FernUniversität
Gesamthochschule in Hagen

FACHBEREICH INFORMATIK

LEHRGEBIET PRAKTISCHE INFORMATIK IV

Prof. Ralf-Hartmut Güting

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
zum Thema

Redesign und Reimplementierung des Systemkerns des erweiterbaren DBMS SECONDO

vorgelegt von

Ulrich Telle
Wolfskaul 12
51061 Köln
Matrikel-Nr. 1471341

Köln, den 27. Juni 2002

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In der vorliegenden Diplomarbeit werden Redesign und Reimplementierung des SECONDO-Systemkerns unter Berücksichtigung verschiedener Anforderungen wie Portabilität, Mehrbenutzerunterstützung und Objektorientierung beschrieben. Eine vollständige Liste der Zielsetzungen ist in Abschnitt 1.3 auf Seite 4 aufgeführt. Zuvor soll jedoch kurz erläutert werden, worum es sich bei dem SECONDO-System überhaupt handelt und welche Gründe zur Redesign-Entscheidung geführt haben.

1.1 Das SECONDO-System

SECONDO ist eine generische Umgebung für die Implementierung von Datenbanksystemen [DG99]. Der Name SECONDO leitet sich von ***second-o****rder signature* (SOS) ab, da Signaturen zweiter Ordnung die formale Basis für die Definition von Datentypen und Abfragesprachen in SECONDO darstellen. Die grundlegende Idee von SOS besteht darin, zwei gekoppelte Signaturen zu verwenden - eine zur Beschreibung des Datenmodells und eine zur Beschreibung der Algebra über diesem Datenmodell. Eine ausführliche Beschreibung dieses Konzepts findet sich in [Güt93].

Das Ziel von SECONDO ist es, Rahmenbedingungen zur Erfüllung der Anforderungen heutiger Anwendungsgebiete wie etwa CAD (Computer Aided Design), GIS (Geographical Information Systems) oder Bilddatenverarbeitung zu bieten, da die vorrangig im Einsatz befindlichen relationalen Datenbanksysteme diesen nur unzureichend gerecht werden. Um dieses Ziel zu erreichen, wird die Strategie verfolgt, die vom Datenmodell unabhängigen Teile eines Datenbankmanagementsystems von den abhängigen Teilen mit Hilfe des SOS-Formalismus zu trennen.

Leichte Erweiterbarkeit wird durch das Konzept der Algebra-Module erreicht. Mit deren Hilfe können neue Datentypen und Operatoren über den Algebren definiert

und implementiert werden. Die Algebra-Module, in denen die jeweilige datenmodellabhängige Funktionalität bereitgestellt wird, werden durch eine Anzahl von Kernmodulen allgemeiner Funktionalität ergänzt.

In Abbildung 1.1 wird ein grober Überblick der Architektur des SECONDO-Systems gezeigt. In der Darstellung stehen weiße Kästchen für feststehende Systemteile, während graue Kästchen für erweiterbare Teile stehen, die von einer spezifischen Implementierung abhängig sind.

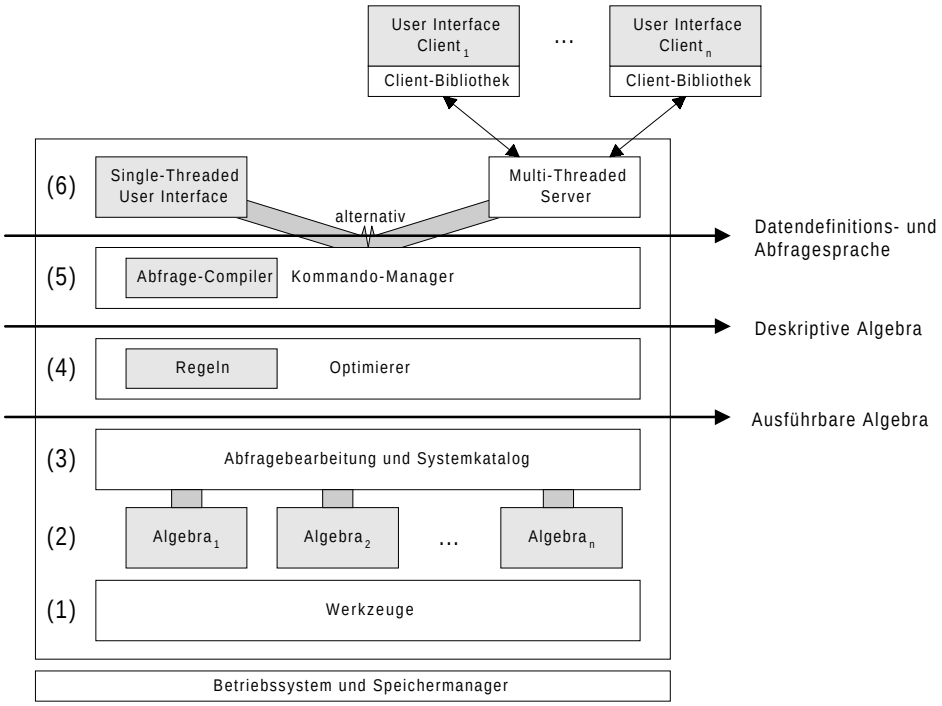

Abbildung 1.1: SECONDO-Architektur

Über der Betriebssystemebene finden sich auf Ebene 1 diverse unterstützende Module wie eine Bibliothek zur Behandlung verschachtelter Listen, Katalogisierungsfunktionen, ein Tupelmanager und schließlich der Parser für Signaturen zweiter Ordnung. Die verschiedenen Algebra-Module, die wesentlich die Erweiterbarkeit von SECONDO ausmachen, befinden sich auf Ebene 2; in ihnen werden unter Zuhilfenahme der Werkzeuge der Ebene 1 Typkonstruktoren und Operatoren der ausführbaren Abfragealgebra definiert und implementiert. Auf Ebene 3 findet die Abfragebearbeitung und die Verwaltung des Systemkatalogs statt. Der Abfrageoptimierer auf Ebene 4 wandelt mit Hilfe von Transformationsregeln eine Abfragebeschreibung in einen effizienten Auswertungsplan um. Der Kommandomanager stellt auf Ebene 5 eine prozedurale Schnittstelle für die Funktionalität der niedrigeren Ebenen zur Verfügung. Auf Ebene 6 finden sich schließlich die Benutzerschnittstellen, wobei entweder ein eigenständiges Ein-Benutzer-Programm oder ein Mehr-Benutzer-Server erzeugt werden kann.

1.2 Ausgangssituation

Die derzeitige SECONDO-Version läuft unter SOLARIS 2.6 auf SUN Sparc-Systemen und ist im Kern in Modula und in einigen Teilen in C bzw. C++ realisiert. Für die persistente Speicherung stützt sich SECONDO auf das Speicherverwaltungssystem SHORE (Scalable Heterogeneous Object REpository) [The95].

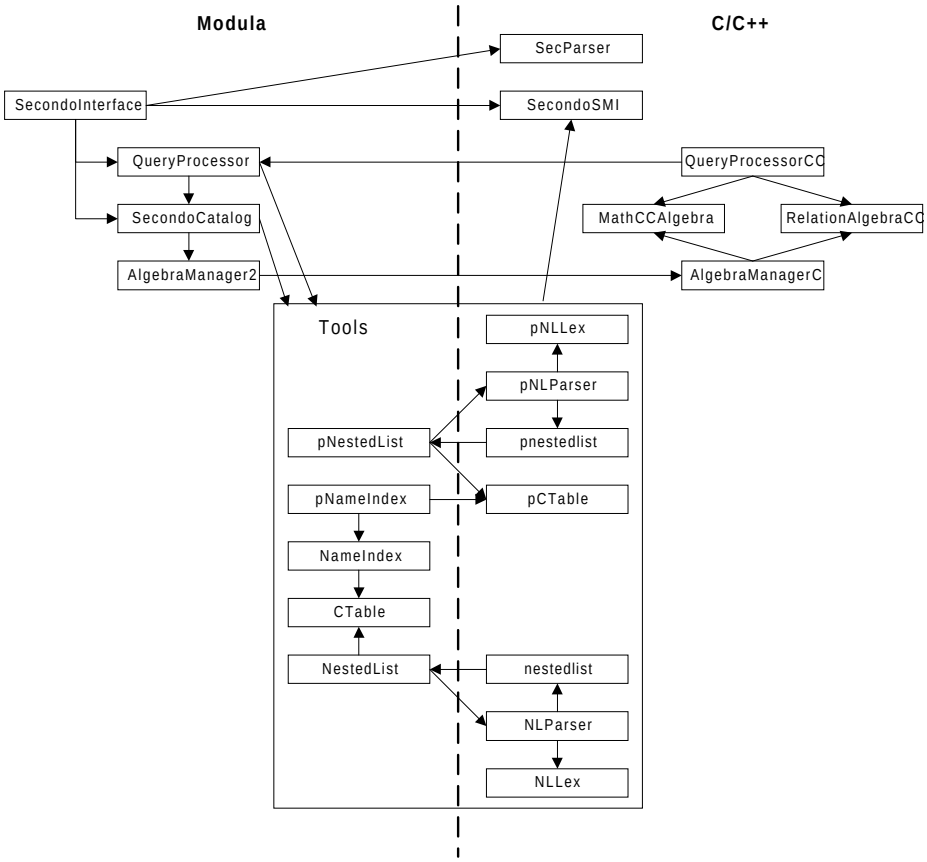

Abbildung 1.2: Modul- und Sprachabhängigkeiten in SECONDO

Im Hinblick auf die gewünschte *Portabilität* des Systems ergeben sich die nachfolgend genannten Schwierigkeiten.

Zum einen sind wegen der vergleichsweise geringen Verbreitung von Modula für andere Plattformen nicht unbedingt geeignete Compiler verfügbar. Der für die Übersetzung der Modula-Komponenten erforderliche EPC Modula-2 Compiler (Version 2.0.9) wird seit der Übernahme von *Edinburgh Portable Compilers Ltd* durch *Analog Devices* in 1999 nicht mehr angeboten und gewartet. Somit muss selbst unter SOLARIS mittelfristig mit Problemen bei der Weiterentwicklung von SECONDO gerechnet werden. Erschwerend kommt hinzu, dass sich die Compiler im jeweils unterstützten Modula-Dialekt und der verfügbaren Laufzeitbibliothek zum Teil deutlich voneinander unterscheiden, so dass die Portierung auf einen anderen Modula-2 Compiler mit erheblichem Aufwand verbunden sein kann.

Zum anderen erschwert gemischtsprachige Programmierung in der Regel die Portierung auf andere Betriebssysteme in hohem Maße, da Unterschiede in den Aufrufkonventionen und der Laufzeitumgebung berücksichtigt werden müssen.

In Abbildung 1.2 auf der vorherigen Seite werden einige wesentliche Modul- und Sprachabhängigkeiten in SECONDO dargestellt, wobei in der linken Hälfte die Modula-Komponenten und in der rechten Hälfte die C- bzw. C++-Komponenten angeordnet sind. Nicht selten treten Aufrufabfolgen der Art *C++ ruft Modula, Modula ruft C, C ruft Modula, Modula ruft C++* auf.

Anzumerken ist zu guter Letzt, dass SHORE nur auf einer begrenzten Menge von Plattformen zur Verfügung steht und derzeit nicht weiterentwickelt wird.

Daraus ergibt sich die Notwendigkeit, zum einen eine Implementierung anzustreben, die durchgängig in nur einer Programmiersprache erfolgt, und zum anderen nach Alternativen für die persistente Datenhaltung zu suchen.

1.3 Aufgabenstellung

Der SECONDO-Systemkern soll dahingehend überarbeitet werden, dass folgende Kriterien erfüllt werden:

- Programmierung einheitlich in C++,
- Gewährleistung hoher Portabilität (LINUX, SOLARIS und WINDOWS müssen in jedem Fall unterstützt werden, die Machbarkeit einer Portierung auf PALM-OS soll geprüft werden),
- einfache Speichersystem-Schnittstelle,
- Versionen für Mehrbenutzer- und Einbenutzerbetrieb mit Transaktionsunterstützung sowie für Einbenutzerbetrieb ohne Transaktionsunterstützung,
- Speichersystemimplementierung exemplarisch auf Basis der BERKELEY DB und auf Basis von ORACLE,
- sichere Mehrbenutzer-Verwaltung des Systemkatalogs und
- objektorientierte Verwaltung von Typen und Operatoren der Algebra-Module.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in vier Teile:

Im ersten Teil, Kapitel 2 ab Seite 6, werden die Vorüberlegungen und Konzepte für die Überarbeitung des SECONDO-Systemkerns auf der Basis von Analysen der Schwachstellen des bisherigen SECONDO-Systems und möglicher Architektur-Alternativen dargestellt. Hinsichtlich der Systemarchitektur für den Mehrbenutzerbetrieb werden Varianten der Client/Server-Architektur ausführlich diskutiert. Die derzeitige Implementierung des Speichersystems wird analysiert, um die Anforderungen, welche Alternativrealisierungen mindestens erfüllen müssen, zu identifizieren. Abschließend werden die vor allem im Hinblick auf Mehrbenutzerbetrieb vorhandenen Schwachstellen der bisherigen Systemkatalogimplementierung sowie die Verwaltung von Typen und Operatoren der verschiedenen Algebra-Module untersucht.

Im Fokus des zweiten Teils, Kapitel 3 ab Seite 28, stehen die Entwürfe zur Umsetzung der neuen Konzepte sowie die detaillierte Darstellung der Eigenschaftsprofile der verschiedenen Module.

Im dritten Teil, Kapitel 4 ab Seite 41, werden einerseits die systemtechnischen Randbedingungen und andererseits die Implementierungskonzepte und Realisierungsprobleme bei der Überarbeitung, Portierung bzw. Neuentwicklung der Schnittstellen und Module beschrieben. Viel Wert wurde dabei auf eine ausführliche und durchgängig in Englisch verfasste Dokumentation der Schnittstellen und Implementierungsdetails gelegt.

Der vierte Teil, Kapitel 5 ab Seite 57, gibt einen Überblick über die neue Systemarchitektur und ihre Besonderheiten, wobei speziell die Randbedingungen für Entwickler, die sich mit der Implementierung von Erweiterungen für das SECONDO-System befassen, Berücksichtigung finden.

Den Abschluss der Arbeit bildet ab Seite 83 eine kurze Zusammenfassung der erreichten Ergebnisse sowie ein Ausblick auf mögliche zukünftige Erweiterungen.

In den Anhängen ab Seite 86 finden sich die Literaturverzeichnisse sowie die wichtigsten Teile der Programmdokumentation.

Kapitel 2

Analyse

Zu Beginn der Analysephase stand die Frage, in welcher Programmiersprache die Reimplementierung des SECONDO-Systemkerns erfolgen sollte. Zur Wahl standen Java und C++. Einerseits sprach für die Wahl von Java, dass diese Sprache weitgehend plattformunabhängig ist, in der Lehre die Sprachen Pascal und Modula ablöst und die Studenten sie daher zunehmend beherrschen – ein im Hinblick auf die spätere Weiterentwicklung von SECONDO beachtenswerter Aspekt. Andererseits war zu berücksichtigen, dass die Laufzeiteffizienz von C++-Programmen deutlich höher als die von Java-Programmen ist und dass bereits einige Teile des Systems in C bzw. C++ realisiert wurden. Die Wahl fiel schließlich auf C++, um eine komplette Neuprogrammierung zu vermeiden, die Portierung vorhandener Algebra-Module zu erleichtern und die Performanzvorteile auszunutzen. Bei der Überarbeitung der Module des Query-Prozessors und des Systemkatalogs, die beide in Modula entwickelt wurden, sollte nicht nur auf C++ umgestellt, sondern auch eine objektorientierte und mehrbenutzerfähige Version realisiert werden.

In den folgenden Abschnitten werden zunächst mögliche Varianten der Client/Server-Architektur untersucht und einige Anmerkungen zur Benutzer- und Rechteverwaltung gemacht. Im Anschluss daran wird das bisher verwendete Speichersystem und mögliche Alternativen dazu dargestellt. Schließlich wird die Struktur des Systemkatalogs sowie die Verwaltung von Typen und Operatoren der Algebra-Module analysiert.

2.1 Client/Server-Architektur

Datenbankanwendungen können anhand der unterschiedlichen Aufgaben (Schnittstelle zum Benutzer, Bearbeitung der Benutzeranfragen und Speicherung der Daten) in folgende drei Komponenten aufgeteilt werden:

1. einen *Client*, der die spezifischen Benutzeranforderungen erfüllt (z.B. grafische Anzeige der Daten),

2. einen *Server*, der die Datenbankanfragen bearbeitet (z.B. Bereitstellung der Ergebnismenge einer Auswahl, Aktualisierung der Daten usw.), sowie

3. einen *Speichermanager*, der die – in der Regel persistente – Speicherung der Daten verwaltet.

2.1.1 Netzwerkarchitektur

Die drei Komponenten Client, Server und Speichermanager können in einem Rechnernetzwerk auf unterschiedliche Weise verteilt sein:

Variante 1: Client, Server und Speichermanager bilden eine in sich abgeschlossene Programmeinheit. Somit laufen alle drei Komponenten gemeinsam auf einem Rechner.

Variante 2: Client und Server bilden eine Programmeinheit und laufen gemeinsam auf einem Rechner, während der Speichermanager auf einem separaten Rechner im Netzwerk läuft.

Variante 3: Server und Speichermanager bilden eine Programmeinheit und laufen gemeinsam auf einem Rechner, während der Client auf einem separaten Rechner im Netzwerk läuft.

Variante 4: Client, Server und Speichermanager laufen jeweils auf separaten Rechnern im Netzwerk.

In Abbildung 2.1 auf der nächsten Seite sind die vier unterschiedlichen Varianten der Verteilung der Komponenten in einem Rechnernetzwerk schematisch dargestellt. Je nach Verteilung ergeben sich daraus diverse Vor- und Nachteile (siehe Tabelle 2.1 auf der nächsten Seite).

Für den Einzelbenutzerbetrieb bietet sich Variante 1 an, da eine aufwendige Netzwerkkommunikation entfällt. Für den Mehrbenutzerbetrieb hingegen ist in der Regel Variante 3 die günstigste Realisierungsform, da die Kommunikation zwischen Server und Speichermanager besonders effizient ist.

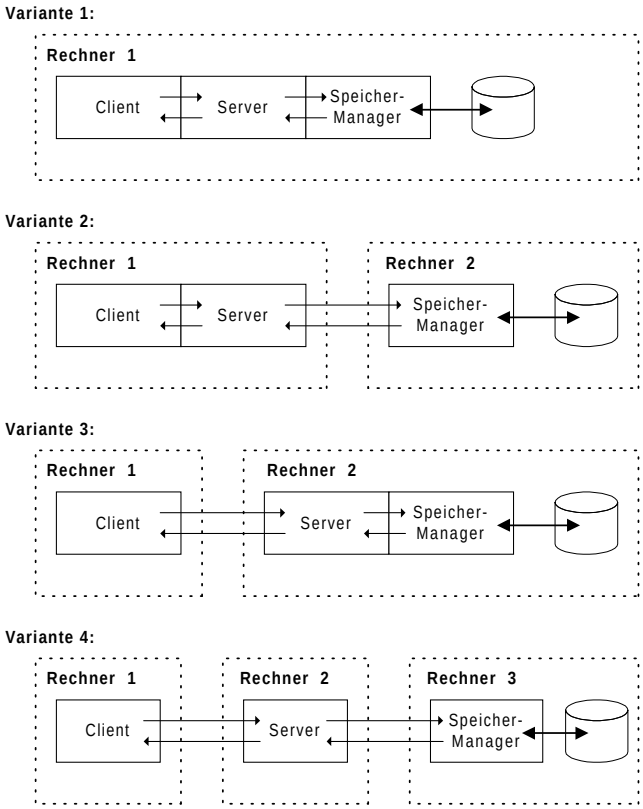

Abbildung 2.1: Verteilung der Komponenten im Netzwerk

Variante	Vorteile	Nachteile
1	keine Netzwerkkommunikation erforderlich, gut geeignet für Einzelbenutzerbetrieb	ungeeignet für Mehrbenutzerbetrieb
2	einfache Kommunikation zwischen Client und Server	„große“ Clients (kompletter Servercode muss eingebunden werden); hohe Netzlast, da alle vom Server zu verarbeitenden Daten über das Netzwerk transportiert werden müssen
3	„kleine“ Clients, effiziente Kommunikation zwischen Server und Speichermanager, gut geeignet für Mehrbenutzerbetrieb	komplexer Server, da für reibungslosen Mehrbenutzerbetrieb gesorgt werden muss
4	prinzipiell verteilte Datenspeicherung möglich, ansonsten wie Variante 3	komplexer Server, hohe Netzlast

Tabelle 2.1: Vor- und Nachteile der verschiedenen Client/Server-Varianten

2.1.2 Server-Architektur

Da ein Datenbanksystem wie SECONDO im Allgemeinen nicht nur für einen einzelnen Benutzer zur Verfügung stehen soll, ist die Konstruktion eines Datenbankservers für die gleichzeitige Bedienung einer größeren Anzahl von Anwendern unabdingbar. Für die Konzeption der Architektur eines solchen Servers ergeben sich verschiedene Möglichkeiten:

- Ein sogenannter *iterativer* Server verarbeitet die Anfragen der Clients sequentiell nacheinander. Da kein neuer Client bedient werden kann, bevor die Bedienung des aktuellen Clients abgeschlossen ist, stellt diese Variante nur in seltenen Fällen eine geeignete Realisierungsform dar.

- Ein sogenannter *paralleler* Server kann die Anfragen mehrerer Clients gleichzeitig, d.h., parallel zueinander, bearbeiten. Hierzu wird entweder für jeden Client ein eigener Thread (Abbildung 2.2) oder aber ein eigenständiger Prozess (Abbildung 2.3 auf der nächsten Seite) gestartet. In den Abbildungen wird neben der Bedienung der Clients auch die Verbindung zum Speichermanager am Beispiel der BERKELEY DB angedeutet. Für die Beurteilung der Varianten ist die Art dieser Verbindung von erheblicher Bedeutung.

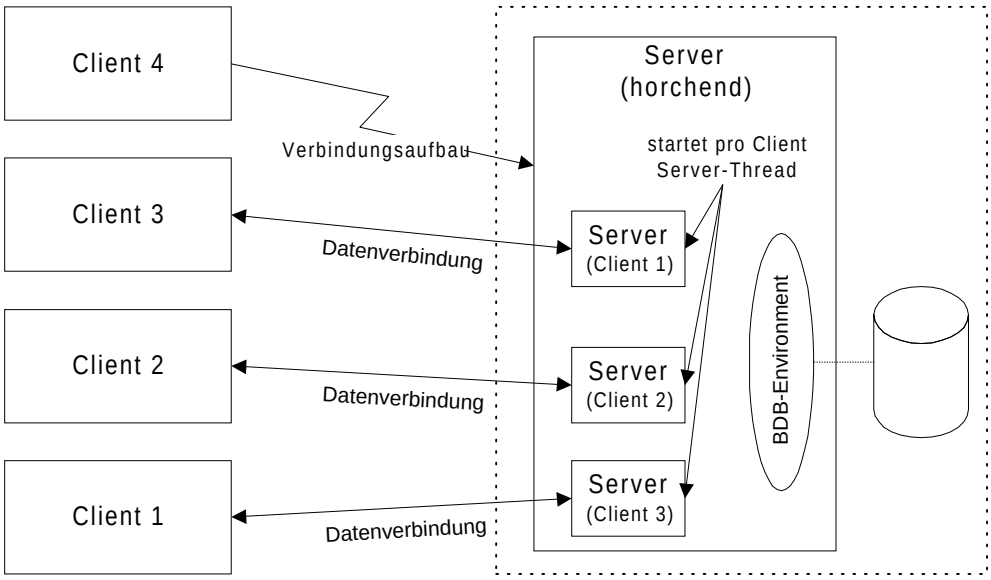

Abbildung 2.2: Paralleler Server, separater Thread für jeden Client

Mischformen der beiden genannten Varianten sind prinzipiell möglich und kommen in der Praxis auch zum Einsatz. Ist die Anzahl der Clients beispielsweise hoch, während die Anforderungen jedes einzelnen Clients jedoch gering und über größere

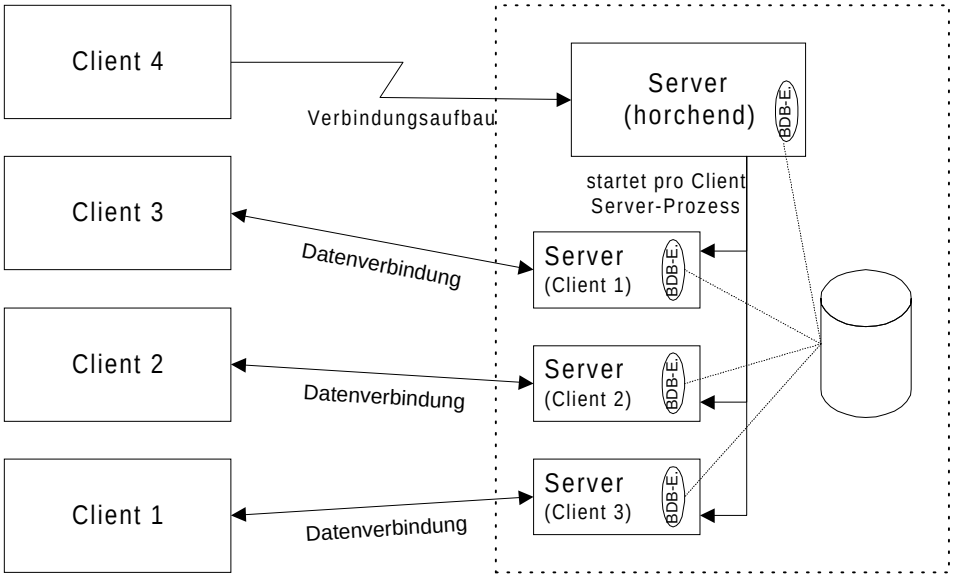

Abbildung 2.3: Paralleler Server, separater Prozess für jeden Client

Zeitspannen verteilt sind, so kann ein iterativer Server in einem solchen Fall durchaus effizient arbeiten.

Für die Bedienung der Clients jeweils einen eigenen Thread im Server zu starten, hat den Vorteil, dass die Erzeugung eines Threads erheblich schneller und Systemressourcen schonender durchgeführt werden kann als die Erzeugung eines eigenständigen Prozesses. Zudem können Threads leichter miteinander kommunizieren, da ihnen ein gemeinsamer Speicheradressraum zur Verfügung steht.

Die Nutzung gemeinsamer Speicherbereiche birgt allerdings die Gefahr, dass sich Threads in unerwünschter Weise gegenseitig beeinflussen und stören können. Dies wird noch dadurch verschärft, dass sich – zumindest bei Verwendung des BERKELEY DB-Speichermanagers – alle Threads eine gemeinsame Speichermanager-Umgebung teilen müssen, über die die Zugriffe auf den persistenten Speicher koordiniert werden. Der Absturz eines Threads kann somit den gesamten Server-Prozess in Mitleidenschaft ziehen.

Wenn für die Bedienung der Clients unabhängige Server-Prozesse erzeugt werden, dann ist dies zwar eine vergleichsweise *teure* Operation und erschwert die Kommunikation dieser Prozesse untereinander. Es entfallen jedoch die bei der Verwendung von Threads geschilderten Gefahren. Jeder Prozess läuft in einem eigenen Adressraum und verfügt über eine eigene Speichermanager-Umgebung. Gleichzeitige, dieselben Daten ändernde Zugriffe durch mehrere Prozesse werden durch den Speichermanager verhindert.

Neben den genannten Vor- und Nachteilen müssen bei der Entscheidung, ob der SECONDO-Server auf der Basis von Threads oder Prozessen realisiert werden sollte, auch Fragen der Portabilität berücksichtigt werden. Threads werden in verschiedenen Betriebssystemen zum Teil sehr unterschiedlich behandelt. LINUX unterscheidet sich beispielsweise von den meisten anderen Betriebssystemen darin, dass Threads wie eigenständige Prozesse behandelt werden [O'S97, Wal97]. Zwar definiert der POSIX-Standard eine für die meisten UNIX-Systeme gültige Thread-Schnittstelle, aber in dieser fehlen wichtige Aspekte wie Unterbrechung und Reaktivierung von Threads [But97]. Die neueren 32-Bit-WINDOWS-Systeme unterstützen zwar Threads; diese entsprechen jedoch nicht dem POSIX-Standard.

Da die Wahrung der Portabilität des SECONDO-Quellcodes bei der Verwendung von Threads erheblich erschwert, wenn nicht gar unmöglich gemacht wird, wird in der vorliegenden Arbeit die *klassische* Server-Realisierung mit eigenständigen Prozessen gewählt.

Bezüglich der Netzwerkarchitektur (vgl. Seite 7) ergibt sich bei einer Realisierung des Speichermanagers auf Basis der BERKELEY DB die Variante 3, in der der Speichermanager integraler Bestandteil des Servers ist. Dahingegen ergibt sich bei Verwendung eines relationalen Datenbankmanagementsystems (RDBMS) für die persistente Datenspeicherung in der Regel Variante 4, auch wenn das RDBMS auf dem selben Rechner wie der SECONDO-Server laufen sollte. Letzteres wäre aus Performanzgründen zu bevorzugen.

In einem System, das unter hoher Last betrieben wird (d.h., wenn sehr viele Client-Anfragen innerhalb einer kurzen Zeitspanne zu verarbeiten sind), kann das Starten eines Threads bzw. eines Prozesses bei Eintreffen einer Client-Anfrage die Performanz erheblich beeinträchtigen. In solchen Umgebungen kann durch vorausschauendes Erzeugen zusätzlicher Threads oder Prozesse die Problematik zumindest gemildert werden. Allerdings fällt hierdurch ein höherer Verwaltungsaufwand an, da die eintreffenden Client-Anfragen einem freien Thread oder Prozess zugeordnet werden müssen.

2.1.3 Benutzer- und Rechteverwaltung

Im bisherigen SECONDO-System existiert keinerlei Benutzer- oder Rechteverwaltung. Fragen der Autorisierung und Authentifizierung spielen jedoch in modernen Datenbanksystemen eine wichtige Rolle.

In den meisten produktiven Systemen ist es erforderlich zu registrieren, welcher Benutzer die Daten erzeugt bzw. bearbeitet hat, und zu überwachen, dass nur autorisierte Benutzer lesend und/oder ändernd auf die Daten zugreifen können. In

kommerziellen Datenbanksystemen stehen hierfür leistungsfähige Benutzer- und Rollenverwaltungsfunktionen zur Verfügung.

Im Rahmen dieser Arbeit soll zwar keine Benutzer- oder Rechteverwaltung eingeführt und implementiert werden, jedoch sollen nach Möglichkeit die von einer Einführung betroffenen Bereiche und Funktionen identifiziert, registriert und wenn möglich gekapselt werden, so dass eine Ergänzung um diese Funktionalität mit begrenztem Aufwand umgesetzt werden kann.

2.1.4 Schlussfolgerung

Im ersten Ansatz wird eine klassische Client/Server-Architektur implementiert, bei der für jeden anfragenden Client ein dedizierter Server-Prozess gestartet wird. Auf die Verwendung von Multi-Threading wird zugunsten der Portabilität und wegen der damit verbundenen Einschränkungen bei der Verwendung der BERKELEY DB verzichtet.

Zur Steigerung der Performanz können spätere Implementierungen z.B. vorausschauend Server-Prozesse starten oder eine Architektur, wie sie von ORACLE bekannt ist (siehe Bild 2.5 auf Seite 21), einsetzen. Die Realisierung solcher Erweiterungen ist jedoch nicht Gegenstand dieser Arbeit.

2.2 Speichersystem

Für jedwedes Datenbanksystem kommt natürlich der persistenten Speicherung von Informationen entscheidende Bedeutung zu. Um die Anforderungen an ein neues Speichersystem zu definieren und Kriterien für seine Eignung aufzustellen, ist es erforderlich, zunächst die Leistungsmerkmale und Schwachpunkte des bisher in SECONDO verwendeten Speichersystems zu beleuchten.

Nach einer kurzen Vorstellung des Projektes SHORE, das als Basis des derzeitigen Speichersystems in SECONDO dient, wird ab Seite 14 beschrieben, welche seiner Eigenschaften in SECONDO konkret genutzt werden.

Vor dem Hintergrund dieses Wissens werden die Leistungsmerkmale zweier Alternativen für das Speichersystem – der BERKELEY DB ab Seite 16 und eines relationalen Datenbanksystems ab Seite 20 – genauer untersucht.

2.2.1 SHORE

SHORE (Scalable Heterogeneous Object REpository) wurde in den frühen 90er Jahren an der Universität von Wisconsin entwickelt [CDF+94]. Es ist für die Plattformen SOLARIS, LINUX und WINDOWS NT 4.0 verfügbar. Die offizielle Weiterentwicklung wurde 1997 beendet. Allerdings wurden im Januar und Oktober 2001 sowie im Februar 2002 neue Zwischenstände veröffentlicht, die eine Vielzahl an Verbesserungen bezüglich Performanz, Robustheit und Unterstützung für aktuelle Compiler, insbesondere gcc-2.95.2, mit sich bringen.

Charakteristisch für SHORE ist, dass alle Objekte typisiert sein müssen. Für die Definition von Objekten steht die *Shore Data Language* (SDL) zur Verfügung, die auf der *Object Data Language* (ODL) der Object Database Management Group (ODMG) basiert. SDL unterstützt die sprachunabhängige Beschreibung von objektorientierten Datentypen. Diese Beschreibungen werden in SHORE abgelegt und verwaltet, so dass Anwendungen Objekte mit Hilfe typsicherer Referenzen bearbeiten können.

Client/Server-Architektur und Mehrbenutzerbetrieb

SHORE besteht aus einem Server-Prozess, der auf jedem Rechner mit SHORE-Anwendungen laufen muss, und einer Client-Bibliothek, die in jede dieser Anwendungen eingebunden werden muss. Einen Überblick über die Prozessarchitektur gibt Abbildung 2.4 auf der nächsten Seite.

SHORE-Server agieren in einem Rechnernetz mittels Peer-to-Peer-Kommunikation. Jeder SHORE-Server verwaltet lokal einen Seitenspeicher, der Seiten aus lokalen Dateien – sofern vorhanden – und Seiten von entfernten Servern enthalten kann. Datenobjekte, die nicht lokal vorhanden sind, werden von einem anderen SHORE-Server angefordert.

Außerdem bietet SHORE für den Mehrbenutzerbetrieb Unterstützung in Form von Transaktionskontrolle, Sperrmechanismen und Logging für die Wiederherstellung nach Systemstörungen. Zusätzlich werden Sicherheitsaspekte berücksichtigt.

Speicherstrukturen

Für die persistente Speicherung unterstützt SHORE aus Datensätzen bestehende Dateien. Jeder Datensatz kann nahezu beliebig groß sein. Der Zugriff auf Datensätze erfolgt über Objektidentifikatoren oder über sequentielles Lesen von Dateien. Weiterhin stehen Implementierungen von B-Bäumen und R-Bäumen zur Verfügung.

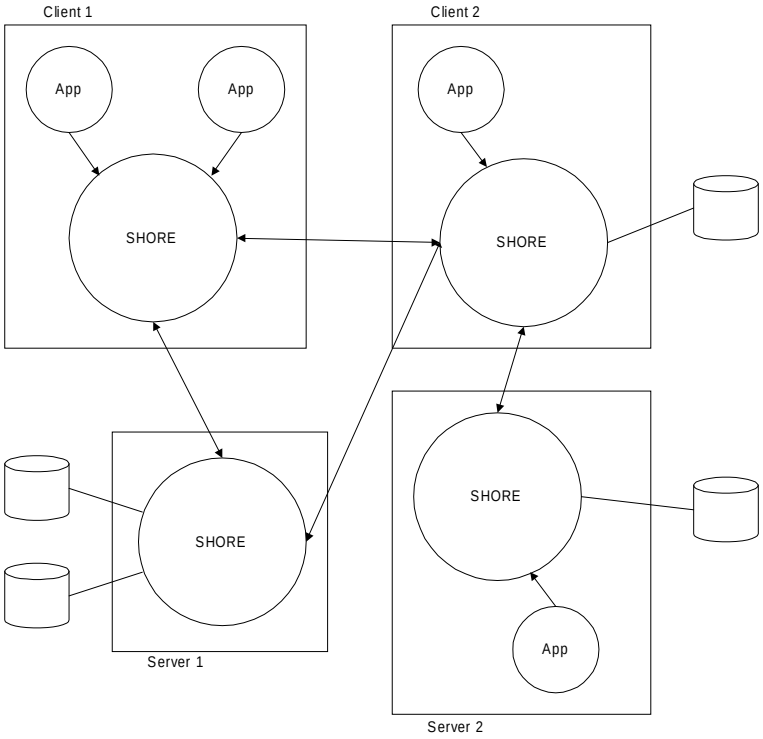

Abbildung 2.4: Prozess-Architektur von SHORE

Auf den Basisstrukturen aufbauend können sogenannte SHORE-Value-Added Server (SVAS) entwickelt werden, die typisierte Objekte und UNIX-ähnliche Verzeichnisstrukturen unterstützen. Beispielsweise gibt es einen SVAS, der das Standard-NFS-Protokoll implementiert, so dass Applikationen SHORE-Objekte wie UNIX-Dateien bearbeiten können.

2.2.2 SECONDO

Das derzeitige Speichersystem von SECONDO baut auf der Funktionalität von SHORE auf, verwendet aber nur einen vergleichsweise kleinen Teil der gesamten SHORE-Funktionalität. Selbst von der zur Zeit im Speichersystem bereitgestellten Funktionalität wird in den übrigen Teilen von SECONDO nur eingeschränkt Gebrauch gemacht.

Für die persistente Speicherung von Daten wird beim Start des Systems ein Festplattenspeicherbereich zugeordnet, innerhalb dessen *Dateien* (Files) verwaltet werden. Unter einer ***Datei*** wird eine Menge von logisch zusammen gehörenden ***Datensätzen*** (Records) verstanden, welche die Nutzdaten enthalten. Falls die SECONDO-Datenbank noch nicht existiert, wird der Festplattenspeicherbereich von und für SHORE allokiert und initialisiert.

Es wird zwischen drei Arten von Dateien unterschieden:

Regular (Standard): Der Zugriff auf die Datei wird protokolliert.

Temporary Es wird kein Protokoll geschrieben. Der Zugriff erfolgt schneller, aber der Effekt eines *Rollbacks* ist undefiniert und nach einem Systemabsturz existiert die Datei nicht mehr.

Load Eine Datei wird zunächst als temporäre Datei angelegt, wird aber zum Zeitpunkt des *Commit* in eine reguläre Datei umgewandelt.

Derzeit werden ausschließlich *reguläre* Dateien verwendet.

Für den konkurrierenden Zugriff auf die Datensätze sind zwar entsprechend den zur Verfügung stehenden Möglichkeiten von SHORE unterschiedliche Modi definiert, jedoch werden für die unterstützten Zugriffsarten nur Standard-Modi verwendet.

Für jede Datei kann ein spezieller Datensatz für die Speicherung von Metadaten genutzt werden. Von dieser Möglichkeit wird im derzeitigen System allerdings kein Gebrauch gemacht.

Ein Datensatz kann nur im Kontext einer Datei verwendet werden und enthält benutzerdefinierte Folgen von Bytes variabler Länge. Die Bearbeitung partieller Datensätze wird unterstützt. Beim Lesen eines Datensatzes wird für diesen eine Sperre eingerichtet, die konkurrierende Zugriffe synchronisiert. Ein sequentielles Lesen aller Datensätze einer Datei ist möglich; die Reihenfolge, in der die Datensätze bereitgestellt werden, ist dabei jedoch nicht definiert.

Neben den genannten Dateien wird eine persistente Speicherung von Paaren aus Schlüsseln und Werten basierend auf B^+-Bäumen bereitgestellt. Diese Funktionalität wird derzeit im SECONDO-Systemkern selbst nicht verwendet, jedoch existieren Algebra-Module, die auf B-Bäume oder R-Bäume in SHORE zurückgreifen. Sowohl Schlüssel als auch Wert können eine variable Größe bis maximal zur Seitengröße[1] haben. Schlüssel können aus beliebigen C-Standardtypen wie `int` oder `float` oder variabel langen Zeichenketten zusammengesetzt sein. Zuordnungen zwischen Schlüsseln und Werten können neu erzeugt oder gelöscht werden. Desweiteren können alle Schlüssel-Wert-Paare, die einer Suchbedingung genügen, in Folge verarbeitet werden.

Da Dateien nur über ihren SHORE-Identifikator angesprochen werden können, wird zur Erhöhung der Benutzerfreundlichkeit ein spezielles Verzeichnis angeboten, über das den SHORE-Identifikatoren ein Name in Form einer Zeichenkette zugeordnet werden kann.

[1] Die Seitengröße in SHORE beträgt je nach Konfiguration 8 bis 64 kB

Das Speichersystem arbeitet transaktionsorientiert. Transaktionen werden implizit automatisch gestartet und entweder regulär beendet (*Commit*) oder abgebrochen (*Abort*). *Commit* macht alle Änderungen seit Beginn der Transaktion persistent, während *Abort* den Zustand vor Beginn der Transaktion wieder herstellt. In beiden Fällen werden bestehende Sperren auf Datensätze aufgehoben.

2.2.3 BERKELEY DB

Die Berkeley-Datenbank (BERKELEY DB) ist ein Datenverwaltungssystem für Anwendungen, die eine hochperformante, mehrbenutzerfähige und ausfallsichere Datenspeicherung benötigen. Das System hat sich in unterschiedlichsten Umgebungen auch im 24-Stunden-7-Tage-Betrieb bewährt. Durch ihre Skalierbarkeit ist die BERKELEY DB sowohl für Mehrbenutzerbetrieb mit großen Datenmengen als auch für Einbenutzerbetrieb mit beschränkten Ressourcen geeignet. Die Bibliothek selbst benötigt unter 300 Kilobyte Hauptspeicher, ist jedoch in der Lage, Datenbanken bis zu einer Größe von 256 Terabyte zu verwalten, sofern dies vom unterlagerten Betriebssystem unterstützt wird. Einzelne Datenwerte können eine Länge von bis zu 2^{32} Bytes haben.

Die BERKELEY DB wird als Programmbibliothek für Software-Entwickler bereitgestellt und wird direkt in die jeweilige Applikation eingebunden. Schnittstellen stehen u.a. für die Sprachen C, C++, Perl, Tcl und Java zur Verfügung. Die BERKELEY DB läuft auf einer breiten Palette von Systemen, darunter die meisten UNIX- oder UNIX-ähnlichen Systeme, EMBEDIX, QNX, VXWORKS sowie Microsoft WINDOWS 95/98/NT/2000. [OBS99]

Durch die Einbindung der BERKELEY DB-Bibliothek in eine Anwendung läuft sie im selben Adressraum wie diese. Für die Durchführung der Datenbankoperationen ist somit keine Interprozesskommunikation notwendig, weder lokal noch über Netzwerk. Die BERKELEY DB ist multi-threading-fähig, auch wenn die Bibliothek selbst nicht mit Threads arbeitet.

Architektur

Da sich die BERKELEY DB auf die Verwaltung von Schlüssel/Wert-Paaren beschränkt, ist die Bezeichnung *Datenbank* vielleicht etwas hochtrabend. Das System bietet weder die direkte Unterstützung einer Abfragesprache wie SQL (Structured Query Language), noch verwaltet es Angaben über die Struktur und die Datentypen der gespeicherten Informationen. Die BERKELEY DB ist also weder eine relationale noch eine objektorientierte Datenbank. Dennoch eignet sie sich für die Datenspei-

cherung in komplexen Anwendungssystemen oder auch in Datenbanksystemen. In der – insbesondere im Web-Umfeld populären – Datenbank MySQL werden transaktionsfähige Tabellen u.a. auf der Basis der BERKELEY DB realisiert.

Die BERKELEY DB beinhaltet eine einfache Programmierschnittstelle (API) für die Datenverwaltung. Entwurfsziel der BERKELEY DB war und ist, eine einfache, sichere und performante persistente Datenhaltung zu ermöglichen. Daher wurde nicht auf Industrie-Standard-Schnittstellen wie ODBC (Open DataBase Connectivity), OleDB (Object Linking and Embedding for DataBases) oder SQL gesetzt.

Im wesentlichen besteht die BERKELEY DB aus fünf Hauptkomponenten:

Zugriffsmethoden Mit Hilfe der Zugriffsmethoden werden Datenbankdateien erzeugt und verwaltet. Die zur Verfügung stehenden Methoden werden in einem eigenen Abschnitt auf der nächsten Seite ausführlicher dargestellt. Datensätze können sowohl im Ganzen oder in Teilen bearbeitet werden. Die BERKELEY DB unterstützt auch Join-Operationen auf zwei oder mehr Datenbanken.

Speicherpool Diese Komponente stellt alle für die Verwaltung eines gemeinsam genutzten Speicherbereichs benötigten Funktionen zur Verfügung. Dieser Speicher-Cache dient dazu, den gemeinsamen Zugriff mehrerer Prozesse oder Threads auf eine Datenbank zu koordinieren.

Auf Maschinen mit sehr großem Hauptspeicher kann die BERKELEY DB diesen für die Verwaltung von Cache, Log-Sätzen und Sperren verwenden. Entsprechend selten muss auf den Hintergrundspeicher (Festplatte) zugegriffen werden.

Mit Hilfe der BERKELEY DB können Anwendungen, die keine persistente Datenspeicherung benötigen, auch Datenbanken anlegen, die nur im Hauptspeicher gehalten werden. Hierbei entfällt der Verwaltungsaufwand, der ansonsten durch das Ein/Ausgabe-System verursacht würde.

Transaktionen Die Transaktionskomponente erlaubt es, eine Menge von Änderungsoperationen als atomare Einheit zu betrachten, so dass entweder alle oder keine der Änderungen wirksam wird.

Sperren Die Sperrkomponente verwaltet im Mehrbenutzerbetrieb den gesicherten gemeinsamen Zugriff auf die Datenbankdateien, so dass zwei Benutzer gleichzeitig verändernden Zugriff auf Datensätze erlangen können. Die BERKELEY DB kann entweder ganze Datenbanken oder einzelne Seiten innerhalb einer Datenbank sperren. Ein Sperrmechanismus auf Datensatzebene existiert dagegen nicht. Aus diesem Grund sollte die Seitengröße nicht zu groß gewählt werden, damit jede Seite möglichst nur eine kleine Anzahl von

Datensätzen enthält. Standardmäßig wird die Seitengröße passend zur physischen Blockgröße des unterlagerten Dateisystems gewählt. Die Seitengröße ist jedoch keine systemweit festgelegte Größe, sondern kann für jede Datenbank individuell – entsprechend den Belangen der Anwendung – eingestellt werden. Wenn Benutzer Sperranforderungen stellen, die gegenseitig voneinander abhängig sind, kann es zu Systemverklemmungen (Deadlocks) kommen. Deadlocks werden von der BERKELEY DB erkannt und automatisch aufgelöst, indem eine der beteiligten Transaktionen abgebrochen wird.

Logging Mit Hilfe der Logging-Komponente, bei der es sich um ein sogenanntes *write-ahead logging* handelt, wird in der BERKELEY DB das Transaktionskonzept umgesetzt. Darüber hinaus dient die Logging-Komponente zur Speicherung von Informationen, die zur Wiederherstellung eines konsistenten Zustands nach einem Systemabsturz benötigt werden. Dadurch ist die BERKELEY DB im Stande, sowohl Programm- als auch Systemabstürze, ja sogar Hardwarefehler ohne Datenverluste zu überstehen. Damit der Aufwand zur Wiederherstellung im Katastrophenfall nicht überhand nimmt, stellt die BERKELEY DB auch einen Checkpoint-Mechanismus zur Verfügung. Nach einem Systemabsturz müssen Datenbank und Log-Dateien nur ab dem vorletzten Checkpoint betrachtet werden.

Fast alle Komponenten können unabhängig voneinander genutzt werden.

Zugriffsmethoden

Unter einer Zugriffsmethode wird hier die dateibasierte Speicherungsstruktur und die auf dieser verfügbaren Operationen verstanden.

Die BERKELEY DB bietet derzeit vier Zugriffsmethoden an: *B$^+$tree*, *Hash*, *Queue* und *Recno*. Alle Methoden arbeiten auf Datensätzen, die aus einem Schlüssel- und einem Datenwert zusammengesetzt sind. Bei der *B$^+$tree*- und *Hash*-Zugriffsmethode können die Schlüssel eine beliebige Struktur haben, während bei der *Queue*- und *Recno*-Methode jedem Datensatz eine Satznummer zugeordnet wird, die als Schlüssel dient. Der Datenwert kann bei sämtlichen Methoden eine beliebige Struktur aufweisen.

Nachfolgend sollen die Methoden kurz beschrieben werden:

B$^+$tree Diese Zugriffsmethode wird mit Hilfe sortierter, balancierter B-Bäume realisiert. Einfügen, Löschen und Suchen von Datensätzen benötigt einen zeitlichen Aufwand von $O(log_b N)$, wobei b die durchschnittliche Zahl von Schlüsseln pro Seite und N die Gesamtzahl der Schlüssel darstellt.

Hash Diese Zugriffsmethode wird durch erweitertes lineares Hashing implementiert [Lit80]. Einfügen, Löschen und Suchen von Datensätzen benötigt in der Regel nur wenige Zugriffe und ist somit bei großen Satzmengen günstiger als die B^+tree-Methode. Allerdings ist kein Zugriff auf die Datensätze in sortierter Folge möglich.

Queue Diese Zugriffsmethode[2] kann nur zur Speicherung von Datensätzen fester Länge verwendet werden, wobei die Datensätze über eine logische Satznummer identifiziert werden. Sie ermöglicht insbesondere ein schnelles Einfügen am Ende bzw. Löschen am Anfang. Ein wahlfreier Zugriff über die logische Satznummer ist ebenfalls möglich.

Recno Diese Zugriffsmethode erlaubt das Speichern von Datensätzen sowohl fester, als auch variabler Länge. Als Schlüssel werden logische Satznummern verwendet, wobei diese automatisch generiert[3] oder aber vom Anwender vorgegeben werden. Aus Anwendersicht sind die Datensätze mit eins beginnend fortlaufend durchnummeriert. Falls gewünscht, werden die Datensätze automatisch neu nummeriert, wenn ein Datensatz zwischen bestehenden Datensätzen eingefügt oder gelöscht wird. Damit das Einfügen, Löschen und Suchen von Datensätzen effizient abgewickelt werden kann, wird intern zur Speicherung eine B-Baum-Struktur verwendet.[4] Bei Bedarf können die Datensätze auch in einer gewöhnlichen Textdatei abgelegt werden.

Client/Server-Betrieb

Grundsätzlich ist es möglich, einen BERKELEY DB-Server einzusetzen, so dass diverse Clients auch über Netzwerkverbindungen das Speichersystem der BERKELEY DB nutzen können. Die BERKELEY DB enthält für diesen Zweck eine auf dem RPC-Protokoll[5] basierende Client/Server-Implementierung. Allerdings ist diese diversen Einschränkungen unterworfen:

---

[2]Die Queue-Zugriffsmethode bietet als einzige einen Sperrmechanismus auf Satzebene, im Gegensatz zur Sperre von Seiten bei allen anderen Zugriffsmethoden der BERKELEY DB. Im Hinblick auf das Sperrverhalten und die damit verbundene Performanz konkurrierender Zugriffe sollte dies bei der Implementierung der Speicherschnittstelle beachtet werden.

[3]Die automatische Generierung von Satznummern erfolgt, wenn Datensätze an eine Datei **angehängt** werden.

[4]Daraus resultiert leider bei schreibenden Zugriffen eine eingeschränkte Performanz konkurrierender Zugriffe, da nicht nur die Seiten, auf denen sich der Datensatz selbst befindet, sondern auch die Seiten der von der Änderung potentiell betroffenen inneren Knoten gesperrt werden.

[5]RPC = Remote Procedure Call, entwickelt von Sun Microsystems

1. Der BERKELEY DB-Server steht nur für UNIX-Systeme zur Verfügung.

2. Der RPC-Client/Server-Code bietet keine Unterstützung für benutzerdefinierte Vergleichsfunktionen, wie sie etwa für B-Bäume benötigt werden.

3. Der direkte Zugriff auf Sperrmechanismen, Protokollmechanismen oder Pufferbereiche im Speicher ist über RPC nicht möglich.

4. Client und Server müssen die exakt gleiche Version der BERKELEY DB-Bibliothek verwenden. Andernfalls werden die Client-Anfragen vom Server mit einer Fehlermeldung abgewiesen.

5. Der BERKELEY DB-Server verfügt über keinerlei Authentifizierungsmechanismen. Werden diese benötigt, müssen im Client- bzw. Server-Code entsprechende Anpassungen vorgenommen werden.

6. Zur Zeit unterstützt der BERKELEY DB-Server kein Multi-Threading, so dass die Anzahl von Clients, die sinnvoll gleichzeitig bedient werden können, stark eingeschränkt ist.

Die genannten Einschränkungen und Nachteile lassen den Einsatz eines BERKELEY DB-Servers im Kontext von SECONDO nicht sinnvoll erscheinen.

2.2.4 Relationale Datenbank am Beispiel von ORACLE

Auf den ersten Blick erscheint es widersinnig, das Speichersystem eines Datenbanksystems auf der Basis eines bestehenden relationalen Datenbanksystems zu realisieren. Wenn jedoch am Einsatzort von SECONDO bereits ein relationales Datenbanksystem eingeführt ist, für das sowohl Wartung, Systempflege sowie Backup- und Wiederanlaufprozeduren organisiert, als auch geschultes Personal vorhanden ist, dürfte es von Vorteil sein, keine proprietäre Lösung zu verwenden, für die erst die notwendige Infrastruktur geschaffen werden müsste. Daher soll in den folgenden Abschnitten am Beispiel des relationalen Datenbanksystems ORACLE untersucht werden, ob sich die Eigenschaften eines Speichersystems wie der BERKELEY DB mit vertretbarem Aufwand nachbilden lassen.

Architektur

In diesem Abschnitt soll die Architektur von ORACLE nicht in allen Details beschrieben werden, sondern nur in dem Umfang wie es im Kontext der Diplomarbeit sinnvoll und notwendig ist.

ORACLE ist seit der Version 8 ein relationales Datenbankmanagementsystem mit einigen *objekt-relationalen Erweiterungen*[6], das für komplexe Datenbankanwendungen mit Mehrbenutzerbetrieb in verteilten Umgebungen geeignet ist. Eigenschaften der BERKELEY DB, wie z.B. die Unterstützung von Mehrbenutzerbetrieb, Transaktions- und Recoverymanagement, beherrscht ORACLE daher ganz selbstverständlich.

Das objekt-relationale Modell erlaubt dem Anwender sowohl die Struktur eigener Objekttypen als auch die Methoden auf diesen zu definieren und diese Datentypen innerhalb des relationalen Modells zu nutzen. Im Abschnitt *Zugriffsmethoden* weiter unten wird näher untersucht, inwieweit auf solche benutzerdefinierten Datentypen zurückgegriffen werden kann oder muss, um die Zugriffsmethoden der BERKELEY DB nachzubilden.

In einer ORACLE-Datenbanksystemumgebung hat man es stets mit einer Client/-Server-Architektur zu tun. ORACLE erzeugt Server-Prozesse, die die Anfragen von mit ihnen verbundenen Client-Prozessen bearbeiten. Dabei bestehen verschiedene Konfigurationsmöglichkeiten, die sich in der Zahl der durch einen Server-Prozess versorgten Client-Prozesse unterscheiden (siehe Abbildung 2.5). In einer Konfiguration mit dedizierten Servern wird für jeden Client ein eigener Server-Prozess gestartet; in einer Konfiguration mit multi-threaded Servern teilen sich dagegen viele Client-Prozesse eine kleine Anzahl von Server-Prozessen, wodurch in der Regel die Systemressourcen optimaler ausgenutzt werden können.

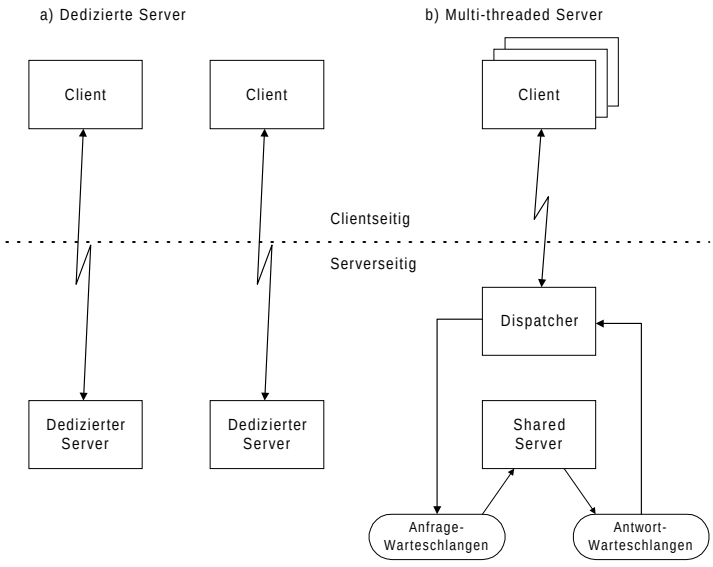

Abbildung 2.5: ORACLE-Architektur: Konfigurationsvarianten

[6]Die objekt-relationalen Erweiterungen stehen mit ORACLE8i – entsprechend Version 8.1.x – in allen Editionen zur Verfügung, während sie in 8.0.x-Versionen nur in der *Enterprise Edition* genutzt werden können.

Zugriffsmethoden

Mit Hilfe der objekt-relationalen Erweiterungen von ORACLE ab Version 8i lassen sich nahezu beliebig komplexe Datenstrukturen abbilden. Allein in dieser Hinsicht bietet ORACLE also erheblich weitreichendere Möglichkeiten als die BERKELEY DB. Hier soll jedoch nur untersucht werden, wie und mit welchem Aufwand die Zugriffsmethoden der BERKELEY DB in ORACLE umgesetzt werden können.

Die *Recno*- und die *Queue*-Zugriffsmethoden lassen sich mit sehr einfachen Tabellen nachbauen, bei denen der Primärschlüssel der logischen Satznummer entspricht. Die Datenwerte können als *Binary Large OBject* (BLOB) abgelegt werden.

Die Umsetzung der Variante der *Recno*-Zugriffsmethode, bei der die logischen Satznummern bei Einfügungen oder Löschungen von Datensätzen umnummeriert werden, lassen sich durch einfache Trigger nach Einfüge- und Lösch-Anweisungen realisieren. Man muss sich allerdings dessen bewusst sein, dass durch diese Trigger $O(n)$ Datensätze aktualisiert werden müssen, also ein erheblicher Zusatzaufwand entsteht.

Für die *B^+tree*- und die *Hash*-Zugriffsmethode ist der Primärschlüssel bei der BERKELEY DB eine beliebige binäre Struktur, für die in der Anwendung eine Vergleichs- bzw. eine Hash-Funktion definiert sein muss, mittels derer die lexikographische Ordnung der Schlüssel ermittelt wird.

ORACLE lässt keine BLOBs als Primärschlüssel zu. Für benutzerdefinierte Datentypen lassen sich zwar Vergleichs- bzw. Abbildungsfunktionen[7] definieren, die aber bedauerlicherweise nur in ORDER-BY-Klauseln genutzt werden können. Es ist jedoch nicht ohne weiteres möglich über benutzerdefinierten Datentypen einen Index aufzubauen. ORACLE bietet zwar sogenannte *Domänen-Indizes* für die Lösung solcher Fragestellungen an, jedoch bedingt dies die Programmierung von *Data Cartridges*, die in etwa mit Algebra-Modulen in SECONDO vergleichbar sind. Der Aufwand hierfür wäre unverhältnismäßig groß. Daher stellt ein Domänen-Index keine dem hier gegebenen Problem angemessene Lösung dar, sondern es muss eine pragmatischere Vorgehensweise gefunden werden.

In der Praxis kommen nur selten Schlüssel mit beliebiger Länge und beliebiger Struktur vor. Daher bietet es sich an, für die wichtigsten Fälle spezielle Lösungen anzubieten. Die Unterstützung von Indizes für Schlüssel mit den Datentypen *Ganz-*

[7] Vergleichsfunktionen bestimmen zu zwei Datenwerten, in welcher Relation sie zueinander stehen; Abbildungsfunktionen bilden Datenwerte auf ganze Zahlen ab, bestimmen also ohne Vergleich mit einem anderen Datenwert die Position des betrachteten Datenwerts in der linearen Ordnung der Datenwerte.

zahl, *Gleitkommazahl* oder *Zeichenkette*[8] stellt in ORACLE kein Problem dar. Auch komplex zusammengesetzte Schlüssel lassen sich vergleichsweise einfach behandeln, sofern die Schlüsselkomponenten sich dergestalt auf eine Zeichenkette abbilden lassen, dass für die entstehenden Zeichenketten die lexikographische Ordnung gilt. Eine direkte Abbildung der Schlüsselkomponenten auf entsprechende Indexkomponenten wäre zwar ebenfalls denkbar, würde jedoch im Vergleich zu der Allgemeingültigkeit der BERKELEY DB eine erhebliche Einschränkung darstellen, da die Schlüsselstruktur (ähnlich wie bei SHORE) als Zusammensetzung elementarer Datentypen definiert werden müsste.

In einer ORACLE-Datenbank kann die unterschiedliche Behandlung der B^+*tree*- und *Hash*-Zugriffsmethoden durch entsprechende Attributierung der Indexstruktur nachvollzogen werden. Normalerweise werden Indizes als B-Bäume verwaltet; es kann jedoch auch eine Hash-Struktur angefordert werden.

Wie beschrieben bietet die BERKELEY DB im Client/Server-Betrieb keine volle Unterstützung der B^+*tree*- und der *Hash*-Zugriffsmethoden, da die zugehörigen Vergleichs- und Hash-Funktionen in den Server integriert werden müssten. Da ORACLE stets als Datenbankserver betrieben wird, ist zu überprüfen, ob und wie dieses Problem gelöst werden kann.

Im Gegensatz zur BERKELEY DB bietet ORACLE definierte Schnittstellen, mit denen externe Funktionen eingebunden werden können. Grundsätzlich ist daher die Integration spezifischer Vergleichs- und Hash-Funktionen möglich. Da die Verwendung solcher Funktionen in SECONDO nur durch die Entwickler von Algebren, nicht jedoch durch Anwender erfolgt, müssen nur die Verfahrensweisen und Schnittstellen entsprechend festgelegt werden. In der Schnittstelle zum Speichersystem sind die Systemunterschiede allerdings zu berücksichtigen, da Funktionen in der BERKELEY DB über ihre Adresse, in ORACLE jedoch über ihren Namen bekannt gemacht werden müssen.

2.2.5 Schlussfolgerung

Da mit Hilfe der BERKELEY DB alle im bisherigen SECONDO-Speichersystem verfügbaren Eigenschaften implementiert werden können, soll die BERKELEY DB das künftige Standard-Speichersystem für die persistente Datenhaltung in SECONDO werden. Exemplarisch wird unter WINDOWS eine Variante des Speichersystems auf der Basis von ORACLE8i entwickelt.

Für eine Einzelbenutzerversion unter PALMOS kann die BERKELEY DB bedau-

[8] Zeichenketten (vom Typ VARCHAR2) sind in ORACLE auf maximal 4000 Zeichen begrenzt.

erlicherweise nicht ohne weiteres eingesetzt werden, obwohl sie wegen der Unterstützung rein speicherbasierter Lösungen prinzipiell geeignet wäre. Nachfragen bei Sleepycat Software[9] haben ergeben, dass keine – zumindest keine öffentlich bekannte – Portierung der BERKELEY DB für PALMOS existiert.

Eine Nachfrage bei Prof. Margo Seltzer von der Harvard-Universität aufgrund einer Ankündigung im Internet[10] führte zu der Erkenntnis, dass eine Portierung auch kein triviales Unterfangen darzustellen scheint, da mehrere studentische Teams daran bislang gescheitert sind.

Für die Erstellung einer SECONDO-Version für PALMOS müsste also eine weitere Speichersystem-Version erstellt werden, was den Rahmen dieser Diplomarbeit sprengen würde.

2.3 Systemkatalog

Im bisherigen SECONDO-System dient der Systemkatalog zur Verwaltung von Datentypen, Objekten, Typkonstruktoren, Operatoren und Datenbanken. Informationen zu Datentypen und Objekten werden persistent gespeichert, während Informationen zu Typkonstruktoren, Operatoren und Datenbanken bei Systemstart in den Hauptspeicher geladen und dort verwaltet werden.

Sowohl die persistente Speicherung als auch die Verwaltung im Hauptspeicher wird auf Basis sogenannter kompakter Tabellen (*Compact Tables*) realisiert. Eine kompakte Tabelle ist eine Sequenz von Elementen gleicher Größe, die mit natürlichen Zahlen, beginnend mit 1, indiziert werden. Wie bei einem Vektor kann wahlfrei auf jedes beliebige Element zugegriffen werden. Bei Bedarf wird die Liste dynamisch vergrößert. Frei werdende Elemente werden wiederverwendet, so dass die Speicherung aller Elemente möglichst kompakt erfolgt.

Zu Datentypen, Objekten, Typkonstruktoren, Operatoren und Datenbanken gibt es jeweils eine Indexstruktur – realisiert als ein in einer kompakten Tabelle gespeicherter AVL-Baum – und eine Eigenschaftentabelle. In der Indexstruktur werden die Namen der Elemente und Indizes auf den zugehörigen Eintrag in der Eigenschaftentabelle verwaltet; in der Eigenschaftentabelle werden Attribute der Elemente und Indizes auf ihre Werte verwaltet. Die eigentlichen Werte werden mit Hilfe

[9] Entwicklung und kommerzielle Unterstützung der BERKELEY DB,
`http://www.sleepycat.com`

[10] Angekündigter Seminarbeitrag der Studenten Sanmay Das, Matthew Clar u. Jeffery Enos zum Thema *A Relational DBMS with Transaction Support for the* PALMOS im Rahmen der Vorlesung *CS 265r. Database Systems* von Prof. Margo Seltzer,
`http://icg.harvard.edu/~cs265/sched.html`

geschachtelter Listen (*Nested Lists*) dargestellt. Geschachtelte Listen werden dabei durch vier kompakte Tabellen (für Knoten, numerische Werte, Namen und Texte) repräsentiert.

Die derzeitige Darstellungsweise des persistenten Teils des Systemkatalogs weist insbesondere im Mehrbenutzerbetrieb erhebliche Schwächen auf. Der schreibende Zugriff auf die Verwaltungsinformationen der kompakten Tabellen kann leicht zu Systemverklemmungen führen.

Im Mehrbenutzerbetrieb dürfen Änderungen am Systemkatalog durch einen Benutzer erst nach erfolgreichem Abschluss einer Transaktion für andere Benutzer sichtbar werden. Daraus ergibt sich für die Server-Prozesse, die jeweils die Anforderungen eines Benutzers bearbeiten, die Notwendigkeit, gewünschte Änderungen am Systemkatalog zunächst lokal zwischenzuspeichern und erst bei Beendigung der Transaktion durch eine *Commit*-Anweisung wirksam werden zu lassen. Nur wenn alle Änderungen korrekt durchgeführt werden können, wird die Transaktion erfolgreich abgeschlossen. Andernfalls ist implizit ein *Rollback* durchzuführen.

Wenn Schreiboperationen auf dem Systemkatalog gesammelt am Ende einer Transaktion durchgeführt werden, statt Einträge in der persistenten Repräsentation des Systemkatalogs unmittelbar einzufügen, zu löschen oder zu aktualisieren, kann das Risiko einer Systemverklemmung erheblich gesenkt werden.

Die Speicherung von **Datenobjekten** in SECONDO basiert wesentlich auf der Annahme, dass sie durch ein einzelnes Speicherwort eindeutig beschrieben werden können. Für Objekte einer ganzen Reihe von Datentypen trifft dies auch zu: Ganzzahlen, Gleitkommazahlen (einfacher Genauigkeit), Wahrheitswerte, Relationen (mit Hilfe ihres SHORE-Identifikators) und andere. Es gibt jedoch auch Datentypen, bei denen Objekte dieses Typs beispielsweise allein durch die Adresse ihrer Klasseninstanz identifiziert werden. Vielfach mag es keinen Sinn machen, Objekte dieser Datentypen persistent zu machen, jedoch wird dies derzeit nicht überwacht und kann beim Laden solcher Objekte zu ernsthaften Problemen bis hin zum Systemabsturz führen. Ein Mechanismus, der entweder eine solche Überwachung gewährleistet oder aber eine sichere persistente Speicherung erlaubt, ist daher für die neue Version des Systems unabdingbar.

2.4 Verwaltung von Typen und Operatoren

Typen und Operatoren werden in SECONDO durch Algebra-Module bereitgestellt. Auf dieser Ebene werden Typen und Operatoren zumindest in den C++-Algebren bereits in objektorientierter Weise mit Hilfe entsprechender Klassen verwaltet. Der

Query-Prozessor und der Systemkatalog sind bislang in Modula implementiert, das keine Klassen und Objekte kennt. Daher werden die Adressen der diversen Funktionen, die Typen und Operatoren bereitstellen, in globalen Vektoren gesammelt und über eine einfache Indizierung angesprochen. Kenntnisse über innere Zusammenhänge, etwa bei überladenen Operatoren, gehen dabei zum Teil verloren. Durch die Art der Implementierung können dadurch momentan maximal zehn Algebra-Module geladen werden.

Das Laden von Algebra-Modulen erfolgt im derzeitigen SECONDO-System zum Teil durch statisches, zum Teil aber auch durch dynamisches Einbinden (Linken). Welche Algebra-Module dynamisch geladen werden sollen, wird mit Hilfe einer Konfigurationsdatei spezifiziert. Allerdings ist das dynamische Binden in hohem Maße systemspezifisch: während unter UNIX-Systemen wie LINUX oder SOLARIS das dynamische Binden eine automatische Auflösung offener Referenzen durch den Binder auslöst, muss dies unter WINDOWS explizit durch die Applikation selbst erfolgen oder mit Hilfe von Link-Bibliotheken vorbereitet werden.

Insbesondere im Mehrbenutzerbetrieb bietet das dynamische Binden Vorteile, da gleiche Code-Teile nicht mehrfach in den Hauptspeicher geladen werden, und somit sparsamer mit Systemressourcen umgegangen wird. Außerdem ist ein Binden des Systemkerns nur dann notwendig, wenn sich die Schnittstelle der Algebra-Module ändert.

Da Algebren insbesondere im Query-Prozessor, aber auch im Systemkatalog über Indizes angesprochen werden, deren konkreter Wert von der Reihenfolge, in der die Algebren geladen werden, abhängt, ist allerdings Vorsicht geboten. Da diese Indizes auch in der persistenten Datenrepräsentation Eingang finden, ist die Konsequenz dieser Vorgehensweise nämlich, dass stets die gleichen Algebren in exakt gleicher Reihenfolge geladen werden müssen, damit es nicht zu schwer identifizierbaren Fehlern kommt.

Das dynamische Einbinden von Algebra-Modulen kann also nur dann sinnvoll und ohne Risiko eingesetzt werden, wenn begleitende Maßnahmen sicherstellen, dass einem Algebra-Modul unabhängig von der Initialisierungsreihenfolge stets der gleiche Index zugeordnet wird.

2.5 Zusammenfassung

Bei der Überarbeitung von SECONDO wird eine einheitliche Programmierung in C++ angestrebt. Auch die in C vorliegenden Teile sollten dabei – soweit möglich – im C++-Modus des Compilers übersetzt werden, um u.a. aufgrund strikterer Typ-

prüfungen zu robusteren Programmen zu gelangen.

Es werden parallel Versionen für die Betriebssysteme LINUX und WINDOWS entwickelt. Darüberhinaus sollte nach Möglichkeit die Lauffähigkeit unter SOLARIS gewährleistet werden. Aus den in Abschnitt 2.2.5 auf Seite 23 genannten Gründen wird auf die Unterstützung von PALMOS im Rahmen dieser Diplomarbeit verzichtet.

Das Speichersystem wird in erster Linie auf Basis der BERKELEY DB realisiert. Die Realisierbarkeit einer rein speicherbasierten Fassung mit Hilfe der BERKELEY DB wird geprüft, jedoch nicht implementiert. Prototypisch wird unter WINDOWS auch eine auf einer ORACLE-Datenbank basierende Version des Speichersystems erstellt, soweit dies technisch möglich ist.

Neben Einbenutzer-Versionen mit bzw. ohne Transaktionsunterstützung wird eine mehrbenutzerfähige Client/Server-Version von SECONDO erstellt. Die Benutzerschnittstelle wird in allen Varianten textbasiert (TTY) realisiert. Eventuell erforderliche Anpassungen in der vorhandenen Java-Version der Benutzerschnittstelle sind nicht Gegenstand dieser Diplomarbeit.

Der Systemkatalog wird dahingehend überarbeitet, dass ein möglichst reibungsloser Mehrbenutzerbetrieb von SECONDO gewährleistet ist. Die Verwaltung von Algebren mit ihren Typen und Operatoren wird dabei in objektorientierter Weise realisiert. Außerdem ist die sichere persistente Speicherung von Objekten hinsichtlich der Zuordnung ihrer Datentypen zu Algebren zu gewährleisten.

Anpassungen der verschiedenen Algebren wie Standard-C++, Relation-C++, Math, StDB usw. auf das neue Speichersystem sowie den überarbeiteten Systemkatalog fallen – nach Aufgabenstellung – eigentlich nicht zu den im Rahmen dieser Diplomarbeit zu erledigenden Aufgaben. Für Testzwecke wird jedoch die Standard-C++-Algebra sowie die Function-C++-Algebra angepasst. Ebenso werden nur die Teile der *Werkzeuge* (vgl. Abbildung 1.1 auf Seite 2), die für den Systemkern unabdingbar sind, überarbeitet.

Insbesondere wird der vor allem für die Relation-Algebra wichtige *Tupelmanager* **nicht** im Rahmen dieser Diplomarbeit auf das neue Speichersystem umgestellt, da sich bei der Analyse dieser Komponente zwei Schwachpunkte herausgestellt haben, deren Beseitigung mit erheblichem Aufwand verbunden wäre. Die derzeitige Implementierung hat sowohl im Hinblick auf Hauptspeicher als auch auf Plattenspeicher Speicherlecks, da Referenzen auf belegten Speicher verloren gehen können. Außerdem funktioniert die persistente Speicherung von C++-Klassenobjekten nur für einfach strukturierte Klassen, die keine dynamisch allokierten Membervariablen enthalten.

Kapitel 3

Entwurf

Auf der Basis der Analyse-Ergebnisse werden in den folgenden Abschnitten Konzepte für die verschiedenen Systemkomponenten erarbeitet und die erforderlichen Systemeigenschaften präzisiert. Zunächst wird ein Entwurf der Client/Server-Architektur vorgestellt. Die Anforderungen an das Speichersystem unter Berücksichtigung der eingesetzten Basis-Software (BERKELEY DB bzw. ORACLE) werden im Anschluss daran ausführlich diskutiert. Zu guter Letzt werden die gewünschten Eigenschaften des Systemkatalogs und der Algebra-Verwaltung beleuchtet.

3.1 Client/Server-Architektur

Das neue Client/Server-System für SECONDO basiert auf einer Reihe von Komponenten, die hier zunächst einmal kurz vorgestellt werden sollen:

Monitor Die Steuerung des Gesamtsystems liegt in der Verantwortung des *Monitors*. Dieses Programm erlaubt es, das SECONDO-System zu starten, während des Betriebs Statusinformationen anzuzeigen und schließlich das System gesichert zu beenden.

Registrar Die Verwaltung von Statusinformationen über angemeldete Benutzer, verwendete Datenbanken und Systemmeldungen obliegt dem *Registrar*. Er dient dazu, eine einheitliche Schnittstelle für globale Informationen über das System für alle übrigen Komponenten bereitzustellen. Weiterhin registriert und kontrolliert er die parallelen Zugriffe auf SECONDO-Datenbanken.

Checkpoints Bei Verwendung der BERKELEY DB als Speichersystem ist es erforderlich, in regelmäßigen Abständen Kontrollpunkte zu erzeugen, um die Archivierung von Log-Dateien und den Aufwand des Wiederanlaufs nach

eventuellen Systemstörungen zu minimieren. Für diese Aufgabe ist die Komponente *Checkpoints* zuständig. Für das ORACLE-basierte Speichersystem ist *Checkpoints* nicht erforderlich.

Listener Für die Entgegennahme von Verbindungsanforderungen stellt der *Listener* einen Kommunikationskanal zur Verfügung, dessen Adresse allen Benutzern des Systems bekannt zu machen ist. Als rudimentärer Zugriffsschutz für das System werden beim Verbindungsaufbau die Client-IP-Adressen gegen eine Liste zugelassener IP-Adressen bzw. -Adressbereiche geprüft. Clients mit nicht-autorisierten IP-Adressen werden abgewiesen.

Server Alle Anforderungen eines Benutzers werden durch einen dedizierten *Server* erfüllt, der nach erfolgreichem Verbindungsaufbau durch den *Listener* gestartet wird.

Client Wenn Benutzer mit dem System arbeiten wollen, bietet ihnen der *Client* die hierfür benötigte Schnittstelle. Zunächst versucht der *Client* eine Verbindung zum *Listener* herzustellen. Falls die Verbindung aufgebaut werden kann, werden die Benutzeranforderungen an den *Server* zur Verarbeitung geleitet und die Ergebnisse von dort abgeholt. Prinzipiell können die *Clients* sehr vielgestaltig sein: von einer einfachen kommandozeilenorientierten Anwendung bis zur Spezialanwendung mit graphischer Oberfläche.

Das Zusammenwirken dieser Komponenten wird in Abbildung 3.1 auf der nächsten Seite dargestellt. Der dynamische Ablauf gliedert sich in folgende Schritte:

1. Der Systemverwalter startet zunächst auf dem Serverrechner den *Monitor*. Durch Eingabe des entsprechenden Kommandos wird das System aktiviert. Hierzu gehört bei Verwendung der BERKELEY DB als Speichersystem auch ein eventuell notwendiger Wiederanlauf der BERKELEY DB-Umgebung nach einer Systemstörung, da der Wiederanlauf isoliert – ohne andere parallele Zugriffe auf die BERKELEY DB-Dateien – erfolgen muss.

2. Da der *Registrar* wichtige Dienste für die Koordinierung des Zugriffs auf die SECONDO-Datenbanken bereitstellt, wird er vom *Monitor* vor allen anderen Diensten gestartet.

3. Falls das Speichersystem auf der BERKELEY DB basiert, muss als nächstes der *Checkpoints*-Dienst aktiviert werden. Bei anderen Speichersystemen kann dieser Schritt entweder entfallen (z.B. bei ORACLE) oder wird durch andere Maßnahmen ersetzt.

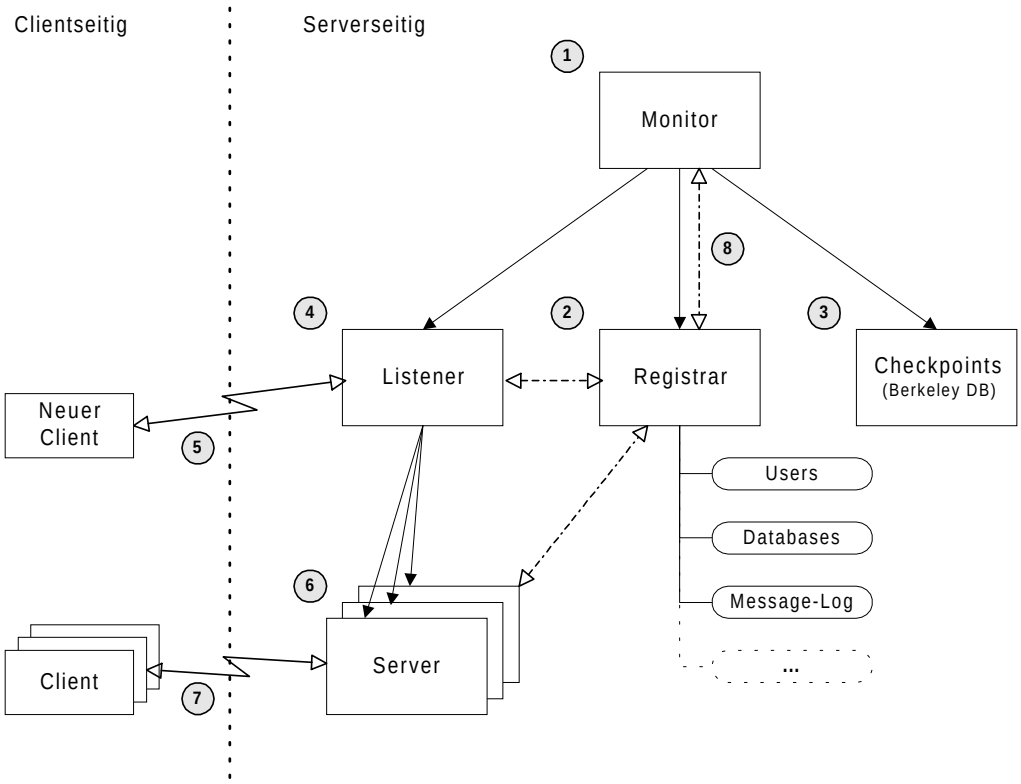

Abbildung 3.1: Client/Server-Architektur

4. Um schließlich Verbindungsanforderungen entgegennehmen zu können, wird der *Listener* gestartet, der auf dem wohlbekannten Kommunikationskanal auf Verbindungsanforderungen wartet.

5. Wenn eine Verbindungsanforderung von einem *Client* eintrifft, nimmt der *Listener* diese entgegen und überprüft zunächst die formale Zulässigkeit, indem z.B. die Zulassung der IP-Adresse des *Clients* überprüft wird.

6. Nach erfolgreicher Überprüfung der Zulässigkeit einer Verbindung startet der *Listener* für den *Client* einen eigenen *Server*-Prozess.

7. Die weitere Bearbeitung der Anforderungen des *Clients* übernimmt jeweils ein *Server* pro *Client*. Zu Beginn könnte der Server eine Überprüfung der Identität des Benutzers anhand der Benutzerkennung und des Passworts durchführen. Eine Passwortüberprüfung ist in SECONDO zur Zeit allerdings nicht implementiert.

8. Der Systemverwalter kann mit Hilfe des *Monitors* und des *Registrars* den Systemzustand überwachen. Wenn schließlich das System heruntergefahren werden soll, informiert der *Monitor* zunächst den *Listener*. Dieser nimmt ab diesem Zeitpunkt keine weiteren Verbindungen mehr entgegen und benachrichtigt seinerseits alle von ihm gestarteten *Server*, dass die Bearbeitung der

Client-Anforderungen zum nächstmöglichen Zeitpunkt beendet werden soll. Der *Monitor* wartet auf die Beendigung des *Listeners* und beendet im Anschluss daran den *Registrar* sowie ggf. den *Checkpoints*-Dienst.

3.2 Speichersystem

3.2.1 Namensgebung

Während SHORE stets mit eindeutigen Identifikatoren (32-Bit-Integer) für Dateien und Datensätze arbeitet, werden in der BERKELEY DB Dateien bzw. in ORACLE Tabellen mit Hilfe von Namen (Zeichenketten) identifiziert. Die Portierung vorhandener Code-Teile auf das neue Speichersystem gestaltet sich einfacher, wenn an Identifikatoren auf Basis von 32-Bit-Integer-Zahlen festgehalten wird.

Für die Generierung eindeutiger Identifikatoren zur Benennung „anonymer“ Dateien bzw. Tabellen bietet sich die Verwaltung einer Sequenz an, die bei jedem Lesezugriff hochgezählt wird. In ORACLE werden Sequenzen direkt unterstützt, bei der BERKELEY DB ist die Implementierung über eine Datei für Sequenzen ebenfalls relativ leicht zu realisieren. Wie in der bisherigen Implementierung des Speichersystems auf SHORE-Basis (persistent root index) ist ein Mechanismus vorzusehen, der es ermöglicht, zu einem Identifikator den zugehörigen Namen zu bestimmen.

3.2.2 Speichersystemumgebung

Jede Implementierung des Speichersystems ist von unterschiedlichen Parametern wie etwa Namen von Verzeichnissen, Größenfestlegungen für Speicherseiten, Indexbereiche oder Cache-Bereiche usw. abhängig. In der Schnittstelle des Speichersystems sollten jedoch so wenig Abhängigkeiten von der konkreten Implementierung wie möglich vorhanden sein. Daraus ergibt sich die Notwendigkeit, eine Speichersystemumgebung zu schaffen, die möglichst ohne Eingriff in den Quellcode von außen zur Laufzeit parametrisierbar ist. Hierfür bietet sich eine Konfigurationsdatei an, die ähnlich wie eine *.ini*-Datei unter WINDOWS aufgebaut sein könnte, die also benannte Sektionen aufweist, in denen verschiedene Parameter gruppiert werden können. Um eine feingranulare Parametrisierung zu ermöglichen, ist es sinnvoll, in der Speichersystemschnittstelle die Spezifikation eines Kontextes vorzusehen. Der Name des Kontextes entspricht dabei dem Namen einer Sektion der Konfigurationsdatei.

Für die BERKELEY DB müssen mindestens folgende Angaben – eventuell teilweise kontextabhängig – spezifiziert werden:

- Pfad, wo sich die Datenbankdateien befinden
- Seitengröße
- Cache-Größe

Für ORACLE müssen mindestens folgende Angaben – eventuell teilweise kontextabhängig – spezifiziert werden:

- Bezeichnung der Datenbank (Connect-String o.ä.)
- Benutzerkennung für den Datenbankzugriff
- Passwort zu o.g. Benutzerkennung
- Tablespace-Attribute
- Indexspace-Attribute

3.2.3 Satzorientierte Zugriffsmethoden

Die Modellierung der Klassen in der bisherigen Implementierung ist sehr stark geprägt durch die Eigenschaften von SHORE. Ein Datensatz (record) wird in SHORE zwar im Kontext einer Datei (file) angelegt, ist aber ein völlig eigenständiges Objekt, das allein durch seinen Identifikator eindeutig bestimmt ist und ohne Kenntnisse über die zugehörige Datei bearbeitet werden kann. Abhängig davon, ob ein Datensatz gelesen oder geschrieben wird, wird er mit Hilfe zweier verschiedener Klassen (`SMI_Record` für das Schreiben bzw. `SMI_Pin` für das Lesen) angesprochen. Diese Aufteilung wurde vorgenommen, weil in SHORE der Zugriff auf die Datensätze direkt über die zugehörigen Datenpuffer im Hauptspeicher abgewickelt wird, die für die Dauer des Zugriffs vor einer Verlagerung im Hauptspeicher geschützt werden müssen (in SHORE-Terminologie „Pinnen“). Während dieser Schutz beim Schreiben implizit durch SHORE aktiviert wird, muss dies beim Lesen explizit durchgeführt werden. Die Aufteilung der Lese- und Schreibaktivitäten in zwei verschiedene Klassen ist trotzdem nur bedingt nachvollziehbar, da die unterschiedliche Behandlung auch durch Einführung zusätzlicher Methoden hätte gelöst werden können.

Während in der BERKELEY DB Datensätze im Grunde keine eigenständige Rolle spielen, liegen die Verhältnisse in ORACLE zumindest im Falle von *BLOBs* etwas anders, da in einem Datensatz nur Referenzen auf *BLOBs*, nicht jedoch ihre eigentlichen Daten gespeichert werden. Um auf die Inhalte eines *BLOBs* zugreifen

zu können, muss zunächst die Referenz auf den Inhalt beschafft werden, um anschließend Lese-, Schreib- und sonstige Funktionen darauf ausführen zu können. Da die Speichersystemschnittstelle für beide Implementierungsvarianten gleichermaßen geeignet sein soll, erscheint es daher gerechtfertig, eine eigenständige Datensatzklasse beizubehalten.

In der BERKELEY DB werden die Dateien, in denen die Datensätze gespeichert sind, als *Datenbanken* bezeichnet. Im Gegensatz dazu spricht man in ORACLE von *Tabellen*. Um für das Speichersystem eine einheitliche Sprechweise zu haben, wird hierfür der Oberbegriff *SmiFile* eingeführt. Nachfolgend sind die benötigten Eigenschaften satzorientierter *SmiFiles* aufgelistet:

Erzeugen eines *SmiFile* Es ist zwischen anonymen und benannten *SmiFiles* zu unterscheiden. Es muss festgelegt werden, ob die Datensätze feste oder variable Länge haben, da sich hierdurch Optimierungsmöglichkeiten ergeben können. Die feste Länge der Datensätze ist zu spezifizieren, während die maximale variable Länge nicht *a priori* festgelegt werden muss. Schließlich sollte der Kontext des *SmiFile* angegeben werden.

Die Schnittstellenklassen des Speichersystems verwalten keine eigenen Datensatzpuffer, sondern transferieren nur Daten zwischen Hauptspeicherbereichen der Applikation und der Implementierung des Speichersystems, also der BERKELEY DB bzw. von ORACLE. Es liegt in der Verantwortung der Applikation, ein Überschreiten der maximalen Puffergrößen zu vermeiden.

Öffnen eines *SmiFile* Für anonyme *SmiFiles* ist ihr Identifikator, für benannte *SmiFiles* ihr Name anzugeben. Falls von den Standardvorgaben abgewichen wird, sind ggf. weitere Parameter wie Kontext, Datensatztyp und Datensatzlänge zu spezifizieren.

Datensatzzugriff Für den Zugriff auf die Datensätze eines *SmiFile* sind Methoden zum Erzeugen neuer Datensätze, Selektieren sowie Löschen vorhandener Datensätze erforderlich. Methoden für das Lesen, Schreiben bzw. Aktualisieren kompletter oder partieller Datensätze werden in einer speziellen Datensatz-Klasse untergebracht.

Beim Selektieren eines Datensatzes ist eine Angabe darüber, ob der Datensatz in der Folge aktualisiert werden soll, erforderlich, da in diesem Fall sofort eine Schreibsperre gesetzt werden kann, wodurch potentielle Blockaden eher vermieden werden können.

Iterator Zur Verarbeitung aller Datensätze eines *SmiFile* muss ein Iterator zur Verfügung stehen, mit dem sukzessiv ein Datensatz nach dem anderen bearbeitet werden kann. Es muss feststellbar sein, ob weitere Datensätze vorhanden

sind. Der aktuelle Datensatz muss aktualisiert (geschrieben) werden können. Eine Repositionierung auf den Anfang ist durchführbar.

Attribute Auf den Namen des *SmiFile* und des Kontextes, den Identifikator, die (maximale) Satzlänge und den aktuellen Zustand kann lesend zugegriffen werden. Ein Umbenennen eines *SmiFile* wird zur Zeit nicht unterstützt.

3.2.4 Schlüsselorientierte Zugriffsmethoden

Es bietet sich an, für die verschiedenen Schlüsseltypen (Ganzzahl, Gleitkommazahl, Zeichenkette, zusammengesetzter Schlüssel), die einer linearen Ordnung genügen, eigene Klassen zu definieren.

Da in kommerziellen Datenbanksystemen wie DB2 oder ORACLE die Gesamtlänge von Schlüsseln zum Teil engen Grenzen[1] unterliegt, wird für das Speichersystem in SECONDO ebenfalls eine Begrenzung eingeführt, auch wenn etwa die BERKELEY DB dies nicht erfordern würde.

Die BERKELEY DB kennt keine zusammengesetzten Schlüssel. Wenn die Satzschlüssel eine innere Struktur aufweisen, die von den Funktionen der BERKELEY DB berücksichtigt werden soll, so muss eine Vergleichsfunktion für ein Paar von Schlüsseln bereitgestellt werden, die feststellt, in welcher Relation die Schlüssel zueinander stehen. Die Nachbildung dieser Verfahrensweise in ORACLE würde eine aufwendige Programmierung extern einzubindender Funktionen nach sich ziehen. Eine Vereinfachung der Behandlung zusammengesetzter Schlüssel kann jedoch dadurch erreicht werden, dass durch den Entwickler einer SECONDO-Algebra für diesen Fall Abbildungsfunktionen bereitgestellt werden, die zusammengesetzte Schlüssel in Zeichenketten und umgekehrt umwandeln. Die Zeichenketten müssen dabei einer lexikographischen Ordnung genügen. Auf diese Weise kann die aufwendige Erstellung zusätzlicher Funktionen für die verschiedenen Speichersysteme vermieden werden.

Nachfolgend sind die benötigten Eigenschaften schlüsselorientierter *SmiFiles* aufgelistet:

Erzeugen eines *SmiFile* Wie bei den satzorientierten Zugriffsmethoden ist zwischen anonymen und benannten *SmiFiles* zu unterscheiden. Auch der Kontext des *SmiFile* soll spezifizierbar sein. Weiterhin sollte festgelegt werden, ob die Schlüssel eindeutig sein müssen oder ob Duplikate erlaubt sein sollen.

[1] In DB2 darf die Gesamtlänge aller Spalten in einem Index nicht mehr als 255 Byte betragen; in ORACLE ist die Gesamtlänge zwar nicht so eng begrenzt, sondern nur die Anzahl der Spalten auf ca. 30 je Index; indizierbare Zeichenketten können allerdings maximal 4000 Byte lang sein.

Öffnen eines *SmiFile* Es werden die gleichen Eigenschaften wie bei satzorientierten Zugriffsmethoden benötigt.

Datensatzzugriff Zusätzlich zu den Eigenschaften, wie sie bei satzorientierten Zugriffsmethoden benötigt werden, muss eine einheitliche Behandlung der verschiedenen Schlüsseltypen gewährleistet werden. Dies kann dadurch geschehen, dass eine polymorphe Schlüsselklasse entworfen wird, die die Unterschiede der Schlüsseldarstellung kapselt, die aber zur Laufzeit eine Feststellung des Schlüsseltyps erlaubt. Im Falle zusammengesetzter Schlüssel muss der Schlüsselklasse die Adresse der zu verwendenden Schüsselabbildungsfunktion mitgegeben werden.

Falls die Schlüssel nicht eindeutig sind, ist das Lesen von Datensätzen nur mit Hilfe eines Iterators möglich.

Iterator Neben der Verarbeitung aller Datensätze wie bei satzorientierten Zugriffsmethoden muss auch für die Verarbeitung ausgewählter Datensätze ein Iterator zur Verfügung stehen. Für die Auswahl der Datensätze kann der Schlüsselbereich dabei sowohl durch eine untere als auch eine obere Grenze eingeschränkt werden. Fehlt die Angabe einer Untergrenze, werden alle Datensätze mit Schlüsselwerten bis zur Obergrenze selektiert; fehlt die Angabe einer Obergrenze, werden alle Datensätze mit Schlüsselwerten ab der Untergrenze selektiert.

Falls Schlüsselduplikate erlaubt sind, werden die Duplikate in nicht näher definierter Reihenfolge gelesen.

Attribute Es werden die gleichen Eigenschaften wie bei satzorientierten Zugriffsmethoden benötigt.

3.2.5 Transaktionen

In einem Datenbankverwaltungssystem sind verschiedene Informationseinheiten gegen konkurrierende Aktualisierungen zu schützen. Dabei ist es häufig erforderlich, mehrere Manipulationen zu Transaktionen zusammenzufassen, so dass entweder jeder oder kein Bestandteil einer Transaktion Gültigkeit erlangt (Commit bzw. Rollback). Zu unterscheiden ist dabei zwischen definierenden (DDL[2]) und manipulierenden (DML[3]) Anweisungen.

[2]Data Definition Language

[3]Data Manipulation Language

Im SQL-Standard ist festgelegt, dass sich eine Commit- oder Rollback-Anweisung stets auf **alle** in einer Transaktion zusammengefassten Anweisungen bezieht, also sowohl auf definierende als auch auf manipulierende. Allerdings wird diese Vorgabe in verschiedenen Datenbanksystemen unterschiedlich gehandhabt.

In ORACLE wird beispielsweise vor und nach einer DDL-Anweisung implizit eine Commit-Anweisung durchgeführt. Dies bedeutet, dass definierende Änderungen nicht durch eine Rollback-Anweisung rückgängig gemacht werden können. Außerdem wird eine offene Transaktion dadurch implizit mit einer Commit-Anweisung abgeschlossen.

In der BERKELEY DB verhält es sich ähnlich wie in ORACLE, da definierende Änderungen keiner Transaktion zugeordnet werden können, also ebenfalls nicht durch eine Rollback-Anweisung rückgängig gemacht werden können. Ein Unterschied im Systemverhalten ergibt sich jedoch daraus, dass in Bezug auf manipulierende Anweisungen keine implizite Commit-Anweisung durchgeführt wird.

Für die Realisierung des Speichersystems wird folgendes Verhalten festgelegt, das für jede Implementierung zu gelten hat und konsistent umgesetzt werden muss:

Jede Anweisung, also auch DDL-Anweisungen, soll dem im SQL-Standard definierten Transaktionsverhalten genügen. Daraus ergibt sich für die BERKELEY DB-Version, dass DDL-Anweisungen registriert werden müssen, um im Falle eines Rollbacks ihre Wirkung rückgängig machen zu können. In der ORACLE-Version müssen DDL-Anweisungen in einer separaten Session ausgeführt werden und ebenfalls registriert werden.

Darüber hinaus wird festgelegt, dass jede Anweisung automatisch durch ein Commit abgeschlossen wird, sofern sie nicht zu einem Transaktionsblock gehört, der durch die Anweisungen `BEGIN TRANSACTION` und `COMMIT/ROLLBACK` eingeschlossen ist.

Das Anlegen oder Löschen einer kompletten Datenbank erfolgt außerhalb des Transaktionskonzept, kann also nicht durch ein `ROLLBACK` rückgängig gemacht werden.

Ein Datenbanksystem verarbeitet in der Regel nicht nur Benutzerdaten, sondern für die eigene interne Verwaltung auch Systemdaten. Um Blockaden durch Änderungen an den Systemdaten zu vermeiden, wird ein Zwei-Phasen-Konzept benutzt: während eine Benutzertransaktion aktiv ist, werden nur lesende Zugriffe auf die Systemdaten durchgeführt. Änderungen werden gesammelt bei der Beendigung der Benutzertransaktion durchgeführt, wodurch die Systemdatenstrukturen nur kurzzeitigen Sperren unterworfen sind.

3.3 Systemkatalog

3.3.1 Transaktionsunterstützung

In der bisherigen SECONDO-Version gab es im Mehrbenutzerbetrieb – vermutlich aufgrund der Eigenheiten der verwendeten persistenten kompakten Tabellen (*Compact Tables*) – bei Änderungen am Systemkatalog einige Probleme. Die Verwaltung des Systemkatalogs muss also Änderungsanforderungen derart behandeln, dass die gegenseitige Beeinflussung von Transaktionen korrekt aufgelöst wird.

Wesentlich für einen reibungslosen Mehrbenutzerbetrieb ist, dass während der Bearbeitung einer Transaktion auf den Systemkatalog nur lesend zugegriffen wird und Änderungen erst im Zuge der Abarbeitung der die Transaktion abschließenden *Commit*-Anweisung durchgeführt werden.

Durch eine Transaktion ausgelöste Änderungen am Systemkatalog müssen daher zunächst lokal zwischengespeichert werden. Grundsätzlich sind dabei zwei unterschiedliche Vorgehensweisen möglich:

1. Falls der Systemkatalog vollständig in den Hauptspeicher geladen werden kann, können Änderungen im Verlauf einer Transaktion auf dieser lokalen Kopie durchgeführt werden. Trotzdem wird es sich wohl nicht vermeiden lassen, im Verlauf einer Transaktion bei der Bearbeitung von Anweisungen auch auf den globalen Systemkatalog lesend zuzugreifen, da zwischenzeitlich für die eigene Transaktion relevante Änderungen durch andere Transaktionen wirksam geworden sein können.

2. Falls der Systemkatalog **nicht** vollständig in den Hauptspeicher geladen werden kann, müssen Änderungen im Verlauf einer Transaktion zunächst in einem lokalen Teilkatalog durchgeführt werden. Bei späteren Zugriffen auf den Systemkatalog muss dieser Teilkatalog berücksichtigt werden, **bevor** Lesezugriffe auf den globalen Systemkatalog erfolgen.

In beiden Varianten müssen zumindest **die** Anweisungen registriert werden, die durch die zugrundeliegende Implementierung des Speichersystems durch eine *Rollback*-Anweisung nicht automatisch zurückgenommen werden können. Im Falle der BERKELEY DB gehören hierzu das Anlegen und Löschen von physischen Dateien, da diese Aktionen nicht dem Transaktionsmechanismus unterworfen sind. Im Falle von ORACLE ist das Anlegen und Löschen von Tabellen betroffen, da DDL-Anweisungen automatisch sofort wirksam werden. Insbesondere das Löschen von Dateien bzw. Tabellen ist mit besonderer Sorgfalt zu behandeln: das tatsächliche

Löschen darf erst im Zuge der abschließenden *Commit*-Anweisung erfolgen, da sonst im Falle eines *Rollbacks* eine Wiederherstellung faktisch unmöglich wäre.

Aufgrund dessen, dass in beiden Varianten Lesezugriffe auf den globalen Systemkatalog im Verlauf einer Transaktion unvermeidlich sind, ergeben sich aus einer vollständigen lokalen Kopie des Systemkatalogs keine offensichtlichen Vorteile. Daher wird für die Implementierung Variante 2 gewählt.

Intern arbeiten der Systemkatalog und der Query-Prozessor mit geschachtelten Listen auf der Basis kompakter Tabellen. Da alle geschachtelten Listen in einem einzigen *Nested-List*-Container abgelegt werden, ist die Umwandlung einer geschachtelten Liste in eine kompakte, für die persistente Speicherung geeignete Repräsentation leider relativ aufwendig, da zumindest zwei Durchläufe durch die jeweilige Liste erforderlich wären. Daher wird für die persistente Speicherung nicht auf persistente kompakte Tabellen, sondern auf die textuelle Darstellung geschachtelter Listen zurückgegriffen. Ein separater persistenter Namensindex ist nicht erforderlich, da sowohl BERKELEY DB als auch ORACLE effiziente schlüsselbasierte Zugriffsstrukturen bieten.

3.3.2 Verwaltung von Datenbanken

Hinsichtlich der Verwaltung von SECONDO-Datenbanken wird davon ausgegangen, dass die grundlegende Einrichtung (Erstellen von Konfigurationsdateien, Anlegen von Verzeichnissen bei der BERKELEY DB, Anlegen einer Datenbank-Instanz bei ORACLE) außerhalb von SECONDO vorgenommen wird. Innerhalb dieses Rahmens gestattet allerdings das System das Anlegen und Löschen mehrerer SECONDO-Datenbanken.

Ein SECONDO-Server-System kann nur SECONDO-Datenbanken verwalten, die auf dem gleichen Speichersystem basieren. Durch geeignete Wahl der Konfigurationsparameter sollte es jedoch möglich sein, bei Bedarf auch mehrere Server für ggf. auch unterschiedliche Speichersysteme auf einem Rechner auszuführen. Es wird jedoch empfohlen, auf einem Rechner nur ein einziges SECONDO-Server-System zu betreiben.

3.3.3 Persistente Speicherung von Objekten

Wie in Abschnitt 2.3 auf Seite 24 bereits erwähnt, werden Objekte in der bisherigen SECONDO-Version innerhalb des Systems mit Hilfe eines einzelnen Speicherwortes dargestellt. Für die Handhabung von Objektwerten im Hauptspeicher durch Systemkomponenten wie den Query-Prozessor reicht diese Vorgehensweise aus und wird

auch in der neuen SECONDO-Version beibehalten. Die persistente Speicherung von Objektwerten unterlag damit bisher impliziten Einschränkungen, deren Einhaltung jedoch durch das System nicht überwacht wurden.

Es wäre natürlich möglich, die persistente Speicherung von Objekten, für deren vollständige Darstellung ein einzelnes Speicherwort nicht ausreicht, zu verbieten und dieses Verbot durch geeignete Maßnahmen zu überwachen. Allerdings würden dadurch die Probleme nur teilweise gelöst, wie das Beispiel der Relationen zeigt.

Im Fall von Relationen beinhaltet das Speicherwort den Verweis auf einen Eintrag in einem von der Relationenalgebra selbst verwalteten Relationenkatalog, in dem zu jeder Relation innerhalb einer SECONDO-Datenbank Metadaten wie beispielsweise die Identifikatoren der zugehörigen SHORE-Files für die Speicherung der Tupel und der in ihnen enthaltenen *Large Objects* gespeichert sind. Ein gravierender Nachteil dieser Vorgehensweise ist, dass die Relationenalgebra originäre Aufgaben des Systemkatalogs übernehmen muss, nämlich die Verwaltung einer Liste aller Relationenobjekte in einer Datenbank. Darüber hinaus muss sie überwachen, ob zwischenzeitlich ein Wechsel der SECONDO-Datenbank stattgefunden hat, da in diesem Fall auch ein Wechsel des Relationenkatalogs stattfinden muss.

Hieraus wird deutlich, dass der Systemkatalog in geeigneter Weise die Speicherung von Daten und Metadaten zu einem Objekt unterstützen sollte, um dadurch eine Algebra von objektübergreifenden Katalogisierungsaufgaben zu befreien. Der Systemkatalog verfügt jedoch nicht über Wissen zur inneren Struktur eines Objekts. Er braucht dies normalerweise auch nicht, müsste aber zur Objektstruktur entweder implizite Annahmen machen oder explizite Anforderungen formulieren (und natürlich irgendwie überwachen), um die Speicherung überhaupt durchführen zu können. Diese Vorgehensweise wäre jedoch unnötig komplex und fehlerträchtig. Stattdessen ist folgende Aufgabenteilung sinnvoller: der Systemkatalog stellt nur einen Mechanismus zur Speicherung und Wiederherstellung von Objektdaten zur Verfügung, überlässt aber die konkrete Speicherung dem Objekt selbst.

Der gesuchte Mechanismus muss so geartet sein, dass einerseits eine flexible persistente Speicherung von Objekten ermöglicht wird, andererseits aber kein hoher Anpassungsaufwand für sämtliche Algebra-Module erzeugt wird. Dieses Ziel kann erreicht werden, indem ein Standardspeicherungsmechanismus bereitgestellt wird, der nur bei Bedarf durch einen objekttypspezifischen Mechanismus ersetzt wird. Da ein Algebra-Modul stets eine Objektausgabefunktion für eine *Nested-List*-Darstellung eines Objekts bereitzustellen hat, bietet sich an, den Standardmechanismus darauf basieren zu lassen.

Nach diesen Vorüberlegungen kann der Mechanismus nunmehr konkretisiert werden. Der Systemkatalog beinhaltet in Zukunft ein zusätzliches *SmiFile*, aus der je-

dem Objekt ein *SmiRecord* zur persistenten Speicherung seiner Daten oder Metadaten zur Verfügung gestellt wird. Zusätzlich zu den übrigen Objektinformationen muss gegenüber der bisherigen Vorgehensweise nur die Satznummer des *SmiRecords* gespeichert werden. Enthält das zugehörige Algebra-Modul keine eigene Methode, wird standardmäßig die *Nested-List*-Darstellung des Objekts in Form einer Zeichenkette im *SmiRecord* abgelegt.

Für Objekte der Standardalgebra kann durch den Standardmechanismus auf einfache Weise Persistenz geboten werden; für die Relationenalgebra wird es dagegen erforderlich sein, geeignete eigene Methoden bereitzustellen. Für andere Algebra-Module wird man diese Frage von Fall zu Fall entscheiden müssen.

3.4 Verwaltung von Typen und Operatoren

Wie in der Analyse ab Seite 25 angemerkt wurde, werden Algebra-Module intern anhand einer Nummer identifiziert. Während bislang die zugeordnete Nummer von der Reihenfolge, in der die Module geladen wurden, abhängig war, soll zukünftig jeder Algebra eindeutig eine Nummer zugeordnet werden. Die Liste aller verfügbaren Algebren wird in einer Konfigurationsdatei der Algebraverwaltung gepflegt. Zu jeder Algebra wird dort ihre eindeutige Nummer, ihr Name sowie ihr Typ (deskriptiv, ausführbar, hybrid) aufgeführt. Außerdem kann ein Entwickler darüber entscheiden, welche dieser Algebren tatsächlich in das System eingebunden und beim Start des Systems geladen und initialisiert werden sollen.

Die Entscheidung darüber, ob ein Algebra-Modul statisch in das System eingebunden oder dynamisch beim Start des Systems geladen wird, bleibt dem Entwickler des jeweiligen Algebra-Moduls überlassen. Zur Zeit ist vorgesehen, dass zum Zeitpunkt des Bindens sämtliche Algebra-Module vorliegen. Das Laden unbekannter Algebra-Module erst zur Laufzeit ist prinzipiell zwar auch möglich, wäre jedoch nur dann sinnvoll durchführbar, wenn der SECONDO-Parser auch die Algebra-Spezifikation dynamisch auswerten könnte.

Die Schnittstelle zum Query-Prozessor und zum Systemkatalog wird über den Algebra-Manager realisiert. Jede Algebra verwaltet die in ihr enthaltenen Typkonstruktoren und Operatoren selbständig, stellt dem Algebra-Manager jedoch Informationen über die jeweilige Anzahl an Typen und Operatoren bereit, so dass der Algebra-Manager anhand der Algebra-Nummer sowie eines Index auf die Menge der Typen bzw. Operatoren innerhalb dieser Algebra die gewünschte Information (Typ- und Operatornamen, Typeigenschaften, Operatorspezifikationen, Adressen von Funktionen usw.) bereitstellen kann.

Kapitel 4

Implementierung

Für die konkrete Umsetzung der Entwurfskonzepte war zunächst zu klären, welche Entwicklungsumgebungen auf den zu unterstützenden Betriebssystemplattformen zur Verfügung stehen und ggf. eine Auswahl zu treffen. Die verschiedenen Probleme, die bei der Vorbereitung und Durchführung der Implementierung auftraten, und wie sie gelöst wurden, werden daran anschließend beschrieben.

4.1 Entwicklungsumgebung

In Tabelle 4.1 sind die für Redesign und Reimplementierung eingesetzten Entwicklungsumgebungen aufgelistet.

Betriebssystem	**Compiler**
SuSE Linux 7.2	GNU gcc 2.95.3
Windows 98, Windows NT 4.0 SP 6a	MinGW GNU gcc 2.95.3, UnxUtils
Solaris 2.7 (Sparc / FernUni)	GNU gcc 2.95.3

Tabelle 4.1: Eingesetzte Betriebssysteme und Compiler

Außerdem wurden insbesondere für die Realisierung des Speichersystems die in Tabelle 4.2 aufgeführten Software-Komponenten eingesetzt.

Komponente	**Version**	**URL**
BERKELEY DB	4.0.14	`http://www.sleepycat.com`
Oracle Personal Ed.	8.1.6	`http://www.oracle.com`
OCI C++ Library	0.5.6	`http://ocicpplib.sourceforge.net`

Tabelle 4.2: Verwendete Softwarekomponenten

Unter LINUX und SOLARIS fällt die Wahl der Entwicklungswerkzeuge recht leicht, da die GNU Compiler Collection kostenlos zur Verfügung steht. Eingesetzt wird die Version GNU gcc 2.95.3. Auf den Einsatz der im Sommer 2001 freigegebenen Version GNU gcc 3.0 wird verzichtet, da keine Kompatibilität mit Bibliotheken der Vorversionen gegeben ist.

Ganz anders stellt sich die Situation unter WINDOWS dar. Sowohl für die BERKELEY DB als auch für ORACLE wird unter WINDOWS von den Herstellern nur der MS Visual C++ Compiler 5.0 (oder höher) unterstützt. Visual C++ ist jedoch ein kommerzielles Entwicklungswerkzeug und daher in der Anschaffung entsprechend kostspielig. Mögliche Alternativen sind Borland C++ und die derzeit verfügbaren Portierungen der GNU Compiler Collection, Cygwin und MinGW.

Borland bietet seinen Compiler, der in dem kommerziellen Produkt *Borland C++ Builder* integriert ist, als Kommandozeilenversion kostenfrei zum Download[1] an. Die Quellcode-Anpassungen der BERKELEY DB für den Borland Compiler waren zwar mit relativ geringem Aufwand zu bewerkstelligen, jedoch weist die Borland-Laufzeitumgebung Inkompatibilitäten zur Microsoft-Laufzeitumgebung auf, die bereits bei einfachen Beispielprogrammen zu Laufzeitfehlern führen. Mehrfache Rücksprachen mit einem Support-Mitarbeiter der Firma SleepyCat erbrachten bedauerlicherweise keine Lösung für die aufgetretenen Probleme. Ein Einsatz der Borland-Version der BERKELEY DB kommt daher derzeit nicht in Frage. Hinsichtlich des auf ORACLE basierenden Speichersystems sieht es nicht viel besser aus: bis zu den Versionen 8.0.x wurden von ORACLE sowohl Microsoft Visual C++ als auch Borland C++ unterstützt. Beginnend mit den Versionen 8.1.x hat ORACLE die Unterstützung des Borland-Compilers teilweise eingestellt, wenngleich für die OCI-Schnittstelle[2] noch Import-Bibliotheken für Borland zur Verfügung stehen. Die OCI-C++-Bibliothek lässt sich allerdings mit dem Borland-Compiler nicht fehlerfrei übersetzen.

Cygwin (GNU+Cygnus+Windows) ist eine UNIX-Umgebung für WINDOWS, die kostenfrei aus dem Internet[3] heruntergeladen werden kann. Cygwin besteht aus zwei Hauptkomponenten: einer DLL, die als UNIX-Emulationsschicht dient, und einer Sammlung diverser Werkzeuge, mit denen unter WINDOWS eine nahezu UNIX-konforme Umgebung zur Verfügung steht. Einerseits wird die Portierung von UNIX-Programmen sehr erleichtert, andererseits wirkt sich die Emulationsschicht nachteilig auf die Performanz aus. Auch die direkte Nutzung von WINDOWS-Funktionalität ist – wenn überhaupt – nur mit deutlich höherem Aufwand zu erreichen.

---

[1] `http://www.borland.com/bcppbuilder/freecompiler/`

[2] OCI = Oracle Call Interface

[3] `http://cygwin.com/`

MinGW (Minimalist GNU For Windows) ist eine Sammlung von Header-Dateien und Import-Bibliotheken, die es erlauben, den GNU gcc Compiler zur Erzeugung nativer WINDOWS-32-Bit-Programmen zu nutzen, ohne von zusätzlichen DLLs abhängig zu sein. Zur Zeit stehen die GNU Compiler Collection (GCC), die GNU Binary Utilities (Binutils), der GNU Debugger (Gdb) , das GNU Make sowie einige weitere ausgewählte Werkzeuge zum kostenfreien Download[4] zur Verfügung. Als Ergänzung zu *MinGW* bietet sich *UnxUtils*[5], eine Sammlung von UNIX-Werkzeugen für WIN32-Plattformen, an, da in der *MinGW*-Distribution einige nützliche Hilfsprogramme wie etwa *rm* fehlen.

Die Gewährleistung der Portabilität von SECONDO dürfte sicherlich leichter fallen, wenn auf allen unterstützten Betriebssystemplattformen der gleiche Compiler – nämlich GNU gcc – eingesetzt wird.

Der *Borland*-Compiler und die *Cygwin*-Portierung kommen als Entwicklungswerkzeug wegen der oben genannten Nachteile nicht in Frage, so dass nur *MinGW* als frei verfügbare Alternative zu *Microsoft Visual C++* übrigbleibt. Um die BERKELEY DB einwandfrei mit *MinGW* übersetzen zu können, waren nur marginale Quellcode-Anpassungen erforderlich. Die Beispielprogramme waren unter mehreren WINDOWS-Varianten (98, ME, NT 4.0) lauffähig. Auch die OCI-C++-Bibliothek ließ sich mit *MinGW* problemlos übersetzen. Die größte Schwierigkeit bestand zunächst in der Anbindung an die OCI-DLLs von ORACLE, da *MinGW* von ORACLE offiziell nicht unterstützt wird. Um trotzdem zu einer funktionierenden Import-Bibliothek für *MinGW* zu gelangen, war es erforderlich, aus der OCI-DLL von ORACLE zunächst eine Definitionsdatei mit den verfügbaren Einsprungpunkten zu gewinnen. Hierfür wurde das frei verfügbare Programm `impdef`[6] verwendet. Anschließend konnte mit Hilfe des Programms `dlltool` der *MinGW*-Entwicklungsumgebung aus der Definitionsdatei eine Import-Bibliothek erzeugt werden.

4.2 Behandlung globaler Variablen und Funktionen

In der bisherigen Implementierung werden in zahlreichen SECONDO-Modulen globale Variablen verwendet. Um Namenskonflikte auszuschließen und den Code reentrant-fähig zu machen (falls in zukünftigen Implementierungen Multi-Threading zum Einsatz kommen sollte), sind globale Variablen soweit wie möglich zu vermeiden.

---

[4] `http://www.mingw.org/`

[5] `http://unxutils.sourceforge.net/`

[6] `http://www.cygwin.com/ml/cygwin/1997-04/msg00221.html`

In den meisten Fällen kann diese Forderung durch Kapselung globaler Variablen in Klassen erfüllt werden. Problematischer stellt sich die Situation bei den automatisch generierten Code-Teilen im Zusammenhang mit der Klasse *NestedList* dar. *Lex* und *Yacc* produzieren nur C-Code, in dem globale Variablen für die Kommunikation der verschiedenen Unterprogramme benutzt werden. Für *Lex* gibt es bei den GNU Werkzeugen eine Version namens *Flex++*, die in der Lage ist, C++-Klassen zu erzeugen. Das GNU-Äquivalent zu *Yacc* ist *Bison*. Bislang verfügt *Bison* nicht über Optionen zur Erzeugung von C++-Klassen. Bei entsprechender Suche im Internet stößt man zwar auf verschiedene *Bison++*-Varianten, die jedoch zumeist auf veralteten *Bison*-Versionen basieren und auf deren Einsatz daher verzichtet wird. Stattdessen werden zwei verschiedene Methoden, Parser in Form von C++-Klassen zu generieren, verwendet:

- Unmittelbare Generierung einer C++-Klasse für den Parser, indem der mit Hilfe von *Bison* erzeugte C-Code nachträgliche mit dem Hilfsprogramm `sed` verändert wird. Diese Manipulationen sind erforderlich, um einige Symbole (Methodennamen, Member- und Klassenvariablen) der Parserklasse zuzuordnen. Diese Technik wird für das Modul *NestedList* eingesetzt.
- Programmierung einer C++-Wrapperklasse zur Kapselung der von *Bison* generierten globalen Parser-Methoden. Eine nachträgliche Manipulation des Quellcodes entfällt. Diese Technik wird für den SECONDO-Parser benutzt.

4.3 Client/Server-Architektur

4.3.1 Netzwerkkommunikation

TCP/IP und Internet Sockets

Die Kommunikation zwischen SECONDO-Client und SECONDO-Server basiert auf *TCP*, einem zuverlässigen, verbindungsorientierten Protokoll aus der *TCP/IP* [7]*-Protokollfamilie*, da Zuverlässigkeit einen entscheidenden Faktor für ein Datenbanksystem darstellt.

Für den Datenaustausch zwischen Client und Server werden *Internet Stream Sockets* eingesetzt, die sich aus Sicht der Anwendung wie sequentielle Dateien verhalten, da TCP die transferierten Daten als einen kontinuierlichen Strom von Bytes behandelt.

[7] Der Name *TCP/IP* leitet sich aus den beiden wichtigsten Mitgliedern dieser Protokollfamilie ab: ***T**ransmission **C**ontrol **P**rotocol* und ***I**nternet **P**rotocol*.

Diesem Umstand wird in der implementierten C++-Socket-Klasse Rechnung getragen, indem sie eine zu C++-Ein/Ausgabeströmen kompatible Schnittstelle bereitstellt. Dadurch können sämtliche Standarddatentypen wie Ganzzahlen, Gleitkommazahlen oder Zeichenketten sowie alle Klassen, für die die Strom-Operatoren $<<$ und $>>$ [8] implementiert sind, problemlos über eine Socket-Verbindung übertragen werden.

Unter UNIX-ähnlichen Betriebssystemen werden Sockets genauso behandelt wie normale Datei-Deskriptoren, so dass sie prinzipiell direkt einem Ein/Ausgabestrom der C++-Standardbibliothek zugeordnet werden können. Von dieser Möglichkeit konnte jedoch kein Gebrauch gemacht werden, da unter WINDOWS-Betriebssystemen Socket-Deskriptoren und Datei-Deskriptoren inkompatibel sind. Stattdessen wurde eine von der `streambuf`-Klasse abgeleitete Klasse programmiert, die die Verwaltung der Ein- und Ausgabepuffer eines Stroms übernimmt und unterlagert die Ein- und Ausgabe auf dem zugeordneten Socket durchführt. Im Konstruktor eines Strom-Objekts wird dann eine abgeleitete `streambuf`-Instanz mitgegeben, um den Strom mit dem gewünschten Socket zu verknüpfen. Diese Vorgehensweise hat gegenüber der Zuordnung von Deskriptoren noch den Vorteil, dass der Status des Sockets relativ leicht auf den Status des Stroms abgebildet werden kann.

4.3.2 Prozesssteuerung und -kommunikation

In einem Client/Server-System, das serverseitig in der Regel aus mehreren Prozessen besteht, kommt der Prozesssteuerung und -kommunikation eine besondere Bedeutung zu: ein Prozess muss in der Lage sein, Kind-Prozesse zu starten, mit diesen zu kommunizieren bzw. ihre Ausführung zu beeinflussen, sie kontrolliert zu beenden und ihren Beendigungsstatus zu überprüfen.

Prozesserzeugung

Unter UNIX-ähnlichen Betriebssystemen wird die Erzeugung von Prozessen durch die Funktion `fork` der Laufzeitbibliothek des Systems übernommen. `fork` erstellt eine identische Kopie des Eltern-Prozesses und setzt sowohl Kind- als auch Eltern-Prozess hinter dem `fork`-Aufruf fort, wobei der Rückgabewert von `fork` darüber Auskunft gibt, welches der Kind- und welches der Eltern-Prozess ist. Darüberhinaus werden auch offene Datei-Deskriptoren dupliziert, so dass der Kind-Prozess bei

[8] Der Eingabeoperator $>>$ wird möglicherweise in SECONDO nicht genutzt werden, da dieser Operator das Einlesen von Daten nach jedem Leerzeichen (oder sonstigen sogenannten *whitespace*-Zeichen) beendet, so dass eine Zeile, die solche Zeichen enthält, nicht in einem Stück eingelesen werden kann. Stattdessen wird teilweise eine Methode zum Lesen ganzer Zeilen verwendet.

Bedarf mit geöffneten Dateien weiterarbeiten kann. Üblicherweise werden nicht benötigte Datei-Deskriptoren sowohl vom Eltern- als auch vom Kind-Prozess geschlossen, um gegenseitige störende Einflüsse zu verhindern. [Ste93]

Unter WINDOWS stellt die Systembibliothek keine `fork`-Funktion zur Verfügung. Stattdessen gibt es die Funktion `CreateProcess`, mit Hilfe derer ein ausführbares Programm geladen und als eigenständiger Prozess gestartet werden kann. Dies entspricht in UNIX-Systemen im wesentlichen der Situation, dass der Kind-Prozess direkt nach dem `fork` eine der `exec`-Funktionen verwendet, um die Kopie des Eltern-Prozesses durch ein anderes Programm zu überlagern. Geöffnete Datei-Deskriptoren werden nicht automatisch an den Kind-Prozess weitergegeben, sondern nur dann, wenn folgende Bedingungen erfüllt sind: zum einen müssen die jeweiligen Deskriptoren überhaupt *vererbbar*[9] sein und zum anderen muss bei der Erzeugung des neuen Prozesses angegeben werden, ob tatsächlich offene Deskriptoren vererbt werden sollen. Schließlich ist anzumerken, dass unter WINDOWS Prozesse generell unabhängig voneinander sind, d.h., keine Eltern-Kind-Beziehung haben. Im Gegensatz zu Kind-Prozessen unter UNIX können daher Prozesse unter WINDOWS nicht feststellen, durch welchen Prozess sie erzeugt wurden.

Um eine möglichst hohe Portabilität der Prozesssteuerung zu erreichen, wurde die Erzeugung eines Prozesses in eine Funktion gekapselt, die unter UNIX den Kind-Prozess durch ein neues Programm überlagert (`exec`-Funktion) und unter WINDOWS einerseits die Vererbung offener Deskriptoren anfordert und andererseits die Prozessidentifikation des erzeugenden Prozesses als versteckten Kommandozeilenparameter an den erzeugten Prozess übergibt.

Prozesssteuerung

Während unter UNIX-ähnlichen Betriebssystemen ein Signalmechanismus zur Verfügung steht, mit Hilfe dessen einem Prozess bestimmte Ereignisse mitgeteilt werden können, ist ein solcher Mechanismus unter WINDOWS nur rudimentär vorhanden.

Für das SECONDO-System ist es insbesondere bedeutsam, dass ein Eltern-Prozess über die Beendigung eines Kind-Prozesses informiert wird und dass ein Eltern-Prozess einen Kind-Prozess zur Beendigung auffordern kann. Unter UNIX kann diese Anforderung durch die Implementierung von Signal-Handlern für die Signale `SIGCHLD` und `SIGTERM` relativ leicht erfüllt werden.

Unter WINDOWS können Signale nur innerhalb eines Prozesses erzeugt und behan-

[9] Bei der Erzeugung eines Deskriptors (handle) kann bzw. muss unter WINDOWS spezifiziert werden, ob er an einen anderen Prozess *vererbbar* sein soll oder nicht.

delt werden, können also zur Erfüllung der genannten Anforderung überhaupt nicht genutzt werden. Es ist noch nicht einmal auf einfache Weise möglich, einen anderen Prozess zu einer geordneten Beendigung aufzufordern.[10] Daher wird der UNIX-Signalmechanismus mit Hilfe von Threads und lokalen Sockets nachgebildet.[11] Der Eltern-Prozess startet zusätzlich zum Kind-Prozess einen Thread, der auf die Beendigung (das Signal `SIGCHLD`) des Kind-Prozesses wartet; der Kind-Prozess startet einen Thread, der die Aufgabe hat, auf einem lokalen Socket Signale (u.a das Signal `SIGTERM`) zu empfangen.

Die implementierten Applikations- und Prozessklassen unterstützen für die applikationsspezifische Prozesskommunikation zusätzlich die Signale `SIGUSR1` und `SIGUSR2`.

Prozesse und Socket-Kommunikation

Ein Eltern-Prozess, der auf einem globalen Socket Verbindungsanforderungen annimmt, muss in der Lage sein, den bei der Annahme der Verbindung erzeugten Client-Socket an einen Kind-Prozess zu übergeben. Wenn eine solche Kommunikationsverbindung beendet werden soll, müssen sowohl der Eltern- als auch Kind-Prozess den Client-Socket unbedingt schließen.

Die Übergabe eines Sockets an den Kind-Prozess ist unter UNIX unproblematisch, da offene Deskriptoren dem Kind-Prozess per se zur Verfügung stehen. In der hier verwendeten Art der Prozesserzeugung muss also nur die entsprechende Deskriptornummer an den Kind-Prozess übermittelt werden. Der Eltern-Prozess kann den Socket sofort nach der Erzeugung des Kind-Prozesses schließen.

Unter WINDOWS stellt sich die Situation ungleich komplexer dar. Zum einen ist ein Socket-Handle nicht unter allen WINDOWS-Varianten automatisch vererbbar und muss daher vor der Übergabe an einen Kind-Prozess als vererbbar dupliziert werden. Zum anderen darf der Eltern-Prozess die Socket-Handles erst dann schließen, wenn der Kind-Prozess beendet worden ist.

Diese Problematik wird durch Kooperation der Klassen *Socket*, *ProcessFactory* und *Application* gelöst. Der *ProcessFactory*-Methode zur Erzeugung des Kind-Prozesses kann optional eine Referenz auf eine *Socket*-Instanz übergeben werden. Der in der *Socket*-Instanz enthaltene Socket-Handle wird vererbbar dupliziert und in Form eines zusätzlichen Kommandozeilenparameters an den erzeugten Prozess überge-

[10] Die WINDOWS-Systemfunktion `TerminateProcess` bricht einen Prozess unmittelbar ab und räumt noch nicht einmal die Systemressourcen auf – von den prozesseigenen ganz zu schweigen.

[11] Ein Nachteil dieser Vorgehensweise ist, dass nur Prozesse, die die hier bereitgestellten Komponenten nutzen, miteinander darüber kommunizieren können – die Steuerung von beliebigen Prozessen ist nicht möglich.

ben. Auf Basis dieses Handles wird im Konstruktor der *Application*-Instanz des Kind-Prozesses eine neue *Socket*-Instanz erzeugt, mit der in der Folge ohne zusätzliche Maßnahmen gearbeitet werden kann.

Die Klasse *ProcessFactory* verwaltet intern eine Liste aller erzeugten Kind-Prozesse, um deren Status nachzuhalten und somit in der Lage zu sein, eventuell übergebene Socket-Handles erst nach Beendigung des zugehörigen Kind-Prozesses zu schließen.

4.4 Speichersystem

4.4.1 Klassenstruktur

Aus den Ausführungen im Abschnitt 3.2 auf Seite 31 lässt sich für die Implementierung die in Abbildung 4.1 dargestellte Klassenstruktur ableiten.

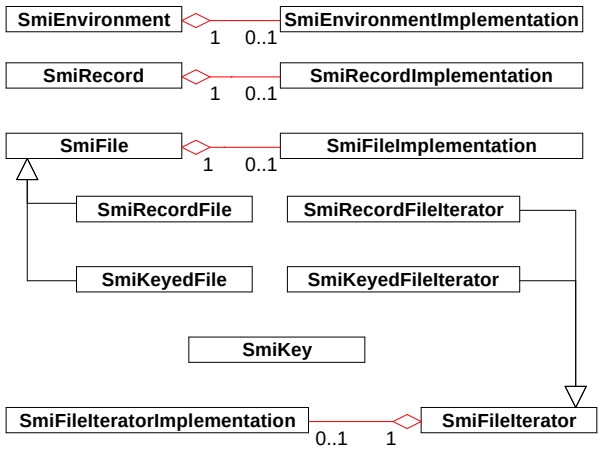

Abbildung 4.1: Klassendiagramm für das Speichersystem

Da in C++ leider keine saubere Trennung von Schnittstelle und Implementierung einer Klasse gewährleistet ist, führen Änderungen an den privaten Teilen einer Klasse in der Regel dazu, dass alle diese Klasse verwendenden Module ebenfalls neu übersetzt werden müssen. Da die Speichersystemschnittstelle in zwei Varianten zu implementieren ist, nämlich einmal basierend auf der BERKELEY DB und einmal auf ORACLE, ist dies besonders nachteilig. Daher werden die privaten Anteile der Schnittstellen-Klassen in Implementierungsklassen ausgelagert, da Referenzen auf Klassen keine vollständige Klassendefinition benötigen. Die Konstruktoren der betroffenen Schnittstellenklassen instantiieren jeweils die zugehörige Implementierungsklasse. Auf diese Weise kann die Entscheidung, welche Basisimplementierung des Speichersystems verwendet werden soll, auf die Link-Phase verschoben werden.

4.4.2 BERKELEY DB

Die Implementierung auf Basis der BERKELEY DB unterstützt sowohl einen Mehrbenutzer- als auch einen Einzelbenutzerbetrieb. Als *Benutzer* gilt in diesem Zusammenhang nicht nur eine reale Person, sondern auch ein Programm. Selbst auf einem Einzelplatzrechner können mehrere Programme gleichzeitig aktiv sein. Daher ist im Einzelbenutzerbetrieb darauf zu achten, dass die angesprochene BERKELEY DB auf keinen Fall parallel von einem anderen Prozess – sei es eine Person oder sei es ein Programm – bearbeitet werden darf. Die Nichtbeachtung dieser Randbedingung kann zu einer Zerstörung der Datenbank-Dateien führen.

Im Mehrbenutzerbetrieb sind bei der BERKELEY DB einige begleitende Maßnahmen erforderlich, um einen reibungslosen Betrieb zu gewährleisten.

Zum einen muss beim Start des Systems eine Überprüfung der BERKELEY DB-Umgebung erfolgen, um festzustellen, ob ggf. ein Wiederanlauf nach einem Systemabsturz durchzuführen ist. Falls ja, darf währenddessen kein weiterer Prozess auf die gleiche BERKELEY DB-Umgebung zugreifen.

Zum anderen ist eine Blockade-Erkennung zu aktivieren, um eventuell auftretende gegenseitige Blockaden durch Abbruch von ein oder mehreren Transaktionen aufzulösen. Hierfür stehen verschiedene Möglichkeiten zur Verfügung:

1. Starten des Hilfsprogramms **db_deadlock**, welches von der BERKELEY DB bereitgestellt wird,
2. Starten eines eigenen Hilfsprogramms mit der gleichen Funktionalität wie **db_deadlock**,
3. Aktivieren der automatischen Blockade-Erkennung.

Variante 1 hat den Vorteil, ohne Programmieraufwand zur Verfügung zu stehen, aber den Nachteil, dass der Prozess, in dem dieses Hilfsprogramm gestartet wird, nicht die in den SECONDO-Modulen verwendeten Prozesskommunikationsmechanismen unterstützt und daher nicht in portabler Weise kontrolliert werden kann. Variante 2 ist mit überschaubarem Programmieraufwand zu realisieren, bedeutet aber wie Variante 1 einen zusätzlich zu startenden Prozess. Variante 3 lässt sich als Option der BERKELEY DB-Umgebung einstellen und ist daher ohne zusätzlichen Aufwand nutzbar. Da Variante 3 die benötigte Funktionalität bereitstellt und den geringsten Aufwand nach sich zieht, wird sie zunächst für die Blockade-Erkennung genutzt. Falls sich im Betrieb unter Praxisbedingungen dadurch Nachteile zeigen sollten, wäre diese Entscheidung ggf. zu überdenken und ein Ersatz von Variante 3 durch Variante 2 zu erwägen.

Schließlich ist es wichtig, regelmäßig Sicherungspunkte (checkpoints) zu erzeugen, so dass ein Wiederanlauf nach einem Systemabsturz nur begrenzte Teile der Log-Dateien zur Wiederherstellung benötigt. Für die Archivierung der BERKELEY DB-Dateien kann auf die von der BERKELEY DB bereitgestellten Hilfsprogramme zurückgegriffen werden.

In den ersten Experimenten mit der Implementierung des Speichersystems auf Basis der BERKELEY DB hat sich leider herausgestellt, dass die Erwartungen an den Transaktionsmechansimus zu hoch gestellt waren. Im ersten Entwurf war vorgesehen, Benutzertransaktionen automatisch zu starten und alle Kommandos bis zur Bestätigung durch ein *Commit*-Kommando bzw. bis zum Abbruch durch ein *Rollback*-Kommando in einer Transaktion zu klammern. Da Benutzertransaktionen vergleichsweise lange dauern können und unter Transaktionskontrolle stattfindende Leseoperationen jegliche Schreiboperationen auf den gleichen Datensätzen verhindern, steigt die Wahrscheinlichkeit für Systemblockaden sehr stark an. Ein sequentielles Lesen einer kompletten BERKELEY DB-Datei blockiert diese z.B. für jegliche Schreiboperationen. In der Dokumentation der BERKELEY DB wird daher empfohlen, Leseoperationen **nicht** unter Transaktionskontrolle durchzuführen. Wenn eine Anwendung eine Transaktion startet und innerhalb dieser sowohl lesend als auch schreibend auf einer BERKELEY DB-Datei arbeitet, und dabei der Empfehlung folgend nur die Schreiboperationen unter Transaktionskontrolle durchführt, kann es leicht passieren, dass sich die Anwendung selbst blockiert. Dies wird durch den Blockade-Erkennungsalgorithmus jedoch **nicht** entdeckt. Eine Totalblockade des SECONDO-Systems wäre letztlich die unausweichliche Folge.

Rücksprachen mit einem der BERKELEY DB-Entwickler, Keith Bostic, von der Firma Sleepycat ergaben eine völlige Bestätigung der bei den Experimenten beobachteten Phänomene. Letztlich bedeutet dies, dass Transaktionen bei der BERKELEY DB möglichst nur genutzt werden sollten, um mehrere BERKELEY DB-Dateien betreffende Operationen konsistent durchzuführen, nicht aber, um umfangreiche, komplexe Änderungsoperationen durchzuführen – es sei denn, die Applikation ist in der Lage, fehlgeschlagene Transaktionen ohne hohen Aufwand zu wiederholen. Im interaktiven Gebrauch dürfte das jedoch nur selten gegeben sein.

Als Ausweg aus dem Dilemma wurde festgelegt, dass einzelne Kommandos ohne Transaktionskontrolle bzw. in einer Art *Autocommit*-Modus bearbeitet werden, d.h. jedes Kommando wird für sich allein als Transaktion aufgefasst. In vielen Anwendungsfällen wird diese Arbeitsweise ausreichend sein. Wenn jedoch (u.U. sehr komplexe) Anweisungsfolgen innerhalb eines Programms generiert werden, für das die Wiederholung einer Transaktion in der Regel relativ unproblematisch sein dürfte, wäre eine leistungsfähige Transaktionsbearbeitung nützlich. Daher be-

steht die Möglichkeit, mit dem Kommando `BEGIN TRANSACTION` gezielt eine Transaktion einzuleiten und mit den Kommandos `COMMIT TRANSACTION` bzw. `ROLLBACK TRANSACTION` zu beenden oder abzubrechen. Wenn von dieser Möglichkeit Gebrauch gemacht wird, ist generell zu empfehlen, wenn irgend möglich, zunächst alle Leseoperationen und erst danach alle Schreiboperationen durchzuführen, um die Wahrscheinlichkeit von Blockaden zu reduzieren.

Da die BERKELEY DB in der Regel seitenorientierte Lese- und Schreibsperren verwendet und nur bei der satzorientierten Zugriffsmethode mit fester Satzlänge in der Lage ist, satzorientierte Sperren zu verwalten, kommt es leider auch in solchen Situationen leicht zu Blockaden (die durch Abbruch einer der beteiligten Transaktionen aufgelöst werden), in denen eigentlich auch ein konkurrierender Schreibzugriff möglich sein müsste.

Die beschriebenen Einschränkungen und Probleme bezüglich des Transaktionsmechanismus der BERKELEY DB haben noch einmal eine Recherche nach möglichen Alternativen ausgelöst. Im Rahmen dieser Recherche rückte INNODB ins Blickfeld.

INNODB ist eine Entwicklung des Finnen Heikki Tuuri und wird derzeit als ein Tabellenhandler des weitverbreiteten frei verfügbaren Datenbanksystems MYSQL bereitgestellt. Die Eigenschaften von INNODB lassen dieses System auf den ersten Blick als nahezu ideale Wahl erscheinen:

1. konsistentes Lesen wie bei einer ORACLE-Datenbank durch Mehrfachversionierung auf der Basis von Rollback-Segmenten, dadurch verzögerungsfreie Updates
2. satz- und tabellenorientierte Sperren
3. konsistente Cursor (keine Phantomdatensätze)
4. automatische Blockade-Erkennung
5. Transaktionsunterstützung

und vieles mehr.

Die erste Ernüchterung stellte sich jedoch bereits beim flüchtigen Überfliegen des zugehörigen Quellcodes ein. Zum einen fehlt in der MYSQL-Distribution jegliche Dokumentation zu INNODB und zum anderen besteht eine hohe Verflechtung mit anderen MYSQL-Codeteilen. Eine Nachfrage per Mail bei Heikki Tuuri bestätigte diesen Eindruck. Seine Empfehlung lautete daher, direkt auf der Programmierschnittstelle von MYSQL aufzusetzen. Daraus ergeben sich jedoch erhebliche Einschränkungen: zum einen sind einattributige Schlüssel auf maximal 255 Byte be-

grenzt, zum anderen können BLOBs (Binary Large OBjects) nur als komplette Einheiten geschrieben werden. Während die geringe maximale Schlüssellänge vielleicht nur in wenigen Fällen ein ernsthaftes Problem darstellen dürfte, könnte die Handhabung von BLOBs nur dadurch verbessert werden, dass das Speichersystem sie selbst geeignet partitioniert und in separaten Datensätzen abspeichert. Wegen des damit verbundenen, erheblichen Programmiermehraufwands wird auf eine Implementierung auf Basis von MYSQL im Rahmen dieser Diplomarbeit verzichtet.

4.4.3 ORACLE

Die Implementierung des Speichersystems auf Basis von ORACLE wurde im Rahmen dieser Diplomarbeit nur unter WINDOWS ausgetestet, wobei der ORACLE-Server unter WINDOS NT installiert war und Client-Verbindungen von verschiedenen WINDOWS-Versionen (98/ME/NT) aus aufgebaut wurden. Grundsätzlich sollte jedoch auch ein Betrieb unter LINUX und SOLARIS relativ problemlos möglich sein, da die OCI-C++-Bibliothek ursprünglich für UNIX-Systeme entwickelt wurde und für diese Diplomarbeit an die MINGW-Compilerumgebung unter WINDOWS angepasst wurde.

Da das ORACLE-Datenbanksystem mehrbenutzerorientiert ist und auch bei einer lokalen persönlichen Installation als Server arbeitet, wird von der SECONDO-Speichersystem-Schnittstelle nur der Mehrbenutzerbetrieb unterstützt, d.h. eine Anforderung, das Speichersystem in Einzelbenutzermodus zu öffnen wird wie im Mehrbenutzermodus behandelt.

Eine Schwierigkeit bei der Implementierung ergab sich daraus, dass in ORACLE vor und nach allen Datendefinitonsanweisungen (wie beispielsweise `CREATE TABLE`) implizit eine `COMMIT`-Anweisung ausgeführt wird. Da es jedoch innerhalb einer benutzerdefinierten Transaktion möglich sein muss, neue *SmiFiles* anzulegen, ohne die Transaktion zu unterbrechen, werden für jeden Client zwei Verbindungen zu ORACLE aufgebaut: eine für die Bearbeitung von Datenmanipulationsanweisungen und eine für die Bearbeitung von Datendefinitionsanweisungen.

In der Implementierung werden *SmiFiles* auf relationale Tabellen und *SmiRecords* auf Zeilen dieser Tabellen abgebildet. Während die Speicherung der Schlüssel von *SmiRecords* mit Hilfe adäquater Datentypen (`NUMBER` für numerische Schüssel, `VARCHAR2` für alle anderen) erfolgt, wird der eigentliche Datenteil eines *SmiRecords* stets als BLOB verwaltet.

Die maximale Schlüssellänge ist einerseits durch die maximale Länge eines Wertes vom Typ `VARCHAR2`, nämlich 4000 Bytes, begrenzt, hängt jedoch andererseits auch von der physischen Seitengröße, mit der die Datenbank installiert wurde, ab

(siehe Tabelle 4.3). Die maximale Schlüssellänge muss daher in der Header-Datei *SecondoSMI* als Wert der Konstante `SMI_MAX_KEYLEN` korrekt definiert werden, damit es zur Laufzeit nicht zu Fehlern kommt.

Seitengröße	max. Schlüssellänge
2 kB	758 Bytes
4 kB	1578 Bytes
8 kB	3218 Bytes
16 kB	4000 Bytes

Tabelle 4.3: Maximale Schlüssellängen in ORACLE

Der Datenteil eines *SmiRecord* wird in ORACLE als BLOB gespeichert. Da von ORACLE in einer Tabelle nur Referenzen auf BLOBs gespeichert, die BLOBs selbst jedoch an anderer Stelle angelegt werden, ergibt sich beim Einfügen eines neuen *SmiRecord* folgende Schwierigkeit:

Beim Einfügen eines Datensatzes kann zunächst nur eine Referenz auf ein leeres BLOB angelegt werden. Zu diesem Zweck steht die eingebaute Funktion `EMPTY_BLOB` zur Verfügung. Um das BLOB selbst einzufügen, muss zunächst die Referenz ermittelt werden. Neben der klassischen Methode, den gerade eingefügten Satz anschließend zu selektieren (`SELECT`-Anweisung), bietet ORACLE eine Erweiterung der `INSERT`-Anweisung um eine `RETURNING`-Klausel an, die es erlaubt, die Referenz auf das neue BLOB unmittelbar beim Einfügen zu ermitteln. Die OCI-C++-Bibliothek unterstützt diese effiziente Variante jedoch zur Zeit nicht. Stattdessen muss also die klassische Methode, die neu eingefügte Zeile zu selektieren, verwendet werden, um in der Folge das BLOB schreiben zu können.

Solange die Datensätze über einen eindeutigen Schlüssel verfügen, ist dies natürlich kein Problem. Da jedoch auch *SmiFiles* ohne eindeutigen Schlüssel unterstützt werden sollen, wird für solche *SmiFiles* automatisch eine weitere Datenspalte mit einer Sequenznummer ergänzt, die zusammen mit dem Schlüssel eine eindeutige Identifizierung jeder Datenzeile erlaubt.

4.5 Systemkatalog

Aus der bisherigen Modula-Implementierung des Systemkatalogs konnte außer den grundlegenden Schnittstellendefinitionen nur wenig in die neue C++-Implementierung übernommen werden.

Ein Grund dafür war, dass in der SHORE-basierten Implementierung des Speichersystems Datensätze nur über ihre Satznummer identifiziert werden konnten. Um auch Zeichenketten als Satzschlüssel verwenden zu können, wurde zusätzlich zur Datendatei eine Namensindex-Datei gepflegt. Eine Beibehaltung dieses Konzepts hätte bedeutet, im neuen Speichersystem bereits vorhandene Funktionalität nachzuprogrammieren, da dort Zeichenketten als Satzschlüssel direkt unterstützt werden. Die auf dem Namensindex-Konzept aufbauenden Codeteile konnten somit nicht übernommen werden.

Ein weiteres Konzept, das ersetzt werden musste, betrifft die persistente Speicherung von Typinformationen. Diese liegen intern stets als geschachtelte Listen vor und wurden bisher aufwendig in eine äquivalente persistente Darstellung überführt. Für die Speicherung wurde die Typinformation aus der internen *Nested-List*-Darstellung in eine Zeichenkette umgewandelt, in eine temporäre Datei geschrieben, von dort mit Hilfe eines Parsers wieder eingelesen und in die persistenten geschachtelten Listen gespeichert; für das Einlesen wurde der umgekehrte Weg beschritten.

Die Klasse *NestedList* stellt eigentlich einen Container für eine beliebige Anzahl geschachtelter Listen dar, wobei die Listen nicht notwendigerweise in zusammenhängenden Speicherbereichen abgelegt sind. Für die persistente Speicherung wird jedoch jeweils eine einzelne Liste in kompakter Darstellung benötigt. Daraus ergab sich auch für die neue SECONDO-Version die Schwierigkeit, die interne Darstellung einer Liste in eine geeignete kompakte persistente Darstellung zu überführen. Um unter Beibehaltung der inneren Knotenstruktur einer Liste zu einer kompakten Form zu kommen, wären zwei Listendurchläufe erforderlich: einer, um die Anzahl der Knoten je unterschiedlichem Knotentyp zu bestimmen, und einer, um die Transformation durchzuführen. Diese Vorgehensweise erschien zu aufwendig.

Stattdessen werden die Listen zur persistenten Speicherung wie bisher in ihre Darstellung als Zeichenkette überführt, sodann jedoch ohne weitere Transformationen direkt gespeichert. Beim Lesen muss diese Zeichenkette zwar mit einem Parser in ihre Listendarstellung umgewandelt werden, jedoch entfällt der Umweg über eine temporäre Datei, weil es in C++ möglich ist, Leseoperationen auf einer Zeichenkette (`stringstream`) durchzuführen. Sowohl für das Lesen als auch für das Schreiben ist somit nur noch jeweils ein Listendurchlauf erforderlich.

Um neben ausführbaren Algebren, auf die die SECONDO-Referenzimplementierung begrenzt ist, auch deskriptive Algebren unterstützen zu können, werden in der neuen SECONDO-Version nunmehr für jede Datenbank statt einem zwei Kataloge benötigt. Daher wurde die bisherige Katalogschnittstelle in zwei Bereiche aufgeteilt. Im algebra-unabhängigen Bereich finden sich alle Funktionen für die Verwaltung ganzer Datenbanken wie z.B. das Anlegen, Öffnen und Schließen; im algebra-abhängigen

Bereich finden sich dagegen alle Funktionen für die Verwaltung von Typkonstruktoren, Operatoren, Datentypen und Objekten.

Neben Objektwerten müssen auch Objektmodelle persistent gespeichert werden können. Da nicht immer beide Informationen (Wert und Modell) zu einem Objekt vorhanden sind, stellt der Katalog einem Objekt sowohl für den Wert als auch für das Modell ein eigenes *SmiRecord* für die Speicherung zur Verfügung. Die eigentliche Speicherung wird durch zwei Methoden des Typkonstruktors eines Objekts vorgenommen. Für den Algebra-Entwickler stehen Standard- oder Dummymethoden zur Verfügung. Die Standardmethoden lesen bzw. speichern unter Zuhilfenahme algebra-spezifischer Methoden die Darstellung des Wertes oder des Modells als geschachtelte Listen. Die Dummymethoden lesen und speichern keinerlei Information; ihre Verwendung kann sinnvoll sein, wenn die interne Objektdarstellung als einzelnes Speicherwort auch für die persistente Speicherung geeignet ist.

4.6 Verwaltung von Typen und Operatoren

In der bisherigen SECONDO-Version registrierte sich eine Algebra automatisch beim Algebra-Manager, wenn ihr statischer Konstruktor ausgeführt wurde. Diese Vorgehensweise bringt jedoch ein nicht-deterministisches Element in die Algebra-Verwaltung, da nicht der Algebra-Entwickler, sondern der C++-Compiler festlegt, wann und in welcher Reihenfolge die Konstruktoren der verschiedenen Algebra-Module aufgerufen werden.

Da die den Algebren, Typkonstruktoren und Operatoren zugeordneten Nummern vor allem im Query-Prozessor eine wesentliche Rolle spielen, sollte die Zuordnung der Nummern deterministischen Regeln folgen. Hierfür wurde, wie bereits in Abschnitt 3.4 auf Seite 40 erwähnt, eine Konfigurationsdatei geschaffen, in der alle verfügbaren Algebra-Module mit fest zugeordneten eindeutigen Nummern aufgelistet werden, unabhängig davon, ob sie in einem laufenden SECONDO-System tatsächlich aktiv verwendet werden oder nicht.

Wenn beim Binden des ausführbaren SECONDO-Systems die Algebra-Module nicht als Objekt-Dateien, sondern als statische oder dynamische Bibliotheken vorliegen, werden sie vom Binder in der Regel nicht mit eingebunden, wenn nicht mindestens eine Funktion in ihnen von einer der Systemkomponenten referenziert wird. Aus diesem Grund wird im neuen SECONDO-System eine Algebra-Initialisierungsfunktion eingeführt, über die jedes Algebra-Modul verfügen muss, und die im Algebra-Manager mit Hilfe der Konfigurationsdatei eingetragen und somit referenziert wird.

Im C++-Quelltext des Algebra-Managers muss zu diesem Zweck einerseits für jede Initialisierungsfunktion ein Funktionsprototyp sowie ein Eintrag in einer Tabelle der vorhandenen Algebra-Module erzeugt werden. Die Liste der vorhandenen Algebra-Module in der Konfigurationsdatei spezifiziert die einzelnen Module mit Hilfe von Makros. Die Konfigurationsdatei wird in den Quelltext des Moduls `AlgebraList` zweimal eingebunden, wobei die Makros jeweils unterschiedlich interpretiert werden: einmal für die Generierung der Funktionsprototypen und einmal für die Generierung der Tabelleneinträge. Hierdurch wird gewährleistet, dass die Information über die vorhandenen Algebra-Module so redundanzfrei wie möglich bereitgehalten wird. Redundanz könnte nur dann vollständig vermieden werden, wenn sämtliche Algebra-Module als dynamische Bibliotheken, die erst bei Systemstart explizit geladen werden, zur Verfügung stünden. Andernfalls sind die erforderlichen Algebra-Module zusätzlich im Haupt-Makefile zu spezifizieren.

Kapitel 5

Das neue SECONDO

In den folgenden Abschnitten wird zunächst ein Überblick über die neue Systemarchitektur und das Kommunikationsprotokoll von SECONDO gegeben. Danach werden die Bereiche von SECONDO, in denen Erweiterungen des Systems möglich sind, unter dem Blickwinkel eines Entwicklers betrachtet. Eine Beschreibung der kommando-orientierten Bedienoberfläche SecondoTTY bildet den Abschluss dieses Kapitels.

5.1 Überblick

5.1.1 Systemarchitektur

In Abbildung 5.1 auf der nächsten Seite ist die neue SECONDO-Architektur schematisch dargestellt. Ein Vergleich der neuen mit der alten Architektur (vgl. Abbildung 1.1 auf Seite 2) zeigt vor allem auf der untersten Ebene (Ebene 1) deutliche Veränderungen. Auf der obersten Ebene (Ebene 6) bietet sich ein gegenüber der alten Darstellung strukturell ähnliches, aber differenzierteres Bild. Auf den übrigen Ebenen hat sich die äußere Struktur nicht verändert.

Neben den bisherigen Werkzeugen enthält Ebene 1 zwei wesentliche neue Schnittstellenkomponenten:

1. Das Speichermanagement-Interface bietet für alle SECONDO-Komponenten eine einheitliche Schnittstelle zur persistenten Speicherung von Informationen, unabhängig von der konkreten Implementierung. Neben den in dieser Arbeit exemplarisch vorgestellten Implementierungen auf Basis der BERKELEY DB und des ORACLE DBMS sind ohne weiteres andere Realisierungen etwa auf Basis von MYSQL oder vergleichbaren Systemen denkbar, ohne dass dies Auswirkungen auf die übrigen Teile von SECONDO hat.

2. Das System-Interface kapselt betriebssystemspezifische Funktionalität vor allem im Bereich der Applikations- und Prozesssteuerung sowie der Prozess- und Netzwerkkommunikation.

Nur mit Hilfe dieser beiden Komponenten war die Portierung auf mehrere Betriebssystemplattformen erfolgreich durchführbar, so dass die neue SECONDO-Version nunmehr unter LINUX, SOLARIS und WINDOWS zur Verfügung steht. Eine Portierung auf weitere Plattformen wird zwar möglicherweise ein nicht-triviales Unterfangen sein – wie das erwähnte Beispiel PALMOS deutlich macht –, ist aber zumindest auf einige wenige Module beschränkt.

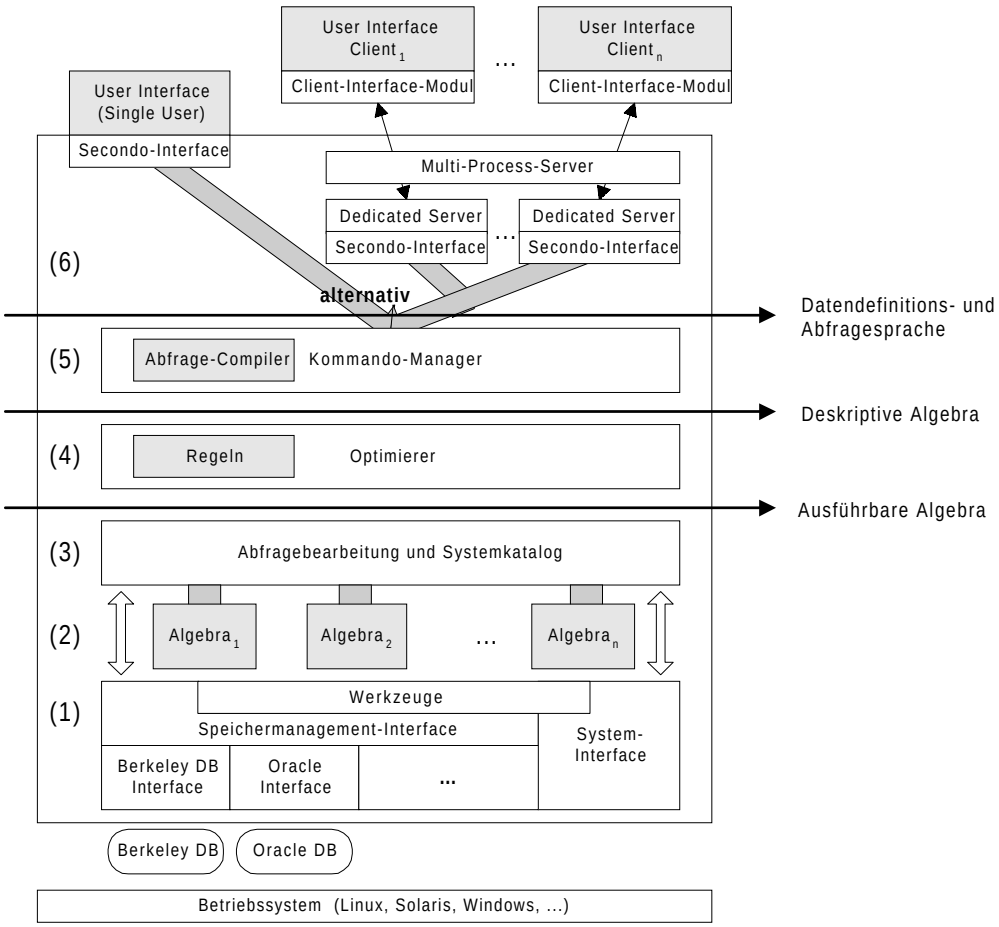

Abbildung 5.1: Neue SECONDO-Architektur

Auf Ebene 6 hat sich in erster Linie die Client/Server-Architektur verändert. Während in der alten Version der SECONDO-Server mit Hilfe von Komponenten aus dem bisherigen Speicherverwaltungssystem SHORE realisiert worden war, basiert die neue Version im wesentlichen nur auf der System-Interface-Komponente. Ein weiterer erwähnenswerter Unterschied besteht darin, dass das User-Interface-Modul für die Mehrbenutzer- und für die Einzelbenutzer-Version völlig identisch ist, da alle Unterschiede im SECONDO-Interface-Modul gekapselt wurden. Dadurch ergibt

sich bei Änderungen im User-Interface-Modul erheblich geringerer Pflegeaufwand.

Gegenüber der bisherigen SECONDO-Referenzimplementierung wurde im Systemkern (Ebenen 3 bis 5) zum einen die Unterstützung deskriptiver Algebren durch den Query-Prozessor und den Systemkatalog (siehe auch Abschnitt 5.1.1) und zum anderen die Behandlung von Abstraktionen und Funktionsobjekten integriert. Der Systemkatalog erlaubt darüberhinaus nun auch die Speicherung zusätzlicher Daten und Metadaten zu persistenten Objekten. Hierdurch wird es möglich, Algebra-Module von Aufgaben zu entlasten, die natürlicherweise in den Verantwortungsbereich des Systemkatalogs fallen.

Die interne Verwaltung von Algebra-Modulen (Ebene 2) wurde grundlegend überarbeitet, um eine deterministische Zuordnung von Algebra-Identifikatoren zu gewährleisten. Hieraus ergeben sich für einen Algebra-Entwickler einige einzuhaltende Randbedingungen, die in Abschnitt 5.2.1 auf Seite 68 näher erläutert werden.

Klassenstruktur

Das Klassendiagramm in Abbildung 5.2 auf der nächsten Seite gibt einen Überblick über die C++-Klassenstruktur hinsichtlich der Vererbungs-, Aggregations- und Assoziationsbeziehungen. Verbindungslinien mit einem Dreieckssymbol bezeichnen Vererbungsbeziehungen; Aggregationen werden durch Verbindungslinien mit einer Raute kenntlich gemacht, wobei 1:n-Kardinalitäten durch den Buchstaben *n* verdeutlich werden; Assoziationen, die für funktionale Abhängigkeiten zwischen Klassen stehen, werden schließlich durch gestrichelte Linien dargestellt.

Stellvertretend für die Klassen der Speichersystemschnittstelle ist im Diagramm nur die Klasse *SmiEnvironment* aufgeführt, um die Darstellung nicht zu unübersichtlich werden zu lassen. Details zur Klassenstruktur der Speichersystemschnittstelle können dem Klassendiagramm in Abbildung 4.1 auf Seite 48 entnommen werden.

Die Dokumentation der Klassen und Klassenschnittstellen befindet sich in den Anhängen C bis G ab Seite 91.

Aufbau der Datenbank

Innerhalb einer Instanz des SECONDO-Systems kann es mehrere SECONDO-Datenbanken geben, die vom Anwender des Systems angelegt werden. Das Speichersystem stellt eine Liste aller Datenbanken einer SECONDO-Instanz zur Verfügung. Für die Verwaltung einer einzelnen Datenbank und die in ihr enthaltenen *SmiFiles* verwendet das Speichersystem je Datenbank drei weitere spezielle Informationsstrukturen: *Sequences*, *FileCatalog* und *FileIndex*. *Sequences* dienen der Generie-

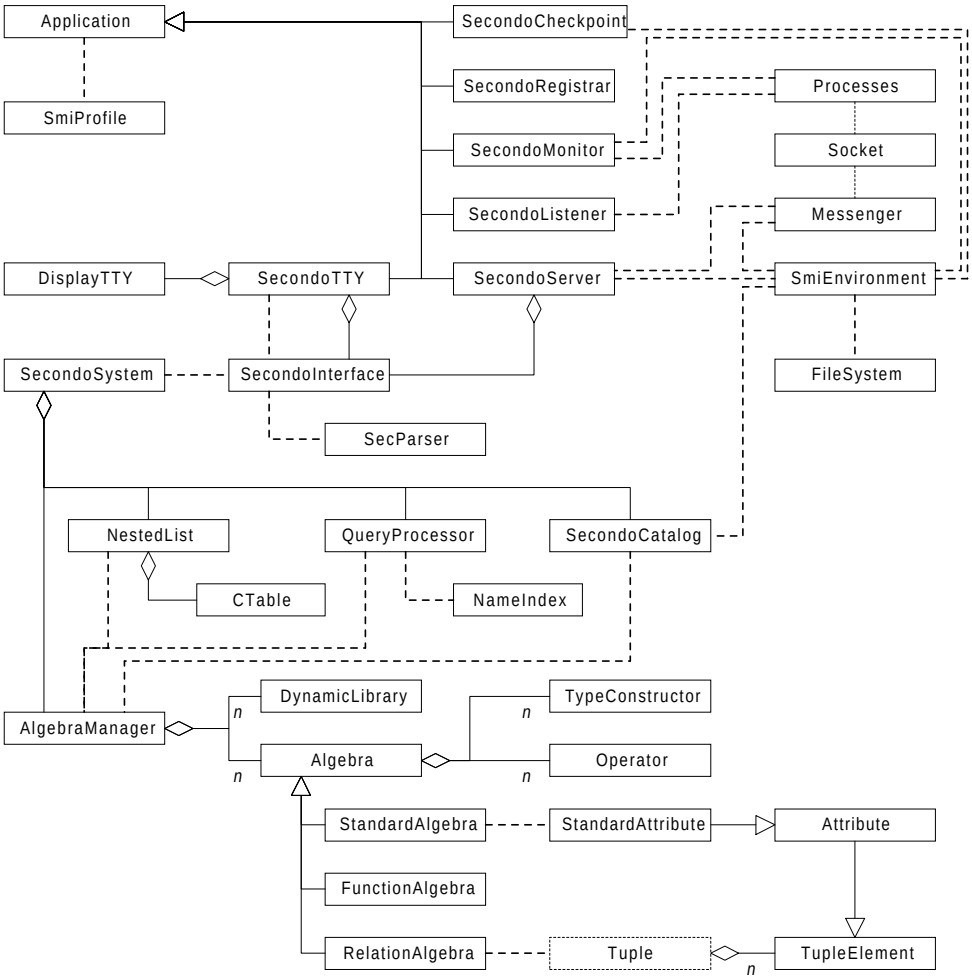

Abbildung 5.2: Klassendiagramm

rung eindeutiger numerischer Identifikatoren für *SmiFiles*. Neben der numerischen Identifikation kann ein *SmiFile* auch einen Namen erhalten. Eine Liste der benannten *SmiFiles* wird im *FileCatalog* verwaltet, dessen Primärschlüssel die eindeutige *SmiFileId* ist; als Sekundärschlüssel, der mit Hilfe des *FileIndex* verwaltet wird, dient der Name.

Aus Effizienzgründen werden für die genannten vier Basisinformationsstrukturen spezielle Eigenschaften der zugrundeliegenden Implementierung ausgenutzt – z.B. die direkte Unterstützung von Sekundärindizes bei der BERKELEY DB oder die Sequenzen, Indizes und Systemtabellen bei ORACLE. Einzelheiten hierzu können der Programmdokumentation in Anhang E ab Seite 157 entnommen werden.

Jede SECONDO-Datenbank enthält für die Unterstützung deskriptiver sowie ausführbarer Algebren zwei Systemkataloge – für jeden Algebra-Typ einen. In diesen werden Informationen über Typkonstruktoren, Operatoren, Typen und Objekte verwaltet. In Abbildung 5.3 auf der nächsten Seite ist der Aufbau einer SECONDO-Datenbank schematisch dargestellt und wird nachfolgend näher erläutert.

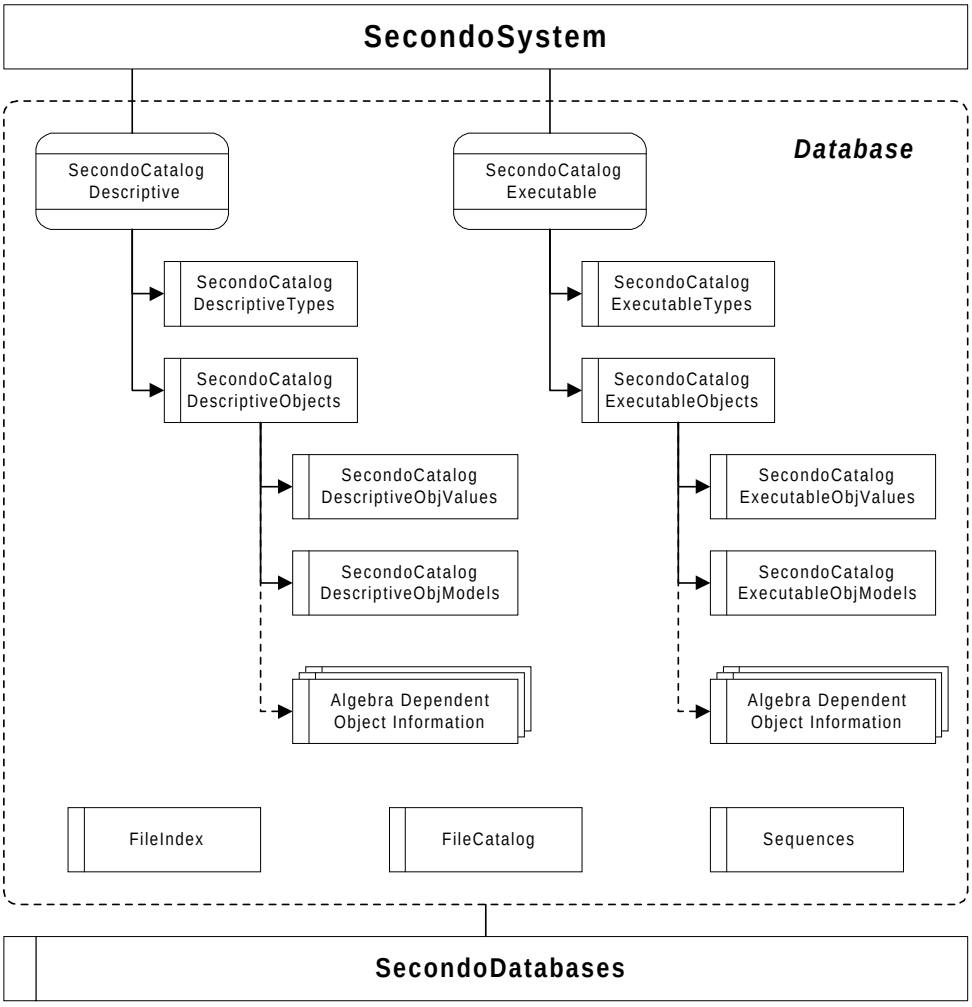

Abbildung 5.3: Aufbau einer SECONDO-Datenbank

Die Menge der Typkonstruktoren und Operatoren wird durch die aktivierten Algebra-Module definiert und verändert sich während der Laufzeit nicht. Es genügt also, Informationen über die Typkonstruktoren und Operatoren beim Systemstart mit Hilfe der Algebra-Module in die Katalogstruktur zu laden. Dahingegen kann ein Anwender nach Belieben eigene Typen und Objekte anlegen, bearbeiten und entfernen. Für die persistente Speicherung von Informationen über Typen und Objekte (Namen, Typausdrücke, Algebra- und Typkonstruktoridentifikation usw.) enthält jeder Katalog zwei *SmiFiles*. Zusätzlich stellt ein Systemkatalog zwei weitere *SmiFiles* für Objekte zur Verfügung, in denen Objektwerte und -modelle abgelegt werden können, falls ihr Platzbedarf ein einzelnes Speicherwort übersteigt. Algebra-Module können zu Objekten selbständig weitere *SmiFiles* anlegen, z.B. für die Verwaltung von Tupelmengen einer Relation.

Die Systemkatalog-Schnittstelle wird in der Dokumentation der Module `SecondoSystem` und `SecondoCatalog` in Anhang G ab Seite 211 beschrieben.

5.1.2 Client/Server-Kommunikation

Im Mehrbenutzerbetrieb überträgt das Client-Programm SECONDO-Kommandos zur Bearbeitung an einen SECONDO-Server. Die Ergebnisse der Bearbeitung werden vom Server an den Client zurückgegeben. Für die Kommunikation zwischen Client und Server wurde ein recht einfach gehaltenes Protokoll festgelegt, das auf einer Mischung von XML-Syntax (für die Protokollstruktur) und *Nested-List*-Darstellung (für die Nutzdaten) basiert.

Der Kommunikationsbedarf zwischen Client und Server eines SECONDO-Systems kann in drei Bereiche aufgeteilt werden: Auf- und Abbau der Verbindung, Ausführung von SECONDO-Kommandos, und spezielle Anfragen hinsichtlich der Zuordnung von Typen zu Algebren und Typkonstruktoren.

Verbindungsauf- und abbau

In Abbildung 5.4 auf der nächsten Seite ist das Protokoll für Auf- und Abbau der Verbindung dargestellt. Auf den Verbindungswunsch eines Clients antwortet der Server entweder mit einer negativen oder positiven Antwort, abhängig von der Überprüfung der IP-Adresse des Clients.

Für die Zugriffskontrolle auf das SECONDO-System wurde ein einfacher Mechanismus implementiert, der bei einer Verbindungsanfrage eines Client an den Server die IP-Adresse des Clients anhand einer IP-Adressliste überprüft und in Abhängigkeit vom Prüfergebnis den Zugriff erlaubt oder verweigert.

Zwei Betriebsarten werden dabei unterstützt: entweder wird der Zugriff allen Clients gewährt, die nicht in einer Ausschlussliste (Blacklist) geführt werden, oder der Zugriff wird nur Clients gewährt, die explizit in einer Zulassungsliste (Whitelist) aufgeführt sind.

Falls die Verbindung angenommen wird, wartet der Server auf die Übermittlung der Anmeldungsdaten. Falls die Anmeldedaten in Ordnung waren, wird eine Begrüßungsmeldung übertragen. In der vorliegenden SECONDO-Version wird zur Zeit noch kein Authentifizierungsmechanismus auf Basis von Benutzerkennungen und Passwörtern unterstützt, jedoch wird die Benutzerkennung im System registriert. Anschließend wartet der Server auf die Übertragung von SECONDO-Kommandos.

Wenn der Client die Verbindung beenden möchte, wird eine entsprechende Nachricht an den Server geschickt, der daraufhin die Verbindung beendet. Falls die Verbindung unvorhergesehen unterbrochen wird, setzt der Server die letzte Transaktion des Clients zurück und schließt die betroffene Datenbank.

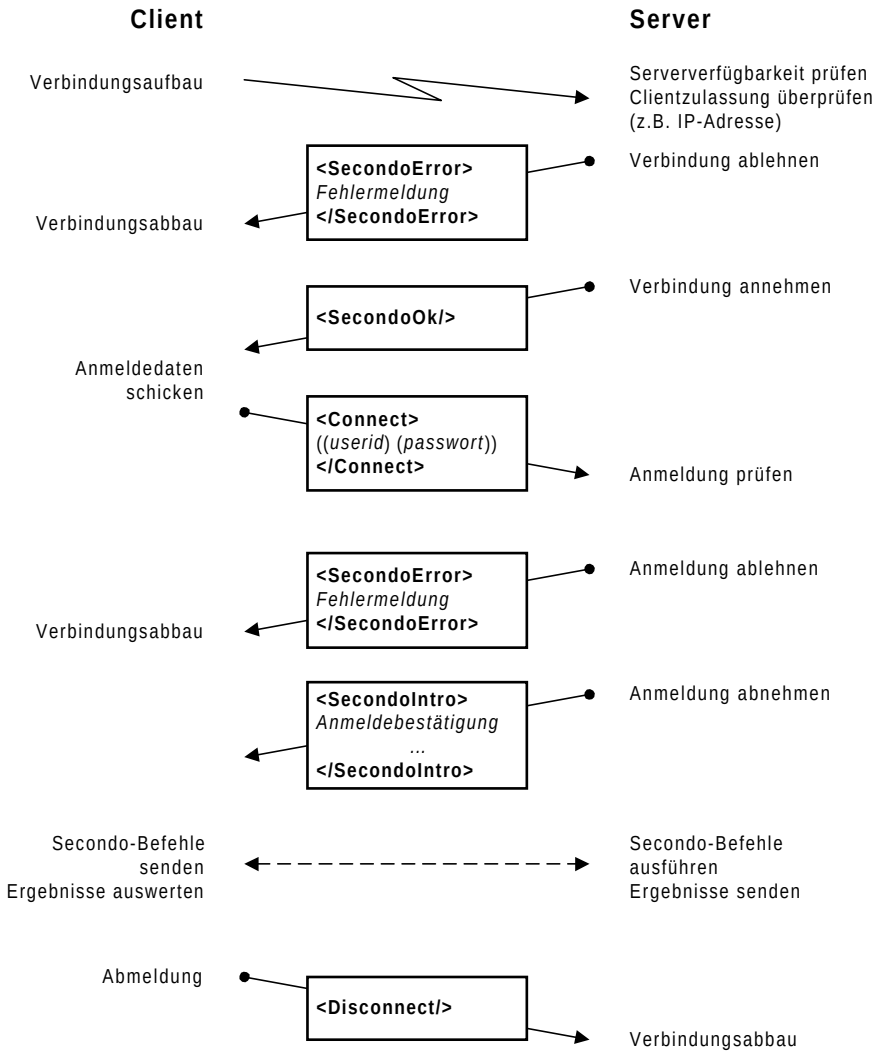

Abbildung 5.4: Verbindungsauf- und Abbau

Ausführung von SECONDO-Kommandos

Die Ausführung von SECONDO-Kommandos erfolgt in der Regel über das in Abbildung 5.5 auf der nächsten Seite dargestellte Protokoll. Es gibt jedoch zwei spezielle Kommandos, bei denen zusätzlich eine Datei zwischen Client und Server übertragen werden muss: die Sicherung einer Datenbank in eine Datei und die Wiederherstellung einer Datenbank aus einer Datei.

Die Vorgehensweise bei der Sicherung einer Datenbank ist in Abbildung 5.6 auf der nächsten Seite dargestellt. Der Client stellt an den Server die Anforderung, eine Datenbank zu sichern. Falls die Sicherung erfolgreich durchgeführt werden konnte, fordert der Server den Client auf, sich zum Empfang der Sicherungsdatei bereit zu machen, andernfalls erfolgt eine Fehlermeldung. Falls der Client empfangsbereit ist, fordert er den Server auf, die Daten der Datei zu übertragen. Zum Abschluss erhält der Client noch eine Statusmeldung vom Server.

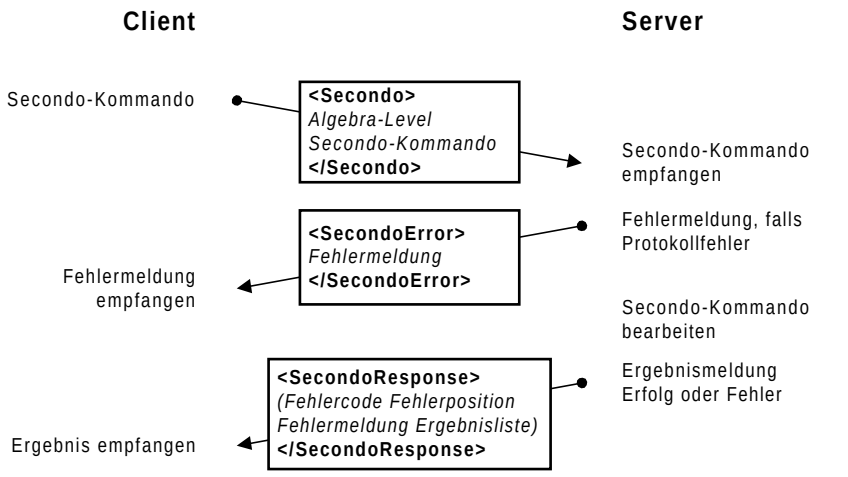

Abbildung 5.5: Übermittlung von SECONDO-Kommandos

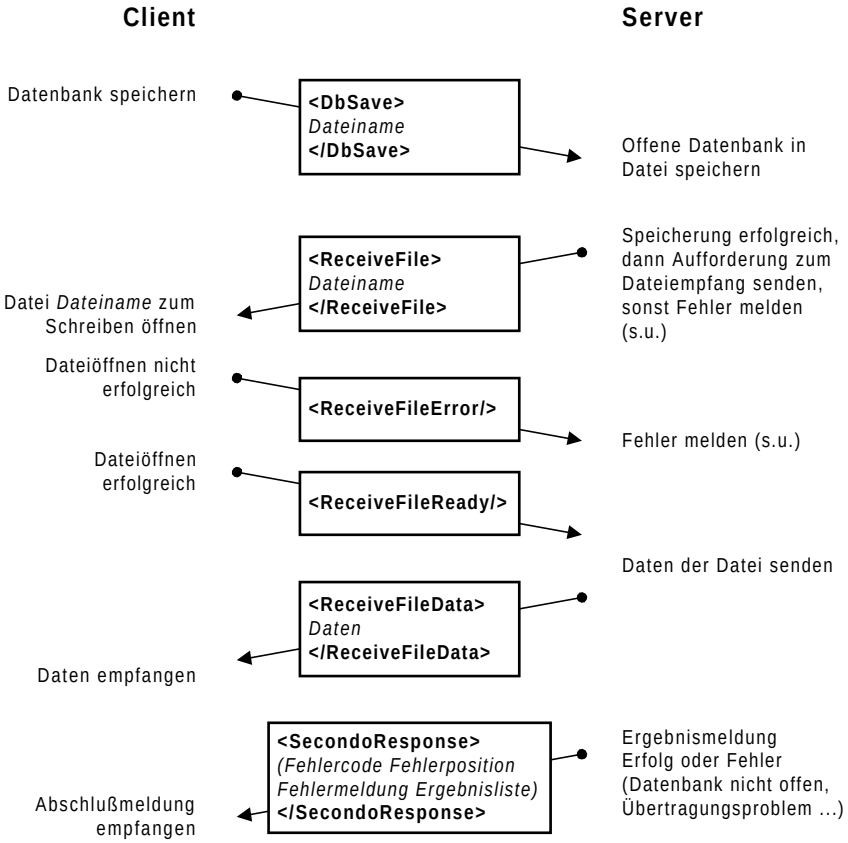

Abbildung 5.6: Datenbank sichern

In Abbildung 5.7 auf der nächsten Seite ist die Vorgehensweise bei der Wiederherstellung einer Datenbank dargestellt. Der Client stellt an den Server die Anforderung, eine Datenbank aus einer Sicherungsdatei wieder herzustellen. Wenn der Server empfangsbereit ist, fordert er den Client auf, die Daten der Datei zu übertragen. Falls die Sicherungsdatei korrekt übertragen werden konnte, wird versucht,

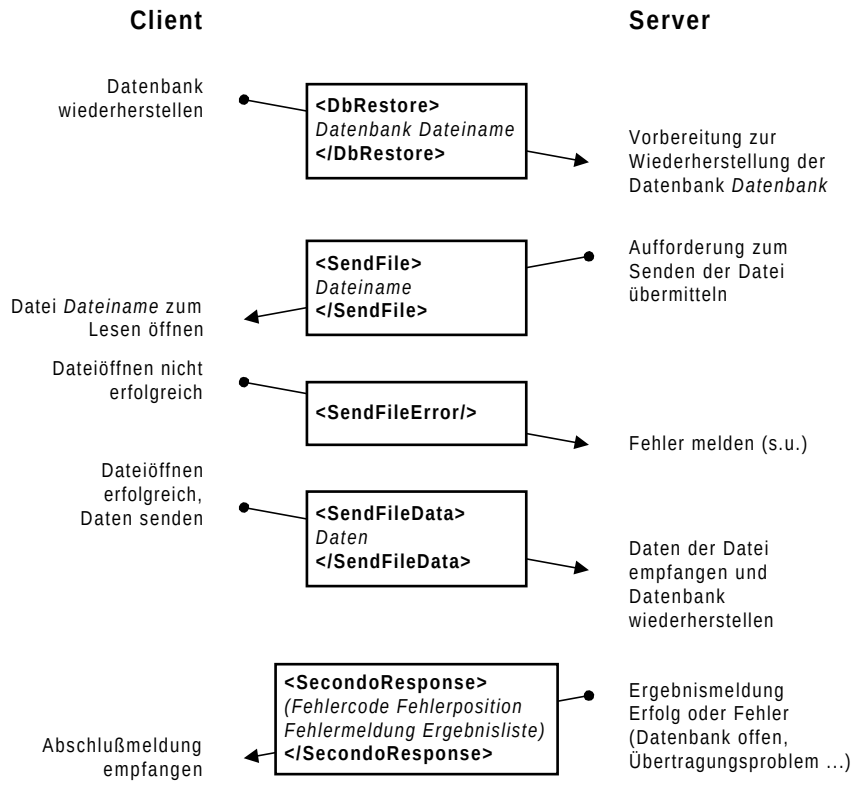

Abbildung 5.7: Datenbank wiederherstellen

die Datenbank wieder herzustellen. Zum Abschluss schickt der Server eine Statusmeldung an den Client, in der mitgeteilt wird, ob die Wiederherstellung erfolgreich durchgeführt werden konnte.

Spezielle Kommandos zur Typidentifizierung

Damit die Ergebnisse der Ausführung eines SECONDO-Kommandos vom Client in geeigneter Weise dargestellt werden können, ist häufig eine Zuordnung der Ergebnistypen zu bestimmten Algebren und Typkonstruktoren erforderlich oder zumindest nützlich. Aus diesem Grund unterstützt das SECONDO-Interface drei spezielle Anfragen zur Typidentifizierung, für die in Abbildung 5.8 auf der nächsten Seite das jeweilige Kommunikationsprotokoll dargestellt ist. Zum einen kann ein Typausdruck in eine Darstellung umgewandelt werden, in der jeder Typ durch ein numerisches Paar, bestehend aus Algebra- und Typkonstruktor-Identifikator, ersetzt wird (`NumericType`). Zum anderen können zu einem benannten Typ die zugehörigen Algebra- und Typkonstruktor-Identifikatoren bestimmt werden (`GetTypeId`). Zu guter Letzt kann zu einem Typausdruck der zugehörige (benannte) Typ samt seiner Algebra- und Typkonstruktor-Identifikatoren ermittelt werden (`LookUpType`).

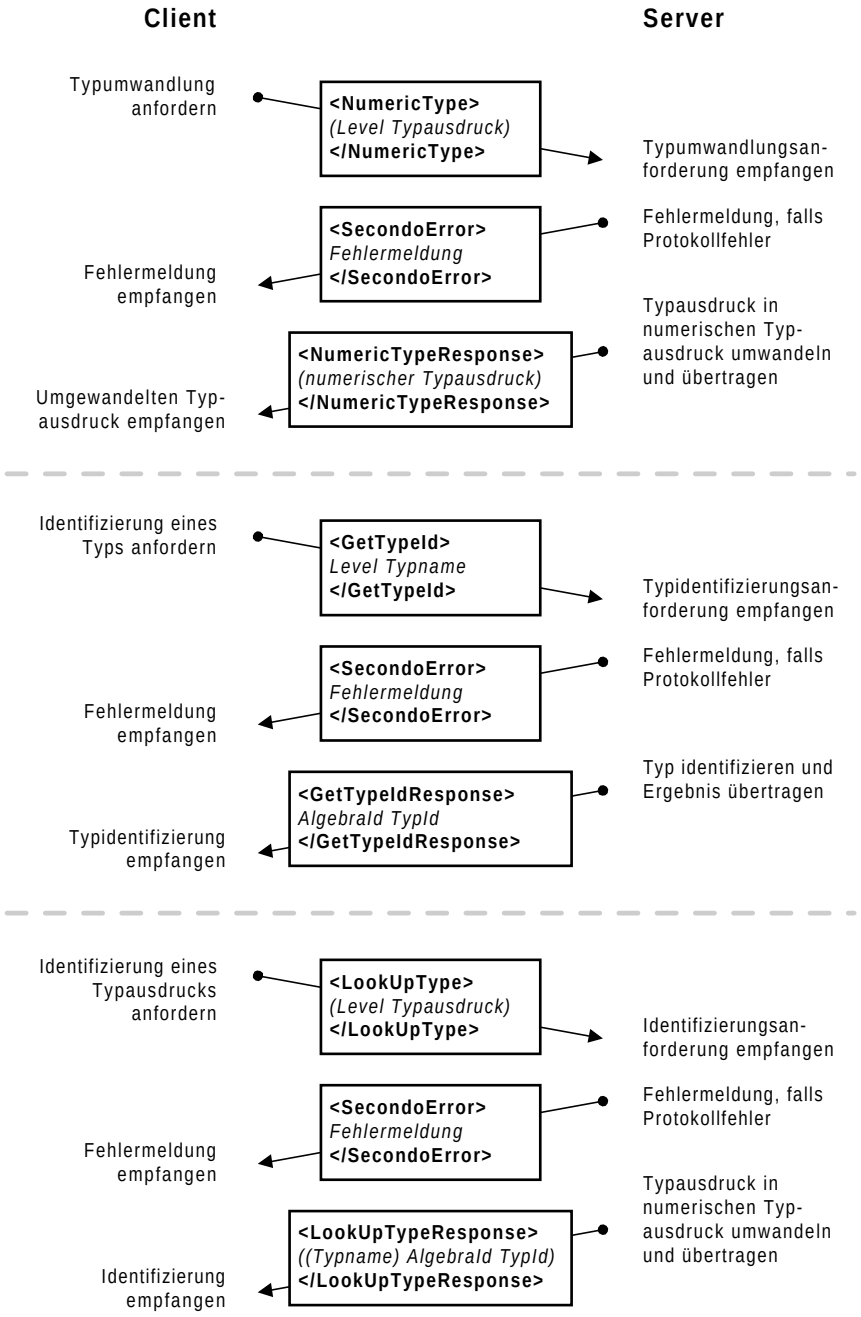

Abbildung 5.8: Spezielle Anfragen zur Identifizierung von Typen

5.2 Erweiterbarkeit

Für die Durchführung von Änderungen oder Erweiterungen am SECONDO-System ist ein Grundverständnis der Code-Organisation erforderlich. Abbildung 5.9 zeigt einen Überblick über die Verzeichnisstruktur.

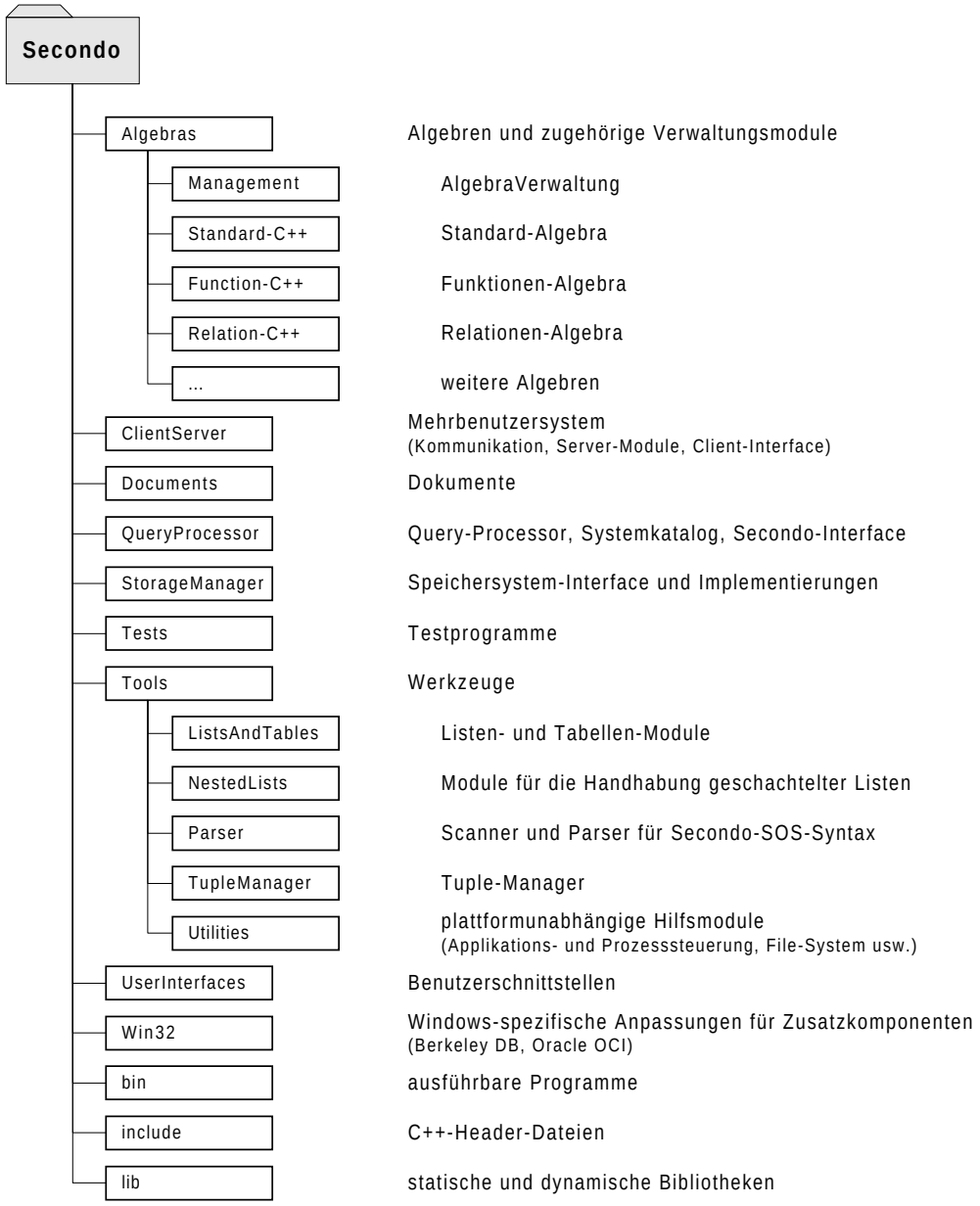

Abbildung 5.9: Verzeichnisstruktur von SECONDO

Je nach Aufgabenbereich sind verschiedene Unterverzeichnisse betroffen: Arbeiten am Systemkern werden in der Regel schwerpunktmäßig Module im Verzeichnis `QueryProcessor` haben; die Bereitstellung von Algebra-Modulen betrifft in erster Linie Dateien im Verzeichnis `Algebras` und dessen Unterverzeichnissen; zusätzliche Benutzerschnittstellen werden im Verzeichnis `UserInterfaces`

untergebracht; Auswirkungen einer zusätzlichen Speichersystemimplementierung sind auf das Verzeichnis `StorageManager` begrenzt. Sämtliche Werkzeug-Komponenten befinden sich in Unterverzeichnissen des Verzeichnisses `Tools`. Insbesondere sind alle systemabhängigen Module im Verzeichnis `Tools/Utilities` untergebracht – mit Ausnahme des Moduls für die Netzwerkkommunikation über Sockets, welches im Verzeichnis `ClientServer` abgelegt ist.

Die Verzeichnisse `include`, `lib` und `bin` sind für C++-Header-Dateien, Laufzeitbibliotheken bzw. ausführbare Programme vorgesehen.

5.2.1 Algebra-Module

Die Erweiterbarkeit des SECONDO-Systems basiert wesentlich darauf, dass über definierte Schnittstellen zusätzliche Algebra-Module eingebunden werden können. Für die Implementierung einer neuen Algebra wird von der Klasse *Algebra* eine neue Klasse abgeleitet, die die neue Algebra repräsentiert. Von dieser Klasse muss genau eine statische Instanz angelegt werden. Die Beschreibungen der weiteren Vorgehensweise bei der Implementierung einer Algebra in [GFB+97] und im Tutorial von Stefan Dieker [Die97] haben im wesentlichen weiter Gültigkeit.

Zu beachten ist jedoch, dass die Schnittstellen vieler Algebra-Funktionen C++-konformer definiert wurden, d.h. dass insbesondere Pointer durch Referenzen (Beispiel: `ListExpr*` wurde zu `ListExpr&`) und C-Zeichenketten (`char*`) konsequent durch den C++-Datentyp `string` ersetzt wurden. Weiterhin wurde der Datentyp `ADDRESS` in allen Funktionen, die über den Algebra-Manager aufgerufen werden, durch den Datentyp `Word`, eine Struktur mit Varianten (`union`), ersetzt, um von Annahmen über den Speicherbedarf eines *Wortes* in der zugrunde liegenden Prozessorarchitektur unabhängig zu sein. Eine Beschreibung der benötigten Datentypen und Funktionsprototypen findet sich in den Header-Dateien `AlgebraTypes` und `AlgebraManager` in Anhang D.

Gegenüber den bisherigen Algebra-Implementierungen benötigen Typkonstruktoren zusätzliche Parameter: die Adresse einer *Property*-Funktion[1], Adressen von *Modell*-Funktionen[2] sowie bei Bedarf Adressen spezieller *Persist*-Funktionen für Objektwerte und -modelle. Auch Operatorkonstruktoren benötigen zusätzliche Parameter, nämlich eine Spezifikation in Form einer Zeichenkette sowie Adressen von *Modell*-Funktionen. Schließlich sollte im Konstruktor einer Algebra nach den

[1] *Property*-Funktionen fehlten in bisherigen C/C++-Algebren, waren jedoch in den Modula-Algebren in der Regel vorhanden.

[2] Exemplarisch wurden bei der Portierung der C++-Standard-Algebra die vorhandenen Modell-Funktionen der Modula-Implementierung integriert.

Aufrufen der Methode `AddTypeConstructor` für die Typkonstruktoren ihre Methode `AssociateKind` ausgeführt werden.[3]

Neu hinzugekommen ist auch eine Initialisierungsfunktion, die eine Algebra mit Referenzen auf den Query-Prozessor und den im SECONDO-System verwendeten *Nested-List*-Container versorgt und eine Referenz auf die Algebra-Instanz an den Algebra-Manager zurückgibt. Eine typische Implementierung dieser Funktion zeigt Tabelle 5.1; dabei sind die fettgedruckten Namen Platzhalter für die tatsächlichen Namen der Algebra und der Algebra-Instanz. Die Funktion muss als C-Funktion deklariert werden, damit sie im Falle des dynamischen Ladens gefunden werden kann.

```
static NestedList* nl;
static QueryProcessor* qp;

extern "C" Algebra*
Initialize AlgebraName( NestedList* nlRef,
                        QueryProcessor* qpRef )
{
  nl = nlRef;
  qp = qpRef;
  return (&algebrainstance);
}
```

Tabelle 5.1: Algebra-Initialisierungsfunktion

Schließlich muss eine neue Algebra dem System bekannt gemacht werden, indem sie in die Konfigurationsdatei `AlgebraList.i` eingetragen wird. Hierbei ist der Algebra eine eindeutige Nummer zuzuordnen und zu definieren, in welcher Form (statisch oder dynamisch) die Algebra zur Laufzeit in SECONDO eingebunden wird. Davon abhängig sind auch die benötigten Einträge in den verschiedenen *Makefiles*. Für das Makefile im Verzeichnis des neuen Algebra-Moduls kann das Makefile der Standard-Algebra als Muster dienen. Im Makefile im übergeordneten Verzeichnis `Algebras` ist unterhalb des Eintrags `makedirs` eine Zeile der Form „`$(MAKE) -C` **`NeueAlgebra`**“ und unterhalb des Eintrags `buildlibs` eine Zeile der Form „`$(MAKE) -C` **`NeueAlgebra`** `buildlibs`“ zu ergänzen, wobei ***NeueAlgebra*** für den Namen des Verzeichnisses steht, in dem sich die Quelltexte des neuen Algebra-Moduls befinden. Falls die Algebra statisch eingebunden werden soll,

[3] In der bisherigen SECONDO-Version wurde die „Sorten“-Zuordnung mit Hilfe globaler Funktionen in nicht-objektorientierter Weise durchgeführt.

ist darüberhinaus in den betriebssystemspezifischen Makefiles eine entsprechende Ergänzung der Variablen `ALGLIB` vorzunehmen.

5.2.2 Alternative Speichersysteme

Eine Grundanforderung an die Implementierung eines alternativen Speichersystems ist die Einhaltung und volle Unterstützung der Speichersystemschnittstelle, deren detaillierte Beschreibung der Dokumentation in Anhang E ab Seite 157 entnommen werden kann. Vor der Implementierung weiterer Speichersysteme empfiehlt es sich, zunächst mindestens eine der vorhandenen beiden Implementierungen im Detail zur Kenntnis zu nehmen.

Zum einen sind die Klassen *SmiEnvironment*, *SmiFile* und *SmiFileIterator* (sowie die von diesen beiden abgeleiteten Klassen für die beiden *SmiFile*-Typen) sowie *SmiRecord* auf Basis des neuen Speichersystems zu implementieren. Zum anderen wird jede Implementierung die in Abschnitt 5.1.1 auf Seite 59 beschriebenen Basisinformationsstrukturen (*Sequences*, *FileCatalog* und *FileIndex*) bereitstellen müssen, auf die sich die Methoden zur Verwaltung von *SmiFiles* stützen.

Abhängig von den Eigenschaften des neuen Basisspeichersystems kann insbesondere die Behandlung von Datensätzen (*SmiRecords*) Schwierigkeiten in der Umsetzung mit sich bringen. Konzeptionell ist nämlich das Einfügen, Löschen und Selektieren von Datensätzen getrennt vom Lesen oder Schreiben des Inhalts[4] der Datensätze. Wenn nun das Basisspeichersystem z.B. keine Cursor-Unterstützung[5] oder vergleichbare Funktionalität bietet, erfordert jeder Lese- oder Schreibzugriff auf den Datensatz eine erneute Selektion, die in der Regel allerdings durch das Speichersystem unter Ausnutzung von Cache-Speicher sehr effizient durchgeführt werden kann. Eine weitere Schwierigkeit kann die Umsetzung des Konzepts, dass der Inhalt von Datensätzen partiell gelesen oder geschrieben werden kann, mit sich bringen. In MySQL beispielsweise können zur Zeit BLOBs zwar sehr einfach partiell gelesen, aber nur als Ganzes geschrieben werden. In einem solchen Fall müsste u.U. eine eigene Zerlegung in Bruchstücke geeigneter Größe verwaltet werden.

5.2.3 Benutzerschnittstellen

Die Entwicklung weiterer Benutzerschnittstellen, wie z.B. graphische Bedienoberflächen, ist im Blick auf die Anbindung an SECONDO recht einfach zu realisieren, falls die Programmierung in C++ erfolgt. Für diesen Fall steht das Modul

[4] Die Daten sind in allgemeinen als BLOBs gespeichert.

[5] Ein Cursor kennzeichnet in der Regel die Datensatzmenge einer Selektion.

`SecondoInterface` zur Verfügung, dessen Schnittstelle in Anhang C beschrieben wird. Abhängig davon, welches Objektmodul in das ausführbare Programm eingebunden wird, entsteht eine Einzelbenutzer- oder Mehrbenutzer-Client-Version.

Für eine Einzelbenutzerversion ist das Objektmodul `SecondoInterface.o` einzubinden, für eine Mehrbenutzer-Client-Version `SecondoInterfaceCS.o`. In beiden Fällen ist zusätzlich sowohl das Objektmodul `SecondoInterface-General.o` als auch die SECONDO-Tool-Bibliothek `libsdbtool` einzubinden. Für die Einzelbenutzerversion sind darüber hinaus weitere Objektmodule und Bibliotheken zu spezifizieren. Der einfachste Weg ist, eine Kopie des entsprechenden Eintrags des Programms `SecondoTTY` im Makefile für das neu entwickelte Programm zu modifizieren.

Um die SECONDO-Schnittstelle verwenden zu können, ist zunächst eine Instanz der Klasse *SecondoInterface* anzulegen. Anschließend ist die Methode *Initialize* mit den für eine Verbindung zu SECONDO benötigten Parametern (Name einer Konfigurationsdatei, SECONDO-Server-Adresse, SECONDO-Server-Port, Benutzerkennung und Passwort) aufzurufen. Server-Adresse und -Port werden nur im Mehrbenutzerbetrieb benötigt. Nach erfolgreicher Initialisierung kann die Methode *Secondo* für die Ausführung von SECONDO-Kommandos verwendet werden. Eine bestehende Verbindung muss vor der Beendigung des Programms mit der Methode *Terminate* unterbrochen werden.

Falls bei der Bearbeitung von SECONDO-Kommandos Fehler auftreten, können die zugehörigen Fehlercodes mit Hilfe der Methode *GetErrorMessage* in einen englischen Meldungstext übersetzt werden. Für die Typidentifizierung stehen weiterhin die Methoden `NumericTypeExpr`, `GetTypeId` und `LookUpTypeExpr` zur Verfügung. Fast alle Methoden der SECONDO-Schnittstelle verwenden als Ein- oder Ausgabeparameter geschachtelte Listen. Mit der Methode *GetNestedList* kann daher eine Referenz auf den verwendeten *Nested-List*-Container ermittelt werden.

Schwieriger wird es, wenn die Implementierungssprache nicht C++ ist. In solchen Fällen ist zu empfehlen, nur Mehrbenutzer-Client-Versionen zu erstellen, um den Programmieraufwand zu begrenzen. Eine Vorgehensweise bei der Implementierung ist, die C++-Implementierung des Moduls `SecondoInterface` trotzdem zu nutzen, indem sie mittels geeigneter Kapselung (Wrapper) in die andere Sprachumgebung eingebunden wird. In Java kann hierfür das JNI[6] verwendet werden. Andernfalls ergibt sich die Notwendigkeit, das client-seitige Kommunikationsprotokoll (siehe Abschnitt 5.1.2 auf Seite 62) nachzuprogrammieren. Hierfür wird zusätzlich Funktionalität für die Socket-Kommunikation sowie für geschachtelte Listen benötigt.

[6] Java Native Interface

5.3 Einsatz des Systems

Bevor SECONDO eingesetzt und bei Bedarf angepasst oder erweitert werden kann, muss das System installiert und konfiguriert werden. In den folgenden Abschnitten wird beschrieben, welche Schritte nötig sind, um aus der SECONDO-Distribution ein lauffähiges System zu erstellen. Zuerst wird die Installation in Abhängigkeit vom Zielbetriebssystem und daran anschließend die Konfiguration des SECONDO-Systems im Einzelnen erläutert. Zuletzt wird die Bedienung des Systems sowohl im Einzelbenutzer- als auch im Mehrbenutzerbetrieb beschrieben.

5.3.1 Installation

Zunächst ist auf der Festplatte ein neues leeres Verzeichnis anzulegen, in das der Inhalt des Verzeichnises `Secondo` auf der CD-ROM zu kopieren ist. Daran anschließend ist in der Datei `makefile.config` das Kommentarzeichen # in genau der Zeile zu entfernen, die dem Betriebssystem entspricht, unter dem SECONDO installiert werden soll. Für jedes unterstützte Betriebssystem gibt es ein systemspezifisches Makefile, in dem u.a. Verzeichnisse, Dienstprogramme und Compiler-Optionen spezifiziert und ggf. speziellen Bedürfnissen angepasst werden können. In der Regel brauchen nur einige wenige Verzeichnisangaben geändert zu werden, bevor die Installation mit dem Programm `make` durchgeführt werden kann.

In jedem Fall ist sicher zu stellen, dass die benötigten Werkzeuge in aktuellen Versionen verfügbar sind. Im Rahmen der vorliegenden Diplomarbeit wurden folgende Versionen verwendet: `gcc 2.95.3`, `make 3.79`, `flex 2.5.4` und `bison 1.33`. Mit älteren Versionen kann es unter Umständen zu Problemen während der Installation kommen.

Abhängig vom jeweiligen Betriebssystem sind vorab zusätzliche Komponenten zu installieren. Die weitere Vorgehensweise wird nachfolgend für jedes unterstützte Betriebssystem separat beschrieben.

LINUX

Bevor die SECONDO-Programme und -Bibliotheken erzeugt werden können, muss die BERKELEY DB installiert sein. In der Regel ist die BERKELEY DB zwar Bestandteil einer LINUX-Distribution, es muss jedoch überprüft werden, ob die richtige Version vorhanden ist. SECONDO setzt mindestens Version 4.0.14 voraus. Gegebenenfalls ist die BERKELEY DB entsprechend der ihr beiliegenden Dokumentation zu installieren; beim Aufruf des `configure`-Skripts muss unbedingt die

Aufruf: `make [Optionen]`	
Option[1]	**Bedeutung**
`shared={`**`no`**`|yes}`	Erzeugen statischer bzw. dynamischer SECONDO-Bibliotheken
`smitype={`**`bdb`**`|ora}`	Wahl des Speichersystems (`bdb` = BERKELEY DB, `ora` = ORACLE)
`tests`	Erstellen von Testprogramme für einige Komponenten
`clean`	Verzeichnisse für eine „frische“ Installation vorbereiten
[1] Die fettgedruckten Parameter sind die Standardeinstellung der Option.	

Tabelle 5.2: Installation von SECONDO

C++-Unterstützung mit Hilfe der Option `--enable-cxx` aktiviert werden.

Im Anschluss daran ist das systemspezifische Makefile `makefile.linux` zu modifizieren. Der Variablen `BUILDDIR` ist der Name des SECONDO-Installationsverzeichnisses zuzuweisen; der Variablen `BDBDIR` das Installationsverzeichnis der BERKELEY DB. Weitere Änderungen sollten in der Regel nicht erforderlich sein; die Werte der Variablen `BDBINCLUDE` (Verzeichnis der BERKELEY DB-Header-Dateien) und `BDBLIBDIR` (Verzeichnis der BERKELEY DB-Bibliotheken) sollten jedoch auf Gültigkeit überprüft werden.

Nunmehr kann SECONDO erzeugt werden, indem das Programm `make` aufgerufen wird. Die möglichen Optionen sind in Tabelle 5.2 aufgeführt. Zu beachten ist, dass die Wahl des ORACLE-basierten Speichersystems derzeit nicht unterstützt wird.

Bevor die SECONDO-Programme aufgerufen werden können, ist einerseits die Umgebungsvariable `LD_LIBRARAY_PATH` um das `lib`-Verzeichnis von SECONDO zu ergänzen und andererseits die Konfigurationsdatei der Installation anzupassen.

SOLARIS

In einem ersten Schritt ist die BERKELEY DB Version 4.0.14 entsprechend der ihr beiliegenden Dokumentation zu installieren. Die C++-Unterstützung muss beim Aufruf des `configure`-Skripts unbedingt mit Hilfe der Option `--enable-cxx` aktiviert werden.

Im nächsten Schritt ist das systemspezifische Makefile `makefile.solaris` zu modifizieren. Der Variablen `BUILDDIR` ist der Name des SECONDO-Installationsverzeichnisses zuzuweisen; der Variablen `BDBDIR` das Installationsverzeichnis der BERKELEY DB. Die Werte der Variablen `BDBINCLUDE` (Verzeichnis der BER-

KELEY DB-Header-Dateien) und `BDBLIBDIR` (Verzeichnis der BERKELEY DB-Bibliotheken) sollten anschließend auf Gültigkeit überprüft werden.

Je nach Konfiguration des SOLARIS-Systems kann es erforderlich sein, das `BinUtils`-Paket der GNU Compiler Collection zu installieren, da der SUN-Assembler bei einigen Modulen Schwierigkeiten macht. In der Variablen `DEFAULTCCFLAGS` ist die Option `-B` auf das `BinUtils`-Verzeichnis zu setzen. Weitere Änderungen sollten in der Regel nicht erforderlich sein.

Nunmehr kann SECONDO erzeugt werden, indem das Programm `make` aufgerufen wird. Die möglichen Optionen sind in Tabelle 5.2 auf der vorherigen Seite aufgeführt. Zu beachten ist, dass die Wahl des ORACLE-basierten Speichersystems und die Erzeugung dynamischer Bibliotheken derzeit nicht unterstützt wird.

Wie unter LINUX ist einerseits die Umgebungsvariable `LD_LIBRARAY_PATH` um das `lib`-Verzeichnis von SECONDO zu ergänzen und andererseits die Konfigurationsdatei der Installation anzupassen, bevor die SECONDO-Programme aufgerufen werden können.

WINDOWS

Bevor die SECONDO-Installation erfolgen kann, muss zunächst die *MinGW*-Umgebung eingerichtet werden. Neben *MinGW* selbst muss auf jeden Fall das Paket `w32api` für die Unterstützung der WINDOWS-Systemaufrufe installiert werden. Leider fehlen *MinGW* einige nützliche Hilfsprogramme wie z.B. `rm` zum Löschen bzw. `cp` zum Kopieren von Dateien. Die beiden genannten Programme können aus der *UnxUtils*-Sammlung ergänzt werden (siehe auch Abschnitt 4.1 auf Seite 41).[7]

Die Programme `flex` und `bison` sind in der *MinGW*-Distribution nicht enthalten, unter der Internet-Adresse `http://gnuwin32.sourceforge.net` stehen jedoch aktuelle Versionen zur Verfügung. Nach der Installation dieser Programme müssen die Umgebungsvariablen `BISON_SIMPLE` und `BISON_HAIRY` auf die entsprechenden Dateien `bison.simple` bzw. `bison.hairy` verweisen.

In den Distributionen der BERKELEY DB und der OCI C++-Bibliothek fehlt bislang eine Installationsunterstützung für die *MinGW*-Entwicklungsumgebung. Für diese Basiskomponenten des Speichersystems waren kleinere Code-Modifikationen in den Originaldistributionen erforderlich. Anpassungen, die speziell das Betriebssystem WINDOWS betreffen, sind im Unterverzeichnis `Win32` zu finden. Das Ver-

[7] Eine vollständige Installation der *UnxUtils*-Sammlung ist nicht zu empfehlen, da in ihr zum Teil veraltete Programmversionen enthalten sind. Seit etwa Mitte Mai 2002 steht mit *MSYS* eine UNIX-ähnliche Umgebung für *MinGW* zur Verfügung, die es erlauben soll, auch unter WINDOWS Entwicklungswerkzeuge wie `configure`, `autoconf` usw. zu verwenden. Im Rahmen dieser Diplomarbeit konnte das jedoch nicht mehr überprüft werden.

zeichnis enthält daher neben den modifizierten Quellcode-Dateien auch Makefiles für die automatische Erstellung der BERKELEY DB-Laufzeitbibliothek sowie der OCI C++-Klassenbibliothek.

Zunächst sind die Distributionsarchive der BERKELEY DB und der OCI C++-Bibliothek zu entpacken. Das systemspezifische Makefile `makefile.win32` ist danach zu modifizieren. Der Variablen `BUILDDIR` ist der Name des SECONDO-Installationsverzeichnisses zuzuweisen; der Variablen `BDBDIR` das Installationsverzeichnis der BERKELEY DB. Die Werte der Variablen `BDBINCLUDE` (Verzeichnis der BERKELEY DB-Header-Dateien) und `BDBLIBDIR` (Verzeichnis der BERKELEY DB-Bibliotheken) sollten anschließend auf Gültigkeit überprüft werden. Falls mit dem auf ORACLE basierenden Speichersystem gearbeitet werden soll, sind zusätzlich den Variablen `ORACLEHOME`[8] und `OCICPPDIR` die Namen des ORACLE- bzw. des OCI C++-Installationsverzeichnisses zuzuweisen. Weitere Änderungen sollten in der Regel nicht erforderlich sein.

Im nächsten Schritt werden die BERKELEY DB und bei Bedarf die OCI C++-Bibliothek erzeugt, indem in das Verzeichnis `win32` gewechselt und dort das Programm `make` aufgerufen wird. Zur Auswahl der ORACLE-Unterstützung ist die Option `smitype=ora` anzugeben.

Durch Aufruf des Programms `make` im SECONDO-Verzeichnis kann im Anschluss SECONDO erzeugt werden. Die möglichen Optionen sind in Tabelle 5.2 auf Seite 73 aufgeführt.

Bevor die SECONDO-Programme aufgerufen werden können, ist zum einen die Umgebungsvariable `PATH` um das `bin`-Verzeichnis von SECONDO zu ergänzen und zum anderen die Konfigurationsdatei der Installation anzupassen.

5.3.2 Konfiguration

Nachdem das SECONDO-System erfolgreich installiert wurde, muss es konfiguriert werden, bevor mit ihm aktiv gearbeitet werden kann. Die Konfiguration erfolgt mit Hilfe einer Konfigurationsdatei. In der SECONDO-Distribution ist eine Beispielkonfigurationsdatei enthalten, die als Vorlage dienen kann. Standardmäßig hat die Konfigurationsdatei den Namen `SecondoConfig.ini`; wird ein anderer Name verwendet, muss er beim Start des Systems explizit spezifiziert werden.

Die Konfigurationsdatei gliedert sich in mehrere Sektionen. Neben einer allgemeinen gibt es für jedes unterstützte Speichersystem eine eigene Sektion. Zusätzlich

[8] Es wird vorausgesetzt, dass auf dem Rechner, auf dem SECONDO installiert wird, entweder eine ORACLE-Client- oder ORACLE-Datenbank-Installation durchgeführt wurde – jeweils einschließlich SQL*NET- und OCI-Unterstützung.

Sektion `Environment`	
Name	**Bedeutung**
`SecondoHome`	Verzeichnis, in dem sich die zu SECONDO gehörigen Dateien befinden
`RegistrarName`	Name des lokalen Sockets des SECONDO-Registrar-Programms
`RegistrarProgram`	Name der ausführbaren Datei des SECONDO-Registrar-Programms
`ListenerProgram`	Name der ausführbaren Datei des SECONDO-Listener-Programms
`SecondoHost`	Host-Adresse des SECONDO-Servers
`SecondoPort`	Port-Nummer des SECONDO-Servers
`RulePolicy`[1]	Basisregel für die IP-Adressprüfung **`ALLOW`** = grundsätzlich zulassen `DENY` = grundsätzlich abweisen
`RuleSetFile`[1]	Name einer Datei mit Regeln für die IP-Adressprüfung

Sektion `BerkeleyDB`	
Name	**Bedeutung**
`ServerProgram`	Name der ausführbaren Datei des SECONDO-Server-Programms
`CheckpointProgram`	Name der ausführbaren Datei des BERKELEY DB-Checkpoint-Programms
`CheckpointTime`[1]	Zeit in Minuten zwischen zwei Checkpoints
`LogDir`[1]	Verzeichnis für Log-Dateien
`CacheSize`[1]	Größe des Speicher-Cache in Kilobyte
`MaxLockers`[1]	Maximalzahl gleichzeitiger Sperreinheiten
`MaxLocks`[1]	Maximalzahl von Sperren
`MaxLockObjects`[1]	Maximalzahl gleichzeitig gesperrter Objekte

Sektion `OracleDB`	
Name	**Bedeutung**
`ServerProgram`	Name der ausführbaren Datei des SECONDO-Server-Programms
`ConnectString`	Verbindungsinformation für die ORACLE-Datenbank-Instanz des SECONDO-Systems (normalerweise ein TNS-Name)
`SecondoUser`	Benutzerkennung für den technischen ORACLE-Benutzer für das SECONDO-System
`SecondoPswd`	Passwort des ORACLE-Benutzers

[1] Optionale Einträge. Fehlt ein Eintrag, werden Standardwerte angenommen.

Tabelle 5.3: Erforderliche Einträge in der SECONDO-Konfigurationsdatei

kann für jeden im SECONDO-System definierten *SmiFile*-Kontext eine weitere Sektion definiert sein. Die erforderlichen Einträge jeder Sektion sind in Tabelle 5.3 auf der vorherigen Seite aufgelistet.

Die Sektion **`Environment`** enthält die Parameter, die unabhängig vom verwendeten Speichersystem sind. Im Eintrag `SecondoHome` ist das Verzeichnis anzugeben, in dem die zum System gehörigen Datendateien, Log-Dateien usw. abgelegt werden sollen. Für den Mehrbenutzerbetrieb sind darüberhinaus die Einträge `SecondoHost` und `SecondoPort` anzupassen. Der Host-Name kann entweder als symbolischer Name (z.B. `robinson.fernuni-hagen.de`) oder als IP-Adresse (z.B. `132.176.69.10`) eingetragen werden. Die Port-Nummer muss ggf. mit dem jeweiligen Systemverwalter abgestimmt werden, um Konflikte mit anderen Server-Diensten zu vermeiden.

Die Einträge `RulePolicy` und `RuleSetFile` legen die Vorgehensweise bei der Überprüfung der IP-Adressen der Clients fest. Der Eintrag `RuleSetFile` verweist auf eine Regeldatei. Ein Beispiel ist in Tabelle 5.4 dargestellt. Wird im Eintrag `RulePolicy` der Wert `ALLOW` angegeben, so sind alle Clients zugelassen, mit Ausnahme von denen, deren IP-Adresse eine der Regeln mit Kennung **`0`** (*Blacklist*) erfüllt. Hat der Eintrag `RulePolicy` dagegen den Wert `DENY`, so werden alle Clients abgewiesen, mit Ausnahme von denen, deren IP-Adresse eine der Regeln mit Kennung **`1`** (*Whitelist*) erfüllt. Eine Client-IP-Adresse wird auf Übereinstimmung mit der IP-Adresse der Regel unter Berücksichtigung der IP-Maske geprüft, indem eine bitweise logische Und-Verknüpfung der Client-IP-Adresse mit der IP-Maske durchgeführt und das Ergebnis anschließend auf Gleichheit mit der IP-Adresse der Regel geprüft wird. Bei Übereinstimmung entscheidet die Kennung über die Zulassung (`1`) oder Abweisung (`0`).

IP-Adresse	IP-Maske	Kennung
`132.176.69.0`	`255.255.255.0`	`1`
`132.176.70.10`	`255.255.255.255`	`0`

Clients, deren IP-Adressen in den ersten drei Stellen mit `132.176.69` übereinstimmen, werden zugelassen; der Client mit der IP-Adresse `132.176.70.10` wird abgewiesen.

Tabelle 5.4: Beispiel einer Regeldatei für IP-Adressprüfungen

Die übrigen Parameter in dieser Sektion müssen in der Regel nicht geändert werden.

In der Sektion **`BerkeleyDB`** werden die Parameter gesetzt, die das auf der BERKELEY DB basierende Speichersystem betreffen. In der Regel können die Parameter auf ihren Standardeinstellungen belassen werden. Zur Bestimmung sinnvoller An-

gaben für die optionalen Parameter sollte die Dokumentation der BERKELEY DB [Sle01] konsultiert werden.

Falls SECONDO unter WINDOWS mit dem auf ORACLE basierenden Speichersystem genutzt werden soll, so sind die Parameter in der Sektion **`OracleDB`** anzupassen. Im Parameter `ConnectString` ist die Verbindungsinformation für ORACLE anzugeben. Hierbei handelt es sich in der Regel um den sogenannten TNS-Namen, unter dem die für SECONDO eingerichtete Datenbankinstanz im Netzwerk angesprochen werden kann. Diese Information wird üblicherweise durch den verantwortlichen Datenbankadministrator bereitgestellt. Für die Herstellung einer Verbindung zur ORACLE-Datenbank müssen zusätzlich der Name des für SECONDO eingerichteten technischen Benutzers sowie das zugeordnete Passwort unter den Parametern `SecondoUser` und `SecondoPswd` eingetragen werden.

Sektion `Default`[1]	
Name	**Bedeutung**
`OraTableSpace`	Angaben zum ORACLE-Tablespace in der `CREATE TABLE`-Anweisung für *SmiFiles*
`OraIndexSpace`	Angaben zum ORACLE-Indexspace in der `CREATE TABLE`-Anweisung für *SmiFiles*

[1] Falls z.B. Algebra-Module eigene Kontext-Namen verwenden, ist statt `Default` der jeweilige Kontext-Name anzugeben.

Tabelle 5.5: Optionale Sektionen in der SECONDO-Konfigurationsdatei

Für jeden definierten *SmiFile*-Kontext kann es in der Konfigurationsdatei eine Sektion geben, die jeweils den Namen des Kontextes trägt. Tabelle 5.5 zeigt die definierten Parameter. Im Kernsystem ist zum einen der Kontext **`Default`**, der immer dann verwendet wird, wenn in den *SmiFile*-Methoden kein Kontext angegeben ist, und zum anderen der Kontext **`SecondoCatalog`**, der für *SmiFiles* des Systemkatalogs verwendet wird, definiert. Algebra-Entwickler können darüberhinaus weitere Kontext-Vereinbarungen einführen.

Zur Zeit sind für die *SmiFile*-Sektionen nur Parameter, die das ORACLE basierte Speichersystem betreffen, verfügbar. Mit Hilfe der Parameter `OraTableSpace` und `OraIndexSpace` kann für einen Kontext festgelegt werden, welche Tablespace- und Indexspace-Eigenschaften für die in diesem Kontext angelegten *SmiFiles* gelten sollen.

5.3.3 Bedienung

Für die Nutzung des SECONDO-Systems steht mit dem Programm `SecondoTTY` eine einfache, kommando-orientierte Bedienoberfläche zur Verfügung. Abhängig davon, ob der Einsatz in einer Einzelbenutzer- oder Mehrbenutzerumgebung erfolgt, und welche Implementierung des Speichersystems genutzt wird, sind unterschiedliche Versionen des Programms zu starten:

- im Einzelbenutzerbetrieb mit Speichersystem auf Basis der BERKELEY DB: `SecondoTTYBDB`
- im Einzelbenutzerbetrieb mit Speichersystem auf Basis von ORACLE: `SecondoTTYORA`
- im Mehrbenutzerbetrieb: `SecondoTTYCS`

Beim Aufruf von `SecondoTTY` müssen entweder über die Kommandozeile oder über Umgebungsvariablen Optionen gesetzt werden, die für die Herstellung der Verbindung zum SECONDO-System nötig sind. Darüber hinaus gibt es Optionen zur Spezifizierung einer Ein- bzw. Ausgabedatei. Eine Übersicht über alle Optionen ist in Tabelle 5.6 dargestellt.

Aufruf: `SecondoTTY{BDB|ORA|CS}`[1] `[Optionen]`

Option	**Bedeutung**	**Umgebungsvariable**[2]
`-c config`	SECONDO-Konfigurationsdatei	`SECONDO_CONFIG`
`-i input`	Name der Eingabedatei (Vorgabe: `stdin`)	
`-o output`	Name der Ausgabedatei (Vorgabe: `stdout`)	
`-u user`	Benutzerkennzeichen	`SECONDO_USER`
`-s pswd`	Passwort	`SECONDO_PSWD`
`-h host`[3]	Host-Adresse des SECONDO-Servers	`SECONDO_HOST`
`-p port`[3]	Portnummer des SECONDO-Servers	`SECONDO_PORT`

[1] BDB – Berkeley DB Single User, ORA – Oracle Single User, CS – Client/Server Multi User

[2] Kommandozeilen-Optionen haben Vorrang vor Umgebungsvariablen.

[3] Host-Adresse und Port-Nummer werden nur im Mehrbenutzerbetrieb benötigt.

Tabelle 5.6: Aufruf von `SecondoTTY`

Einzelbenutzerbetrieb

Außer der Spezifizierung der SECONDO-Konfigurationsdatei sind im Einzelbenutzerbetrieb keine weiteren Optionen für das Programm `SecondoTTY` erforderlich

– zumindest solange in SECONDO keine Benutzerverwaltung implementiert ist. Es muss gewährleistet werden, dass auf die angesprochene SECONDO-Datenbank keine anderen Prozesse zeitgleich zugreifen können.

Mehrbenutzerbetrieb

Bevor SECONDO im Mehrbenutzerbetrieb genutzt werden kann, ist der SECONDO-Server zu starten. Für den Betrieb des Servers werden mehrere Programme benötigt: `SecondoMonitor`, `SecondoRegistrar`, `SecondoListener`, `SecondoServer` und ggf. `SecondoCheckpoint`. Einzelheiten zu den Aufgaben dieser Programme können im Abschnitt 3.1 auf Seite 28 nachgelesen werden.

Zur Verwaltung des SECONDO-Systems wird je nach verwendeter Implementierung des Speichersystems zunächst das Programm `SecondoMonitorBDB` (Speichersystem auf Basis der BERKELEY DB) oder `SecondoMonitorORA` (Speichersystem auf Basis von ORACLE) aufgerufen, wobei die SECONDO-Konfigurationsdatei entweder als erster und einziger Kommandozeilenparameter oder über die Umgebungsvariable `SECONDO_CONFIG` anzugeben ist (siehe Tabelle 5.7).

Aufruf: `SecondoMonitor{BDB|ORA}`[1] `[Optionen]`

Option	**Bedeutung**	**Umgebungsvariable**[2]
`config`	SECONDO-Konfigurationsdatei	`SECONDO_CONFIG`

[1] Speichersytem basiert auf: BDB – BERKELEY DB, ORA – ORACLE

[2] Kommandozeilen-Optionen haben Vorrang vor Umgebungsvariablen.

Tabelle 5.7: Aufruf von `SecondoMonitor`

Falls keine Angabe der Konfigurationsdatei erfolgte, wird zu guter Letzt im aktuellen Arbeitsverzeichnis nachgesehen, ob dort die Datei `SecondoConfig.ini` vorhanden ist. Falls keine Konfigurationsdatei gefunden wurde, beendet sich das Monitor-Programm mit einer entsprechenden Fehlermeldung. Andernfalls werden die Konfigurationsparameter auf Gültigkeit untersucht. War die Überprüfung erfolgreich, so wird automatisch das Programm `SecondoRegistrar` zur Unterstützung von Verwaltungsaufgaben und – falls das Speichersystem auf der BERKELEY DB basiert – das Programm `SecondoCheckpoint` gestartet.

Mit Hilfe der in Tabelle 5.8 auf der nächsten Seite aufgeführten Kommandos wird SECONDO-System verwaltet und überwacht. Insbesondere aktiviert das Kommando `STARTUP` das Programm `SecondoListener`, das die Verbindungswünsche der Clients entgegennimmt und für jeden Client einen SECONDO-Server startet. Wenn das SECONDO-System heruntergefahren werden soll, wird zunächst der Listener

mit Hilfe des Kommandos SHUTDOWN deaktiviert. Anschließend können die übrigen Systemkomponenten durch Eingabe des Kommandos QUIT beendet werden.

Kommando	Bedeutung
?, HELP	Informationen zu den Kommandos anzeigen
STARTUP	SECONDO-Listener starten
SHUTDOWN	SECONDO-Listener stoppen
SHOW LOG	Neue Log-Einträge anzeigen
SHOW USERS	Aktuell angemeldete Benutzer anzeigen
SHOW DATABASES	Aktuell benutzte SECONDO-Datenbanken anzeigen
SHOW LOCKS	Aktuelle Datenbanksperren anzeigen
QUIT	SECONDO-Listener – falls erforderlich – stoppen und SECONDO-Monitor beenden

Tabelle 5.8: Kommandos das SECONDO-Monitors

Kommandoverarbeitung

Sobald die Bedienoberfläche SecondoTTY erfolgreich initialisiert und die Verbindung zu SECONDO bzw. zum SECONDO-Server hergestellt wurde, meldet sich das Programm mit dem Eingabe-Prompt „Secondo =>“ und wartet auf die Eingabe von Kommandos.

Kommando	Bedeutung
?, HELP	Informationen zu den Kommandos anzeigen
{FILE}	Kommandos aus Datei 'FILE' lesen (verschachtelt möglich)
D, DESCRIPTIVE	Level auf 'DESCRIPTIVE' setzen
E, EXECUTABLE	Level auf 'EXECUTABLE' setzen
H, HYBRID	Level auf 'HYBRID' setzen (Kommandos werden zuerst auf Level 'DESCRIPTIVE' und anschließend auf Level 'EXECUTABLE' ausgeführt.)
SHOW LEVEL	Aktuellen Level anzeigen
DEBUG 0[1]	Debug-Modus ausschalten
DEBUG 1[1]	Debug-Modus einschalten
DEBUG 2[1]	Debug-Modus und Trace-Modus einschalten
Q, QUIT	Programm beenden
# ...	Kommentar eingeben (Das erste Zeichen in einer Zeile muss # sein.)

[1] Der Debug-Modus kann nur in der Einzelbenutzerversion aktiviert werden.

Tabelle 5.9: Interne Kommandos von SecondoTTY

In Tabelle 5.9 auf der vorherigen Seite sind die internen Kommandos aufgeführt, die ohne Interaktion mit SECONDO ausgeführt werden.

Außerdem können alle gültigen SECONDO-Kommandos eingegeben werden, die in Tabelle 5.10 aufgeführt sind. Nähere Einzelheiten zu diesen Kommandos können der Programmdokumentation in Anhang G ab Seite 211 entnommen werden.

Kommando	**Bedeutung**
`create database <identifier>`	Datenbank anlegen
`delete database <identifier>`	Datenbank löschen
`open database <identifier>`	Datenbank öffnen
`close database`	Datenbank schließen
`save database to <filename>`	Datenbank speichern
`restore database <identifier> from <filename>`	Datenbank wiederherstellen
`type <identifier> = <type expression>`	Neuen Typ definieren
`delete type <identifier>`	Typ löschen
`create <identifier> : <type expression>`	Objekt anlegen
`update <identifier> := <value expression>`	Objektwert aktualisieren
`delete <identifier>`	Objekt löschen
`query <value expression>`	Abfrage ausführen
`list databases`	Liste der Datenbanken anzeigen
`list type constructors`	Liste der Typkonstruktoren anzeigen
`list operators`	Liste der Operatoren anzeigen
`list types`	Liste der Typen anzeigen
`list objects`	Liste der Objekte anzeigen
`begin transaction`	Transaktion beginnen
`commit transaction`	Transaktion beenden
`abort transaction`	Transaktion abbrechen

Tabelle 5.10: Liste der `Secondo`-Kommandos

Bei der Kommandoeingabe ist zu beachten, dass interne Kommandos auf genau eine Zeile begrenzt sind, während SECONDO-Kommandos sich über mehrere Zeilen erstrecken können. Für Fortsetzungszeilen wird als Eingabe-Prompt „`Secondo ->`“ angezeigt. Wird als letztes Zeichen einer Zeile ein Semikolon eingegeben, beendet es das Kommando, ist aber selbst nicht Bestandteil des Kommandos. Alternativ kann zur Beendigung eines Kommandos eine Leerzeile eingegeben werden.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Erreichung der Ziele

Durch eine einheitliche Programmierung in C++ konnten die Voraussetzungen für eine hohe Portabilität des SECONDO-Systems geschaffen werden. Systemspezifische Besonderheiten wurden in geeignete Klassen gekapselt, so dass SECONDO nunmehr unter LINUX, SOLARIS und WINDOWS lauffähig ist. Unter anderem entstanden dabei betriebssystemunabhängige Klassen für die Prozess- und Netzwerkkommunikation sowie die Applikationssteuerung.

Für die persistente Datenspeicherung wurde eine einfach zu nutzende Speichersystemschnittstelle definiert und exemplarisch für Speichersysteme auf der Basis der BERKELEY DB und des ORACLE-Datenbanksystems implementiert. Darauf aufbauend wurden sowohl eine Einzelbenutzer- als auch eine Client/Server-Version von SECONDO entwickelt. In beiden Versionen können SECONDO-Kommandos in Transaktionen zusammengefasst werden.

Das SECONDO-Interface wurde derart erweitert, dass die Schnittstelle gleichermaßen für den Single-User- wie den Mehrbenutzerbetrieb geeignet ist. Weiterhin wurde der SECONDO-Systemkatalog fast vollständig neu entwickelt und stützt sich auf die mehrbenutzerfähige Speichersystemschnittstelle. Schließlich wurde die Algebra-Verwaltung grundlegend überarbeitet, so dass die Algebren nunmehr in objektorientierter Weise ins System eingebunden werden.

Die Möglichkeiten für eine Portierung von SECONDO auf PALMOS wurden geprüft. Die eingesetzten Werkzeuge aus der GNU Compiler Collection sind zwar auch für PALMOS verfügbar, jedoch stellt die Portierung des Speichersystems eine hohe Hürde dar. Wahrscheinlich müsste eine Implementierung der Speichersystemschnittstelle von Grund auf neu entwickelt werden, da eine Portierung der BERKELEY DB mit unkalkulierbar hohem Aufwand verbunden wäre.

Über die ursprüngliche Aufgabenstellung hinaus wurde zum einen neben der Unterstützung ausführbarer Algebren auch die für deskriptive sowie die Behandlung von Abstraktionen und Funktionsobjekten in das SECONDO-System integriert. Zum anderen wurden die Standard-Algebra sowie die Function-Algebra auf die neue Architektur umgestellt und die Bedienoberfläche `SecondoTTY` vollständig neu implementiert, um die Systemfunktionalität austesten zu können.

6.2 Zukünftige Erweiterungen

Damit die neue SECONDO-Version die Grundfunktionalität eines relationalen Datenbanksystems bietet, ist die Portierung des Tupelmanagers sowie der Relation-Algebra vordringlich. Die Umstellung weiterer vorhandener Algebra-Module auf die neuen Schnittstellen kann je nach Bedarf erfolgen.

Im Hinblick auf eine bessere Transaktionsunterstützung ist zu überlegen, ob eine zusätzliche Implementierung der Speichersystemschnittstelle auf der Basis von MYSQL durchgeführt wird, um die herausragenden Eigenschaften der INNODB im Mehrbenutzerbetrieb für SECONDO auszunutzen.

Für Anwendungen im mobilen Einsatz könnte eine ausschließlich hauptspeicherbasierte Einzelbenutzer-Version nützlich sein. Eventuell ließe sich eine solche Version auf Basis der BERKELEY DB realisieren, denn beim Anlegen von *SmiFiles* werden nur Hauptspeicherbereiche genutzt, falls auf die Übergabe von Dateinamen verzichtet wird.

Schließlich könnte die Programmierung im Bereich der persistenten Speicherung weiter vereinfacht werden, wenn die Speichersystemschnittstelle um C++-Streamkonforme Ein/Ausgabe-Methoden auf *SmiRecords* erweitert würde.

Im Bereich der Netzwerkkommunikation wird in der Zukunft wahrscheinlich eine Erweiterung der *Socket*-Klasse um die Unterstützung von IPv6[1] wünschenswert sein. Je nach Einsatzgebiet könnte auch Bedarf für die Unterstützung sicherer Kommunikation (SSL[2] o.ä.) entstehen.

Die Erweiterung von SECONDO um eine Benutzerverwaltung wäre wahrscheinlich insbesondere für den Mehrbenutzerbetrieb wünschenswert. Über das SECONDO-Interface kann bereits jetzt eine Benutzerkennung und ein Passwort mitgegeben werden, auch wenn derzeit noch keine Authentifizierung implementiert ist. Für die Verwaltung der Zugriffsrechte der Benutzer innerhalb des Systems sollte zunächst

[1] Internet Protocol Version 6

[2] Secure Socket Layer

ein Rechtekonzept erarbeitet werden.

In einer der nächsten SECONDO-Versionen sollte die derzeitige Netzwerkkommunikationsprotokoll-Implementierung auf die Verwendung von XML-Komponenten wie z.B. ein XML-Parser umgestellt werden. Mit dem Einsatz von XML-Technologien könnte die Einhaltung der Protokollvereinbarungen leichter und weniger fehleranfällig umgesetzt werden. Dies spielt insbesondere dann eine Rolle, wenn Client-Implementierungen in anderen Sprachen als C++ erfolgen, da in diesem Fall in der Regel die client-seitigen Protokollanteile neu programmiert werden müssen. Außerdem wäre zu erwägen, das Protokoll in Richtung SOAP[3] zu erweitern, um SECONDO unter Umständen zukünftig auch als Web-Service anbieten zu können.

[3]SOAP = Simple Object Access Protocol

Anhang A

Literaturverzeichnis

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, Boston, San Francisco, New York, 1997.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference*, pages 383–394, Minneapolis, Mai 1994.

[DG99] Stefan Dieker and Ralf-Hartmut Güting. Plug and play with query algebras; SECONDO a generic DBMS development environment. Informatik-Berichte 249, Fernuniversität Hagen, Februar 1999.

[Die97] Stefan Dieker. *Tutorial: Secondo Algebra Implementation*. FernUniversität Hagen, 1997.

[GFB+97] R.H. Güting, C. Freundorfer, L. Becker, S. Dieker, and H. Schenk. Secondo/QP: Implementation of a generic query processor. Informatik-Report 215, Fernuniversität Hagen, Februar 1997.

[Güt93] Ralf Hartmut Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. In *SIGMOD Conference*, pages 277–286, 1993.

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, Montreal, Quebec, Kanada, Oktober 1980.

[LR99] (Primary Authors) L. Leverenz and D. Rehfield. *Oracle8i Concepts*. Oracle Corporation, Redwood Shores, CA, USA, 1999.

[OBS99] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, California, Juni 1999.

[O'S97] Bryan O'Sullivan. Answers to frequently asked questions for comp.programming.threads. `http://www.serpentine.com/~bos/threads-faq/`, September 1997.

[Sle01] Sleepycat Software, Inc. *Berkeley DB*. New Riders Publishing, Indianapolis, 2001.

[Ste93] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, Boston, San Francisco, New York, 21. edition, 1993.

[The95] The SHORE Team. SHORE: Combining the best features of OODBMS and file systems. In *Proceedings of the ACM SIGMOD International Conference*, page 486, San Jose, 1995.

[Wal97] Sean Walton. Linux threads frequently asked questions. `http://www.ibiblio.org /pub/Linux/docs/faqs/Threads-FAQ/html/`, Januar 1997.

Anhang B

Kommentiertes Literaturverzeichnis

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, Boston, San Francisco, New York, 1997.

> Das Buch enthält eine ausführliche Beschreibung der Thread-Schnittstelle des Posix-Standards.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference*, pages 383–394, Minneapolis, Mai 1994.

[DG99] Stefan Dieker and Ralf-Hartmut Güting. Plug and play with query algebras; SECONDO a generic DBMS development environment. Informatik-Berichte 249, Fernuniversität Hagen, Februar 1999.

> Dieser Bericht stellt das SECONDO-System als äußerst flexible erweiterbare Umgebung für die Implementierung von Datenbanksystemen vor. Zunächst wird das Konzept der *second-order signature* als Basis des Systems vorgestellt. Anschließend wird die Architektur sowie die Arbeitsweise des Systems beschrieben.

[Die97] Stefan Dieker. *Tutorial: Secondo Algebra Implementation*. FernUniversität Hagen, 1997.

> Dieses Tutorial beschreibt die Vorgehensweise bei der Erstellung eines Algebra-Moduls für Secondo. Die Schnittstellenfunktionen zum Query-Prozessor werden detailliert beschrieben.

[GFB+97] R.H. Güting, C. Freundorfer, L. Becker, S. Dieker, and H. Schenk. Secondo/QP: Implementation of a generic query processor. Informatik-Report 215, Fernuniversität Hagen, Februar 1997.

In diesem Artikel werden Konzepte zur Implementierung eines allgemeinen Query Prozessors als Teil eines erweiterbaren Datenbanksystems vorgestellt. Nach einer kurzen Rekapitulation von Signaturen zweiter Ordnung als Spezifikationsformalismus wird die Schnittstelle zwischen dem Query Prozessor und den Algebra-Modulen beschrieben. In einer Algebra werden Typkonstruktoren und Operatoren realisiert. Nach einer Auflistung der Struktur einer Algebra wird die Implementierung der Operatorauswertungsfunktionen in Bezug auf die verschiedenen Arten von Operatoren genauer untersucht. Für die Auswertung von Abfrageplänen und Algebraausdrücken spielt die Typzuordnung eine wichtige Rolle. Wie die Typüberprüfung und Typzuordnung bei der Konstruktion des Operatorbaums implementiert werden kann, wird in allen Einzelheiten vorgeführt. In diesem Artikel nicht berücksichtigt sind Modellabbildungs- und Kostenfunktionen, die zur Unterstützung von Abfrageoptimierung benötigt werden.

[Güt93] Ralf Hartmut Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. In *SIGMOD Conference*, pages 277–286, 1993.

Mit *Signaturen zweiter Ordnung* wird ein System aus zwei gekoppelten mehrsortigen Signaturen vorgestellt, das sich in besonderer Weise zur Spezifikation erweiterbarer Datenbanksysteme eignet. Die erste Signatur definiert dabei ein Typsystem, während die zweite Signatur die Ausdrücke der ersten als Sorten für die Definition von Operatoren der Abfrage- oder Ausführungssprache verwendet. Zunächst wird der Systemrahmen informell vorgestellt. Es wird gezeigt, wie mit dem Konzept der Signatur zunächst ein Typsystem und darauf aufbauend die Operatoren definiert werden. Um zu einer lesbaren Abfragesprache zu gelangen, wird eine erweiterte Syntax eingeführt. Der Vorteil besteht darin, daß sowohl auf der Modellebene als auch auf der Darstellungsebene die gleiche Sprache verwendet werden kann. Nach der informellen Vorstellung wird dann das Konzept streng formal entwickelt. Auf der Darstellungsebene werden Beispiele von Spezifikationen detailliert betrachtet. Schließlich wird gezeigt, wie sich Optimierungsregeln und Updatefunktionen in den Systemrahmen integrieren lassen.

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, Montreal, Quebec, Kanada, Oktober 1980.

In diesem Text wird die Methode des erweiterten linearen Hashings dargestellt, die Grundlage der Hash-Zugriffsmethode der Berkeley-DB ist.

[LR99] (Primary Authors) L. Leverenz and D. Rehfield. *Oracle8i Concepts*. Oracle Corporation, Redwood Shores, CA, USA, 1999.

In diesem Handbuch werden die Konzepte, nach denen die Architektur des Oracle-Datenbankmanagementsystem aufgebaut ist, beschrieben. Für die vorliegende Diplomarbeit waren insbesondere die Teile von besonderem Interesse, die sich mit der Client/Server-Architektur und den objekt-orientierten Erweiterungen (speziell die Definition von Vergleichs- oder Mapping-Funktionen für selbstdefinierte Datentypen) befassen.

[OBS99] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, California, Juni 1999.

Dieser Artikel beschreibt die Historie und die wichtigsten Eigenschaften der Berkeley-DB.

[O'S97] Bryan O'Sullivan. Answers to frequently asked questions for comp.programming.threads. `http://www.serpentine.com/~bos/threads-faq/`, September 1997.

[Sle01] Sleepycat Software, Inc. *Berkeley DB*. New Riders Publishing, Indianapolis, 2001.

In diesem Buch die Konzepte und die Architektur der Berkeley-DB sowie die Schnittstellen zu C, C++, Java und Tcl beschrieben. Im wesentlichen stimmt der Text mit der Online-Dokumentation überein.

[Ste93] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Professional Computing Series. Addison-Wesley, Boston, San Francisco, New York, 21. edition, 1993.

In diesem Buch werden fortgeschrittene Techniken der UNIX-Programmierung aus nahezu allen Bereichen - von der Ein/Ausgabe bis zur Prozesssteuerung - ausführlich behandelt. Für diese Arbeit waren die Kapitel zur Prozesssteuerung und -kommunikation von besonderer Bedeutung.

[The95] The SHORE Team. SHORE: Combining the best features of OODBMS and file systems. In *Proceedings of the ACM SIGMOD International Conference*, page 486, San Jose, 1995.

[Wal97] Sean Walton. Linux threads frequently asked questions. `http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/`, Januar 1997.