DISS. ETH Nr. 12 348

Transaktionsorientierte Verwaltung und Suche von Dokumenten in einer Mehrprozessordatenbankumgebung

ABHANDLUNG
zur Erlangung des Titels

DOKTOR DER TECHNISCHEN WISSENSCHAFTEN
der
EIDGENÖSSISCHEN TECHNISCHEN HOCHSCHULE ZÜRICH

vorgelegt von

HELMUT KAUFMANN

Dipl. Informatik-Ing. ETH
geboren am 10. September 1967
von Österreich

Angenommen auf Antrag von
Prof. Dr. H.-J. Schek, Referent
Prof. Dr. G. Gonnet, Korreferent

1997

Den Menschen, die mich lieben

Inhaltsverzeichnis

Zusammenfassung

Datenbanksysteme sind traditionell auf eine effiziente transaktionsorientierte Verarbeitung von Standarddaten ausgelegt. Eine Verarbeitung von Volltextdaten ist mit Datenbanksystemen bis heute nicht oder nur unter teilweiser Aufgabe der ACID-Eigenschaften möglich, da die Indexierung von Volltextdaten typischerweise zu einem transaktionellen Leistungsengpass führt. Grund für diesen Engpass sind Pseudokonflikte, die bei Einsatz herkömmlicher Protokolle zur Mehrbenutzerverwaltung, beispielsweise 2PL, unvermeidlich sind. Konkret bedeutet dies, dass im Fall von Online-Updates mit massiven Behinderungen, d.h. zeitlichen Verzögerungen, parallel laufender Suchoperationen aufgrund von Konflikten auf der Ebene der textuellen Indexstrukturen zu rechnen ist. Im erste Teil der vorliegenden Arbeit wird auf diese Problematik eingehend eingegangen.

In einer Reihe von Forschungsarbeiten wurde in der Vergangenheit auf theoretischer Basis nachgewiesen, dass es durch den Einsatz neuer Transaktionsmodelle, insbesondere Mehrschichtentransaktionen, möglich ist, die oben genannten Leistungsengpässe zu entschärfen. Die theoretisch erhaltenen Resultate konnten anhand prototypischer Implementierungen, die meist von Grund auf neue Datenbanksysteme realisieren oder aber auf bestehenden Client-Server-Systemen basieren, verifiziert werden.

Im Laufe der letzten Jahre ist ein Trend weg von Client-Server-basierten Informationssystemen zu Systemen auf Basis sogenannter Middlewareprodukte, im OLTP-Bereich insbesondere Transaktionsmonitore (TP-Monitore), zu beobachten. Bei diesen Systemen arbeitet ein Benutzer nicht mehr direkt mit einer Datenbank, beispielsweise via SQL, sondern greift auf die Daten nur über sogenannte "Services" zu. Ein Beispiel für einen solchen Service wäre beispielsweise das Einfügen eines neuen Dokuments in die Datenbank oder das Suchen nach Dokumenten, welche bestimmte Wörter enthalten. Der Benutzer arbeitet somit mit semantisch reichen Operationen, die Implementierung der Services bleibt ihm verborgen. Eine Benutzertransaktion besteht in einer solchen Umgebung aus dem Aufruf einer Reihe von Services. Das Konzept des Arbeitens mit einem TP-Monitor weist grosse Ähnlichkeiten mit einem Datenbanksystem mit zweischichtiger Transaktionsverwaltung auf: Transaktionen auf der oberen Schicht bestehen aus Operationen auf der unteren Schicht. Services sind somit den semantisch reichen Operationen eines Mehrschichtensystems sehr ähnlich.

Die vorliegende Arbeit macht sich die Ähnlichkeit von TP-Monitoren und Mehrschichtendatenbanksystemen zunutze: Ausgehend von einem TP-Monitor und einen relationalen Datenbanksystem wird im zweiten Teil der Arbeit der TP-Monitor um eine generische Komponente (Scheduler, Recover-Komponente) erweitert, welche es ermöglicht, zweischichtige Transaktionen auszuführen. Konkret heisst dies, dass jeder Aufruf eines Services durch den Benutzer im Rahmen einer eigenständigen Transaktion des unterliegenden relationalen Datenbanksystems ausgeführt wird, wobei der erweiterte TP-Monior garantiert, dass sich die im unterliegenden

System gespeicherten Daten immer — auch im Fehlerfall — in einem konsistenten Zustand befinden. Die Erweiterung erfolgt dabei in Form von Services, d.h. es sind keine Änderungen am Code der eingesetzten Systeme notwendig. Auf Basis dieses generische System, TPM/ONT genannt, werden anschliessend Services zum Verwalten und Suchen von Volltextdokumenten und deren Indexstrukturen realisiert.

Im dritten Teil der Arbeit wird das im ersten Teil realisierte System in einer Umgebung mit bis zu 25 parallel arbeitenden Benutzern leistungsmässig untersucht. Es wird nachgewiesen, dass es mit Hilfe eines zweischichtigen Datenbanksystems möglich ist, Volltextdokumente transaktionsorientiert und unter voller Wahrung der ACID-Eigenschaften online zu verarbeiten. Insbesondere wird gezeigt, dass es mit Hilfe dieses Systems möglich ist, im laufenden Betrieb neue Dokumente einzufügen, ohne hierdurch parallel ablaufende Transaktionen leistungsmässig relevant zu behindern. Die Ergebnisse werden dabei jenen eines Systems mit einschichtiger Transaktionsverwaltung gegenübergestellt. Schlussendlich wird gezeigt, dass durch Ausnutzen von Intratransaktionsparallelität (insbesondere der parallelen Verarbeitung der Textindexstrukturen eines Dokuments) eine weitere Leistungssteigerung gegenüber der traditionellen, streng sequentiell ablaufenden Dokumentenverarbeitung möglich ist.

Abstract

Traditional database systems provide efficient transaction oriented processing of structured data. However, usually, processing of textual data in a database system has either no or only limited support. For example, while it may be possible to build index structures over textual data, in practice, the resulting bottlenecks make this impracticable. The reason for these bottlenecks are pseudo conflicts arising from traditional protocols for concurrency control, such as two-phase locking, which were designed to prevent conflict between transactions rather than within transactions. In the first part of this thesis, we discuss these problems of combining efficient textual processing with traditional database systems in detail.

Existing research proposals for advanced transaction models, such as multi-level transactions, provide a theoretical basis for reducing the level of conflicts in a database system. The potential for adopting such protocols to solve the problem of online updates of indexed textual data has previously been validated, but only in a research environment. This has been done either by building new prototype database systems or by extending existing client–server database systems with a preprocessor for textual data. However, over the last few years, there has been a shift in database technology: In the past, applications were mostly based on traditional client–server technology. Today, applications are often based on middleware products. Especially in the area of online transaction processing, we have noticed an increasing number of applications built using transaction processing monitors (TP monitors). In a system based on TP monitors, users do not directly interact with the database system using, for example, SQL. Instead, they interface the database system using so-called "services". In the context of textual data, typical services are the insertion of a textual document including the updates of the textual index structures or the search for all documents containing certain keywords. Of course, TP monitors provide full transactional support. In particular, it is possible to combine a number of service calls to a single transaction. If we compare the concepts of multi-level database systems, in particular a two level system, with TP monitors, we find that these concepts are very similar: In a two level database system, a user transaction consists of the execution of a number of higher level operations, each of which is coded using lower level operations. In a TP monitor environment, a user transaction is made up of a number of service calls. Each of these services consist of a number of calls to the underlying database system. Therefore, one could say that the higher level operations of a two level database system are very similar to the services of a TP monitor and lower level operations are similar to the calls of the database system.

In the second part of this thesis, we show how the similarity of multi-level database systems and TP monitors can be exploited in order to build a fully operational multi-level database system: Based on commercial relational database systems (Oracle and Sybase) as well as a commercial TP monitor (Tuxedo), the TP monitor is extended with a generic higher-level transaction manager. The resulting transaction manager provides support for the correct execution of two-

level transactions. This means that a higher-level operation maps to a service call, which is executed as an independent transaction of the underlying database system. The consistency of the data stored in the database is guaranteed — also in the case of a system failure — by the higher level transaction manager. It is worth noting, that the extension is entirely based on services, i.e. it is neither necessary to change the code of the TP monitor, nor that of the database system. On the basis of this system, called TPM/ONT (Transaction Processing Monitor with Support for Open Nested Transactions), services for handling textual documents including their index structures have been implemented.

In the third part of the thesis, we benchmark the implemented system against traditional database systems (Oracle and Sybase). In this benchmark, we investigate the behaviour of the system in an environment with up to 25 concurrent users, which are concurrently inserting new documents as well as searching for existing ones. The results show that the implemented system based on TPM/ONT provides efficient online support for textual data including the online indexing of this data. Finally, we present the ability of TPM/ONT to execute higher level operations in parallel and how the transaction-oriented processing of textual documents can benefit from such a parallelisation.

1 Einleitung

Motivation und Problemstellung

Der Anteil elektronisch verfügbarer Dokumente hat in den letzten Jahren stetig zugenommen und liegt heute geschätzten 80 Prozent. Mit der Verbreitung der Datenautobahn ist nicht nur mit einem weiteren prozentuellen Anstieg der elektronisch verfügbaren Information, sondern auch mit einer drastischen Steigerung elektronischer Information überhaupt zu rechnen.

Daten werden künftig nicht mehr als Hardcopy, sondern elektronisch abgelegt werden. Sowohl an die heutigen, wie an die zukünftigen Ablagesysteme werden dabei zwei Grundforderungen gestellt: Zum einen müssen jederzeit Dokumente abgelegt und entfernt werden können, zum anderen muss eine sowohl schnelle, als auch fehlertolerante Suche nach archivierten Dokumenten möglich sein.

Datenbanksysteme, wie beispielsweise Oracle, Sybase oder Ingres, erlauben es, Datensätze im laufenden Betrieb zu bearbeiten und gleichzeitig mehrere Benutzer mit Information zu bedienen. Das Abfragen von Daten in diesen Systemen erfolgt mit einer Anfragesprache, welche die interne Struktur der verwalteten Dokumente in die Anfrage miteinbezieht. Die Verwaltung von Volltextdokumenten werden jedoch von diesen Systemen nicht oder nur mangelhaft unterstützt.

Bei Information Retrieval Systemen steht die effiziente Beantwortung von Benutzeranfragen im Vordergrund. Änderungen an der Datenbasis, also das Hinzufügen neuer oder das Entfernen alter Dokumente, erfolgen in vielen dieser Systeme immer noch offline über Nacht. Änderungen im laufenden Betrieb werden in der Regel nicht durchgeführt, da das Einfügen und Löschen von Dokumenten umfangreiche Änderungen an den Textindexstrukturen mit sich zieht und deshalb mit wesentlichen Behinderungen parallel ablaufender Suchoperationen gerechnet werden muss.

Forschungsgruppen im Bereich Information Retrieval haben in der Vergangenheit die Problematik, Dokumente in laufenden Betrieb einzufügen und zu löschen, wiederholt aufgegriffen. Die als Lösung für dieses Problem vorgestellten Methoden zeichnen sich jedoch dadurch aus, dass sie Änderungen am Datenbestand nur mit Einschränkungen bezüglich der Transaktionsverwaltung erlauben. Ein typisches Beispiel hierfür ist die Beschränkung, dass jede Benutzeroperation (Suchen, Einfügen, Löschen) jeweils im Rahmen einer eigenständigen Transaktion ausgeführt wird. Dieses Vorgehen mag in vielen Anwendungen des Information Retrieval akzeptabel sein. In typischen Datenbankanwendungen hingegen ist eine solche Einschränkung in der Regel zu restriktiv: Typischerweise umfasst eine Transaktion mehrere Operationen, wobei die einzelnen Operationen oftmals voneinander abhängig sind und daher nur gesamthaft oder gar nicht ausgeführt werden dürfen. Diese Transaktionsorientierung ist mit heutigen Information Retrieval Systemen nicht möglich.

Zusammenfassend ist festzustellen, dass sich heutige Datenbanksysteme für die transaktionsorientierte Bearbeitung grosser Mengen verschiedener Datensätze, nicht aber von Fliesstextdokumenten eignen. Information Retrieval Systeme hingegen bieten sich sich für die Suche von und in Volltextdokumenten an, ermögliche jedoch keine effizienten und vor allem keine transaktionsorientierten Änderungen ihres Datenbestands im laufenden Betrieb.

Da die beiden Systeme bisher in verschiedenen Anwendungsbereichen zum Einsatz kamen, wurden ihre Unzulänglichkeiten in der Vergangenheit in Kauf genommen. In vielen Anwendungsbereichen werden heute jedoch Informationssysteme benötigt, deren Anforderungen durch den Einsatz eines Datenbanksystems oder eines Information Retrieval Systems allein nicht abgedeckt werden können. Dies ist in den Gegebenheiten der heutigen Datenbestände, wie Dynamik, Umfang und Strukturierung, und in den Erwartungen bezüglich Verfügbarkeit und Sicherheit begründet:

- *Die Datenbestände sind dynamisch:* Es fallen kontinuierlich neue Daten an, in global tätigen Unternehmen während 24 Stunden am Tag.

- *Die Datenbestände sind umfangreich:* Immer mehr Daten werden elektronisch erzeugt und übermittelt. Die Anzahl und Grösse elektronisch verfügbarer Daten nimmt stetig zu. Eine elektronische Archivierung ist unumgänglich.

- *Die Datenbestände sind semistrukturiert:* Die Datensätze bestehen aus einer Kombination von strukturierten Anteilen (Standarddatentypen wie Zahlen, Datum, Zeichenketten...) und unstrukturierten Anteilen (Fliesstext).

- *Die Datenbestände müssen sofort verfügbar sein:* Viele Daten, beispielsweise die Meldungen einer Presseagentur, sind den Benutzern unmittelbar nach deren Eintreffen zur Verfügung zu stellen. Ein Nachfahren der Datenbasis im Batch-Modus — wie in zahlreichen Anwendungsgebieten des Information Retrieval — ist nicht ausreichend.

  In anderen Informationssystemen ist das Nachführen im Batch-Modus in der Nacht oder am Wochenende schon aufgrund des Volumens neuer Daten nicht mehr möglich. Solche Systeme werden deshalb heute bereits in zwei Instanzen betrieben. Zu bestimmten Zeitpunkten wird jeweils von der Instanz zum Anfragen auf die Instanz zum Einspielen von Daten umgeschaltet, die neuen Daten sind unter Umständen nur mit deutlicher Verzögerung verfügbar.

- *Die Datenbestände müssen gesichert sein:* Das zur Verwaltung eingesetzte System hat zu garantieren, dass die Information einschliesslich aller Änderungen auch im Fall eines Systemversagens wiederherstellbar ist. Information ist eine kostbare und kostspielige Ressource. Für viele Unternehmen ist sie daher überlebenswichtig.

Um den gestiegenen Bedürfnissen und Anforderungen der Benutzer gerecht zu werden, bedarf es in Zukunft eines Systems, das sowohl ein effizientes Bearbeiten von Daten im laufenden Betrieb, als auch ein schnelles Suchen nach Daten jeglicher Strukturierung erlaubt. Ideal wäre somit ein System, das die Vorteile von Datenbank- und Information Retrieval Systemen in sich vereinigt.

Ziel der Arbeit

Mitte der siebziger und zu Beginn der achtziger Jahre haben sich Forschungsgruppen mit einer Synthese von Datenbanksystemen und Information Retrieval Systemen befasst und unter anderem die Integration textueller Indexstrukturen in bestehende, relationale Datenbanksysteme untersucht. Einige der realisierten Prototypen entstanden dabei unter der Prämisse, dass ein existierendes Datenbanksystem eingesetzt werden sollte, dessen Sourcecode nicht zur Verfügung steht und somit die textuellen Indexstrukturen nicht in den Kern des Datenbanksystems integriert werden können. Die Textindexstrukturen wurden daher "on top" des Datenbanksystems in Form invertierter Listen, welche in Benutzerrelationen der Form InvList(Wort, Dokumenten-ID) gespeichert wurden, realisiert. Die Verarbeitung dieser Textindexstruktur erfolgte mit Hilfe eines Präprozessors. Diese Prototypen zeigten Schwächen in drei Bereichen:

1. *Suboptimale Hardware:* Die damals zur Verfügung stehende Hardware war technologisch gesehen nicht für die Verarbeitung von Textindexstrukturen geeignet, da invertierte Listen viel Speicherplatz auf den Sekundärspeichermedien benötigen. Zur Beantwortung einer Anfrage müssen diese vom Sekundär- in den Primärspeicher übertragen werden, was bei einer damaligen mittleren Zugriffszeit von über 50ms, sowie niedrigen Datentransferraten sehr zeitintensiv war. Zudem standen nur Einprozessormaschinen zur Verfügung, deren Leistungsgrenze schnell erreicht war.

2. *Inadäquate Mehrbenutzerverwaltung:* Kommerzielle Datenbanksysteme setzten und setzen typischerweise ein Zwei–Phasen–Sperrprotokoll (2PL) zur Mehrbenutzerkontrolle ein. Werden die Textindexstrukturen als invertierte Listen in Benutzerrelationen der Form InvList(Wort, Dokumenten-ID) gespeichert, kommt es unweigerlich zu sperrbedingten Behinderungen konkurrierender Operationen, wie anhand eines Beispiels deutlich wird: Ein Benutzer A fügt ein Dokument X mit einem Textattribut *Datenbanken und Informationssysteme* ein, während gleichzeitig ein Benutzer B nach Dokumenten sucht, welche die Wörter *Datenbanken* und *Retrieval* enthalten. Beim Einfügen von und bei der Suche nach Dokumenten wird jeweils auf den Textindex InvList(Wort, Dokumenten-ID) zugegriffen. Es ist unschwer zu erkennen, dass die Einfüge- und die Suchoperation nicht gleichzeitig ausgeführt werden können, da die beiden Operationen ein gemeinsames Tupel *(Datenbanken, X)* in InvList aufweisen, auf das Benutzer A schreibend und Benutzer B lesend zugreifen will. Ein solcher gleichzeitiger Zugriff ist daher bei Einsatz einer Transaktionsverwaltung mit 2PL nicht möglich.

   Folge dieses konkurrierenden Zugriffs ist die Verzögerung einer der beiden Transaktionen bis zum Abschluss der anderen. Hat das einzufügende Dokument beispielsweise 100 zu indexierende Wörter und beträgt die mittlere Plattenzugriffszeit 50ms, werden für die Indexierung rund 5 Sekunden benötigt. Dies ist unbefriedigend, wenn man bedenkt, dass diese Behinderung im vorliegenden Beispiel aus Sicht des Benutzers B unnotwendig ist: Das Dokument, das von Benutzer A eingefügt wird, qualifiziert sich nicht auf die Anfrage von Benutzer B, da es den Suchbegriff *Retrieval* nicht enthält. Aus Sicht der Einfügeoperation ist eine parallel ablaufende Suchoperation semantisch gesehen ebenfalls belanglos, da eine Suchoperation in keinem Fall Einfluss auf die Einfügeoperation hat.

3. *Mangelnde Funktionalität:* Eine Präprozessorlösung ist aus Sicht des Benutzers aus drei Gründen unbefriedigend: Erstens wird die Wartung der textuellen Indexstrukturen nicht

durch das Datenbanksystem unterstützt. Beim Einfügen, Ändern und Löschen von Dokumenten muss diese Wartung explizit durch Aufruf des Präprozessors durch den Benutzer selbst vorgenommen werden. Zweitens ist dem Datenbanksystem die Existenz der Textindexstrukturen nicht bekannt. Diese können daher vom System auch nicht für eine Anfrageoptimierung verwendet werden. Soll ein Textindex für die Beantwortung einer Anfrage eingesetzt werden, muss dies explizit durch Miteinbezug des Textindexes in die Abfrage durch den Benutzer erfolgen. Drittens stellt das Datenbanksystem keine Operationen auf Texte zur Verfügung. So existiert beispielsweise kein relationaler Operator, mit dem nach allen Dokumenten gesucht werden kann, welche bestimmte Wörter enthalten.

Seit der Einführung der ersten relationalen Datenbanksysteme vor zwanzig Jahren kam es sowohl auf dem Hardware-, als auch auf dem Kommunikationssektor zu bedeutenden Entwicklungen, an deren Ende heute leistungsfähige Client–Server-Systeme stehen, die auf Mehrprozessorrechnern betrieben werden. Daten werden in heutigen Informationssystemen typischerweise in grossen Disk-Array mit Zugriffszeiten im Bereich weniger Millisekunden und Übertragungsgeschwindigkeiten von einigen Hundert Megabyte pro Sekunde gespeichert. Dank dieser Entwicklung — deren Ende noch nicht abzusehen ist — ist der erste oben genannte Nachteil damaliger Systeme daher heute kein Thema mehr.

Aufgrund dieser Leistungsfähigkeit wurde und wird vielfach der Wunsch geäussert, Datenbanksysteme nicht nur für klassische Anwendungen, beispielsweise im Bankenbereich, sondern auch in anderen Anwendungsgebieten einzusetzen. Bei den durchgeführten Umsetzungsversuchen wurde jedoch sehr schnell deutlich, dass Datenbanksysteme für Nichtstandardanwendungen aufgrund der verwendeten 2PL–Transaktionsverwaltung nur beschränkt einsetzbar sind. Im Rahmen von Forschungsprojekten wurden daher neue Transaktionsmodelle entwickelt, welche den Bedüfnissen von Nichtstandardanwendungen Rechnung tragen. Das für die vorliegende Arbeit relevante "neue" Transaktionsmodell ist die Transaktionsverwaltung mittels offen geschachtelter Mehrschichtentransaktionen [Wei86, WS92], konkret Zweischichtentransaktionen. Dokumentenverwaltung ist in einem System mit zweischichtiger Transaktionsverwaltung beispielsweise mittels folgender Operationen auf den beiden Schichten möglich: Auf der unteren Schicht finden sich Operationen zum Schreiben eines einzelnen Worts eines Dokuments mit einer bestimmten ID in invertierte Listen (w^0(Wort, ID)), sowie zum Suchen der Dokumenten-IDs aller Dokumente, die ein bestimmtes Wort enthalten (r^0(Wort) $\rightarrow$ {ID}). Auf der oberen Ebene findet man eine Operation zum Einfügen von Dokumenten ($w^1(w_1, w_2, \ldots, w_n)$) (hier charakterisiert durch die Wörter, aus denen sie bestehen) sowie eine Operation zum Suchen aller Dokumente, die bestimmte Wörter enthalten ($r^1(w_1, w_2, \ldots, w_m)$). Die Einfügeoperation ($w^1(w_1, w_2, \ldots, w_n)$) auf der oberen Schicht setzt sich aus dem wiederholten Aufruf der Einfügeoperation w^0(Wort, ID) auf der unteren Schicht zusammen, wobei jede Operation auf der unteren Schicht im Rahmen einer eigenständigen Transaktion durchgeführt wird. Analog hierzu besteht die Suchoperation $r^1(w_1, w_2, \ldots, w_m)$ aus dem wiederholten Aufruf der Suchoperation $r^0(w_i) \rightarrow$ {ID} $(1 \leq i \leq m)$, welche jeweils wiederum in eigenständigen Transaktionen auf der unteren Ebene ausgeführt werden. Im Anschluss an dieses Lesen werden die erhaltenen Mengen von Dokumenten-IDs geschnitten und somit alle Dokumente ermittelt, welche alle Wörter $w_1, \ldots, w_m$ enthalten.

Für die parallele Ausführung von Such- und Leseoperationen auf der oberen Ebene gilt, dass eine parallele Ausführung möglich ist, falls sich das einzufügende Dokument nicht auf die Suche qualifiziert. Im Rahmen einer Suchoperation parallel zu einer Leseoperation ist es aufgrund

der frühzeitigen Sperrfreigabe auf der unteren Ebene durchaus möglich, dass eine Operation $r^0(\text{Wort}) \rightarrow \{\text{ID}\}$ eine Dokumenten-ID eines Dokuments ermittelt, das gerade eingefügt wird und daher für die Suchoperation noch nicht sichtbar sein sollte, da die einfügende Transaktion noch nicht abgeschlossen ist. Dieses Sichtbarwerden ist jedoch unkritisch: Da sichergestellt ist, dass sich das einzufügende Dokument nicht auf die Suche qualifiziert, existiert mindestens ein Suchbegriff, der nicht im einzufügenden Dokument enthalten ist. Beim Lesen der invertierten Liste zu diesem Suchbegriff ist daher die ID des einzufügenden Dokuments nicht enthalten. Beim anschliessenden Schneiden der Dokumenten-IDs wird diese Dokumenten-ID eliminiert. Die Suchoperation auf der oberen Ebene ist somit korrekt.

Im Fall der transaktionsorientierten Verarbeitung von Dokumenten unter Einsatz einer mehrschichtigen Transaktionsverwaltung ist dank frühzeitiger Freigabe der Sperren auf den Textindexstrukturen (jeweils am Ende einer r^0- bzw. w^0-Operation) mit einer deutlichen Steigerung des Transaktionsdurchsatzes, sowie einer Verkürzung der Antwortzeiten zu rechnen. Sofern es gelingt, die Mehrschichtentransaktionsverwaltung in das Datenbanksystem zu integrieren, kann der zu Beginn erwähnte transaktionsbedingte Leistungsengpass somit behoben und ein relationales Datenbanksystem zur Verwaltung von Textdokumenten eingesetzt werden

Als letzte der vorgängig aufgezählten Unzulänglichkeiten relationaler Systeme bezüglich der Verwaltung von Dokumenten (suboptimale Hardware, inadäquate Mehrbenutzerverwaltung und mangelnde Funktionalität) verbleibt somit nur die mangelnde Funktionalität: Die Wartung von Indexstrukturen sowie der Einsatz dieser in Anfragen sollte nicht durch den Benutzer via eines Präprozessors vorgenommen werden. Vielmehr sollten ihm semantisch reiche Operationen zum Einfügen, Suchen, Ändern und Löschen von Dokumenten zur Verfügung gestellt werden.

Die Notwendigkeit, Benutzer nicht direkt mit einer Datenbank interagieren zu lassen, sondern ihnen den Zugriff auf die Daten nur über semantisch reiche Operationen zu erlauben, besteht in einer Reihe von Anwendungen. Ein typisches Beispiel hierfür sind Bankapplikationen. Bei solchen muss ein Applikationsprogrammierer beispielsweise den Geldtransfer von einem Konto auf das andere nicht mittels Embedded–SQL programmieren, sondern er verwendet einen sogennanten *Service* (=Bibliotheksfunktion), welcher die Überweisung vornimmt. Im Regelfall wird ein solcher Service nicht auf eine einzige Applikation zugeschnitten, sondern wird generisch aufgebaut, sodass er von einer Reihe von Applikationen wiederverwendet werden kann. Vorteil einer solchen Realisierung ist, dass dem Benutzer seinerseits die Komplexität einer Operation verborgen bleibt und andererseits im Fall sich ändernder Bedingungen nur der Code des Services angepasst werden muss, nicht aber die Applikationsprogramme, welche ihn verwenden.

Heute ist eine Reihe sogenannter *Middlewareprodukte* auf dem Markt verfügbar, welche die Kapselung von Datenbankfunktionalität in Form von Services unterstützen. Sie werden als "Middlewareprodukte" bezeichnet, weil sie in der Mitte zwischen Benutzer und Datenbank plaziert werden. Ein Beispiel für solche Middleware-Produkte sind *Transaktionsmonitore*. Hauptkennzeichen eines solchen Monitors sind erstens die obengenannten Services, welche die Interaktion mit der Datenbank übernehmen. Die Services werden in der Sprache des unterliegenden Datenbanksystems, beispielsweise Embedded–SQL im Fall relationaler Systeme, realisiert. Diese Services werden im Rahmen von Applikationsprogrammen aufgerufen. Das zweite Merkmal von Transaktionsmonitoren ist, dass Applikationsprogramme Services transaktionsorientiert aufrufen können. Das heisst, es ist möglich mehrere Services im Rahmen einer einzigen Transaktion aufzurufen.

Mit Hilfe eines Transaktionsmonitors wäre es somit möglich, dem Benutzer ein transaktionsorientiertes System zur Verwaltung von Dokumenten zur Verfügung zu stellen, das der eingangs aufgestellten Forderung nach adäquater Funktionalität, d.h. semantisch reichen Operationen zum Einfügen, Suchen, Ändern und Löschen von Dokumenten, gerecht wird.

Stellt man einen Transaktionsmonitor und ein zweischichtiges Datenbanksystem einander gegenüber, sind eine Reihe von Parallelen in diesen beiden Systemen zu erkennen: Die Services des Transaktionsmonitors zeigen Ähnlichkeit mit den Operationen der oberen Schicht des zweischichtigen Datenbanksystems: Bei beiden handelt es sich um semantisch reiche Operationen. Services werden in der Sprache des unterliegenden Datenbanksystems implementiert. Dies korrespondiert in einem zweischichtigen Datenbanksystem mit der Realisierung der semantisch reichen Operationen auf der oberen Schicht mittels primitiver Operationen auf der unteren. Was ein zweischichtiges Datenbanksystem von einem Transaktionsmonitor unterscheidet, ist die Transaktionsverwaltung: Ein mehrschichtiges Datenbanksystem besitzt auf jeder Schicht einen eigenständigen Transaktionsmanager. In einem System mit Transaktionsmonitor ist nur auf der unteren Ebene ein Transaktionsmanager vorhanden, nämlich der im verwendeten Datenbanksystem eingebaute. Auf der oberen Schicht ist jedoch kein Transaktionsmanager vorhanden.

Transaktionsmonitore sind offene Systeme, d.h. sie können einfach um neue Services erweitert werden. Im Rahmen dieser Arbeit wird gezeigt, dass es möglich ist, diese Offenheit für die Realisierung eines zweischichtigen Datenbanksystems zu nutzen. Konkret wird die Implementierung eines vollständigen, zweiten Transaktionsmanagers für die obere Schicht eines Zweischichtensystems "on top" des kommerziell verfügbaren Transaktionsmanagers *Tuxedo* vorgestellt. Die Integration dieses zweiten Transaktionsmanagers in den Transaktionsmonitor erfolgt dabei in Form gewöhnlicher Services. Dies hat den Vorteil, dass existierende Technologie — Transaktionsmonitorund Datenbanksystem — nicht verändert werden muss und somit zukünftige, leistungsfähigere Versionen dieser Produkte ohne Änderungen am Sourcecode übernommen werden können.

Auf Basis dieses TPM/ONT genannten erweiterten Transaktionsmonitors ist es möglich, ein zweischichtiges Datenbanksystem zur Dokumentenverwaltung, *TPM/ONT-Text* genannt, zu realisieren, wobei als unterliegendes Datenbanksystem abwechslungsweise Oracle oder Sybase zum Einsatz kommen können. Dank Einsatz einer mehrschichtigen Transaktionsverwaltung ist es mit TPM/ONT-Text möglich, Dokumente transaktionsorientiert zu verwalten. Insbesondere ist es mit diesem System möglich, Dokumente im laufenden Betrieb einzufügen und zu löschen, ohne hierdurch parallel laufende Suchtransaktionen wesentlich zu behindern.

In dieser Arbeit wird somit gezeigt, wie existierende Datenbanktechnologie (Relationale Datenbanksysteme und Transaktionsmonitore) und Ergebnisse aus der Forschung (Mehrschichtentransaktionen) gewinnbringend zu einem lauffähigen mehrschichtigen Datenbanksystem kombiniert werden können, das für die transaktionsorientierte Verwaltung von Dokumenten effizient eingesetzt werden kann.

Die Hauptbeiträge dieser Arbeit sind zusammengefasst folgende:

- Es wird das Konzept zweischichtiger Transaktionen bezüglich der Transformation von Intra-Transaktionsparallelität auf der Benutzerebene in Inter-Transaktionsparallelität auf der Datenbankebene untersucht.

- Es wird gezeigt, wie ein existierendes relationales Datenbanksystem durch die Erweiterung eines Transaktionsmonitors um Funktionalität aus Forschungsergebnissen zu einem zweischichtigen Datenbanksystem ausgebaut werden kann.

  Der realisierte Prototyp TPM/ONT, eine Erweiterung des Transaktionsmonitors Tuxedo mit den Datenbanksystemen Oracle und Sybase als Speichersysteme auf der unteren Ebene, wird im Detail vorgestellt. Dieser Prototyp ist aufgrund seines Aufbaus nicht auf die Verwaltung von Dokumenten beschränkt, sondern kann auch in anderen Anwendungen eingesetzt werden.

- Anhand von Messungen wird nachgewiesen, dass die Kosten für die Ausführung zweischichtiger Transaktionen, insbesondere die in der Theorie immer befürchteten hohen Commit-Kosten der Transaktionen auf der unteren Schicht, gering sind.

- Die Möglichkeit der Verwaltung textueller Indexstrukturen durch relationale Datenbanksysteme wird untersucht. Insbesondere werden mögliche logische und physische Datenstrukturen unter dem Gesichtspunkt ihrer Änderbarkeit im laufenden Betrieb analysiert.

- Es wird nachgewiesen, dass TPM/ONT erfolgreich zur transaktionsorientierten Verarbeitung von Textobjekten eingesetzt werden kann: Die durchgeführten Messungen, in welchen TPM/ONT mit einschichtigen Datenbanksystemen verglichen wird, zeigen zum einen, dass es bei Einsatz von TPM/ONT dank der frühzeitigen Freigabe von Sperren auf den Textindexstrukturen möglich ist, Dokumente im laufenden Betrieb einzufügen, ohne parallel ablaufende Suchtransaktionen zu behindern. Zum anderen weisen die Messungen nach, dass aufgrund der Fähigkeit, gewisse Teile der Mehrschichtentransaktionen parallel auszuführen — so können beispielsweise alle Textattribute eines Dokuments gleichzeitig indexiert werden — sowohl der Transaktionsdurchsatz gesteigert, als auch gleichzeitig die durchschnittlichen Antwortzeiten reduziert werden können.

  Abschliessend bleibt anzumerken, dass der Schwerpunkt der Arbeit auf der *transaktionsorientierten* Dokumentenverwaltung und nicht auf der Information Retrieval Funktionalität liegt. Die vorliegende Arbeit beschränkt sich daher auf boolesches Retrieval. Die hier gewonnenen Ergebnisse lassen sich in grossen Teilen jedoch auch auf ein System mit nicht-booleschem Retrieval übertragen. Eine solche Übertragung hätte den Rahmen dieser Arbeit insbesondere bezüglich Messungen gesprengt, und muss daher Thema weiterer Arbeiten bleiben.

Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt aufgebaut: In Kapitel 2 wird ein Überblick über die Verwaltung und die Suche von Dokumenten in elektronischen Informationssystemen gegeben. Hierzu wird zuerst der Begriff des Information Retrieval Modells zur Charakterisierung von Information und Anfragen erläutert und anschliessend auf die Approximation von Volltexten durch Deskriptoren eingegangen. Sodann werden Realisierungsvarianten zur Speicherung dieser Deskriptoren in relationalen Datenbanksystemen vorgestellt. Am Ende des Kapitels wird ein Überblick über verwandte Arbeiten gegeben und die vorliegende abgegrenzt.

Kapitel 3 behandelt die Transaktionsverwaltung, welche eine zentrale Rolle in einem Datenbanksystem spielt. Anhand von Beispielen wird gezeigt, welche Probleme durch den Einsatz

einer herkömmlichen Transaktionsverwaltung bei der Integration textueller Daten auftreten und wie die hieraus resultierenden Verarbeitungsengpässe mit Hilfe mehrschichtiger Transaktionen behoben werden können.

In Kapitel 4 wird ein Datenbanksystem zur Ausführung zweischichtiger Transaktionen vorgestellt. Dieses System baut im Gegensatz zu anderen Ansätzen auf bestehender Datenbanktechnologie auf. Konkret wird für die Implementierung der Transaktionsmonitor Tuxedo, der um einen vollständigen, zweiten Transaktionsmanager erweitert wurde, sowie wahlweise eines der relationalen Datenbanksysteme Oracle oder Sybase eingesetzt.

Kapitel 5 schliesslich zeigt, wie mit Hilfe des im vorhergehenden Kapitel vorgestellten Prototypsystems transaktionsorientierte Operationen zur Verwaltung von und Suche in Textdokumenten realisiert werden können.

Die Effizienz des in Kapitel 4 realisierten und in Kapitel 5 um Operationen zur Verwaltung von und Suche in grossen Dokumentenkollektionen erweiterten Systems wird in Kapitel 6 anhand detaillierter Leistungsuntersuchungen vorgestellt.

In Kapitel 7 werden die Ergebnisse der vorliegenden Arbeit zusammengefasst und Minimalanforderungen für die Realisierung künftiger transaktionsorientierter Systeme zur Dokumentenverwaltung aufgestellt. Abschliessend wird aufgezeigt, in welchen anderen Anwendungsgebieten das realisierte zweischichtige Datenbanksystem ebenfalls eingesetzt werden kann.

2 Dokumentenverwaltung und -suche in elektronischen Informationssystemen

Datenbanksysteme stellen ihren Benutzern heute meist nur eine leistungsmässig unzureichende Funktionalität in Bezug auf die Verwaltung von Textdaten zur Verfügung. Wenn in Zukunft für die Verwaltung von Textdokumenten, welche meist durch stark strukturierte Information ergänzt sind, Datenbanksysteme eingesetzt werden, ist eine Erweiterung bestehender Datenbanksysteme um Zugriffsstrukturen für Volltextdaten unumgänglich. In diesem Kapitel wird eine solche Erweiterung vorgestellt. Hierzu wird zunächst der Begriff *Information Retrieval Modell* besprochen, welcher der Charakterisierung von Information Retrieval Problemen dient. Anschliessend werden *Deskriptoren* zur Approximation von Fliesstextdokumenten vorgestellt und logische sowie physische Speicherungsmöglichkeiten von Deskriptoren in einem relationalen Datenbanksystem besprochen.

Im letzten Teil dieses Kapitels wird ein Überblick über Forschungsarbeiten im Bereich der Integration von Information Retrieval– und Datenbanksystemen sowie deren kommerzielle Umsetzung gegeben und die vorliegende Arbeit von diesen abgegrenzt.

2.1 Information Retrieval Modelle

Der Begriff des Information Retrieval hat sich seit seiner Einführung in den fünfziger Jahren stark gewandelt. Während man früher darunter die Suche in und nach Volltextdokumenten verstand, wird der Begriff heute immer mehr im Zusammenhang mit der Suche nach Information in irgendeinem Medium — beispielsweise in Bild- und Tondokumenten [Wec95], aber auch in Grafikdokumenten [LM95] — verwendet. *Das* Information Retrieval an sich existiert nicht. Um ein Problem mit Hilfe von Information Retrieval zu lösen, ist es notwendig, ein sogenanntes Information Retrieval Modell (IRM) aufzustellen, welches das Problem mittels dreier Komponenten charakterisiert: *Dokumentenrepräsentation*, *Anforderungsprofil* des Benutzers, problemadäquate *Suchfunktion*.

Obwohl — wie bereits erwähnt — der Begriff des Information Retrieval für die Suche nach Information in verschiedenen Medien (Bild, Ton, Text, ...) verwendet wird, beschränkt sich diese Arbeit auf Textdokumente.

Dokumentenrepräsentation

Welches Informationssystem für die Verwaltung von Dokumenten eingesetzt werden soll, hängt überwiegend vom Aufbau der zu speichernden Information ab. Dies bedeutet, dass vorab ge-

klärt werden muss, ob es sich um *unstrukturierte, strukturierte* oder *semistrukturierte* Dokumente handelt.

Dokumente werden als *unstrukturiert* bezeichnet, wenn sie ausschliesslich aus Fliesstext bestehen, d.h. abgesehen von der Formatierung (Sätze, Absätze, Kapitel, ...) keine innere Struktur aufweisen (siehe Abbildung 2.1). Unstrukturierte Dokumente eignen sich besonders zur Spei-

```
(c't) Die Kette von Betrugsmanövern rund um den PC reisst nicht ab. Nach
wertlosen Cache-Dummies gelangten jetzt DX4-Prozessoren mit gefälschter
Beschriftung auf den Markt. Sie werden mit sattem Profit als 100-MHz-CPUs
verkauft, obwohl sie nur mit 75 MHz stabil laufen.
Die Hintermänner dieser neuerlichen Betrugsaffäre machen sich die
Tatsache zunutze, dass die meisten Digitalchips unter alltäglichen
Betriebsbedingungen eine leicht überhöhte Taktfrequenz verkraften. Bei
der Selektion werden immer Reserven vorgesehen, zum Beispiel für heisse
Sommertage. Auch treten manche Fehlfunktionen nicht offensichtlich in
Erscheinung, sondern lassen sich nur mit den komplexen Testverfahren der
Hersteller als solche identifizieren.
```

Abbildung 2.1: Unstrukturiertes Dokument

cherung in einem Dateisystem oder in einem Information Retrieval System.

Im Gegensatz hierzu werden unter dem Begriff *strukturierte* Dokumente jene Informationseinheiten verstanden, die durch Aggregation einer Menge von Attributen aus Standarddatentypen[1] gebildet werden. Oftmals werden diese Dokumente als eine Kollektion von *(Name, Wert)*–Paaren dargestellt, wie sie in Abbildung 2.2 zu sehen ist. Strukturierte Dokumente wer-

Personalnummer	:	534.67.372.153FL
Name	:	Kaufmann
Vorname	:	Helmut
Geburtsdatum	:	10. September 1967
Strasse	:	Unterfeld 22
Wohnort	:	Triesen
Staat	:	Fürstentum Liechtenstein
Hobbies	:	Lesen, Squash

Abbildung 2.2: Strukturiertes Dokument

den in der Regel in Datenbanksystemen verwaltet, wobei Dokumente gleicher Gattung (z.B. Personenbeschreibungen) zu Relationen (im Fall eines relationalen Systems) oder zu Klassen (im Fall eines objektorientierten Systems) zusammengefasst werden.

Betrachtet man die verschiedenen Dokumente in einem Unternehmen, ist festzustellen, dass fast alle Arten von Information — so auch der in Abbildung 2.3 auf Seite 21 gezeigte Bericht — sowohl strukturierte (Titel, Autoren, ...) als auch unstrukturierte Anteile (Fliesstext) enthalten. Solche Dokumente werden als *semistrukturiert* bezeichnet. Eine Verwaltung semistrukturierter Dokumente ist sowohl mit Information Retrieval Systemen als auch mit Datenbanksystemen möglich: Kommerzielle IRS, wie beispielsweise BASISPlus, erlauben nicht nur die Speicherung

[1] *Standarddatentypen* sind Zahlen, Datum, Währung, kurze Zeichenketten etc.

reiner Fliesstextdokumente, sondern ermöglichen auch eine Modellierung von Dokumenten in Form von Relationen mit mengenwertigen Attributen. Kommerzielle relationale Datenbanksysteme wie Oracle, Sybase oder Ingres erlauben, Fliesstexte in Attributen des Typs BLOB[2] zu speichern. Für welchen Systemtyp man sich entscheidet, hängt von den Benutzeranforderungen und der benötigten Suchfunktion ab.

Benutzeranforderungen

Benutzer eines Informationssystems stellen an dieses je nach Bedarf und in Abhängigkeit der gespeicherten Daten verschiedene Anforderungen. Eine einfache Klassifizierung wie im Fall der Dokumentenrepräsentation ist daher nicht möglich. Eine beispielsweise typische Anforderung ist der Wunsch, Dokumente zu finden, die bestimmte Wörter oder Phrasen enthalten oder die einen ähnlichen Informationsgehalt aufweisen wie ein vorgegebenes Musterdokument.

Ferner werden weitere Anforderungen gestellt, die zwar nicht Teil des Information Retrieval Modells sind, die Wahl des Informationssystems jedoch nachhaltig beeinflussen:

- *Verfügbarkeit:* Ein Informationssystem muss rund um die Uhr verfügbar sein. Dies ist besonders in Unternehmen mit weltweiten Niederlassungen notwendig.
- *Aktualität:* Neue Informationen müssen unmittelbar verfügbar sein, daher sind Batch-Updates über Nacht in vielen Fällen nicht ausreichend.
- *Datensicherheit:* Es muss sichergestellt werden, dass die Integrität des Datenbestands auch im Fall eines Systemversagens gewährleistet ist.
- *Datenschutz:* Der Zugriff auf Daten muss kontrolliert ablaufen, d.h. Daten sollen nicht allgemein zugänglich sein.

Diesen Anforderungen kann heute mittels Einsatz von Datenbanksystemen Rechnung getragen werden.

Suchfunktionen

Die im Bereich des Information Retrieval verwendeten Suchmodelle lassen sich in *boolesche* Suchmodelle einerseits und *nichtboolesche* andererseits unterteilen.

Bei einem booleschen Suchmodell wird eine Anfrage in Form eines prädikatenlogischen Ausdrucks formuliert. Die Antwortmenge wird durch Prüfung dieses Ausdrucks gegen die einzelnen in der Datenbank gespeicherten Dokumente ermittelt. Dieses Suchmodell wird vorwiegend im Datenbankbereich eingesetzt.

Das Hauptproblem des booleschen Suchmodells liegt darin, dass sich nur jene Dokumente auf eine Anfrage qualifizieren, die dem prädikatenlogischen Ausdruck exakt entsprechen. Werden beispielsweise alle Dokumente gesucht, welche die Wörter *boolesche, nichtboolesche, Suchmodelle, Information* und *Retrieval* enthalten, müssen alle fünf Wörter im Dokument enthalten

[2]Binary Large OBject

sein, damit sich das Dokument qualifiziert. Oft ist der Benutzer jedoch auch an Dokumenten interessiert, die nur einzelne der Schlüsselwörter beinhalten. Selbstverständlich können diese Dokumente auch mit Hilfe des booleschen Suchmodells gefunden werden [Coo88]. Die dazu erforderlichen Suchprädikate werden jedoch sehr lang und deren Formulierung unpraktisch und fehleranfällig. Im Bereich des Information Retrieval wird daher immer mehr zu nichtbooleschen Suchmodellen übergegangen, die einer Suche nach Dokumenten mittels vager Anfragen Rechnung tragen. Das bekannteste nichtboolesche Suchmodell dürfte heute das Vektorraummodell [WZW85] sein, auf das hier stellvertretend für alle nichtbooleschen Modelle eingegangen wird.

Beim Vektorraummodell wird jedem Dokument d einer Kollektion sowie jeder Anfrage q ein n-dimensionaler Vektor (Dokumentenvektor und Anfragevektor) zugeordnet. Der Vektor ist dabei eine Approximation eines Dokuments oder einer Anfrage. Die Ähnlichkeit zwischen der Anfrage und einem Dokument kann durch den Vergleich des Anfragevektors mit dem Dokumentenvektor, beispielsweise durch Berechnung des Winkels zwischen den Vektoren, ermittelt werden. Als Ergebnis erhält man für jedes Dokumentenvektor/Anfragevektor–Paar eine Zahl s (in der Regel gilt: $0 \leq s \leq 1$), welche die Ähnlichkeit zwischen dem Dokument und der Anfrage beziffert, wobei ein Wert von $s = 1$ die Gleichheit von Anfrage und Dokument bedeutet. Anschliessend werden alle Dokumente, deren Übereinstimmung s über einem gewissen Schwellwert liegt, so geordnet, dass das Dokument mit maximalem s am Beginn und jenes mit minimalem s am Ende zu liegen kommt. Diese so geordnete Antwortkollektion wird dann dem Benutzer als Resultat auf seine Anfrage präsentiert.

Ein nichtboolesches Suchmodell hat gegenüber einem booleschen den Vorteil, dass man auf eine Anfrage nicht nur jene Dokumente erhält, die exakt der Anfrage entsprechen, sondern auch jene, welche der Anfrage nur ähnlich sind, wobei das System sicherstellt, dass die ähnlichsten Dokumente am Anfang der Resultatsliste und die unähnlichsten am Ende zu liegen kommen. Ein gravierender Nachteil nichtboolescher Suchmodelle ist jedoch, dass die Berechnung der Ähnlichkeit zwischen Dokumenten und Anfragen sowohl zeit- als auch rechenintensiv ist. Ausserdem erlaubt es das Vektorraummodell nicht, nur jene Dokumente als Resultat zurückzuliefern, die genau der Anfrage entsprechen[3].

Zu vermerken ist, dass nicht die Suche nach neuen Information Retrieval Methoden, sondern die Integration bestehender Methoden in heutige Datenbanksysteme Schwerpunkt dieser Arbeit ist. Auf die sehr weitläufigen Arbeiten auf dem Gebiet des Information Retrieval wird daher in dieser Arbeit nicht weiter eingegangen. Für eine Einführung in das Information Retrieval wird auf [Sal75, Rij81, SM83, Sal89] verwiesen.

Modell der vorliegenden Arbeit

Im Rahmen der vorliegenden Arbeit wird die Integration textueller Information in relationale Datenbanksysteme untersucht. Im Mittelpunkt steht die transaktionsorientierte Verwaltung von und die Suche nach *semistrukturierten* Dokumenten.

An das System wird die Anforderung gestellt, die Suche nach semistrukturierten Objekten zu unterstützen, wobei bei Textattributen eine Suche nach einzelnen Wörtern möglich sein muss.

[3] [WZRW89] stellt eine Methode vor, mit welcher boolesche Anfragen durch eine geeignete Wahl des Anfragevektors annäherungsweise beantwortet werden können.

Obwohl in der Vergangenheit wiederholt auf die Vorzüge nichtboolescher Suchmodelle gegenüber booleschen hingewiesen wurde, wird in der vorliegenden Arbeit für die Dokumentensuche das boolesche Suchmodell verwendet. Die hierbei für die Auswertung boolescher Suchanfragen eingesetzte Zugriffsstruktur (invertierte Listen) kann jedoch auch für die Auswertung nichtboolescher Anfragen eingesetzt werden [Coo88]. Die vorgestellten Indexstrukturen sowie die zugehörigen Operationen auf diesen sind daher Basis für die Auswertung nichtboolescher Suchanfragen.

2.2 Volltextsuche in Datenbanksystemen

Heute verfügbare Datenbanksysteme sind auf die leistungsorientierte Verarbeitung strukturierter Dokumente ausgelegt. Um häufig gestellte Benutzeranfragen effizient beantworten zu können, stellen die meisten Datenbanksysteme mehr oder weniger mächtige Möglichkeiten zum physischen Entwurf der gespeicherten Daten zur Verfügung. Beispiele hierfür sind Zugriffsstrukturen in Form von Bäumen, das Clustering von Daten nach Merkmalsklassen (horizontale Partitionierung) und die vertikale Partitionierung. Mit Hilfe dieser Strukturen können Punkt- und Bereichsanfragen auf den indexierten Daten effizient, d.h. mit einer geringen Anzahl von Sekundärspeicherzugriffen, beantwortet werden.

Im Gegensatz hierzu unterstützen heutige Datenbanksysteme die Verwaltung von Fliesstextattributen semistrukturierter Dokumente nicht oder nur eingeschränkt. Oracle und Sybase beispielsweise beschränken sich in der Verarbeitung von Volltextattributen ausschliesslich auf die Speicherung der Texte in Attributen der Typen `VARCHAR` oder `RAW`[4]. Attribute dieser Typen können nur eingeschränkt in Suchkriterien verwendet werden: Der Test auf Gleichheit sowie die Suche nach Teilstrings (Präfixe, Postfixe, Infixe) werden vom Datenbanksystem nicht unterstützt. Besteht ein Suchkriterium nur aus Attributen dieser beiden Typen, wird die Anfrage durch das sequentielle Lesen des Datenbestands, verbunden mit einem Vergleich zwischen jedem einzelnen Dokument und der Anfrage, beantwortet. Werden nebst Textattributen auch andere Suchkriterien spezifiziert, werden diese zuerst — wenn möglich mit Hilfe einer Zugriffsstruktur — ausgewertet und anschliessend wird die Suchbedingung auf den Textattributen geprüft.

Das sequentielle Durchsuchen einer Relation ist ein legitimer Ansatz, der eine effiziente Anfragebeantwortung im Fall kleiner Datenmengen oder unselektiver Anfragen erlaubt. Sobald jedoch Datenkollektionen von mehreren Hunderttausenden oder gar Millionen Dokumenten zu durchsuchen sind, werden die aus einer sequentiellen Suche resultierenden Antwortzeiten für den Benutzer inakzeptabel. Für die Suche in grossen Kollektionen müssen daher Indexstrukturen zur Verfügung gestellt werden, die eine effiziente Suche nach Dokumenten mit gewissen Merkmalen in den Volltextattributen erlauben. Welche Merkmale wie aus einem Text ermittelt werden, ist dabei stark von der Applikation und der Semantik des Textes[5] abhängig. In einer Vielzahl von Applikationen handelt es sich bei Merkmalen eines Textes — im Information Retrieval oftmals auch *Deskriptoren* genannt — um individuelle Wörter, Wortfragmente oder Phrasen. Diese Deskriptoren können dabei aus den Texten entweder *manuell* oder *automatisch*

[4] Auf die kommerzielle Erweiterung von Oracle um Textindexstrukturen (*SQL*TextRetrieval* und *SQL*TextServer*) wird in Abschnitt 2.4.2 eingegangen.

[5] Aus offensichtlichen Gründen wird für die Extraktion von E-Mail-Adressen ein anderer Algorithmus eingesetzt als für die Extraktion von Geburtsdaten.

gewonnen werden. Bei einer manuellen Gewinnung, auch *manuelle Deskribierung* genannt, weist ein Spezialist einem Text eine Menge von Deskriptoren zu. Diese Art der Deskribierung ist sehr zeitaufwendig und wird nur selten, beispielsweise in Bibliotheken, eingesetzt. In den meisten Information Retrieval Systemen werden die Deskriptoren automatisch durch ein Programm ermittelt. Oftmals — so auch in der vorliegenden Arbeit — kommt dabei ein *wortbasierter* Algorithmus zum Einsatz. Bei diesem wird in einem ersten Schritt ein Text in seine einzelnen Wörter zerlegt und häufig vorkommende, sogenannte *Stopwörter*, werden entfernt. In einem zweiten Schritt werden die verbleibenden Wörter auf ihre Wortstämme reduziert und etwaige Duplikate entfernt. Die so letztlich verbleibenden Wörter sind die Deskriptoren des Textes.

Nach der Extraktion der Deskriptoren werden diese — eventuell zusammen mit ergänzenden Angaben, beispielsweise ihrer Position innerhalb des Textes — in Zugriffsstrukturen eingefügt. [Fal85, GBYS92] gibt einen Überblick über eine Vielzahl einsetzbarer Zugriffsstrukturen einschliesslich deren Verarbeitung. Von besonderer Bedeutung — in der Vergangenheit sowie in der Gegenwart wiederholt untersucht und eingesetzt — sind dabei *Patricia–Bäume, Signaturen* und *invertierte Listen*:

- *Patricia–Bäume* [GBYS92], kurz PAT-Trees, sind Präfixbäume, die ein effizientes Suchen nach Wörtern und Phrasen, beispielsweise *to be or not to be*, erlauben. Die Proximity–Suche, also die Angabe einer maximalen Distanz zwischen zwei oder mehreren Suchbegriffen, wird ebenfalls unterstützt.

- *Signaturen* und *Referenzstrings* [FC84, Sch77] erlauben die Suche nach Wörtern und Wortfragmenten (beliebige Maskierung einzelner oder mehrerer Buchstaben). Sie liefern jedoch aufgrund ihrer internen Struktur immer eine Obermenge der gesuchten Antwortmenge, die im Anschluss an die Suche mittels weiterer Methoden, beispielsweise einer linearen Suche, auf die endgültige Antwortmenge reduziert werden muss (engl.: false drops elimination).

- *Invertierte Listen* [Knu73, HFRYL92] unterstützen Punkt- und Präfixanfragen ohne Notwendigkeit einer False Drops–Elimination wie im Fall von Signaturen.

In Datenbanksystemen werden im Allgemeinen dynamische Datenbestände verwaltet, die von vielen Benutzern gleichzeitig abgefragt und geändert werden. Um jederzeit Anfragen effizient beantworten zu können, sind gleichzeitig mit jeder Änderung des Datenbestands die Zugriffsstrukturen über den betroffenen Daten nachzuführen, um Inkonsistenzen zu vermeiden. Es können daher in Datenbanksystemen nur solche Zugriffsstrukturen zum Einsatz kommen, die ein effizientes Ändern im laufenden Betrieb erlauben: PAT-Trees gestatten ein effizientes Hinzufügen neuer Dokumente; das Löschen und Ändern von Dokumenten hingegen ist nur mit grossem Aufwand möglich. Signaturen müssen im Fall grosser Dokumentenkollektionen in einem Baum, einem sogenannten *Signaturbaum* [Dep89], gespeichert werden. Invertierte Listen schliesslich erlauben es, Dokumente jederzeit hinzuzufügen und zu löschen.

Bei der Auswahl einer geeigneten Zugriffsstruktur muss des weiteren die Frage geklärt werden, welche Art von Suchanfragen der Benutzer typischerweise stellt. In vielen Anwendungen sind die Anfragen wortbasiert, d.h. es wird nach dem Auftreten bestimmter Wörter oder Wortkombinationen innerhalb eines Dokuments gefragt, wobei die einzelnen Wörter oftmals am

Ende maskiert werden. Mit Hilfe der erwähnten Zugriffsstrukturen (PAT–Trees, Signaturen und invertierte Listen) können solche Anfragen effizient beantwortet werden.

Im Rahmen dieser Arbeit werden invertierte Listen für die Indexierung von Volltextdaten eingesetzt, da diese

- effizient in einem relationalen Datenbanksystem physisch gespeichert und verarbeitet werden können. Dabei sind keine Änderungen am Datenbanksystem selbst notwendig, was bei PAT–Trees und Signaturen nicht der Fall ist.
- im laufenden Betrieb geändert werden können. Dies heisst, dass Benutzer neue Dokumente einfügen und alte löschen können, während andere Benutzer gleichzeitig nach Dokumenten suchen. Dies ist in vielen Anwendungen, beispielsweise einem *Newsfeed* einer Presseagentur, notwendig, da neue Daten sofort verfügbar sein müssen und somit ein Nachführen des Datenbestands offline nicht möglich ist.

Im folgenden Abschnitt wird gezeigt, wie invertierte Listen in einem relationalen Datenbanksystem logisch und physisch modelliert und verarbeitet werden können. Die Verwaltung von Indexstrukturen wird dabei nur funktional betrachtet. Die Problematik der Indexverwaltung im Mehrbenutzerbetrieb wird in Kapitel 5 eingehend diskutiert.

2.3 Physische Modellierung invertierter Listen

Im letzten Abschnitt wurde angesprochen, dass Fliesstexte mittels Deskriptoren approximiert werden können. Mathematisch kann dies wie folgt definiert werden: Gegeben seien Dokumente d_i, welche in Form von Tupeln einer Relation

$$D(\underline{\text{Dok-ID}}, a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m)$$

modelliert sind. Der Primärschlüssel der Relation D ist durch *Dok-ID* gegeben, wobei für die nachfolgenden Betrachtungen angenommen wird, dass *Dok-ID* vom Typ `Zahl` ist. Bei a_i $(0 \leq i \leq n)$ handelt es sich um Attribute mit *Standarddatentyp* (Zahl, Datum, String) und bei t_j $(0 \leq j \leq m)$ um Attribute vom Typ *Volltext*.

Mathematisch gesehen besteht zwischen dem Primärschlüssel *Dok-ID* und den Attributen a_i und t_j eine funktionale Abhängigkeit

$$\text{Dok-ID} \rightarrow a_i \ (0 \leq i \leq n)$$

$$\text{Dok-ID} \rightarrow t_j \ (0 \leq j \leq m)$$

Werden die Textattribute t_j jeweils durch eine Menge von Deskriptoren $\{Des\}$ approximiert, gilt ferner

$$\text{Dok-ID} \rightarrow t_j \rightarrow \{\text{Des}_{t_j}\} \ (0 \leq j \leq m)$$

Um Anfragen der Form $\sigma_{x \subset t_j}(D)$[6] effizient beantworten zu können, muss die Inverse dieser Funktion berechnet werden:

$$\text{Des}_{t_j} \rightarrow \{\text{Dok-ID}\}$$

[6]Leseweise: Gesucht sind alle Dokumente der Kollektion D, welche den Deskriptor x im Attribut t_j enthalten.

In einem relationalen Datenbanksystem kann die Ausprägung dieser Funktion logisch gesehen in einer Relation

$$\text{NestedInvList}_{t_j}(\underline{\text{Des}}, \{\text{Dok-ID}\})$$

mit Primärschlüssel *Des* gespeichert werden. Da relationale Datenbanksysteme — mit Ausnahme von Prototypsystemen wie [Pau88a] — nur die Speicherung von Relationen erlauben, welche sich in der Ersten Normalform befinden, muss *NestedInvList* durch Entnestung in

$$\text{InvList}_{t_j}(\underline{\text{Des, Dok-ID}})$$

überführt werden. Diese Relation befindet sich in Erster Normalform. Mit Hilfe dieser Relation können alle Dokumente der Relation D, welche in einem Volltextattribut t_j den Deskriptor x enthalten durch Auswertung des Ausdrucks

$$D \times \pi_{\text{Dok-ID}}(\sigma_{\text{Des}=x}(\text{InvList}_{t_j}))$$

ermittelt werden.

Beim physischen Entwurf einer Datenbank werden die im Rahmen des logischen Entwurfs gewonnenen logischen Schemata auf physische Schemata abgebildet. Die Wahl der physischen Speicherungsstruktur wird dabei nachhaltig von den Operationen (Hinzufügen neuer, Ändern oder Löschen bestehender Daten sowie Anfragen von Daten) auf den Daten beeinflusst.

Invertierte Listen, wie sie obenstehend als $NestedInvList_{t_j}$ bzw. $InvList_{t_j}$ eingeführt wurden, können in einem heutigen relationalen Datenbanksystem im wesentlichen auf drei Arten modelliert werden: Als Bitvektoren, Mengen von Dokumenten-IDs oder in Form "normaler" Relationen. Diese Realisierungsvarianten werden folgend vorgestellt und ihre Vor- und Nachteile diskutiert.

2.3.1 Bitlisten

Speicherungsstruktur

Die vorgestellte Funktion $\text{Des}_{t_j} \rightarrow \{\text{Dok-ID}\}$ kann dahingehend interpretiert werden, dass für jeden Deskriptor festgehalten wird, in welchen Dokumenten er im Attribut t_j auftritt.

Graphisch kann dies in Form einer zweidimensionalen Matrix IL^{t_j} dargestellt werden (siehe Abbildung 2.4 auf Seite 22). Sofern die Dokumente und Deskriptoren jeweils durch fortlaufende Zahlen identifiziert werden, gilt für die einzelnen Elemente $\text{IL}_{i,j}^{t_j}$ der Matrix:

$$\text{IL}_{i,j}^{t_j} = \begin{cases} 1 & \text{falls der Deskriptor } i \text{ im Attribut } t_j \text{ des Dokuments } j \text{ enthalten} \\ 0 & \text{sonst} \end{cases}$$

Betrachtet man die Matrix zeilenweise, können die einzelnen Elemente als Nullen und Einsen eines Bitstrings interpretiert werden. Jeder dieser Bitstrings kann zusammen mit dem Identifikator des Deskriptors als Tupel einer Relation

$$\text{InvListBitlist}_{t_j}(\underline{\text{Des}}, \text{Bitstring})$$

gespeichert werden. *Des* ist dabei vom Typ `Zahl` und *Bitstring* vom Typ `BLOB`.

Um Anfragen der Art $\sigma_{\text{Des}=x}(\text{InvListBitlist}_{t_j})$[7] effizient zu beantworten, wird über dem At-

[7] d.h.: Gesucht sind alle Dokumente, die im Attribut t_j den Deskriptor x enthalten.

tribut *Des* ein eindeutiger Index (unique index) angelegt.

Nachfolgend werden die notwendigen Operationen betrachtet, die für das Warten der Bitlisten bei Einfügen, Löschen und Ändern eines Dokuments notwendig sind.

Operationen

Einfügeoperation Algorithmus 1 zeigt das Einfügen eines Dokuments d in die Dokumentenkollektion D.

```
did=did+1
INSERT d INTO D
for j = 1 to m do
  Extrahieren der Deskriptoren aus t_j → {M}
  for i = 0 to |M| do
    SELECT Bitstring FROM InvListBitlist_{t_j} WHERE Des=M_i
    length=LENGTH(Bitstring)
    if length < did then
      Extend(Bitstring, did)
    end if
    Set(Bitstring, did)
    UPDATE Bitstring
  end for
end for
```

Algorithmus 1: Bitlisten-Indexierung: Einfügen eines Dokuments

Hierbei wird implizit angenommen, dass den Dokumenten fortlaufende Nummern als Dokumentenidentifikation *(did)* zugewiesen werden. Das Einfügen eines Dokuments bedeutet somit das Anhängen weiterer Bits an die einzelnen Postinglisten[8]. Eine Wiederverwendung von Dokumenten-IDs bereits gelöschter Dokumente ist möglich. In diesem Fall wird der Bitstring nicht vergrössert.

Die im Algorithmus verwendete Operation `Extend(Bitstring, m)` vergrössert einen bestehenden Bitstring auf m Bits und setzt alle Bits zwischen *length* (Länge des Bitstrings vor dem Einfügen des neuen Dokuments) und l auf den Wert Null. Diese Operation kann vermieden werden, wenn beim Einfügen eines neuen Dokuments *alle* Bitstrings um jeweils ein Bit vergrössert werden. Dies ist jedoch leistungsmässig nicht praktikabel, weil grosse Datenmengen gelesen, geändert und wieder auf das Sekundärspeichermedium geschrieben werden müssten.

Der Aufwand für das Einfügen von Deskriptoren wird überwiegend von der Funktionalität des unterliegenden Datenbanksystems bestimmt. Kann dieses keine Teil-BLOBs lesen und schreiben, ist der Aufwand für das Ändern eines Bitstrings proportional zu dessen Länge und somit zur Grösse der Dokumentenkollektion D. Wird das Lesen und Schreiben von Teilobjekten unterstützt, ist der Aufwand geringer, da nur ein Teil der Bitstrings gelesen und geschrieben werden muss.

[8]Postingliste=Liste aller Dokumenten-IDs jener Dokumente, die ein bestimmtes Merkmal enthalten

Löschoperation Das Entfernen eines Dokuments d mit der Dokumentenidentifikation *did* aus der Kollektion D wird in Algorithmus 2 gezeigt. Analog zum Löschen von Dokumenten, die nur

```
SELECT * FROM D WHERE Dok-ID=did→ d
for j = 0 to m do
  Extrahieren der Deskriptoren aus t_j → {M}
  for i = 1 to |M| do
    SELECT Bitstring FROM InvListBitlist_{t_j} WHERE Des=M_i
    Clear(Bitstring, did)
    UPDATE Bitstring
  end for
end for
DELETE FROM D WHERE Dok-ID=did
```

Algorithmus 2: Bitlisten-Indexierung: Löschen eines Dokuments

aus Standarddatentypen bestehen, muss auch hier das Dokument vorab gelesen werden, um ein effizientes Löschen seiner Deskriptoren zu ermöglichen. Alternativ könnte die `FOR`–Schleife durch den Ausdruck

```
for all Bitliste do
  SELECT Bitstring ...
  if IsSet(Bitstring, did) then
    Clear(Bitstring, did)
    UPDATE Bitstring
  end if
end for
```

ersetzt werden. In diesem Fall werden alle Bitlisten unabhängig davon gelesen, ob das Bit *did* gesetzt ist oder nicht. Wie bei der Alternative für das Einfügen von Dokumenten (Erweiterung jeweils aller Bitlisten) ist auch dies mit Leistungseinbussen gegenüber dem Algorithmus 2 verbunden.

Der Aufwand für das Löschen von Dokumenten ist gleich gross wie für das Einfügen neuer Dokumente.

Anfragebeantwortung Um eine boolesche Anfrage der Form[9]

$$\begin{aligned} \text{Anfrage} &= \text{DisExp}_1 \vee \text{DisExp}_2 \vee \ldots \vee \text{DisExp}_{n-1} \vee \text{DisExp}_n \\ \text{DisExp}_i &= \text{KonExp}_{i1} \wedge \text{KonExp}_{i2} \wedge \ldots \wedge \text{KonExp}_{im_i} \\ \text{KonExp}_{ij} &= (\text{Name}_{ij}, \text{Wert}_{ij}) \end{aligned} \tag{2.1}$$

zu beantworten, muss gemäss Algorithmus 2.3.1 vorgegangen werden. In diesem Zusammenhang bedeutet $\text{KonExp}_{ij} = (\text{Name}_{ij}, \text{Wert}_{ij})$, dass dem gesuchten Dokument im Attribut Name_{ij} der Deskriptor Wert_{ij} zugeordnet sein soll. Das Symbol $\oplus$ steht hierbei für die bitweise *OR*–Operation und $\otimes$ für die bitweise *AND*–Operation.

Wird konsequent nach dem oben angeführten Algorithmus vorgegangen und auf die Auswertung gemeinsamer Teilausdrücke verzichtet (also keine *common subexpression elimination* vorgenommen), ist der Aufwand proportional zur Anzahl der Suchterme mal der Länge der Bitlisten (Anzahl der Dokumente in der Kollektion).

[9] Ausdruck in disjunktiver Normalform

```
DispExp=Bitliste mit allen Bits auf Null gesetzt
for i = 1 to n do
  SELECT Bitstring INTO KonExp_i
  FROM InvListBitlist_Name_i1 WHERE Des= Wert_i1
  for j = 2 to |Wert_i| do
    SELECT Bitstring INTO temp
    FROM InvListBitlist_Name_ij WHERE Des= Wert_ij
    if j = 1 then
      KonExp_i=temp
    else
      KonExp_i=KonExp_i⊗ temp
    end if
  end for
  DispExp⊕KonExp_i
end for
```

Algorithmus 3: Bitlisten-Indexierung: Suche von Dokumenten

Diskussion

Zipf stellt in seiner Abhandlung [Zip49] über das menschliche Verhalten fest, dass die einzelnen Wörter der englischen Sprache — dies trifft auch bei fast allen anderen Sprachen zu — in einem Text nicht gleichverteilt auftreten. Ordnet man die in einem Text vorkommenden Wörter nach ihrer absteigenden Auftretenshäufigkeit, findet man, dass das Produkt aus der Auftretenshäufigkeit und der Position eines Wortes in dieser Häufigkeitsrangliste nahezu konstant ist, d.h. sehr viele Wörter kommen in nur wenigen Dokumenten, sehr wenige in sehr vielen vor. Diese Art der Verteilung ist unter dem Begriff *Zipf'sche Verteilung* bekannt geworden.

Aufgrund dieser Verteilung sind viele Bitlisten nur sehr schwach besetzt. Oftmals ist nur ein einziges Bit gesetzt, die anderen Bits haben den Wert Null.

Obwohl der Benutzer bei Verwendung eines sehr selektiven Suchbegriffs nur mit wenigen Trefferdokumenten rechnen kann, wird der Aufwand zur Ermittlung der Trefferdokumente durch die Grösse der Dokumentenkollektion und nicht durch die Anzahl der Treffer bestimmt. Dieser Nachteil kann durch eine Komprimierung, wie sie beispielsweise in [MZ94, MZK95] vorgeschlagen wird, vermindert werden. Hierdurch erhöht sich jedoch der Aufwand für das Löschen und Ändern[10] von Dokumenten, da die komprimierten Bitlisten zu grossen Teilen gelesen, dekomprimiert, geändert, komprimiert und wieder gespeichert werden müssen. Die Verwendung komprimierter Bitlisten eignet sich daher für statische Dokumentenkollektionen oder solche, bei denen nur Dokumente hinzugefügt, nicht aber gelöscht oder geändert werden. Die Verwendung in einer stark dynamischen Umgebung mit vielen Löschungen ist leistungsmässig problematisch.

Bitlisten haben ferner den Nachteil, dass die gängigen Datenbanksysteme serverseitig keine Bitoperationen (`IsSet, Set, Clear`) anbieten. Alle vorgestellten Algorithmen müssen daher auf der Klientenseite realisiert werden, sofern nicht eine neue Funktion für das Durchführen von Bitoperationen in das bestehende System integriert werden kann. Die klientenseitige Abarbeitung bedeutet auf der einen Seite eine geringere Belastung für den Server, auf der

[10] Verweise auf neue Dokumente können effizient am Ende der Bitlisten eingefügt werden.

anderen Seite aber eine durch den Transfer verursachte erhöhte Belastung des Netzwerks sowie des Klientenrechners.

Die Modellierung invertierter Listen in Form der vorgestellten Bitlisten ist aufgrund des enormen Aufwands nicht zu empfehlen.

2.3.2 Dokumenten-IDs in BLOBs

Speicherungsstruktur

Eine Alternative zur Speicherung der Dokumenten-IDs in Form von Bitlisten, die aufgrund der Zipf'schen Verteilung von Merkmalen in den meisten Fällen sehr schwach besetzt sind, ist die Speicherung in einer Relation

InvListBLOB(Des, Dok-ID-BLOB)

Im Gegensatz zur Bitlistenrealisierung werden im Attribut *Dok-ID-BLOB* die Dokumenten-IDs nicht jeweils in Form eines Bits pro Dokument codiert, sondern die Dokumenten-IDs werden direkt gespeichert. In Programmiersprachen–Terminologie handelt es sich bei *Dok-ID-BLOB* um ein Attribut des Typs `ARRAY OF INTEGER`, wenn davon ausgegangen wird, dass der Identifikator eines Dokuments eine Zahl ist.

Operationen

Die Algorithmen 4, 5 und 6 zeigen Funktionen für das Einfügen, Suchen und Löschen von Dokumenten.

```
INSERT d INTO D
for j = 1 to m do
  Extrahieren der Deskriptoren aus t_j → {M}
  for i = 1 to |M| do
    SELECT OR CREATE Dok-ID-BLOB FROM InvListBLOB_{t_j}
    WHERE Des=M_i
    Dok-ID in Dok-ID-BLOB einfügen
    UPDATE Dok-ID-BLOB
  end for
end for
```

Algorithmus 4: BLOB-Indexierung: Einfügen eines Dokuments

Diskussion

Unterstützt das Datenbanksystem das Lesen und Schreiben von Teil-BLOBs, ist für das Einfügen eines neuen Deskriptors mit demselben Aufwand wie für das Einfügen von Bitlisten zu rechnen. Der Aufwand für das Löschen eines Dokuments hingegen ist grösser: Während in der Bitlistenrealisierung bekannt ist, welches Bit beim Löschen auf Null zu setzen ist, muss in der

Text Search Using Database Systems Revisited
— Some Experiments —

Helmut Kaufmann and Hans-Jörg Schek

Swiss Federal Institute of Technology (ETH Zürich)
Department of Computer Science—Databases
{kaufmann, schek}@inf.ethz.ch

Abstract. With the increasing availability of information in electronic form, the integration of textual data into database systems is becoming more and more important. Motivated by recent technology development, we describe how a preprocessor for simple text retrieval can be realised on top of a relational database system. This approach shows a surprisingly good performance compared to a commercially available information retrieval system and compared to another relational preprocessor product for text search.

1 Introduction and Motivation

In the late seventies and early eighties a considerable amount of research was devoted to the comparison of database systems (DBS) and information retrieval systems (IRS) and to a synthesis of these. Principle differences between both are described in [Rij81]. Specifically, discussions of data model issues are given in, for example, [SP82, DKA+86]; architectural issues in [Bil82, Sch84, LDE+85, LKD+88, Fuh89]. Prototype systems built on top of relational systems are described in [Mac79, SSL+83]. Our own early experience was gained by a prototype system [Pol80] built on top of IBM's SQL/DS by using the reference string indexing method [Sch77]. Although these early attempts showed some promise, the consensus was more or less that relational systems were not really suitable for text retrieval. Many groups started to build new next generation database systems, e.g. [SR86, SW86, SPSW90], rather than putting a document management and search preprocessor on top of existing database systems.

Now, more than ten years later, we believe that this discussion must be resumed for the following reasons:

- **Technology Evolution:** Relational database technology has evolved together with the dramatic changes in hardware and communication technology. Sophisticated client-server architectures and transaction technology provide parallel search and update of many users in a scalable way. They more and more use multi-processor hardware and provide not only inter- but also intra-transaction parallelism.
- **Higher Level of Abstraction:** Increasingly, relational database systems are being used as storage managers upon which sophisticated object managers and application-oriented tools are built. Some consider the relational

Abbildung 2.3: Semistrukturiertes Dokument

DispExp={}
for $i = 1$ **to** n **do**
  SELECT Dok-ID-BLOB INTO KonExp_i FROM $\text{InvListBLOB}_{\text{Name}_{i1}}$
  WHERE Des= $Wert_{i1}$
  for $j = 2$ **to** $|\text{Wert}_i|$ **do**
    SELECT Dok-ID-BLOB INTO temp FROM $\text{InvListBLOB}_{\text{Name}_{ij}}$
    WHERE Des= $Wert_{ij}$
    $\text{KonExp}_i = \text{KonExp}_i \cap \text{temp}$
  end for
  $\text{DispExp} = \text{DispExp} \cup \text{KonExp}_1$
end for

Algorithmus 5: BLOB-Indexierung: Suchen von Dokumenten

	Dok_1	Dok_2	Dok_3	Dok_4	Dok_5	...	Dok_n
Deskriptor_1	0	0	1	1	0	...	1
Deskriptor_2	1	1	0	0	0	...	1
Deskriptor_3	0	0	0	0	0	...	0
⋮							
Deskriptor_{m-1}	0	1	1	0	0	...	1
Deskriptor_m	0	0	0	0	1	...	0

Abbildung 2.4: Speicherung invertierter Listen in Bitarrays

SELECT * FROM D WHERE Dok-ID=did$\rightarrow d$
for $j = 1$ **to** m **do**
  Extrahieren der Deskriptoren aus $t_j \rightarrow \{M\}$
  for $i = 1$ **to** $|M|$ **do**
    SELECT Dok-ID-BLOB FROM InvListBLOB_{t_j} WHERE Des=M_i
    Entfernen der Dok-ID *did* aus dem Dok-ID-BLOB
    UPDATE Dok-ID-BLOB
  end for
end for

Algorithmus 6: BLOB-Indexierung: Löschen eines Dokuments

BLOB–Variante zuerst die Dokumenten-ID im BLOB gesucht und anschliessend entfernt werden. Zu bemerken ist, dass die Sucheffizienz von der Länge des zu lesenden BLOBs abhängig ist. Angenommen, für die Speicherung einer Dokumenten-ID werden 40 Bits benötigt, ist die BLOB-Realisierung bezüglich Transfer der Daten vom Sekundär- in den Primärspeicher effizienter als die Bitlistenvariante, falls die Selektivität eines Deskriptors geringer als $\frac{1}{40}$ ist, d.h. weniger als 2.5 Prozent beträgt. Dies ist gemäss Zipf'scher Verteilung für einen Grossteil der Deskriptoren der Fall.

Zur Länge von BLOBs: Um ein effizientes Löschen von Dokumenten zu ermöglichen, ist es empfehlenswert, eine Postingliste in kurze Teilpostinglisten aufzuteilen. In diesem Fall müssen beim Löschen nur kleine BLOBs gelesen und wieder geschrieben werden.

Der Engpass bei der Verarbeitung invertierter Listen ist im Transfer dieser vom Sekundär- in den Primärspeicher begründet. Da eine kleine Menge langer, benachbart gespeicherter Teilpostinglisten schneller von Disk gelesen werden kann als eine grosse Menge kurzer, hat die Anzahl der Teilpostinglisten einen entscheidenden Einfluss auf die Verarbeitungsgeschwindigkeit der einzelnen Operationen. Die maximale Grösse der einzelnen Teilpostinglisten ist daher unter Berücksichtigung der Auftretenshäufigkeit der Operationen Lesen, Einfügen und Löschen festzulegen.

Abschliessend ist in Bezug auf die Verwendung von BLOBs zur Speicherung invertierter Listen festzustellen, dass in einem realen System eine Kombination der vorgestellten Speicherungsvarianten empfehlenswert ist: Postinglisten unselektiver Deskriptoren werden am effizientesten in Bitlisten (1 Bit pro Dokument), Postinglisten selektiver Deskriptoren hingegen vorteilhaft in Form von `ARRAY OF Dok-ID` gespeichert. Das Wachstum der in beiden Fällen zur Speicherung der Postinglisten verwendeten BLOBs sollte dabei in Abhängigkeit von der Auftretenshäufigkeit der Lese-, Einfüge- und Löschoperationen beschränkt werden.

2.3.3 Relationen

Speicherungsstruktur

Eine Alternative zur Modellierung invertierter Listen mittels BLOBs ist die 1:1 Abbildung der logischen Relation

$$\text{InvList}_{t_j}\,(\underline{\text{Des, Dok-ID}})$$

auf eine physische Relation

$$\text{InvListRelation}(\underline{\text{Des, Dok-ID}})$$

Diese physische Speicherungsform, bei der beide Attribute Standarddatentypen besitzen, wurde bereits in [Mac79, Mac83], vorgeschlagen.

Um ein effizientes Ermitteln aller Dokumenten-IDs für einen bestimmten Deskriptor d zu ermöglichen, wird — wie in der Bitlistenrealisierung — ein Zugriffspfad über den Attributen *Des* und *Dok-ID* angelegt [KS95]. Der aus diesem Index resultierende Leistungsvorteil bei der Beantwortung von Anfragen wird untenstehend eingehend besprochen.

Operationen

Anfragebeantwortung Eine Anfrage, die wiederum in disjunktiver Normalform gemäss Gleichung 2.1 vorliegt, wird mittels in Relationen gespeicherter invertierter Listen gemäss Algorithmus 7 beantwortet.

```
DisExp={}
for i = 1 to n do
  SELECT Dok-ID INTO KonExp_i
  FROM InvListRelation_Name_i1 WHERE Des=Wert_ij
  for j = 2 to m_i do
    SELECT Dok-ID INTO temp
    FROM InvListRelation_Name_ij WHERE Des=Wert_ij
    KonExp_i=KonExp_i∩temp
  end for
  DisExp=DisExp ∪ KonExp_i
end for
```

Algorithmus 7: Relationen-Indexierung: Suchen von Dokumenten

Um die einzelnen Postinglisten effizient miteinander schneiden ($\cap$) und vereinigen ($\cup$) zu können, müssen die von den einzelnen `SELECT`–Anweisungen retournierten Dokumenten-IDs in sortierter Form vorliegen.

Eine Sortierung liegt vor, wenn die Relation *InvListRelation* physisch nach den Attributen *Des* und *Dok-ID* sortiert gespeichert wird. Dies kann beispielsweise durch Speicherung der Daten

in einem B*–Baum erreicht werden, dessen Schlüssel durch die Attribute *Des, Dok-ID* gegeben ist.

Heute verfügbare Datenbanksysteme erlauben es normalerweise nicht, die interne Repräsentation einer Relation frei zu wählen. Sie ermöglichen somit keine Speicherung einer Relation in einer B*–Baumstruktur. Durch Anlegen eines eindeutigen Index (unique index) über den Attributen der Relation *InvListRelation* kann jedoch derselbe Effekt erzielt werden. Eine einzelne `SELECT`–Anweisung kann dann allein durch Zugriff auf den B*–Baum beantwortet werden. Durch das Anlegen des B*–Baums wird das gespeicherte Datenvolumen mehr als verdoppelt, da die Daten nicht nur in der Relation, sondern auch zusätzlich im B*–Baum gespeichert werden. Obwohl dies keinen negativen Einfluss auf die Sucheffizienz hat, wirkt sich die Duplizierung auf das Leistungsverhalten beim Einfügen, Löschen und Ändern von Dokumenten ungünstig aus.

Eine Alternative zum Anlegen eines Index ist das physisch benachbarte Speichern (Cluster), wobei nach dem Attribut *Des* geklustert wird. Der Vorteil dieser Speicherungsmethode liegt im relativ geringen Sekundärspeicherbedarf, da das Clusterattribut in der Regel pro physischer Datenbankseite nur einmal gespeichert wird und — im Gegensatz zur B*–Baum–Variante — keine Daten dupliziert werden müssen. Nachteilig wirkt sich dabei aus, dass die Dokumenten-IDs nicht sortiert vorliegen.

Das Suchen in Information Retrieval Systemen ist die häufigste Operation. Es muss deshalb jene Speicherungsstruktur zum Einsatz kommen, die das Suchen von Dokumenten am effizientesten unterstützt: B*–Bäume.

Einfügeoperation Algorithmus 8 zeigt das Vorgehen beim Einfügen eines Dokuments in die Dokumentenkollektion. Nach dem Einfügen des Dokuments selbst müssen wiederum die Textindexstrukturen manuell nachgeführt werden. In einer ersten Fassung geschieht dies tupelorientiert, d.h. jeder Deskriptor wird mittels einer eigenen `INSERT`-Anweisung eingefügt. Dies ist mit hohem Kommunikationsaufwand verbunden, da jedes Tupel einzeln vom Klienten zum Server transferiert wird. Kommerzielle Datenbanksysteme, beispielsweise Oracle, erlauben nicht nur das Einfügen einzelner Tupel, sondern ganzer Tupelmengen. Durch den Einsatz mengenorientierter Operationen vermindert sich der Kommunikationsaufwand und damit die Zeit für das Einfügen der Deskriptoren in eine invertierte Liste deutlich.

Der Aufwand für das Einfügen der Deskriptoren ist ausschliesslich von der Kardinalität von M abhängig. Bei Ausführung der `INSERT`–Anweisung wird zuerst jeder Deskriptor in die Relation *InvListRelation* eingefügt und anschliessend der B*–Baum über dieser Relation gewartet. Nacheinander in derselben Transaktion eingeführte Deskriptoren kommen mit grosser Wahrscheinlichkeit auf derselben physischen Datenseite zu liegen. Beim Nachführen des Index muss jedoch mit dem Ändern einer Datenseite pro Deskriptor gerechnet werden, da die eingefügten Deskriptoren auf verschiedenen Blättern des B*–Baums zu liegen kommen.

Löschoperation Wie das Einfügen ist auch das Löschen von Dokumenten mit einem Nachführen der Indexstrukturen verbunden, wobei auch dieses entweder tupel- oder mengenorientiert gemäss Algorithmus 9 erfolgen kann. Der Aufwand für das Löschen eines Dokuments ist mit dem für das Einfügen eines Dokuments ident.

Tupelorientiertes Einfügen:
did=did+1
INSERT d INTO D
for $j = 1$ **to** m **do**
  Extrahieren der Deskriptoren aus $t_j \rightarrow \{M\}$
  for $i = 1$ **to** $|M|$ **do**
    INSERT INTO InvListRelation VALUES(M_i, did)
  end for
end for
Mengenorientiertes Einfügen:
did=did+1
INSERT d INTO D
for $j = 1$ **to** m **do**
  Extrahieren der Deskriptoren aus $t_j \rightarrow \{M\}$
  INSERT INTO InvListRelation VALUES($\{M\}$, did)
end for

Algorithmus 8: Relationen-Indexierung: Einfügen eines Dokuments

Tupelorientiertes Löschen:
SELECT * INTO d FROM D WHERE Dok-ID=did
for $j = 1$ **to** m **do**
  Extrahieren der Deskriptoren aus $t_j \rightarrow \{M\}$
  for $i = 1$ **to** $|M|$ **do**
    DELETE FROM InvListRelation$_{t_j}$
    WHERE Des=M_i AND Dok-ID=did
  end for
end for
Mengenorientiertes Löschen:
SELECT * INTO d FROM D WHERE Dok-ID=did
for $j = 1$ **to** m **do**
  Extrahieren der Deskriptoren aus $t_j \rightarrow \{M\}$
  DELETE FROM InvListRelation$_{t_j}$
  WHERE Des IN M AND Dok-ID=did
end for

Algorithmus 9: Relationen-Indexierung: Löschen eines Dokuments

Diskussion

Einer der Hauptvorteile der Realisierung invertierter Listen mittels Relationen ist die Möglichkeit, Anfragen direkt auf der Serverseite auszuwerten. Dies ist machbar, weil zur Ermittlung der Resultatmenge nur SQL–, nicht aber Bitlistenoperationen benötigt werden.

Die Kosten für das Einfügen neuer und das Ändern oder Löschen bestehender Dokumente sind im Gegensatz zur Bitlistenrealisierung nicht mehr von der Anzahl der Gesamtdokumente abhängig, sondern ausschliesslich von der Anzahl der zu verarbeitenden Deskriptoren des einzelnen Dokuments.

Ein gewichtiger Nachteil dieser Realisierung ist allerdings, dass für die Speicherung der Indexstrukturen viel Platz benötigt wird. Dazu folgendes Beispiel: Datenbanksysteme werden heute oftmals mit einer Seitengrösse von 4 Kilobyte betrieben. Angenommen, die Attribute *Des* und

Dok-ID belegen zusammen 8 Byte, haben in einem Blattknoten eines B*–Baums, in dem diese Attribute gespeichert werden, rund 500 Tupel Platz. Da ein Datenbanksystem in einem Index zu jedem Schlüssel auch einen Verweis auf das eigentliche Tupel speichert (bei Oracle eine ROWID von 6 Byte Länge), finden auf einer Blattseite nur noch 285 Tupel Platz. Geht man von einer durchschnittlichen I/O-Zeit von 10 Millisekunden aus, können pro Sekunde 100 I/Os durchgeführt und $285 * 100 = 28'500$ Dokumenten-IDs zur Verarbeitung in den Hauptspeicher transferiert werden. Diese Zahl reduziert sich um weitere 30 Prozent, wenn man von einem 70 prozentigen Füllgrad der Datenseiten ausgeht. Somit können vom Sekundärspeicher pro Sekunde lediglich 20'000 Dokumenten-IDs gelesen werden.

2.4 Verwandte Arbeiten

2.4.1 Forschungsarbeiten

Die Integration von Information Retrieval Systemen und Datenbanksystemen hat sich in den letzten Jahren zunehmend zu einem beliebten Forschungsgebiet entwickelt. Im Folgenden wird ein Überblick über eine Reihe von Arbeiten auf diesem Gebiet gegeben, wobei diese in *Architekturen und Datenmodelle, Transaktionsverwaltung* und *Invertierte Listen* unterteilt werden. Viele der Arbeiten behandeln nicht nur einen einzelnen dieser Aspekte. In diesem Fall werden sie jedoch nicht überall aufgeführt.

Architekturen und Datenmodelle

Bereits 1979 — wenige Jahre nach der Vorstellung des relationalen Datenmodells durch Codd und kurze Zeit nach Erscheinen der ersten SQL–Definition — zeigt Macleod [Mac79] die Schwierigkeiten auf, die bei der Verarbeitung textueller Daten mittels SQL auftreten. Zur Behebung dieser schlägt Macleod die Einführung eines Präprozessors "on top" bestehender Datenbanksysteme vor.

1981 zeigt Crawford [Cra81] auf, dass mit Hilfe des relationalen Datenmodells einfache, aber effiziente Indexstrukturen für Textdaten modelliert werden können. Biller [Bil82] stellt 1982 eine wachsende Nachfrage nach Informationssystemen zur Verwaltung semistrukturierter Dokumente fest. Diese Systeme werden von ihm als *DBMIRS* — ein Akronym aus Data Base Management and Information Retrieval System — bezeichnet. In seinem Artikel untersucht Biller, wie sich die für ein DBMIRS notwendigen Datenstrukturen in das ANSI–SPARC Schichtenmodell [ANS75] einordnen lassen. Er hält fest, dass auf der externen Ebene im wesentlichen drei Arten von Daten existieren: *Dokumente* (mit strukturierten und unstrukturierten Attributen), *Wörterbücher*[11] (Sammlungen von Deskriptoren, welche der Indexierung der Dokumente dienen) sowie *Beziehungen*[12] zwischen Dokumenten und Dokumenten, zwischen Dokumenten und Einträgen in den Wörterbüchern sowie zwischen verschiedenen Wörterbucheinträgen (zwecks Definition von Synonymen, Ober- und Unterbegriffen, ...). Die dem Benutzer auf externer Ebene zur Verfügung stehenden Operationen sind die klassischen Anfrageoperationen, wie Selektion und Projektion sowie diverse Textsuch- und Manipulationsoperationen,

[11] Im Originaltext als *Dictionaries* bezeichnet.
[12] Im Originaltext als *Relations* bezeichnet.

beispielsweise die Suche nach allen Dokumenten, denen bestimmte Wörterbucheinträge zugeordnet sind. Biller schlägt für die Realisierung von DBMIRS vor, die benötigten externen Schemata entweder direkt auf die konzeptionellen Schemata hierarchischer oder relationaler Datenbanksysteme mittels Applikationsprogrammen abzubilden. Biller hält in diesem Zusammenhang fest, dass die zur Verfügung stehenden Datenstrukturen aus Effizienzgründen nicht ideal sind. Er diskutiert daher die direkte Abbildung der externen Schemata auf interne, was jedoch eine Erweiterung bestehender Datenbanksysteme zur Folge hätte.

Schek und Pistor halten in [SP82] fest, dass Information Retrieval Systeme und Datenbanksysteme in der Forschung ursprünglich als zwei unabhängige Gebiete betrachtet und demzufolge als getrennte Systeme entwickelt wurden. Die Autoren führen jedoch wie Biller aus, dass in vielen realen Anwendungen kombinierte Systeme benötigt werden. Ein Beispiel hierfür ist das Patienteninformationssystem eines Krankenhauses. In einem solchen System werden u.a. Krankengeschichten verwaltet, die teils aus formatierter Information (beispielsweise Name, Sozialversicherungsnummer und Geburtsdatum eines Patienten), teils aus unformatierter Information (Diagnose, ...) bestehen. Schek und Pistor zeigen, dass traditionelle relationale Datenbanksysteme für die Verwaltung von Deskriptoren, die unformatierten Informationen zugewiesen werden, nur eingeschränkt verwendet werden können. Durch eine Erweiterung des relationalen Datenmodells um relationenwertige Attribute ist eine effiziente Verarbeitung solcher Daten möglich. Diese von Schek und Pistor vorgestellte Erweiterung ist unter dem Begriff *Non First Normal Form (NF^2)* bekannt. Ein Forschungsprototyp, der dieses Datenmodell unterstützt und für die Evaluation der Integration von Information Retrieval Funktionalität in Datenbanksysteme verwendet wird, ist *DASDBS* [Sch87, SPSW90, SSW91].

Eine Erweiterung relationaler Anfragesprachen um Operatoren zur automatischen Indexierung von Volltexten wird von Stonebraker et al. 1983 vorgeschlagen [SSL+83]. Konkret wird das Datenbanksystem *Ingres* unter anderem um einen Operator `BREAK` erweitert. Dieser hat die Aufgabe, die einzelnen Wörter aus einem Fliesstext zu extrahieren. Die extrahierten Wörter können anschliessend in einer geeigneten Zugriffsstruktur des Datenbanksystems, beispielsweise einem B*–Baum, gespeichert und zur effizienten Suche nach Dokumenten mit bestimmten Wörtern eingesetzt werden.

Mitte der achtziger Jahre wird immer deutlicher, dass zukünftige Datenbanksysteme nicht nur in den klassischen Datenbankbereichen wie Banken und Versicherungen eingesetzt werden, in denen ausschliesslich Standarddatentypen zur Modellierung der Daten verwendet werden, sondern auch in Nichtstandardbereichen mit Nichtstandarddatentypen wie Volltexten oder geographischen Datentypen. In der Folge werden neue Systemarchitekturen vorgestellt, die sich mit vertretbarem Aufwand um neue Datentypen, Funktionen und Zugriffsstrukturen erweitern lassen [LDE+85, Sto86, LKD+88].

Mit dem Aufkommen objektorientierter Datenbanksysteme wird auch die Integration von IR-Funktionalität in diese Systeme untersucht. Als Beispiel einer solchen Integration ist *Eclair* zu nennen [HW92]. Bei Eclair handelt es sich um eine Klassenbibliothek für das Datenbanksystem ONTOS, welche dem Benutzer ein Framework zur Realisierung von IR–Applikationen zur Verfügung stellt. Ein solches Framework hat den Vorteil, dass es einfach um neue Dokumentenarten, beispielsweise um Text- oder Tondokumente, erweitert werden kann und die einfache Einbindung neuer Suchfunktionen möglich ist. Diese Erweiterbarkeit von Eclair wird anhand einer Beispielapplikation mit 10'000 INSPEC Dokumenten funktional, nicht aber leistungsmässig, verifiziert.

Ein komplett anderer Ansatz wird mit dem Information Retrieval System *Spider* gewählt [Sch93, KS96]. Das Spider System besteht aus zwei Komponenten, einem sogenannten relationalen *DB-Server* (realisiert mittels Sybase) und einem *IR–Server*. Alle Dokumente werden im DB-Server in Form benutzerdefinierter Relationen gespeichert. Zugriffsstrukturen, welche für die Auswertung von IR–Anfragen mittels gewichtetem Retrieval auf diesen Dokumenten benötigt werden, sind im IR-Server — einer Eigenentwicklung — gespeichert. Benutzer von Spider haben nun einerseits die Möglichkeit, Datenbankanfragen an den DB-Server oder IR-Anfragen an den IR-Server zu senden. Datenbankanfragen können durch den DB-Server allein effizient ausgewertet werden. Bei IR-Anfragen wird mit Hilfe der im IR-Server gespeicherten Indexstrukturen eine Rangliste der Trefferdokumente, repräsentiert durch deren Primärschlüssel, erstellt. Anschliessend kann der Benutzer über diese Primärschlüssel auf die Dokumente, welche im DB-Server gespeichert sind, zugreifen. Da Spider aus zwei unabhängigen Systemkomponenten — von denen nur eine mit einer Transaktionskomponente ausgestattet ist — besteht, könnte es im Fall von Änderungen am Datenbestand zu Inkonsistenzen kommen. Dies wird mit Hilfe eines speziellen Transaktionskonzepts verhindert, auf welches im nächsten Abschnitt eingegangen wird.

Transaktionsverwaltung

Die beschriebenen Arbeiten beschäftigen sich fast ausschliesslich mit der datenbezogenen Konzeption eines kombinierten Information Retrieval Systems und Datenbanksystems. Die Problematik der Transaktionsverwaltung wurde in diesen Arbeiten jedoch nicht oder nur am Rande angesprochen.

Dass der Einsatz herkömmlicher Transaktionsmodelle, beispielsweise der Einsatz einer Transaktionsverwaltung mit Striktem Zwei-Phasen-Sperrprotokoll, zu Leistungsengpässen bei der Durchführung von Änderungen am Datenbestand im laufenden Betrieb führen kann, wurde jedoch sehr bald erkannt [DPS83]. Schek und Weikum schlagen deshalb in [Sch84, WS84] vor, in einem kombinierten IRS/DBMS–System die klassische einschichtige Transaktionsverwaltung durch eine mehrschichtige zu ersetzen und Sperren auf Dokumenten mit Hilfe eines prädikatbasierten Ansatzes zu realisieren. Diesem Gedanken folgen auch Lum, Dadam, Erbe et al. in [LDE+85].

Brown, Callan und Croft evaluieren in [BCC94] das Laden von Indexdaten für das probabilistische IR-System INQUERY anhand eines Teils der TIPSTER-Kollektion in Mneme [Mos90], einem transaktionsorientierten Speichersubsystem für persistente Objekte. Sie zeigen, dass durch ein batchorientiertes Laden der Deskriptoren mehrerer Dokumente die Indexierungszeiten im Vergleich zum dokumentenorientierten Einfügen reduziert werden können. Ein batchorientiertes Einfügen ist jedoch in vielen Anwendungen, in denen den Benutzern neue Daten unmittelbar zur Verfügung zu stellen sind, nicht möglich.

Erni untersucht mit dem Prototypsystem PlentyOfText [Ern95] die Möglichkeit, textuelle Indexstrukturen in Form invertierter Listen in das Datenbanksystem Plenty [Has95], welches zweischichtige Transaktionen unterstützt, zu integrieren.

Einen anderen Weg gehen Schäuble und Knaus mit Spider [KS96]. Sie argumentieren, dass in kombinierten IRS/DBS–Systemen nur ein Mindestmass an Transaktionskontrolle notwendig ist, da ein Unterschied zwischen Transaktionen in klassischen Datenbankanwendungen und

Transaktionen in Information Retrieval Anwendungen besteht. Schäuble und Knaus relaxieren deshalb die ACID Eigenschaften der klassischen Transaktionsverwaltung und geben die Isolation von Transaktionen auf. Dieser Ansatz, der er im Wesentlichen bedeutet, dass jede IR–Operation in Form einer eigenen Datenbanktransaktion abgearbeitet wird, erlaubt es, den Transaktionsdurchsatz, der durch die Verwendung klassischer Sperrverwalter limitiert wird, zu steigern.

Kamath und Ramamritham schliesslich unterstreichen in [KR96] die Notwendigkeit, den Benutzern eines Informationssystems jederzeit die aktuellsten Daten zur Verfügung zu stellen. Sie führen aus, dass Aktualität in vielen Anwendungen, beispielsweise einem Newsfeed, von grösster Bedeutung ist. Sie schlagen deshalb für IR–Systeme einen Scheduler vor, der zu Beginn einer Anfragebeantwortung prüft, ob zur Zeit eine andere Transaktion Dokumente einfügt, welche sich auf die Frage qualifizieren. Wenn ja, werden jene Daten der Einfügetransaktion, welche für die Anfragebeantwortung benötigt werden, schnellstmöglich in die Datenbank eingebracht; danach wird die Anfrage *vor* Ende der Einfügetransaktion gestartet. Dies ist möglich, da davon ausgegangen wird, dass Transaktionen niemals vom Benutzer zurückgesetzt werden und somit immer eine Forward–Recovery durchgeführt werden kann.

Sowohl das Transaktionsmodell von Schäuble und Knaus wie jenes von Kamath und Ramamritham unterscheiden sich von dem in dieser Arbeit verwendeten Modell insofern, alsdass es bei letzterem möglich ist, *mehrere* Operationen zu einer Transaktion zusammenzufassen und bereits durchgeführte Änderungen durch einen Transaktionsabbruch rückgängig zu machen. Die Ausführung von Datenbanktransaktionen, d.h. Transaktionen, die aus einer zu Beginn nicht notwendigerweise bekannten Sequenz von Operationen bestehen, ist daher möglich.

Aufbau und Verwaltung invertierter Listen

Ein batchorientiertes Verfahren zum Aufbau grosser invertierter Listen in einem transaktionslosen System wird von Tomasic, Garcia-Molina und Shoens in [TGMS93] diskutiert. Der Schwerpunkt dieser Arbeit liegt in der Evaluation von Verfahren für das effiziente Schreiben invertierter Listen auf ein Sekundärspeichermedium. Das vorgestellte Verfahren wird anhand von 868 Megabyte Usenet–News mit rund 140'000 Dokumenten verifiziert.

Dieses Verfahren geht — wie [BCC94] — davon aus, dass die zu indexierende Dokumentenkollektion entweder statisch ist oder ausschliesslich wächst, d.h. dass die bestehenden Dokumente nach dem Einfügen nicht mehr geändert oder gelöscht werden. Bei den vorgestellten Verfahren wird auf den Aspekt parallel arbeitender Benutzer (Leser wie auch Schreiber) nicht eingegangen. Eine genauere Betrachtung der verwendeten Algorithmen zeigt, dass diese nicht für den Einsatz in einer Mehrbenutzerumgebung ausgelegt sind: Die Rücksetzung von Transaktionen ist aufgrund der verwendeten Datenstrukturen nicht möglich.

Intensive Forschungsarbeit auf dem Gebiet invertierter Listen wird am CITRI in Melbourne betrieben. Der Schwerpunkt der dortigen Arbeit liegt auf der Komprimierung [ZMSD91, MZS94] sowohl der zu verwaltenden Texte als auch der zugehörigen invertierten Listen. Während die Komprimierung der Texte selbst nicht in Frage gestellt wird und diese in Bezug auf die Vermeidung I/O-bedingter Engpässe prinzipiell auch für Indexstrukturen von Vorteil wäre, existieren keine Untersuchungen über das Verhalten der Indexstrukturen für dynamische Datenbestände: Es wird wiederum davon ausgegangen, dass nur Daten hinzugefügt, nicht aber gelöscht oder

geändert werden. Zudem wird in allen Arbeiten die Transaktionsverwaltung, insbesondere der Aspekt parallel durchgeführter Änderungen, vernachlässigt.

Relationale Systeme und Spezialsysteme

Die Idee, bestehende relationale Datenbanksysteme um einen Textdatentyp zu erweitern und die für die Textsuche notwendigen Indexdatenstrukturen in Form invertierter Listen direkt in Relationen zu speichern, wurde erstmals von Macleod [Mac79] aufgegriffen.

Unabhängig von Macleod wird im Rahmen des AIM-Projekts der IBM Mitte der siebziger Jahre SQL/DS um Zugriffsstrukturen für Texte [Sch77] erweitert und das Leistungsverhalten des erweiterten Systems untersucht [EHPR81].

Obwohl alle diese Erweiterungen befriedigende Leistungen erbringen, kommt man dennoch zum Schluss, dass relationale Systeme für die Verwaltung textueller Daten und Indexstrukturen nicht optimal geeignet sind. Viele Forschungsgruppen beginnen deshalb mit dem Entwurf und der Realisierung neuer Datenbanksysteme, den sogenannten *Datenbanksystemen der nächsten Generation*. Diese erlauben die Integration neuer Datentypen, Operationen und Zugriffsstrukturen. Zwei bekannte Prototypsysteme, denen diese Überlegungen zugrundeliegen, sind Ingres/Postgres [SSL+83, SR86] und DASDBS [SW86, SPSW90].

Auch die aufgekommenen objektorientierten Datenbanksysteme werden neuerdings für die Speicherung textueller Indexstrukturen eingesetzt [HW92, BCCM93].

2.4.2 Kommerzielle Informationssysteme

Mit der Verfügbarkeit leistungsfähiger Systeme wurde der Gedanke, Textdaten und deren Indexstrukturen "on top" bestehender Datenbanksysteme zu verwalten, wieder aufgegriffen und kommerziell vermarktet. Als Beispiele für kommerzielle Datenbanksysteme zur Verwaltung von Texten seien *SQL*TR, SQL*TextServer* und *Illustra TextBlade* der Datenbanksysteme Oracle bzw. Illustra genannt [Ora94, Ill94].

Oracle Text*Retrieval und Text*Server

Oracle Text*Retrieval und Text*Server [SQL92, Ora94] sind Erweiterungen des Datenbanksystems Oracle, die es erlauben, Textdaten in die Datenbank einzufügen, sie auf einer Wortbasis zu indexieren und in Anfragen miteinzubeziehen. Text*Retrieval wie auch Text*Server realisieren Textindizes in Form von Bitlisten (siehe Abschnitt 2.3.1). Diese können entweder komprimiert oder unkomprimiert gespeichert werden. Beide Systeme zeigen ein gutes Leistungsverhalten, wenn keine oder nur sehr wenige Textinformationen im laufenden Betrieb geändert werden. Sollen jedoch dynamisch Dokumente eingefügt, geändert oder gelöscht werden, zeigen beide Systeme deutliche Leistungsschwächen. Der Hersteller empfiehlt daher, nur die Primärdaten im laufenden Betrieb zu ändern und die Indexstrukturen zu einem späteren Zeitpunkt nachzuführen.

Illustra TextBlade

Illustra [Ill95], die kommerzielle Variante von Postgres, erlaubt neue Datentypen mitsamt ihren Operationen und Zugriffsstrukturen als sogenannte *Blades* in das System zu integrieren. Zur Zeit sind Blades für räumliche Daten, Zeitreihen, statistische Operationen (ausschliesslich Operationen, keine neuen Typen und Zugriffsstrukturen), Bilder und Texte verfügbar.

Als Zugriffsstruktur für Texte verwendet Illustra eine proprietäre Indexstruktur, über die keine näheren Informationen, auch nicht über das Leistungsverhalten des Systems, verfügbar sind. Die grösste jemals in Illustra geladene Datenmenge beträgt laut Illustra 30'000 Dokumente (Patientendaten in einem Spital). Die Suchzeit auf diesen Daten — ohne parallel ablaufende Änderungsoperationen — beträgt weniger als eine Sekunde.

BASISPlus

BASISPlus [Inf90] ist ein hybrides Information Retrieval/Datenbanksystem. Es basiert auf dem relationalen Datenmodell, erlaubt jedoch im Gegensatz zu traditionellen relationalen Systemen die Definition mengenwertiger Attribute, d.h. die Relationen müssen sich nicht in Erster Normalform befinden. BASISPlus bietet die Möglichkeit, semistrukturierte Dokumente zu speichern, wortbasierte Zugriffsstrukturen über Textattributen anzulegen und diese zur effizienten Anfragebeantwortung einzusetzen.

BASISPlus ist das Nachfolgemodell von BASIS/K, einem Information Retrieval System, das seine Daten in Dateien verwaltet. Beim Umstieg von BASIS/K auf BASISPlus wurde das Dateisystem um eine Transaktionskomponente erweitert, um Mehrbenutzerfähigkeit auch im Fall mehrerer paralleler Änderungsoperationen zu ermöglichen.

Leistungsuntersuchungen, beispielsweise [KS95], zeigen, dass sich BASISPlus nicht für den Einsatz in einer sich dynamisch ändernden Dokumentenumgebung eignet: Selbst wenn keine parallelen Änderungen am Datenbestand vorgenommen werden, können Anfragen mit nur unbefriedigenden Antwortzeiten realisiert werden. Dies liegt grösstenteils an der Transaktionskomponente des Systems. Wird diese ausgeschaltet, d.h. die Datenbank im "read-only" Modus hochgefahren, verbessern sich die Antwortzeiten deutlich.

Eine Datenbank wird in BASISPlus daher oftmals in zwei Instanzen parallel betrieben. Auf der einen Instanz werden nur Anfragen vorgenommen, auf der anderen können Daten gelesen und geschrieben werden. Zu bestimmten Zeitpunkten werden Änderungen durch Umschalten zwischen den Instanzen verfügbar gemacht. Dieser Lösungsansatz ist unbefriedigend, da den Benutzern die neuesten Daten nicht unmittelbar nach ihrer Eingabe zur Verfügung stehen.

EuroSpider

EuroSpider [Sch93, KS96], das Produkt der gleichnamigen Firma, ist eine Kombination aus einem Information Retrieval System und einem Datenbanksystem. Während die Information Retrievalkomponente auf dem Forschungsprototypen *Spider* basiert, wird datenbankseitig das kommerziell erhältliche Datenbanksystem Sybase eingesetzt. EuroSpider legt Primärdaten, d.h. die Dokumente selbst, in Sybase ab. Die Dauerhaftigkeit einmal eingefügter Daten sowie die Konsistenz der Daten wird durch Sybase sichergestellt. Die für das schnelle Auswerten von

Anfragen — EuroSpider unterstützt gewichtetes Retrieval — benötigten Indexstrukturen werden von der Retrievalkomponente des Systems in regelmässigen Abständen inkrementell aus den geänderten, im Datenbanksystem gespeicherten Daten gewonnen. Anfragen auf Textdaten werden dann mit Hilfe der IR-Komponente beantwortet, Anfragen auf Datenbankdaten mittels Sybase. Die ACID-Eigenschaften, insbesondere die Isolation von Transaktionen, sind nur mit Einschränkungen gewährleistet.

Internet Search Engines

Mit dem Aufkommen der Datenautobahn hat auch die Menge der verfügbaren Information auf dem Internet stark zugenommen. Eine Reihe von Firmen bietet daher sogenannte *Search Engines* an. Eine Search Engine erhält vom Benutzer eine Menge von Stichwörtern und ermittelt alle Web-Seiten, welche alle oder einige dieser Stichwörter enthalten. Alle heute verfügbaren Search Engines reihen die Trefferdokumente nach gewissen Kriterien, z.B. zu Beginn jene Dokumente, welche alle Stichwörter enthalten, am Schluss jene, welche nur ein Stichwort enthalten.

Hauptproblem der Search-Engines ist das Einfügen neuer oder die Aktualisierung bestehender Daten: Diese Operationen laufen grundsätzlich in zwei Schritten ab. Zuerst muss ein neues oder bestehendes Dokument vom Web-Server, auf dem es sich befindet, zur Search Engine transferiert werden. Diese Aufgabe wird normalerweise von einem *Web Crawler*, einem Programm, das kontinuierlich eine Web-Seite nach der anderen liest, erledigt. Anschliessend werden die Stichwörter aus den einzelnen Web-Seiten extrahiert und in den Textindex der Search Engine eingefügt. Wird danach eine indexierte Web-Seite durch den Benutzer geändert oder gar gelöscht, ist diese Änderung für die Benutzer einer Search Engine erst sichtbar, wenn der Web-Crawler sie das nächste Mal liest und der Search Engine zur Neuindexierung übergibt. Da zwischen zwei aufeinanderfolgenden Aufrufen derselben Web-Seite durch einen Web-Crawler in der Regel Tage oder gar Wochen liegen, ist es belanglos, ob die Änderungen am Datenbestand für den Benutzer unmittelbar nach dem Lesen durch den Web-Crawler sichtbar sind oder erst ein paar Stunden später. Search Engines werden daher — ähnlich wie BASISPlus — oft in zwei Instanzen, einer Änderungsinstanz und einer Leseinstanz betrieben, zwischen denen in regelmässigen Abständen umgeschaltet wird.

2.5 Abgrenzung der Arbeit

Im Mittelpunkt steht die Untersuchung, inwiefern bestehende relationale Datenbanksysteme für die transaktionsorientierte Verwaltung textueller Daten und deren Indexstrukturen geeignet sind. Die vorliegende Arbeit geht somit in Richtung der Untersuchungen von Macleod [Mac79] und Stonebraker [SSL+83]. Diese Arbeiten behandeln jedoch die ausschliesslich funktionale Erweiterung bestehender Datenbanksysteme. Transaktionsorientierte Leistungsengpässe wie sie in [WS84] besprochen wurden, zeigen weder Macleod noch Stonebraker auf.

Verwandte Gebiete dieser Arbeit wurden auch bereits im Rahmen der Forschung der Gruppe Schek angesprochen: [Pau88b] stellt ein Speichersystem für Nichtstandarddaten vor, das sich zur Speicherung von Textdaten eignet. [ZPD90]. [Dep89] realisiert Signaturbäume zur effizienten Suche in Volltextdokumenten. In allen diesen Arbeiten wird im Gegensatz zur vorliegenden Arbeit jedoch kein kommerzielles relationales Datenbanksystem, sondern der Forschungs-

prototyp DASDBS [Sch87, SPSW90] eingesetzt. [WS84] zeigt auf, wie die für ein effizientes Information Retrieval benötigten Daten mit Hilfe eines Präprozessors in einem relationalen Datenbanksystem verwaltet werden können, welche transaktionellen Schwächen dieser Ansatz aufweist und wie diese mittels mehrschichtiger Transaktionen [Wei91] behoben werden können. Er bleibt jedoch den Nachweis, dass dieser Ansatz leistungsmässig auch wirklich von Vorteil ist und der erzielte Leistungsgewinn nicht von den Kosten für die Verwaltung von Mehrschichtentransaktionen übertroffen wird, schuldig. Dieser wird im Rahmen der vorliegenden Arbeit erbracht, womit diese im weiteren Sinn als Fortführung von [WS84] angesehen werden kann.

3 Transaktionsverarbeitung in Datenbanksystemen

Aufgabe eines Informationssystems im Allgemeinen und eines Datenbanksystems im Speziellen ist es, einer beliebigen Anzahl von Benutzern das gleichzeitige Bearbeiten einer grossen Menge von Daten zu ermöglichen. Interaktionen eines Benutzers mit dem System bestehen dabei oftmals nicht nur aus einer einzelnen Anweisung, beispielsweise einer Such- oder Einfügeoperation, sondern aus einer Sequenz davon. Eine solche wohldefinierte Folge von Operationen eines Benutzers wird als *Transaktion* bezeichnet. In diesem Kapitel werden zuerst die für diese Arbeit notwendigen Grundbegriffe der klassischen Transaktionsverwaltung der Vollständigkeit halber wiederholt, anschliessend wird auf transaktionsbedingte Leistungsengpässe im Zusammenhang mit der Verwaltung von Textindexstrukturen eingegangen. Abschliessend wird ein erweitertes Transaktionsmodell, *offen geschachtelte Transaktionen*, zur effizienten, transaktionsorientierten Verwaltung von Textindexstrukturen vorgestellt.

3.1 Grundbegriffe

Mit dem Begriff der Transaktion verbindet man im Datenbankbereich folgende vier fundamentale Eigenschaften [HR83, GR93], welche im Englischen unter dem Begriff *ACID properties*[1] bekannt sind:

- *Atomarität* (engl. *atomicity*): Alle Änderungen einer Transaktion werden entweder vollständig oder gar nicht durchgeführt. Auf Verlangen des Benutzers oder im Fehlerfall werden alle bereits durchgeführten Änderungen rückgängig gemacht.
- *Integritätserhaltung* (engl. *consistency*): Eine Transaktion transformiert eine Datenbank von einem konsistenten über einen eventuell inkonsistenten in einen wiederum konsistenten Zustand. Dies bedeutet, dass am Ende einer Transaktion alle für die Datenbank definierten Integritätsbedingungen erfüllt sind.
- *Isolation* (engl. *isolation*): Jeder Transaktion T wird der Eindruck vermittelt, es gäbe keine parallel laufenden Transaktionen. Änderungen dieser Transaktion T sind für andere Transaktionen bis zum Abschluss von T nicht sichtbar.
- *Dauerhaftigkeit* (engl. *durability*): Sobald eine Transaktion abgeschlossen wird, stellt das System sicher, dass die Änderungen auch tatsächlich in die Datenbank eingebracht werden. Dieses Einbringen wird auch im Fall eines Systemversagens (Hardware wie Software) garantiert.

[1] ACID=Atomicity, Consistency, Isolation und Durability

Um diese Eigenschaften gegenüber dem Benutzer garantieren zu können, müssen diese vom Datenbanksystem autonom, d.h. ohne Beeinflussung von aussen, sichergestellt werden. Hierzu bedienen sich Datenbanksysteme im Wesentlichen zweier Mechanismen, der sogenannten *Mehrbenutzerkontrolle* und der *Recovery*: Die Mehrbenutzerkontrolle koordiniert den konkurrierenden Zugriff der parallel arbeitenden Benutzer auf die Daten, die Recovery ist für die Wiederherstellung eines konsistenten Datenbankzustands im Fall eines gewollten oder ungewollten Transaktionsabbruchs verantwortlich. Im Zusammenhang mit der Mehrbenutzerkontrolle und der Recovery ist vielfach von *Datenbankobjekten, -operationen, -aktionen* und *-transaktionen* sowie *Schedules* die Rede. Diese Begriffe werden nachfolgend eingeführt.

Unter einer *Datenbank* D versteht man eine Menge *Datenbankobjekte* d_j, d.h. Informationseinheiten, auf die eine Reihe verschiedener *Datenbankoperationen* o_i $(1 \leq i \leq n)$ anwendbar sind. In Bezug auf Mehrbenutzerkontrolle und Recovery sind dies die kleinstmöglichen Informationseinheiten, die mittels Datenbankoperationen bearbeitet werden können.

Die Ausführung einer Datenbankoperation o_i auf einem Datenbankobjekt d_j bezeichnet man als *Datenbankaktion* $a_{i,j} = o_i(d_j)$. In der Regel führt ein Benutzer im Rahmen einer Arbeitseinheit T nicht nur eine, sondern mehrere Datenbankaktionen a durch. Diese Arbeitseinheiten, auch *Transaktionen* genannt, kennzeichnet man durch eine Indexierung der Aktionen und Operationen $a^T_{i,j} = o^T_i(d_j)$. Anfang und Ende einer Transaktion werden zudem durch spezielle Operationen markiert: b^T spezifiziert den Beginn, c^T das erfolgreiche Beenden und r^T den Abbruch einer Transaktion T. Beim Abschluss einer Transaktion mit c^T werden alle Änderungen dieser Transaktion persistent in der Datenbank gespeichert. Bei Transaktionsabbruch hingegen werden alle durchgeführten Änderungen am Datenbestand durch das System rückgängig gemacht.

Schliesslich wird die Abfolge von Datenbankaktionen $a^{T_l}_{i,j}$ verschiedener Transaktionen T_l als *Schedule* bezeichnet. Die *Ausführungsreihenfolge* der einzelnen Aktionen wird dabei durch das Symbol $<$ gekennzeichnet: $a^T_{i,j} < a^{T'}_{k,l}$ bedeutet, dass $a^T_{i,j}$ vor $a^{T'}_{k,l}$ ausgeführt wird.

Manipuliert immer nur ein einziger Benutzer Daten einer Datenbank, so befindet sich die Datenbank am Ende einer jeden Benutzertransaktion in einem korrekten Zustand, sofern die zur Bearbeitung der Daten verwendeten Programme korrekt sind. Ein serieller, durch die sequentielle Ausführung einer Folge von korrekten Transaktionen $T_1, T_2, \ldots, T_n$ definierter Schedule ist somit immer korrekt. Es ist aber Aufgabe des Datenbanksystems, auch im Fall mehrerer parallel ablaufender Transaktionen nur die Ausführung korrekter Schedules S zuzulassen.

Für die Beurteilung der *Korrektheit* eines Schedules wurde in der Vergangenheit bereits eine Reihe von Kriterien, beispielsweise *Final–State–Serialisierbarkeit*, *View–Serialisierbarkeit* oder *Konflikt–Serialisierbarkeit*, aufgestellt. In der Praxis, d.h. bei der Realisierung der Transaktionsverwaltung von Datenbanksystemen, konnten sich jedoch nur wenige dieser Kriterien durchsetzen: Das im Rahmen dieser Arbeit eingesetzte Datenbanksystem Sybase beispielsweise verwendet als Korrektheitskriterium die Konflikt–Serialisierbarkeit, das ebenfalls eingesetzte Datenbanksystem Oracle eine Variante der Mehrversionen–Serialisierbarkeit. Der in der gegenständlichen Arbeit realisierte Prototyp *TPM/ONT* schliesslich beruht auf der Mehrschichten–Serialisierbarkeit. Da nur diese Korrektheitskriterien für das Verständnis dieser Arbeit relevant sind, beschränkt sich auch die untenstehende Diskussion auf diese. Für eine ausführliche Erläuterung dieser und anderer Korrektheitskriterien wird auf [Pap86, BHG87, Ora95] verwiesen.

3.2 Klassische Einschichten–Transaktionsverwaltung

Heute erhältliche Datenbanksysteme stellen dem Benutzer semantisch reiche Operationen, beispielsweise SQL, zum Manipulieren ihrer Datenbankobjekte zur Verfügung. Intern werden diese auf zwei semantisch primitive Operationen, nämlich das *Lesen (Read)* und *Schreiben (Write)* von internen Datenbankobjekten, beispielsweise Datenbankseiten oder Tupel, abgebildet. Nachfolgend werden *Mehrbenutzerkontrolle* und *Recovery* von Datenbanksystemen betrachtet, in welchen nur diese beiden Operationen verfügbar sind.

Das wohl am häufigsten angewandte Korrektheitskriterium ist die Konflikt-Serialisierbarkeit. Grundidee der Konflikt–Serialisierbarkeit ist, dass Operationen auf Datenbankobjekten miteinander entweder verträglich sind oder nicht. Diese (Un-)Verträglichkeit wird im Allgemeinen in Form einer *Konfliktmatrix* con $= [\text{con}_{kl}]$ festgehalten. Im Fall des Read/Write–Modells gilt:

$con =$

	Read	Write
Read	+	–
Write	–	–

Ein Plus (+) bedeutet, dass zwei Operationen miteinander verträglich sind, ein Minus (−), dass sie es nicht sind. Ob ein gegebener Schedule S konflikt-serialisierbar ist oder nicht, geht aus der sogenannten *Serialisierungsordnung* $\preceq$ [Wei91] der an dem Schedule beteiligten Transaktionen hervor. Diese lässt sich aus einem gegebenen Schedule S und der Konfliktmatrix *con* wie folgt ableiten:

$$\preceq = \{(T_r, T_s) | \exists a_{k,j}^{T_r}, a_{l,j}^{T_s} : (a_{k,j}^{T_r} < a_{l,j}^{T_s}) \wedge (\text{con}_{kl} = -) \wedge (r \neq s)\}$$

Der Schedule S wird dann als *konflikt-serialisierbar* und damit korrekt bezeichnet, wenn seine Serialisierungsordnung gleich der Serialisierungsordnung irgendeiner seriellen Ausführung der an S beteiligten Transaktionen $T_1, T_2, \ldots$ ist.

In der Praxis ist es nicht möglich, Datenbankaktionen wahllos zuzulassen, zu bestimmten Zeitpunkten den Betrieb einer Datenbank zu unterbrechen, den seit dem Start der Datenbank aufgebauten Schedule S zu betrachten und dann zu entscheiden, ob dieser korrekt ist oder nicht. Es muss vielmehr inkrementell entschieden werden, ob die Ausführung einer Datenbankaktion $a_{i,j}^T$ durch eine Transaktion T einen korrekten oder inkorrekten Schedule erzeugen wird. In der Praxis wird daher für die Erzeugung eines Schedules eine Softwarekomponente, *Scheduler* genannt, eingesetzt, welche dank eines speziellen Zulassungsprotokolls für die einzelnen Datenbankaktionen im laufenden Betrieb feststellen kann, ob der bisherige Schedule korrekt ist oder nicht. Ein in der Praxis für ein solches Scheduling eingesetztes Protokoll ist das sogenannte *Strikte Zwei–Phasen–Sperrprotokoll (S2PL)*[2]. Bei diesem Protokoll wird vor der Durchführung einer Datenbankaktion $a_{l,j}^T = \text{o}_l^T(d_j)$ durch eine Transaktion T eine sogenannnte *Sperre* auf das Objekt d_j für die Ausführung der Operation $\text{o}_l^T(d_j)$ durch T angefordert. Eine Sperre anfordern heisst dabei, dass der Scheduler überprüfen muss, ob irgendeine andere, zur Zeit noch nicht abgeschlossene Transaktion T' bereits eine Datenbankaktion $a_{k,j}^{T'} = \text{o}_k^{T'}(d_j)$ durchgeführt hat (d.h. $a_{k,j}^T < a_{l,j}^{T'}$), wobei $con_{kl} = -$ ist. In diesem Fall wird die Ausführung von $a_{l,j}^T$ bis nach dem Abschluss der Transaktion T' durch $c^{T'}$ oder $r^{T'}$ verzögert. Am Ende der Transaktion T werden alle Sperren, welche im Laufe dieser Transaktion T angefordert wurden, wieder freigegeben.

[2] engl. Strict Two Phase Locking

Mit Hilfe des Strikten Zwei–Phasen–Sperrprotokolls kann die Isolation von Transaktionen sichergestellt werden. Voraussetzung hierfür ist, dass die einzelnen Transaktionsprogramme korrekt sind und alle Transaktionen erfolgreich beendet werden, d.h. keine Transaktion durch den Benutzer oder das System abgebrochen wird. Im Fall eines Transaktionsabbruchs oder eines Systemversagens kann es zu einer Verletzung der Eigenschaften Atomarität und Dauerhaftigkeit kommen. Um auch diese Eigenschaften garantieren zu können, muss dem Datenbanksystem folgende Information zur Verfügung stehen[3]:

- *Atomarität:* Eine sogenannte *Undo-Information* muss verfügbar sein. Die Undo-Information erlaubt es, den Zustand, in dem sich die Datenbank zu Beginn einer Transaktion befunden hat, wiederherzustellen. Diese Information wird benötigt, um Änderungen nicht mit c^T beendeter Transaktionen T rückgängig zu machen (sog. *Rückwärts Recovery* oder engl. *Backward Recovery*).

  In vielen Datenbanksystemen werden für die Backward Recovery sogenannte *Before Images* verwendet, d.h. vor Änderung eines Objekts wird sein momentaner Zustand festgehalten. Diese Before Images können im Fehlerfall wieder in die Datenbank eingespielt werden.

- *Dauerhaftigkeit:* Änderungen an Datenbankobjekten müssen protokolliert sein. Diese Information wird *Redo-Information* genannt. Ausgehend von einem initialen Datenbankzustand können mit Hilfe dieser Information sämtliche Änderungsoperationen am Datenbestand durch erfolgreich abgeschlossene Transaktionen rekonstruiert werden (sog. *Vorwärts Recovery* oder eng. *Forward Recovery*).

  In vielen Systemen besteht die Redo-Information aus sogenannten *After Images*, also dem Zustand des Objekts nach einer Änderungsoperation.

Da es auch während der Recovery zu einem Systemversagen kommen kann, muss sichergestellt werden, dass die zur Verfügung stehende Undo- und Redo-Information eine korrekte Wiederherstellung auch bei wiederholtem Systemversagen erlaubt. Werden für die Recovery die bereits erwähnten Before und After Images verwendet, wird von einer *zustandsorientierten* Recovery gesprochen. Diese ist idempotent, d.h. sie kann beliebig oft wiederholt werden. Bestehen Undo- und Redo-Information jedoch nicht aus absoluten Zuständen, sondern beispielsweise aus Deltas oder aus Informationen über die durchgeführte Operation, spricht man von einer *operationsorientierten* Recovery. Diese ist im Allgemeinen nicht idempotent, d.h. eine wiederholte Durchführung der Recovery mündet letztlich in einen inkonsistenten Datenbankzustand. Die Recoverykomponente muss daher registrieren, welche Objekte sich bereits in einem konsistenten Zustand befinden und welche nicht. Ein in diesem Zusammenhang oftmals angewandtes Verfahren, das auf sogenannten *Log-Satz-Nummern* basiert, wird in [GR93] beschrieben.

Anzumerken bleibt, dass sowohl das Schreiben von Undo- als auch Redo-Information sehr zeitaufwendig ist. Da sicherzustellen ist, dass diese Informationen persistent sind, müssen sie mittels synchroner Schreiboperationen auf ein Sekundärspeichermedium geschrieben werden. Mit heutigen Speichermedien werden hierzu grössenordnungsmässig 6–9 ms benötigt. Dies erscheint sehr gering, ist jedoch in zahlreichen Anwendungen mit sehr kurzen Transaktionen

[3] Es wird davon ausgegangen, dass das Datenbanksystem zu jedem Zeitpunkt geänderte Objekte in die Datenbank einbringen darf, also eine *Steal*–Strategie bei der Pufferverwaltung eingesetzt wird [HR83].

ein prozentuell zu berücksichtigender Faktor, insbesondere wenn die Transaktion auf Datenbankobjekten arbeitet, welche sich ausschliesslich im Hauptspeicher befinden: Der Zugriff auf den Hauptspeicher ist mit 60 ns rund 100'000 mal schneller als das Schreiben von Undo- oder Redo-Information.

Nachfolgend wird aufgezeigt, wie sich das Strikte–Zwei–Phasen–Sperrprotokoll auf die Verwaltung von Volltextdokumenten leistungsmässig auswirkt.

3.3 Textindexstrukturen: Leistungsengpass Zwei–Phasen–Sperrprotokoll

Das vorgestellte Strikte Zwei-Phasen-Sperrprotokoll hat zwei gravierende Nachteile: Es ist *blockierend* und *nicht verklemmungsfrei*. Blockierend bedeutet, dass eine Transaktion T unterbrochen wird, wenn auch nur eine einzige Sperre nicht gewährt werden kann. Nicht verklemmungsfrei heisst, dass eventuell zwei Transaktionen T und T' wechselseitig aufeinander warten und deshalb eine der beteiligten Transaktionen zurückgesetzt, d.h. abgebrochen werden muss. Hat die abgebrochene Transaktion bereits Datenbankobjekte verändert, müssen diese Änderungen rückgängig gemacht werden, sofern sie bereits in die Datenbank eingebracht wurden. Dies ist mit grossem Aufwand verbunden, falls die zurückzusetzende Transaktion viele Änderungen am Datenbestand vorgenommen hat.

Transaktionen, in deren Rahmen semistrukturierte Dokumente eingefügt werden, sind mit einer grossen Anzahl von Änderungen am Datenbestand verbunden. Gemäss [Zip49] kommen, wie bereits erwähnt, bestimmte Deskriptoren in fast allen Dokumenten vor, andere wiederum in sehr wenigen. Die Wahrscheinlichkeit, dass verschiedene Volltextdokumente gemeinsame Deskriptoren aufweisen, ist daher sehr gross. Werden im Rahmen gleichzeitig ablaufender Transaktionen gemeinsame Deskriptoren $\{des_i\}$ in die Datenbank eingefügt, behindern sich diese Transaktionen gegenseitig, da sie versuchen, die Deskriptoren gleichzeitig in die textuellen Indexstrukturen einzufügen: Angenommen, dass die textuellen Indexstrukturen logisch als invertierte Listen modelliert und in B–Bäumen physisch gespeichert und dass den einzufügenden Dokumenten aufsteigend sortierte Zahlen als Primärschlüssel zugeordnet werden, kommen die Indexeinträge für einen Deskriptor des_i, welcher in allen einzufügenden Dokumenten enthalten ist, auf derselben Blattseite zu liegen. Verwendet das Datenbanksystem ein Striktes Zwei-Phasen-Sperrprotokoll auf Seitenebene, behindern sich die Einfügetransaktionen gegenseitig, da zu jedem Zeitpunkt jeweils nur eine Transaktion die für das Einfügen notwendige Schreibsperre erhält.

Die oben unmathematisch mit "gross" bezeichnete Wahrscheinlichkeit, dass Dokumente, welche gleichzeitig durch verschiedene Transaktionen eingefügt werden, gemeinsame Deskriptoren enthalten und sich dadurch gegenseitig behindern, wurde bereits anhand eines einfachen und bereits in [DPS83] eingeführten Modells unter folgenden, vereinfachenden Annahmen analytisch untersucht:

- Jedes Dokument wird durch eine Menge von l Deskriptoren $\{des\}$ beschrieben.
- Deskriptoren treten in verschiedenen Dokumenten unabhängig voneinander mit einer gleichverteilten Wahrscheinlichkeit p auf.

- Eine Suchanfrage besteht aus k konjunktiv verknüpften Deskriptoren. Das heisst: Auf eine Anfrage qualifizieren sich jene Dokumente, welche alle k Deskriptoren enthalten.

Die Wahrscheinlichkeit, dass ein einzufügendes Dokument mindestens einen Deskriptor mit mindestens einem von n parallel einzufügenden Dokumenten gemeinsam hat – und sich somit Einfügetransaktionen gegenseitig behindern — beträgt

$$P_{Konflikt}(1\ Einfüger,\ n\ Einfüger) = 1 - (1-p)^{ln}$$

Zu bemerken ist, dass eine solche Behinderung semantisch betrachtet nicht notwendig ist, da sowohl der Endzustand der Datenbank als auch das Ergebnis der Einfügetransaktionen von der Einfügereihenfolge der Deskriptoren unabhängig ist.

Weisen die Dokumente mindestens zwei gemeinsame Deskriptoren des_1, des_2 auf, kann es zudem zu einer Verklemmung der beiden Einfügetransaktionen kommen: Fügt nämlich eine Transaktion T zuerst Deskriptor des_1 ein und T' den Deskriptor des_2, blockieren sich die beiden Transaktionen gegenseitig, wenn sie anschliessend versuchen, den jeweils anderen Deskriptor einzufügen. Um dies zu verhindern, muss die Deskriptormenge $\{des_i\}$ jedes einzufügenden Dokuments partiell geordnet werden.

Eine ähnliche Problematik zeigt sich, wenn eine Suchtransaktion parallel zu einer Einfügetransaktion durchgeführt wird: Haben beide Transaktionen auch nur einen einzigen gemeinsamen Deskriptor *des*, wird die Abarbeitung der einen Transaktion bis zum Ende der anderen verzögert, da die Lesetransaktion eine Lesesperre, die Einfügetransaktion eine inkompatible Schreibsperre auf *des* anfordert. Die Wahrscheinlichkeit, dass bei n parallel ablaufenden Einfügetransaktionen und einer Suchtransaktion mindestens ein einzufügendes Dokument mindestens einen (aber nicht alle) Suchdeskriptor enthält, beträgt

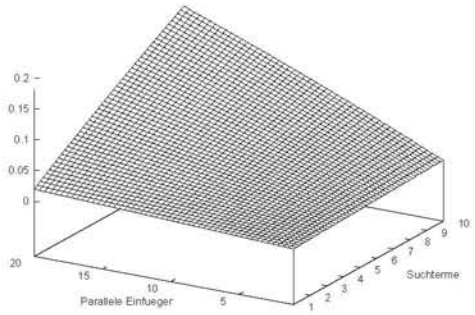
$$P_{Pseudokonflikt}(1\ Sucher,\ n\ Einfüger\ ) = 1 - (1-p)^{kn}$$

Die durch gemeinsame Deskriptoren entstehende gegenseitige Behinderung von Transaktionen ist vom semantischen Standpunkt aus nicht notwendig: Da keines der einzufügenden Dokumente *alle* Suchdeskriptoren enthält, qualifiziert sich keines dieser Dokumente auf die Suche. Der durch die Anforderung der Schreib-/Lesesperren auf den gemeinsamen Deskriptoren verursachte Konflikt auf der Indexebene wird daher als *Pseudokonflikt* bezeichnet.

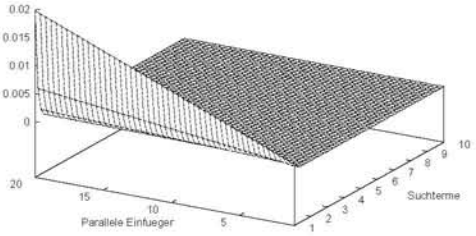
Die Wahrscheinlichkeit hingegen, dass sich mindestens eines der n parallel einzufügenden Dokumente auf die Anfrage qualifiziert und somit die Suchoperation nicht parallel zum Einfügen eines Dokuments durchgeführt werden kann, beträgt

$$P_{Konflikt}(1\ Sucher,\ n\ Einfüger) = 1 - (1-p^k)^n$$

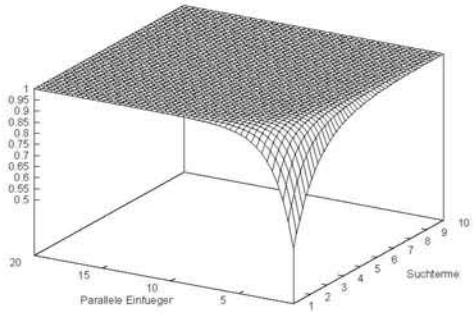
Diese Wahrscheinlichkeit, *Konfliktwahrscheinlichkeit* genannt, ist für beliebige n, p und k stets kleiner oder gleich der Wahrscheinlichkeit eines Pseudokonflikts. Abbildung 3.1 zeigt die Konflikt- und Pseudokonfliktwahrscheinlichkeit einer Suchtransaktion mit k ($1 \leq k \leq 10$) Suchtermen, wenn sie parallel zu n ($1 \leq n \leq 20$) Einfügetransaktionen für $p = 0.001$ (sehr selektive Suchterme) und $p = 0.5$ (unselektive Suchterme) ausgeführt wird. Die Graphiken zeigen, dass die Konfliktwahrscheinlichkeit mit zunehmender Anzahl Suchterme deutlich abnimmt. Die Pseudokonfliktwahrscheinlichkeit hingegen — und damit die semantisch gesehen unnotwendigen



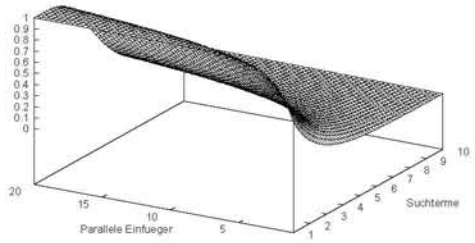
(a) Pseudokonfliktw'keit für p=0.001



(b) Konfliktw'keit für p=0.001



(c) Pseudokonfliktw'keit für p=0.5



(d) Konfliktw'keit für p=0.5

Abbildung 3.1: Pseudokonflikt- und Konfliktwahrscheinlichkeiten für eine Suchtransaktion parallel zu einer Menge von Einfügetransaktionen

Behinderungen von Suchtransaktionen durch Einfügetransaktionen und umgekehrt — nimmt mit zunehmender Anzahl Suchdeskriptoren drastisch zu.

In einem Information–Retrieval System findet man überwiegend reine Lesetransaktionen. Es ist daher wünschenswert, diese Transaktionen möglichst effizient, d.h. ohne Behinderung durch parallele Änderungstransaktionen, auszuführen. Dies ist jedoch mit einem konflikt-serialisierenden Datenbanksystem, das auf einem Strikten–Zwei–Phasen–Sperrprotokoll basiert, aufgrund von Pseudokonflikten nicht möglich. Nachfolgend wird daher das Korrektheitskriterium der Mehrversionen–Serialisierbarkeit, das die Ausführung von Lesetransaktionen ohne Behinderung durch parallele Änderungstransaktionen erlaubt, vorgestellt.

3.4 Mehrversionen–Serialisierbarkeit

In einem Datenbanksystem mit einer Mehrversionen–Transaktionsverwaltung erzeugt jede Datenbankaktion $a_{k,j}^T = o_k^T(d_j)$, welche ein Datenbankobjekt d_j verändert, eine neue *Version* dieses Objekts. Der Vorteil mehrerer Versionen eines Datenbankobjekts soll anhand eines Beispiels illustriert werden. Zwei Transaktionen T und T', welche eine Reihe von Lese– und Schreib–Operationen (o_r und o_w) auf zwei Postinglisten P_x und P_y durchführen, erzeugen folgenden Schedule:

$$o_r^T(P_x)\ o_r^{T'}(P_x)\ o_w^{T'}(P_x)\ o_r^{T'}(P_y)\ o_w^{T'}(P_y)\ o_r^T(P_y)$$

Dieser Schedule ist nicht konflikt-serialisierbar: T liest das Objekt P_x vor T', T' schreibt jedoch das Objekt P_y bevor T dieses liest.

Wäre die Transaktion T nur einen Bruchteil schneller gewesen, wäre folgender Schedule entstanden:

$$o_r^T(P_x)\ o_r^{T'}(P_x)\ o_w^{T'}(P_x)\ o_r^{T'}(P_y)\ o_r^T(P_y)\ o_w^{T'}(P_y)$$

Dieser Schedule ist konflikt-serialisierbar. Beide Transaktionen hätten somit erfolgreich abgeschlossen werden können. Da die Ausführungsreihenfolge jedoch nicht mehr rückwirkend geändert werden kann, hätte man der Transaktion T alternativ den Wert von P_y zur Verfügung stellen können, den es vor der Änderung durch T' hatte.

Verschieden Datenbankhersteller haben die Vorteile einer versionierenden Transaktionsverwaltung erkannt und verwenden diese als Grundlage für die Realisierung ihrer eigenen Transaktionsverwaltung. Oracle beispielsweise stellt seinen Anwendern in der Version 7.2[4] eine Transaktionsverwaltung mit folgenden Eigenschaften zur Verfügung [Ora95]:

- Vor Ausführung einer Operation o_i^T (SQL–Anweisung) einer Transaktion T wird der Zustand der Datenbank virtuell eingefroren und die Operation auf diesen — *Snapshot* genannten — Daten ausgeführt. Dieser Snapshot enthält nur Objekte d_j abgeschlossener Transaktionen. Die zu diesem Zeitpunkt gültigen Versionen der Datenbankobjekte werden mit d_j^{b,o_i^T} bezeichnet (b steht dabei für *before*).

- Eine Operation o_i^T, die ein Objekt ändert, erzeugt eine neue Version dieses Objekts. Diese wird mit d_j^{a,o_i^T} bezeichnet (a steht dabei für *after*).

- Operationen verschiedener parallel laufender Transaktionen, die Daten einfügen, behindern einander nicht, solange keine Integritätsbedingungen, insbesondere die Primärschlüsselbedingungen, verletzt werden. Dies ist auch bei Daten, die auf derselben Datenseite zu liegen kommen, möglich, da Oracle nicht auf Seiten-, sondern auf Tupelebene sperrt.

- Eine Operation o_l^T, welche eine Version eines Objekts d_j im Rahmen einer Transaktion T lesen will, wird durch eine vorgängige Änderungsoperation $o_u^{T'}$ auf dieses Objekt durch eine parallel zu T laufende Transaktion T' nicht behindert. Die der Leseoperation o_l^T zur

[4] Diese Version wurde für die Leistungsevaluation eingesetzt.

Verfügung gestellte Version d_j^{b,o_l^T} von d_j ist

$$d_j^{b,o_l^T} = \begin{cases} d_j^{a,o_u^T} & \text{falls } c^{T'} < o_l^T \\ d_j^{b,o_u^T} & \text{falls } o_l^T < c^{T'} \\ d_j^{b,o_u^T} & \text{falls } r^{T'} < o_l^T \end{cases}$$

Aus dieser Definition folgt, dass die durch eine Transaktion durchgeführten Änderungen nach deren Abschluss für alle anderen Transaktionen sofort sichtbar sind.

- Eine Änderungsoperation $o_u^{T'}$ einer Transaktion T', welche ein Objekt d_j ändern will, das durch eine parallel ablaufende Transaktion T vorgängig durch o_l^T gelesen wurde, wird durch diese Leseoperation nicht behindert. Die der Änderungsoperation zur Verfügung gestellte Version von d_j ist $d_j^{b,o_u^{T'}} = d_j^{b,o_l^T} = d_j^{a,o_l^T}$.

- Aus den beiden vorhergehenden Charakteristika folgt: Führt eine Transaktion T ein Leseoperation o_l^T auf ein Objekt d_j durch, das anschliessend im Rahmen einer Transaktion T' mittels $o_u^{T'}$ verändert wird, gilt für das erneute Lesen des Objekts d_j durch die Transaktion T mittels $o_{l'}^T$:

$$d_j^{b,o_{l'}^T} = d_j^{a,o_u^T} \neq d_j^{b,o_l^T} \text{ falls } o_l^T < c^{T'} < o_{l'}^T$$

  Das heisst, dass ein Datenbankobjekt bei wiederholtem Lesen unterschiedliche Werte annehmen kann (keine *repeatable reads*).

- Parallel laufende Transaktionen T und T', die mittels o_u^T und $T_u^{T'}$ parallel Änderungen an einem Objekt d_j durchführen wollen, behindern einander. Falls $o_u^T < o_u^{T'}$ ist die der Operation $o_u^{T'}$ zur Verfügung gestellte Version

$$d_j^{b,T'} = \begin{cases} d_j^{a,o_u^T} & \text{mit } c^T < o_u^{T'} \\ d_j^{b,o_u^T} & \text{mit } r^T < o_u^{T'} \end{cases}$$

  Mit $o_u^{T'} < o_u^T$ ist die der Operation o_u^T zur Verfügung gestellte Version

$$d_j^{b,T} = \begin{cases} d_j^{a,o_u^{T'}} & \text{mit } c^{T'} < o_u^T \\ d_j^{b,o_u^{T'}} & \text{mit } r^{T'} < o_u^T \end{cases}$$

Nachteil der von Oracle realisierten Transaktionsverwaltung ist, dass der erzeugte Schedule nicht notwendigerweise zu einem seriellen Schedule äquivalent ist, da Änderungen an Daten sofort nach Abschluss einer Änderungstransaktion für alle anderen Transaktionen sichtbar werden. In vielen Applikationen, vor allem im zeitkritischen OLTP–Bereich[5] wird dies zugunsten kürzerer Antwortzeiten und erhöhtem Durchsatz jedoch in Kauf genommen.

Zur Effizienz der Verwaltung mehrerer Versionen eines Objekts kann angemerkt werden, dass die verschiedenen Versionen eines Objekts nicht extra in der Datenbank gespeichert werden müssen, sondern aus der Undo-Information der Recovery–Komponente gewonnen werden

[5]Online Transaction Processing,

können. Dies hat den Nachteil, dass die hierzu notwendige Undo-Information unter Umständen sehr lange aufbewahrt werden muss, um langlaufenden Transaktionen den Zugriff auf eine bestimmte Version eines Objekts zu ermöglichen; das kann zu drastischen Leistungsengpässen führen, wenn grosse Teile des Undo-Logs nach der benötigten Version eines Objekts durchsucht werden müssen. Ist der zur Verfügung stehende Sekundärspeicherplatz zudem knapp bemessen, kann es vorkommen, dass Undo-Information gelöscht werden muss und eine Transaktion nicht mehr mit einer bestimmten Version bedient werden kann. In diesem Fall muss die Transaktion zurückgesetzt werden.

Mit Hilfe der in Oracle realisierten Transaktionsverwaltung ist es — wie bereits erwähnt — möglich, Lesetransaktionen ohne Behinderung durch gleichzeitig ablaufende Einfügetransaktionen abzuarbeiten. Versuchen jedoch zwei unabhängige Transaktionen, ein oder mehrere gemeinsame Datenbankobjekte — beispielsweise Postinglisten — zu ändern, ist dies weder mit der Transaktionsverwaltung von Oracle noch mit einer anderen versionierenden Transaktionsverwaltung möglich.

Nachfolgend wird deshalb ein weiteres Verfahren zur Mehrbenutzerverwaltung, die sogenannte *Mehrschichten–Transaktionsverwaltung*, vorgestellt. Diese erlaubt es, unter gewissen Umständen in Konflikt stehende Operationen, wie das Ändern von Postinglisten, parallel auszuführen, ohne die ACID-Eigenschaften zu verletzen.

3.5 Mehrschichten–Transaktionsverwaltung

Die vorangegangenen Diskussionen haben gezeigt, dass bei der Bearbeitung von Dokumenten massive Sperrkonflikte und somit Behinderungen der verarbeitenden Transaktionen auftreten können, die im Extremfall zum Abbruch einzelner Transaktionen führen. Das restriktive Halten von Sperren bis zum Ende einer Transaktion ist in vielen Fällen jedoch nicht notwendig, da dies aus Sicht des Benutzers keinen Einfluss auf das Endergebnis einer Transaktion hat.

Betrachtet man beispielsweise zwei parallel laufende Transaktionen T und T': Im Rahmen von T werden zwei Dokumente d_1 und d_2 in die Datenbank D eingefügt. Das Dokument d_1 enthält die Deskriptoren a und b, das Dokument d_2 nur den Deskriptor c. Im Rahmen der zweiten Transaktion T' werden jene Dokumente gesucht, welche die Deskriptoren a, b und c enthalten. Abbildung 3.2 zeigt einen möglichen Ablauf der beiden Transaktionen in einem Datenbanksystem mit Striktem Zwei–Phasen–Sperrprotokoll. Auffallend ist, dass das Lesen der Postingliste P_a durch Transaktion T' aufgrund der in Änderung befindlichen Postingliste P_a durch T bis zum Ende der Transaktion T verzögert wird. Diese Verzögerung kann in einem Datenbanksy-

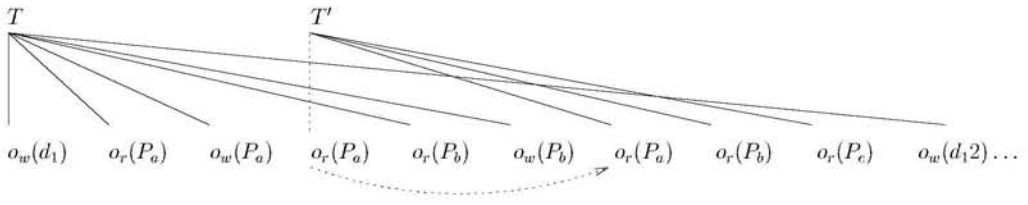


Abbildung 3.2: Einfügen und Suchen von Dokumenten ohne vorzeitige Sperrfreigabe

stem mit Strikten Zwei–Phasen–Sperren nur vermieden werden, wenn die Schreibsperren auf

die einzelnen Postinglisten unmittelbar nach erfolgter Durchführung einer Änderungsoperation freigegeben werden (Abbildung 3.3). In herkömmlichen Datenbanksystemen wird dieses

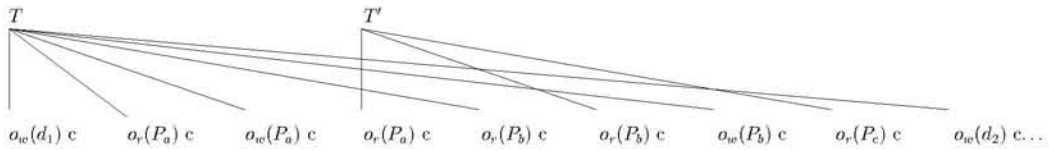


Abbildung 3.3: Einfügen und Suchen von Dokumenten mit vorzeitiger Sperrfreigabe

Freigeben durch ein Beenden der Datenbanktransaktion (Commit-Operation c) erreicht. Dieses Vorgehen hat jedoch den Nachteil, dass Atomarität und Isolation von Transaktionen nicht mehr gewährleistet sind:

- *Atomarität*: Versagt das System nach dem Einfügen von Dokument d_1 und des Deskriptors a, ist das Dokument persistent in der Datenbank gespeichert, jedoch nur teilweise indexiert.

- *Isolation*: Die parallel zur Einfügetransaktion T ablaufende Lesetransaktion T' liest die durch T geänderte Postingliste P_a bereits vor dem Ende von T und stellt fest, dass das Dokument d_1 den Deskriptor a enthält. Es wird somit ein inkonsistenter Zwischenzustand sichtbar.

  Im konkreten Fall würde ein Abbruch von T in Bezug auf T' jedoch ohne Folgen bleiben: Da die Lesetransaktion alle Dokumente ermittelt, welche a, b und c enthalten, d_1 jedoch nur a und b, nicht aber c enthält, qualifiziert sich d_1 nicht auf die Anfrage.

  Würde die Lesetransaktion nach Dokumenten suchen, welche ausschliesslich den Deskriptor b enthalten, hätte das Sichtbarwerden dieses Zwischenzustands gravierende Folgen: Wird T abgebrochen, hätte T' das Dokument d_1 als Treffer ermittelt, obwohl dieses Dokument nicht in der Datenbank existiert.

Um trotz frühzeitiger Sperrfreigabe sowohl die Atomarität als auch die Isolation von Transaktionen sicherzustellen, ist die herkömmliche Transaktionsverwaltung zu erweitern. Für das obige Beispiel heisst das:

- *Atomarität:* Es ist festzuhalten, welche Postinglisten bereits geändert wurden. Versagt das System den Dienst, kann entweder eine *Vorwärts–Recovery* durchgeführt werden, d.h. die nicht nachgeführten Textindexstrukturen werden nachgeführt, oder eine *Rückwärts–Recovery*, d.h. bereits durchgeführte Änderungen der Textindexstrukturen werden rückgängig gemacht und das Dokument wird aus der Datenbank entfernt.

- *Isolation:* Zu Beginn einer Suchanfrage ist zu prüfen, ob gleichzeitig ein Dokument eingefügt wird, das sich auf die Anfrage qualifiziert. Wenn ja, wird die Anfrage erst nach Abschluss der Einfügetransaktion zugelassen.

Das Problem der Verzögerung — und eventuell auch einer Verklemmung — von Transaktionen tritt ebenfalls auf, wenn zwei Transaktionen T und T' versuchen, gleichzeitig Dokumente

mit gemeinsamen Deskriptoren (beispielsweise a und c) in die Datenbank einzufügen (Abbildung 3.4). Auch in diesem Fall kann die Behinderung durch eine frühzeitige Sperrfreigabe reduziert und die Verklemmung verhindert werden, was jedoch wiederum den Verlust der Isolations- und Atomaritätseigenschaften der Transaktionen zur Folge hat.

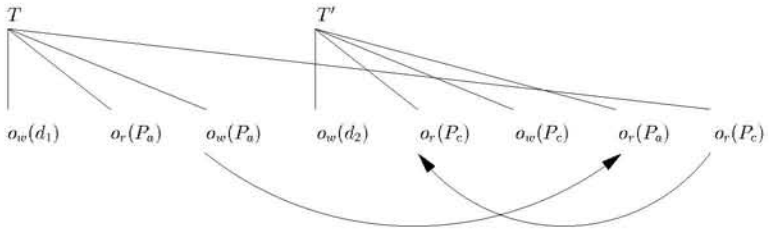


Abbildung 3.4: Paralleles Einfügen von Dokumenten ohne vorzeitige Sperrfreigabe

Die frühzeitige Freigabe von Sperren ist nicht nur im Fall von Textindexstrukturen von Vorteil, sondern kann auch in anderen Anwendungsgebieten zu einer deutlichen Leistungsverbesserung von Datenbankapplikationen beitragen. Das hinter dieser Erweiterung stehende Prinzip ist in der Literatur unter dem Namen *Mehrschichtentransaktionen*, einem Spezialfall *offen geschachtelter Transaktionen*, bekannt [Wei86, WS92].

Mehrschichtentransaktionen liegt der Gedanke zugrunde, Daten auf verschiedenen semantischen Ebenen mittels semantisch mehr oder weniger reichen Operationen zu manipulieren. Auf der untersten Ebene L_0 finden sich nur primitive Operationen, nämlich das Lesen und Schreiben atomarer Objekte, beispielsweise von Seiten des Sekundärspeichermediums. Auf der nächsthöheren Ebene L_1 finden sich semantisch reichere Operationen. Jede Operation auf Ebene L_1 besteht dabei aus einer Folge von Operationen auf Ebene L_0, die im Rahmen einer eigenständigen Transaktion auf Ebene L_0 ausgeführt werden. Auf der wiederum nächsthöheren Ebene L_2 finden sich semantisch noch reichere Operationen. Jede dieser Operationen wird auf eine Folge von Operationen der Ebene L_1 abgebildet, welche zusammen eine Transaktion der Ebene L_1 definieren. Grundsätzlich sind beliebig viele Ebenen L_i denkbar. In der Praxis findet man jedoch meist zwei- oder dreischichtige Systeme, da sich in nur wenigen Applikationen semantisch reichere Operationen auf noch höheren Ebenen formulieren lassen.

Für die Verwaltung von Textdokumenten ist ein dreischichtiges Datenbanksystem vorteilhaft [WS84]. Auf Ebene L_1 stellt das System folgende Operationen bereit:

- *InsDoc*$(a_1, a_2, \ldots, a_m, t_1, t_2, \ldots, t_n) \rightarrow$ *Dok-ID* fügt ein Dokument in die Datenbank ein und liefert den Schlüssel des Dokuments zurück. Zugriffsstrukturen über den Textattributen $t_1, t_2, \ldots, t_n$ werden nicht gewartet.
- *RetDoc(Dok-ID)* $\rightarrow a_1, a_2, \ldots, a_m, t_1, t_2, \ldots, t_n$ liest das Dokument mit Schlüssel *Dok-ID*.
- *DelDoc(Dok-ID)* löscht das Dokument mit Schlüssel *Dok-ID*. Verweise auf dieses Dokument in den Zugriffsstrukturen über den Textattributen werden nicht gelöscht.
- *InsDes(Index, Des, Dok-ID)* fügt einen Deskriptor *Des* für das Dokument *Did* in die textuelle Indexstruktur *Index* ein.

- *RetDes(Index, Des)* $\rightarrow \{$*Dok-ID*$\}$ retourniert die Schlüssel $\{$*Dok-ID*$\}$ jener Dokumente, welche den Deskriptor *Des* in der textuellen Indexstruktur *Index* enthalten.
- *DelDes(Index, Des, DOK-ID)* löscht den Verweis des Deskriptors *des* auf das Dokument *Dok-ID* aus der textuellen Indexstruktur *Index*.

Auf Ebene L_2 sind folgende Operationen definiert:

- *InsertDocument*$(a_1, a_2, \ldots, a_m, t_1, t_2, \ldots, t_n)$ fügt ein Dokument in die Datenbank ein und indexiert die textuellen Attribute $t_1, t_2, \ldots, t_n$. Dies geschieht durch Aufruf der L_1–Operation *InsDoc* und anschliessenden wiederholten Aufrufen der L_1–Operation *InsDes* für die Indexierung der Textattribute $t_1, t_2, \ldots, t_n$.
- *RetrieveDocument(Predicate)* $\rightarrow \{a_1, a_2, \ldots, a_m, t_1, t_2, \ldots, t_n)\}$ ermittelt jene Dokumente, welche einem Suchprädikat *Predicate* genügen. Dies erfolgt durch wiederholten Aufruf der L_1–Operation *RetDes* zur Ermittlung der Dokumenten-IDs der Trefferdokumente und anschliessenden Zugriffen auf diese mittels der L_1–Operation *RetDoc*.
- *DeleteDocument(Predicate)* löscht jene Dokumente, welche einem Prädikat *Predicate* genügen. Hierzu werden wiederum die Dokumenten-IDs der Trefferdokumente mittels *RetDes* ermittelt und diese sowie deren Indexstrukturen anschliessend durch wiederholten Aufruf der L_1–Operationen *DelDoc* und *DelDes* gelöscht.

In einem mehrschichtigen Datenbanksystem kann eine Transaktion T, die aus dem Einfügen zweier Dokumente besteht (einem Dokument d_1, charakterisiert durch die Deskriptoren a und b, sowie einem Dokument d_2 mit dem Deskriptor c), sowie eine Transaktion T', die nach Dokumenten mit Deskriptoren a, b und c sucht, mit Hilfe dieser Operationen gemäss Abbildung 3.5 ausgeführt werden. Betrachtet man nur den Schedule der Ebene L_0, stellt man fest, dass dieser

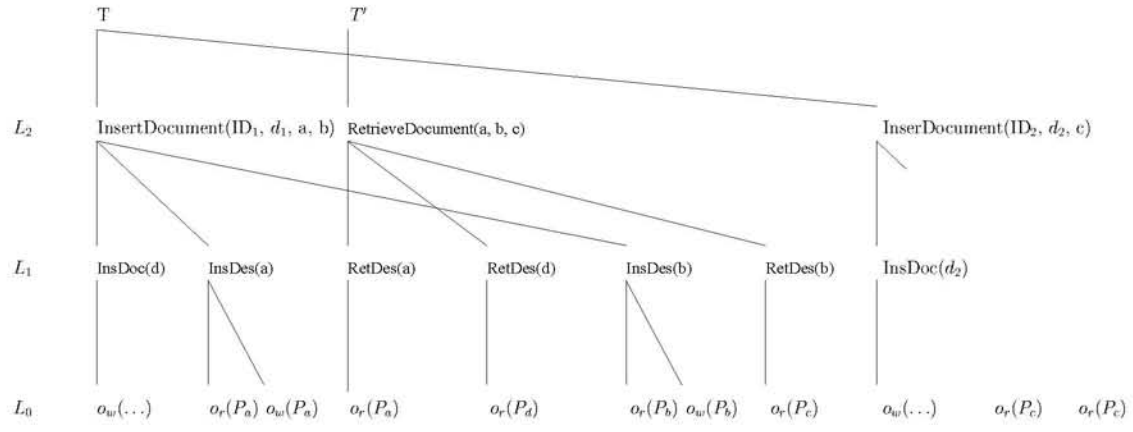


Abbildung 3.5: Texttransaktionen in Mehrschichtendatenbanksystemen

in Bezug auf die Transaktionen T und T' nicht konfliktserialisierbar ist: T schreibt zuerst die Postingliste P_a, die anschliessend von T' gelesen wird (also: $T < T'$), später schreibt T die Postingliste P_c, die bereits vorher von T' gelesen wurde (also: $T' < T$)[6]. Betrachtet man die

[6]Dies ist nur möglich, weil die Sperren auf Ebene L_0 jeweils am Ende einer L_1–Aktion freigegeben werden.

Transaktionen vom Standpunkt der semantisch höheren Ebene L_2, stellt man fest, dass das Ergebnis der beiden Transaktionen T und T' trotz eines Zyklus auf Ebene L_0 korrekt ist.

Um diese Korrektheit von Benutzertransaktionen $T, T', \ldots$ trotz einer frühzeitigen Sperrfreigabe zu garantieren, muss in einem ersten Schritt auf jeder Ebene L_i definiert werden, welche Operationen dieser Ebene miteinander kommutieren und deshalb ohne Einschränkungen parallel ausgeführt werden können. Auf der untersten Ebene L_0 findet man die bereits vorgestellten primitiven Schreib/Lese–Operationen mit der auf Seite 37 vorgestellten Konfliktmatrix. Für die Ebenen L_i $(i > 0)$ gilt: Zwei Operationen o_x und x_y einer Ebene L_i werden als *kommutierbar* bezeichnet, wenn für jeden Zustand σ der Datenbank gilt:

- Der nach aussen sichtbare Datenbankzustand, welcher aus der Ausführung von $a_x; o_y$ resultiert, ist derselbe wie jener, der durch $o_y; o_x$ entsteht.
- Die Operationen o_x und o_y haben dieselben Rückgabewerte, unabhängig davon, in welcher Reihenfolge die Operationen ausgeführt werden, sofern diese Zustände nach aussen sichtbar werden. Das heisst, es darf durchaus Operationen auf Ebene L_j $(j < i)$ geben, welche zwischen der Ausführung von $o_x(o_y(\sigma))$ und $o_y(a_x(\sigma))$ unterscheiden können.

Mit dieser Definition wird für die Textoperationen auf Ebene L_1 die Konfliktmatrix con^1 wie folgt definiert:

$con^1 =$

	InsDoc	RetDoc	DelDoc	InsDes	RetDes	DelDes
InsDoc	+	–	–	+	+	+
RetDoc	–	+	–	+	+	+
DelDoc	–	–	+	+	+	+
InsDes	+	+	+	–	–	+
RetDes	+	+	+	–	+	–
DelDes	+	+	+	+	–	–

Die Konfliktmatrix auf Ebene L_2 ist durch

$con^2 =$

	InsertDocument	RetrieveDocument	DeleteDocument
InsertDocument	+	–	–
RetrieveDocument	–	+	–
DeleteDocument	–	–	–

gegeben.

Um L_i–Aktionen auf einer Ebene L_i korrekt parallel ausführen zu können, wird auf jeder Ebene eine eigenständige Mehrbenutzerverwaltung benötigt. Diese garantiert die isolierte Ausführung der L_i–Transaktionen und somit die Isolation der L_{i+1}-Aktionen. Mehrschichtentransaktionen basieren — wie bereits erwähnt — auf dem Prinzip, dass Objekte auf verschiedenen semantischen Ebenen manipuliert werden, wobei das semantische Wissen um die Operationen und Objekte von Ebene zu Ebene zunimmt. Höhere Ebenen abstrahieren somit von den Realisierungsdetails der unteren Schichten. Stehen auf einer Ebene L_i zwei L_i–Aktionen miteinander in Konflikt, besteht daher auf jeder unterliegenden Ebene L_j $(j < i)$ zwischen zwei von diesen abstammenden L_j–Aktionen ebenfalls mindestens ein Konflikt (Axiom 1 [Wei86]). Um die Korrektheit eines Mehrschichtenschedules zu garantieren, muss die Mehrbenutzerkontrolle auf Ebene L_i dafür sorgen, dass solche in Konflikt stehenden L_i–Aktionen in derselben Reihenfolge serialisiert werden, wie deren L_j–Aktionen auf den unterliegenden Schichten. Diese Forderung kann beispielsweise durch die Realisierung eines jeweils eigenständigen Schedulers auf jeder

Ebene L_i erfüllt werden, wobei der L_i–Scheduler dafür zu sorgen sorgen hat, dass zwei in Konflikt stehende L_i–Aktionen auf jeder tieferen Schicht L_k $(k < i)$ in derselben Reihenfolgee serialisiert werden. Der Nachweis, dass der hieraus entstehende Schedule korrekt ist, wird in [Wei91] gegeben.

In Systemen, in denen die einzelnen Schichten unabhängig voneinander implementiert sind, kann diese Serialisierungsreihenfolge von Transaktionen auf Ebene L_{i-1} eventuell nicht durch den Scheduler der Ebene L_i von oben herab bestimmt werden. Es wäre somit möglich, dass für zwei Aktionen a^T und $a^{T'}$ auf Ebene L_i die Serialisierungsreihenfolge $a^T < a^{T'}$ gilt, diese jedoch vom Scheduler auf Ebene L_{i-1} ohne Wissen der Ebene L_i umgekehrt wird. Der hieraus entstehende Schedule wäre nicht mehr korrekt.

Eine Möglichkeit, eine bestimmte Serialisierungsreihenfolge auf Ebene L_{i-1} durch den Scheduler der Ebene L_i zu erzwingen, ist das Sequentialisieren der Aktionen a^T und $a^{T'}$ auf Ebene L_i: Die L_{i-1}–Transaktion zur Ausführung der Aktion $a^{T'}$ wird erst gestartet, wenn die L_{i-1}–Transaktion zur Ausführung der Aktion a^T abgeschlossen ist. Wird auf Ebene L_i ein solcher sequentialisierender Scheduler eingesetzt, bedeutet dies, dass die Serialisierungsreihenfolge von L_i–Aktionen auf der unterliegenden Schicht L_{i-1} vom L_{i-1}–Scheduler beliebig festgelegt werden kann: Kommutieren zwei L_i–Aktionen auf Ebene L_i, ist die Serialisierungsreihenfolge auf der Ebene L_{i-1} belanglos; kommutieren sie nicht, sorgt bereits der L_i–Scheduler, dass die L_{i-1}–Aktionen der L_i–Aktion $a^{T'}$ erst nach Abschluss (=Commit) der L_{i-1}–Aktionen von a^T ausgeführt werden und somit die Ausführung der L_i–Aktionen im Rahmen sequentiell ausgeführter L_{i-1}–Transaktionen erfolgt.

Im Rahmen dieser Arbeit wird für die Verwaltung von Dokumenten ein dreischichtiges Datenbanksystem mit einem sequentialisierenden Scheduler auf Ebene L_2 eingesetzt. Wird bei der Ausführung einer L_2-Aktion a^T (*InsertDocument, RetrieveDocument* oder *DeleteDocument*) einer Transaktion T festgestellt, dass diese mit einer L_2–Aktion $a^{T'}$ einer anderen Transaktion T' in Konflikt steht, wird die Ausführung der Aktion $a^{T'}$ bis zum Ende der Transaktion T verzögert. Diese Verzögerung bewirkt, dass folgende Konflikte zwischen Operationen der Ebene L_1 in der Praxis nicht mehr auftreten können:

- *InsDoc–RetDoc* und *RetDoc–InsDoc*
- *InsDoc–DelDoc* und *DelDoc–InsDoc*
- *DelDoc–RetDoc* und *RetDoc–DelDoc*

Für diese Aktionspaare stellt bereits der L_2–Scheduler sicher, dass diese nur unter bestimmten Umständen zur parallelen Ausführung auf Ebene L_1 zugelassen werden:

- *InsDoc–RetDoc* und *RetDoc–InsDoc*: Diese L_1–Operationen werden im Rahmen der L_2–Operationen *InsertDocument* und *RetrieveDocument* aufgerufen. Eine parallele Ausführung dieser L_2–Operationen wird auf Ebene L_2 nur dann zugelassen, wenn sich das einzufügende Dokument nicht auf die Anfrage qualifiziert.
- *InsDoc–DelDoc* und *DelDoc–InsDoc*: Diese L_1–Operationen werden im Rahmen der L_2–Operationen *InsertDocument* und *DeleteDocument* ausgeführt. Eine parallele Ausführung dieser L_2–Operationen wird auf Ebene L_2 nur dann zugelassen, wenn sich das einzufügende Dokument nicht auf das Löschprädikat qualifiziert.

- *DelDoc–RetDoc* und *RetDoc–DelDoc*: Diese L_1–Operationen werden im Rahmen der L_2–Operationen *DeleteDocument* und *RetrieveDocument* aufgerufen. Eine parallele Ausführung dieser L_2–Operationen wird auf Ebene L_2 nur dann zugelassen, wenn sich Such- und Löschprädikat nicht überlappen.

Dank der Verzögerung von L_2–Operationen durch den L_2–Scheduler sind zudem folgende Konflikte auf Ebene L_1 semantisch gesehen nicht notwendig, da es sich aus Sicht der Ebene L_2 um Pseudokonflikte auf den Indexstrukturen handelt:

- *InsDes–RetDes* und *RetDes–InsDes*
- *InsDes–DelDes* und *DelDes–InsDes*
- *DelDes–RetDes* und *RetDes–DelDes*

Dass es sich tatsächlich um Pseudokonflikte handelt, kann konstruktiv am Beispiel *InsDes–RetDes* und *RetDes–InsDes* gezeigt werden: Diese L_1–Operationen werden nur im Rahmen der L_2–Operationen *InsertDocument* und *RetrieveDocument* aufgerufen, wobei wiederum gilt, dass der L_2–Scheduler eine parallele Ausführung dieser L_2–Operationen nur zulässt, wenn sich das einzufügende Dokument nicht auf die Anfrage qualifiziert. Wird beispielsweise ein Dokument mit Dokumenten-ID *Dok-ID* mit Deskriptoren a, b und c eingefügt, während gleichzeitig nach Dokumenten mit den Deskriptoren b und e gesucht wird, gilt: Die Ausführung der Operationen *InsDes–RetDes* und *RetDes–InsDes* ist nicht kommutativ, denn

$$InsDes(\ldots, b, \ldots); RetDes(\ldots, b, \ldots) \neq RetDes(\ldots, b, \ldots); InsDes(\ldots, b, \ldots)$$

Im ersten Fall ermittelt *RetDes* die Dokumenten-ID des neu eingefügten Dokuments, im zweiten nicht. Wird anschliessend jedoch $RetDes(\ldots, e, \ldots)$ ausgeführt und die Menge der Trefferdokumente ermittelt, befindet sich das Dokument *Dok-ID* nicht unter diesen, da es den Deskriptor e nicht enthält. Die Ausführungsreihenfolge der Operationen *InsDes–RetDes* und *RetDes–InsDes* ist daher semantisch gesehen irrelevant und kann ohne Beeinflussung der Korrektheit beliebig festgelegt werden.

Aufgrund dieser Überlegungen ist es möglich, die ursprüngliche Konfliktmatrix con^1 der Ebene L_1 durch

$con^1 =$

	InsDoc	RetDoc	DelDoc	InsDes	RetDes	DelDes
InsDoc	+	n/a	n/a	+	+	+
RetDoc	n/a	+	n/a	+	+	+
DelDoc	n/a	n/a	+	+	+	+
InsDes	+	+	+	P	P	+
RetDes	+	+	+	P	+	P
DelDes	+	+	+	+	P	P

zu ersetzen. Hierbei bedeutet *n/a*, dass ein solcher Konflikt aufgrund des L_2–Schedulings nicht auftreten kann und *P*, dass es sich um einen Pseudokonflikt handelt, der aufgrund der Semantik der L_2–Operationen keinen Einfluss auf das Ergebnis einer L_2–Aktion hat (siehe *Kompatibilität* in [WS92]). Auf Ebene L_1 sind daher alle Operationen miteinander kompatibel und können ohne Beschränkung parallel zueinander ausgeführt werden. Bei der Verarbeitung von Textoperationen ist somit kein Scheduling von Operationen der Ebene L_1 notwendig, da die

Ausführungsreihenfolge der L_1–Operationen irrelevant ist. Wenn kein L_1–Scheduler eingesetzt wird, können jedoch beim Aufruf der Operation *RetDes* teilweise inkorrekte Dokumenten-IDs ermittelt werden: Wird parallel zu *RetDes* die Operation *InsDes* ausgeführt, liest *RetDes* unter Umständen Daten nicht abgeschlossener Benutzertransaktionen (dirty reads). Wird eine Anfrage alleine über Postinglisten ausgewertet, d.h. die Postinglisten aller beteiligten Deskriptoren gelesen und die Menge der sich qualifizierenden Dokumente ermittelt bevor auf diese mittels *RetDoc* zugegriffen wird, sind diese lokal auftretenden Inkonsistenzen für das Resultat einer Anfrage belanglos. Wird jedoch ein optimierter Algorithmus zur Auswertung einer Anfrage eingesetzt, wie er beispielsweise in [KS95] vorgestellt wird, muss die Schnittstelle von *RetDoc* erweitert werden: In [KS95] wird die Suche nach Dokumenten, welche n konjunktiv verknüpfte Deskriptoren enthalten wie folgt ermittelt: Zuerst werden die Postinglisten von m ($m < n$) Deskriptoren gelesen und deren Schnittmenge ermittelt. Anschliessend werden jene Dokumente, welche diese m Deskriptoren enthalten, gelesen und das Auftreten der restlichen $n - m$ Deskriptoren durch eine Suche in den Dokumenten selbst überprüft. Die Schnittstelle von *RetDoc* muss daher um einen Parameter $\{des_k\}$ erweitert werden, in welchem die noch zu überprüfenden $n - m$ Deskriptoren übergeben werden. Die Funktionalität von *RetDoc* muss zudem wie folgt erweitert werden: Nach dem Lesen der Postinglisten der ersten m Deskriptoren wird die Menge jener Dokumenten-IDs berechnet, welche die ersten m Deskriptoren enthalten. Diese enthält unter Umständen Dokumenten-IDs von Dokumenten, welche gerade eingefügt werden. In *RetDes* muss daher damit gerechnet werden, dass beim Lesen von Dokumenten mittels Dokumenten-IDs diese eventuell nicht mehr vorhanden sind, falls die einfügende Transaktion zurückgesetzt wurde. Derselbe Effekt kann ebenfalls auftreten, wenn parallel zu einer Suche Dokumente gelöscht werden: Nach dem Ermitteln der Schnittmenge der ersten m Deskriptoren wird wiederum auf die sich potentiell qualifizierenden Dokumente zugegriffen, welche zu diesem Zeitpunkt allenfalls bereits vollständig aus der Datenbank entfernt worden sind und daher nicht mehr gelesen werden können.

Zusammenfassend bedeutet dies, dass durch parallel ausgeführte L_1–Operationen auf den Indexstrukturen inkonsistente Zwischenzustände anderer L_2–Transaktionen sichtbar werden. L_1–Operationen können daher nicht davon ausgehen, dass sie auf konsistenten Daten arbeiten. Vielmehr müssen bei der Realisierung der L_1–Operationen die möglichen inkonsistenten Zwischenzustände ermittelt und behandelt werden, um die Konsistenz sicherzustellen.

Abbruch von Mehrschichtentransaktionen Bei den bisher betrachteten Mehrschichtentransaktionen wurde davon ausgegangen, dass diese immer erfolgreich abgeschlossen werden. Ist dies nicht der Fall, kann es zu einer Verletzung der *Atomarität* kommen. Dass und wie diese auch in einem mehrschichtigen System sichergestellt werden kann, wird anhand des Beispiels in Abbildung 3.5 illustriert. Dabei wird angenomme, dass die Transaktion T während des Änderns der Postingliste P_b unvorhergesehen terminiert. Dann ist zu diesem Zeitpunkt sowohl das Dokument d_1 persistent in der Datenbank gespeichert, als auch die Postingliste P_a bereits geändert. Diese durch T durchgeführten Änderungen können in einem dreistufigen Verfahren rückgängig gemacht werden (Abbildung 3.6):

- In einem ersten Schritt wird die auf Ebene L_0 noch nicht persistente Änderung der Postingliste P_b durch die Transaktionsverwaltung der Ebene L_0 rückgängig gemacht. Hierzu wird ein zustandsorientierter Recoveryalgorithmus (before images) verwendet.

- In einem zweiten Schritt werden die auf Ebene L_0 bereits persistenten Änderungen rückgängig gemacht. Dies ist nicht mit Hilfe einer zustandsorientierten Recovery möglich, da parallel ablaufende Transaktionen (beispielsweise eine zweite Einfügetransaktion) unter Umständen dieselben L_1-Objekte (hier: Postinglisten) manipuliert haben. Es muss daher eine *operationsorientierte* Recoverystrategie verwendet werden: Für jede auf Ebene L_1 durchgeführte und auf Ebene L_0 transaktionsmässig bereits abgeschlossene Aktion a wird eine sogenannte *inverse Aktion* a^{-1}, auch *Kompensationsaktion* genannt, durchgeführt, welche die von a verursachten Änderungen rückgängig macht. Individuelle Kompensationsaktionen müssen dabei für alle Aktionen, welche Änderungen am Datenbestand vornehmen, bereitgestellt werden. Im konkreten Fall entfernt die $InsDes^{-1}$ den Verweis des Deskriptors a auf das Dokument d_1 aus der textuellen Indexstruktur und $InsDoc^{-1}$ löscht das Dokument aus der Datenbank.

  Um diesen zweiten Schritt überhaupt durchführen zu können, muss Information über die auf jeder Ebene ausgeführten Aktionen vorhanden sein. Diese Information wird in ebenenspezifischen Logdateien, sogenannten L_i*–Logs*, gespeichert.

- In einem dritten Schritt müssen die abgeschlossenen L_2–Aktionen nicht abgeschlossener Benutzertransaktionen durch Ausführung von L_2–Kompensationsaktionen ausgeglichen werden.

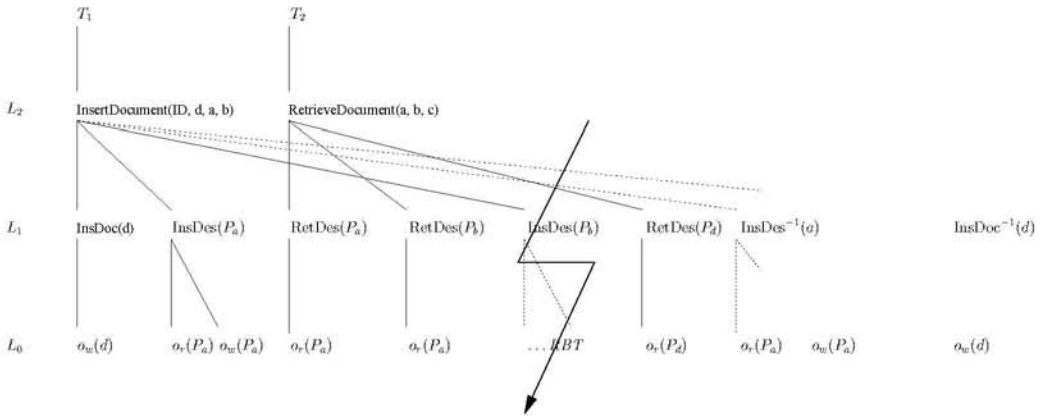


Abbildung 3.6: Mehrschichtentransaktionen: Transaktionsabbruch

Abschliessend sei vermerkt, dass eine Kompensationsaktion a^{-1} auf Ebene L_i nur ausgeführt werden kann, wenn die hierfür notwendigen L_i–Sperren erfolgreich angefordert werden können. Diese Sperren können im Fall einer Recovery immer gewährt werden, wenn die Aktion a vor ihrer Ausführung nicht nur ihre eigenen Sperren, sondern auch bereits jene ihrer Kompensationsaktion a^{-1} anfordert. Zwei Aktionen $a = o_1(d_j)$ und $a' = o_2(d_j)$ mit $con^i_{o_1,o_2} = +$ kommutieren daher nur, wenn auch $con^i_{o_1^{-1},o_2^{-1}} = +$, $con^i_{o_1,o_2^{-1}} = +$ und $con^i_{o_1^{-1},o_2} = +$ gilt.

Da die Sperren für die inverse Aktion a^{-1} im Rahmen der Aktion a vorangefordert werden, wird nachfolgend auf eine explizite Ausführung der Kompensationsoperationen in den Konfliktmatrizen verzichtet.

Crash–Recovery Wird die Abarbeitung von Mehrschichtentransaktionen zu einem nicht näher bestimmbaren Zeitpunkt unfreiwillig unterbrochen, können auf allen Ebenen Aktionen existieren, die zum Zeitpunkt des Systemversagens nur teilweise ausgeführt waren. Um einen konsistenten Systemzustand wiederherzustellen, muss deshalb in einem ersten Schritt eine traditionelle L_0–Crash–Recovery durchgeführt werden. Am Ende dieses Schritts sind alle Änderungen erfolgreich abgeschlossener L_1–Aktionen in die Datenbank eingebracht. Änderungen aller nicht erfolgreich abgeschlossener L_1–Aktionen sind rückgängig gemacht worden. In einem zweiten Schritt müssen Änderungen abgeschlossener L_1–Aktionen nicht abgeschlossener L_2–Aktionen durch Ausführung von L_1–Kompensationsaktionen rückgängig gemacht werden. In einem dreischichtigen Datenbanksystem ist die Recovery an diesem Punkt beendet und die Daten befinden sich in einem konsistenten Zustand. In einem System mit mehr als drei Ebenen hingegen müssen anschliessend ebenenweise die zum Crash–Zeitpunkt nicht abgeschlossenen Transaktionen kompensiert werden.

Intra–Transaktionsparallelität Durch die frühzeitige Freigabe von Sperren auf Datenbankobjekten kann der Grad der Intertransaktionsparallelität, also der tatsächlich zu einem Zeitpunkt parallel zueinander ablaufenden Benutzertransaktionen, gesteigert werden. Mit Hilfe von Mehrschichtentransaktionen kann aber auch *Intratransaktionsparallelität* erzielt werden: Voneinander unabhängige Operationen derselben Transaktion können parallel ausgeführt werden. Dadurch werden die Antwortzeiten reduziert. So können beispielsweise die Referenzen auf ein Dokument in allen betroffenen Postinglisten gleichzeitig geändert werden.

Es darf an dieser Stelle jedoch nicht verschwiegen werden, dass die Transaktionsverarbeitung in einem Mehrschichtensystem auch mit zusätzlichen Kosten verbunden ist. Insbesondere ist auf jeder Ebene L_i sowohl ein Lock-, als auch ein Logmanager zu realisieren und zu jeder Operation eine Kompensationsoperation zur Verfügung zu stellen. Ausserdem ist zu erwarten, dass das andauernde Starten und Beenden von sehr kurzen Transaktionen auf Ebene L_0 mit grossem Zeitaufwand verbunden ist, wenn für diese L_0–Transaktionen Dauerhaftigkeit sichergestellt werden muss. In vielen Anwendungen wäre daher der aus Mehrschichtentransaktionen stammende Nutzen geringer als die Kosten. Dass dies bei einem Systems zur Dokumentenverwaltung nicht der Fall ist, werden die Leistungsuntersuchungen dieser Arbeit zeigen.

3.6 Paralleles Arbeiten in Datenbanksystemen heute: Möglichkeiten und Limitierungen

Heutige Datenbanksysteme erlauben ein paralleles Verarbeiten gleichzeitig ablaufender Transaktionen, die jeweils aus einer Folge von Datenbankaktionen bestehen. Ebenso gestatten heutige Datenbanksysteme, beispielsweise Oracle, voneinander unabhängige Teile einer *einzelnen* SQL-Anweisung intern mit Hilfe mehrerer Prozesse parallel abzuarbeiten, die einzelnen Ergebnisse zusammenzufügen und an den Benutzer zurückzugeben. Die beiden folgenden Beispiele zeigen dies.

- ```
  SELECT COUNT(RelA.*), COUNT(RelB.*)
  FROM   RelA, RelB
  ```

  entspricht der parallelen Ausführung der Anweisungen
  ```

```
SELECT COUNT(*) FROM RelA
und
SELECT COUNT(*) FROM RelB
```

- ```
  SELECT A FROM RelA WHERE A=12
  INTERSECT
  SELECT A FROM RelB WHERE A=12
  INTERSECT
  SELECT A FROM RelC WHERE A=12
  ```

  kann durch paralleles Auswerten der einzelnen `SELECT` Anweisungen und anschliessende Durchführung der `INTERSECT` Anweisungen ermittelt werden.

Diese Parallelität wird als *Operatorparallelität* bezeichnet.

Intratransaktionsparallelität, das heisst das parallele Ausführen einzelner Anweisungen oder der Anweisungssequenzen einer einzelnen Transaktion, wird von derzeit kommerziell erhältlichen Datenbanksystemen nicht unterstützt. Beim Entwurf dieser Systeme ging man davon aus, dass alle Operationen einer Transaktion *sequentiell* abgearbeitet werden. Im Fall einer parallelen Abarbeitung von Anweisungssequenzen sind die vorhandenen internen Datenstrukturen der Sperrverwaltung für eine korrekte Verarbeitung nicht mehr ausreichend, wie anhand des folgenden Beispiels deutlich gemacht wird: Im Rahmen einer Transaktion T werden zwei Textdokumente in die Datenbank eingefügt, wobei jeweils eine — zu Beginn leere — Postingliste P_{des} nachgeführt wird. In einem Datenbanksystem, das in Bezug auf Postinglisten nur deren Lesen und Schreiben unterstützt, wird diese Transaktion bei Verwendung von R/W–Sperren und einem Strikten Zwei-Phasen Sperrprotokoll gemäss Abbildung 3.7 abgearbeitet.

Operation	P_{des}	Bemerkung
Transaktionsbeginn	{}	
⋮	{}	Dokument d_1 einfügen → *Dok-ID*$_1$
ReadLock(P_{des})	{}	
Read(P_{des})	{}	
⋮	{}	
WriteLock(P_{des})	{}	
Write(P_{des})	{Dok-ID$_1$}	
⋮		Dokument d_2 einfügen → *Dok-ID*$_2$
ReadLock(P_{des})	{*Dok-ID*$_1$}	(bereits gewährt)
Read(P_{des})	{*Dok-ID*$_1$}	
⋮	{*Dok-ID*$_1$}	Dokument *Dok-ID*$_2$ einfügen
WriteLock(P_{des})	{*Dok-ID*$_1$}	(bereits gewährt)
Write(P_{des})	{*Dok-ID*$_1$, *Dok-ID*$_2$}	
⋮	{*Dok-ID*$_1$, *Dok-ID*$_2$}	
Transaktionsende	{*Dok-ID*$_1$, *Dok-ID*$_2$}	

Abbildung 3.7: Sequentielles Einfügen von Dokumenten

Da das Einfügen zweier Dokumente voneinander unabhängig ist, liegt es nahe, diese Operationen parallel zueinander, aber im Rahmen derselben Transaktion auszuführen. Eine vom
  ```

Scheduler des Datenbanksystems gewählte Ausführungsreihenfolge der einzelnen Operationen ist in Abbildung 3.8 dargestellt. Während die Postingliste P_{des} im Fall der sequentiellen

Einfüger 1	Einfüger 2	P_{des}	Bemerkung
⋮		{}	
ReadLock(P_{des}		{}	
Read(P_{des})		{}	
	ReadLock(P_{des})	{}	(•) bereits gewährt
⋮	Read(P_{des})	{}	(∘)
	WriteLock(P_{des})	{}	
	Write(P_{des})	{*Dok-ID*$_2$}	
WriteLock(P_{des})		{*Dok-ID*$_2$}	(⋆) bereits gewährt
Write(P_{des})		{*Dok-ID*$_1$}	

Abbildung 3.8: Paralleles Einfügen von Dokumenten

Ausführung am Ende korrekterweise *Dok-ID*$_1$ und *Dok-ID*$_2$ enthält, befindet sich bei paralleler Ausführung nur *Dok-ID*$_1$ in der Postingliste, da die Sperrverwaltung des Datenbanksystems bei der Gewährung der Sperren nicht erkannte, dass die Lese- und Schreibsperren auf P_{des} von zwei verschiedenen Threads (Einfüger 1 und Einfüger 2) ein und derselben Transaktion angefordert wurden. Aus Sicht der Datenbank handelt es sich bei der Sperranforderung (•) um eine wiederholte Anforderung der Sperre (∘). Da diese Sperre der Transaktion schon gewährt wurde, wird sie nochmals erteilt. Nachfolgend wird mit (⋆) eine Änderung des Sperrmodus auf P_{des} beantragt. Diese Änderung, d.h. die Aufwertung der Lesesperre zu einer Schreibsperre, ist unter der Annahme zulässig, dass keine andere Transaktion eine Lesesperre auf P_{des} hält. Aus Sicht des Systems handelt es sich bei der Transaktion, welche die Lesesperre (•) angefordert hat, wiederum um dieselbe Transaktion. Die Änderung des Sperrmodus wird daher fälschlicherweise gewährt.

Hätte es sich bei *Einfüger 1* und *Einfüger 2* um zwei unabhängige Transaktionen gehandelt, wäre die Änderung des Sperrmodus in einem Datenbanksystem mit einem Strikten Zwei–Phasen–Sperrprotokoll bis nach Abschluss von *Einfüger 2* nicht gewährt worden. Nach Ausführung beider Transaktionen hätte aber dasselbe korrekte Endergebnis wie bei einer seriellen Ausführung der beiden Transaktionen vorgelegen.

Ein intratransaktionsparalleles Arbeiten ist daher mit derzeitigen Datenbanksystemen aufgrund ihrer Architektur nicht möglich, obwohl es in vielen Applikationen zu einer Leistungssteigerung beitragen könnte. Dieses Problem wurde vielerorts bereits erkannt. Beispiele hierfür sind der SQL3–Standard [SQL93], welcher zum ersten Mal Schnittstellen für eine asynchrone Ausführung *einzelner* SQL–Anweisungen vorsieht, sowie der XA+ Standard [X/O94]. Der letztgenannt Standard — der eine einheitliche Transaktions–Schnittstelle für verteilte Datenbanken beschreibt — definiert unter anderem sogenannte *Transaction Branches*. Diese erlauben es, nach dem Starten einer Benutzertransaktion *Anweisungssequenzen* parallel zueinander auszuführen, wobei jede dieser Anweisungssequenzen separate Sperren auf die von dieser Sequenz zu verarbeitenden Objekte anfordert. Zu vom Benutzer definierten Zeitpunkten werden diese Transaction Branches synchronisiert und alle von diesen angeforderten Sperren an die Benutzertransaktion vererbt, d.h., die Sperren werden gegenüber anderen, parallel laufenden Benutzertransaktionen gehalten, gegenüber nachfolgenden Transaction Branches derselben Benutzertransaktion jedoch freigegeben. In der Terminologie der Mehrschichtentransaktionen

wird mit Hilfe des XA+–Interfaces ein zweischichtiges Datenbanksystem mit *geschlossen geschachtelten* Transaktionen realisiert. Geschlossen geschachtelt bedeutet, dass die Sperren auf den manipulierten Datenbankobjekten bis zum Ende der Benutzertransaktion gehalten werden. Ein Erhöhung des Intertransaktionsparallelitätsgrades durch kürzer gehaltene Sperren ist somit nicht möglich. Die Operationen der Ebene L_0 sind die Anweisungen des Datenbanksystems, beispielsweise SQL. Die Sperren sind die Sperren des Datenbanksystems. Die Operationen der Ebene L_1 sind Anweisungssequenzen, wobei diese immer mit anderen Anweisungssequenzen kommutieren. Semantisch reiche Sperren und Kompensationsoperationen existieren auf Ebene L_1 nicht: Da die Sperren auf Ebene L_0 immer bis zum Ende einer Benutzertransaktion gehalten werden, ist dies nicht notwendig, da sowohl Isolation als auch Atomarität bereits durch die Transaktionsverwaltung der Ebene L_0 garantiert werden.

In der Praxis wird dieser Teil des XA+–Standards heute noch nicht eingesetzt. In Oracle beispielsweise werden Transaction Branches synchronisiert, d.h. der Start eines Transaction Branches wird verzögert, sofern ein anderer Transaction Branch derselben Transaktion noch aktiv ist, sich also noch nicht in der Synchronisationsphase befindet. Dies genügt dem XA+–Standard, ein intratransaktionsparalleles Arbeiten wird dadurch jedoch nicht unterstützt.

4 TPM/ONT: Ein System zur Ausführung offen geschachtelter Transaktionen

In Kapitel 3 wurden Schwächen heutiger Datenbanksysteme aufgezeigt, deren Ursachen im Einsatz konventioneller Sperrprotokolle für die Transaktionsverwaltung liegen und die zu empfindlichen Leistungseinbussen führen können: Zum einen werden Sperren auf Datenbankobjekte bis zum Ende einer Transaktion gehalten und dadurch parallel arbeitende Benutzer in ihrer Arbeit behindert (geringe *Inter*transaktionsparallelität), zum anderen können voneinander unabhängige Teile einer Benutzertransaktion nicht gleichzeitig ausgeführt werden (keine *Intra*transaktionsparallelität).

Eine Möglichkeit, sowohl die Inter- als auch die Intratransaktionsparallelität zu erhöhen, liegt im Einsatz eines Datenbanksystems mit einer mehrschichtigen Transaktionsverwaltung, welche die Ausführung offen geschachtelter Transaktionen unterstützt. Ein solches System kann grundsätzlich auf zwei Arten erstellt werden: Entweder wird ein bereits bestehendes Datenbanksystem um eine Komponente zur Ausführung von Mehrschichtentransaktionen erweitert, respektive ein neues Datenbanksystem realisiert, oder es wird eine Mehrschichtentransaktionsverwaltung "on top" eines bestehenden Datenbanksystems verwirklicht. Die erstgenannte Variante ist allein vom finanziellen Standpunkt aus betrachtet eher unrealistisch.

Eine aus Sicht des Software–Engineering vielversprechende Variante ist die Realisierung eines Datenbanksystems mit einer Mehrschichtentransaktionsverwaltung, welches auf bereits bestehender, kommerziell erhältlicher Software aufbaut. Die Mehrschichtentransaktionsverwaltung wird in diesem Fall als eine Applikation der unterliegenden Systeme verwirklicht, wobei idealerweise der Code der eingesetzten Systeme nicht geändert wird. Dadurch ist es möglich, zukünftige Versionen der unterliegenden Basissysteme ohne Änderungen zu übernehmen und von Leistungsverbesserungen direkt zu profitieren. Im Rahmen der gegenständlichen Arbeit wird diese Variante näher untersucht und ein zweischichtiges System vorgestellt, das auf den kommerziell erhältlichen Datenbanksystemen Oracle und Sybase, sowie dem Transaktionsmonitor Tuxedo aufbaut.

Im ersten Teil dieses Kapitels werden Realisierungsansätze zweischichtiger Datenbanksysteme basierend auf traditionellen Datenbanksystemen vorgestellt. Die Nachteile dieser Ansätze werden aufgezeigt und schrittweise eliminiert. Diese Ansätze führen zu *TPM/ONT*, dem im Rahmen dieser Arbeit entworfenen und implementierten Prototyp zur Ausführung offen geschachtelter Mehrschichtentransaktionen.

4.1 Mehrschichtentransaktionen: Realisierungsvarianten

Für die Diskussion möglicher Realisierungsvarianten einer mehrschichtigen Transaktionsverwaltung wird eine Benutzertransaktion T betrachtet, die in Form eines eingebetteten SQL–Programms realisiert ist:

```
EXEC SQL BEGIN TRANSACTION
   o1:
      ⋮
      EXEC SQL ...
      EXEC SQL ...
      ⋮
   o2:
      ⋮
      EXEC SQL ...
      EXEC SQL ...
      ⋮
   on:
      ⋮
      EXEC SQL ...
      ⋮
EXEC SQL COMMIT TRANSACTION
```

Diese Transaktion besteht aus voneinander unabhängigen Benutzeroperationen o_i — beispielsweise dem Einfügen oder dem Suchen von Dokumenten — welche sich jeweils aus einer Folge von Datenbankaktionen (SQL–Anweisungen) zusammensetzen. Der Aufbau derselben hat Ähnlichkeit mit der Transaktion eines zweischichtigen Datenbanksystems: Die Benutzeroperationen o_i entsprechen semantisch reichen Operationen (L_1–Operationen), die einzelnen SQL–Anweisungen semantisch primitiven Operationen (L_0–Operationen).

In einem traditionellen Datenbanksystem wird die Benutzertransaktion im Rahmen eines Klientenprozesses sequentiell abgearbeitet. Im Laufe dieser Transaktion angeforderte Sperren auf Datenbankobjekte werden bis zum Ende der Transaktion gehalten. Ist die Benutzertransaktion datenintensiv und langlaufend, kann es zu den bereits besprochenen Behinderungen und damit Verzögerungen aufgrund der Sperranforderungen parallel ablaufender Transaktionen kommen.

Diese Verzögerungen können nur verkürzt werden, indem jede Benutzeroperation o_i von einer eigenen Transaktionsklammer umschlossen wird. Wie in Abschnitt 3.5 erläutert, werden infolge der damit verbundenen vorzeitigen Sperrfreigabe die Atomarität und die Isolation der Benutzertransaktionen nicht mehr garantiert. Zur Sicherstellung der Isolation muss das Benutzerprogramm um Sperranforderungen auf semantisch reiche Objekte (L_1–Sperren im Sinne der Mehrschichtentransaktionen) ergänzt werden. Um die Atomarität der Benutzertransaktionen zu garantieren, muss zudem die Ausführung der einzelnen Benutzeroperationen o_i protokolliert werden (L_1–Log im Sinne der Mehrschichtentransaktionen). Somit stellt sich das Benutzerprogramm wie folgt dar:

```
Benutzertransaktion beginnen
```

```
o_1:
⋮
o_i:
   EXEC SQL BEGIN TRANSACTION
   Sperren auf L_1-Objekte anfordern (⋆)
   IF (Sperren gewährt) THEN
     ⋮
     EXEC SQL
     ⋮
     Durchführung von o_i protokollieren
     Ausführung von o_i beenden
     EXEC SQL COMMIT TRANSACTION
   ELSE (* Verklemmung *)
     EXEC SQL ROLLBACK TRANSACTION
     Bisherige Benutzeroperationen o_j (1 ≤ j < i) kompensieren
     Sperren der Benutzertransaktion auf L_1-Objekte freigeben (⋆)
     Benutzertransaktion beenden
   ENDIF
o_{i+1}:
⋮
o_n:
   EXEC SQL BEGIN TRANSACTION
   ⋮
   EXEC SQL COMMIT TRANSACTION

Benutzertransaktion beenden:
   Sperren der Benutzertransaktion auf L_1-Objekte freigeben (⋆)
```

Die mit ($\star$) gekennzeichneten Operationen zur Verwaltung von Sperren auf semantisch reichen Objekten greifen auf Daten zu, welche von allen Benutzertransaktionen benötigt werden. Werden alle Benutzertransaktionen auf demselben Rechner abgearbeitet, können die für diese Operationen notwendigen Datenstrukturen in einem gemeinsam genutzten Hauptspeichersegment (shared memory) abgelegt werden. Sind hingegen die Klientenprozesse auf verschiedene Rechner verteilt, muss ein separater Prozess *(Mehrschichtenkoordinator (MSK))* zur Verarbeitung der Sperroperationen zur Verfügung gestellt werden. Hierdurch ensteht ein System, wie es in Abbildung 4.1 gezeigt wird.

Diese Realisierungsvariante mit einem Mehrschichtenkoordinator ist unvorteilhaft:

- Kann eine Benutzertransaktion nicht vollständig durchgeführt werden, muss die gerade in Abarbeitung befindliche Benutzeroperation o_i auf SQL-Ebene abgebrochen werden. Zu diesem Zeitpunkt hat die Benutzertransaktion bereits $i-1$ Operationen o_j $(1 \leq j < i)$ im Rahmen jeweils eigenständiger Datenbanktransaktionen bereits durchgeführt. Die Änderungen dieser Datenbanktransaktionen sind persistent in die Datenbank eingebracht worden und können nur noch durch Kompensation rückgängig gemacht werden. Da der Mehrschichtenkoordinator nur L_1-Lockinformationen, nicht aber den Code für die Ausführung der Kompensationsoperationen der bereits durchgeführten Benutzeroperationen o_j $(1 \leq j < i)$ besitzt, muss die Kompensation durch das Benutzerprogramm selbst erfolgen. Es liegt somit in der alleinigen Verantwortung des Benutzerprogramms, bereits persistente Änderungen einer abgebrochenen Benutzertransaktion rückgängig zu

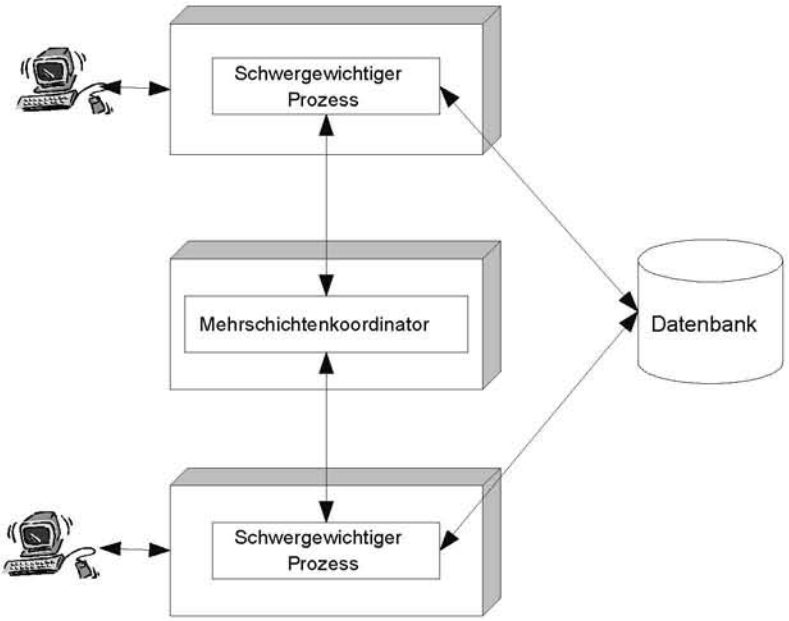


Abbildung 4.1: Datenbanksystem mit Mehrschichtenkoordinator

machen und einen konsistenten Zustand der Datenbank wiederherzustellen.

Da die Rücksetzung der Benutzertransaktion durch das Benutzerprogramm selbst erfolgt, kann es im Fall eines Absturzes desselben vorkommen, dass sich die Datenbank dauerhaft in einem inkonsistenten Zustand befindet: Der erste Teil der Benutzertransaktion (Benutzeroperationen o_j $(1 \leq j < i)$) ist auf Datenbankebene bereits persistent, der zweite konnte nicht mehr ausgeführt werden. Dieser Missstand kann in der vorliegenden Realisierungsvariante nicht behoben werden, da das Benutzerprogramm nicht mehr existiert und somit keine Kompensationsoperationen durchführen kann. Zudem kann es zu einer teilweisen oder kompletten Verklemmung des Datenbanksystems kommen, wenn die von der abgestürzten Benutzertransaktion angeforderten L_1–Sperren nicht mehr freigegeben werden.

- Voneinander unabhängige Operationen o_i einer Benutzertransaktion können nicht parallel abgearbeitet werden: L_1-Operationen bestehen aus einer Reihe von SQL-Operationen. Eine asynchrone und damit parallele Verarbeitung von SQL-Sequenzen ist aus den in Abschnitt 3.8 genannten Gründen nicht möglich.

Ein intratransaktionsparalleles Arbeiten ist möglich, sofern für jede parallel auszuführende Benutzeroperation o_i ein leichtgewichtiger Prozess (lightweight process) gestartet wird, welcher jeweils eine Verbindung zum Datenbanksystem aufbaut, o_i im Rahmen einer eigenständigen Datenbanktransaktion ausführt und anschliessend die Verbindung wieder abbaut (Abbildung 4.2)[1]. Dieser Ansatz scheitert derzeit jedoch in der Praxis: Kommerzielle Datenbanksysteme erlauben die parallele Ausführung voneinander unabhängiger Transaktionen durch

[1] Aus Leistungsüberlegungen heraus könnte alternativ zu Beginn der Benutzertransaktion eine gewisse Anzahl leichtgewichtiger Prozesse vorgeneriert werden, welche anschliessend für die Abarbeitung der Benutzeroperationen o_i verwendet werden.

leichtgewichtige Prozesse nicht, da diese Systeme davon ausgehen, dass Benutzerprogramme immer strikt sequentiell ausgeführt werden. In den bestehenden Implementierungen heutiger System sind daher oftmals globale Variablen vorhanden, welche eine parallele Ausführung von Transaktionen verhindern. Dies sind Restriktionen, die im Rahmen zukünftiger Versionen beseitigt werden.

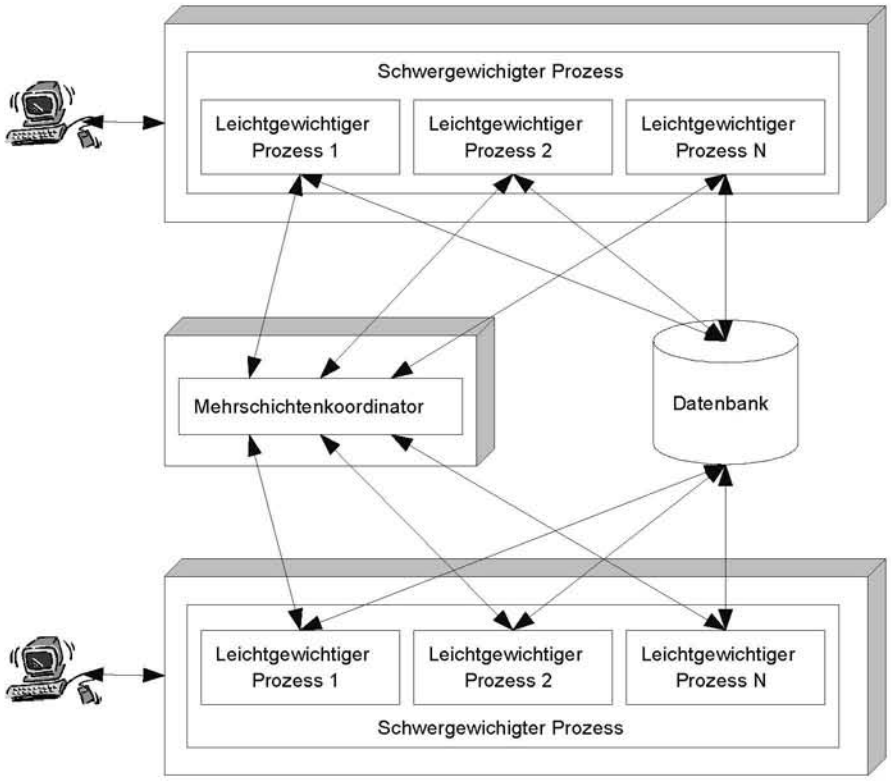


Abbildung 4.2: Intratransaktionsparalleles Arbeiten mittels leichtgewichtiger Prozesse

Ein intratransaktionsparalleles Arbeiten ist mit heutigen Datenbanksystemen nur mittels einer Architektur möglich, wie sie in Abbildung 4.3 gezeigt wird. Jeder Benutzertransaktion wird eine Reihe schwergewichtiger Prozesse (engl. *heavy weighted processes*) zur Verfügung gestellt. Jeder dieser Prozesse enthält den Code der Benutzeroperationen o_i und ist mit der Datenbank verbunden. Will eine Benutzertransaktion eine Operation o_i durchführen, veranlasst sie deren Abarbeitung durch einen schwergewichtigen Prozess mittels Interprozess–Kommunikationsmechanismen, beispielsweise gemeinsamen Hauptspeicher, Queues oder Pipes. Jede Benutzeroperation wird in dieser Umgebung im Rahmen einer separaten Transaktion des Datenbanksystems durchgeführt.

Der entscheidende Nachteil dieser Lösung liegt im hohen Ressourcenverbrauch: Jeder Benutzerprozess ist indirekt über seine schwergewichtigen Prozesse nicht nur einmal, sondern mehrmals mit der Datenbank verbunden. Im Fall eines symmetrisch aufgebauten Datenbanksystems starten damit n datenbankseitige Serverprozesse, die jeweils Hauptspeicherressourcen benötigen und deren Startzeiten — werden die Server nicht vorgeneriert — im Sekundenbereich liegen. Bei einem asymmetrischen Datenbanksystem erstellen die schwergewichtigen Prozesse insge-

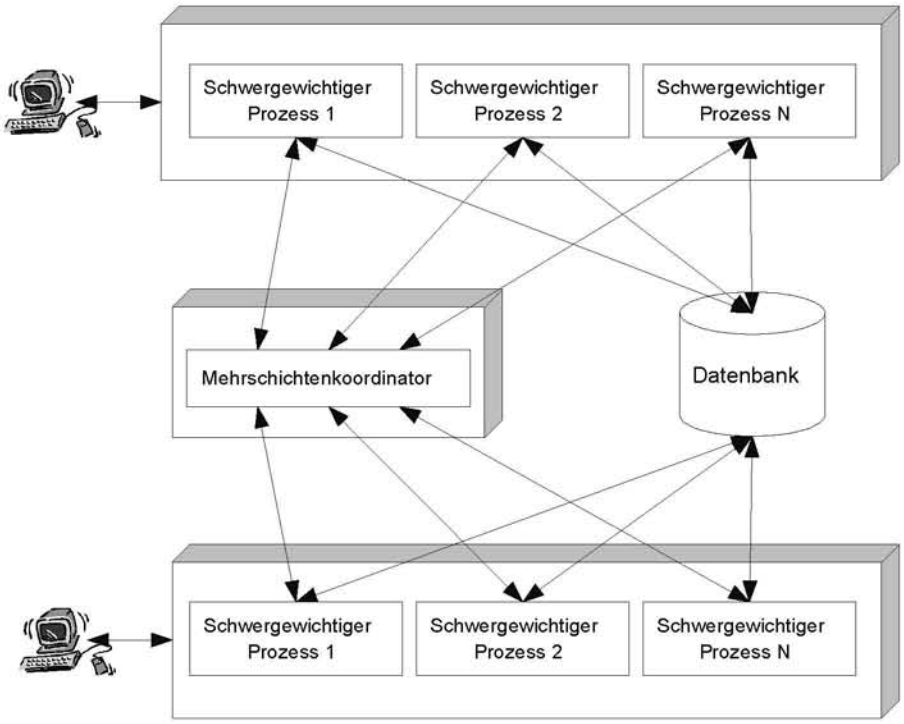


Abbildung 4.3: Intratransktionsparalleles Arbeiten mittels separater schwergewichtiger Prozesse

samt n Verbindungen mit den datenbankseitigen Serverprozessen. Der Benutzer kann in den meisten Systemen keinen Einfluss darauf nehmen, mit welchem Datenbankserver seine Klientenprozesse verbunden werden. So kann es im Extremfall vorkommen, dass alle Verbindungen der Klientenprozesse mit einem *einzigen* Server aufgebaut werden; in diesem Fall vermindert sich der Grad der effektiven Intratransaktionsparallelität der Benutzertransaktion, da zu jedem Zeitpunkt immer nur eine Benutzeroperation durch den Server bearbeitet werden kann. Im anderen Extrem werden alle Verbindungen der Klientenprozesse mit verschiedenen Servern aufgebaut; in diesem Fall wird der Grad der Intratransaktionsparallelität dieses Benutzerprogramms maximal.

Ungeachtet dessen, ob es sich um ein Datenbanksystem mit symmetrischer oder asymmetrischer Architektur handelt, wird das Betriebssystem durch die grosse Anzahl schwergewichtiger Prozesse (mehrere pro Benutzerprozess) zeitlich, aber auch speichertechnisch stark belastet. Zudem wird wegen des Auf- und Abbauens von Datenbankverbindungen die Leistung des Datenbanksystems vermindert. Ideal wäre der Einsatz eines Systems, bei welchem sich die Anwender gemeinsam genutzter schwerer Prozesse bedienen. Diese bei Inbetriebnahme des Datenbanksystems gestarteten Prozesse eröffnen jeweils eine Verbindung zum Datenbanksystem und halten diese kontinuierlich aufrecht. Ein solches System wird in Abbildung 4.4 gezeigt: Mehrere Benutzerprogramme teilen sich eine fixe Anzahl schwergewichtiger Prozesse, in welchen die L_1-Operationen der einzelnen Benutzertransaktionen ausgeführt werden. Die Initiierung von Benutzeroperationen erfolgt wiederum über IPC–Mechanismen (Pipes, Queues, Shared Memory). Das asynchrone Absetzen von IPC-Aufrufen durch eine Transaktion und somit das

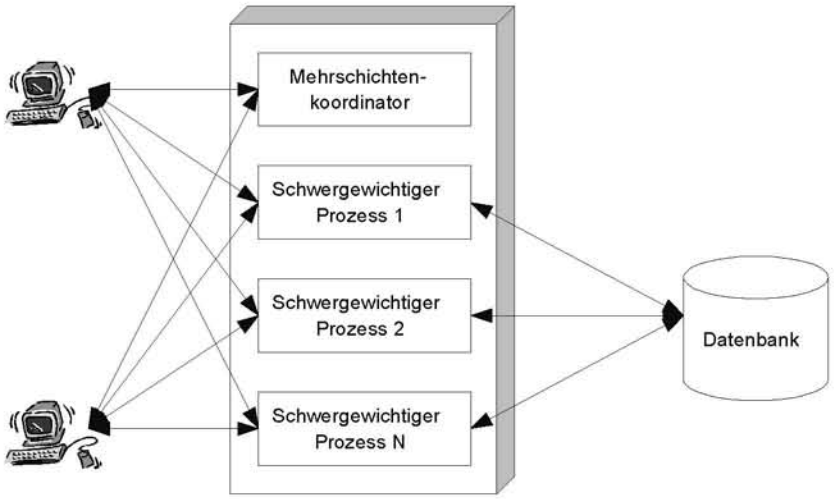


Abbildung 4.4: Intratransaktionsparalleles Arbeiten mittels gemeinsam genutzter schwergewichtiger Prozesse

parallele Ausführen von Benutzeroperationen ist möglich.

Ein solches System gestattet, Mehrschichtentransaktionen "on top" eines bestehenden Datenbanksystems zu realisieren, wobei dieses sogar eine einfache Lastkontrolle erlaubt: Sind nur wenige Benutzer im System, können asynchron abgesetzte IPC-Aufrufe einer Benutzertransaktion parallel zueinander ausgeführt werden. Befinden sich viele Benutzer im System, können IPC–Aufrufe immer noch asynchron abgesetzt werden. Auf Seite der schwergewichtigen Prozesse werden sie jedoch sequentialisiert, falls mehr Operationen o_i parallel ausgeführt werden sollen als schwergewichtige Prozesse zur Verfügung stehen.

Weiterhin ungelöst bleibt in allen bisher besprochenen Realisierungsvarianten die Rücksetzung von Transaktionen, deren Benutzerprozesse infolge eines Systemversagens nicht mehr verfügbar sind und daher die Ausführung der Kompensationsoperationen nicht mehr initiieren können.

Die in Abbildung 4.4 abgebildete Architektur zur Ausführung von Mehrschichtentransaktionen zeigt grosse Ähnlichkeit mit sogenannten *Transaktionsmonitoren.* Es werden daher nachfolgend Transaktionsmonitore vorgestellt. Dabei wird gezeigt, wie diese für die Realisierung einer Mehrschichtentransaktionsverwaltung eingesetzt werden können. Insbesondere wird untersucht, ob sie — im Gegensatz zu den bisherigen Varianten — ein korrektes Arbeiten im Fall des Absturzes von Benutzerprozessen unterstützen.

4.2 Transaktionsmonitore

Heutige Informationssysteme sind im Allgemeinen nach dem Client–Server Paradigma aufgebaut, d.h. der Benutzer ist über eine logische Verbindung direkt mit dem Datenbanksystem verbunden. Diese Architektur hat eine Reihe von Nachteilen:

- *Kommunikation:* Jede Interaktion mit der Datenbank (beispielsweise eine SQL–Operation) bedeutet ein Senden und Empfangen vieler Datenpakete über das Netzwerk. Die Lauf-

zeiten können sehr hoch sein.

- *Codereplikation:* Die Applikationslogik sowie der Code für die Durchführung einer Benutzertransaktion ist in jedem Klienten vorhanden.
- *Verteilung:* Werden im Rahmen einer Benutzertransaktion mehrere Datenbanksysteme angesprochen, muss das Benutzerprogramm wissen, auf welchen Datenbanksystemen sich welche Informationen befinden. Zudem müssen Transaktionen über Datenbankgrenzen hinweg durch den Benutzer koordiniert werden.

In vielen Anwendungen kommen daher *Transaktionsmonitore*, kurz *TP–Monitore* oder *TPM* genannt, zum Einsatz. TP-Monitore gehören zur Klasse der sogenannten *Middleware*–Produkte.

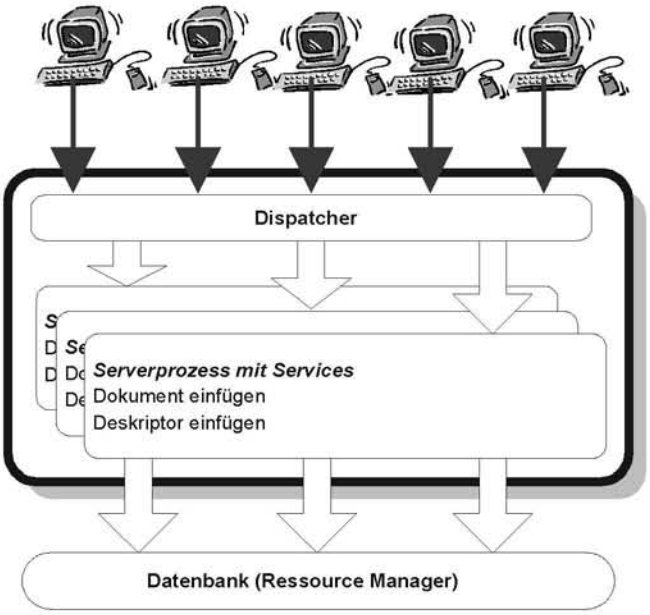


Abbildung 4.5: Architektur eines Transaktionsmonitors

Das sind Softwarekomponenten, welche als Bindeglied zwischen Applikationsprogrammen einerseits und Datenbanksystemen andererseits dienen und den Applikationen zusätzliche Funktionalität bieten. Transaktionsmonitore sind heute vielfach nach dem *X/Open Distributed Transaction Processing* Modell [X/O96] gemäss Abbildung 4.5 aufgebaut. In dieser Umgebung realisieren *Applikationsprogramme* eine gewisse Funktionalität, indem sie eine Reihe von Operationen auf gemeinsamen Ressourcen — beispielsweise Datenbanken — transaktionsorientiert ausführen. Soll eine Operation ausgeführt werden, sendet das Applikationsprogramm einen entsprechenden Auftrag an den Dispatcher, der diesen an einen Serverprozess (der diese Operation anbietet) zur Ausführung weiterleitet. Wird eine Operation von mehreren Serverprozessen angeboten, ist es dem Dispatcher in der Regel freigestellt, diese von einem beliebigen ausführen zu lassen. Ein Beispiel folgt auf der nächsten Seite.

Transaktionen eines Applikationsprogramms, kurz *AP–Transaktionen*, bestehen im Allgemeinen nicht nur aus einer einzelnen Operation, sondern aus einer Menge von Operationen, welche unter Umständen mehrere Serverprozesse involvieren. Der Dispatcher muss daher darüber

buchführen, auf welchen Serverprozessen ein Applikationsprogramm im Rahmen einer Transaktion Operationen durchführt. Diese Information wird benötigt, um am Ende einer AP-Transaktion die Änderungen auf den beteiligten Ressource–Managern über ein 2PC–Verfahren persistent oder rückgängig zu machen.

Im vorangegangenen Abschnitt wurde wiederholt von "Operationen", welche durch einen "Ressource–Manager" auf einer "Ressource" ausgeführt werden, gesprochen, ohne jedoch diese Begriffe genauer definiert zu haben. Der X/Open DTP–Standard versteht unter einem Ressource–Manager eine Softwarekomponente, welche den Zugriff auf eine von mehreren Benutzern genutzte Ressource erlaubt. Eine Ressource ist in der Regel eine Datenbank, kann aber auch ein Drucker sein. Die Kommunikation zwischen Dispatcher, Serverprozessen und Ressource-Managern erfolgt dabei über genormte Schnittstellen, dem *XA*- sowie dem *TX-Interface* [X/O91, X/O94, X/O95], welche von den meisten heute erhältlichen Datenbanksystemen angeboten werden. Die nachfolgenden Betrachtungen gehen von Datenbanken als einzigem Ressourcetyp aus und besprechen den Aufbau des Ressource–Manager anhand des TP–Monitors Tuxedo, der in dieser Arbeit eine Schlüsselrolle spielt.

Für Tuxedo ist ein Ressource–Manager — im Sinne von X/Open DTP — ein Programm, das als *Server* bezeichnet wird. Ein Server baut zum Startzeitpunkt eine Verbindung mit einer Datenbank auf und hält diese kontinuierlich aufrecht. Jeder Server stellt eine Menge sogenannter *Services* — das sind Operationen im Sinne von X/Open DTP — zur Verfügung, deren transaktionsorientierte Ausführung von Applikationsprogrammen über den Dispatcher veranlasst werden kann. Jeder Service interagiert mit einem transaktionsorientierten System in einer für dieses System verständlichen Sprache, beispielsweise Embedded SQL.

In einem System zur Verwaltung von Textdaten würden Services für das Einfügen und Löschen von Dokumenten, für das Nachführen textueller Indexstrukturen, sowie für die Suche nach Dokumenten mit bestimmten Merkmalen zur Verfügung gestellt werden. Ein Tuxedo–Service kann daher als semantisch reiche Operation eines zweischichtigen Datenbanksystems gewertet werden. Die semantisch primitiven Operationen sind in einer solchen Umgebung die einzelnen Interaktionen mit dem Datenbanksystem, beispielsweise Embedded SQL–Anweisungen.

In einer Produktionsumgebung besteht ein TP–Monitor aus einer grösseren Zahl von Servern mit teils identischen, teils verschiedenen Services. Welche Services von welchen Servern angeboten werden, ist dem Dispatcher bekannt. Aufträge eines Applikationsprogramms, *Service Calls* genannt, können daher über den Dispatcher an einen beliebigen Server weitergeleitet werden, welcher den benötigten Service zur Verfügung stellt. Dieser Ansatz hat den Vorteil, dass ein Benutzerprozess nicht spezifizieren muss, auf welchem Server ein Service auszuführen ist. Dem TP–Monitor ist es daher möglich, eine Lastbalancierung durchzuführen und den momentan am wenigsten belasteten Server mit der Ausführung eines Services zu beauftragen.

Transaktionsmonitore, so auch Tuxedo, erlauben es, Operationen einer einzelnen AP-Transaktion entweder synchron oder asynchron an den TP-Monitor zur Ausführung zu übergeben. Werden dem Monitor zudem von einem Applikationsprogramm mehrere Aufträge gleichzeitig zur asynchronen Abarbeitung übergeben, können diese parallel ausgeführt werden. Dies bedeutet, dass zwei oder mehrere Server zur selben Zeit Services für ein und dieselbe AP–Transaktion durchführen. Technisch muss an dieser Stelle zwischen zwei Fällen unterschieden werden:

- *Teilweise Verbindung mit derselben Datenbank:* Mindestens zwei Server sind mit derselben Datenbank verbunden. Ein paralleles Ausführen der Services ist im Allgemeinen

nicht möglich, da es dadurch zu den bereits diskutierten Inkonsistenzen kommen kann. Der TP–Monitor serialisiert deshalb diese Service Calls.

- *Verbindung mit verschiedenen Datenbanken:* Die beteiligten Server sind mit verschiedenen Datenbanken verbunden. Ein paralleles Abarbeiten ist ohne Einschränkung möglich, da auf den beteiligten Datenbanksystemen jeweils eine eigenständige Transaktion (RM–Transaktion) gestartet wird.

TP-Monitore können in ihrer bestehenden Form somit nur zur Parallelisierung von Service Calls, deren ausführende Server mit verschiedenen Datenbanken verbunden sind, eingesetzt werden. Dies schränkt ihre Anwendung im Bereich der Parallelisierung stark ein.

Mit Hilfe von TP-Monitoren können jedoch die zu Beginn aufgezählten Nachteile konventioneller Client–Server–Architekturen behoben werden:

- *Kommunikation:* Es kommunizieren ausschliesslich die Server mit der Datenbank. Da sich die Server im Regelfall auf demselben Rechner wie die Datenbank befinden, kann die Kommunikation effizient über den gemeinsamen Hauptspeicher erfolgen. Die Netzwerkbelastung für die Kommunikation zwischen Applikationsprogramm und Dispatcher bzw. Dispatcher und Server ist geringer als im Fall von beispielsweise Embedded SQL: Das Netzwerk wird nur für die Initiierung eines Service Calls und dem Transfer der Resultatdaten benötigt. Ganz im Gegensatz zu Embedded SQL, wo für jede SQL–Anweisung mehrere Netzwerkzugriffe auf die Datenbank benötigt werden.

- *Codereplikation:* Der Applikationscode der einzelnen Operationen (beispielsweise für das Einfügen eines Dokuments) ist nur in den Servern, nicht aber den Applikationsprogrammen, repliziert.

- *Verteilung:* Aus Sicht des Benutzers existiert nur der TP–Monitor als Kommunikationspartner. Dieser leitet Service–Calls des Benutzers autonom und transaktionsorientiert an die entsprechenden Server weiter.

Mit TP-Monitoren nicht zu beheben sind die bereits diskutierten Leistungsengpässe, welche durch die Sperrverwaltung der beteiligten Datenbanksysteme hervorgerufen werden. Ausserdem ist ein intratransaktionsparalleles Arbeiten nur eingeschränkt möglich. Im Folgenden stellen wir eine Erweiterung von TP–Monitoren vor, welche es erlaubt, offen geschachtelte Zweischichtentransaktionen auszuführen und dadurch diese Leistungsengpässe zu reduzieren.

4.3 TPM/ONT: Architektur und Grundprinzip

Im Rahmen dieser Arbeit wurde ein Datenbanksystem mit einer zweischichtigen Transaktionsverwaltung realisiert, welche die Ausführung offen geschachtelter Transaktionen erlaubt. Die Architektur dieses Systems — *TPM/ONT (Transaction Processing Monitor with Support for Open Nested Transactions)* genannt — wird in Abbildung 4.6 gezeigt. Kern des Systems ist der Transaktionsmonitor Tuxedo [Nov95] sowie wahlweise das Datenbanksystem Oracle oder Sybase.

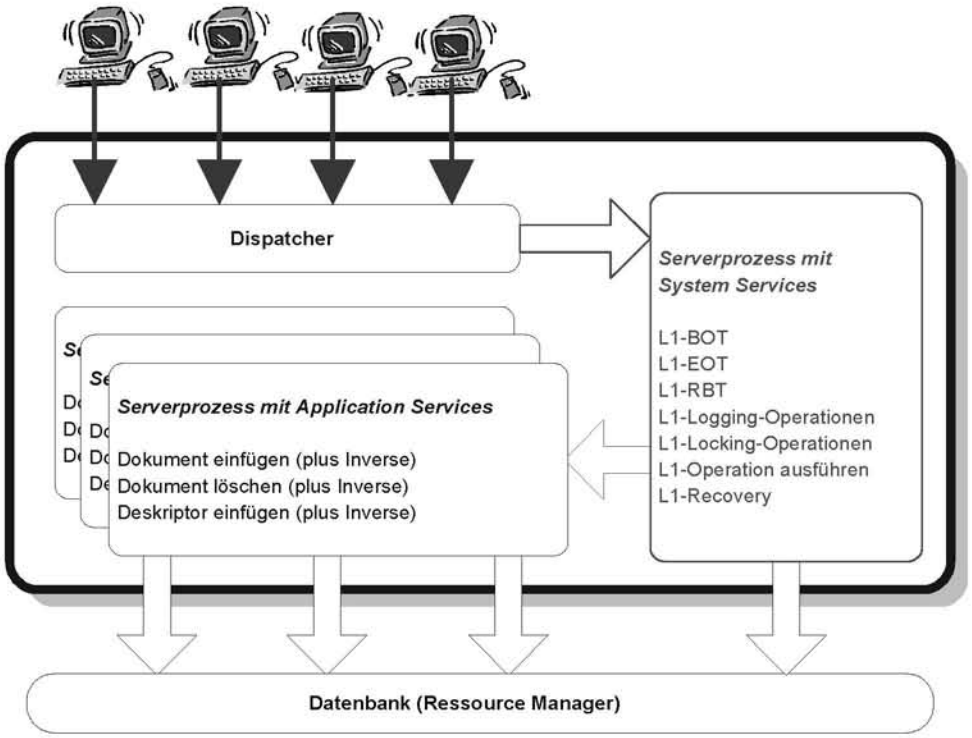


Abbildung 4.6: Realisierung einer Mehrschichtentransaktionsverwaltung mittels TP-Monitoren

Die gezeigte Architektur unterscheidet sich rein äusserlich nicht von der eines herkömmlichen TP-Monitors wie in Abbildung 4.5 dargestellt. Die Unterschiede liegen in der Art und Weise, wie AP–Transaktionen ausgeführt werden. In einer traditionellen TP-Monitor–Anwendung besteht ein Applikationsprogramm aus einer sequentiell abzuarbeitenden Folge von Service Calls, welche im Rahmen einer *einzigen* Transaktion des unterliegenden Datenbanksystems ausgeführt werden. Im Gegensatz hierzu wird in TPM/ONT jeder Service Call einer AP–Transaktion im Rahmen einer eigenständigen Transaktion des unterliegenden Datenbanksystems (RM–Transaktion) ausgeführt, d.h. zu Beginn eines Service Calls wird jeweils eine RM–Transaktion gestartet und am Ende des Service Calls beendet. Dank der Verwendung voneinander unabhängiger RM–Transaktionen für die Ausführung von Services ist es möglich, diese seitens der Applikationsprogramme nicht nur asynchron zu beginnen, sondern diese auch effektiv parallel zueinander auszuführen. Intratransaktionsparalleles Arbeiten ist somit durch die Abbildung von Intratransaktionsparallelität der AP–Ebene auf Intertransaktionsparallelität der RM–Ebene möglich. Mit dem Ende der RM–Transaktion am Ende jedes Service Calls werden zudem alle Sperren dieser RM–Transaktion freigegeben. Dies bedeutet eine Reduktion der Sperrkonflikte auf RM–Ebene und damit einen höheren Parallelitätsgrad.

Um Atomarität und Isolation von AP–Transaktionen sicherzustellen — diese sind durch die Verwendung jeweils eigenständiger RM–Transaktionen für die Ausführung der einzelnen Services nicht mehr garantiert — stellt TPM/ONT zusätzlich zu den applikationsspezifischen Services, *Applikationsservices* genannt, eine Reihe weiterer Services, sogenannte *Systemservices*, zur Verfügung.

Im Folgenden werden zuerst diese von TPM/ONT realisierten Systemservices — welche ohne Änderungen für beliebige Applikationen eingesetzt werden können — im Detail vorgestellt. Anschliessend wird das Vorgehen bei der Integration anwendungsspezifischer Applikationsservices beschrieben. Schliesslich wird Tcl/ONT, eine Erweiterung der Sprache Tcl [Ous94], für die Realisierung von Applikationen vorgestellt.

4.4 TPM/ONT: Systemservices

Ein grundlegendes Prinzip von Datenbanksystemen mit mehrschichtiger Transaktionsverwaltung ist die Unabhängigkeit der einzelnen Schichten: Auf jeder Ebene des Systems existieren Objekte mit eigenständiger Sperr- und Logverwaltung, auf welchen ebenenspezifische Operationen angewendet werden.

TPM/ONT ist ein zweischichtiges Datenbanksystem, dessen untere Ebene L_0 ein kommerzielles Datenbanksystem (Oracle bzw. Sybase) ist. Die Objekte dieser Ebene sind Tupel, welche mittels SQL–Operationen manipuliert werden. Die Sperr- und Logverwaltung dieser Ebene erfolgt durch die Transaktionsverwaltung des verwendeten Datenbanksystems.

Auf der darüberliegenden Ebene L_1 befindet sich der TP–Monitor Tuxedo. Die Operationen dieser Ebene sind Applikationsservices, welche applikationsspezifische, semantisch reiche Objekte manipulieren. Tuxedo stellt jedoch weder Operationen zum Sperren dieser Objekte noch die Verwaltung von Loginformationen zur Verfügung. Tuxedo ist daher um diese Komponenten zu erweitern. Diese werden in Form von Services zur Verfügung gestellt, da die Änderung des Codes des TP-Monitors nicht möglich ist.

Nachfolgend wird der in TPM/ONT realisierte prädikatorientierte L_1–Sperrverwalter vorgestellt, anschliessend wird auf die Dienste für die Durchführung des L_1–Logging eingegangen und das Vorgehen bei der Rücksetzung von AP–Transaktionen einschliesslich der Crash–Recovery erläutert. Ausserdem werden Services besprochen, welche beim Starten bzw. Beenden von TPM/ONT ausgeführt werden müssen.

4.4.1 Sperrverwaltung auf semantisch reicher Ebene

Bei der Besprechung von Mehrschichtentransaktionen wurde bereits erläutert, dass in einem mehrschichtigen Datenbanksystem ein Sperren von Objekten auf jeder Abstraktionsebene des Systems stattfindet. In TPM/ONT kommen auf Ebene L_0 die Sperren des eingesetzten Ressource Managers zum Einsatz (z.B. Seitensperren in Sybase oder Tupelsperren in Oracle). Diese werden vom Ressource Manager bei der Ausführung von Datenbankaktionen (z.B. Embedded SQL–Anweisungen) selbständig angefordert und bis zum Ende der Transaktion gehalten.

In TPM/ONT kommt auf Ebene L_1 ein prädikatbasierter Sperrverwalter zum Einsatz, wie er beispielsweise in [DPS83, EGLT76] beschrieben wird. Prädikate werden über den einzelnen Attributen eines Objekts aufgebaut. In der derzeitigen Realisierung können Prädikate über Attributen folgender Typen definiert werden:

- Ganzzahlen (`INTEGER`)

- Fliesskommazahlen (`FLOAT`)
- Datum (`DATE`)
- Zeichenketten (`TEXT`)

Nachfolgend werden die von TPM/ONT zur Verfügung gestellten Operationen für die Definition von Prädikaten und die Anforderung von Sperren im Rahmen einer Datenbankaktion $A^T_{i_n,n} = o_{i_n}(D_n)$ (d.h. Anwendung der Operation o_{i_n} auf jedes $d \in D_n$ durch eine Transaktion T) vorgestellt. Folgende Annahmen werden hierbei zugrundegelegt:

- Es existiert eine L_1–Konfliktmatrix *con*, die definiert, welche L_1–Operationen miteinander kommutieren und welche nicht.

  Hierbei gilt, dass zwei L_1–Operationen o_i und o_j kommutieren falls $con_{o_{i_k},o_{i_n}} = +$, bzw. nicht kommutieren, falls $con_{o_{i_k},o_{i_n}} = -$.

- Anderen, zur Zeit noch nicht abgeschlossenen L_1–Transaktionen, wurden bereits Sperren auf Objektmengen D_k $(1 \leq k < n)$ für die Ausführung der Operationen o_{i_k} gewährt.

- Eine Sperre auf die durch D_n definierte Objektmenge für die Ausführung der Datenbankaktion $A^T_{i_n,n} = o_{i_n}(D_n)$ durch die Transaktion T kann durch den L_1–Sperrverwalter von TPM/ONT gewährt werden, falls

  $$\nexists k\ (1 \leq k < n) : (con_{o_{i_k},o_{i_n}} = -) \wedge (D_n \cap D_k \neq \{\})$$

  Das heisst: Eine Sperre für die Ausführung der Datenbankaktion $A^T_{i_n,n} = o_{i_n}(D_n)$ wird gewährt, wenn zur Zeit keine andere Transaktion aktiv ist, welche eine Sperre auf ein Objekt $d \in D_n$ für die Ausführung der Operation o_{i_k} hält, welche nicht mit o_{i_n} kommutiert.

Der in TPM/ONT realisierte L_1–Sperrverwalter stellt dem Programmierer von Applikationsservices konkret folgende Operationen zur Verfügung:

- `L1_Lock_Initialize_Manager(con)` initialisiert den Lockmanager. Die für die Ermittlung der Verträglichkeit von L_1–Operationen benötigte Konfliktmatrix `con` muss vom Programmierer der Applikationsservices definiert und dieser Funktion als Parameter übergeben werden.

- `L1_Lock_Create_Interval(Att, Type, Low, High, LowClosed, HighClosed)` → `Pred` definiert ein Prädikat `Pred` für das mit *Att* bezeichnete Attribut vom Typ *Type* für ein Intervall im Bereich `Low` bis `High`.

  Um Sperren über Volltextattributen effizient zu unterstützen, werden diese durch Signaturen approximiert [DPS83]. Die Signatur eines Textes wird dabei durch Hashing der einzelnen Wörter des Textes gewonnen.

  Wird ein Prädikat für ein Attribut mit Datentypen `INTEGER`, `FLOAT` oder `DATE` definiert, kann hierbei für den oberen und unteren Intervallrand separat angegeben werden (*LowClosed* und *HighClosed*), ob das Intervall offen oder geschlossen ist. Für Attribute vom Typ `TEXT` sind die Parameter `High`, `LowClosed` und `HighClosed` ohne Bedeutung.

  Beispiel: Durch den Aufruf der Funktion

L1_Lock_Create_Interval(X, FLOAT, 1.2, 17.2, Open, Closed)→PredX

wird eine Menge von Objekten spezifiziert, die in einem Fliesskommaattribut X einen Wert im Intervall $]1.2, 17.2]$ aufweisen. Durch den Aufruf von

L1_Lock_Create_Interval(Y, TEXT, "Bill Clinton", NULL, NULL, Discard, Discard)→PredY

wird eine Menge von Objekten spezifiziert, welche in einem Textattribut Y die Wörter *Bill* sowie *Clinton* enthalten.

- `L1_Lock_And(Pred`$_1$`, Pred`$_2$`)→Pred` verknüpft zwei Prädikate $Pred_1$ und $Pred_2$ konjunktiv miteinander.

  Beispiel: Durch den Aufruf

  `L1_Lock_And(PredX, PredY)`

  wird die Menge jener Objekte definiert, die in einem Fliesskommaattribut X einen Wert im Intervall $]1.2, 17.2]$ aufweisen und im Textattribut Y die Wörter *Bill* sowie *Clinton* enthalten.

- `L1_Lock_Or(Pred`$_1$`, Pred`$_2$`)→Pred` verknüpft zwei Prädikate $Pred_1$ und $Pred_2$ disjunktiv miteinander.

- `L1_Lock_Acquire(APTAID, Pred, Op)` fordert eine Sperre auf die durch *Pred* definierte Tupelmenge für die Ausführung einer Operation *Op* im Rahmen der AP-Transaktion `APTAID` an.

  Kann die Sperre nicht gewährt werden, wird die Ausführung des anfordernden Prozesses bis zur Gewährung der Sperre unterbrochen. Die Sperranforderung wird zurückgewiesen, falls die Anforderung zu einer Verklemmung (Deadlock) führt.

Am Ende einer AP–Transaktion oder im Fall eines Abbruchs einer einzelnen L_1–Operation müssen die angeforderten Sperren durch das System wieder freigegeben werden. Hierzu stehen folgende Operationen zur Verfügung:

- `L1_Lock_Release(APTAID, Pred)` gibt eine L_1–Sperre, welche für die Ausführung einer L_1–Operation angefordert wurde, frei. Ein solches Freigeben ist notwendig, wenn die zu dieser Operation gehörige RM–Transaktion aus irgendeinem Grund nicht erfolgreich abgeschlossen werden kann.

- `L1_Lock_Release_All(APTAID)` gibt alle Sperren, die im Rahmen einer AP-Transaktion angefordert wurden, frei. Diese Operation wird am Ende der AP–Transaktion ausgeführt.

4.4.2 Logverwaltung auf semantisch reicher Ebene

Die im letzten Abschnitt beschriebene Sperrverwaltung auf der Ebene L_1 von TPM/ONT gewährleistet die Isolation von AP–Transaktionen und somit auch die Konsistenz der Datenbank, sofern alle AP–Transaktionen jeweils erfolgreich abgeschlossen werden können, d.h. nicht zurückgesetzt werden müssen. Bei Systemversagen oder beim Abbruch einer AP–Transaktion durch Benutzer oder TP–Monitor befindet sich das Datenbanksystem unter Umständen in einem inkonsistenten Zustand. In diesem Fall müssen alle Änderungen am Datenbestand, welche durch diese AP–Transaktion verursacht wurden, rückgängig gemacht werden.

Besteht eine AP–Transaktion aus einer oder mehreren noch nicht abgeschlossenen L_1–Operationen, welche jeweils im Rahmen eigenständiger RM–Transaktionen ausgeführt wurden, wird die AP–Transaktion durch den Abbruch aller beteiligten RM–Transaktionen des unterliegenden Datenbanksystems gewährleistet. Die Transaktionsverwaltung auf Ebene L_1 hat lediglich die Aufgabe, den Abbruch der betroffenen RM–Transaktionen zu initiieren.

Sind jedoch eine oder mehrere L_1–Operationen bereits abgeschlossen, können deren Änderungen durch Kompensationsoperationen rückgängig gemacht werden, da die Änderungen der L_1–Operationen auf Ebene L_0 bereits persistent sind. Die Information, welche L_1–Operationen einer AP–Transaktion bereits abgeschlossen sind und daher kompensiert werden müssen, muss der L_1–Transaktionsverwaltung in Form von Logsätzen (sogenannten L_1*–Logsätzen*) zur Verfügung gestellt werden. In TPM/ONT wird die L_1–Loginformation im unterliegenden Datenbanksystem gespeichert. Nimmt eine L_1–Operation o Änderungen am Datenbestand vor, wird ein L_1–Logsatz in Form eines Tupels der Relation

```
L1Log(Log-Sequenz-Nummer     NUMBER PRIMARY KEY,
      Transaktions-ID        NUMBER,
      Kompensationsoperation STRING,
      Parameter              BLOB)
```

in die Datenbank geschrieben. Bei *Log-Sequenz-Nummer* handelt es sich dabei um eine fortlaufende Nummer, die jedem Logsatz in der Reihenfolge seines Eintreffens bei der Logverwaltung zugeteilt wird. Mit *Kompensationsoperation* wird ein Service bezeichnet, welcher ausgeführt werden muss, um die L_1–Operation o zu kompensieren. In *Parameter* sind jene Daten festgehalten, welche von der Kompensationsoperation zur korrekten Ausführung benötigt werden.

Um die Integrität des Systems sicherzustellen, muss der L_1–Logsatz einer L_1–Operation *atomar* mit den Änderungen dieser Operation auf Ebene L_0 in das unterliegende Datenbanksystem geschrieben werden: Wird die L_1–Operation — und damit die durch diese Operation definierte RM–Transaktion — erfolgreich abgeschlossen, ist der L_1–Logsatz in der Datenbank persistent und steht für Fall eines späteren Rücksetzens der kompletten AP–Transaktion für die Kompensation der durchgeführten L_1–Operation zur Verfügung. Wird die RM–Transaktion hingegen abgebrochen, werden alle durchgeführten Änderungen einschliesslich dem Einfügen des L_1–Logsatzes durch das Datenbanksystem rückgängig gemacht. Da die L_0–Transaktion in diesem Fall bereits vollständig rückgängig gemacht wurde, kann der L_1–Logsatz aus der Datenbank gelöscht werden, da zu keinem späteren Zeitpunkt eine Kompensation auf Ebene L_1 mehr notwendig ist.

TPM/ONT stellt dem Programmierer von Applikationsservices eine Operation zur Verfügung:

- `L1_Log_Insert(APTAID, Operation, Data)` schreibt einen L_1–Logsatz.

Den nachfolgend beschriebenen Systemservices *L1_BOT, L1_EOT* und *L1_RBT_ALL* wird zusätzlich eine Reihe weiterer Services zur Verwaltung der Logsätze sowie der Rücksetzung von AP–Transaktionen bereitgestellt:

- `L1_Log_Delete(LSN)` entfernt den L_1–Logsatz mit der Logsequenznummer *LSN* aus der L_1–Logdatei.
- `L1_Log_Delete_All(APTAID)` entfernt alle L_1–Logsätze der AP–Transaktion *APTAID.*
- `L1_Log_Read(LSN)`→`Operation,Daten` liest den L_1–Logsatz *LSN* und gibt den Namen der Kompensations*operation* sowie die für die Ausführung dieser Operation notwendigen *Daten* zurück.
- `L1_Log_Committed(APTAID)`→ {`LSN`} ermittelt die Logsatznummern aller L_1–Logsätze der AP–Transaktion *APTAID* in der umgekehrten Reihenfolge ihres Eintreffens bei der L_1–Logverwaltung.

  Diese Information wird benötigt um eine AP–Transaktion vollständig zurückzusetzen.
- `L1_Log_All_Committed()`→ {`LSN`} ermittelt und retourniert die Logsatznummern der L_1–Logsätze aller momentan nicht abgeschlossenen AP–Transaktionen in der umgekehrten Reihenfolge ihres Eintreffens bei der L_1-Logverwaltung.

  Diese Information wird im Fall eines Systemversagens benötigt, um im Anschluss an den Neustart von TPM/ONT alle zum Zeitpunkt des Absturzes nicht abgeschlossenen AP–Transaktionen zurückzusetzen.

4.4.3 Transaktionskontrolle auf semantisch reicher Ebene

In einem Datenbanksystem mit einem klassischen Transaktionsmonitor als Middleware–Komponente besteht eine Transaktion aus einer Folge von Service–Calls, welche durch das Applikationsprogramm initiiert werden. Beginn und Ende einer TP-Monitor–Transaktion werden — wie im Fall eines Datenbanksystems ohne TP-Monitor — durch spezielle Operatoren (*tx_begin, tx_commit* und *tx_rollback* im Fall des X/Open–Standards [X/O95]) markiert. In der TP-Monitor–Umgebung kommt diesen Befehlen eine besondere Bedeutung zu:

- Beginn einer AP–Transaktion (*tx_begin*): Der TP–Monitor generiert einen internen Transaktions–Identifikator und vermerkt den Beginn der Transaktion in einer Logdatei. Zu diesem Zeitpunkt werden noch keine Transaktionen auf den unterliegenden Datenbanksystemen (die Server des TP–Monitors sind unter Umständen mit verschiedenen Datenbanksystemen verbunden) gestartet, da a priori nicht bekannt ist, welche Services der Benutzer im Rahmen dieser Transaktion ausführen wird.

  Eine RM–Transaktion auf einem unterliegenden Datenbanksystem wird vom TP–Monitor erst initiiert, wenn der Benutzer im Rahmen dieser AP–Transaktion einen Service aufruft, der auf ein Datenbanksystem zurückgreift, auf das diese AP–Transaktion bisher nicht zugegriffen hat.

- Ende einer AP–Transaktion (*tx_commit*): Alle Datenbanksysteme, auf denen im Rahmen der AP–Transaktion RM–Transaktionen gestartet wurden, beenden diese mittels eines Zwei–Phasen–Commit (2PC), wobei die Koordination des 2PC vom TP-Monitor übernommen wird. Auf welchen Datenbanken RM–Transaktionen gestartet wurden, ist dem TP-Monitor zu diesem Zeitpunkt aufgrund der ausgeführten Services bekannt.

- Abbruch einer AP–Transaktion (*tx_rollback*): Alle RM–Transaktionen, die im Rahmen einer AP–Transaktion begonnen wurden, werden auf Datenbankebene abgebrochen.

In einem klassischen Datenbanksystem ist durch die Verwendung des Zwei–Phasen–Commit–Protokolls garantiert, dass sämtliche im Rahmen einer Benutzertransaktion durchgeführten Änderungen am Datenbestand jederzeit rückgängig oder mittels Prepare to Commit und Commit persistent gemacht werden können.

Mit TPM/ONT wird ein eigenständiger Transaktionsmanager auf Ebene L_1 realisiert. Dieser stellt für die L_1–Transaktionskontrolle (Beginn, Ende und Abbruch von AP–Transaktionen) folgende Operationen in Form eigenständiger Services zur Verfügung:

- `L1_BOT()`→ `APTAID` beginnt eine neue AP–Transaktion. Dieser Service generiert einen eindeutigen Identifikator für diese Transaktion und liefert diesen an das Applikationsprogramm zurück. Dieser Identifikator wird anschliessend für die Durchführung von Sperr- und Logoperationen auf der semantisch reichen Ebene verwendet (siehe Abschnitte 4.4.1 und 4.4.2).

  Ein expliziter L_1–Logsatz für den Beginn der AP–Transaktion wird nicht geschrieben, da die Existenz einer AP–Transaktion implizit durch die Log-Einträge der einzelnen L_1–Operationen respektive den L_1–EOT–Eintrag der AP–Transaktion zu erkennen ist.

- `L1_EOT(APTAID)` beendet eine AP–Transaktion: Alle L_1–Sperren, die im Rahmen dieser Transaktion durch die Applikationsservices angefordert wurden, werden durch Aufruf des Services `L1_Lock_Release_All(APTAID)` der L_1–Lockverwaltung freigegeben und der Transaktionsidentifikator *APTAID* wird invalidiert.

  In einem traditionellen Datenbanksystem wird am Ende einer Transaktion ein expliziter EOT-Logsatz geschrieben. Im Fehlerfall kann die Recoverykomponente anhand dieser Logsätze alle abgeschlossenen Transaktionen ermitteln und für diese eine Forward–Recovery durchführen. Im Fall von TPM/ONT ist eine Forward–Recovery auf Ebene L_1 nicht notwendig: Die Persistenz von AP–Transaktionen ist durch die Persistenz der RM–Transaktionen gegeben, welche bereits durch das unterliegende Datenbanksystem garantiert wird. Die L_1–Logsätze werden somit nach der Ausführung der `L1_EOT`–Operation nicht mehr benötigt und können unverzüglich aus dem L_1–Log mittels Aufruf des Service `L1_Log_Delete_All(APTAID)` aus der Datenbank entfernt werden.

  Es ist anzumerken, dass die erwarteten Kosten für das Entfernen aller L_1–Logsätze einer AP–Transaktion nicht höher sind als das Schreiben eines L_1–EOT–Logsatzes, sofern sich die Logrelation komplett im Hauptspeicher befindet. In beiden Fällen werden die Kosten durch den synchronen Schreibvorgang zur Persistenzsicherung der Logdatei dominiert.

- `L1_RBT(APTAID)` setzt eine Transaktion auf Ebene L_1 in einem zweistufigen Verfahren zurück: In einem ersten Schritt werden alle zu diesem Zeitpunkt laufenden L_1–

Operationen dieser Transaktion abgebrochen. Da die L_1–Operationen im Rahmen individueller Services des TP-Monitors abgearbeitet werden, bedeutet der Abbruch eines Services das Rücksetzen der RM–Transaktion des unterliegenden Datenbanksystems mittels `EXEC SQL ROLLBACK`. In einem zweiten Schritt sind alle bereits abgeschlossenen L_1–Operationen mittels L_1–Kompensationsoperationen zu kompensieren. Zu diesem Zeitpunkt sind alle Änderungen am Datenbestand, welche durch die AP–Transaktion durchgeführt wurden, rückgängig gemacht.

Diese Kompensationsoperationen müssen in TPM/ONT durch den Programmierer von Applikationsservices bereitgestellt werden.

Der verwendete TP–Monitor Tuxedo unterstützt den Abbruch laufender Services nicht. Im Fall eines AP–Transaktionsabbruchs muss in TPM/ONT zuerst das Ende aller momentan aktiven Services dieser Transaktion abgewartet werden. Anschliessend werden wiederum alle erfolgreich abgeschlossenen L_1–Operationen kompensiert.

Die Kompensation aller bereits abgeschlossenen L_1–Operationen einer AP–Transaktion *APTAID* wird gemäss Algorithmus 10 durchgeführt: Die Kompensationsoperation o^{-1}

```
CommittedSubtransactions = L1_Log_Committed(APTAID)
for all committed subtransactions t in CommittedSubtransactions do
  EXEC SQL BEGIN TRANSACTION
  LogRecord = L1_Log_Read(t)
  Ausführen der Kompensationsoperation für t
  L1_Log_Delete(t)
  EXEC SQL COMMIT TRANSACTION
end for
```

Algorithmus 10: Transaktionsabbruch: Kompensation aller bereits abgeschlossenen L_1–Operationen einer AP–Transaktion *APTAID*

einer Operation o führt eine operationsorientierte Backward Recovery durch (siehe Diskussion auf Seite 38) und ist somit nicht idempotent. Daher muss der L_1–Logsatz der Operation o im Rahmen der Kompensationstransaktion gelöscht werden, um eine nochmalige Ausführung von o^{-1} im Fall einer wiederholten Recovery zu vermeiden. Alternativ hierzu kann ein neuer L_1–Logsatz geschrieben werden, der die Ausführung von o^{-1} festhält. Dies erschwert jedoch eine erneute Recovery, falls diese nicht vollständig durchgeführt und daher wiederholt werden muss. In TPM/ONT werden L_1–Logsätze gelöscht.

Die einzelnen Kompensationsoperationen könnten auch im Rahmen einer *einzigen* RM–Transaktion ausgeführt werden. Dies führt jedoch zu einem unnötig langen Sperren der beteiligten Datenbankobjekte, was wiederum zu einer Verschlechterung der Antwortzeiten der parallel hierzu ablaufenden Transaktionen führen kann. Es ist daher im Allgemeinen von Vorteil, die einzelnen Kompensationsoperationen im Rahmen eigenständiger RM–Transaktionen abzuarbeiten.

Da jede L_1–Operation in TPM/ONT im Rahmen einer eigenen RM–Transaktion ausgeführt wird, kann in TPM/ONT auf den Einsatz des XA-Interfaces verzichtet werden, da dieses nur benötigt wird, wenn sich eine Transaktion über mehrere Service Calls erstreckt. Tuxedo

erlaubt dies und bedient sich dazu eines speziellen Ressource-Managers namens `TMS`. Dieser nimmt XA-Aufrufe jeweils entgegen und bestätigt sofort mit einer Mitteilung, dass der XA-Service erfolgreich ausgeführt wurde. Wird dieser Ressource-Manager eingesetzt, muss die Transaktionskontrolle wieder explizit durch den Applikationsprogrammierer durchgeführt werden. Für TPM/ONT heisst das konkret, dass bei Einsatz eines relationalen Datenbanksystems die Befehle `EXEC SQL COMMIT`, bzw. `EXEC SQL ROLLBACK` am Ende jeder L_1–Operation explizit ausgeführt werden müssen.

4.4.4 Crash–Recovery

An Datenbanksysteme wird die Anforderung gestellt, auch nach dem Auftreten eines Fehlers, beispielsweise eines Stromausfalls oder eines Sekundärspeicherdefekts, ein Weiterarbeiten ohne Datenverlust zu ermöglichen. Dem muss auch ein zweischichtiges Datenbanksystem wie TPM/ONT gerecht werden. Die bisher vorgestellten Systemservices sind hinreichend, um Zweischichtentransaktionen in einer perfekten Umgebung korrekt auszuführen. Dies schliesst die benutzerinitiierte Rücksetzung von AP–Transaktionen durch den Systemservice `L1_RBT` mit ein.

In den bisherigen Betrachtungen wurde immer davon ausgegangen, dass die einzelnen Serverprozesse von TPM/ONT ununterbrochen zur Verfügung stehen. Der Fall, dass einzelne oder alle Serverprozesse ausfallen und das System in einen temporär instabilen Zustand versetzen, wurde nicht betrachtet. Um auch in solchen Fällen ein korrektes Verhalten des Datenbanksystems sicherzustellen, sind die möglichen Zustände, in denen sich die Server von TPM/ONT befinden und die Übergänge zwischen diesen, zu untersuchen (siehe Abbildung 4.7).

PRESTART Server noch nicht gestartet: Zustand des Servers unmittelbar nach dem Start von Tuxedo.

TERM Server nicht mehr verfügbar: Zustand des Servers nach dem ordnungsgemässen Herunterfahren von TPM/ONT.

DEAD Server ausgefallen: Ausfall durch fehlerhaften Befehl, beispielsweise Dereferenzierung eines "dangling pointers", Betriebssystemfehler oder Ausfall eines Rechners.

IDLE Server passiv: Erwartet Initiierung eines System- oder Applikationsservices durch AP–Transaktion.

BUSY Server aktiv: Führt einen Applikations- oder Systemservice aus.

Bei den Zuständen IDLE und BUSY sowie den Übergängen zwischen diesen handelt es sich um die Normalzustände bzw. -übergänge, die keiner weiteren Erklärung bedürfen. Bei den Übergängen PRESTART → IDLE, IDLE → TERM und DEAD → IDLE handelt es sich um Zustandsänderungen, welche der Ausführung spezieller Operationen bedürfen, wie nachfolgend vorgestellt wird.

Wird dem Benutzer ein neuer Server zur Verfügung gestellt, ist dieser in einen definierten Initialzustand zu versetzen, insbesondere müssen von mehreren Servern gemeinsam genutzte Ressourcen initialisiert werden (Übergang `PRESTART→IDLE`).

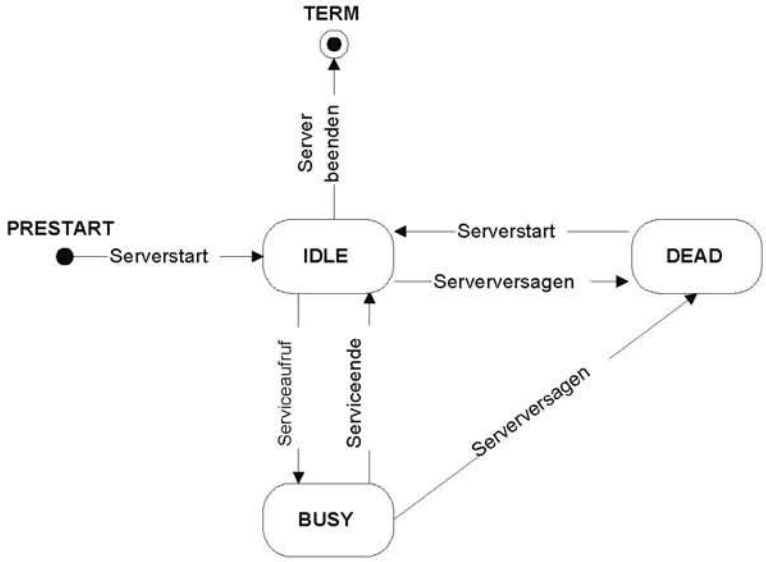


Abbildung 4.7: Zustandsdiagramm der Server eines TP-Monitors

Beim Start von TPM/ONT ist vor Ausführung der ersten AP–Transaktion ein spezieller Systemservice, `L1_RBTALL`, auszuführen. Aufgabe dieses Services ist es, alle AP–Transaktionen, die zum Startzeitpunkt im L_1–Log als noch nicht abgeschlossen aufgeführt sind, zurückzusetzen. In TPM/ONT werden alle L_1–Logsätze einer AP–Transaktion am Ende derselben aus der Logdatei entfernt. Somit ist keine AP–Transaktion, für welche L_1–Logsätze existieren, beendet. Nicht abgeschlossene AP–Transaktionen existieren, falls TPM/ONT vorher aus irgendeinem Grund nicht fehlerfrei beendet oder beendet wurde, ohne das Ende der gerade laufenden AP–Transaktionen abgewartet zu haben.

Da die Benutzerprozesse, die solche AP–Transaktionen gestartet haben, nicht mehr mit TPM/-ONT in Verbindung stehen, können diese beim Neustart von TPM/ONT durch die Benutzerprozesse weder erfolgreich abgeschlossen noch zurückgesetzt werden. Die bereits durchgeführten L_1–Operationen dieser AP–Transaktionen müssen daher durch einen speziellen Systemservice, `L1_RBT_ALL` genannt, kompensiert werden.

```
CommittedSubTransactions = L1_Log_All_Committed()
for all committed subtransactions t in CommittedSubtransactions do
  EXEC SQL BEGIN TRANSACTION
  LogRecord = L1_Log_Read(t)
  Ausführen der Kompensationsoperation für t
  L1_Log_Delete(t)
  EXEC SQL COMMIT TRANSACTION
end for
```

Algorithmus 11: Crash Recovery: Kompensation aller nicht abgeschlossenen AP–Transaktionen mittels `L1_RBT_ALL`

Der in Abbildung 11 gezeigte Service wird in TPM/ONT im Rahmen der Prozedur `tvsrinit` eines eigenständigen Systemservers, *Startupserver* genannt, ausgeführt. Bei `tvsrinit` han-

delt es sich um eine spezielle Prozedur, die von jedem Tuxedo–Server implementiert werden muss. `tvsrinit` wird unmittelbar nach dem Start eines Servers und vor Verwendung eines Services durch Tuxedo ausgeführt. Um AP–Transaktionen überhaupt kompensieren zu können, müssen die Kompensationsoperationen zur Verfügung stehen. Diese sind in Form von Services der Applikationsserver realisiert. Der Startupserver muss daher *nach* dem Hochfahren aller System- und Applikationsserver gestartet werden, damit `L1_RBT_ALL` erfolgreich ausgeführt werden kann.

Soll ein Server — und somit alle von ihm angebotenen Services — nicht mehr zur Verfügung gestellt werden, ist dieser zu terminieren. Der Server geht somit vom Zustand `IDLE` in den Zustand `TERM` über. Teilt sich ein Server Ressourcen, beispielsweise Hauptspeichersegmente, mit anderen Servern, sind diese über die Statusänderung dieses Servers zu informieren. In TPM/ONT geschieht dies im Rahmen der Prozedur `tpsvrdone`, welche von Tuxedo jeweils unmittelbar vor Beendigung eines Servers ausgeführt wird.

Tritt während der Ausführung eines Applikations- oder Systemservices ein gravierender Fehler auf, geht der Server vom Zustand `BUSY` in `DEAD` über. Mit Hilfe der Recovery–Komponente des unterliegenden Datenbanksystems werden alle Änderungen am Datenbestand, welche dieser Service im Rahmen der momentanen RM–Transaktion durchgeführt hat, rückgängig gemacht. Zudem veranlasst Tuxedo die erneute Ausführung dieses Applikationsservices auf einem anderen Server und startet gleichzeitig den terminierten Server wieder neu (Übergang `DEAD` nach `IDLE`, siehe unten).

Tuxedo geht davon aus, dass Server voneinander unabhängig sind und nebst den unterliegenden Datenbanksystemen keine Ressourcen gemeinsam haben. Im Fall eines Serverversagens sind mit der Rücksetzung der RM–Transaktion, welche der terminierte Server gestartet hat, alle gemeinsamen Ressourcen wieder freigegeben. Andere Server sind somit vom Absturz eines Servers nicht betroffen.

In TPM/ONT hingegen werden Ressourcen, insbesondere L_1–Sperren, zwischen Servern geteilt. TPM/ONT stellt daher einen Systemservice `L1_CHECK` zur Verfügung, der diese Ressourcen im Fehlerfall freigibt. Weiterhin prüft `L1_CHECK` in periodischen Abständen, ob bzw. welche Applikationsprogramme, welche AP–Transaktionen gestartet, aber noch nicht beendet haben, verfügbar sind. Ist ein Applikationsprogramm nicht mehr verfügbar, da dessen Prozess beendet wurde, veranlasst `L1_CHECK` den Abbruch dieser AP–Transaktion durch den Aufruf des Systemservices `L1_RBT`.

Wird ein Server, der sich im Zustand `IDLE` befindet, beispielsweise durch ein `kill` Signal terminiert, geht dieser ebenfalls in den Zustand `DEAD` über. Aktionen seitens TPM/ONT sind nicht notwendig, da der Server zum Zeitpunkt der Terminierung keine gemeinsamen Ressourcen mit anderen Servern hält.

Analog zum Übergang `PRESTART`→`IDLE`, bei welchem dem Systemserver besondere Aufgaben zukommen, werden vom letzten herunterzufahrenden Server der Zweischichtentransaktionsverwaltung ebenfalls spezielle Aufgaben ausgeführt; insbesondere werden von diesem gemeinsam genutzte Ressourcen freigegeben. Zu diesem Zeitpunkt unbeendete AP–Transaktionen können nicht mehr zurückgesetzt werden, da die hierfür benötigten Applikations- und Systemservices nicht mehr verfügbar sind. Diese AP–Transaktionen verbleiben im System und werden beim nächsten Start von TPM/ONT — wie oben beschrieben — zurückgesetzt.

4.5 TPM/ONT: Realisierung von Applikationsservices

Die vorgestellten Systemdienste stellen die notwendigen Operationen für die korrekte Abarbeitung zweischichtiger Transaktionen zur Verfügung. Auf diesen aufbauend können semantisch reiche Operationen, d.h. Operationen der Ebene L_1, realisiert werden. Diese werden in TPM/ONT ebenfalls in Form von Services, den sogenannten *Applikationsservices*, verwirklicht.

Die Bereitstellung eines Applikationsservices erfolgt in drei Schritten:

1. Implementierung der semantisch reichen Operation (`Do-Operation`)
2. Implementierung der zugehörigen inversen Operation (`Undo-Operation`)
3. Definition der Konfliktmatrix con_1 der Ebene L_1

Die Do-Operation wie auch die Undo-Operation eines Applikationsservices werden in der Sprache des unterliegenden Datenbanksystems implementiert, beispielsweise als eine Folge von Embedded SQL–Anweisungen im Fall von Oracle und Sybase. Grundsätzlich kann eine L_1–Operation Daten in mehreren Datenbanksystemen manipulieren. Die vorgestellten Algorithmen zur Transaktionskontrolle auf Ebene L_1 müssen in diesem Fall nicht geändert werden, sofern mittels Operationen auf Ebene L_0 sichergestellt wird (beispielsweise ein Zwei–Phasen–Commit), dass am Ende einer L_1–Operation die RM–Transaktionen auf den beteiligten Datenbanksystemen entweder gesamthaft erfolgreich abgeschlossen (commit) oder zurückgesetzt (abort) werden.

Eine Do-Operation setzt sich jeweils aus drei Hauptkomponenten zusammen (Algorithmus 12): Erzeugung eines L_1–Sperrprädikats und Anforderung einer L_1–Sperre, Interaktion mit der unterliegenden Datenbank im Rahmen einer eigenständigen RM–Transaktion und Schreiben eines L_1–Logsatzes. Die Erzeugung des L_1–Sperrprädikats sowie das Schreiben von L_1–Loginformationen mittels der in Abschnitt 4.4 vorgestellten Operationen sind von vitaler Bedeutung: Werden L_1–Sperren nicht korrekt definiert bzw. wird nicht genügend Undo–Information für die Ausführung der Undo-Operation geschrieben, kann die Konsistenz der Datenbank nicht garantiert werden. Den L_1–Sperren bzw. dem L_1–Log, welche immer applikationsspezifisch definiert sind, muss daher besondere Beachtung geschenkt werden.

Die Undo–Operationen haben grundsätzlich denselben Aufbau wie die Do–Operationen, jedoch wird das Schreiben des L_1–Logsatzes durch das Löschen des Logsatzes jener Operation, welche mit dieser Undo–Operation kompensiert wird, ersetzt.

Nach der Realisierung der Do- und Undo-Operationen muss in einem letzten Schritt die Konfliktmatrix con_1 erweitert werden. Stellte TPM/ONT bisher n semantisch reiche Operationen zur Verfügung, hatte die Konfliktmatrix eine Grösse von $n \times n$. Mit der Einführung eines neuen Applikationsservices vergrössert sich die Matrix auf $(n+1) \times (n+1)$. In der neu hinzugekommenen Zeile und Spalte der Konfliktmatrix wird festgehalten, welche der bisher zur Verfügung gestandenen Applikationsservices mit dem neu hinzugekommenen kommutieren und welche nicht.

Stehen Code für die Do- und Undo-Operationen sowie die erweiterte Konfliktmatrix zur Verfügung, kann der neue Applikationsservice dem Benutzer bereitgestellt werden. Hierbei können zwei Ansätze verfolgt werden. Im ersten Fall werden alle Do- und Undo-Operationen in Form

```
EXEC SQL BEGIN TRANSACTION
* L1-Sperrprädikat erzeugen
* L1-Sperre anfordern
if L1-Sperre gewährt then
  {Beginn der Interaktion mit der Datenbank}
  * ⋮
  * EXEC SQL...
  * ⋮
  * EXEC SQL ...
  * ⋮
  {Ende der Interaktion mit der Datenbank}
  if Ausführung erfolgreich then
    L1_Log_Write(...)
    EXEC SQL COMMIT TRANSACTION
  else
    EXEC SQL ROLLBACK TRANSACTION
  end if
else {L1-Sperre nicht gewährt}
  EXEC SQL ROLLBACK TRANSACTION
end if
```

Algorithmus 12: Aufbau von L_1–Operationen (Do-Operationen)

selbständiger Services angeboten: Dem TP-Monitor wird mitgeteilt, dass zwei neue Services (je einer für die Durchführung der Do- sowie der Undo-Operation) zur Verfügung stehen. Diese Services können dann direkt über IPC–Mechanismen aufgerufen werden. Dieses Vorgehen ist jedoch mit grösserem administrativem Aufwand auf Seite des TP–Monitors verbunden.

Beim zweiten Ansatz stellt der TP-Monitor lediglich *einen* Service, nachfolgend `L1_EXEC (APTAID, OpName, ParamsIn, ParamsOut)` genannt, zur Verfügung. Dabei ist *APTAID* ein Transaktions–Identifikator, *OpName* der Name einer Do-Operation; *ParamsIn* sind die Parameter, welche für die Ausführung von *OpName* benötigt werden. Eine AP–Transaktion ruft den Service `L1_EXEC` mit den Parametern *APTAID, OpName* und *ParamsIn* auf, welcher seinerseits den Service *OpName* ausführt. Etwaige Rückgabewerte des ausgeführten Service werden in *ParamsOut* zurückgeliefert. In der momentanen Realisierung von TPM/ONT wird dieser Ansatz verwirklicht. Der Code des realisierten Services `L1_EXEC` findet sich in Algorithmus 13. Der zu implementierende Do/Undo-Code einer L_1–Operation reduziert sich in der zweiten Variante auf die mit (*) bezeichneten Zeilen. Auffallend ist, dass die L_1–Sperren nicht innerhalb von `L1_EXEC`, sondern im Rahmen der einzelnen Do/Undo-Implementierungen angefordert werden. Dafür gibt es zwei Erklärungen: Zum einen sind die L_1–Sperrprädikate applikationsspezifisch und werden ausschliesslich durch die Do-Operation erzeugt, zum anderen werden in vielen Anwendungen Teile der Daten, die für die Berechnung eines Sperrprädikats benötigt werden, auch in der Do-Operation selbst verwendet. Durch die Anforderung der Sperren innerhalb der Do-Operationen wird die zeitlich aufwendige Mehrfachberechnung dieser Daten vermieden.

```
EXEC SQL BEGIN TRANSACTION
op=Code der Operation CodeName in der dynamischen Programmbibliothek
if op gefunden then
  Operation op ausführen
  if Ausführung erfolgreich then
    L1_Write_Log(...)
    EXEC SQL COMMIT TRANSACTION
    return SUCCESS
  else
    EXEC SQL ROLLBACK TRANSACTION
    return FAILURE
  end if
else {Code für op nicht vorhanden}
  return FAILURE
end if
```

Algorithmus 13: Ausführung von L_1–Operationen mittels `L1_EXEC`

4.6 TPM/Tcl: Ausführung von AP–Transaktionen

Die beiden vorangegangenen Abschnitte behandelten die Systemservices, welche ausschliesslich der Transaktionsverwaltung dienen und zeigen auf, wie neue Applikationsservices — das sind semantisch reiche Operationen der Ebene L_1 — realisiert werden. Diese Services stehen den Applikationsprogrammierern nach dem Start von TPM/ONT zur Verfügung. Nachfolgend werden jene Operationen besprochen, welche den Applikationsprogrammierern zum Aufruf dieser Services bereitgestellt werden. Diese bestehen aus Services zur Transaktionskontrolle, Operationen zum Transfer von Daten vom Applikationsprogramm zu den Servern und schliesslich aus Funktionen zum Aufruf einzelner Applikationsservices.

Transaktionskontrolle Eine Benutzertransaktion besteht in einer Umgebung mit einem TP–Monitor aus einer Anweisung *Begin of Transaction* (`tx_begin`), einer Folge von Serviceaufrufen und aus einer Anweisung *End of Transaction* respektive *Rollback Transaction* (`tx_commit` bzw. `tx_rollback`).

Auch in TPM/ONT wird eine AP–Transaktion durch Anweisungen zur Transaktionskontrolle geklammert. Diese Funktionen stehen dem Applikationsprogrammierer in Form dreier Funktionen `cBOT()`→`APTAID`, `cEOT(APTAID)` und `cRBT(APTAID)` zur Verfügung. Diese rufen über die Tuxedo-interne Kommunikationsschnittstelle die Systemservices `L1_BOT`, `L1_EOT` und `L1_RBT` auf. Der beim Aufruf von `cBOT` erzeugte Transaktionsidentifikator wird für alle nachfolgenden Ausführungen von Applikations- und Systemservices benötigt.

Datentransfer Applikations- und Systemservices benötigen für ihre Ausführung meistens Daten des Klientenprogramms (z.B. den AP–Transaktionsidentifikator *APTAID*) oder sie übermitteln Resultate an dieses (z.B. Ergebnisse einer Abfrage, Statusinformation). Transaktionsmonitore stellen Operationen für den Transfer solcher Daten zwischen Applikationsprogrammen und Servern bereit.

Tuxedo stellt für den Datenaustausch grundsätzlich zwei Mechanismen zur Verfügung: Die

Datenübergabe mit Hilfe von *Views* oder von *Fielded Buffers* [Nov95, FML Guide]. Views erlauben es dem Applikationsprogrammierer, einfache, d.h. lineare, Datenstrukturen (records of integer, string, boolean, ...) in einer Metasprache zu definieren und diese mittels eines sogenannten *View Compilers* nach C oder COBOL zu übersetzen. Die durch die Kompilation entstehenden Datentypen werden anschliessend in den Benutzerprogrammen zur Deklaration von Variablen verwendet. Diese Variablen können sodann vom Benutzerprogramm mit Werten belegt und beim Aufruf eines Services als Parameter übergeben werden. Views haben den Nachteil, dass die notwendigen Datentypen nur mittels des View-Compilers erzeugt werden können. Ändert sich die Definition eines Datentyps, müssen alle Services — so auch die Systemservices — welche ihn verwenden, neu übersetzt werden.

Alternativ zu Views können für den Datentransfer sogenannte *Fielded Buffers* eingesetzt werden. Ein Fielded Buffer ist ein Übergabebereich einer benutzerdefinierten Grösse, in dem jedes beliebige Datum in Form von *(Namen, Wert)*–Paaren gespeichert werden kann. Einzige Bedingung für *Wert* ist dabei, dass es sich um eine flache Datenstruktur handelt oder dass Funktionen zur Verfügung gestellt werden, welche diese Datenstruktur linearisieren bzw. wieder expandieren. Eine wichtige Eigenschaft von Views, die im Rahmen dieser Arbeit auch genutzt wird, besteht darin, mehrere (Namen, Wert)–Paare mit gleichem *Namen* in einem Fielded Buffer zu speichern, d.h. mengenwertige Attribute zu definieren. Einem Programmierer stehen nicht nur Funktionen zum Erzeugen und Freigeben von Fielded Buffers einer bestimmten Grösse und Operationen zum Einfügen, Löschen und Ändern von (Namen, Wert)–Paaren zur Verfügung, sondern auch Funktionen zur Ermittlung der Anzahl der (Name, Wert)–Paare für einen bestimmten Namen. Fielded Buffers und die Operationen auf diesen sind den *Transfer Areas* des DASDBS–Kernsystems [Pau88b] sehr ähnlich, wenn man davon absieht, dass eine Nestung (ein Attribut darf weder ein Tupel noch eine Menge von Tupeln enthalten) nicht möglich ist. Fielded Buffers sind dank ihres dynamischen Aufbaus flexibler als Views und werden daher für den Datentransfer in TPM/ONT eingesetzt.

Ausführung von Applikationsservices Nach dem applikationsseitigen Start einer AP–Transaktion mittels `cBOT` kann mit der Ausführung von Applikationsservices begonnen werden. TPM/ONT kann diese synchron oder asynchron zum Applikationsprogramm ausführen.

Soll ein Applikationsservice synchron ausgeführt werden, ruft der Applikationsprogrammierer die Operation `cCALL(APTAID, ServiceName, ParamsIn, ParamsOut)` auf. *APTAID* ist hierbei jener AP–Transaktionsidentifikator, der vorgängig mittels `cBOT` ermittelt wurde; *ServiceName* ist der Name des Applikationsservices, der ausgeführt werden soll und *ParamsIn* ist ein Fielded Buffer, der alle applikationsspezifischen Daten enthält, welche zur Ausführung des genannten Applikationsservices benötigt werden. Da Applikationsservices nicht direkt aufgerufen werden können, sondern nur indirekt über den Systemservice `L1_EXEC` (siehe Abschnitt 4.5), veranlasst `cCALL` die Ausführung von `L1_EXEC` mit Parametern *(APTAID, ServiceName, ParamsIn, ParamsOut)* mittels Tuxedo-spezifischer IPC–Mechanismen (Ausführung der Operation `tpcall`). Die Ausführung des AP–Programms wird anschliessend bis nach Ausführung des Applikationsservices *ServiceName* unterbrochen. Erzeugt der aufgerufene Service Daten, werden diese in *ParamsOut*, einem Fielded Buffer, geschrieben.

Soll ein Service asynchron ausgeführt werden, geschieht dies durch Aufruf der Operation `cACALL(APTAID, ServiceName, ParamsIn)→AID` und `cWAIT(AID, ParamsOut)`. `cACALL` erhält als Parameter wiederum einen AP–Transaktionsidentifikator *APTAID*, den Namen des

auszuführenden Applikationsservices *ServiceName* sowie die Parameter *ParamsIn* dieses Services. `cACALL` startet den Service `L1_EXEC` mit Parametern *APTAID, ServiceName* und *ParamsIn* über einen asynchronen IPC–Aufruf (Ausführung der Operation `tpacall`) und retourniert an den Benutzer unverzüglich einen Identifikator *AID* für diesen Aufruf. Anschliessend wird die Ausführung des Applikationsprogramms fortgesetzt. Das Applikationsprogramm kann danach die Funktion `cWAIT(AID, ParamsOut)` mit dem vorher erhaltenen Identifikator *AID* aufrufen. Diese Funktion wartet das Ende des mit *AID* bezeichneten Service Calls ab und liefert etwaige Rückgabewerte in Form eines Fielded Buffers *ParamsOut* zurück. Ein spezieller Wert für den AID–Identifikator ist der Wert `ANY`: Dieser Wert weist `cWAIT` an, auf die Beendigung *irgendeines* Service Calls zu warten und dessen Ergebnisse zurückzuliefern.

Klientenprogrammierung in Tcl/Tk In der ersten Implementierung von TPM/ONT wurden Applikationsprogramme in der Programmiersprache C geschrieben, d.h. die applikationsseitigen Operationen für die Transaktionskontrolle, den Datentransfer und die Aufrufe von Applikationsservices wurden in Form von C-Funktionen zur Verfügung gestellt.

Im Laufe der Arbeit zeigte sich jedoch, dass die Verwendung von C für die Entwicklung applikationsseitiger Prototypen zu umständlich ist, da bereits die kleinsten Codeänderungen eine komplette Neuübersetzung des Applikationsprogramms nach sich ziehen. Alle im Rahmen dieser Arbeit entstandenen Applikationsprogramme wurden daher mittels Tcl (Tool Command Language) [Ous94], einer Interpretersprache, realisiert. Hierzu war es notwendig, den Tcl–Interpreter um die oben beschriebenen Operationen zur Transaktionskontrolle (`cBOT, cEOT, cRBT`), zur Manipulation von Fielded Buffers und zum Aufruf von Applikationsservices (`cCALL, cACALL, cWAIT`) zu erweitern. Eine solche Erweiterung ist in Tcl dank dynamisch ladbarer Operationen möglich. Zudem kann der erweiterte Interpreter, *Tcl/ONT* genannt, mit Tk (Tool Kit) gekoppelt werden. Tk ist ebenfalls eine Erweiterung von Tcl, die es erlaubt, Tcl–Programme mit einer graphischen Oberfläche auszustatten. Diese Erweiterung wurde für die Visualisierung der zeitlichen Abläufe von TPM/ONT–Programmen verwendet.

5 TPM/ONT–Text: Transaktionsorientierte Dokumentenverwaltung und -suche mittels TPM/ONT

In Kapitel 3 wurde aufgezeigt, dass die Verwaltung von und die Suche nach semistrukturierten Dokumenten in Datenbanksystemen mit einer traditionellen Transaktionsverwaltung zu Leistungsengpässen führen kann.

Durch den Einsatz semantikbasierter Transaktionsmodelle, insbesondere der vorgestellten offen geschachtelten Mehrschichtentransaktionen, werden diese Engpässe dank einer frühzeitigen Sperrfreigabe entschärft. Damit erhöht sich der Grad der *Inter*–Transaktions–Parallelität. Zudem gestattet es die mehrschichtige Transaktionsverwaltung, semantisch unabhängige Operationen einer einzelnen Transaktion parallel abzuarbeiten. *Intra*transaktionsparalleles Arbeiten, beispielsweise das Warten voneinander unabhängiger Textindizes eines Dokuments, ist dadurch möglich.

In diesem Kapitel werden zuerst die in dieser Arbeit verwendeten Operationen zur Verwaltung von und Suche nach semistrukturierten Dokumenten vorgestellt. In der Folge wird ein dreischichtiges Datenbanksystem zur transaktionsorientierten Verarbeitung dieser Operationen besprochen. Schliesslich wird gezeigt, wie dieses dreischichtige System auf TPM/ONT, das ausschliesslich die Ausführung zweischichtiger Transaktionen unterstützt, abgebildet werden kann.

5.1 Dokumentenverwaltung und -suche mittels Transaktions-Monitoren

In der vorliegenden Arbeit wird das Leistungsverhalten transaktionsorientierter Systeme für die Verwaltung von und die Suche nach semistrukturierten Dokumenten untersucht. Es wird speziell das Verhalten von Systemen in einer sich dynamisch ändernden Dokumentenumgebung betrachtet, in welcher neue Dokumente kontinuierlich im laufenden Betrieb sowohl zur Verfügung gestellt, als auch entfernt werden, während mehrere Benutzer gleichzeitig nach Dokumenten suchen. Über die Speicherung der semistrukturierten Dokumente und deren textuelle Indexstrukturen werden folgende Annahmen gemacht:

- Ein Dokument besteht aus einer Anzahl von Nichttextattributen $a_1, a_2, \ldots, a_n$ sowie einer Menge von Textattributen $t_1, t_2, \ldots, t_m$. Intern wird jedes Dokument durch einen eindeutigen, systemgewarteten Identifikator *Dok-ID* identifiziert.

- Jedes Dokument wird intern als Tupel einer Relation
$$D(\textit{Dok-ID}, a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m)$$
modelliert und in einem relationalen Datenbanksystem gespeichert.

- Über jedem Textattribut t_j $(1 \leq j \leq m)$ existiert ein auf Deskriptoren basierender Textindex IL_j zur Unterstützung boolescher Anfragen.

  Die Deskriptoren werden aus den Textattributen durch Extraktion der einzelnen Wörter gewonnen. Es ist möglich, diese Wörter auf ihre Wortstämme zu reduzieren und Stopworte zu eliminieren. Das System soll es erlauben, für jedes Attribut t_j individuelle Extraktions- und Wortstammreduktions–Algorithmen und Stopwortlisten zu spezifizieren. Die hierfür notwendige Information muss in Form von Metadaten zur Verfügung stehen.

  Jeder Textindex IL_j ist als invertierte Liste modelliert und physisch im unterliegenden Datenbanksystem in Form einer Relation $IL_j(\textit{Des-ID}, \textit{Dok-ID})$ gespeichert.

- Um mit einer möglichst kleinen invertierten Liste IL_j eine schnelle Suche zu unterstützen, werden die Deskriptoren *Des* nicht durch die extrahierten Zeichenketten, sondern durch Deskriptor-IDs *Des-ID* in Form von Zahlen repräsentiert. Die eindeutige Zuordnung von Zeichenketten zu Zahlen wird in einer zusätzlichen Relation
$$\text{Mapping}(\underline{\text{Des, Des-ID}})$$
festgehalten. Neuen Deskriptoren werden aufsteigende Deskriptor-IDs zugeordnet.

Auf Ebene L_2 stehen folgende bereits diskutierte Operationen zur Verfügung:

- *InsertDocument*$(a_1, a_2, \ldots, a_n, t_1, t_2, \ldots t_m)$ fügt ein Dokument mit Nichttextattributen a_i und Textattributen t_j in die Datenbank ein.

- *RetrieveDocument(Predicate)* $\rightarrow \{a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m\}$ sucht und retourniert alle Dokumente, die einem Prädikat genügen.

- *DeleteDocument(Predicate)* entfernt alle Dokumente aus der Datenbank, die einem Prädikat genügen.

Bei den Operationen *InsertDocument* und *DeleteDocument* hat das System die Aufgabe, die auf den Attributen definierten Indexstrukturen zu warten. Bei der Suche nach Dokumenten mittels *RetrieveDocument* — der in der Regel häufigsten Operation — werden diese Indexstrukturen zur effizienten Auswertung eingesetzt. Die vorgestellten Operationen zum Einfügen, Suchen und Löschen von Dokumenten können Benutzern in Form von Services eines traditionellen TP–Monitors mit einschichtiger Transaktionsverwaltung bereitgestellt werden. Der Pseudocode der hierfür zu realisierenden Services ist in den Abbildungen 14, 15 und 16 aufgezeigt.

5.2 Ausführung durch ein dreischichtiges Datenbanksystem

Bei den vorgestellten Operationen zum Einfügen, Suchen und Löschen von Dokumenten kommt es in einem einschichtigen TP–Monitorsystem zu sperrbedingten Engpässen oder gar Verklemmungen. Eine Vielzahl dieser Engpässe wird dabei durch Pseudokonflikte auf der Indexebene

Erzeugen eines neuen Dokumenten-Identifikators *Dok-ID*
INSERT INTO D VALUES ($Dok\text{-}ID, a_1, a_2, \ldots, a_n, t_1, \ldots, t_m$)
for $i = 1$ **to** m **do**
  ⋮
  Entfernen der Duplikate aus {Des} → {Des}
  Des-ID={}
  for all w **in** {Des} **do**
    SELECT Des-ID INTO *did* FROM Mapping WHERE Des=w
    if keine Des-ID für w vorhanden **then**
      Erzeugen einer neuen Des-ID → did
      INSERT INTO Mapping VALUES(w, did)
    end if
    Des-ID=Des-ID∪did
  end for
  for all w **in** Des-ID **in ascending order do**
    INSERT INTO IL_i VALUES(w, Dok-ID)
  end for
end for

Algorithmus 14: Einfügen eines Dokuments

ResDoks={}
for $i = 1$ **to** m **do**
  for $j = 1$ **to** Anzahl spezifizierte Suchwörter für t_{x_i} **do**
    SELECT Des-ID INTO *did* FROM Mapping WHERE Des=$w_j^{x_i}$
    SELECT Dok-ID INTO *temp* FROM $IL_{t_{x_i}}$ WHERE Des-ID=did
    if j=1 **then**
      TempDoks = temp
    else
      TempDoks = TmpDoks ∩ temp
    end if
  end for
  if i=1 **then**
    ResDoks = TempDoks
  else
    ResDoks = ResDoks ∩ TempDoks
  end if
  if ResDoks = {} **then**
    Suche beenden, kein Dokument qualifiziert sich
  end if
end for
SELECT * FROM *D* WHERE Dok-ID IN {ResDoks}

Algorithmus 15: Suchen von Dokumenten

```
Dokumenten-IDs der zu löschenden Dokumente ermitteln → {Dok-ID}
for all d = 1 in {Dok-ID} do
  for i = 1 to m do
    Extrahieren der einzelnen Wörter aus t_i → {Des} des Dokuments d
    Entfernen der Stopworte aus {Des} → {Des}
    Reduzieren der Wörter {Des} auf ihre Wortstämme → {Des}
    Entfernen der Duplikate aus {Des} → {Des}
    Des-ID={}
    for all w in {Des} do
      SELECT Des-ID ID INTO did FROM Mapping WHERE Deskriptor=w
      Des-ID=Des-ID∪did
    end for
    for all w in {Des-ID} in ascending order do
      DELETE FROM IL_i WHERE Des-ID=w AND Dokument-ID=d
    end for
  end for
  DELETE FROM D WHERE Dokument-ID=d
end for
```

Algorithmus 16: Löschen eines Dokuments

verursacht. Diese Leistungsengpässe können durch den Einsatz eines mehrschichtigen Datenbanksystems entschärft werden. Nachfolgend wird speziell der Einsatz eines Datenbanksystems mit einer dreischichtigen Transaktionsverwaltung untersucht, dessen unterste Ebene L_0 von einem relationalen Datenbanksystem gebildet wird und auf deren oberster Ebene L_2 die Operationen *InsertDocument*, *DeleteDocument* und *RetrieveDocument* zur Verfügung stehen. Die Kompatibilität der Operationen dieser Ebene wurde bereits in Abschnitt 3.5 besprochen.

Alle semantisch reichen Operationen der Ebene L_2 — sowie die obenstehend nicht explizit beschriebenen Kompensationsoperationen — werden auf Operationen der Ebene L_1 abgebildet, welche nachfolgend besprochen werden.

Einfügen eines neuen Dokuments Das Einfügen von Dokumenten gemäss Algorithmus 14 besteht aus zwei voneinander unabhängigen Teilen: Dem Einfügen des Dokuments in die Datenbank und dem manuellen Indexieren der einzelnen Textattribute. Das Indexieren eines einzelnen Textattributs t lässt sich wiederum in das Extrahieren der Deskriptoren aus t und das Einfügen dieser Deskriptoren in die invertierte Liste unterteilen. Für das Einfügen eines Dokuments werden daher auf Ebene L_1 drei Operationen zur Verfügung gestellt:

- *MInsDok*$(a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m) \rightarrow$ *Dok-ID* fügt ein Dokument in die Datenbank ein und retourniert den internen Identifikator dieses Dokuments..
- *MExtDes(Attribut, Extrakt, Stop)* $\rightarrow$ ({*Des-ID*}) extrahiert die Wörter aus dem Textattribut *Attribut*, entfernt die Stopwörter und reduziert die verbleibenden Wörter auf den Wortstamm. Es wird dabei die für das Attribut definierte Extraktionsfunktion *Extrakt* bzw. Stopwortliste *Stop* verwendet. Die resultierenden Deskriptoren werden anschliessend auf Deskriptor-IDs abgebildet und diese retourniert.

  Tritt bei der Extraktion der Deskriptoren *Des* aus einem Textdokument ein Deskriptor zum ersten Mal auf, existiert kein Eintrag für *Des* in der *Mapping*–Relation. In diesem

Fall muss eine *Des-ID* für den Deskriptor *Des* erzeugt und das Paar (*Des*, *Des-ID*) in die Mapping-Relation eingefügt werden.

- *MInsDes*(*IL*, {*Des-ID*}, *Dok-ID*) fügt die Deskriptoren {*Des-ID*} des Dokuments *Dok-ID* in die invertierte Liste *IL* ein.

Suchen von Dokumenten Die Suche von Dokumenten erfolgt ebenfalls in einem zweistufigen Verfahren: Zuerst werden die Postinglisten der Suchdeskriptoren gelesen und die Schnittmenge gebildet. Daraus resultieren jene Dokumente, welche sich auf die Anfrage qualifizieren. Diese werden anschliessend gelesen und dem Benutzer zur Verfügung gestellt. Für die Suche von Dokumenten stehen folgende L_1–Operationen zur Verfügung:

- *MRetDes*(*IL*, *Des*) $\rightarrow$ {(*AttName*, *Dok-ID*}) ermittelt die IDs aller Dokumente aus der invertierten Liste *IL*, welche den Deskriptor *Des* enthalten.

  Falls der Deskriptor *Des* ein Stopwort ist, retourniert diese Operation einen Fehlercode.

- *MRetDok*(*Dok-ID*) $\rightarrow$ (*Dok-ID*, $a_1, a_2, \ldots a_n, t_1, t_2, \ldots, t_m$) liest das durch die Dokumenten-ID *Dok-ID* identifizierte Dokument.

Löschen von Dokumenten Wie das Vorgehen beim Einfügen besteht auch das Löschen von Dokumenten gemäss Algorithmus 16 aus drei Schritten: Dem Ermitteln aller Dokumente, welche gelöscht werden sollen, dem Löschen der Deskriptoren aus den invertierten Listen und dem Löschen der Dokumente aus der Datenbank. Auf Ebene L_1 werden dafür zwei weitere Operationen benötigt:

- *MDelDes*(*IL*, {*Des-ID*}, *Dok-ID*) löscht die Deskriptoren *Des-ID* des Dokuments *Dok-ID* aus der invertierten Liste *IL*.

- *MDelDokDok-ID*) löscht das durch die Dokumenten-ID *Dok-ID* identifizierte Dokument aus der Datenbank.

Die zu diesen L_1–Operationen gehörende Konfliktmatrix wird — unter Berücksichtigung der Überlegungen in Abschnitt 3.5 betreffend Pseudokonflikte auf Ebene L_1 und dem Einsatz eines serialisierenden Schedulers auf Ebene L_2 — in Abbildung 5.1 gezeigt. Auffallend ist, dass alleine die Operation *MExtDes* mit sich selbst in Konflikt steht. Dies hat zwei Ursachen:

- Angenommen, zwei parallele Benutzertransaktionen T und T' extrahieren beim Ausführen von *MExtDes* jeweils einen bisher unbekannten Deskriptor Des^T und $Des^{T'}$. Wird *MExtDes* für Transaktion T vor *MExtDes* für Transaktion T' und werden neuen Deskriptoren Zahlen in aufsteigender Reihenfolge zugeordnet, erhält *des* die Zahl n, der Deskriptor $Des^{T'}$ die Zahl $n+1$. Ist die Ausführungsreihenfolge umgekehrt, wird dem Deskriptor *des* die Zahl $n+1$ und $Des^{T'}$ die Zahl n zugeordnet. Das Ergebnis der Operationen ist daher von der Ausführungsreihenfolge abhängig. Die Operationen kommutieren nicht.

  Vom semantischen Standpunkt aus gesehen ist dieser Konflikt nicht notwendig: Deskriptor-IDs werden nur in den internen Textindexstrukturen verwendet. Welche Deskriptor-ID einem Deskriptor zugewiesen wird, ist ohne Belang, sofern sie eindeutig ist.

	MInsDok	MExtrDes	MInsDes	MDelDok	MDelDes	MRetDok	MRetDes
MInsDok	+	+	+	n/a	+	n/a	+
MExtDes	+	−	+	+	+	+	+
MInsDes	+	+	+	+	P	+	P
MDelDok	n/a	+	+	n/a	+	n/a	+
MDelDes	+	+	P	+	P	+	P
MRetDok	n/a	+	+	n/a	+	+	+
MRetDes	+	+	P	+	P	+	+

Abbildung 5.1: Konfliktmatrix auf Ebene L_1 (vollständig rücksetzbar)

- Angenommen, zwei parallele Benutzertransaktionen T und T' extrahieren beim Ausführen der Operation $MExtDes^T$ und $MExtDes^{T'}$ einen bisher unbekannten Deskriptor *Des*. Wird $MExtDes^T$, vor $MExtDes^{T'}$ ausgeführt, erzeugt $MExtDes^T$ eine neue Deskriptor-ID und fügt diese in die Mapping–Relation ein. $MExtDes^{T'}$ verwendet anschliessend die neu erzeugte Deskriptor-ID. Wird die Benutzertransaktion T zurückgesetzt, müssen sämtliche Änderungen kompensiert werden. Dies heisst, dass der Eintrag für den neuen Deskriptor aus der Mapping–Relation entfernt werden muss. Kommutieren die beiden Operationen, ist dies jedoch nicht möglich, da die neue Deskriptor-ID bereits von T' gelesen und verwendet wurde.

  Würden die Änderungen an der Mapping-Relation im Fall eines Transaktionsabbruchs nicht rückgängig gemacht werden, würden zwei *MExtDes*–Operationen kommutieren. Ein solches Vorgehen ist semantisch gesehen ohne Nachteil: Wird ein solches Mapping zwischen Deskriptor und Deskriptor-ID nicht entfernt, steht es nachfolgenden *MExtDes*–Operationen zur Verfügung und muss dann nicht erst erzeugt werden.

Aus dieser Sichtweise heraus ist es möglich, die in Abbildung 5.1 definierte Konfliktmatrix durch die in Abbildung 5.2 gezeigte zu ersetzen. *o/B* bedeutet dabei, dass ein Konflikt zwischen zwei Operationen semantisch gesehen ohne Bedeutung ist und diese Operationen daher kommutieren.

	MInsDok	MExtrDes	MInsDes	MDelDok	MDelDes	MRetDok	MRetDes
MInsDok	+	+	+	n/a	+	n/a	+
MExtDes	+	o/B	n/a	+	+	+	+
MInsDes	+	n/a	+	+	P	+	P
MDelDok	n/a	+	+	n/a	+	n/a	+
MDelDes	+	+	P	+	P	+	P
MRetDok	n/a	+	+	n/a	+	+	+
MRetDes	+	+	P	+	P	+	+

Abbildung 5.2: Konfliktmatrix auf Ebene L_1 (unvollständig, aber korrekt rücksetzbar)

5.3 Effiziente Dokumentenverwaltung und -suche mittels TPM/ONT

In den bisherigen Diskussionen eines Systems zur Verwaltung von und Suche nach Textdokumenten wurde wiederholt darauf hingewiesen, dass ein solches System mit Vorteil durch ein dreischichtiges Datenbanksystem realisiert wird. Mit TPM/ONT steht jedoch nur ein zweischichtiges System zur Verfügung. Im Abschnitt 3.5 über Mehrschichtentransaktionen wurde bereits erläutert, dass auf einer Ebene L_i auf die Realisierung eines eigenständigen Schedulers verzichtet werden kann, sofern alle Operationen dieser Ebene miteinander kompatibel sind. Dies ist bei einem System zur Dokumentenverwaltung auf Ebene L_1 der Fall, sofern auf Ebene L_2 sichergestellt wird, dass Konflikte zwischen L_1–Operationspaaren, die ein n/a in der Konfliktmatrix aufweisen, bereits auf Ebene L_2 korrekt behandelt werden. TPM/ONT — ein zweischichtiges Datenbanksystem — kann daher zur Verwaltung semistrukturierter Dokumente eingesetzt werden. Die im letzten Abschnitt vorgestellten Operationen *MInsDok,MExtDes, MInsDes, MDelDok, MDelDes, MRetDok,MRetDes* sind in TPM/ONT die Operationen der Ebene L_1. Jede dieser Operationen wird in Form eines eigenständigen Services des TP-Monitors realisiert. Um die Korrektheit von Benutzertransaktionen sicherzustellen, werden zusätzlich folgende Services bereitgestellt:

- *LockInsert*$(a_{1,2}, \ldots, a_n, t_1, t_2, \ldots, t_m)$ beschreibt das einzufügende Dokument durch ein Prädikat (erzeugt aus den einzelnen Text- und Nichttextattributen) und fordert eine Einfügesperre für das durch dieses Prädikat beschriebene Dokument an.
- *LockRetrieve(Predicate)* erhält als Eingabeparameter ein Prädikat, das die zu suchenden Dokumente beschreibt und fordert für die durch dieses Prädikat beschriebene Dokumentenmenge eine Suchsperre an.
- *LockDelete(Predicate)* erhält als Eingabeparameter ein Prädikat, das die zu löschenden Dokumente beschreibt und fordert für die durch dieses Prädikat beschriebene Dokumentenmenge eine Löschsperre an.

Wie in einem klassischen Datenbanksystem müssen diese L_1–Lockoperationen vor Ausführung ihrer korrespondierenden L_1–Operationen (`MInsDok`, `MRetDok`, `MDelDok`) ausgeführt werden. Kann eine Sperre aufgrund eines Sperrkonflikts nicht gewährt werden, wird die anfordernde Transaktion verzögert, bis die Transaktion, welche diesen Konflikt verursacht hat, abgeschlossen worden ist.

Um die Konsistenz der Datenbank auch im Fehlerfall garantieren zu können, müssen für alle vorgestellten Operationen Kompensationsoperationen (Undo–Operationen) gemäss Abschnitt 4.5 definiert werden:

- *MInsDok*$^{-1}$*(Dok-ID)* löscht das Dokument mit Identifikator *Dok-ID* aus der Datenbank.
- *MExtrDes*$^{-1}$() ist eine Nulloperation: Im Rahmen dieser Operation wird die Datenbasis unter Umständen verändert (Erzeugen neuer Deskriptor-IDs, siehe Diskussion auf Seite 87 ff.). Solche Änderungen müssen jedoch nicht kompensiert werden, da diese keine integritätsverletzenden Auswirkungen auf die Datenbank haben.

- $MInsDes^{-1}(IL, \{Des\text{-}ID\}, Dok\text{-}ID)$ entfernt die Deskriptoren $\{Des\text{-}ID\}$ des Dokuments *Dok-ID* aus der invertierten Liste *IL*.
- $MDelDes^{-1}(IL, \{Des\text{-}ID\}, Dok\text{-}ID)$ fügt die Deskriptoren $\{Des\text{-}ID\}$ des Dokuments *Dok-ID* wieder in die invertierte Liste *IL* ein.
- $MDelDok^{-1}(Dok\text{-}ID, a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m)$ fügt ein gelöschtes Dokument d wieder in die Datenbank ein.
- $MRetDes^{-1}()$ ist eine Nulloperation, da im Rahmen von *MRetDes* nur Daten gelesen wurden.
- $MRetDok^{-1}()$ ist eine Nulloperation, da im Rahmen von *MRetDok* nur Daten gelesen wurden.

Um diese Undo-Operationen erfolgreich auszuführen, wird im Rahmen ihrer Do-Operationen die hierfür notwendige Undo-Information geschrieben: Für die Operation *MInsDoc* genügt es, im L_1–Log die Dokumenten-ID des neu eingefügten Dokuments festzuhalten. Bei *MDelDok* hingegen muss das komplette Dokument auf das L_1–Log geschrieben werden, um es im Fall eines AP–Transaktionsabbruchs wieder in die Datenbank einfügen zu können.

Für die Ausführung der Operation $MInsDes^{-1}$ wäre es ausreichend, die Dokumenten-ID des neu indexierten Attributs zu kennen. Ein effizientes Löschen von Verweisen aus einer invertierten Liste ist jedoch nur möglich, wenn die Deskriptor-IDs bekannt sind. Aus Effizienzüberlegungen ist es notwendig, diese im L_1–Log festzuhalten. Für die Durchführung der Operation $MDelDes^{-1}$ schliesslich müssen die gelöschten Deskriptor-IDs in das L_1–Log geschrieben werden.

Logisch gesehen handelt es sich bei Textindexstrukturen um eine *abgeleitete* Information, welche aus den Primärdaten, den Attributen der Textdokumente, gewonnen wird. Somit könnten auch die Daten zur Durchführung der Undo-Operationen auf Textindexstrukturen aus den Primärdaten gewonnen werden, sofern diese zu Beginn einer Undo-Operation zur Verfügung stehen. Für das Einfügen und Löschen von Textdokumenten bedeutet dies konkret:

- Die Operation *MInsDoc* muss *vor* der Indexierung der Textattribute durch `MInsDes` abgeschlossen sein. Die Indexierung der einzelnen Textattribute kann parallel zueinander erfolgen.

  Im Rahmen der Ausführung der Operation *MInsDes* wird nicht mehr das ganze Attribut auf das L_1–Log geschrieben, sondern nur die Dokumenten-ID und der Name des indexierten Attributs.

  Wird eine AP-Transaktion abgebrochen, werden zuerst die Textindexstrukturen geändert. Da sich das Dokument zu diesem Zeitpunkt noch persistent in der Datenbank befindet, können die Undo-Operationen $MInsDes^{-1}$ auf die einzelnen Attribute des Dokuments zugreifen, die Deskriptoren aus den Attributen berechnen und diese aus den Textindizes löschen.

- Die Operation *MDelDoc* darf erst *nach* dem Löschen aller Verweise auf das zu löschende Dokument in den Textindexstrukturen durchgeführt werden. Das Ausführen von *MDelDes* für die einzelnen Textattribute t_j kann parallel erfolgen.

Die Undo-Information der Operation *MDelDoc* besteht wiederum aus der Dokumenten-ID sowie dem Namen des Attributs, dessen Textindex geändert wird.

Im Fall eines AP-Transaktionsabbruchs wird zuerst das Dokument mittels $MDelDok^{-1}$ wieder persistent in die Datenbank eingefügt. Dieses steht sodann für die Berechnung der Deskriptoren durch $MDelDes^{-1}$ wieder zur Verfügung.

Die hieraus resultierenden applikationsseitigen Aufrufe zum Einfügen, Suchen und Löschen von Dokumenten finden sich in den Algorithmen 17, 18 und 19. Dazu folgendes:

- Beim Einfügen von Dokumenten kann die Anforderung des Sperrprädikats (*LockInsert*) und das Einfügen des Dokuments in die Datenbank (*MInsDok*) in einem Service zusammengefasst werden.
- Die Services *MExtDes* und *MInsDes* zur Indexierung der einzelnen Textattribute t_j könnten ebenfalls in einem einzigen Service zusammengefasst werden. Soll aber auch das Einfügen der Deskriptoren eines einzelnen Textattributs in die invertierten Listen parallelisiert werden, sind diese Services einzeln aufzurufen.

  Dasselbe gilt für das Löschen der Deskriptoren aus einer invertierten Liste.

```
{Einfügen eines Dokuments d(Dok-ID, a_1, a_2, ..., a_n, t_1, t_2, ..., t_m)}
FML-Buffer BufIn erzeugen und Dokument d in den Buffer einfügen
FML-Buffer BufOut für etwaige Rückgabewerte erzeugen
cCALL(APTAID, LockInsert, BufIn, BufOut)
if Sperre gewährt then
  for i = 1 to m do
    FML-Buffer BufIn leeren
    Attributs t_i und Name der invertierten Liste in den FML-Buffer BufIn schreiben
    cACALL(APTAID, MExtrDes, BufIn) → CallID_i
  end for
  for i = 1 to 2 * m do
    FML-Buffer BufOut leeren
    cWAIT(ANY, BufOut) → WaitID
    if WaitID IN {CallID_1, CallID_2, ... CallID_m} then
      {Die von cWAIT retournierte ID stammt vom Aufruf des Services MExtrDes}
      cCALL(APTAID, MInsDes, BufOut)
    end if
  end for
end if
FML-Buffer BufIn entfernen
FML-Buffer BufOut entfernen
```

Algorithmus 17: Applikationsseitiger Ablauf zum Einfügen eines Dokuments in TPM/ONT

```
{Suchen aller Dokumente mit Wort d_i in Attribut mit Namen n_i}
FML-Buffer BufIn erzeugen und Dokument d in den Buffer einfügen
FML-Buffer BufOut für etwaige Rückgabewerte erzeugen
BufIn mit (d_i, n_i)-Paaren füllen
cCALL(APTAID, LockRetrieve, BufIn)
if Sperre gewährt then
  ResDoks= {}
  for i = 1 to Anzahl spezifizierte Wörter do
    FML-Buffer BufIn leeren
    (d_i, n_i)-Paar in FML-Buffer BufIn einfügen
    cACALL(APTAID, MRetDes, BufIn)
  end for
  for i = 1 to Anzahl spezifizierte Wörter do
    FML-Buffer BufIn leeren
    cWAIT(ANY, BufOut)
    if Rückgabewert von cWAIT ≠ Stopwort then
      TempDoks=Des-IDs aus BufOut
      ResDoks=ResDok∪TempDoks
    end if
  end for
  for j = 1 to Anzahl Dokumenten-IDs in ResDoks do
    FML-Buffer BufIn und BufOut leeren
    Dokumenten-ID ResDoks_j in FML-Buffer BufIn einfügen
    cCALL(APTAID, MRetDok, BufIn, BufOut)
    {Resultatdokument in BufOut weiterverarbeiten}
  end for
else
  {Sperre nicht gewährt}
end if
```

Algorithmus 18: Applikationsseitiger Ablauf zum Suchen von Dokumenten in TPM/ONT

{Löschen von Dokumenten $d(\textit{Dok-ID}, a_1, a_2, \ldots, t_1, t_2, \ldots, t_m)$ die einem Prädikat genügen}
{Dokumenten-IDs mit Hilfe des Suchalgorithmus ermitteln, jedoch anstelle einer Suchsperre eine Löschsperre anfordern}
for $i = 1$ **Anzahl zu löschende Dokumente do**
  Dokument lesen $\rightarrow d(\textit{Dok-ID}, a_1, a_2, \ldots, t_1, t_2, \ldots, t_m)$
  for $j = 1$ **to** m (=Anzahl Textattribute des Dokuments) **do**
    FML-Buffer *BufIn* leeren
    Dokumenten-ID ResDok_i in FML-Buffer *BufIn* schreiben
    Name des Attributs t_j in FML-Buffer *BufIn* schreiben
    Inhalt des Attributs t_j in FML-Buffer *BufIn* schreiben
    cACALL(APTAID, MExtr, BufIn)$\rightarrow$ $\textit{Call-ID}_i$
  end for
  for $j = 1$ **to** $2m$ **do**
    FML-Buffer *BufOut* leeren
    cWAIT(ANY)$\rightarrow$ *BufOut*
    if Call-ID der cWAIT-Anweisung **in** $\{\textit{Call-ID}_1, \textit{Call-ID}_2, \ldots, \textit{Call-ID}_m\}$ **then**
      cACALL(APTAID, MDelDes, BufOut)
    end if
  end for
  FML-Buffer *BufIn* und *BufOut* leeren
  Dokumenten-ID *Dok-ID* in FML-Buffer *BufIn* schreiben
  cCALL(APTAID, MDelDok, BufIn, BufOut)
end for

Algorithmus 19: Applikationsseitiger Ablauf zum Löschen von Dokumenten in TPM/ONT

6 Leistungsuntersuchungen

In den letzten Jahren haben Forschungsgruppen immer wieder Realisierungen neuer Transaktionsmodelle vorgestellt. Vielfach mussten sie sich jedoch den Vorwurf gefallen lassen, nicht nachweisen zu können, dass ihre wohl interessanten und theoretisch fundierten Konzepte den herkömmlichen Modellen in der Praxis überlegen sind. TPM/ONT implementiert kein neues Transaktionsmodell. Die verwendeten offen geschachtelten Transaktionen sind in der Literatur bekannt [Wei91]. Im Unterschied zu anderen Prototypen zur Leistungsuntersuchung offen geschachtelter Transaktionen, beispielsweise [Has95], wird mit TPM/ONT kein von Grund auf neues Datenbanksystem realisiert. Es werden bestehende Systeme, und zwar der Transaktionsmonitor Tuxedo und wahlweise die relationalen Datenbanksysteme Oracle bzw. Sybase um eine Komponente zur korrekten Ausführung zweischichtiger Transaktionen erweitert. Dies hat den entscheidenden Vorteil, dass die mit TPM/ONT erzielten Ergebnisse direkt mit den Resultaten verglichen werden können, die auf einem Transaktionsmonitor mit einschichtiger Transaktionsverwaltung basieren.

In diesem Kapitel wird durch ausführliche Leistungsuntersuchungen der Nachweis erbracht, dass sich mittels TPM/ONT textuelle Daten und deren Indexstrukturen effizienter verwalten lassen als mit traditionellen Datenbanksystemen mit einschichtiger Transaktionsverwaltung. Die vorgestellten Leistungsuntersuchungen sind zweigeteilt: Im ersten Teil der Untersuchungen wird TPM/ONT mit Oracle als Ressource–Manager betrieben. Oracle setzt eine mehrversionierende Transaktionsverwaltung auf Tupelebene ein. Bei der Manipulation von Dokumenten ist daher mit keinen Sperrkonflikten zu rechnen. Ziel der Untersuchungen mit diesem Datenbanksystem ist es, einerseits die Kosten für die Ausführung mehrschichtiger Transaktionen mittel TPM/ONT gegenüber einschichtigen Transaktionen zu ermitteln und andererseits mögliche Leistungssteigerungen durch die Parallelisierung von Algorithmen (Leistungsgewinn durch Intra–Transaktions–Parallelität) aufzuzeigen. Im zweiten Teil der nachfolgenden Untersuchungen wird als Ressource–Manager das ebenfalls relationale Datenbanksystem Sybase eingesetzt. Sybase setzt zur Transaktionsverwaltung ein Striktes Zwei–Phasen Sperrprotokoll auf Seitenebene ein. Bei der Manipulation von Dokumenten ist aufgrund dieses Protokolls mit starken Behinderungen im Mehrbenutzerbetrieb zu rechnen. Ziel der Leistungsuntersuchungen mit Sybase ist es, eine mögliche Leistungssteigerung durch ein verkürztes Halten von Sperren auf Ebene des Ressource–Managers zu untersuchen.

6.1 Messumgebung

6.1.1 Datenkollektion

Die Auswahl einer geeigneten Testkollektion ist für die Glaubwürdigkeit einer leistungsorientierten Evaluation von grosser Bedeutung. Das Hauptanliegen der vorliegenden Arbeit ist es, aufzuzeigen, dass semistrukturierte Dokumente mit heutiger Datenbanktechnologie effizient online verwaltet werden können. "Online verwaltet" bedeutet in diesem Zusammenhang, dass Dokumente nicht im Batch–Betrieb über Nacht, sondern jederzeit eingefügt, geändert und gelöscht werden können, während gleichzeitig andere Benutzer nach Dokumenten suchen. Für die Leistungsuntersuchung sollten daher mit Vorzug Dokumentenkollektionen eingesetzt werden, welche sich auch in der Realität häufig ändern. Solche Dokumentenkollektionen sind beispielsweise *Usenet–News*. Diese können wie folgt charakterisiert werden:

```
Subject: Clinton's health at risk !!!!!
Date: Fri, 16 Aug 1996 05:04:36 GMT
From: jakala@netcom.com (henry jakala)
Organization: NETCOM Online Communication Services

1) Dole has lower cholesterol counts than Clinton
2) Dole is at a weight commensurate with his build, Clinton is overweight and
flabby-assed
3) Dole maintains a well balanced diet, Clinton is constantly stuffing his face with
artery clogging junk from McDonald's

from a medical standpoint based Dole is the healthier of the two even though he's 20
years older than Clinton

final point to ponder: Dole' records have been made public - where are Slick's and
what's he hiding ?
```

Abbildung 6.1: Beispiel eines Usenet–Artikels

- Usenet News sind stark dynamische Daten: Während 24 Stunden am Tag werden irgendwo neue Artikel geschrieben, die innerhalb weniger Stunden weltweit verfügbar sind.
- Usenet News sind in über 10'000 Gruppen unterteilt, beispielsweise Computer, Kultur, Politik, ...
- In jeder Gruppe erscheinen täglich ein paar wenige bis mehrere hundert oder tausend neue Artikel.
- In allen Gruppen zusammen erscheinen rund 250'000 neue Artikel pro Tag.
- Geht man von einer durchschnittlichen Länge von 2 Kilobyte pro Artikel aus, fällt pro Tag rund ein halbes Gigabyte neuer Information an.
- Ein einzelner Artikel ist ein semistrukturiertes Dokument. Der strukturierte Anteil eines Artikels besteht aus einem Attribut *Autor*, der einen Artikel an einem bestimmten *Datum*

zu einem bestimmten *Thema* geschrieben und an bestimmte *Gruppen* geschickt hat. Der unstrukturierte Teil ist der eigentliche *Inhalt* des Artikels. Für ein Beispiel eines Artikels siehe Abbildung 6.1.

- Im Volltextteil eines Artikels finden sich — nach Entfernung aller Stopwörter und Durchführung einer Wortstammreduktion — rund 100 verschiedene Wörter.

Leser von Usenet Artikeln greifen auf diese in der Regel auf zwei Arten zu: Im sogenannten *Newsreader*–Modus starten sie ein spezielles Programm, wählen eine bestimmte Gruppe aus und lesen (sequentiell) alle Artikel dieser Gruppe, die seit dem letzten Zugriff auf diese Gruppe neu hinzugekommen sind. Im *Search*–Modus spezifiziert ein Benutzer eine Reihe von Stichwörtern und erhält alle Artikel einer oder mehrerer Newsgruppen, die diese Stichwörter enthalten. Diese Art der Informationssuche wird heute von einer Reihe von kommerziellen Internet Search Engines (siehe Abschnitt 2.4.2) oder Forschungsprototypen wie *Tapestry* [TGNO92], *SIFT* [YGM95] oder [BM96] angeboten. Nachfolgend wird ein Informationssystem zur Unterstützung des Search-Modus untersucht. Im Gegensatz zu den vorgenannten Systemen (Tapestry und SIFT) werden neue Dokumente nicht unmittelbar gegen eine Menge von Anfrageprofilen geprüft und anschliessend entfernt, sondern online in einer Datenbank gespeichert, wo nach ihnen jederzeit — ebenfalls online – gesucht werden kann.

Unter der Annahme, dass rund 250'000 neue Artikel zu je rund zwei Kilobyte pro Tag anfallen, ist mit 500 Megabyte neuer Information täglich zu rechnen[1]. Aufgrund dieser Datenflut müssen selbst in grossen Datenbanksystemen ältere Artikel aus der Datenbank entfernt werden, um Platz für neue Artikel zu schaffen. Dies ist für die Benutzer im Allgemeinen unproblematisch, da die in den Artikeln enthaltene Information oftmals innert sehr kurzer Zeit veraltet und damit uninteressant ist.

Werden pro Tag ebensoviele Artikel eingefügt wie entfernt, sind etwa 500'000 Artikel täglich zu bearbeiten. Geht man davon aus, dass jeder Artikel im Durchschnitt rund 200 Deskriptoren enthält, die in invertierten Listen gespeichert werden, benötigt man für das Einfügen oder Löschen eines Dokuments ein bis zwei Sekunden. Wird nur ein einziger Benutzerprozess zur Verarbeitung eingesetzt, werden über 500'000 Sekunden (rund sechs Tage) für die Verarbeitung der News–Daten eines einzigen Tags benötigt.

Durch den Einsatz mehrerer Einfüge- und Löschprozesse reduziert sich die Verarbeitungszeit für ein Tagesgeschäft (Abbildung 6.2). Aufgrund gegenseitiger Behinderungen dieser parallel ablaufenden Prozesse wird kein linearer Speedup erreicht. Hingegen erreicht man sehr schnell den Punkt, an dem in einem herkömmlichen Datenbanksystem durch Hinzufügen neuer Ressourcen (parallele Einfüge- bzw. Löschprozesse) keine Leistungsverbesserung mehr erzielt werden kann. Unter diesen Umständen ist es nicht möglich, die Daten eines Tages innerhalb von 24 Stunden zu verarbeiten. Die nachfolgenden Leistungsmessungen zeigen, dass die Antwortzeiten jedoch durch den Einsatz *TPM/ONT-Text* weiter reduziert werden können.

Für alle durchgeführten und nachfolgend präsentierten Messungen werden Artikel der Usenet–Gruppe *talk.politics.libertarian* verwendet. Diese werden in einer Relation

```
TPL(Dok-ID      NUMBER,
    Datiert     DATE,
```

[1]Diese 500 Megabyte umfassen lediglich die Rohdaten, nicht jedoch die Zugriffsstrukturen über diesen Daten.

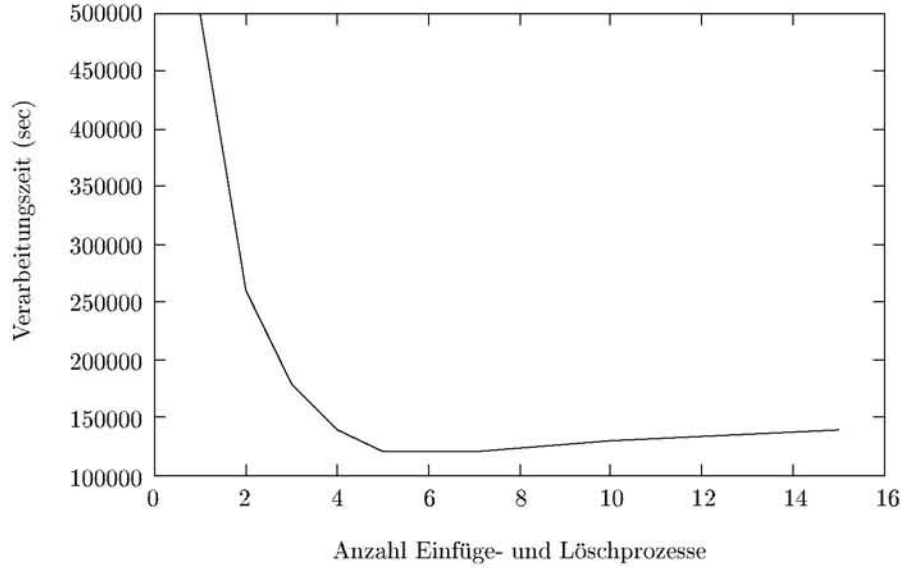


Abbildung 6.2: Verarbeitungszeit für die Usenet News eines Tages

```
Absender     STRING
Organisation STRING,
Titel        TEXT,
Inhalt       TEXT)
```

physisch gespeichert. Dabei bezeichnet *Datiert* das Datum, an dem der Artikel geschrieben wurde, *Absender* den Namen des Autors und *Organisation* den Namen der Institution, bei welcher der Autor beschäftigt ist. *Titel* beschreibt das Thema des Artikels und *Inhalt* ist der Volltextteil des Artikels.

Um eine schnelle wortbasierte Suche nach einzelnen Artikeln zu unterstützen, wird jeweils ein wortbasierter Textindex über den Attributen *Titel* und *Inhalt* angelegt. Zusätzlich wird ein ebenfalls wortbasierter Index über einem virtuellen Textattribut, *Kombiniert* genannt, gewartet. Dieses virtuelle Attribut besteht aus den aneinandergefügten Attributen *Absender, Organisation, Titel* und *Inhalt*, also aus allen Volltextattributen der Relation *TPL*. Der Index über diesem Attribut erlaubt es, effizient nach allen Artikeln der Gruppe zu suchen, die in irgendeinem Textattribut ein oder mehrere bestimmte Wörter enthalten.

Jeder Textindex wird als invertierte Liste modelliert und physisch in Relationen gespeichert (siehe Abschnitt 2.3.3). Die drei Textindizes werden nachfolgend mit I_{Titel}, I_{Inhalt} und $I_{Kombiniert}$ abgekürzt.

Nach Entfernung der Stopwörter und Durchführung einer Wortstammreduktion findet man im Durchschnitt im Attribut *Titel* fünf, in den Attributen *Organisation* und *Autor* zusammen wiederum fünf und im Attribut *Inhalt* rund 85 verschiedene Wörter pro Artikel. Beim Einfügen oder Löschen eines Dokuments müssen daher rund 180 Einträge in den invertierten Listen I_{Titel}, I_{Inhalt} und $I_{Kombiniert}$ gewartet werden.

6.1.2 Systemkonfiguration

Rechner, Betriebssystem Alle Experimente werden auf einem SparcCenter 2000 durchgeführt. Dieses ist mit zehn Prozessoren mit einer Taktfrequenz von 40 MHz sowie 300 MByte Hauptspeicher ausgestattet. Das verwendete Betriebssystem ist Solaris 2.5.

Transaktionsmonitor Als Basis für TPM/ONT wird der Transaktionsmonitor *Tuxedo* in der Version 5.1 mit 20 Serverprozessen eingesetzt.

Datenbanksysteme Im ersten Teil der nachfolgenden Leistungsuntersuchungen wird das Datenbanksystem *Oracle*, Version 7.2.3, verwendet. Das System wird symmetrisch betrieben, d.h. jedem Tuxedo–Serverprozess steht ein eigener Oracle–Serverprozess zur Verfügung.

Im zweiten Teil wird das Datenbanksystem Sybase Version 10.0.2 eingesetzt. Sybase erlaubt es, maximal soviele Serverprozesse zu betreiben, wie Prozessoren verfügbar sind. Es wird jedoch empfohlen, die Anzahl der Serverprozesse auf 80% der verfügbaren Prozessoren zu beschränken. Sybase wird daher mit 8 Serverprozessen betrieben.

Der Datenbankpuffer beträgt bei beiden Datenbanksystemen 2500 Seiten.

Datenallokation Für die Leistungsuntersuchungen werden Usenet–Artikel der Gruppe *talk.politics.libertarian* verwendet.

Für die Speicherung dieser Daten werden 12 Platten eingesetzt, wobei jeweils zwei Platten an einen SCSI–Kontroller angeschlossen sind. Auf Platte 1 wird die Relation *TPL* gespeichert. Die Textindizes I_{Titel} und I_{Inhalt} werden auf den Platten 2, 3 und 4 verteilt gespeichert. Diese Platten bilden ein vom Betriebssystem verwaltetes Metadevice. Die Daten auf diesem Device werden mit einem Granulat von 40 Kilobyte (1 Track) nach einem Round–Robin–Verfahren über die drei Platten verteilt. Der Textindex $I_{Kombiniert}$ wird auf einem Metadevice derselben Art über die Platten 5, 6 und 7 gespeichert. Von den Platten 8 bis 12 schliesslich werden jeweils zwei Platten für die Verwaltung der Datenbank-internen Undo und Redo-Logdateien und eine Platte zum Verwalten temporärer Daten verwendet.

Zu Beginn einer jeden Messserie befinden sich — falls nicht explizit anders erwähnt — 40'000 indexierte Dokumente in der Datenbank.

Güte der Messungen Alle Experimente werden sooft wiederholt, bis bei einer Schwankung von $\pm 10\%$ um den Mittelwert ein Konfidenzniveau von 90% erreicht wird.

6.2 Leistungsmessung: Datenbanksystem mit Mehrversionen–Transaktionsverwaltung (Oracle)

In einer ersten Serie von Messungen wird das leistungsmässige Verhalten von TPM/ONT-Text mit einem mehrversionierenden Datenbanksystem als Ressource Manager, speziell Oracle, untersucht. Ziel ist es, folgende Fragen zu klären:

Wie verhalten sich die Antwortzeiten und der Transaktionsdurchsatz für das Einfügen eines Dokuments,

1. wenn auf der einen Seite ein traditioneller Transaktionsmonitor und auf der anderen Seite TPM/ONT-Text verwendet wird, wobei in beiden Fällen alle Operationen einer Transaktion sequentiell ausgeführt werden (keine Intratransaktionsparallelität)?
2. wenn mehrere Benutzer gleichzeitig Dokumente mit respektive ohne TPM/ONT in die Datenbank einfügen?
3. wenn die einzelnen Operationen einer Einfügetransaktion in TPM/ONT nicht sequentiell, sondern parallel durchgeführt werden, sofern sich die Operationen nicht gegenseitig behindern?
4. wenn sich die Einfügetransaktionen gegenseitig behindern, falls Statistikinformationen im laufenden Betrieb nachgeführt werden?
5. wenn Dokumente eingefügt werden und gleichzeitig nach Dokumenten gesucht wird und sich Einfüge- und Lesetransaktionen gegenseitig behindern?

6.2.1 Referenzmessungen: TPM/ONT–Ausführungskosten

Mehrschichtigen Datenbanksystemen wird vielfach vorgeworfen, die Realisierung einer eigenständigen Transaktionsverwaltung auf jeder Ebene des Systems sei zeitintensiv, die Kosten seien oftmals höher als der erzielte Nutzen. In einer ersten Untersuchung werden daher die Kosten für die Ausführung von Mehrschichtentransaktionen mittels TPM/ONT ermittelt und mit dem Aufwand für die Ausführung flacher Transaktionen mittels Tuxedo verglichen.

Der serverseitige Aufwand für die Ausführung einer flachen Transaktion mit Tuxedo setzt sich zusammen aus den Kosten für das Starten und Beenden der RM–Transaktion durch Tuxedo (`tx_begin` und `tx_commit`), für die Koordination derselben über das X/A–Interface und für die Interaktionen mit der Datenbank (Embedded–SQL-Anweisungen).

In TPM/ONT resultiert der Aufwand aus dem Starten und Beenden der AP–Transaktion durch Aufruf der System-Services `L1_BOT` und `L1_EOT`, dem Starten und Beenden einer RM–Transaktion am Beginn resp. Ende jeder L_1–Operation mittels `EXEC SQL BEGIN TRANSACTION` bzw. `EXEC SQL COMMIT TRANSACTION`, dem Anfordern und Verwalten von L_1–Sperren, dem Schreiben von L_1–Loginformation sowie wiederum aus dem Aufwand für die Interaktionen mit der Datenbank selbst.

Um den Aufwand experimentell zu ermitteln, wird ein und dieselbe AP–Transaktion auf der einen Seite als einschichtige Transaktion mit Tuxedo ohne TPM/ONT und auf der anderen Seite als zweischichtige Transaktion mit TPM/ONT ausgeführt. Um vergleichbare Funktionalität zu erhalten, wird dabei in beiden Fällen auf die Ausführung asynchroner Service Calls verzichtet: Mehrere parallel abgesetzte Service Calls können von TPM/ONT parallel verarbeitet werden, während sie ohne TPM/ONT durch den Transaktionsmonitor sequentialisiert werden.

Für einen Kostenvergleich werden die Usenet–Artikel eines Monats (7'234 Artikel) durch einen einzelnen Einfügeprozess in eine zu Beginn leere Oracle–Datenbank eingefügt. Das Einfügen eines einzelnen Dokuments ohne TPM/ONT erfolgt dabei gemäss Algorithmus 20, das Einfügen

mit TPM/ONT gemäss Algorithmus 21. Tabelle 6.1 zeigt die durchschnittlichen Kosten für das Einfügen eines einzelnen Dokuments. Wie bei allen folgenden Messergebnissen bleiben bei der Berechnung der Kosten die ersten 2'000 eingefügten Dokumente unberücksichtigt (Einschwingphase des Datenbanksystems). Wie der Aufstellung zu entnehmen ist, bestehen zwischen einer

{Einfügen eines Dokuments $d(\textit{Dok-ID}, a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m)$}
tx_begin()
Buffer *BufIn* erzeugen und Dokument d in den Buffer einfügen
Buffer *BufOut* für etwaige Rückgabewerte erzeugen
tpcall(MDokIns, BufIn, BufOut)
for $i = 1$ **to** m **do**
  Buffer *BufIn* leeren
  Name und Wert des Attributs t_i in den FML-Buffer *BufIn* schreiben
  tpcall(MExtrDes, BufIn, BufOut)
  Buffer *BufIn* leeren
  tpcall(MInsDes, BufOut, BufIn)
end for
Buffer *BufIn* entfernen
Buffer *BufOut* entfernen
tx_commit()

Algorithmus 20: *Insert/Traditionell*: Applikationsseitiger Algorithmus zum Einfügen eines Dokuments in einer einschichtigen Datenbankumgebung (Tuxedo ohne TPM/ONT)

{Einfügen eines Dokuments $d(\textit{Dok-ID}, a_1, a_2, \ldots, a_n, t_1, t_2, \ldots, t_m)$}
cBOT()→APTAID
FML-Buffer *BufIn* erzeugen und Dokument d in den Buffer einfügen
FML-Buffer *BufOut* für etwaige Rückgabewerte erzeugen
cCALL(APTAID, LockEinfügen, BufIn, BufOut)
cCALL(APTAID, MDocIns, BufIn, BufOut)
for $i = 1$ **to** m **do**
  FML-Buffer *BufIn* leeren
  Name und Wert des Attributs t_i in den FML-Buffer *BufIn* schreiben
  cCALL(APTAID, MExtrDes, BufIn, BufOut)
  FML-Buffer *BufIn* leeren
  cCALL(APTAID, MInsDes, BufOut, BufIn)
end for
FML-Buffer *BufIn* entfernen
FML-Buffer *BufOut* entfernen
cEOT(APTAID)

Algorithmus 21: *Insert/ONT-Sequentiell*: Applikationsseitiger Algorithmus zum Einfügen eines Dokuments in einer zweischichtigen Datenbankumgebung ohne asynchrone Ausführung der einzelnen Operationen (Tuxedo mit TPM/ONT)

einschichtigen und einer mehrschichtigen Ausführung einer Einfügetransaktion keine signifikanten Unterschiede in der Gesamtausführungszeit. Dies ist auf den ersten Blick zu erwarten, da in beiden Fällen die gleichen Operationen sequentiell in derselben Reihenfolge durchgeführt werden. Bei genauerer Überlegung würde man ein deutlich schlechteres Abschneiden der mehrschichtigen Transaktion gegenüber der einschichtigen erwarten: Jede L_1–Operation einer Mehrschichtentransaktion wird im Rahmen einer eigenständigen Oracle-Transaktion aus-

	einschichtig	zweischichtig
Eingefügte Deskriptoren	173	173
Benötigte Oracle Transaktionen	1	4
Ausführungszeit (ms)	1'435	1'463
CPU-Zeit Oracle (ms)	390	472
Grösse L_0–RedoLog (KByte)	60	62
Ausführungszeit BOT (ms)	1	10
Dokumenteneinfügezeit (ms)	65	98
Indexierungszeit (sec)	1270	1345
Ausführungszeit EOT (ms)	98	10

Tabelle 6.1: Durchschnittliche Kosten für das Einfügen eines Dokuments mittels einschichtiger und zweischichtiger Transaktionen

geführt, die mit einem `COMMIT` abgeschlossen wird, was jeweils mit einem synchronen Schreiben eines Redo-Logsatzes auf eine Platte verbunden ist. Bei den Einfügetransaktionen fällt jedoch das synchrone Schreiben der Redo-Logsätze am Ende einer Transaktion nicht ins Gewicht. Der hierfür benötigte Zeitaufwand ist verglichen mit der Gesamtausführungszeit einer Transaktion vernachlässigbar.

Auffallend ist, dass der Aufwand für das Starten einer Tuxedo–Transaktion gleich Null ist, während das Beenden einer Tuxedo–Transaktion deutlich aufwendiger ist als das Abschliessen einer Transaktion in TPM/ONT. Der Grund hierfür liegt darin, dass im Fall von Tuxedo ohne TPM/ONT zu Beginn einer Transaktion keine Kommunikation mit den unterliegenden Ressource–Managern notwendig ist. Zu diesem Zeitpunkt ist nämlich noch nicht bekannt, auf welchen Datenbanksystemen die nachfolgenden Service Calls ausgeführt werden. Der Aufruf von `tx_begin` verursacht somit praktisch keinen Zeitaufwand. Bei TPM/ONT hingegen wird zu Beginn einer AP–Transaktion der Systemservice `L1_BOT` aufgerufen. Dieser erzeugt einen Identifikator für die AP–Transaktion und liefert diesen an den Benutzer zurück. Dieser Aufruf ist mit Kommunikationskosten in der Grössenordnung von 8–15 ms Sekunden verbunden.

Beim Beenden einer AP–Transaktion ist es umgekehrt. Im Fall von Tuxedo ohne TPM/ONT wird beim Aufruf von `tx_commit` ermittelt, auf welchen Datenbanksystemen im Rahmen dieser Transaktion Services ausgeführt wurden. Im Anschluss daran werden die Transaktionen auf den beteiligten Ressource-Managern beendet und das erfolgreiche Abschliessen in der Tuxedo–internen Logdatei vermerkt. Das Abschliessen einer RM–Transaktion ist kostenintensiv, da alle noch nicht geschriebenen Redo–Logsätze dieser Transaktion synchron geschrieben werden müssen. Im Gegensatz dazu werden beim Einsatz von TPM/ONT L_0–Logsätze nicht beim Aufruf von `L1_EOT`, sondern bereits im Rahmen der Ausführung der einzelnen L_1–Operationen geschrieben. Am Ende einer AP–Transaktion werden nur noch Redo-Logsätze für jene Datenbankoperationen geschrieben, die im Rahmen der Ausführung von `L1_EOT` ausgeführt werden. In der momentanen Realisierung sind dies die Logsätze für das Löschen der L_1–Logeinträge dieser Transaktion (siehe Abschnitt 4.4.2). Die Ausführungskosten der `COMMIT`–Operation sind aufgrund des kleineren L_0–Logs deutlich geringer als der `COMMIT`–Aufwand einer flachen Transaktion.

Im Zusammenhang mit Redo-Logs ist die Frage von Interesse, wieviel Zeit eine Datenbank für

das Schreiben von Redo-Logs benötigt. Beim Einfügen eines einzelnen Usenet–Artikels werden Redo–Logsätze mit einer durchschnittlichen Gesamtlänge von 70 Kilobyte erzeugt, die im Laufe der Einfügetransaktion, spätestens aber am Ende der Transaktion im Rahmen der `COMMIT`-Verarbeitung geschrieben werden müssen. Das Schreiben dieser Information ist notwendig, um die Dauerhaftigkeit von Änderungsoperationen auf der Ebene L_0 sicherzustellen. In einem mehrschichtigen System wie TPM/ONT ist die Sicherstellung der Persistenz auf Ebene L_0 durch das Schreiben von Redo-Logsätzen nicht notwendig, da Änderungen auf der unteren Ebene aufgrund der Loginformation der höheren Ebene jederzeit nachgefahren werden können. Zu beachten gilt, dass auf Ebene L_0 weiterhin Undo–Logs zur Sicherstellung der Atomarität von Transaktionen geschrieben werden müssen.

Oracle — als einschichtiges Datenbanksystem — gestattet normalerweise nicht, das Redo–Logging auszuschalten, da ansonsten die Dauerhaftigkeit von Änderungsoperationen nicht mehr garantiert ist. Trotzdem erlaubt ein nicht dokumentierter interner Parameter von Oracle jegliches Logging (d.h. Undo und Redo) zu unterbinden und dadurch den Transaktionsdurchsatz zu steigern. Durch das Aktivieren dieses Parameters wird jedoch auch das Undo–Logging unterbunden; im Fall eines Systemversagens oder eines Transaktionsabbruchs auf Ebene L_0 ist die Konsistenz der Datenbank nicht mehr sichergestellt. Ungeachtet dessen wird im Folgenden der Einfluss dieses Parameters auf die Ausführungskosten mehrschichtiger Transaktionen untersucht. Der Aufwand für das Schreiben von Undo–Logs wird dabei nicht berücksichtigt, da Undo–Logs nur vor Auslagerung eines L_0–Datenbankobjekts geschrieben werden müssen. Dies kann jederzeit asynchron erfolgen. Das Schreiben von Undo–Logs beeinflusst somit die Ausführungszeiten von Benutzertransaktionen nur unwesentlich.

```
{Einfügen eines Dokuments d(Dok-ID, a_1, a_2, ..., a_n, t_1, t_2, ..., t_m)}
cBOT()→APTAID
FML-Buffer BufIn erzeugen und Dokument d in den Buffer einfügen
FML-Buffer BufOut für etwaige Rückgabewerte erzeugen
cCALL(APTAID, LockEinfügen, BufIn, BufOut)
cCALL(APTAID, MDocIns, BufIn, BufOut)
for i = 1 to m do
  FML-Buffer BufIn leeren
  Name und Wert des Attributs t_i in den FML-Buffer BufIn schreiben
  cCALL(APTAID, MExtrDes, BufIn, BufOut)
  for j = 1 to Anzahl retournierter Deskriptoren step N do
    FML-Buffer BufIn leeren
    Deskriptoren des_j bis des_{j+N-1} aus BufOut nach BufIn kopieren
    cCALL(APTAID, MInsDes, BufIn, BufIn)
  end for
end for
FML-Buffer BufIn entfernen
FML-Buffer BufOut entfernen
cEOT(APTAID)
```

Algorithmus 22: *Insert/ONT-Sequentiell–N*: Applikationsseitiger Ablauf zum Einfügen eines Dokuments mit TPM/ONT ohne asynchrone Ausführung der einzelnen Operationen, Aufteilung der Deskriptoren beim Einfügen in die invertierte Liste

Das Wiederholen der Messserie mit TPM/ONT gemäss Algorithmus 21 ohne Schreiben von Loginformation zeigt in Bezug auf die Antwortzeiten nur eine geringe Leistungsverbesserung

von rund 10 Prozent gegenüber den Messergebnissen mit Redo–/Undo–Logging (Tabelle 6.2). Werden jedoch einzelne lange L_1–Operationen gemäss Algorithmus 22 in eine grosse Anzahl

	mit L_0-Logging	ohne L_0–Logging
Eingefügte Deskriptoren	173	173
Benötigte Oracle Transaktionen	4	4
Ausführungszeit (ms)	1'463	1'315
CPU-Zeit Oracle (ms)	370	360
Grösse L_0–Redolog (KByte)	62	0
Ausführungszeit BOT (ms)	10	10
Dokumenteneinfügezeit (ms)	98	80
Indexierungszeit (ms)	1345	1'215
Ausführungszeit EOT (ms)	10	10

Tabelle 6.2: Durchschnittliche Kosten für das Einfügen eines Dokuments gemäss Algorithmus 21 mit und ohne Logging auf Ebene L_0

kurzer L_1–Operationen aufgeteilt (für das Einfügen der Deskriptoren in eine invertierte Liste wird nicht mehr eine einzige, sondern es werden mehrere L_1–Operationen verwendet) und sequentiell ausgeführt, steigen die Kosten für das Einfügen und Indexieren der Dokumente an (Tabelle 6.3). Eine nähere Untersuchung der Kosten einschliesslich einem Vergleich mit den Kosten für die Ausführung der gleichen Operationen mit Redo-Logging zeigt, dass die höheren Kosten vor allem durch den Kommunikationsaufwand bestimmt sind (ein Service Call pro durchgeführter L_1–Operation) und somit nur indirekt durch die Mehrschichtentransaktionsverwaltung verursacht werden.

	$N = \infty$	$N = 25$	$N = 10$
Eingefügte Deskriptoren	173	173	173
Benötigte Oracle Transaktionen	4	6.3	9.6
Ausführungszeit (ms)	1'315	1'443	1'498
CPU-Zeit Oracle (ms)	360	400	440
Ausführungszeit BOT (ms)	11	11	11
Dokumenteneinfügezeit (ms)	80	80	80
Indexierungszeit (ms)	1'215	1'329	1'400
Ausführungszeit EOT (ms)	10	10	10

Tabelle 6.3: Durchschnittliche Kosten für das Einfügen von Dokumenten ohne L_0–Logging mittels zweischichtiger Transaktionen gemäss Algorithmus 21 ($N = \infty$) bzw. Algorithmus 22 ($N = 25, N = 10$)

Zusammenfassend kann festgehalten werden, dass das Logging auf Ebene L_0 beim Einfügen von Dokumenten aufgrund der Länge einer einzelnen Transaktion nur einen geringen Einfluss auf die Ausführungszeiten hat.

Das unnotwendige Schreiben von L_0–Loginformation bei mehrschichtigen Transaktionen kann jedoch dann zu drastischen Leistungseinbussen gegenüber einschichtigen Transaktionen führen,

wenn im Rahmen einer einzelnen AP–Transaktion sehr viele L_1–Operationen ausgeführt werden, die sich gegenseitig nicht behindern und die jeweils nur wenige Datenbankobjekte bearbeiten. In diesem Fall sollten L_1–Operationen zusammengefasst als *eine* L_1–Operation im Rahmen *einer* RM–Transaktion ausgeführt werden.

6.2.2 Leistungsverhalten bei Einsatz von Intra–Transaktions–Parallelität

Der Einbenutzerbetrieb, der im letzten Abschnitt als Grundlage für die Kostenschätzung von Mehrschichtentransaktionen verwendet wurde, ist in einer realen Anwendung eher die Ausnahme. Im Allgemeinen werden Daten in einem Informationssystem von mehreren Benutzern gleichzeitig bearbeitet. Im Rahmen der folgenden Messserie wird daher das Leistungsverhalten ein- und mehrschichtiger Einfügetransaktionen im Mehrbenutzerbetrieb mit 5, 10 und 20 parallel arbeitenden Benutzern untersucht. Jeder Benutzer wird durch einen eigenständigen Prozess simuliert. Dieser erhält als Eingabeparameter eine Datei voneinander verschiedener Usenet–Artikel. Diese werden unabhängig von den anderen Prozessen sequentiell und ohne Unterbruch in die Datenbank eingebracht. Die einzelnen Artikel werden im einschichtigen Fall mittels *Insert/TPM-Traditionell* (Algorithmus 20) und im mehrschichtigen Fall mit *Insert/ONT-Sequentiell, Insert/ONT-Sequentiell-N, Insert/ONT-Parallel* und *Insert/ONT-Parallel-N* (Algorithmen 21, 22, 23 und 24) verarbeitet. Mit Ausnahme von *Insert/ONT-Parallel* und *Insert/ONT-Parallel-N* wurden alle Algorithmen bereits vorgestellt. Diese beiden Algorithmen können wie folgt charakterisiert werden: Bei Einsatz von *Insert/ONT-Parallel* werden die einzelnen Attribute eines Dokuments nicht sequentiell, sondern parallel indexiert, wobei jedes Attribut im Rahmen einer eigenständigen RM–Transaktion indexiert wird; bei Verwendung von *Insert/ONT-Parallel-N* werden die Deskriptoren nicht im Rahmen *einer* RM–Transaktion verarbeitet, sondern es werden jeweils N Deskriptoren mittels einer eigenständigen RM–Transaktion in die invertierte Liste eingefügt. Eine graphische Beschreibung der für das Einfügen eines einzelnen Dokuments verwendeten Algorithmen findet sich in Abbildung 6.3. Bei den Algorithmen *Insert/ONT-Sequentiell–N* und *Insert/ONT-Parallel* wird angenommen, dass die Attribute *Inhalt* und *Kombiniert* jeweils M Deskriptoren enthalten. Mit $N = \frac{M}{2}$ müssen daher pro Attribut jeweils zwei L_1–Einfügeoperationen (`MInsDes`) ausgeführt werden.

	Benutzer		
	5	10	20
Insert/TPM-Traditionell	2.1	3.0	5.4
Insert/ONT-Sequentiell	2.1	3.0	5.3
Insert/ONT-Sequentiell-50	2.0	2.9	5.1
Insert/ONT-Parallel	1.5	2.4	5.1
Insert/ONT-Parallel-50	1.3	2.4	–

Tabelle 6.4: Durchschnittliche Einfügezeiten im Mehrbenutzerbetrieb (Sekunden)

Mit diesen Messungen soll geklärt werden,

- welcher Einfügealgorithmus einzusetzen ist, wenn eine bestimmte Anzahl von Benutzern parallel Dokumente einfügt, um die Antwortzeiten eines einzelnen Benutzers zu minimieren.

```
{Einfügen eines Dokuments d(Dok-ID, a_1, a_2, ..., a_n, t_1, t_2, ..., t_m)}
cBOT()→APTAID
FML-Buffer BufIn erzeugen und Dokument d in den Buffer einfügen
FML-Buffer BufOut für etwaige Rückgabewerte erzeugen
cCALL(APTAID, LockEinfügen, BufIn, BufOut)
cCALL(APTAID, MDocIns, BufIn, BufOut)
for i = 1 to m do
  FML-Buffer BufIn leeren
  Name und Wert des Attributs t_i in den FML-Buffer BufIn schreiben
  cACALL(APTAID, MExtrDes, BufIn)→ AId_i
end foroutstandingCalls=m
while outstandingCalls> 0 do
  cWAIT(ANY)→ AId
  if AId in {AId_x} (1 ≤ x ≤ m) then
    for j = 1 to Anzahl retournierter Deskriptoren step N do
      FML-Buffer BufIn leeren
      Deskriptoren des_j bis des_{j+N-1} aus BufOut nach BufIn kopieren
      cACALL(APTAID, MInsDes, BufIn)
      outstandingCalls=outstandingCalls+1
    end for
  end if
end while
FML-Buffer BufIn entfernen
FML-Buffer BufOut entfernen
cEOT(APTAID)
```

Algorithmus 23: *Insert/ONT-Parallel–N*: Applikationsseitiger Ablauf zum Einfügen eines Dokuments mit TPM/ONT mit asynchroner Ausführung der einzelnen Operationen (TPM/ONT)

```
{Einfügen eines Dokuments d(Dok-ID, a_1, a_2, ..., a_n, t_1, t_2, ..., t_m)}
cBOT()→APTAID
FML-Buffer BufIn erzeugen und Dokument d in den Buffer einfügen
FML-Buffer BufOut für etwaige Rückgabewerte erzeugen
cCALL(APTAID, LockEinfügen, BufIn, BufOut)
cCALL(APTAID, MDocIns, BufIn, BufOut)
for i = 1 to m do
  FML-Buffer BufIn leeren
  Name und Wert des Attributs t_i in den FML-Buffer BufIn schreiben
  cACALL(APTAID, MExtrDes, BufIn)→ AId_i
end for
for i = 1 to 2m do
  cWAIT(ANY)→ AId
  if AId in {AId_x (1 ≤ x ≤ m) then
    Rückagbe–Buffer von AId→BufIn
    cACALL(APTAID, MInsDes, BufIn)→ AId_{m+i}
  end if
end for
FML-Buffer BufIn entfernen
FML-Buffer BufOut entfernen
cEOT(APTAID)
```

Algorithmus 24: *Insert/ONT-Parallel*: Applikationsseitiger Ablauf zum Einfügen eines Dokuments mit TPM/ONT mit asynchroner Ausführung der einzelnen Operationen (TPM/ONT)

- mit welchem Einfügealgorithmus mit einer bestimmten Anzahl von Einfügeprozessen eine vorgegebene Menge von Dokumenten in minimaler Zeit, d.h. mit maximalem Durchsatz, in die Datenbank eingebracht werden kann.

Tabelle 6.4 zeigt die durchschnittlichen Einfügezeiten für ein Dokument mit 5, 10 und 20 parallelen Einfügeprozessen bei Verwendung von 20 Tuxedo–Serverprozessen[2]. Es fällt auf, dass die Antwortzeiten gegenüber dem Einbenutzerbetrieb nicht linear mit der Anzahl der parallel arbeitenden Benutzer ansteigen: Während im Einbenutzerbetrieb die durchschnittliche Einfügezeit für ein einzelnes Dokument mit *Insert/ONT–Sequentiell* bei rund 1.6 Sekunden liegt, steigt sie im Mehrbenutzerbetrieb mit 20 Benutzern lediglich um einen Faktor drei an.

Im Einbenutzerbetrieb ist der Transaktionsdurchsatz durch die Inverse der durchschnittlichen Einfügezeit gegeben. Im Mehrbenutzerbetrieb ist der Transaktionsdurchsatz nicht durch diese Inverse gegeben, sondern muss empirisch ermittelt werden (siehe Tabelle 6.5). Anhand der

	Benutzer		
	5	10	20
Insert/TPM-Traditionell	2.1	3.2	3.3
Insert/ONT-Sequentiell	2.1	3.1	3.4
Insert/ONT-Sequentiell-50	2.1	3.1	3.5
Insert/ONT-Parallel	2.8	3.6	3.0
Insert/ONT-Parallel-50	3.1	3.6	–

Tabelle 6.5: Durchschnittlicher Durchsatz im Mehrbenutzerbetrieb (Dokumente/Sekunde)

Messergebnisse aus den Tabellen 6.4 und 6.5 können die aufgeworfenen Fragen beantwortet werden:

- Ist die Anzahl paralleler Benutzer vorgegeben, können mit dem parallelisierenden Algorithmus *TPM/ONT-Parallel* minimale durchschnittliche Antwortzeiten erreicht werden. Eine weitere Intra–Parallelisierung ist nur dann von Vorteil, wenn wenige Benutzer parallel arbeiten.

- Soll der Transaktionsdurchsatz maximiert werden, beispielsweise wenn eine grosse Anzahl von Dokumenten einzufügen ist, muss *TPM/ONT-Parallel* mit 5 Benutzern eingesetzt werden. Dieser Algorithmus erlaubt es, rund 4 Dokumente pro Sekunde oder 15'000 Dokumente pro Stunde zu indexieren.

Zu den vorliegenden Messungen ist erklärend festgehalten:

- Der Transaktionsdurchsatz kann bei Verwendung von mehr als zehn Einfügeprozessen nur mehr unbedeutend gesteigert werden: Bei Einsatz von zehn parallelen Einfügeprozessen werden auf den Sekundärspeichermedien, auf denen die textuellen Indexstrukturen gespeichert sind, rund 60 Lese- und Schreiboperationen pro Sekunde durchgeführt. Die

[2]Bei 20 Benutzern konnten Antwortzeit und Durchsatz aufgrund Tuxedo-interner Restriktionen nicht ermittelt werden.

Externspeichermedien haben damit praktisch ihre Leistungsgrenze erreicht. Durch Hinzufügen neuer Platten kann dieser I/O–bedingte Leistungsengpass entschärft werden. Werden beispielsweise doppelt soviel Platten für die Speicherung der Indexstrukturen I_{Inhalt} und $I_{Kombiniert}$ eingesetzt, halbiert sich die Anzahl der Schreib- und Leseoperationen pro Platte. Dadurch erhöht sich der Transaktionsdurchsatz und verkürzen sich die Antwortzeiten.

- *Insert/ONT-Sequentiell* zeigt in allen Fällen geringe Antwortzeitverbesserungen gegenüber *Insert/TPM-Traditionell*. Da beide Algorithmen Dokumente nach demselben Schema einfügen, kann dieser Leistungsunterschied einerseit mit den Kosten für die Transaktionsverwaltung (Aufrufe an das XA–Interface) erklärt werden. Diese Kosten entfallen bei TPM/ONT. Daraus folgt, dass die Kosten für die Verwaltung flacher Transaktionen mittels XA–Interface grösser sind als die Kosten für die Verwaltung zweischichtiger Transaktionen. Andererseits ist die Ausführung mehrerer kurzer Transaktionen aufgrund des geringeren Verwaltungsaufwands (z.B. kleiner Sperrstrukturen) kostengünstiger als die Ausführung einer langen Transaktion.

- Nach dem Einfügen von jeweils 500'000 Tupel in die invertierten Listen der Attribute *Body* und *Kombiniert* haben die B*-Bäume über diesen eine Höhe von drei erreicht. Die Antwortzeiten erhöhen sich nachfolgend nicht mehr.

- In vielen Anwendungen ist die Wahrscheinlichkeit gering, dass zwei aufeinanderfolgende Transaktionen Daten manipulieren, die auf derselben Datenbankseite zu liegen kommen. Beim parallelen Einfügen von Dokumenten ist dies jedoch sehr wahrscheinlich, da einige Deskriptoren gemäss Zipf in fast allen Dokumenten zu finden sind: Diese Deskriptoren werden im Index über der invertierten Liste, in die sie eingefügt werden, auf derselben Datenbankseite gespeichert: Zwei parallel einzufügenden Dokumenten werden durch TPM/ONT-Text die Dokumenten-IDs i und $i+1$ zugewiesen. Für einen Deskriptor *des*, der in beiden Dokumenten im Attribut *Att* vorkommt, werden die Tupel (des, i) und $(des, i+1)$ in die invertierte Liste I_{Att} eingetragen. Diese beiden Tupel kommen im B*-Baum über I_{Att} nebeneinander zu liegen. In einem seitensperrenden Datenbanksystem kann dies zu einem sperrbedingten Leistungsengpass führen.

6.2.3 Leistungsverhalten bei verkürztem Halten von Sperren

Die bisherigen Messungen zeigen, dass durch den Einsatz einer Mehrschichtentransaktionsverwaltung wie TPM/ONT–Text die durchschnittliche Antwortzeit einer Einfügetransaktion im Ein- wie auch im Mehrbenutzerbetrieb zum Teil erheblich gesenkt werden können. Auf den Transaktionsdurchsatz hat TPM/ONT jedoch keinen grossen Einfluss. Dies hat zwei Gründe:

1. Im parallelen Betrieb mit zehn und zwanzig Benutzern sind die zur Verfügung stehenden Ressourcen (Platten) in der momentanen Systemkonfiguration vollständig ausgelastet. Eine Durchsatzerhöhung mittels Erhöhung des Intratransaktionsparallelitätsgrades ist daher nicht möglich.

2. Das in den bisherigen Messungen eingesetzte Datenbanksystem Oracle setzt zur Transaktionsverwaltung Sperren auf Tupelebene ein. Für die vorgestellten Einfügealgorithmen ist

dieses Sperrgranulat von Vorteil, da sich dadurch gleichzeitig ablaufende Transaktionen nicht gegenseitig behindern.

Da es bereits bei der einschichtigen Ausführung der Einfügealgorithmen zu keinerlei sperrbedingten Behinderungen zwischen Transaktionen kommt, kann TPM/ONT-Text von einer Verkürzung der Sperrdauer nicht profitieren.

Werden jedoch beim Einfügen eines Dokuments Statistikinformation wie Dokumentenhäufigkeiten nachgeführt, um die Suche nach Dokumenten effizienter zu unterstützen [KS95], muss auch in Oracle mit massiven Behinderungen parallel ablaufender Einfügetransaktionen gerechnet werden: Gemäss dem Zipf'schen Gesetz ist die Wahrscheinlichkeit, dass zwei Dokumente mindestens einen gemeinsamen Deskriptor aufweisen, sehr gross. Beim Warten der Dokumentenhäufigkeiten werden daher teilweise dieselben Daten geändert. Diese Behinderung wird im Rahmen der folgenden Messserie näher untersucht. Dabei wird zugrundegelegt, dass die Dokumentenhäufigkeiten in einer Relation[3]

```
DokumentenHäufigkeit_I(Des-ID NUMBER, Häufigkeit NUMBER)
```

gespeichert und im Rahmen des Applikationsservices `MInsDes` (siehe Seite 87) gewartet werden. Der hierfür neue Applikationsservice `MInsDes` wird in Abbildung 25 gezeigt.

for $i = 1$ **to** $|M|$ **do**
  INSERT INTO InvListRelation VALUES(M_i, Dok-ID)
  UPDATE DokumentenHäufigkeit
  SET Häufigkeit=Häufigkeit+1 WHERE Des-ID=Dok-ID
end for

Algorithmus 25: Relationen-Indexierung: Einfügen von Deskriptoren $\{M\}$ in eine invertierte Liste einschliesslich Nachführen von Statistikinformation

Bei Einsatz von 20 Tuxedo–Servern wird folgendes Leistungsverhalten erwartet:

- Unabhängig davon, welche mehrschichtigen Algorithmen zum Einfügen der Dokumente eingesetzt werden, sollten diese aufgrund kürzerer Sperrkonfliktzeiten in Bezug auf Durchsatz und Antwortzeit effizienter sein als der einschichtige Algorithmus *Insert/TPM–Traditionell.*

- Bei 20 Benutzern wird im Durchschnitt jedem Benutzer eine Ressource zugeteilt. Der Durchsatz für *Insert/ONT-Sequentiell* und *Insert/ONT-Parallel* sowie *Insert/ONT-Sequentiell–N* und *Insert/ONT-Parallel–N* sollte daher jeweils gleich sein. Durch den Einsatz von *Insert/ONT-Sequentiell–N* anstatt *Insert/ONT-Sequentiell* (beziehungsweise *Insert/ONT-Parallel-N* anstatt *Insert/ONT-Parallel*) sollten die Antwortzeiten reduziert und der Durchsatz erhöht werden, da die Sperren auf den Dokumentenhäufigkeiten weniger lange gehalten werden.

[3]Im Rahmen dieser Messserie wird für jede invertierte Liste I eine eigene Relation zur Speicherung der Dokumentenhäufigkeiten angelegt und gewartet.

- Bei 5, respektive 10 Benutzern erhält jeder Benutzer im Durchschnitt zwei bzw. vier Ressourcen. Die Anwendung intratransaktionsparalleler Algorithmen sollte daher vorteilhaft sein. Wie bei 20 Benutzern gilt auch hier, dass durch den Einsatz von *Insert/ONT-Parallel-N* anstatt *Insert/ONT-Parallel* ein höherer Durchsatz und geringere Antwortzeiten aufgrund kürzerer Sperrkonfliktzeiten erzielt werden sollten.

	Benutzer		
	5	10	20
Insert/TPM-Traditionell	—	—	—
Insert/ONT-Sequentiell	3.0	6.3	16.2
Insert/ONT-Sequentiell–50	3.0	5.6	11.3
Insert/ONT-Sequentiell–25	3.1	4.6	8.6
Insert/ONT-Parallel	3.7	6.4	12.0
Insert/ONT-Parallel-50	2.5	4.4	

Tabelle 6.6: Durchschnittliche Einfügezeiten im Mehrbenutzerbetrieb mit Behinderungen (Sekunden)

	Benutzer		
	5	10	20
Insert/TPM-Traditionell	—	—	—
Insert/ONT-Sequentiell	1.6	1.5	1.16
Insert/ONT-Sequentiell–50	1.5	1.7	1.70
Insert/ONT-Sequentiell–25	1.4	2.2	2.15
Insert/ONT-Parallel	1.3	1.4	1.14
Insert/ONT-Parallel-50	1.8	2.1	–

Tabelle 6.7: Durchsatz im Mehrbenutzerbetrieb mit Behinderungen (Dokumente/Sekunde)

Die Tabellen 6.6 und 6.7 zeigen die durchschnittlichen Antwortzeiten und den Transaktionsdurchsatz für den Betrieb mit fünf, zehn und zwanzig Benutzern. Für *Insert/TPM–Traditionell* konnten aufgrund der massiven gegenseitigen Behinderungen der einzelnen Transaktionen und damit langen Laufzeiten keine Ergebnisse ermittelt werden. Die angeführten Erwartungen werden durch diese Ergebnisse bestätigt; zudem kann folgendes festgestellt werden:

- Durch den Einsatz intratransaktionsparalleler Algorithmen (*Insert/ONT-Parallel-N* anstatt *Insert/ONT-Parallel*) kann selbst dann eine Antwortzeitreduktion bei gleichem Durchsatz erzielt werden, wenn alle Ressourcen dauernd belegt sind.
- Ab rund zehn einfügenden Benutzern kann der Transaktionsdurchsatz aufgrund der massiven Behinderungen durch Hinzufügen weiterer Einfügeprozesse nicht mehr gesteigert werden. Die Antwortzeiten steigen in diesem Fall erwartungsgemäss linear mit der Anzahl paralleler Benutzer.
- Das Ermitteln der Antwortzeiten und des Transaktionsdurchsatzes für *Insert/ONT-Parallel-25* ist nicht möglich, da Tuxedo die Anzahl asynchroner Service Calls pro Applikationsprozess limitiert und diese Zahl mit *Insert/ONT-Parallel-25* überschritten wird.

Zu erwarten ist jedoch, dass sowohl die durchschnittliche Antwortzeit als auch der Transaktionsdurchsatz aufgrund kürzer gehaltener Sperren auf den Statistikrelationen reduziert werden kann.

Die auftretenden Behinderungseffekte zeigen jedoch grosse Ähnlichkeit mit jenen, die beim Einsatz seitensperrender Systeme, wie beispielsweise Sybase, auftreten. Diese Behinderung, insbesondere die Behinderung von Suchtransaktionen durch Einfügetransaktionen wird im nächsten Abschnitt untersucht.

6.3 Leistungsmessung: Datenbanksystem mit Striktem Zwei–Phasen Sperrprotokoll (Sybase)

Die bisher vorgestellten Messungen wurden mit Oracle als Ressource–Manager durchgeführt. Oracle ist ein Datenbanksystem, das zur Sicherstellung der ACID–Eigenschaften eine mehrversionierende Transaktionsverwaltung auf Tupelebene einsetzt. Da in Oracle somit Einfügeoperationen nur Sperren auf jenen Objekten (Tupel) halten, welche sie gerade einfügen, treten bei diesem Datenbanksystem keine auf Sperrkonflikten basierenden Engpässe beim gleichzeitigen Einfügen mehrerer Dokumente auf. Ebenso werden reine Suchtransaktionen durch parallel ablaufende Einfügetransaktionen nicht behindert. Die mit TPM/ONT gegenüber einer einschichtigen Transaktionsverwaltung erzielten Leistungsgewinne bezüglich Antwortzeit und Durchsatz sind daher ausschliesslich auf die parallele Indexierung der einzelnen Attribute zurückzuführen.

Kommt hingegen ein Datenbanksystem zum Einsatz, welches keine Mehrversionen zur Verfügung stellt, sondern zur Transaktionskontrolle das Zwei–Phasen Sperrprotokoll einsetzt, ist mit massiven gegenseitigen Behinderungen parallel ablaufender Lese- und Einfügetransaktionen zu rechnen. In diesem Kapitel wird untersucht, wie sich ein solches Datenbanksystem, konkret *Sybase*, verhält, wenn Benutzer nach Dokumenten suchen, während andere Benutzer parallel dazu neue Dokumente in die Datenbank einfügen. Hierzu werden nach einer analytischen Untersuchung Messungen zur Klärung folgender Fragen vorgestellt:

1. Wie stark wird eine Lesetransaktionen durch eine parallel ausgeführte Einfügetransaktion behindert, wenn einerseits das einschichtige Datenbanksystem Sybase und andererseits TPM/ONT mit Sybase als Speichersystem auf Ebene L_0 eingesetzt wird?

2. Wie stark ist diese Behinderung von der Anzahl paralleler Einfügetransaktionen abhängig?

3. Wie verhält sich das System, wenn nicht nur eine, sondern mehrere Lesetransaktionen parallel zu einer variierenden Anzahl von Einfügetransaktionen verarbeitet werden?

Sybase zeigt Probleme bei der Koppelung mit dem XA–Interface (siehe auch [KS96]). Für die Messserien im Fall einschichtiger Transaktionen musste daher auf das XA–Interface verzichtet werden. Für Einfügetransaktionen heisst dies, dass für das Einfügen nur ein einziger Service notwendig ist. Im Rahmen dieses Services wird zuerst das Dokument im Rahmen einer eigenen Transaktion in die Datenbank eingefügt, anschliessend werden die einzelnen Attribute im Rahmen unabhängiger Transaktionen sequentiell indexiert.

6.3.1 Referenzmessungen: Nur Suchtransaktionen, nur Einfügetransaktionen

Um die Behinderungen von Einfügetransaktionen durch Suchtransaktionen (und umgekehrt) bewerten zu können, werden in einer ersten Messserie Antwortzeit und Durchsatz für Such- und Einfügetransaktionen im Einbenutzerbetrieb sowie im Betrieb mit fünf und zwanzig parallelen Benutzern ermittelt. Zu jedem Zeitpunkt befinden sich bei diesen Messserien jeweils nur Such- oder nur Einfügetransaktionen im System.

Suchtransaktionen

Im Rahmen einer Suchtransaktion entnimmt ein Klientenprozess eine Anfrage (=Menge von Deskriptoren) aus einer Datei und ruft einen Suchservice auf, der alle Dokumente der Relation *TP_LIB* ermittelt, welche sich auf diese Anfrage qualifizieren. Treffer einer Anfrage sind dabei jene Dokumente der Relation *TP_LIB*, welche alle Deskriptoren im Attribut *Kombiniert* enthalten. Die einzelnen Anfragen wurden vorgängig künstlich aus Artikeln der Usenet–Newsgruppe *talk.politcs.libertarian* gewonnen, indem aus den vorliegenden Artikeln einzelne Zeilen des Attributs *Inhalt* nach dem Zufallsprinzip ausgewählt wurden. Jede dieser ausgewählten Zeilen stellt eine Anfrage dar, die im Durchschnitt aus vier Deskriptoren besteht (nach Stopwortelimination und Duplikatentfernung).

Um die Treffermenge zu ermitteln, wird serverseitig im einschichtigen Fall Algorithmus 26 *(Retrieve–TPM/Traditionell)* und im mehrschichtigen Algorithmus 27 *(Retrieve–TPM/ONT)* eingesetzt. Diese beiden Algorithmen – welche in Abbildung 6.4 graphisch dargestellt sind – unterscheiden sich im wesentlichen in einem Punkt: Bei Einsatz von *Retrieve–TPM/Traditionell* wird anfangs eine L_0–Transaktion begonnen und am Schluss beendet, während bei *Retrieve–TPM/ONT* die L_0–Transaktion jeweils nach dem Lesen der Postingliste jedes einzelnen Schlüsselwortes beendet und eine neue begonnen wird. Das Lesen der Postingliste eines einzelnen Deskriptors ist somit eine Subtransaktion im Sinne der Mehrschichtentransaktionsverwaltung.

Erwartete Messergebnisse: Bei der Ausführung von Suchoperationen treten keine Sperrkonflikte zwischen den einzelnen Transaktionen auf, da von allen Transaktionen nur Lesesperren angefordert werden. Es ist daher ein leistungsmässig ähnliches Verhalten ein- und mehrschichtiger Transaktionen zu erwarten.

Tabelle 6.8 zeigt die durchschnittlichen Antwortzeiten sowie den Durchsatz für 1, 5 und 20 parallel ablaufende Suchtransaktionen unter Verwendung von *Retrieve–TPM/Traditionell* und *Retrieve–TPM/ONT*. Im Einbenutzerbetrieb liegt die durchschnittliche Antwortzeit für *Retrieve–TPM/ONT* rund 150 ms über dem Wert für *Retrieve–TPM/Traditionell*. Dies bedeutet, dass die Kosten für die Koordination von Mehrschichtentransaktionen (Starten und Beenden der L_1–Transaktion, L_1–Sperren und L_1–Logging) bei 11% liegen. Beim Einsatz von fünf bzw. zwanzig parallel ablaufenden Einfügetransaktionen steigen die Antwortzeiten sowohl beim Einsatz einer ein- als auch einer mehrschichtigen Transaktionsverwaltung um einen Faktor zwei bzw. vier an. Die für die Verwaltung der Mehrschichtentransaktionen anfallenden Kosten erhöhen sich mit steigender Benutzeranzahl nur unmerklich. Dieses Verhalten ist zu erwarten: Die kostenintensiven Anteile an diesen Mehrschichtentransaktionen sind die Aufrufe der System-Services `L1_BOT` und `L1_RBT`, sowie das Extrahieren der L_1–Prädikatsperren. Diese Kosten sind von der Anzahl parallel laufender Transaktionen unabhängig. Im Mehrbenutzerbetrieb muss zudem geprüft werden, ob die angeforderte Sperre überhaupt erteilt werden kann.

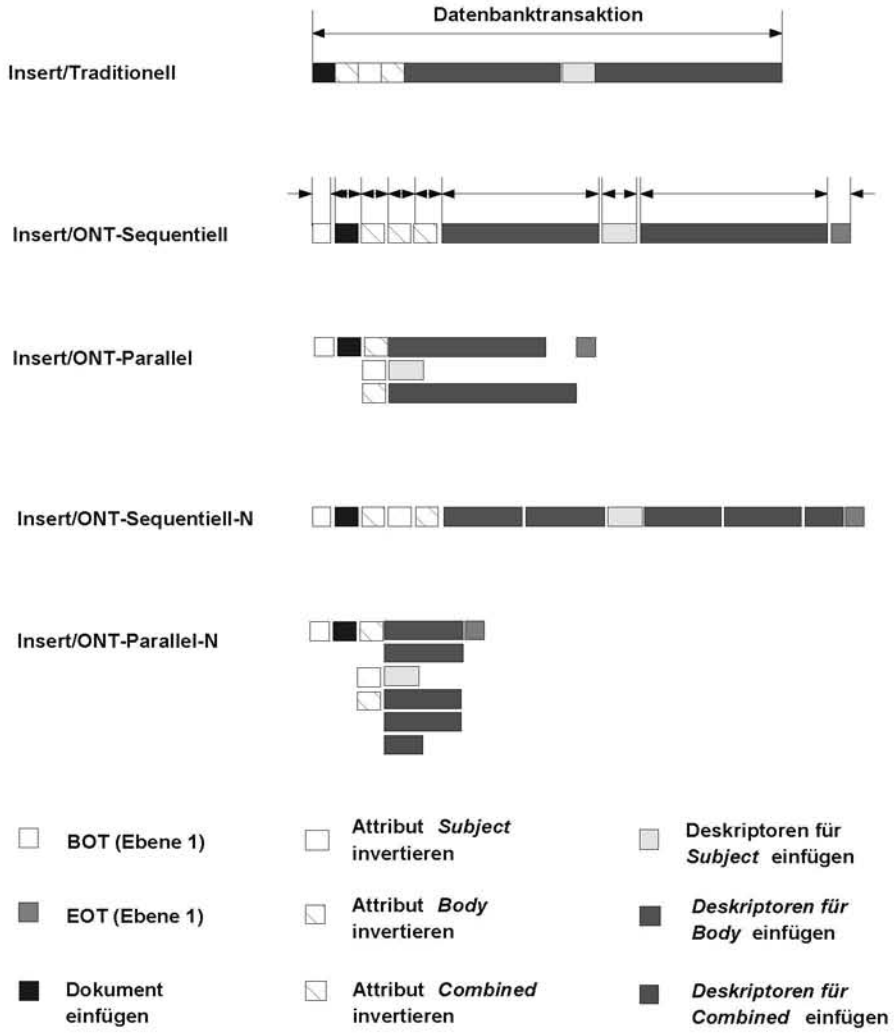


Abbildung 6.3: Eingesetzte Einfügealgorithmen

	Benutzer					
	1		5		20	
	Antw	Durch	Antw	Durch	Antw	Durch
Retrieve/Traditionell	1.30	0.77	2.18	2.25	6.42	3.02
Retrieve/ONT	1.45	0.69	2.31	2.10	6.77	2.87

Tabelle 6.8: Suche von Dokumenten: Durchschnittliche Antwortzeit (in Sekunden) und Durchsatz (Dokumente/Sekunde)

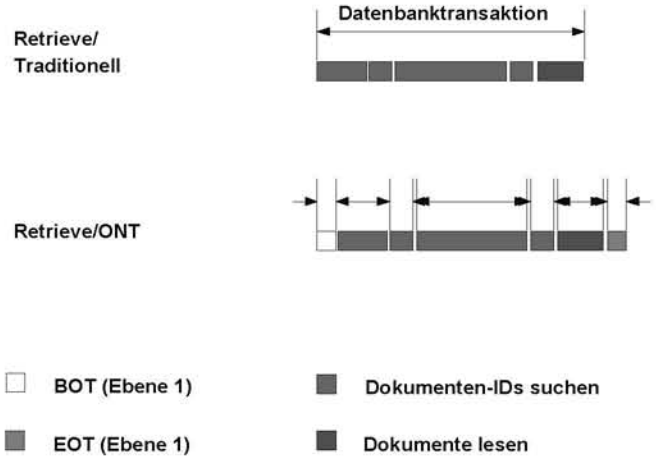


Abbildung 6.4: Eingesetzte Suchalgorithmen

Suchen aller Dokumente, welche die Wörter $W = \{w_1, w_2, \ldots, w_n\}$ im Attribut X enthalten
Wortstammreduktion durchführen $W \rightarrow W$
Entfernen der Stopworte aus $W \rightarrow W$
Entfernen der Duplikate aus $W \rightarrow W$
Begin L_0**-*Transaction***
for all w **in** {W} **do**
  SELECT Des-ID INTO *Dok-ID* FROM Mapping WHERE Des=w
  if keine Des-ID für d vorhanden **then**
    Suche beenden, da sich kein Dokument qualifizieren kann
  end if
  Des-ID=Des-ID∪Dok-ID
end for
result={}
for i=1 **to** |Des-ID| **do**
  SELECT Dok-ID INTO tmp FROM IL_X WHERE Des-ID=d
  if i=1 **then**
    result=tmp
  else
    result=∩tmp
  end if
end for
Commit L_0**-*Transaction***

Algorithmus 26: *Retrieve/Traditionell:* Serverseitiger Algorithmus zum Suchen von Dokumenten in einer einschichtigen Datenbankumgebung (Tuxedo ohne TPM/ONT)

```
Suchen aller Dokumente, welche die Wörter W = {w_1, w_2, ..., w_n} im Attribut X enthalten
Wortstammreduktion durchführen W → W
Entfernen der Stopworte aus W → W
Entfernen der Duplikate aus W → W
Begin L_1-Transaction
Begin L_0-Transaction
for all w in {W} do
  SELECT Des-ID INTO did FROM Mapping WHERE Deskriptor=w
  if keine Des-ID für d vorhanden then
    Suche beenden, da sich keine Dokument qualifizieren kann
  end if
  Des-ID=Des-ID∪did
end for
Commit L_0-Transaction
L_1-Sperre für Suchoperation anfordern
result={}
for i=1 to |Des-ID| do
  Begin L_0-Transaction
  SELECT Dok-ID INTO tmp FROM IL_X WHERE Des-ID=d
  Commit L_0-Transaction
  if i=1 then
    result=tmp
  else
    result=∩tmp
  end if
end for
Commit L_1-Transaction
```

Algorithmus 27: *Retrieve/ONT:* Serverseitiger Algorithmus zum Suchen von Dokumenten in einer mehrschichtigen Datenbankumgebung (Tuxedo mit TPM/ONT)

Dieser Test ist im vorliegenden Fall kostengünstig, da zu jedem Zeitpunkt nur L_1–Operationen ausgeführt werden, welche miteinander kompatibel sind (d.h. ein + in der Konfliktmatrix aufweisen). Die Kompatibilität von Suchoperationen kann bereits anhand der Konfliktmatrix festgestellt werden. Die Durchführung des kostenintensiven Prädikattests, d.h. eines Tests, ob eine Sperre auf Objekte angefordert wurde, auf die bereits eine andere Transaktion eine unverträgliche Sperre hält, kann daher entfallen.

Einfügetransaktionen

In einer zweiten Referenz–Messserie wird das Verhalten des Datenbanksystems bei der Ausführung von Einfügeoperationen untersucht. Gemessen werden die durchschnittlichen Antwortzeiten sowie der Durchsatz im Einbenutzerbetrieb sowie im Betrieb mit 5 und 20 parallelen Einfügetransaktionen.

Für das Einfügen eines Dokuments kommen die bereits im letzten Kapitel vorgestellten Einfügealgorithmen *Insert/Traditionell*, *Insert/ONT-Sequentiell*, *Insert/ONT-Sequentiell–N* mit $N = 25$ und $N = 10$ sowie *Insert/ONT-Parallel* und *Insert/ONT-Parallel–N* mit $N = 25$ zum Einsatz (Algorithmen 20, 21, 22, 23, 24). Folgende Einschränkungen müssen dabei beachtet werden: Tuxedo beschränkt intern die Anzahl asynchroner Serviceaufrufe pro Klientenprozess. Beim Einsatz von *Insert/ONT-Parallel–25* wird daher nicht pro 25 Deskriptoren ein eigener

asynchroner Serviceaufruf ausgeführt, sondern es erfolgt pro Textattribut *ein* Serviceaufruf des Services *Insert/ONT-Parallel*, wobei jeweils 25 Deskriptoren im Rahmen einer eigenständigen L_0–Transaktion in die invertierte Liste eingefügt werden. Eine weitere Tuxedo–Restriktion erlaubt es zudem nicht, für *Insert/ONT-Parallel–N* Messungen für mehr als 10 Einfügetransaktionen durchzuführen.

Analog zum Einfügen im Fall von Oracle unter gleichzeitigem Nachführen von Statistikinformation wird folgendes Verhalten erwartet:

- Im Einbenutzerbetrieb sollten sich alle Algorithmen ungefähr gleich verhalten, da es zu keinen Konflikten auf Ebene L_0 kommt und — abgesehen von Commits auf Ebene L_0 im Fall von TPM/ONT — derselbe Code ausgeführt wird.
- Im Mehrbenutzerbetrieb ist beim Einsatz von *Insert/Traditionell* aufgrund der starken Behinderungen auf den invertierten Listen mit einem praktisch linearen Anstieg der Antwortzeiten bei gleichbleibendem Durchsatz zu rechnen.
- Im Mehrbenutzerbetrieb darf mit einer Verbesserung des Durchsatzes gerechnet werden, wenn anstatt *Insert/Traditionell* der Algorithmus *Insert/ONT-Sequentiell* eingesetzt wird, da dieser jedes Textattribut im Rahmen einer Transaktion indexiert. Im konkreten Fall sollte sich der Durchsatz verdoppeln, da sich jedes Dokument aus zwei praktisch gleich langen Volltextattributen (Inhalt und Kombiniert) zusammensetzt und die Sperrdauer durch die Verwendung von *Insert/ONT-Sequentiell* auf den Textindexstrukturen I_{Inhalt} und $I_{Kombiniert}$ damit halbiert wird.
- Durch den Einsatz von *Insert/ONT-Sequentiell–N* anstatt *Insert/ONT-Sequentiell* sollte sich die durchschnittliche Sperrdauer auf den Textindexstrukturen weiter verkürzen. Unter der Annahme, dass jedes Textattribut rund 80 Deskriptoren enthält, reduziert sich die Sperrdauer im Fall von $N = 25$ um rund 70% und für $N = 10$ um durchschnittlich 90%. Mit einer Reduzierung der Antwortzeiten im gleichen Ausmass darf jedoch aufgrund der erhöhten Kosten für die Ausführung mehrschichtiger Transaktionen nicht gerechnet werden.
- Bei Verwendung der parallelen Einfügealgorithmen *Insert/ONT-Parallel* und *Insert/-ONT-Parallel–N* ist mit einer weitere Verkürzung der Antwortzeiten zu rechnen. Diese hat ihre Ursache im höheren Grad an Intratransaktionsparallelität, welche mit einer verkürzten Sperrdauer auf Ebene L_1 verbunden ist.

Tabelle 6.9 zeigt die Resultate der durchgeführten Messungen. Die Ergebnisse für *Insert/Traditionell* und *Insert/ONT-Sequentiell* entsprechen den vorgestellten Erwartungen. Unerwartet hingegen sind die Ergebnisse für *Insert/ONT-Sequentiell–N*: Für $N = 25$ wird noch eine Verbesserung von rund 22% gegenüber *Insert/ONT-Sequentiell* erreicht. Für $N = 10$ hingegen verschlechtern sich Antwortzeit und Durchsatz. Dies ist auf die hohen Kosten für die Durchführung der Commit–Operationen auf Ebene L_0 zurückzuführen. Aufgrund der schlechten Ergebnisse für *Insert/ONT-Sequentiell–10* wird dieser Algorithmus mit $N = 10$ nachfolgend nicht weiter betrachtet.

Ein weiteres interessantes Ergebnis stellen die Werte für *Insert/ONT-Parallel* dar: Erwartungsgemäss verbessern sich im Einbenutzerbetrieb Antwortzeiten und Durchsatz. Bei Erhöhung

	Benutzer					
	1		5		20	
	Antw	Durch	Antw	Durch	Antw	Durch
Insert/Traditionell	5.8	0.17	27	0.18	102	0.19
Insert/ONT-Sequentiell	5.9	0.17	21	0.46	38	0.46
Insert/ONT-Sequentiell-25	6.3	0.15	16	0.59	30	0.63
Insert/ONT-Sequentiell-10	9.2	0.11	30	0.32	52	0.36
Insert/ONT-Parallel	3.0	0.31	10	0.45	39	0.45
Insert/ONT-Parallel-25	3.4	0.27	8	0.58	—	—

Tabelle 6.9: Einfügen von Dokumenten: Durchschnittliche Antwortzeit (in Sekunden) und Durchsatz (Dokumente/Sekunde)

auf 5 parallele Einfüger kommt es jedoch nur noch zu einer Verbesserung der Antwortzeiten verglichen mit ihren sequentiellen Äquivalenten *Insert/ONT-Sequentiell* und *Insert/ONT-Sequentiell–25*: Eine Durchsatzverbesserung kann aufgrund der limitierten Ressourcen, insbesondere der Sybase–Server (=8), nicht mehr erreicht werden.

Bei Einsatz der parallelen Einfügealgorithmen *Insert/ONT-Parallel* und *Insert/ONT-Parallel-25* ist eine weitere Reduktion der durchschnittlichen Antwortzeit möglich. Diese Reduktion hat — verglichen mit *Insert/ONT-Sequentiell* und *Insert/ONT-Sequentiell–25* — jedoch keinen Einfluss auf den Durchsatz, da bereits bei Einsatz von *Insert/ONT-Sequentiell–25* die vorhandenen Sybase–Ressourcen ausgeschöpft sind.

6.3.2 Leistungsverhalten bei gleichzeitiger Ausführung von Einfüge- und Lesetransaktionen

Wie bereits mehrfach erwähnt, werden in einem Datenbanksystem in der Regel Daten gelesen, während gleichzeitig neue Daten hinzugefügt werden. Dabei ist aufgrund gemeinsam benötigter Information (Textindizes) mit Behinderungen zu rechnen. Mit den nachfolgenden Messungen wird untersucht, wie stark sich diese Behinderungen auf parallel ablaufende Such- und Einfügetransaktionen auswirken. Die Messungen gliedern sich dabei in zwei Teile: In einem ersten werden die Behinderungen einer einzelnen Suchtransaktion in Abhängigkeit von der Anzahl paralleler Einfügetransaktionen untersucht, im zweiten wird ein Szenario mit einer variablen Anzahl paralleler Such- und Einfügetransaktionen untersucht.

Mehrere Einfügetransaktionen parallel zu einer Suchtransaktion

Bei Suchtransaktionen, die parallel zu einer Einfügetransaktion ausgeführt werden, muss mit zwei Arten von Behinderungen gerechnet werden: Zum einen mit "echten" Konflikten, wenn sich ein einzufügendes Dokument auf die Suchanfrage qualifiziert, zum anderen mit Pseudokonflikten, wenn das einzufügende Dokument einige, aber nicht alle Suchdeskriptoren enthält. Während bei echten Konflikten eine der beiden Transaktionen das Ende der anderen zwecks Sicherung der Integrität abwarten muss, ist eine Verzögerung im Fall von Pseudokonflikten

unnotwendig, in einschichtigen Datenbanksystemen mit Striktem Zwei–Phasen–Sperrprotokoll allerdings nicht zu vermeiden.

Im Rahmen dieser Messreihe wird die Behinderung einer einzelnen Suchtransaktion durch Einfügetransaktionen untersucht. Zu diesem Zweck werden Experimente mit drei Messanordnungen, *Mixed–Traditionell* und *Mixed–TPM/ONT*, durchgeführt:

- *Mixed–Traditionell* ermittelt das Verhalten des Datenbanksystems bei Einsatz einer einschichtigen Transaktionsverwaltung. Für die Suche von Dokumenten wird *Retrieve/Traditionell* und für das Einfügen *Insert/Traditionell* eingesetzt (Algorithmen 26 und 20).
- *Mixed–TPM/ONT-Sequentiell-25* untersucht das Verhalten des Datenbanksystems bei Einsatz einer mehrschichtigen Transaktionsverwaltung. Hierbei wird *Retrieve/ONT* für die Dokumentensuche und *Insert/ONT-Sequentiell–N* mit $N = 25$ für das Einfügen von Dokumenten verwendet (Algorithmen 27 und 22).
- *Mixed–TPM/ONT-Parallel-25* setzt für die Untersuchung des Systemverhaltens im Mehrschichtbetrieb *Retrieve/ONT* für die Dokumentensuche und *Insert/ONT-Parallel–N* mit $N = 25$ für das Einfügen von Dokumenten (Algorithmen 27 und 24) ein.

Für alle Messanordnungen werden Messungen für 1, 5 und 20 parallel zur Suchtransaktion ablaufende Einfügetransaktionen durchgeführt. Gemäss den in Abschnitt 3.3 aufgestellten analytischen Überlegungen ist mit einer deutlichen Reduktion der Suchzeiten von *Mixed–TPM/ONT-Sequentiell-25* und *Mixed–TPM/ONT-Parallel-25* gegenüber *Mixed–Traditionell* zu rechnen.

Tabelle 6.10 fasst die Messergebnisse für die Dokumentensuche für *Mixed–Traditionell*, *Mixed–TPM/ONT-Sequentiell-25* und *Mixed–TPM/ONT-Parallel-25* zusammen. Im Fall einer einzel-

	Einfügetransaktionen		
	1	5	20
Mixed–Traditionell	1.35	2.54	12.29
Mixed–TPM/ONT-Sequentiell-25	1.53	2.35	5.08
Mixed–TPM/ONT-Parallel-25	2.25	3.2	—

Tabelle 6.10: Gemischter Such-/Einfüge–Betrieb: Durchschnittliche Antwortzeit für das Suchen von Dokumenten (in Sekunden)

nen Einfügetransaktion parallel zu einer Suchtransaktion liegt die mittlere Suchzeit bei Einsatz von TPM/ONT um 13% höher als bei Einsatz einer einschichtigen. Dieses Ergebnis hat zwei Ursachen: Zum einen ist die Wahrscheinlichkeit eines Pseudokonflikts gering, zum anderen ist die Ausführung der Mehrschichtentransaktionen im Gegensatz zu den vorangegangenen Messungen mit Kosten verbunden, die höher als der Nutzen sind: Werden — wie in Abschnitt 6.3.1 — zu jedem Zeitpunkt jeweils nur Such- oder nur Einfügetransaktionen ausgeführt, muss bei Anforderungen eines Sperrprädikats *kein* aufwendiger Prädikattest durchgeführt werden, da alle L_1–Operationen miteinander kommutieren. Im Fall einer gleichzeitigen Verarbeitung von Such- und Einfügetransaktionen muss dieser Prädikattest durchgeführt werden. Die Kosten dieser Operation, zusammen mit denen für das Starten und Beenden der L_1–Transaktionen, sind in Summe höher als der Gewinn, welcher aus der Vermeidung von Pseudokonflikten resultiert.

Wird die Anzahl paralleler Einfügetransaktionen erhöht, kann durch den Einsatz von TPM/-ONT eine Leistungsverbesserung erzielt werden, da mit der Erhöhung der Anzahl paralleler Einfügetransaktionen gleichzeitig die Pseudokonfliktwahrscheinlichkeit ansteigt. Bereits bei fünf parallelen Einfügetransaktionen halten sich die Kosten für die Ausführung mehrschichtiger Transaktionen und deren Nutzen verglichen mit der einschichtigen Transaktionsverwaltung die Waage. Bei einer weiteren Erhöhung übersteigt der Nutzen deutlich deren Kosten. Auffallend ist, dass durch *Mixed–TPM/ONT-Parallel-25* keine Leistungsverbesserung mehr möglich ist. Dies hat seine Ursache im erhöhten Ressourcenverbrauch. Die vorhandenen Sybase–Server werden stärker belastet als bei Einsatz von *Mixed–TPM/ONT-Sequentiell-25*. Daher steht weniger Rechenleistung für die Verarbeitung von Anfragen zur Verfügung; die Antwortzeiten verschlechtern sich.

Zusammenfassend kann festgehalten werden, dass der Einsatz von Mehrschichtentransaktionen für das Einfügen und Suchen von Dokumenten im Mehrbenutzerbetrieb mit vielen Benutzern zu einer deutlichen Reduktion der Antwortzeiten der Suchtransaktionen führt.

Mehrere Einfügetransaktionen parallel zu mehreren Suchtransaktionen

In einer letzten Messreihe wird untersucht, wie sich Antwortzeiten und Durchsatz von Such- und Einfügetransaktionen in einem realen Anwendungsszenario mit jeweils *mehreren* parallelen Such- und Einfügetransaktionen verhalten. Als Anwendungsbeispiel wird ein Informationssystem betrachtet, welches wiederum die Verwaltung von und Suche nach Usenet–Artikeln erlaubt. Dieses System wird unter folgenden Randbedingungen betrieben:

- Zu jedem Zeitpunkt ist es maximal $max_{Benutzer}$ Benutzern gleichzeitig möglich, mit dem System zu arbeiten, d.h. der maximale Grad der Intertransaktionsparallelität (max_{Inter}) beträgt $max_{Benutzer}$.
- Neu eintreffende Artikel sollen den Benutzern so schnell wie möglich zur Verfügung gestellt werden. Treffen zu einem Zeitpunkt mehrere Artikel ein, werden diese parallel eingefügt.
- Im vorliegenden Informationssystem steht die möglichst schnelle Bedienung von Suchtransaktionen im Vordergrund. Suchtransaktionen werden daher vom System bevorzugt behandelt. Konkret heisst dies, dass die maximale Anzahl paralleler Einfügetransaktionen $current_{Einfüger}$ von der Anzahl der zu einem Zeitpunkt zu bedienenden Suchtransaktionen $current_{Sucher}$ abhängig ist und vom System automatisch auf

  $$current_{Einfüger} = max_{Inter} - current_{Sucher}$$

  festgelegt wird.

TPM/ONT erlaubt in der momentanen Implementierung keine automatische Zulassungsbeschränkung einzelner Transaktionstypen, wie sie für die Erfüllung der letztgenannten Anforderung notwendig wäre. Dieses Verhalten muss daher in den nachfolgenden Messungen künstlich nachgebildet werden. Hierzu werden separate Messungen für

$$max_{Inter} = max_{Benutzer} = 25$$

für folgende vier Fälle durchgeführt:

- 5 Suchtransaktionen parallel zu 20 Einfügetransaktionen
- 10 Suchtransaktionen parallel zu 15 Einfügetransaktionen
- 15 Suchtransaktionen parallel zu 10 Einfügetransaktionen
- 20 Suchtransaktionen parallel zu 5 Einfügetransaktionen

Für jeden dieser Fälle werden Messungen im Einschicht-, wie im Mehrschichtbetrieb durchgeführt. Dabei kommen wiederum die in Abschnitt 6.3.2 beschriebenen Messanordnungen *Mixed–Traditionell*, *Mixed–TPM/ONT-Sequentiell-25* und *Mixed–TPM/ONT-Parallel-25* zum Einsatz. Erwartet werden Ergebnisse, wie sie bereits in Abschnitt 6.3.2 erzielt wurden.

Tabelle 6.11 zeigt die Messergebnisse der durchgeführten Experimente. Hervorzuheben ist im einschichtigen Fall besonders das schlechte Abschneiden der Einfügetransaktionen. Vergleicht man diese Zahlen mit denen aus Tabelle 6.9 (nur Einfügetransaktionen), stellt man fest, dass der Transaktionsdurchsatz der Einfügetransaktionen durch die parallel laufenden Suchtransaktionen deutlich reduziert wird. Eine Erklärung hierfür ist ansatzweise in [Syb93, Kapitel 13: Locking] zu finden:

> *Demand locks prevent any more shared locks from being set. SQL Server sets a demand lock to indicate that a transaction is next in line to lock a table or page. This avoids situations in which read transactions acquire overlapping shared locks, monopolizing a table or page, so that a write transaction waits indefinitely for its exclusive lock.*
>
> *After waiting on several different read transactions, SQL Server gives a Demand lock to the write transaction. As soon as the existing read transactions finish, the write transaction is allowed to proceed. Any new read transactions must then wait for the write transaction to finish, when its exclusive lock is released*

Dies bedeutet, dass Transaktionen Leseperren (SL) gemäss Abbildung 6.5 anfordern können und diese auch erhalten: T_1 erhält eine Lesesperre auf x. T_2 fordert eine Schreibsperre (XL) auf x und wird verzögert. T_3 fordert anschliessend eine Lesesperre auf x und erhält diese, obwohl T_2 bereits eine Schreibsperre angefordert hat. Anschliessend wird T_1 beendet. T_2 versucht nun erneut, die Sperre auf x zu erwerben, erhält sie jedoch nicht, da diese jetzt von T_3 gehalten wird. Nach Gewährung einer weiteren Lesesperre auf x durch T_4 fordert Sybase ein *Demand Lock (DL)* auf x für T_2 und verhindert somit, dass weitere Lesesperren auf x an andere Transaktionen (z.B. T_5) gewährt werden. Nach Abschluss von T_3 wird dadurch die Schreibsperre auf x an T_2 gewährt. Anhand dieses Beispiels wird deutlich, dass es bei Einsatz einer einschichtigen Transaktionsverwaltung aufgrund überlappender Gewährung von Lesesperren zum ansatzweisen Verhungern der Einfügetransaktionen und dadurch zur Bevorzugung der Lesetransaktionen kommen kann.

Kommt die mehrschichtige Transaktionsverwaltung von TPM/ONT zum Einsatz, ist die Wahrscheinlichkeit überlappender Lesesperren aufgrund der kürzeren Sperrdauer auf Ebene L_0 geringer. Der Leistungsengpass kann daher durch den Einsatz der mehrschichtigen Transaktionsverwaltung TPM/ONT vermindert werden. Dadurch verbessern sich die Antwortzeiten der Einfügetransaktionen im Mittel um einen Faktor 5–8, während der Durchsatz in ähnlichem Masse zunimmt. Die Antwortzeiten für Lesetransaktionen steigen dabei mit 15% verhältnismässig gering an.

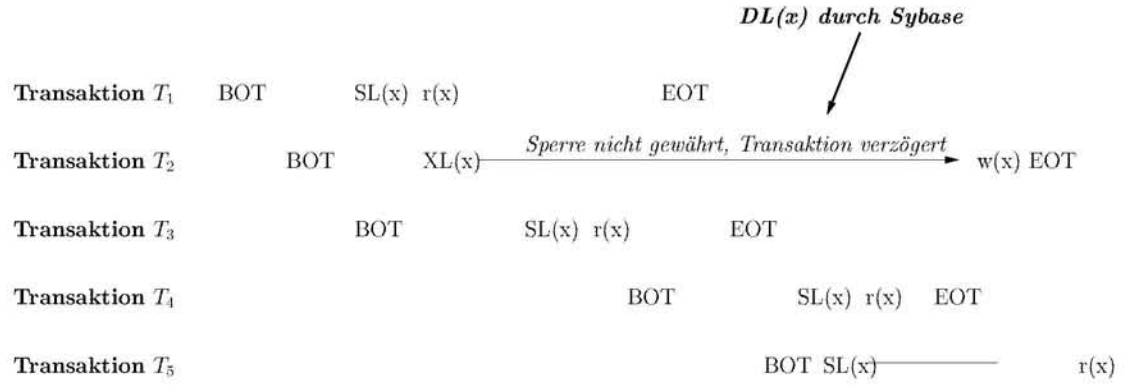


Abbildung 6.5: Verhungern von Einfügetransaktionen

Zusammenfassend kann festgehalten werden, dass durch den Einsatz einer mehrschichtigen Transaktionsverwaltung das Gesamtverhalten eines Informationssystems zur Verwaltung und Suche von Volltextdaten in einer Mehrprozessorumgebung gegenüber einer einschichtigen Transaktionsverwaltung deutlich verbessert werden kann.

6.4 Vergleich mit einem transaktionslosen Suchsystem

Die vorliegenden Messergebnisse könnten auf den ersten Blick den Eindruck erwecken, dass durch den Einsatz einer mehrschichtigen Transaktionsverwaltung sämtliche Leistungsengpässe eines Datenbanksystems beseitigt werden und insbesondere ein ähnlich effizientes Arbeiten wie mit einem transaktionslosen Informationssystem, das meist nur statische Dokumentenkollektionen handhabt, möglich ist. Dies ist natürlich nicht der Fall: Der Einsatz eines transaktionsorientierten anstatt eines transaktionslosen Systems hat eine Reihe klarer Vorteile. Beispiele hierfür sind die Möglichkeit, im laufenden Betrieb neue Dokumente einzufügen, während gleichzeitig andere Benutzer nach Dokumenten suchen, die garantierte Dauerhaftigkeit von Daten, die in vielen Anwendungen von grosser Bedeutung ist oder eine klar definierte Zugriffskontrolle. Diese Vorteile haben jedoch auch ihren Preis. So müssen für die Textindexierung Datenstrukturen eingesetzt werden, die ein Ändern im laufenden Betrieb erlauben (und die aufgrund ihrer Struktur nicht so effizient sind wie "read-only" Indizes), mit Hilfe von Sperren muss der konkurrierende Zugriff von Benutzern auf Daten geregelt, Logsätze zur Garantie der Dauerhaftigkeit müssen geschrieben und Zugriffsberechtigungen geprüft werden. Dieser "Luxus" verursacht nicht zu vernachlässigende Kosten, die bei Einsatz transaktionsloser Informationssysteme nicht anfallen.

Die Erörterung der Frage, wie hoch die Kosten für den Einsatz eines transaktionsorientierten Systems gegenüber einem transaktionslosen sind, ist nicht eigentliche Aufgabe dieser Arbeit, da von der Grundvoraussetzung ausgegangen wurde, dass ein transaktionsorientiertes System eingesetzt werden muss, um die speziellen Bedürfnisse der Benutzer — beispielsweise das Einfügen neuer Dokumente im laufenden Betrieb oder die Dauerhaftigkeit der Daten — befriedigen zu können. Die Verwendung eines transaktionsorientierten Systems stand daher immer ausser Frage.

Nichtsdestotrotz ist die Frage, wie hoch die effektiven Kosten der transaktionsorientierten Verwaltung von Dokumenten sind, natürlich einerseits berechtigt und anderseits auch sehr interessant und soll daher nachfolgend zumindest in theoretischer Form andiskutiert werden. Hierzu wird ein einfacher Leistungsvergleich zwischen TPM/ONT-Text und *Altavista*, der heute wohl populärsten Internet Search-Engine, angestellt. Bei diesem Vergleich muss man sich jedoch immer bewusst sein, dass es sich bei TPM/ONT, wie es in dieser Arbeit realisiert und betrieben wurde, und Altavista, wie es auf dem Internet verfügbar ist, um sehr unterschiedliche Systeme handelt, was schon allein die nachfolgende Liste zeigt:

- *Die Suchmodelle sind unterschiedlich:* TPM/ONT setzt ein boolesches Suchmodell ein und erlaubt keine Proximity-Suche, während Altavista gewichtetes Retrieval mit Proximity-Suche unterstützt.

- *Die Hardware ist unterschiedlich:* Für die Ermittlung der Messergebnisse für TPM/ONT wird ein SparcCenter 2000 von SUN mit 10 Sparc–Prozessoren eingesetzt. Die Prozessoren sind mit 40 Megahertz getaktet. Das System verfügt über 300 Megabyte realen Hauptspeicher. Für die Speicherung der Textindexstrukturen werden drei Festplatten eingesetzt. Die maximale kumulierte Datentransferrate dieser Festplatten über den Bus in den Hauptspeicher liegt bei unter 5 Megabyte pro Sekunde. Die eingesetzte Hardware eignet sich gut für *vergleichende* Messungen. Für die Verarbeitung von Datenmengen im Bereich mehrerer Gigabyte ist der verwendete Rechner einerseits unterdimensioniert (RAM, Anzahl Festplatten), andererseits entspricht er aufgrund der rasanten technologischen Entwicklung nicht mehr dem Stand der Technik (insbesondere die CPU-Leistung).

  Altavista setzt die modernste heute von DEC erhältliche Hardware ein: Für die Anfrageauswertung wird ein AlphaServer 8400 5/300 mit 10 Alpha-Prozessoren (getaktet mit 437 Megahertz) und 6 Gigabyte Hauptspeicher eingesetzt. Die textuellen Indexstrukturen werden in einem RAID gespeichert. Der leistungsstarke Bus des AlphaServer 8400 erlaubt es, Daten mit bis zu 1200 Megabyte pro Sekunde vom angeschlossenen RAID in den Hauptspeicher zu transferieren. Für die Beantwortung von Anfragen wird eine von DEC nicht näher genannte Zahl solcher AlphaServer eingesetzt.

  Bemerkt werden muss, dass diese Server nur für die Auswertung von Anfragen mittels Schlüsselwörter eingesetzt werden. Will der Benutzer im Anschluss an eine Suche auf Daten (beispielsweise die Artikel einer Newsgruppe) zugreifen, werden diese von einem unabhängigen Server aufbereitet und an den Benutzer transferiert.

- *Die Aktualität der Daten ist unterschiedlich*: TPM/ON unterstützt das Einfügen von Dokumenten im laufenden Betrieb. Neue Dokumente sind somit sofort verfügbar.

  Altavista baut den Textindex mittels separater Rechner offline auf und stellt den neuen Textindex den Search Engines einmal täglich zur Verfügung. In Altavista sind daher nur tagesaktuelle Dokumente verfügbar.

Ein aussagekräftiger Vergleich der beiden Systeme ist aufgrund der obigen Punkte sowie der Tatsache, dass über die Intera von Altavista nur sehr wenig Information öffentlich verfügbar ist, nicht ohne weiteres möglich. Es kann daher kein einfacher quantitativer Vergleich der beiden Systeme mit absoluten Zahlen angestellt werden. Vorgestellt wird daher eine Reihe grundsätzlicher Überlegungen bezüglich des Leistungsverhaltens unter der Annahme, dass TPM/ONT eine ähnlich leistungsstarke Hardware wie Altavista zur Verfügung steht.

Die unterschiedlichen Anforderungen an die Aktualität von Daten hat direkten Einfluss auf die Wahl der eingesetzten Indexstrukturen. TPM/ONT bietet aktuelle Daten an und setzt als Textindexstruktur invertierte Listen ein. Diese Indexstruktur erlaubt ein effizientes Indexieren neuer Dokumente im laufenden Betrieb. Nachteil dieser Indexstruktur ist, dass sie mit 8 Byte pro Posting viel Speicherplatz benötigen. Werden invertiert Listen physisch in einem B*-Baum gespeichert, sind die Blätter dieses B*-Baums im Durchschnitt zu $\frac{2}{3}$ gefüllt, respektive zu $\frac{1}{3}$ leer. Rechnet man diesen freien Platz in die Länge eines einzelnen Postings mit ein, werden für dessen Speicherung rund 10.5 Bytes ($= l_{Eintrag}^{TPM/ONT}$) benötigt.

Über den in Altavista verwendeten Textindex ist nur wenig Information öffentlich verfügbar. Die Grösse eines Postings muss daher wie folgt approximativ bestimmt werden: Altavista indexiert 30 Millionen Dokumente mit einem Gesamtumfang von rund 100 Gigabyte Information und erlaubt es, komplexe Anfragen wie "Wort X nahe bei Wort Y" zu stellen. Es muss daher jedes Wort in diesen 100 Gigabyte einzeln identifizierbar sein. Für die eindeutige Identifikation jedes einzelnen Worts in einer 100 Gigabyte grossen Datenkollektion werden $\lceil \log_2(10^{11}) \rceil = 37$ Bits benötigt. Mit grosser Wahrscheinlichkeit werden diese Postings in Form von BLOBs gespeichert, wie sie in Abschnitt 2.3.2 (Seite 20) diskutiert wurden. Unter der Annahme, dass die Postings im BLOB um 25% auf je 28 Bits=3.5 Byte ($= l_{Eintrag}^{Altavista}$) komprimiert werden können, benötigt Altavista für die Speicherung eines Postings nur $\frac{1}{3}$ des Speicherplatzes von TPM/ONT-Text.

Der Textindex von Altavista hat einen Umfang von rund $3 * 10^{10}$ Byte (=30 Gigabye). Unter der Annahme, dass in allen Dokumenten rund 10^6 *verschiedene* Wörter auftreten und diese — entgegen der Zipf'schen Verteilung vereinfachend angenommen — gleichverteilt auftreten, hat eine Postingliste zu einem Wort $\frac{3*10^{10}}{10^6} = 3 * 10^4$ Einträge. Bei einem Speicherplatzbedarf von 3.5 Byte pro Eintrag in der Postingliste liegt die Länge einer Postingliste in Altavista bei rund 10^5 Bytes ($= l_{Liste}^{Altavista}$).

TPM/ONT-Text unterstützt im Gegensatz zu Altavista nur sehr einfache Anfragen. Wörter, die mehrmals in einem Dokument vorkommen, werden nur einmal im Textindex gespeichert. Wird vereinfachen angenommen, dass jedes Wort in einem Dokument zweimal vorkommt, bedeutet dies, dass in einer Postingliste zu einem Wort in Altavista doppelt soviele Postings zu finden sind wie in TPM/ONT. Berücksichtigt man diese, hat eine Postingliste in TPM/ONT nur $1.5 * 10^4$ Einträge und die Länge der Postingliste in TPM/ONT-Text beträgt $1.5 * 10^4 * l_{Eintrag}^{TPM/ONT} \approx 1.6 * 10^5$ Byte ($= l_{Liste}^{TPM/ONT}$).

Die Zeit zur Beantwortung einer Benutzeranfrage durch Auswertung eines Textindex wird heute durch jene Zeit dominiert, die für den Transfer der Textindexstrukturen vom Sekundärspeicher in den Hauptspeicher benötigt wird. Mit CPU-Engpässen ist aufgrund hoher Taktraten und dem Einsatz von Mehrprozessormaschinen nicht zu rechnen.

Unter Zuhilfenahme der bisher gemachten Annahmen ergibt sich folgende Abschätzung des Antwortzeitverhältnisses der beiden Systeme für die Beantwortung einfacher Anfrage der Form *"Gesucht werden alle Dokumente, welche eine Menge von Schlüsselwörter A, B, C, ..." enthalten*": Postinglisten sind in TPM/ONT um rund 60% ($k_{Liste}^{Mehraufand} = l_{Liste}^{TPM/ONT} / l_{Liste}^{Altavista}$) länger als in Altavista. Die für den reinen Transfer einer einzelnen Postingliste vom Sekundär- in den Primärspeicher in TPM/ONT benötigte Zeit ist daher um ebenfalls 60% höher als jene in Altavista. Rechnet man zudem den Aufwand für den Einsatz der Datenbank pessimistisch mit einem Faktor $k_{Transfer}^{Mehraufwand} = 3$ der Transferzeit ein, wird für den Transfer von Postinglisten

bei Einsatz TPM/ONT rund $k_{Transfer}^{Mehraufwand} * k_{Liste}^{Mehraufand} = 3 * 1.6 \approx 5$ Mal mehr Zeit benötigt als von Altavista. Geht man von dem vorgängig erwähnten I/O-Engpass aus, d.h. lässt man die Zeit für die Berechnung der Trefferdokumente ausser Acht, benötigt TPM/ONT rund fünfmal mehr Zeit für die Auswertung einfacher Anfragen als Altavista.

Sollen komplexere Anfragen beantwortet werden, beispielweise Proximity-Anfragen, ändert sich die benötigte Ausführungszeit in Altavista nur unwesentlich, da sehr viele Anfragen direkt mit Hilfe des Textindex beantwortet werden können. Bei Einsatz von TPM/ONT-Text erhöht sich die Ausführungzeit jedoch deutlich, wie anhand eines Beispiels verdeutlicht werden kann: Werden alle Dokumente gesucht, welche die Wörter X und Y im Abstand von maximal 5 Wörtern enthalten, müssen zuerst alle Dokumente ermittelt werden, welche sowohl X, wie auch Y enthalten. Dies kann mittels Textindex-Auswertung geschehen. Anschliessend muss jedoch auf jedes hierdurch ermittelte Dokument zugegriffen werden und die Bedingung *"im Abstand von maximal 5 Wörtern"* überprüft werden. Dies ist mit grossem zeitlichen Aufwand (je ein separater Sekundärspeicherzugriff pro Dokument plus lineares Durchsuchen desselben) verbunden: Müssen beispielsweise 1'000 potentielle Trefferdokumente, d.h. Dokumente, welche die Wörter X und Y irgendwo enthalten, auf die Bedingung *"im Abstand von maximal 5 Wörtern"* überprüft werden, liegt alleine die Transferzeit der potentiellen Dokumente vom Sekundär- in den Primärspeicher im Bereich 20 Sekunden ($1'000 * 20$ms).

6.5 Skalierbarkeit der Messergebnisse

TPM/ONT erlaubt es, nicht nur eine, sondern verschiedene Dokumentenkollektionen zu definieren und zu verarbeiten. Das heisst, dass es möglich ist, im Rahmen eines physischen Entwurfs einer Datenbank ein Dokumentenkollektion je nach Anforderung in eine Menge kleinerer Kollektionen zu unterteilen und getrennt zu indexieren. Im Fall von Usenet-News wäre es beispielsweise möglich, jede einzelne Newsgruppe (z.B. talk.libertarian.politics, comp.databases, comp.databses.oracle) in separaten Tabellen zu speichern und jedes einzelne Textattribut dieser Tabellen separat zu indexieren. Durch eine solche Partitionierung wird einerseits eine semantische Unterteilung erreicht, andererseits werden kurze Antwortzeiten beim Suchen nach Dokumenten erzielt, da nur relativ kleine Dokumentenkollektionen (und damit kleine Textindizes) in eine Anfrage invoviert sind.

Die präsentierten Messungen basierten alle auf einer einzelnen Dokumentenkollektion mit 40'000 Usenet Artikeln. Eine einzelne Usenet-Gruppe mit 40'000 aktuellen Dokumenten ist sehr gross. Die durchgeführten Messungen sind in Bezug auf Antwortzeiten und Durchsatz bei Suchoperationen daher eher obere Schranken.

In anderen Anwendungsbereichen sind jedoch durchaus grössere Dokumentenkollektionen denkbar. Nachfolgend wird daher auf das Verhalten des Systems im Fall grösserer Dokumentenkollektionen eingegangen.

Einfügen und Löschen von Dokumenten

Der Aufwand für das Einfügen neuer Dokumente ist in TPM/ONT von der Anzahl der bereits eingefügten Dokumente weitgehend unabhängig: Für jeden einzufügenden Deskriptor eines

Dokuments erfolgt ein logischer I/O zum Lesen der Datenbankseite, auf welcher der Deskriptor eingefügt werden muss, anschliessend wird er eingefügt und die Datenbankseite in Abhängigkeit von der Pufferverdrängungsstrategie wieder auf den Sekundärspeicher geschrieben.

Einen grossen Einfluss auf die Einfügedauer hingegen hat die Grösse des Datenbankpuffers. Steht TPM/ONT ein Puffer in der Grössenordnung mehrerer Gigabyte — wie in Altavista — zur Verfügung, können Dokumente praktisch ohne Schreibzugriffe auf den Textindex eingefügt werden. Die Einfügezeit wird in diesem Fall durch die Zeit für das Schreiben des Redo/Undologs durch das Datenbanksystem bestimmt.

Der Aufwand für das Löschen eines Dokuments aus einer Dokumentenkollektion ist wie der Aufwand für das Einfügen von Dokumenten nur von der Anzahl Postings der Dokumente, nicht aber von der Grösse der Dokumentenkollektion abhängig.

Einfügen neuer Dokumente parallel zur Suche nach Dokumenten

Werden gleichzeitig zur Suche nach Dokumenten auch neue Dokumente eingefügt, kommt es zu den in dieser Arbeit bereits ausführlich behandelten Pseudokonflikten und damit einer Behinderung von Transaktionen. Mit steigendem Datenvolumen und gleichbleibender Benutzeranzahl vergrössert sich zwar die Wahrscheinlichkeit eines Pseudokonflikts nicht, jedoch hat das wachsende Datenvolumen Einfluss auf die Zeit, während der eine Transaktion aufgrund eines Pseudokonflikts blockiert ist. Hier muss zwischen drei Fällen der Behinderung durch Pseudokonflikte unterschieden werden:

- *Suchtransaktion behindert Einfügetransaktion:* Muss eine Einfügetransaktion n Zeiteinheiten warten, weil eine Lesetransaktion gerade eine benötigte invertierte Liste liest, muss die Einfügetransaktion bei einer x-mal grösseren Datenkollektion (und damit x-mal grösseren Textindexstruktur im Fall invertierter Listen) rund $n * x$ Zeiteinheiten warten. Die Behinderungszeit ist somit linear von der Grösse der Dokumentenkollektion abhängig.

- *Einfügetransaktion behindert Suchtransaktion:* Die Behinderung einer Suchtransaktion durch eine Einfügetransaktion ist von der Grösse der Dokumentenkollektion unabhängig, da die Zeit für das Einfügen eines einzelnen Dokuments von der Grosse der Dokumentenkollektion unabhängig ist (siehe oben).

- *Einfügetransaktion behindert Einfügetransaktion:* Die Zeit für das Einfügen einzelner Dokumente ist von der Grösse der Dokumentenkollektion unabhängig (siehe oben).

Beim Einfügen neuer Dokumente in das Datenbanksystem parallel zum Suchen nach Dokumenten kann es jedoch nicht nur zu Pseudokonflikten, sondern auch "echten" Konflikten kommen, falls sich ein einzufügendes Dokument auf eine gerade laufende Anfrage qualifiziert. In einem Datenbanksystem — so auch in TPM/ONT — wird eine der beiden Transaktionen bis zum Ende der anderen verzögert um eine konsistente Sicht auf den Datenbestand sicherzustellen. Wird die Einfügetransaktion verzögert, ist die Verzögerungsdauer ungefähr linear von der Grösse des Datenbestands abhängig (die Dauer für die Ermittlung der Trefferdokumente einer Suchanfrage ist von der Grösse des Textindex abhängig, dieser nimmt linear mit der Grösse

der Dokumentenkollektion zu). Wird hingegen die Suchtransaktion behindert, ist die Behinderungsdauer von der Grösse der Dokumentenkollektion unabhängig, da die Einfügezeit von der Grösse der Dokumentenkollektion unabhängig ist.

Im Zusammenhang mit der Behinderungen von Transaktionen muss sich jedoch im Fall dieser Anwendung die Frage gestellt werden, ob diese vom applikatorischen Standpunkt aus gesehen überhaupt notwendig ist: In einer Anwendung wie Altavista, definiert jede Operation[4] aus Sicht des Benutzers eine eigene Transaktion. Wird im Rahmen einer Transaktion nur ein einziges Dokument eingefügt, kann zu Beginn der Einfügetransaktion ein Redo-Record geschrieben und anschliessend mit der Indexierung des Dokuments begonnen werden. Versagt das System während der Indexierung, kann dank des Redo-Records jederzeit eine Forward-Recovery durchgeführt werden. Da zudem sicher ist, dass die Einfügetransaktion nie durch den Benutzer abgebrochen wird, kann einer parallel zur Einfügetransaktion laufenden Suchtransaktion in jedem Fall erlaubt werden, auf die Textindexstrukturen das gerade einzufügenden Dokuments bereits vor Ende der Einfügetransaktion zuzugreifen, da sichergestellt ist, dass das einzufügende Dokument in jedem Fall in die Datenbank eingebracht wird.

Eine solche Anpassung der Transaktionsverwaltung an die Applikation "Internet Search Engine" wäre in TPM/ONT durch eine einfache Änderung der L_2–Konfliktmatrix — keine Konflikte zwischen den Operationen *InsertDocument* und *RetrieveDocument* — sowie der Realisierung einer Forward-Recovery im Rahmen der Kompensationsoperation $InsertDocument^{-1}$ möglich.

Abschliessend kann zur Skalierbarkeit von TPM/ONT bezüglich der Grösse der Dokumentenkollektion festgehalten werden, dass im Vergleich zu transaktionslosen Informationssystemen wie Altavista sicherlich mit einem Leistungsverlust im Rahmen einer Grössenordnung zu rechnen ist. In einer Reihe von Anwendungen sind die dadurch entstehenden Kosten jedoch durch den Vorteil, neue Dokumente sofort zur Verfügung stellen zu können, mit Sicherheit zu rechtfertigen.

Steigende Anzahl parallel arbeitender Benutzer

Eine letzte interessante Fragestellung bezüglich der Skalierbarkeit von TPM/ONT ist, wie sich das System bei steigender Anzahl parallel arbeitender Benutzer verhält. Hierüber können jedoch nur spekulative Aussagen gemacht werden, da das Verhalten eines Datenbanksystems — insbesondere wenn parallel zu Lesetransaktionen auch Schreibtransaktionen durchgeführt werden — eine Dynamik entwickelt, die von vielen Faktoren, insbesondere der Grösse des Datenbankpuffers, der Charakteristika der eingsetzten Sekundärspeichermedien sowie der Ankunftsintervalle der Lese- und Einfügetransaktionen abhängt. Eine analytische Untersuchung eines solchen Systems ist daher nur schwer möglich. Um Aussagen über die Skalierbarkeit bei steigender Benutzeranzahl treffen zu können, müssten daher entweder weitere Messungen mit entsprechend vielen Benutzern (und eventuell grösseren Datenmengen) durchgeführt oder ein entsprechendes Datenbanksystem simuliert werden. Beide Varianten würden den Rahmen dieser Arbeit jedoch sprengen und werden hier nicht weiter betrachtet, sollten jedoch Gegenstand zukünftiger Untersuchungen im Rahmen weiterer Arbeiten sein.

[4] Einfügen eines Dokuments, Suchen nach Dokumenten

	Leser		Einfüger	
	Antw	Durch	Antw	Durch
Mixed–Traditionell	6.6	0.7	151	0.12
Mixed–TPM/ONT-Sequentiell-25	6.5	0.6	42	0.46

(a) 5 Leser, 20 Einfüger

	Leser		Einfüger	
	Antw	Durch	Antw	Durch
Mixed–Traditionell	7.3	1.3	112	0.10
Mixed–TPM/ONT-Sequentiell-25	7.7	1.1	46	0.31

(b) 10 Leser, 15 Einfüger

	Leser		Einfüger	
	Antw	Durch	Antw	Durch
Mixed–Traditionell	6.7	2.1	169	0.048
Mixed–TPM/ONT-Sequentiell-25	6.4	2.2	32	0.303
Mixed–TPM/ONT-Parallel-25	7.9	1.6	26	0.367

(c) 15 Leser, 10 Einfüger

	Leser		Einfüger	
	Antw	Durch	Antw	Durch
Mixed–Traditionell	7.8	2.5	141	0.033
Mixed–TPM/ONT-Sequentiell-25	8.0	2.4	27	0.176
Mixed–TPM/ONT-Parallel-25	8.2	2.0	17	0.226

(d) 20 Leser, 5 Einfüger

Tabelle 6.11: Gemischter Lese/Einfüge–Betrieb: Durchschnittliche Antwortzeit (in Sekunden) und Durchsatz (Transaktionen pro Sekunde) für das Suchen und Einfügen von Dokumenten

7 Schlussbetrachtung

7.1 Zusammenfassung

Im Rahmen der vorliegenden Arbeit wurde der Frage nachgegangen, inwieweit heutige Datenbanksysteme für eine transaktionsorientierte Verwaltung semistrukturierter Dokumente eingesetzt werden können. Im speziellen wurde untersucht, ob für diese Aufgabe relationale Systeme effizient einsetzbar sind.

Hauptproblem heutiger, insbesondere kommerziell erhältlicher relationaler Datenbanksysteme in Bezug auf Dokumente mit Fliesstextanteil ist, dass sie aus Effizienzüberlegungen weder Zugriffsstrukturen über Volltextattributen zur Verfügung stellen, noch deren Integration in das System erlauben.

In dieser Arbeit wurde ein bereits früher vorgestellter Ansatz zur Indexierung von Volltextattributen verfolgt, um effizient nach Dokumenten mit bestimmten Wörtern in Volltextattributen suchen zu können. Aus jedem Textattribut werden die einzelnen Wörter extrahiert und diese zusammen mit einem Verweis auf das Dokument in einer Benutzerrelation in Form einer invertierten Liste gespeichert. Diese Zugriffsstruktur kann — wie eine frühere Leistungsmessung [KS95] bereits gezeigt hat — effizient für die Beantwortung textueller Anfragen eingesetzt werden.

Untersuchungen haben gezeigt, dass die Wartung textueller Indexstrukturen in einem Datenbanksystem mit einer traditionellen, auf dem Zwei-Phasen-Sperrprotokoll beruhenden Transaktionsverwaltung sehr schnell zu einem Leistungsengpass führen kann. Wird in einem solchen Informationssystem nicht nur nach Dokumenten gesucht, sondern werden im laufenden Betrieb dynamisch neue Dokumente hinzugefügt oder bestehende gelöscht, kommt es im Mehrbenutzerbetrieb zu einer sperrbedingten Verzögerung parallel ablaufender Transaktionen.

Durch den Einsatz eines Datenbanksystems wie Oracle, das eine auf Versionen basierende Transaktionsverwaltung auf Tupelebene einsetzt, lässt sich dieser Leistungsengpass entschärfen. Steht jedoch nur ein seitensperrendes Datenbanksystem wie Sybase zur Verfügung, kommt es zu den bereits beschriebenen Engpässen.

Sperrbedingte Behinderungen können nur durch ein frühzeitiges Freigeben der Sperren vor dem Ende der Transaktion, welche diese angefordert hat, behoben werden. In einem klassischen, d.h. einschichtigen Datenbanksystem ist dies mit der Aufgabe der Atomarität und Isolation von Transaktionen verbunden. Die hieraus resultierenden Inkonsistenzen des Datenbestands sind in vielen Applikationen jedoch nicht akzeptabel.

Um Sperren ohne Aufgabe der ACID-Eigenschaften freigeben zu können, wurde daher in dieser Arbeit ein Prototyp mit einer zweischichtigen Transaktionsverwaltung realisiert. Dieses

TPM/ONT genannte System erlaubt die Ausführung offen geschachtelter Transaktionen. Im Unterschied zu anderen Prototypen wurde mit TPM/ONT jedoch kein von grundauf neues System implementiert, sondern ein System entwickelt, das grösstenteils auf existierender, kommerzieller Datenbanktechnologie basiert. Kernstück des Systems ist der Transaktionsmonitor *Tuxedo*, der um einen Transaktionsmanager zur Ausführung semantisch reicher Operationen erweitert wurde. Alle durchgeführten Erweiterungen wurden als Services des TP–Monitors realisiert. TPM/ONT konnte somit ohne Änderungen am Code des Transaktionsmonitors Tuxedo verwirklicht werden. Die durch diese Sperr- und Logkomponente realisierte Transaktionsverwaltung bildet zusammen mit der eines unterliegenden Datenbanksystems, welches der TP-Monitor als Ressource verwendet, ein zweischichtiges Datenbanksystem im Sinne von [Wei91, WS92]. In diesem System entspricht eine Operation der Ebene L_1 der Ausführung eines einzelnen Services des Transaktionsmonitors. Jeder Service wird dabei im Rahmen einer eigenen Transaktion des unterliegenden Datenbanksystems, der Ebene L_0, ausgeführt. Die Sicherstellung der ACID–Eigenschaften auf Ebene L_1 — diese sind auf Datenbankebene nicht mehr garantiert, da jede L_1–Operation im Rahmen einer unabhängigen Transaktion ausgeführt wird — erfolgt über den bereits erwähnten Sperr- bzw. Logmanager.

Ganz im Gegensatz zu einem nicht erweiterten TP-Monitor ist es TPM/ONT möglich, mehrere Operationen der Ebene L_1 einer einzelnen Benutzertransaktion parallel auszuführen. *Intra*transaktionsparallelität auf Operationsebene L_1 wird dabei durch *Inter*transaktionsparallelität auf Datenbankebene, d.h. der Ebene L_0, realisiert.

Bei TPM/ONT handelt es sich um ein generisches System, d.h. es ist nicht nur auf die Ausführungen mehrschichtiger *Text*transaktionen limitiert, sondern erlaubt die Ausführungen beliebiger zweischichtiger Transaktionen. Die Einbindung der hierfür notwendigen neuen Operationen auf Ebene L_1 erfolgt in TPM/ONT in zwei Schritten. In einem ersten werden die Operationen selbst sowie die dazugehörigen Kompensationsoperationen dem TP–Monitor in Form von Services bereitgestellt. In einem zweiten Schritt wird definiert, mit welchen bisherigen L_1–Operationen die neu eingeführten kommutieren und mit welchen nicht. Anschliessend können die neuen wie alle bisherigen L_1-Operationen im Rahmen von Benutzerprogrammen ausgeführt werden.

Auf den Prototypen TPM/ONT aufbauend wurde für die Verwaltung von und die Suche in Dokumentenkollektionen eine Reihe von L_1-Operationen realisiert und in TPM/ONT integriert. Zur Durchführung der Leistungsevaluation wurden Operationen zum Einfügen, Lesen und Löschen von Dokumenten sowie den zugehörigen Operationen zum Warten der textuellen Indexstrukturen verwirklicht.

Die mit TPM/ONT durchgeführten Messungen zeigen, dass die Verwaltung von Volltextdokumenten einschliesslich deren Indexstrukturen in einem relationalen Datenbanksystem dank der Vorteile einer zweischichtigen Transaktionsverwaltung effizient möglich ist. Haupterkenntnisse der vorliegenden Arbeit in Bezug auf die Dokumentenverwaltung und Suche mittels relationaler Datenbanksysteme, aber auch auf die Ausführung von Mehrschichtentransaktionen mittels TPM/ONT sind:

- In einem Datenbanksystem wie Oracle, in dem sich dank des Einsatzes einer versionierenden Transaktionsverwaltung Benutzer gegenseitig nur dann behindern, wenn sie versuchen, dasselbe Datum (Tupel) gleichzeitig zu ändern, kann eine Verbesserung der Antwortzeiten ausschliesslich durch ein paralleles Ausführen voneinander unabhängiger

L_1–Operationen erreicht werden.

Stehen auf Seite des TP-Monitors genügend Ressourcen, d.h. Server zur Verfügung, werden diese Operationen tatsächlich *parallel* und nicht nur *pseudo-parallel* ausgeführt.

Aber auch wenn nicht genügend Ressourcen für eine parallele Ausführung bereitstehen oder eine Benutzertransaktion alle ihre Operationen strikt sequentiell ausführt, werden die durchschnittlichen Antwortzeiten durch TPM/ONT im Vergleich zur einschichtigen Ausführung verkürzt. Diese Leistungsverbesserung hat seinen Ursprung in den internen Datenstrukturen des eingesetzten Datenbanksystems. Diese wachsen mit der Anzahl der im Rahmen einer Transaktion geänderten Tupel. Vor einer Änderungsoperation einer Transaktion werden diese Datenstrukturen traversiert, um Sperren auf die zu ändernden Objekte anzufordern. Werden im Rahmen einer langen Transaktion, beispielsweise beim Einfügen eines Dokuments, viele Tupel geändert, wird für das Traversieren dieser Datenstrukturen viel Zeit benötigt. Durch das Aufteilen der langen einschichtigen Transaktion in mehrere kurze zweischichtige, können diese Datenstrukturen früher freigegeben werden. Sie müssen anschliessend nicht mehr traversiert werden, was zu kürzeren Antwortzeiten führt.

- Werden bei der Mutation eines Dokuments Statistikinformationen für eine effizientere Anfragebeantwortung nachgeführt, treten in einem einschichtigen Datenbanksystem aufgrund der Zipf'schen Verteilung von Deskriptoren Hot-Spots auf, es kommt zu einer massiven Behinderung parallel ablaufender Transaktionen. TPM/ONT entschärft durch die frühzeitige Freigabe der Sperren auf den Statistikdaten diese Hot-Spots und steigert dadurch den Transaktionsdurchsatz.

  Durch die frühzeitige Sperrfreigabe haben parallel zu einer Änderungstransaktion laufende Suchoperationen unter Umständen Zugriff auf inkorrekte statistische Daten. Diese Inkonsistenz wird zugunsten eines höheren Transaktionsdurchsatzes in Kauf genommen, da hierdurch nur der Ausführungsplan einer Suchoperation beeinflusst wird, was jedoch keinen Einfluss auf das Ergebnis der Suche hat.

- Wird nicht Oracle, sondern ein seitensperrendes Datenbanksystem wie Sybase als Ressource–Manager eingesetzt, kommt es bereits bei Operationen, die nicht auf gemeinsamen Daten an sich, sondern auf Daten, die auf derselben Datenseite liegen, zu massiven Behinderungen. Grund dafür ist das verwendete grobe Sperrgranulat (Seitensperren) und die lange Sperrdauer. Werden beispielsweise zwei Dokumente durch zwei unabhängige Transaktionen parallel eingefügt, werden diesen Dokumenten aufeinanderfolgende Dokumenten-IDs zugeordnet. Haben diese Dokumente auch nur einen Deskriptor gemeinsam — was gemäss der Zipf'schen Verteilung von Deskriptoren mit hoher Wahrscheinlichkeit der Fall ist — kommen die Verweise dieser Deskriptoren auf derselben Seite in der invertierten Liste[1] zu liegen. Hierdurch wird eine der beiden Transaktionen bis zum Ende der anderen unnötig verzögert.

  Leseoperationen, die parallel zu Änderungsoperationen ausgeführt werden, behindern sich gegenseitig massiv. Bei der Transaktionsverwaltung von Sybase kann es zudem vorkommen, dass Einfügetransaktionen durch sich überlappende Lesesperren paralleler Suchtransaktionen ansatzweise verhungern. Durch das Verkürzen der Transaktions- und damit der Sperrdauer auf Ebene L_0, gekoppelt mit prädikatorientierten Sperren auf

[1] Genauer gesagt: Im B*-Baum, der als Zugriffsstrukturen über der invertierten Liste angelegt wird.

Ebene L_1, können durch Pseudokonflikte auf der Indexebene hervorgerufene Leistungsengpässe grösstenteils beseitigt werden.

7.2 Anforderungen an künftige Datenbanksysteme

7.2.1 Verwaltung von und Suche in Kollektionen semistrukturierter Dokumente

Die vorgestellten Messungen haben gezeigt, dass semistrukturierte Dokumente mittels Informationssystemen, die auf heutigen Datenbanksystemen aufbauen, effizient verwaltet werden können. Sollte in Zukunft ein Datenbanksystem speziell zur Verwaltung semistrukturierter Dokumente entworfen und implementiert werden, sollte dieses — aus Sicht der vorliegenden Arbeit — folgenden Anforderungen genügen:

- Textuelle Zugriffsstrukturen müssen wie Zugriffsstrukturen über anderen Datentypen intern vom Datenbanksystem verwaltet werden, d.h. sie sind vom Datenbanksystem selbst zu warten und bei einer Suche in den Ausführungsplan miteinzubeziehen.

- Dem Administrator einer Datenbank sollte die Möglichkeit gegeben werden, neue datenadäquate Operationen zur Extraktion von Deskriptoren aus Textattributen in das Datenbanksystem einzubinden und eigene Stopwortlisten zu definieren.

- Um sperrbedingte Leistungsengpässe auf den textuellen Indexstrukturen zu vermeiden, dürfen für die Transaktionskontrolle auf diesen nicht Locks eingesetzt werden, sondern es sind Latches, d.h. Kurzzeitsperren, zu verwenden. Die korrekte Ausführung von Transaktionen muss durch Verwendung von Locks auf den Dokumenten selbst sichergestellt werden. Ein solches Vorgehen entspricht in etwa den prädikatbasierten Sperren auf Ebene L_1 in TPM/ONT-Text.

- Werden statistische Informationen wie Dokumentenhäufigkeiten im laufenden Betrieb nachgeführt, müssen für diese ebenfalls Latches anstatt Locks verwendet werden. Im Fall eines Transaktionsabbruchs müssen durchgeführte Änderungen mittels einer operationsorientierten Recovery kompensiert werden.

  Parallel ablaufenden Suchtransaktionen kann der Zugriff auf diese teilweise inkorrekten Daten jederzeit gestattet werden (engl.: *dirty reads*). Dies hat — wie bereits ausgeführt — keinen Einfluss auf das Resultat einer Abfrage.

7.2.2 Erhöhung des Transaktionsdurchsatzes

Die Leistungsuntersuchungen anhand der Applikation "Dokumentenverwaltung" haben gezeigt, dass die Antwortzeiten von Transaktionen durch den Einsatz einer zweischichtigen Transaktionsverwaltung mit Unterstützung offen geschachtelter Transaktionen verkürzt werden können. Dies hat seine Ursache zum einen in der frühzeitigen Freigabe der Sperren der unteren Ebene, zum anderen in der parallelen Ausführung von Operationen der semantisch reichen Operationen.

Künftige Datenbanksysteme sollten daher die parallele Ausführung voneinander unabhängiger Teile einer Transaktion erlauben. Im Gegensatz zu TPM/ONT sollte das Datenbanksystem diese Teile jedoch im Rahmen *einer* Datenbanktransaktion ausführen.

Um dies zu ermöglichen, muss die Sperrverwaltung heutiger Datenbanksysteme erweitert werden: Im Rahmen einer Benutzertransaktion muss die Möglichkeit geschaffen werden, voneinander unabhängige Sequenzen von SQL–Operationen als solche zu deklarieren[2]. Die Sperrverwaltung muss für jede dieser Sequenzen vor Ausführung einer Operation eigene Sperren auf die betroffenen Datenbankobjekte erwerben. Diese werden am Ende einer Sequenz gegenüber anderen Sequenzen derselben Benutzertransaktion freigegeben, gegenüber anderen jedoch gehalten.

Die Forderung nach einer parallelen Ausführung von Operationen wird auch im neuen SQL3–Standard aufgestellt. SQL3 beschränkt sich jedoch auf die asynchrone Ausführung *einzelner* SQL–Operationen und nicht einer Folge von SQL–Anweisungen wie es in TPM/ONT möglich ist.

Sind Datenbanksysteme in der Lage, Operationssequenzen parallel auszuführen, sollte ermöglicht werden, angeforderte Sperren vor Ende einer Transaktion freizugeben. Durch diese frühzeitige Freigabe kann der Grad der Intertransaktionsparallelität in vielen Applikationen erhöht werden. In einem einschichtigen Datenbanksystem ist dies jedoch mit der Aufgabe der ACID–Eigenschaften verbunden und daher nicht durchführbar. Bei Einsatz eines offen-geschachtelten Transaktionsmodells ist eine frühzeitige Freigabe möglich, wenn eine Sperr- und Logverwaltung auf höherer Ebene realisiert wird, welche die Semantik der durchgeführten Operationen ausnutzt. Im Fall eines Transaktionsabbruchs müssen die durchgeführten Änderungen mittels einer operationsbasierten Recovery (Kompensationsoperationen) rückgängig gemacht werden.

Sowohl das Extrahieren von semantisch reichen Sperren aus einer Sequenz komplexer SQL–Anweisungen als auch das automatische Erzeugen von Kompensationsoperationen aus diesen Sequenzen darf in einem produktiven System wegen der Fehleranfälligkeit nicht dem Applikationsprogrammierer überlassen werden, sondern muss automatisch erfolgen. Beide Operationen können nur durch die Analyse des Programmflusses einer Operation automatisch durchgeführt werden. Ob eine solche Analyse überhaupt möglich ist und wenn ja, ob sie auch zeitlich lohnenswert ist, bleibt ungeklärt.

Technisch möglich hingegen ist schon heute die Erzeugung von Sperren und Kompensationsoperationen für einzelne SQL-Anweisungen, wie beispielsweise in [Sch96] gezeigt wird. Die Betrachtung einzelner Anweisungen anstatt Anweisungssequenzen hat jedoch den Nachteil, dass in der Regel mehr Objekte gesperrt werden als notwendig ist.

7.2.3 Verteilte Datenbanksysteme

Die Messungen mit TPM/ONT haben gezeigt, dass durch den Einsatz einer mehrschichtigen Transaktionsverwaltung selbst im Fall einer synchronen Ausführung von L_1-Operationen oftmals kürzere Antwortzeiten erreicht werden als mit der Ausführung einer flachen Transaktion unter Verwendung des XA–Interfaces.

[2] Dies entspricht einzelnen *Transaction Branches* der XA+–Spezifikation [X/O94].

Es liegt daher nahe, TPM/ONT nicht nur zur Ausführung von Transaktionen in einem zentralen Datenbanksystem, sondern auch in einer verteilten, eventuell sogar heterogenen Datenbankumgebung einzusetzen. An der vorliegenden prototypischen Realisierung von TPM/ONT müssen in einer verteilten Umgebung Änderungen an der Sperr- und Logverwaltung der Ebene L_1 vorgenommen werden: Die Systemservices zum Sperren von L_1–Operationen nützen bisher das Wissen aus, dass sich alle Server von TPM/ONT auf derselben Maschine befinden. Es ist daher möglich, gemeinsame Datenstrukturen in einem gemeinsam benutzten Hauptspeichersegment anzulegen. In einer verteilten Umgebung ist dies nicht mehr möglich, da ein Server in der Regel auf dem Rechner gestartet wird, auf dem sich sein Ressource-Manager, d.h. das Datenbanksystem, befindet. Es ist daher notwendig, auf einem dedizierten Rechner spezielle Server zur Durchführung der Sperroperationen zu installieren.

Ähnlich verhält es sich mit der Verwaltung des L_1–Logs. Hier wird heute das Wissen ausgenützt, dass alle Server mit demselben Ressource–Manager verbunden sind. L_1–Logsätze können daher von jedem Server in dessen Datenbanksystem geschrieben werden. In einer verteilten Umgebung ist es notwendig, sich für ein ausgewähltes Datenbanksystem zu entscheiden, in welchem alle Logsätze für die Kompensation semantisch reicher Operationen persistent gespeichert werden.

Mit Hilfe dieser Erweiterung wäre es möglich, Mehrschichtentransaktionen in einer verteilten Umgebung effizient zu realisieren und insbesondere die durch das Zwei–Phasen–Commitprotokoll entstehenden Engpässe zu umgehen.

Literaturverzeichnis

[ANS75] ANSI/X3/SPARC Study Group on Database Management Systems: Interim Report. FDT Bulletin (ACM SIGMOD), 7(2), 1975.

[BCC94] E.W. Brown, J.P. Callan und W.B. Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings International Conference on Very Large Databases*, Seiten 192–202, Santiago, Chile, September 1994.

[BCCM93] E.W. Brown, J.P. Callan, W.B. Croft und J.E.B. Moss. Supporting Full-Text Information Retrieval with a Persistent Object Store. Technical Report 93-67, Dept. of Computer Science. University of Massachusetts, Amherst, MA 01003, USA, August 1993.

[BHG87] P.A. Bernstein, V. Hadzilacos und N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Bil82] H. Biller. On the Architecture of a System Integrating Data Base Management and Information Retrieval. In G. Goos und H. Hartmanis, Hrsg., *Research and Development in Information Retrieval*, Jgg. 146 of *Lecture Notes in Computer Science*, Seiten 80–97. Springer, Mai 1982.

[BM96] T.A.H. Bell und A. Moffat. The Design of a High Performance Information Filtering System. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seiten 12–20, Zurich, Switzerland, August 1996.

[Coo88] W.S. Cooper. Getting Beyond Boole. *Information Processing & Management*, 24(3):243–248, 1988.

[Cra81] R.G. Crawford. The Relational Model in Information Retrieval. *Journal of the ASIS*, Seiten 51–64, 1981.

[Dep89] U. Deppisch. *Signaturen in Datenbanksystemen.* Dissertation, TU Darmstadt, Fachbereich Informatik, Februar 1989.

[DPS83] P. Dadam, P. Pistor und H.-J. Schek. A Predicate Oriented Locking Approach for Integrated Information Systems. In R.E.A. Mason, Hrsg., *Proceedings of the IFIP*, 1983.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie und I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[EHPR81] K. Elhardt, F. Höckenrainwer, R. Poloczek und B. Ruhbach. SQL-TR: A System R Extension for Text Retrieval. Internal report, IBM Heidelberg Scientific Center, 1981.

[Ern95] A. Erni. PlentyOfText. Diplomarbeit, ETH Zürich, Departement Informatik, August 1995.

[Fal85] C. Faloutsos. Access Methods for Text. *ACM Computing Surveys*, 17(1):49–74, Marz 1985.

[FC84] C. Faloutsos und S. Christodoulakis. Signature Files: An Access Method for Documents and its Analytical Performace Evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, 1984.

[GBYS92] G.H. Gonnet, R.A. Baeza-Yates und T. Snider. New Indices for Text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, Kapitel 5, Seiten 66–82. Addison-Wesley, 1. Auflage, 1992.

[GR93] J. Gray und A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Has95] Ch. Hasse. *Inter- und Intratransaktionsparallelität in Datenbanksystemen.* Dissertation, ETH Zürich, Departement Informatik, 1995.

[HFRYL92] D. Harman, E. Fox, R.Baeza-Yates und W. Lee. Inverted Files. In W. B. Frakes und R. Baeza-Yates, Hrsg., *Information Retrieval (Data Structures and Algorithms)*, Kapitel 3, Seiten 28–43. Prentice Hall, 1992.

[HR83] T. Härder und A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), Dezember 1983.

[HW92] D.J. Harper und A.D.M. Walker. ECLAIR: An Extensible Class Library for Information Retrieval. *The Computer Journal*, 35(3):256–267, 1992.

[Ill94] Illustra Information Technologies. *Text DataBlade Guide*, 1.2. Auflage, Juni 1994.

[Ill95] Illustra Information Technologies. *Illustra User's Guide*, 3.2. Auflage, Oktober 1995.

[Inf90] Information Dimension Inc., 655 Metro Place South, Dublin, Ohio 43017-1396. *BASISPlus Database Administration Guide*, Juni 1990. Release L.

[Knu73] D.E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, 1973.

[KR96] M. Kamath und K. Ramamritham. Efficient Transaction Support for Dynamic Information Retrieval Systems. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, August 1996.

[KS95] H. Kaufmann und H.-J. Schek. Text Search in Database Systems Revisited—Some Experiments. In *Proc. of the the 13th Britisch National Conference on Databases*, Lecture Notes in Computer Science, Seiten 201–225. Springer, Juli 1995.

[KS96] D. Knaus und P. Schäuble. The System Architecture and the Transaction Concept of the SPIDER Information Retrieval System. *Bulletin of the Technical Committee on Data Engineering*, 19(1):43–52, Marz 1996. Special Issue on Integrating Text Retrieval and Databases.

[LDE+85] V. Lum, P. Dadam, R. Erbe, J. Günauer, P. Pistor, G. Walch, H. Werner und J. Woodfill. Design of an Integrated DBMS to Support Advanced Applications. In *International Conference on Foundations of Data Organization*, Kyoto, 1985.

[LKD+88] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch und M. Wallrath. Design and Implementation of Extensible Database Management System Supporting User Defined Data Types and Functions. In *Proceedings International Conference on Very Large Databases*, Los Angeles, California, 1988.

[LM95] O. Lorenz und G. Monagan. A Retrieval System for Graphical Documents. In *Fourth Annual Symposium on Document Analysis and Information Retrieval*, Seiten 291–300, April 1995.

[Mac79] I.A. Macleod. SEQUEL as a Language for Document Retrieval. *Journal of the American Society for Information Science*, 30(5):243–249, September 1979.

[Mac83] I.A. Macleod. A Model for Integrated Information Systems. In *Proceedings International Conference on Very Large Databases*, Florence, 1983.

[Mos90] J.E.B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Office Information Systems*, 8(2):103–139, April 1990.

[MZ94] A. Moffat und J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. Technical Report CITRI/TR-94-2, Collaborative Information Technology Research Institute, Victoria, Australia, Februar 1994.

[MZK95] A. Moffat, J. Zobel und S. T. Klein. Improved Inverted File Processing for Large Text Databases. In *Proceedings of the Australian Computer Science Conference*, 1995.

[MZS94] A. Moffat, J. Zobel und N. Sharman. Text Compression For Dynamic Document Databases. Technical Report CITRI/TR-94-4, Collaborative Information Technology Research Institute, Victoria, Australia, Mai 1994.

[Nov95] Novell, Inc. and BEA Systems. *TUXEDO Guides and Reference*, 1995.

[Ora94] Oracle TextServer, Marz 1994. Servers to manage very large document databases.

[Ora95] Oracle Corporation, Redwood Shores, CA. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7 (Oracle White Paper)*, Juli 1995.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[Pau88a] H.-B. Paul. *DAS Database Kernel System for Standard and Non-standard Applications —Architecture, Implementation, Applications—*. Dissertation, TU Darmstadt, Fachbereich Informatik, 1988.

[Pau88b] H.-B. Paul. *DAS Database Kernel System for Standard and Non-standard Applications —Architecture, Implementation, Applications—*. Dissertation, TU Darmstadt, Fachbereich Informatik, 1988.

[Rij81] C.J. van Rijsbergen. *Information Retrieval.* Butterworth, 1981.

[Sal75] G. Salton. *Dynamic Information and Library Processing.* Prentice Hall, 1975.

[Sal89] G. Salton. *Automatic Text Processing.* Addison-Wesley, 1989.

[Sch77] H.-J. Schek. The Reference String Access Method and Partial Match Retrieval. Technical Report TR 77.12.008, IBM Germany, Heidelberg Scientific Center, Dezember 1977.

[Sch84] H.-J. Schek. Nested Transactions in a Combined IR–DBMS Architecture. In C.J. van Rijsbergen, Hrsg., *Proceeding of the 3rd BCS/ACM Symposium on Research and Development in Information Retrieval*, The British Computer Society Workshop Series, Seiten 55–70, Cambridge, Juli 1984. British Computer Society, Cambridge University Press.

[Sch87] H.-J. Schek. A Database Kernel System Supporting Application-Specific Layers—Architecture of the DASDBS Family. *Informationstechnik* **it**, 29(3):153–164, Marz 1987.

[Sch93] P. Schäuble. SPIDER: A Multiuser Information Retrieval System for Semistructured and Dynamic Data. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seiten 318–327, 1993.

[Sch96] W. Schaad. *Transaktionsverwaltung in heterogenen, föderierten Datenbanksystemen.* Dissertation, ETH Zürich, Departement Informatik, 1996.

[SM83] G. Salton und M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[SP82] H.-J. Schek und P. Pistor. Data Structures for an Integrated Database Management and Information Retrieval System. In *Proceedings International Conference on Very Large Databases*, Seiten 197–207, Mexico, 1982.

[SPSW90] H.-J. Schek, H.-B. Paul, M.H. Scholl und G. Weikum. The DASDBS Project: Objectives, Experiences and Future Prospects. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):25–43, Marz 1990. Special Issue on Database Prototype Systems.

[SQL92] SQL*TextRetrieval–Technical Overview, 1992. Version 2.

[SQL93] Draft Database Language SQL (SQL3). ISO-ANSI Working Draft, Februar 1993.

[SR86] M.R. Stonebraker und L.A. Rowe. The Design of POSTGRES. In *Proc. ACM SIGMOD Conf. on Management of Data*, Seiten 340–355, Washington, D.C., Mai 1986. ACM.

[SSL+83] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash und A. Guttmann. Document Processing in a Relational Database System. *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.

[SSW91] H.-J. Schek, M.H. Scholl und G. Weikum. The Background of the DASDBS & COSMOS Projects. In *Proc. Int'l Conf. on Mathematical Foundations of Database Systems (MFDBS)*, Rostock, Germany, Mai 1991. LNCS, Springer Verlag, Heidelberg.

[Sto86] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Bulletin of the Technical Committee on Data Engineering*, 1986.

[SW86] H.-J. Schek und G. Weikum. DASDBS: Concepts and Architecture of a Database System for Advanced Applications. Technical Report DVSI-1986-T1, TU Darmstadt, 1986.

[Syb93] *System Administration Guide for SYBASE SQL Server*, September 1993.

[TGMS93] A. Tomasic, H. Garcia-Molina und K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. Technical Note STAN-CS-TN-93-1, Stanford University, Department of Computer Science, Stanford, CA 94305-2140, August 1993.

[TGNO92] D. Terry, D. Goldberg, D. Nichols und B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM SIGMOD Conf. on Management of Data*, Seiten 321–330, 1992.

[Wec95] M. Wechsler. Eine Indexierungsmethode für Information Retrieval auf Audiodokumenten. In R. Kuhlen und M. Rittberger, Hrsg., *Hypertext—Information Retrieval—Multimedia*, number 20 in Schriften zur Informationswissenschaft, Seiten 117–128. Hochschulverband für Informationswissenschaften, April 1995.

[Wei86] G. Weikum. *Transaction Management in Layered Database System Architectures.* Dissertation, TU Darmstadt, Fachbereich Informatik, Dezember 1986.

[Wei91] G. Weikum. Principles and Realization Strategies of Multi-Level Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, Marz 1991.

[WS84] G. Weikum und H.-J. Schek. Architectural Issues of Transaction Management in Layered Systems. In *Proceedings International Conference on Very Large Databases*, Singapore, 1984.

[WS92] G. Weikum und H.-J. Schek. *Database Transaction Models for Advanced Applications*, Kapitel Concepts and Applications of Multilevel Transactions and Open Nested Transactions, Seiten 515–546. Morgan Kaufmann, 1992.

[WZRW89] S.K.M. Wong, W. Ziarko, V.V. Raghavan und P.C.N. Wong. Extended Boolean Query Processing in the Generalized Vector Space Model. *Information Systems*, 14, 1989.

[WZW85] S.K.M. Wong, W. Ziarko und P.C.N. Wong. Generalized Vector Space Model in Information Retrieval. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seiten 18–25, New York, 1985. ACM.

[X/O91] X/Open Ltd., Reading, England. *Distributed Transaction Processing: The XA Specification*, 1991.

[X/O94] X/Open Ltd., Reading, United Kingdom. *Distributed Transaction Processing: The XA+ Specification Version 2*, Juni 1994.

[X/O95] X/Open Ltd., Reading, United Kingdom. *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, April 1995.

[X/O96] X/Open Ltd., Reading, United Kingdom. *Distributed Transaction Processing: Reference Model Version 3*, Februar 1996.

[YGM95] T. Yan und H. Garcia-Molina. SIFT–A Tool for Wide-Area Information Dissemination. In *Proc. 1995 USENIX Technical Conference*, Seiten 177–186, 1995.

[Zip49] G.K. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley Press, 1949.

[ZMSD91] J. Zobel, A. Moffat und R. Sacks-Davis. An Efficient Indexing Technique for Full-Text Database Systems. Technical Report CITRI/TR-92-21, Collaborative Information Technology Research Institute, Victoria, Australia, Februar 1991.

[ZPD90] P. Zabback, H.-B. Paul und U. Deppisch. Office Documents on a Databse Kernel – Filing, Retrieval, and Archiving. In *Proc. ACM Conf. on Office Information Systems*, Seiten 261–270, Cambridge, Ma., April 1990.

Ganz zum Schluss

Ich weiss nicht, wie es anderen gegangen ist, die am Ende ihrer Dissertation, nachdem alles vorbeit ist, vor einem weissen Blatt Papier sitzen und verzweifelt nach den richtigen Worten für eine Danksagung suchen. Ich zumindest habe das Gefühl, nur noch eines sagen zu müssen:

Danke!

Mein Dank gilt allen, die mich auf meinem bisherigen Lebensweg begleitet haben, insbesondere aber jenen Menschen, die mich bei der Erstellung meiner Disseration unterstützt haben:

- Hans–Jörg Schek: Danke für die lehrreiche Zeit an der ETH und die Übernahme des Referats
- Gaston Gonnet: Danke für die Übernahme des Korreferats
- Lukas Relly und Kai Warszas: Danke für die Zeit, die Ihr in Diskussionen über meine Arbeit, aber auch über Gott und die Welt, mit mir verbracht habt
- Moira Norrie: Vielen Dank, dass Du immer Zeit für mich hattest, wenn ich Deine Hilfe benötigt habe
- Hanni Hilgarth und Jürg Gutknecht: Danke für die gute Zusammenarbeit im Abteilungssekretariat
- Antoinette Förster: Danke für die tolle Arbeit, die Du täglich im Sekretariat geleistet hast
- Claus Hagen, Uwe Röhm und Heiko Schuldt: Danke für Eure Diskussionsbereitschaft und die Möglichkeit, zusammen mit Euch beim Squash ein wenig abzuspannen
- Bettina Kemme: Danke für die kurze, aber sehr interessante Zeit, die wir am Ende meiner ETH-Laufbahn im selben Büro verbracht haben
- Tomas Felner und Marco Schmidt: Danke, dass Ihr meine andauernden Konfigurationswünsche betreffend "mein" SparcCenter immer verlässlich erfüllen konntet
- Meiner Mutter, Ilse Kaufmann: Herzlichen Dank für die viele Zeit, die Du mit dem Korrekturlesen meiner Arbeit verbracht hast
- Heidi Rosseel: Danke, dass Du immer Verständnis für mich hattest

Lebenslauf

Name	Helmut Ludwig Kaufmann, Dipl. Informatik-Ing. ETH
Staatsbürgerschaft	Österreich
Geburtsdatum	10. September 1967
Geburtsort	Vöcklabruck, Oberösterreich
Ausbildung	1974–1978 Primarschule in Triesen, Liechtenstein
	1978–1986 Neusprachliches Gymnasium der Zisterzienser in Bregenz–Mehrerau, Österreich
	1986–1991 Informatikstudium an der Eidgenössischen Technischen Hochschule Zürich
	1991–1996 Doktorand am Institut für Informationssysteme–Fachgruppe Datenbanken der ETH Zürich (Gruppe Prof. Dr. Hans-Jörg Schek)
Beruflicher Werdegang	April 1991 – November 1996 Assistent/Wissenschaftlicher Mitarbeiter am Institut für Informationssysteme–Datenbanken der ETH Zürich (Gruppe Prof. Dr. Hans-Jörg Schek)
	April 1994–Juni 1996 Sekretär der Abteilung für Informatik der ETH Zürich