von Weikum et al. Anschließend entwickeln wir auf der Basis unserer Charakterisierung von Snapshot Isolation einen Algorithmus zur föderierten Concurrency-Control, der beweisbar auch dann die korrekte Ausführung föderierter Transaktionen gewährleistet, wenn einige der beteiligten Datenbanksysteme nur eingeschränkte Isolation Levels unterstützen. In Abschnitt 3.7 diskutieren wir dann die Verwendbarkeit eingeschränkter Isolation Levels für föderierte Transaktionen und zeigen, dass sich die Isolation Levels aus dem SQL-Standard der ANSI auf föderierte Transaktionen übertragen lassen. Wiederum aufbauend auf unseren theoretischen Ergebnissen aus Kapitel 2 stellen wir anschließend in Abschnitt 3.8 zwei Algorithmen vor, die beweisbar Snapshot Isolation für föderierte Transaktionen sicherstellen, wenn alle Komponentensysteme selbst Snapshot Isolation garantieren.

Alle Algorithmen, die wir in dieser Arbeit entwickelt haben, wurden in ein Prototypsystem integriert, das auf der Basis des existierenden föderierten Datenbanksystems *VHDBS* entwickelt wurde. In Kapitel 4 beschreiben wir zunächst in Abschnitten 4.1 Konzepte und wichtige Implementierungsdetails des VHDBS-Systems und stellen anschließend in Abschnitt 4.2 Grundlagen und Probleme des Object Transaction Services der OMG vor, der im Prototyp zur Realisierung des Zweiphasen-Commitprotokolls verwendet wird. Die Integration der Transaktionsverwaltung TraFIC in das VHDBS-System ist Gegenstand der folgenden Abschnitte 4.3 und 4.4. In den Abschnitten 4.5 und 4.6 stellen wir ausführlich die Implementierung der Komponenten der Transaktionsverwaltung vor.

Anhand dieses Prototyps untersuchen wir in Kapitel 5 experimentell die Leistungsfähigkeit der Algorithmen, die wir entwickelt haben. Wir stellen dazu in Abschnitt 5.1 zunächst eine Suite von Benchmarkprogrammen vor, die quantitative Urteile über die Performancepotentiale der einzelnen Strategien erlauben. Umfassende Messungen der Strategien zur Gewährleistung globaler Serialisierbarkeit in Abschnitt 5.2 und zur Gewährleistung globaler Snapshot Isolation in Abschnitt 5.3 zeigen, dass die neu entwickelten Strategien für den praktischen Einsatz ausreichend leistungsfähig sind.

Spezielle Aspekte dieser Arbeit wurden auf wissenschaftlichen Konferenzen präsentiert. [SWW+99] diskutiert die graphische Charakterisierung von Snapshot Isolation und skizziert erste Algorithmen für eine globale Concurrency Control, [SW00a] und [SW00b] entwickeln diese Algorithmen weiter und geben erste Ergebnisse der Benchmarks wieder. [SW99b] schließlich gibt einen Überblick über die Implementierung der Transaktionsverwaltung.

2 Transaktionen in zentralisierten Datenbanksystemen

Transaktionen spielen in der Datenbankwelt eine wichtige Rolle. Sie kapseln zusammengehörende Abläufe, etwa das Ausführen einer Überweisung oder das Buchen eines Fluges, zu einer Ausführungseinheit, die als Ganzes erfolgreich beendet oder vorher abgebrochen werden kann. Datenbanksysteme gewährleisten für Transaktionen die folgenden vier besonderen Eigenschaften, die sogenannten *ACID-Eigenschaften*:

- *Atomarität*: Nach dem erfolgreichen Beenden einer Transaktion werden alle Änderungen, die sie in der Datenbank gemacht hat, persistent gemacht, oder überhaupt keine. Wird eine Transaktion abgebrochen, hat sie keinen Effekt auf die Datenbank.
- *Konsistenzerhaltung*: Transaktionen erhalten Konsistenzbedingungen, die auf den Daten einer Datenbank definiert sind. Man fordert dazu, dass Transaktionen die Datenbank von einem konsistenten Zustand in einen anderen überführen. Das System kann über aktive Elemente wie Trigger und Constraints die Einhaltung einiger Typen von Konsistenzbedingungen sicherstellen.
- *Isolation*: Parallel laufende Transaktionen verhalten sich so, als wären sie alleine im System; insbesondere sehen sie keine partiellen Effekte der anderen Transaktionen.
- *Dauerhaftigkeit*: Änderungen von erfolgreich beendeten Transaktionen überleben Fehlerzustände des Datenbanksystems, wie Softwarefehler und Hardwareschäden. Datenbanksysteme verwenden dazu zum Beispiel Spiegelplatten, Backups von Logdateien und von Festplatten.

Wir beschränken uns in diesem Kapitel auf Aspekte der Gewährleistung von Atomarität und Isolation von Transaktionen in zentralisierten Datenbanksystemen, die wir später bei der Betrachtung von föderierten Transaktionen benötigen. Wir stellen dazu zunächst im folgenden Abschnitt 2.1 das Ausführungsmodell vor, auf das wir uns in dieser Arbeit berufen, und geben wichtige, grundlegende Definitionen an. Anschließend stellen wir in Abschnitt 2.2 relativ kurz verschiedene existierende Konzepte zur Gewährleistung von Serialisierbarkeit vor, insbesondere Konflikt- und Mehrversionsserialisierbarkeit sowie Mehrschichtentransaktionen. In den folgenden Abschnitten befassen wir uns mit Isolation Levels, einer Möglichkeit zur Performancesteigerung unter partiellem Verlust von Konsistenzbewahrung, und stellen neuartige Konzepte zu Snapshot Isolation, einem in der Praxis verbreiteten Isolation Level, vor. In Abschnitt 2.5 beschäftigen wir uns mit der Gewährleistung der Atomarität von Transaktionen und berücksichtigen dabei erstmals auch Mehrversionenschedules sowie Isolation Levels. Den Abschluss dieses Kapitels bildet eine kurze Vorstellung über Transaktionsmechanismen in verbreiteten Datenbanksystemen in Abschnitt 2.6.

2.1 Begriffe und Definitionen

Eine Datenbank besteht in unserem Ausführungsmodell aus einer Menge von atomaren Objekten, zum Beispiel Seiten oder Tupeln, die von einem Datenbanksystem verwaltet werden. Auf diese Objekte können Anwendungen lesend oder schreibend zugreifen. Wird Objekt x gelesen, schreiben wir dafür kurz *r(x)*, für eine Schreiboperation schreiben wir *w(x)*.

Anwendungen werden als Programme in einer Programmiersprache spezifiziert. Wir nehmen vereinfachend an, dass ein solches Programm alle Aktionen auf der Datenbank in einer Transaktion kapselt; wir sprechen dann von einem *Transaktionsprogramm*. Wird ein solches Programm ausgeführt, greift es lesend und schreibend auf die Datenbank zu. Die Operationen, die dabei ausgeführt werden, werden zu einer *Transaktion* zusammengefasst:

Definition 2.1: (Transaktion)

Eine *Transaktion* t_i ist eine Abfolge, d.h. eine totale Ordnung <, von Lese- und Schreiboperationen auf Objekten in einer Datenbank. Wir kennzeichnen die Operationen von t_i mit dem Index *i*; die Operation $r_i(x)$ ist eine Leseoperation von t_i auf Objekt x, $w_i(x)$ eine Schreiboperation. Zusätzlich enthält eine Transaktion entweder eine *Commit-Aktion* C_i oder eine *Abort-Aktion* A_i, die bezüglich < nach allen übrigen Operationen der Transaktion angeordnet sind. Sie enthält außerdem eine Pseudooperation B_i, die den Beginn der Transaktion markiert und bezüglich < vor allen übrigen Aktionen dieser Transaktion angeordnet ist.

Die Menge der Objekte, die eine Transaktion t_i liest, bezeichnen wir als *Readset von* t_i oder kurz $RS(t_i)$, analog heißt die Menge der Objekte, die t_i schreibt, *Writeset von* t_i oder kurz $WS(t_i)$. ■

Wenn wir im folgenden Transaktionen darstellen, werden wir zur Vereinfachung die Pseudooperation B_i in der Regel nicht explizit aufführen, wenn ihre exakte Position nicht notwendig ist.

Oft werden Transaktionen als partielle Ordnung von Lese- und Schreiboperationen definiert, z.B. in [BHG87] und [WV01]. Wir beschränken uns auf totale Ordnungen, da wir von Transaktionsprogrammen ausgehen, die sequentiell abgearbeitet werden. Die üblichen Anwendungen in der Praxis erfüllen dies, so dass diese Beschränkung keine harte Einschränkung darstellt.

Zur Vereinfachung der Argumentation in den folgenden Abschnitten vereinbaren wir zwei Konventionen für erlaubte Transaktionen:

- Jede Transaktion darf ein Objekt höchstens einmal lesen und schreiben.
- Wenn eine Transaktion ein Objekt schreibt, so muss sie es vorher lesen.

Keine dieser beiden Konventionen stellt eine harte Einschränkung für die Ausführungen dar, die wir modellieren können. In realen Ausführungen geht dem Ändern eines Objektes üblicherweise immer ein Lesen desselben Objekts voran. (Auch beim Einfügen eines neuen Objektes muss die Seite, auf der das Objekt schließlich gespeichert wird, zunächst gelesen werden, um das Objekt platzieren zu können.) Mehrfache Lese- und Schreiboperationen können durch temporäre Variablen des Transaktionsprogramms vermieden werden. Eine erneute Leseoperation kann dann auf die gespeicherte Kopie zugreifen, und die Schreiboperationen werden in einer einzigen am Ende der Transaktion gebündelt.

Werden mehrere Transaktionen ausgeführt, ergibt sich eine Reihenfolge, in der die Operationen (eventuell verschränkt) ausgeführt werden:

Definition 2.2: **(Historie, Schedule)**

Eine *Historie* einer Transaktionsmenge T={t_1, ...} ist eine Abfolge, d. h. eine totale Ordnung <, aller Aktionen aller Transaktionen in T, bei der die vorgegebene Ordnung der Aktionen innerhalb der Transaktionen erhalten bleibt.

Die Menge aller Operationen, die in einer Historie h enthalten sind, heißt *op(h)*. Die Menge aller Transaktionen in h, die mit Abort beendet werden, heißt *abort(h)*, die Menge aller Transaktionen, die mit Commit beendet werden, *commit(h)*.

Die *Projektion* einer Historie h auf eine Teilmenge M der in ihr vorkommenden Transaktionen $t_1..t_n$ beschränkt die Historie auf die Operationen der Transaktionen in M. Insbesondere ist *CP(h)* die Projektion von h auf commit(h).

Ein Präfix einer Historie heißt *Schedule*. Die Menge aller Transaktionen eines Schedules s, die weder mit Commit noch mit Abort beendet wurden (und somit noch aktiv sind), heißt *active(s)*. ■

Da jede Historie ein Präfix von sich selbst ist, ist sie gleichzeitig ein Schedule. Wir sprechen daher im folgenden nur von Schedules und stellen besonders heraus, wenn sich Aussagen nur auf Historien beziehen.

Transaktionen arbeiten in der Regel nicht auf einer leeren Datenbank, sondern mit Objekten, die frühere Transaktionen geschrieben haben. Wir modellieren dies durch eine initiale Transaktion t_0, die wir jedem Schedule hinzufügen und die bezüglich der Ordnung < vor allen "richtigen" Transaktionen angeordnet ist. Sie enthält für jedes Objekt, das im Schedule gelesen oder geschrieben wird, eine initiale Schreiboperation, die den Wert des Objektes zu Beginn der Ausführung setzt. Analog modellieren wir den Endzustand der Datenbank nach Ausführung des Schedules durch eine finale Transaktion t_∞, die alle Objekte liest, die während der Ausführung des Schedules verändert wurden. Beide Transaktionen schreiben wir in der Regel nur dann explizit hin, wenn auf sie Bezug genommen wird, andernfalls lassen wir sie weg.

Innerhalb eines Schedules lesen Transaktionen Werte, die frühere Transaktionen geschrieben haben. Man fasst die Information, welche Transaktion den Wert eines Objektes geschrieben hat, den eine andere Transaktion liest, in der *Liest-von-Relation* zusammen:

Definition 2.3: (Liest-von-Relation)

Sei s ein Schedule, und sei $r_j(x) \in op(s)$ eine Operation in s. Wir sagen, t_j *liest x von* t_i, $i \neq j$, wenn $w_i(x)$ die letzte Schreiboperation in s ist, für die $w_i(x) < r_j(x)$ gilt.

Wir definieren die *Liest-von-Relation RF(s) von s* als

RF(s) := { (t_i,x,t_j) | t_j liest x von t_i} ■

Eine besondere Klasse von Schedules sind solche, bei denen die Transaktionen streng nacheinander ausgeführt werden, sogenannte serielle Schedules:

Definition 2.4: (serieller Schedule)

Ein Schedule heißt seriell, wenn für je zwei verschiedene Transaktionen t_i und t_j, die im Schedule vorkommen, alle Operationen von t_i entweder vor oder nach allen Operationen von t_j stehen. ■

Sie bilden die Basis für die theoretische Betrachtung einer korrekten, verschränkten Ausführung von Transaktionen, zu der wir im nächsten Abschnitt kommen.

2.2 Traditionelles Konzept der Serialisierbarkeit

2.2.1 Motivation

Ein Ziel, dass wir in jedem Fall gewährleisten wollen, ist die Erhaltung der Konsistenz der Datenbank. Wir setzen voraus, dass jede Transaktion für sich betrachtet, d.h. wenn sie alleine ausgeführt wird, die Konsistenz erhält. Wenn mehrere Transaktionen ineinander verschränkt ausgeführt werden, kann die Konsistenz verletzt werden; wir wollen nur solche Ausführungen zulassen, die konsistenzerhaltend sind. Offensichtlich ist eine Ausführung korrekt, bei der alle Transaktionen sequentiell nacheinander ausgeführt werden (also ohne Schachtelung). Es liegt daher nahe, die geschachtelte Ausführung von Transaktionen dann als korrekt zu bezeichnen, wenn sie die gleichen Auswirkungen wie eine serielle Ausführung der Transaktionen hat. Man spricht dann von *Äquivalenz* der beiden Ausführungen. Ein Schedule, der äquivalent zu einem seriellen Schedule ist, heißt *serialisierbar*; die Reihenfolge der Transaktionen im äquivalenten seriellen Schedule wird *Serialisierungsreihenfolge* genannt.

Verschiedene Äquivalenzbegriffe führen auf verschiedene Klassen von „guten“ Schedules. In den folgenden Abschnitten stellen wir einige davon kurz vor. Wir beschränken uns dabei auf die Klassen, die in der Praxis (und auch im später vorgestellten Prototypen) eingesetzt werden. Im Gegensatz zu anderen, umfassenderen Klassen wie FSR (final-state-serialisierbare Schedules, [Papa79,BSW79]) und VSR (view-serialisierbare Schedules, [Yanna84]) lässt sich in polynomieller Zeit prüfen, ob ein Schedule zu einer solchen Klasse gehört. Außerdem gibt es effiziente Scheduling-Algorithmen, die solche Scheduleklassen erzeugen.

Wir nehmen zunächst an, dass alle Transaktionen in einem Schedule schließlich mit Commit beendet werden. Sollten in einem tatsächlichen Schedule s Transaktionsabbrüche oder nicht beendete Transaktionen vorkommen, betrachten wir stattdessen CP(s). Erweiterungen auf Schedules mit zurückgesetzten Transaktionen diskutieren wir, analog zum Vorgehen in [WV01], in Abschnitt 2.5.

2.2.2 Konfliktserialisierbarkeit

Eine in der Praxis weit verbreitete Klasse von serialisierbaren Schedules bilden die konfliktserialisierbaren Schedules. Zur Definition von Serialisierbarkeit wird dazu zunächst der Begriff des Konflikts eingeführt: Zwei Operationen auf demselben Objekt x stehen *in Konflikt*, wenn mindestens eine von ihnen eine Schreiboperation ist. Die Reihenfolge dieser Operationen darf sich in einem *konfliktäquivalenten Schedule* nicht ändern, da sich sonst die Semantik des Schedules verändern könnte. Dies führt unmittelbar zur Definition von konfliktserialisierbaren Schedules:

Definition 2.5: (Konfliktäquivalenz, CSR)

Zwei Schedules heißen *konfliktäquivalent*, wenn sie dieselben Transaktionen und Operationen enthalten und die Reihenfolge von Operationen, die in Konflikt stehen, in beiden Schedules gleich sind.

Ein Schedule s heißt *konfliktserialisierbar*, wenn CP(s) konfliktäquivalent zu einem seriellen Schedule ist. Die Klasse aller konfliktserialisierbaren Schedules heißt *CSR*. ■

Ein großer Vorteil dieses Kriteriums ist es, dass sich die Zugehörigkeit zu CSR in polynomieller Zeit überprüfen lässt, so dass praxistaugliche Verfahren möglich sind. Man kann dazu etwa den *Konfliktgraphen CG(s)* eines Schedules s verwenden, der die Transaktionen im Schedule als Knoten hat und genau dann eine Kante von Transaktion t_i nach Transaktion t_j, wenn eine Operation p von t_i mit einer Operation q von t_j in Konflikt steht und $p < q$ gilt. Die wichtige Äquivalenzaussage ist dann die folgende:

Theorem 2.1: (Charakterisierung von CSR)

Ein Schedule s ist genau dann in CSR, wenn sein Konfliktgraph CG(s) azyklisch ist. ■

Besonders im Falle verteilter Transaktionen sind zwei Teilklassen von CSR wichtig, die wir im nächsten Kapitel noch verwenden werden:

Definition 2.6: (OCSR, COCSR)

Ein Schedule s heißt

- *ordnungserhaltend konfliktserialisierbar*, wenn je zwei Transaktionen t_i und t_j, die in s vollständig nacheinander ablaufen, in einem konfliktäquivalenten seriellen Schedule die gleiche Reihenfolge behalten. Die Klasse aller ordnungserhaltend konfliktserialisierbaren Schedules heißt *OCSR*.
- *commit-ordnungserhaltend konfliktserialisierbar*, wenn die Reihenfolge der Transaktionen in einem konfliktäquivalenten seriellen Schedule der Reihenfolge der Commitoperationen in s entspricht. Die Klasse aller

commit-ordnungserhaltenden konfliktserialisierbaren Schedules heißt *COCSR*. ■

Algorithmen, die auf dem bereits vorgestellten Konfliktgraphen basieren, haben wegen des damit verbundenen Verwaltungsoverheads in der Praxis wenig Bedeutung erzielt. Statt dessen verwenden viele reale Systeme *Sperrprotokolle*, d.h., jede Transaktion sperrt ein Objekt, bevor sie darauf zugreift, und gibt es irgendwann nach dem Zugriff wieder frei. Dabei werden verschiedene *Sperrmodi* unterschieden, im allgemeinen mindestens ein *exklusiver* Sperrmodus, der das alleinige Zugriffsrecht zum Ändern eines Objekts sicherstellt, und ein *gemeinsamer* Sperrmodus, der mehreren Transaktionen das Lesen eines Objekts ermöglicht. Fordert eine Transaktion eine Sperre an, wenn eine andere Transaktion bereits eine Sperre auf dem gleichen Objekt in einem nicht verträglichen Modus hält (z.B. exklusiv vs. gemeinsam), so muss sie warten, bis alle nicht verträglichen Sperren freigegeben werden. Dabei können *Verklemmungen (Deadlocks)* auftreten, bei denen mehrere Transaktionen gegenseitig darauf warten, dass Sperren freigegeben werden. Solche Situationen müssen vom System erkannt und, z.B. durch Abbrechen einer der wartenden Transaktionen, behoben werden. Praktisch alle Systeme sperren zweiphasig: In der ersten Phase werden ausschließlich Sperren erworben (sog. *Wachstumsphase*), in der zweiten Phase werden die erworbenen Sperren wieder freigegeben, aber keine neuen dazu erworben. Insbesondere fordert also eine Transaktion höchstens einmal eine Sperre auf einem Objekt an; es ist aber möglich, eine vorhandene gemeinsame Sperre zu einer exklusiven Sperre zu erweitern.

Dieser Sperralgorithmus ist als *Zweiphasensperren (2PL)* bekannt. Man kann zeigen, dass 2PL nur konfliktserialisierbare Schedules erzeugt, aber nicht die ganze Klasse CSR [WV01]. Halten alle Transaktionen ihre exklusiven Sperren bis zu ihrem Ende, spricht man von *striktem Zweiphasensperren (S2PL)*; geben sie alle Sperren erst an ihrem Ende frei, spricht man von *starkem 2PL (SS2PL)*. Starkes 2PL generiert ausschließlich commit-ordnungserhaltend konfliktserialisierbare Schedules [WV01].

2.2.3 Mehrversionenserialisierbarkeit

Die Konfliktserialisierbarkeit eines Schedules scheitert oft daran, dass eine andere Transaktion bereits das Objekt überschrieben hat, das eine Transaktion lesen will, und so ein Zyklus im Konfliktgraph entsteht, wie im folgenden Beispiel:

$$r_1(x)\ r_2(x)\ r_2(y)\ w_2(x)\ w_2(y)\ r_1(y)\ c_2\ c_1$$

Könnte t_1 noch den alten Wert von y lesen, so wäre der Schedule serialisierbar. In einem realen System wird dieser alte Wert sowieso in der Regel bis zum Ende von t_2 aufbewahrt werden müssen, um ihn im Falle des Abbruchs von t_2 wiederherstellen zu können. Es liegt daher nahe, eine transparente Versionierung einzuführen, also alte Versionen von Objekten aufzuheben und, unsichtbar für die Anwendung, jeder Leseoperation die "passende" Version zuzuordnen. Dies führt zu sogenannten Mehrversionenschedules:

Definition 2.7: **(Mehrversionenschedule)**

Ein *Mehrversionenschedule* einer Transaktionsmenge $T=\{t_1, ...\}$ ist ein Schedule mit einer zusätzlichen Versionsfunktion, die jeder Leseoperation

$r_i(x)$ im Schedule eine Schreiboperation $w_j(x)$ zuordnet, die bezüglich der Ordnung < vor der Leseoperation angeordnet ist. Die Leseoperation wird dann auch als $r_i(x_j)$ geschrieben, wobei x_j die Version bezeichnet, die durch die Schreiboperation von Transaktion t_j erzeugt wurde. ■

Die Liest-von-Relation eines Mehrversionenschedules lässt sich aus den Leseoperationen ablesen, die er enthält. Kommt z.B. die Operation $r_j(x_i)$ im Schedule vor, so liest die Transaktion t_j x von t_i.

Die Schedules, die wir im letzten Abschnitt betrachtet haben, lassen sich als Spezialfall von Mehrversionenschedules darstellen, nämlich als Schedules, deren Versionsfunktion jeder Leseoperation immer die jüngste der vorherigen Schreiboperationen auf diesem Objekt zuordnet. Diese Schedules nutzen also nur eine Version eines Objektes:

Definition 2.8: **(Einversionenschedule)**

Ein *Einversionenschedule* einer Transaktionsmenge T={t_1, ...} ist ein Mehrversionenschedule, dessen Versionsfunktion jede Leseoperation $r_i(x)$ auf die jüngste vorhergehende Schreiboperation $w_j(x)$ abbildet (d.h., $w_j(x) < r_i(x)$ und es ist keine andere Operation dazwischen, die x verändert). ■

Es ist nun naheliegend, Mehrversionenschedules dann als korrekt anzusehen, wenn sie äquivalent zu einem seriellen Einversionenschedule sind, also zu einem Schedule, in dem die Versionierung keine Rolle mehr spielt [BHG87, Pa86]. Die dabei verwendete Äquivalenz ist die View-Äquivalenz, d.h. die Liest-von-Relationen der beiden Schedules müssen identisch sein:

Definition 2.9: **(MVSR)**

Ein Mehrversionenschedule s ist *mehrversions-serialisierbar*, wenn CP(s) view-äquivalent zu einem seriellen Einversionenschedule ist. Die Klasse aller mehrversions-serialisierbaren Schedules heißt *MVSR*. ■

Ein wesentliches Hilfsmittel, um zu prüfen, ob ein gegebener Schedule zu MVSR gehört, ist eine Graphkonstruktion, der sogenannte *Mehrversions-Serialisierungsgraph*:

Definition 2.10: **(Versionsordnung, MVSG)**

Für einen gegebenen Mehrversionenschedule s und ein gegebenes Objekt x ist eine *Versionsordnung $\ll$ für x* eine totale Ordnung der Versionen von x in s. Eine *Versionsordnung für s* ist die Vereinigung der Versionsordnungen für alle Objekte in s.

Für einen gegebenen Mehrversionenschedule s und eine Versionsordnung $\ll$ für s ist der *Mehrversions-Serialisierungsgraph für s und $\ll$, MVSG(s,$\ll$),* der gerichtete Graph mit den mit Commit beendeten Transaktionen in s als Knoten und den folgenden Kanten:

- Zu jeder Operation $r_j(x_i)$ im Schedule gibt es eine Kante $t_i \rightarrow t_j$ (*WR-Kante*).
- Zu jedem Paar $r_k(x_j)$ und $w_i(x_i)$ von Operationen mit paarweise verschiedenen i, j und k gibt es eine Kante
  (i) $t_i \rightarrow t_j$, falls $x_i \ll x_j$ (*WW-Kante*),
  (ii) $t_k \rightarrow t_i$, falls $x_j \ll x_i$ (*RW-Kante*). ■

Das folgende zentrale Theorem aus [BHG87] charakterisiert die Schedules, die zur Klasse MVSR gehören, mit Hilfe der zugehörigen MVSG. Es ist die Basis für die Korrektheitsbeweise vieler praktischer Concurrency-Control-Protokolle, die Mehrversionsschedules verwenden:

Theorem 2.2: **(Charakterisierung von MVSR)**

Ein Mehrversionenschedule s gehört genau dann zur Klasse MVSR, wenn es eine Versionsordnung $\ll$ gibt, so dass der zugehörige Graph MVSG(s,$\ll$) azyklisch ist. ■

Das Theorem selbst eignet sich nicht, um algorithmisch die Zugehörigkeit eines gegebenen Schedules zu MVSR zu prüfen, da es exponentiell viele verschiedene Versionsordnungen gibt, die man alle durchprobieren müsste. Es kann daher, ähnlich wie im Einversionsfall, sinnvoll sein, nur eine Teilklasse von MVSR zu betrachten, für die die Zugehörigkeit eines Schedules in polynomieller Zeit entschieden werden kann. Wir definieren dazu die Klasse der *mehrversions-konfliktserialisierbaren* Schedules:

Definition 2.11: (MCSR)

Ein Mehrversionenschedule s ist *mehrversions-konfliktserialisierbar*, wenn CP(s) durch endlich viele Anwendungen der folgenden Kommutativitätsregel in einen seriellen Einversionenschedule transformiert werden kann:

Zwei Operationen p,q$\in$ op(CP(s)), die bezüglich < direkt benachbart sind mit p<q, können genau dann vertauscht werden, wenn p keine Lese- oder q keine Schreiboperation auf dem gleichen Objekt ist.

Die Klasse aller mehrversions-konfliktserialisierbaren Schedules heißt *MCSR*. ■

Während bei MVSR die Versionsfunktion von vornherein feststeht, ergibt sie sich bei MCSR als Resultat der wiederholten Anwendung der Kommutativitätsregel: Die (triviale) Versionsfunktion des schließlich erzeugten seriellen Einversionenschedules wird auf den Ausgangsschedule übertragen.

Wendet man die Regeln von MCSR dagegen auf Schedules an, die bereits eine Versionsfunktion besitzen, können durch das Vertauschen der Operationen ungültige Schedules entstehen, die insbesondere eine unzulässige Versionsfunktion haben: Wird z.B. die Schreiboperation, von der eine Leseoperation liest, hinter die Leseoperation verschoben, liest sie eine Version, die erst später erzeugt wird. Natürlich sind nur die

schoben, liest sie eine Version, die erst später erzeugt wird. Natürlich sind nur die Schedules Element von MCSR, bei denen eine wiederholte Vertauschung nach der Kommutativitätsregel schließlich zu einem gültigen Schedule führt.

Hintergrund für die Kommutativitätsregel ist der Konfliktbegriff, den MCSR verwendet; es handelt sich um einen völlig anderen als bei als CSR. Bei MCSR stehen lediglich Operationspaare $r_i(x)$, $w_j(x)$ in Konflikt, wenn sie in dieser Reihenfolge stehen, wir sprechen dann von einem *Mehrversionskonflikt zwischen t_i und t_j*. Hintergrund dafür ist, dass die Leseoperation $r_i(x)$ mehr Versionen "zur Auswahl" hat, wenn die Schreiboperation davor kommutiert wird.

Im allgemeinen ist MCSR eine echte Teilklasse vom MVSR, d.h. es gibt Schedules, die zwar in MVSR sind, aber nicht in MCSR. Für unser spezielles Ausführungsmodell wird jedoch in [Papa86] gezeigt, dass die beiden Klassen identisch sind:

Theorem 2.3: **(Äquivalenz MCSR und MVSR)**

Wenn jeder Schreibaktion einer Transaktion eine Leseaktion auf dem gleichen Objekt vorausgeht, gilt MCSR=MVSR. ■

Analog zur Charakterisierung von CSR lässt sich auch die Zugehörigkeit eines Schedules zu MCSR über Zyklenfreiheit eines Graphen charakterisieren, der die Transaktionen als Knoten hat und genau dann eine Kante von Transaktion t_i nach Transaktion t_j, wenn t_i und t_j einen Mehrversionskonflikt haben. Daraus ergibt sich sofort das folgende Korollar, da der Graph zu einem Schedule nur kleiner werden kann, wenn man Operationen vom Ende des Schedules entfernt, also einen Präfix des Schedules betrachtet:

Korollar 2.1:

MCSR ist präfix-abgeschlossen. ■

Basierend auf diesen theoretischen Ansätzen wurde eine Reihe von Protokollen entwickelt, z.B. das *Mehrversionen-Zeitmarkenprotokoll MVTO*, das *Mehrversionen-Sperrprotokoll MV2PL* und *MVSGT*, das auf dem Mehrversions-Serialisierungsgraphen basiert; Details dazu findet man z.B. in [BHG87] und in [WV01]. Alle diese Protokolle haben aber in realen Systemen keinen Durchbruch erzielt. Ein in der Praxis dagegen recht weit verbreitetes Protokoll, das eine Teilmenge von MVSR erzeugt, ist *ROMV* (*Read-Only Multiversion Protocol*, [DuBou82, CFL+82]). Es handelt sich dabei um ein hybrides Protokoll, das den in der Praxis sehr häufigen Fall von reinen Lesetransaktionen besonders behandelt: Für sie verwendet das Protokoll ein Mehrversionsverfahren, das jeder Leseoperation die letzte Version zuordnet, die vor dem Beginn der Transaktion committed wurde. Insbesondere werden keinerlei Lesesperren angefordert oder sonstige Maßnahmen zur Concurrency Control für diesen Transaktionstyp durchgeführt. Dadurch können reine Lesetransaktionen sehr effizient behandelt werden und müssen nie auf Sperren warten. Für Transaktionen, die auch Objekte verändern, wird das übliche strikte zweiphasige Sperrverfahren für Lese- und Schreibaktionen verwendet, das

wir in Abschnitt 2.2.2 vorgestellt haben. Solche Transaktionen profitieren also überhaupt nicht von den zusätzlich vorhandenen Versionen.

2.2.4 Mehrschichtentransaktionen

Wir erweitern nun unser bisheriges Ausführungsmodell, indem wir Operationen nicht wie bisher auf Seiten bzw. Tupeln arbeiten lassen, sondern semantisch reichere Operationen auf "Objekten" betrachten. Solche Objekte können z.B. die verschiedenen Konten einer Bank mit den Operationen Einzahlen und Abbuchen, Zähler mit Inkrementieren und Dekrementieren oder auch Indexstrukturen des Datenbanksystems sein, die die Operationen Einfügen, Löschen und Suchen unterstützen. Eine solche Operation besteht, wenn sie schließlich auf der Datenbank ausgeführt wird, in der Regel aus mehreren einfacheren Operationen, bis schließlich Operationen auf Seitenebene folgen. Dies führt zu *Mehrschichtentransaktionen*, die die Operationen und ihre Unteroperationen als Baum darstellen, und *Mehrschichtenschedules*, die die verschränkte Ausführung von solcher Transaktionen beschreiben [WS84, Weik86, Weik91, WS92]. Die Schichten eines solchen Schedules werden auch als *Levels* L_i bezeichnet, die von unten nach oben, beginnend bei 0, durchnummeriert werden. Die Ausführung jeder Unteroperation wird in einer *Subtransaktion* gekapselt, die mit Commit beendet wird, nachdem die Operation erfolgreich ausgeführt wurde.

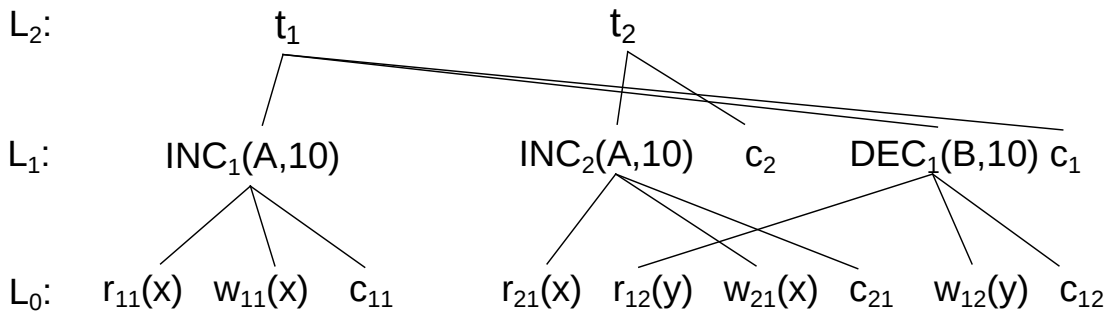

Abbildung 2.1 - Beispiel für einen Mehrversionenschedule

Abbildung 2.1 zeigt ein Beispiel für den Ablauf zweier Transaktionen im Mehrschichtenmodell. Auf der obersten Schicht sind dabei die Transaktionen selbst dargestellt. In der Mittelschicht stehen die Operationen, die die Transaktionen ausführen; dabei handelt es sich um semantisch reiche Operationen, in diesem Fall Inkrementieren (INC) und Dekrementieren (DEC) von Zählern. Die unterste Schicht enthält die Subtransaktionen auf der Seitenebene, die von den darüberliegenden Operationen ausgelöst werden. Sie werden unmittelbar nach der Ausführung der Operationen mit Commit beendet, daher können z.B. Sperren freigegeben werden, die sie gehalten haben, und werden nicht bis zum Ende der gesamten Transaktion gehalten. Die Parallelität in der unteren Schicht ist also potentiell wesentlich höher.

An diesem Beispiel sieht man sofort, dass die bisher entwickelten Korrektheitskriterien hier nicht ohne weiteres greifen: Der Teilschedule auf dem untersten Level L_0 ist konfliktserialisierbar, da es sich für das Datenbanksystem um drei voneinander unabhängige Transaktionen handelt, der äquivalente serielle Schedule ist $t_{11}t_{21}t_{12}$. Es ist aber unklar, ob man daraus die Korrektheit des Gesamtschedules folgern kann, da t_1 und t_2 auf der mittleren Schicht geschachtelt ausgeführt werden. Zur Lösung führt es, auch auf den

inneren Schichten Maßnahmen zur Concurrency Control zu verwenden. Man nutzt dazu typischerweise Zusatzwissen über die Semantik der ausgeführten Operationen auf den mittleren Schichten aus, um Informationen über die Kommutativität der Operationen zu gewinnen. Im Beispiel kann man aus Wissen über die Anwendung und die Implementierung der Operation ableiten, dass zwei INC-Operationen vertauschbar sind: Das Ergebnis bleibt unabhängig von der Reihenfolge ihrer Ausführung das gleiche.

Das theoretische Korrektheitskriterium für Mehrschichtenschedules ist *Baumreduzierbarkeit* [WV01], das ein konstruktives Kriterium auf Basis der Kommutativität von Operationen der einzelnen Levels darstellt. Man versucht dabei, kommutative Operationen auf Level L_{i-1} so zu vertauschen, dass die Operationen einer Aktion auf Level L_i vollständig isoliert stehen, d.h. dass zwischen den Operationen der Aktion keine Operationen anderer Subtransaktionen angeordnet sind. Die Operationen einer solchen isolierten Aktion, die dann einen Teilbaum des gesamten Schedules bilden, darf man dann aus dem Schedule entfernen. Aus der totalen Ordnung der Operationen auf dem untersten Level leitet man eine Ordnung der Operationen auf den Levels darüber ab, dabei ist die Ordnung auf den höheren Levels u.U. partiell, weil zwei Operationen auf einem höheren Level nur dann angeordnet werden können, wenn die zu ihnen gehörenden Operationen auf den unteren Levels vollständig geordnet sind. Ein Mehrschichtenschedule heißt dann *baumreduzierbar*, wenn man ihn mit diesen Regeln zu einer Anordnung seiner Wurzelknoten reduzieren kann. Die Schedules, die man so erhält, sind die, die man als korrekt ansieht.

Für praktische Systeme ist man aber weniger an Kriterien interessiert, die einen gegebenen Schedule überprüfen, sondern mehr an solchen, mit denen man sicherstellen kann, dass nur korrekte Schedules erzeugt werden. Man betrachtet dazu die Schedules auf den einzelnen Schichten, die sogenannten *Level-to-Level-Schedules*. Dann kann man zeigen, dass die Baumreduzierbarkeit aller erzeugten Schedules gewährleistet ist, wenn alle Scheduler ordnungserhaltend konfliktserialisierbare Schedules erzeugen. Insbesondere Scheduler, die stark zweiphasig sperren, gehören dazu. Selbst konfliktserialisierbare Schedules sind ausreichend, wenn man sicherstellen kann, dass *Konflikttreue* vorliegt, d.h. dass Operationen, die auf einer höheren Schicht nicht vertauschbar sind, auch auf der Schicht darunter mindestens einen Konflikt haben [WV01].

2.3 Eingeschränkte Korrektheitskriterien

2.3.1 Motivation

Serialisierbarkeit gewährleistet die vollständige Isolation von Transaktionen gegenüber parallel laufenden anderen Transaktionen. Um dies zu erreichen, sind Maßnahmen zur Concurrency Control wie die Verwaltung von Sperren oder Graphstrukturen notwendig, die oft Auswirkungen auf die erzielbare Performance aller Transaktionen haben. Viele existierende Anwendungen benötigen aber gar nicht wirklich Serialisierbarkeit, sie sind mit weniger starken Korrektheitskriterien zufrieden, die zudem in der Regel eine bessere Performance erlauben.

Solche eingeschränkten Kriterien können zum Beispiel verwendet werden, wenn man nur an approximativen Werten wie der Zahl der Verkäufe in einem Jahr interessiert ist. Man kann sie auch sinnvoll einsetzen, wenn man sicher weiß, dass keine Probleme mit parallel laufenden Transaktionen auftreten können. Andererseits gibt es auch viele An-

wendungsfälle, wo man auf korrekte, d.h. serialisierbare Ausführung angewiesen ist. Man kann zum Beispiel jeden Sitz in einem Flugzeug nur höchstens einmal vergeben, daher dürfen parallel laufende Transaktionen nicht den gleichen Sitzplatz buchen. Auch Anwendungen aus dem Bankbereich, die Geld transferieren oder Bewertungen anhand von aktuellen Vermögensständen abgeben, sind auf korrekte Daten angewiesen. Man muss sich also immer im Klaren darüber sein, ob die konkrete Anwendung eingeschränkte Korrektheit verträgt oder nicht.

Bereits 1976 wurde in [GLPT76] die Einführung von eingeschränkten Isolationsgraden, sogenannten *Isolation Levels*, vorgeschlagen. Dort wurden die verschiedenen Levels zum einen aus Systemsicht über das Anfordern und Freigeben von Lese- und Schreibsperren definiert, zum anderen durch das Vermeiden charakteristischer Phänomene. Die erste Variante der Definition lässt sich naturgemäß nicht ohne weiteres auf Scheduler anwenden, die nicht mit Sperren arbeiten. Im ANSI-SQL-Standard [ANSI92] wurden die Definition von Isolation Levels über die Vermeidung informell beschriebener, problematischer Phänomene übernommen. Berenson et al haben in [BBGM+95] gezeigt, dass diese Darstellung aber andere Klassen als die Definition über Sperren erzeugt, und eigene Verallgemeinerungen vorgeschlagen, die den sperrbasierten Ansätzen nahe kommen. In den folgenden Abschnitten stellen wir beide Ansätze vor und vergleichen sie. Erst in jüngerer Zeit hat es Ansätze gegeben, Isolation Levels auch für nichtsperrende Scheduler formal zu definieren; die dabei entstandenen Kriterien sind teilweise inkompatibel mit den ANSI-Definitionen [Adya99].

Die Auswahl des geeigneten Isolation Levels für eine konkrete Anwendung ist im allgemeinen nicht trivial: Wählt man einen zu konservativen Level, verschenkt man möglicherweise Performance; entscheidet man sich dagegen für einen zu schwachen Level, riskiert man inkorrekte und inkonsistente Daten. Bisher haben sich nur wenige Arbeiten mit diesem Auswahlverfahren beschäftigt, das wichtigste ist das IsoTest-Projekt von Patrick und Elizabeth O'Neil [OO97, OOSF00], in dem versucht wird, für gegebene Anwendungen den minimal notwendigen Isolation Level zu bestimmen, unter dem die Anwendung noch korrekt ausgeführt wird.

In diesem Abschnitt werden wir zum einen die Problematik des Einsatzes von Isolation Levels klarmachen, indem wir mögliche resultierende Inkonsistenzen aufzeigen. Zum anderen werden wir verschiedene existierende Definitionsmöglichkeiten für Isolation Levels vergleichen und, besonders im Hinblick auf reale Systeme, bewerten.

2.3.2 Definitionen für Isolation Levels

Wir stellen in diesem Abschnitt verschiedene Möglichkeiten zur Definition von Isolation Levels vor, die man in der Literatur finden kann, vergleichen sie miteinander und zeigen charakteristische Inkonsistenzen auf, die sich durch den Einsatz der Isolation Levels ergeben. Wir beginnen im folgenden Abschnitt mit den Definitionen im SQL92-Standard des ANSI [ANSI92], diskutieren anschließend den sperrbasierten Ansatz von Gray et. al. [GLPT76] und stellen dann einen Ansatz von Berenson et. al. [BBGM+95] vor, der die Definitionen der ANSI kritisch betrachtet. In Abschnitt 2.3.2.4 betrachten wir einen weiteren, nicht vom Standard erfassten Isolation Level, den man in realen Produkten finden kann. Wir erweitern anschließend die Definition von Isolation Levels auf Mehrversionen-Scheduler. Im letzten Abschnitt vergleichen wir die verschiedenen Isolation Levels qualitativ miteinander.

2.3.2.1 Definitionen über Vermeidung von Phänomenen

Da reale Systeme viele verschiedene Protokolle zur Concurrency Control verwenden, müssen allgemeine Definitionen für Isolation Levels unabhängig vom verwendeten Protokoll sein. Im ANSI-SQL92-Standard [ANSI92] führt man daher den Begriff des *Phänomens* ein, eine Folge von Operationen, die zu einem fehlerhaften Verhalten führt. Isolation Levels werden dann über die Vermeidung einer Teilmenge dieser Phänomene definiert, so dass die Definitionen prinzipiell für alle möglichen Scheduler passen. Im einzelnen werden im Standard die folgenden Phänomene beschrieben:

- *Dirty Read*

  Transaktion t_1 ändert den Wert von Objekt x. Bevor t_1 beendet wird, liest eine weitere Transaktion t_2 dann den neuen Wert von x. Wenn t_1 nun zurückgesetzt wird, hat t_2 einen Wert gelesen, der offiziell nie in der Datenbank gestanden hat.

- *Unrepeatable Read*

  Transaktion t_1 liest den Wert von Objekt x. Eine weitere Transaktion t_2 ändert oder löscht anschließend Objekt x und wird mit Commit beendet. Wenn t_1 nun versucht, den Wert von x erneut zu lesen, erhält sie den geänderten Wert bzw. stellt fest, dass das Objekt gelöscht wurde.

- *Phantom*

  Transaktion t_1 liest eine Menge von Objekten, die ein Suchprädikat P erfüllen. Eine andere Transaktion t_2 fügt anschließend neue Objekte ein, die das Suchprädikat erfüllen, und wird mit Commit beendet. Wenn t_1 nun die Query mit dem gleichen Suchprädikat erneut ausführt, erhält sie eine andere Ergebnismenge als beim ersten Mal zurück.

Basierend auf dem Vermeiden dieser Phänomene werden dann insgesamt vier Isolation Levels definiert:

- *Read Uncommitted*

  Dieser Isolation Level erlaubt alle drei beschriebenen Phänomene. Insbesondere können Transaktionen also Änderungen von Transaktionen sehen, die noch nicht beendet sind und daher später noch zurückgesetzt werden können. Um einen korrekten Abbruch zu gewährleisten, müsste dann auch die lesende Transaktion zurückgesetzt werden.

- *Read Committed*

  Unter diesem Isolation Level wird Dirty Read vermieden, d.h. jede Leseoperation sieht nur Daten, die von bereits mit Commit beendeten Transaktionen geschrieben wurden. Unrepeatable Reads und Phantome können aber auftreten. Für manche Anwendungen reicht dies aber aus, etwa für solche, die nur einen ungefähren Überblick über eine große Datenmenge erhalten wollen, bei dem es nicht auf absolut exakte Werte ankommt.

- *Repeatable Read*

  Zusätzlich zu Dirty Read wird unter diesem Isolation Level auch Unrepeatable Read verboten, d.h. wiederholtes Lesen eines Objektes durch eine Transaktion führt zu

den gleichen Ergebnissen, selbst wenn inzwischen eine andere Transaktion die gelesenen Werte überschrieben hat. Phantome können aber auftreten.

- *Serializability*

  Der ANSI-Standard fordert, dass dieser Isolation Level Serialisierbarkeit im üblichen Sinne gewährleisten soll. Gleichzeitig definiert er aber Serialisierbarkeit auch über die Vermeidung aller drei genannten Phänomene Dirty Reads, Unrepeatable Reads und Phantome. Man erlaubt damit aber auch nichtserialisierbare Schedules, wie wir im folgenden Abschnitt sehen werden.

In der folgenden Tabelle werden die Definitionen der vier Isolation Levels aus dem ANSI-SQL-Standard zusammengefasst:

Isolation Level	**Dirty Read**	**Unrepeatable Read**	**Phantom**
Read Uncommitted	Möglich	möglich	möglich
Read Committed	Unmöglich	möglich	möglich
Repeatable Read	Unmöglich	unmöglich	möglich
Serializable	Unmöglich	unmöglich	unmöglich

2.3.2.2 Definitionen für Sperrverfahren

Viele Systeme verwenden Sperrverfahren für die Concurrency Control, in der Regel striktes Zweiphasensperren, das konfliktserialisierbare Schedules erzeugt. Durch das Halten der Sperren bis zum Transaktionsende wird die mögliche Parallelität allerdings recht stark eingeschränkt, man könnte also durch frühzeitige Sperrfreigabe Leistung gewinnen, allerdings in der Regel auf Kosten der Korrektheit. Auf dieser Basis wurden die ersten Definitionen von eingeschränkten Korrektheitsgraden in [GLPT76] vorgenommen, die dort "Degrees of Consistency" genannt werden. Wir stellen diese Definitionen hier kurz vor, insbesondere unterscheiden wir dabei, ob Transaktionen Lese- und Schreibsperren auf Objekten und Prädikaten anfordern und wann sie sie wieder freigeben.

- *Degree 0*

  Transaktionen, die unter diesem Isolation Level laufen, erwerben lediglich kurze Schreibsperren für die Dauer der Schreiboperation, die die Atomarität der Schreiboperation sicherstellen, aber keine Lesesperren. Hier hat man praktisch keine Konsistenz mehr. Im ANSI-Standard hat dieser Level keine Entsprechung.

- *Degree 1*

  Transaktionen halten ihre Schreibsperren jetzt bis zum Ende der Transaktion, aber verwenden keine Lesesperren. Damit sind alle drei Phänomene aus dem ANSI-Standard möglich.

- *Degree 2*

  Transaktionen halten ihre Schreibsperren bis zum Ende der Transaktion und verwenden kurze Lesesperren für die Dauer einer Leseoperation. Weil eine Transaktion nun wegen des Sperrkonfliktes keine Objekte mehr lesen kann, die eine andere, ak-

tive Transaktion geändert hat, vermeidet dieser Level Dirty Reads, er entspricht damit dem Read Committed aus dem Standard.

- *Degree 3*

  Transaktionen verwenden striktes Zweiphasensperren. Die erzeugten Schedules sind damit konfliktserialisierbar.

Zwischen Degree 2 und 3 gibt es noch Platz für Variationen, die in [GLPT76] nicht betrachtet wurden. Insbesondere kann man einen Level definieren, der Unrepeatable Reads vermeidet:

- *Locking Repeatable Read*

  Transaktionen halten ihre Schreibsperren und die Lesesperren auf Objekten bis zum Ende der Transaktion. Außerdem sperren sie das Suchprädikat einer Query bis zum Ende der Ausführung der Query.

2.3.2.3 Definitionen nach Berenson et. al.

Im ANSI-SQL-Standard sind die Phänomene, die vermieden werden sollen, nicht formal, d.h. als Folge von Operationen auf Objekten, sondern nur in textueller Form definiert. Um über die Korrektheit von Schedules argumentieren zu können, müssen die Phänomene daher zunächst auf der Ebene von Lese- und Schreiboperationen formuliert werden. Berenson et. al. schlagen dazu in [BBGM+95] zwei verschiedene Möglichkeiten vor: Zunächst eine *enge Interpretation* der ANSI-Phänomene, die exakt die Operationssequenz aus der textuellen ANSI-Definition darstellt, die verboten wird, und zum anderen eine *verallgemeinerte Interpretation*, die eine weiter gefasste Operationssequenz verbietet, die den jeweiligen problematischen Ausgang haben *könnte*. Sie argumentieren, dass die Intention der ANSI-Definitionen die verallgemeinerte Variante ist, obwohl die enge Variante spezifiziert wurde. In der Tat stellt man fest, dass die verallgemeinerte Interpretation exakt der auf Sperren basierenden Definition entspricht, die wir im vorherigen Abschnitt dargestellt haben. Wir stellen nun beide Interpretationen für die Phänomene anhand der Operationssequenzen, die sie beschreiben, vor und zeigen die Unterschiede auf. Da die Sequenzen für die verallgemeinerte Interpretation immer Präfixe der Sequenzen für die enge Interpretation sind, heben wir den Teil für die verallgemeinerte Interpretation durch Fettdruck hervor.

- *Dirty Write*: $w_1(x)$ $w_2(x)$ (c_1 oder a_1, c_2 oder a_2 in beliebiger Reihenfolge)

  Dieses Phänomen taucht im ANSI-Standard überhaupt nicht auf. Praktisch ist es dagegen von großer Wichtigkeit, solche Ausführungen nicht zuzulassen, da sie große Probleme bei der Recovery verursachen, wie wir später in Abschnitt 2.5.1 noch zeigen werden. Zusätzlich können Inkonsistenzen auftreten, wie z.B. im folgenden Beispiel:

  $$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$$

  Sollte es eine Bedingung geben, die x und y betrifft (z.B. x+y>100), so kann diese durch diesen Schedule verletzt werden, obwohl t_1 und t_2 die Einhaltung der Bedingung prüfen, weil der Wert von x, der durch t_2 geschrieben wurde, aber der Wert von y, der durch t_1 geschrieben wurde, in der Datenbank bleibt. Verbietet man Dirty Writes, tritt dieses Problem nicht mehr auf, weil t_2 x erst nach dem Ende von t_1

schreiben darf und so nur die Werte, die von t_2 geschrieben wurden, in die Datenbank gelangen.

- *Dirty Read*: **$w_1(x)$ $r_2(x)$** (a_1,c_2 in beliebiger Reihenfolge)

  Im Sinne der ANSI-Definition tritt nur dann ein Problem auf, wenn t_1 nach dem Lesen durch t_2 abgebrochen wird. Praktisch gibt es aber auch dann Inkonsistenzen, wenn t_1 und t_2 erfolgreich enden, wie man am folgenden Beispielschedule sieht:

  $r_1(x)$ $w_1(x)$ $r_2(x)$ $r_2(y)$ c_2 $r_1(y)$ $w_1(y)$ c_1

  Die Transaktion t_2 sieht bereits den von t_1 geänderten Wert von x, aber den alten Wert von y; wenn es also eine Abhängigkeit von x und y gibt, könnte es sein, dass t_2 inkonsistente Werte sieht. Offenbar muss man tatsächlich die verallgemeinerte Interpretation fordern, um diese Art von Anomalie vermeiden zu können.

- *Unrepeatable Read*: **$r_1(x)$ $w_2(x)$** c_2 $r_1(x)$ c_1

  Hier sieht die ANSI-Definition das Problem im erneuten Lesen des Werts von x durch t_1, die dann einen anderen Wert sieht als vorher. (In unserem Ausführungsmodell kann dieser Fall nicht auftreten, da eine Transaktion jedes Objekt höchstens einmal lesen darf.) Auch hier kann man mit einem Beispiel motivieren, dass man besser die verallgemeinerte Interpretation fordern sollte:

  $r_1(x)$ $r_2(x)$ $w_2(x)$ $r_2(y)$ $w_2(y)$ c_2 $r_1(y)$

  Die Transaktion t_1 sieht noch den alten Wert von x, aber schon den von t_2 geänderten Wert von y. Wenn sie x noch einmal lesen würde, würde sie dieses Problem erkennen; das tut sie aber nicht, und damit ist dieser Schedule unter der engen Definition zulässig, obwohl er eine potentielle Inkonsistenz zeigt, nämlich z.B. eine Verletzung einer möglicherweise vorhandenen Invariante x+y>100.

- *Phantom*: **$r_1[P]$ $w_2(x \in P)$** c_2 $r_1[P]$ c_1

  Man kann dies als eine Art "Sonderfall" des Unrepeatable Read sehen, da t_1 wiederholt die Objekte liest, die das Prädikat P erfüllen. Anhand eines Beispiels kann man nun wieder zeigen, dass die enge Interpretation nicht alle Fälle abdeckt, die man als Phantome bezeichnen würde, während die verallgemeinerte Interpretation alle Fälle abdeckt:

  $r_1[P]$ $w_2(\text{insert } x \in P)$ $r_2(z)$ $w_2(z)$ c_2 $r_1(z)$ c_1

  Hier wertet t_1 das Prädikat P nur einmal aus, so dass die enge Interpretation hier kein Problem sieht. Tatsächlich kann es aber z.B. sein, dass in z die Gesamtzahl aller Objekte in der Datenbank gespeichert wird, die das Prädikat P erfüllen, z.B. die Zahl der Mitarbeiter einer Abteilung. Die Transaktion t_1 sieht also ein Objekt, nämlich x, "zu wenig". Nach der verallgemeinerten Interpretation ist dieser Schedule dagegen unzulässig.

Anhand der Beispiele erkennt man, dass es nicht ausreichend ist, ausschließlich exakt die Phänomene zu betrachten, die in den textuellen ANSI-Definitionen beschrieben sind, da man sonst viele Inkonsistenzen nicht ausschließen kann. Andererseits sind die verallgemeinerten Interpretationen, die von Berenson et. al. vorgeschlagen werden, in vielen Fällen zu restriktiv, da sie praktisch das verschränkte Ausführen von Lese- und Schreiboperationen ganz verbieten. Um Unrepeatable Reads zu vermeiden, wird bei-

spielsweise das Überschreiben eines Objektes so lange verboten, bis die lesende Transaktion beendet ist, obwohl in der großen Mehrzahl aller Ausführungen später keine Probleme auftreten werden. In [BBGM+95] zeigen Berenson et. al., dass dieser Ansatz für Einversionenschedules tatsächlich den Definitionen über Sperren entspricht. Unterschiede ergeben sich also höchstens bei Protokollen, die mehrere Versionen eines Objektes verwalten; diese Richtung verfolgen sie aber für die üblichen Isolation Levels nicht weiter.

Wir sprechen im folgenden von *ANSI Read Committed*, wenn wir den Isolation Level nach der engen Interpretation meinen, und von *verallgemeinertem Read Committed* oder einfach *Read Committed*, wenn wir die verallgemeinerte Interpretation meinen. Analoges gilt für Read Uncommitted und Repeatable Read.

2.3.2.4 Weitere Isolation Levels

Der ANSI-SQL-Standard definiert nur die Isolation Levels, die sich durch die Vermeidung der drei genannten Phänomene ergeben. Tatsächlich kann man aber zeigen, dass es weitere Phänomene gibt, die zu inkonsistenten Datenbankzuständen führen können, die aber selbst dann vorkommen können, wenn alle drei Phänomene aus dem Standard vermieden werden. In realen Systemen findet man einige Isolation Levels, die sich durch die Vermeidung dieser Phänomene auszeichnen. Wir stellen sie in diesem Abschnitt kurz vor.

Der Isolation Level *Cursor Stability* [Date90] vermeidet neben Dirty Writes und Dirty Reads auch eine Variante des sogenannten *Lost Update*-Phänomens, bei dem eine Änderung einer Transaktion verloren geht, die über einen SQL-Cursor auf eine Objektmenge zugreift:

$$r_1(x)\ r_2(x)\ w_2(x)\ c_2\ w_1(x)\ c_1$$

Die Änderung, die Transaktion t_2 an Objekt x vorgenommen hat, ist nach dem Commit von t_1 verloren, da t_1 auf dem alten Wert von x gearbeitet und den durch t_2 geänderten Wert von x nie gesehen hat.

Die Idee dieses Levels besteht darin, mit einem Cursor über eine Menge von Objekten navigieren und diese Objekte auch ändern zu können, ohne dass die Änderung durch einen parallel laufende Transaktion überschrieben wird. Dies wäre möglich, wenn man nur eine kurze Lesesperre während des Lesens des aktuellen Objektes halten würde. Eine mögliche Implementierung dieses Levels besteht darin, die Lesesperre auf dem Objekt, auf dem aktuell der Cursor steht, zu halten, bis der Cursor weiterbewegt wird. Ändert man das Objekt, muss man natürlich die Lese- zu einer Schreibsperre konvertieren, die man dann bis zum Ende der Transaktion hält.

2.3.2.5 Isolation Levels und Mehrversions-Concurrency-Control

Die Definitionen von Gray et. al. und Berenson et. al. für Isolation Levels haben sich nur auf Scheduler bezogen, die nur eine einzige Version eines Objektes verwalten können. Einige aktuelle Systeme verwenden aber Mehrversionsalgorithmen zur Concurrency Control, wie wir sie in Abschnitt 2.2.3 vorgestellt haben. Auch in solchen Umgebungen will man aber für manche Anwendungen durch den Einsatz von Isolation Levels die mögliche Performance erhöhen. Die phänomenbasierte Definition der ANSI deckt solche Systeme prinzipiell ab, wir haben aber gezeigt, dass die Vermeidung der explizit

genannten Phänomene nicht ausreicht. Es ist also notwendig, auch für den Mehrversionsfall die verschiedenen Isolation Levels zu definieren. Im folgenden betrachten wir dazu die noch einmal die Phänomene aus Abschnitt 2.3.2.2 in ihrer verallgemeinerten Interpretation und entwickeln daraus Anforderungen an einen Mehrversionen-Scheduler, die notwendig sind, um diese Phänomene zu vermeiden.

- *Dirty Writes*

  Mehrversionen-Scheduler erzeugen für jede Schreiboperation eine neue Version eines Objektes. Dirty Writes können daher in diesem Modell nicht vorkommen, so dass wir auch keine besonderen Maßnahmen treffen müssen, um sie zu vermeiden.

- *Dirty Reads*

  Leseoperationen wird in der Mehrversionstheorie von der Versionsfunktion eine vorher erzeugte Version des gelesenen Objektes zugeordnet. Um Dirty Reads zu vermeiden, müssen wir also eine Einschränkung der Versionsfunktion fordern: *Einer Leseoperation darf nur noch eine Version* des *gelesenen Objektes zugeordnet werden, die von einer Transaktion erzeugt wurde, die zum Zeitpunkt der Leseoperation bereits mit Commit beendet ist*. In der Praxis kann es viele erlaubte Versionen geben; um die Kompatibilität mit dem Einversionsfall zu wahren, kann man fordern, dass immer die zeitlich letzte Version zugeordnet werden muss.

- *Unrepeatable Reads*

  Um Unrepeatable Reads zu vermeiden, müssen wir fordern, dass die Versionsfunktion wiederholten Leseoperationen einer Transaktion auf dem gleichen Objekt immer die gleiche Version zuordnet. Das ist keine große Einschränkung für den Scheduler, erfordert aber, dass alte Versionen ggf. lange aufgehoben werden und schränkt damit die Garbage Collection alter Versionen ein. In unserem Ausführungsmodell fordern wir sogar, dass die Transaktion selbst sich implizit so verhält, indem sie jedes Objekt höchstens einmal liest.

- *Phantome*

  Wie wir in Abschnitt 2.3.2.2 gesehen haben, können "indirekte" Phantome auftreten, indem abgeleitete Daten wie die Zahl aller Objekte, die ein Suchprädikat erfüllen, in der Datenbank geändert werden. Es genügt daher nicht, nur zu fordern, dass wiederholtes Auswerten des gleichen Suchprädikats die gleichen Versionen der Objekte ergibt; praktisch wäre das ohnehin nur mit einigem Aufwand zu realisieren, man müsste die Anfragen und ihre Ergebnisse zwischenspeichern. Stattdessen fordern wir, dass alle Leseoperationen einer Transaktion den Zustand der Datenbank zu einem festen Zeitpunkt sehen, d.h. dass einer Operation, die ein Objekt liest, die zum Zeitpunkt des Beginns der Transaktion jüngste Version dieses Objektes zugeordnet wird, deren erzeugende Transaktion bereits mit Commit beendet war. Die Transaktion sieht also einen "Schnappschuss" der stabilen Datenbank zum Zeitpunkt ihres Starts.

Basierend auf diesen Anforderungen können wir nun Isolation Levels für Mehrversionsscheduler definieren:

Definition 2.12: **(Isolation Levels für Mehrversionsscheduler)**

Ein Mehrversionsscheduler erzeugt Schedules des Isolation Levels

- *Multiversion Read Committed*, wenn er einer Leseoperation die zum Zeitpunkt der Leseoperation jüngste Version des gelesenen Objektes zuordnet, die von einer Transaktion erzeugt wurde, die zum Zeitpunkt der Leseoperation bereits mit Commit beendet ist;
- *Multiversion Repeatable Read*, wenn er zusätzlich jedem wiederholten Lesen des gleichen Objektes durch die gleiche Transaktion die gleiche Version des Objektes zuordnet;
- *Multiversion Snapshot*, wenn er einer Leseoperation die zum Zeitpunkt des Anfangs der Transaktion jüngste Version des gelesenen Objektes zuordnet, die von einer Transaktion erzeugt wurde, die zu diesem Zeitpunkt bereits mit Commit beendet ist. ■

Obwohl Multiversion Snapshot alle im ANSI-Standard genannten Phänomene vermeidet, erlaubt er auch nichtserialierbare Schedules. Das liegt daran, dass alle bisher definierten Isolation Levels für Mehrversions-Scheduler ausschließlich die Behandlung von Leseoperationen einschränken. Durch Wechselwirkungen von Schreiboperationen mehrerer Transaktionen können nichtserialisierbare Schedules wie der folgende entstehen:

$$r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ c_1\ w_2(x_2)\ c_2$$

Hier geht die Änderung, die z.B. eine Erhöhung sein könnte, von x durch t_1 verloren, weil t_2 basierend auf dem alten Wert von x arbeitet und die Änderung von t_1 überschreibt. Es handelt sich also um eine Variante des Lost Update-Problems, das wir in Abschnitt 2.3.2.4 vorgestellt haben. Im Unterschied zum Einversionsfall steht hier aber die Leseoperation von t_2 *nach* der Schreiboperation von t_1, aber durch die Versionsfunktion tritt das Problem hier trotzdem auf. In einem seriellen Schedule kann dieser Fall nicht auftreten. Will man dieses Phänomen vermeiden, kann man z.B. verbieten, dass parallel laufende Transaktionen ein gemeinsames Objekt verändern. Dies führt unmittelbar zu *Snapshot Isolation,* einem Isolation Level, der von einigen wichtigen Datenbanksystemen unterstützt wird. Wir stellen Snapshot Isolation wegen seiner Wichtigkeit in der Praxis im folgenden Abschnitt 2.4 ausführlich vor.

2.3.2.6 Beziehungen zwischen den Isolation Levels

Eine naheliegende Lösung ist es, zwei Isolation Levels anhand der jeweiligen Menge von Schedules zu vergleichen, die sie zulassen, ähnlich wie man es auch mit den Subklassen von VSR macht. Man könnte zum Beispiel aus der Beziehung "Die Menge der Schedules von Level 1 ist Teilmenge der Schedules von Level 2" folgern, dass Level 1 schwächer als Level 2 sei, weil er weniger Schedules erlaubt dieser Level. Leider ist dieses Kriterium aber praktisch nicht anwendbar, da alle Isolation Levels (einschließlich "Serializable" nach der Phänomen-basierten Definition) unvergleichbar mit der Menge aller (view- bzw. konflikt-) serialisierbaren Schedules wären: Natürlich erlauben alle Isolation Levels auch Schedules, die nicht serialisierbar sind, andererseits verbieten sie manche serialisierbaren Schedules.

Andererseits interessiert man sich ja gerade für die nichtserialisierbaren Schedules, die in einem Isolation Level erlaubt sind. Um zwei Isolation Levels zu vergleichen, betrachtet man daher sinnvollerweise nur die nichtserialisierbaren Schedules, die sie zulassen. Wir sehen einen Isolation Level dabei als schwächer als einen anderen an, wenn er mehr Schedules zulässt als der andere. Formal definieren wir dazu eine Relation zwischen Isolation Levels [BBGM+95]:

Definition 2.13 : (Vergleich von Isolation Levels)

Die Menge der nichtserialisierbaren Schedules, die unter einem Isolation Level L erlaubt sind, heißt *N(L)*.

Ein Isolation Level L_1 heißt

- *schwächer* als ein Isolation Level L_2 (oder L_2 *stärker* als L_1), in Zeichen $L_1 < L_2$, wenn $N(L_1) \supset N(L_2)$,
- *äquivalent* zu einem Isolation Level L_2, in Zeichen $L_1 = L_2$, wenn $N(L_1) = N(L_2)$,
- *nicht stärker* als ein Isolation Level L_2, in Zeichen $L_1 \leq L_2$, wenn L_1 schwächer als oder äquivalent zu L_2 ist,
- *unvergleichbar* mit einem Isolation Level L_2, in Zeichen $L_1 <> L_2$, wenn weder $N(L_1) \supset N(L_2)$ noch $N(L_2) \supset N(L_1)$ gilt. ■

Mit dieser Definition kann man nun die verschiedenen Isolation Levels, die auf Sperren sowie engen und verallgemeinerten Interpretationen der ANSI-Definitionen basieren, miteinander vergleichen. Berenson et. al. [BBGM+95] zeigen u.a., dass die mittels Sperren definierten Levels äquivalent zu den entsprechenden mit der verallgemeinerten Interpretation erhaltenen Levels sind. Die Levels der engen Interpretation sind schwächer als die der verallgemeinerten Interpretation, weil sie mehr nichtserialisierbare Schedules erlauben.

Im letzten Abschnitt haben wir Isolation Levels für Mehrversionenschedules eingeführt. Um sie mit den übrigen Isolation Levels vergleichen zu können, müssen wir zunächst die Einversionenschedules zu Mehrversionenschedules machen. Wir verwenden dazu eine Standard-Versionsfunktion, die jeder Leseoperation die letzte vorhergehende Schreiboperation zuordnet, ohne Rücksicht darauf, ob die schreibende Transaktion bereits mit Commit beendet wurde. Jede Schreiboperation erzeugt eine neue Version.

Im praktischen Einsatz ist natürlich auch von großem Interesse, wie viele Schedules ein Isolation Level insgesamt erlaubt, da man für die Preisgabe der Korrektheit der Ausführung natürlich möglichst viele erlaubte Schedules und damit eine möglichst große Performance erzielen will. Insbesondere die Isolation Levels, die aus der verallgemeinerten Interpretation der ANSI-Definitionen abgeleitet werden, verbieten, wie wir in Abschnitt 2.3.2.2 gezeigt haben, relativ viele serialisierbare Schedules, allerdings erlauben sie nicht weniger serialisierbare Schedules als ein strikt in zwei Phasen sperrender Scheduler.

2.4 Snapshot Isolation

2.4.1 Motivation

Das Protokoll ROMV, das wir in Abschnitt 2.2.3 kurz vorgestellt haben, hat enorme Performancevorteile für reine Lesetransaktionen, die nie blockiert werden. Sobald aber mindestens eine Änderung an der Datenbank vorgenommen wird, müssen für *alle* Operationen strikte Sperren erworben werden, so dass die Performance solcher gemischter Transaktionen, speziell unter hoher Last, leidet. Um auch die Leistung dieser Transaktionen zu verbessern, liegt es daher nahe, auch auf die Leseoperationen von gemischten Transaktionen das ROMV-Protokoll für Lesetransaktionen anzuwenden, so dass auch diese Operationen nie blockiert werden. Für Schreiboperationen müssten nach wie vor exklusive Sperren erworben werden. Da Transaktionen in der Praxis häufig wesentlich mehr Lese- als Schreiboperationen ausführen, verspricht man sich davon eine wesentliche Verbesserung der Performance. Leider ist damit aber Serialisierbarkeit nicht mehr sichergestellt, was man am folgenden Schedule erkennt:

$$r_1(x_0)\ r_2(x_0)\ w_1(x_1)\ c_1\ w_2(x_2)\ c_2$$

Der Schedule ist nicht serialisierbar, da jede mögliche serielle Reihenfolge der beiden Transaktionen die Liest-von-Beziehung verletzt. Insbesondere liegt das Problem darin, dass t_1 und t_2 parallel das gleiche Objekt ändern wollen, nachdem sie beide den gleichen alten Wert gelesen haben; hier zeigt sich also ein Lost-Update-Problem, da die Änderung einer der beiden Transaktionen verloren geht. Es ist nun naheliegend, das Problem zu beheben, indem man solche Ausführungsmuster verbietet. Das führt unmittelbar zu Snapshot Isolation: Man verbietet, dass zwei parallel laufende Transaktionen ein gemeinsames Objekt verändern; tritt der Fall doch auf, so muss eine der beiden zwangsweise zurückgesetzt werden.

Obwohl damit viele Problemfälle abgedeckt werden, werden wir später sehen, dass dies nicht genügt, um wirklich Serialisierbarkeit zu gewährleisten. Je nach Anwendung kann aber der Performancegewinn den Verlust an Konsistenz aufwiegen.

2.4.2 Formale Definitionen

Bevor wir in den folgenden Abschnitten Eigenschaften von Snapshot Isolation diskutieren können, müssen wir zunächst formal definieren, wann ein Schedule zur Klasse Snapshot Isolation gehört:

Definition 2.14: **(Snapshot Isolation)**

Ein Mehrversionenschedule von Transaktionen $T=\{t_1, ...\}$ gewährleistet *Snapshot Isolation (SI)* für die beteiligten Transaktionen, wenn die beiden folgenden Bedingungen gelten:

(SI-V) *SI-Versionsfunktion*: Die Versionsfunktion bildet jede Leseoperation $r_i(x)$ auf die Schreiboperation $w_j(x_j)$ der Transaktion t_j ab, deren Commit zeitlich am kürzesten vor der Begin-Operation von t_i liegt,

oder formal: $r_i(x)$ wird auf die Operation $w_j(x_j)$ abgebildet, für die $w_j(x_j)<C_j<B_i<r_i(x)$ gilt und es keine anderen Operationen $w_h(x_h)$, C_h ($h \neq j$) gibt mit $w_h(x_h)<B_i$ und $C_j<C_h<B_i$.

(SI-W) *Nichtüberlappende Schreibmengen*: Die Menge der Objekte, die zwei parallele Transaktionen schreiben, sind disjunkt,

oder formal: Wenn für zwei Transaktionen t_i und t_j entweder $B_i<B_j<C_i$ oder $B_j<B_i<C_j$ gilt, dann dürfen t_i und t_j nicht beide das gleiche Objekt x schreiben.

Die Klasse aller Schedules mit dieser Eigenschaft heißt *SI*. ■

Im Gegensatz zur Darstellung in [BBGM+95] legen wir uns in dieser Definition nicht darauf fest, welche Transaktion abgebrochen werden muss, wenn zwei parallele Transaktionen versuchen, das gleiche Objekt zu ändern. Möglich sind zwei Alternativen:

- *First-Committer-Wins*: Paralleles Schreiben wird zunächst zugelassen, aber nur die Änderungen der Transaktion, die zuerst ihr Commit absetzt, werden in die Datenbank geschrieben. Die andere Transaktion wird vom System automatisch zurückgesetzt. Hier hat man den Vorteil, dass man die zweite Transaktion nicht behindert, wenn die erste aus einem anderen Grund vor ihrem Commit abgebrochen werden sollte.
- *First-Writer-Wins*: Nachdem eine Transaktion ein Objekt verändert hat, wird jede weitere Änderung dieses Objektes durch eine andere Transaktion so lange verzögert, bis die erste Transaktion abgeschlossen ist. Wenn sie mit Commit beendet wurde, werden alle wartenden Transaktionen automatisch abgebrochen, andernfalls können sie weiterarbeiten. Auf diese Weise vermeidet man, dass Transaktionen, die vom System abgebrochen werden müssen, unnötige Arbeit tun, da sie warten müssen.

Reale Systeme wie Oracle und PostgreSQL implementieren in der Regel die zweite Variante, indem sie Schreibsperren auf Objektebene verwenden.

2.4.3 Snapshot Isolation und Serialisierbarkeit

In Abschnitt 2.4.1 hatten wir bereits erwähnt, dass Snapshot Isolation auch nicht serialisierbare Schedules erlaubt, so dass inkonsistente Daten entstehen können. Gleichzeitig gibt es aber auch serialisierbare Schedules, die nicht Element von SI sind. In diesem Abschnitt beleuchten wir die Beziehung von Snapshot Isolation und (Mehrversions-) Serialisierbarkeit näher und stellen Möglichkeiten vor, wie man durch zusätzliche Maßnahmen Serialisierbarkeit sicherstellen kann.

Wir zeigen zunächst, dass nicht alle mehrversions-serialisierbaren Schedules auch Element von SI sind. Dazu betrachten wir den folgenden Schedule:

$$r_1(x_0)\ w_1(x_1)\ r_2(y_0)\ c_1\ r_2(x_1)\ c_2$$

Dieser Schedule ist serialisierbar, insbesondere ist er äquivalent zu dem seriellen Einversionenschedule t_1t_2. Er ist aber nicht Element von SI, da t_2 die "falsche" Version von x liest, nämlich x_1 statt der eigentlich "richtigen" Version x_0. Snapshot Isolation enthält also schon wegen seiner speziellen Versionsfunktion nicht alle möglichen mehrversions-serialisierbaren Schedules. Dies ist gleichzeitig der einzige Grund, weshalb ein Schedule aus MVSR nicht in SI sein kann, insbesondere zeigt die folgende einfache Überlegung, dass die Forderung disjunkter Writesets keine zusätzlichen Schedules aus MVSR ausschließt:

Nehmen wir an, dass in einem Schedule zwei parallel laufende Transaktionen t_1 und t_2 jeweils eine neue Version von Objekt x erzeugen. Die Forderung disjunkter Writesets schließt dann aus, dass dieser Schedule in SI sein kann. Weil in unserem Modell jedes Objekt gelesen werden muss, bevor es geschrieben werden darf, und weil die beiden Transaktionen parallel laufen, lesen sie die alten Versionen x_{k1} und x_{k2}. Jeder Versuch, einen äquivalenten seriellen Schedule zu konstruieren, muss nun fehlschlagen: Man muss t_{k1} und t_{k2} vor t_1 und t_2 anordnen, da t_{k1} und t_{k2} unmöglich von t_1 oder t_2 gelesen haben können. Ordnet man nun t_1 vor t_2 an, liest t_2 x von t_1 statt von t_{k2}; analog kann man t_2 nicht vor t_1 anordnen, da sonst t_1 von t_2 statt von t_1 lesen würde. Der Ausgangsschedule war also überhaupt nicht mehrversions-serialisierbar und damit nicht Element von MVSR. Die einzigen mehrversions-serialisierbaren Schedules, die nicht Element von SI sind, sind also wirklich diejenigen mit einer anderen Versionsfunktion.

Ein wesentlich schwerwiegenderes Problem stellen Schedules dar, die zwar Element von SI sind, aber nicht mehrversions-serialisierbar. Ein solcher Schedule ist z. B. der folgende:

$$r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ c_1\ w_2(y_2)\ c_2$$

Wir nehmen an, dass eine Konsistenzbedingung über x und y definiert ist, z. B. "x+y>100", und dass jede Transaktion für sich diese Konsistenzbedingung erhält. Dies kann etwa dadurch geschehen, dass im Transaktionsprogramm zunächst die aktuellen Werte von x und y gelesen werden und anhand dieser Werte entschieden wird, ob eine Änderung von x oder y möglich ist. Die Transaktionen t_1 und t_2 im Beispiel tun genau dies. Durch die gezeigte parallele Ausführung der beiden Transaktionen kann die Konsistenzbedingung aber verletzt werden, während jede serielle Ausführung der beiden Transaktionen die Bedingung erhält. Insbesondere ist dieser Schedule also nicht mehrversions-serialisierbar.

Serialisierbarkeit lässt sich also durch den Vergleich der Writesets paralleler Transaktionen alleine nicht erzielen. Man kann allerdings zeigen, dass es ausreichen würde, den Writeset einer Transaktion, die mit Commit beendet werden soll, mit den Readsets aller bereits mit Commit beendeten Transaktionen zu vergleichen, die parallel zu dieser gelaufen sind. Wir stellen dazu zunächst fest, dass wir, ohne die Semantik der Transaktion zu ändern, fiktiv alle Leseoperationen einer Transaktion t_i an ihrem Beginn B_i und alle Schreiboperationen unmittelbar vor ihrem Commit C_i bündeln können. Wir verwenden nun das *BOCC*-Protokoll (backward oriented concurrency control, [KR81, Härd84]), das Änderungen einer Transaktion t_i, die mit Commit beendet werden soll, erst nach einer erfolgreichen Validierung in die Datenbank schreibt. BOCC vergleicht dazu $RS(t_i)$ mit den Writesets aller parallel zu t_i gelaufenen Transaktionen, die bereits mit Commit beendet und erfolgreich validiert wurden; nur wenn alle Writesets disjunkt mit dem Readset von t_i sind, darf t_i sein Commit ausführen. BOCC lässt sich in unserem Fall auch anwenden, da die Änderungen einer Transaktion erst nach ihrem Commit von den übrigen gesehen werden können. Auf diese Weise ließe sich Serialisierbarkeit gewährleisten, man müsste aber Read- und Writesets verwalten und beim Commit auf Disjunktheit prüfen, was recht viel Aufwand bedeuten kann. Da BOCC die Validierungsphase als kritischen Abschnitt betrachtet, in dem immer nur eine Transaktion sein kann, führt dies in der Praxis zu erheblichen Performanceeinschränkungen. Weil üblicherweise viel mehr Daten gelesen als geschrieben werden, verursacht das Vergleichen der Writesets bei Snapshot Isolation wesentlich weniger Overhead, der durch die First-

Writer-Wins-Technik noch weiter verringert werden kann; dafür muss aber eine geringere Konsistenzgarantie in Kauf genommen werden.

Das mit BOCC verwandte Verfahren *FOCC* (forward optimistic concurrency control) hat den Vorteil, dass es in seiner Validierungsphase den Writeset der zu committenden Transaktion mit den Readsets aller gerade laufenden Transaktionen vergleicht, so dass keine Daten über das Commit einer Transaktion hinaus aufgehoben werden müssen. Es lässt sich aber in diesem Zusammenhang nicht anwenden: Wir haben oben festgestellt, dass alle Leseoperationen einer Transaktion fiktiv an ihrem Anfang gebündelt werden können, so dass man in FOCC immer mit dem gesamten Readset einer Transaktion vergleichen könnte. In einer laufenden Transaktion ist dies aber noch gar nicht bekannt, so dass man entweder das Commit verzögern oder die Readsets aller Transaktionen vorher bekannt geben müsste. Beide Möglichkeiten sind nicht praktikabel.

Auf Applikationsebene könnte man versuchen, durch geeignetes Anfordern von Sperren das Verletzen von Konsistenzbedingungen zu vermeiden. Dazu muss man aber voraussetzen, dass dem Applikationsprogrammierer alle Konsistenzbedingungen, die es jemals geben wird, zum Zeitpunkt der Programmentwicklung bekannt sind. Wie der folgende Schedule zeigt, kann es aber selbst dann zu Problemen kommen:

$$r_3(x_0)\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ c_1\ r_3(y_0)\ w_3(y_3)\ c_3$$

Wir nehmen wieder an, dass ein Constraint über x und y definiert ist, die Transaktion t_1 ändert den Wert von x unter Berücksichtigung dieses Constraints. Zwischen dem Lesen von x und dem Ändern von y durch t_3 kann viel Zeit (und entsprechend viel Code) liegen; möglicherweise hat sich die Änderung von y auch erst durch Eingaben des Benutzers ergeben, so dass zum Zeitpunkt des Lesens von x noch gar nicht klar war, dass der Constraint zwischen x und y überhaupt eine Rolle spielt. Um diesen Fall durch Sperren abzudecken, müsste jedes Objekt, auf das zugegriffen wird, auf der Applikationsebene exklusiv gesperrt werden, unabhängig davon, ob es nur gelesen oder später auch geschrieben wird. Dadurch verliert man aber gerade die Performancevorteile (z.B. die Blockierungsfreiheit für Leseoperationen), die SI bietet, so dass diese Alternative praktisch nicht in Frage kommt. Allein durch Maßnahmen auf der Applikationsebene lassen sich diese problematischen Fälle also in der Praxis nicht vermeiden.

2.4.4 Graphbasierter SR-Test für SI-Datenbanken

Im letzten Abschnitt haben wir verschiedene Möglichkeiten diskutiert, wie man Serialisierbarkeit in Datenbanken sicherstellen könnte, die nur Snapshot Isolation gewährleisten. In diesem Abschnitt leiten wir nun einen graphbasierten Algorithmus her, mit dem man für Schedules, die bereits die für SI "richtige" Versionsfunktion haben, entweder Mitgliedschaft in SI oder in MVSR sicherstellen kann. Wir erweitern dazu den Mehrversions-Serialisierungsgraphen aus Abschnitt 2.2.3 um geeignete Kantenbeschriftungen. Als Versionsordnung, die zu einem solchen Graphen immer gehört, kommt alleine die folgende *SI-Versionsordnung* in Frage:

Definition 2.15: (SI-Versionsordnung)

Die *SI-Versionsordnung* $\ll_s$ zweier Versionen eines Objektes ist die Ordnung, die durch die Reihenfolge der Commitoperationen der Transaktionen vorgegeben wird, die die Versionen erzeugt haben, oder formal:

$x_i \ll_s x_j :\Leftrightarrow C_i < C_j$ ■

Dies ist die einzige Versionsordnung, die für serialisierbare Schedules in SI zu einem azyklischen MVSG führt. Um dies zu verdeutlichen, benutzen wir den folgenden Schedule als Beispiel:

$$r_1(x_0)\ w_1(x_1)\ c_1\ r_2(x_1)\ w_2(x_2)\ c_2\ r_3(x_2)\ w_3(x_3)\ c_3$$

Dieser Schedule ist serialisierbar, also gibt es einen azyklischen Mehrversions-Serialisierungsgraphen, außerdem ist er Element von SI. Wählt man nun z.B. die Versionsordnung $x_2 \ll x_1 \ll x_3$, erhält man (u.a.) die WR-Kanten $t_1 \rightarrow t_2$, $t_2 \rightarrow t_3$ und die RW-Kante $t_3 \rightarrow t_1$, so dass ein Zyklus geschlossen wird. Dies liegt daran, dass in unserem Modell immer zuerst gelesen wird, bevor geschrieben wird, und außerdem das gleiche Objekt nicht parallel geschrieben werden darf. Dadurch impliziert die Reihenfolge der Schreiboperationen auf einem Objekt die einzig mögliche serielle Anordnung der Transaktionen, ohne die Liest-von-Relation zu verändern. Die SI-Versionsordnung entspricht genau dieser Reihenfolge.

Mit Hilfe der SI-Versionsordnung können wir nun den SI-Mehrversions-Serialisierungsgraph als Erweiterung des üblichen MVSG definieren:

Definition 2.16: (SI-MVSG)

Der SI-Mehrversions-Serialisierungsgraph SI-MVSG eines gegebenen Mehrversionenschedules s, der (SI-V) erfüllt, ist ein gerichteter Graph mit den Transaktionen als Knoten und den folgenden Kanten:

- Für jede Operation $r_j(x_i)$ im Schedule gibt es eine Kante $t_i \rightarrow t_j$, die mit "x" beschriftet wird (*WR-Kante*).
- Für jedes Paar von Operationen $r_k(x_j)$ und $w_i(x_i)$ mit paarweise verschiedenen i, j, k gibt es eine Kante

  (i) $t_i \rightarrow t_j$, falls $x_i \ll_s x_j$ (*WW-Kante*),
  (ii) $t_k \rightarrow t_i$, falls $x_j \ll_s x_i$ (*RW-Kante*),

  die in beiden Fällen mit "x" beschriftet wird.

Kanten im Graph, die mit "x" beschriftet sind, heißen *x-Kanten*. Ein Zyklus im Graphen, der nur aus x-Kanten besteht, heißt *x-Zyklus*. ■

Der SI-MVSG unterscheidet sich vom MVSG durch die feste Versionsordnung und die Beschriftungen der Kanten. Analog zu Theorem 2.2 aus Abschnitt 2.2.3 können wir nun die Mitgliedschaft von Schedules in SI und MVSR mit Hilfe dieses Graphen charakterisieren:

Theorem 2.4: **(Äquivalenz von SI und zyklusfreiem SI-MVSG)**

Ein Mehrversionenschedule, der (SI-V) erfüllt, ist

a) in MVSR genau dann, wenn sein zugehöriger SI-MVSG zyklusfrei ist,

b) in SI genau dann, wenn es kein Objekt x gibt, so dass der SI-MVSG einen x-Zyklus hat. ■

Beweis:

a) Da wir voraussetzen, dass der Schedule (SI-V) erfüllt, und wir außerdem voraussetzen, dass jeder Schreiboperation eine Leseoperation auf demselben Objekt vorausgeht, ist die SI-Versionsordnung die einzige Versionsordnung, die den MVSG eines Schedules azyklisch machen kann. Der Satz folgt dann direkt aus Theorem 2.2, da der SI-MVSG bis auf die Beschriftung die gleichen Kanten enthält wie der gewöhnliche MVSG.

b) "⇐":

Nehmen wir an, der Schedule sei nicht in SI. Weil der Schedule (SI-V) erfüllt, muss also (SI-W) verletzt sein. Es gibt also mindestens zwei parallele Transaktionen t_i und t_j, die dasselbe Objekt x schreiben. Weil jedes Objekt gelesen werden muss, bevor es geschrieben werden darf, lesen beide Transaktionen x, bevor sie es schreiben. Nehmen wir an, t_i lese x_k und t_j lese x_l. Nach der Definition der Versionsfunktion muss t_k dann mit Commit beendet worden sein, bevor t_i begonnen hat, also gilt $x_k \ll_s x_i$ nach der Definition der SI-Versionsordnung $\ll_s$. Das gleiche gilt für t_l und t_j, also erhalten wir $x_l \ll_s x_j$. Außerdem muss t_k vor t_j mit Commit beendet werden, da t_i von t_k liest (also war t_k bereits mit Commit beendet, als t_i begonnen hat) und t_j und t_i parallel ausgeführt werden, so dass das Ende von tj nach dem Anfang von t_i liegen muss. Dies ergibt $x_k \ll_s x_j$ und, analog, $x_l \ll_s x_i$. Der SI-MVSG enthält also die Kanten

$t_i \rightarrow t_j$ beschriftet mit "x", wegen $r_i(x_k)$, $w_j(x_j)$ und $x_k \ll_s x_j$, sowie
$t_j \rightarrow t_i$ beschriftet mit "x", wegen $r_j(x_l)$, $w_i(x_i)$ und $x_l \ll_s x_i$.

Aber dies ist ein x-Zyklus, also ein Widerspruch zur vorausgesetzten x-Zykel-Freiheit.

"⇒":

Nehmen wir an, es gebe einen x-Zyklus $t_{i1} \rightarrow t_{i2} \rightarrow ... \rightarrow t_{in}=t_{i1}$ im SI-MVSG des Schedules s, $s \in SI$. Wenn es mehr als einen gibt, wählen wir einen minimaler Länge. Ohne Beschränkung der Allgemeinheit nehmen wir weiterhin an, dass die Transaktionen im Zyklus so nummeriert sind, dass $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n=t_i$.

Wir zeigen zunächst, dass es mindestens eine Kante in diesem Zyklus geben muss, die wegen eines RW-Paars eingefügt wurde. Sowohl WR- als auch WW-Kanten sind Kanten, die von einer Transaktion, die eine ältere Version von x (im Sinne von $\ll_s$) schreibt, zu einer Transaktion, die eine jüngere Version von x schreibt. Weil die SI-Versionsordnung $\ll_s$ durch die Commitordnung der Transaktionen definiert ist, bedeutet eine solche Kante von t_i nach t_j, dass t_i sein Commit vor dem Commit von t_j

ausführt, formal $C_i<C_j$. Aber es handelt sich um einen Zyklus. Das würde bedeuten, dass $C_1 < C_1$, was natürlich nicht sein kann, also muss es mindestens eine Kante $t_i \rightarrow t_{i+1}$ geben, für die $C_i > C_{i+1}$ gilt. Wir wählen die erste solche Kante.

Diese Kante wurde zum Graphen hinzugefügt, weil $r_i(x_l)$ und $w_{i+1}(x_{i+1})$ im Schedule vorkommen und $C_l < C_{i+1}$ gilt. Außerdem gilt $C_l < B_i$ und es gibt keine andere Commitoperation einer Transaktion dazwischen, die x schreibt, da t_i x von t_l liest. Wenn wir dies nun zusammensetzten, erhalten wir $C_l < B_i < C_{i+1} < C_i$. Wir haben also gezeigt, dass C_i und C_{i+1} parallel zueinander ablaufen. Das alleine bedeutet noch nicht notwendigerweise, dass der Schedule nicht in SI ist, da t_i nicht unbedingt x schreiben muss.

Da ein Objekt gelesen werden muss, bevor es geschrieben werden darf, muss t_{i+1} eine Version x_p von x lesen, bevor x_{i+1} geschrieben wird. Um einen Widerspruch zu erzielen, betrachten wir nun die verschiedenen Möglichkeiten für die Transaktion t_p, von der die Transaktion t_{i+1} lesen kann.

(i) Wenn t_{i+1} von einer Transaktion liest, die ihr Commit vor t_l ausgeführt hat, erhalten wir $C_p < B_{i+1} < C_l$. Aber das bedeutet, dass t_l und t_{i+1} parallel abgelaufen sind und das gleiche Objekt x geschrieben haben, so dass der Schedule nicht in SI gewesen sein kann, was ein Widerspruch zur Annahme ist.

(ii) Wenn t_{i+1} x von t_l liest, erhalten wir l=p und $C_l < B_{i+1} < C_{i+1}$. Das heißt nun aber, dass es eine x-Kante im SI-MVSG von t_l nach t_{i+1} gibt, die eine WW-Kante ist. Aber dann kann t_l nicht auf dem Zyklus liegen, oder wir könnten die Kanten $t_l \rightarrow t_i \rightarrow t_{i+1}$ durch die Kanten $t_l \rightarrow t_{i+1}$ ersetzen und erhielten einen kürzeren Zyklus, was im Widerspruch zur Minimalität des gewählten Zyklus steht.

Also wissen wir, dass t_i eine eingehende x-Kante hat, die nicht von t_l kommt. Die Kante von t_l ist aber t_i's einzige eingehende x-Kante des Typs WR, da t_i x nur von t_l liest. Daher muss die eingehende x-Kante vom Typ RW oder WW sein. Aus der Definition des SI-MVSG folgt dann aber, dass t_i x schreiben muss, um eine solche eingehende Kante zu haben. Wir haben also gezeigt, dass die parallelen Transaktionen t_i und t_{i+1} beide x schreiben, was ein Widerspruch dazu ist, dass s in SI ist.

(iii) Wenn t_{i+1} x von einer Transaktion t_q liest, die ihr Commit nach dem Anfang von t_i ausgeführt hat, erhalten wir die Ordnung $C_l < B_i < C_q < B_{i+1} < C_{i+1} < C_i$. Wenn t_l nicht auf dem Zyklus wäre, könnten wir wie oben zeigen, dass t_i x schreibt, so dass s nicht in SI wäre. Daher muss t_l auf dem Zyklus liegen.

Wenn es keine andere Transaktion zwischen t_l und t_q gibt, die x schreibt, enthält der Graph die Kanten $t_l \rightarrow t_q$ (t_q liest x von t_l) und $t_q \rightarrow t_{i+1}$ (t_{i+1} liest x von t_q). Wenn wir die Kanten $t_l \rightarrow t_i \rightarrow t_{i+1}$ durch die Kanten $t_l \rightarrow t_q \rightarrow t_{i+1}$ ersetzen, erhalten wir einen Zyklus mit der gleichen Länge. Außerdem können wir die Kante von einer größeren zu einer kleineren Version ($t_i \rightarrow t_{i+1}$) durch Kanten ersetzen, die die Versionsordnung respektieren ($C_l < C_q < C_{i+1}$). Wie wir vorher gezeigt haben, muss es dann eine andere Kante geben, die nicht der Versionsordnung folgt. Wir können also den Beweis bei dieser Kante fortsetzen. Da der Zyklus endliche Länge hat, können wir dies nur endlich oft tun, bis einer der anderen Fälle angewendet werden kann.

Wenn es eine Sequenz von Transaktionen t_{r1} ... t_{rm} zwischen t_l und t_q gibt, die x schreiben, muss t_{r1} x von t_l lesen, t_{r2} muss x von t_{r1} lesen, und so weiter, bis schließlich t_q x von t_{rm} liest. Daher gibt es die Kanten $t_l \rightarrow t_{r1}$ und $t_{r1} \rightarrow t_{ij+1}$ (wegen $r_{r1}(x_l)$, $w_{ij+1}(x_{ij+1})$, und $x_{ij+1} \gg_s x_l$). Jetzt können wir die Kanten $t_l \rightarrow t_{ij} \rightarrow t_{ij+1}$ durch $t_l \rightarrow t_{r1} \rightarrow t_{ij+1}$ ersetzen, was wiederum Kanten sind, die die Versionsordnung respektieren, und dasselbe Argument wie zuvor lässt sich anwenden. ■

Als Beispiel betrachten wir die beiden folgenden Schedules

$$s_1 := r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ w_2(y_2)\ C_1\ C_2$$
$$s_2 := r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ w_2(x_2)\ C_1\ C_2$$

Der Schedule s_1 ist in SI enthalten, während s_2 kein Element von SI ist (die Transaktionen t_1 und t_2 laufen parallel zueinander ab und schreiben beide x). Die SI-MVSGs beider Schedules, die in Abbildung 2.2 dargestellt sind, enthalten einen Zyklus, daher sind beide Schedules kein Element von MVSR. Dagegen enthält nur der SI-MVSG von s_2 (auf der rechten Seite von Abbildung 2.2) einen Zyklus aus Kanten, die mit dem gleichen Objekt "x" beschriftet sind. Daher ist s_2 nicht in SI, während s_1 ein Element von SI ist.

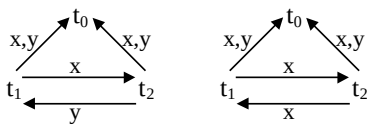

Abbildung 2.2 - SI-MVSGs für die beiden Schedules s_1 (links) und s_2 (rechts)

Aus der Konstruktion des SI-MVSG lässt sich ein Algorithmus ableiten, der zur Laufzeit garantiert, dass nur Schedules erzeugt werden, die Element von SI bzw. von MVSR sind. Er verwendet eine Online-Version des SI-MVSG:

Definition 2.17: (Online SI-MVSG)

Der *Online-Snapshot-Isolation-Multiversions-Serialisierungsgraph* (*OSI-MVSG)* für einen Schedule, der (SI-V) erfüllt, ist ein gerichteter Graph, der die Transaktionen von s als Knoten hat und in den Kanten nach den folgenden Regeln eingefügt werden, wenn Operationen hinzukommen:

(i) Wenn eine Transaktion beginnt, wird sie zum Graphen hinzugefügt.

(ii) Wenn eine Transaktion t_i die Operation $r_i(x)$ ausführt, dann wird für alle Transaktionen t_j im Graphen, die vorher eine Operation $w_j(x)$ ausgeführt haben,
   a) eine Kante $t_i \rightarrow t_j$ zum Graph hinzugefügt, falls t_j und t_i parallel zueinander ablaufen,
   b) eine Kante $t_j \rightarrow t_i$ zum Graph hinzugefügt, falls t_j bereits mit Commit beendet war, als t_i begonnen hat,
   und jede dieser Kanten wird mit x beschriftet.

(iii) Wenn eine Transaktion t_i die Operation $w_i(x)$ ausführt, dann wird für alle Transaktionen t_j im Graph, die bereits die Operation $r_j(x)$ ausgeführt haben, eine Kante $t_j \rightarrow t_i$ in den Graphen eingefügt, die mit x beschriftet ist.

Außerdem wird für jede Transaktion t_j im Graphen, die parallel zu t_i abläuft und bereits die Operation $w_j(x)$ ausgeführt hat, eine Kante $t_j \rightarrow t_i$ in den Graphen eingefügt, die mit x beschriftet ist.

(iv) Eine mit Commit beendete Transaktion wird zusammen mit ihren ausgehenden Kanten aus dem Graphen entfernt, wenn alle Transaktionen im Graph, die parallel zu ihr gelaufen sind, ebenfalls beendet sind und sie eine Quelle des Graphen ist.

(v) Eine mit Abort beendete Transaktion wird unmittelbar aus dem Graphen entfernt, zusammen mit ihren ein- und ausgehenden Kanten. ■

Obwohl der OSI-MVSG dem üblichen Konfliktgraphen sehr ähnlich ist, gibt es einen bedeutsamen Unterschied: In Regel (ii) a) werden Kanten erzeugt, die nicht den üblichen Konfliktkanten entsprechen, sondern genau entgegengesetzt gerichtet sind; dies spiegelt die (SI-V)-Eigenschaft von Snapshot Isolation wieder.

Mit Abort beendete Transaktionen werden unmittelbar aus dem Graphen entfernt, weil sie keinen Einfluss mehr auf die Korrektheit des erzeugten Schedules haben können. Wir werden sie daher bei den folgenden Betrachtungen nicht mehr berücksichtigen. Bevor solche Transaktionen abgebrochen wurden, können sie allerdings dazu beigetragen haben, dass eine andere Transaktion zurückgesetzt wurde, weil ihre letzte Operation zu einem Zyklus im Graph geführt hat, wie etwa im folgenden Schedule:

$$r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x1)\ w_2(y_2)$$

Nachdem die Ausführung so weit fortgeschritten ist, hat der OSI-MVSG die gleiche Gestalt wie der SI-MVSG von Schedule s_1 in Abbildung 2.2. Das Einfügen der Kanten, die zu $w_2(y_2)$ gehören, hat also einen Zyklus im Graphen verursacht, also wird t_2 vom System zurückgesetzt. Wenn nun anschließend t_1, z.B. durch den Benutzer, abgebrochen wird, hat das System einen legalen Schedule abgelehnt, da t_1 für die Serialisierbarkeit nun keine Rolle mehr spielt. Um dieses Problem zu entschärfen, kann man z.B. mit dem Zyklustest so lange wie möglich warten, d.h. spätestens bis zum Ausführen des Commits einer Transaktion. Da selbst das nicht alle möglichen Fälle abdeckt, z.B. auch nicht das Beispiel, könnte man auch die Commits von Transaktionen auf einem Zyklus so lange verzögern, bis alle beteiligten Transaktionen beendet sind, und dann in einer Art "Group Commit" die abzubrechenden bestimmen und die übrigen mit Commit beenden.

Die praktische Relevanz des OSI-MVSG ergibt sich aus dem folgenden Theorem:

Theorem 2.5 (Korrektheit des Zyklustests im OSI-MVSG):

Ein Concurrency-Control-Algorithmus, der auf dem OSI-MVSG basiert, erlaubt nur Schedules, die

(i) serialisierbar sind, wenn er eine Operation ablehnt, sobald die hinzugefügten Kanten zu einem Zyklus im Graphen führen,

(ii) zur Klasse SI gehören, wenn er eine Operation auf einem Objekt x ablehnt, sobald die hinzugefügten Kanten zu einem x-Zyklus im Graphen führen. ■

Das Ablehnen einer Operation führt üblicherweise dazu, dass die Transaktion, die die Operation abgesetzt hat, zurückgesetzt wird bzw. selbst ein Abort auslöst.

Aus dem Theorem ergibt sich, dass der Algorithmus insbesondere dazu genutzt werden kann, um Serialisierbarkeit sicherzustellen, wenn das Datenbanksystem selbst nur Snapshot Isolation beherrscht. In diesem Fall ist nämlich (SI-V) automatisch erfüllt, und es genügt, die Kanten ohne ihre Beschriftungen zu betrachten.

Beweis:

Wir zeigen zunächst, dass jede Kante des SI-MVSG auch im OSI-MVSG auftritt, solange keine Transaktionen aus dem Graphen entfernt werden. Das Theorem folgt dann unmittelbar aus Theorem 2.4: Wenn es keinen Zyklus im OSI-MVSG gibt, dann kann es auch keinen Zyklus im SI-MVSG geben, der höchstens weniger Kanten hat.

(WR-Kanten) Wir nehmen an, dass es eine Operation $r_j(x_i)$ im Schedule gibt, so dass die Kante $t_i \rightarrow t_j$ zum SI-MVSG hinzugefügt wird. Diese Kante wird wegen Regel (ii,b) der Definition des Online SI-MVSG zum Online SI-MVSG hinzugefügt, wenn Transaktion t_j diese Operation abschickt.

(WW-Kanten) Für Operationen $r_k(x_j)$ und $w_i(x_i)$ im Schedule wird eine Kante $t_i \rightarrow t_j$ zum SI-MVSG hinzugefügt, wenn t_i vor t_j mit commit beendet wurde. (tj wiederum wurde mit Commit beendet, bevor tk gestartet wurde.) Da tj x gelesen haben muss, bevor sie x schreiben durfte, wurde beim Abschicken dieser Leseoperation die Kante $t_i \rightarrow t_j$ zum OSI-MVSG hinzugefügt, wieder nach Regel (ii,b).

(RW-Kanten) Für Operationen $r_k(x_j)$ und $w_i(x_i)$ im Schedule wird eine Kante $t_k \rightarrow t_i$ zum SI-MVSG hinzugefügt, wenn t_i nach t_j mit Commit beendet wurde. Wenn t_i seine Schreiboperation abgeschickt hat, bevor t_k seine Leseoperation abgeschickt hat, dann müssen t_i und t_k parallel zueinander laufen, sonst hätte t_k nicht von t_j gelesen. Also wird nach Regel (ii,a) eine Kante $t_k \rightarrow t_i$ zum OSI-MVSG hinzugefügt, wenn t_k seine Leseoperation abschickt. Wenn t_i seine Schreiboperation nach t_k's Leseoperation abschickt, dann wird nach Regel (iii) eine Kante $t_k \rightarrow t_i$ zum OSI-MVSG hinzugefügt, wenn t_i seine Schreiboperation abschickt.

Diese drei Fälle sind alle, bei denen Kanten zum SI-MVSG hinzugefügt werden. Also treten alle Kanten im SI-MVSG daher auch im OSI-MVSG auf. Was wir nun noch zeigen müssen, ist, dass wir keine Zyklen im Graph verpassen, wenn wir mit Commit beendete Transaktionen aus dem Graph entfernen, sofern die Bedingungen aus Regel (iv) der Definition des OSI-MVSG eingehalten werden, unter denen eine Transaktion entfernt werden darf.

Wir nehmen also an, dass wir die mit Commit beendete Transaktion t_i aus dem OSI-MVSG entfernen, die eine Quelle im Graph ist und deren parallel laufende Transaktionen ebenfalls beendet sind. Offensichtlich liegt t_i nicht auf einem Zyklus, da t_i eine Quelle ist. Damit t_i in der Zukunft überhaupt auf einem Zyklus liegen kann, muss also eine Kante zum Graph hinzugefügt werden, die in t_i hineingeht. Eine solche Kante kann nur zu t_i hinzugefügt werden, wenn eine parallel laufende Transaktion t_j ein Objekt liest, das t_i geschrieben hat. Da aber alle parallel zu t_i laufenden Transaktionen bereits beendet sind, kann dies nicht vorkommen. Es ist daher möglich, t_i aus dem Graphen zu entfernen. ■

2.4.5 Andere Kriterien

In jüngster Zeit wurden zwei weitere Kriterien in der Literatur vorgeschlagen, mit denen man feststellen kann, ob ein gegebener Schedule Element von SI ist, bzw. die es erlauben, basierend auf einem Scheduler, der SI erzeugt, durch zusätzliche Maßnahmen serialisierbare Schedules zu erhalten. Wir stellen beide Ansätze kurz vor und zeigen die Unterschiede zu unserer Lösung auf.

Adya et. al. [Adya99,ALO00] führen graphbasierte Kriterien ein, mit denen sich die Zugehörigkeit von Transaktionen eines Schedules zu den verschiedenen Isolation Levels nachweisen lässt. Auf diesem Weg kommen sie zu einer Graphcharakterisierung von Snapshot Isolation, die alle Kanten des SI-MVSG enthält. Zusätzlich enthält ihr Graph aber noch Kanten, die Informationen über die Reihenfolge von Anfang und Commit der Transaktionen enthalten, so dass man auch anhand des Graphen prüfen kann, ob die Versionsfunktion zulässig ist. Das ist ein Vorteil gegenüber dem SI-MVSG, der die Richtigkeit der Versionsfunktion voraussetzt. Um die Zugehörigkeit eines Schedules zu SI zu prüfen, genügt aber ein Zyklustest im Graphen alleine (wie beim SI-MVSG) nicht. Zusätzlich ist ein weiterer Test notwendig, denn es wird gefordert, dass aus der Existenz eines bestimmtes Kantentyps im Graphen auch das Vorhandensein anderer Kanten folgt. Dies ist ähnlich aufwendig wie das Prüfen der Versionsfunktion in unserem Ansatz. Darüber hinaus haben Adya et. al. keine Resultate, die etwas über die Serialisierbarkeit eines Schedules, basierend auf ihrer Graphkonstruktion, aussagen.

Fekete et. al. [Feke99,FLO+01] stellen dagegen einen Ansatz vor, um serialisierbare Ausführungen einer *Menge von Transaktionsprogrammen* sicherzustellen, selbst wenn der Scheduler des Datenbanksystems nur Snapshot Isolation gewährleisten kann. Sie analysieren dazu die gegebenen Transaktionsprogramme im Hinblick auf mögliche Konflikte, die bei der Ausführung zwischen den resultierenden Transaktionen entstehen können. Basierend darauf wird eine Graphstruktur, der *SC-Graph*, konstruiert, der aus verschiedenen Kantentypen besteht. Sie zeigen, dass jede mögliche verschachtelte Ausführung der Transaktionsprogramme unter einem Scheduler, der Snapshot Isolation erlaubt, genau dann zu einem serialisierbaren Schedule führt, wenn ein bestimmter Zyklustyp nicht im SC-Graph auftritt. Zusätzlich schlagen sie Maßnahmen vor, um Transaktionsprogramme anzupassen, falls solch ein Zyklus auftreten sollte. Bereits die Ableitung der Konflikte involviert jedoch den Datenbank-Administrator, der die Programme selbst analysieren und die Konflikte erkennen muss; eine automatische Analyse ist nicht vorgesehen. Dies macht diesen Ansatz, obwohl er in der Theorie sehr nützlich ist, für die üblicherweise in der Praxis auftretenden großen Anwendungssysteme wenig hilfreich.

2.5 Gewährleistung von Atomarität

Bislang haben wir nur Schedules betrachtet, in denen alle Transaktionen schließlich mit Commit beendet werden. In der Praxis enden aber Transaktionen nicht immer erfolgreich, sondern sie können aus verschiedenen Gründen abgebrochen werden. Dies kann zum Beispiel geschehen, wenn das System sie als Opfer zur Auflösung eines Deadlocks ausgewählt hat, weil sie eine Konsistenzbedingung der Datenbank verletzt haben, weil das Anwendungsprogramm abgestürzt ist oder weil das Datenbanksystem selbst ausgefallen ist. Das System darf daher nur solche Schedules erlauben, die es ermöglichen,

Effekte von zurückgesetzten Transaktionen ungeschehen zu machen, ohne dass die Korrektheit des Gesamtsystems darunter leidet. In diesem Abschnitt erweitern wir daher unser bisheriges Modell um Abbrüche von Transaktionen.

Im folgenden Abschnitt stellen wir kurz etablierte Ansätze für die Rücksetzbarkeit von Einversions-Schedules vor. Anschließend erweitern wir diese Ansätze auf Mehrversionenschedules. In Abschnitt 2.5.3 geben wir einen Überblick über Transaktionsabbrüche bei Mehrschichtentransaktionen. Den Abschluss bildet eine Diskussion über den Einfluss von Isolation Levels auf die Rücksetzbarkeit von Schedules.

2.5.1 Etablierte Lösungen

Das Rücksetzen einer Transaktion geschieht üblicherweise, indem das Datenbanksystem die Änderungen, die die Transaktion ausgeführt hat, rückgängig macht, z.B. indem es jedes von einer Transaktion geänderte Objekt mit Hilfe eines Logbuches auf den Zustand vor der Änderung zurücksetzt. Es ist daher naheliegend, diese Behandlung eines Transaktionsabbruchs genau so in das Ausführungsmodell aufzunehmen. Wir definieren daher zunächst die inverse Operation $w_i^{-1}(x)$, die die Änderung von Objekt x durch Transaktion t_i rückgängig macht. Wenn im Schedule Transaktion t_i zurückgesetzt wird, im Schedule durch das Abort a_i der Transaktion dargestellt, führt das System also die inversen Aktionen zu den Schreiboperationen von t_i aus und beendet die Transaktion mit Commit. Es ist dabei wichtig, dass die Inversen in der umgekehrten Reihenfolge der Vorwärtsoperationen ausgeführt werden. Auf der syntaktischen Ebene des Schedules beschreibt die Expansion des Schedules dieses Vorgehen, d.h. im expandierten Schedule stehen statt der Abbruchoperation die inversen Operationen in umgekehrter Reihenfolge und anschließend die Commitoperation. Wir betrachten dazu folgenden Schedule als Beispiel:

$$r_1(x)\ r_1(y)\ w_1(y)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2$$

Hier liest t_2 x von t_1, obwohl t_1 später zurückgesetzt wird. Diese Ausführung wird man also nicht als korrekt ansehen. Die Expansion dieses Schedules sieht wie folgt aus:

$$r_1(x)\ r_1(y)\ w_1(y)\ w_1(x)\ r_2(x)\ w_1^{-1}(x)\ w_1^{-1}(y)\ c_1\ w_2(x)\ c_2$$

Man versucht nun, nicht in Konflikt stehende Operationen so lange zu vertauschen, bis eine inverse Operation unmittelbar neben ihrer Vorwärtsoperation steht, dann kann man dieses Paar aus dem Schedule entfernen, da sich ihre Effekte gerade aufheben. Leseoperationen von abgebrochenen Transaktionen kann man immer entfernen, da sie keine Effekte auf die Datenbank haben. Eine solcher expandierter Schedule ist dann korrekt, wenn sich alle Schreiboperationen von abgebrochenen Transaktionen auf diese Weise entfernen lassen und der resultierende Schedule serialisierbar ist. Am Beispiel sieht man, dass man zwar $w_1^{-1}(y)$ nach links bis zur entsprechenden Schreiboperation kommutieren kann, aber $w_1^{-1}(x)$ wird durch $r_2(x)$ blockiert, der Schedule ist also nicht korrekt. Dieses Vorgehen führt auf die Klasse der *reduzierbaren* Schedules [SWY93]:

Definition 2.18: (RED, PRED)

Ein Schedule s heißt *reduzierbar*, wenn seine Expansion exp(s) durch endlich viele Anwendung der folgenden Regeln in einen seriellen Schedule transformiert werden kann:

(i) Kommutativitätsregel (KR):
Zwei Operationen $p,q \in op(exp(s))$, die bezüglich $<$ direkt benachbart sind mit $p<q$, können genau dann vertauscht werden, wenn sie nicht in Konflikt stehen.

(ii) Undo-Regel (UR):
Zwei Operationen $p,q \in op(exp(s))$, die bezüglich $<$ direkt benachbart sind mit $p<q$, und die Inverse voneinander sind (d.h., $p=w_i(x)$ sowie $q=w_i^{-1}(x)$ für ein x und ein i), können aus dem Schedule entfernt werden.

(iii) Null-Regel (NR):
Leseoperationen von aktiven oder abgebrochenen Transaktionen im Schedule s können aus dem Schedule entfernt werden.

(iv) Ordnungsregel (OR):
Zwei kommutative, ungeordnete Operationen können beliebig geordnet werden.

Die Menge aller reduzierbaren Schedules heißt *RED*.

Ein Schedule heißt *präfix-reduzierbar*, wenn alle Präfixe des Schedules reduzierbar sind. Die Menge alle präfix-reduzierbaren Schedules heißt *PRED*. ■

Mit (Präfix-)Reduzierbarkeit hat man ein Kriterium, mit dem man entscheiden kann, ob in einem gegebenen Schedule alle abgebrochenen Transaktionen korrekt zurückgesetzt werden können. In der Praxis ist man aber eher an Kriterien interessiert, die es erlauben, beim Absetzen einer Operation zu entscheiden, ob der resultierende Schedule korrekt bleibt. Man kommt so zu verschiedenen Klassen von erlaubten Schedules:

- Ein Schedule ist *rücksetzbar*, wenn für jede Transaktion t_i, die mit Commit beendet wurde und von einer anderen Transaktion t_j liest, $c_j<c_i$ gilt. Man vermeidet damit den Effekt, dass eine mit Commit beendete Transaktion von einer zurückgesetzten liest. Die Klasse aller rücksetzbaren Schedules heißt *RC* [Hadz88].
- Ein Schedule *vermeidet kaskadierende Abbrüche*, wenn jede Transaktion nur von bereits mit Commit beendeten Transaktionen liest. Andernfalls müsste die lesende Transaktion abgebrochen werden, wenn die Transaktion, von der sie liest, selbst zurückgesetzt wird, der Transaktionsabbruch hätte sich also fortgepflanzt. Die Klasse dieser Schedules heißt *ACA* [Hadz88].
- Ein Schedule heißt *strikt*, wenn zwischen einer Schreiboperation einer Transaktion t_i und einer folgenden Operation einer anderen Transaktion die Transaktion t_i mit Commit oder Abort beendet werden muss. Die Klasse aller strikten Schedules heißt *ST* [Hadz88].
- Ein Schedule heißt *rigoros*, wenn zwischen zwei Operationen verschiedener Transaktionen, die in Konflikt stehen, die erste Transaktion mit Commit oder Abort beendet sein muss. Die Klasse aller rigorosen Schedules heißt *RG* [BGRS91].
- Ein Schedule heißt *log-rücksetzbar*, wenn er rücksetzbar ist und für jeden Konflikt der Form $w_i(x)<w_j(x)$ im Schedule

- $a_i<w_j(x)$ oder $c_i<c_j$, falls t_j mit Commit beendet wird, oder

- $a_j<a_i$ sonst

gilt. Die Klasse aller log-rücksetzbaren Schedules heißt *LRC* [AAE94].

Die einzelnen Klassen stellen immer schärfere Anforderungen an erlaubte Schedules; man kann zeigen, dass RG⊂ST⊂ACA⊂RC und ST⊂LRC, wobei alle Inklusionen echt sind [WV01]. Praktisch relevant ist dabei vor allem RG, denn man kann zeigen, dass die Schedules, die von starkem 2PL erzeugt werden, gerade die rigorosen sind. Die übrigen sind keine Teilmenge von CSR; man sieht also, dass Serialisierbarkeit und korrektes Zurücksetzen abgebrochener Transaktionen orthogonale Eigenschaften von Schedules sind.

Das Ziel dieser Ansätze ist es, ein Kriterium zu entwickeln, mit dem man zur Laufzeit (präfix-)reduzierbare Schedules garantieren kann. Das folgende Theorem [AAE94, WV01] zeigt, dass man dazu neben Serialisierbarkeit nur die LRC-Bedingungen sicherstellen muss, und erlaubt somit die Konstruktion solcher Scheduler.

Theorem 2.6:

PRED = CSR ∩ LRC ■

2.5.2 Erweiterung auf Mehrversionenschedules

Bisher gibt es praktisch keine Arbeiten, die das Rücksetzen von Transaktionen in Mehrversionenschedules betrachten, obwohl existierende Systeme Versionen von Objekten zur Performancesteigerung benutzen. Offensichtlich gibt es dabei keine Probleme mit dem Rücksetzen von Transaktionen. Wir entwickeln im folgenden eine Theorie, die zeigt, dass dies kein Zufall ist, sondern dass sich solche Mehrversionsschedules, wie sie praktische Systeme erzeugen, problemlos zurücksetzen lassen. Wir werden dazu die Definitionen aus dem vorherigen Abschnitt auf Mehrversionsschedules erweitern. Um die Argumentation einfacher zu machen, zeigen wir die Ergebnisse für Schedules aus MCSR; nach Theorem 2.3 gelten sie dann auch für Schedules aus MVSR, da wir in unserem Modell blindes Schreiben verbieten.

Zunächst müssen wir eine Schreibweise für die Inverse einer Schreiboperation $w_i(x_i)$ definieren. Wir schreiben sie als $w_i^{-1}(x_i)$ und meinen damit, dass die von Transaktion t_i geschriebene Version gelöscht wird. Die Expansion exp(s) eines Mehrversionenschedules wird analog wie im Einversionsfall definiert. Auf dieser Basis können wir nun das Konzept der Reduzierbarkeit (RED) auf Mehrversionenschedules übertragen.

Definition 2.19: **(MC-RED)**

Ein Mehrversionenschedule s heißt *mehrversions-reduzierbar*, wenn seine Expansion exp(s) durch endlich viele Anwendung der folgenden Regeln in einen seriellen Einversionenschedule transformiert werden kann:

(i) Kommutativitätsregel (KR):

Zwei Operationen $p,q \in op(exp(s))$, die bezüglich $<$ direkt benachbart sind mit $p<q$, können genau dann vertauscht werden, wenn p keine Lese- oder q keine (normale oder inverse) Schreiboperation auf dem gleichen Objekt ist.

(ii) Undo-Regel (UR):

Zwei Operationen $p,q \in op(exp(s))$, die bezüglich $<$ direkt benachbart sind mit $p<q$, und die Inverse voneinander sind (d.h., $p=w_i(x_i)$ sowie $q=w_i^{-1}(x_i)$ für ein x und ein i), können aus dem Schedule entfernt werden.

(iii) Null-Regel (NR):

Leseoperationen von aktiven oder abgebrochenen Transaktionen im Schedule s können aus dem Schedule entfernt werden.

(iv) Ordnungsregel (OR):

Zwei kommutative, ungeordnete Operationen können beliebig geordnet werden.

Die Menge aller mehrversions-reduzierbaren Schedules heißt *MC-RED*. ■

Die Kommutativitätsregel KR entspricht gerade der Vertauschbarkeitsregel in der Definition von MCSR. Wir betrachten das folgende Beispiel:

$$r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ c_2\ a_1$$

Wenn man diesen Schedule expandiert und dann die Regeln zur Reduktion anwendet, erhält man (nach Vertauschen von $w_1(x_1)$ und $r_2(x_1)$ und anschließendem Undo) den Schedule

$$r_2(x_1)\ c_2\ c_1$$

Das ist offensichtlich zwar ein serieller, aber kein gültiger Einversionenschedule, da eine nichtexistente Version von x (nämlich x_1) gelesen wird. Da dies die einzige Möglichkeit der Reduktion ist, ist der Ausgangsschedule also nicht mehrversions-reduzierbar.

Analog zu PRED im letzten Abschnitt definieren wir nun noch *MC-PRED* als die Menge der Mehrversionenschedules, für die alle Präfixe mehrversions-reduzierbar sind.

Nach diesen eher algorithmischen Kriterien betrachten wir nun die syntaktischen Kriterien Rücksetzbarkeit (RC), Vermeidung kaskadierender Abbrüche (ACA), Striktheit (ST), Rigorosität (RG) und Log-Rücksetzbarkeit (LRC). Im Gegensatz zum Fall ohne Versionen kommt es beim Schreiben eines Objektes zu keinem Überschreiben des alten Werts, da jedes Schreiben eine neue Version anlegt. Dies gilt natürlich nur, solange es keine Beschränkung der Anzahl der Versionen eines Objektes gibt. Zwei Schreiboperationen stehen also in diesem Sinn nicht in Konflikt. Analog sind eine Lese- und eine Schreiboperation nur dann in Konflikt, wenn sie mit der gleichen Version arbeiten, wenn also die Leseoperation die Version liest, die die Schreiboperation vorher erzeugt hat. Konfliktrelation und Liest-von-Relation fallen hier also zusammen.

Man sieht, dass sich die Definitionen für RC und ACA aus dem letzten Abschnitt ohne weiteres auf Mehrversionenschedules übertragen lassen, da sie ausschließlich auf der Liest-von-Beziehung basiert sind. Striktheit verbietet zusätzlich zu ACA das Überschreiben von Objekten, wenn die Transaktion, die das Objekt zuletzt geändert hat, noch nicht beendet ist. Im Mehrversionsfall werden (ohne Beschränkung der Versionszahl) aber keine Objekte überschrieben, sondern neue Versionen erzeugt, ST fällt hier also mit ACA zusammen. Mit einem analogen Argument kann man zeigen, dass in diesem Fall RG und ST zusammenfallen. Weil es außerdem keine Konflikte zwischen Schreiboperationen gibt, fallen auch RC und LRC zusammen. Es gilt also der folgende Satz:

Theorem 2.7:

Für Mehrversionenschedules (ohne Begrenzung der Anzahl der Versionen) gilt:

(i) RC=LRC
(ii) ACA=ST=RG ■

Ähnlich wie in Theorem 2.6 im vorherigen Abschnitt kann man folgenden Zusammenhang zwischen präfixreduzierbaren Mehrversionenschedules einerseits und rücksetzbaren, mehrversions-konfliktserialisierbaren Schedules andererseits zeigen:

Theorem 2.8:

Ein Mehrversionenschedule s ist genau dann präfixreduzierbar, wenn er mehrversions-konfliktserialisierbar und rücksetzbar ist, d.h.

MC-PRED = MCSR $\cap$ RC. ■

Zum Beweis dieses Satzes argumentieren wir ähnlich wie im Beweis von Theorem 2.6 in [WV01] und zeigen zunächst das folgende Lemma:

Lemma 2.1:

Für einen Mehrversionenschedule $s \in$ RC können alle Operationen von Transaktionen, die nicht mit Commit beendet wurden, durch Anwenden der Regeln KR, UR, NR und OR aus exp(s) entfernt werden. ■

Beweis:

Die Behauptung ist klar für Leseoperationen von nicht mit Commit beendeten Transaktionen, da sie nach der Nullregel unmittelbar aus exp(s) entfernt werden dürfen. Wir müssen das Lemma also nur für Schreiboperationen solcher Transaktionen zeigen. Sei also t_i eine solche Transaktion, so dass $w_i(x_i) \in op(s)$. Wir nehmen zunächst an, dass $w_i(x_i)$ die letzte Operation in s ist. In exp(s) steht dann $w_i^{-1}(x_i)$ unmittelbar nach $w_i(x_i)$, so dass das Paar gemäß der Undo-Regel gelöscht werden kann. Ist $w_i(x_i)$ nicht die letzte

Operation in s, so kann $w_i(x_i)$ unter Anwendung der Kommutativitätsregel so lange nach rechts geschoben werden, bis es unmittelbar vor $w_i^{-1}(x_i)$ steht, denn Schreiboperationen darf man immer nach rechts kommutieren. Dann kann man, wie gerade gezeigt, $w_i(x_i)$ und $w_i^{-1}(x_i)$ durch Anwenden der Undoregel löschen.

Es bleibt noch zu zeigen, dass der Restschedule immer noch gültig ist. Sei dazu t_j eine Transaktion aus s mit $r_j(x_i) \in op(s)$. Weil $s \in RC$ und t_i nicht mit Commit beendet wurde, kann dann auch t_j nicht mit Commit beendet worden sein. Dann kann die Leseoperation aber durch Anwenden der Nullregel aus dem Schedule entfernt werden, der übrigbleibende Schedule ist dann gültig. ■

Beweis von Theorem 2.8:

"$\supseteq$": Sei $s \in MCSR \cap RC$. Wir nehmen an, dass $s \notin$ MC-PRED. Dann gibt es ein Präfix s' von s, das nicht reduzierbar ist. Aus der Definition von RC folgt, dass RC präfix-abgeschlossen ist, MCSR ist nach Korollar 2.1 präfix-abgeschlossen; es gilt also $s' \in MCSR \cap RC$. Nach Lemma 2.1 können alle Operationen von nicht mit Commit beendeten Transaktionen aus s' entfernt werden, sei s" das Ergebnis dieses Vorgangs. Jetzt enthält s" genau die mit Commit beendeten Transaktionen von s'. Da $s' \in MCSR$, kann s" durch endliche Anwendung von Regel KR in einen äquivalenten seriellen Einversionenschedule umgewandelt werden. Also ist $s' \in$ MC-PRED, im Widerspruch zur Annahme.

"$\subseteq$": Sei $s \in$ MC-PRED. Wir nehmen an, dass $s \notin MCSR \cap RC$. Wenn $s \notin MCSR$ wäre, könnte s auch nicht in MC-PRED sein, da die Vertauschungsregel von MC-PRED die gleiche wie die von MCSR ist und in CP(s) nur mit Commit beendete Transaktionen enthalten sind. Es genügt daher, $s \notin RC$ zu betrachten. Wir nehmen also an, s enthalte ein Operationspaar $w_i(x_i)$ und $r_j(x_i)$, außerdem werde t_j mit Commit beendet. Wegen $s \notin RC$ wird entweder t_i (nach $r_j(x_i)$) mit Abort beendet, oder das Commit von t_i erfolgt nach dem Commit von t_j. Im ersten Fall entsteht durch die Expansion von s der Teilschedule $w_i(x_i) \ldots r_j(x_i) \ldots w_i^{-1}(x_i)\ c_i$. Nach der Reduktion bleibt $r_j(x_i)$ stehen, was keine gültige Operation mehr ist, da die Version x_i nicht mehr erzeugt wird. Der Schedule ist also nicht mehrversions-präfix-reduzierbar, was im Widerspruch zur Annahme steht. Im zweiten Fall betrachten wir den Präfix von s unmittelbar vor c_i. Er enthält die gleiche Operationsfolge wie oben, und da jetzt c_i nicht mehr im Schedule ist, greift nach seine Expansion das gleiche Argument; auch dieser Schedule ist nicht präfix-reduzierbar. Wegen dieses Widerspruchs muss die Hypothese $s \notin$ RC falsch gewesen sein, was die Behauptung zeigt. ■

2.5.3 Rücksetzen in Mehrschichtentransaktionen

In Abschnitt 2.2.4 haben wir gezeigt, dass es Mehrschichtentransaktionen ermöglichen, Subtransaktionen nach der Ausführung einer semantisch reichen Operation mit Commit zu beenden und damit Sperren auf den unteren Levels frühzeitig freizugeben, so dass potentiell eine höhere Parallelität erzielt werden kann. Wenn nun die Transaktion abgebrochen wird, müssen wegen der Atomarität ihre Effekte vollständig rückgängig gemacht werden. Subtransaktionen, die gerade bearbeitet werden, werden auf L_0 mit den Mitteln, die wir in den beiden vorherigen Abschnitten vorgestellt haben, zurückgesetzt. Wenn aber Subtransaktionen der Transaktion bereits beendet waren, stehen ihre Effekte bereits in der Datenbank. Das System muss daher, ähnlich wie im ungeschichteten Fall, *inverse Operationen* erzeugen, um diese Subtransaktionen zurückzusetzen. Abbildung

2.3 zeigt ein Beispiel, in dem Transaktion t_1 abgebrochen wird, während sie gerade B dekrementiert, also bevor die Subtransaktion t_{12} beendet ist. Die Aktionen, die das System ausführt, um t_1 zurückzusetzen, sind in Abbildung 2.4 dargestellt. Um t_{12} zurückzusetzen, wird auf L_0 eine inverse Operation für die Schreiboperation $w_{12}(y)$ erzeugt, die den Effekt der Operation rückgängig macht. Das Inkrementieren von A durch t_1 war bereits beendet, also muss das System eine inverse Operation erzeugen, die selbst als Subtransaktion t_{13} ausgeführt wird und in diesem Fall A wieder um den gleichen Wert dekrementiert. Nachdem alle Operationen von t_1 rückgängig gemacht wurden, wird t_1 mit Commit beendet.

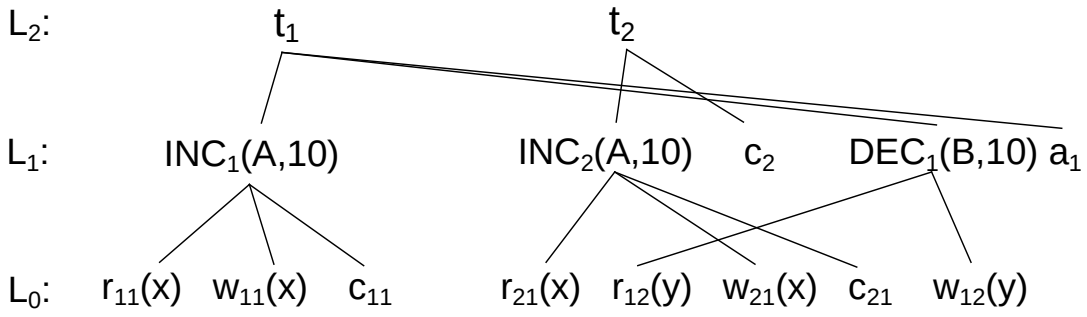

Abbildung 2.3 - Transaktionsabbruch in einem Mehrschichtenschedule

Formal definiert man ähnlich wie im ungeschichteten Fall die Expansion eines Mehrschichtenschedules, in der die Abort-Operation einer Transaktion durch eine Folge von inversen Operationen aller Subtransaktionen ersetzt wird, die zum Zeitpunkt des Abbruchs bereits beendet waren. Auch hier kann man wieder das Konzept der Reduzierbarkeit, hier in Gestalt der *Baum-Präfix-Reduzierbarkeit*, definieren; man nennt die Mehrschichtenschedules baum-präfix-reduzierbar, bei denen die Expansion aller Präfixe baumreduzierbar ist [WV01].

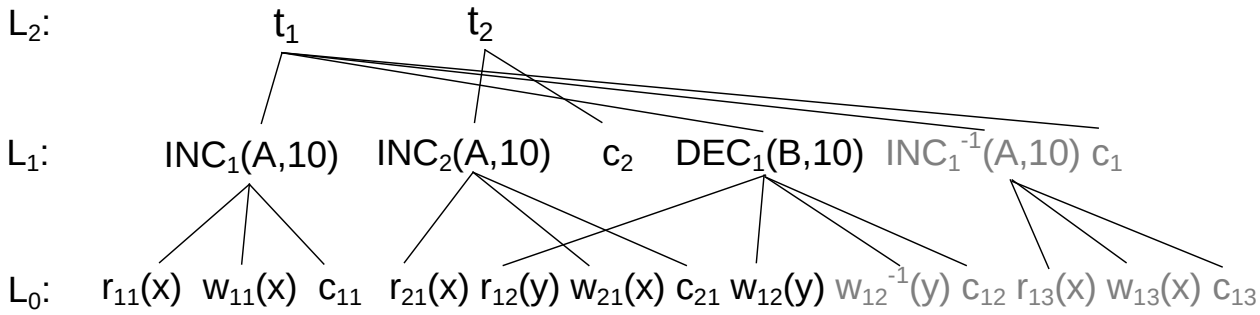

Abbildung 2.4 - Rücksetzen einer Mehrschichtentransaktion durch inverse Operationen

2.5.4 Einfluss von Isolation Levels

In Abschnitt 2.5.1 hatten wir gezeigt, dass Serialisierbarkeit und Rücksetzbarkeit orthogonal sind. Insbesondere bedeutet das, dass man durch die eingeschränkte Korrektheit, die Isolation Levels bieten, nicht automatisch Rücksetzbarkeit verlieren muss. In diesem Abschnitt untersuchen wir daher, welche Auswirkungen die verschiedenen Isolation Levels auf die Rücksetzbarkeit von Schedules haben.

Wir betrachten zunächst Dirty Writes, die in praktisch allen Systemen und allen Isolation Levels verboten sind. Dies liegt unter anderem daran, dass sie eine automatische Recovery unmöglich machen. Um dies klarzumachen, sehen wir uns den Schedule $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ a_1$ an, der ein typisches Dirty Write enthält. Wenn nun t_1 abbricht, ist der Schedule nicht reduzierbar, er ist auch nicht log-rücksetzbar, weil t_2 sonst vor t_1 abgebrochen werden müsste. Tatsächlich hat das System keine Möglichkeit, den von t_1 geschriebenen Wert durch einen alten Wert zu ersetzen, ohne den von t_2 geschriebenen Wert zu überschreiben. Und sollte t_2 später auch abgebrochen werden, ist vollkommen unklar, welchen Wert das System an x zuweisen soll.

Isolation Levels, die zwar Dirty Writes vermeiden, aber Dirty Reads erlauben (z.B. Read Uncommitted), machen ebenfalls Probleme bei der Recovery. Ein typischer Schedule ist z.B. $r_1(x)\ w_1(x)\ r_2(x)\ a_1$, den wir schon mehrmals gesehen haben: Transaktion t_2 liest einen Wert, den es nie gegeben hat, da t_1 abgebrochen wurde. Isolation Levels, die Dirty Reads erlauben, sind also a priori nicht rücksetzbar (und nicht reduzierbar, wie wir in Abschnitt 2.5.1 gesehen haben). Um diese Eigenschaft zu erhalten, muss ein solches System also dafür sorgen, dass Transaktionen wie t_2 im Beispiel abgebrochen werden, wenn die Transaktionen, von denen sie gelesen haben, zurückgesetzt werden. Dann sind die entstehenden Schedules sogar log-rücksetzbar, weil Dirty Writes verboten sind.

Werden Dirty Writes und Dirty Reads vermieden, sind die erlaubten Schedules nach der bisherigen Argumentation log-rücksetzbar und damit von der Recoveryseite unproblematisch, außerdem vermeiden sie kaskadierende Aborts. Insbesondere erhalten wir damit $SI \subset LRC \cap ACA$; trotzdem gilt $SI \not\subset$ MC-PRED, weil $SI \not\subset$ MCSR.

2.6 Transaktionsverwaltung in existierenden Datenbanksystemen

In diesem Abschnitt beschreiben wir die Transaktionsverwaltung existierender Datenbanksysteme und ordnen die Schedules, die sie generieren, in die theoretischen Klassen ein, die wir in diesem Kapitel vorgestellt haben. Wir werden diese Informationen im nächsten Kapitel nutzen, um die Einsetzbarkeit von Algorithmen für verteilte Transaktionen in realen Systemumgebungen beurteilen zu können.

2.6.1 Oracle 8i

Oracle 8i [Orac99c] bietet die Optionen "*Read Committed"* und "*Serializable"* für Transaktionen. Vor der ersten Operation einer Transaktion muss der gewünschte Modus gewählt werden.

Das Datenbanksystem verwendet einen Mehrversionsscheduler, der auf dem ROMV-Protokoll basiert. Voreingestellt für alle Transaktionen ist der Modus "Read Committed", der dem Isolation Level Multiversion Read Committed aus Abschnitt 2.3.2.5 entspricht. Dirty Writes werden durch Schreibsperren auf zu ändernden Objekten vermieden, die bis zum Ende der Transaktion gehalten werden; Oracle verwaltet immer nur eine "aktuelle" Version eines Objektes und mehrere "alte", die von bereits beendeten Transaktionen geschrieben wurden. Dazu werden die alten Versionen (bzw. die Datenbankseiten, in denen sie liegen) zusammen mit den Commitzeitpunkten der Transaktionen, die sie geschrieben haben, in sogenannten Rollback-Segmenten aufbewahrt, die gleichzeitig beim Zurücksetzen von Transaktionen benutzt werden. Die verschiedenen

Versionen einer Seite in den Rollback-Segmenten sind verkettet, so dass bei einem Lesezugriff einer Transaktion diese Kette solange entlang gelaufen werden kann, bis die zeitlich richtige Version gefunden wurde.

Im Modus "Serializable" realisiert Oracle Snapshot Isolation mit der First-Writer-Wins-Strategie, die durch Sperren realisiert ist. Vor jedem Schreibzugriff auf ein Objekt fordert die Transaktion automatisch eine exklusive Sperre darauf an, sollte eine andere Transaktion das Objekt bereits gesperrt haben, muss sie warten: Wird die andere Transaktion mit Commit beendet, wird die wartende abgebrochen, andernfalls erhält sie die Sperre und darf weiterarbeiten. Gleichzeitig wird geprüft, ob eine bereits beendete Transaktion, die parallel zu dieser Transaktion gelaufen ist, dieses Objekt geändert hat. Wenn dies der Fall ist, wird die Transaktion abgebrochen. Durch diese beiden Maßnahmen wird sichergestellt, dass parallel laufende Transaktionen nie das gleiche Objekt verändern können. In den Oracle-Handbüchern werden die Anomalien, die durch Snapshot Isolation auftreten können, beschrieben. Zu ihrer Behebung werden Maßnahmen auf der Ebene der Anwendungsprogramme vorgeschlagen, die in der Regel im Anfordern zusätzlicher Schreibsperren durch die Anwendungsprogramme ("select for update") bestehen. Dieser "Ausweg" belastet also die Anwendungsentwickler und reduziert deren Produktivität.

Für reine Lesetransaktionen bietet Oracle den Modus "read-only" an, in dem die Transaktion genau den Zustand der Datenbank zum Zeitpunkt ihres Starts sieht. Dieser Modus wird genau wie der "serializable"-Modus realisiert; er stammt aus früheren Versionen von Oracle, in denen es für Transaktionen, die auch schreiben, nur den Level "read committed" gab.

Garantierte Serialisierbarkeit kann nicht durch den Datenbankscheduler, sondern nur auf Anwendungsebene durch explizite Sperranforderungen hergestellt werden. In früheren Versionen gab es die Option, durch Tabellensperren Serialisierbarkeit zu gewährleisten; da dies hohe Performancenachteile hat, wurde diese Option mit Release 8i abgeschafft.

Änderungen, die Transaktionen vornehmen, werden sofort auf der Datenbank ausgeführt, gleichzeitig wird die Änderung im Redo-Log vermerkt und der alte Inhalt jeder geänderten Seite in ein Rollback-Segment geschrieben. Alte Versionen von Seiten werden über eine Verkettung der Einträge in den Rollbacksegmenten gefunden, ebenso wie die Änderungen, die eine Transaktion vorgenommen hat. Das Rücksetzen einer Transaktion geschieht, indem alle geänderten Seiten in den Rollback-Segmenten gefunden und zurückgenommen werden.

2.6.2 Microsoft SQL Server 2000

SQL Server 2000 [Micr99] verwendet ein Einversionsverfahren zur Concurrency Control. Obwohl die Dokumentation auch davon spricht, dass optimistische Techniken realisiert sein sollen, wird nur ein auf Sperren basierter Algorithmus beschrieben, der gemeinsame und exklusive Sperren auf Prädikaten in Form von Indexeinträgen, Records, Seiten und Tabellen sowie intentionale Sperren auf Seiten und Tabellen verwendet. SQL Server 2000 bietet alle in Abschnitt 2.3.2.2 über Sperren definierte Isolation Levels. Der vom System vorgegebene Level ist Locking Read Committed. Im höchsten Isolation Level Serializable wird Serialisierbarkeit durch starke Zweiphasensperren gewährleistet, so dass die Klasse der von SQL Server 2000 erzeugten Schedules eine Teilmenge von COCSR ist.

Änderungen, die Transaktionen vornehmen, werden zunächst in eine Logdatei eingetragen und dann in der stabilen Datenbank ausgeführt. Andere Transaktionen, die zur gleichen Zeit aktiv sind, können diese Änderungen wegen der verwendeten Sperren nur sehen, wenn sie im Isolation Level Read Uncommitted ausgeführt werden. Wird die Transaktion später zurückgesetzt, werden ihre Änderungen anhand des Logs ungeschehen gemacht, kaskadierende Abbrüche anderer Transaktionen werden aber nicht ausgelöst.

2.6.3 IBM DB2 Universal Database Release 7

DB2 Universal Database [IBM00b] verwendet ein sperrbasiertes Einversionsverfahren zur Concurrency Control. Neben den üblichen gemeinsamen und exklusiven Sperrmodi auf Prädikatsperren in Form von Sperren von Indexeinträgen sowie Sperren auf Record- und Tabellenebene sowie den intentionalen Sperren auf Tabellen verwendet es zusätzliche Sperrmodi zur Concurrency Control auf Indexstrukturen. Ein Mechanismus zur Sperreskalation konvertiert Recordsperren einer Transaktion in Tabellensperren, wenn die Zahl der Recordsperren einen einstellbaren Schwellwert übersteigt. DB2 bietet Varianten der in Abschnitt 2.3.2.2 vorgestellten Isolation Levels, die aber anders benannt sind (Serializable wird hier z.B. Repeatable Read genannt). Auch der Isolation Level Cursor Stability, der in Abschnitt 2.3.2.4 eingeführt wurde, wird von DB2 unterstützt. Der vom System vorgegebene Level ist Cursor Stability. Im höchsten Isolation Level Repeatable Read wird Serialisierbarkeit durch starke Zweiphasensperren gewährleistet, so dass die Klasse der von DB2 erzeugten Schedules eine Teilmenge von COCSR ist.

Änderungen, die Transaktionen vornehmen, werden zunächst in eine Logdatei eingetragen und dann in der stabilen Datenbank ausgeführt. Andere Transaktionen, die zur gleichen Zeit aktiv sind, können diese Änderungen wegen der verwendeten Sperren nur sehen, wenn sie im Isolation Level Read Uncommitted ausgeführt werden. Wird die Transaktion später zurückgesetzt, werden ihre Änderungen anhand des Logs ungeschehen gemacht, kaskadierende Abbrüche anderer Transaktionen werden aber nicht ausgelöst.

2.6.4 Informix UniVerse Release 9.6.1

UniVerse [Info00a,Info00b,Info00c] verwendet ein sperrbasiertes Einversionsverfahren zur Concurrency Control. Es bietet dazu Prädikatsperren in Form von Sperren von Indexeinträgen, Record- und Tabellensperren in den Modi gemeinsam und exklusiv sowie intentionale Sperren auf Tabellenebene zusammen mit automatischer Sperreskalation, die Recordsperren einer Transaktion in eine Tabellensperre konvertiert, wenn die Zahl der Recordsperren einen einstellbaren Schwellwert übersteigt. Transaktionen haben zunächst die Freiheit, Sperren zu jedem Zeitpunkt anzufordern und freizugeben, ohne dass das System Korrektheitsgarantien übernimmt. Zusätzlich kann jeder Transaktion ein Isolation Level zugewiesen werden, das System kümmert sich dann selbst um die Anforderung und Freigabe der Sperren. UniVerse bietet alle in Abschnitt 2.3.2.2 über Sperren definierte Isolation Levels. Der vom System vorgegebene Level ist Locking Read Uncommitted, empfohlen wird Locking Read Committed. Im höchsten Isolation Level 4 wird Serialisierbarkeit durch starke Zweiphasensperren gewährleistet, so dass die Klasse der von UniVerse erzeugten Schedules eine Teilmenge von COCSR ist.

Das System erlaubt "geschachtelte" Transaktionen, d.h. ein Transaktionsprogramm kann eine neue Transaktion beginnen, während es bereits eine aktive Transaktion hat. Nach dem Ende einer solchen Subtransaktion erbt die vorher aktive Transaktion die neu erworbenen Sperren. Im Gegensatz zu den in Abschnitt 2.2.4 eingeführten Mehrschichtentransaktionen kann somit keine höhere Parallelität erzielt werden, lediglich das Granulat für das Rücksetzen von Transaktionen wird feiner.

Änderungen von Transaktionen werden zunächst in einem privaten Cache zwischengespeichert und sind somit nicht für andere Transaktionen sichtbar. Erst nach dem Commit einer Transaktion werden ihre Änderungen in die stabile Datenbank geschrieben. Das Rücksetzen von aktiven Transaktionen ist daher sehr einfach, da nur der private Cache verworfen werden muss.

2.6.5 Sybase Adaptive Server Enterprise Release 11.5

Sybase Adaptive Server [Syba97a,Syba97b] verwendet ein sperrbasiertes Einversionsverfahren zur Concurrency Control. Er bietet dazu Prädikatsperren in Form von Sperren von Indexeinträgen, Seiten- und Tabellensperren in den Modi gemeinsam und exklusiv sowie intentionale Sperren auf Tabellenebene zusammen mit automatischer Sperreskalation, die Seitensperren einer Transaktion in eine Tabellensperre konvertiert, wenn die Zahl der Seitensperren einen einstellbaren Schwellwert übersteigt. Jeder Transaktion wird ein Isolation Level zugewiesen, das System fordert die notwendigen Sperren für diesen Isolation Level an und gibt sie zu gegebener Zeit wieder frei. Der Adaptive Server bietet alle in Abschnitt 2.3.2.2 über Sperren definierte Isolation Levels außer Locking Repeatable Read. Der vom System vorgegebene Level ist Locking Read Committed. Zusätzlich können Transaktionen Sperren selbst anfordern und den Isolation Level für eine einzelne Operation wechseln, um z. B. für eine Query die neuesten Daten aus der Datenbank zu erhalten. Im höchsten Isolation Level 3 wird Serialisierbarkeit durch starke Zweiphasensperren gewährleistet, so dass die Klasse der von Adaptive Server erzeugten Schedules eine Teilmenge von COCSR ist.

Änderungen, die Transaktionen vornehmen, werden zunächst in eine Logdatei eingetragen und dann in der stabilen Datenbank ausgeführt. Andere Transaktionen, die zur gleichen Zeit aktiv sind, können diese Änderungen wegen der verwendeten Sperren nur sehen, wenn sie im Isolation Level Read Uncommitted ausgeführt werden. Wird die Transaktion später zurückgesetzt, werden ihre Änderungen anhand des Logs ungeschehen gemacht, kaskadierende Abbrüche anderer Transaktionen werden aber nicht ausgelöst.

2.6.6 Ardent Technologies O_2 Release 5

Das objektorientierte Datenbanksystem O_2 [Unid98], das mittlerweile nicht mehr vertrieben wird, hat im Vergleich zu den bisher vorgestellten relationalen bzw. objektrelationalen Systemen eine recht einfache Unterstützung für Transaktionen. Es verwendet ein sperrbasiertes Einversionenverfahren, das nur ansatzweise dokumentiert wird. Transaktionen können entweder reine Lesetransaktionen sein, dann lesen sie Objekte in der Datenbank, ohne Sperren zu verwenden; dies ist gleichzeitig die voreingestellte Variante und entspricht ungefähr dem Isolation Level Read Committed. Transaktionen, die Änderungen an der Datenbank vornehmen wollen, müssen entsprechend markiert werden und wenden dann ein zweiphasiges Sperrprotokoll an, das gemeinsame und exklu-

sive Sperren auf Objekten verwendet. Änderungen von Transaktionen werden erst beim Commit in die Datenbank geschrieben. Lesetransaktionen können zu ändernden Transaktionen konvertiert werden, wobei das System dann die aktuellen Werte aller bisher gelesenen Objekte erneut aus der Datenbank liest, jetzt unter Verwendung von Lesesperren; das Transaktionsprogramm muss selbst prüfen, ob sich Werte geändert haben. Explizite Isolation Levels werden nicht unterstützt. Zur Vermeidung von Verklemmungen wird vorgeschlagen, die komplette Transaktion ohne Rücksicht und auch ohne Hinweis auf mögliche Performancenachteile in einen kritischen Abschnitt zu packen; die dazu notwendigen Methoden werden mitgeliefert.

Das System erlaubt "geschachtelte" Transaktionen, d.h. ein Transaktionsprogramm kann eine neue Transaktion beginnen, während es bereits eine aktive Transaktion hat. Die Änderungen, die eine solche Subtransaktion gemacht hat, werden erst nach dem Ende der obersten Transaktion in der Transaktionshierarchie in die Datenbank geschrieben. Damit dieses Vorgehen zu einem korrektem Ablauf führt, muss nach dem Ende einer Subtransaktion die vorher aktive Transaktion die neu erworbenen Sperren erben; die Dokumentation sagt nichts darüber aus, ob das geschieht. Im Gegensatz zu den in Abschnitt 2.2.4 eingeführten Mehrschichtentransaktionen kann durch die Verwendung dieser geschachtelten Transaktionen keine höhere Parallelität erzielt werden, lediglich das Granulat für das Rücksetzen von Transaktionen wird feiner.

Weil die Änderungen, die eine Transaktion vorgenommen hat, erst bei ihrem Commit in die stabile Datenbank geschrieben werden, ist das Rücksetzen einer abgebrochenen Transaktion sehr einfach.

2.7 Zusammenfassung

Wir haben in diesem Kapitel theoretische Grundlagen und praktische Algorithmen zur Gewährleistung von Atomarität und Isolation in zentralisierten Datenbanksystemen vorgestellt. Neben wichtigen, etablierten Konzepten haben wir dabei neuartige Ansätze zur Charakterisierung von Snapshot Isolation und den Bezügen zu Mehrversionsserialisierbarkeit sowie zur Rücksetzbarkeit von Mehrversionsschedules entwickelt. Beispiele realer Systeme haben gezeigt, dass die theoretischen Konzepte praktisch umsetzbar sind.

Wir haben auch gesehen, dass in der Praxis häufig eingeschränkte Korrektheitskriterien in Form schwächerer Isolation Levels eingesetzt werden, die potentiell eine höhere Performance ermöglichen, aber keine vollständige Isolation mehr gewährleisten. Wir haben mögliche Probleme aufgezeigt, die beim Einsatz von Isolation Levels entstehen können. Vielen Programmierern in der Praxis sind diese Schwierigkeiten nicht geläufig, so dass die Gefahr besteht, dass Anwendungen unerwartete Ergebnisse liefern. Wir haben verschiedene Ansätze zur Definition von Isolation Levels verglichen und neue Isolation Levels für Mehrversionenscheduler definiert. Für den weitverbreiteten Isolation Level Snapshot Isolation, der z.B. vom Marktführer Oracle verwendet wird, haben wir eine graphbasierte Charakterisierung angegeben und Algorithmen entwickelt, mit denen man selbst auf Systemen, die nur Snapshot Isolation gewährleisten können, serialisierbare Schedules sicherstellen kann.

3 Transaktionen in föderierten Datenbanksystemen

3.1 Motivation

Genau wie in zentralisierten Datenbanken möchte man in föderierten Datenbanken zusammengehörende Aktionen in Transaktionen kapseln und sowohl ihre korrekte als auch ihre vollständige Ausführung sicherstellen. In vielen Bereichen, in denen Föderationen eine Rolle spielen, sind Transaktionen essentiell für die vorhandenen Anwendungen, z.B. im E-Commerce oder in Reisebuchungssystemen. Typischerweise bedeutet fehlerhafte Ausführung von Transaktionen in diesem Kontexten den Verlust von mehr oder weniger vielen Einnahmen, bzw. das Entstehen von mehr oder weniger hohen Kosten.

Man möchte daher auch für föderierte Transaktionen die üblichen *ACID-Eigenschaften* garantieren, also Atomarität, Konsistenzerhaltung, Isolation und Dauerhaftigkeit. Die Datenbanksysteme, die an der Föderation teilnehmen, gewährleisten bereits die Dauerhaftigkeit der Änderungen von Transaktionen, die mit Commit beendet wurden, so dass dazu keine besonderen Maßnahmen auf der föderierten Ebene notwendig sind. Mechanismen zur Erhaltung der Konsistenz der föderierten Datenbank, vergleichbar zu Triggern und Constraints bei zentralisierten Datenbanken, gehören nicht unmittelbar zur Transaktionsverwaltung, wir betrachten sie daher in dieser Arbeit nicht weiter, sondern beschränken uns auf Atomarität und Isolation.

Ähnlich wie im zentralisierten Fall sind besondere Maßnahmen notwendig, um diese beiden Eigenschaften sicherzustellen. Da die Komponentensysteme in einer Föderation autonom sind und daher in der Regel keine Unterstützung für eine Zusammenarbeit mit anderen Systemen bieten, müssen alle Algorithmen und Protokolle alleine auf der föderierten Ebene angesiedelt werden. Als zusätzliche Schwierigkeit kommt die Heterogenität der beteiligten Systeme hinzu, die sich unter anderem hinsichtlich der eingesetzten Verfahren zur Concurrency Control stark unterscheiden können.

Viele existierende Produkte, die föderierte Transaktionen unterstützen, konzentrieren sich ausschließlich auf die Gewährleistung von Atomarität, insbesondere werden oft verteilte Transaktionen mit verteilter Atomarität gleichgesetzt. Neben den zahlreichen TP-Monitoren und verteilten Datenbanksystemen ist das Industriepapier von Ram et.al. über die praktische Anwendung verteilter Transaktionen [RDD99] ein weiteres Beispiel dafür. Ein wichtiger Grund für diese Ansicht ist sicher, dass es sofort einsichtig ist, dass es zu Problemen mit der Atomarität von Transaktionen kommen kann, die auf mehr als eine Datenbank zugreifen: Führt die Transaktion in einer Datenbank ein Commit aus, während sie in der anderen zurückgesetzt wird, ist ihre Atomarität verletzt; man muss also Maßnahmen ergreifen, um diesen Fall nicht eintreten zu lassen. Wie wir in diesem Kapitel noch sehen werden, unterstützen praktisch alle aktuellen Systeme Verfahren zur Gewährleistung globaler Atomarität.

In der Forschung hat man erkannt, dass auch die Isolation föderierter Transaktionen im allgemeinen nicht einfach zu garantieren ist, und dass man daher mehr als nur Verfahren benötigt, die Atomarität garantieren [BS88, BGS92, CHHK+99]. Zahlreiche theoretische Arbeiten beschäftigen sich mit globaler Serialisierbarkeit und stellen Verfahren vor, die globale Serialisierbarkeit sicherstellen. Die notwendigen zusätzlichen Maßnahmen auf der föderierten Ebene hängen dabei stark von den Eigenschaften der beteiligten

Datenbanksysteme ab. In existierenden Produkten sind diese Verfahren allerdings nicht berücksichtigt worden, so dass dort der Anwendungsentwickler selbst dafür sorgen muss, dass globale Serialisierbarkeit gewährleistet wird. Zusätzlich ignorieren die in der Literatur vorgeschlagenen Algorithmen vollständig, dass vielfach eingeschränkte Isolation Levels eingesetzt werden, um die erzielbare Performance unter Inkaufnahme gewisser Konsistenzverluste zu erhöhen, bzw. dass manche realen Produkte serialisierbare Schedules überhaupt nicht erzeugen können. Ein besonders wichtiger Fall dabei ist, dass mindestens eines der beteiligten Komponentensysteme nur Snapshot Isolation garantiert, weil das der stärkste Isolation Level ist, den der Marktführer Oracle gewährleisten kann.

In diesem Fall kann nicht nur die Konsistenz der lokalen Daten, sondern auch die der globalen Daten nicht mehr garantiert werden. Wir zeigen die Probleme, die auftreten können, anhand des Beispiels einer weltweiten Buchhandlung, die Niederlassungen in Deutschland, Frankreich und Spanien hat. Jede dieser Niederlassungen hat ihre eigene lokale Datenbank, die in eine firmeneigene föderierte Datenbank integriert ist. Wir nehmen an, dass die deutsche und französische Datenbank Snapshot Isolation garantieren, während die spanische Datenbank durch starkes Zweiphasensperren konfliktserialisierbare Schedules erzeugt.

Der erste problematische Fall sind föderierte Transaktionen, die sowohl auf Datenbanken zugreifen, die serialisierbare Schedules erzeugen, als auch auf solche, die nur Snapshot Isolation gewährleisten. Nehmen wir zum Beispiel an, dass eine Transaktion t_1 überprüft, wie viele Exemplare es insgesamt von einem bestimmten Buch gibt, etwa um neue Exemplare nachzubestellen. Eine andere Transaktion t_2 ändert gleichzeitig die föderierte Datenbank, nachdem einige Exemplare von der deutschen zur spanischen Niederlassung transportiert wurden, indem sie die Anzahl der Exemplare in der einen Datenbank erhöht und in der anderen verringert. Die entstehende Ausführung könnte etwa so aussehen (die Operationen von t_1 sind kursiv dargestellt, und die Zeit schreitet von oben nach unten fort):

Deutsche Niederlassung (mit SI)	**Spanische Niederlassung (mit SS2PL)**
	t_2: Erhöhen der Anzahl der Exemplare des Buchs
t_1: Bestimmen der Anzahl der Exemplare des Buchs	
t_2: Verringern der Anzahl der Exemplare des Buchs	
	t_2: Commit t_2
	t_1: Bestimmen der Anzahl der Exemplare des Buchs
	t_1: Commit t_1

In der Datenbank der deutschen Niederlassung sieht Transaktion t_1 die Änderungen nicht, die Transaktion t_2 vorgenommen hat. In der Datenbank der spanischen Niederlassung dagegen sieht sie die Änderungen, da t_1 dort warten muss, bis t_2 zum Zeitpunkt ihres Commits ihre exklusive Sperre freigibt. Transaktion t_1 sieht also einen inkonsistenten Zustand der Datenbank, nämlich eine zu hohe Gesamtzahl der Exemplare dieses Buchs, und könnte möglicherweise, basierend auf diesen inkonsistenten Informationen, die Konsistenz der Datenbank noch weiter schädigen, indem sie diesen "falschen" Wert in die Datenbank zurückschreibt. Hätte auch die Datenbank der deutschen Niederlassung starkes 2PL verwendet, könnte dieser Fall nicht eintreten, weil dann eine globale Verklemmung der beiden Transaktionen eingetreten wäre und eine von ihnen zwangsweise zurückgesetzt worden wäre.

Selbst wenn alle lokalen Datenbanken Snapshot Isolation gewährleisten, können Probleme auftreten. In unserem Beispiel könnte es etwa die globale Bedingung geben, dass mindestens eine gewisse Anzahl von Exemplaren eines Buches vorrätig sein muss. Wenn die Gesamtzahl unter diese Grenze fällt, sollen neue Exemplare bestellt werden. Der Programmierer hat der Prozedur, die den Verkauf eines Buchs abwickelt, einen Test hinzugefügt, der die Einhaltung dieser Bedingung prüft. Der folgende Pseudocode beschreibt diesen Test:

```
procedure sell(bookid, copies_sold)
begin
  count1:= select quantity from StoreGermany.stock where id=bookid;
  count2:= select quantity from StoreFrance.stock where id=bookid;
  Wähle eine Niederlassung aus und verringere die dortige Anzahl
  um copies_sold
  if (count1+count2≥limit) and (count1+count2-copies_sold<limit)
  then Löse Nachbestellung des Buchs aus
end
```

Wird nun diese Prozedur zweimal nebenläufig für das gleiche Buch ausgeführt, kommt es zu Problemen, wenn jeweils eine unterschiedliche Niederlassung ausgewählt wird. Nehmen wir an, dass die erste Ausführung die deutsche Niederlassung auswählt und die zweite die französische. Dann sind beide Ausführungen in den lokalen Datenbanken korrekt im Sinne von Snapshot Isolation, denn die beiden nebenläufigen Transaktionen verändern nicht das gleiche Objekt. Es kann aber vorkommen, dass die Gesamtzahl der Exemplare dieses Buchs unter die vorgegebene Grenze fällt, ohne dass es eine der Transaktionen feststellen kann, weil keine Transaktion die Änderungen der nebenläufigen Transaktion sieht. Es kann daher passieren, dass eine Bedingung verletzt wird, die mehr als eine Datenbank umfasst, obwohl die Transaktionen alle Möglichkeiten nutzen, um dies zu erkennen. In diesem speziellen Beispiel würden auch zukünftige Ausführungen dieser Prozedur keine Bücher mehr nachbestellen, sobald die Grenze einmal unterschritten wird.

Natürlich kann ein Anwendungsprogrammierer diese Probleme erkennen und Lösungen finden, um sie zu beheben. Typische Anwendungen aus der Praxis sind aber zu komplex, um alle problematischen Fälle zu entdecken, so dass eine generelle Lösung im föderierten Datenbanksystem die bessere Alternative ist. Insbesondere für unternehmenskritische Anwendungen ist man an Protokollen interessiert, die beweisbar nur korrekte Ausführungen zulassen. Die bisher vorgestellten Lösungen scheitern aber, sobald in den lokalen Datenbanken Isolation Levels verwendet werden.

In diesem Kapitel erweitern wir zunächst die im letzten Kapitel eingeführte Notation, so dass wir auch formal über föderierte Transaktionen argumentieren können. Anschließend stellen wir in Abschnitt 3.3 die verbreiteten Verfahren zur Gewährleistung globaler Atomarität vor. In Abschnitt 3.4 kategorisieren wir die bisher in der Literatur vorgestellten Techniken zur Sicherstellung globaler Serialisierbarkeit, vergleichen sie qualitativ und präsentieren die beiden praktisch relevantesten Techniken detaillierter. In Abschnitt 3.5 stellen wir die sehr rudimentäre Transaktionsunterstützung in zwei etablierten Produkten zur Föderation heterogener Datenbanken vor. Abschnitt 3.6 behandelt die Gewährleistung globaler Serialisierbarkeit, wenn in den lokalen Datenbanken eingeschränkte Isolation Levels verwendet werden, indem existierende Protokolle erweitert und neue entwickelt werden. In den Abschnitten 3.7und 3.8 diskutieren wir, wie man für föderierte Transaktionen Isolation Levels sicherstellen kann, speziell präsentieren

und vergleichen wir verschiedene Algorithmen, die globale Snapshot Isolation garantieren.

3.2 Erweiterung der Notation

Bisher haben wir ausschließlich Transaktionen betrachtet, die nur auf Objekte in einer einzigen Datenbank zugreifen. Damit wir auch über solche Transaktionen argumentieren können, die Objekte aus mehreren Datenbanken verwenden, müssen wir unsere Notation erweitern.

Definition 3.1: (Globale und lokale Transaktionen)

Eine *globale* oder *föderierte Transaktion (GT)* ist eine Transaktion, die auf Objekte in mindestens zwei verschiedenen Datenbanken zugreift. Im Gegensatz dazu benutzt eine *lokale Transaktion (LT)* nur Objekte in einer einzigen Datenbank.

Die Projektion einer globalen Transaktion t_i auf eine Datenbank DB_k ist die Menge der Aktionen von t_i, die sich auf Objekte aus DB_k beziehen, zusammen mit ihrer entsprechenden Ordnung <. Wir nennen diese Projektion *globale Subtransaktion (GST)* und bezeichnen sie mit $t_i^{(k)}$. ■

Nach dieser Definition unterscheiden wir globale und lokale Transaktionen allein auf einer syntaktischen Ebene, basierend auf den Datenbanken, wo sie Operationen initiieren. In der Praxis ist ein zusätzlicher, wesentlicher Unterschied, dass lokale Transaktionen direkt auf das jeweilige Datenbanksystem zugreifen, statt über eine zusätzliche föderierte Softwareschicht zu gehen. Insbesondere sind lokale Transaktionen für das föderierte System nicht ohne weiteres sichtbar und können daher nicht einfach in ein globales Scheduling integriert werden.

Um deutlich machen zu können, auf welche Datenbank eine Operation einer Transaktion zugreift, erweitern wir unsere Notation wie folgt:

Eine Lese- bzw. Schreiboperation von Transaktion t_i in Datenbank DB_k, die auf Objekt x zugreift, bezeichnen wir mit $r_i^{(k)}(x)$ bzw. $w_i^{(k)}(x)$. Die Begin-Pseudooperation und die Commit-Operation von Transaktion t_i in Datenbank DB_k, die immer dann angegeben werden, wenn sie relevant sind, bezeichnen wir entsprechend mit $B_i^{(k)}$ bzw. $C_i^{(k)}$.

Wenn zwei globale Transaktionen t_i und t_j nebenläufig ablaufen, schreiben wir $t_i \parallel t_j$; laufen sie nacheinander ab in der Reihenfolge t_i vor t_j, schreiben wir $t_i < t_j$. Analoge Schreibweisen verwenden wir für die entsprechenden Beziehungen von Subtransaktionen globaler Transaktionen.

Wir nennen den Schedule aus globalen Transaktionen, der bei der Ausführung im föderierten System entsteht, *globalen Schedule*. Die Schedules in den Komponentensystemen heißen entsprechend *lokale Subschedules*, sie enthalten neben den globalen Subtransaktionen auch rein lokale Transaktionen.

3.3 Gewährleistung von Atomarität

3.3.1 Globale Atomarität

Transaktionen in föderierten Datenbanksystemen arbeiten in der Regel mit mehreren Komponentendatenbanksystemen, die selbst Atomarität für Subtransaktionen garantieren. Um die Atomarität von föderierten Transaktionen sicherzustellen, genügt dies aber nicht. Man muss zusätzlich gewährleisten, dass der Ausgang aller Subtransaktionen einer föderierten Transaktion gleich ist: Wird mindestens eine Subtransaktion abgebrochen, so müssen auch alle anderen Subtransaktionen dieser Transaktion abgebrochen werden. Die föderierte Transaktion darf nur dann zum Commit kommen, wenn alle ihre Subtransaktionen erfolgreich ihr Commit ausgeführt haben. Nur so kann sichergestellt werden, dass keine partiellen Änderungen in einem der Komponentensysteme bestehen bleiben und so die Atomarität der globalen Transaktionen verletzt wird.

In den folgenden Abschnitten stellen wir zunächst das verbreitete Zweiphasen-Commitprotokoll zur Gewährleistung globaler Atomarität vor, das von praktisch allen aktuellen Datenbanksystemen unterstützt wird. In Abschnitt 3.3.3 betrachten wir die Atomarität von Transaktionen in Mehrschichtenschedules. Den Abschluss dieses Abschnittes bildet ein qualitativer Vergleich der vorgestellten Verfahren.

3.3.2 Das Zweiphasen-Commitprotokoll

Das *Zweiphasen-Commitprotokoll (2PC)* [Gray78, LSGG+79, LS76] stellt sicher, dass entweder alle Subtransaktionen einer föderierten Transaktion mit Commit beendet oder alle abgebrochen werden; es handelt sich also um ein *atomares Commit-Protokoll (ACP)*. Um dies zu erreichen, müssen sich die beteiligten Komponentensysteme, im folgenden *Teilnehmer* genannt, zunächst auf einen Ausgang der föderierten Transaktion einigen, bevor sie dann in einem zweiten Schritt das erhaltene Ergebnis umsetzen. Die Abstimmung zwischen den Teilnehmern übernimmt dabei ein ausgezeichneter Prozess, der *Koordinator*, der auch identisch mit einem der Teilnehmer sein kann. Das Protokoll verarbeitet den Commitwunsch einer föderierten Transaktion in zwei Phasen:

- In der ersten Phase, der *Stimmphase*, fragt der Koordinator bei den beteiligten Teilnehmern an, ob sie ihre Subtransaktion der föderierten Transaktion mit Commit beenden können. Die Teilnehmer prüfen dies; dazu kann z.B. ein Zyklustest im lokalen Konfliktgraph nötig sein. Anschließend melden sie dem Koordinator, ob ein Commit aus ihrer Sicht möglich ist oder nicht. Ab diesem Zeitpunkt müssen sie garantieren, dass sie diese Entscheidung auch umsetzen können, selbst wenn die lokale Datenbank wegen eines Soft- oder Hardwarefehlers neu gestartet werden müsste; dazu müssen die entsprechenden Logsätze ins lokale Log geschrieben werden. Der Koordinator sammelt die Stimmen der Teilnehmer; haben alle ihre Stimme abgegeben, beginnt die nächste Phase.
- In der zweiten Phase, der *Entscheidungsphase*, entscheidet der Koordinator anhand der abgegebenen Stimmen, ob die föderierte Transaktion mit Commit oder Abort beendet werden soll, und teilt das Ergebnis den Teilnehmern mit. Der Koordinator entscheidet genau dann für Commit, wenn alle Teilnehmer bestätigt haben, dass sie ihre Subtransaktion mit Commit beenden können. Hat mindestens ein Teilnehmer für Abort gestimmt, wird die gesamte föderierte Transaktion zurückgesetzt. Die Teilnehmer setzen dann das Ergebnis der Abstimmung um, nachdem der Koordina-

tor es ihnen mitgeteilt hat, und melden dies dem Koordinator zurück. Erst dann kann der Koordinator alle Informationen über die föderierte Transaktion löschen.

Dieses Protokoll gewährleistet die Atomarität von föderierten Transaktionen, sofern keine Nachrichten verloren gehen und Koordinator sowie Teilnehmer nicht während des Ablaufs des Protokolls durch Fehler neugestartet werden müssen. Da in der Praxis aber genau diese Fälle auftreten können, wird das Zweiphasen-Commitprotokoll um besondere Maßnahmen für Fehlerfälle erweitert. Fällt der Koordinator zum Beispiel aus, wenn er gerade bei den Teilnehmern nach ihrer Entscheidung gefragt hat, kann es passieren, dass einer der Teilnehmer antwortet, während der Koordinator gerade neu gestartet wird und somit diese Nachricht nicht sieht. Nach seinem Neustart wartet er also vergeblich auf die Nachricht dieses Teilnehmers. Um dieses Problem zu beheben, kann der Koordinator, nachdem er eine gewisse Zeit auf Nachrichten gewartet hat, nochmals die Entscheidung aller Teilnehmer anfordern. Auf ähnliche Weise kann man alle auftretenden Fehlerfälle behandeln, Details finden sich in [WV01].

In existierenden verteilten oder föderierten Datenbanksystemen hat sich das Zweiphasen-Commitprotokoll als Mittel zur Gewährleistung der Atomarität verteilter Transaktionen durchgesetzt. In Gestalt des XA-Standards der X/Open-Group [XOpen96] wird es von den gängigen, aktuellen Systemen unterstützt. TP-Monitore wie BEA Tuxedo oder IBM TXSeries und andere Middleware wie der Object Transaction Service OTS von CORBA [OMG97] garantieren die Atomarität von Transaktionen über verschiedenen Datenbanken, indem sie als Koordinator ein Zweiphasen-Commitprotokoll über das XA-Interface der beteiligten Systeme steuern.

Speziell ältere Datenbanksysteme unterstützen dagegen oft das Zweiphasen-Commit-Protokoll nicht, sondern erlauben nur ein unmittelbares Commit oder Abort föderierter Subtransaktionen. Solche Systeme können aber auch Kandidaten für eine Integration in föderierte Datenbanken sein. Um auch zusammen mit solchen Komponentensystemen Atomarität für föderierte Transaktionen garantieren zu können, haben Wolski und Veijalainen in [WV91] vorgeschlagen, einen Agenten auf das Datenbanksystem aufzusetzen, der an Stelle des Datenbanksystems am 2PC teilnimmt. Der Agent stellt dabei sicher, dass er eine globale Subtransaktion nach endlich vielen Versuchen im lokalen System mit Commit beenden kann, wenn er dies dem Koordinator in der Stimmphase mitgeteilt hat. Er muss dazu die Operationen, die die Subtransaktion ausgeführt hat, in ein Log schreiben und sie ggf. erneut ausführen, wenn das Commit in der lokalen Datenbank fehlschlagen sollte. Dies kann z.B. passieren, wenn das lokale Datenbanksystem wegen eines Fehlers neugestartet werden muss oder weil die entstandene lokale Ausführung nicht serialisierbar war. Man muss dabei fordern, dass das lokale Datenbanksystem mindestens rigorose Schedules erlaubt.

3.3.3 Kompensationsaktionen

Wir haben in Abschnitt 2.2.4 Mehrschichtentransaktionen als Mittel zur Gewährleistung von Serialisierbarkeit vorgestellt. Auch in föderierten Datenbanksystemen können sie eingesetzt werden, um globale Serialisierbarkeit sicherzustellen [DSW94, SSW95]. Man verwendet dabei eine zweischichtige Architektur, wobei in Schicht L_1 die Operationen der föderierten Transaktionen und in L_0 die Seitenoperationen in den Komponentensystemen ausgeführt werden. Die Operationen der föderierten Transaktionen sind entweder Methodenaufrufe von Objekten in den lokalen Datenbanken, falls diese das

unterstützen, oder Anweisungen in der Sprache der jeweiligen lokalen Datenbank, also z.B. SQL.

Wie im zentralisierten Fall wird jede solche Operation innerhalb der Komponentendatenbank in einer eigenen Subtransaktion ausgeführt, so dass ihre Änderungen unmittelbar nach Ausführung der Operation in der Datenbank stehen. Sollte die föderierte Transaktion, die diese Operation ausgeführt hat, später abgebrochen werden, müssen ihre Effekte durch eine geeignete *Kompensationsaktion* rückgängig gemacht werden, wie wir das für den zentralisierten Fall bereits in Abschnitt 2.5.3 vorgestellt haben. Als Besonderheit kommt hier noch dazu, dass wir auch für SQL-Anweisungen, die Änderungen an einer lokalen Datenbank vorgenommen haben, Kompensation sicherstellen müssen. Während der Anwendungsentwickler Kompensationsaktionen für Methodenaufrufe bereitstellen muss, können die Operationen zur Kompensation der Effekte von SQL-Anweisungen direkt aus ihnen berechnet werden: Die inverse Operation zu einer Einfügeoperation ist beispielsweise das Löschen des eingefügten Objektes, und zur Kompensation der Änderung eines Attributes eines Tupels muss wieder der alte Wert des Tupels gesetzt werden.

Das föderierte Datenbanksystem muss die Kompensationsaktionen, die zum Ungeschehenmachen der Effekte von Operationen föderierter Transaktionen notwendig sind, vor dem Commit der jeweiligen Subtransaktion auf stabilen Speicher schreiben, damit es über sie verfügen kann, wenn die föderierte Transaktion zurückgesetzt werden muss.

Lokale Transaktionen, die unabhängig vom föderierten System direkt auf die Komponentendatenbanken zugreifen, beeinträchtigen die Atomarität von föderierten Transaktionen nicht. Wir werden aber später sehen, dass sie Einfluss auf die globale Serialisierbarkeit haben können, so dass hier besondere Maßnahmen notwendig werden.

Kompensationsaktionen werden außerdem in Sagas [GS87] und im S-Transaction-Model von Veijalainen et al [VEH92] zur Sicherstellung der Atomarität verteilter Transaktionen eingesetzt. In der Literatur sind außerdem Verfahren beschrieben worden, die auch ohne Mehrschichtentransaktionen über Kompensationsaktionen die Atomarität verteilter Transaktionen sicherstellen; dazu gehören unter anderem die Arbeiten von Levy, Korth und Silberschatz [LKS91a, LKS91b].

Andere Arbeiten setzen darauf, das Commit für eine Komponentendatenbank niemals vor dem definitiven Commit der föderativen Ebene zu initiieren. Dadurch wird die Notwendigkeit für Kompensation auf der föderativen Ebene eliminiert. Dual zu den Verfahren mit Kompensation benötigt man jedoch ggf. Redo-Schritte, um die Änderungen einer erfolgreich beendeten föderativen Transaktionen in den beteiligten Komponentendatenbanken nachzufahren [MR91, HSL94, BST92, WV91, VW92].

3.3.4 Qualitativer Vergleich der Verfahren

Das Zweiphasen-Commitprotokoll hat den großen Vorteil, dass es durch den XA-Standard zum Industriestandard geworden ist und so eine sehr große Verbreitung unter den aktuellen Datenbanksystemen hat. Zusätzlich erfordert es einen geringen Overhead auf der föderierten Ebene, der dazu oftmals von Middlewarekomponenten wie TP-Monitoren übernommen wird, so dass der Implementierungsaufwand gering bleibt. Der einzige Vorbehalt ist, dass die Verwendung des Zweiphasen-Commitprotokolls die Autonomie der Komponentensysteme beeinträchtigen kann: Die Sperrwartezeiten bzw.

Konfliktrate lokaler Transaktionen können u.U. stark erhöht werden, weil föderierte Subtransaktionen ihre Sperren erst freigeben, wenn das globale Protokoll zu einem Ergebnis gekommen ist. Fällt der Koordinator aus, bevor dieses Ergebnis erzielt wurde, werden lokale Transaktionen potentiell auf unbegrenzt lange Zeit blockiert. In letzter Konsequenz bedeutet der Einsatz eines 2PC also eine potentielle Leistungseinbuße bzw. ein Leistungsrisiko für die lokalen Transaktionen.

Verbessern lässt sich dies dadurch, dass die Subtransaktionen der föderativen Transaktionen jeweils so früh wie möglich ein eigenständiges Commit durchführen. Dies führt auf einen Mehrschichtentransaktionsansatz, der dann zwingend mit Kompensationsaktionen arbeiten muss. Ein solcher Algorithmus erfordert zwar einen gewissen Overhead zur Gewährleistung von Atomarität wegen der Speicherung der notwendigen Kompensationsaktionen, verspricht jedoch speziell dann eine bessere Performance, wenn auf den höheren Schichten Kommutativität von Operationen ausgenutzt werden kann.

3.4 Etablierte Techniken zur globalen Serialisierbarkeit

3.4.1 Globale Serialisierbarkeit

In föderierten Datenbanksystemen ist nicht ausschließlich der Scheduler auf der föderierten Ebene für die korrekte Ausführung globaler Transaktionen zuständig. Zusätzlich arbeiten lokale Scheduler in den Komponentensystemen, die an der Föderation beteiligt sind, die die Serialisierungen der lokalen Subschedules bestimmen. Man versucht nun, eine äquivalente serielle Form des globalen Schedules abzuleiten, indem man die Serialisierungsreihenfolgen der globalen Subtransaktionen aus den Komponentensystemen auf den globalen Schedule überträgt. Dies ist natürlich nur möglich, wenn die lokalen Serialisierungsordnungen der Subtransaktionen *verträglich* sind, d.h. wenn sie sich nicht widersprechen. Wird z.B. die Subtransaktion einer globalen Transaktion t_1 von einem lokalen Scheduler vor einer Subtransaktionen einer anderen globalen Transaktion t_2 serialisiert, muss diese Reihenfolge auch in allen anderen Komponentensystemen gelten, wo beide Transaktionen Operationen ausgeführt haben. Dies führt unmittelbar zum Begriff der globalen Serialisierbarkeit:

Definition 3.2: (globale Serialisierbarkeit)

Ein globaler Schedule s heißt *global serialisierbar*, wenn alle seine lokalen Subschedules serialisierbar sind und die Serialisierungsreihenfolgen der lokalen Subschedules verträglich sind, d.h. wenn eine Subtransaktion $t_i^{(k)}$ in der lokalen Datenbank DB_k im zum lokalen Subschedule äquivalenten Schedule vor einer Subtransaktion $t_j^{(k)}$ steht, dann muss das auch für die Subtransaktionen dieser Transaktionen in allen übrigen lokalen Datenbanken gelten, in denen beide Transaktionen Operationen ausführen. ■

Es ist nicht trivial, globale Serialisierbarkeit zu gewährleisten. Insbesondere gilt nicht automatisch globale Serialisierbarkeit, wenn die lokalen Systeme selbst Serialisierbarkeit gewährleisten. Es genügt nicht einmal, globale Transaktionen sequentiell auszuführen, da lokale Transaktionen, die an der globalen Kontrolle vorbei auf eine lokale Datenbank zugreifen, Probleme machen können. Um dies zu verdeutlichen, betrachten wir

den folgenden globalen Schedule von zwei Transaktionen t_1 und t_2, die Operationen in zwei Datenbanken ausführen:

DB_1 $r_1(x)$ $w_1(x)$ C_1 $r_2(x)$ $w_2(x)$ C_2
DB_2: $r_L(a)$ $r_1(a)$ $w_1(a)$ C_1 $r_2(b)$ C_2 $r_L(b)$ $w_L(b)$ C_L

Die beiden globalen Transaktionen laufen vollständig nacheinander ab. Die Serialisierungsreihenfolge in DB_1 ist auch $t_1^{(1)}$ vor $t_2^{(1)}$, da sonst keine Transaktionen ablaufen. In DB2 dagegen gibt es eine lokale Transaktion t_L, die parallel zu t_1 und t_2 läuft. Sie sorgt dafür, dass die einzig mögliche Serialisierung $t_2^{(2)} \rightarrow t_L \rightarrow t_1^{(2)}$ ist, da t_2 von $t_L^{(2)}$ liest, aber t_L selbst von $t_1^{(2)}$ liest. Die lokalen Serialisierungsreihenfolgen in DB_1 und DB_2 sind also nicht verträglich, daher ist der globale Schedule nicht serialisierbar. Auf der globalen Ebene kann dies ohne weiteres nicht erkannt werden, da die lokale Transaktion t_L dort nicht bekannt ist.

Wir klassifizieren in diesem Kapitel zunächst die verschiedenen Ansätze, die zur Lösung des Problems der globalen Serialisierbarkeit entwickelt wurden, und diskutieren ihre Kombination mit den Verfahren zur globalen Atomarität aus Abschnitt 3.3. Anschließend beurteilen wir ihre Einsetzbarkeit in praktischen Systemen, unter Berücksichtigung der in Abschnitt 2.6 vorgestellten Eigenschaften einiger verbreiteter Systeme. Schließlich stellen wir zwei ausgewählte Techniken, die Tickettechnik und Mehrschichtentransaktionen, ausführlicher vor, da wir später Verfahren entwickeln werden, die auf diesen Techniken basieren.

3.4.2 Klassifikation der Verfahren zur globalen Concurrency Control

In diesem Abschnitt stellen wir die verschiedenen Ansätze, die zur Lösung des Problems der globalen Serialisierbarkeit vorgeschlagen wurden, vor und klassifizieren sie. Von Kategorie zu Kategorie ist dabei zunehmend mehr Aufwand zur föderierten Concurrency Control notwendig. Bei allen Verfahren, die Steuerungs- bzw. Kontrollmaßnahmen auf der föderativen Ebene umfassen, sind jeweils pessimistische (d.h. sperrende) wie auch optimistische (d.h. auf Konfliktzyklustests und Zurücksetzen basierende) Varianten denkbar. Allen vorgestellten Verfahren ist gemeinsam, dass sie mindestens serialisierbare lokale Subschedules voraussetzen. Eine gute Übersicht über bekannte Verfahren wird in [BGS92] gegeben, umfangreichere Betrachtungen finden sich in [Lehn96].

- *Ausnutzung lokaler Invarianten*: Wenn alle Komponentendatenbanksysteme ausschließlich commit-ordnungserhaltend konfliktserialisierbare Schedules (COSCR, siehe Definition 2.6) erzeugen, kann man zeigen, dass dann alle globalen Schedules ohne weitere Maßnahmen serialisierbar sind [GRS94]. Die globale Serialisierungsreihenfolge entspricht dabei der Reihenfolge der globalen Commit-Operationen. Dieses Verfahren ist auch als *implizite Ticket-Technik* bekannt [GRS94].

- *Ausnutzung lokaler Invarianten mit föderierter Kontrolle*: Wenn die Komponentendatenbanksysteme lediglich ordnungserhaltend konfliktserialisierbare Schedules erzeugen (OCSR, siehe Definition 2.6), sind bereits Zusatzmaßnahmen auf der föderierten Ebene notwendig, um globale Serialisierbarkeit zu gewährleisten. Allerdings bringen sie relativ wenig zusätzlichen Overhead ein. Ein Verfahren dieser Art ist das altruistische Sperrverfahren [AGS87, SGA89, SGS94], ein anderes wird in [CBS93] vorgestellt.

- *Ausnutzung lokal exportierter Serialisierungsinformationen*: Wenn die Komponentendatenbanken zur Laufzeit Informationen über die lokal vorgenommenen Serialisierungen explizit zur Verfügung stellen, kann man diese auf der föderierten Ebene ausnutzen, um globale Serialisierbarkeit sicherzustellen. Üblicherweise wird dazu ein Zyklustest im globalen Serialisierungsgraph durchgeführt, der aus der Vereinigung der lokalen Serialisierungsgraphen entsteht. Vorgeschlagene Verfahren in diesem Bereich nutzen z.B. explizite Informationen, die die Komponentendatenbanken nach außen geben (wie Serialisierungsreihenfolgen oder -graphen), oder profitieren von Eigenschaften der lokalen Scheduler, wobei die Serialisierungsreihenfolge der Transaktionen bereits aus der Reihenfolge ausgezeichneter Operationen (sog. Serialisierungspunkte) abgeleitet werden kann [Pu88, DE90, MRBK+92].

- *Ausnutzung lokal erzwungener Konflikte mit föderierter Kontrolle*: Die *explizite Ticket-Technik* [GRS91, GRS94] fügt jeder föderierten Transaktion lesende und schreibende Zugriffe auf ein ausgezeichnetes Datum in jeder beteiligten Komponentendatenbank hinzu und erzeugt dadurch künstlich Konflikte zwischen ihnen. Anhand der Verhältnisse der gelesenen Werte können außerdem die Serialisierungsreihenfolgen in den lokalen Datenbanksystemen abgeleitet werden. Die optimistische Variante dieser Technik prüft die korrekte Serialisierungsreihenfolge erst unmittelbar vor dem Commit einer Transaktion und löst dabei u.U. Transaktionsabbrüche aus; die pessimistische Variante führt die Ticketzugriffe so aus, dass keine ungültigen Serialisierungsreihenfolgen entstehen können.

- *Föderierte Steuerung lokaler Serialisierungen*: Wenn über die lokalen Schedules der Komponentendatenbanken keinerlei Information vorliegt, muss die Serialisierbarkeit alleine durch Maßnahmen auf der föderierten Ebene sichergestellt werden. Ein solches Verfahren muss sehr konservativ arbeiten und von potentiellen lokalen Konflikten ausgehen. Auf dieser Grundlage kann dann wieder ein Zyklustest abgeleitet werden, der jedoch auf einem relativ groben Graphen beruht [RS88]. Dabei müssen die potentiellen Einflüsse lokaler Transaktionen grundsätzlich unberücksichtigt bleiben.

- *Ausnutzung der föderierten Operationssemantik*: Die Subtransaktionen föderierter Transaktionen entsprechen in praktischen Anwendungen oft speziell exportierten „Methoden“ der Komponentendatenbanken. Es bietet sich daher an, die Semantik dieser „Methoden“ bzw. der in den Transaktionen aufgerufenen Datenbankoperationen auszunutzen. Kommutative Methoden bzw. in ihrer Reihenfolge beliebig vertauschbare Methoden werden dabei als bezüglich der Concurrency-Control kompatible Operationen spezifiziert. Selbst wenn eine explizite Spezifikation solcher Eigenschaften nicht vorliegt, kann man immer noch Kommutativitätseigenschaften der aufgerufenen SQL-Operationen ausnutzen. Dieser Ansatz führt auf eine Anwendung von Mehrschichtentransaktionen, die wir in Abschnitt 2.2.4 vorgestellt haben. Im Kontext föderativer Datenbanken ergeben sich zusätzliche Schwierigkeiten, vor allem hinsichtlich der Wechselwirkungen mit lokalen Transaktionen. Konzeptionelle Lösungsansätze für diese Probleme werden z. B. in [VEH92], [DSW94], [Scha96] und [Muth97] vorgestellt. Auf der föderierten Ebene müssen dazu entweder pessimistische Sperren [DSW94, Muth97] oder optimistische Konflikttests realisiert werden [Ripk98].

3.4.3 Kombination mit Verfahren zur Gewährleistung von Atomarität

Die Verfahren zur Gewährleistung der Serialisierbarkeit lassen sich *nicht* beliebig mit den Verfahren für die Transaktionsatomarität kombinieren. Vielmehr müssen diese beiden Aspekte einer föderierten Transaktionsverwaltung eng aufeinander abgestimmt sein, um bei akzeptabler Leistung korrekt zu arbeiten. Ein Teil der existierenden Arbeiten vernachlässigt leider diesen Aspekt, so dass sich daraus keine praktisch einsetzbaren Verfahren ableiten lassen.

Wir können die folgenden wichtigen Grundsätze zur Kombination der Verfahren zur Gewährleistung von Atomarität aus Abschnitt 3.3 und zur globalen Concurrency Control aus Abschnitt 3.4.2 festhalten:

Werden zur globalen Concurrency Control Mehrschichtentransaktionen genutzt, um die Semantik von föderierten Operationen ausnutzen zu können, müssen notwendigerweise Kompensationsaktionen zur Sicherstellung der Atomarität verwendet werden. Umgekehrt ist eine solche semantische Concurrency Control notwendig, wenn globale Transaktionen, die mit Kompensation arbeiten, bis zum Abschluss ihrer möglicherweise notwendigen Kompensationsschritte aus Anwendungssicht isoliert erscheinen sollen, also zu global serialisierbaren Abläufen führen sollen.

Wenn zur Gewährleistung von globaler Serialisierbarkeit ein anderes Verfahren eingesetzt wird, das z.B. besondere Eigenschaften der lokalen Datenbanksysteme ausnutzt, kann man das Zweiphasen-Commitprotokoll nach X/Open zur Sicherstellung von Atomarität verwenden. Kompensationsaktionen können dann nicht genutzt werden.

3.4.4 Qualitativer Vergleich der Verfahren

Da es möglich sein soll, beliebige Datenbanksysteme in föderierte Systeme zu integrieren, können keine Voraussetzungen über eine Kooperation der lokalen Datenbanken gemacht werden. Statt dessen müssen sie als in sich abgeschlossene, autonome Einheiten betrachtet werden, mit denen nur über die jeweilige Datenbanksprache (SQL oder eine objektorientierte Sprache) in Verbindung getreten werden kann. Außerdem sollen weiterhin lokale Transaktionen am föderierten Datenbanksystem vorbei direkt auf den lokalen Datenbanken arbeiten können und dabei möglichst wenig behindert werden. Betrachtet man die vorgestellten Techniken unter diesen Aspekten, kommt man zu folgenden Bewertungen.

- *Ausnutzung lokaler Invarianten*: Diese Methode, beispielsweise in Form der impliziten Ticket-Technik, ist nur für Komponentensysteme mit speziellen Eigenschaften einsetzbar. Für diese Systeme bildet sie eine sehr elegante Lösung, die keinen Overhead auf der föderierten Ebene generiert und keine inhärenten negativen Auswirkungen auf lokale Transaktionen hat. Allerdings halten globale Transaktionen ihre Sperren immer bis zu ihrem Ende, bzw. muss eine globale Transaktion bis zu ihrem Ende bei einem Konflikttest im lokalen System berücksichtigt werden. Der erzielbare Parallelitätsgrad und damit potentiell der Transaktionsdurchsatz ist daher gegenüber Verfahren, die die Semantik von Subtransaktionen mit eigenständigem Commit ausnutzen, notwendigerweise limitiert. Gegenüber allen anderen Verfahrensklassen schneidet die Methode in ihrer Leistung mindestens gleichwertig, wenn nicht sogar besser ab.

- *Ausnutzung lokaler Invarianten mit föderierter Kontrolle*: Die Annahmen, die diese Verfahren über die von den Komponentensystemen erzeugten lokalen Schedules machen, werden zunehmend restriktiver, so dass man in immer weniger Anwendungsfällen davon ausgehen kann, dass sie von den beteiligten Systemen erfüllt werden. Gleichzeitig generieren die zusätzlichen Kontrollmaßnahmen auf der föderierten Ebene einen gewissen Overhead, so dass diese Verfahren nicht mehr so einfach wie die Verfahren zu realisieren sind, die ausschließlich auf lokalen Invarianten beruhen. Auch führen alle in der Literatur vorgeschlagenen Verfahren dieser Kategorie zu erheblichen Einschränkungen der Parallelität (bis hin zu einer Zwangssequentialisierung aller föderierten Transaktionen); dies gilt beispielsweise für das altruistische Sperrverfahren sowie alle Methoden, die mit Site Locks bzw. Site Graphs arbeiten [AGS87, SGA89, BS88].

- *Ausnutzung lokal exportierter Serialisierungsinformation*en: Da diese Verfahren Informationen über lokale Strukturen wie Wartegraphen oder Sperrtabellen benötigen, setzen sie einen erheblichen Grad der Mitarbeit der Komponentensysteme voraus. In der Praxis wird dies von den wenigsten Systemen unterstützt, so dass diese Verfahren wenig praxistauglich sind. Zu dieser Kategorie zählen insbesondere alle Strategien, die sich auf Serialisierungspunkte berufen [Pu88, DE90, MRB+92].

- *Ausnutzung lokal erzwungener Konflikte mit föderierter Kontrolle*: Diese Verfahren können wegen der zusätzlichen erzwungenen lokalen Konflikte potentiell den Parallelitätsgrad stark beeinträchtigen. Andererseits sind sie gegenüber anderen Verfahren mit föderierter Steuerung relativ einfach zu implementieren, und sie generieren nur begrenzten Laufzeit-Overhead. In Situationen, in denen die rein auf lokalen Invarianten basierenden Verfahren nicht eingesetzt werden können, ist daher ein Einsatz dieser Verfahren, speziell der expliziten Tickettechnik [GRS94], durchaus sinnvoll.

- *Föderierte Steuerung lokaler Serialisierungen:* Da diese Verfahren keine Annahmen über die Interna der Komponentensysteme machen, müssen sie mehr oder weniger zwingend mit einem sehr groben Granulat arbeiten, beispielsweise indem sie ganze Relationen sperren. Dies schränkt die mögliche Parallelität drastisch ein. Gleichzeitig generieren diese Verfahren merklich mehr Overhead als etwa die explizite Ticket-Technik, so dass sie in praktischen Systemen nicht zum Einsatz kommen.

- *Ausnutzung der föderierten Operationssemantik:* Der Implementierungsaufwand und der potentielle Laufzeit-Overhead dieser Verfahren ist der höchste aller hier vorgestellten Kategorien. Auf der anderen Seite aber haben sie mit Abstand das beste Potential für sehr hohe Parallelität und damit hohen Durchsatz und gute Antwortzeiten. Mit nicht gerade geringem Implementierungsaufwand, aber doch effektiv realisierbar lassen sich auch lokale Transaktionen berücksichtigen.

	Praxis-tauglichkeit	Overhead	Leistungs-fähigkeit
Ausnutzung lokaler Invarianten	sehr gut	sehr wenig	mittel bis hoch
Ausnutzung lokaler Invarianten mit föderierter Kontrolle	einsetzbar	wenig	gering
Ausnutzung lokal exportierter Serialisierungsinformationen	nicht praktikabel	wenig	gering
Ausnutzung lokal erzwungener Konflikte mit föderierter Kontrolle	sehr gut	wenig	mittel
Föderierte Steuerung lokaler Serialisierungen	nicht praktikabel	mittel	sehr gering
Ausnutzung der föderierten Operationssemantik	sehr gut	viel	potentiell hoch

Tabelle 1: Vereinfachende Zusammenfassung der Stärken und Schwächen der verschiedenen Verfahrenskategorien zur föderierten Transaktionsverwaltung

Wir fassen die Bewertung der verschiedenen Verfahrensklassen zur Sicherstellung der globalen Serialisierbarkeit in Tabelle 1 zur besseren Übersicht zusammen. Dabei nehmen wir an, dass jedes der Verfahren mit gemäß der Diskussion in Abschnitt 3.4.3 jeweils passenden Verfahren zur globalen Atomarität kombiniert wird.

Insgesamt ergibt sich, dass die Verfahren, die sowohl praktisch einsetzbar sind, als auch ausreichende Leistung versprechen, die impliziten und die expliziten Varianten der Tickettechnik sowie Mehrschichtentransaktionen sind, wobei keins der Verfahren die beste Lösung für alle Anwendungsfälle garantieren kann. Wir stellen diese Verfahren in den folgenden Abschnitten vor.

3.4.5 Die Ticket-Technik

Die Tickettechnik [GRS91,GRS94] leitet die lokale Serialisierungsreihenfolge auf elegante und flexible Art aus dedizierten Operationen auf einem sogenannten Ticket-Objekt ab, die jeder globalen Subtransaktion hinzugefügt werden. Die so gewonnenen Informationen werden auf der föderierten Ebene genutzt, um globale Serialisierbarkeit zu gewährleisten.

Ein "Ticket" ist ein ausgezeichnetes Objekt pro Komponentendatenbank, das einen numerischen Datentyp hat. Es wird ausschließlich für die globale Concurrency Control benutzt. Jede globale Subtransaktion muss zu einem beliebigen Zeitpunkt ihres Ablaufs, aber noch vor ihrem Commit, den aktuellen Wert des Tickets lesen und einen erhöhten Wert zurückschreiben. (Man spricht dann davon, dass sich die Subtransaktion "ein Ticket nimmt".) Durch dieses Vorgehen wird eine Kette von Schreib-Lese- und Lese-Schreib-Konflikten zwischen den globalen Subtransaktionen erzeugt, so dass das Verhältnis der Ticket-Werte zweier globaler Subtransaktionen ihre lokale Serialisierungsreihenfolge widerspiegelt.

Um die Arbeitsweise der Tickettechnik zu verdeutlichen, betrachten wir noch einmal den folgenden Schedule, mit dem wir bereits am Anfang von Abschnitt 3.4 die Problematik der globalen Serialisierbarkeit gezeigt haben:

$$r_L(a)\ r_1(a)\ w_1(a)\ c_1\ r_2(b)\ c_2\ r_L(b)\ w_L(b)\ c_L$$

Die beiden globalen Subtransaktionen t_1 und t_2 laufen vollständig seriell ab und haben keinen Konflikt. Durch die lokale Transaktion t_L, die unsichtbar für die föderierte Ebene ausgeführt wird, ergibt sich nun aber wegen der Konflikte auf a zwischen t_L und t_1 und auf b zwischen t_2 und t_L als einzig mögliche Serialisierungsreihenfolge $t_2 \rightarrow t_L \rightarrow t_1$. Die Reihenfolge von t_1 und t_2 wurde also vertauscht, ohne dass dies auf der föderierten Ebene erkannt werden kann. Fügt man nun zu den beiden globalen Transaktionen den Ticketzugriff hinzu, ergibt sich der folgende Schedule:

$$r_L(a)\ r_1(a)\ w_1(a)\ r_1(T)\ w_1(T)\ c_1\ r_2(b)\ r_2(T)\ w_2(T)\ c_2\ r_L(b)\ w_L(b)\ c_L$$

Durch den Ticketzugriff ist nun ein direkter Konflikt zwischen t_1 und t_2 entstanden. Der Konfliktgraph dieses Schedules hat einen Zyklus $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_1$, so dass der Schedule nicht mehr konfliktserialisierbar ist und somit bereits vom lokalen System abgelehnt wird. Problemfälle dieser Art können also mit der Tickettechnik vermieden werden.

Es kann aber immer noch vorkommen, dass in verschiedenen lokalen Systemen unverträgliche Serialisierungsreihenfolgen erzeugt werden. Um dies zu erkennen, nutzt man auf der föderierten Ebene die Ticketwerte der Subtransaktionen in den einzelnen Komponentensystemen aus. Man verwaltet dazu einen Ticket-Graphen, der die globalen Transaktionen als Knoten hat. In diesen Graphen trägt man immer dann eine Kante von Transaktion t_i zu Transaktion t_j ein, wenn der Ticket-Wert einer Subtransaktion von t_i in einer lokalen Datenbank kleiner als der einer Subtransaktion von t_j in der gleichen Datenbank ist. In [GRS94] wurde gezeigt, dass der globale Schedule genau dann serialisierbar ist, wenn der Ticket-Graph keinen Zyklus enthält; dabei wurde vorausgesetzt, dass alle lokalen Scheduler mindestens serialisierbare Schedules erzeugen.

Die Ticket-Technik ist einfach und effizient zu implementieren, man muss im wesentlichen den Ticketzugriff zu jeder Subtransaktion hinzufügen und den globalen Ticketgraphen verwalten. Neben dem Eintragen und Löschen von Knoten und Kanten muss man den Graphen spätestens, wenn eine Transaktion ihr Commit ausführen will, auf einen Zyklus testen. Wird einer gefunden, müssen eine oder mehrere Transaktionen, die auf dem Zyklus liegen, zurückgesetzt werden, bis der Zyklus durchbrochen wurde.

Zusätzliche Optimierungen sind möglich, wenn Komponentensysteme besondere Eigenschaften haben. Im Falle, dass ein Komponentensystem ausschließlich rigorose Schedules erzeugt, kann man zeigen, dass kein expliziter Ticketzugriff notwendig ist. Stattdessen kann man den Zeitpunkt, zu dem die Transaktion ihr Commit ausgeführt hat, als implizites Ticket verwenden und Kanten in den globalen Ticketgraphen eintragen, die der Commitreihenfolge in diesem Komponentensystem entsprechen. Falls alle lokalen Scheduler Schedules aus der Klasse ACA erzeugen, muss man zwar den Ticketzugriff ausführen, kann aber auf das Verwalten des Ticketgraphs verzichten; diese Variante ist als implizite Tickettechnik bekannt [GRS91].

Die Tickettechnik zeichnet sich besonders dadurch aus, dass sie nur so viel zusätzlichen Aufwand wie unbedingt notwendig generiert; stattdessen profitiert sie von Eigenschaften der Komponentensysteme, wie sie in vielen Systemen in der Praxis vorkommen. Sie

ist daher ein sehr eleganter und vielseitig verwendbarer Algorithmus für eine föderierte Concurrency Control.

Neben diesen Vorteilen hat die Tickettechnik allerdings auch Nachteile. Zunächst ist offensichtlich, dass das Ticketobjekt ein potentieller Engpass in den Komponentendatenbanken ist, da alle globalen Subtransaktionen das Ticketobjekt lesen und schreiben müssen. Der beste Zeitpunkt, zu dem der Ticketzugriff innerhalb einer Transaktion erfolgen soll, hängt außerdem vom Schedulingverfahren in jeder lokalen Datenbank ab; er ist im allgemeinen nicht einfach zu bestimmen (Diskussionen zu einigen gängigen Schedulingverfahren findet man in [GRS94]). Schließlich werden reine Lesetransaktionen durch das Schreiben des Ticketobjekts zu Lese-Schreib-Transaktionen, so dass das Datenbanksystem spezielle Optimierungen für reine Lesetransaktionen (z.B. in Oracle mit der Option "read only") nicht mehr einsetzen kann.

In [ZE93] wird eine Verallgemeinerung der Tickettechnik vorgestellt. [TW97] und [HAD97] zeigen Anwendungen der Tickettechnik zur Realisierung von geschachtelten Transaktionen.

3.4.6 Mehrschichtentransaktionen

Mehrschichtentransaktionen, die wir in Abschnitt 2.2.4 vorgestellt haben, können in einer Variante auch in föderierten Datenbanksystemen eingesetzt werden, um globale Serialisierbarkeit sicherzustellen [SWS91, SS93, DSW94, SSW95]. Man verwendet dabei eine zweischichtige Architektur, wobei in Schicht L_1 die Operationen der föderierten Transaktionen und in L_0 die Seitenoperationen in den Komponentensystemen ausgeführt werden. Wenn die Komponentensysteme an ihrem Interface Objekte und deren Methoden anbieten, wie es z.B. objektorientierte Systeme, aber auch CORBA-Server tun, können Operationen auf der föderierten Ebene die Aufrufe solcher Objekte sein. Andernfalls sind es Anweisungen in der Sprache der jeweiligen lokalen Datenbank, also z.B. SQL-Anfragen und Einfügeoperationen. Wie im zentralisierten Fall wird jede solche Operation innerhalb der Komponentendatenbank in einer eigenen Subtransaktion ausgeführt.

Um Serialisierbarkeit gewährleisten zu können, müssen wir nach der Diskussion in Abschnitt 2.2.4 auf L_0, also in den Komponentensystemen, ordnungserhaltend konfliktserialisierbare Schedules fordern. Viele existierende Systeme setzen zweiphasiges Sperren ein und erfüllen damit diese Voraussetzung, so dass sie in einer solchen Umgebung eingesetzt werden können. Auch auf L_1 müssen wir ordnungserhaltend konfliktserialisierbare Schedules sicherstellen. Wir setzen dazu auch auf diesem Level ein Sperrverfahren ein, bei dem wir die Objekte sperren, auf die föderierte Transaktionen zugreifen. Jede Methode eines Objektes entspricht dabei einem Sperrmodus; wenn eine föderierte Transaktion die Methode ausführen will, muss sie zunächst das Objekt im entsprechenden Modus sperren. Anhand der Semantik der Methoden muss ein Administrator entscheiden, ob zwei Methoden kompatibel sind, und entsprechend ihre Sperrmodi als verträglich markieren. Kann man auch über Anweisungen in der Datenbanksprache auf die lokalen Datenbanken zugreifen, müssen auch für diese Operationen (gemeinsame oder exklusive) Sperren erworben werden. Wenn es möglich ist, dasselbe Objekt sowohl über eine Methode als auch über die Datenbanksprache anzusprechen, müssen auch Verträglichkeiten von Methodensperrmodi und gemeinsamen bzw. exklusiven Sperren spezifiziert werden.

Zur Verwaltung der Sperren werden üblicherweise Prädikatsperren verwendet, d.h. man sperrt nicht die Objekte, auf die eine Operation zugreift, sondern ein Prädikat, dass die Objektmenge insgesamt beschreibt. Dieses Prädikat kann man bei Anfragen aus dem Suchprädikat ableiten, bei Einfüge- und Löschoperationen aus den aktuellen Attributen des Objektes. Für die Operation

```
SELECT P FROM PARTS WHERE P.COLOR=GREEN OR (P.PRICE>100 AND P.WEIGHT>7)
```

sperrt man beispielsweise die Prädikate `COLOR=GREEN` und `PRICE=100 AND WEIGHT<7` im gemeinsamen Sperrmodus. Wenn die Anfrage einen Join enthält, wird das Prädikat in der Anfrage in einzelne Prädikate für jede beteiligte Tabelle aufgeteilt. Das Join-Prädikat selbst kann dabei ignoriert werden, aber die übrigen Filterprädikate werden berücksichtigt, so dass man faktisch statt des Joins das kartesisches Produkt der beteiligten Relationen sperrt. Auf die gleiche Weise lassen sich Prädikate ableiten, die die Objekte beschreiben, die von Einfüge-, Lösch- und Änderungsoperationen sowie von Methodenaufrufen betroffen sind.

Weil die eventuell gehaltenen Sperren auf Level L_0 nach dem Ende einer Operation freigegeben werden, können lokale Transaktionen, die unter Umgehung der föderierten Ebene direkt auf eine lokale Datenbank zugreifen, Änderungen sehen, die innerhalb dieser Operation gemacht wurden. Da die föderierte Transaktion, die diese Änderungen gemacht hat, aber noch nicht beendet ist, kann es sein, dass die gelesenen Daten inkonsistent sind oder später ungültig werden, wenn die föderierte Transaktion abgebrochen werden sollte. Um diese Probleme zu vermeiden, müssen auch lokale Transaktionen in die Sperrverwaltung auf Level L_1 integriert werden. In [DSW94] wird dazu vorgeschlagen, die Sperren datenbankweise in sogenannten *Agenten* zu verwalten und lokale Transaktionen so zu ändern, dass sie ihre Zugriffe auf die Datenbank über diesen Agenten machen. Im allgemeinen ist dies allerdings nicht immer einfach zu realisieren, da die lokalen Transaktionen selbst nicht ohne weiteres geändert werden können. Es kann aber möglich sein, durch Eingriffe ins Laufzeitsystem der lokalen Datenbank alle Zugriffe transparent über den Agenten umzuleiten.

3.5 Transaktionsunterstützung in existierenden Produkten

Da Föderationen von heterogenen Datenbanksystemen in der Praxis immer wichtiger werden, erweitern immer mehr Anbieter von Datenbanksoftware ihre Produkte um die Möglichkeit, auch auf die Systeme von Fremdanbietern zuzugreifen. In diesem Abschnitt stellen wir exemplarisch zwei Lösungen vor, die Föderationskomponenten in Oracle 8.1.6 und in IBM's DB2 Universal Database. Wir konzentrieren uns dabei auf die Unterstützung, die diese Produkte für Atomarität und Isolation föderierter Transaktionen bieten.

3.5.1 Oracle 8.1.6

Oracle [Orac99b] erlaubt es, verteilte Datenbanken aus Oracle- und Nicht-Oracle-Komponenten aufzubauen und mit den üblichen SQL-Anweisungen anzusprechen. Die Umsetzung der verschiedenen Datentypen und SQL-Dialekte wird von den *Heterogeneous Services* vorgenommen, die jede Datenbank eines Fremdherstellers über ein produktspezifisches *Transparent Gateway* [Orac96, Orac99a] ansprechen.

In Föderationen, die nur aus Oracle-Datenbanken bestehen (also in verteilten Oracle-Datenbanken), stellt Oracle eine konsistente Sicht von Transaktionen auf die verteilten Daten sicher, indem die beteiligten Datenbanken regelmäßig ihre internen Zeitmarken synchronisieren. Aus Performancegründen geschieht diese Synchronisation aber nicht ständig, sondern nur nach dem Ende einer verteilten Anfrage und am Anfang einer Transaktion. Es kann daher vorkommen, dass eine Transaktion keine konsistenten Daten sieht. In der Dokumentation wird ein Beispiel gegeben, dass eine zweimalige Ausführung derselben verteilten SQL-Anweisung ohne Änderungsoperationen dazwischen zu verschiedenen Ergebnissen führen kann. Als Lösung wird vorgeschlagen, entweder vor jeder verteilten Query eine andere verteilte Anfrage auszuführen, die keine Ergebnisse zurückliefert, aber dafür sorgt, dass die Zeitmarken synchronisiert werden, oder die aktuelle Transaktion zu beenden und eine neue zu beginnen, was natürlich der Idee der Kapselung mehrerer Queries in einer Transaktion zuwider läuft. Ähnlich wie schon im Falle der Gewährleistung von Serialisierbarkeit innerhalb einer einzigen Oracle-Instanz erfordern diese Lösungen Zusatzaufwand vom Anwendungsprogrammierer, der durch ein geschickteres Design der Synchronisation vermieden werden könnte.

Wenn auch Datenbanksysteme von Fremdherstellern an der Föderation teilnehmen, wird überhaupt keine Garantie für eine konsistente Sicht auf die Daten abgegeben. Die Dokumentation warnt vor möglichen Problemen, insbesondere wird auf die für Oracle-Anwender ungewohnte Möglichkeit der Blockierung von Leseoperationen durch Sperrkonflikte hingewiesen. Als Lösung empfiehlt man, die Programmierrichtlinien des anderen Systems zu berücksichtigen. Wir werden in den folgenden Abschnitten sehen, dass dieser Ansatz im allgemeinen nicht zum Ziel führt.

Während Oracle also für föderierte Transaktionen keine konsistente Sicht auf die Daten garantieren kann, ist die Gewährleistung von Atomarität unkritisch. Oracle verwendet dazu das etablierte Zweiphasen-Commitprotokoll und kann alle Fremdsysteme einbinden, die das XA-Interface zur Steuerung des Zweiphasen-Commitprotokolls unterstützen.

3.5.2 IBM DB2 Universal Database Release 7

Zur Unterstützung föderierter Datenbanksysteme wurde das unabhängig entwickelte Produkt *DataJoiner* [IBM98] in DB2 integriert [IBM00a, IBM00c]. Man kann damit verteilte Datenbanken aus DB2- und damit verwandten Datenbanken sowie Oracle-Instanzen aufbauen. Oracle-Datenbanken werden über den Mediator *DB2 Relational Connect* eingebunden, der Konvertierungen von Datentypen und SQL-Dialekt vornimmt.

Über das föderierte Interface lassen sich zunächst nur Anfragen absetzen. Um für diese eine konsistente Sicht auf die Daten zu gewährleisten, verwendet DB2 in allen beteiligten Systemen vergleichbare Isolation Levels. Da Oracle nicht die gleichen Isolation Levels wie DB2 anbietet, kommt es dabei zwangsweise zu Unschärfen bei der Abbildung, so wird z.B. DB2's Modus Repeatable Read in Oracle's "Read-Only"-Modus übersetzt, in dem es reinen Lesetransaktionen eine konsistente Sicht auf die Daten garantiert, aber keine Änderungen erlaubt. Dies wird dann ein Problem, wenn föderierte Transaktionen auch schreibend auf Datenbanken zugreifen sollen, was sie unter Umgehung der föderierten Ebene direkt durch Operationen auf den Datenbanken

erledigen müssen. In der Dokumentation wird nicht geklärt, wie sich dies mit den Einstellungen für reine Lesetransaktionen verträgt, die in Oracle verwendet werden.

Zur Gewährleistung der Atomarität globaler Transaktionen setzt DB2 ähnlich wie Oracle auf das Zweiphasen-Commitprotokoll. Es unterstützt das übliche XA-Interface, so dass es auch in Anwendungsfällen eingesetzt werden kann, in denen TP-Monitore die Koordination von Transaktionen übernehmen.

3.6 Serialisierbarkeit unter lokaler Snapshot Isolation

3.6.1 Lokale Isolation Levels und globale Serialisierbarkeit

Alle Algorithmen zur Gewährleistung globaler Serialisierbarkeit, die wir in diesem Kapitel diskutiert haben, erfordern zwingend, dass alle beteiligten Komponentensysteme mindestens serialisierbare Schedules erzeugen. In der Praxis gibt es aber wichtige Systeme wie Oracle, die gerade diese Eigenschaft nicht sicherstellen können, sondern bestenfalls einen Isolation Level wie Snapshot Isolation garantieren. Zusätzlich kann man daran interessiert sein, lokal Isolation Levels einzusetzen, um einen höheren Transaktionsdurchsatz erzielen zu können, weil dadurch Sperren früher freigegeben werden. Auf der föderierten Ebene sollen aber auch unter diesen Bedingungen serialisierbare Schedules erzeugt werden, weil für wichtige Anwendungen solche Korrektheitsgarantien wesentlich sind.

Wir zeigen anhand des folgenden Schedules beispielhaft ein Problem, das auftreten kann, wenn mindestens ein Komponentensystem nur ein eingeschränktes Korrektheitskriterium, hier Read Committed, unterstützt:

DB_1: $r_1(x)$ $w_1(x)$ c_1 $r_2(x)$ $w_2(x)c_2$

DB_2: $r_2(a)$ $r_1(a)$ $w_1(a)$ c_1 $w_2(a)$ c_2

Der Scheduler der Komponentendatenbank DB_1 erzeugt serialisierbare Schedules, dort wird t_1 vor t_2 serialisiert. Der Scheduler in DB_2 garantiert nur Read Committed, der dort erzeugte Schedule ist nicht mehr serialisierbar. Insbesondere geht die Änderung, die t_1 an Objekt a vorgenommen hat, verloren, da sie unmittelbar danach von t_2 überschrieben wird, ohne dass t_2 vorher den von t_1 geschriebenen Wert gelesen hätte. In DB_1 dagegen sieht t_2 den von t_1 geänderten Wert des Objektes x, bevor t_2 den neuen Wert von x in die Datenbank schreibt. Die globale Datenbank ist also potentiell inkonsistent geworden, weil in DB_2 ein schwächerer Isolation Level verwendet wurde.

Die Definition von globaler Serialisierbarkeit lässt sich a priori nicht mehr anwenden, wenn einige Systeme schwächere Isolation Levels verwenden, da sie serialisierbare lokale Subschedules voraussetzt. Wir müssen also mit entsprechenden Algorithmen auf der föderierten Ebene dafür sorgen, dass unzulässige lokale Schedules, also solche, die nicht serialisierbar sind, unterbunden werden. Im allgemeinen ist dies ein schwieriges Problem, da auf der föderierten Ebene nur SQL-Anweisungen oder Methodenaufrufe der föderierten Transaktionen, aber keine oder nur wenige Details über ihre Ausführung in den Komponentendatenbanksystemen bekannt sind. Insbesondere weiß man nicht, auf welche Datenbankseiten eine Subtransaktion tatsächlich zugreift, so dass man keine Informationen über Konflikte zwischen verschiedenen Subtransaktionen hat, aus denen man zum Beispiel einen Konfliktgraphen konstruieren könnte. Man ist daher im allge-

meinen auf Näherungslösungen angewiesen, die aus den Operationen der Subtransaktionen auf der föderierten Ebene Konflikte in den Komponentensystemen ableiten.

In den folgenden Abschnitten beschränken wir uns auf Möglichkeiten, wie globale Serialisierbarkeit sichergestellt werden kann, wenn einige oder alle lokalen Systeme nur Snapshot Isolation garantieren. Dieser Fall ist sehr praxisrelevant, weil Systeme wie der Marktführer Oracle bestenfalls diesen Isolation Level sicherstellen können. Verfahren für andere Isolation Levels lassen sich mit ähnlichen Techniken ableiten, wir diskutieren die dabei entstehenden Probleme kurz am Ende von Abschnitt 3.6.3.

Wir betrachten zunächst das Verhalten der Tickettechnik und einer auf Mehrschichtentransaktionen basierenden Technik unter dieser Voraussetzung und stellen vor, wie man diese Strategien an diese Bedingungen anpassen kann. Anschließend entwickeln wir einen neuen Algorithmus auf Basis des Online-SI-MSVG, den wir in Abschnitt 2.4.4 vorgestellt haben. In Abschnitt 3.6.4 zeigen wir Möglichkeiten auf, wie sich die vorgestellten Verfahren in föderierten Systemen anwenden lassen, in denen manche Systeme Snapshot Isolation und manche Serialisierbarkeit gewährleisten. Den Abschluss bildet ein qualitativer Vergleich der dargestellten Verfahren.

3.6.2 Verhalten und Verbesserungen bekannter Verfahren

Die etablierten Algorithmen zur globalen Concurrency Control setzen voraus, dass die lokalen Scheduler serialisierbare Schedules erzeugen. Es ist daher zunächst unklar, ob diese Verfahren auch eingesetzt werden können, wenn lokal nur Snapshot Isolation garantiert wird.

Manche Verfahren lassen sich in diesem Fall überhaupt nicht einsetzen. Dazu gehören alle Verfahren, die implizite Aussagen über die lokale Serialisierungsreihenfolge von Transaktionen benutzen (z.B. Serialisierungspunkte oder Commitzeitpunkte). Algorithmisch orientierte Verfahren, die nur explizite oder gar keine Informationen aus den beteiligten Komponentendatenbanken verwerten, versprechen eher eine Lösung des vorhandenen Problems im heterogenen Umfeld; in Frage kommen also die Familie der Tickettechniken und Mehrschichtentransaktionen. In den folgenden Abschnitten diskutieren wir die Anwendbarkeit und eventuell notwendige Anpassungen dieser beiden Verfahren für lokale Scheduler, die Snapshot Isolation gewährleisten.

3.6.2.1 Die Ticket-Technik

Voraussetzung für die Anwendbarkeit der Tickettechnik ist, dass die lokalen Scheduler serialisierbare Schedules generieren, andernfalls ist die Korrektheit des globalen Schedules nicht mehr gewährleistet. Wir nehmen zunächst an, dass es keine lokalen Transaktionen gibt, und zeigen, welche Auswirkungen das Hinzufügen des Ticketzugriffs in lokalen Datenbanksystemen hat, die nur Snapshot Isolation garantieren. Dazu betrachten wir den folgenden (nichtserialisierbaren) Beispielschedule von zwei Transaktionen t_1 und t_2:

$$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(y)\ c_1\ c_2$$

Dieser Schedule ist, wie wir bereits in Kapitel 2 gesehen haben, zwar Element von SI, aber nicht serialisierbar. Fügt man nun zu jeder Transaktion den Ticketzugriff hinzu, so erhält man den folgenden Schedule:

$$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(y)\ r_1(T)\ w_1(T)\ c_1\ r_2(T)\ w_2(T)\ c_2$$

Nun verändern die beiden parallel laufenden Transaktionen t_1 und t_2 das Ticketobjekt T, der Schedule kann daher nicht mehr in SI sein. Der lokale Scheduler wird also diesen Schedule ablehnen und eine der beiden Transaktionen, vermutlich die zweite, zwangsweise zurücksetzen.

Allgemein lässt sich sagen, dass von mehreren parallel laufenden Transaktionen immer nur eine mit Commit beendet werden kann, sobald zu allen der Ticketzugriff hinzugefügt wird; die übrigen werden vom System zwangsweise zurückgesetzt und müssen dann erneut gestartet werden. Auf diese Weise wird eine Sequentialisierung der Transaktionen erzwungen. Der entstehende Schedule ist trivialerweise serialisierbar, da er bereits seriell ist, und die Ticketwerte spiegeln die Reihenfolge der Transaktionen wider. Die Tickettechnik kann also prinzipiell eingesetzt werden, um lokale Systeme mit Schedulern, die Snapshot Isolation garantieren, in föderierte Systeme ohne lokale Transaktionen einzubinden, allerdings um den Preis, dass globale Transaktionen auf diesen Komponenten sequentiell ausgeführt werden müssen. Ihre Subtransaktionen in anderen Komponentensystemen können natürlich weiterhin parallel ablaufen. Trotzdem bewirkt die sequentielle Ausführung in SI-Systemen in der Regel einen dramatischen Performanceverlust, so dass sich der Einsatz der Tickettechnik in dieser Systemumgebung normalerweise verbietet.

Wenn zusätzlich lokale Transaktionen in einem der SI-Systeme aktiv sind, genügt selbst eine sequentielle Ausführung der globalen Subtransaktionen nicht mehr, wie man an folgendem Beispiel mit zwei globalen Subtransaktionen t_1 und t_2 und einer lokalen Transaktion t_L erkennt:

$$r_L(y_0)\ r_1(y_0)\ w_1(y_1)\ r_1(T_0)\ w_1(T_1)\ c_1\ r_2(x_0)\ r_2(T_1)\ w_2(T_2)\ c_2\ r_L(x_0)\ w_L(x_L)\ c_L$$

Da die lokale Transaktion auf der föderierten Ebene nicht sichtbar ist, wird ihr auch kein Ticketzugriff hinzugefügt. Die beiden globalen Subtransaktionen laufen sequentiell ab, so dass der zusätzliche Ticketzugriff ihren Ablauf nicht stören kann. Dieser Schedule ist immer noch in SI, aber nicht serialisierbar; Aussagen über globale Serialisierbarkeit sind also nicht mehr möglich, da bereits lokal kein serialisierbarer Schedule mehr gewährleistet werden kann.

Eine mögliche (und in der Tat die einzige) Lösung für dieses Problem ist es, auch zu den lokalen Transaktionen einen Ticketzugriff hinzuzufügen. Dies kann z.B. durch eine geringfügige Änderung des Codes erreicht werden, der beim Commit ausgeführt wird. In diesem Fall werden aber nicht nur die globalen, sondern auch die lokalen Transaktionen durch den Ticketzugriff sequentialisiert. Der daraus resultierende Performanceverlust betrifft nun also auch die lokalen Transaktionen und verletzt somit die Autonomie des lokalen Datenbanksystems. Eine solche Lösung wäre also sicherlich für die allermeisten Szenarien nicht einsetzbar.

3.6.2.2 Die erweiterte Ticket-Technik

In diesem Abschnitt stellen wir eine Erweiterung der Tickettechnik vor, die für reine Lese-Subtransaktionen die Zwangssequentialisierung, die sich unter lokaler Snapshot Isolation bei der üblichen Tickettechnik ergibt, vermeidet. Da viele praktische Anwendungen durch reine Lesetransaktionen dominiert sind, zwischen denen nur ab und zu ein paar Transaktionen Änderungen ausführen, kann für solche Anwendungen globale Serialisierbarkeit mit einer ausreichend hohen Leistung erreicht werden.

Wir betrachten zunächst globale Transaktionen in Föderationen, die ausschließlich aus Komponentensystemen bestehen, die Snapshot Isolation garantieren. Die Integration von lokalen Transaktionen sprechen wir am Ende dieses Abschnitts an, die Integration von Subsystemen, die serialisierbare Schedules erzeugen, diskutieren wir im Rahmen von Abschnitt 3.6.4. Wir setzen außerdem voraus, dass es dem System bekannt ist, ob eine Transaktion in allen Datenbanken nur liest oder auch mindestens eine davon ändert. Um dies zu erreichen, kann man beispielsweise fordern, dass jede Transaktion vom Client besonders markiert wird, wenn sie nur lesend auf die Datenbank zugreift; unmarkierte Transaktionen werden behandelt, als würden sie die Datenbank lesen und schreiben.

Zunächst diskutieren wir die Behandlung von globalen Transaktionen, die in mindestens einer lokalen Datenbank Änderungen vornehmen. Bereits im letzten Abschnitt haben wir gezeigt, dass der Ticketzugriff eine sequentielle Ausführung dieser Transaktionen erzwingt. Wir entwickeln nun zwei Varianten einer erweiterten Tickettechnik: eine eher pessimistische Version, die ändernde Transaktionen von vornherein sequentiell ausführt und so Abbrüche wegen des Ticketzugriffs verhindert, und eine optimistische Version, die eine Sequentialisierung schreibender Transaktionen durch den Ticketzugriff erzielt.

Die *pessimistische erweiterte Tickettechnik* kann immer dann sinnvoll eingesetzt werden, wenn relativ viele globale Transaktionen Änderungen in mehreren lokalen Datenbanken ausführen. In diesem Kontext besteht ein hohes Potential für Abbrüche nach Ticketkonflikten, es ist daher sinnvoll, alle Transaktionen, die Objekte ändern, schon sequentiell auszuführen und somit paralleles Schreiben des Tickets und die daraus resultierenden Transaktionsabbrüche zu verhindern. Der globale Transaktionsmanager muss also dafür sorgen, dass globale Transaktionen, die nicht bereits zu ihrem Beginn als reine Leser gekennzeichnet sind, nacheinander ausgeführt werden. Trotz der sequentiellen Reihenfolge müssen die Transaktionen das Ticket in jeder Komponentendatenbank lesen und erhöhen, um ihr Verhältnis zu parallel laufenden Lesetransaktionen und eventuell vorhandenen lokalen Transaktionen behandeln zu können.

Die *optimistische erweiterte Tickettechnik* ist prädestiniert für den Fall, dass insgesamt nur wenige globale Transaktionen Änderungen vornehmen, die dazu nur auf einzelne lokale Datenbanken schreibend, aber auf andere auch nur lesend zugreifen. Die überwiegende Mehrzahl aller Transaktionen greift nur lesend auf die Komponentensysteme zu. In einer solchen Umgebung ist das Potential für Transaktionsabbrüche wegen eines gemeinsamen Zugriffs auf das Ticketobjekt relativ gering, so dass auf eine zwangsweise Sequentialisierung der globalen Transaktionen verzichtet werden kann. Diese Technik behandelt nicht eine globale Transaktion als ganzes, sondern jede Subtransaktion für sich. Die Subtransaktionen werden wie üblich ausgeführt und führen erst an ihrem Ende den Ticketzugriff durch. Es genügt also, sie erst unmittelbar vor ihrem Commit als Lese- oder Änderungssubtransaktion zu kennzeichnen. Subtransaktionen, die nur lesend auf eine Komponentendatenbank zugreifen, müssen das Ticket nur lesen, die übrigen Subtransaktionen müssen das Ticket lesen und erhöhen. Ändern dabei zwei nebenläufig ablaufende Subtransaktionen das Ticket, wird eine von ihnen und damit auch die Transaktion, zu der sie gehört, abgebrochen. Dadurch erzielt man natürlich wieder eine sequentielle Ausführung von globalen Transaktionen, die Daten in der gleichen Komponentendatenbank geändert haben; globale Transaktionen, die in disjunkten Datenbanken Daten ändern, können dagegen parallel ablaufen.

Weil bei der optimistischen Technik erst am Ende einer Subtransaktion festgelegt sein muss, ob sie nur gelesen oder auch Daten geändert hat, ist jetzt auch in eingeschränktem Umfang eine automatische Ableitung dieser Eigenschaft denkbar. Man kann dazu die Operationen analysieren, die von der Subtransaktion abgesetzt wurden, um festzustellen, ob sie jemals während ihrer Laufzeit die Datenbank geändert hat. Dieses Verfahren erfordert aber mehr Aufwand der föderativen Ebene als das erste, daher wird man üblicherweise das erste Verfahren vorziehen. Dieser Ansatz ist darüber hinaus fehleranfällig, wenn es in der lokalen Datenbank aktive Elemente (wie z.B. Trigger) gibt, die selbständig eine Änderung der Datenbank auslösen können, ohne dass die Subtransaktion selbst dies veranlasst. Dies kann zum Beispiel der Fall, wenn über einen Trigger Zugriffe auf eine Tabelle gespeichert werden. In diesem Fall muss die Existenz dieses Triggers dem föderierten System bekannt sein, wenn man nicht vollständig auf solche aktiven Elemente verzichten will. Vergleichbare Probleme treten auf, wenn es möglich ist, über das föderative Interface Methoden in der Datenbank (z.B. Stored Procedures) aufzurufen, über deren interne Realisierung und damit über ihre Effekte auf die Datenbank nichts bekannt ist. In solchen Fällen muss man in der Regel konservativ annehmen, dass eine solche Methode und damit die gesamte Subtransaktion Änderungen ausgeführt hat.

Die Behandlung von reinen Lesetransaktionen ist bei der pessimistischen und der optimistischen Technik gleich. Die optimistische Technik wendet darüber hinaus das hier vorgestellte Verfahren auch für Subtransaktionen an, die ausschließlich lesend auf eine Komponentendatenbank zugreifen. Wenn wir im folgenden von Lesetransaktionen sprechen, meinen wir auch diese Art von Subtransaktionen.

Reine Lesetransaktionen werden in Systemen, die Snapshot Isolation gewährleisten, in der Regel weder blockiert noch aus Gründen der Concurrency Control vom System abgebrochen, solange das System noch alle benötigten Versionen der gelesenen Objekte finden kann. Sie lassen sich daher üblicherweise mit hoher Performance ausführen. Der übliche Ticketzugriff verursacht nun aber, wie wir bereits gesehen haben, potentiell viele Abbrüche und damit einen erheblichen Performanceverlust. Eine sorgfältige Analyse der Situation zeigt aber, dass es für die Korrektheit bereits genügt, wenn reine Lesetransaktionen das Ticketobjekt nur lesen und damit die Performancenachteile praktisch vermeiden.

Der Ticketwert einer Subtransaktion zeigt ihre relative Position im zu der tatsächlichen Ausführung im lokalen System äquivalenten seriellen Schedule. In einem System DB_k, dass Snapshot Isolation gewährleistet, hängt der Wert des Tickets der zugehörigen Subtransaktion einer Transaktion t_i nur von ihrem lokalen Startzeitpunkt, also von der Position von $B_i^{(k)}$, ab. Die Subtransaktion liest nur von anderen Transaktionen, deren Commit vor $B_i^{(k)}$ liegt, also muss ihre Position im äquivalenten Schedule hinter diesen liegen. Ihr Ticketwert muss also größer sein als der von allen Transaktionen, die vorher mit Commit beendet wurden. Andererseits sieht sie keine Änderungen von Transaktionen, deren Commit nach $B_i^{(k)}$ liegt, also liegt sie im äquivalenten seriellen Schedule vor diesen. Ihr Ticketwert muss daher kleiner sein als der Ticketwert der später mit Commit beendeten Transaktionen. Parallel oder nacheinander ablaufende reine Lesetransaktionen beeinflussen sich gegenseitig nicht, ihre Ticketwerte ergeben sich also alleine aus ihrem Verhältnis zu schreibenden Transaktionen.

Eine möglicher korrekter Ticketwert für eine reine Lesetransaktion ist also ein Wert, der zwischen dem aus der Datenbank gelesenen Ticketwert und dem nächsten Wert des

Tickets liegt, der ihm von einer folgenden Transaktion zugewiesen wird. Dieser Ansatz lässt sich sehr einfach implementieren: Hat zum Beispiel das Ticketobjekt einen ganzzahligen numerischen Wert, können wir dafür sorgen, dass Schreibtransaktionen immer nur gerade Ticketwerte in die Datenbank schreiben, etwa indem sie den aktuellen Ticketwert immer um 2 erhöhen. Reine Lesetransaktionen verwenden dann statt des gelesenen Ticketwerts den nächsthöheren ungeraden Wert. Mehrere reine Lesetransaktionen können auf diese Weise den gleichen Ticketwert erhalten, aber das beeinflusst die Korrektheit nicht.

Wir erhalten also das folgende Theorem:

Theorem 3.1: (Optimistische erweiterte Tickettechnik in SI-Systemen)

In einem föderierten Datenbanksystem ohne lokale Transaktionen, dessen Komponentensysteme Snapshot Isolation garantieren, werde ein Ticketgraph nach den folgenden Regeln verwaltet:

- Eine Subtransaktion einer globalen Transaktion, die in einer Komponentendatenbank Änderungen vorgenommen hat, liest vor ihrem Commit den aktuellen Wert des Tickets in dieser Datenbank und schreibt ihn um 2 erhöht zurück; dieser neue Wert ist ihr Ticket.
- Eine Subtransaktion einer globalen Transaktion, die nur lesend auf eine Komponentendatenbank zugegriffen hat, liest vor ihrem Commit den aktuellen Wert des Tickets in dieser Komponentendatenbank; sie verwendet den um 1 erhöhten Wert als ihr Ticket.
- Der Ticketgraph hat die globalen Transaktionen als Knoten und für jedes Komponentensystem, in dem beide Aktionen ausgeführt haben, eine Kante von t_i nach t_j (bzw. von t_j nach t_i), wenn das Ticket der Subtransaktion von t_i in dieser Komponentendatenbank kleiner (größer) als das der Subtransaktion von t_j ist.
- Eine mit Commit beendete Transaktion kann zusammen mit allen ihren Kanten aus dem Graphen entfernt werden, wenn alle Transaktionen, die gleichzeitig mit ihr aktiv waren, beendet sind und sie eine Quelle im Graph ist.
- Eine abgebrochene Transaktion wird zusammen mit allen ihren Kanten aus dem Graphen entfernt.

Wenn Transaktionen zurückgesetzt werden, die zu einem Zyklus im Ticketgraph führen, dann stellt dieses Verfahren globale Serialisierbarkeit sicher.

■

Beweis:

Wir müssen zeigen, dass durch die spezielle Art des Ticketzugriffs lokal serialisierbare Schedules entstehen und dass die Serialisierungsreihenfolge durch das Verhältnis der Ticketwerte widergespiegelt wird. Die Korrektheit des Verfahrens folgt dann sofort aus der Korrektheit der üblichen Tickettechnik.

Es ist klar, dass die lokalen Schedules serialisierbar sind, da alle Subtransaktionen, die Änderungen ausführen, das Ticket ändern und somit zwangsweise sequentiell ausgeführt werden, wie wir in Abschnitt 3.6.2.1 gezeigt haben. Das Verhältnis ihrer Ticketwerte gibt außerdem ihre Ausführungsreihenfolge wieder.

Für eine Subtransaktion t_S, die nur gelesen hat, entscheidet ihr Startzeitpunkt darüber, welchen Zustand der Datenbank sie sieht. Sei t_i die letzte Subtransaktion, die Änderungen ausgeführt hat und vor dem Anfang von t_S mit Commit beendet wurde, und sei t_k die nächste folgende Subtransaktion, die Änderungen ausführt. Dann ist t_i vollständig vor t_S abgelaufen und steht auch im äquivalenten seriellen Schedule vor t_S, da t_S die Änderungen von t_i sehen kann. Insbesondere liest t_S den Ticketwert, den t_i geschrieben hat. Der Ticketwert von t_S ist also um eins höher als der von t_i, so dass das Verhältnis der Werte ihre Serialisierungsreihenfolge wiedergibt. Transaktion t_k ist entweder nebenläufig mit t_S oder beginnt nach dem Ende von t_S. In beiden Fällen sieht t_S die Änderungen von t_k nicht, so dass t_S im äquivalenten Schedule vor t_k stehen muss. Der Ticketwert von t_k ist um zwei größer als der von t_i, also insbesondere auch größer als der von t_S, daher gibt auch das Verhältnis der Ticketwerte von t_S und t_k ihre Serialisierungsreihenfolge wieder. ■

Auch die Korrektheit des pessimistischen Ansatzes für Transaktionen, die Änderungen vornehmen, folgt unmittelbar aus diesem Theorem, da er lediglich noch mehr Subtransaktionen sequentiell ausführt, ansonsten aber die gleichen Argumente gelten. Als Beispiel für die Anwendung des dem pessimistischen Ansatz für schreibende Transaktionen betrachten wir den folgenden Schedule, in dem wegen der besseren Übersichtlichkeit die Ticketoperationen und die Lesezugriffe von t_1, t_2 und t_3 weggelassen wurden:

DB_1: $w_1(x)$ c_1 $r_W(x)$ $w_2(x)$ c_2 $r_B(x)$ c_W c_B

DB_2: $w_1(y)$ c_1 $r_W(y)$ $r_B(y)$ $w_3(y)$ c_3 c_W c_B

DB_3: $w_1(z)$ c_1 $w_2(z)$ c_2 $r_W(z)$ $r_B(z)$ $w_3(z)$ c_3 c_W c_B

Die schreibenden Transaktionen t_1 bis t_3 werden sequentiell ausgeführt. Transaktion t_1 schreibt in DB_1 den Ticketwert 6, in DB_2 den Wert 4 und in DB_3 den Wert 8, t_2 den Wert 8 in DB_1 und 10 in DB_3, und t_3 6 in DB_2 und 12 in DB_3. Parallel dazu laufen zwei reinen Lesetransaktionen t_B und t_W, die Aktionen in allen drei Komponentendatenbanken ausführen. Wenn wir uns t_W in der ersten Datenbank DB_1 ansehen, stellen wir fest, dass der Ticketwert für diese Subtransaktion größer als 6 sein muss (denn das ist der Wert, den t_1 geschrieben hat), aber kleiner als 8 (der Wert, den t_2 geschrieben hat). Ein möglicher Ticketwert für t_W in dieser Datenbank ist also 7. Analog ergeben sich die Ticketwerte für die übrigen Subtransaktionen.

Mit diesen Ticketwerten können wir nun überprüfen, ob die Ausführung der beiden reinen Lesetransaktionen zu einer global serialisierbaren Ausführung führt, d.h., ob der globale Ticketgraph zyklusfrei ist. Wir betrachten dazu zunächst Transaktion t_B. In den Datenbanken DB_1 und DB_3 beginnt sie ihre Ausführung nach dem Commit von t_2 (und t_1), also ist ihr Ticketwert größer als der von t_2 und t_1. (Zum Beispiel ist der Ticketwert von t_2 in DB_1 8, während der von t_B 9 ist.) In Datenbank DB_2 hat t_2 keine Operationen ausgeführt, und t_B wurde begonnen, nachdem t_1 mit Commit beendet wurde. Zusammenfassend beginnt also t_B immer nach dem Commit von t_1 und t_2. Analog kann man zeigen, dass t_B immer vor dem Beginn von t_3 beginnt. Der Ticketgraph ist also zyklusfrei; in einem äquivalenten Schedule muss t_B nach t_1 und t_2, aber vor t_3 ausgeführt werden.

```
void OptimisticCommit(transaction ti) raises RolledBack
{
  for all databases db where ti had operations
  {
    if ti was read-only in db then
      ticket:= select value+1 from db.ticket;
    else
      update db.ticket set value=value+2;
    ticket:= select value from db.ticket;
    for all transactions t in TicketGraph that had operations in db
    {
      if t's ticket in db < ticket then add edge t→ti
      else if t's ticket in db > ticket then add edge ti→t
    }
  }
  if (cycletest()=true) then
  { Abort ti;
    raise RolledBack;
  }
}

void PessimisticCommit(transaction ti) raises RolledBack
{
  for all databases db where ti had operations
  {
    if ti was read-only then
      ticket:= select value+1 from db.ticket;
    else
      update db.ticket set value=value+2;
    ticket:= select value from db.ticket;
    for all transactions t in TicketGraph that had operations in db
    {
      if t's ticket in db < ticket then add edge t→ti
      else if t's ticket in db > ticket then add edge ti→t
    }
  }
  if (cycletest()=true) then
  { Abort ti;
    raise RolledBack;
  }
}
```

Abbildung 3.1 - Pseudocode der Commitoperationen der erweiterten Tickettechniken

Transaktion t_W führt ihre ersten Operationen in DB_2 und DB_3 aus, bevor t_3 mit Commit beendet wurde; ihre Ticketwerte in beiden Datenbanken sind daher niedriger als die von t_3. Andererseits beginnt sie in DB_1 vor dem Commit von t_2, in DB_3 dagegen erst nach dem Commit von t_2; t_W sieht daher in DB_3 Objekte, die t_2 vorher geändert hat, in DB_1 aber nicht. Der Zustand der globalen Datenbank, den t_W sieht, ist also inkonsistent. Die Ticketwerte von t_2 und t_W erlauben es, dieses Problem festzustellen: Der Ticketwert von t_W ist in DB_1 kleiner als der von t_2 (7 bzw. 8), während er in DB_3 größer ist (11 bzw. 10). Der globale Ticketgraph hat daher einen Zyklus zwischen t_W und t_2, der (spätestens) erkannt wird, wenn t_W ihr Commit ausführen will.

Abbildung 3.1 zeigt noch einmal zusammenfassend die Commitoperationen der beiden Varianten der erweiterten Tickettechnik als Pseudocode. Man sieht dabei deutlich, dass

die optimistische Variante im Unterschied zur pessimistischen Variante auf Subtransaktionsbasis entscheidet, welche Art des Ticketzugriffs gemacht werden muss.

Lokale Transaktionen werden ähnlich wie globale Subtransaktionen in der optimistischen Technik behandelt: Lokale Transaktionen, die die Datenbank verändert haben, müssen das Ticket lesen und schreiben; solche, die nur lesend auf die Datenbank zugegriffen haben, müssen überhaupt nicht auf das Ticket zugreifen, weil sie keinen Einfluss auf die Serialisierungsreihenfolge des lokalen Schedules haben können.

Obwohl die sequentielle Ausführung von Transaktionen, die Änderungen in irgendeiner Komponentendatenbank (bei der pessimistischen Methode) bzw. in mindestens einer gemeinsamen Datenbank (bei der optimistischen Methode) ausführen, sehr restriktiv ist, ist dieser Ansatz nicht schlechter als die übliche Tickettechnik. Für Transaktionen, die nur lesend auf lokale Datenbanken zugreifen, ist dieser neue Ansatz dagegen sehr effizient und führt nur dann zu Abbrüchen, wenn die globale Serialisierbarkeit sonst nicht mehr gewährleistet werden könnte. Die optimistische Variante wendet dieses leistungssteigernde Verfahren auch auf nur lesende Subtransaktionen von Transaktionen an, die in anderen Komponentensystemen auch Änderungen vornehmen, und hat so ein noch besseres Leistungspotential, wenn die Zugriffsmuster der ausgeführten Transaktionen dies zulassen.

3.6.2.3 Mehrschichtentransaktionen

Um Mehrschichtentransaktionen anwenden zu können, müssen alle lokalen Scheduler grundsätzlich ordnungserhaltend konfliktserialisierbare Schedules erzeugen. Garantiert mindestens einer der lokalen Scheduler nur Snapshot Isolation, ist diese Voraussetzung nicht mehr gegeben. Lediglich konfliktserialisierbare Schedules genügen dann, wenn Konflikttreue gegeben ist, d.h. wenn Operationen, die auf einer höheren Schicht (hier L_1) nicht vertauschbar sind, auch auf der Schicht darunter (hier L_0) mindestens einen Konflikt haben.

Wir betrachten den Spezialfall, dass auf der föderierten Ebene nur SQL-Anweisungen, aber keine Methodenaufrufe ausgeführt werden, und zeigen zunächst, dass dann lokal serialisierbare Ausführungen entstehen, selbst wenn einige lokale Systeme nur Snapshot Isolation gewährleisten. Anschließend zeigen wir, dass dann auch Ordnungserhaltung vorliegt. Zusammen ergibt dies die Anwendbarkeit des Mehrschichtenansatzes auf Föderationen, in denen manche Systeme konfliktserialisierbare Schedules erzeugen, aber manche nur Snapshot Isolation gewährleisten. Dieser Fall ist sehr praxisrelevant, weil man damit Oracle-Systeme in eine solche mehrschichtige Architektur einbinden kann.

In Abschnitt 3.4.6 hatten wir das semantische Sperrverfahren beschrieben, das wir auf L_1 einsetzen. Es verwendet die Prädikate der SQL-Anweisungen, die auf L_1 ausgeführt werden, um eine konservative Beschreibung der Objekte abzuleiten, auf die während der Ausführung einer SQL-Anweisung auf L_0 zugegriffen wird. Man wird dabei im allgemeinen wesentlich mehr Objekte sperren, als tatsächlich notwendig wäre, weil man z.B. bei Joins oder komplexen Anfragen mit Subqueries aus Aufwandsgründen einfachere Prädikate ableiten muss. Für die geschachtelte Anfrage

```
update kunden k set bonus=1 where k.knr in (select knr from bestellungen)
```

müsste man z.B. Tabellensperren, d.h. das Prädikat `true`, auf beiden Tabellen erwerben, und zwar eine exklusive Sperre auf der Tabelle `kunden` und eine Lesesperre auf der Ta-

belle `bestellungen`. Wir können das folgende Lemma aufstellen, das Eigenschaften unseres Sperrverfahrens beschreibt:

Lemma 3.1: (Eigenschaften des Sperrverfahrens)

Das Sperrverfahren, das auf L_1 eingesetzt wird, hat die folgenden Eigenschaften:

(i) Die Sperrprädikate, die zu einer L_1-Operation gehören, beschreiben eine echte oder unechte Obermenge der Objekte, auf die während der Ausführung der Operation auf L_0 zugegriffen wird.

(ii) Wenn zwei L_1-Operationen auf L_0 auf das gleiche Objekt mit in Konflikt stehenden Operationen zugreifen, so sperren sie auf L_1 überlappende Prädikate in unverträglichen Sperrmodi.

(iii) Wenn zwei L_1-Operationen auf L_1 überlappende Prädikate in unverträglichen Sperrmodi sperren, so müssen sie keinen Konflikt auf L_0 haben. ■

Beweis:

Teil (i) folgt unmittelbar aus der Arbeitsweise des Sperrverfahrens. Teil (ii) gilt, weil die Prädikate auf L_1 immer mindestens die Objekte beschreiben, auf die während der Ausführung einer Operation auf L_0 zugegriffen wird. Andererseits können die Prädikate zu viele Objekte beschreiben, weil sie konservativ gewählt werden, daher kann es auf L_1 einen Konflikt geben, der auf L_0 nicht auftritt, was Teil (iii) zeigt. ■

Weil wir jede SQL-Anweisung in einer eigenen Subtransaktion ausführen, können wir alle L_1-Sperren, die für diese Anweisung notwendig sind, erwerben, bevor die Anweisung selbst ausgeführt wird. Für lokale Transaktionen, die mehr als eine SQL-Anweisung ausführen, ist dies nicht ohne weiteres möglich, ohne die zugehörigen Transaktionsprogramme zu verändern. Wir konzentrieren uns daher im folgenden auf föderierte Systeme ohne lokale Transaktionen. Am Ende des Abschnitts diskutieren wir kurz den Einfluss lokaler Transaktionen. Aus dem Lemma folgt insbesondere, dass unsere L_1-Operationen nicht konflikttreu sind, weil es auf L_1 Konflikte geben kann, zu denen keine korrespondierenden Konflikte auf L_0 gehören. Wir können aber mit Hilfe des Lemma das folgende Theorem zeigen:

Theorem 3.2: (Eigenschaften möglicher lokaler Schedules)

Wenn alle Sperren, die für die Ausführung einer Subtransaktion benötigt werden, am Anfang der Subtransaktion erworben werden, dann sind die Schedules, die auf L_0 entstehen können, ordnungserhaltend serialisierbar.

■

Beweis:

Wir zeigen zunächst, dass alle möglichen lokalen Schedules serialisierbar sind. In Abschnitt 2.4 hatten wir gezeigt, dass die nichtserialisierbaren Schedules, die unter Snapshot Isolation möglich sind, gerade folgende Gestalt haben:

$$r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ r_2(y_0)\ w_1(x_1)\ c_1\ w_2(y_2)\ c_2$$

Hier kann es zu einer Verletzung einer Bedingung zwischen x und y (z.B. x+y>1000) kommen, und zwar selbst dann, wenn jede Transaktion die Einhaltung dieser Bedingung explizit prüft. Die Transaktionen t_1 und t_2 sind Subtransaktionen von verschiedenen globalen Transaktionen g_1 und g_2. Auf L_1 hat daher g_1 eine Lesesperre auf einen Prädikat erworben, das nach Lemma 3.1(i) (mindestens) die Objekte x und y beschreibt, ebenso g_2. Zusätzlich muss g_1 eine Schreibsperre auf einem Prädikat erworben haben, das mindestens Objekt x beschreibt, und g_2 eine Schreibsperre auf einem Prädikat, das mindestens Objekt y beschreibt. Das kann aber nicht der Fall sein, weil g_1 seine Schreibsperre erst dann erwerben kann, wenn g_2 seine Lesesperre freigegeben hat, da sich die beiden Prädikate überdecken müssen, weil sie beide mindestens Objekt x beschreiben. Analog kann man zeigen, dass g_2 seine Schreibsperre erst erwerben kann, wenn g_1 seine Lesesperre freigegeben hat. Es muss also zu einer globalen Verklemmung gekommen sein, so dass der gezeigte lokale Schedule nicht aufgetreten sein kann.

Für andere, denkbare Ausführungsreihenfolgen kann man mit ähnlichen Argumenten zeigen, dass keine Probleme entstehen können. Zusammenfassend kann man also sagen, dass unter diesen Voraussetzungen lokale Serialisierbarkeit trotz lokaler Snapshot Isolation gewährleistet werden kann.

Zusätzlich müssen wir noch zeigen, dass die möglichen lokalen Schedules auch ordnungserhaltend serialisierbar sind. Wir nehmen dazu an, dass zwei L_1-Operationen der globalen Transaktionen g_1 und g_2 auf L_1 einen Konflikt haben, auf L_0 dagegen konfliktfrei sind. Dies kann vorkommen, weil die Prädikate nur eine Obermenge der Objekte beschreiben, auf die die Subtransaktionen tatsächlich zugreifen. Sie werden in Subtransaktionen t_1 bzw. t_2 vollständig nacheinander ausgeführt, sie seien aber im äquivalenten seriellen lokalen Schedule vertauscht. Dann muss es eine andere Subtransaktion t_3 einer globalen Transaktion g_3 geben, die parallel zu t_1 und t_2 läuft und die vor t_1, aber nach t_2 serialisiert werden muss. Wir leiten nun ab, welche Arten von Konflikt es zwischen den einzelnen Transaktionen geben kann.

Dazu betrachten wir zunächst t_1 und t_3. Weil t_3 vor t_1 serialisiert wird und nicht beide das gleiche Objekt verändern können, muss t_3 ein Objekt lesen, das t_1 verändert. (Würden sie beide das gleiche Objekt nur lesen, wären sie nicht in Konflikt.) Wegen Lemma 3.1(i) haben dann auch die Prädikate, die für t_1 und t_3 auf L_1 gesperrt werden, eine Überlappung und werden in unverträglichen Modi gesperrt. Dann muss aber die zweite der beiden Transaktionen warten, bis die erste ihre L_1-Sperre freigegeben hat. Weil die Anforderung aller notwendigen L_1-Sperren am Beginn einer Subtransaktion vor der Ausführung von Operationen in der Komponentendatenbank geschieht, kann die Sperre für t_3 nicht nach dem Commit von g_1 angefordert worden sein, da sonst t_1 und t_3 nicht nebenläufig sein könnten. Auf L_1 muss also das Commit von g_3 vor der Sperranforderung für t_1 ausgeführt werden – aber auch dann können t_1 und t_3 nicht parallel gelaufen sein. Es kann also keine Subtransaktion t_3 geben, die parallel zu t_1 ausgeführt wird und einen Wert liest, den t_1 ändert.

Nun betrachten wir t_2 und t_3. Mit analogen Argumenten wie oben ergibt sich, dass t_2 und t_3 nicht parallel ausgeführt werden können. Die geordneten Transaktionen t_1 und t_2 bleiben also auch im äquivalenten seriellen Schedule in dieser Reihenfolge. ■

Was passiert nun, wenn nicht alle Sperren, die während der Ausführung einer Subtransaktion benötigt werden, zu ihrem Beginn erworben werden? Insbesondere lokale Trans-

aktionen gehören zu dieser Kategorie, wenn man sie nicht anpassen kann. Erlaubt man in der obigen Diskussion t_3, während seiner Ausführung Sperren zu erwerben, kann keine Ordnungserhaltung mehr garantiert werden. Wenn t_3 vor dem Beginn von t_2 auf Objekte zugreift, die t_2 nicht benutzt, beginnt t_3 lokal vor t_2. Selbst wenn dann t_3 mit dem Ändern des kritischen Objektes, das t_2 vorher gelesen hat, bis nach dem Commit von g_2 wartet, sind t_2 und t_3 immer noch parallel, und es ergibt sich die Abhängigkeit $t_2 \rightarrow t_3$. Analog kann t_3 auch nach dem Commit von g_1 noch den alten Wert des Objektes lesen, den t_1 geändert hat, weil lokal Snapshot Isolation garantiert wird und t_3 vor t_1 begonnen hat, so dass sich die Abhängigkeit $t_3 \rightarrow t_1$ ergibt und somit eine Vertauschung von t_1 und t_2 im äquivalenten lokalen Schedule. Wir dürfen also nur solche lokalen Transaktionen erlauben, die alle Sperren, die sie während ihres Ablaufs benötigen werden, an ihrem Beginn anfordern.

Werden auf der föderierten Ebene Methoden bereitgestellt, z.B. von einem CORBA-Server, der selbst wieder auf eine Oracle-Datenbank zugreift, muss die Implementierung dieser Methoden bekannt sein. Nur dann ist es möglich, Prädikate anzugeben, die die Objektmenge charakterisieren, auf die während der Ausführung der Methoden zugegriffen wird. Andernfalls bleibt es als einziger Ausweg, die Methoden zu identifizieren, bei denen diese Probleme auftreten können, und die ihnen zugeordneten semantischen Sperren so stark zu machen, dass die problematischen Fälle ausgeschlossen werden. Gegebenenfalls bedeutet dies, alle Operationen seriell auszuführen, wenn man nicht ausschließen kann, dass es zu den gezeigten Anomalien kommt. Offensichtlich ist eine sehr sorgfältige Analyse der Anwendung notwendig, um diesen Weg wählen zu können, wenn man nicht konservativ sperren und damit möglicherweise sehr viel Performance verlieren will.

Mehrschichtentransaktionen können also in Umgebungen, in denen manche Systeme nur Snapshot Isolation gewährleisten, nur dann „narrensicher“, also bedenkenlos verwendet werden, wo auf der föderierten Ebene nur Anweisungen in der Datenbanksprache abgesetzt werden, die jeweils in einer Subtransaktion gekapselt sind, oder Methoden mit bekannter Implementierung aufgerufen werden. Außerdem darf es keine lokalen Transaktionen geben, bzw. die vorhandenen lokalen Transaktionen müssen zu ihrem Beginn alle Sperren anfordern, die sie während ihrer Ausführung benötigen werden. Andernfalls muss man auf der Ebene der Applikationslogik argumentieren, dass es nicht zu Problemen kommen kann.

3.6.3 Ein Graphalgorithmus für globale Serialisierbarkeit

Der Online-SI-MVSG, oder kürzer OSI-MVSG, den wir in Abschnitt 2.4.4 eingeführt haben, erlaubt es, Serialisierbarkeit sicherzustellen, obwohl der Scheduler der Datenbank nur Snapshot Isolation gewährleistet. Wie wir gezeigt haben, müssen wir lediglich solche Operationen ablehnen, die zu einem Zyklus im OSI-MVSG führen, indem wir die betreffende Transaktion zurücksetzen. Wir können diesen Algorithmus auch zur globalen Concurrency Control in föderierten Datenbanksystemen verwenden.

Der globale OSI-MVSG, der auf der globalen Ebene verwaltet werden muss, ist die Vereinigung der OSI-MVSG, die wir separat für jedes lokale Datenbanksystem verwalten. Insbesondere betrachten wir für den OSI-MVSG eines Komponentensystems nur die entsprechenden Subtransaktionen der globalen Transaktionen in diesem System. Wir müssen außerdem sicherstellen, dass kein Knoten zu früh aus dem globalen Graphen

entfernt wird: Im globalen OSI-MVSG darf eine Transaktion erst dann aus dem Graphen entfernt werden, wenn sie eine Quelle im Graph ist und alle Transaktionen beendet sind, die in irgendeinem lokalen System parallel zu ihr gelaufen sind. Wir erhalten also das folgende Ergebnis:

Theorem 3.3: (globaler OSI-MVSG)

In einem föderierten Datenbanksystem, dessen Komponentensysteme Snapshot Isolation garantieren, erlaubt ein Algorithmus zur föderierten Concurrency Control, der auf dem Zyklustest im globalen OSI-MVSG basiert, nur global serialisierbare Schedules, wenn er jede Operation ablehnt, die zu einem Zyklus im globalen OSI-MVSG führt. ■

Beweis:

Wir haben in Theorem 2.5 gezeigt, dass mit jedem lokalen OSI-MVSG für das zugehörige Komponentensystem Serialisierbarkeit gewährleistet werden kann. Auch dieser Algorithmus sorgt für serialisierbare lokale Schedules, weil jeder Zyklus in einem lokalen OSI-MVSG auch ein Zyklus im globalen OSI-MVSG ist, so dass eine der Transaktionen auf dem Zyklus zurückgesetzt wird. Wir müssen nun noch zeigen, dass die Serialisierungsreihenfolgen der globalen Transaktionen in allen Komponentensystemen verträglich sind. Weil die lokalen OSI-MVSG aber gerade die lokale Serialisierungsreihenfolge beschreiben, führen unverträgliche lokale Serialisierungsreihenfolgen zu einem Zyklus im globalen OSI-MVSG, der von unserem Algorithmus erkannt und behandelt wird. ■

Um die Kanten des Graphs konstruieren zu können, müssen wir die Lese- und Schreiboperationen der Transaktionen in den Komponentensystemen beobachten. Die üblichen Datenbanksysteme bieten jedoch an ihrem Interface keine Methoden dafür an, so dass wir selbst eine Näherung der Objektmengen bestimmen müssen, die föderierte Transaktionen in den verschiedenen Komponentensystemen gelesen und geändert haben. Wir können dazu nur die Operationen verwenden, die sie auf der föderierten Ebene abschicken. Eine Lösung dieses Problems wurde in [SSW95] vorgestellt, die potentielle Konflikte zwischen den Prädikaten in SQL-Anweisungen findet. Da jede Aktion einer föderierten Transaktion über die föderierte Ebene abgewickelt wird, kennen wir jede Aktion und können daraus Prädikate ableiten, die die Objekte charakterisieren, die diese Aktion betroffen hat.

Lesezugriffe auf Objekte werden dabei normalerweise von SELECT-Anweisungen verursacht, daher können wir das Suchprädikat einer solchen Anfrage verwenden, um alle Objekte zu charakterisieren, die von dieser Anfrage zurückgegeben werden. Wenn eine Subtransaktion zum Beispiel die Anfrage

```
SELECT P FROM PARTS WHERE P.COLOR=GREEN OR (P.PRICE>100 AND P.WEIGHT>7)
```

absetzt, charakterisieren die Prädikate COLOR=GREEN und PRICE=100 AND WEIGHT<7 exakt die Teilmengen der Tabelle PARTS, die die Subtransaktion liest.

Elementare Ausdrücke in den zur Charakterisierung verwendeten Prädikaten sind Vergleiche von Objektattributen mit Skalaren. Wenn die Anfrage einen Join enthält, wird das Anfrageprädikat in einzelne Teilprädikate für jede beteiligte Tabelle aufgeteilt. Das

Join-Prädikat selbst kann dabei ignoriert werden, aber die übrigen Teilprädikate werden berücksichtigt. Auf diese Weise wird die Menge der Objekte, auf die bei der Auswertung des Joins zugegriffen wird, höchstens über-, aber keinesfalls unterschätzt. Enthält die Anfrage eine Unteranfrage, müssen auch die Objekte repräsentiert werden, auf die bei der Auswertung dieser Unteranfrage zugegriffen wird. Weil die so entstehenden Prädikate schnell sehr unhandlich werden und nicht mehr effizient miteinander verglichen werden können, müssen die Prädikate vereinfacht werden, also mehr Objekte als unbedingt nötig beschreiben. Im Extremfall kann das zum Prädikat `true` führen, das alle Objekte einer Relation beschreibt.

Auf die gleiche Weise lassen sich Prädikate ableiten, die die Objekte beschreiben, die von Einfüge-, Lösch- und Änderungsoperationen betroffen sind.

Basierend auf diesen Prädikaten können wir nun bestimmen, ob das Writeset von t_i, also die Menge der Objekte, die t_i geändert hat, und das Readset von t_j *potentiell überlappen*, d.h. ob die Konjunktion der Prädikate erfüllbar ist. Dieser Test kann möglicherweise zu viele Überlappungen ergeben, da wir die Menge der Objekte, auf die eine Transaktion zugreift, auf diese Weise nur approximieren. Wir sind dabei aber konservativ, d.h. alle wirklichen Überlappungen werden durch diesen Test gefunden, daher können wir die Korrektheit eines Algorithmus garantieren, der diese approximativen Ergebnisse verwendet.

```
transaction transactions[];

void AddReadEdge(transaction ti, predicate p) raises CycleDetected
{
  for all t in transactions
  {
    if (t≠ti) then
      if (t.writeset overlaps p) then
        if (t is commmitted) then add edge t→ti;
        else add edge ti→t;
      end if;
    end if;
  }
  if (cycletest()=true) raise CycleDetected;
}

void AddWriteEdge(transaction ti, predicate p) raises CycleDetected
{
  for all t in transactions
  {
    if (t≠ti) then
      if (t.readset overlaps p) then add edge t→ti;
      if ((t.writeset overlaps p)and(t is not commmitted)) then
        add edge t→ti;
    end if;
  }
  if (cycletest()=true) raise CycleDetected;
}
```

Abbildung 3.2 – Pseudocode für das Einfügen von Kanten in den Online SI-MVSG

Die Realisierbarkeit des Einsatzes von Prädikaten zur Approximation von Read- und Writesets, insbesondere die Anwendbarkeit auf eine ausreichend große Klasse von praktisch relevanten SQL-Anweisungen, wurde bereits experimentell in [SSW95] bestätigt.

Basierend auf dieser Repräsentation einer Approximation der Read- und Writesets von Transaktionen können wir nun den globalen Online SI-MVSG aktualisieren, wenn globale Transaktionen Leseoperationen oder Modifikationen ausführen. Abbildung 3.2 zeigt die dazu notwendigen Operationen in Pseudocode.

Mit der hier gezeigten Technik zur Approximation der Read- und Writesets von Transaktionen können prinzipiell auch Verfahren entwickelt werden, die globale Serialisierbarkeit sicherstellen, wenn lokal schwächere Isolation Levels als Snapshot Isolation verwendet werden. Man konstruiert dazu auf der föderierten Ebene aus den approximierten Read- und Writesets sowie aus Informationen über Anfangs- und Endzeitpunkte von Subtransaktionen einen Konfliktgraphen für jedes lokale System. Da man aber keine genauen Informationen über die Ausführungen der Operationen hat, muss man starke Restriktionen fordern, damit die abgeleiteten Konflikte auch der Situation in den lokalen Datenbanken entsprechen. Hat man etwa lokal nur den Level Read Committed, so muss man verbieten, dass eine Transaktion ihr Commit ausführt, während eine Operation einer anderen Transaktion Daten liest, die die zu beendende Transaktion geschrieben hat. Vermeidet man solche Fälle nicht, kann man nicht mit Sicherheit sagen, ob die lesende Transaktion die Änderungen gesehen hat oder nicht. Unter lokaler Snapshot Isolation kann dieser Fall wegen der besonderen Versionsfunktion nicht vorkommen. Angesichts dieser drastischen Einschränkungen, die keine gute Performance solcher Algorithmen erwarten lassen, haben wir uns in dieser Arbeit nicht weiter damit befasst.

3.6.4 Kombinationsverfahren

Da föderierte Systeme in der Regel heterogen sind, wird man nur selten ausschließlich Komponentensysteme haben, die Snapshot Isolation garantieren. Statt dessen werden manche Systeme Snapshot Isolation, manche Systeme aber serialisierbare Schedules gewährleisten. In diesem Abschnitt stellen wir Erweiterungen der Verfahren vor, die wir in den vorherigen Abschnitten vorgestellt haben, so dass sie auch in dieser Situation global serialisierbare Schedules erzeugen.

Grundsätzlich sollten die jetzt vorgestellten Verfahren so kombiniert werden, dass jeweils das beste für jedes Komponentensystem ausgewählt wird. Dann müssen aber die jeweiligen Serialisierungsreihenfolgen in einen globalen Graphen eingetragen und dieser auf Zyklen getestet werden, wenn nicht gezeigt werden kann, das die Verfahren kompatible Reihenfolgen erzeugen.

Die erweiterte Tickettechnik kann sehr einfach mit Systemen kombiniert werden, die serialisierbare Schedules generieren: In diesen Datenbanksystemen wendet man einfach eine Variante der üblichen Tickettechnik an, die wir in Abschnitt 3.4.5 vorgestellt haben. Man kann insbesondere die verschiedenen Optimierungen, z.B. die implizite Tickettechnik, benutzen, falls die Eigenschaften der lokalen Systeme dies zulassen. Die Ticketkanten für alle beteiligten Systeme trägt man dann in den globalen Ticketgraphen ein. Wir erhalten das folgende Theorem:

Theorem 3.4: (Kombination der Tickettechniken)

Setzt man in einem föderierten Datenbanksystem in Komponentensystemen, die Snapshot Isolation garantieren, eine Variante der erweiterten Tickettechnik ein, und in Komponentensystemen, die serialisierbare Schedules erzeugen, die übliche Tickettechnik, so sind alle globalen Schedules serialisierbar, wenn man alle Ticketkanten in den gleichen Graphen einträgt und eine Transaktion zurücksetzt, deren Ticket zu einem Zyklus im globalen Graphen führt. ■

Beweis:

Die Kanten, die aufgrund der Verhältnisse der Ticketwerte der Subtransaktionen in den Komponentensystemen in den Graphen eingetragen werden, spiegeln gerade die Serialisierungsreihenfolgen in den Komponentensystemen wider. Der globale Schedule ist also genau dann serialisierbar, wenn der Ticketgraph keinen Zyklus enthält. ■

Es wäre insbesondere recht einfach, die optimistische Variante in bestehende Middleware zur Integration mehrerer Oracle-Instanzen (z.B. *Oracle Distributed Databases* [Orac99b]) einzubauen und so auch globale Serialisierbarkeit zu garantieren.

Auch die graphbasierte Technik aus Abschnitt 3.6.3 lässt sich auf diese Weise erweitern: Für Komponentensysteme, die serialisierbare Schedules erzeugen, wird wieder eine der Varianten der Tickettechnik benutzt. Die Ticketkanten trägt man als zusätzliche Kanten in den OSI-MVSG ein. Wir erhalten das folgende Theorem:

Theorem 3.5: (Kombination von OSI-MVSG und Tickettechniken)

Setzt man in einem föderierten Datenbanksystem in Komponentensystemen, die Snapshot Isolation garantieren, die auf dem Online-SI-MVSG basierende Technik ein, und in Komponentensystemen, die serialisierbare Schedules erzeugen, die übliche Tickettechnik, so sind alle globalen Schedules serialisierbar, wenn man die Ticketkanten als zusätzliche Kanten in den OSI-MVSG einträgt und eine Transaktion zurücksetzt, die zu einem Zyklus im globalen Graphen führt. ■

Beweis:

Sowohl die Ticketkanten als auch die Kanten im OSI-MVSG geben die Serialisierungsreihenfolgen in den lokalen Systemen wieder. Man erhält also globale Serialisierbarkeit, wenn man sicherstellt, dass der OSI-MVSG zu keinem Zeitpunkt einen Zyklus enthält. ■

3.6.5 Qualitativer Vergleich der vorgestellten Verfahren

Keines der in diesem Abschnitt vorgestellten drei Verfahren ist in allen Situationen besser als die übrigen. Jedes hat Vorteile in gewissen Situationen, aber auch Nachteile in anderen.

Das theoretisch leistungsfähigste Verfahren, was den möglichen Parallelitätsgrad von Transaktionen angeht, sind *Mehrschichtentransaktionen*, weil die Sperren in den Komponentensystemen nur für die Dauer einer Operation gehalten werden. Sobald aber lokal Snapshot Isolation verwendet wird, können sie nur in den Spezialfällen verwendet werden, wo auf der föderierten Ebene nur SQL-Anweisungen und Methoden, deren Implementierung bekannt ist, eingesetzt werden. Selbst dann können nur für einfache Anfragen und Änderungen exakte Prädikate gesperrt werden, ansonsten muss man konservative Näherungslösungen in Kauf nehmen, die die Performance nachteilig beeinträchtigen können. Ein weiterer wichtiger Nachteil ist, dass praktisch keine lokalen Transaktionen erlaubt werden dürfen, um die Korrektheit dieses Verfahrens garantieren zu können. Zur Laufzeit entsteht zusätzlicher Aufwand durch die Sperrverwaltung auf der föderierten Ebene und das Mitschreiben der Kompensationsaktionen für den Fall eines Abbruchs einer globalen Transaktion.

Die *erweiterte Tickettechnik* erfordert für reine Lesetransaktionen sehr wenig zusätzlichen Aufwand auf der föderierten Ebene und in Komponentensystemen, die Snapshot Isolation garantieren. Sie ist daher die Methode der Wahl für Anwendungen, die nahezu ausschließlich solche Transaktionen ausführen. Datenbanksysteme mit serialisierbaren lokalen Schedules lassen sich einfach integrieren, insbesondere erlauben es die verschiedenen Varianten der Tickettechnik, die notwendigen Zusatzmaßnahmen für diese Systeme auf die lokal garantierte Teilklasse von CSR abzustimmen. Für Transaktionen, die lesen und schreiben, hat die erweiterte Tickettechnik dagegen große Performancenachteile, weil sie Transaktionen diesen Typs sequentiell ausführt. In Anwendungsgebieten, die von solchen Transaktionen dominiert sind, kann sie daher praktisch nicht eingesetzt werden.

Das graphbasierte Protokoll schließlich, das auf dem OSI-MVSG basiert, muss vom Performancepotential her zwischen den beiden anderen Ansätzen eingeordnet werden. Es erlaubt insbesondere für Transaktionen, die lesen und schreiben, einen höheren Parallelitätsgrad als die erweiterte Tickettechnik. Durch den zusätzlichen Aufwand für die prädikatbasierte Verwaltung der Read- und Writesets ist die erzielbare Leistung für reine Lesetransaktionen dagegen geringer als bei dieser. Mehrschichtentransaktionen erfordern einen noch größeren Aufwand, erlauben aber mehr Parallelität, wenn Semantik von Methoden ausgenutzt werden kann.

Die beste Gesamtleistung erzielt man, wenn man für die Subtransaktionen in jedem Komponentensystem das Verfahren benutzt, das am besten auf die Anforderungen passt. Werden in einem Komponentensystem, das Snapshot Isolation garantiert, z.B. praktisch nur Subtransaktionen ausgeführt, die ausschließlich lesen, so bietet sich dort die erweiterte Tickettechnik an. Für solche Komponentensysteme, in denen sowohl reine Lese- als auch gemischte Subtransaktionen ausgeführt werden, empfiehlt sich die graphbasierte Technik. Datenbanksysteme, die serialisierbare Schedules generieren, können mit einer Variante der Tickettechnik integriert werden.

3.7 Gewährleistung globaler Isolation Levels

Im letzten Abschnitt haben wir gezeigt, wir man in föderierten Datenbanksystemen serialisierbare Schedules sicherstellen kann. Eine wichtige Voraussetzung war dabei, dass die lokalen Schedules selbst serialisierbar sind. In existierenden Systemen werden aber oft schwächere Isolation Levels eingesetzt, weil sie bessere Performance unter Preisga-

be eines Teils der Konsistenz versprechen. Auch in föderierten Systemen können Isolation Levels sinnvoll eingesetzt werden, wenn die eingesetzten Anwendungen mit der potentiell eingeschränkten Konsistenz der Daten leben können. In diesem Abschnitt diskutieren wir, welche Isolation Levels man für globale Schedules garantieren kann, wenn die lokalen Systeme selbst bestimmte Isolation Levels erzeugen.

Wir können zunächst das folgende einfache Theorem zeigen:

Theorem 3.6: (globaler und lokale Isolation Levels)

In einem föderierten Datenbanksystem ist der globale Isolation Level ohne zusätzliche Maßnahmen auf der föderierten Ebene nicht stärker als der schwächste der lokalen Isolation Levels. ■

Beweis:

Sei DB_k eines der Komponentensysteme, in dem der schwächste Isolation Level aller lokalen Systeme gewährleistet wird, wir nennen diesen Isolation Level L_k. Auf der globalen Ebene werde der Isolation Level L_G gewährleistet. Um zu zeigen, dass L_G nicht stärker als L_k ist, müssen wir nach Definition 2.13 zeigen, dass $N(L_G) \supseteq N(L_k)$ ist. Wir betrachten dazu einen Schedule in DB_k. Dieser Schedule kann als globaler Schedule aufgefasst werden, in dem alle Transaktionen ausschließlich Operationen in DB_k ausführen. Weil auf der globalen Ebene keine zusätzlichen Maßnahmen getroffen werden, wird dieser Schedule auch vom föderierten System zugelassen. ■

Wir haben damit aber noch keine Aussage darüber gemacht, wie stark L_G mindestens ist. Auf dieser abstrakten Ebene ist das auch nicht möglich, da L_G Schedules erlaubt, die auf Objekte in mehreren Datenbanken zugreifen und somit von keiner einzelnen Komponentendatenbank zugelassen werden. Jeder Versuch, $N(L_G)$ auf eine andere Weise als in Theorem 3.6 mit der Menge der Schedules zu vergleichen, die ein lokales System erlaubt, muss also fehlschlagen.

Für den Sonderfall, dass alle Komponentensysteme den gleichen Isolation Level garantieren, können wir für die üblichen Isolation Levels nach der verallgemeinerten Interpretation in Abschnitt 2.3.2.3 zeigen, dass dann auch auf der globalen Ebene dieser Isolation Level sichergestellt werden kann. Dazu müssen wir voraussetzen, dass ein atomares Commitprotokoll verwendet wird, das sich konzeptionell so verhält, als werde es von einem zentralen Koordinator gesteuert:

Definition 3.3: (ACP mit zentralem Koordinator)

Ein atomares Commitprotokoll *basiert konzeptionell auf einem zentralen Koordinator*, wenn es die beiden folgenden Eigenschaften hat:

- Die relative Ordnung der Commitoperationen von verschiedenen Transaktionen ist in allen Komponentensystemen die gleiche (d.h. wenn $t_i^{(l)}$ in DB_l sein Commit vor $t_j^{(l)}$ ausführt, dann muss das auch für $t_i^{(k)}$ und $t_j^{(k)}$ in den übrigen Datenbanken DB_k gelten).

- Die Commitoperationen globaler Transaktionen sind total geordnet. ■

Um diese Anforderungen zu erfüllen, muss ein realer Algorithmus nicht wirklich einen zentralen Koordinator benutzen, es können auch andere Verfahren eingesetzt werden. Das verbreitete Zweiphasen-Commitprotokoll, das wir in Abschnitt 3.3.1 vorgestellt haben, verwendet üblicherweise einen zentralen Koordinator, um diese Eigenschaften sicherzustellen.

In diesem Fall führen globale Transaktionen alle Commitoperationen ihrer lokalen Transaktionen zum gleichen Zeitpunkt aus, den wir im globalen Schedule durch die Commitoperation der globalen Transaktion markieren. Damit ist die relative Ordnung des Commits einer globalen Transaktion und der Operationen von anderen globalen Transaktionen im globalen Schedule die gleiche wie in den lokalen Subschedules. Wenn also im globalen Schedule die Commitoperation c_i einer Transaktion t_i vor der Schreiboperation $w_j(x)$ einer anderen globalen Transaktion t_j ausgeführt wird, dann gilt dies auch in dem Komponentensystem, in dem $w_j(x)$ schließlich ausgeführt wird. In Systemen, die striktes Zweiphasensperren verwenden, ist dies automatisch gewährleistet. Unter diesen Voraussetzungen können wir das folgende Theorem zeigen:

Theorem 3.7: (Gewährleistung globaler Isolation Levels)

Ein föderiertes Datenbanksystem, das ein atomares Commitprotokoll mit einem konzeptuell zentralisierten Koordinator verwendet, gewährleistet für globale Transaktionen den Isolation Level

- Read Uncommitted, wenn alle lokalen Systeme Read Uncommitted gewährleisten,
- Read Committed, wenn alle lokalen Systeme Read Committed gewährleisten,
- Repeatable Read, wenn alle lokalen Systeme Repeatable Read gewährleisten.

Wenn alle lokalen Systeme Phantome vermeiden, können auch im globalen Schedule keine Phantome auftreten. ■

Beweis:

Wenn ein Komponentensystem Dirty Writes vermeidet, muss zwischen zwei Schreiboperationen verschiedener Transaktionen auf dem gleichen Objekt immer die erste der Transaktionen mit Commit beendet worden sein. Wir haben also im lokalen Schedule folgende Operationen:

$$... \; w_1(x) \; ... \; c_1 \; ... \; w_2(x)$$

Wegen der oben beschriebenen Ordnungserhaltung haben wir diese Reihenfolge aber auch im globalen Schedule, d.h. auch im globalen Schedule können keine Dirty Writes vorkommen; für den globalen Schedule wird also Read Uncommitted gewährleistet. Analog können wir argumentieren, dass im globalen Schedule Dirty Reads bzw. Unrepeatable Reads vermieden werden, wenn die Komponentensysteme Dirty Reads bzw.

Unrepeatable Reads vermeiden, und daher für den globalen Read Committed bzw. Repeatable Read garantiert wird.

Wir zeigen nun, dass auch im globalen Schedule keine Phantome auftreten können, wenn alle beteiligten Systeme Phantome vermeiden. Das Auftreten eines Phantoms bedeutet nach der verallgemeinerten Definition aus Abschnitt 2.3.2.3, dass eine föderierte Transaktion ein Prädikat auswertet, also die Objekte liest, die das Prädikat erfüllen, und eine zweite (föderierte oder lokale) Transaktion ein Objekt ändert, einfügt oder löscht, das dieses Prädikat erfüllt, bevor die erste Transaktion beendet ist. Wegen der Eigenschaft der lokalen Systeme muss sich das Prädikat dabei auf mehr als eine Datenbank beziehen, sonst folgt bereits aus der lokalen Vermeidung von Phantomen, dass auch die föderierte Anfrage keine Phantome sehen kann.

Wir nehmen also an, dass eine Transaktion eine verteilte Anfrage ausführt und dabei auf zwei Tabellen S und T in unterschiedlichen Datenbanken DB_1 und DB_2 zugreift. Eine solche Anfrage sieht konzeptuell so aus:

```
select * from DB1, DB2 where P1(DB1) and P1(DB2) and P1(global)
                          or P2(DB1) and P2(DB2) and P2(global)
                          or ...
```

$P_i(DB_1)$ und $P_i(DB_2)$ sind dabei Prädikate, die die Tupel beschreiben, die aus DB_1 bzw. DB_2 zum Ergebnis der Anfrage beitragen, und die P_i(global) sind Prädikate, die Eigenschaften von Tupeln aus beiden Datenbanken verknüpfen, z.B. mit föderierten Joins. In den globalen Prädikaten können prinzipiell auch Relationen vorkommen, die im entsprechenden lokalen Prädikat nicht eingeschränkt werden, z.B. in Form einer Subquery

```
objekte.farbe in (select farbe from moegliche_farben where typ=objekte.typ)
```

mit der Relation `objekte` in DB_1 und der Relation `moegliche_farben` in DB_2. Wir beschränken uns zunächst auf globale Prädikate ohne Subqueries; durch Induktion über die Schachtelungstiefe kann man dann zeigen, dass auch für Anfragen mit Subqueries Phantome vermieden werden. Alle Prädikate können leer sein, also keine Einschränkung der Tupel angeben. Jedes der mit `OR` verknüpften Monome führt zu einer eigenen Ergebnismenge; wir nehmen an, dass jedes Monom in einer eigenen Teilanfrage A_i separat ausgewertet wird.

Zur Ausführung einer solchen Teilanfrage A_i sind mehrere Alternativen denkbar. In jedem Fall müssen aus DB_1 und DB_2 entweder alle Tupel der betroffenen Relationen gelesen werden oder nur die, die eine in $P_i(DB_1)$ bzw. $P_i(DB_2)$ vorhandene Einschränkung erfüllen, falls es eine solche gibt. Die lokalen Systeme sehen also entweder eine Anfrage nach allen Tupeln einer Relation (also mit dem Prädikat `true` in der Suchanfrage), oder eine Anfrage nach Tupeln, die ein gegebenes Prädikat erfüllen. Weil sie lokal Phantome vermeiden, können andere Transaktionen keine Objekte in diesen Relationen einfügen, ändern oder löschen, die diese Prädikate erfüllen.

Zusätzlich müssen dann noch die Prädikate ausgewertet werden, die sich auf Tupel aus beiden Komponentendatenbanken beziehen. Die Implementierung kann dazu entweder alle Tupel aus den betroffenen Relationen auf die föderierte Ebene laden und dort prüfen, welche die Prädikate erfüllen; in diesem Fall gilt die gleiche Argumentation wie für rein lokale Prädikate. Die andere Möglichkeit ist, nur die Tupel aus einer Komponentendatenbank auf die föderierte Ebene zu laden und danach die Partnertupel aus der an-

deren Komponentendatenbank durch eine eigene Anfrage zu besorgen. Ein Prädikat der Form

```
A.attribut=B.attribut
```

mit möglichen Werten 1 und 2 für `B.attribut` kann z.B. ausgewertet werden, indem die beiden Anfragen

```
select * from A where A.attribut = 1
select * from A where A.attribut = 2
```

ausgeführt und die Ergebnisse entsprechend behandelt werden. Im Komponentensystem, wo Relation A liegt, können keine Objekte modifiziert werden, die eins der beiden angegebenen Prädikate erfüllen. Weil zur Bestimmung der möglichen Werte von `B.attribut` aber ebenfalls ein Prädikat benutzt wird, können auch im Komponentensystem, wo B liegt, keine neuen Werte von `B.attribut` hinzukommen.

Insgesamt haben wir also gezeigt, dass nach dem Ausführen einer föderierten Anfrage in keinem Komponentensystem Objekte hinzugefügt, geändert oder gelöscht werden können, die das Prädikat der föderierten Anfrage erfüllen. Das heißt aber, dass global Phantome vermieden werden. ■

Vermeiden alle lokalen Systeme Phantome, folgt allerdings nicht automatisch globale Serialisierbarkeit, wie wir in Abschnitt 3.4.1 gesehen haben: Alle Systeme, die serialisierbare Schedules erzeugen, vermeiden Phantome, trotzdem sind zusätzliche Eigenschaften oder Maßnahmen notwendig, um globale Serialisierbarkeit sicherzustellen.

Wenn alle lokalen Systeme Snapshot Isolation garantieren, folgt nicht automatisch globale Snapshot Isolation, obwohl man das nach der bisherigen Diskussion in diesem Abschnitt erwarten könnte. Wir diskutieren dieses Problem im folgenden Abschnitt ausführlich und stellen mögliche Abhilfen vor.

3.8 Gewährleistung globaler Snapshot Isolation

3.8.1 Problemstellung

In Abschnitt 3.6 haben wir gesehen, dass auf der föderierten Ebene in der Regel ein recht hoher Aufwand nötig ist, um globale Serialisierbarkeit zu gewährleisten, wenn einige oder alle lokalen Systeme nur Snapshot Isolation garantieren. Der Grund dafür ist, dass die "lokal fehlende Korrektheit" durch zusätzliche Maßnahmen auf der globalen Ebene hergestellt werden muss. Ähnlich wie im zentralen Fall kann man nun das globale Korrektheitskriterium etwas lockern und auch global nur Snapshot Isolation fordern. Weil jetzt die lokalen und das globale Kriterium identisch sind, kann man erwarten, dass insgesamt weniger Aufwand notwendig ist, um das globale Kriterium Snapshot Isolation sicherzustellen.

Wir beschränken uns zunächst auf föderierte Systeme, in denen alle Komponentensysteme selbst Snapshot Isolation gewährleisten. In Abschnitt 3.8.5 werden wir später sehen, dass es nicht einfach ist, solche Datenbanksysteme zu integrieren, die serialisierbare Schedules erzeugen. Das folgende Beispiel eines Schedules in einem föderierten System mit zwei Komponenten zeigt, dass selbst in einem solchen homogenen System zusätzlicher Aufwand notwendig ist, um globale Snapshot Isolation zu garantieren:

DB_1: $r_1^{(1)}(a_0)\ r_2^{(1)}(x_0)\ w_2^{(1)}(x_2)$ $C_2\ r_1^{(1)}(x_0)$ C_1

DB_2: $r_2^{(2)}(y_0)\ w_2^{(2)}(y_2)\ C_2$ $r_1^{(2)}(y_2)\ w_1^{(2)}(y_1)\ C_1$

In Datenbank DB_1 laufen die Subtransaktionen $t_1^{(1)}$ und $t_2^{(1)}$ der beiden globalen Transaktionen t_1 und t_2 parallel, also lesen sie beide von einer vorherigen Transaktion (hier ist das die initiale Transaktion t_0). Nur $t_2^{(1)}$ ändert ein Objekt, der Subschedule ist also Element von SI. Der Subschedule in der anderen Datenbank ist auch Element von SI: Die beiden Subtransaktionen $t_1^{(2)}$ und $t_2^{(2)}$ laufen nacheinander ab und lesen die richtigen Werte, die die SI-Versionsfunktion vorsieht. Kombiniert man aber beide Subschedules zu einem globalen Schedule, erhält man (ohne die Nummern, die die Datenbank angeben):

$$r_1(a_0)\ r_2(x_0)\ w_2(x_2)\ r_2(y_0)\ w_2(y_2)\ C_2\ r_1(x_0)\ r_1(y_2)\ w_1(y_1)\ C_1$$

Dieser Schedule ist nicht Element von SI: Transaktion t_1 liest y von t_2 statt von t_0, also gilt (SI-V) nicht, und beide Transaktionen laufen parallel, schreiben aber beide y, also gilt SI-W ebenfalls nicht. Die Ursache dieses Problems liegt darin, dass der lokale Scheduler in DB_2 nicht weiß, dass t_2 auf der globalen Ebene viel früher als lokal in DB_2 angefangen hat, und deshalb t_2 Versionen zuweist, die zwar lokal die richtigen sind, aber die globale Korrektheit beeinflussen. Gleiches gilt für globale vs. lokale Writesets und globale vs. lokale Parallelität von Transaktionen. Um global Snapshot Isolation zu gewährleisten, genügen also rein lokale Maßnahmen nicht, statt dessen sind zusätzlich Algorithmen auf der föderierten Ebene notwendig.

Für die Diskussion im ganzen Abschnitt 3.8 nehmen wir an, dass alle föderierten Transaktionen ein atomares Commitprotokoll (ACP) verwenden, das auf einem konzeptionell tentralen Koordinator basiert, um ihr über die beteiligten Komponentensysteme verteiltes Commit zu synchronisieren (siehe Definition 3.3).

Basierend auf dieser Voraussetzung stellen wir im folgenden zwei Algorithmen vor, mit denen man globale Snapshot Isolation gewährleisten kann, sofern die Komponentensysteme selbst Snapshot Isolation für die lokalen Subschedules sicherstellen. Der erste Algorithmus, den wir im nächsten Abschnitt darstellen, synchronisiert dazu den Beginn aller Subtransaktionen einer globalen Transaktion. In Abschnitt 3.8.3 diskutieren wir einen optimistischen Algorithmus, der Verletzungen von (SI-V) im globalen Schedule erkennt. Anschließend vergleichen wir die vorgestellten Algorithmen qualitativ. Den Abschluss dieses Abschnitts bildet eine kurze Diskussion über die Möglichkeiten, Subsysteme einzubinden, die serialisierbare Schedules erzeugen.

3.8.2 Synchronisation der Subtransaktionen

Wir haben anhand des Beispiels im letzten Abschnitt gesehen, dass ein Grund für die auftretenden Probleme darin liegt, dass die lokalen Subtransaktionen teilweise später als die globale Transaktion beginnen, so dass die lokalen Scheduler aus globaler Sicht unzulässige Schedules erzeugen. Diese Schwierigkeiten würden nicht auftreten, wenn alle Subtransaktionen einer globalen Transaktion zur gleichen Zeit wie die globale Transaktion im globalen Schedule beginnen würden. Wenn wir das erreichen, kann jede Verletzung der Snapshot-Isolation-Eigenschaft des globalen Schedules auf eine korrespondierende Verletzung in einer der lokalen Schedules abgebildet werden. Weil die lokalen Scheduler aber die SI-Eigenschaft garantieren, kann es keine solchen lokalen Verletzungen geben, so dass auch der globale Schedule Element von SI sein muss.

Der naheliegende Weg, die lokalen Subtransaktionen simultan zu starten, ist es, in allen lokalen Datenbanksystemen, in denen die Transaktion Operationen ausführen wird, eine explizite, lokale Subtransaktion zu beginnen. Leider genügt dies alleine nicht, um das Problem zu lösen: Eine globale Transaktion t_1 könnte ihre atomare Commitoperation ausführen, während eine andere globale Transaktion t_2 gerade ihre lokalen Subtransaktionen startet. Wenn mindestens eine dieser Subtransaktionen vor und eine andere nach dem globalen Commit von t_1 ausgeführt werden, kann global keine Snapshot Isolation mehr garantiert werden. Der folgende Schedule zeigt dieses Problem exemplarisch:

DB_1: $r_1^{(1)}(x_0)\ w_1^{(1)}(x_1)\ B_2^{(1)}\ C_1 \qquad r_2^{(1)}(x_0) \qquad\qquad C_2$
DB_2: $r_1^{(2)}(y_0)\ w_1^{(2)}(y_1) \qquad C_1\ B_2^{(2)} \qquad\qquad r_2^{(2)}(y_1)\ w_2^{(2)}(y_2)\ C_2$

Der globale Schedule ist nicht Element von SI, obwohl beide lokalen Schedules Element von SI sind. Es genügt also nicht, lediglich die Subtransaktionen frühzeitig zu starten.

Das Problem tritt deshalb auf, weil eine globale Commitoperation zwischen dem Starten der Subtransaktionen einer globalen Transaktion ausgeführt wurde. Um es zu vermeiden, müssen wir also die Commitoperation so lange verzögern, bis alle lokalen Subtransaktionen begonnen wurden. Analog müssen wir das Starten der lokalen Subtransaktionen aufschieben, bis eine gerade laufende Commitoperation beendet ist.

Wenn wir diese Synchronisation von Commitoperationen und Starts von globalen Transaktionen sowie ein atomares Commitprotokoll verwenden, können wir also annehmen, dass alle Subtransaktionen einer globalen Transaktion t_i zum gleichen Zeitpunkt beginnen, den wir durch die Pseudooperation B_i kennzeichnen, und zur gleichen Zeit ihr Commit ausführen, so dass wir in allen lokalen Datenbanken die gleiche Commitoperation C_i verwenden können. Daher sind auch die Versionsfunktionen in den Komponentensystemen synchronisiert. Wir können also das folgende Theorem ableiten:

Theorem 3.8:

Sei s ein Schedule in einem föderierten Datenbanksystem, dessen Komponentensysteme für lokale Schedules Snapshot Isolation garantieren, und das ein ACP verwendet, das auf einem konzeptuell zentralen Koordinator basiert. Wenn jede föderierte Transaktion an ihrem Anfang in jeder lokalen Datenbank, auf die sie möglicherweise zugreifen wird, eine lokale Subtransaktion beginnt, und die föderierte Ebene verhindert, dass globale Commitoperationen parallel dazu ausgeführt werden, dann ist s Element von SI. ■

Beweis von Theorem 3.8:

Wir zeigen, dass der globale Schedule Element von SI ist, indem wir nachweisen, dass die beiden Eigenschaften (SI-V) und (SI-W) gelten, die Schedules in SI auszeichnen.

(SI-V): Nehmen wir an, dass es im globalen Schedule eine Transaktion t_i gibt, die in Datenbank DB_l eine Version des Objektes x liest, die eine andere Transaktion t_j geschrieben hat, aber dass t_j die falsche Transaktion im Sinne der globalen Versionsfunktion ist. Dann wurde t_j entweder noch nicht mit Commit beendet, wurde nach dem Anfang von t_i mit Commit beendet, oder eine andere Transak-

tion t_k hat x verändert und zwischen dem Commit von t_j und dem Anfang von t_i ihr Commit ausgeführt. Da aber Transaktionsstarts und Commitoperationen in allen lokalen Datenbanksystemen synchron ablaufen, muss jeder dieser Fälle auch in DB_l gelten, wenn er global eintritt. Auch der lokale Schedule in DB_l verletzt also (SI-V), das ist aber ein Widerspruch gegen die Annahme, dass alle lokalen Schedules Element von SI sind.

(SI-W): Nehmen wir nun an, dass es im globalen Schedule zwei Transaktionen t_i und t_j gibt, die parallel zueinander ausgeführt werden und in Datenbank DB_l Objekt x schreiben. Weil die Transaktionsstarts und Commitoperationen in allen lokalen Datenbanksystemen synchron ablaufen, müssen auch $t_i^{(l)}$ und $t_j^{(l)}$ parallel zueinander ablaufen. Aber dann ist (SI-W) auch in DB_l verletzt, so dass auch der lokale Schedule nicht Element von SI sein kann, was wieder ein Widerspruch zur Annahme ist. ■

Dieser Ansatz kann recht einfach implementiert werden, hat aber potentiell Performancenachteile: Es ist am Anfang einer globalen Transaktion nicht unbedingt bekannt, auf welche Komponentensysteme sie im Laufe ihrer Ausführung zugreifen wird. Man kann dies auch nicht aus dem zugrundeliegenden Transaktionsprogramm ableiten, da z.B. anhand von Benutzereingaben entschieden wird, auf welche Daten zugegriffen wird. Um auf der sicheren Seite zu sein, muss die globale Transaktion also in einer konservativ ausgewählten Teilmenge der Datenbanksysteme in der Föderation eine Subtransaktion beginnen, im schlimmsten Fall in allen. Obwohl jeder einzelne Start einer Subtransaktion recht wenig Ressourcen benötigt, kann der Gesamtoverhead, der durch die hohe Zahl dieser Operationen entsteht, spürbaren Einfluss auf den möglichen Transaktionsdurchsatz haben. Darüber hinaus müssen Commitoperationen verzögert werden, falls gerade Subtransaktionen gestartet werden, was eine ernste Beeinträchtigung der möglichen Antwortzeiten für Transaktionen bedeuten kann.

Würde man diesen Ansatz dagegen statt des in Abschnitt 3.5.1 beschriebenen in die Oracle-Software zur Steuerung verteilter Transaktionen einbauen, wäre der notwendige Overhead wesentlich kleiner. Man müsste nicht bereits am Anfang der verteilten Transaktion alle notwendigen Subtransaktionen beginnen, sondern könnte den gleichen Effekt erreichen, indem man jeder Subtransaktion systemintern die Zeitmarke der verteilten Transaktion selbst zuweist. Sind die Zeitmarken aller beteiligten Systeme synchronisiert, garantiert dies globale Snapshot Isolation ohne wesentlichen Zusatzaufwand. Dies kann von Software außerhalb der Datenbank nicht erreicht werden, da es unmöglich ist, die Zeitmarke einer Transaktion von außen zu setzen. Allerdings würde diese Lösung nur bei verteilten Oracle-Datenbanken greifen, nicht bei beliebigen Föderationen.

3.8.3 Ein optimistischer SI-Test

Wenn wir uns nochmals das Beispiel aus Abschnitt 3.8.1 betrachten, stellen wir fest, dass dort eine besondere Situation vorliegt: Die beiden globalen Transaktionen laufen parallel zueinander ab, aber die Subtransaktion $t_2^{(2)}$ von t_2 in DB_2 liest von der anderen Subtransaktion $t_1^{(2)}$. Dadurch wird die SI-Eigenschaft des globalen Schedules verletzt, obwohl alle lokalen Schedules Element von SI sind:

- Subtransaktion $t_2^{(2)}$ liest von $t_1^{(2)}$ in Datenbank DB_2, obwohl die globalen Transaktionen parallel ablaufen, also gilt für den globalen Schedule (SI-V) nicht.

- Beide Subtransaktionen $t_1^{(2)}$ und $t_2^{(2)}$ ändern das gleiche Objekt x in Datenbank DB_2, so dass (SI-W) für den globalen Schedule nicht gilt. Weil wir verlangen, dass eine Transaktion ein Objekt zuerst liest, bevor sie es schreiben darf, muss auch (SI-V) für den globalen Schedule verletzt sein. Es genügt daher, sicherzustellen, dass (SI-V) für den globalen Schedule gilt.

Diese Situation ist in der Tat die einzige Möglichkeit dafür, dass der globale Schedule kein Element von SI sein kann, obwohl alle lokalen Schedules Snapshot Isolation gewährleisten. Wir müssen also vermeiden, dass zwei globale Transaktionen parallel ablaufen, während zwischen ihren Subtransaktionen in mindestens einer Komponentendatenbank eine Liest-von-Beziehung besteht. Das folgende Theorem zeigt, dass dies ausreicht, um (SI-V) für den globalen Schedule sicherzustellen und damit globale Snapshot Isolation zu gewährleisten:

Theorem 3.9: (Charakterisierung der globalen SI-Versionsfunktion)

Sei s ein Schedule in einem föderierten Datenbanksystem, dessen Komponentensysteme für lokale Schedules Snapshot Isolation garantieren, und das ein ACP verwendet, das auf einem konzeptuell zentralen Koordinator basiert. Die Versionsfunktion von s, d.h. die Vereinigung der Versionsfunktionen der lokalen Subschedules von s, erfüllt genau dann (SI-V), wenn es keine parallel laufenden globalen Transaktionen t_i und t_j gibt, so dass in einer Datenbank DB_k $t_j^{(k)}$ (transitiv) von $t_i^{(k)}$ liest. ■

Beweis:

<u>"Es gibt keine ..." ⇒ "(SI-V) gilt":</u>

Wir nehmen zunächst an, (SI-V) gelte nicht. Dann gibt es eine Transaktion t_j, die eine Version eines Objektes x liest, die entweder "zu neu" oder "zu alt" ist. Das Objekt x liegt dabei in einer Datenbank DB_l. Alle Lese- und Schreiboperationen, die auf x zugreifen, sind daher Teil von Subtransaktionen in dieser Datenbank.

Wenn die gelesene Version zu alt ist, liest t_i x nicht von der letzten Transaktion t_j, die x geschrieben hat und mit Commit beendet wurde, bevor t_i gestartet wurde. Also muss es eine andere Transaktion t_k geben, die dazwischen mit Commit beendet wurde: $C_j < C_k < B_k$, und sowohl t_j als auch t_k ändern x. Die Subtransaktion $t_i^{(l)}$ von t_i in DB_l kann nicht vor dem Anfang von t_i selbst beginnen, also gilt $B_i < B_i^{(l)}$. Weil wir ein ACP annehmen, ergibt sich damit $C_i^{(l)} < C_k^{(l)} < B_i^{(l)}$. Das heißt, dass $t_i^{(l)}$ x nicht von der letzten Transaktion liest, die vor dem Start von $t_i^{(l)}$ im lokalen Datenbanksystem mit Commit beendet wurde. Dies ist aber ein Widerspruch, weil DB_l für alle lokalen Transaktionen Snapshot Isolation garantiert.

Wenn die gelesene Version zu neu ist, liest t_i x von einer Transaktion t_j, die noch nicht mit Commit beendet war, als t_i gestartet wurde. Die Transaktionen t_i und t_j laufen also global parallel ab. In DB_l liest aber $t_i^{(l)}$ x von $t_j^{(l)}$, dies ist ein Widerspruch zur Annahme.

"(SI-V) gilt" ⇒ "Es gibt keine ...":

Wir nehmen an, dass zwei globale Transaktionen t_i und t_j parallel zueinander ablaufen, aber dass in einer lokalen Datenbank DB_l $t_i^{(l)}$ x von $t_j^{(l)}$ liest. Dann ist es aber klar, dass t_i von einer parallel laufenden Transaktion liest, so dass (SI-V) nicht mehr erfüllt ist, wir haben also sofort einen Widerspruch gezeigt. ■

Aus dieser notwendigen und hinreichenden Bedingung, die ein Schedule erfüllen muss, um (SI-V) zu erfüllen, können wir das folgende Theorem ableiten, dass ein Kriterium für die Zugehörigkeit eines globalen Schedules zu SI angibt.

Theorem 3.10: **(notwendige und hinreichende Bedingung für globale Snapshot Isolation)**

Sei s ein Schedule in einem föderierten Datenbanksystem, dessen Komponentensysteme für lokale Schedules Snapshot Isolation garantieren, und das ein ACP verwendet, das auf einem konzeptuell zentralen Koordinator basiert. Dann ist s genau dann Element von SI, wenn es keine parallel laufenden globalen Transaktionen t_i und t_j gibt, so dass in einer Datenbank DB_k $t_j^{(k)}$ (transitiv) von $t_i^{(k)}$ liest. ■

Beweis:

Nach Theorem 3.9 wissen wir bereits, dass s die Bedingung (SI-V) erfüllt. Um zu zeigen, dass s auch die Bedingung (SI-W) erfüllt, nutzen wir die Charakterisierung durch den SI-MVSG, die wir in Abschnitt 2.4.4 vorgestellt haben. Der SI-MVSG eines globalen Schedules ist die Vereinigung der SI-MVSGs der lokalen Subschedules. Wir können nun aus der SI-Eigenschaft der lokalen Subschedules sofort folgern, dass auch der globale Schedule Element von SI sein muss: Wenn alle lokalen Subschedules Element von SI sind, gibt es kein Objekt x, so dass es in irgendeinem lokalen SI-MVSG einen x-Zyklus gibt. Weil aber ein Objekt immer in genau einer Datenbank abgelegt wird, gibt es auch im globalen Graphen keinen x-Zyklus, also ist s nach Theorem 2.4 Element von SI. ■

Dieses Theorem bildet die Basis für einen Algorithmus, mit dem man für globale Schedules Snapshot Isolation gewährleisten kann, unter der Voraussetzung, dass alle lokalen Systeme selbst Snapshot Isolation garantieren. Die Idee dabei ist, die Einhaltung der Bedingungen des Theorems zu beobachten; sollten zwei Transaktionen sie verletzen, muss man auf der föderierten Ebene entsprechende Maßnahmen treffen, z.B. eine der beiden Transaktionen zwangsweise zurücksetzen. Es ist dabei leicht zu prüfen, ob zwei globale Transaktionen parallel zueinander ablaufen. Analog ist es möglich, festzustellen, ob zwei Subtransaktionen in einer Komponentendatenbank nacheinander ablaufen; das ist notwendig, damit die spätere überhaupt von der früheren lesen kann.

Wesentlich schwieriger ist es, die Liest-von-Beziehung zu überwachen, da die Komponentensysteme normalerweise keine Informationen darüber geben, welche Version eines Objektes von einer Leseoperation gelesen wurde. Unser Algorithmus bildet daher auf der Basis der Operationen, die globale Subtransaktionen ausführen, eine konservative Näherung (also eine Obermenge) der Liest-von-Beziehung. Der Einfachheit halber betrachtet wir zunächst keine rein lokalen Transaktionen, sondern beschränken die lokalen

Schedules auf Subtransaktionen globaler Transaktionen. Wir können die Aktionen, die diese Subtransaktionen ausführen, auf der föderierten Ebene beobachten und daraus Prädikate ableiten, die die Objekte charakterisieren, die von dieser Aktion betroffen sind. Leseoperationen werden zum Beispiel in der Regel von SQL-Anweisungen angestoßen, so dass wir das Suchprädikat einer solchen Anweisung zur Charakterisierung der Objekte verwenden können, die von dieser Anweisung zurückgegeben werden. Wir haben die Details dieser Vorgehensweise bereits in Abschnitt 3.6.3 vorgestellt.

Mit dieser Technik können wir Approximationen der Read- und Writesets von Transaktionen bestimmen, während sie ausgeführt wird. Eine Subtransaktion $t_i^{(k)}$ liest dann potentiell (transitiv) von einer anderen Subtransaktion $t_j^{(k)}$, wenn $t_j^{(k)} < t_i^{(k)}$ und der Schnitt von Writeset($t_j^{(k)}$) und Readset($t_i^{(k)}$) nicht leer ist.

Was geschieht nun, wenn wir auch lokale Transaktionen in den Komponentensystemen erlauben? Wir betrachten dazu eine lokale Transaktion t_L, die vollständig zwischen $t_j^{(k)}$ und $t_i^{(k)}$ ausgeführt wird, d.h. $C_j^{(k)} < B_L < C_L < B_i^{(k)}$. Subtransaktion $t_i^{(k)}$ liest dann nicht mehr von $t_j^{(k)}$, sondern von t_L. Für jedes betroffene Objekt x gibt es dann die folgenden Operationen im lokalen Schedule:

$$... r_i(x) ... w_i(x_i) ... c_i ... r_L(x_i) ... w_L(x_L) ... c_L ... r_j(x_L) ...$$

Man sieht, dass $t_j^{(k)}$ immer noch transitiv von $t_i^{(k)}$ liest, so dass der Algorithmus immer noch korrekt arbeitet. Analog kann man argumentieren, wenn eine Sequenz lokaler Transaktionen zwischen $t_i^{(k)}$ und $t_j^{(k)}$ ausgeführt wird. Unser Algorithmus kann also mit lokalen Transaktionen umgehen, ohne dass er ihre Aktionen beobachten muss, was praktisch sehr schwierig, wenn nicht unmöglich, in jedem Fall aber eine Verletzung der Autonomie des lokalen Systems wäre.

Weil der Algorithmus nur Informationen der globalen Ebene verwenden kann, sieht er nicht, wenn Subtransaktionen in den Komponentensystemen automatische Aktionen im Kontext der Subtransaktion auslösen, z.B. Änderungen, die innerhalb von feuernden Triggern ausgeführt werden. Wenn man solche Effekte in das Protokoll aufnehmen will, muss das föderierte System Informationen über alle Trigger und vergleichbare aktive Elemente in den Komponentendatenbanken haben, so dass es sie beim Aufbau der Read- und Writesets berücksichtigen kann.

Hat man auf der föderierten Ebene keine Informationen über die aktiven Elemente in mindestens einer lokalen Datenbank, muss der Algorithmus angepasst werden. Wir können nun nicht mehr ableiten, ob eine Subtransaktion $t_i^{(k)}$ einer globalen Transaktion in einer Komponentendatenbank DB_k von einer anderen Subtransaktion $t_j^{(k)}$ liest, da wir die Read- und Writesets nicht vollständig kennen. Eine notwendige Bedingung, damit $t_i^{(k)}$ von $t_j^{(k)}$ lesen kann, ist, dass $t_j^{(k)}$ vollständig vor $t_i^{(k)}$ abgelaufen ist, dass also $t_j^{(k)} < t_i^{(k)}$ gilt. Ist diese Bedingung nicht erfüllt, kann (SI-V) nicht wegen dieses Paars von Subtransaktionen verletzt werden, weil $t_i^{(k)}$ nicht von $t_j^{(k)}$ lesen kann. Wir können also das folgende Korollar zu Theorem 3.10 aufstellen, das eine hinreichende Bedingung für globale Snapshot Isolation formuliert:

Korollar 3.1: **(hinreichende Bedingung für globale Snapshot Isolation)**

Sei s ein Schedule in einem föderierten Datenbanksystem, dessen Komponentensysteme für lokale Schedules Snapshot Isolation garantieren und das ein atomares Commitprotokoll verwendet, das auf einem konzeptuell zentralen Koordinator basiert. Dann ist s Element von SI, wenn es keine parallel laufenden globalen Transaktionen t_i und t_j gibt, bei denen in mindestens einer Datenbank DB_k $t_j^{(k)} < t_i^{(k)}$ gilt. ■

Der oben vorgestellte Algorithmus bleibt also korrekt, wenn wir den Aufwand für die Ableitung der Read- und Writesets sowie den Test auf Überdeckung einsparen. Die Bedingung an die erlaubten Schedules ist aber nun restriktiver als in Theorem 3.10; wir verbieten also mehr Schedules, auch solche, die globale Snapshot Isolation gewährleisten. Das sind gerade die, bei denen zwei parallele globale Transaktionen Subtransaktionen in einem Komponentensystem haben, die vollständig nacheinander ablaufen, aber nicht voneinander lesen. Diese Transaktionen werden also unnötigerweise abgebrochen, so dass die erzielbare Performance potentiell schlechter ist als die des aufwändigeren Algorithmus. In Situationen, bei denen nicht alle Operationen einer Subtransaktion anhand der auf der föderierten Ebene abgesetzten Aktionen abgeleitet werden können, weil z.B. in der Datenbank auf der föderierten Ebene unbekannte aktive Elemente existieren, ist dieser Algorithmus aber eine mögliche Lösung, um überhaupt globale Snapshot Isolation garantieren zu können.

3.8.4 Qualitativer Vergleich der Verfahren

Die Synchronisation der Subtransaktionen aus Abschnitt 3.8.2 ist vom Implementierungsaufwand her die Technik mit dem geringsten Aufwand. Trotzdem hat sie mögliche Performanceprobleme, weil sie Verzögerungen von Transaktionsstarts und -commits verursachen kann und so die Antwortzeit von Transaktionen negativ beeinflussen kann. Weil zudem die Menge der Komponentensysteme, in denen Subtransaktionen begonnen werden, konservativ gewählt werden muss, fällt hier pro Transaktionsstart ein nicht unerheblicher Zusatzaufwand an. Diese Technik bietet sich daher vor allem dann an, wenn nur wenige Komponentensysteme an der Föderation teilnehmen oder die Datenbanken, auf die eine Transaktion zugreifen wird, von vornherein bekannt sind. Ein weiteres mögliches Einsatzgebiet wäre natürlich in Oracles Datenbanksoftware zur Kontrolle verteilter Transaktionen, da dann die Synchronisation der Zeitmarken der einzelnen Komponentensysteme ausreichen würde, um Snapshot Isolation zu gewährleisten.

Die optimistische Technik, die in Abschnitt 3.8.3 vorgestellt wurde, kann zu unnötigen Transaktionsabbrüchen führen, weil sie nicht die wirkliche Liest-von-Beziehung, sondern nur eine auf Prädikaten basierende Näherung berücksichtigen kann. Dies ist aber inhärent darin begründet, dass die föderierte Ebene die Details der Verarbeitung in den Komponentensystemen nicht beobachten kann. Trotz dieses Nachteils kann diese Technik eine bessere Performance erzielen als die Synchronisation der Subtransaktionen, weil sie nur für die Systeme zusätzlichen Overhead bringt, die wirklich an der Ausführung teilhaben, und keine Verzögerung von Transaktionsstarts und -commits benötigt. Die restriktive Variante, die bereits eine Verletzung der Snapshot-Isolation-Eigenschaft

vermutet, wenn zwei global nebenläufige Transaktionen sequentiell ablaufende Subtransaktionen haben, ist zwar wesentlich weniger aufwändig als die Auswertung der Prädikate. Sie ruft aber in der Regel wesentlich mehr unnötige Abbrüche hervor, so dass der aufwändigeren Technik der Vorzug gegeben werden sollte, wann immer dies möglich ist.

3.8.5 Integration von SR-Subsystemen

In den vorherigen Abschnitten haben wir gezeigt, wie man globale Snapshot Isolation garantieren kann, wenn alle Komponentensysteme für die Subtransaktionen globaler Transaktionen Snapshot Isolation sicherstellen. In der Praxis ist man auch daran interessiert, globale Snapshot Isolation zu gewährleisten, wenn einige Komponentensysteme ein anderes Korrektheitskriterium verwenden; besonders interessant ist dabei der Fall, dass einige Datenbanken Snapshot Isolation unterstützen, während andere serialisierbare Schedules erzeugen. Wir zeigen jetzt kurz, dass sich diese Kombination nicht effizient unterstützen lässt.

Wir nehmen dazu an, dass ein Komponentensystem Schedules erzeugt, die kaskadierende Abbrüche vermeiden, also Schedules aus der Klasse ACA. Wenn wir sicherstellen wollen, dass lokale Schedules zur Klasse SI gehören, müssen wir u.a. dafür sorgen, dass Transaktionen die "richtigen" Daten lesen. Die erste Operation einer Transaktion t_i bestimmt dabei, welche Daten sie später lesen darf: Wir müssen vermeiden, dass t_i Daten sieht, die von einer anderen Transaktion geschrieben wurden, die nach der ersten Operation von t_i mit Commit beendet wurde. Betrachten wir zum Beispiel den folgenden Schedule:

$$r_1(x)\ r_2(y)\ w_2(y)\ c_2\ r_1(y)\ w_1(y)\ c_1$$

Transaktion t_1 sieht hier die Änderung von y, die t_2 gemacht hat, obwohl t_2 erst nach dem Anfang von t_1 mit Commit beendet wurde. Der Schedule erfüllt also, wenn man ihn als Mehrversionenschedule ansieht, indem man ihn mit einer Standardversionsfunktion versieht, nicht die Bedingung an die Versionsfunktion, die Snapshot Isolation fordert. Solche Schedules müssen also vermieden werden, wenn wir globale Snapshot Isolation sicherstellen wollen. Im Beispiel müssten wir also t_1 abbrechen, obwohl der Schedule selbst serialisierbar ist. Um solche Fälle zu erkennen, können wir, ähnlich wie in dem am Ende von Abschnitt 3.6.3 beschriebenen Ansatz zur Gewährleistung von Serialisierbarkeit mit lokalen Isolation Levels, Approximationen der Read- und Writesets von Transaktionen verwalten und Transaktionen abbrechen, die auf der Basis dieser Informationen eine falsche Version lesen könnten. Analog dazu können wir feststellen, wenn zwei nebenläufige Transaktionen dasselbe Objekt verändern wollen, und dann eine von ihnen abbrechen. Zusätzlich zu diesen lokalen Maßnahmen muss man sicherstellen, dass die Versionsfunktionen der lokalen Schedules kompatibel sind; Verfahren dazu haben wir in den vorherigen Abschnitten vorgestellt.

Ein Algorithmus wie der vorgestellte schränkt die mögliche Parallelität von Transaktionen stark ein, da er viele Transaktionen abbrechen wird, weil sie "falsche" Versionen im Sinne von Snapshot Isolation lesen. Die lokalen Schedules wären dabei sogar ohne diese Abbrüche serialisierbar. Das Erzwingen eines Isolation Levels bringt hier also keine höhere Parallelität, sondern im Gegensatz dazu sogar Performanceeinbußen. Der Aufwand, der notwendig wäre, um lokale Snapshot Isolation zu gewährleisten, steht also in keinem Verhältnis zum erzielbaren Nutzen. Aus diesem Grund ist es nicht sinnvoll, in

gemischten Systemen, wie wir sie in diesem Abschnitt betrachtet haben, globale Snapshot Isolation sicherzustellen; statt dessen sollte man dann besser gleich globale Serialisierbarkeit mit einer der in Abschnitt 3.6 gezeigten Techniken garantieren. Der Laufzeitoverhead für diese Algorithmen, speziell die Verwaltung der Read- und Writesets des OSI-MVSG, ist vergleichbar; im Gegensatz zu dem in diesem Abschnitt vorgestellten Ansatz erlauben sie aber eine höhere Parallelität.

3.9 Zusammenfassung

Wir haben in diesem Kapitel die Probleme dargestellt, die sich stellen, wenn für Transaktionen in föderierten Datenbanksystemen Atomarität und Isolation gewährleistet werden sollen, und etablierte Algorithmen vorgestellt und bewertet, die diese Probleme lösen. Dabei hat sich herausgestellt, dass viele Algorithmen, die die Isolation föderierter Transaktionen sicherstellen, so spezielle Anforderungen an die beteiligten Datenbanken oder so große Performancenachteile haben, dass sie für eine praktische Anwendung nicht in Frage kommen.

Existierende Systeme zur Föderation von Datenbanken unterstützen von diesen Algorithmen in der Regel das Zweiphasen-Commitprotokoll, mit dem sich auch in heterogenen Föderationen Atomarität sicherstellen lässt. Globale Serialisierbarkeit dagegen wird von praktisch allen Systemen vernachlässigt. Statt dessen verlagert man die Verantwortung für diesen Bereich auf den Programmierer der Anwendung, der selbst notwendige Maßnahmen treffen muss. Ein weiterer Grund dafür, dass die Vielzahl der entwickelten Algorithmen zur Gewährleistung globaler Serialisierbarkeit nicht Einfluss in Systeme gefunden haben, liegt neben den schon genannten darin, dass die Algorithmen weit verbreitete Besonderheiten, wie die Verwendung eingeschränkter Isolation Levels, nicht unterstützen, sondern lokale Serialisierbarkeit fordern. Häufig eingesetzte Systeme wie Oracle können daher von diesen Algorithmen nicht eingebunden werden.

Wir haben in diesem Kapitel Algorithmen vorgestellt, die erstmalig auch dann globale Serialisierbarkeit garantieren können, wenn einige oder alle lokalen Systeme nur den wichtigen Isolation Level Snapshot Isolation gewährleisten, indem wir die Tickettechnik erweitert und unsere theoretischen Ergebnisse aus Abschnitt 2.4 auf den föderierten Fall übertragen haben. Außerdem haben wir diskutiert, wie man auch für föderierte Transaktionen eingeschränkte Isolation Levels gewährleisten und damit potentiell auch für diese Transaktionen Performance gewinnen kann. Einen besonderen Schwerpunkt bildet dabei wieder Snapshot Isolation, für deren Gewährleistung auf der föderierten Ebene wir zwei Algorithmen vorgestellt haben.

4 Prototypimplementierung

4.1 Das föderierte Datenbanksystems VHDBS

In diesem Abschnitt beschreiben wir zunächst das föderierte Datenbanksystem VHDBS [Wu96,WW97,HWW98], das als Ausgangspunkt für diese Arbeit genommen wurde, weil föderierte Transaktionen nur im Zusammenhang mit einem föderierten Datenbanksystem sinnvoll betrachtet werden können. Insbesondere der quantitative Vergleich verschiedener Transaktionsstrategien durch Benchmarks ist nur in einer realen Systemumgebung möglich.

Wir stellen in Abschnitt 4.1.1 zunächst die wesentlichen Aspekte der Architektur von VHDBS vor. In Abschnitt 4.1.2 gehen wir auf einige Implementierungsdetails von VHDBS näher ein, die wir im weiteren Verlauf der Arbeit noch benötigen.

4.1.1 Die Architektur von VHDBS

VHDBS[1] [HWW98] ist ein föderiertes Datenbanksystem, das im Rahmen eines Projektes für die Deutschen Telekom AG vom Fraunhofer Institut für Software und Systemtechnik in Dortmund entwickelt wurde [WW96,Weiß97]. Es integriert existierende, weitgehend autonome Datenbanken, die von heterogenen Datenbanksystemen verwaltet werden. Seine Architektur basiert auf dem Wrapper-Mediator-Ansatz, der von Wiederhold vorgeschlagen wurde [Wied92]. Existierende Komponentensysteme werden durch systemspezifische *Datenbankadapter* gekapselt, die Datenmodell und Anfragesprache zwischen der föderierten Ebene und den Komponentensystemen übersetzen. Auf der föderierten Ebene nimmt ein Mediator, der *Föderationsserver*, Anfragen entgegen, zerlegt sie in Teilanfragen an die Komponentensysteme und setzt die Teilergebnisse zum Ergebnis der föderierten Query zusammen. Als föderierte Anfragesprache wird eine Teilmenge der Object Query Language OQL der ODMG [ODMG97] verwendet, die um an die SQL-Syntax angelehnte Operationen zum Modifizieren von Objektattributen erweitert wurde. Als Komponentensysteme werden derzeit Oracle 8i, Microsoft SQL Server und das objektorientierte System O_2 unterstützt. Auf die lokalen Datenbanken kann weiterhin durch lokale Transaktionen direkt zugegriffen werden.

Die Architektur von VHDBS ist in Abbildung 4.1 dargestellt. Das System besteht aus den folgenden Hauptmodulen:

- Der *Föderationsserver* ist die zentrale Komponenten von VHDBS. Er stellt alle Dienste der föderierten Ebene zur Verfügung, insbesondere Dienste zur Verwaltung des föderierten Schemas und zum Ausführen von föderierten Anfragen und Modifikationen. Der Föderationsserver zerlegt eine Anfrage in Teilanfragen für die beteiligten Komponentensysteme und führt sie parallel über die jeweiligen Datenbankadapter aus. Die Ergebnisse dieser Teilanfragen werden in einer ausgezeichneten O_2-Datenbank, der *Föderationsdatenbank*, gespeichert, wo sie zur endgültigen Ergebnismenge kombiniert werden, indem z.B. föderierte Joins berechnet werden.
- Der *Metaserver* verwaltet Metainformationen, die unter anderem Definitionen des föderierten Schemas und der lokalen Schemata umfassen, sowie Informationen

[1] Verteiltes heterogenes Datenbanksystem

über die verwendeten Komponentensysteme. Außerdem ist hier die Benutzerverwaltung angesiedelt.

- Die Komponentendatenbanken werden durch systemspezifische *Datenbankadapter* an das föderierte System angebunden. Sie erlauben Zugriff auf die Daten in den Komponentensystemen über ein einheitliches Interface. Jeder Datenbankadapter für ein spezifisches Datenbanksystem (oder eine Klasse von Systemen) bildet das föderierte Datenmodell auf das Datenmodell des lokalen Systems und die föderierte Anfragesprache auf die lokale Anfragesprache ab.

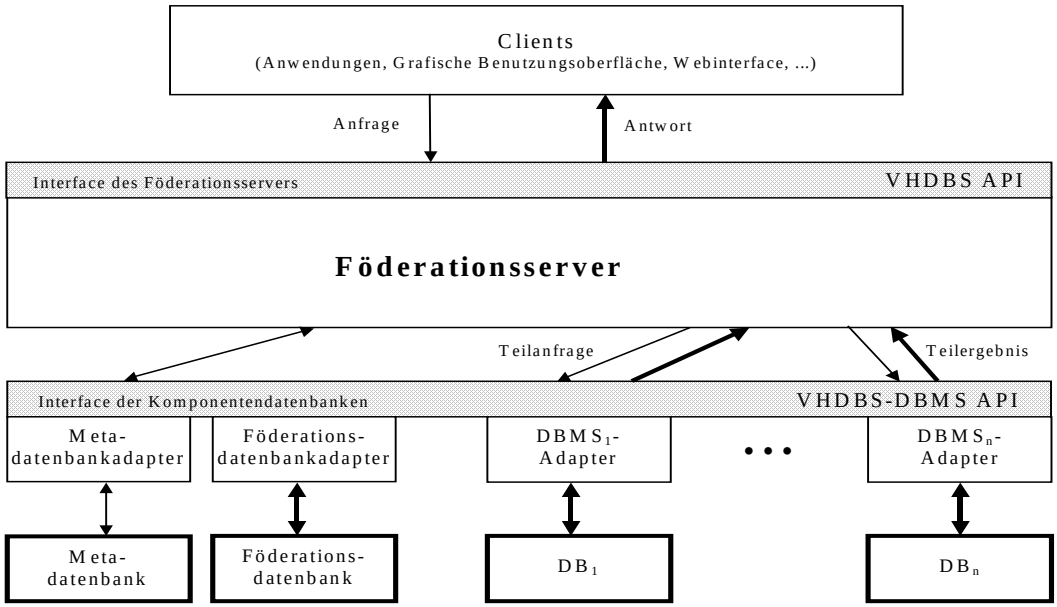

Abbildung 4.1 - Architektur des VHDBS-Systems

Die Interfaces aller Module dieser Architektur werden mit der Interface-Definitions-Sprache IDL des CORBA-Standards spezifiziert, so dass die gesamte Kommunikation zwischen ihnen über CORBA abgewickelt werden kann. Auch die Daten in den Komponentensystemen werde in CORBA-Objekten gekapselt. Auf diese Weise können die Module nach Bedarf zu CORBA-Servern gruppiert werden, so dass eine Anpassung an die Bedürfnisse der Anwendung möglich ist.

Die Kommunikation zwischen solchen Servern wird vollständig über CORBA abgewickelt, so dass die Server über lokale oder globale Netze verteilt werden können. Das gleiche gilt für die Kommunikation mit Clients der Föderation, so dass ein einfacher Zugriff sowohl von Anwendungsprogrammen (z.B. über die Anbindung von CORBA an C++) als auch von Webanwendungen (z.B. über die Anbindung von CORBA an Java) möglich ist.

4.1.2 Implementierungsaspekte

In diesem Abschnitt stellen wir Aspekte der Implementierung von VHDBS vor, die für das Verständnis der weiteren Ausführungen in diesem Kapitel notwendig sind. Wir vereinfachen die tatsächliche Implementierung dabei an manchen Stellen, soweit die Details für diese Arbeit nicht erforderlich sind; weiterführende Informationen können [WW96, Weiß97] entnommen werden.

Wie wir bereits im vorgehenden Abschnitt kurz gesagt haben, werden alle Objekte und Relationen in den Komponentendatenbanken sowie in der Meta- und in der Föderationsdatenbank durch CORBA-Objekte dargestellt. Alle Objekte werden dabei vom Basisinterface `dbObject` abgeleitet, das elementare Methoden enthält, mit denen der Typ eines Objektes bestimmt und zwei Objekte verglichen werden können. Weiter gibt es die elementaren Objektklassen `dbInteger`, `dbReal`, `dbString` und `dbBoolean`, die ganze Zahlen, reelle Zahlen, Strings und boole'sche Werte repräsentieren. Tupel in Relationen werden durch den Typ `dbTuple` realisiert, der Methoden `get` und `set` zum Lesen und Ändern von Attributen enthält, die durch die genannten elementaren Typen dargestellt werden. Relationen, die in der VHDBS-Terminologie eine spezielle Art von *Repository* für Daten sind, werden durch Objekte vom Typ `dbCollection` repräsentiert, die unter anderem Methoden enthalten, um Tupel einzufügen und zu löschen. Der Typ `dbCollection` unterstützt außerdem ein Cursorkonzept in Gestalt eines `dbCursor`-Objektes, mit dem man mittels der Operationen `first` und `next` über alle Tupel in der Relation iterieren kann.

VHDBS unterscheidet zwei Arten von Repositories: *Spiegelrepositories*, die unmittelbar Relationen in einem Komponentensystem entsprechen, und *Kombirepositories*, die das Ergebnis von Anfragen sind, die auf andere Kombirepositories und auf Spiegelrepositories in mehreren Komponentensystemen zugegriffen haben können. Alle Kombirepositories werden materialisiert, d.h. sie werden zum Zeitpunkt der Anfrageauswertung als `dbCollection` in der Föderationsdatenbank gespeichert. Sie werden nicht automatisch aktualisiert, wenn sich die Relationen ändern, aus denen das Kombirepository entstanden ist.

Anfragen werden in VHDBS in einer Variante der Object Query Language OQL der ODMG formuliert. Jede Anfrage hat folgenden schematischen Aufbau:

```
SELECT <Attribute> FROM <Repositories> WHERE <Suchprädikat>
```

Im Attributteil werden die Attribute der Objekte angegeben, die im Ergebnis enthalten sein sollen; das jeweilige Repository, aus dem das Objekt stammt, wird mit einem Kürzel bezeichnet, das im FROM-Teil angegeben wird. Neben Attributen können auch Methodenaufrufe angegeben werden, außerdem werden ähnlich wie in SQL Aggregationsfunktionen und ‚*' zur Selektion aller Attribute eines Objektes unterstützt. Im FROM-Teil werden die Repositories spezifiziert, mit denen die Anfrage arbeitet. Dabei können sowohl Spiegel- als auch Kombirepositories benutzt werden.

Eine typische Anfrage ist etwa die folgende, die alle Angestellten und ihr Gehalt bestimmt, die am TraFIC-Projekt mitarbeiten:

```
SELECT a.name, a.gehalt
FROM p IN Projekte, a IN Angestellte, aip IN arbeitet_in_Projekt
WHERE a.id=aip.angestellter AND p.pid=aip.projekt
      AND p.bezeichnung='TraFIC';
```

Zur Auswertung wird diese Anfrage über die Methode `select` an den Föderationsserver übergeben, der sie zunächst analysiert und einen Syntaxbaum erstellt; der Syntaxbaum für die Beispielanfrage ist in Abbildung 4.2 dargestellt. Intern wird dieser Baum durch die Datenstruktur `OQLSequence` repräsentiert.

Der Föderationsserver berechnet anschließend einen Ausführungsplan für die Anfrage, führt Teilanfragen über die Datenbankadapter in den Komponentendatenbanken aus und

schreibt die Zwischenergebnisse in die Föderationsdatenbank. Auch die Datenbankadapter haben dazu eine Methode `select` in ihrem Interface. Möglichst viele Abhängigkeiten zwischen Objekten verschiedener Repositories (z.B. Joins) werden durch eine geeignete Formulierung der Teilanfragen ausgewertet, die übrigen müssen im Föderationsserver berechnet werden. Das Ergebnis der Anfrage liegt dann als `dbCollection` in der Föderationsdatenbank; an den Client wird diese `dbCollection` zurückgegeben.

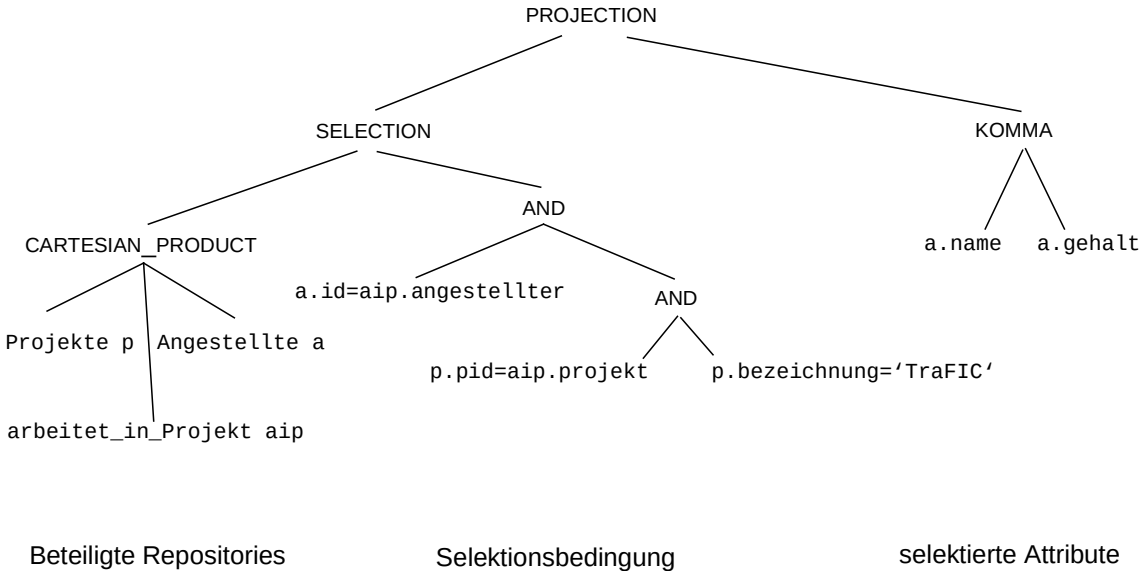

Abbildung 4.2 - Syntaxbaum für die Beispielanfrage

Modifikationen, d.h. Änderungen von Objektattributen, Methodenaufrufe sowie Einfügen in oder Löschen von Objekten aus Repositories, können entweder in der Anfragesprache formuliert und über den Föderationsserver ausgeführt werden oder unmittelbar über Methoden der entsprechenden Klasse ausgeführt werden. Im einzelnen gibt es

- die Methode `set` der Klasse `dbTuple`, um ein Attribut des Tupels zu ändern,
- die Methode `execMeth` der Klasse `dbTuple`, um eine Methode auszuführen, und
- die Methoden `insertElement` bzw. `deleteElement` der Klasse `dbCollection`, um ein Objekt in ein Repository einzufügen bzw. um es daraus zu löschen.

Um Änderungen von Objektattributen in der Anfragesprache zu formulieren, wird zunächst durch eine Anfrage die zu ändernde Objektmenge bestimmt. Anschließend werden die auszuführenden Änderungen mit Hilfe eines Cursors auf jedes der so bestimmten Objekte angewendet. Um also das Gehalt aller Mitarbeiter am TraFIC-Projekt zu erhöhen, könnte man die folgende Anfrage formulieren:

```
FOR a IN (SELECT ang FROM p IN Projekte, ang IN Angestellte,
                 aip IN arbeitet_in_Projekt
          WHERE a.id=aip.angestellter AND p.pid=aip.projekt
            AND p.bezeichnung='TraFIC')
DO
   ang.gehalt:=ang.gehalt + 10.000;
END;
```

Zur Ausführung dieser Anweisung bestimmt der Föderationsserver zunächst wie oben beschrieben eine `dbCollection`, die alle Tupel enthält, die als Ergebnis der `SELECT`-

Anweisung erhalten wurden. Für jedes dieser Tupel wird anschließend die Methode `set` zum Ändern des Attributes `gehalt` aufgerufen. Einfüge- und Löschoperationen können ähnlich formuliert werden, das Ausführen von Methoden haben wir bereits oben im Rahmen der Formulierung von Anfragen beschrieben.

Zur Realisierung von VHDBS wurde die CORBA-Implementierung Orbix von Iona verwendet [Iona95, Iona99a]. Die Techniken, die im weiteren Verlauf dieser Arbeit beschrieben werden, verwenden an bestimmten Stellen Funktionalität, die nicht Teil des CORBA-Standards ist, sondern Orbix-spezifische Erweiterungen des Standards darstellen.

4.2 Konzepte und Probleme des Object Transaction Service

Zur Unterstützung von Transaktionen wurde CORBA um den *Object Transaction Service* [OMG94,OMG97], kurz OTS, erweitert. Wir beschreiben im folgenden Abschnitt 4.2.1 zunächst wichtige Konzepte des OTS. Anschließend zeigen wir in Abschnitt 4.2.2 auf, warum eine Transaktionsunterstützung für VHDBS, die ausschließlich auf dem Object Transaction Service basiert, nicht ausreichend wäre.

4.2.1 Transaktionen im Object Transaction Service

Der Object Transaction Service erlaubt die Zusammenfassung von Operationen auf unabhängigen Datenquellen zu Transaktionen, deren Atomarität er durch ein verteiltes Zweiphasen-Commitprotokoll sicherstellt. Jede Datenquelle wird dabei über einen sogenannten *Resource Manager* angesprochen, der die Datenquelle in das globale Commitprotokoll einbindet. Wenn zum Beispiel Datenbanksysteme angesprochen werden, führt der jeweilige Resource Manager über das XA-Interface [XOpen96,GR93,BN97] des Datenbanksystems die Schritte des 2PC durch. Es ist aber auch möglich, eigene Datenquellen einzubinden, für die man selbst die notwendigen Operationen implementieren muss, die zur Integration in das verteilte Commitprotokoll notwendig sind; der OTS sorgt dafür, dass diese Operationen zum richtigen Zeitpunkt aufgerufen werden.

Es gibt zwei verschiedene Programmiermodelle, um transaktionale Applikationen mit dem Object Transaction Service zu realisieren:

- *Das indirekt-implizite Programmiermodell*: In diesem Modell liegt die Verantwortung für die Kombination mehrerer Methodenaufrufe zu einer Transaktion vollständig beim OTS. Ein Client muss lediglich explizit eine Transaktion beginnen, indem er die `begin`-Methode eines ausgezeichneten Objektes, des `Current`-Objektes, aufruft. Ab diesem Zeitpunkt werden alle folgenden Methodenaufrufe von Objekten, die vorher als transaktional gekennzeichnet wurden, innerhalb dieser Transaktion ausgeführt. Die Interfaces dieser Objekte müssen dazu vom Pseudointerface `TransactionalObject` abgeleitet werden, damit der OTS weiß, dass er zu Aufrufen der Methoden dieses Interfaces implizit die aktuelle Transaktion als zusätzlichen, für die Anwendung unsichtbaren Parameter hinzufügen muss. Der OTS kümmert sich auch um die automatische Integration von Resource Managern in die Transaktion, indem er zum Beispiel Methoden ihres XA-Interfaces benutzt. Das Beenden der Transaktion geschieht durch den Aufruf der `commit`- oder `abort`-Methode des `Current`-Objektes.

- *Das direkt-explizite Programmiermodell*: In diesem Modell muss das Anwendungsprogramm selbst dafür sorgen, dass mehrere Methodenaufrufe im Kontext einer Transaktion ausgeführt werden; der OTS stellt lediglich dazu benötigte Mechanismen bereit. Eine Transaktion wird in diesem Modell durch drei verschiedene Objekte dargestellt: Ein `Control`-Objekt, das die eigentliche Transaktion repräsentiert und Methoden bereitstellt, mit denen man die beiden übrigen Objekte erhalten kann; ein `Terminator`-Objekt, das Methoden zum Commit und Abort der Transaktion bereitstellt, und ein `Coordinator`-Objekt, mit dem man Resource Manager in den Ablauf der Transaktion integrieren kann. Ein `TransactionFactory`-Objekt dient zum Starten einer Transaktion, man erhält als Rückgabe ein `Control`-Objekt. Jeder Methode, die im Kontext einer Transaktion aufgerufen werden soll, muss explizit das `Control`-Objekt der aktuellen Transaktion als Parameter übergeben werden. Die Implementierung der Methode muss dann mit OTS-Methoden lokal eine Repräsentation dieser Transaktion erzeugen und mit Methoden des `Coordinator`-Objekts gegebenenfalls benötigte Resource Manager in die Transaktion einbinden, bevor auf sie zugegriffen werden kann. Zum Beenden der Transaktion wird schließlich eine der Methoden des `Terminator`-Objektes benutzt.

Neben den üblichen flachen Transaktionen unterstützt der OTS auch eine Variante der in Abschnitt 2.2.4 vorgestellte Mehrschichtentransaktionen, allerdings mit zwei wesentlichen Unterschieden: Sperren, die Subtransaktionen erwerben, werden an ihrem Ende nicht freigegeben, sondern an die Vatertransaktion vererbt. Außerdem werden die Änderungen, die innerhalb einer Subtransaktion gemacht wurden, noch nicht endgültig in die Datenbank geschrieben, dies geschieht erst beim Commit der obersten Transaktion. Es ist immer möglich, noch nicht beendete Subtransaktionen zurückzusetzen und damit ihre Effekte aufzuheben. Man erzielt mit diesen sogenannten *geschlossen geschachtelten Transaktionen* also keinen Gewinn an Parallelität, muss sich auf der anderen Seite aber auch nicht um Kompensation partieller Effekte kümmern. Der wesentliche Vorteil ist, dass man durch die Aufteilung einer Transaktion in Subtransaktionen sehr einfach einen Teil der Ausführung zurücksetzen kann, indem man die entsprechende Subtransaktion zurücksetzt. Man erreicht also auf diese Weise eine Verfeinerung des möglichen Granulats für die Rollback-Operation.

Um die Isolation nebenläufiger Transaktionen zu gewährleisten, stellt CORBA in Gestalt des *Concurrency Control Service OCCS* [OMG96] einen Sperrmechanismus zur Verfügung, mit dem CORBA-Objekte in den üblichen Modi gemeinsam und exklusiv gesperrt werden können. Die Sperren werden stark zweiphasig verwaltet, d.h. alle Sperren werden erst am Ende der Transaktion, die sie angefordert hat, freigegeben.

4.2.2 OTS als Transaktionskomponente von VHDBS

VHDBS wurde ursprünglich als föderiertes System entwickelt, mit dem nur lesend auf die Komponentendatenbanken zugegriffen wurde, so dass keine besonderen Maßnahmen zur Unterstützung föderierter Transaktionen notwendig waren. In einer späteren Projektphase sollten auch Modifikationen der Daten über die föderierte Ebene möglich sein, was zusätzliche Protokolle zur Gewährleistung globaler Atomarität und Serialisierbarkeit notwendig machte.

Folgt man dem CORBA-Ansatz, so liegt es nahe, den Object Transaction Service mit dem indirekt-impliziten Programmiermodell einzusetzen, um VHDBS um die Unter-

stützung von föderierten Transaktionen zu erweitern. Um dies zu tun, muss man die folgenden (einfachen) Modifikationen an VHDBS vornehmen:

- Alle Interfaces, die im Kontext einer Transaktion verwendet werden sollen, müssen explizit als transaktional markiert werden, indem sie vom Pseudointerface `TransactionalObject` erben.
- Die Datenbankadapter müssen so geändert werden, dass sie keine expliziten SQL-Anweisungen zur Transaktionskontrolle benutzen. Statt dessen übernimmt der OTS transparent die Steuerung der Subtransaktionen über das XA-Interface der lokalen Datenbank.
- Zu allen Clientprogrammen müssen OTS-Anweisungen hinzugefügt werden, um Transaktionen zu beginnen und zu beenden. Der OTS fügt dann implizit die aktuelle Transaktion als (unsichtbaren) Parameter zu jedem Aufruf eines transaktionalen Objektes hinzu, so dass es nicht notwendig ist, diese Aufrufe selbst zu ändern.

Auf diese Weise lässt sich einfach eine Unterstützung für föderierte Transaktionen in VHDBS integrieren, aber man löst nur einen Teil des Problems: Der OTS steuert das Zweiphasen-Commitprotokoll über das XA-Interface der beteiligten Komponentensysteme und garantiert damit die Atomarität von föderierten Transaktionen. Auf der Seite der Concurrency Control dagegen bietet CORBA in Gestalt des Concurrency Control Service OCCS lediglich verschiedene Sperrmodi, mit denen der Zugriff auf CORBA-Objekte synchronisiert werden kann, was für föderierte Transaktionen, die auf Daten in lokalen Datenbanksystemen zugreifen, nicht eingesetzt werden kann. Der OTS bietet weder Protokolle an, die globale Serialisierbarkeit gewährleisten, noch erlaubt er, solche Protokolle auf einfache Weise einzubinden. Wie wir in Kapitel 3 gesehen haben, sind solche Protokolle für eine korrekte Ausführung absolut notwendig, daher löst OTS alleine das Problem nicht.

Selbst wenn man durch Eingriffe in die interne Verarbeitung von VHDBS Verfahren zur Gewährleistung globaler Serialisierbarkeit einbinden könnte, hätte dieser OTS-zentrierte Ansatz wesentliche Nachteile:

- Die Kontrolle über den Ablauf der Transaktion, insbesondere die Ausführung der Commit-Operation, läge ausschließlich beim OTS. Manche Strategien erfordern aber weitergehende Kontrollmöglichkeiten, z.B. Verzögerungen der Ausführung eines Commit oder Zugriffe auf die lokalen Datenbanksysteme beim Commit (z.B. die Tickettechnik).
- Die Mehrschichtentransaktionsstrategie kann nur sinnvoll eingesetzt werden, wenn die Steuerung der Transaktionen nicht durch den OTS erfolgt, weil die Variante geschachtelter Transaktionen, die der OTS bietet, die Vorteile der Mehrschichtenstrategie gerade zunichte macht, da sie Sperren erst am Ende der obersten Transaktion freigibt.
- Clients von VHDBS müssten sich mit der technischen Realisierung der Transaktionen befassen, insbesondere müssten sie je nach gewählter Strategie an bestimmten Stellen der Transaktion Methoden der Transaktionsverwaltung aufrufen (z.B. eine explizite Methode am Anfang der Transaktion, falls dies der beste Zeitpunkt für den Ticketzugriff in einem lokalen Scheduler ist). Die Details der Transaktionsverwal-

tung und der eingesetzten Protokolle zur globalen Concurrency Control sollten aber vollständig innerhalb der Transaktionsverwaltung gekapselt sein.

Seit dem Abschluss der Konzeption der Transaktionsverwaltung für VHDBS sind zwar einige dieser Nachteile beseitigt worden, speziell ist in der aktuellen OTS-Spezifikation 1.1 der OMG [OMG97] ein `Synchronization`-Interface vorgesehen, mit dem man Methoden angeben kann, die nach dem Commit-Wunsch der Transaktion, aber vor den Anstoßen des eigentlichen 2PC aufgerufen werden. Trotzdem kommt die OTS-zentrierte Lösung wegen der immer noch vorhandenen Probleme für die Transaktionsverwaltung in VHDBS nicht in Frage.

4.3 Die Transaktionsverwaltung in VHDBS

4.3.1 Architektur

Wegen der Nachteile, die wir im letzten Abschnitt herausgearbeitet haben, werden Transaktionen in VHDBS nicht ausschließlich mit den Transaktionsmechanismen des Objekt Transaction Service realisiert. Wir nehmen statt dessen einen zusätzlichen föderierten Transaktionsmanager, genannt TraFIC[2], in das System auf [SW98, SW99a, SW99b]. Dieser Manager bietet einen Satz von Strategien zur Gewährleistung von globaler Atomarität und Serialisierbarkeit an, aus denen Clients diejenigen auswählen können, die ihren Anforderungen am besten entsprechen.

Der Aufbau dieses Servers ist in Abbildung 4.3 dargestellt, er enthält die folgenden Komponenten:

- Interfaces zur Transaktionskontrolle, d.h. zum Starten und Beenden von Transaktionen
- Interfaces zur Ausführung von Queries und Updates auf den Objekten und Repositories der lokalen Datenbankadapter, sowohl der Komponentendatenbanksysteme als auch der Föderationsdatenbank
- Strategien zur Gewährleistung einer atomaren und serialisierbaren Ausführung der globalen Transaktionen, insbesondere die in Kapitel 3 vorgestellten Varianten der Tickettechniken, Mehrschichtentransaktionen und die graphbasierte Technik.
- Strategien zur Gewährleistung von globaler Snapshot Isolation für globale Transaktionen, insbesondere die in Abschnitt 3.8 vorgestellten Techniken.
- Mechanismen zur Realisierung der vorgenannten Strategien, die teilweise von mehreren Strategien wiederverwendet werden.
- Interfaces zur Auswahl und Konfiguration der Strategien und Mechanismen

Das Interface jeder dieser Komponenten wird in der CORBA-Interface-Definitionssprache IDL beschrieben, so dass es unabhängig von der tatsächlichen Implementierung bleibt. Durch diese klare Definition der Interfaces zwischen den Komponenten können Erweiterungen einfach vorgenommen werden, insbesondere können neue Strategien und Mechanismen einfach eingebaut und vorhandene Implementierungen einfach ausgetauscht werden.

---

[2] Transaktionsverwaltung in Föderierten Informationssystemen auf der Basis von CORBA

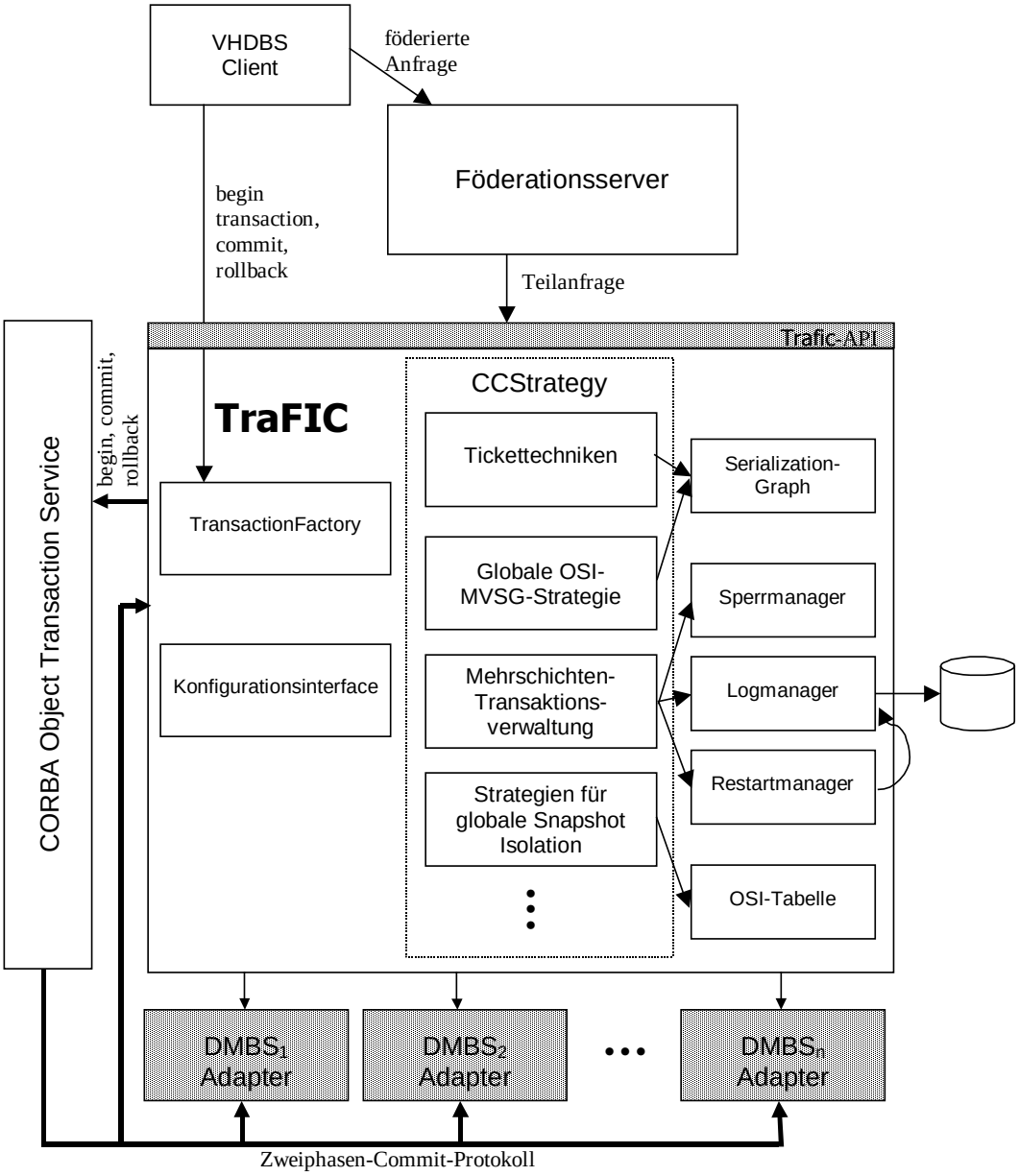

Abbildung 4.3 - Aufbau von TraFIC

Diese Architektur erlaubt also eine flexible Anpassung von TraFIC an die Gegebenheiten im praktischen Einsatz. Vom Standpunkt der Skalierbarkeit und Verfügbarkeit aus dagegen ist es nicht optimal, einen einzigen, zentralen TraFIC-Server zu haben. Zum einen wird die Leistung dieses Servers bei hoher Last schnell sinken, und zum anderen wird das gesamte System bei seinem Ausfall lahmgelegt.

Eine hohe Verfügbarkeit kann man relativ einfach dadurch erreichen, dass man einen Backup-Server auf einem anderen Rechner vorsieht, der bei Ausfall des TraFIC-Servers übernimmt. Natürlich müssen in diesem Fall alle Transaktionen, die zum Zeitpunkt des Ausfalls aktiv waren, vom Backup-Server zurückgesetzt werden, bevor die weitere Ausführung erlaubt werden darf.

Bis zu einem gewissen Maß kann man das System an steigende Last anpassen, indem man die verschiedenen Komponenten auf eigenständige Rechner verteilt. Da alle Kommunikation zwischen den Komponenten über CORBA erfolgt, ist dazu keinerlei Änderung an der bisherigen Architektur notwendig. Steigt die Last weiter, kann jedoch jede einzelne Komponente zum Engpass werden. Das System wird daher nur dann vollständig skalierbar, wenn man jede Komponente mehrfach instantiieren kann, so dass sich die anfallende Last auf diese Instanzen verteilen kann. Die verwendeten Strategien zur globalen Concurrency-Control erfordern jedoch konzeptionell eine zentrale Instanz, um Serialisierbarkeit zu gewährleisten. In diesem Fall muss also die Kohärenz der nunmehr verteilt gehaltenen globalen Daten, die sich sonst nur in einem Server befinden, sichergestellt werden. Durch gängige Kohärenzprotokolle, wie sie auch in verteilten Datenbanken eingesetzt werden [Lome94, Rahm94, Dadam96], könnte man dieses Problem aber mit dem entsprechenden Aufwand lösen.

Dieses Problem betrifft im wesentlichen alle Mechanismen, die globale Graphen aufbauen oder wie die Sperrmanager alle Informationen zu einer Komponentendatenbank verwalten müssen. Die übrigen Mechanismen wie der Logmanager, aber auch der eigentliche TraFIC-Server mit den Implementierungen der `TransactionFactory` und der Strategien können mehrfach instantiiert werden, wenn sie zu Performanceengpässen werden sollten.

4.3.2 Arbeitsweise

Der Grundgedanke bei der Integration von TraFIC in das VHDBS-System ist es, alle Aufrufe von Methoden, die im Kontext einer Transaktion umgeleitet werden müssen, nicht direkt an die Datenbankadapter weiterzugeben, sondern über TraFIC umzuleiten.

Im einzelnen läuft eine Transaktion wie folgt ab:

Ein Client von VHDBS beginnt eine Transaktion, indem er das `TransactionFactory`-Objekt des TraFIC-Servers benutzt. Er erhält als Rückgabe ein `Transaction`-Objekt, das in TraFIC die Transaktion repräsentiert.

Anschließend

- stellt er Anfragen, indem er Methoden des Föderationsservers aufruft, der die Anfragen nach dem Parsen und Optimieren in Teilanfragen für die einzelnen Datenbanken zerlegt und sie an TraFIC zur weiteren Bearbeitung weitergibt; oder er
- macht Modifikationen, indem er wiederum Methoden des Föderationsservers oder direkt Methoden von TraFIC aufruft, um die von den Anfragen zurückgegebenen Objekte zu ändern.

TraFIC trifft eventuell notwendige Maßnahmen und führt dann den Aufruf im Datenbankadapter aus, insbesondere kümmert sich TraFIC selbst um die Verwaltung der notwendigen OTS-Transaktionen.

Schließlich beendet der Client die Transaktion durch Aufruf einer Methode des `Transaction`-Objektes. Bevor TraFIC das Commit der Transaktion ausführt, wird sichergestellt, dass die entstandene Ausführung global serialisierbar ist und die Atomarität der Transaktion gewährleistet werden kann.

4.4 Integration der Transaktionsverwaltung in VHDBS

In diesem Abschnitt stellen wir Details der Integration der Transaktionsverwaltung in das bestehende föderierte Datenbanksystem VHDBS vor.

4.4.1 Operationen zur Transaktionskontrolle

Zur Gewährleistung von Atomarität und zur einfachen Integration der Datenbanksysteme in einen Transaktionskontext stützt sich TraFIC auf Dienste des Object Transaction Service. Die Aufrufe des OTS werden dabei vollständig in TraFIC gekapselt, zum Starten und Beenden von Transaktionen verwenden die Clients ausschließlich Interfaces von TraFIC. Die notwendigen Aufrufe an den darunterliegenden OTS werden von TraFIC selbst ausgeführt. Für die Clients geschieht dies vollständig transparent, sie haben keinen Zugriff auf den OTS. Die Datenbankadapter dagegen müssen an den OTS angepasst werden, da ihre Zugriffe auf die Datenbanken im Kontext von OTS-Transaktionen stattfinden muss.

Zum Starten einer Transaktion dient das `TransactionFactory-Interface` von TraFIC. Es enthält als einzige Methode `BeginTransaction`, durch deren Aufruf eine neue Transaktion begonnen wird. Sie gibt ein `Transaction`-Objekt zurück, das die neue Transaktion repräsentiert.

```
interface TransactionFactory
{
   Transaction BeginTransaction();
};
```

Auf dem zurückgegebenen `Transaction`-Objekt werden die übrigen Operationen zur Kontrolle des Transaktionsablaufes ausgeführt. Die Methode `CommitTransaction` wird benutzt, um die von der Transaktion gemachten Änderungen endgültig in die Datenbanken zu schreiben. Die Methode `AbortTransaction` bricht die Transaktion ab und macht alle Änderungen rückgängig, die die Transaktion bisher vorgenommen hat. Beide Methoden lösen die Exception `NotInTransaction` aus, wenn die Transaktion bereits beendet war. Mit der Methode `SetIsolationLevel` kann für diese Transaktion ein Isolation Level gewählt werden.

```
interface Transaction
{
   void CommitTransaction() raises (NotInTransaction);
   void AbortTransaction()  raises (NotInTransaction);
   void SetIsolationLevel(in IsolationLevel l);
};
```

Neben den hier gezeigten Methoden sind auch Methoden für die Ausführung von Anfragen und Modifikationen auf den lokalen Datenbanken vorhanden, die im folgenden Abschnitt angesprochen werden.

4.4.2 Operationsintegration

4.4.2.1 Grundsätzliches

Es gibt in VHDBS zwei grundsätzlich verschiedene Möglichkeiten, wie Anfragen und Modifikationen ausgeführt werden können:

- Formulieren von Anfragen und Modifikationen als OQL-Befehlssequenz und Ausführen über die `Select`-Methode des Föderationsservers, der die Sequenz zerlegt und an die Datenbankadapter weitergibt.
- Direktes Modifizieren von Objekten (`dbTuple`, `dbCollection`) durch Aufrufen von Objektmethoden wie `dbTuple.set()` und `dbCollection.insertElement()`. Der Föderationsserver ist daran nicht beteiligt, da die Methoden unmittelbar von den Objekten in den Datenbankadaptern ausgeführt werden.

TraFIC muss über alle Operationen informiert werden, die Clients von VHDBS ausführen, um die notwendigen Maßnahmen zur Gewährleistung globaler Serialisierbarkeit und Atomarität treffen zu können. Wie wir bereits im letzten Abschnitt vorgestellt haben, werden dazu alle Anfragen und Modifikationen, die Clients oder der Föderationsserver an die Datenbankadapter schicken, über TraFIC umgeleitet.

Abbildung 4.4 zeigt den schematischen Ablauf am Beispiel des Föderationsservers, die Behandlung der Clients ist analog.

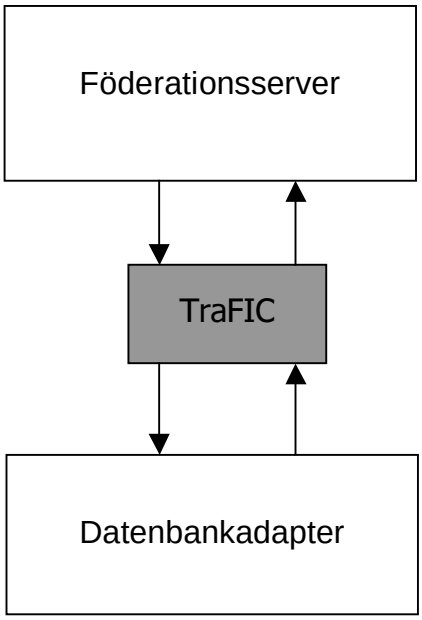

Abbildung 4.4 - Umleiten über TraFIC

Wenn der Föderationsserver eine Aktion ausführen will, sendet er sie an TraFIC, indem er eine Methode des `Transaction`-Objektes aufruft, das die aktuelle Transaktion repräsentiert. Dort können dann erforderliche Maßnahmen zur Gewährleistung globaler Serialisierbarkeit und Atomarität getroffen werden, falls die aktuelle Strategie dies vorsieht. Anschließend schickt TraFIC die Aktion an den zuständigen Datenbankadapter, der sie ausführt und das Ergebnis – in Form einer Objektmenge bei einer Anfrage oder eines Rückgabewertes bei einer Modifikation – an TraFIC zurückschickt, von wo es unmittelbar an den Föderationsserver weitergeleitet wird.

Durch die klare Trennung von Föderationsserver und TraFIC wird der modulare Aufbau von VHDBS gewahrt, zusätzlich bleibt für Clients die Möglichkeit erhalten, direkt Objekte zu ändern und ihre Methoden aufzurufen, ohne den Umweg über den Föderationsserver gehen zu müssen. Das Leistungspotential des Föderationsservers wird nicht ein-

geschränkt, da er die gleiche Zahl von Methodenaufrufen wie ohne die Einbindung von TraFIC machen muss.

4.4.2.2 Ablauf einer Anfrage

Anfragen werden immer über die `Select()`-Methode des Föderationsservers gestellt, direkte Methodenaufrufe durch Clients kommen nicht vor. Es genügt also, diesen Fall zu betrachten. Der Ablauf der Ausführung einer Anfrage ist in Abbildung 4.5 dargestellt. Für den Nachrichtenfluss ist es dabei unerheblich, ob die Föderationsdatenbank oder eine Komponentendatenbank angesprochen wird. Im einzelnen werden sind die folgenden Aktionen notwendig:

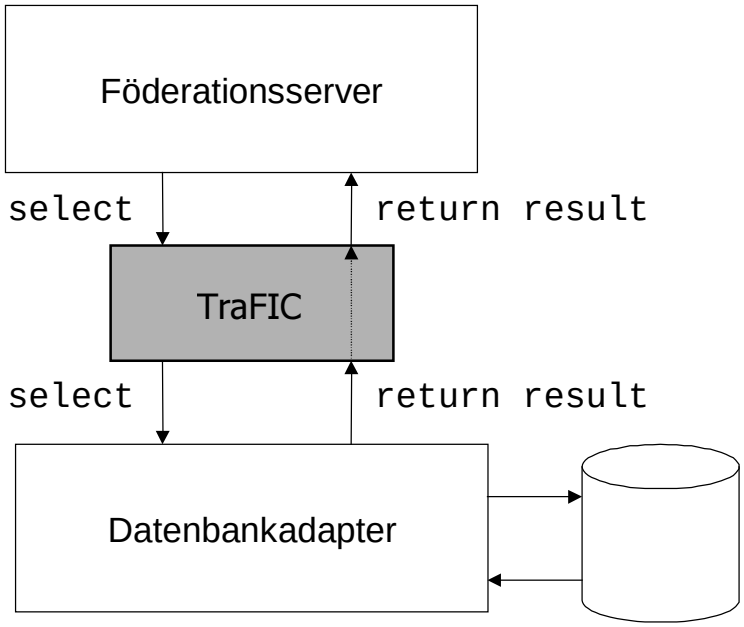

Abbildung 4.5 - Ablauf einer Anfrage

- Der Föderationsserver trennt die evtl. komplexe föderative Anfrage in ihre Einzelbestandteile auf und schickt diese Teilanfragen einzeln an TraFIC, statt sie direkt an die zuständigen Datenbankadapter zu schicken. Er ruft dazu die Methode `select` des `Transaction`-Objektes auf, das die aktuelle Transaktion repräsentiert.
- TraFIC trifft nun erforderliche Maßnahmen zur Wahrung der Transaktionseigenschaften und leitet dann die Teilanfragen an die zuständigen Datenbankadapter weiter.
- Der Datenbankadapter führt die jeweilige Teilanfrage aus, erzeugt die entsprechenden Objekte und gibt sie an TraFIC zurück bzw. meldet einen Fehler, falls die Ausführung der Anfrage in der Datenbank fehlschlägt.
- TraFIC wertet evtl. Fehler aus und gibt die Ergebnisobjekte an den Föderationsserver zurück.

4.4.2.3 Ablauf einer Modifikation

Unter dem Begriff „Modifikation“ werden Einfüge-, Änderungs- und Löschoperationen sowie Methodenaufrufe von Objekten in einer Komponentendatenbank zusammengefasst, also alle Operationen, die die lokale Datenbank potentiell ändern. Die Behandlung von Änderungen von Objekten in der Föderationsdatenbank beschreiben wir im nächsten Abschnitt.

Wie in Abschnitt 4.4.2.1 erwähnt, gibt es zwei verschiedene Möglichkeiten, wie Clients Modifikationen durchführen können: Entweder mit einer OQL-Befehlssequenz über die `Select()`-Methode des Föderationsservers oder durch einen Methodenaufruf eines Objektes, das vorher als Ergebnis einer Anfrage zurückgegeben wurde. Grundsätzlich unterscheiden sich diese beiden Möglichkeiten nur darin, wer TraFIC aufruft: Im ersten Fall ist es der Föderationsserver, im zweiten Fall der Client. Im folgenden wird beispielhaft das Vorgehen bei einer Modifikation über den Föderationsserver beschrieben.

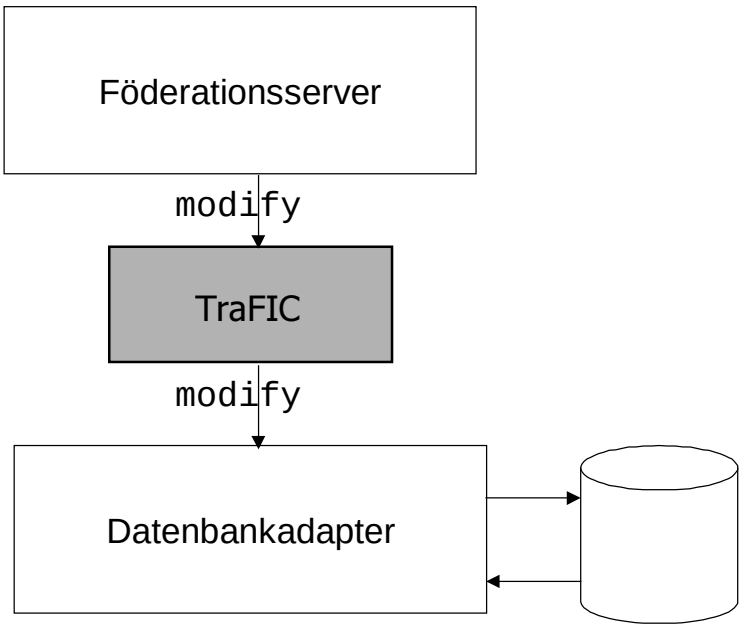

Abbildung 4.6 - Ablauf einer Modifikation in einer KDB

Der Ablauf einer Modifikation von Objekten in einer Komponentendatenbank ist vergleichbar mit dem Ablauf einer Anfrage, aber sogar noch etwas einfacher: Aus dem Aufbau von des OQL-Dialekts von VHDBS folgt unmittelbar, dass sich jede Modifikation nur auf Objekte in genau einer Datenbank beziehen kann. Der Schritt zur Zerlegung der Modifikation entfällt daher. Im einzelnen werden die folgenden Aktionen ausgeführt (siehe Abbildung 4.6):

- Der Föderationsserver schickt die Modifikation an TraFIC, statt sie direkt an die zuständigen Datenbankadapter zu schicken. Er ruft dazu die Methode `select` des `Transaction`-Objektes auf, das die aktuelle Transaktion repräsentiert.
- TraFIC trifft nun erforderliche Maßnahmen zur Wahrung der Transaktionseigenschaften und leitet dann die Modifikation an den zuständigen Datenadapter weiter.

- Der Datenbankadapter führt die Modifikation in der lokalen Datenbank aus und meldet an TraFIC einen Rückgabewert, der Auskunft über evtl. Fehler bei der Ausführung der Modifikation in der Datenbank gibt.
- TraFIC wertet evtl. Fehler aus und meldet sie ggf. an den Föderationsserver zurück.

4.4.2.4 Behandlung von Objekten in der Föderationsdatenbank

Wenn die Objekte, die geändert werden sollen, in Kombirepositories und damit in der Föderationsdatenbank enthalten sind, genügt der oben gezeigte Ablauf nicht. Zusätzlich muss nun noch die Änderung an die Originalobjekte propagiert werden, aus denen das geänderte Objekt entstanden ist. Dazu müssen die Objekte bestimmt werden, aus denen ein zu änderndes Objekt entstanden ist, und die Änderung dann ebenfalls auf diesen Objekten ausgeführt werden; dies wird von TraFIC erledigt. Weil alle Daten in der Föderationsdatenbank ausschließlich temporäre Kopien von Objekten in den Komponentendatenbanken sind, auf die nur die Transaktion zugreifen kann, die diese Kopien angelegt, hat, sind keine besonderen Maßnahmen zur Gewährleistung von Isolation auf diesen Daten notwendig. Ebenso müssen diese Daten nicht in die Atomaritätsbetrachtungen eingebunden werden, da sie nach Abschluss der Transaktion wieder gelöscht werden.

Abbildung 4.7 zeigt den Ablauf einer Modifikation in der FDB schematisch für den Fall, dass das zu ändernde Objekt unmittelbar aus Objekten einer einzigen Komponentendatenbank entstanden ist.

Ablauf:

- Der Föderationsserver schickt die Modifikation an TraFIC, statt sie direkt an den Föderationsdatenbankadapter zu schicken.
- TraFIC leitet die Modifikation an den Föderationsdatenbankadapter weiter.
- Der Föderationsdatenbankadapter führt die Modifikation in der Föderationsdatenbank aus.
- TraFIC bestimmt die Objekte, aus denen das geänderte Objekt entstanden ist, und die Datenbanken, in denen diese Objekte liegen, mit den Methoden, die VHDBS zur Verfügung stellt (siehe dazu [Weiß97]).
- TraFIC trifft nun auch für diese Modifikationen die erforderliche Maßnahmen zur Wahrung der Transaktionseigenschaft und leitet sie anschließend an die zuständigen Datenbankadapter weiter.
- Wenn es sich bei dem zuständigen Datenbankadapter um einen Komponentendatenbankadapter handelt, führt dieser die Modifikation aus und gibt einen Ergebniscode an TraFIC zurück.
- Wenn es sich bei dem zuständigen Datenbankadapter um den Föderationsdatenbankadapter handelt, etwa weil das ursprünglich geänderte Objekt aus anderen Objekten in der Föderationsdatenbank entstanden ist, wendet TraFIC den in diesem Abschnitt geschilderten Algorithmus zur Propagation erneut an.

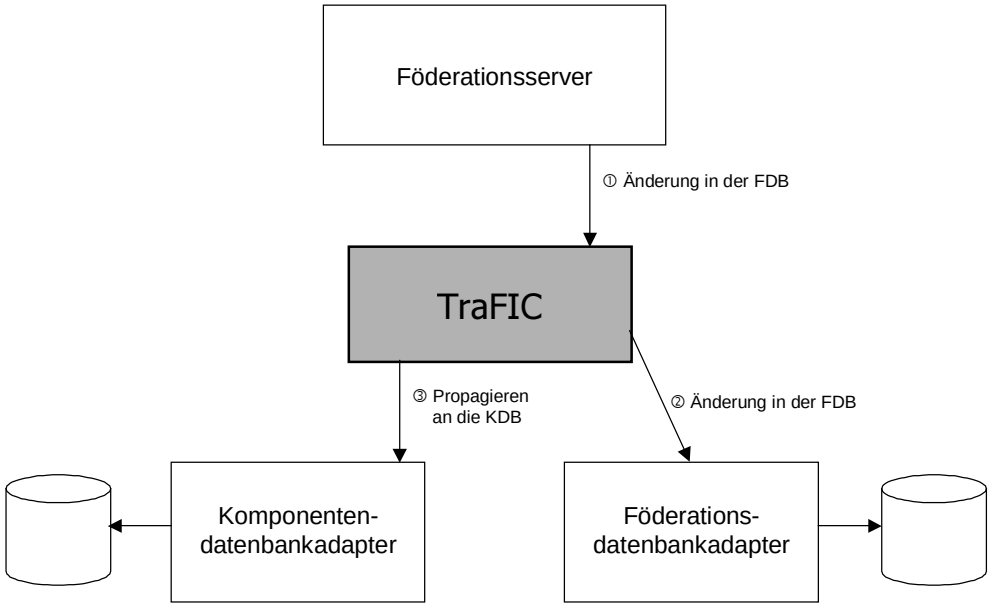

Abbildung 4.7 - Ablauf einer Modifikation in der FDB

Der Föderationsdatenbankadapter darf bei diesem Ansatz die Propagation also nicht mehr selbst ausführen. Weil dies das wesentliche Merkmal war, das den Föderationsdatenbankadapter von einem "normalen" Adapter für O_2-Datenbanken unterscheidet, genügt es nun, einen „gewöhnlichen“ Datenbankadapter als Föderationsserver zu nehmen.

4.4.2.5 Transparente Integration von Clients

Das Konzept des `Transaction`-Objektes, über das alle Methodenaufrufe gemacht werden, die im Kontext der Transaktion ausgeführt werden, ist zwar sehr elegant; es ist aber unschön, dass Methodenaufrufe außerhalb des Transaktionskontextes weiterhin direkt an den Föderationsserver geschickt werden. Es wäre wesentlich besser, wenn ein ähnliches Konzept wie im Object Transaction Service eingesetzt werden könnte: Ein Client muss lediglich explizit eine Transaktion beginnen und sie später explizit mit Commit beenden, kann aber ansonsten sowohl transaktionale als auch nichttransaktionale Methoden gleichartig aufrufen. Das Laufzeitsystem sollte selbständig transaktionale Methoden erkennen und sie über den Transaktionsmanager umleiten, d.h. die entsprechende Methode des `Transaction`-Objektes der aktuellen Transaktion aufrufen. Dies sollte vollständig transparent für den Client geschehen.

Um dies zu realisieren, wird das Konzept der *Smart Proxies* genutzt, das die verwendete CORBA-Implementierung Orbix als Erweiterung des CORBA-Standards anbietet. Orbix verwendet automatisch erzeugte Proxy-Objekte in jedem Client, die als „Stellvertreter“ für die wirklichen Objekte im Server auftreten. Sie nehmen die Methodenaufrufe entgegen, die der Client ausführt, leiten sie an das Originalobjekt weiter und geben anschließend die Ergebnisse an den Client zurück. Der Anwendungsprogrammierer hat zusätzlich die Möglichkeit, selbst Proxy-Objekte zu definieren (die sog. Smart Proxies), deren Methoden statt der Methoden der Orbix-eigenen Proxyobjekte aufgerufen werden.

Zunächst merkt sich das Laufzeitsystem nach erfolgreichem Aufruf von `BeginTransaction` das zurückgegebene `Transaction`-Objekt, das die aktuelle Transaktion repräsentiert. Dazu wird ein Smart Proxy für das `TransactionFactory`-Objekt von TraFIC eingesetzt, der das von Trafic zurückgegebene Objekt zwischenspeichert. Zusätzlich setzen wir auf der Clientseite Smart Proxies für alle Objektklassen ein, deren Methoden vom Client zur Durchführung von Modifikationen genutzt werden, insbesondere sind dies `dbTuple` und `dbCollection`. Methoden dieser Klassen, über deren Aufruf TraFIC nicht informiert werden muss, brauchen im Smart Proxy nicht neudefiniert zu werden, in diesem Fall wird automatisch die entsprechende Methode des Orbix-eigenen Proxies verwendet. Die für TraFIC interessanten Methoden werden in den Smart Proxies so neudefiniert, dass sie die entsprechende Methode des aktuellen `Transaction`-Objektes aufrufen statt der Methode des Originalobjektes.

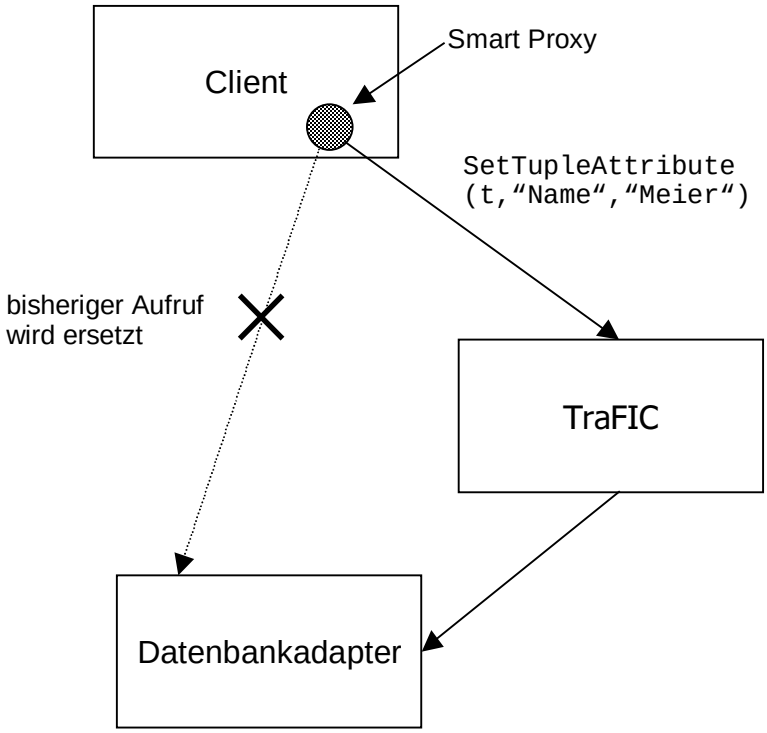

Abbildung 4.8 - Smart Proxies zur Einbindung von TraFIC

Abbildung 4.8 zeigt diesen Vorgang anhand eines Beispiels. Wenn der Client die Methode `set(“Name“,“Meier“)` des `dbTuple t` aufruft, wird von Orbix automatisch die entsprechende Methode des Smart Proxies aufgerufen, der zu `t` gehört. Dieser Smart Proxy nutzt die entsprechende Methode des aktuellen `Transaction`-Objektes, um TraFIC über den Methodenaufruf zu informieren. In diesem Fall wäre das die Methode `SetTupleAttribute(t,“Name“,“Meier“)`. TraFIC trifft nun wie üblich die erforderlichen Maßnahmen und leitet den Aufruf an das Originalobjekt weiter.

Der Client hat auch die Möglichkeit, über die Methode `Select` des Föderationsservers Anfragen und Modifikationen, die in OQL formuliert sind, auszuführen. Der Föderationsserver analysiert dann die OQL-Befehlssequenz und setzt sie schließlich in Anfragen und Modifikationen in den Datenbankadaptern um, wie sie im vorherigen Abschnitt behandelt wurden. Hier sind also zwei Teilprobleme zu untersuchen:

- **Problem 1** – Wie können Anfragen und Modifikationen, die der Client über die `select`-Methode an den Föderationsserver weitergibt, im Kontext einer Transaktion ausgeführt werden?
- **Problem 2** – Wie können die Anfragen und Modifikationen, die der Föderationsserver nach der Analyse der OQL-Befehle in den Datenbankadaptern ausführt, im Kontext einer Transaktion ausgeführt werden?

Zur Lösung des ersten Problems kann zunächst festgestellt werden, dass TraFIC an dieser Stelle noch nicht eingebunden werden muss. Es genügt, wenn TraFIC in die Ausführung der Anfragen und Modifikationen eingebunden wird, die der Föderationsserver nach Auswertung der OQL-Befehle durchführt. Es ist allerdings notwendig, dem Föderationsserver die Referenz des `Transaction`-Objektes der aktuellen Transaktion zu übergeben, damit er später TraFIC über dieses Objekt die Methodenaufrufe mitteilen kann. Dazu wird die folgende Vorgehensweise gewählt, die in Abbildung 4.9 graphisch verdeutlicht wird:

- Das Interface des Föderationsservers wird um eine weitere Methode `SelectTA` erweitert, die die gleichen Parameter wie `Select` sowie zusätzlich eine Referenz des `Transaction`-Objektes der aktuellen Transaktion erhält:

```
dbCollection SelectTA(in string OQLString, in boolean noDefine,
                      in Trafic::Transaction t)
             raises (dbError);
```

- Im Client wird ein Smart Proxy des Föderationsserverobjektes installiert, der alle Aufrufe der Methode `Select` abfängt. Zusammen mit den übrigen Parametern wird das aktuelle `Transaction`-Objekt an die Methode `SelectTA` des Föderationsservers übergeben.
- Die Implementierung der Methode `SelectTA` im Föderationsserver merkt sich dieses Objekt, indem sie es dem aktuellen Thread zuordnet, und ruft dann die Originalmethode `Select` des Föderationsservers auf. Wenn dabei eine Exception ausgelöst werden sollte, wird sie an den Client zurückgegeben.

Dieser Ansatz hat wieder den Vorteil, dass keine Änderungen in den bestehenden Clients notwendig sind.

Das zweite Problem lässt sich auf das bereits gelöste Problem zurückführen, wie direkte Methodenaufrufe der Clients an TraFIC gemeldet werden können. Da der Föderationsserver die gleichen Methodenaufrufe wie die Clients nutzt, können auch die gleichen Smart Proxies verwendet werden. Hinzu kommt noch ein Smart Proxy für jedes Datenbankadapterobjekt, auf dem der Föderationsserver die Methode `Select` aufruft, um Anfragen durchzuführen.

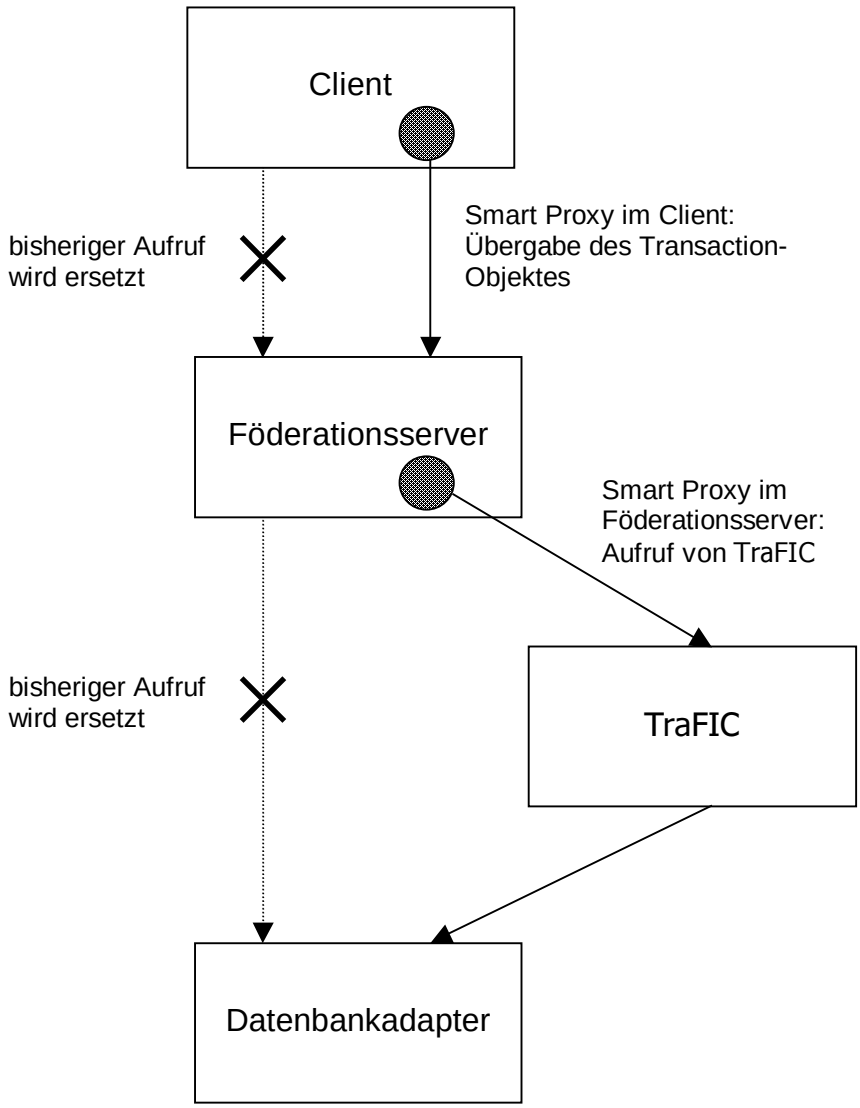

Abbildung 4.9 - Verbinden von TraFIC und Föderationsserver

4.4.3 Anbindung der Datenbankadapter an TraFIC

4.4.3.1 Implementierung als OTS-Server

Die Datenbankadapter sind in der OTS-Terminologie Server, die eine Ressource bereitstellen (nämlich die Datenbank); daher müssen sie als OTS-Server implementiert werden. Zusätzlich muss berücksichtigt werden, dass keine SQL-Kommandos zum Auf- und Abbau der Verbindung zur Datenbank genutzt werden dürfen, sondern der gesamte Verbindungsauf- und abbau ausschließlich über den OTS geschehen muss. Daher müssen insbesondere alle

```
EXEC SQL CONNECT ...
```

aus dem Quellcode entfernt werden, außerdem alle

```
EXEC SQL COMMIT WORK;
```

und

```
EXEC SQL ROLLBACK WORK;
```

Besonderheiten bei Oracle:

Wegen einer Beschränkung durch das XA-Interface von Oracle kann in einem Oracle-Datenbankadapter immer nur beim Starten des Servers die Benutzerkennung angegeben werden, unter der die Zugriffe auf die Datenbank durch diesen Server erfolgen sollen. Der OTS verbindet sich beim ersten Zugriff des Servers auf die Datenbank mit der Kennung, die bei der Registrierung der XA-Resource übergeben wird. Es ist nicht vorgesehen, zu einem späteren Zeitpunkt eine andere Verbindung explizit zu öffnen. Dadurch ist es insbesondere nicht möglich, dass mehrere Benutzer mit dem gleichen Datenbankadapter arbeiten und dabei zum Zugriff auf die Datenbank ihre eigene Benutzerkennung verwenden. Sie müssen statt dessen unter der im Datenbankadapter jeweils vorgegebenen Benutzerkennung arbeiten. Für die Zwecke dieser Arbeit ist dies ausreichend. Für einen praktischen Einsatz des Systems kann man für jeden möglichen Benutzer einer Datenbank einen eigenen Datenbankadapter instantiieren, der sich beim Start mit der Benutzerkennung dieses Benutzers bei der Datenbank anmeldet. Dies lässt sich einfach realisieren, indem dem Datenbankadapter beim Start ein Parameter übergeben wird, der den Namen des gewünschten Benutzers enthält. Zum Beispiel kann man dem Datenbankadapter, der die Zugriffe des Benutzers „`scott`“ auf die Datenbank „`oracle`“ ausführt, den Servernamen „`oracle_scott`“ geben.

Besonderheiten bei O_2:

Das XA-Interface von O_2 ist weder multi-session- noch multi-thread-fähig. Das bedeutet, dass immer nur eine XA-Transaktion auf O_2 zugreifen darf; erst nach deren Ende kann die nächste Transaktion über das XA-Interface starten. Aus diesem Grund eignet sich O_2 nicht für einen ernsthaften Einsatz in verteilten Umgebungen.

4.4.3.2 Propagieren des OTS-Transaktionskontextes zwischen TraFIC und Adapter

Für die Weitergabe des OTS-Transaktionskontextes zwischen Client und Server sind in OTS zwei verschiedene Möglichkeiten vorgesehen, deren Anwendbarkeit für TraFIC in diesem Abschnitt diskutiert wird.

Möglichkeit 1 – implizite Propagierung

Die in OTS vorgesehene Standardmethode, den OTS-Transaktionskontext bei transaktionalen Methodenaufrufen vom Client zum Server zu transportieren, ist die implizite Propagierung. Um das hier zu tun, müssen lediglich das Interface der Datenbankadapter selbst und alle Interfaces, die sie implementieren (`dbObject`, `dbCollection`, ...), vom Interface `TransactionalObject` abgeleitet werden.

In diesem Fall ist es aber nicht mehr möglich, Methoden der von den Datenbankadaptern implementierten Interfaces außerhalb des Kontextes einer OTS-Transaktion aufzurufen, da dann alle Methodenaufrufe innerhalb einer Transaktion geschehen müssen. Insbesondere bedeutet dies, dass der Föderationsserver selbst keine Methoden von `dbObject`, `dbCollection` und den übrigen Interfaces benutzen dürfte bzw. alle diese Methoden über TraFIC umgeleitet werden müssten. Weil der Föderationsserver diese

Interfaces auch für interne Operationen verwendet, bedeutet dies eine zu starke Einschränkung, so dass hier eine andere Alternative gewählt werden muss.

Möglichkeit 2 – explizite Propagierung

Die andere Möglichkeit, die in OTS vorgesehen ist, ist es, den OTS-Transaktionskontext explizit als Parameter der Methoden zu spezifizieren, die im Kontext von Transaktionen benutzt werden sollen. Dies hat den Vorteil, dass nur einige Methoden eines Interfaces transaktional sein können, während die übrigen auch außerhalb von Transaktionen verwendet werden können.

Diese Möglichkeit wird daher in TraFIC gewählt. Die Interfaces, die von den Datenbankadaptern implementiert werden, müssen dazu so geändert werden, dass bei allen Methoden zur Anfrage- und Modifikationsausführung zusätzlich ein Parameter `CosTransactions::Control_ptr` übergeben wird, der den aktuellen OTS-Transaktionskontext enthält. Um die Implementierung einfach zu halten, fügen wir dazu explizit neue Methoden zu den Interfaces hinzu, die den um den Suffix „TA“ ergänzten Namen der Originalmethoden haben. Beispielhaft hier ein Ausschnitt aus dem geänderten `DBAdapter`-Interface:

```
interface DBAdapter
{
  dbCollection selectTA(in OQLSequence condition,
                        in CosTransactions::Control control)
               raises(dbError);
}
```

Am Anfang jeder dieser Methoden muss dann eine entsprechende OTS-Transaktion hergestellt werden (Zur Erklärung der einzelnen Methodenaufrufe siehe [Iona99b]). Nachdem dieses geschehen ist, wird die Originalmethode des Datenbankservers aufgerufen, die dann im Kontext dieser Transaktion ausgeführt wird:

```
OpTA(param,CosTransactions::Control_ptr control)
{
  CosTransactions::Coordinator_var coord;
  coord = control->get_coordinator();

  CosTransactions::PropagationContext_var context;
  context = coord->get_txcontext();

  CosTransactions::TransactionFactory_var factory;
  factory = CosTransactions::TransactionFactory::\
            _bind(„TransactionFactory“);

  control = factory->recreate(context);

  CosTransactions::Current_var current =
                               CosTransactions::Current::IT_create();
  current->resume(control);

  Op(param); // call original operation

  CosTransactions::Control_var control2;
  control2 = current->suspend();
}
```

4.4.3.3 Ergänzungen im Interface

Das Interface der Datenbankadapter wird um eine Methode erweitert, mit der man explizit den Isolation Level für die aktuelle Transaktion setzen kann:

```
void setIsolationLevelTA(in IsolationLevel l,
     in CosTransactions::Control control) raises(dbError);
```

Der Parameter IsolationLevel ist dabei ein Aufzählungstyp, der die unterstützten Isolation Levels enthält:

```
enum IsolationLevel
  {
    ReadUncommitted,
    ReadCommitted,
    RepeatableRead,
    SnapshotIsolation,
    Serializable
  };
```

Diese Methode wird von den Implementierungen der Strategien von TraFIC verwendet. Zur Zeit werden nur `SnapshotIsolation` und `Serializable` benutzt.

Zusätzlich werden auch Methoden zum Interface der Datenbankadapter hinzugefügt, die den Ticketzugriff realisieren. Im einzelnen sind dies `TicketAccessRW`, die das Ticketobjekt in der Datenbank um zwei erhöht und den neuen Wert zurückgibt, und `TicketAccessRO`, die das Ticketobjekt nur liest und den um eins höheren Wert zurückgibt. Beide Methoden werden von den Tickettechniken eingesetzt, wobei auch in Datenbanken, die nicht Snapshot Isolation gewährleisten, das Ticket um zwei erhöht wird.

In relationalen Datenbanken sieht das Erhöhen des Tickets etwa so aus:

```
update ticket set value=value+2;
select value from ticket;
```

In objektorientierten Datenbanken wie O_2 implementiert man datenbankseitig eine Klasse `Ticket`, die eine Methode `TicketAccess` hat, um das Ticket zu erhöhen und den neuen Wert zurückzuliefern:

```
class Ticket
{
  private:
  int value;

  public:
  int Ticket::TicketAccess()
  {
    value=value+1;
    return value;
  }
};
```

4.4.4 Behandlung von Ausfällen von VHDBS-Komponenten

Im Zusammenhang mit der Diskussion der Integration von TraFIC in VHDBS muss auch die Frage betrachtet werden, wie TraFIC auf den Ausfall von VHDBS-Komponenten reagieren muss, um insbesondere die Atomarität von föderativen Transaktionen sicherzustellen. Dabei geht es vorrangig darum, wie Ausfälle erkannt werden; dann ist es konzeptionell naheliegend, jede aktive Transaktion, die die ausgefallene Komponente

benutzt hat, zurückzusetzen, da die einzelnen Komponenten keine transparente Recovery unterstützen. Um das Zurücksetzen von Transaktionen kümmert sich der Restartmanager, der in Abschnitt 4.5.5 beschrieben wird, zusammen mit dem Logmanager (siehe Abschnitt 4.5.4). Der Ausfall von TraFIC selbst wird hier nicht behandelt; er wird wie ein Neustart des Systems behandelt, nach dem wieder der Restartmanager Transaktionen zurücksetzt, die zum Zeitpunkt des Ausfalls aktiv waren.

Die Komponenten von VHDBS, deren Ausfall Auswirkungen auf den Betrieb von TraFIC hat, sind die Datenbankadapter, der Föderationsserver und die Clients von VHDBS.

4.4.4.1 Ausfall eines Datenbankadapters

Der am einfachsten zu behandelnde Fall ist der Ausfall eines Datenbankadapters oder der zugehörigen Komponentendatenbank. In beiden Fällen erhält TraFIC bei der Ausführung einer Methode eines Objekts im Datenbankadapter eine Fehlermeldung. Als Reaktion darauf kann TraFIC entweder die Transaktion abbrechen und dem Client eine `TRANSACTION_ROLLEDBACK`-Exception zurückgeben, oder dem Client eine Fehlermeldung zurückmelden; weitere Maßnahmen zur Fehlerbehebung, insbesondere das Abbrechen der Transaktion, fallen dann in den Aufgabenbereich des Clients.

4.4.4.2 Ausfall eines Clients

Der Ausfall eines Clients unterscheidet sich darin grundsätzlich vom Ausfall eines Datenbankadapters, dass TraFIC nicht darüber informiert wird, wenn z.B. die Rückgabe eines Ergebnisses an den Client scheitert, weil dieser in der Zwischenzeit ausgefallen ist. Ausfälle des Partners bei der Ausführung einer Operation können in Orbix nämlich auf einfache Art und Weise (d.h. durch das Abfangen einer Exception) nur von Clients und nicht von Servern festgestellt werden. Das ist natürlich insbesondere dann ein großes Problem, wenn der Client während der Ausführung des Commits einer Transaktion ausfällt. TraFIC darf in diesem Fall natürlich kein Commit ausführen, sondern muss die Transaktion abbrechen.

Zur Lösung dieses Problems bietet Orbix einen sog. `IOCallback` an, dessen Methode `CloseFD` aufgerufen wird, wenn eine Verbindung zu einem Client oder Server abbricht. Leider kann man hier aber nur die Nummer der Datei erhalten, die der Verbindung zugeordnet ist, und keine Informationen über den ausgefallenen Client bzw. Server, und schon gar keine über die beteiligten Transaktionen.

Um dies zu realisieren, verwaltet TraFIC selbst eine Tabelle, in der Verbindungsinformation (also Dateinummern) und Transaktionen einander zugeordnet werden. Zur Verwaltung dieser Tabelle setzen wir ein Filterobjekt im TraFIC-Server ein, das alle eingehenden Methodenaufrufe abfängt. Filter sind orbixspezifische Erweiterungen des CORBA-Standards, die eingesetzt werden, um an einem Orbix-Server eintreffende Methodenaufrufe zu erkennen und ggf. vorab zu verarbeiten, bevor die eigentliche Methode im Server aufgerufen wird. Betreffen die abgefangenen Methodenaufrufe ein `Transaction`-Objekt, bestimmt der Filter mit einer orbixspezifischen Methode die Dateinummer der Verbindung und trägt in der Tabelle ein, dass über diese Verbindung eine Operation im Kontext der Transaktion ausgeführt wurde. Der Eintrag einer Transaktion wird erst wieder entfernt, wenn das `Transaction`-Objekt gelöscht wird; das geschieht mittels `DropTransaction()` erst nach dem Ende der Transaktion.

Fällt nun ein Client aus, wird TraFIC darüber durch den `IOCallback` informiert. Anhand der Dateinummer kann nun in der Tabelle nachgesehen werden, welche Transaktionen über diese Verbindung abgewickelt wurden, diese können dann zurückgesetzt werden.

Abbildung 4.10 zeigt die Erweiterung der Architektur von TraFIC, um Ausfälle von Clients erkennen und behandeln zu können. In dem dargestellten Beispiel hat der Client über die Verbindung mit der Dateinummer 8 die Transaktion mit der ID 1 begonnen und dann über den Föderationsserver eine Operation im Kontext dieser Transaktion abgesetzt, die in TraFIC über die Verbindung mit der Dateinummer 6 ankommt. Außerdem hat der gleiche Client eine weitere Transaktion mit der ID 2 begonnen und einen direkten Methodenaufruf im Kontext dieser Transaktion durchgeführt.

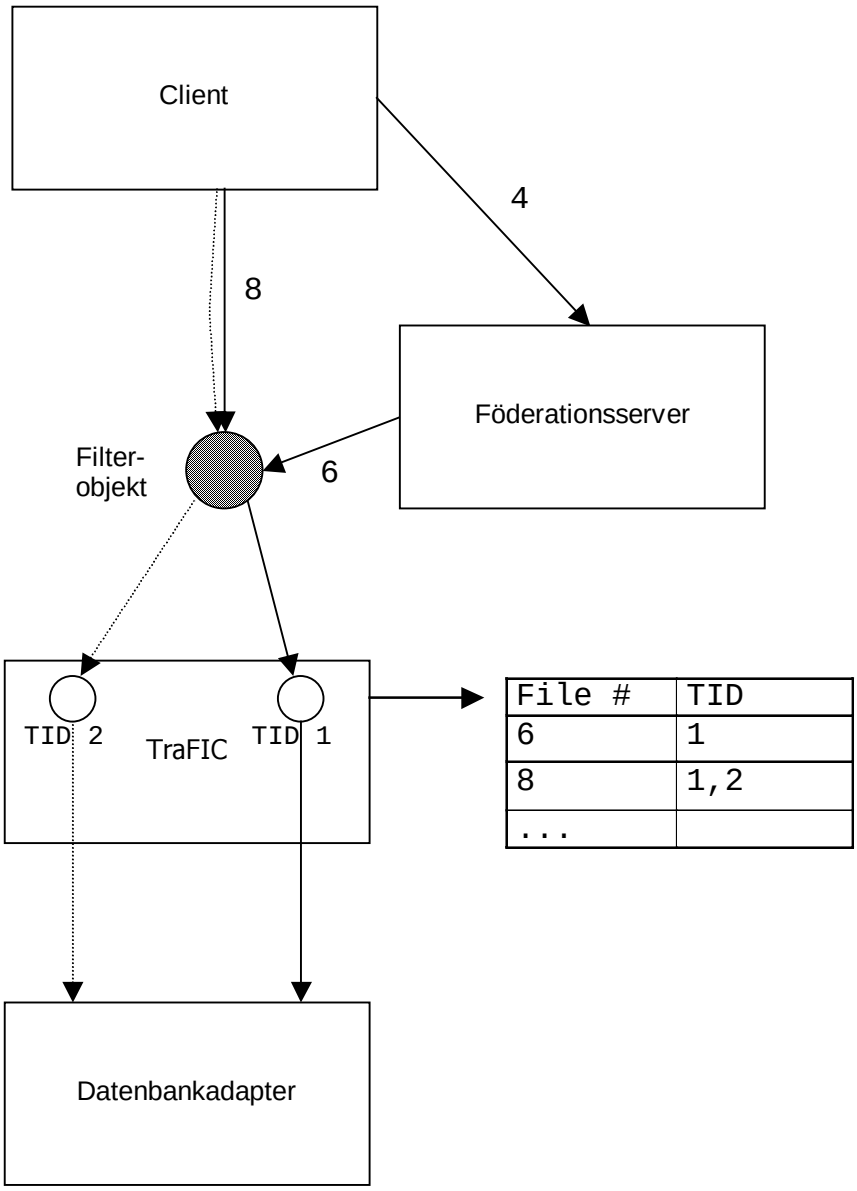

Abbildung 4.10 - Ergänzung der Architektur zum Erkennen von Client-Ausfällen

4.4.4.3 Ausfall des Föderationsservers

Der Föderationsserver ist für TraFIC lediglich ein besonderer Client, für den grundsätzlich keine Sonderbehandlung notwendig ist. Es ist aber sinnvoll, auf den Ausfall des Föderationsservers nicht mit dem Abbruch der Transaktionen zu reagieren, in deren Kontext er Operationen ausgeführt hat.

Da Transaktionen ausschließlich von VHDBS-Clients und nie vom Föderationsserver selbst begonnen und beendet werden, kann die Verantwortung für den Abbruch einer Transaktion bei Ausfall des Föderationsservers vollständig auf den Client verlagert werden, da dieser eine entsprechende Fehlermeldung vom Orbix-Laufzeitsystem erhält. Der Client kann nun selbst entscheiden, ob er mit einer ggf. neu gestarteten Instanz des Föderationsservers weiterarbeiten oder die Transaktion abbrechen will.

Die Realisierung dieser Variante wird aber dadurch erschwert, dass es in einem Orbix-Filter nicht möglich ist, den Ursprung eines eintreffenden Requests herauszufinden. Daher kann nicht unterschieden werden, ob es sich bei einer abgerissenen Verbindung um eine Verbindung zum Föderationsserver oder zu einem Client handelt. Wenn der Ausfall des Föderationsservers gesondert behandelt werden soll, sind also weitere Maßnahmen notwendig. Eine mögliche Lösung ist, zu jeder bisherigen Interfacemethode von TraFIC eine neue hinzuzunehmen, die nur vom Föderationsserver verwendet wird. Wird im Filter ein Aufruf einer solchen Methode erkannt, ist klar, dass die zugehörige Verbindung zum Föderationsserver gehört. Wegen des für diese Arbeit geringen Nutzens, dem ein hoher Implementierungsaufwand gegenübersteht, wurde diese Lösung nicht implementiert.

4.5 Mechanismen der Transaktionsverwaltung

In diesem Abschnitt werden allgemeine Mechanismen von TraFIC dargestellt, die von den übergeordneten Strategien zur Gewährleistung von Atomarität und Serialisierbarkeit genutzt werden. Es gibt dabei Mechanismen, die von mehreren Strategien benutzt werden können, und Mechanismen, die exklusiv von einer einzigen Strategie verwendet werden.

Die folgenden Mechanismen stellen allgemeine Funktionalitäten bereit, die von mehreren Strategien angewendet werden:

- Ein Satz von Datenstrukturen und Methoden *verwaltet Prädikate*, die konservative Approximationen der Objektmengen beschreiben, auf die Transaktionen zugegriffen haben. Er enthält außerdem Methoden, um *Überdeckungen dieser Prädikate* effizient zu erkennen und somit feststellen zu können, ob zwei Transaktionen einen Konflikt haben.
- Der *Deadlock-Erkenner* erkennt zyklische Blockierungen globaler Transaktionen und macht Vorschläge zu ihrer Behebung, z.B. durch die Auswahl einer geeigneten Opfertransaktion, die abgebrochen werden soll. Er verwaltet dazu tatsächliche oder angenommene Konflikte zwischen globalen Transaktionen in einem Konfliktgraphen und erkennt globale Verklemmungen durch einen Zyklus in diesem Graphen.

Die graphbasierte Strategie zur Gewährleistung von globaler Serialisierbarkeit und die Tickettechniken verwenden den folgenden Mechanismus:

- Der *Serialisierungsgraph* verwaltet in einem gerichteten Graphen zum einen die Ticketwerte der globalen Transaktionen in den verschiedenen lokalen Datenbanken, zum anderen den globalen OSI-MVSG. Er erkennt nicht serialisierbare Ausführungen durch einen Zyklus in diesem Graphen.

Das Mehrschichtentransaktionsverfahren nutzt die folgenden Mechanismen:

- Der *Sperrmanager* verwaltet die semantischen Sperren der Objekte einer Komponentendatenbank, insbesondere Prädikatsperren und Methodensperren. Die Erkennung der Blockierung einer globalen Verklemmung Transaktionen wurde in den Deadlock-Erkenner ausgelagert, weil es neben zyklischen Sperrkonflikten auf der globalen Ebene durch indirekte Konflikte in den lokalen Datenbanken auch zu anderen Typen globaler Blockierungen kommen kann, die sogar dann auftreten können, wenn auf der globalen Ebene kein sperrendes Verfahren eingesetzt wird.
- Der *Logmanager* führt Buch über bereits beendete Subtransaktionen von globalen Transaktionen und die dazugehörigen Kompensationsaktionen. Im Falle eines Fehlers von TraFIC mit nachfolgendem Neustart wird diese Information vom
- *Restartmanager* benutzt, um die bereits beendeten Subtransaktionen von zum Fehlerzeitpunkt aktiven Transaktionen zurückzusetzen.

Die optimistische Strategie zur Gewährleistung globaler Snapshot Isolation verwendet den folgenden Mechanismus:

- Die *OSI-Tabelle* verwaltet Prädikate, die Approximationen der Objektmengen beschreiben, auf die Transaktionen zugegriffen haben, und erkennt mit den in Theorem 3.10 aufgestellten Regeln Verletzungen der Snapshot-Isolation-Eigenschaft des globalen Schedules.

4.5.1 Verwaltung von Prädikaten und Überdeckungstests

Einige Mechanismen von TraFIC, insbesondere die Sperrmanager, der Serialisierungsgraph und die OSI-Tabelle, verwalten Prädikate, die Approximationen der Objektmengen beschreiben, auf die globale Transaktionen zugegriffen haben. Außerdem müssen sie prüfen, ob verschiedene Transaktionen mit den gleichen Objekten gearbeitet haben; die dann folgenden Maßnahmen sind vom jeweiligen Mechanismus abhängig. Es ist daher naheliegend, die Funktionalität zum Verwalten von Prädikaten und zum Prüfen, ob sich zwei Prädikate überdecken, in ein Modul zu stecken, das von den genannten Mechanismen genutzt wird. Dadurch lässt sich eine Menge Implementierungsaufwand sparen. Wir beschreiben nun zunächst interne Datenstrukturen, die zur Beschreibung der Daten in den Komponentensystemen notwendig sind, und erläutern dann den Umgang mit Prädikaten.

Interne Strukturen

Die Mechanismen verwenden intern unter anderem die Klassen `lmType` (zur Verwaltung von Typen), `lmRepository` (Datenbankrepositories) und `lmVHDBSRepository` (Spiegelrepositories). Diese Klassen werden nun im einzelnen vorgestellt. Dabei wird aus Platzgründen darauf verzichtet, den Quellcode ausführlich abzubilden.

Zu jedem Repository in einem Datenbankadapter wird eine Instanz der Klasse `lmRepository` angelegt, die in einer verketteten Liste mit Wurzel `lmRepositories` eingetra-

gen wird. Eine Instanz dieser Klasse entspricht also einer Relation in einer relationalen Komponentendatenbank bzw. einem Repository in einer objektorientierten Datenbank. In dieser Instanz wird der Name gespeichert, mit der das Repository in der Datenbank angesprochen werden muss. Prädikate werden immer auf Ebene dieser Repositories verwaltet.

Zu einem solchen Datenbankrepository kann es ein oder mehrere VHDBS-Spiegelrepositories geben, die auf es abgebildet werden. Diese Spiegelrepositories werden durch die Klasse `lmVHDBSRepository` repräsentiert. Eine Instanz dieser Klasse enthält einen Zeiger auf die Instanz von `lmRepository` des Datenbankrepositories.

Die Unterscheidung von Datenbank- und Spiegelrepositories ist deshalb notwendig, weil die Verwaltung der Prädikate auf der Ebene der Datenbankrepositories geschehen muss. Andernfalls könnten gleichzeitig zwei Zugriffe auf das gleiche Datenbankrepository über verschiedene Spiegelrepositories erfolgen, ohne dass sich die zugehörigen Prädikate überdecken würden, da sie sich auf verschiedene Spiegelrepositories bezögen. In der `OQLSequence`, die der Föderationsserver an einen Datenbankadapter übergibt, ist nur der Name des zugegriffenen Spiegelrepositories enthalten, daher müssen beide Konzepte im Sperrmanager berücksichtigt werden.

Die Typen der benutzten Repositories werden als Instanzen der Klasse `lmType` verwaltet. Sie beinhaltet neben dem Namen eine Liste der Attribute, wenn es sich um einen Tupeltyp handelt, bzw. einen Zeiger auf die `lmType`-Instanz des Komponententyps, wenn es sich um einen Mengentyp handelt. Attribute werden durch die Klasse `lmAttribute` repräsentiert, die neben dem Namen des Attributs seinen Typ, dargestellt als Aufzählungstyp `lmAttrType`, und Information darüber enthält, ob es sich um ein Schlüsselattribut handelt.

Repräsentation boolescher Ausdrücke

Prädikate werden verwendet, um die Objekte zu charakterisieren, die selektiert oder modifiziert werden sollen. Als elementare Ausdrücke werden dabei Vergleiche von Objektattributen mit Skalaren oder, im Falle von Joins, Vergleiche von Objektattributen untereinander verwendet. Vergleiche von Attributen untereinander schränken dabei die zugegriffenen Objekte nicht direkt, sondern nur indirekt ein, so dass wir sie nicht weiter berücksichtigen und konservativ annehmen, dass sie keine Einschränkung vornehmen. Die übrigen Vergleiche werden immer in der Form

$$\texttt{untere\_grenze} \leq \texttt{a.wert} \leq \texttt{obere\_grenze}$$

spezifiziert, gegebenenfalls mit vorhergehendem `NOT`-Operator. Dadurch wird ein Intervall angegeben, in dem der Wert des Attributs liegt. Die beiden Grenzen können auch den Wert $-\infty$ bzw. ∞ haben. Sind beide Grenzen maximal, wird der gesamte Wertebereich des Attributs getroffen. Durch den einheitlichen Aufbau lässt sich bei der späteren Ausführung sehr effizient testen, ob ein Vergleich erfüllt ist oder nicht. Es lässt sich einfach zeigen, dass jede andere Art von Vergleichsoperation in diese Gestalt gebracht werden kann. Beispielsweise lässt sich

$$\texttt{a.wert<7}$$

konvertieren zu

$$(-\infty \leq \texttt{a.wert} \leq 7) \texttt{ and not } (\ 7 \leq \texttt{a.wert} \leq 7).$$

Ein solcher elementarer Vergleich wird durch eine Instanz der Klasse `lmLiteral` repräsentiert. Nichtnumerische Datentypen können analog verarbeitet werden, wenn man einen geeigneten Vergleichsoperator nimmt, z.B. für Strings lexikographische Ordnung und für Datumsangaben oder Zeitpunkte die zeitliche Reihenfolge.

Anschließend wird das Prädikat in disjunktive Normalform umgewandelt, indem die üblichen Regeln (z.B. deMorgan-Formeln, Ausklammern) angewendet werden. In der so entstandenen Normalform kommen potentiell Attribute von Objekten aus mehreren Relationen vor. Wir verwalten aber die Prädikate für jede Relation getrennt, also werden anschließend die Teilprädikate für die einzelnen Relationen bestimmt. Dazu wird für jede Relation der betreffende Anteil aus jedem Monom der DNF extrahiert; die Zwischenergebnisse werden dann durch `OR` zu einem neuen Prädikat verbunden.

Als Ergebnis erhält man für jede Relation ein Polynom, das durch eine Instanz der Klasse `lmPolynom` repräsentiert wird und aus einer Disjunktion von Monomen besteht, die von Instanzen der Klasse `lmMonom` repräsentiert werden, die in einer Liste linear verkettet sind. In jedem solchen Monom müssen nicht alle Attribute eines Typs eingeschränkt werden. Attribute, die nicht im Monom vorkommen, können jeden möglichen Wert haben; wir ergänzen daher für sie einen Vergleich, der den gesamten Wertebereich des Attributs trifft, also z.B. in der Form

$$-\infty \leq \texttt{a.wert} \leq \infty$$

Prüfen der Überdeckung zweier Prädikate

Um die Überdeckung zweier solcher durch Polynome dargestellten Prädikate zu prüfen, gehen wir schrittweise vor und prüfen die Überdeckung der einzelnen Monome. Haben zwei beliebige Monome der beiden Polynome eine Überdeckung, so auch die beiden Polynome.

Um die Überdeckung zweier Monome zu prüfen, wird die Überdeckung der einzelnen Intervalle geprüft, die durch jeden elementaren Vergleich spezifiziert wurden. Weil wir für Attribute, die nicht im Prädikat vorkommen, den gesamten Wertebereich eingesetzt haben, finden wir für jedes Attribut einen Vergleichspartner. Die beiden Monome haben genau dann keine Überdeckung, wenn es mindestens bei einem Attribut keine Überdeckung der Intervalle gibt.

4.5.2 Der Deadlock-Erkenner

Der Deadlock-Erkenner ist wie der Serialisierungsgraph ein eigenständiger Server, der das `DeadlockGraph`-Interface implementiert. Er wird bei der Mehrschichtenstrategie vom Sperrmanager benutzt, um zyklische Wartebeziehungen zwischen globalen und/ oder lokalen Transaktionen festzustellen.

Es gibt verschiedene Möglichkeiten, um globale Verklemmungen zu erkennen. Die einfachste ist sicher die timeout-basierte Variante, bei der die Ausführung von Anfragen und Modifikationen in den lokalen Datenbankadaptern mit Antwortzeitschranken versehen wird, nach deren Ablauf eine Blockierung angenommen wird. Es ist aber wegen Einschränkungen der verwendeten CORBA-Implementierung nicht ohne weiteres möglich, eine solche Technik in einen externen Mechanismus auszulagern: CORBA-Aufrufe, die an einen Server abgeschickt wurden, können vom Client nicht mehr selbst abgebrochen werden. Die einzige Möglichkeit, so etwas zu realisieren, ist es, im Clientprogramm selbst die maximale Zeit zu setzen, die auf die Antwort auf einen CORBA-

Aufruf gewartet wird. Wir verwenden daher im externen Deadlock-Erkenner eine graphbasierte Technik, um verteilte Verklemmungen zu erkennen und aufzulösen. Ambitioniertere Strategien zur Behandlung globaler Verklemmungen werden z.B. in [KKG99] beschrieben; in dieser Arbeit beschränken wir uns aus Aufwandsgründen auf den relativ einfachen, graphbasierten Ansatz.

Der vom Deadlock-Erkenner verwaltete Graph hat die globalen und lokalen Transaktionen als Knoten und Wartebeziehungen zwischen ihnen als Kanten. Starten und Beenden von Transaktionen sowie Anfang und Ende des Wartens einer Transaktion auf eine andere werden ihm von den Sperrmanagern mitgeteilt. Wenn er über eine neue Wartebeziehung informiert wird, trägt er sie in den Graphen ein und prüft, ob dadurch ein Zyklus entstanden ist. Wenn das der Fall ist, ist eine globale Verklemmung entstanden, die dem Sperrmanager durch das Auslösen der Exception `DeadlockGraph::CycleDetected` signalisiert wird.

Nicht alle Wartebeziehungen sind auf der globalen Ebene bekannt, lediglich direkte Konflikte zwischen globalen Transaktionen auf dem gleichen Objekt können erkannt werden, wenn ein globales Sperrprotokoll verwendet wird, etwa bei der Mehrschichtenstrategie. Wenn die Komponentensysteme ihre Wartebeziehungen nicht nach außen bekannt geben, kann man nicht feststellen, ob eine globale Transaktion wegen einer Sperre blockiert wird, die eine lokale Transaktion hält, oder lediglich eine Operation ausführt, die sehr lange benötigt. Beispielsweise könnte es sein, dass auf der globalen Ebene die globale Transaktion GTA_2 auf eine globale Sperre wartet, die eine andere globale Transaktion GTA_1 hält. In einem lokalen Datenbanksystem wartet GTA_1 selbst auf eine lokale Sperre, die die lokale Transaktion LTA hält, die ihrerseits auf eine Sperre wartet, die die globale Transaktion GTA_2 hält. Alle Konflikte sind dabei auf verschiedenen Objekten. Abbildung 4.11 zeigt diese Problematik, die lokalen Wartebeziehungen sind dabei gestrichelt eingezeichnet.

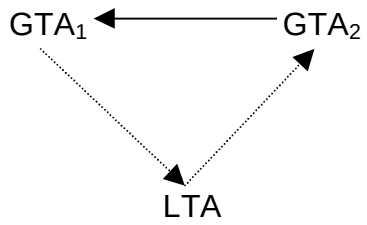

Abbildung 4.11 - Globale Verklemmung durch indirekte Wartebeziehung

Als mögliche Lösung für diese Problematik implementiert der Deadlock-Erkenner zusätzlich einen „potentiellen Wartegraphen“ (PWG). Dazu wird ein möglicher Konflikt zweier globaler Transaktionen in einem LDBS schon dann angenommen, wenn die beiden Transaktionen gleichzeitig in diesem LDBS aktiv sind. Die Verwendung dieser Alternative wird aber nicht empfohlen, da sie zu ungenau ist und zu viele Transaktionen unnötig abbricht. Statt dessen sollten zum Erkennen solcher Konflikte clientseitig die Timeoutmechanismen genutzt werden, die alle beteiligten Datenbanksysteme und die CORBA-Implementierung bieten.

4.5.3 Die Sperrmanager

Ein Sperrmanager verwaltet semantische Sperren für die Objekte einer Komponentendatenbank. Es gibt mehrere Sperrmanager im System, wobei immer ein Sperrmanager für die Sperren der Objekte eines Datenbankadapters zuständig ist. Dies ist möglich, da der Föderationsserver immer Teilanfragen an genau einen Datenbankadapter absetzt und der zuständige Sperrmanager daher eindeutig feststeht. Ebenso liegen zu ändernde Objekte immer in genau einer lokalen Datenbank.

Der Sperrmanager, der für den Datenbankadapter `A` zuständig ist, hat den Servernamen `LM4A`. Auf diese Weise kann jederzeit der richtige Server gefunden werden. Im folgenden wird die Implementierung eines solchen Sperrmanagers beschrieben.

Interface

Zur Anforderung einer Lesesperre wird dem Sperrmanager die OQL-Anweisung in Form einer `OQLSequence` übergeben, für die eine Sperre erworben werden soll. Daraus extrahiert der Sperrmanager dann die zu sperrenden Prädikate und fordert die notwendigen Sperren an. Zusätzlich gibt es Methoden zum Anfordern einer Schreibsperre vor dem Einfügen oder Löschen eines Objekts, zum Anfordern einer Schreibsperre vor der Änderung eines Attributs eines Objekts und zum Anfordern einer semantischen Sperre vor der Ausführung einer Methode, so dass das Interface des Sperrmanagers für globale Transaktionen folgendes Aussehen hat:

```
long AcquireGlobalQueryLock(in Transaction t, in OQLSequence action)
     raises (Deadlock);
long AcquireGlobalUpdateLock(in Transaction t, in dbTuple obj)
     raises (Deadlock);
long AcquireGlobalUpdateAttributeLock(in Transaction t,
                                      in dbTuple obj, in string attr)
     raises (Deadlock);
long AcquireGlobalMethodLock(in Transaction t, in dbTuple obj,
                             in string method)
     raises (Deadlock);
```

Zusätzlich gibt es analoge Interfacemethoden, um Sperren lokaler Transaktionen anzufordern und freizugeben, wobei dort statt eines `Transaction`-Objektes lediglich eine numerische Transaktions-ID übergeben wird. Diese Methoden sind allerdings im Prototypen nicht implementiert, weil der Aufwand für die Umleitung der Operationen der lokalen Transaktionen über die Sperrmanager (siehe Abschnitt 3.4.6) zu hoch gewesen wäre.

Interne Strukturen

Der Sperrmanager verwendet intern neben den Klassen, die von der Prädikatverwaltung bereitgestellt werden, unter anderem die Klassen `lmTransaction` (Transaktionen), `lmLock` (Sperren) und `lmLockMode` (Sperrkompatibilitäten). Diese Klassen werden nun im einzelnen vorgestellt. Dabei wird aus Platzgründen darauf verzichtet, den Quellcode ausführlich abzubilden.

Für jeden Repositorytyp, der durch die in Abschnitt 4.5.1 beschriebene Klasse `lmType` repräsentiert wird, werden in der Metadatenbank Informationen über die Kompatibilität der Methoden dieses Typs mit anderen Methoden sowie mit Lese- und Schreibsperren abgelegt. Diese Informationen werden in einer Liste von Instanzen der Klasse `lmLock-`

`Mode` gespeichert; ein Zeiger in der entsprechenden `lmType`-Instanz zeigt auf die erste dieser Instanzen.

Beispiel:

In einer Bankanwendung gebe es eine Klasse „Konto“, die die Methoden „GeldAbheben“ (GA), „GeldEinzahlen“ (GE) und „KontostandAbfragen“ (KA) habe. In der Metadatenbank würden dann die folgenden Informationen abgelegt:

Methode	Kompatible Methoden	Kompatibel mit Lesesperre auf „Konto“	Kompatibel mit Schreibsperre auf „Konto“
`GeldAbheben`	-[3]	nein	nein
`GeldEinzahlen`	`GE`	nein	nein
`KontostandAbfragen`	`KA`	ja	nein

Aus diesen Informationen kann der Sperrmanager die folgende Kompatibilitätsmatrix aufbauen:

	GA	GE	KA	RD	WR
GA	-	-	-	-	-
GE	-	+	-	-	-
KA	-	-	+	+	-
RD	-	-	+	+	-
WR	-	-	-	-	-

Informationen über Repositories, Typen, Typattribute, Methoden und Methodenkompatibilitäten werden beim Starten des Sperrmanagers durch die Methode `lmRepository::init` aus der Metadatenbank gelesen und in die internen Strukturen geschrieben. Dadurch werden zur Laufzeit ständige Zugriffe auf die Metadatenbank, die die Performance beeinträchtigen würden, vermieden. Der Nachteil dieses Ansatzes ist, dass jede Änderung des Typsystems einen Neuaufbau dieser lokalen Informationen notwendig macht.

Realisierung von Lesesperren

Lesesperren werden vor der Ausführung einer Anfrage angefordert, die als `OQLSequence` an den Sperrmanager übergeben wird. Der prinzipielle Aufbau einer solchen `OQLSequence` ist in Abbildung 4.12 gezeigt. Der Sperrmanager entnimmt ihr die Informationen über die Repositories, die an der Anfrage beteiligt sind, und merkt sich jeweils in einer Instanz der Klasse `lmQueryTypeInfo` das Repository (in Form eines Pointers auf die entsprechende Instanz von `lmRepository`) und den in der Anfrage dafür verwendeten Bezeichner. Der Bedingungsteil wird dann rekursiv durchlaufen, dabei wird ein boolescher Ausdruck erzeugt, der eine konservative Approximation der Objektmenge beschreibt, auf die während der Ausführung der Anfrage zugegriffen wird. Mit den Methoden zur Prädikatverwaltung, die in Abschnitt 4.5.1 vorgestellt wurden, wird aus diesem booleschen Ausdruck für jedes Repository, das in der Anfrage vorkommt, ein Po-

[3] Zwei Aufrufe von „GeldAbheben“ sind nicht kompatibel, wenn das Konto nicht beliebig weit überzogen werden darf.

lynom (repräsentiert durch eine Instanz der Klasse `lmPolynom`) erzeugt, das aus einer Disjunktion von Monomen besteht, die von Instanzen der Klasse `lmMonom` repräsentiert werden. Zusätzlich wird eine Instanz der Klasse `lmLock` erzeugt, die die angeforderte Sperre repräsentiert und eine Liste aller `lmPolynom`-Instanzen enthält, die während dieses Prozesses erzeugt wurden.

Jedes erhaltene Teilprädikat wird nun gesperrt, indem jedes seiner Monome gesperrt wird. Der folgende Algorithmus wird grundsätzlich auch bei der Anforderung von Schreib- und Methodensperren verwendet. Er wurde als Methode `lockLiteral(lmLiteral, Mode)` der Klasse `lmRepository` implementiert. Darin wird für alle auf dem Repository bereits gesperrten Monome geprüft,

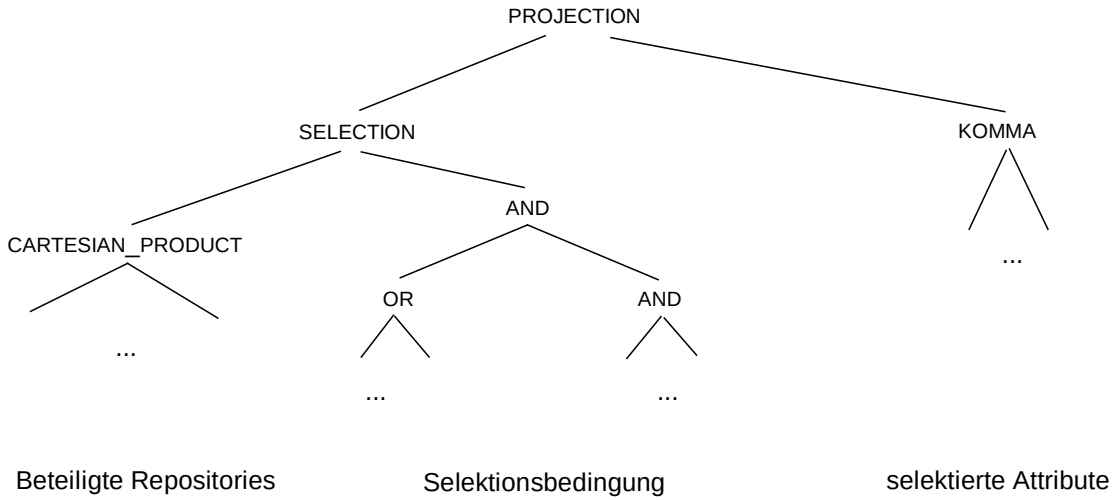

Abbildung 4.12 - Struktur einer Anfrage als OQLSequence

- ob es sich um eine eigene Sperre handelt, denn dann kann sofort weitergemacht werden,
- ob es sich um eine kompatible Sperre handelt – dazu bietet die Klasse `lmType` die Methode `lockCompatible`, die Verträglichkeit zweier Sperrmodi prüft,
- ob das zu sperrende Monom eine Überdeckung mit dem gesperrten hat, was mit den in Abschnitt 4.5.1 gezeigten Techniken realisiert wird.

Wenn ein Monom nicht gesperrt werden kann, muss der ausführende Thread warten, bis eine andere Transaktion die blockierende Sperre freigegeben hat. Dazu wird eine Instanz der Klasse `lmWaitingLiteral` erzeugt, die das zu sperrende Monom und die ID des Threads enthält. Diese Instanz wird in eine Liste in der `lmRepository`-Instanz eingetragen, die alle Threads enthält, die auf eine Sperre auf diesem Repository warten. Anschließend wird der Thread angehalten, bis er später nach Freigabe der blockierenden Sperre wieder aufgeweckt wird (siehe weiter unten). Danach beginnt er wieder von neuem mit dem Test auf Überdeckung mit allen gesperrten Monomen, weil es noch Überdeckungen mit anderen, gesperrten Monomen geben kann. Um Verklemmungen zu erkennen, wird vor dem Anhalten des Threads entweder ein Timeout vorgesehen, nachdem das Warten abgebrochen wird, oder ein Eintrag in den globalen Wartegraphen vorgenommen. Der Eintrag im Wartegraphen muss wieder entfernt werden, wenn die Transaktion wieder aufgeweckt wird.

Wenn die Sperre auf einem Monom gewährt werden kann, wird das Monom in die Liste der auf diesem Repository gesperrten Monome eingetragen und mit dem nächsten weitergemacht.

Sind alle Monome aller beteiligten Repositories erfolgreich gesperrt, kann die Ausführung der Anfrage erfolgen.

Realisierung von Schreibsperren

Das Sperren eines Objektes zum Löschen oder Einfügen erfordert es, den aktuellen Zustand des Objektes zu sperren, damit andere Transaktionen nicht vor dem Commit der ändernden Transaktion darauf zugreifen können. Als Prädikat formuliert, bedeutet das, das Monom

$$\texttt{Attribut}_1\texttt{=Objekt.Attributwert}_1\ \texttt{and ... and}\ \texttt{Attribut}_n\texttt{=Objekt.Attributwert}_n$$

auf dem Repository zu sperren, indem das Objekt liegt. Genau dieses implementiert die entsprechende Methode des Sperrmanagers. Die aktuellen Objektwerte werden dazu durch die VHDBS-Methoden besorgt. Das Sperren des Prädikats selbst erfolgt mit der oben beschriebenen Methode `lockLiteral`. Dies ist möglich, da das Prädikat bereits ein Monom ist, das nur Attribute eines Repositories enthält.

Zum Ändern des Werts eines Objektattributs darf dagegen nicht ausschließlich der aktuelle Wert gesperrt werden, da andere Transaktionen sonst den geänderten Wert sehen könnten. Um dieses zu vermeiden, wird im zu sperrenden Prädikat das gesamte Intervall des geänderten Attributs gesperrt.

Realisierung von Methodensperren

Methodensperren werden analog zu Schreibsperren implementiert. Lediglich beim Aufruf von `lockLiteral` wird statt des Schreibmodus der zu der Methode gehörende Sperrmodus angegeben.

Freigabe von Sperren

Sperren einer Transaktion werden erst freigegeben, wenn die Transaktion (mit Commit oder Abort) beendet wird. Globale Transaktionen geben ihre Sperren dann mit der Methode `FreeGlobalLocks` frei.

Wenn eine Sperre freigegeben wird, müssen alle Teilmonome dieser Sperre aus den Listen der beteiligten Repositories gelöscht werden. Außerdem müssen alle Transaktionen, die auf ein solches Monom warten, aufgeweckt werden. Dazu wird jedes freigegebene Monom mit den Monomen in der Liste mit den wartenden Monomen verglichen; gibt es eine Überdeckung, so muss die entsprechende Transaktion aufgeweckt werden.

Wenn der globale Deadlock-Erkenner verwendet wird, um Deadlocks zu erkennen, muss diesem außerdem das Ende der Transaktion signalisiert werden.

Threadarchitektur

Für jede Sperranforderung, die im Sperrmanager eingeht, wird ein neuer Thread erzeugt, der die Ausführung übernimmt. Wäre das nicht der Fall, würde bereits das Warten einer Transaktion den Sperrmanager blockieren. Es würde prinzipiell auch genügen, einen Thread für jede Transaktion zu haben, da eine Transaktion nie auf eine eigene Sperre warten muss.

Berücksichtigung lokaler Transaktionen

Die Implementierung des Sperrmanagers unterstützt keine lokalen Transaktionen. Um diese zu integrieren, müsste ein datenbankabhängiges Interface geschaffen werden, das die Aufrufe der lokalen Applikationen transparent abfängt und die notwendigen Sperren anfordert. Die technische Realisierung eines solchen Interfaces ist sehr aufwendig, insbesondere ist eine Analyse der Programmierschnittstelle des Datenbanksystems notwendig. Da die Auswirkungen von lokalen Transaktionen auf die Mehrschichtenstrategie bereits in [Scha96] ausführlich untersucht wurde, haben wir in dieser Arbeit darauf verzichtet.

Intern ist der Sperrmanager durchaus auf lokale Transaktionen vorbereitet, wenn das entsprechende Interface bereitgestellt wird, ist es relativ einfach möglich, Sperren lokaler Transaktionen zu verwalten.

4.5.4 Der Logmanager

Der Logmanager verwaltet ein Log, in dem ausschließlich Informationen über die Kompensationsaktionen für abgeschlossene Subtransaktionen von noch nicht beendeten globalen Transaktionen festgehalten werden. Eine Subtransaktion entspricht dabei immer genau einer Änderungsoperation oder einem Methodenaufruf einer globalen Transaktion. Die Logsätze müssen nicht über das Commit einer Transaktion hinaus aufbewahrt werden. Sie werden benötigt, um bereits mit Commit beendete Subtransaktionen bei Abbruch der Vatertransaktion zurückzusetzen. Dies ist in zwei Fällen erforderlich, entsprechend wird in diesen beiden Fällen lesend auf das Log zugegriffen: zum einen beim Abbruch einer globalen Transaktion (z.B. aufgrund der Auflösung einer Verklemmung) und zum anderen beim Neustart des föderierten Datenbanksystems aufgrund eines Systemfehlers.

Das Log muss in einem persistenten Speicher gehalten werden, der Systemabstürze überdauert. Insbesondere muss gewährleistet sein, dass die Kompensationsaktion einer Operation genau dann im Log steht, wenn die Operation selbst Änderungen in der stabilen Datenbank gemacht hat. Das Schreiben ins Log muss also als Teil der Subtransaktion ausgeführt werden. Es bietet sich an, entweder eine eigene persistente Datenstruktur im Filesystem abzulegen oder eines der Datenbanksysteme zu nutzen. Um den Aufwand zur Integration der Logoperationen in den Kontext der Subtransaktion, also im wesentlichen das Bereitstellen eines XA-Interfaces, in Grenzen zu halten, wählen wir die Variante mit dem Log in einem Datenbanksystem, das alle nötigen Voraussetzungen mitbringt.

Verwaltung des Logs:

Zu jeder Operation, die innerhalb einer Transaktion ausgeführt wird und die Datenbank potentiell verändert, muss eine Kompensationsaktion ins Log geschrieben werden. Für die Berechnung der Kompensationsaktion und das Auslösen ihres Schreibens ins Log ist die Implementierung der Mehrschichtenstrategie zuständig. Das Schreiben selbst wird durch den Logmanager ausgeführt. Dadurch wird erreicht, dass nur der Logmanager die Struktur des Logs kennen muss. Die Realisierung des Logs kann geändert werden, ohne die Implementierung der Strategie selbst anpassen zu müssen.

Die Implementierung des Logmanagers verwendet die beiden folgenden Relationen, um das Log zu realisieren:

```
CREATE TABLE TraficTransactions
             ( TID     INTEGER,
               State   INTEGER
               PRIMARY KEY (TID));

CREATE TABLE TraficLog
             ( TID      INTEGER,
               SID      INTEGER,
               EntryID  INTEGER,
               DBServer VARCHAR(50),
               UndoOp   VARCHAR2(250),
               PRIMARY KEY (TID,SID,EntryID),
               FOREIGN KEY (TID) REFERENCES TraficTransactions);
```

In der Relation `TraficTransactions` werden Informationen über die Transaktionen gehalten, die Aktionen ausgeführt haben. `TID` ist die eindeutige Transaktions-ID, die von TraFIC für jede Transaktion vergeben wird. `State` gibt an, ob die Transaktion aktiv ist (0) oder bereits beendet wurde (1).

In der Relation `TraficLog` werden die eigentlichen Kompensationsaktionen verwaltet. Ein Tupel dieser Tabelle entspricht dabei im Regelfall einer solchen Kompensationsaktion. `TID` ist wieder die Transaktions-ID, `SID` die laufende Nummer der Kompensationsaktion dieser Transaktion, die durch den Logmanager vergeben wird. In `UndoOp` wird die Kompensationsaktion als OQL-Anweisung gespeichert; ist der Text der Aktion zu lang für dieses Feld, wird sie auf mehrere Tupel mit gleicher TID und SID verteilt, die mit dem Eintrag `EntryID` durchnumeriert werden. `DBServer` schließlich enthält den Namen des Datenbankadapters, in dem die Kompensationsaktion ausgeführt werden muss.

Aktionen im laufenden Betrieb

Folgende Einträge in das Log werden im laufenden Betrieb vorgenommen:

- Bei BOT (Beginn der Vatertransaktion): kein Eintrag
- Bei BOS (Beginn einer Subtransaktion): kein Eintrag
- Operation einer Subtransaktion: Schreiben des Undo-Logsatzes im Kontext dieser Subtransaktion
- Bei EOS (Ende einer Subtransaktion): Commit des Schreibens des Undo-Logsatzes zusammen mit dem Commit der Subtransaktion (2PC); dies geschieht automatisch durch OTS.
- Bei EOT: (Ende der Vatertransaktion): Nach erfolgreichem Commit der Vatertransaktion werden die Undo-Logsätze nicht mehr benötigt, sie können also auf eine geeignete Weise gelöscht werden. Die beiden Möglichkeiten, die es dafür gibt, werden weiter unten diskutiert.

Im Interface des Logmanagers spiegeln sich diese fünf Zeitpunkte durch die folgenden fünf Methoden wider:

```
  void BeginTransaction(in Transaction t) raises(dbError);
  void BeginSubtransaction(in Transaction t, in long stid)
                           raises(dbError);
  void WriteUndoLogEntry(in Transaction t, in long stid,
       in string undoop, in RepServer kdb) raises(dbError);
  void EndSubtransaction(in Transaction t, in long stid)
                         raises(dbError);
```

```
void EndTransaction(in Transaction t) raises(dbError);
```

In der derzeitigen Implementierung müssen nur die Operationen `BeginTransaction` (Anfang einer globalen Transaktion), `WriteUndoLogEntry` (Schreiben eines Logeintrags für eine Subtransaktion) und `EndTransaction` (Ende der globalen Transaktion) wirklich etwas tun. Durch das allgemein gehaltene Interface kann aber zu einem späteren Zeitpunkt die Implementierung des Logmanagers geändert werden, ohne dass die Schnittstellen angepasst werden müssen.

Aktionen bei BOT:

Am Anfang jeder Transaktion muss diese mit der Methode `BeginTransaction` beim Logmanager angemeldet werden. Reine Lesetransaktionen können darauf verzichten, da sie niemals Einträge ins Log vornehmen müssen. Die Implementierung dieser Methode trägt die Transaktion mit Status 0 in die Relation `TraficTransactions` ein.

Aktionen beim Ausführen einer Operation:

Zu jeder Operation einer Transaktion, die die Datenbank modifiziert, muss eine Kompensationsaktion ins Log eingetragen werden, so dass die Effekte der Operation gegebenenfalls wieder zurückgenommen werden können. Das Eintragen ins Log geschieht dabei innerhalb derselben Transaktion wie das Ausführen der Änderung in der Datenbank, so dass genau dann ein Kompensationseintrag im Log steht, wenn die Änderung in der Datenbank wirksam wurde. Die Implementierung der Mehrschichtenstrategie berechnet zunächst die notwendige Kompensationsaktion und übergibt sie dann mit der Methode `WriteUndoLogEntryTA` an den Logmanager. Zusätzliche Parameter dabei sind der Kontext der OTS-Transaktion, die zum Schreiben ins Log genutzt werden soll, und die Komponentendatenbank, in der die Kompensationsaktion gegebenenfalls ausgeführt werden soll. Die Methode fügt dann im Kontext dieser Transaktion ein entsprechendes Tupel in die Relation `TraficLog` ein; als `SID` wird dabei die laufende Nummer der Kompensationsaktion dieser Transaktion gewählt. Sollte die Länge der Kompensationsaktion zu groß sein, wird die Information auf mehrere Tupel verteilt.

Aktionen bei EOT:

Wie bereits angesprochen gibt es zwei Möglichkeiten, um am Ende einer Transaktion sicherzustellen, dass ihre Effekte später nicht mehr kompensiert werden:

1. Möglichkeit: Sofortiges Löschen

Bevor das Commit der Vatertransaktion bestätigt wird, werden alle Undo-Logsätze, die zu dieser Transaktion gehören, aus dem Log gelöscht. Das Problem hier ist, dass die Logdatenbank durch die häufigen Löschungen stark belastet wird und es daher zu Performanceproblemen kommen kann. Obwohl dieser Ansatz sehr einfach ist, wird er daher verworfen.

2. Möglichkeit: Markieren der beendeten Transaktion

In der Relation `TraficTransactions` werden alle gerade aktiven Transaktionen vermerkt. Das Commit der Transaktion wird lediglich durch das Verändern des Statuseintrages des Eintrags in dieser Relation festgehalten. Eine nachfolgende Recovery berücksichtigt nur diejenigen Transaktionen, die in `TraficTransactions` enthalten, aber nicht als mit Commit beendet gekennzeichnet sind.

Da jetzt keine Löschungen aus dem Log anfallen, können potentiell mehr Einfügeoperationen parallel stattfinden. Gleichzeitig bedeutet das aber auch, dass sich die Logrelation ständig vergrößert, so dass von Zeit zu Zeit nicht mehr benötigte Einträge gelöscht werden müssen, was durch die Methode `FlushLog()` geschieht. Dies kann aber zu Zeiten erfolgen, wo es wenig sonstige Zugriffe auf die Logdatenbank gibt, zum Beispiel nachts. Die aktuelle Implementierung verwendet diese Methode.

Kompensieren von Transaktionen auf Seite des Logmanagers

Das Kompensieren von Transaktionen ist Aufgabe des Restartmanagers, und zwar sowohl im laufenden Betrieb nach Transaktionsabbrüchen als auch während der Recovery nach einem vorhergehenden Ausfall. Zur Kapselung des Zugriffes auf das Log verwendet er dabei Methoden des Logmanagers, die wir im einzelnen in Abschnitt 4.5.5 im Rahmen des Restartmanagers vorstellen.

Optimierung des Logzugriffes

Wegen der speziellen Struktur der Zugriffe auf die Logdatenbank ist es möglich, durch geeignete Wahl eines Isolation Levels einen besseren potentiellen Durchsatz zu erreichen. Es gibt drei Arten von Zugriffen auf die Logrelation:

- **Einfügen von Undo-Logsätzen:** Hier sorgt bereits der Logmanager für eindeutige Schlüssel, indem geeignete IDs und SIDs vergeben werden, so dass es hier nicht zu Konflikten beim Einfügen kommt. In diesem Bereich ist also eigentlich auf der Ebene der Tupel überhaupt keine Concurrency Control seitens des Logdatenbanksystems erforderlich, möglicherweise aber auf Seiten- und Indexebene.
- **Lesen von Logsätzen beim Zurücksetzen einer Transaktion:** Hier werden alle Logsätze von bereits mit Commit beendeten Subtransaktionen dieser Transaktion gelesen, dagegen nicht die Logsätze von Subtransaktionen, die noch aktiv sind. Wenn der Abbruch einer Transaktion eingeleitet wird, werden auch die noch aktiven Subtransaktionen beendet, so dass keine weiteren Einträge in das Log mehr durch Subtransaktionen vorgenommen werden, die zu dieser Transaktion gehören. Logeinträge werden außerdem grundsätzlich nicht mehr geändert, nachdem sie geschrieben wurden. Es genügt also, hier den Isolation Level „Read Committed“ zu fordern.
- **Lesen von Logsätzen beim Neustart:** Beim Neustart werden erst nach dem vollständigen Ablauf der Recovery neue globale Transaktionen erlaubt, so dass es nicht zu Konflikten mit Einfügeoperationen kommen kann.

Es ist also möglich, durch Setzen des Isolation Levels **„Read Committed“** weitere potentielle Parallelitätsgewinne zu erzielen.

4.5.5 Der Restartmanager

Der Restartmanager ist zuständig für das Kompensieren von Transaktionen, wenn die Mehrschichtenstrategie verwendet wird, und zwar sowohl im laufenden Betrieb nach Transaktionsabbrüchen als auch während der Recovery nach einem vorhergehenden Ausfall.

Recovery nach einem vorhergehenden Ausfall

Beim Neustart von TraFIC nach einem vorhergehenden Ausfall muss der Restartmanager die Kompensationsaktionen für die bereits mit Commit beendeten Subtransaktionen anstoßen, deren Vatertransaktionen zum Zeitpunkt des Fehlers noch nicht beendet waren. Dazu selektiert er über den Logmanager aus der Log-Datenbank alle zum Zeitpunkt des Auftretens des Fehlers aktiven globalen Transaktionen; das sind alle, die noch nicht als beendet markiert sind und für die es mindestens einen Eintrag in `TraficLog` gibt. Transaktionen, die zwar aktiv waren, aber keinen Logeintrag geschrieben haben, haben auch noch keine mit Commit beendeten Subtransaktionen, die Änderungen vorgenommen haben. Für sie muss daher auch kein Undo durchgeführt werden. Anschließend wird in beliebiger Transaktionsreihenfolge das Undo durchgeführt, indem die jeweiligen Kompensationsaktionen in der Reihenfolge absteigender SIDs aus dem Log gelesen und ausgeführt werden. Die Transaktionsreihenfolge ist deshalb beliebig, weil die Transaktionen zum Zeitpunkt ihrer Ausführung wegen der semantischen Isolationseigenschaft keine Aktionen ausgeführt haben, die miteinander in Konflikt standen.

Es gibt zwei Möglichkeiten, wie die Recovery ablaufen kann:

- Die gesamte Recovery wird in einer einzigen Transaktion durchgeführt, in der nacheinander mit der Methode `GetNextUndoOperation()` des Logmanagers alle Logeinträge der zum Zeitpunkt des Absturzes aktiven Transaktionen gelesen und die entsprechenden Kompensationsaktionen ausgeführt werden. Diese Lösung hat den Nachteil, dass im Falle eines erneuten Systemfehlers während der Recovery beim nächsten Neustart alle Kompensationsaktionen erneut durchgeführt werden müssen.
- Jede Kompensationsaktion wird in einer eigenen Transaktion ausgeführt, in der gleichzeitig auch der Undo-Eintrag aus dem Log gelöscht wird. Lesen und Löschen des Logeintrags erledigt die Methode `GetAndDropUndoOperation()` des Logmanagers. Alternativ könnte man auch einen Kompensations-Logsatz eintragen. Bei dieser inkrementellen Variante wird verhältnismäßig viel Aufwand für die Verwaltung der Transaktionen notwendig, dafür gewinnt man aber erheblich, wenn es während der Ausführung der Recovery zu einem Systemfehler kommt.

Der grundsätzliche Ablauf beider Varianten der Recovery ist folgender:

Zunächst ruft der Restartmanager die Methode `BeginRecovery` des Logmanagers auf. Diese Methode baut eine Liste der auszuführenden Kompensationsaktionen in internen Strukturen des Logmanagers auf. Im einzelnen tut sie dabei folgendes:

1) Durch einen Scan von `TraficTransactions` werden alle Transaktionen bestimmt, die Status 0 haben, also noch nicht beendet sind.
2) Durch eine Selektion auf der Relation `TraficLog` werden die SIDs der Kompensationsaktionen bestimmt, die für jede Transaktion ausgeführt werden müssen, und in eine Liste geschrieben. Die auszuführende Aktion wird an dieser Stelle noch nicht gelesen, um nicht zuviel Speicher zu belegen.
3) Der Zeiger `CurrentUndo` wird auf die erste SID der ersten Transaktion in der Tabelle gesetzt.

Zum Zugriff auf diese interne Tabelle des LogManagers dienen die Methoden des Interfaces `Recovery`, das vom LogManager implementiert wird. Die Methode `BeginReco-`

very gibt eine Instanz eines solchen Recovery-Objektes zurück, dessen Implementierung die genannte Tabelle enthält.

Anschließend werden über die Methoden GetNextUndoOp für die En-Block-Recovery bzw. GetAndDropNextUndoOp für die inkrementelle Recovery dieses Recovery-Objektes nacheinander die notwendigen Kompensationsaktionen gelesen und über VHDBS ausgeführt. Bei der inkrementellen Variante wird der gesamte Vorgang in einer einzigen OTS-Transaktion durchgeführt, die En-Bloc-Variante kapselt jeweils das Lesen und Ausführen einer Kompensationsaktion in einer OTS-Transaktion.

Die Methode GetNextUndoOpTA liest die nächste auszuführende Kompensationsaktion aus der Datenbank, die Methode GetAndDropNextUndoOpTA liest sie und löscht sie außerdem aus der Datenbank. Beiden Methoden wird der Kontext der OTS-Transaktion übergeben, innerhalb der das Lesen und ggf. Löschen aus dem Log geschehen soll.

Bei der Ausführung von GetNextUndoOpTA werden durch den Logmanager die folgenden Aktionen ausgeführt:

1) Falls es keine auszuführenden Kompensationsaktionen mehr gibt, werden alle Einträge in TraficLog und TraficTransactions gelöscht und die Exception NoRecovery ausgelöst.
2) Die Kompensationsaktion an der aktuellen Stelle der Tabelle wird aus der Datenbank gelesen; sollte die Aktion in mehrere Teile aufgespalten worden sein, müssen diese jetzt zusammengesetzt werden:

   ```
   CurrentEntry:=1
   REPEAT
      SELECT * FROM TraficLog WHERE TID=CurrentUndo.TID
      AND SID=CurrentUndo.SID AND EntryID=CurrentEntry;
      Gelesene Kompensationsaktion evt. an bereits gelesene anhängen, CurrentEntry++
   UNTIL keine Daten mehr
   ```

3) CurrentUndo wird auf das nächste Element in der Tabelle verschoben.
4) Die auszuführende Kompensationsaktion wird zurückgegeben.

Die Ausführung von GetAndDropUndoOp läuft analog zu der obigen ab. Im ersten Schritt müssen allerdings nur die Informationen in der Relation TraficTransactions gelöscht werden, wenn es keine auszuführenden Kompensationsaktionen mehr gibt, da die Daten in TraficLog bereits während der Recovery gelöscht wurden. Zusätzlich werden vor dem Verschieben noch die gerade gelesenen Tupel gelöscht:

```
DELETE FROM TraficLog WHERE TID=CurrentUndo.TID
                          AND SID=CurrentUndo.SID;
```

Das Problem dabei ist, dass die Datenbankadapter die komplexen, in OQL formulierten Kompensationsaktionen nicht direkt ausführen können. Stattdessen muss der Umweg über den Föderationsserver gewählt werden. Um eine solche Operation auszuführen, muss dazu die Methode SelectTA des Föderationsservers verwendet werden, der ein Transaction-Objekt als Parameter übergeben werden muss. Eine naheliegende Lösung wäre es nun, die TransactionFactory von TraFIC zu verwenden, um ein Transaction-Objekt zu erhalten. Das ist aber nicht ausreichend, da zum Zugriff auf das Log nicht das Transaction-Objekt, sondern der OTS-Transaktionskontext erforderlich ist, der von TraFIC intern verwaltet und nicht nach außen gegeben wird.

Die Lösung dieses Problems ist die folgende: Der Restartmanager implementiert selbst das `Transaction`-Interface, z.B. durch eine Implementierungsklasse

```
class RestartTransaction_i: public Trafic::TransactionBOAImpl
{
  CosTransactions::Control_ptr control;
  ...
};
```

Im Konstruktor muss dabei eine OTS-Transaktion begonnen werden, deren `Control`-Objekt in einer Variablen gespeichert wird. Die Implementierungen von `select, SetTupleAttribute` usw. geben nun dieses `Control`-Objekt als Parameter an die entsprechenden Methoden des `Repserver-, dbCollection-` oder `dbTuple`-Interfaces weiter (`selectTA`, `setTA`, ...). Die Methode `CommitTransaction` beendet die OTS-Transaktion.

Die En-Bloc-Recovery läßt sich also folgendermaßen zusammenfassen:

```
RestartTransaction_i *ta=new RestartTransaction_i;
vhdbs=vhdbs::_bind(„:VHDBS“,“”);
recovery=LogManager.BeginRecovery();
repeat
    op=recovery->GetNextUndoOperation(ldbs,ta->control);
    vhdbs->SelectTA(op,ta);
until exception NoRecovery;
ta->CommitTransaction();
```

Die inkrementelle Recovery hat analog den folgenden Aufbau:

```
vhdbs=vhdbs::_bind(„:VHDBS“,“”);
recovery=LogManager.BeginRecovery();
repeat
    RestartTransaction_i *ta=new RestartTransaction_i;
    op=recovery->GetAndDropUndoOp(ldbs,ta->control);
    vhdbs->SelectTA(op,ta);
    ta->CommitTransaction();
until exception NoRecovery;
```

Ablauf beim Start des föderierten Datenbanksystems

Der Restartmanager wird immer beim Start des föderierten Datenbanksystems aufgerufen. Wenn das Log leer ist, weil es sich um einen initialen Start handelt, tut der Restartmanager nichts. Wenn das Log nicht leer ist, handelt es sich um einen Start nach einem vorhergehenden Systemfehler, und die Recovery muss angestoßen werden. Das Interface ist daher sehr einfach: Es gibt eine Methode `DoRecovery()` für die en-Bloc-Recovery, sowie eine entsprechende Methode `DoIncrementalRecovery()` für die inkrementelle Recovery.

Zurücksetzen aktiver Transaktionen

Das Kompensieren einer einzigen Transaktion verläuft analog zur En-Bloc-Recovery. Lediglich das Recovery-Objekt wird nicht durch `BeginRecovery`, sondern durch die Methode `CompensateTransaction(TID)` des Logmanagers besorgt. Der Logmanager legt nun eine ähnliche Tabelle wie vorher an, in der aber nur die Kompensationsaktionen für diese Transaktion enthalten sind. Der Zugriff auf diese Tabelle erfolgt wieder über Methoden des `Recovery`-Objekt. `GetNextUndoOpTA` erkennt, dass nur eine einzige

Transaktion zurückgesetzt wird, und löscht die Informationen dieser Transaktion aus dem Log, wenn alle Kompensationsaktionen ausgeführt wurden.

Einfluss lokaler Transaktionen

Der Algorithmus zur Recovery kann nur funktionieren, wenn lokale Transaktionen über die globale Ebene umgeleitet werden und so ebenfalls semantische Sperren erwerben müssen. Andernfalls wäre es möglich, dass eine lokale Transaktion nach dem Ende einer Subtransaktion einer globalen Transaktion auf Daten zugreift, die die Subtransaktion geschrieben hat, und diese möglicherweise ihrerseits ändert. Würde die globale Transaktion abgebrochen und dabei die zu der Subtransaktion gehörenden Kompensationsaktionen ausgeführt, würde eventuell auch der Effekt der lokalen Transaktion aufgehoben.

4.5.6 Der Serialisierungsgraph

Der Serialisierungsgraph ist ein eigenständiger Server, der das Interface `SerializationGraph` realisiert. Er wird zum einen bei den Tickettechniken dazu benutzt, die Relationen der Ticketwerte der Transaktionen in den einzelnen Datenbanksystemen zu überwachen und gegebenenfalls das Abbrechen einer Transaktion zu verlangen, wenn eine global nicht serialisierbare Ausführung entstanden ist. Zum anderen verwendet ihn die graphbasierte Strategie, um den globalen OSI-MVSG (siehe Abschnitt 3.6.3) zu verwalten und nichtserialisierbare Ausführungen anhand von Zyklen im Graphen zu erkennen.

Der Serialisierungsgraph verwaltet intern eine Liste von Transaktionen, die die Knoten des Graphs darstellen. Bei Aufruf der Methode `AddTransaction` wird eine neue Transaktion in diese Liste aufgenommen. Kanten im Graphen sind entweder Ticketkanten oder Kanten des OSI-MVSG; wir beschreiben nun, wie diese Kanten in den Graphen gelangen.

Ticketkanten:

Zu jeder Transaktion werden die Ticketwerte gespeichert, die diese Transaktion in den einzelnen Datenbanken gelesen hat, außerdem Zeitmarken für ihren Startzeitpunkt in den einzelnen Komponentensystemen und ggf. den Zeitpunkt ihres Commits (realisiert durch einen Zähler) sowie ihr aktueller Status (aktiv oder beendet).

Die Ticketwerte werden dem Serialisierungsgraphen durch die Methode `SetTicketValue` übermittelt. Mit dem Eintragen eines Ticketwerts aktualisiert der Serialisierungsgraph außerdem die Kanten im Graph: Für jede Transaktion, die Knoten im Graph ist und ein Ticket in der gleichen Datenbank gelesen hat, wird die Relation zum Ticket dieser Transaktion geprüft und eine entsprechende Kante in den Graphen eingetragen. Abschließend wird geprüft, ob durch das Einfügen der zusätzlichen Kanten ein Zyklus im Graphen entstanden ist. Wenn das der Fall ist, wird die Exception `SerializationGraph::CycleDetected` zurückgegeben.

MVSG-Kanten:

Um die Kanten des Online-SI-MVSG eintragen zu können, muss der Serialisierungsgraph über alle Operationen einer globalen Transaktion informiert werden, die eine Komponentendatenbank lesen oder ändern. Im Interface des Serialisierungsgraphen gibt es dazu die folgenden Methoden:

- `AddQuery(Transaction,Query)`, um den Serialisierungsgraphen über eine Anfrage einer Transaktion zu informieren. Die Query wird als `OQLSequence` übermittelt, die Informationen darüber enthält, auf welche Komponentendatenbanken sich die Anfrage bezieht.
- `AddUpdateAttribute(Transaction,Object,Attribute)` zur Information über die Änderung eines Attributs eines Objektes.
- `AddInsertObject(Transaction,Object,Repository)` zur Information über das Einfügen eines Objektes in ein Repository.
- `AddUpdateObject(Transaction,Object)` zur Information über das Löschen eines Objektes.

Alle Methoden aktualisieren die Prädikate, die die Objektmenge charakterisieren, auf die die Transaktion zugegriffen hat. Die Ableitung der Prädikate aus den Operationen geschieht dabei genau wie in den Sperrmanagern, wir haben die Details dieses Verfahrens in Abschnitt 4.5.3 geschildert. Zur Verwaltung der Prädikate werden die Datenstrukturen und Methoden verwendet, die wir in Abschnitt 4.5.1 beschrieben haben. Anhand dieser Informationen und den Zeitmarken der Start- und Commitzeitpunkte der Transaktionen werden dann die Kanten berechnet, die in den Graphen eingetragen werden müssen, wie es in Abschnitt 3.6.3 beschrieben wurde. Anschließend wird geprüft, ob die neu eingefügten Kanten einen Zyklus im Graphen geschlossen haben; wenn dies der Fall ist, muss die Operation, die zu diesen Kanten geführt hat, abgelehnt werden.

Commit einer Transaktion:

Der Commit-Wunsch einer Transaktion wird dem Serialisierungsgraphen durch die Methode `CommitTransaction` mitgeteilt. Durch den Zyklustest beim Einfügen einer neuen Kante ist sichergestellt, dass der Graph jetzt zykelfrei ist, also muss kein erneuter Zyklustest durchgeführt werden. Statt dessen wird geprüft, welche Transaktionen aus der Liste entfernt werden können: Das sind gerade die, die beendet sind und eine Quelle im Graph sind; außerdem muss gelten, dass alle übrigen Transaktionen beendet sind, die aktiv waren, als die Transaktion beendet wurde (siehe Theorem 2.5 und Theorem 3.1). Alle Knoten und Kanten, die diese Transaktionen betreffen, können aus dem Graphen entfernt werden.

Abbruch einer Transaktion:

Das Abbrechen einer Transaktion wird dem Serialisierungsgraph durch die Methode `AbortTransaction` signalisiert, die den entsprechenden Knoten, alle zugehörigen Kanten und alle Informationen über Ticketwerte und Prädikate löscht. Zusätzlich werden – wie beim Commit – die übrigen Transaktionen gelöscht, die jetzt eventuell aus dem Graphen entfernt werden können.

4.5.7 Die OSI-Tabelle

Die OSI-Tabelle wird bei der optimistischen Technik zur Gewährleistung globaler Snapshot Isolation (siehe Abschnitt 3.8.3) benutzt, um Verletzungen der SI-Eigenschaft des globalen Schedules zu erkennen. Dazu muss sie sowohl Informationen über Start- und Commitzeitpunkte globaler Transaktionen als auch über die Operationen haben, die sie ausgeführt haben. Da das die gleichen Daten sind, die auch der Serialisierungsgraph benötigt, haben beide das gleiche Interface. Die OSI-Tabelle ist also eine weitere Imp-

lementierung des Interfaces `SerializationGraph`, wobei der Teil zur Verwaltung der Ticketwerte ohne Funktion ist.

Auch die interne Arbeitsweise der OSI-Tabelle ist eng an den Serialisierungsgraphen angelehnt, insbesondere werden Prädikate und Zeitmarken identisch verwaltet. Lediglich an der Stelle, wo im Serialisierungsgraphen die Kanten berechnet werden, die in den Graphen eingetragen werden müssen, prüft die OSI-Tabelle lediglich die Einhaltung der Bedingungen aus Theorem 3.10, indem nach der Erkennung einer Überlappung zweier Prädikate folgender Test ausgeführt wird:

```
// current predicate and predicate l overlap,
// check if global SI still holds

if (trans->isGloballyConcurrent(l->transaction)==TRUE)
  { if (database->areParallel(trans,l->transaction)==FALSE)
    {
      if (l->Mode==WRITE)
      break; // WRITE->READ oder WRITE->WRITE, so abort
    }
  }
}
```

4.6 Strategien der Transaktionsverwaltung

In diesem Abschnitt wird die Realisierung der verschiedenen Strategien vorgestellt, die in Trafic zur Gewährleistung von Serialisierbarkeit und Atomarität der globalen Transaktionen eingesetzt werden. Es handelt sich dabei um die verschiedenen Varianten der Tickettechnik sowie um Mehrschichtentransaktionen, die wir in Abschnitt 3.4 vorgestellt haben, sowie um die Techniken, die wir in den Abschnitten 3.6 und 3.8 entwickelt haben. Wir beginnen einer allgemeinen Beschreibung der Implementierung von Strategien, die Details der einzelnen Strategien folgen in den anschließenden Abschnitten.

4.6.1 Allgemeine Implementierungsaspekte

Jede Strategie wird als eigenständiges Objekt realisiert, das das Standardinterface `CCStrategy` implementiert. Dieses Interface enthält insbesondere eine Methode, die beim Starten von Transaktionen durch die `TransactionFactory` aufgerufen wird und ein `Transaction`-Objekt erzeugt. Das einer Strategie zugeordnete Objekt wird mit einem eindeutigen Marker versehen (die implizite Tickettechnik z.B. mit dem Marker „ITT“, das Mehrschichtentransaktionsverfahren mit „MLTM“ für „Multilevel Transaction Management“). Trafic verwaltet eine Referenz auf das Objekt, das zur aktuell aktiven Strategie gehört. Zusätzlich implementiert jede Strategie das `Transaction`-Interface, so dass sie bei Anfragen und Modifikationen innerhalb einer Transaktion sowie am Ende die notwendigen Maßnahmen zur Concurrency Control einleiten kann. Jede Implementierung des `Transaction`-Interfaces erbt von einer Basisklasse, die Buch darüber führt, auf welche Komponentensysteme eine Transaktion zugreift.

Erhält das `TransactionFactory`-Objekt innerhalb des Trafic-Servers die Aufforderung, eine neue Transaktion zu erzeugen, leitet es diese Aufforderung an das aktuelle Strategieobjekt weiter und gibt das von diesem zurückgegebene `Transaction`-Objekt an den Client zurück. Alle weiteren Anfragen und Modifikationen, die innerhalb der Transaktion ausgeführt werden, können unmittelbar durch Aufrufe von Methoden dieses `Transaction`-Objektes realisiert werden. Die Rolle des `TransactionFactory`-

Objektes in dieser Architektur lässt sich vergleichen mit der des Orbix-Daemons, der auch nur genutzt wird, um die erste Verbindung zwischen Client und Server herzustellen und ansonsten nicht an ihrer Kommunikation beteiligt ist.

Wenn die Strategie gewechselt werden soll, wird, sofern das aufgrund des aktuellen Systemzustandes möglich ist, an das entsprechende Strategieobjekt über den zugehörigen Marker gebunden und die Referenz aktualisiert. In der Regel ist das nur dann möglich, wenn gerade keine Transaktionen aktiv sind, sonst kann die Serialisierbarkeit des globalen Schedules nicht mehr gewährleistet werden.

4.6.2 Strategien zur Gewährleistung globaler Serialisierbarkeit

4.6.2.1 Tickettechniken

Alle Tickettechniken verwenden die gleiche Implementierung von `CCStrategy` und `Transaction`, da sich die Unterschiede im wesentlichen auf die Art des Ticketzugriffs beschränken.

CCStrategy-Implementierung:

Die Auswahl zwischen den einzelnen Varianten wird durch Angabe eines Parameters im Konstruktor der Implementierung `TicketCCS_i` von `CCStrategy` vorgenommen, außerdem wird der Marker übergeben, mit dem später an das Strategieobjekt der gewünschten Strategie gebunden werden kann. Das Erzeugen des `CCStrategy`-Objekts für die implizite Tickettechnik sieht beispielsweise wie folgt aus:

```
TicketCCS_i *CCStrategyITT=new TicketCCS_i(ImplicitTicket,"ITT");
```

Zur Auswahl der impliziten Tickettechnik muss TraFIC also das `CCStrategy`-Objekt mit dem Marker `ITT` auswählen.

Transaction-Implementierung:

BeginTransaction:

Bereits im Konstruktor der Klasse `TicketTransaction_i` wird eine OTS-Transaktion begonnen, deren `Control`-Objekt in einem Attribut der Klasse gespeichert wird. Im Kontext dieser Transaktion werden später die Operationen der Transaktion ausgeführt.

Transaktionale Methoden:

Die Realisierung der verschiedenen transaktionalen Methoden sieht im wesentlichen immer gleich aus: Zunächst muss die OTS-Transaktion lokal wiederhergestellt werden, dann wird die entsprechende Methode des Datenbankadapters aufgerufen, anschließend wird die OTS-Transaktion wieder „verlassen“. Ursprünglich war vorgesehen, für jede Operation eine eigene OTS-Subtransaktion zu beginnen. Es hat sich jedoch herausgestellt, dass sich Subtransaktionen nicht zusammen mit expliziter Propagierung des Transaktionskontextes verwenden lassen, da zwei Subtransaktionen der gleichen Transaktion auf dem Zielserver wie zwei vollständig unabhängige Transaktionen erscheinen. Insbesondere kann es zu Sperrkonflikten in der Datenbank zwischen solchen Subtransaktionen kommen, die unweigerlich zu einer Verklemmung führen, da die Sperren der Subtransaktionen erst mit dem Ende der globalen Transaktion freigegeben werden. Aus diesem Grund werden alle Requests im Kontext der eigentlichen OTS-Transaktion ausgeführt. Da man ein `Control`-Objekt immer nur einem Thread zuordnen kann, bedeutet

dies, dass eine Transaktion immer nur einen Request zu einer Zeit bearbeiten kann; weil VHDBS selbst immer nur einen Thread pro Transaktion benutzt, ist dies aber keine Einschränkung.

Um die Implementierung der Methoden zu vereinheitlichen, wurde ein Satz von Makros geschaffen, der die genannten Schritte ausführt. Anhand der Methode `Select` wird jetzt der Aufbau einer Implementierungsmethode unter Verwendung dieser Makros gezeigt:

```
dbCollection_ptr TicketTransaction_i::select(const OQLSequence &query,
                                             const char *repName,
                                             RepServer_ptr ldbs,
                                             CORBA::Environment &env)
                 throw (CORBA::SystemException,dbError)
{
  dbCollection_ptr result=NULL;

  TRANSACTION_INTRO("select")

  try
  {
    result=ldbs->selectTA(query,repName,mycontrol);
  }
  TRANSACTION_CATCH("select")

  TRANSACTION_FINISH("select")

  TransactionBaseClass::select(query,repName,ldbs,env);

  return result;
}
```

Im Makro `TRANSACTION_INTRO` wird die im Konstruktor begonnene OTS-Transaktion mit der OTS-Methode `resume` an den ausführenden Thread gekoppelt. Da eine Transaktion immer nur einem Thread zugeordnet sein darf, wird zuvor eine exklusive Sperre erworben.

```
#define TRANSACTION_INTRO(op) \
  CosTransactions::Control_ptr mycontrol;\
  CosTransactions::Current_ptr current=\
     CosTransactions::Current::IT_create();\
  mutex_lock(&ControlLock);\
  current->resume(Control);\
  mycontrol=current->get_control();
```

Das Makro `TRANSACTION_CATCH` hat die Aufgabe, Exceptions abzufangen und zu behandeln, die während der OTS-Operationen oder der Ausführung der Methode im Zielserver entstehen. Dabei führen alle Exceptions außer der VHDBS-eigenen `dbError`-Exception grundsätzlich zum Abbruch der Transaktion durch Aufruf der Methode `AbortTransaction`; an den Aufrufer wird in diesem Fall die (CORBA-) Exception `TRANSACTION_ROLLEDBACK` zurückgegeben. Eine auftretende `dbError`-Exception wird direkt an den Aufrufer zurückgegeben; gegebenenfalls muss dieser selbst die Transaktion zurücksetzen, wenn es sich um einen schwerwiegenden Fehler handelt. Sehr wichtig ist es, die Zuordnung der lokalen OTS-Transaktion zum ausführenden Thread aufzuheben (`suspend`), die Sperre des `Control`-Objektes freizugeben und den Zähler der aktiven Operationen zu vermindern, bevor die Transaktion abgebrochen bzw. die Exception an den Aufrufer propagiert wird.

Exemplarisch hier die Behandlung einer `SystemExeption` und einer `dbError`-Exception:

```
#define TRANSACTION_CATCH(op)\
(...)
catch (dbError &dbe)\
{\
  cerr << "dbError exception during " << op << " for TID " << TID <<
endl;
  cerr << dbe.why << endl;
  Control=current->suspend();
  mutex_unlock(&ControlLock);\
  pendingOperations--;\
  throw(dbe);\
}\
catch (CORBA::SystemException &sysEx)\
{\
  cerr << "SysEx during " << op << " for TID " << TID <<" : "<< endl;\
  cerr << &sysEx;\
  Control=current->suspend();
  mutex_unlock(&ControlLock);\
  pendingOperations--;\
  AbortTransaction();\
  ROLLEDBACK_EXCEPTION;\
}
```

Das abschließende Makro `TRANSACTION_FINISH` löst die Zuordnung der OTS-Transaktion zum ausführenden Thread (mit `suspend`), vermindert den Zähler der aktiven Operationen dieser Transaktion und gibt die Sperre auf dem `Control`-Objekt frei.

Alle Methoden, die Modifikationen an Objekten vornehmen können, setzten zusätzlich den Status der Transaktion auf „Read-Write“, nachdem er im Konstruktor mit „Unmarked“ initialisiert wurde. Beim Commit kann dadurch entschieden werden, ob gewisse Optimierungen für Nur-Lese-Transaktionen eingesetzt werden können. Die Methoden, die diese Ergänzung haben, sind `execMeth`, `setTupleAttribute`, `insertElement` und `removeElement`. Zusätzlich kann eine Applikation mittels der Methode `SetIsolationLevel` explizit angeben, eine Nur-Lese-Transaktion zu sein; modifizierende Operationen werden dann nicht erlaubt:

```
if (TransType==Unmarked) TransType=ReadWrite;
if (TransType==ReadOnly)
{ cout << "[Trafic::CCStrategy::ETT] SetTupleAttribute TID " << TID;
  cout << " not allowed in read-only transaction." << endl;
  AbortTransaction();
  ROLLEDBACK_EXCEPTION;
}
```

Beim ersten Zugriff auf eine Oracle-Datenbank muss außerdem der Isolation Level mit der Methode `SetIsolationLevel` des Datenbankadapters gesetzt werden, da Oracle sonst nur Read Committed verwenden würde, wir aber Snapshot Isolation benötigen, um die Korrektheit des Verfahrens garantieren zu können.

CommitTransaction:

Vor dem Commit einer Transaktion muss der Wert des Tickets in jeder beteiligten Datenbank gemäß der gewählten Variante der Tickettechnik bestimmt und u.U. erhöht werden. Die Strategie bedient sich dabei der Liste der verwendeten Datenbanken, die

die Basisklasse der `Transaction`-Implementierung verwaltet. Im einzelnen sehen die Ticketzugriffe der Varianten der Tickettechnik für jede beteiligte Datenbank folgendermaßen aus:

- Die implizite Technik erhöht einen datenbankspezifischen Zähler, der eine „Zeitmarke“ für den Zeitpunkt des Commits darstellt; die Reihenfolge der Commits gibt die Serialisierungsreihenfolge in dieser Datenbank an.
- Die optimistische Technik führt einen Ticketzugriff in der Datenbank aus, indem sie die Methode `TicketAccessRWTA` des Datenbankadapters benutzt, die den alten Ticketwert liest und den um 2 erhöhten Wert in die Datenbank schreibt.
- Die erweiterte Technik für lokale SI-Datenbanken nutzt die vorher gesammelte Information, um zu entscheiden, ob das Ticket nur gelesen oder auch geschrieben werden muss: Bei reinen Lesetransaktionen genügt es, das Ticket mit der Methode `TicketAccessROTA` zu lesen, modifizierende Transaktionen müssen das Ticket auch erhöhen und zurückschreiben.
- Die automatische Tickettechnik bestimmt anhand der Eigenschaften der Datenbank (aus der Metadatenbank), welche Arten von Ticketzugriffen bei Erhaltung der Korrektheit möglich sind, und wählt dabei den mit der besten Performance aus. Dabei wird die implizite Technik immer bevorzugt, die aber nur möglich ist, wenn der Scheduler der Datenbank commit-ordnungserhaltend serialisiert. Die nächstbeste Möglichkeit ist eine Oracle-Datenbank, in der die Transaktion nur gelesen hat, hier genügt das Lesen des Tickets (`TicketAccessROTA`). Andernfalls wird das Ticket gelesen und wieder zurückgeschrieben (`TicketAccessRWTA`).

Die so bestimmten Werte des Tickets in jeder Datenbank werden nun in den Serialisierungsgraphen eingetragen. Kommt es dabei zu einer Exception `SerializationGraph::CycleDetected`, hat der Serialisierungsgraph eine global nicht serialisierbare Ausführung entdeckt, die Transaktion muss also abgebrochen werden. Wenn alle Ticketwerte erfolgreich eingefügt werden konnten, kann OTS veranlasst werden, die verteilte Transaktion mit Commit zu beenden. Dazu wird zunächst das `Terminator`-Objekt bestimmt, das zu der Transaktion gehört, indem die Methode `get_terminator()` des `Control`-Objektes aufgerufen wird. Mit der Methode `commit()` bzw. `rollback()` dieses Objektes kann nun die Transaktion beendet werden, dabei wird von OTS ggf. das Zweiphasen-Commitprotokoll durchgeführt.

Als Beispiel hier ein Teil der Implementierung des Commits einer Transaktion, die Ausnahmebehandlung ist auch hier wieder weggelassen worden:

```
void Trafic::Transaction::CommitTransaction()
{
  CosTransactions::Terminator_ptr term;

  term=Control->get_terminator(); // Terminator bestimmen

  term->commmit(1);                // Transaktion beenden
}
```

Außerdem wird dem Ticketgraphen mitgeteilt, dass die Transaktion mit Commit beendet wurde, damit dieser seine internen Datenstrukturen aktualisieren kann.

AbortTransaction:

Beim Abbrechen der Transaktion wird zunächst dem Ticketgraphen mitgeteilt, dass er alle Knoten und Kanten, die diese Transaktion betreffen, löschen kann. Anschließend wird die OTS-Transaktion mit Hilfe des in der Klasse gespeicherten `Control`-Objektes zurückgesetzt. OTS übermittelt dabei den Abbruch selbsttätig an alle beteiligten Datenbanken.

4.6.2.2 Mehrschichtentransaktionen

Ähnlich wie bei den Tickettechniken beschreiben wir zunächst die Implementierung des Interfaces `CCStrategy`, danach die Implementierung des Interfaces `Transaction`.

CCStrategy

Die Implementierungsklasse von `CCStrategy` hat bei der Mehrschichtenstrategie keine Aufgabe zu erfüllen, außer in `InitStrategy` die weiter unten genannten internen Datenstrukturen zu füllen.

Transaction-Implementierung

Für die Realisierung der Mehrschichtenstrategie können die von OTS angebotenen Subtransaktionen nicht genutzt werden, da dabei die Subtransaktionen ihre gehaltenen Sperren am Ende nicht freigeben und somit das Verfahren keinerlei Performancevorteile hätte. Statt dessen wird für die Ausführung jeder Operation eine eigene OTS-Transaktion begonnen, in deren Kontext die Operation selbst und eventuell das Schreiben der Kompensationsaktion ins Log durchgeführt wird.

BeginTransaction:

Am Anfang einer Transaktion muss diese beim Logmanager mit dessen Methode `BeginTransaction` angemeldet werden.

Transaktionale Methoden:

Die Implementierung der transaktionalen Methoden hat immer den gleichen Aufbau:

- Zunächst wird eine OTS-Transaktion begonnen, in deren Kontext die folgenden Operationen ausgeführt werden.
- Anschließend wird, falls es erforderlich ist, der zuständige Sperrmanager bestimmt und eine Sperre angefordert. Zum lesenden Zugriff auf einen Cursor oder auf Elemente einer Collection sind keine zusätzlichen Sperren erforderlich, da diese bereits bei der Ausführung der eigentlichen Anfrage erworben wurden, so dass `cursor`, `first`, `next`, `more` und `getElement` keine Sperren anfordern.
- Nun wird eine Kompensationsaktion berechnet und über den Logmanager ins Log eingetragen, falls es sich um eine modifizierende Operation handelt (`SetTupleAttribute`, `execMeth`, `insertElement`, `deleteElement`).
- Jetzt wird die entsprechende Methode im Datenbankdadapter aufgerufen, anschließend wird die OTS-Transaktion beendet.

Wie bei der Tickettechnik wurde auch hier ein Satz von Makros bereitgestellt, um die Implementierung der Methoden zu erleichtern und zu vereinheitlichen.

Bestimmen der Kompensationsaktionen

Um eine Änderung eines Attributs eines Objektes zu kompensieren, muss zunächst das geänderte Objekt selektiert werden, anschließend muss das geänderte Attribut wieder auf seinen alten Wert zurückgesetzt werden. Dies wird als OQL-Anweisung mit Hilfe eines OQL-Cursors realisiert; zur Selektion des Objektes werden dabei nur die Schlüsselattribute genutzt, die aus der Metadatenbank bestimmt werden können. Wenn also das Attribut `A` eines Objektes im Repository `Rep`, dessen Schlüsselattribut `S` den Wert 4 hat, von 10 auf 11 geändert wurde, lautet die Kompensationsaktion

```
FOR obj IN (SELECT o FROM o IN Rep WHERE o.S=4) DO obj.A:=10 END;
```

Ein Sonderfall, auf den geachtet werden muss, ist die Änderung eines Schlüsselattributs, hier muss natürlich der neue Wert zur Selektion des Objektes benutzt werden. Problematisch an dieser Anweisung ist, dass man aus einem übergebenen Objekt nur den Namen des Datenbankrepositories erhalten kann, in dem das Objekt liegt. In der obigen Anweisung muss aber ein VHDBS-Spiegelrepository benutzt werden, damit die Anfrage vom Föderationsserver bearbeitet werden kann. Zur Lösung dieses Problems verwendet die Implementierung eine ähnliche Technik wie schon der Sperrmanager: Sie hält Informationen über Typen, Datenbank- und VHDBS-Repositories in internen Strukturen und kann daher ohne Umweg über die Metadatenbank darauf zugreifen. Zur Formulierung der Kompensationsaktion wird aus diesen internen Daten irgendein VHDBS-Repository bestimmt, das auf das Datenbankrepository abgebildet wird, in dem das Objekt liegt. Auch die Information über die Schlüsselattribute wird aus den internen Tabellen entnommen.

Methodenaufrufe werden kompensiert, indem eigens entworfene Kompensationsmethoden aufgerufen werden, die mit einer besonderen Namenskonvention angesprochen werden. Ähnlich wie bei Änderungen wird zunächst das geänderte Objekt selektiert und dann die Kompensationsmethode aufgerufen, zum Beispiel

```
FOR obj IN (SELECT o FROM o IN Rep WHERE o.S=4)
DO obj.compensate_KaufeAnteil() END;
```

Einfügungen werden durch eine entsprechende Löschoperation kompensiert, Löschungen analog durch eine Einfügung.

CommitTransaction:

Beim Commit der globalen Transaktion muss ein entsprechender Eintrag im Log vorgenommen werden, damit die Auswirkungen dieser Transaktion bei einer späteren Recovery nicht rückgängig gemacht werden. Die Implementierung dieser Methode ruft daher die Methode `EndTransaction` des Logmanagers auf und teilt diesem damit das erfolgreiche Commit der Transaktion mit.

Außerdem müssen alle Sperren freigegeben werden, die diese Transaktion hält. Dazu werden aus der Liste, die die Basisklasse verwaltet, die Namen aller benutzten Datenbankadapter gelesen, dann wird an die zugehörigen Sperrmanager gebunden. Schließlich werden die Sperren der Transaktion mit der Methode `FreeGlobalLocks` jedes betroffenen Sperrmanagers freigegeben.

AbortTransaction:

Ein Abbruch einer Transaktion im laufenden Betrieb (wegen eines Fehlers oder auf Wunsch der Transaktion selbst) erfordert es, die Effekte der bereits beendeten Subtrans-

aktionen dieser Transaktion zu kompensieren. Dazu wird der Restartmanager verwendet, dessen bereits beschriebene Methode `CompensateTransaction` die notwendigen Kompensationsaktionen aus dem Log liest und über den Föderationsserver ausführt.

Natürlich müssen nach dem erfolgreichen Kompensieren der Effekte der Transaktion noch alle gehaltenen Sperren freigegeben werden, wie dies auch beim Commit geschieht. Wenn die Sperren bereits vor Abschluss der Kompensation freigegeben würden, könnte es sein, dass andere Transaktionen Effekte der abgebrochenen Transaktion sähen

4.6.2.3 Graphbasierter Algorithmus

Die Implementierung des graphbasierten Algorithmus, den wir in Abschnitt 3.6.3 vorgestellt haben, ist sehr stark an die Implementierung der Tickettechniken angelehnt. Insbesondere verwaltet der graphbasierte Algorithmus die OTS-Transaktion genau wie die Tickettechniken, auch die gleichen Makros werden verwendet. Die Unterschiede liegen darin, welche zusätzlichen Aktionen bei den verschiedenen Operationen notwendig sind.

Transaktionale Methoden:

Neben der Verwaltung der OTS-Transaktion, die bereits bei der Implementierung der Tickettechnik beschrieben wurde, muss der Serialisierungsgraph über die Aktionen informiert werden, die die Transaktion gemacht hat. Die jeweilige Implementierung der transaktionalen Methode ruft dazu die entsprechende Methode des Serialisierungsgraphen auf, also

- `AddQuery(this,query)`, wenn die Transaktion `this` (das ist in C++ ein Pointer auf die Instanz, deren Methode gerade ausgeführt wird, hier also des aktuellen `Transaction`-Objektes) die Anfrage `query` ausführt,
- `AddUpdateAttribute(this,obj,attr)`, wenn die Transaktion Attribut `attr` von Objekt `obj` ändert,
- `AddInsertObject(this,obj,repName)`, wenn die Transaktion Objekt `obj` in das Repository `repName` einfügt, und
- `AddUpdateObject(this,obj)`, wenn die Transaktion Objekt `obj` löscht.

Für Operationen, die sich auf die Föderationsdatenbank beziehen, sind keine Einträge in den globalen Graphen notwendig, weil es dort nur transaktionsprivate Daten gibt. Dort treten also keine Konflikte zwischen Transaktionen auf, der lokale Schedule ist dann immer äquivalent zu einer beliebigen seriellen Anordnung der Transaktionen.

CommitTransaction:

Weil beim Einfügen neuer Kanten sofort geprüft wird, ob ein Zyklus im Graphen entstanden ist, ist dies beim Commit einer Transaktion nicht mehr notwendig. Die Methode `CommitTransaction` des Serialisierungsgraphen markiert wird die Transaktion im Serialisierungsgraph als beendet und enternt sie aus dem Graphen, sobald es nach den Vorgaben des Algorithmus möglich ist.

Danach wird die zur Transaktion gehörende OTS-Transaktion mit Commit beendet. Damit werden alle Änderungen, die die Transaktion in den Komponentensystemen vorgenommen hat, persistent gemacht.

AbortTransaction:

Beim Abbruch einer Transaktion muss nicht nur die entsprechende OTS-Transaktion beendet werden, sondern auch der Serialisierungsgraph informiert werden, damit er die Transaktion und alle ihre Kanten aus dem Graphen löschen kann.

4.6.2.4 Kombinationsverfahren

Neben den bisherigen Strategien, die ausschließlich jeweils einen Algorithmus implementieren, haben wir auch das in Theorem 3.5 beschriebene Kombinationsverfahren realisiert. Es verwendet die auf dem globalen OSI-MVSG basierende Technik für Datenbanken, die Snapshot Isolation gewährleisten, und eine der Tickettechniken für die übrigen Datenbanken. Der Serialisierungsgraph, den wir in Abschnitt 4.5.6 vorgestellt haben, unterstützt dies, indem er sowohl die Kanten der verschiedenen lokalen OSI-MVSG als auch des globalen Ticketgraphen in einer einzigen Graphstruktur verwaltet, so dass auch globale Zyklen erkannt werden können.

Um dies automatisch realisieren zu können, wird in der Metadatenbank von VHDBS dazu vermerkt, von welchem Typ die Scheduler der lokalen Datenbanken sind. Die Implementierung der Methoden des `Transaction`-Interfaces informiert dann, wie wir es im letzten Abschnitt beschrieben haben, den Serialisierungsgraphen über Operationen, die die Transaktion in Datenbanken ausführt, die Snapshot Isolation garantieren. Für Operationen in den übrigen Datenbanken sind keine besonderen Maßnahmen notwendig.

Beim Commit einer Transaktion werden dann für alle Datenbanken, in denen die Transaktion Operationen ausgeführt hat und die nicht Snapshot Isolation garantieren, die Ticketwerte bestimmt und an den Serialisierungsgraphen weitergegeben, der entsprechende Kanten in den globalen Graphen einträgt. Anschließend wird der globale Graph auf Zyklen untersucht. Wenn ein Zyklus gefunden wird, ist der entstandene Schedule nicht serialisierbar, so dass das Commit der Transaktion zurückgewiesen werden muss. Andernfalls wird die zugehörige OTS-Transaktion mit Commit beendet. Die Transaktion kann nun aus dem globalen Graphen entfernt werden, wenn die notwendigen Bedingungen dazu erfüllt sind.

4.6.3 Strategien zur Gewährleistung globaler Snapshot Isolation

4.6.3.1 Synchronisation der Subtransaktionen

Auch die Implementierung der in Abschnitt 3.8.2 beschriebenen Strategie zur Gewährleistung globaler Snapshot Isolation, die auf der Synchronisation der Subtransaktionen basiert, verwendet ein ähnliches Gerüst wie die Tickettechniken zur Verwaltung der OTS-Transaktion, die zu der aktuellen Transaktion gehört. Bereits am Beginn der globalen Transaktion muss in jedem Komponentensystem, das Snapshot Isolation verwendet, eine lokale Subtransaktion begonnen werden; auf andere Komponentensysteme kann unter dieser Strategie nicht zugegriffen werden, wenn man weiter Korrektheit garantieren will. Dazu baut die Strategie bereits in der Methode `InitStrategy` des Inter-

faces `CCStrategy` eine Liste aller möglichen Komponentensysteme auf. In der Implementierung von `BeginTransaction` wird dann in jedem dieser Komponentensysteme eine Subtransaktion begonnen. Dies geschieht, indem über den jeweiligen Datenbankadapter der Isolation Level „Snapshot Isolation“ ausgewählt wird.

Weitere Maßnahmen während der Ausführung von Operationen oder zum Commit der Transaktion sind nicht erforderlich, ausgenommen natürlich die Operationen, die zur Verwaltung der OTS-Transaktion notwendig sind.

4.6.3.2 Optimistischer SI-Test

Das Implementierungsgerüst des optimistischen SI-Tests, den wir in Abschnitt 3.8.3 vorgestellt haben, ist praktisch identisch mit dem des graphbasierten Algorithmus aus Abschnitt 4.6.2.3. Dies ist deshalb möglich, weil die Interfaces des Serialisierungsgraphen und der OSI-Tabelle identisch sind. In dieser Strategie muss lediglich an die globale OSI-Tabelle statt an den Serialisierungsgraphen gebunden werden.

4.7 Konfigurationsoptionen

Mit dem `ConfigManager`-Interface können in TraFIC die verschiedenen Komponenten des Baukastensystems konfiguriert werden.

4.7.1 Auswahl der Isolations- und Atomaritätsstrategie

Durch die Auswahl einer Strategie zur Gewährleistung von globaler Serialisierbarkeit wird immer die „passende“ Atomaritätsstrategie mit ausgewählt. In Kapitel 3 wurde deutlich gemacht, dass die Auswahl dieser beiden Komponenten aufeinander abgestimmt werden muss, um eine brauchbare Lösung zu erzielen.

Die verwendete Strategie wird mit der Methode `SwitchToStrategy` gewechselt, der als Parameter der String mit dem Namen der Strategie übergeben wird. Werden die Strategien in externe Server ausgelagert, dient dieser Name gleichzeitig als Marker, um den zugehörigen Server zu finden. In der Regel muss vor dem Wechsel der Strategie gewartet werden, bis keine Transaktionen mehr aktiv sind, um die Korrektheit zu gewährleisten.

Mögliche Strategien sind:

- `ITT`: implizite Tickettechnik + 2PC
- `OTT`: optimistische explizite Tickettechnik + 2PC
- `ETT`: erweiterte Tickettechnik für SI-Datenbanken
- `ATT`: automatische Tickettechnik, die für jede Komponentendatenbank die Variante der Tickettechnik wählt, die die beste Performance erlaubt, + 2PC
- `MLTM`: Mehrschichtenstrategie, bei der alle Operationen als eigene Subtransaktion ausgeführt werden, mit dem erforderlichen Logging
- `MVSG`: Auf dem OSI-MVSG basierende Strategie für SI-Datenbanken, die für Komponentensysteme, die serialisierbare Schedules erzeugen, die Variante der Tickettechnik verwendet, die die beste Performance erlaubt, + 2PC

- `PSI`: Die pessimistische Variante der Strategie zur Gewährleistung globaler Snapshot Isolation, + 2PC
- `OSI`: Die optimistische Variante der Strategie zur Gewährleistung globaler Snapshot Isolation, + 2PC

Die Exception `StrategyNotAvailable` wird ausgelöst, wenn die gewünschte Strategie nicht ausgewählt werden kann, der dabei übergebene String gibt nähere Auskunft über den Grund des Fehlschlagens.

In Abschnitt 3.4 wurde ausführlich diskutiert, dass es keine Strategie gibt, die für alle Anwendungsfälle optimal ist. Es obliegt daher grundsätzlich dem Administrator, die Anwendungsumgebung zu analysieren und die geeignete Strategie auszuwählen. Er wird sich dabei im wesentlichen auf die Ergebnisse von Performancemessungen stützen müssen.

Hat sich der Administrator entschieden, die automatische Tickettechnik zu nutzen, wählt TraFIC anhand von Informationen über die Komponentendatenbanken die jeweils leistungsfähigste Variante aus. Dazu werden Informationen über die erzeugten Schedules der beteiligten Datenbanksysteme in der Metadatenbank abgelegt. Natürlich bleibt dem Administrator weiterhin die Möglichkeit, explizit eine andere Tickettechnik auszuwählen, die dann für alle Systeme eingesetzt wird.

4.7.2 Weitere Konfigurationsmöglichkeiten

Verwendete Recovery-Methode

Die Recovery-Methode, die beim Warmstart von TraFIC eingesetzt wird, kann mit der Methode `SetRecoveryMode` des ConfigManagers eingestellt werden:

```
enum RecoveryModes {EnBlocRecovery, IncrementalRecovery};
void SetRecoveryMode( in RecoveryModes mode);
```

4.8 Zusammenfassung

Wir haben in diesem Abschnitt die Implementierung einer Transaktionsverwaltung für ein föderiertes Datenbanksystem vorgestellt, die alle Strategien zur Gewährleistung globaler Atomarität und Serialisierbarkeit unterstützt, die wir in den Abschnitten 3.6 und 3.8 vorgestellt haben. Als Basis dieser Entwicklung dient das CORBA-basierte föderierte System VHDBS, das mit einem Wrapper-Mediator-basierten Ansatz heterogene Datenbanksysteme wie Oracle, SQL Server und O_2 integriert. Weil der CORBA-eigene Transaktionsmechanismus OTS insbesondere im Bereich der globalen Concurrency Control gravierende Defizite hat, haben wir eine eigenständige Transaktionskomponente, TraFIC, in VHDBS integriert.

Wir leiten dazu alle Anfragen und Modifikationen, die im Kontext einer Transaktion ausgeführt werden, nicht direkt an die Datenbankadapter, sondern leiten sie über TraFIC um, wo notwendige Maßnahmen getroffen werden. TraFIC verfügt über eine Suite von verschiedenen Strategien zur Gewährleistung globaler Atomarität und Serialisierbarkeit, aus denen die ausgewählt werden kann, die im jeweiligen Anwendungsumfeld die beste Leistung verspricht. Unter anderem werden verschiedene Varianten der Tickettechnik, Mehrschichtentransaktionen und die auf dem OSI-MVSG basierende Strategie unter-

stützt. Daneben bietet TraFIC auch Strategien zur Gewährleistung globaler Snapshot Isolation. Die Strategien stützen sich auf eine Menge von Mechanismen, die Basisfunktionalität implementieren. Alle Komponenten können als separate CORBA-Server instantiiert werden, um so die mögliche Performance zu erhöhen.

5 Experimentelle Evaluation

In diesem Kapitel evaluieren wir experimentell die Leistungsfähigkeit der Techniken zur föderierten Concurrency Control, die wir im dritten Kapitel entwickelt haben. Wir setzen dazu die prototypische Implementierung der Transaktionsverwaltung für das VHDBS-System ein, die wir im letzten Kapitel beschrieben haben. Als Vergleichsbasis dient eine Suite von Benchmarkprogrammen in einem stark vereinfachten Anwendungsszenario, die wir in Abschnitt 5.1 vorstellen. In den Abschnitten 5.2 und 5.3 stellen wir die Ergebnisse der verschiedenen Benchmarks vor, zunächst für die Techniken zur Gewährleistung globalen Serialisierbarkeit, dann für die Verfahren zur globalen Snapshot Isolation.

5.1 Die Benchmarksuite

Die experimentelle Evaluation unserer Techniken basiert auf dem Prototyp der Transaktionsverwaltung für VHDBS, dessen Implementierung wir im letzten Kapitel vorgestellt haben.

5.1.1 Anwendungsszenario

Als Anwendungsumgebung für unsere Benchmarks wählen wir ein (stark vereinfachtes) föderiertes System zur Abwicklung von Aktienkäufen. Wir betrachten dazu zwei verschiedene Typen von Datenbanken, die zu einer Föderation zusammengeschlossen sind:

- Datenbanken von Börsenmaklern, die Informationen über die aktuellen Preise von Aktien und über Wertpapierdepots von Kunden enthalten, sowie
- die Datenbank einer Bank, die Informationen über Wertpapiere enthalten, die Kunden bei den Börsenmaklern erworben haben.

Wir nehmen an, dass in der Datenbank der Bank exakt die gleichen Informationen über Aktien der Kunden auftauchen wie in allen Datenbanken der Börsenmakler zusammen, und fordern dies als globale Konsistenzbedingung für die föderierte Datenbank. Zusätzlich nehmen wir an, dass die Mengen der Aktien, die die Makler anbieten, paarweise disjunkt sind.

In unserer Benchmarkumgebung, die schematisch in dargestellt ist, gibt es zwei Börsenmakler und eine Bank, die jeweils Oracle 8i einsetzen. Die Datenbanken der Börsenmakler haben in dieser idealisierten Umgebung alle den gleichen Aufbau. Sie bestehen aus jeweils drei Relationen:

- Die Relation `Stocks(StockID,Price)` enthält die Aktien, die dieser Makler anbietet, und ihre aktuellen Preise. Das Attribut `StockID` ist der eindeutige Primärschlüssel der Relation. Diese Relation enthält bei allen Maklern 100 Tupel.
- Die Relation `Customer(CustomerID,Name)` enthält die Kunden des Maklers. Alle Makler verwalten dieselben 100 Kunden.
- Die Relation `Portfolios(CustomerID,StockID,Amount)` enthält Informationen darüber, wie viele Exemplare einer Aktie ein Kunde besitzt. Diese Relation enthält ein Tupel für jedes mögliche Paar aus Kunden und Aktien, so dass sie zu Anfang aus 10000 Tupeln besteht.

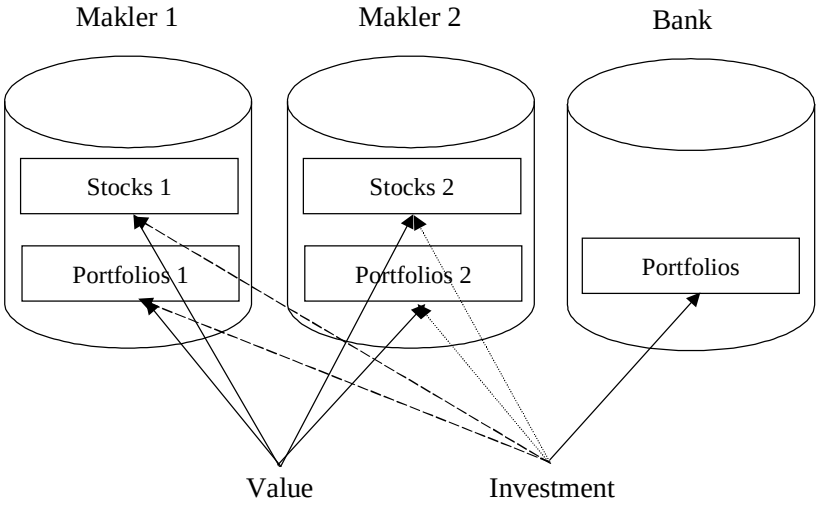

Abbildung 5.1 - Anwendungsszenario für die Benchmarksuite

Die Datenbank der Bank besteht aus den folgenden beiden Relationen:

- Die Relation `Customer(CustomerID,Name)` enthält die Kunden der Bank, es sind wieder dieselben Kunden wie bei den Maklern.
- Die Relation `Portfolio(CustomerID,StockID,Amount)` enthält das Wertpapierportefeuille der einzelnen Kunden. Die Bank hat keine Kenntnisse über den aktuellen Wert einer Aktie, kann aber aus der `StockID` den Makler ableiten, der für die Aktie zuständig ist. Weil die Kunden ausschließlich bei den drei Maklern Aktien kaufen, ist diese Relation logisch die Vereinigung der einzelnen `StockList`-Relationen der Makler, entsprechend enthält sie zu Beginn jeder Messung 20000 Tupel.

Die Datenbanken, die wir für den Benchmark einsetzen, enthalten also relativ wenige Daten. Performanceengpässe können daher ausschließlich aufgrund von Konflikten nebenläufiger Transaktionen entstehen. Unser Benchmark prüft also, wie gut die verschiedenen Verfahren mit häufig auftretenden Konflikten umgehen können, die in der Realität vor allem unter hoher Last vorkommen, und versucht nicht, alle Aspekte der wirklichen Welt widerzuspiegeln.

5.1.2 Transaktionstypen

Als Last lassen wir zwei verschiedene Typen föderierter Transaktionen ablaufen:

- Die *Investment*-Transaktion modelliert die Investition einer festen Summe durch einen Kunden in Exemplare einer einzigen Aktie. Sie muss dazu zunächst den Makler bestimmen, der die Aktie vertreibt, dann aus dessen Datenbank den aktuellen Wert der Aktie lesen und schließlich sowohl in der Datenbank des Maklers als auch in der Datenbank der Bank das Portefeuille des Kunden anpassen. Diese Transaktion vertritt die Klasse der Transaktionen, die sowohl lesend als auch schreibend auf die föderierte Datenbank zugreifen.
- Die *Value*-Transaktion bestimmt den aktuellen Wert des Portefeuilles eines Kunden, indem sie aus den Datenbanken der Makler die Aktien bestimmt, die der Kunde be-

sitzt, und dann mit den aktuellen Preisen aus den gleichen Datenbanken den Gesamtwert berechnet. Diese Transaktion ist eine typische föderierte Lesetransaktion mittlerer Dauer.

Die Investment-Transaktion greift auf genau eine Makler-Datenbank und auf die Datenbank der Bank zu; die Value-Transaktion greift auf die beiden Makler-Datenbanken zu. Abbildung 5.2 zeigt Pseudocode für diese beiden Transaktionstypen.

```
Investment(cid,sid,amount):
 determine stock broker database SB from sid;
 select price from SB.Stocks where StockID=sid;
 number:=amount/price;
 update SB.Portfolios set Amount=Amount+number; }
 update Bank.Portfolios set Amount=Amount+number; }


Value(cid):
 totalvalue:=0;
 for StockID,Amount in
  (select StockID,Amount from SB1.Portfolios where CustomerID=cid)
  { select Price from SB1.Stocks where stockid=StockID;
    totalvalue+=Amount*Price;
  }
 for StockID,Amount in
  (select StockID,Amount from SB2.Portfolios where CustomerID=cid)
  { select Price from SB2.Stocks where stockid=StockID;
    totalvalue+=Amount*Price;
  }
```

Abbildung 5.2 - Pseudocode der Transaktionsprogramme

5.1.3 Messumgebung

Für die in den folgenden Abschnitten gezeigten Messungen wurden die Module von TraFIC als separate CORBA-Server implementiert und über sieben relativ leistungsgleiche Rechner so verteilt, dass die CPU-Last möglichst balanciert war:

Modul	**Rechner**	**Typ**
Lastgenerator	`frankfurt`	Sun SparcStation 20
TransactionFactory, Strategieimplementierungen	`dudweiler`	Sun SparcStation 5
Logmanager	`emden`	Sun SparcStation 4
Sperrmanager, Serialisierungsgraph	`bochum`	Sun SparcStation 4
Oracle-Adapter	`kyoto, mainz, santiago`	Sun SparcStation 4
VHDBS-Server	`london`	Sun SparcStation 10

Die Komponentendatenbanken laufen auf ausreichend dimensionierten Serversystemen unter Solaris (Fachrichtungsserver `sothis` mit acht Prozessoren) und Windows NT (Lehrstuhlserver `niniveh` mit zwei Prozessoren sowie dedizierter PC `sofia`); während der Messungen hat sich gezeigt, dass die Leistungsfähigkeit der Komponentendatenbanken nie kritisch war.

Als Lastgenerator wird ein C++-Programm eingesetzt, das eine Strategie auswählt und während der gewünschten Dauer des Messlaufes eine vorgegebene Anzahl nebenläufiger Investment- und Value-Transaktionen in entsprechenden Threads ausführt; der Multiprogrammierungsgrad ist also ein Parameter der Messläufe. Für jede dieser Transaktionen wird die Antwortzeit bestimmt; am Ende des Messlaufes wird die Antwortzeit für jeden der Transaktionstypen als Mittelwert der Einzelzeiten berechnet. Weil es sich um ein geschlossenes System handelt, kann der Durchsatz für jeden Transaktionstyp nach dem Gesetz von Little als Quotient der gleichzeitig aktiven Transaktionen dieses Typs und seiner mittleren Antwortzeit berechnet werden [Lang92]. Für die Messungen in den folgenden Kapiteln wurde eine Messdauer von zehn Minuten gewählt; länger dauernde Kontrollmessungen haben gezeigt, dass sich mit dieser Messdauer ausreichend genaue Ergebnisse erzielen lassen.

Die absoluten Werte für den Durchsatz, die in den folgenden Abschnitten bestimmt werden, sind relativ gering. Dies liegt zum einen daran, dass wir unsere Benchmarks auf Rechnern ablaufen lassen, die weit hinter der Leistungsfähigkeit moderner Application Server zurückbleiben. Zum anderen wurde das zugrundeliegende VHDBS-System nicht hinsichtlich der erzielbaren Performance optimiert, sondern lediglich prototypisch implementiert.

5.2 Strategien zur globalen Serialisierbarkeit

In diesem Abschnitt evaluieren wir die Strategien zur Gewährleistung globaler Serialisierbarkeit, die wir in Kapitel 3 entwickelt haben. Wir konzentrieren uns dabei zunächst auf die Strategien, die mit lokaler Snapshot Isolation umgehen können, also auf die erweiterte Tickettechnik, die graphbasierte Technik und Mehrschichtentransaktionen. Zusätzlich bestimmen wir die Performance der Pseudostrategie "NONE", die keinerlei zusätzliche Maßnahmen zur Concurrency Control auf der föderierten Ebene trifft, sondern lediglich das ZweiphasenCommitprotokoll verwendet. Die Ergebnisse dieser Strategie sind also eine Schranke für die bestenfalls erzielbare Leistung.

Wir bestimmen in Abschnitt 5.2.1 zunächst den Overhead, der für die Anwendung der einzelnen Strategien notwendig ist. Anschließend betrachten wir in Abschnitt 5.2.2 die Performance der Strategien in lesedominierten Anwendungen und in Abschnitt 5.2.3 die Performance in update-dominierten Anwendungen.

5.2.1 Overhead

Mit den Messungen in diesem Abschnitt bestimmen wir den Overhead, der durch die Anwendung der einzelnen Strategien eingeführt wird. Wir parametrisieren dazu die Transaktionsprogramme so, dass keine Konflikte auf Daten in den Datenbanken entstehen können. Der erzielte Durchsatz hängt also ausschließlich vom notwendigen Aufwand auf der föderierten Ebene ab.

Abbildung 5.3 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.4 den erzielbaren Durchsatz von Value-Transaktionen mit der Pseudostrategie NONE, der erweiterten Tickettechnik (ETT), der graphbasierten Strategie (MVSG) und der Mehrschichten-Transaktionsstrategie (MLTM). Für die Value-Transaktionen haben wir zusätzlich den erzielbaren Durchsatz mit der etablierten optimistischen Tickettechnik gemessen (OTT), für die Investment-Transaktionen ist der Durchsatz dieser Technik identisch mit dem der erweiterten Tickettechnik. Alle Messwerte in diesem und den folgenden Charts geben Transaktionen pro Minute (tpm) wieder.

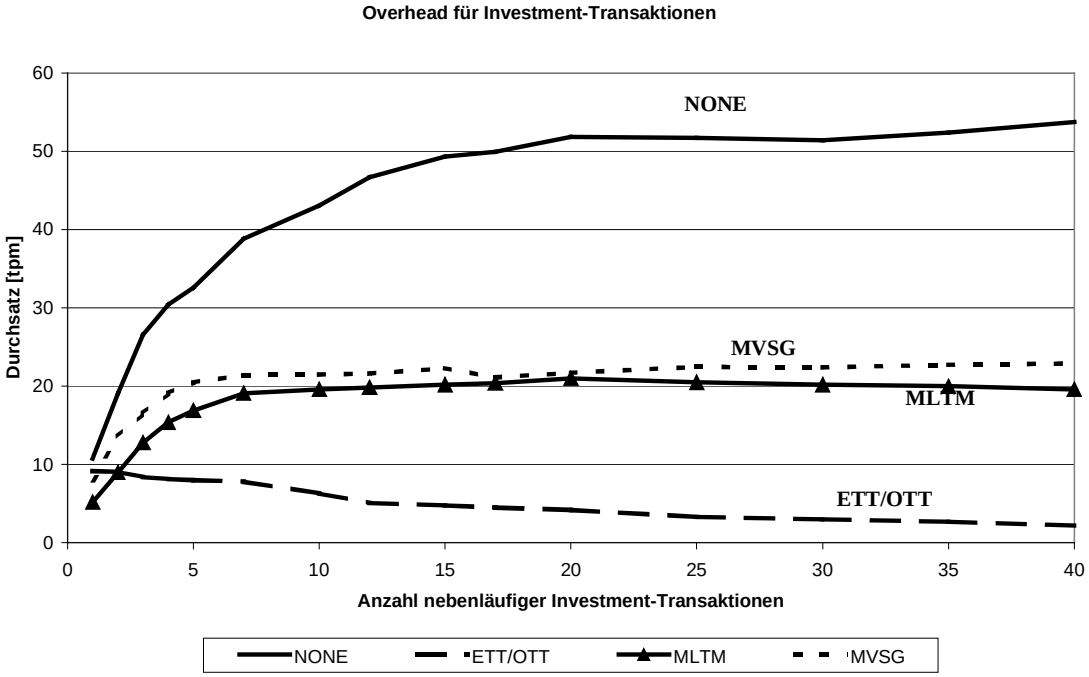

Abbildung 5.3 - Overhead für Investment-Transaktionen unter globaler Serialisierbarkeit

Die graphbasierte Strategie und die erweiterte Tickettechnik erlauben für Value-Transaktionen nahezu den gleichen Durchsatz wie die Pseudostrategie NONE, ihr Overhead ist also vernachlässigbar klein. Der Durchsatz der Value-Transaktionen unter der Mehrschichtenstrategie dagegen ist wesentlich geringer; dies liegt daran, dass eine Value-Transaktion in sehr viele Subtransaktionen zerlegt wird, für die jeweils ein eigenes Zweiphasen-Commit ausgeführt werden muss. Die etablierte Tickettechnik erlaubt schließlich den geringsten Durchsatz, weil sie wegen des gemeinsamen Zugriffs auf das Ticketobjekt jegliche Nebenläufigkeit von Transaktionen verhindert.

Bei den Investment-Transaktionen ergibt sich ein leichter Performancevorteil für die graphbasierte Strategie gegenüber der Mehrschichten-Transaktionsstrategie. Dies liegt daran, dass zwar beide im wesentlichen vergleichbaren Zusatzaufwand auf der föderierten Ebene benötigen, die Mehrschichtenstrategie aber zusätzlich das Zweiphasen-Commit der Subtransaktionen durchführen muss. Beide Strategien sind aber relativ weit von der Leistung der Pseudostrategie NONE entfernt, weil durch die kurzen Investment-Transaktionen die zusätzlichen Operationen schwerer wiegen. Beide Varianten der Tickettechnik schließlich erlauben den geringsten Durchsatz, weil der gemeinsame Zugriff auf das Ticketobjekt auch hier jegliche Nebenläufigkeit von Transaktionen ausschließt.

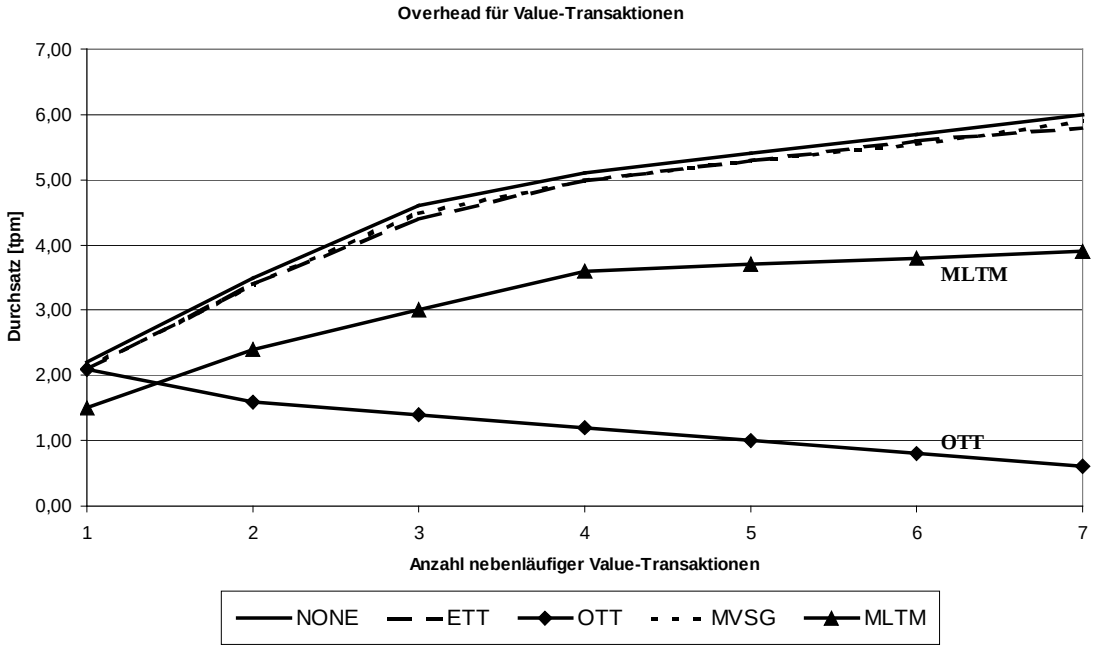

Abbildung 5.4 - Overhead für Value-Transaktionen unter globaler Serialisierbarkeit

5.2.2 Performance in lesedominierten Anwendungen

In diesem Abschnitt beurteilen wir die Leistungsfähigkeit der Strategien in Anwendungen, die von reinen Lesetransaktionen dominiert werden. Wir halten dazu in allen Messungen eine Hintergrundlast von fünf gleichzeitig aktiven Investment-Transaktionen konstant und variieren die Anzahl gleichzeitig aktiver Value-Transaktionen von eins bis 20. Damit dabei Konflikte auf den Daten eine Rolle spielen, verwenden für diese Messungen alle Transaktionen nur Parameter aus einem eingeschränkten Bereich: `Customerid` und `Stockid` werden gleichverteilt aus dem Intervall [1;10] gewählt.

Abbildung 5.5 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.6 den erzielbaren Durchsatz von Value-Transaktionen mit der Pseudostrategie NONE, der erweiterten Tickettechnik in ihrer optimistischen Ausprägung (ETT), der graphbasierten Strategie (MVSG) und der Mehrschichten-Transaktionsstrategie (MLTM).

Den besten Durchsatz für Investment-Transaktionen erzielt bis zu einem sehr hohen Parallelitätsgrad die graphbasierte Strategie. Weil die Gesamtlast auf dem System steigt, nimmt der mögliche Durchsatz mit steigender Zahl nebenläufiger Value-Transaktionen langsam ab, und zwar in einem ähnlichen Maß wie bei der Pseudostrategie NONE. Die zusätzlich notwendigen Maßnahmen der graphbasierten Strategie haben also auch unter hoher Last keinen zusätzlichen negativen Effekt auf den erzielbaren Durchsatz. Die erweiterte Tickettechnik erlaubt nur einen wesentlich geringeren Durchsatz für Investment-Transaktionen, weil sie diesen Transaktionstyp sequentiell ausführen muss. Eine steigende Anzahl nebenläufiger Value-Transaktionen wirkt sich aber weniger stark aus als bei der graphbasierten Strategie, da die erweiterte Tickettechnik für solche reinen Lesetransaktionen nur sehr wenig Zusatzaufwand erfordert. Unter sehr hoher Last ist sie daher die beste der drei Strategien. Die Mehrschichten-Transaktionsstrategie wird von

steigender Value-Last sehr stark beeinflusst, da es verstärkt zu Blockierungen kommt. Zusätzlich besteht jede Value-Transaktion aus vielen, sehr kurzen Subtransaktionen, die einen hohen Verwaltungsaufwand verursachen. Auch der Durchsatz der Investment-Transaktionen leidet darunter: schon unter relativ geringer Last sinkt der mögliche Durchsatz drastisch und bleibt dann der niedrigste aller drei Strategien. Bei allen Strategien sinkt der erzielbare Durchsatz mit steigender Last auf dem System, weil zunehmend mehr Aufwand für die Bearbeitung der Value-Transaktionen notwendig ist.

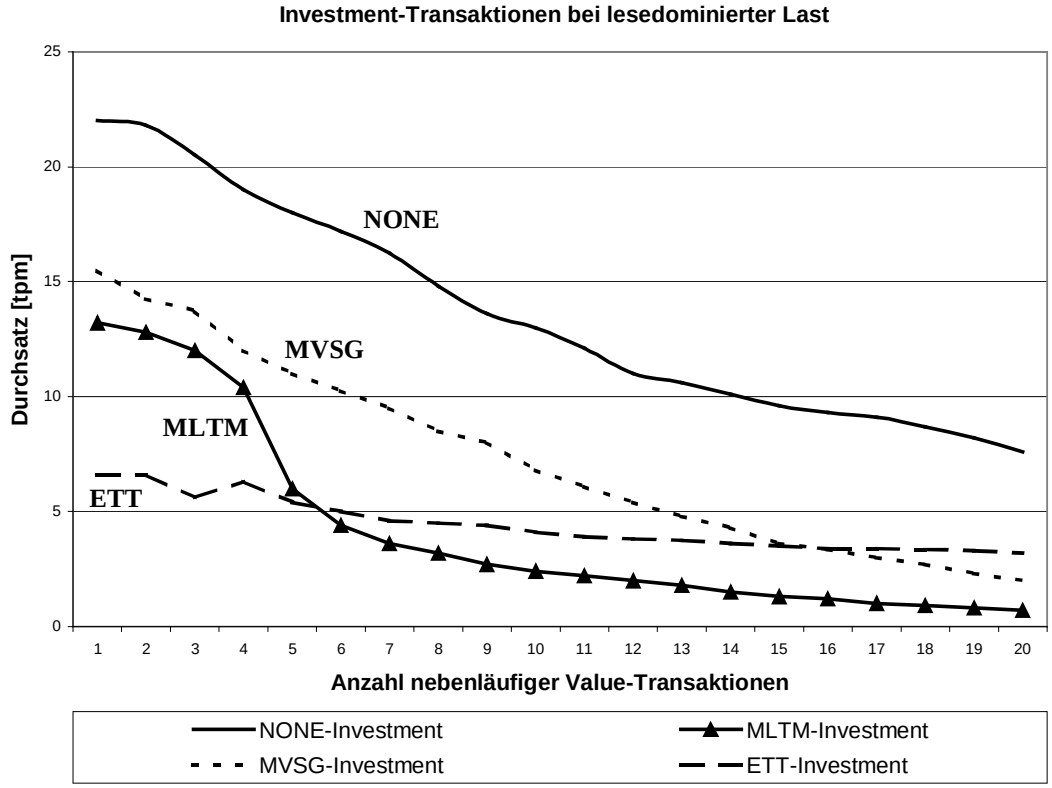

Abbildung 5.5 - Durchsatz von Investment-Transaktionen bei lesedominierter Last

Der Durchsatz der Value-Transaktionen ist für alle Strategien ungefähr gleich, wobei die Mehrschichten-Transaktionsstrategie an letzter Stelle liegt. Ähnlich wie schon bei den Overhead-Messungen liegt die graphbasierte Strategie recht nahe an der Pseudostrategie NONE. Die erweiterte Tickettechnik liegt etwas dahinter, was an der erhöhten Last auf den Gesamtsystem durch die Abbrüche von Investment-Transaktionen verursacht wird, die hier eine Rolle spielen, weil wir die optimistische Variante der erweiterten Tickettechnik gewählt haben. Die Mehrschichten-Transaktionsstrategie hat den geringsten Durchsatz, weil sie neben der Verwaltung der Sperren auf der föderierten Ebene zusätzlich die relativ große Zahl von Subtransaktionen verwalten muss, aus denen jede Value-Transaktion besteht (ca. 50 im Gegensatz zu drei bei einer Investment-Transaktion).

5.2.3 Performance in updatedominierten Anwendungen

In diesem Abschnitt beurteilen wir die Leistungsfähigkeit der Strategien in Anwendungen, die von Transaktionen dominiert werden, die Änderungen vornehmen. Wir halten

dazu in allen Messungen eine Hintergrundlast von fünf Value-Transaktionen konstant und variieren die gleichzeitig aktiven Investment-Transaktionen von eins bis 20. Auch hier schränken wir den Parameterbereich wie für die Messungen in Abschnitt 5.2.2 ein.

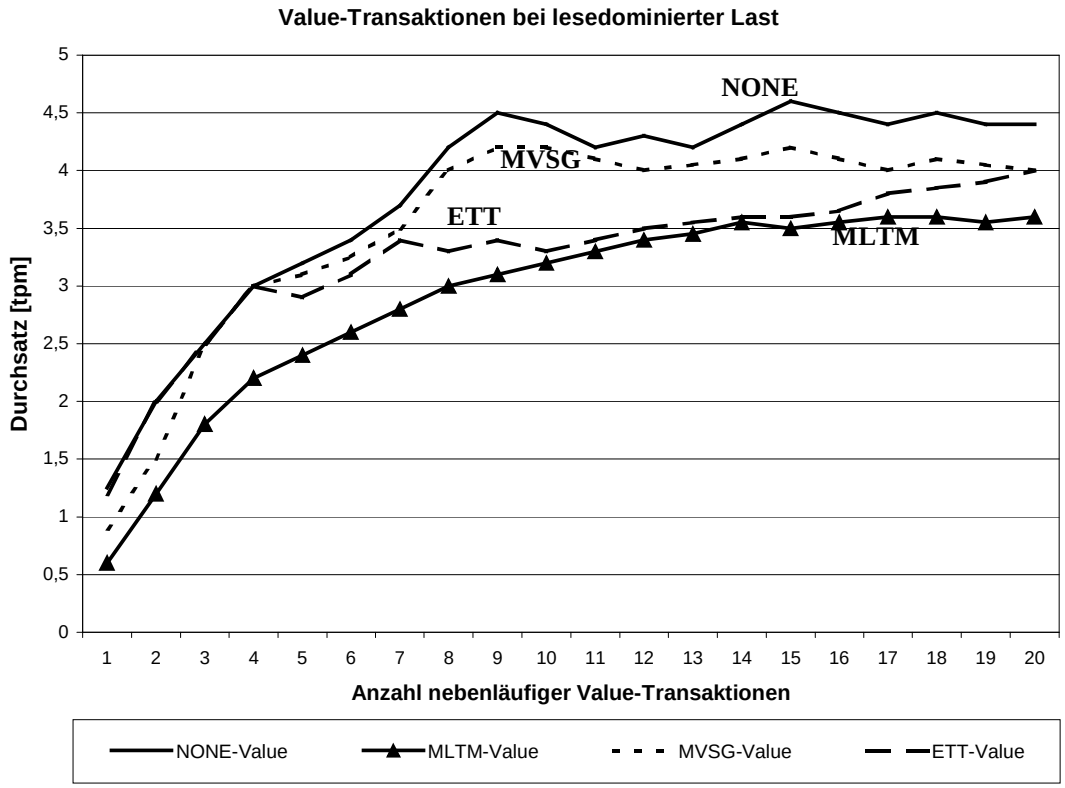

Abbildung 5.6 - Durchsatz von Value-Transaktionen bei lesedominierter Last

Abbildung 5.7 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.8 den erzielbaren Durchsatz von Value-Transaktionen mit der Pseudostrategie NONE, der erweiterten Tickettechnik (ETT), der graphbasierten Strategie (MVSG) und der Mehrschichten-Transaktionsstrategie (MLTM).

Den besten Durchsatz für Investment-Transaktionen erzielt anfangs die graphbasierte Strategie. Ab einer gewissen Zahl nebenläufiger Investment-Transaktionen kommt es allerdings verstärkt zu Transaktionsabbrüchen, weil zwei nebenläufige Investment-Transaktionen das gleiche Tupel verändern wollen, so dass der Durchsatz wieder leicht abnimmt. Die Mehrschichten-Transaktionsstrategie kann in dieser Situation ihre Stärke ausspielen, da sie jede Transaktion in kurze, unabhängige Subtransaktionen zerlegt, so dass es nicht zu zwangsweisen Abbrüchen wegen parallelen Änderungen des gleichen Tupels kommt. Sie erlaubt daher den besten Durchsatz bei großer Last. Beide Techniken sind aber wegen ihres relativ großen Zusatzaufwandes auf der föderierten Ebene recht weit vom idealen Durchsatz der Pseudostrategie NONE entfernt. Der Durchsatz der erweiterten Tickettechnik ist wegen der Zwangssequentialisierung der Transaktionen nochmals wesentlich geringer.

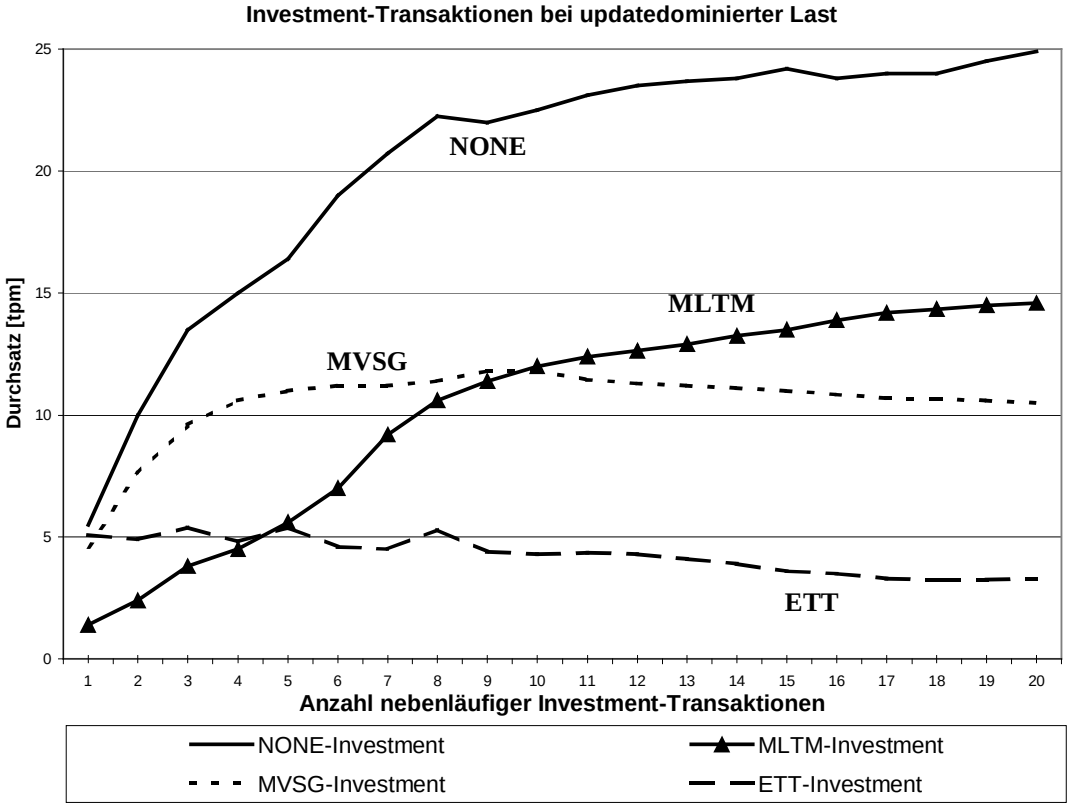

Abbildung 5.7 - Durchsatz von Investment-Transaktionen bei updatedominierter Last

Der Durchsatz der Value-Transaktionen ist für alle Strategien ungefähr gleich, wobei die Mehrschichten-Transaktionsstrategie an letzter Stelle liegt. Ähnlich wie schon bei den Overhead-Messungen liegen die graphbasierte Strategie und die erweiterte Tickettechnik sehr nahe an der Pseudostrategie NONE. Insgesamt sinkt der erzielbare Durchsatz mit steigender Last auf dem System, weil zunehmend mehr Aufwand für die Bearbeitung der Investment-Transaktionen notwendig ist.

5.3 Strategien zur globalen Snapshot Isolation

In diesem Abschnitt evaluieren wir die Strategien zur Gewährleistung globaler Snapshot Isolation, die wir in Abschnitt 3.8 entwickelt haben, also die Synchronisation der Subtransaktionen und den optimistischen SI-Test. Auch hier bestimmen wir zusätzlich die Performance der Pseudostrategie "NONE", deren Ergebnisse eine Schranke für die bestenfalls erzielbare Leistung sind.

Wir bestimmen in Abschnitt 5.3.1 zunächst den Overhead, der für die Anwendung der einzelnen Strategien notwendig ist. Anschließend betrachten wir in Abschnitt 5.3.2 die Performance der Strategien in lesedominierten Anwendungen und in Abschnitt 5.3.3 die Performance in updatedominierten Anwendungen.

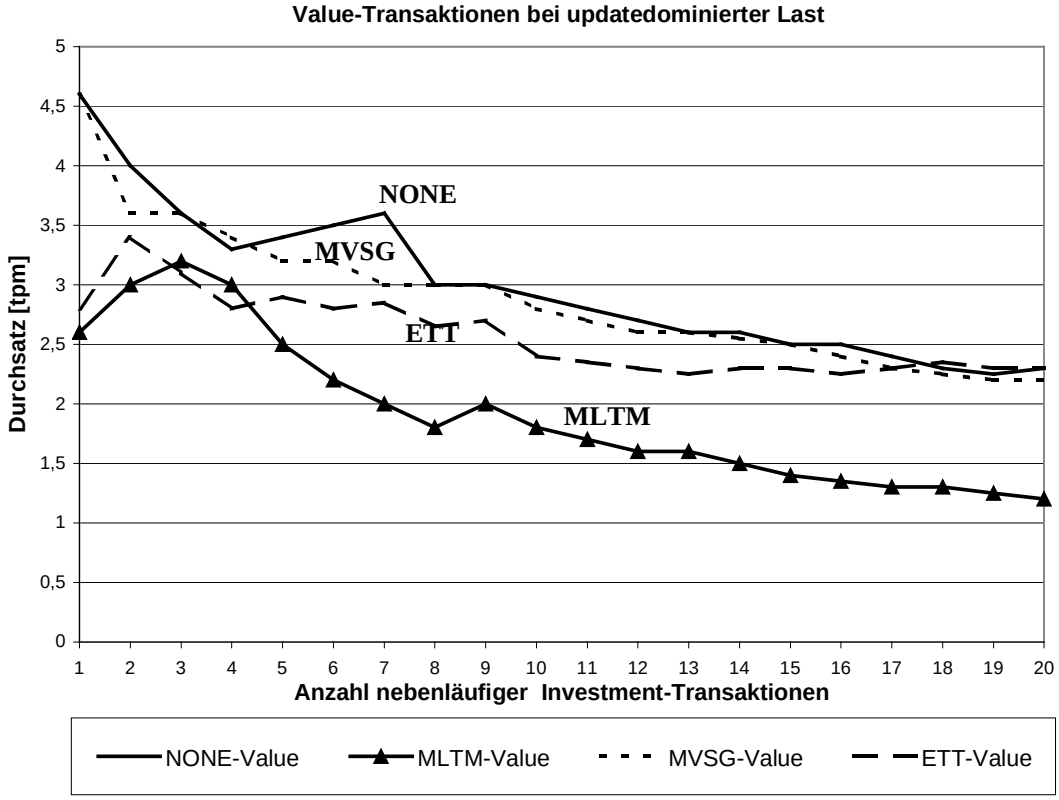

Abbildung 5.8 - Durchsatz von Value-Transaktionen bei updatedominierter Last

5.3.1 Overhead

Mit den Messungen in diesem Abschnitt bestimmen wir den Overhead, der durch die Anwendung der einzelnen Strategien eingeführt wird. Wir parametrisieren dazu die Transaktionsprogramme so, dass keine Konflikte auf Daten in den Datenbanken entstehen können. Der erzielte Durchsatz hängt also ausschließlich vom notwendigen Aufwand auf der föderierten Ebene ab.

Abbildung 5.9 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.10 den erzielbaren Durchsatz für Value-Transaktionen mit der Pseudostrategie NONE, der Synchronisation der Subtransaktionen (SST) und dem optimistischen SI-Test (OSI).

Sowohl für Investment- als auch für Value-Transaktionen ergibt sich ein leichter Performancevorteil für die Synchronisation der Subtransaktionen, der allerdings in der Praxis kaum ins Gewicht fällt. Der Durchsatz von Value-Transaktionen beider Strategien ist sehr nahe am Durchsatz der Pseudostrategie NONE, weil sie im Verhältnis zur Länge der Value-Transaktionen nur wenig zusätzliche Operationen ausführen. Der erzielbare Durchsatz von Investment-Transaktionen beider Strategien ist relativ weit von der Leistung der Pseudostrategie NONE entfernt, weil durch die kurzen Investment-Transaktionen die zusätzlichen Operationen schwerer wiegen.

Vergleicht man die absoluten Messwerte mit denen der Techniken, die serialisierbare Schedules erzeugen, erkennt man, dass der mögliche Durchsatz für Value-Transaktionen bei den besten Strategien etwa gleich ist. Bei Investment-Transaktionen

dagegen bleiben die Strategien für globale Snapshot Isolation leicht hinter der graph-basierten Strategie für globale Serialisierbarkeit zurück.

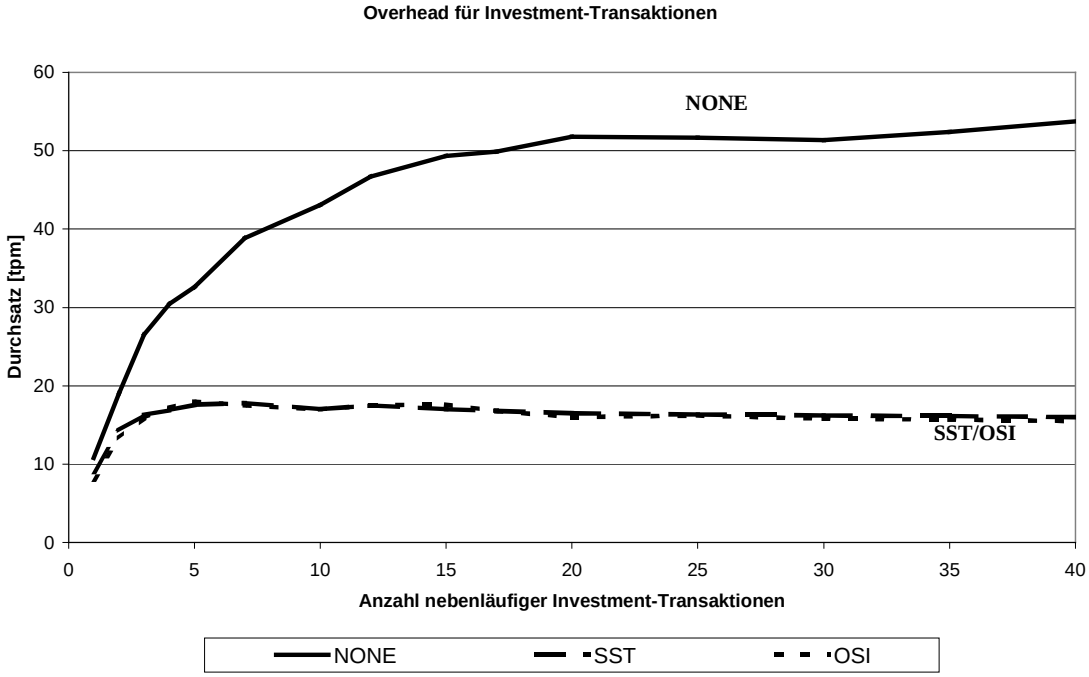

Abbildung 5.9 - Overhead für Investment-Transaktionen

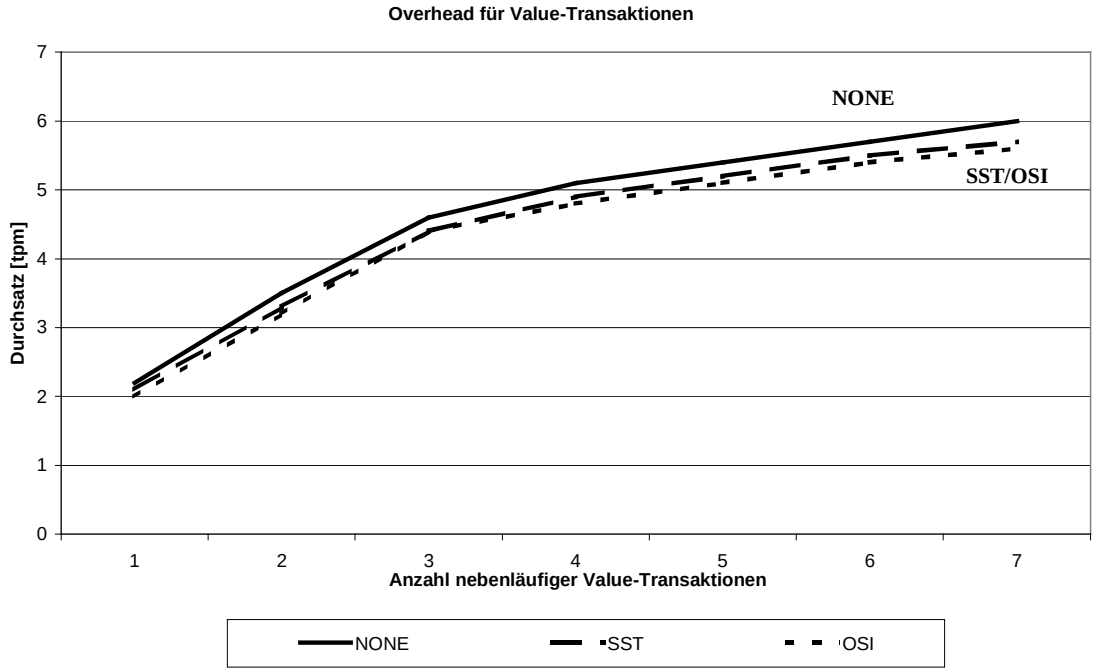

Abbildung 5.10 - Overhead für Value-Transaktionen

5.3.2 Performance in lesedominierten Anwendungen

In diesem Abschnitt beurteilen wir die Leistungsfähigkeit der Strategien in Anwendungen, die von reinen Lesetransaktionen dominiert werden. Wir halten dazu in allen Messungen eine Hintergrundlast von fünf Investment-Transaktionen konstant und variieren die gleichzeitig aktiven Value-Transaktionen von eins bis 20. Damit dabei Konflikte auf den Daten eine Rolle spielen, verwenden für diese Messungen alle Transaktionen nur Parameter aus einem eingeschränkten Bereich: `Customerid` und `Stockid` werden gleichverteilt aus dem Intervall [1;10] gewählt.

Abbildung 5.11 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.12 den erzielbaren Durchsatz von Value-Transaktionen mit der Pseudostrategie NONE, der Synchronisation der Subtransaktionen (SST) und dem optimistischen SI-Test (OSI).

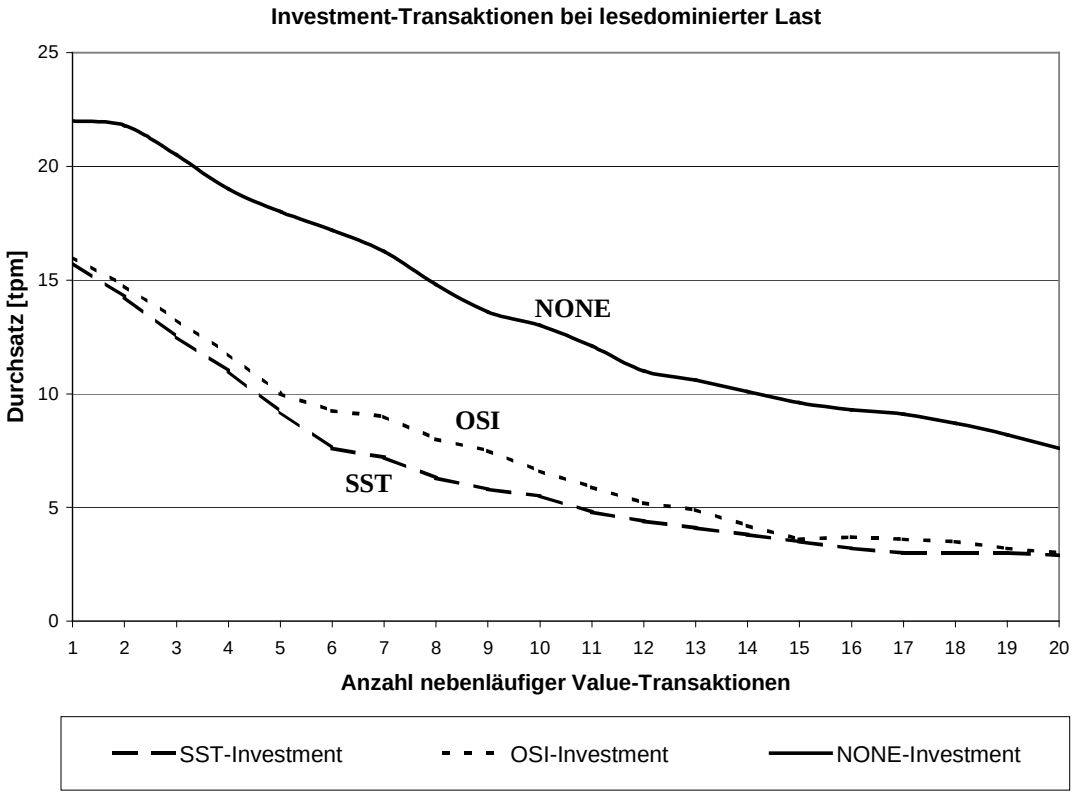

Abbildung 5.11 - Durchsatz von Investment-Transaktionen unter lesedominierter Last

Der Durchsatz von Value-Transaktionen ist für beide Strategien praktisch identisch, mit einem geringfügigen Vorteil für die Synchronisation der Subtransaktionen oberhalb einer gewissen Schwelle. Dies liegt an der Last auf der globalen OSI-Tabelle, die mit steigender Zahl paralleler Value-Transaktionen zunimmt, so dass die Einträge und Tests in dieser Tabelle länger dauern.

Die Investment-Transaktionen werden grundsätzlich mit steigender Value-Last immer langsamer bearbeitet, weil die Gesamtlast auf dem System steigt. Die optimistische SI-Technik erlaubt dabei einen leicht besseren Durchsatz für diesen Transaktionstyp, weil sie Schreib-Schreib-Konflikte nebenläufiger Transaktionen bereits vor dem Commit der Transaktion erkennen kann, nämlich beim Eintragen in die OSI-Tabelle, und daher die

betreffende Transaktion sofort abbrechen kann. Werden die Subtransaktionen synchronisiert, wird dieser Fall erst vom lokalen Datenbanksystem beim Commit der Transaktion erkannt.

Im Vergleich zur besten Strategie für globale Serialisierbarkeit zeigen die beiden Strategien zur globalen Snapshot Isolation einen vergleichbaren Durchsatz beider Transaktionstypen.

5.3.3 Performance in updatedominierten Anwendungen

In diesem Abschnitt beurteilen wir die Leistungsfähigkeit der Strategien in Anwendungen, die von Transaktionen dominiert werden, die Änderungen vornehmen. Wir halten dazu in allen Messungen eine Hintergrundlast von fünf Value-Transaktionen konstant und variieren die gleichzeitig aktiven Investment-Transaktionen von eins bis 20. Auch hier schränken wir den Parameterbereich wie für die Messungen in Abschnitt 5.3.2 ein.

Abbildung 5.13 zeigt den erzielbaren Durchsatz von Investment-Transaktionen, Abbildung 5.14 den erzielbaren Durchsatz von Value-Transaktionen mit der Pseudostrategie NONE, der Synchronisation der Subtransaktionen (SST) und dem optimistischen SI-Test (OSI).

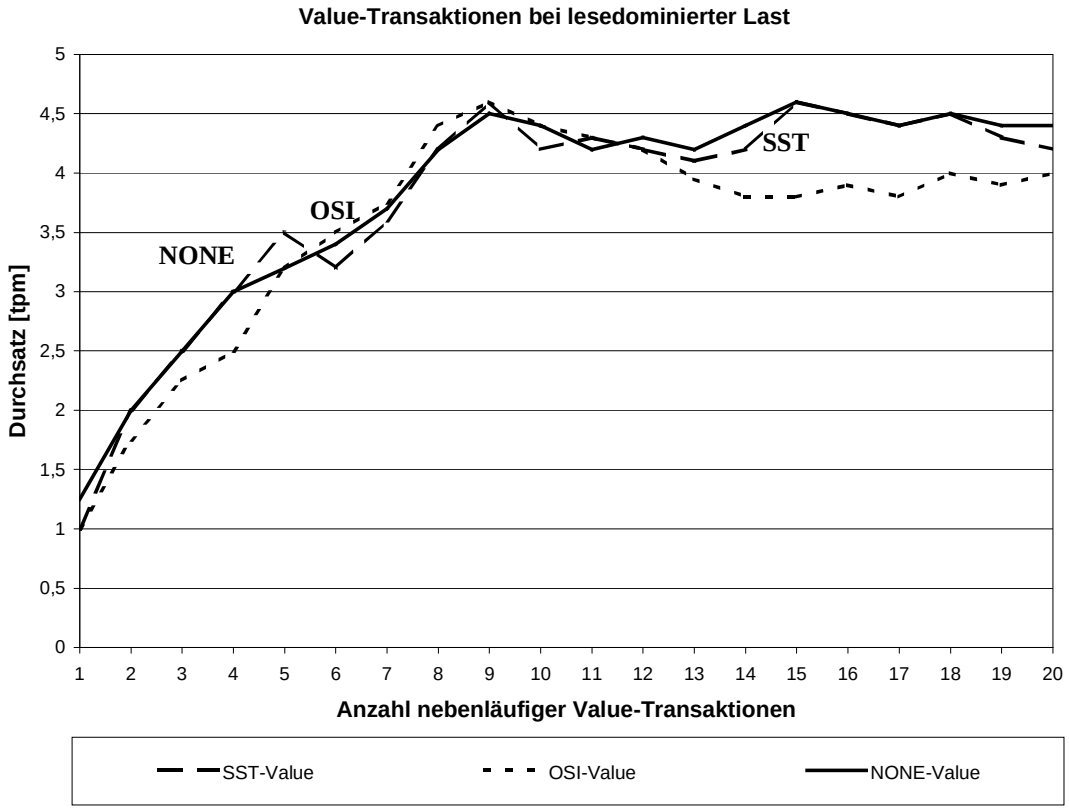

Abbildung 5.12 - Durchsatz von Value-Transaktionen unter lesedominierter Last

Der Durchsatz von Value-Transaktionen nimmt mit steigender Investment-Last ab, weil die Gesamtlast auf dem System zunimmt. Oberhalb einer gewissen Schwelle erlaubt die Synchronisation der Subtransaktionen einen geringfügig besseren Durchsatz als die op-

timistische Technik. Dies liegt wieder an der steigenden Last auf der globalen OSI-Tabelle mit steigender Zahl paralleler Investment-Transaktionen.

Für die Investment-Transaktionen erlaubt die optimistische SI-Technik dabei einen leicht besseren Durchsatz, weil sie wie schon in der lesedominierten Anwendung Schreib-Schreib-Konflikte nebenläufiger Transaktionen bereits vor dem Commit der Transaktion erkennen kann.

Auch hier zeigen die beiden Strategien für globale Snapshot Isolation im Vergleich zur besten Strategie für globale Serialisierbarkeit einen vergleichbaren Durchsatz beider Transaktionstypen.

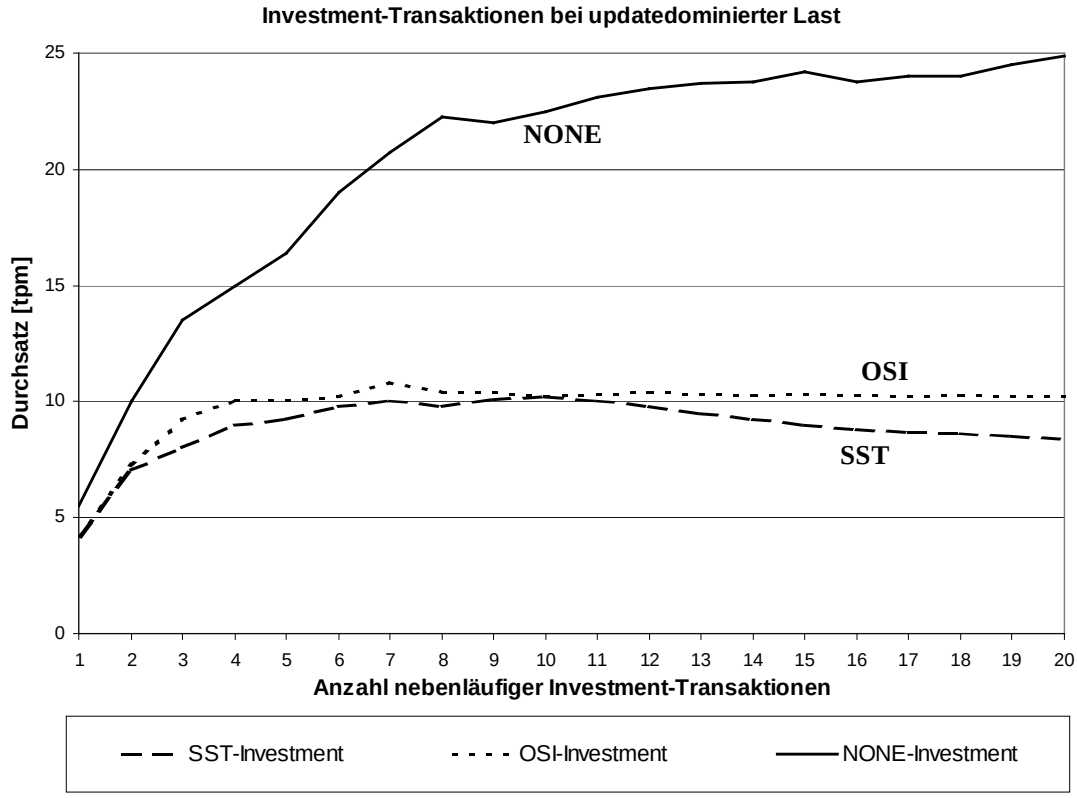

Abbildung 5.13 - Durchsatz von Investment-Transaktionen bei updatedominierter Last

5.4 Zusammenfassung

Wir haben in diesem Kapitel anhand von Benchmarks, die in unserem Prototypsystem ausgeführt wurden, die praktische Einsetzbarkeit der Strategien zur föderierten Concurrency Control gezeigt, die wir in Kapitel 3 entwickelt haben. Dabei hat sich herausgestellt, dass die graphbasierte Strategie unter allen Strategien zur Gewährleistung globaler Serialisierbarkeit unter niedriger bis mittlerer Last den höchsten Durchsatz erlaubt, und zwar sowohl in update- als auch in lesedominierten Anwendungen. Unter hoher Last spielen die erweiterte Tickettechnik in lesedominierten Anwendungen und die Mehrschichten-Transaktionsstrategie in updatedominierten Anwendungen ihre Stärken aus.

Die beiden Strategien, die globale Snapshot Isolation garantieren, erlauben in allen Messungen jeweils ungefähr den gleichen Durchsatz. Die optimistische Strategie hat dabei kleine Vorteile bei Transaktionen, die Änderungen vornehmen, während die Synchronisation der Subtransaktionen für reine Lesetransaktionen geringfügig besser geeignet ist. Im Vergleich zu den besten Techniken, die Serialisierbarkeit garantieren, erlauben die Strategien zur globalen Snapshot Isolation ungefähr den gleichen Durchsatz. Der Verlust gewisser Konsistenzgarantien durch den Einsatz eines Isolation Levels auf der föderierten Ebene bringt also für diese Anwendungssituation keinen Performancegewinn. In der Praxis muss man daher im Einzelfall prüfen, ob föderierte Snapshot Isolation oder globale Serialisierbarkeit im Hinblick auf Performance und Konsistenzgarantien die bessere Wahl ist.

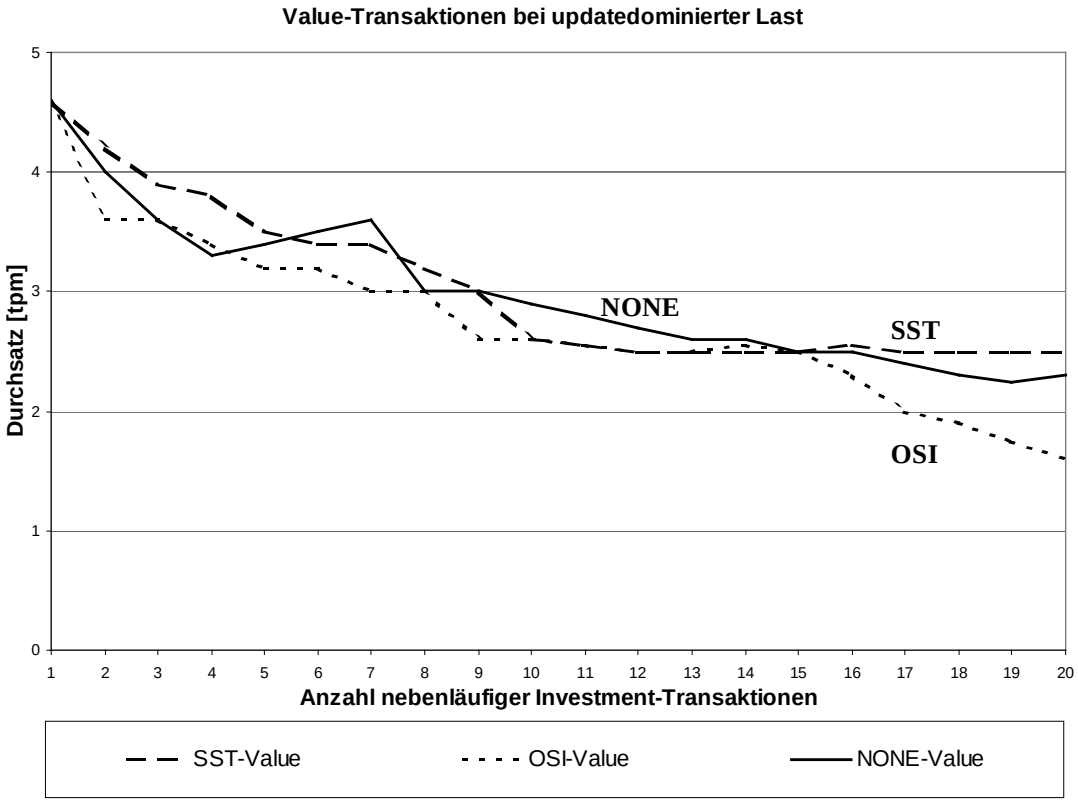

Abbildung 5.14 - Durchsatz von Value-Transaktionen unter updatedominierter Last

6 Zusammenfassung und Ausblick

Bis heute verstehen kommerzielle Systeme zur Föderation heterogener Datenbanken unter der Unterstützung föderierter Transaktionen ausschließlich die Gewährleistung der Atomarität von Transaktionen. Sie setzen dazu das verbreitete Zweiphasen-Commitprotokoll ein, scheitern jedoch häufig an Systemen, die keine native Unterstützung für dieses Protokoll anbieten. Isolation föderierter Transaktionen kann kein System vollständig garantieren, sobald verschiedene Datenbanksysteme involviert sind. Dies liegt vermutlich daran, dass für viele Applikationen eine lose Kopplung der Komponentensysteme ohne weitreichende Konsistenzgarantien ausreichend ist. Moderne Anwendungen, speziell aus dem Bereich des E-Commerce und der weiterreichenden E-Services, sind aber auf konsistente Daten angewiesen. Die vorhandenen föderierten Datenbanksysteme unterstützen diese Anwendungen nicht, sondern verlagern die Verantwortung für die Erhaltung der Konsistenz der verteilten Daten auf Administratoren und Anwendungsprogrammierer, die alle möglichen Probleme vorab erkennen und durch "manuelle" Eingriffe wie das explizite Anfordern von Sperren beheben müssen. Obwohl in der Literatur zahlreiche Verfahren vorgeschlagen wurden, um globale Serialisierbarkeit automatisch sicherzustellen, hat keines Eingang in kommerzielle Systeme gefunden. Dies liegt daran, dass die Verfahren zu starke Voraussetzungen an die Datenbanksysteme machen, die an einer Föderation beteiligt sind. Insbesondere fordern sie mindestens lokale Serialisierbarkeit, ignorieren dabei aber vollständig, dass in realen Anwendungen häufig schwächere Isolation Levels zum Einsatz kommen. Der Marktführer Oracle gewinnt einen Teil seiner hohen Performance sogar dadurch, dass überhaupt keine serialisierbare Ausführung gewährleistet, sondern lediglich Snapshot Isolation erreicht wird. In vielen realen Anwendungsfällen ist es daher nicht möglich, ohne – notwendigerweise unsystematische – Zusatzmaßnahmen überhaupt die Isolation föderierter Transaktionen und damit auch die Konsistenz der verteilten Daten sicherzustellen; dabei ist nie gewährleistet, dass eine noch so gut durchdachte Lösung für ein Anwendungsszenario wirklich in allen Fällen fehlerfrei arbeitet, wenn die Anwendung langfristig weiterentwickelt wird.

Im Gegensatz zu diesem bisher üblichen Weg haben wir in dieser Arbeit theoretisch fundierte und beweisbar korrekte Algorithmen vorgestellt, die in jeder Anwendungssituation die Serialisierbarkeit (und die Atomarität) föderierter Transaktionen sicherstellen, und zwar auch dann, wenn einige der Komponentensysteme nur schwächere Isolation Levels unterstützen. Wir haben dazu zunächst verschiedene formale Definitionen für Isolation Levels aus der Literatur verglichen und anschließend für den besonders wichtigen Fall der Snapshot Isolation eine neuartige Charakterisierung auf Basis einer Graphstruktur entwickelt. Aufbauend auf diesen Ergebnissen haben wir Algorithmen zur föderierten Concurrency Control vorgestellt: Zunächst haben wir die etablierte Tickettechnik von Georgakopoulos et. al. so erweitert, dass sie auch unter lokaler Snapshot Isolation korrekte Ergebnisse liefert. Wir haben gezeigt, dass unter bestimmten Voraussetzungen auch Mehrschichtentransaktionen korrekte Ergebnisse liefern, wenn einige Komponentensyteme Snapshot Isolation garantieren. Zusätzlich haben wir eine neue Technik vorgestellt, die auf einem Zyklustest in einer globalen Variante der graphbasierten Charakterisierung von Snapshot Isolation beruht.

Auch für föderierte Transaktionen kann man daran interessiert sein, durch den Einsatz globaler Isolation Levels Performance unter Verlust gewisser Konsistenzgarantien zu

gewinnen. Wir haben dazu gezeigt, dass sich die Isolation Levels Read Uncommitted, Read Committed und Repeatable Read ohne zusätzlichen Aufwand auf die föderierte Ebene übertragen lassen, wenn alle Komponentensysteme denselben Level unterstützen. Für Snapshot Isolation gilt dies nicht, es sind zusätzliche Algorithmen notwendig. Wir haben dazu zwei Verfahren entwickelt: Ein Verfahren, dass die Startzeitpunkte aller Subtransaktionen einer föderierten Transaktion synchronisiert, und ein Verfahren, dass Verletzungen der Snapshot-Isolation-Eigenschaft des globalen Schedules erkennt und die verursachende Transaktion zurücksetzt.

Um die praktische Einsetzbarkeit unserer Algorithmen zu zeigen, haben wir die vorgestellten Techniken in ein prototypisches föderiertes Datenbanksystem integriert. Umfangreiche Messungen haben ergeben, dass der zusätzliche Aufwand, der für die Gewährleistung der Isolation föderierter Transaktionen unter lokaler Snapshot Isolation notwendig ist, erträglich bleibt. Im einzelnen hat sich für Anwendungsszenarien, die durch reine Lesetransaktionen dominiert werden, die erweiterte Tickettechnik als performanteste Technik herausgestellt, während in gemischten Anwendungsumgebungen die neuentwickelte graphbasierte Technik die beste Leistung erzielt hat.

Wir haben mit dieser Arbeit wichtige Grundlagen gelegt, um Föderationen aus Komponentensystemen aufbauen zu können, die sowohl serialisierbare Schedules als auch Isolation Levels unterstützen. Einige interessante Aspekte konnten wir dabei nicht berücksichtigen, sie sollten aber im Hinblick auf praktische Einsetzbarkeit weiter untersucht werden.

In der Praxis ist man oft nicht nur daran interessiert, einen Isolation Level für alle Anwendungen zu benutzen. Stattdessen gibt es Transaktionsklassen, für die Read Committed ausreicht, während andere mindestens Snapshot Isolation benötigen, um korrekte Ergebnisse zu liefern. Im IsoTest-Projekt versuchen Patrick O'Neil et al [OO97, OOSF00], Kriterien abzuleiten, mit denen man automatisch den mindestens notwendigen Isolation Level für ein Transaktionsprogramm ableiten kann. Für föderierte Transaktionen stellt sich dann die Frage, ob sich diese Ergebnisse auch auf diesen Fall übertragen lassen oder ob, ähnlich wie bei globaler Snapshot Isolation oder Serialisierbarkeit, zusätzlicher Aufwand auf der föderierten Ebene notwendig ist.

Ähnlich wie die Wahl des „besten" Isolation Levels ist auch die Auswahl einer geeigneten Strategie zur föderierten Concurrency Control von den Anwendungsprogrammen abhängig. Wünschenswert wäre es dabei, verschiedene Algorithmen kombinieren zu können, so dass jede Anwendung das für ihr Lastprofil beste Verfahren verwenden kann. Wir haben erste Ansätze dazu bereits in Abschnitt 3.6.4 gemacht, allerdings haben wir uns dort darauf konzentriert, das Verfahren passend zu den jeweiligen Komponentensysteme auszuwählen. Zu untersuchen bleiben zum einen Kombinationen mit Mehrschichtenverfahren, zum anderen die Kombination verschiedener Verfahren für Transaktionsklassen, zu der es bisher noch keinerlei Arbeiten gibt.

Besonders interessant für die Praxis schließlich wäre die Möglichkeit, für eine gegebene Menge von Transaktionsprogrammen und ein Lastprofil anhand von automatischen Benchmarks oder Simulationen die „beste" Strategie für diese Situation zu bestimmen. Auf diese Weise wäre man nicht mehr alleine auf einen Administrator angewiesen, sondern das System selbst könnte geeignete Parametereinstellungen finden.

Literatur

[AAE94] G. Alonso, D. Agrawal, A. El Abbadi: *Reducing Recovery Constraints on Locking-Based Protocols*. In: Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1994.

[ABJ97] V. Atluri, E. Bertino, S. Jajodia: *A Theoretical Formulation for Degrees of Isolation in Databases*. In: Information and Software Technology 39(1), Elsevier Science, 1997.

[Adya99] A. Adya: *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. Thesis, MIT, Cambridge, März 1999.

[AGS87] R. Alonso, H. Garcia-Molina, K. Salem: *Concurrency Control and Recovery for Global Procedures in Federated Database Systems*. In: Data Engineering 10(3), 1987.

[ALO00] A. Adya, B. Liskov, P. O'Neil: *Generalized Isolation Level Definitions.* In: Proceedings of the 16th International Conference on Data Engineering (ICDE), San Diego, CA. IEEE Computer Society Press, 2000.

[BBGM+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil*: A Critique of ANSI SQL Isolation Levels*. In: Proceedings of the 1995 ACM-SIGMOD International Conference on Management of Data, San Jose, CA, 1995.

[BE96] O.A. Bukhres, A.K. Elmagarmid (eds): *Object Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice Hall, 1996.

[BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewciz, A. Silberschatz: *On Rigorous Transaction Scheduling*. In: IEEE Transactions on Software Engineering 17(9), 1991.

[BGS92] Y. Breitbart, H. Garcia-Molina, A. Silberschatz: *Overview of Multidatabase Transaction Management*. In: VLDB Journal 1(2), 1992.

[BGS95] Y. Breitbart, H. Garcia-Molina, A. Silberschatz: *Transaction Management in Multidatabase Systems*. In: W. Kim (Ed.): *Modern Database Systems*, ACM Press, 1995.

[BHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman*: Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[BLL00] A.J. Bernstein, P.M. Lewis, S. Lu: *Semantic Conditions for Correctness at Different Isolation Levels.* In: Proceedings of the 16th International Conference on Data Engineering (ICDE), San Diego, CA, März 2000.

[BN97] P.A. Bernstein, E. Newcomer: *Principles of Transaction Processing.* Morgan Kaufmann, 1997.

[BS88] Y. Breitbart, A. Silberschatz: *Multidatabase Update Issues*. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, IL, 1988.

[BS92] Y. Breitbart, A. Silberschatz: *Strong Recoverability in Multidatabase Systems*. In: Prodeedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, Tempe, 1992.

[BST92] Y. Breitbart, A. Silberschatz, G.R. Thompson: *Transaction Management Issues in a Failure-Prone Environment*. In: VLDB Journal 1(1), 1992.

[BSW79] A.C. Bernstein, D.W. Shipman, W.S. Wong: *Formal Aspects of Serializability in Database Concurrency Control*. In: IEEE Transactions on Software Engineering SE-5, 1979.

[CBS93] J. Chen, A. Bukhres, J. Sharif-Askary: *A Customized Multidatabase Transaction Management Strategy*. In: Proceedings of the 4th Conference on Database and Expert Systems Applications (DEXA), 1993.

[CFL+82] A. Chan, S. Fox, W.K. Lin, A. Nori, D.R. Ries: *The Implementation of an Integrated Concurrency Control and Recovery Scheme*. In: Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, 1982.

[CHHK+99] S. Conrad, W. Hasselbring, U. Hohenstein, R.-D. Kutsche, M. Roantree, G. Saake, F. Saltor: *Engineering Federated Information Systems: Report of EFIS '99 Workshop*. In: SIGMOD Record 28(3), September 1999.

[Conr97] S. Conrad: *Föderierte Datenbanksysteme*. Springer Verlag, 1997.

[CSS99] S. Conrad, G. Saake, K.-U. Sattler: *Informationsfusion – Herausforderung an die Datenbanktechnologie*, in: A. P. Buchmann (Hrsg.): Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), GI-Fachtagung, Freiburg, 1.-3. März 1999.

[Date90] C. J. Date: *An Introduction to Database Systems*. Fifth Edition, Addison-Wesley, 1990.

[DE90] W. Du, A. Elmagarmid: *A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems*. In: Proceedings of the 6th IEEE International Conference on Data Engineering. Los Angeles, CA. IEEE Computer Society Press, 1990.

[DE00] Data Engineering Bulletin, *Special Issue on Database Technology in Electronic Commerce*, IEEE Computer Society, März 2000.

[DSW94] A. Deacon, H.-J. Schek, G. Weikum: *Semantics-based Multilevel Transaction Management in Federated Systems*. Proceedings of the 10th IEEE International Conference on Data Engineering, Houston, 1994.

[DuBou82] D. DuBourdieu: *Implementation of Distributed Transactions*. In: Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, 1982.

[Elma92] A.K. Elmagarmid (Ed.): *Database Transaction Models For Advanced Applications*, Morgan Kaufmann Publishers, 1992.

[Feke99] A. Fekete: *Serializability and Snapshot Isolation*. In: Proceedings of the 10th Australasian Database Conference, Auckland, New Zealand, 1999.

[FLO+01] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha: *Making Snapshot Isolation Data Item Serializable*, submitted for publication, URL: http://www.cs.umb.edu/~isotest/snaptest/snaptest.pdf, 2001.

[Gray78] J. Gray: *Notes on Database Operating Systems.* In: Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60:393-481. Springer Verlag, Berlin, 1978.

[GLPT76] J. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger: *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. IFIP Working Conference on Modelling in Data Base Management Systems, 1976.

[GR93] J.N. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GRS91] D. Georgakopoulos, M. Rusinkiewicz, A.P. Shet: *On Serializability of Multidatabase Transactions Through Forced Local Conflicts*. In: Proceedings of the 7th International Conference on Data Engineering. Kobe, 1991.

[GRS94] D. Georgakopoulos, M. Rusinkiewicz, A.P. Shet*: Using Tickets to Enforce the Serializability of Multidatabase Transactions*. In: IEEE Transactions on Knowledge and Data Engineering 6(1), Februar 1994.

[GS87] H. Garcia-Molina, K. Salem: *Sagas*. In: Proceedings of the 1987 ACM SIGMOD Conference, San Francisco, 1987.

[HAD97] Ugur Halici, B. Arpinar, A. Dogac: *Serializability of Nested Transactions in Multidatabases*. In: Proceedings of the 13th International Conference on Data Engineering, Birmingham, 1997.

[Hadz88] V. Hadzilacos: *A Theory of Reliability in Database Systems*. In: Journal of the ACM 35, 1988.

[Härd84] T. Härder: *Observations on Optimistic Concurrency Control*. Information Systems 9, 1984.

[HBP93] A.R. Hurson, M.W. Bright, S.H. Pakzad: *Multidatabase Systems – Advanced Solution for Global Information Sharing*. In: IEEE Computer Society Press, 1993.

[HSC99] J.M. Hellerstein, M. Stonebraker, R. Caccia: *Independent, Open Enterprise Data Integration*. In: IEEE Data Engineering Bulletin 22(1), 1999.

[HSL94] S.-Y. Hwang, J. Srivastava, J. Li*: Transaction Recovery in Federated Autonomous Databases*. In: Distributed and Parallel Databases 2(2), 1994.

[HWW98] B. Holtkamp, N. Weissenberg, X. Wu: *VHDBS: A Federated Database System for Electronic Commerce*. EURO-MED NET 1998.

[IBM98] IBM Corp.: *DB2 DataJoiner® Administration Supplement Version 2.1.1*, Juli 1998.

[IBM00a] IBM Corp.: *IBM® DB2® Universal Database Administration Guide: Planning*, Version 7, 2000.

[IBM00b] IBM Corp.: *IBM® DB2® Universal Database Administration Guide: Performance*, Version 7, 2000.

[IBM00c] IBM Corp.: *IBM® DB2® Universal Database Application Development Guide*, Version 7, 2000.

[Info00a] Informix Software, Inc.: *Universe - SQL Administration for DBAs*, Version 9.6.1, Part No. 000-6948, Juli 2000.

[Info00b] Informix Corporation: *Universe - Basic*, Version 9.6, Dezember 2000.

[Info00c] Informix Software, Inc.: *UniVerse - Transaction Logging and Recovery*, Version 9.6.1, Part No. 000-6953, Juli 2000.

[Iona95] Iona Technologies: *The Orbix Architecture*, 1995.

[Iona97] Iona Technologies: *The Object Transaction Service*. White Paper, 1997.

[Iona99a] Iona Technologies: *Orbix C++ Programmer's Guide Release 3.0*, Dublin, Februar 1999.

[Iona99b] Iona Technologies: *OrbixOTS Programmer's and Administrator's Guide*, Release 3.0, Dublin, Februar 1999.

[KKG99] N. Krivokapic, A. Kemper, E. Gudes: *Deadlock detection in distributed database systems: a new algorithm and a comparatative performance analysis*. In: VLDB Journal 8(2), October 1999.

[Klee96] J. Kleewein: *Practical Issues with Commercial Use of Federated Databases*. In: Proceedings of the 22th Conference on Very Large Databases, Bombay, 1996.

[KR81] H.T. Kung, P.L. Lehman: *On Optimistic Methods for Concurrency Control*. ACM Transactions on Database Systems 6, 1981.

[Lang92] H. Langendörfer: *Leistungsanalyse von Rechensystemen*. Carl Hanser Verlag, München und Wien, 1992.

[Lehn96] T. Lehnecke: *Föderierte Transaktionsverwaltung – eine vergleichende Betrachtung aktueller Entwicklungen*. Diplomarbeit an der Otto-von-Guericke-Universität Magdeburg, August 1996.

[LKS91a] E. Levy, H.F. Korth, A. Silberschatz: *A Theory of Relaxed Atomicity*. In: Proceedings of the ACM Symposium of the Principles of Distributed Computing, 1991.

[LKS91b] E. Levy, H.F. Korth, A. Silberschatz: *An Optimistic Commit Protocol for Distributed Transaction Management*. In: Proceedings of ACM-SIGMOD International Conference on Management of Data, Denver, CO, 1991.

[LMR90] W. Litwin, L. Mark, N. Roussopoulos: *Interoperability of Multiple Autonomous Databases*. ACM Computing Surveys, Vol. 22, 1990.

[Lome94] D. Lomet: *Private Locking and Distributed Cache Management*. In: Proceedings of the 3rd Int. Conference on Parallel and Distributed Information Systems. Austin, 1994.

[LS76] B. Lampson, H. Sturgis: *Crash Recovery in a Distributed Data Storage System*. Technical Report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, CA, 1976.

[LSGG+79] B.G. Lindsay P.G. Selinger, C. Galtieri, J.N. Gray, R.A. Lorie, T.G. Price, F. Putzolu, B.W. Wade: *Notes on Distributed Databases*, IBM Research Report RH2561, San Jose, CA, 1979.

[Micr99] Microsoft Corp.: *SQL Server 2000 Online Documentation*, 1999.

[MKRZ99] N. M. Mattos, J. Kleewein, M. T. Roth, K. Zeidenstein*: From Object-Relational to Federated Databases*. Invited Paper, in: A. P. Buchmann (Ed.): Datenbanksysteme in Büro, Technik und Wissenschaft *(BTW)*, Freiburg, 1999.

[MR91] P. Muth, T. Rakow: *Atomic Commitment for Integrated Database Systems.* In: Proceedings of the 7th IEEE International Conference on Data Engineering, Kobe, 1991.

[MRBK+92] S. Mehrotra, R. Rastogi, Y Breitbart, H.F. Korth, A. Silberschatz: *The Concurrency Control Problem in Multidatabases: Characteristics and Solutions*. In: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data. San Diego, CA, ACM Press, Juni 1992.

[Muth97] P. Muth: *Application Specific Transaction Management in Multidatabase Systems*. In: Distributed and Parallel Databases 5(4), 1997.

[ODMG97] R.G.G. Cattell, D.K. Barry: *The Object Database Standard ODMG-2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[ÖV98] M.T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1998.

[OMG94] Object Management Group, Inc.: *Object Transaction Service*. OMG Document 94.8.4, 1994.

[OMG96] Object Management Group, Inc: *CORBAServices: Common Object Services Specification*, 1996.

[OMG97] Object Management Group, Inc: *Object Transaction Service 1.1*, 1997.

[OO97] P O'Neil, E. O'Neil: *Project Summary: Isolation Testing in Transactional Systems*. Project Proposal, University of Massachusetts, Boston, 1997.

[OOSF00] P. O'Neil, E. O'Neil, D. Shasha, A. Fekete: *Isolation Testing for Transactional Systems*, University of Massachusetts, Boston, Februar 2000.

[Orac95] Oracle Corporation: *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*. White Paper. 1995.

[Orac96] Oracle Corporation: *Oracle Transparent Gateway for Microsoft SQL Server Installation and User's Guide*, Release 4.0, 1996.

[Orac99a] Oracle Corporation: *Oracle8i Transparent Gateways*, White Paper, August 1999.

[Orac99b] Oracle Corporation: *Oracle8i Distributed Database Systems*, Release 2 (8.1.6), Part No. A76960-01, Dezember 1999.

[Orac99c] Oracle Corporation: *Oracle8i Concepts: Chapter 27, Data Concurrency and Consistency*, Release 2 (8.1.6), Part No. A76965-01, Dezember 1999.

[Papa79] C. Papadimitriou: *The Serializability of Concurrent Database Updates*. Journal of the ACM 26, 1979.

[Papa86] C. Papadimitriou: *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[Pu88] C. Pu: *Superdatabases for Composition of Heterogenous Databases*. In: Proceedings of the 4th IEEE International Conference on Data Engineering, Los Angeles, CA, 1988.

[Rahm94] E. Rahm: *Mehrrechner-Datenbanksysteme*. Addison-Wesley Publishing Company, 1994.

[Raz91] Y. Raz: *The Principle of Commit Ordering or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers*. Technical Report. Digital Equipment Corporation. 1991.

[Raz92] Y. Raz: *The Principle of Commit Ordering or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment*. In: Proceedings of the 18th International Conference on Very Large Databases, Vancouver ,August 1992.

[RDD99] P. Ram, L. Do, P. Drew: *Distributed Transactions in Practice*. In: SIGMOD Record 28(3), September 1999.

[Ripk98] T. Ripke: *Optimistische/Hybride Synchronisation in Mehrschichtsystemen*. 9. Workshop Transaktionen und ihre Anwendung. Warnemünde. 1998.

[Scha96] W. Schaad: *Transaktionsverwaltung in heterogenen, föderierten Datenbanksystemen.* Dissertation. ETH Zürich, 1996.

[SGA89] K. Salem, H. Garcia-Molina, R. Alonso: *Altruistic Locking: A Strategy for Coping with Long Lived Transactions*. In: D. Gadwick, M. Haynie, A. Reuter (Eds.): Lecture Notes in Computer Sciences, High Performance Transaction Systems, Volume 359, Springer Verlag NY, 1989.

[SGS94] K. Salem, H. Garcia-Molina, J. Shands: *Altruistic Locking*. In: ACM Transactions on Database Systems 19(1), 1994.

[SL90] A.P. Sheth, J.A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*. In: ACM Computing Surveys 22(2), 1990.

[SS93] W. Schaad, H.-J. Schek: *Federated Transaction Management Using Open Nested Transactions*. In: Proceedings of the DBTA Workshop on Interoperability of Database Systems and Database Applications, Fribourg, 1993.

[SSW95] W. Schaad, H.-J. Schek, G. Weikum: *Implementation and Performance of Multi-level Transaction Management in a Multidatabase Environment*. In: Proceedings of the 5th IEEE International Workshop on Research Issues in Data Engineering, Taipeh, 1995.

[Ston98] M. Stonebraker: *Are We Working On The Right Problems? (Panel)*. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, 1998.

[SW98] R. Schenkel, G. Weikum: *Transaktionsverwaltung in Föderativen Informationssystemen auf der Basis von CORBA (TRAFIC)*. Entwurfsdokument, Mai 1998.

[SW99a] R. Schenkel, G. Weikum: *Transaktionsverwaltung in Föderativen Informationssystemen auf der Basis von CORBA (TRAFIC)*. Dokumentation der Implementierung, März 1999.

[SW99b] R. Schenkel, G. Weikum: *Experiences with Building a Federated Transaction Manager based on CORBA OTS*. In: Proceedings of the 2nd International Workshop of Engineering Federated Information Systems (EFIS '99), Kühlungsborn, Mai 1999.

[SW00a] R. Schenkel, G. Weikum*: Integrating Snapshot Isolation Into Transactional Federations*. In: Proceedings of the 5th IFCIS International Conference on Cooperative Information Systems (CoopIS 2000), Eilat, Israel, September 6-8, 2000. Lecture Notes in Computer Science Vol. 1901, Springer, 2000.

[SW00b] R. Schenkel, G. Weikum: *Taming the Tiger: How to Cope With Real Database Products in Transactional Federations for Internet Applications*. Workshop "Internet-Datenbanken", Informatik 2000, Berlin, 19.-22. September 2000.

[SWS91] H.-J. Schek, G. Weikum, W.A. Schaad: *A Multilevel Transaction Approach to Federated DBMS Transaction Management*. In: Proceedings of the International Workshop on Interoperability in Multidatabase Systems, Kyoto, 1991.

[SWW+99] R. Schenkel, G. Weikum, N. Weißenberg, X. Wu*: Federated Transaction Management With Snapshot Isolation*. In: Proceedings of the 8th International Workshop on Foundations of Models and Language for Data and Objects – Transactions and Database Dynamics, Schloß Dagstuhl, 1999. Lecture Notes in Computer Science Vol. 1773, Springer, 1999.

[SWY93] H.-J. Schek, G. Weikum, H. Ye: *Towards a Unified Theory of Concurrency Control and Recovery*. In: Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1993.

[Syba97a] Sybase, Inc: *Sybase® Adaptive Server™ Enterprise Performance and Tuning Guide*, Emeryville, CA, September 1997.

[Syba97b] Sybase, Inc: *Sybase® Adaptive Server™ Enterprise System Administration Guide*, Emeryville, CA, September 1997.

[TW97] T. Tesch, J. Wäsch: *Global Nested Transaction Management for ODMG-Compliant Multidatabase Systems*. Arbeitspapiere der GMD, GMD Technical Report No. 1071, 1997.

[Unid98] Unidata, Inc.: *O_2Engine API Reference Manual*, Release 5.0, März 1998.

[VEH92] J. Veijalainen, F. Eliassen, B. Holtkamp: *The S-Transaction Model*. In: [Elma92]

[VW92] J. Veijalainen, A. Wolski: *Prepare and Commit Vertification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases*. In: Proceedings of the 8th International Conference on Data Engineering, Tempe, 1992.

[Weihl89] W.E. Weihl: *Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types*. In: ACM Transactions on Programming Languages and Systems, 11(2), 1989.

[Weik86] G. Weikum: *A Theoretical Foundation of Multi-Level Concurrency Control*. In: Proceedings of the 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1986.

[Weik91] G. Weikum: *Principles and Realization Strategies of Multilevel Transaction Management*. In: ACM Transactions on Database Systems 16(1), März 1991.

[Weiß97] N. Weißenberg: *Konzepte und Entwürfe zur Version 3.0 des VHDBS-Systems*. Fraunhofer ISST, Außenstelle Dortmund. November 1997.

[Wied92] G. Wiederhold: *Mediators in the Architecture of Future Information Systems*. In: IEEE Computer, 25(3), 1992.

[WS84] G. Weikum, H.-J. Schek: *Architectural Issues of Transaction Management in Layered Systems*. In: Proceedings of the 10th Conference on Very Large Data Bases, Paolo Alto, CA. 1984.

[WS92] G. Weikum, H.-J. Schek: *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. In: A.K. Elmagarmid (Ed.): Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Francisco, CA, 1992.

[WV91] A. Wolski, J. Veijalainen: *2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase*. In: N. Rishe, S. Navathe, D.Tal (eds.): *Databases: Theory, Design and Applications*. IEEE Computer Society, 1991.

[WV01] G. Weikum, G. Vossen: *Transactional Information Systems – Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[WW96] N. Weißenberg, X. Wu: *Konzepte und Entwürfe zur Version 2.0 des VHDBS-Systems*. Fraunhofer ISST, Außenstelle Dortmund. Dezember 1996.

[WW97] X. Wu, N. Weißenberg: *A Graphical Interface for Cooperative Access to Distributed and Heterogeneous Database Systems*. International Database Engineering and Applications Symposium, Montreal, 1997.

[Wu96] X. Wu: *An Architecture for Interoperation of Distributed Heterogenous Database Systems*. In: Proceedings of the International Conference on Database and Expert Systems Applications, Zürich. 1996.

[Xopen96] X/Open Company Ltd.: *Distributed Transaction Processing: Reference Model*, X/Open Guide, Version 3, 1996.

[Yanna84] M. Yannakakis: *Serializability by Locking*. In: Journal of the ACM 31, 1984.

[ZE93] A. Zhang, A.K. Elmagarmid: *A Theory of Global Concurrency Control in Multidatabase Systems*. In: VLDB Journal 2(3), 1993.

Index