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.

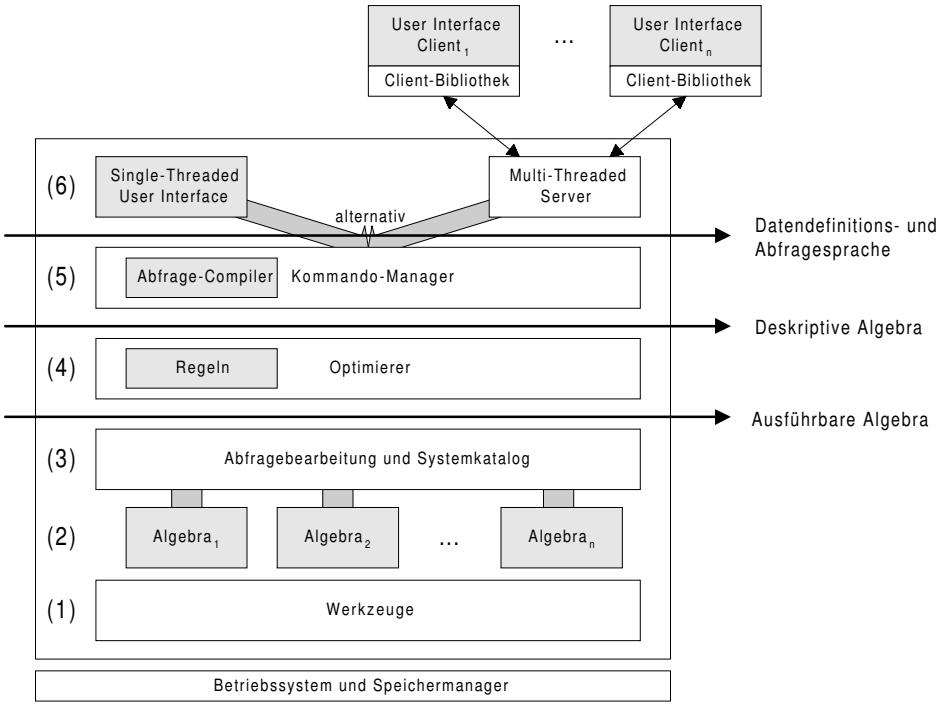

Abbildung 1.1: SECONDO-Architektur

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].

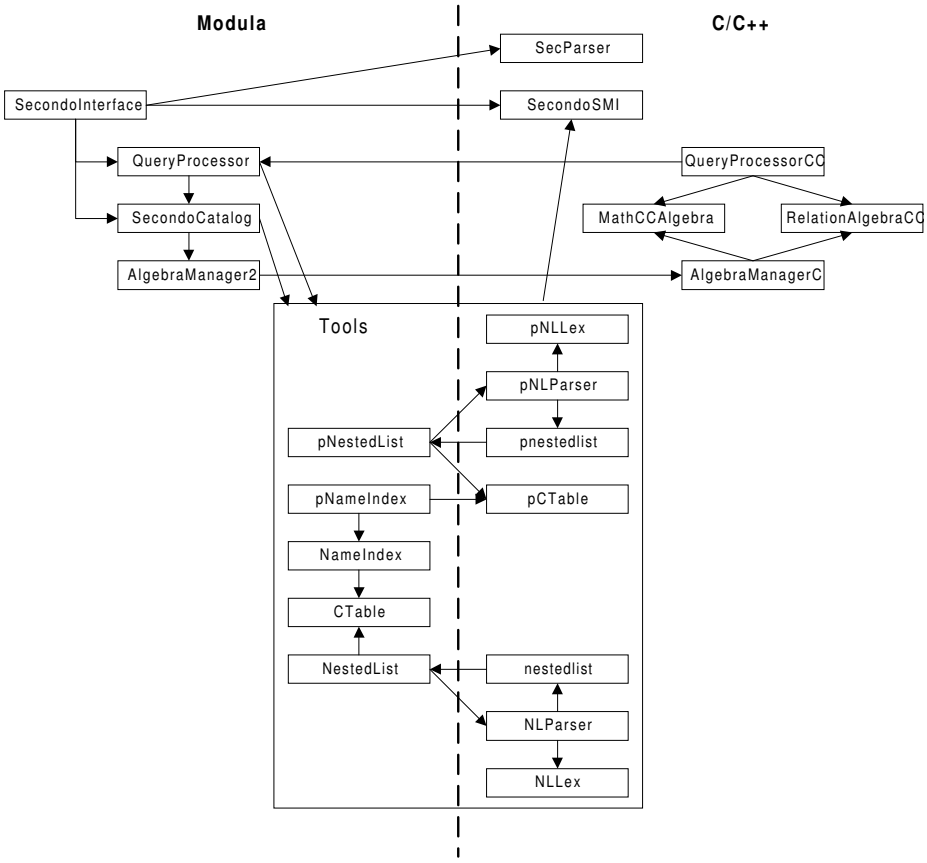

Abbildung 1.2: Modul- und Sprachabhängigkeiten in SECONDO

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.

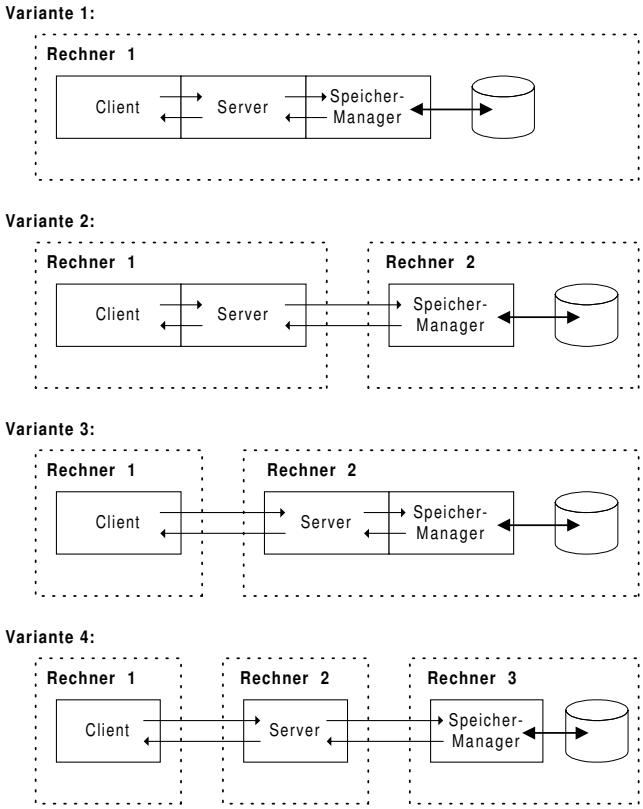

Abbildung 2.1: Verteilung der Komponenten im Netzwerk

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.

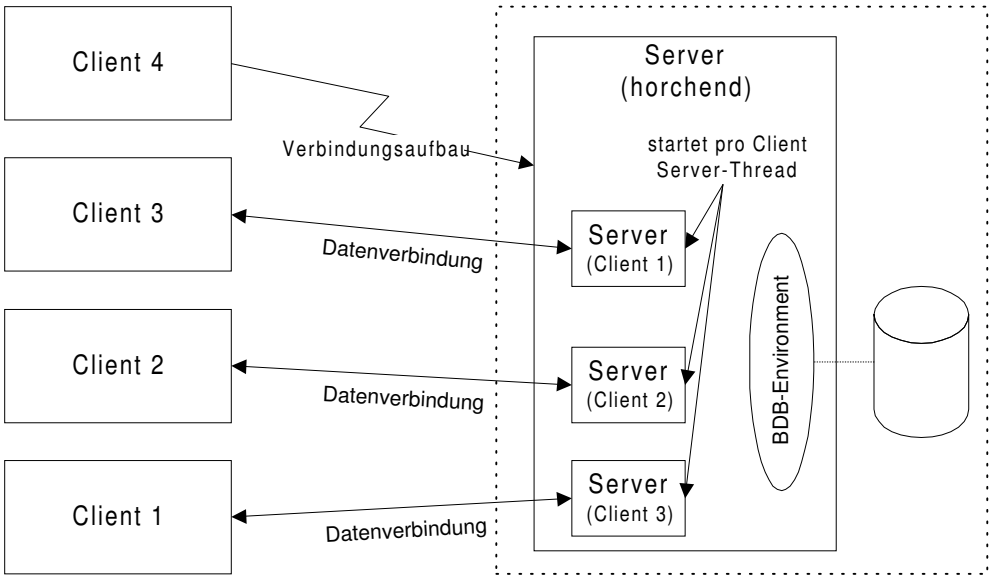

Abbildung 2.2: Paralleler Server, separater Thread für jeden Client

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

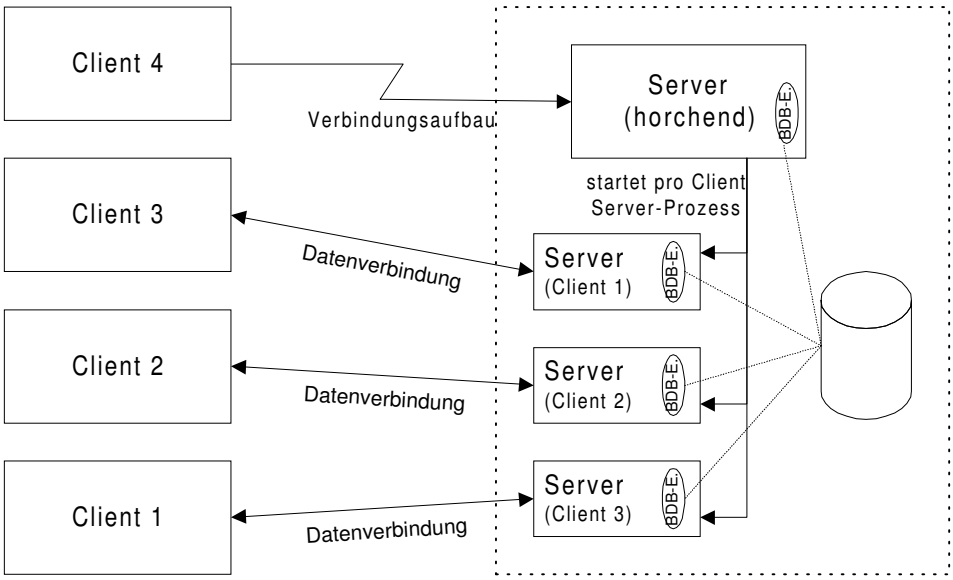

Abbildung 2.3: Paralleler Server, separater Prozess für jeden Client

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.

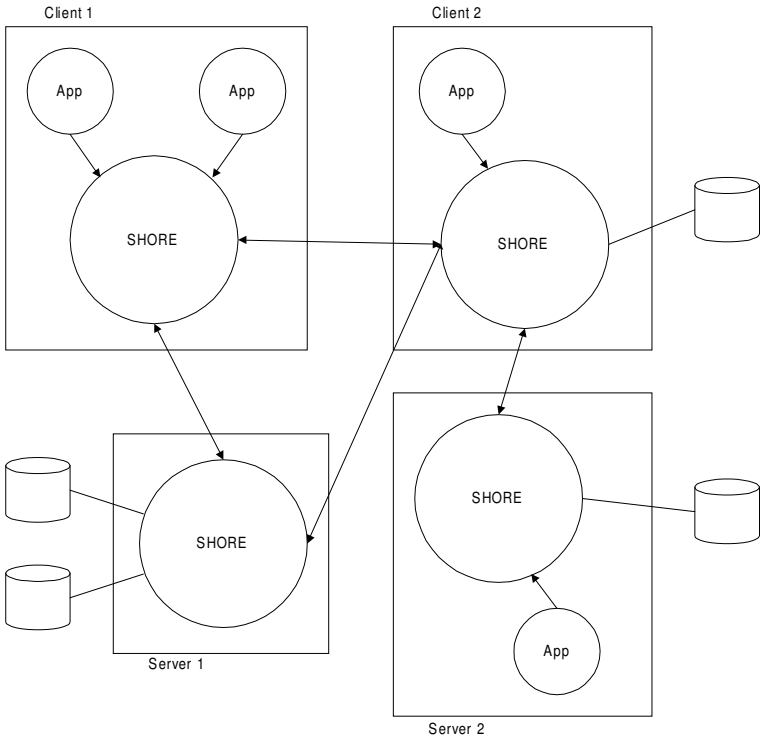

Abbildung 2.4: Prozess-Architektur von SHORE

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.

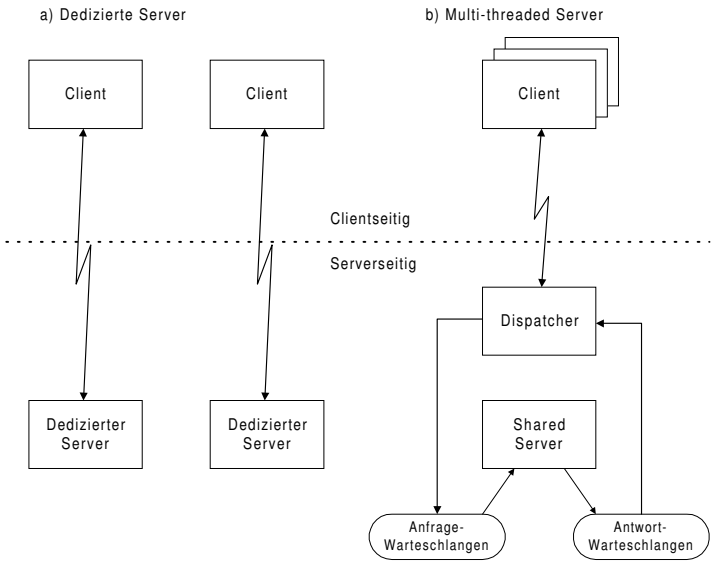

Abbildung 2.5: ORACLE-Architektur: Konfigurationsvarianten

[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.

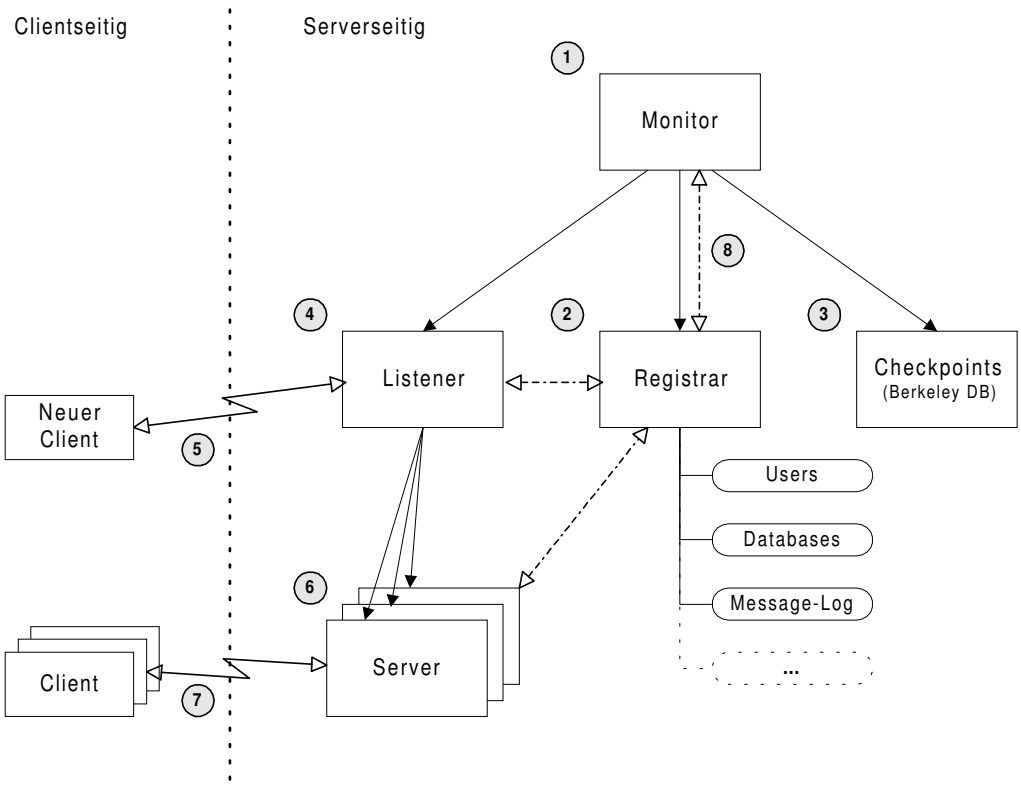

Abbildung 3.1: Client/Server-Architektur

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.

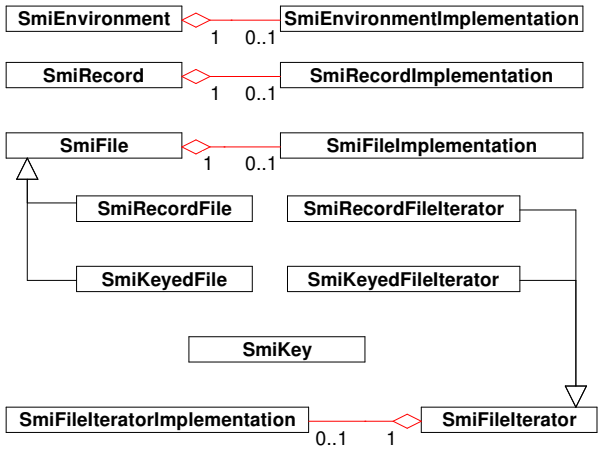

Abbildung 4.1: Klassendiagramm für das Speichersystem

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.

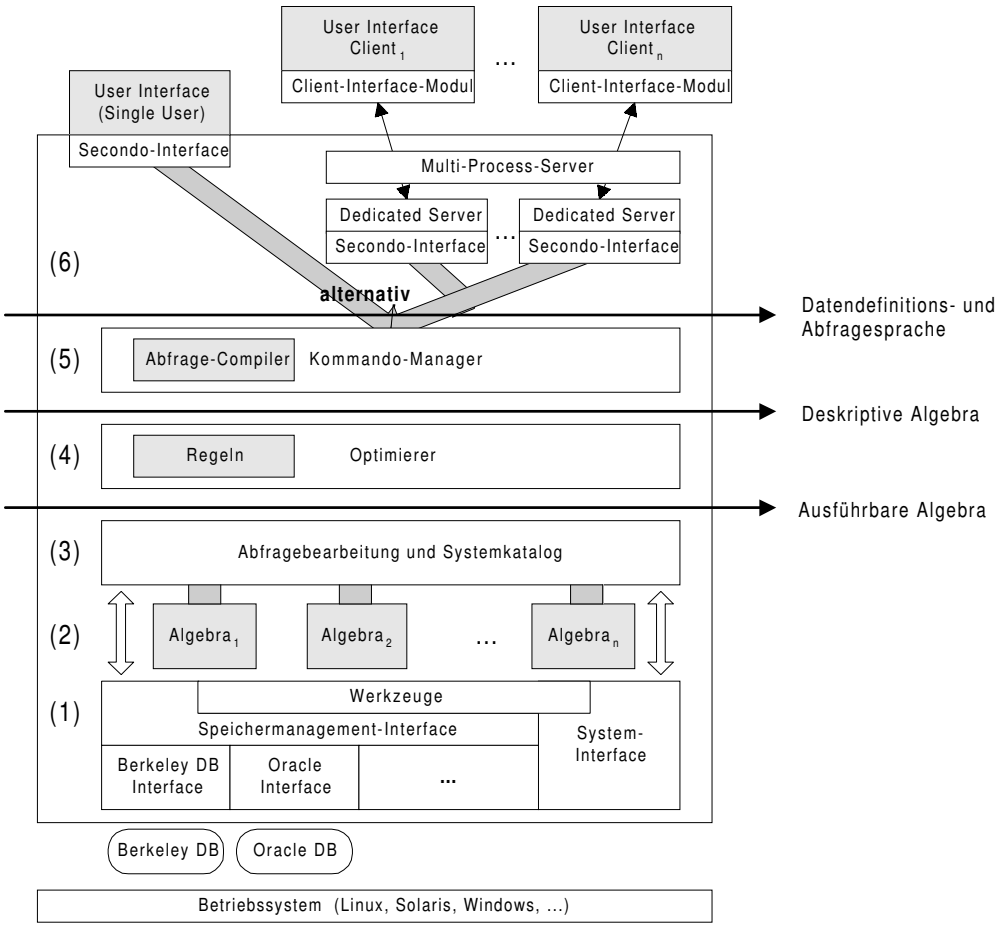

Abbildung 5.1: Neue SECONDO-Architektur

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-

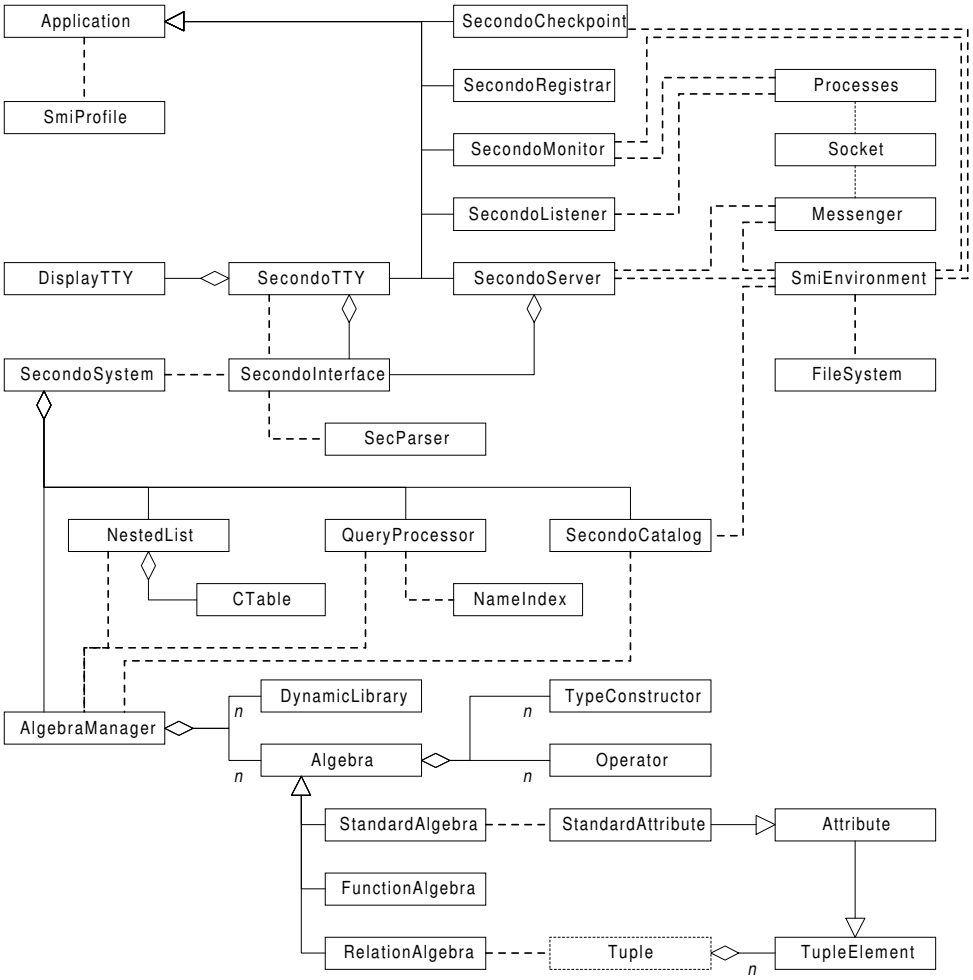

Abbildung 5.2: Klassendiagramm

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.

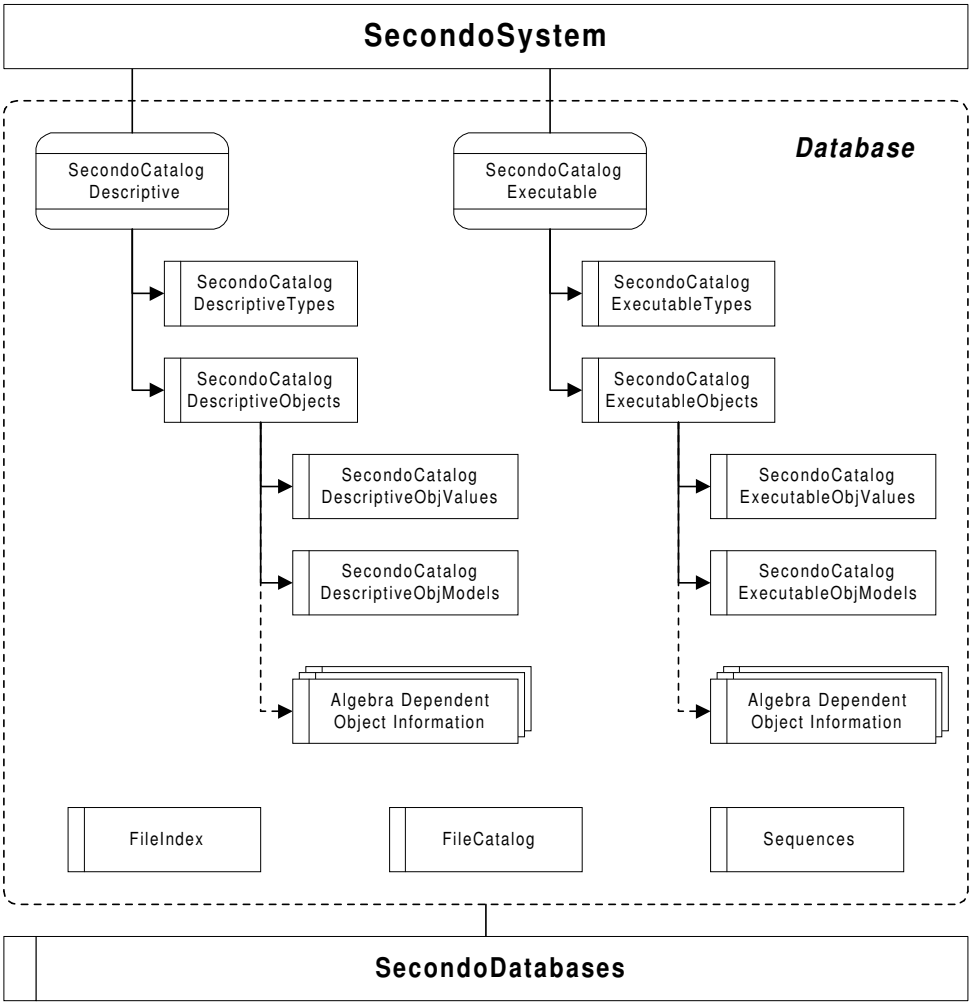

Abbildung 5.3: Aufbau einer SECONDO-Datenbank

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.

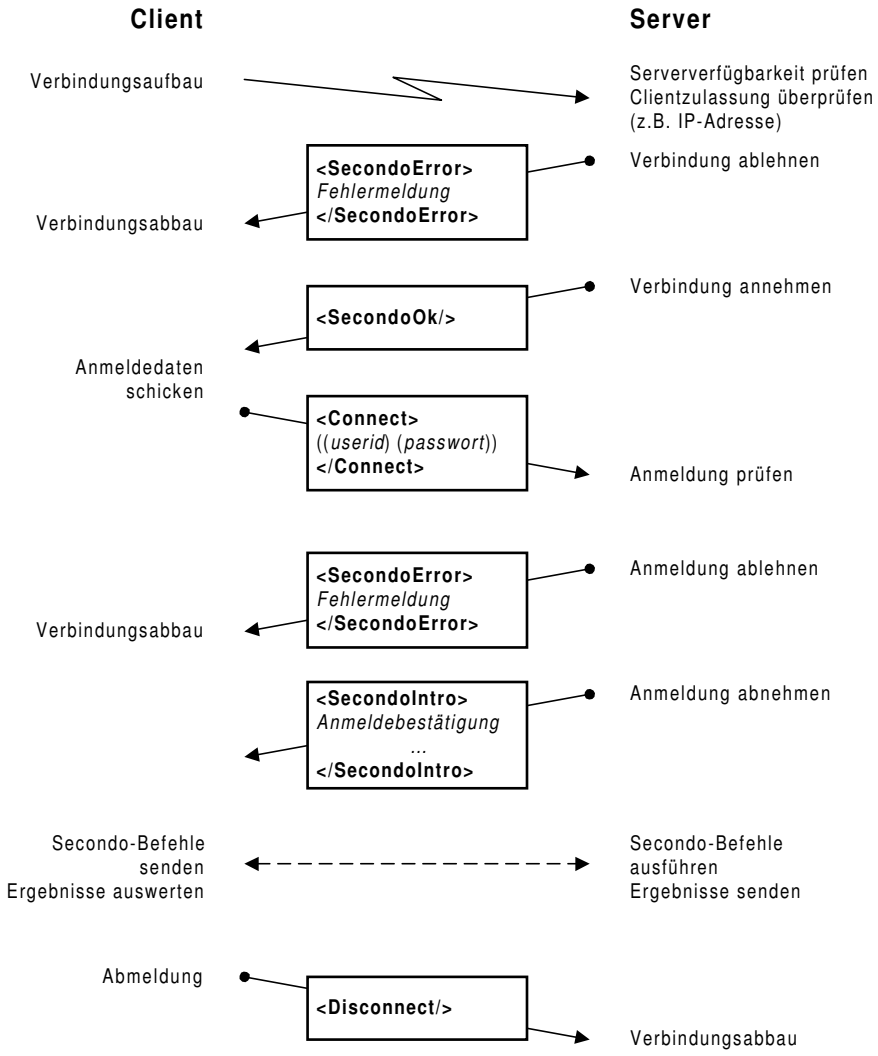

Abbildung 5.4: Verbindungsauf- und Abbau

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.

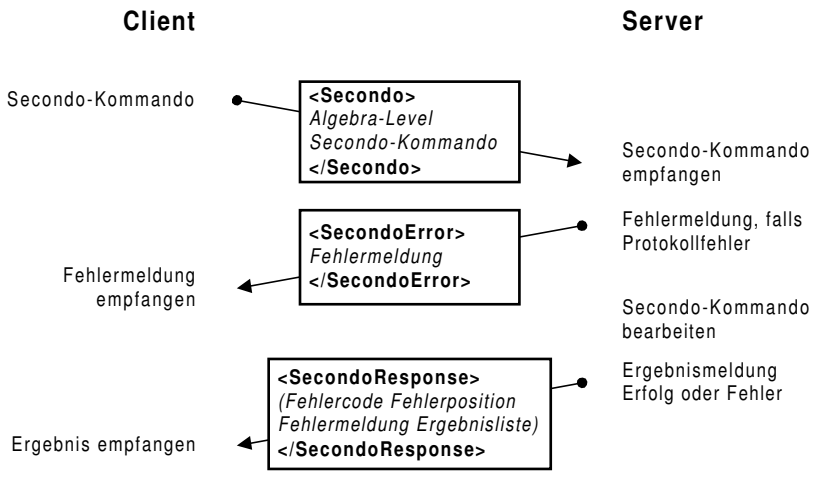

Abbildung 5.5: Übermittlung von SECONDO-Kommandos

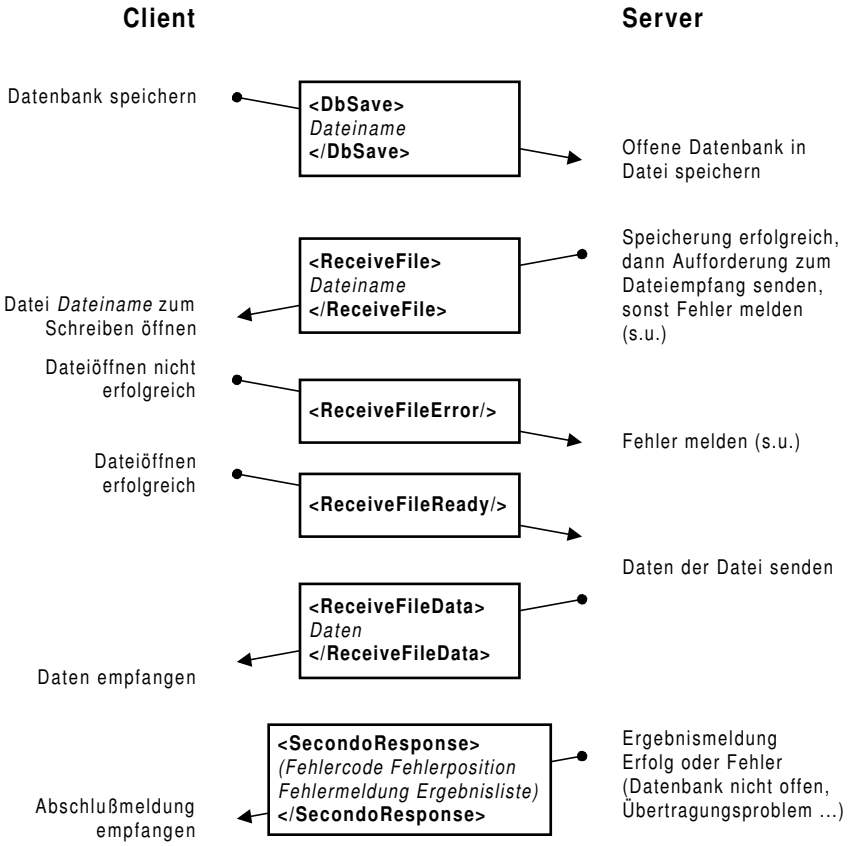

Abbildung 5.6: Datenbank sichern

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,

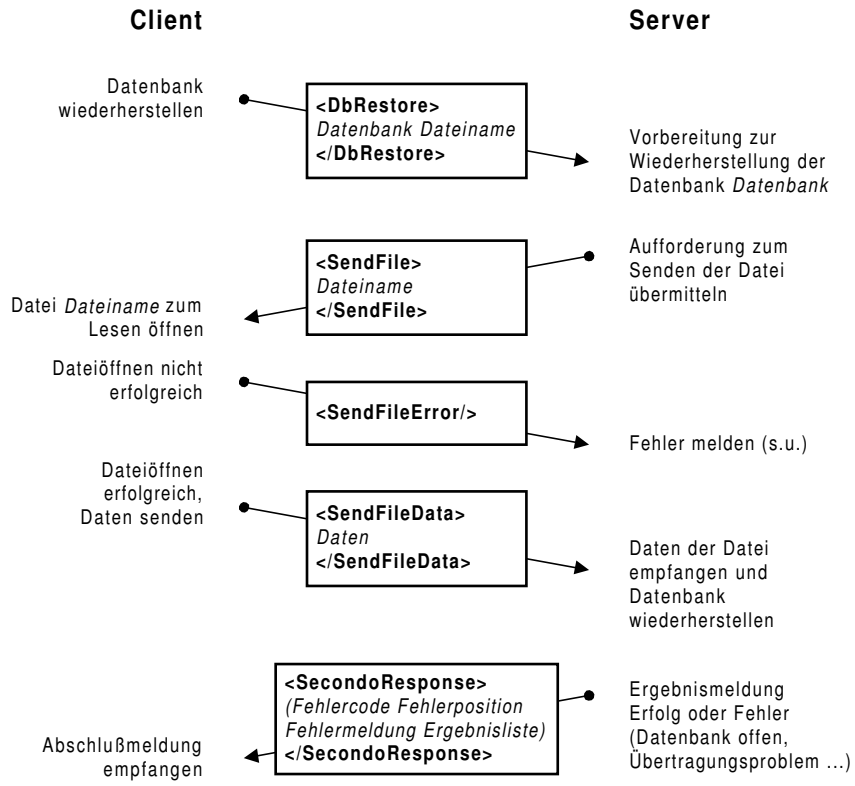

Abbildung 5.7: Datenbank wiederherstellen

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`).

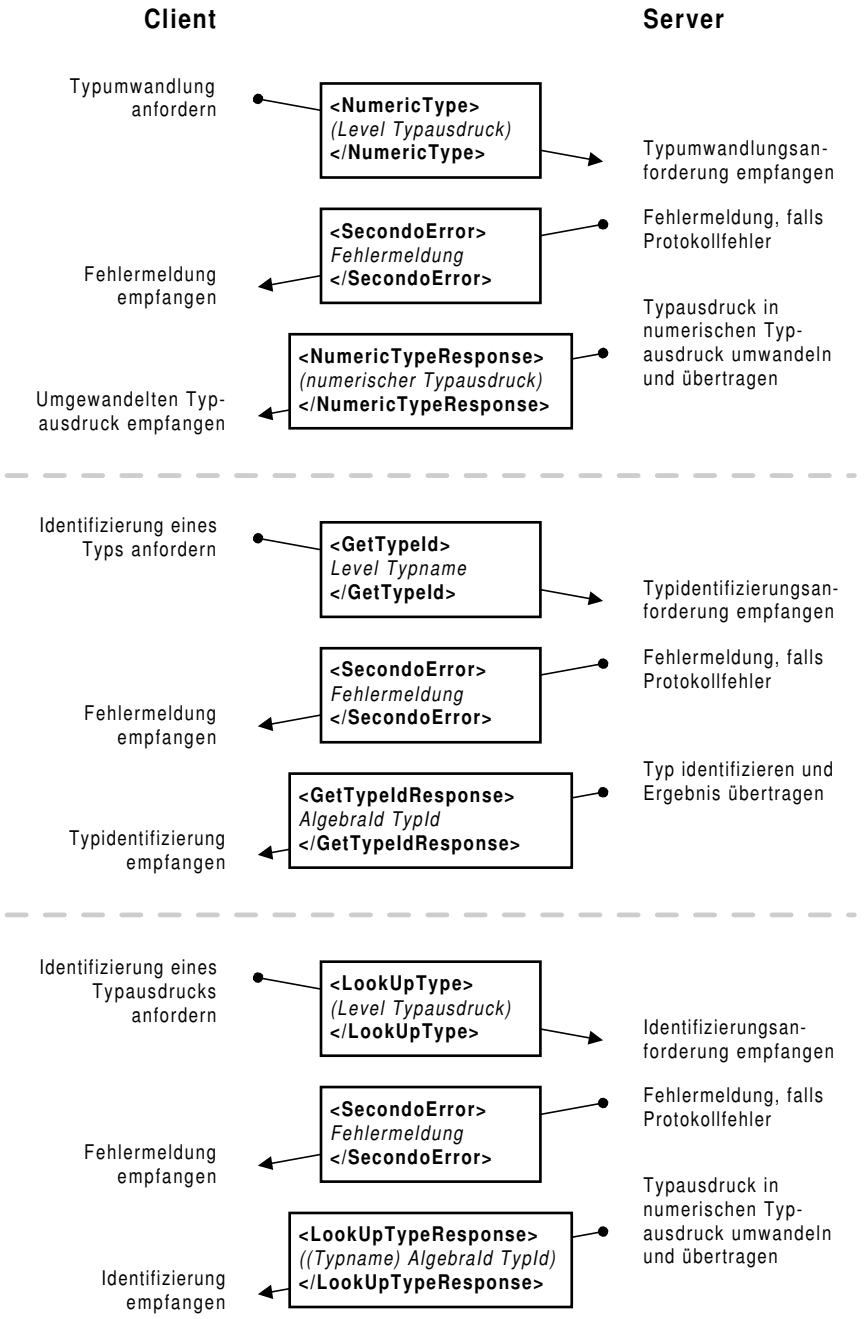

Abbildung 5.8: Spezielle Anfragen zur Identifizierung von Typen

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.

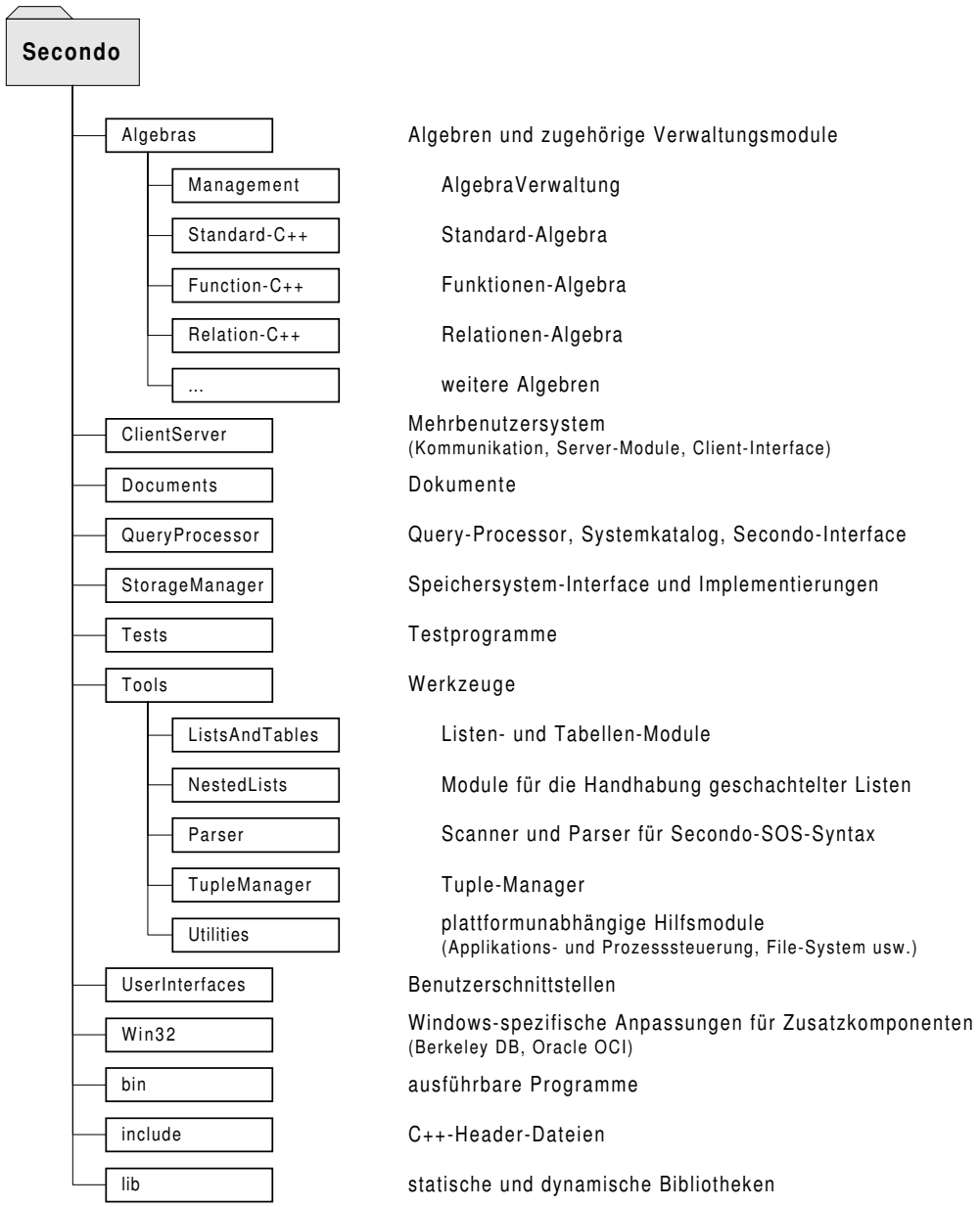

Abbildung 5.9: Verzeichnisstruktur von SECONDO

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.

Anhang C

Programmdokumentation: SECONDO System

In diesem Anhang findet sich die Dokumentation der wichtigsten SECONDO-Kern-Module (Query-Prozessor, Systemkatalog, System-Interface, Parser für SOS-Syntax) und ihrer Schnittstellen.

Die Dokumentation wurde mit Hilfe des PD-Systems aus den Quelltexten gewonnen. Die Kommentierung der Quelltexte erfolgte durchgehend in Englisch.

C.1 Header File: Secondo Configuration

January 2002 Ulrich Telle

C.1.1 Overview

The compilation of SECONDO has to take into account differences in the architecture of the underlying operating system. Usually the compiler defines symbols which can be used to identify the operating system SECONDO is built for.

Depending on the actual operating system several constants and system dependent macros are defined.

C.1.2 Imports, Types

```
#ifndef SECONDO_CONFIG_H
#define SECONDO_CONFIG_H

#define SECONDO_VERSION            "2.0.1"
#define SECONDO_VERSION_MAJOR      2
#define SECONDO_VERSION_MINOR      0
#define SECONDO_VERSION_REVISION   1

#define SECONDO_LITTLE_ENDIAN
```

Define the preprocessor symbol SECONDO_LITTLE_ENDIAN if your machine has a little endian byte order architecture. Otherwise *#undef* this symbol.

TODO: Detection of endianess should be done automatically.

C.1.3 Detect the platform

Windows

```
#if (defined(_WIN32) || defined(_WIN64) || defined(__WIN32__) \
  || defined(__MINGW32__)) && !defined(__CYGWIN__)
#  define SECONDO_WIN32
   // Define Windows version for WINVER and _WIN32_WINNT
   // 0x0400 = Windows NT 4.0
   // 0x0500 = Windows 2000 (NT based)
#  define WINVER       0x0400
#  define _WIN32_WINNT 0x0400
#  define WIN32_LEAN_AND_MEAN
#  include <windows.h>
```

When creating or using shared libraries (i.e. DLLs = Dynamic Link Libraries) on Windows platforms it is necessary to specify for C++ classes whether they are imported from a DLL (when using a DLL) or exported from a DLL (when creating a DLL).

Since the SECONDO system has separate components, namely the database kernel and the storage management interface one must be able to differentiate between these components.

Currently there are two components:

- SDB – the SECONDO DataBase kernel. Define SDB_USE_DLL if compiling modules which want to use a DLL for this component. Define SDB_CREATE_DLL if compiling this component as a DLL.
- SMI – the Storage Management Interface. Define SMI_USE_DLL if compiling modules which want to use a DLL for this component. Define SMI_CREATE_DLL if compiling this component as a DLL.

If none of these symbols is defined it is assumed that a static library is to be built.

NOTE: The GNU C++ compiler does not require these symbols to be present in class definitions since all exportable symbols of a DLL are exported by default. But to be compatible with compilers like Microsoft Visual C++ they should be used.

```
#  if defined(SDB_USE_DLL)
#    define SDB_EXPORT __declspec(dllimport)
#  elif defined(SECONDO_CREATE_DLL)
#    define SDB_EXPORT __declspec(dllexport)
#  else
#    define SDB_EXPORT
#  endif
#  if defined(SMI_USE_DLL)
#    define SMI_EXPORT __declspec(dllimport)
#  elif defined(SMI_CREATE_DLL)
#    define SMI_EXPORT __declspec(dllexport)
#  else
#    define SMI_EXPORT
#  endif
```

Linux

```
#elif defined(__linux__)
#  define SECONDO_LINUX
```

Creating shared libraries requires no special measures.

```
#  define SDB_EXPORT
#  define SMI_EXPORT
```

Solaris

```
#elif defined(unix) && defined(sun)
#  define SECONDO_SOLARIS
```

Creating shared libraries requires no special measures.

```
#  define SDB_EXPORT
#  define SMI_EXPORT
#else
#  error Could not identify the operating system
#endif
```

The following symbol must be defined in order to use the latest POSIX APIs:

```
#ifdef __GNUC__
#  define _GNU_SOURCE 1
#endif
```

Define separator character for pathnames in PATH environment variable:

```
#ifdef SECONDO_WIN32
#  define PATH_SLASH "\\"
#  define PATH_SLASHCHAR '\\'
#  define PATH_SEP ";"
#  define PATH_SEPCHAR ';'
#else
#  define PATH_SLASH "/"
#  define PATH_SLASHCHAR '/'
#  define PATH_SEP ":"
#  define PATH_SEPCHAR ':'
#endif
```

Number of elements in a static array:

```
#define nelems(x) (sizeof((x))/sizeof(*(x)))
```

Size of a structure member:

```
#define sizeofm(struct_t,member) \
  ((size_t)(sizeof(((struct_t *)0)->member)))
```

Default includes:

- stdint.h – defines standard integer types (alternatively *inttypes.h* could be used where *stdint.h* is not available).
- string – defines the C++ string data type.

```
#ifndef SECONDO_SOLARIS
#include <stdint.h>
#else
#include <inttypes.h>
#endif
#include <string>

#endif // SECONDO_CONFIG_H
```

C.2 Header File: Secondo Interface

September 1995 Claudia Freundorfer

January 2, 1997 Ralf Hartmut Güting. Explanation of commands and introduction of error codes.

January 10, 1997 RHG Additional error codes for error lists. Made variable *Errors* available as well as procedure *NumericTypeExpr*.

January 17, 1998 RHG Changes in description, new error code 9.

May 15, 1998 RHG Added a command *model value-expression* which is analogous to *query value-expression* but computes the result model for a given query rather than the result value.

May 2002 Ulrich Telle Port to C++, added initialization and termination methods for hiding the details of setting up the SECONDO environment from the user interface program.

C.2.1 Overview

This module defines the procedure *Secondo* as defined in "The Secondo Project" [Gü95]. The main procedure *Secondo* reads a command and executes it. It possibly returns a result.

The first subsection describes the initialization and termination of the interface.

The second subsection describes the various commands and the errors that may occur for each of them.

The third subsection describes error handling and the specific error codes; it makes error messages available in an array *errors*.

The fourth subsection makes some procedures for type transformation and information available, namely *NumericTypeExpr*, *GetTypeId* and *LookUpTypeExpr*. *NumericTypeExpr* transforms type expressions into a numeric form suitable for writing application programs treating types in a generic way (e.g. representation of values at the user interface). *GetTypeId* returns the algebra and type identifier for a type.

Note that there have been some slight changes in respect to [Gü95] in the treatment of the database state in database commands.

```
#ifndef SECONDO_INTERFACE_H
#define SECONDO_INTERFACE_H

#include <string>
#include <map>
#include "SocketIO.h"
#include "NestedList.h"
#include "AlgebraTypes.h"
```

C.2.2 Class $SecondoInterface$

Creation, Deletion, Initialization and Termination

```
class SecondoInterface
{
 public:
  SecondoInterface();
```

Constructs a SECONDO interface. Depending on the implementation of the interface different member variables are initialized.

```
virtual ~SecondoInterface();
```

Destroys a SECONDO interface.

```
bool Initialize( const string& user, const string& pswd,
                 const string& host, const string& port,
                 string& profile,
                 const bool multiUser = false );
```

Starts up the SECONDO interface. Depending on the implementation not all parameters are required for the interface to be operational.

The current implementation of the SECONDO system does not support user authentification. Nevertheless the *user* identification and the password *pswd* can be specified. In the client/server version this information is passed to the SECONDO server to identify the user session.

In the client/server version the *host* address and the *port* of the SECONDO server are needed to establish a connection to the server, but these parameters may be specified via the configuration file *profile*. The method arguments *host* and *port* take precedence over specifications in the configuration file.

In the single user version only the name of the configuration file *profile* must be specified. Values for *host* and *port* are ignored.

```
void Terminate();
```

Shuts down the SECONDO interface. In the client/server version the connection to the SECONDO server is closed; in the single user version the SECONDO system and the *SmiEnvironment* are shut down.

The SECONDO main interface method

```
void Secondo( const string& commandText,
              const ListExpr commandLE,
              const int commandLevel,
              const bool commandAsText,
              const bool resultAsText,
              ListExpr& resultList,
              int& errorCode,
              int& errorPos,
              string& errorMessage,
              const string& resultFileName =
                              "SecondoResult" );
```

Reads a command and executes it; it possibly returns a result. The command is one of a set of SECONDO commands described below. The parameters have the following meaning.

A SECONDO command can be given at various *levels*; parameter *commandLevel* indicates the level of the current command. The levels are defined as follows:

- 0 – SECONDO executable command in nested list syntax (*list*)

- 1 – SECONDO executable command in SOS syntax (*text*)
- 2 – SECONDO descriptive command after, or without, algebraic optimization (*list*)
- 3 – SECONDO descriptive command after, or without, algebraic optimization (*text*)
- 4 – SECONDO descriptive command before algebraic optimization (*list*)
- 5 – SECONDO descriptive command before algebraic optimization (*text*)
- 7 – Command in some specific query language (e.g. SQL, GraphDB, etc.)

If the command is given in *text* syntax (command levels 1, 3, or 5), then the text string must be placed in *commandText*. If the command is given in *list* syntax, it can be passed either as a text string in *commandText*, in which case *commandAsText* must be `true`, or as a list expression in *commandLE*; in the latter case, *commandAsText* must be `false`.

If the command produces a result (e.g. a query), then the result can be requested to be returned either as a list expression (*resultAsText* is `false`) in the parameter *resultList*, or (*resultAsText* is `true`) in a text file whose name is set to **SecondoResult** by default, but may be overwritten.

Finally, the procedure returns an *errorCode*. There are four general error code numbers:

- 0: no error
- 1: command not recognized
- 9: syntax error in command/expression (found by SECONDO Parser)
- 30: command not yet implemented
- 31: command level not yet implemented

The other error codes are explained below together with the commands that may produce them. If an error occurred (*errorCode* different from 0), then the application can print the error message given in *errors[errorcode]*. Possibly additional information about the error is given in parameter *errorMessage* (e.g. messages by SECONDO Parser) and in the *resultList* which is then a list of errors in the form explained below.

Furthermore, *errorPos* contains a position within the *commandBuffer* where the error was detected (only when the command was given in the text buffer, of course). - not yet implemented. -

Basic Commands

```
type <identifier> = <type expression>
delete type <identifier>
create <identifier> : <type expression>
update <identifier> := <value expression>
delete <identifier>
query <value expression>

(not yet implemented:)
let <identifier> = <value expression>
persistent <identifier>
```

All basic commands are only valid, if currently a database is open.

```
type <identifier> = <type expression>
```

Define a type name *identifier* for the type expression. Possible errors:

- 10: *identifier* already used
- 5: error in type expression
- 6: no database open

In case of error 5, an error list with further information is returned in *resultList*.

```
delete type <identifier>
```

Delete the type name *identifier*. Possible errors:

- 11: *identifier* is not a known type name
- 14: type name is used by an object
- 6: no database open

```
create <identifier> : <type expression>
```

Create an object called *identifier* of the type given by *type expression*. The value is still undefined. Possible errors:

- 10: *identifier* already used
- 4: error in *type expression*
- 6: no database open

In case of error 4, an error list with further information is returned in *resultList*.

```
update <identifier> := <value expression>
```

Assign the value computed by *value expression* to the object *identifier*. Possible errors:

- 12: *identifier* is not a known object name
- 2: error in *value expression*
- 3: *value expression* is correct, but not evaluable (e.g. a stream)
- 8: undefined object value in *value expression*
- 13: type of value is different from type of object
- 6: no database open

```
delete <identifier>
```

Destroy the object *identifier*. Possible errors:

- 12: *identifier* is not a known object name
- 6: no database open

```
query <value expression>
```

Evaluate the value expression and return the result as a nested list. Possible errors:

- 2: error in *value expression*
- 3: *value expression* is correct, but not evaluable (e.g. a stream)
- 8: undefined object value in *value expression*
- 6: no database open

Not yet implemented:

```
let <identifier> = <value expression>
```

Assign the value resulting from *value expression* to a new object called *identifier*. The object must not exist yet; it is created by this command and its type is by definition the one of the value expression. This object is temporary so far; it will be automatically destroyed at the end of a session. Possible errors:

- 10: *identifier* already used
- 2: error in *value expression*

- 3: *value expression* is correct, but not evaluable (e.g. a stream)
- 6: no database open

```
persistent <identifier>
```

Make the object *identifier* persistent, so that it will survive the end of a session. Presumably this object was created by a *let* command earlier. If it is not a temporary object, this command is not an error, but has no effect. Possible errors:

- 12: *identifier* is not a known object name
- 6: no database open

NOTE: Beginning with version 2 of the SECONDO system all objects are persistent by default. That is the *persistent* command is obsolete.

Transaction Commands

```
begin transaction
commit transaction
abort transaction
```

These commands can only be used when a database is open. Any permanent changes to a database can only be made within a transaction. Only a single transaction can be active for this particular client at any time.

```
begin transaction
```

Start a transaction. Possible errors:

- 20: a transaction is already active.
- 6: no database open

```
commit transaction
```

Commit a running transaction; all changes to the database will be effective. Possible errors:

- 21: no transaction is active.
- 6: no database open

```
abort transaction
```

Abort a running transaction; all changes to the database will be revoked. Possible errors:

- 21: no transaction is active.
- 6: no database open

NOTE: All commands not enclosed in a *begin transaction ... commit/abort transaction* block are implicitly surrounded by a transaction.

Database Commands

```
create database <identifier>
delete database <identifier>
open database <identifier>
close database
save database to <filename>
restore database <identifier> from <filename>
```

NOTE: All database commands can not be enclosed in a transaction.

The commands *create database* and *delete database* are only valid when currently there is no open database (`IsDatabaseOpen() == false`). They leave this state unchanged.

The commands *open database* and *restore database* are only valid when currently there is no open database (`IsDatabaseOpen() == false`), the database is open after successful completion.

The command *close database* is only valid if `IsDatabaseOpen() == true`. No database is open after successful completion.

The command *save database* is only valid if `IsDatabaseOpen() == true`, it leaves the state of the database unchanged.

```
create database <identifier>
```

Create a new database. Possible errors:

- 10: *identifier* already used

- 7: a database is open

```
delete database <identifier>
```

Destroy the database *identifier*. Possible errors:

- 25: *identifier* is not a known database name.
- 7: a database is open

```
open database <identifier>
```

Open the database *identifier*. Possible errors:

- 25: *identifier* is not a known database name.
- 7: a database is open

```
close database
```

Close the currently open database. Possible errors:

- 6: no database open

```
save database to <filename>
```

Write the entire contents of the database *identifier* in nested list format to the file *filename*. The structure of the file is the following:

```
(DATABASE <database name>
  (DESCRIPTIVE ALGEBRA)
    (TYPES
      (TYPE <type name> <type expression>)*
    )
    (OBJECTS
      (OBJECT <object name> (<type name>) <type expression>
                                          <value>)*
    )
```

```
  (EXECUTABLE ALGEBRA)
    (TYPES
      (TYPE <type name> <type expression>)*
    )
    (OBJECTS
      (OBJECT <object name> (<type name>) <type expression>
                                          <value>)*
    )
)
```

If the file exists, it will be overwritten, otherwise be created. Possible errors:

- 6: no database open
- 26: a problem occurred in writing the file (no permission, file system full, etc.)

```
restore database <identifier> from <filename>
```

Read the contents of the file *filename* into the database *identifier*. The database is in open state after successful completion. Previous contents of the database are lost. Possible errors:

- 25: *identifier* is not a known database name
- 27: the database name in the file is different from *identifier*
- 7: a database is open
- 28: a problem occurred in reading the file (syntax error, no permission, file system full, etc.)
- 29: the overall list structure of the file is not correct
- 24: there are errors in type or object definitions in the file

In case of error 24, an error list with further information is returned in *resultList*.

Inquiries

```
list databases
list type constructors
list operators
list types
list objects
```

The last two commands are only valid when a database is open.

```
list databases
```

Returns a list of names of existing databases. Possible errors: none.

```
list type constructors
```

Return a nested list of type constructors (and their specifications). For the precise format see [Gü95]. Possible errors: none.

```
list operators
```

Return a nested list of operators (and their specifications). For the precise format see [Gü95]. Possible errors: none.

```
list types
```

Return a nested list of type names defined in the currently open database. The format is:

```
(TYPES
  (TYPE <type name> <type expression>)*
)
```

Possible errors:

- 6: no database open

```
list objects
```

Return a nested list of objects existing in the currently open database. The format is:

```
(OBJECTS
  (OBJECT <object name> (<type name>) <type expression>)*
)
```

This is the same format as the one used in saving and restoring the database except that the *value* component is missing. The type name is written within a sublist since an object need not have a type name. Possible errors:

- 6: no database open

Type transformation and information methods

The following procedure allows an application to transform a type expression into an equivalent form with numeric codes. This may be useful to provide type-specific output or representation procedures. One can organize such procedures via doubly indexed arrays (indexed by algebra number and type constructor number).

```
ListExpr NumericTypeExpr( const AlgebraLevel level,
                          const ListExpr type );
```

Transforms a given type expression into a list structure where each type constructor has been replaced by the corresponding pair (algebraId, typeId). The catalog corresponding to the current *level* (descriptive or executable) is used to resolve type names in the type expression. For example,

---

```
int    ->    (1 1)

(rel (tuple ((name string) (age int)))

->     ((2 1) ((2 2) ((name (1 4)) (age (1 1)))))
```

Identifiers such as *name*, *age* are moved unchanged into the result list. If a type expression contains other constants that are not symbols, e.g. integer constants as in `(array 10 real)`, they are also moved unchanged into the result list.

The resulting form of the type expression is useful for calling the type specific *In* and *Out* procedures.

```
bool GetTypeId( const AlgebraLevel level,
                const string& name,
                int& algebraId, int& typeId );
```

Finds the *algebraId* and *typeId* of a named type. The catalog corresponding to the current *level* (descriptive or executable) is used to resolve the type name.

```
bool LookUpTypeExpr( const AlgebraLevel level,
                     ListExpr type, string& name,
                     int& algebraId, int& typeId );
```

Finds the *name*, *algebraId* and *typeId* of a type given by the type expression *type*. The catalog corresponding to the current *level* (descriptive or executable) is used to resolve the type name.

```
NestedList* GetNestedList();
```

Returns a reference to the nested list container used by the SECONDO system.

C.2.3 Error Messages

```
static string GetErrorMessage( const int errorCode );
```

Error messages 1 through 30 are generated within *SecondoInterface* and directly returned in procedure *Secondo*. Error messages larger than 40 belong to four groups:

- 4x – errors in type definitions in database files
- 5x – errors in object definitions in database files
- 6x – errors found by kind checking procedures
- 7x – errors found by *In* procedures of algebras in the list representations for values
- 8x – errors found by type checking procedures in algebras (this group does not yet exist at the moment)

All such error messages (larger than 40) are appended to a list *errorList*. Each procedure generating error messages has a parameter *errorInfo* containing a pointer to the current last element of *errorList*. It appends the error message as a list with a command of the form

```
errorInfo = Append(errorInfo, <message list>)
```

If errors are appended to the list within the execution of a SECONDO command, then the list *errorList* is returned in parameter *resultList* of procedure *Secondo*. This currently happens for the commands

```
type <identifier> = <type expr>
create <identifier> : <type expr>
restore database <identifier> from <filename>
```

since these commands involve kind checking and checking of value list representations for objects.

The messages that are appended to *errorList* usually have further parameters in addition to the error code number. The list following the error message below describes the error entry appended. The parameters after the error number have the following meaning:

- *i*: number of type definition or object definition in database file (the *i*-th type definition, the *i*-th object definition),
- *n*: type name or object name in that definition,
- *k*: kind name,
- *t*: type expression,
- *j*: error number specific to a given kind *k* or type constructor *tc*,
- *tc*: a type constructor,
- *v*: value list, list structure representing a value for a given type constructor.

```
errors[1]  = "Command not recognized.";
errors[2]  = "Error in (query) expression.";
errors[3]  = "Expression not evaluable. "
             "(Operator not recognized or stream?)";
errors[4]  = "Error in type expression. No object created.";
errors[5]  = "Error in type expression. No type defined.";
errors[6]  = "No database open.";
errors[7]  = "A database is open.";
errors[8]  = "Undefined object value in (query) expression";
errors[9]  = "Syntax error in command/expression";

errors[10] = "Identifier already used.";
errors[11] = "Identifier is not a known type name.";
errors[12] = "Identifier is not a known object name.";
errors[13] = "Type of expression is different from type of "
             "object.";
errors[14] = "Type name is used by an object. Type not deleted.";

errors[20] = "Transaction already active.";
errors[21] = "No transaction active.";
errors[22] = "Begin transaction failed.";
errors[23] = "Commit/Abort transaction failed.";

errors[24] = "Error in type or object definitions in file.";
errors[25] = "Identifier is not a known database name.";
errors[26] = "Problem in writing to file.";
errors[27] = "Database name in file different from identifier.";
errors[28] = "Problem in reading from file.";
errors[29] = "Error in the list structure in the file.";
errors[30] = "Command not yet implemented.";
errors[31] = "Command level not yet implemented.";
errors[32] = "Command not yet implemented at this level.";

errors[40] = "Error in type definition.";             // (40 i)
errors[41] = "Type name doubly defined.";             // (41 i n)
errors[42] = "Error in type expression.";             // (42 i n)

errors[50] = "Error in object definition.";           // (50 i)
errors[51] = "Object name doubly defined.";           // (51 i n)
errors[52] = "Wrong type expression for object.";     // (52 i n)
errors[53] = "Wrong list representation for object."; // (53 i n)

errors[60] = "Kind does not match type expression.";  // (60 k t)
errors[61] = "Specific kind checking error for kind.";
                          // (61 k j ...)

errors[70] = "Value list is not a representation for type "
             "constructor."; // (70 tc v)
errors[71] = "Specific error for type constructor in "
             "value list.";  // (71 tc j ...)
errors[72] = "Value list is not a representation for type "
             "constructor."; // (72 tc)
errors[73] = "Error at a position within value list for type "
             "constructor."; // (73 pos)

errors[80] = "Secondo protocol error.";
errors[81] = "Connection to Secondo server lost.";
```

The error messages 61 and 71 allow a kind checking procedure or an *In* procedure to introduce its own specific error codes (just numbered 1, 2, 3, ...). These error messages may then have further parameters. To interpret such error messages (and return information to the user) one needs to add code branching on these specific error code numbers. Such code may or may not be supplied with an algebra.

```
  void SetDebugLevel( const int level );
```

Sets the debug level of the query processor.

```
 protected:
 private:
  void StartCommand();
  void FinishCommand( int& errorCode );

  bool        initialized;       // state of interface
  bool        activeTransaction; // state of transaction block
  NestedList* nl;                // Reference of
                                 //   nested list container
  Socket*     server;            // used in C/S version only

  static void InitErrorMessages();
  static bool errMsgInitialized;
  static map<int,string> errors;
};
```

References

[Gü95] Güting, R.H., The SECONDO Project. Praktische Informatik IV, Fernuniversität Hagen, working paper, November 1995.

```
#endif
```

C.3 Header File: Secondo System

May 2002 Ulrich Telle Port to C++

C.3.1 Overview

This module implements those parts of the SECONDO catalog which are independent of the algebra level (descriptive or executable).

It manages a set of databases. A database consists of a set of named types, a set of objects with given type name or type expressions and a set of models for objects. Objects can be persistent or not. Persistent objects are implemented by the *Storage Management Interface*. When a database is opened, for each algebra level a catalog with informations about types, type constructors, operators, objects and models of the database is loaded. Furthermore the catalog of each algebra is loaded into memory by calling the procedures of the module *Algebra Manager*.

C.3.2 Interface methods

The class *SecondoSystem* provides the following methods:

System Management	Database Management	Information
GetInstance	CreateDatabase	ListDatabaseNames
StartUp	DestroyDatabase	IsDatabaseOpen
ShutDown	OpenDatabase	GetDatabaseName
GetAlgebraManager	CloseDatabase	
GetQueryProcessor	SaveDatabase	
GetCatalog	RestoreDatabase	
GetNestedList		

C.3.3 Imports

```
#ifndef SECONDO_SYSTEM_H
#define SECONDO_SYSTEM_H

#include "NestedList.h"
#include "AlgebraManager.h"
#include "SecondoCatalog.h"
```

Forward declaration of several classes:

```
class NestedList;
class AlgebraManager;
class QueryProcessor;
class SecondoCatalog;
```

C.3.4 Class $SecondoSystem$

This class implements all algebra level independent functionality of the SECONDO catalog management.

```
class SecondoSystem
{
 public:
  SecondoSystem( GetAlgebraEntryFunction getAlgebraEntryFunc );
  virtual ~SecondoSystem();
  ListExpr ListDatabaseNames();
```

Simply returns the names of existing databases in a list:

```
(<database name 1>..<database name n>)
```

```
  bool CreateDatabase( const string& dbname );
```

Creates a new database named *dbname* and loads the algebraic operators and type constructors for each algebra level into the SECONDO programming interface. Returns `false` if a database under this name already exists.

Precondition: No database is open.

```
  bool DestroyDatabase( const string& dbname );
```

Deletes a database named *dbname* and all data files belonging to it. Returns `false` if the database *dbname* is not known.

Precondition: No database is open.

```
  bool OpenDatabase( const string& dbname );
```

Opens a database with name *dbname*. Returns `false` if database *dbname* is unknown.

Precondition: No database is open.

```
  bool CloseDatabase();
```

Closes the currently opened database.

Precondition: A database is open.

```
  bool IsDatabaseOpen();
```

Returns `true` if a database is in open state, otherwise `false`.

```
  bool SaveDatabase( const string& filename );
```

Writes the currently open database called *dbname* to a file with name *filename* in nested list format. The format is as follows:

```
(DATABASE <database name>
  (DESCRIPTIVE ALGEBRA)
    (TYPES
      (TYPE <type name> <type expression>)*
```

```
    )
    (OBJECTS
      (OBJECT <object name> (<type name>) <type expression>
                                          <value> <model>)*
    )
  (EXECUTABLE ALGEBRA)
    (TYPES
      (TYPE <type name> <type expression>)*
    )
    (OBJECTS
      (OBJECT <object name> (<type name>) <type expression>
                                          <value> <model>)*
    )
)
```

Returns `false` if there was a problem in writing the file.

Precondition: A database is open.

```
int RestoreDatabase( const string& dbname,
                     const string& filename,
                     ListExpr& errorInfo );
```

Reads a database from a file named *filename* that has the same nested list format as described in the method *SaveDatabase* and fills the catalogs for database types and objects. The database is in open state after successful completion. Returns error 1 if *dbname* is not a known database name, error 2, if the database name in the file is different from *dbname* here, error 3, if there was a problem in reading the file, and error 4, if the list structure in the file was not correct. Returns error 5 if there are errors in type definitions and/or object list expressions.

Furthermore, any errors found by kind checking and by *In* procedures are returned in the list *errorInfo*.

Precondition: No database is open.

```
string                GetDatabaseName();
```

Returns the name of the currently open database. An empty string is returned if no database is in open state.

```
static SecondoSystem*  GetInstance();
```

Returns a reference to the single instance of the SECONDO system.

```
static bool            StartUp();
```

Initializes the SECONDO system. The *Storage Management Interface* is started and the algebra modules are loaded into main memory.

```
static bool            ShutDown();
```

Shuts down the SECONDO system and the *Storage Management Interface*. Dynamically loaded algebra modules are unloaded.

```
  static AlgebraManager* GetAlgebraManager();
```

Returns a reference to the associated algebra manager.

```
  static QueryProcessor* GetQueryProcessor();
```

Returns a reference to the associated query processor.

```
  static SecondoCatalog* GetCatalog( const AlgebraLevel level );
```

Returns a reference to the SECONDO catalog of the specified algebra *level*.

```
  static NestedList*     GetNestedList();
```

Returns a reference to the associated nested list container.

```
  static bool BeginTransaction();
```

Begins a transaction.

```
  static bool CommitTransaction();
```

Commits a transaction.

```
  static bool AbortTransaction();
```

Aborts a transaction.

```
 protected:
  SecondoSystem( const SecondoSystem& );
  SecondoSystem& operator=( const SecondoSystem& );
 private:
  bool RestoreCatalog( SecondoCatalog* sc,
                       ListExpr types, ListExpr objects,
                       ListExpr& errorInfo );
  bool RestoreTypes( SecondoCatalog* sc,
                     ListExpr types, ListExpr& errorInfo );
  bool RestoreObjects( SecondoCatalog* sc,
                       ListExpr objects, ListExpr& errorInfo );
```

Are internal methods for restoring a database.

```
  static SecondoSystem* secondoSystem;

  NestedList*     nl;
  AlgebraManager* algebraManager;
  QueryProcessor* queryProcessor;
  SecondoCatalog* scDescriptive;
  SecondoCatalog* scExecutable;

  bool testMode;
  bool initialized;
};

#endif
```

C.4 Header File: Secondo Catalog

September 1996 Claudia Freundorfer

December 20, 1996 RHG Changed definition of procedure *OutObject*.

May 15, 1998 RHG Added treatment of models, especially functions *InObjectModel*, *OutObjectModel*, and *ValueToObjectModel*.

May 2002 Ulrich Telle Port to C++

C.4.1 Overview

This module defines the module *SecondoCatalog*. It manages a set of named types, a set of objects with given type name or type expressions and a set of models for objects for a database at a specific algebra level. Persistency is implemented by the *Storage Management Interface*.

Modifications to the catalog by the methods of this module are registered in temporary data structures in memory and written to disk on completion of the enclosing transaction.

C.4.2 Interface methods

The class *SecondoCatalog* provides the following methods:

Catalog and Types	Object Values and Models	Type Constructors / Operators
SecondoCatalog	ListObjects	ListTypeConstructors
~SecondoCatalog	ListObjectsFull	IsTypeName
Open	CreateObject	GetTypeId
Close	InsertObject	GetTypeName
CleanUp	DeleteObject	GetTypeDS
	InObject	
	GetObjectValue	
	OutObject	
ListTypes	IsObjectName	ListOperators
InsertType	GetObject	IsOperatorName
DeleteType	GetObjectExpr	GetOperatorId
MemberType	GetObjectType	GetOperatorName
LookUpTypeExpr	UpdateObject	
GetTypeExpr	InObjectModel	
NumericType	OutObjectModel	
ExpandedType	ValueToObjectModel	
KindCorrect	ValueListToObjectModel	

C.4.3 Imports

```
#ifndef SECONDO_CATALOG_H
#define SECONDO_CATALOG_H

#include "AlgebraManager.h"
#include "NestedList.h"
#include "NameIndex.h"
#include "SecondoSMI.h"
```

C.4.4 Class *SecondoCatalog*

This class implements all functionality of the SECONDO catalog management dependent of a specific algebra level.

All operations on types and objects are valid only, when the associated database is open. Type constructors and operators may be accessed when no database is open.

```
class SecondoCatalog
{
 public:
  SecondoCatalog( const string& name,
                  const AlgebraLevel level );
```

Creates a new catalog with the given *name* for the specified algebra *level*.

```
  virtual ~SecondoCatalog();
```

Destroys a catalog.

```
  bool Open();
```

Opens the catalog for operation. Returns `true` if the catalog could be opened successfully, otherwise `false`.

```
  bool Close();
```

Closes the catalog. Returns `true` if the catalog could be closed successfully, otherwise `false`.

```
  bool CleanUp( const bool revert );
```

Cleans up the memory representation when a transaction is completed. The switch *revert* has to be set to `true` if the enclosing transaction is aborted.

Database Types

```
  ListExpr ListTypes();
```

Returns a list of types of the whole database in the following format:

```
(TYPES
  (TYPE <type name><type expression>)*
)
```

```
  bool InsertType( const string& typeName,
                   ListExpr typeExpr );
```

Inserts a new type with identifier *typeName* defined by a list *typeExpr* of already existing types in the database. Returns `false`, if the name was already defined.

```
int DeleteType( const string& typeName );
```

Deletes a type with identifier *typename* in the database. Returns error 1 if type is used by an object, error 2, if *typename* is not known.

```
bool MemberType( const string& typeName );
```

Returns `true`, if type with name *typename* is member of the actually open database.

```
bool LookUpTypeExpr( const ListExpr typeExpr,
                     string& typeName,
                     int& algebraId, int& typeId );
```

Returns the algebra identifier *algebraId* and the type identifier *opId* and the name *typeName* of the outermost type constructor for a given type expression *typeExpr*, if it exists, otherwise an empty string as *typeName* and value 0 for the identifiers, and the methods return value is set to `false`.

```
ListExpr GetTypeExpr( const string& typeName );
```

Returns a type expression for a given type name *typename*, if exists.

Precondition: `MemberType( typeName ) == true`.

```
ListExpr NumericType( const ListExpr type );
```

Transforms a given type expression into a list structure where each type constructor has been replaced by the corresponding pair (algebraId, typeId). For example,

---

```
int -> (1 1)

(rel (tuple ((name string) (age int)))

->  ((2 1) ((2 2) ((name (1 4)) (age (1 1)))))
```

Identifiers such as *name*, *age* are moved unchanged into the result list. If a type expression contains other constants that are not symbols, e.g. integer constants as in (array 10 real), they are also moved unchanged into the result list.

The resulting form of the type expression is useful for calling the type specific *In* and *Out* procedures.

```
ListExpr ExpandedType ( const ListExpr type );
```

Transforms a given type definition (a type expression possibly containing type names, or just a single type name) into the corresponding type expression where all names have been replaced by their defining expressions.

Kind Checking

```
bool KindCorrect ( const ListExpr type, ListExpr& errorInfo );
```

Here *type* is a type expression. *KindCorrect* does the kind checking; if there are errors, they are reported in the list *errorInfo*, and `false` is returned. *errorInfo* is a list whose entries are again lists, the first element of an entry is an error code number. For example, an entry

---

```
(1 DATA (hello world))
```

---

says that kind *DATA* does not match the type expression *(hello world)*. This is the meaning of the general error code 1. The other error codes are type-constructor specific.

Database Objects and Models

```
ListExpr ListObjects();
```

Returns a list of *objects* of the whole database in the same format that is used in the procedures *SaveDatabase* and *RestoreDatabase*:

```
(OBJECTS
  (OBJECT <object name>(<type name>) <type expression>)*
)
```

For each object the **value** and **model** component is missing, otherwise the whole database would be returned.

```
ListExpr ListObjectsFull();
```

Returns a list of *objects* of the whole database in the following format:

```
(OBJECTS
  (OBJECT <object name>(<type name>) <type expression>
                                     <value> <model>)*
)
```

```
bool CreateObject( const string& objectName,
                   const string& typeName,
                   const ListExpr typeExpr,
                   const int sizeOfComponents );
```

Creates a new object with identifier *objectName* defined with type name *typeName* (can be empty) and type *typeExpr*. The value is not yet defined, and no memory is allocated. Returns `false`, if the object name is defined already.

```
bool InsertObject( const string& objectName,
                   const string& typeName,
                   const ListExpr typeExpr,
                   const Word valueWord,
                   const bool defined,
                   const Word modelWord );
```

Inserts a new object with identifier *objectName* and value *valueWord* defined by type name *typeName* or by a list *typeExpr* of already existing types (which always exists) into the database catalog. Parameter *defined* tells, whether *valueWord* actually contains a defined value. Further, *modelWord* contains a model for this value, possibly 0, the undefined model. If the object name already exists, the procedure has no effect. Returns `false` if the *objectName* is already in use.

When the given object has no type name, it is mandatory, that *typeName* is an empty string.

```
bool DeleteObject( const string& objectName );
```

Deletes an object with identifier *objectName* in the database. Returns `false` if the object does not exist.

```
Word InObject( const ListExpr typeExpr,
               const ListExpr valueList,
               const int errorPos,
               ListExpr& errorInfo,
               bool& correct );
```

Converts an object of the type given by *typeExpr* and the value given as a nested list into a *Word* representation which is returned. Any errors found are returned together with the given *errorPos* in the list *errorInfo*. *correct* is set to `true` if a value was created (which means that the input was at least partially correct).

NOTE: Works only at the executable level.

```
ListExpr GetObjectValue( const string& objectName );
```

Returns the value of a locally stored database object with identifier *objectName* as list expression to show the value to the database user. If the value is undefined, an empty list is returned.

NOTE: Works only at the executable level.

```
ListExpr OutObject( const ListExpr type,
                    const Word object );
```

Returns for a given *object* of type *type* its value in nested list representation.

NOTE: Works only at the executable level.

```
bool IsObjectName( const string& objectName );
```

Checks whether *objectName* is a valid object name.

```
bool GetObject( const string& objectName,
                Word& word, bool& defined );
```

Returns the value *word* of an object with identifier *objectName*. *defined* tells whether the word contains a meaningful value.

NOTE: Works only at the executable level.

Precondition: `IsObjectName( objectName ) == true`.

```
bool GetObjectExpr( const string& objectName,
                    string& typeName,
                    ListExpr& typeExpr,
                    Word& value,
                    bool& defined,
                    Word& model,
                    bool& hasTypeName );
```

Returns the value *value*, the type name *typeName*, the type expression *typeExpr*, and the *model* of an object with identifier *objectName*. *defined* tells whether *value* contains a defined value. If object has no type name the variable *hasTypeName* is set to `false` and the procedure returns an empty string as *typeName*.

Precondition: `IsObjectName(objectName) == true`.

```
bool GetObjectType( const string& objectName,
                    string& typeName );
```

Returns the type name *typeName* of an object with identifier *objectName*, if the type name exists and an empty string otherwise.

Precondition: `IsObjectName( objectName ) == true`.

```
bool UpdateObject( const string& objectName,
                   const Word word );
```

Overwrites the value of the object with identifier *objectName* with a new value *word*. Returns `false` if object does not exist.

NOTE: Works only at the executable level.

```
Word InObjectModel( const ListExpr typeExpr,
                    const ListExpr modelList,
                    const int objNo );
```

Converts a model of the type given by *typeExpr* and the value given as a nested list into a `Word` representation which is returned.

```
ListExpr OutObjectModel( const ListExpr typeExpr,
                         const Word model );
```

Returns for a given *model* of type *typeExpr* its description in nested list representation.

```
Word ValueToObjectModel( const ListExpr typeExpr,
                         const Word value );
```

Returns for a given *value* of type *typeExpr* its model.

NOTE: Works only at the executable level.

```
Word ValueListToObjectModel( const ListExpr typeExpr,
                             const ListExpr valueList,
                             int& errorPos,
                             ListExpr& errorInfo,
                             bool& correct );
```

Returns for a given *valueList* of type *typeExpr* its model. Any errors found are returned together with the given *errorPos* in the list *errorInfo*. *correct* is set to `true` if a model was created .

NOTE: Works only at the descriptive level.

Algebra Type Constructors

```
ListExpr ListTypeConstructors();
```

Returns a list of type constructors of the actually load algebras in the following format:

```
(
  (<type constructor name> (<arg 1>..<arg n>) <result>) *
)
```

```
bool IsTypeName( const string& typeName );
```

Checks whether *typeName* is a valid name for an algebra type constructor or a database type.

```
bool GetTypeId( const string& typeName,
                int& algebraId, int& typeId );
```

Returns the algebra identifier *algebraId* and the type identifier *opId* of an existing type constructor or database type with name *typeName*.

Precondition: `IsTypeName( typeName ) == true`.

```
string GetTypeName( const int algebraId, const int typeId );
```

Looks up the name of a type constructor defined by the algebra identifier *algebraId* and the type identifier *opId*.

```
ListExpr GetTypeDS( const int algebraId, const int typeId );
```

Looks up the properties of a type constructor defined by the algebra identifier *algebraId* and the type identifier *opId*.

Algebra Operators

```
ListExpr ListOperators();
```

Returns a list of operators specifications in the following format:

```
(
  ( <operator name>
    (<arg type spec 1>..<arg type spec n>)
    <result type spec>
    <syntax>
    <variable defs>
    <formula>
    <explaining text>
  )*
)
```

This format is based on the formal definition of the syntax of operator specifications from [BeG95b, Section3.1].

```
bool IsOperatorName( const string& opName );
```

Checks whether *opName* is a valid operator name.

```
void GetOperatorId( const string& opName,
                    int& algebraId, int& opId );
```

Returns the algebra identifier *algebraId* and the operator identifier *opId* of an existing *opName*.

Precondition: `IsOperatorName( opName ) == true`.

```
string GetOperatorName( const int algebraId,
                        const int opId );
```

Looks for the name of an operator defined by the algebra identifier *algebraId* and the operator *opId*.

```
ListExpr GetOperatorSpec( const int algebraId,
                          const int opId );
```

Returns the operator specification of an operator defined by the algebra identifier *algebraId* and the operator identifier *opId* in the following format:

```
( <operator name>
  (<arg type spec 1>..<arg type spec n>)
  <result type spec>
  <syntax>
  <variable defs>
  <formula>
  <explaining text>
)
```

```
protected:
 bool TypeUsedByObject( const string& typeName );
private:
```

```
  string           catalogName;
  AlgebraLevel     catalogLevel;
  NestedList*      nl;
  AlgebraManager*  am;

  struct CatalogEntry
  {
    int      algebraId;
    int      entryId;
  };
  typedef map<string,CatalogEntry> LocalCatalog;
  LocalCatalog constructors;
  LocalCatalog operators;

  enum EntryState { EntryInsert, EntryUpdate, EntryDelete };

  struct TypesCatalogEntry
  {
    int        algebraId;
    int        typeId;
    string     typeExpr;
    EntryState state;
  };
  typedef map<string,TypesCatalogEntry> TypesCatalog;
  TypesCatalog types;
  SmiKeyedFile typeCatalogFile;

  struct ObjectsCatalogEntry
  {
    int         algebraId;
    int         typeId;
    string      typeName;
    string      typeExpr;
    Word        value;
    bool        valueDefined;
    SmiRecordId valueRecordId;
    Word        model;
    SmiRecordId modelRecordId;
    EntryState  state;
  };
  typedef map<string,ObjectsCatalogEntry> ObjectsCatalog;
  ObjectsCatalog objects;
  SmiKeyedFile   objCatalogFile;
  SmiRecordFile  objValueFile;
  SmiRecordFile  objModelFile;

  bool testMode;
```

If *testMode* is set some preconditions are tested. If an error occurs, `exit` is called.

TODO: `exit` should never be called in the server version. In case of an error it should always be reported to the client.

```
  friend class SecondoSystem;
};

#endif
```

C.5 Header File: Query Processor

September 1996 Claudia Freundorfer

December 23, 1996 RHG Changed procedure *construct* to include checks for correctness of query and evaluability of the operator tree.

January 3, 1997 RHG Added parameter *defined* to procedures *construct* and *annotateX* to allow checking whether all objects have defined values.

May 4, 1998 RHG Added procedure *resultStorage*.

May 4, 1998 RHG Added procedure *getSupplier*.

May 15, 1998 RHG Added procedures *evalModel* and *requestModel*.

June 18, 1998 RHG Added procedures *getNoSons* and *getType*.

January 24, 2001 RHG Change of procedure *Destroy* taken from SecondoReference.

January 26, 2001 RHG Added an *isFunction* parameter to procedure *construct*.

May 2002 Ulrich Telle Port to C++, integrated descriptive algebra level and function mapping.

C.5.1 Overview

This module describes the interface of module *QueryProcessor*. The modules offers all the basic operations for executing an executable command in nested list format. The module *QueryProcessor* offers a data structure to store an operator tree, procedures to build an operator tree from an access plan given by the optimizer and procedures to execute it by quasi-coroutines.

The task of the query processor is to evaluate queries given as nested list expressions. It divides the task into three steps:

1. The given query is *annotated* which means all the symbols occurring in the query are analyzed (e.g. objects or operators are looked up in the system catalog) and the found information is attached to the symbol. The result is an *annotated query*, again a nested list. This is done by method *Annotate*.

   This step includes calling *type mapping functions* for each operator of the query. The type mapping function gets a list of argument types (associated with the arguments of the operator in the query). It then checks whether argument types are correct. If so, it returns a result type, otherwise a special symbol *typeerror*. The result type is used to annotate the operator application in the query.

   This step also includes calling a *selection function* which maps an operator into an evaluation procedure. This is used for overloaded operators, for example + (with four evaluation functions for the possible combinations of int and real arguments). All operators have such a selection function, even if they are not overloaded; in that case the selection function is simple (e.g. identity for the operator number to function number mapping) and it may be the same for all operators of an algebra.

2. An annotated query is taken and transformed into an *operator tree*. Method *Subtree* constructs the tree from the annotated query.

3. Finally, an evaluation method called *Eval* traverses the tree, calling *evaluation functions* for the operators there. The evaluation functions can call (their) parameter func-

tions through special interface procedures to the query processor. They can also produce or consume streams in cooperation with the query processor (that is, *Eval*).

C.5.2 Interface methods

The class *QueryProcessor* provides the following methods:

Creation/Deletion/Test	Operator tree	Operator handling
QueryProcessor	Construct	Argument
~QueryProcessor	Eval	Request
	EvalModel	Received
	Destroy	Open
		Close
		RequestModel
AnnotateX		GetSupplier
SubtreeX		ResultStorage
ListOfTree		GetNoSons
SetDebugLevel		GetType

C.5.3 Imports and Types

```
#ifndef QUERY_PROCESSOR_H
#define QUERY_PROCESSOR_H

#include "AlgebraManager.h"
```

defines the basic types of the query processor such as `ArgVectorPointer`, `Supplier`, `Word`, `Address`, etc.

```
#include "SecondoCatalog.h"
#include "SecondoSystem.h"

struct OpNode;
typedef OpNode* OpTree;

struct VarEntry
{
  int position;
  int funindex;
  ListExpr typeexpr;
};

typedef CTable<VarEntry>  VarEntryCTable;
```

C.5.4 Class $QueryProcessor$

This class implements all methods for the SECONDO query processor.

```
class QueryProcessor
{
 public:
  QueryProcessor( NestedList* newNestedList,
                  AlgebraManager* newAlgebraManager );
```

Creates a query processor instances using the provided nested list container and algebra manager.

```
virtual ~QueryProcessor();
```

Destroys a query processor instance.

Construction and Execution of an Operator Tree

```
void Construct( const AlgebraLevel level,
                const ListExpr expr, bool& correct,
                bool& evaluable, bool& defined,
                bool& isFunction,
                OpTree& tree, ListExpr& resultType );
```

Builds an operator tree *tree* from a given list expression *expr* by calling the procedures *annotateX* and *subtreeX*. The tree is only constructed if *annotateX* does not find a type error. If there is no error, then *correct* is TRUE, the tree is returned in *tree* and the result type of the expression in *resultType*. In addition, for a descriptive query (*level* = *descriptive*), models are evaluated and stored in the tree. If there is a type error, *correct* is set to `false` and *resultType* contains a symbol *typeerror*.

If there is an object with undefined value mentioned in the query, then *defined* is `false`.

Even if there is no type error, a query may not be evaluable, for example, if the outermost operator produces a stream, or the query is just an argument list. The query processor may also view the query as an argument list, if the root operator is not recognized (an error in the query). Therefore, *construct* returns in *evaluable*, whether the constructed tree can indeed be evaluated.

Finally, it is returned in *isFunction* whether the tree represents an abstraction. In this case, it is not evaluable, but we may want to store the function in a database object.

```
void Eval( const OpTree tree, Word& result,
           const int message );
```

Traverses the operator tree *tree* calling operator implementations for each node, and returns the result in *result*. The *message* is `OPEN`, `REQUEST`, or `CLOSE` and is used only if the root node produces a stream.

```
void EvalModel( const OpTree tree, Word& result );
```

Traverses the operator tree *tree* calling operator model mapping functions for each node, and returns the result in *result*and stores it in *subtreeModel*. This is similar to *eval*, but we do not need to handle stream evaluation.

```
void Destroy( OpTree& tree, const bool destroyRootValue );
```

Deletes an operator tree object. If *DestroyRootValue* is `false`, the result value stored in the root node is not deleted.

Handling of Parameter Functions and Stream Operators

```
ArgVectorPointer Argument( const Supplier s );
```

Returns for a given supplier *s* a pointer to its argument vector. Arguments can be set by writing into the fields of this argument vector.

```
void Request( const Supplier s, Word& word );
```

Calls the parameter function (to which the arguments must have been supplied before). The result is returned in *result*.

```
bool Received( const Supplier s );
```

Returns `true` if the supplier responded to the previous *request* by a `YIELD` message; `false` if it responded with `CANCEL`.

```
void Open( const Supplier s );
```

Changes state of the supplier stream to *open*.

```
void Close( const Supplier s );
```

Changes state of the supplier stream to *closed*. No effect, if the stream is closed already.

```
void RequestModel( const Supplier s, Word& result );
```

Calls the parameter function of a model mapping function (to which the arguments must have been supplied before). The result is returned in *result*. This one is used for model evaluation.

```
Supplier GetSupplier( const Supplier s, const int no );
```

From a given supplier *s* that must represent an argument list, get its son number *no*. Can be used to traverse the operator tree in order to access arguments within (nested) argument lists. Values or function or stream evaluation can then be obtained from the returned supplier by the usual calls to *request* etc.

```
Word ResultStorage( const Supplier s );
```

For each operator in an operator tree, the query processor allocates a storage block for the result value (which it also destroys after execution of the query). The operator's evaluation function can call this procedure *resultStorage* to get the address of that storage block. As a parameter *s*, the operator's node address has to be given which is passed to the evaluation function in parameter *opTreeNode*.

```
int GetNoSons( const Supplier s );
```

Returns the number of sons of the operator node *s* of the operator tree.

```
ListExpr GetType( const Supplier s );
```

Returns the type expression of the node *s* of the operator tree.

Procedures Exported for Testing Only

```
ListExpr AnnotateX( const AlgebraLevel level,
                    const ListExpr expr, bool& defined );
```

Annotate query expression of algebra at level *level*. Create tables for variables, reset *valueno* and *functionno*, then call *annotate*. Parameter *defined* tells, whether all objects mentioned in the expression have defined values.

```
OpTree SubtreeX( const AlgebraLevel level,
                 const ListExpr expr );
```

Construct an operator tree from *expr*. Allocate argument vectors for all functions and then call *subtree* to do the job.

```
ListExpr ListOfTree( OpTree tree );
```

Represents an operator tree through a list expression. Used for testing.

```
void SetDebugLevel( const int level );
```

Sets the debug level for the query processor. The following levels are defined:

- **0** – Debug mode is turned off
- **1** – Debug mode is turned on (i.e. results of methods *AnnotateX* and *SubtreeX* are displayed)
- **2** – Debug **and** trace mode are turned on

```
private:
 void GetVariable( const string& name, NameIndex& varnames,
                   const VarEntryCTable& vartable,
                   int& position, int& funindex,
                   ListExpr& typeexpr );
```

Get for variable *name* its *position* (number of parameter in the list of parameters) and the number of the abstraction (function definition) *funindex* defining it, as well as the associated *typeexpr*.

Precondition: `IsVariable( name, varnames ) == true`.

```
void EnterVariable( const string& name,
                    NameIndex& varnames,
                    VarEntryCTable& vartable,
                    const int position,
                    const int funindex,
                    const ListExpr typeexpr );
```

Enter *position* (number of parameter), *funindex* (number of abstraction definition) and *typeexpr* for the variable *name* into tables *varnames* and *vartable*.

Precondition: `IsVariable( name, varnames ) == false`.

```
bool IsVariable( const string& name,
                 NameIndex& varnames );
```

Check whether *name* is the name of a variable, that is, occurs in *varnames*.

```
bool IsIdentifier( const AlgebraLevel level,
                   const ListExpr expr,
                   NameIndex& varnames );
```

Expr may be any list expression. Check whether it is an identifier, that is, a symbol atom which is not registered as a variable or an operator.

```
enum QueryProcessorType
       { QP_MAP, QP_FUN, QP_STREAM,
         QP_CONSTANT, QP_OPERATOR, QP_OBJECT,
         QP_FUNCTION, QP_VARIABLE, QP_IDENTIFIER,
         QP_ABSTRACTION, QP_APPLYOP,
         QP_ARGLIST, QP_APPLYABS, QP_APPLYFUN,
         QP_TYPEERROR, QP_ERROR, QP_APPEND,
         QP_UNDEFINED };
```

enumerates the types a symbol may have while annotating an expression.

```
QueryProcessorType TypeOfSymbol( const ListExpr symbol );
```

Transforms a list expression *symbol* into one of the values of type *QueryProcessorType*. *Symbol* is allowed to be any list. If it is not one of these symbols, then the value *error* is returned.

```
ListExpr Annotate( const AlgebraLevel level,
                   const ListExpr expr,
                   NameIndex& varnames,
                   VarEntryCTable& vartable,
                   bool& defined,
                   const ListExpr fatherargtypes );
```

Annotates a query expression *expr* of either the executable or the descriptive *level*. Use tables *varnames* and *vartable* to store variables occurring in abstractions (function definitions) and to retrieve them in the function's expression. Return the annotated expression.

Parameter *defined* is set to `false` if any object mentioned in the expression has an undefined value. Parameter *fatherargtypes* is used to implement inference of parameter types in abstractions. When a function is analyzed by *annotate-function*, then this list contains the argument types of the operator to which this function is a parameter.

```
ListExpr AnnotateFunction( const AlgebraLevel level,
                           const ListExpr expr,
                           NameIndex& varnames,
                           VarEntryCTable& vartable,
                           bool& defined,
                           const int paramno,
                           const ListExpr typeList,
                           const ListExpr lastElem,
                           const ListExpr fatherargtypes );
```

Annotate an abstraction *expr* which has the form:

```
(fun (x1 t1) ... (xn tn) e)
```

---

and return the annotated version:

```
 ->  ((none abstraction annotate(expr) <functionno>) <type>)
```

---

where *type* is a functional type of the form (map ...). *Functionno* is the index in *ArgVectors* used for the argument vector of this function. Before other actions, its value is assigned to *localfunctionno* to catch the case that *functionno* is incremented during annotation of the function body.

```
  bool IsCorrectTypeExpr( const AlgebraLevel level,
                          const ListExpr expr );
  OpTree Subtree( const AlgebraLevel level,
                  const ListExpr expr );
```

Construct operator tree recursively for a given annotated *expr*. See *Annotate* and *Annotate-Function* for the possible structures to be processed.

```
  void AllocateValuesAndModels( int idx );
  void AllocateArgVectors( int idx );
  SecondoCatalog* GetCatalog( const AlgebraLevel level )
  {
    return (SecondoSystem::GetCatalog( level ));
  };

  NestedList*     nl;
  AlgebraManager* algebraManager;

  int  valueno;
  int  functionno;
  bool testMode;
  bool debugMode;
  bool traceMode;

  vector<Word> values; // MAXVALUE = 200
  vector<Word> models;
  vector<ArgVectorPointer> argVectors; // MAXFUNCTIONS = 30
};

#endif
```

C.6 Header File: Secondo Parser

March 2002 Ulrich Telle

C.6.1 Overview

SECONDO offers a fixed set of commands for database management, catalog inquiries, access to types and objects, queries, and transaction control. Some of these commands require type expression, value expression, or identifier arguments. Whether a type expression or value expression is valid or not is determined by means of the specifications provided by the active algebra modules, while validity of an identifier depends on the contents of the actual database.

SECONDO acccepts two different forms of user input: Queries in nested list syntax and queries following a syntax defined by the active algebra modules. Both forms have positive and negative aspects. On the one hand, nested list syntax remains the same, regardless of the actual set of operators provided by active algebra modules. On the other hand, queries in nested list syntax tend to contain a lot of parentheses, thereby getting hard to formulate and read. This is the motivation for offering a second level of query syntax with two important features:

- Reading and writing type expressions is simplified.
- For each operator of an algebra module, the algebra implementor can specify syntax properties like infix or postfix notation. If this feature is used carefully, value expressions can be much more readable.

User level syntax is provided to support the formulation of interactive user queries in a more intuitive and less error-prone way. Compared to nested list syntax, writing SECONDO commands in user level syntax essentially has three effects:

- There is no need to enclose commands by parentheses. The string list type constructors, for instance, is valid input.
- Formulation of type expressions is simplified and more straightforward.
- SECONDO enables the algebra implementor to define syntactic properties of value expressions using the algebra's operators.

Internally the system uses always the nested list representation. Therefore a method for translating the user level syntax into nested list syntax is necessary. The SECONDO parser class provides such a translation feature.

C.6.2 Interface methods

This module offers the following methods:

Creation/Removal	Parsing
SecParser	Text2List
~SecParser	

C.6.3 Class $SecParser$

The class *SecParser* implements a parser for translating Secondo commands in text form into textual nested list representation.

```
#ifndef SEC_PARSER_H
#define SEC_PARSER_H

class SecParser
{
 public:
  SecParser();
```

Creates a Secondo command parser.

NOTE: The parser is not reentrant.

```
  ~SecParser();
```

Destroys a Secondo parser.

```
  int Text2List( const string& inputString,
                 string& outputString,
                 string& errors );
```

Parses the Secondo command in *inputString* and returns a nested list representation of the command in *outputString*. Returns an error code as follows:

- 0 – parsing was successful
- 1 – parsing was aborted due to errors
- 2 – parsing was aborted due to stack overflow

```
};

#endif
```

Anhang D

Programmdokumentation: Algebra Management

In diesem Anhang findet sich die Dokumentation der wichtigsten Module der SECONDO-Algebra-Verwaltung und ihrer Schnittstellen.

Die Dokumentation wurde mit Hilfe des PD-Systems aus den Quelltexten gewonnen. Die Kommentierung der Quelltexte erfolgte durchgehend in Englisch.

D.1 Header File: Algebra Types

May 2002 Ulrich Telle

D.1.1 Overview

This module defines several types which are important not only to algebra modules but also throughout the whole SECONDO system.

D.1.2 Imports, Types and Defines

```
#ifndef ALGEBRA_TYPES_H
#define ALGEBRA_TYPES_H

#include "NestedList.h"

#ifndef TYPE_ADDRESS_DEFINED
#define TYPE_ADDRESS_DEFINED
typedef void* Address;
#endif
```

Is the type for generic references. To use such references one need to apply an appropriate type cast.

```
union Word
{
//  Word()                    : addr( 0 )       {};
//  Word( Address  newaddr )  : addr( newaddr ) {};
//  Word( ListExpr newlist )  : list( newlist ) {};
//  Word( int      newival )  : ival( newival ) {};
//  Word( float    newrval )  : rval( newrval ) {};
```

Unfortunately C++ does not allow members with constructors in unions. Therefore some inline initialization functions (`SetWord`) are defined below.

```
  Address  addr; // generic reference
  ListExpr list; // nested list expression
  int      ival; // integer value
  float    rval; // floating point value with single precision
};
```

Specifies a generic variant type for a `Word` of memory used for SECONDO objects. To be independent of the underlying processor architecture no assumptions about the size of a `Word` should be made but all required variants should be defined as a separate variant. For each variant a constructor must be added to the list of constructors.

```
static inline Word SetWord( Address  newaddr )
                     { Word w; w.addr = newaddr; return w; };
static inline Word SetWord( ListExpr newlist )
                     { Word w; w.list = newlist; return w; };
static inline Word SetWord( int      newival )
                     { Word w; w.ival = newival; return w; };
static inline Word SetWord( float    newrval )
                     { Word w; w.rval = newrval; return w; };
```

Are several inline initialization functions for *Word* instances.

```
enum AlgebraLevel { UndefinedLevel   = 0,
                    DescriptiveLevel = 1,
                    ExecutableLevel  = 2,
                    HybridLevel      = 3 };
```

Is an enumeration of the algebra levels.

```
#endif
```

D.2 Header File: Algebra

May 2002 Ulrich Telle Port to C++

D.2.1 Overview

A snapshot of a working SECONDO system will show a collection of algebras, each of them "populated" by two different kinds of entities: objects and operators. Operators are fix in terms of number and functionality which are defined by the algebra implementor. In contrast to that, the number of objects is variable and changes dynamically at runtime. Even the types of objects are not predefined, but only their type constructors.

These very different properties of types and objects give rise to a very different C++ representation of operators and objects:

- Operators are instances of a predefined class *Operator*. Thus an implementation of an algebra with n operators contains n definitions of instances of class *Operator*.
- Objects cannot be predefined, because they are constructed and deleted at runtime. Even the possible types of objects cannot be predeclared, because they can be declared at runtime, too. Only an algebras's *type constructors* are well known and fix. An implementation of an algebra with m type constructors contains m definitions of instances of the predefined class *TypeConstructor*.

From a top level point of view, a SECONDO universe is a collection of algebras. This can be implemented by defining an instance of a subclass of the predefined class *Algebra* for each existing algebra. Each of these *Algebra* instances essentially consists of a set of operators and a set of type constructors.

D.2.2 Defines and Includes

```
#ifndef ALGEBRA_H
#define ALGEBRA_H

#include <string>
#include <vector>
#include "AlgebraManager.h"
```

D.2.3 Class $Operator$

An operator instance consists of

- a name
- at least one value mapping function, sometimes called evaluation function
- a type Mapping function, returned the operator's result type with respect to input parameters type
- a selection function calculating the index of a value mapping function with respect to input parameter types

- model and cost mapping functions (reserved for future use)

All properties of operators are set in the constructor. Only the value mapping functions have to be registered later on since their number is arbitrary. This number is set in the constructor (*noF*).

```
class Operator
{
 public:
  Operator( const string& nm,
            const string& spec,
            const int noF,
            ValueMapping vms[],
            ModelMapping mms[],
            SelectFunction sf,
            TypeMapping tm,
            CostMapping cm = Operator::DummyCost );
```

Constructs an operator with *noF* overloaded evaluation functions.

```
  Operator( const string& nm,
            const string& spec,
            ValueMapping vm,
            ModelMapping mm,
            SelectFunction sf,
            TypeMapping tm,
            CostMapping cm = Operator::DummyCost );
```

Constructs an operator with **one** evaluation functions.

```
  virtual ~Operator();
```

Destroys an operator instance.

```
  string Specification();
```

Returns the operator specification as a string.

```
  int      Select( ListExpr argtypes );
```

Returns the index of the overloaded evaluation function depending on the argument types *argtypes*.

```
  int CallValueMapping( const int index,
                        ArgVector args,
                        Word& result,
                        int message,
                        Word& local,
                        Supplier s );
```

Calls the value mapping function of the operator.

```
  Word CallModelMapping( const int index,
                         ArgVector argv,
                         Supplier s );
```

Calls the model mapping function of the operator.

```
ListExpr CallTypeMapping( ListExpr argList );
```

Calls the type mapping function of the operator.

```
ListExpr CallCostMapping( ListExpr argList );
```

Calls the cost mapping function of the operator.

```
static Word     DummyModel( ArgVector, Supplier );
```

Defines a dummy model mapping function for operators.

```
static ListExpr DummyCost( ListExpr );
```

Defines a dummy cost mapping function for operators.

```
private:
 bool AddValueMapping( const int index, ValueMapping f );
```

Adds a value mapping function to the list of overloaded operator functions.

```
 bool AddModelMapping( const int index, ModelMapping f );
```

Adds a model mapping function to the list of overloaded operator functions.

```
 string         name;           // Name of operator
 string         specString;     // Specification
 int            numOfFunctions; // No. overloaded functions
 SelectFunction selectFunc;
 ValueMapping*  valueMap; // Array of size numOfFunctions
 ModelMapping*  modelMap; // Array of size numOfFunctions
 TypeMapping    typeMap;
 CostMapping    costMap;

 friend class AlgebraManager;
};
```

D.2.4 Class $TypeConstructor$

An instance of class *TypeConstructor* consists of

- a name
- a function (“outFunc”) converting a value’s Address-representation to the corresponding nested list representation
- a function (“inFunc”) converting a value’s nested list representation to the corresponding Address-represantation
- a function (“createFunc”) allocating main memory for values which can’t be represented by a single Address (may be valid for all types in the future)

- a function (“deleteFunc”) releasing the memory previously allocated by createFunc.

All properties of an instance of class TypeConstructor are set within the constructor.

```
class TypeConstructor
{
 public:
  TypeConstructor( const string& nm,
                   TypeProperty prop,
                   OutObject out,
                   InObject in,
                   ObjectCreation create,
                   ObjectDeletion del,
                   ObjectCast ca,
                   TypeCheckFunction tcf,
                   PersistFunction pvf = 0,
                   PersistFunction pmf = 0,
                   InModelFunction inm =
                     TypeConstructor::DummyInModel,
                   OutModelFunction outm =
                     TypeConstructor::DummyOutModel,
                   ValueToModelFunction vtm =
                     TypeConstructor::DummyValueToModel,
                   ValueListToModelFunction vltm =
                     TypeConstructor::DummyValueListToModel );
```

Constructs a type constructor.

```
virtual ~TypeConstructor();
```

Destroys an instance of a type constructor.

```
void AssociateKind( const string& kindName );
```

Associates the kind *kindName* with this type constructor.

```
ListExpr Property();
```

Returns the properties of the type constructor as a nested list.

```
ListExpr Out( ListExpr type, Word value );
Word     In( const ListExpr type,
             const ListExpr value,
             const int errorPos,
             ListExpr& errorInfo,
             bool& correct );
Word     Create( int Size );
void     Delete( Word& w );

Word     InModel( ListExpr, ListExpr, int );
ListExpr OutModel( ListExpr, Word );
Word     ValueToModel( ListExpr, Word );
Word     ValueListToModel( const ListExpr typeExpr,
                           const ListExpr valueList,
                           const int errorPos,
                           ListExpr& errorInfo,
                           bool& correct );
```

Are methods to manipulate objects and models according to the type constructor.

```
bool     PersistValue( const PersistDirection dir,
                       SmiRecord& valueRecord,
                       const string& type,
                       Word& value );
bool     PersistModel( const PersistDirection dir,
                       SmiRecord& modelRecord,
                       const string& type,
                       Word& model );
bool     DefaultPersistValue( const PersistDirection dir,
                              SmiRecord& valueRecord,
                              const string& type,
                              Word& value );
bool     DefaultPersistModel( const PersistDirection dir,
                              SmiRecord& modelRecord,
                              const string& type,
                              Word& model );
```

Are methods to support persistence for objects and models according to the type constructor. The same methods are used for saving or restoring an object or model to or from its persistent representation. The direction is given by the parameter *dir*; possible values are `ReadFrom` and `WriteTo`.

An algebra implementor may choose to use a default implementation for these methods. The default methods use nested lists to represent the persistent object an model values. If a type does not need more than one `Word` of storage for representing an object value, a dummy method could be specified by the implementor.

For types like tuples or relations the default methods are not appropriate and should be overwritten. For tuples and relations meta information about the structure is needed and should be stored in the SECONDO catalog through this mechanism. The tuples and relations itself should be stored into *SmiFiles* by the algebra module.

```
static bool DummyPersistValue( const PersistDirection dir,
                               SmiRecord& valueRecord,
                               const string& type,
                               Word& value );
static bool DummyPersistModel( const PersistDirection dir,
                               SmiRecord& modelRecord,
                               const string& type,
                               Word& model );
static Word DummyInModel( ListExpr typeExpr,
                          ListExpr list,
                          int objNo );
static ListExpr DummyOutModel( ListExpr typeExpr,
                               Word model );
static Word DummyValueToModel( ListExpr typeExpr,
                               Word value );
static Word DummyValueListToModel( const ListExpr typeExpr,
                                   const ListExpr valueList,
                                   const int errorPos,
                                   ListExpr& errorInfo,
                                   bool& correct );
```

Are dummy methods used as placeholders for model manipulating type constructor functions.

```
 private:
  string                    name;   // Name of type constr.
  TypeProperty              propFunc;
  OutObject                 outFunc;
  InObject                  inFunc;
  ObjectCreation            createFunc;
  ObjectDeletion            deleteFunc;
  ObjectCast                castFunc;
  PersistFunction           persistValueFunc;
  PersistFunction           persistModelFunc;
  InModelFunction           inModelFunc;
  OutModelFunction          outModelFunc;
  ValueToModelFunction      valueToModelFunc;
  ValueListToModelFunction valueListToModelFunc;
  TypeCheckFunction         typeCheckFunc;
  vector<string>            kinds;  // Kinds of type constr.

  friend class AlgebraManager;
};
```

D.2.5 Class *Algebra*

An instance of class *Algebra* provides access to all information about a given algebra at run time, i.e. a set of type constructors and a set of operators. These properties have to be set once. A straightforward approach is to do these settings within an algebra's constructor. As all algebra modules use different type constructors and operators, all algebra constructors are different from each other. Hence we cannot use a single constructor, but request algebra implementors to derive a new subclass of class *Algebra* for each algebra module in order to provide a new constructor. Each of these subclasses will be instantiated exactly once. An algebra subclass instance serves as a handle for accessing an algebra's type constructors and operators.

```
class Algebra
{
 public:
  Algebra();
```

Creates an instance of an algebra. Concrete algebra modules are implemented as subclasses of class *Algebra*.

```
  virtual ~Algebra();
```

Destroys an algebra instance.

```
  int GetNumTCs() { return (tcs.size()); }
```

Returns the number of type constructors provided by the algebra module.

```
  int GetNumOps() { return (ops.size()); }
```

Returns the number of operators provided by the algabra module.

```
  TypeConstructor* GetTypeConstructor( int index )
                    { return (tcs[index]); }
```

Returns a reference to the type constructor identified by *index*.

```
Operator* GetOperator( int index ) { return (ops[index]); }
```

Returns a reference to the operator identified by *index*.

```
protected:
 void AddTypeConstructor( TypeConstructor* tc );
 void AddOperator( Operator* op );
```

Are used by the subclassed algebra to add its type constructors and operators to the list of type constructors and operators within the base class.

```
private:
 vector<TypeConstructor*> tcs;
 vector<Operator*> ops;

 friend class AlgebraManager;
};

#endif
```

D.3 Header File: Algebra Manager

September 1996 Claudia Freundorfer

9/26/96 RHG Slight revisions of text. Constant *MAXTYPES* introduced.

October 1996 RHG Revised Overview. Introduced types *SelectMapping*, *SelectArray*, and variable *SelectFunction* to accomodate new concept for overloading.

December 20, 1996 RHG Changed format of procedure type *OutObject*.

January 8/9, 1997 RHG Changed format of procedure type *InObject*.

May 4, 1998 RHG Type *ValueMapping* (interface of evaluation functions) changed; additional parameter *opTreeNode* of type *Supplier*.

May 15, 1998 RHG Added some generic functions for models (*InModel*, *OutModel*, *ValueToModel*.

August 25, 1998 Stefan Dieker Added generic functions for type checking and procedure *InitKinds*.

April 2002 Ulrich Telle Port to C++, complete revision

D.3.1 Overview

The SECONDO algebra manager is responsible for registering and initializing all specified algebra modules and provides interface functions for the query processor and the catalog functions to access the type constructors and operators of each registered algebra module.

The SECONDO system needs to know about the algebra modules available at run time. Therefore the developer has to specify which algebra modules are to be included in the build of the system. The list of all available algebra modules is kept in the source file `AlgebraList.i`. Each algebra can be flagged whether it should be included or not.

Currently the list of algebras to be included must be available at link time, although an algebra module may be a shared library which is loaded dynamically into memory only when the SECONDO server is started. To build an algebra modula as a shared library has the advantage that one does not need to rebuild the SECONDO system when only the implementation of the algebra modula changes.

In principal it would be possible to load algebra modules at runtime which are not known at link time. But there are at least two disadvantages:

1. it would restrict the user to command specifications in nested list form, because the code of the SECONDO parser depends on algebra specifications, which are compiled into the code, and
2. it could cause problems when such a dynamically loadable algebra module is referenced by any persistent object, but is not included in the load list any more.

Therefore this feature is currently not implemented.

The *AlgebraManager* manages a set of algebra modules. Every algebra consists of a set of types (type constructors, to be precise) and a set of operators and is defined by descriptions in SOS-Format as described in [Gü92]. A data structure and optionally a "data model" belongs to each type of the algebra. A *model* is a miniature version of the data structure itself used instead of this for cost estimations (e.g. for a relation the number of tuples, the

number of used segments, statistics for the distribution of chosen attributes could be taken into account within the model).

An algebra module offers for each type constructor up to eight generic operations:

- *AssociateKind*, *Property*. Supply the kind and the properties of a type.
- *Create*, *Delete*. Allocate/deallocate internal memory. Not needed for types represented in a single word of storage.
- *Open*, *Close*. Only needed, if the type is *independently* persistent. For example, they will be needed for relations or object classes, but not for tuples or atomic values (if we decide that the latter are not independently persistent).
- *In*, *Out*. Map a nested list into a value of the type and vice versa. Used for delivering a value to the application (among other things).

Every value of every type has a representation as a single `Word` of storage. For a simple type, this can be all. For more complex types, it may be a pointer to some structure. For persistent objects (that are currently not in memory) it can be an index into a catalog which tells where the object is on disk.

If a type (constructor) has a model, then the algebra module offers four further operations:

- *InModel*. Create a model data structure from a nested list.
- *OutModel*. Create a nested list representation from the model data structure
- *ValueToModel*. Create a model data structure from a value.
- *ValueListToModel*.

For support of persistent storage of object values and models two more operations:

- *PersistValue*. Saving and restoring an object value.
- *PersistModel*. Saving and restoring an object model.

Two default implementations (*DefaultPersistValue* and *DefaultPersistModel*) are supplied which store values and models in their nested list representation.

The algebra module offers five (sets of) functions for every operator, namely

- *Set of Evaluation Functions*. Associated with each operator is at least one, but are possibly several, evaluation functions. Each evaluation function maps a list of input values given in an array of *Word* into a result value (also a *Word*). The interface of the procedure is independent from the type of operation. Stream operators and parameter function calls are handled by calling the functions *Request*, *Open*, *Close*, *Cancel*, and *Received* of the module *QueryProcessor*.
- *Type Mapping*. This function maps a list of input types in nested list format (*ListExpr*) into an output type. Used for type checking and mapping.
- *Selection Function*. This function is given a list of argument types together with an operator number; based on these it returns the number of an evaluation function. This is used to determine for an overloaded operator the appropriate evaluation function. Usually the same function can be used for all operators of an algebra; if operators are not overloaded, identity (on the numbers) is sufficient.

- *Model Mapping*. Computes a result model from the argument models.
- *Cost Mapping*. Estimates from the models of the arguments the costs for evaluating the operation (number of CPUs, number of swapped pages). If there are no models, it returns some constant cost.

D.3.2 Defines, Includes, Constants

```
#ifndef ALGEBRA_MANAGER_H
#define ALGEBRA_MANAGER_H

#include <map>
#include "AlgebraTypes.h"
#include "NestedList.h"
#include "SecondoSMI.h"

const int MAXARG = 10;
```

Is the maximal number of arguments for one operator

```
const int OPEN    = 1;
const int REQUEST = 2;
const int CLOSE   = 3;
const int YIELD   = 4;
const int CANCEL  = 5;
```

Are constants for stream processing.

```
enum PersistDirection { ReadFrom, WriteTo };
```

Defines whether the methods managing the persistence of object values and models are persisting an object (`WriteTo`) or restoring an object (`ReadFrom`).

D.3.3 Types

```
typedef Word ArgVector[MAXARG];
typedef ArgVector* ArgVectorPointer;
```

Are the types for generic argument vectors for algebra functions.

```
typedef Address Supplier;
```

Is the type for references to a supplier of information of the operator tree.

```
typedef int (*ValueMapping)( ArgVector args, Word& result,
                             int msg, Word& local,
                             Supplier tree );
```

Is the type of an evaluation function.

```
typedef ListExpr (*TypeMapping)( ListExpr typeList );
```

Is the type of a type mapping procedure.

```
typedef TypeMapping CostMapping;
```

Is the type of a cost mapping procedure.

```
typedef int (*SelectFunction)( ListExpr typeList );
```

Is the type of a selection function.

```
typedef bool (*PersistFunction)( PersistDirection dir,
                                 SmiRecord& valueRecord,
                                 const string& type,
                                 Word& value );
```

Is the type of a function for object value and model persistence.

```
typedef Word (*ModelMapping)( ArgVector args, Supplier tree );

typedef Word (*InModelFunction)( ListExpr typeExpression,
                                 ListExpr modelList,
                                 int objectNumber );

typedef ListExpr (*OutModelFunction)( ListExpr typeExpression,
                                      Word model );

typedef Word (*ValueToModelFunction)( ListExpr typeExpression,
                                      Word value );

typedef Word (*ValueListToModelFunction)(
                const ListExpr typeExpr,
                const ListExpr valueList,
                const int errorPos,
                ListExpr& errorInfo,
                bool& correct );
```

Are the types of model mapping functions and of *in* and *out* functions for models.

```
typedef Word (*InObject)( const ListExpr numType,
                          const ListExpr valueList,
                          const int errorPos,
                          ListExpr& errorInfo,
                          bool& correct );

typedef ListExpr (*OutObject)( const ListExpr numType,
                               const Word object );

typedef Word (*ObjectCreation)( const int size );

typedef void (*ObjectDeletion)( Word& object );

typedef void* (*ObjectCast)(void*);
```

Are the types used for creating, deleting and initializing the algebra objects or components of the objects and for appending new subobjects.

This shows also the types of the generic functions for the type constructors. This is not yet satisfactory, will be revised.

```
typedef bool (*TypeCheckFunction)( const ListExpr type,
                                   ListExpr& errorInfo );
```

Is the type for type checking functions, one for each type constructor.

```
typedef ListExpr (*TypeProperty)();
```

Is the type of property functions, one for each type constructor.

D.3.4 Class *AlgebraManager*

```
struct AlgebraListEntry;
class Algebra;
class QueryProcessor;
```

Are *forward declarations* of used data structures and classes.

```
typedef Algebra*
          (*AlgebraInitFunction)( NestedList* nlRef,
                                  QueryProcessor* qpRef );
```

Is the prototype of the algebra initialization functions. For each algebra the algebra manager calls an initialization function to get a reference to the algebra and to provide the algebra with references to the global nested list container and the query processor.

```
class DynamicLibrary;

struct AlgebraListEntry
{
  AlgebraListEntry()
    : algebraId( 0 ), algebraName( "" ),
      level( UndefinedLevel ),
      algebraInit( 0 ), dynlib( 0 ), useAlgebra( false ) {}
  AlgebraListEntry( const int algId, const string& algName,
                    const AlgebraLevel algLevel,
                    const AlgebraInitFunction algInit,
                    DynamicLibrary* const dynlibInit,
                    const bool algUse )
    : algebraId( algId ),   algebraName( algName ),
      level( algLevel ),    algebraInit( algInit ),
      dynlib( dynlibInit ), useAlgebra( algUse ) {}
  int                   algebraId;
  string                algebraName;
  AlgebraLevel          level;
  AlgebraInitFunction   algebraInit;
  DynamicLibrary*       dynlib;
  bool                  useAlgebra;
};
```

Is the type for entries in the list of algebras. Each algebra has a unique identification number *algebraId*, a name *algebraName* and one of the following levels

- *Descriptive*
- *Executable*

- *Hybrid* – the algebra is *descriptive* **and** *executable*.

Additionally the address of the initialization function of the algebra is registered. For algebras to be used this forces the linker to include the algebra module from an appropriate link library. For an algebra to be loaded dynamically the address of the initialization function is not set. Instead the method *LoadAlgebras* uses the algebra name to identify and load the shared library and to identify and call the initialization function of the algebra module. A reference to the shared library is stored in the member variable *dynlib* to be able to unload the library on termination.

Finally there is a flag whether this algebra should be included in the initialization process or not.

```
typedef AlgebraListEntry& (*GetAlgebraEntryFunction)( const int j );

#define ALGEBRA_LIST_START \
static AlgebraListEntry algebraList[] = {

#define ALGEBRA_LIST_END \
  AlgebraListEntry( -1, "", UndefinedLevel, 0, 0, false ) };

#define ALGEBRA_LIST_INCLUDE(ALGNO,ALGNAME,ALGTYPE) \
 AlgebraListEntry( ALGNO, #ALGNAME,\
                   ALGTYPE##Level,\
                   &Initialize##ALGNAME, 0, true ),

#define ALGEBRA_LIST_EXCLUDE(ALGNO,ALGNAME,ALGTYPE) \
 AlgebraListEntry( ALGNO, #ALGNAME,\
                   ALGTYPE##Level, 0, 0, false ),

#define ALGEBRA_LIST_DYNAMIC(ALGNO,ALGNAME,ALGTYPE) \
 AlgebraListEntry( ALGNO, #ALGNAME,\
                   ALGTYPE##Level, 0, 0, true ),

#define ALGEBRA_PROTO_INCLUDE(ALGNO,ALGNAME,ALGTYPE) \
extern "C" Algebra* \
Initialize##ALGNAME( NestedList* nlRef,\
                     QueryProcessor* qpRef );

#define ALGEBRA_PROTO_EXCLUDE(ALGNO,ALGNAME,ALGTYPE)

#define ALGEBRA_PROTO_DYNAMIC(ALGNO,ALGNAME,ALGTYPE)
```

These preprocessor macros allow to easily define all available algebras. To start the list the macro `ALGEBRA_LIST_START` is used exactly once, and the macro `ALGEBRA_LIST_END` is used exactly once to terminate the list.

The macros `ALGEBRA_PROTO_INCLUDE`, `ALGEBRA_PROTO_EXCLUDE` and `ALGEBRA_PROTO_DYNAMIC` are used to create prototypes for the algebra initialization functions, the list entries for the algebra modules to be included or excluded from loading by the algebra manager are generated by the macros `ALGEBRA_LIST_INCLUDE`, `ALGEBRA_LIST_EXCLUDE` and `ALGEBRA_LIST_DYNAMIC`.

In the algebra list definition file `AlgebraList.i` the developer uses special versions of these macros, namely `ALGEBRA_INCLUDE`, `ALGEBRA_EXCLUDE` and `ALGEBRA_DYNAMIC`. They are expanded to the appropriate prototype and list entry macros as required. These macros have three parameters:

1. *the unique identification number* which must be a positive integer, it is recommended but not absolutely necessary to order the entries of the list in ascending order. No identification number may occur more than once in the list.

2. *the algebra name* which is used to build the name of the initialization function: the algebra name is appended to the string `Initialize`.

3. *the level of the algebra* which may be one of the following: *Descriptive*, *Executable* or *Hybrid*.

As mentioned above the list of algebras is specified in the source file `AlgebraList.i` which is included by the algebra manager source file.

Example:

```
ALGEBRA_INCLUDE(1,StandardAlgebra,Hybrid)
ALGEBRA_DYNAMIC(2,FunctionAlgebra,Executable)
ALGEBRA_EXCLUDE(3,RelationAlgebra,Hybrid)
```

This means:

- the *StandardAlgebra* will be **included** and must be available at link time. It has the id number 1 and is defined on descriptive **and** executable level.
- the *FunctionAlgebra* will be **included**, but will be loaded dynamically at runtime. It has the id number 2 and is defined on executable level only.
- the *RelationAlgebra* will be **excluded**. It has the id number 3 and is defined on descriptive **and** executable level.

```
class AlgebraManager
{
 public:
  AlgebraManager( NestedList& nlRef, GetAlgebraEntryFunction getAlgebraEnt
```

Creates an instance of the algebra manager. *nlRef* is a reference to the nested list container which should be used for nested lists.

```
virtual ~AlgebraManager();
```

Destroys an algebra manager.

```
void LoadAlgebras();
```

All existing algebras are loaded into the SECONDO programming interface. The catalog of every algebra contains the specifications of type constructors and operators of the algebra as defined above.

This procedure has to be started before the use of other functions of the database, otherwise no algebra function (operator) or type constructor can be used.

```
void UnloadAlgebras();
```

The allocated memory for the algebra catalogs is returned. No algebra function (operator) or type constructor can be used anymore.

```
bool IsAlgebraLoaded( const int algebraId );
```

Returns `true`, if the algebra module *algebraId* is loaded. Otherwise `false` is returned.

```
bool IsAlgebraLoaded( const int algebraId,
                      const AlgebraLevel level );
```

Returns `true`, if the algebra module *algebraId* is loaded and has the specified *level*. Otherwise `false` is returned.

```
int CountAlgebra();
int CountAlgebra( const AlgebraLevel level );
```

Returns the number of loaded algebras, all or only of the specified *level*.

```
bool NextAlgebraId( int& algebraId, AlgebraLevel& level );
```

Returns the identification number *algebraId* and the *level* of the next loaded algebra. On the first call *algebraId* must be initialized to zero.

```
bool NextAlgebraId( const AlgebraLevel level,
                    int& algebraId );
```

Returns the identification number *algebraId* of the next loaded algebra of the specified *level*. On the first call *algebraId* must be initialized to zero.

```
int OperatorNumber( const int algebraId );
```

Returns the number of operators of algebra *algebraId*.

```
string Ops( const int algebraId, const int operatorId );
```

Returns the name of operator *operatorId* of algebra *algebraId*.

```
ListExpr Specs( const int algebraId, const int operatorId );
```

Returns the specification of operator *operatorId* of algebra *algebraId* as a nested list expression.

```
SelectFunction
  Select( const int algebraId, const int operatorId );
```

Returns the address of the select function of operator *operatorId* of algebra *algebraId*.

```
ValueMapping
  Execute( const int algebraId, const int opFunId );
```

Returns the address of the evaluation function of the - possibly overloaded - operator *opFunId* of algebra *algebraId*.

```
ModelMapping
  TransformModel( const int algebraId, const int opFunId );
```

Returns the address of the model mapping function of the - possibly overloaded - operator *opFunId* of algebra *algebraId*.

```
TypeMapping
  TransformType( const int algebraId, const int operatorId );
```

Returns the address of the type mapping function of operator *operatorId* of algebra *algebraId*.

```
TypeMapping
  ExecuteCost( const int algebraId, const int operatorId );
```

Returns the address of the cost estimating function of operator *operatorId* of algebra *algebraId*.

```
int ConstrNumber( const int algebraId );
```

Returns the number of constructors of algebra *algebraId*.

```
string Constrs( const int algebraId, const int typeId );
```

Returns the name of type constructor *typeId* of algebra *algebraId*.

```
ListExpr Props( const int algebraId, const int typeId );
```

Returns the type properties of type constructor *typeId* of algebra *algebraId* as a nested list expression.

```
InObject InObj( const int algebraId, const int typeId );
```

Returns the address of the object input function of type constructor *typeId* of algebra *algebraId*.

```
OutObject OutObj( const int algebraId, const int typeId );
```

Returns the address of the object output function of type constructor *typeId* of algebra *algebraId*.

```
ObjectCreation
  CreateObj( const int algebraId, const int typeId );
```

Returns the address of the object creation function of type constructor *typeId* of algebra *algebraId*.

```
ObjectDeletion
  DeleteObj( const int algebraId, const int typeId );
```

Returns the address of the object deletion function of type constructor *typeId* of algebra *algebraId*.

```
ObjectCast Cast( const int algebraId, const int typeId );
```

Returns the address of the type casting function of type constructor *typeId* of algebra *algebraId*.

```
bool PersistValue( const int algebraId, const int typeId,
                   const PersistDirection dir,
                   SmiRecord& valueRecord,
                   const string& type, Word& value );
bool PersistModel( const int algebraId, const int typeId,
                   const PersistDirection dir,
                   SmiRecord& modelRecord,
                   const string& type, Word& model );
```

Save or restore object or model values for objects of type as constructed by type constructor *typeId* of algebra *algebraId*. Return `true`, if the operation was successful.

```
InModelFunction
  InModel( const int algebraId, const int typeId );
```

Returns the address of the model input function of type constructor *typeId* of algebra *algebraId*.

```
OutModelFunction
  OutModel( const int algebraId, const int typeId );
```

Returns the address of the model output function of type constructor *typeId* of algebra *algebraId*.

```
ValueToModelFunction
  ValueToModel( const int algebraId, const int typeId );
```

Returns the address of the value to model mapping function of type constructor *typeId* of algebra *algebraId*.

```
ValueListToModelFunction
  ValueListToModel( const int algebraId, const int typeId );
```

Returns the address of the value list to model mapping function of type constructor *typeId* of algebra *algebraId*.

```
TypeCheckFunction TypeCheck( const int algebraId,
                             const int typeId );
```

Returns the address of the type check function of type constructor *typeId* of algebra *algebraId*.

```
bool CheckKind( const string& kindName,
                const ListExpr type,
                ListExpr& errorInfo );
```

Checks if *type* is an element of kind *kindName*.

First parameter is the type expression to be checked. Second parameter is a list of which every element is an error message (also a list). The procedure returns `true` if the type

expression is correct. Otherwise it returns `false` and adds an error message to the list. An error message has one of two formats:

```
(60 <kind> <type expression>)
```

This means "kind *kind* does not match *type expression*". The second format is:

```
(61 <kind> <errno> ...)
```

This means "Error number *errno* in type expression for kind *kind*". The specific error numbers are defined in the kind checking procedure; the list may contain further information to describe the error.

```
 private:
  NestedList*               nl;
```

Is a referenced to a global nested list container.

```
  int                       maxAlgebraId;
```

Is the highest algebra id occuring in the list of algebras.

```
  vector<Algebra*>          algebra;
```

Is an array for references to all loaded algebra modules.

```
  vector<AlgebraLevel>      algType;
```

Is an accompanying array of level specifications for all loaded algebra modules.

```
  multimap<string,TypeCheckFunction> kindTable;
  GetAlgebraEntryFunction getAlgebraEntry;
};

#endif
```

D.4 Header File: Attribute

May 1998 Stefan Dieker

April 2002 Ulrich Telle Adjustments for the new Secondo version

D.4.1 Overview

Classes implementing attribute data types have to be subtypes of class attribute. Whatever the shape of such derived attribute classes might be, their instances can be aggregated and made persistent via instances of class *Tuple*, while the user is (almost) not aware of the additional management actions arising from persistency.

D.4.2 Class *Attribute*

The class *Attribute* defines several pure virtual methods which every derived attribute class must implement.

NOTE: Changes in the interface of the class *Attribute* might occur due to changes in the *Tuple Manager* when it is ported to the new SECONDO version.

```
#ifndef ATTRIBUTE_H
#define ATTRIBUTE_H

#include "TupleElement.h"

class Attribute : public TupleElement
{
 public:
  virtual int        Compare( Attribute *attrib ) = 0;
  virtual Attribute* Clone()     = 0;
  virtual bool       IsDefined() = 0;
  virtual int        Sizeof()    = 0;
};

#endif
```

D.5 Header File: Standard Attribute

May 1998 Stefan Dieker

December 1998 Friedhelm Becker

April 2002 Ulrich Telle Adjustments for the new Secondo version

D.5.1 Overview

The data types in the standard algebra are classes derived from the class *StandardAttribute* which defines a pure virtual method *GetValue*. The method *GetValue* is important for the delivery of object values to the SECONDO query processor.

```
#ifndef STANDARDATTRIBUTE_H
#define STANDARDATTRIBUTE_H

#include "Attribute.h"

class StandardAttribute : public Attribute
{
 public:
  virtual void* GetValue() = 0;
};

#endif
```

D.6 Header File: Standard Data Types

December 1998 Friedhelm Becker

D.6.1 Overview

This file defines four classes: CcInt, CcReal, CcBool and CcString. They are the data types which are provided by the Standardalgebra.

```
#ifndef STANDARDTYPES_H
#define STANDARDTYPES_H

#include <string>
#include "StandardAttribute.h"
```

D.6.2 CcInt

```
class CcInt : public StandardAttribute
{
 public:
  CcInt();
  CcInt( bool d, int v );
  ~CcInt();
  bool     IsDefined();
  int      GetIntval();
  void*    GetValue();
  void     Set( int v );
  void     Set( bool d, int v );
  int      Compare(Attribute * arg);
  int      Sizeof() ;
  CcInt*   Clone() ;
  ostream& Print( ostream &os ) { return (os << intval); }
 private:
  bool defined;
  int  intval;
};
```

D.6.3 CcReal

```
class CcReal : public StandardAttribute
{
 public:
  CcReal();
  CcReal( bool d, float v );
  ~CcReal();
  bool     IsDefined();
  float    GetRealval();
  void*    GetValue();
  void     Set( float v );
  void     Set( bool d, float v );
  int      Compare( Attribute* arg );
  int      Sizeof() ;
  CcReal*  Clone() ;
  ostream& Print( ostream &os ) { return (os << realval); }
```

```
 private:
  bool  defined;
  float realval;
};
```

D.6.4 CcBool

```
class CcBool : public StandardAttribute
{
 public:
  CcBool();
  CcBool( bool d, int v );
  ~CcBool();
  bool     IsDefined();
  bool     GetBoolval();
  void*    GetValue();
  void     Set( bool d, bool v );
  int      Compare( Attribute * arg );
  int      Sizeof()  ;
  CcBool*  Clone() ;
  ostream& Print( ostream &os ) { return (os << boolval); }
 private:
  bool defined;
  bool boolval;
};
```

D.6.5 CcString

```
class CcString : public StandardAttribute
{
 public:
  CcString();
  CcString( bool d, const string& v );
  ~CcString();
  bool      IsDefined();
  string*   GetStringval();
  void*     GetValue();
  void      Set( bool d, const string& v );
  int       Compare( Attribute* arg );
  int       Sizeof()  ;
  CcString* Clone() ;
  ostream&  Print( ostream &os ) { return (os << stringval); }
 private:
  bool   defined;
  string stringval;
};

#endif
```

D.7 Header File: Tuple Element

May 1998 Stefan Dieker

April 2002 Ulrich Telle Adjustments for the new Secondo version

D.7.1 Overview

The *Tuple Manager* is an important support component for the relational algebra. Relations consist of tuples, tuples consist of tuple elements. Classes implementing attribute data types have to be subtypes of class attribute. Whatever the shape of such derived attribute classes might be, their instances can be aggregated and made persistent via instances of class *Tuple*, while the user is (almost) not aware of the additional management actions arising from persistency.

D.7.2 Types

```
#ifndef TUPLE_ELEMENT_H
#define TUPLE_ELEMENT_H

#ifndef TYPE_ADDRESS_DEFINED
#define TYPE_ADDRESS_DEFINED
typedef void* Address;
#endif

#ifndef TYPE_FLOB_DEFINED
#define TYPE_FLOB_DEFINED
typedef void FLOB;
#endif
```

Are type definitions for a generic address pointer and for a *fake large object*.

D.7.3 Class $TupleElement$

This class defines several virtual methods which are essential for the *Tuple Manager*.

```
class TupleElement // renamed, previous name: TupleElem
{
 public:
  TupleElement(){};
  virtual ~TupleElement() {};
  virtual int      NumOfFLOBs() { return (0); };
  virtual FLOB*    GetFLOB( int ){ return (0); };
  virtual ostream& Print( ostream& os ) { return (os << "??"); };
};

#endif
```

Anhang E

Programmdokumentation: Storage Management Interface

In diesem Anhang findet sich die Dokumentation der wichtigsten Module des SECONDO-Speichersystems und ihrer Schnittstellen.

Die Dokumentation wurde mit Hilfe des PD-Systems aus den Quelltexten gewonnen. Die Kommentierung der Quelltexte erfolgte durchgehend in Englisch.

E.1 Header File: Storage Management Interface

April 2002 Ulrich Telle

E.1.1 Overview

The **Storage Management Interface** provides all types and classes needed for dealing with persistent data objects in SECONDO.

Essential for all operations on persistent data objects is the class *SmiEnvironment* which provides methods for startup and shutdown of the storage management, for transaction handling and for error handling.

Internally the *SmiEnvironment* handles the connection to the underlying implementation of the persistent information storage management. Currently implementations based on the BERKELEY DB library and on the ORACLE DBMS are available. The decision which implementation is used is taken at link time of the SECONDO server. Fine tuning of implementation dependent parameters is done by means of a configuration file which is read at system startup. For future extensions (i.e. user management and access control) it is possible to store the current user identification in the *SmiEnvironment*.

Additionally the *SmiEnvironment* introduces the concept of *databases*. Within one *SmiEnvironment* there may exist several databases, but a user may access only **one** database at a single time. A valid database name consists of at most `SMI_MAX_DBNAMELEN` (currently 15) alphanumeric characters or underscores. The first character has to be alphabetic. Database names are **not** case sensitive.

Persistent objects are stored as records in so called *SmiFiles*. An *SmiFile* is a handle to its representation in the interface implementation. Two kinds of *SmiFiles* are provided: one for record oriented access and one for key oriented access. An *SmiFile* always has a *context*, default or user specified. The context allows to adjust implementation dependent parameters to benefit from special knowledge about the objects to be stored in this context. These parameters are specified in the configuration file under the appropriate context section heading.

An *SmiFile* is always represented by a unique numeric identifier. Additionally it may have a name. Without a name an *SmiFile* is said to be *anonymous*. Within a context a name must be unique. *SmiFile* names and context names are case sensitive and may have at most `SMI_MAX_NAMELEN` (currently 31) alphanumeric characters or underscores. The first character has to be alphabetic.

An *SmiFile* for record oriented access is called *SmiRecordFile*. Records can be of fixed or variable size. The use of *SmiFiles* with fixed length records is recommended whereever appropriate since they may be more efficient – depending on the implementation. Records can only be appended to the end of an *SmiFile*, but can be accessed for reading, update or deletion in random order using their record number. An iterator for sequential access to all records of an *SmiFile* is provided.

An *SmiFile* for key oriented access is called *SmiKeyedFile*. An *SmiKeyFile* is capable of holding pairs of keys and data. Both keys and data may be of variable size. While the size of the data is only restricted by physical limits of the available hardware, the size of keys is restricted to at most 4000 bytes. (This limit is imposed by the VARCHAR2 datatype of ORACLE, but depending on the actual database configuration the maximal possible key length might be lower. Other relational database systems allow only much smaller keys, i.e. MYSQL restricts the combined key length to 512 bytes, but a single column key cannot exceed 255 bytes and that would be the restriction for an implementation based on MYSQL.)

As key types signed integers, floating point numbers, and strings are supported. Keys may also have a more complex structure as long as the user provides a function for mapping a key to a byte string which obeys to a lexical order. Keys may be unique or may allow for duplicate data. The use of iterators is mandatory for access to duplicates. Iterators for access to all records or records within a specified range of keys are provided.

An *SmiRecord* handle is used to access complete or partial records of an *SmiFile*.

E.1.2 Transaction handling

The storage management interface provides methods to start, commit and abort transactions. All access to persistent objects should be done within user transactions. User transactions are **not** started automatically. Due to limitations of the transaction support of the BERKELEY DB it is recommended that applications using this interface should run in a sort of auto-commit mode by surrounding very few atomic operations with calls to the start and commit transaction methods. Only application which are able to repeat transactions easily in case of failure may use more complex transactions.

Although both current implementations of the storage management interface support transaction logic for data manipulation statements, there is no support for transaction logic for data definition statements (creation and deletion of *SmiFiles*). Therefore the update of the SmiFile catalog and the deletion of *SmiFiles* are postponed until completion or rollback of a transaction. If the *SmiFile* catalog update fails, the transaction is aborted and rolled back automatically.

E.1.3 Interface methods

The class *SmiEnvironment* provides the following methods:

Environment	Transaction handling	Error handling
StartUp	BeginTransaction	CheckLastErrorCode
ShutDown	CommitTransaction	GetLastErrorCode
SetUser	AbortTransaction	SetError
CreateDatabase		
OpenDatabase		
CloseDatabase		
EraseDatabase		
ListDatabases		
IsDatabaseOpen		
CurrentDatabase		

The classes *SmiRecordFile* and *SmiKeyedFile* inherit the following methods from their base class *SmiFile*:

Creation/Removal	Open/Close	Information
Create	Open	GetContext
Drop	Close	GetName
		GetFileId
		IsOpen

The class *SmiRecordFile* provides the following methods:

Creation/Removal	Record Selection	Record Modification
SmiRecordFile	SelectRecord	AppendRecord
~SmiRecordFile	SelectAll	DeleteRecord

The class *SmiKeyedFile* provides the following methods:

Creation/Removal	Record Selection	Record Modification
SmiKeyedFile	SelectRecord	InsertRecord
~SmiKeyedFile	SelectRange	DeleteRecord
	SelectLeftRange	
	SelectRightRange	
	SelectAll	

The class *SmiRecord* provides the following methods:

Creation/Removal	Persistence	Querying
SmiRecord	Read	Size
~SmiRecord	Write	
Finish	Truncate	

The classes *SmiRecordFileIterator* and *SmiKeyedFileIterator* provide the following methods:

Creation/Removal	Access	Querying
SmiRecordFileIterator	Next	EndOfScan
~SmiRecordFileIterator	DeleteCurrent	
SmiKeyedFileIterator	Restart	
~SmiKeyedFileIterator	Finish	

The class *SmiKey* provides the following methods:

Creation/Removal	Access	Key mapping
SmiKey	GetKey	Map
~SmiKey	KeyDataType	Unmap

E.1.4 Imports, Constants, Types

```
#ifndef SECONDO_SMI_H
#define SECONDO_SMI_H

#include "SecondoConfig.h"
#include <string>

const string::size_type SMI_MAX_NAMELEN      =   31;
```

Specifies the maximum length of a context or file name.

```
const string::size_type SMI_MAX_DBNAMELEN    =   15;
```

Specifies the maximum length of a database name.

```
const string::size_type SMI_MAX_KEYLEN       = 3200;
```

Specifies the maximum length of keys.

NOTE: The maximum length of keys depends on several factors, namely the storage management system and physical properties of the underlying system:

System	Blocksize	Max. key length
Berkeley DB		4000
MySQL		255
Oracle 8i	2 kB	758
Oracle 8i	4 kB	1578
Oracle 8i	8 kB	3218
Oracle 8i	16 kB	4000

```
const string::size_type SMI_MAX_KEYLEN_LOCAL =   32;
```

Specifies the maximum length of keys which are stored locally within an instance of the *SmiKey* class. Extra memory is allocated for longer keys.

```
typedef long          SmiError;
```

Is the type for error codes of the storage management interface.

```
typedef unsigned long SmiFileId;
```

Is the type for the unique file identifiers.

```
typedef unsigned long SmiRecordId;
```

Is the type for record identifiers.

```
typedef unsigned long SmiSize;
```

Is the type for record sizes or offsets.

```
typedef void (*MapKeyFunc)( const void*    inKey,
                            const SmiSize  inLen,
                                  void*    outKey,
                            const SmiSize  maxOutLen,
                                  SmiSize& outLen,
                            const bool     doMap );
```

Is the type of user functions for mapping composite key types to strings and vice versa.

E.1.5 Class *SmiEnvironment*

This class handles all aspects of the environment of the storage environment including the basics of transactions.

```
class SmiFile;  // Forward declaration of SmiFile class

class SMI_EXPORT SmiEnvironment
{
 public:
  enum SmiType { SmiBerkeleyDB, SmiOracleDB };
```

Enumerates the different implementations of the storage management interface:

- **SmiBerkeleyDB** – Implementation based on the BERKELEY DB
- **SmiOracleDB** – Implementation based on ORACLE

```
enum RunMode { SingleUserSimple, SingleUser,
               MultiUser, MultiUserMaster };
```

Lists the types of run modes supported by the *SmiEnvironment*:

- **SingleUserSimple** – access to the *SmiEnvironment* is restricted to exactly **one** single process. Not observing this restriction can cause unpredictable behavior and possibly database corruption. Transactions and logging are usually disabled. Using this mode is not recommended, except for read-only databases.
- **SingleUser** – access to the *SmiEnvironment* is restricted to exactly **one** single process. Not observing this restriction can cause unpredictable behavior and possibly database corruption. Transactions and logging are enabled.
- **MultiUser** – the *SmiEnvironment* may be accessed by more than one process. Transactions, logging and locking are enabled.
- **MultiUserMaster** – the *SmiEnvironment* may be accessed by more than one process. Transactions, logging and locking are enabled. This mode should be used by the process which acts as the dispatcher for client requests to allow additional implementation dependent initialization.

NOTE: In any multi user mode the SECONDO registrar must be running. The behaviour of the storage management system in single user mode is implementation dependent.

```
static SmiType GetImplementationType();
```

Returns the implementation type of the storage management interface.

NOTE: This information is availabe before calling the *StartUp* method, thus allowing to perform implementation dependent activities.

```
static SmiEnvironment* GetInstance();
```

Returns a pointer to the SECONDO Storage Management Environment (seldom needed).

```
static bool StartUp( const RunMode mode,
                     const string& parmFile,
                     ostream& errStream );
```

Initializes the *SmiEnvironment* of the storage manager interface. Parameters are read from the configuration file *parmFile*. Error messages are written to the provided output stream *errStream*. A user transaction is implicitly started.

```
static bool ShutDown();
```

Shuts down the storage manager interface. An open user transaction is aborted implicitly. It is necessary to close **all** open *SmiFiles*, iterators and record handles before shutting down the system.

```
static bool IsDatabaseOpen();
```

Returns `true` if a database is currently open, otherwise `false` is returned.

```
static string CurrentDatabase();
```

Returns the name of the currently open database. If no database is open a blank string is returned.

```
static bool CreateDatabase( const string& dbname );
```

Creates a new SECONDO database under the name *dbname*. The function returns `true`, if the database could be created; it returns `false`, if a database with the given name already exists or if an error occured.

```
static bool OpenDatabase( const string& dbname );
```

Opens an existing SECONDO database having the name *dbname*. The functions returns `true`, if the database could be opened, otherwise it returns `false`.

```
static bool CloseDatabase();
```

Closes a previously created or opened SECONDO database. The function returns `true`, if the database could be closed successfully. Otherwise it returns `false`.

```
static bool EraseDatabase( const string& dbname );
```

Erases the SECONDO database named *dbname*. The function returns `true`, if the database could be erased. Otherwise it returns `false`.

NOTE: It is an error to call this function, if a database is already open, regardless of the name.

```
static bool ListDatabases( string& dbname );
```

Lists the names of existing databases, one at a time, and delivers them in *dbname*. The function returns `true` as long as there are names of database available and returns `false` after the last name has been delivered.

```
static bool SetUser( const string& uid );
```

Stores the identification *uid* of the current user in the *SmiEnvironment*. In a future extension it may be used for user management and access control.

```
static bool BeginTransaction();
static bool CommitTransaction();
static bool AbortTransaction();
```

Are provided for transaction handling. Transactions are never implicitly started. Therefore an explicit call to *BeginTransaction* is always necessary. To be able to rollback requests for deleting *SmiFiles* such requests are registered throughout the transaction and carried out only if the transaction completes successfully. *Named SmiFiles* are registered in a file catalog. Changes to this catalog take place when a transaction is committed. When updates to the file catalog fail, the transaction is implicitly aborted.

```
static SmiError CheckLastErrorCode();
static SmiError GetLastErrorCode();
static SmiError GetLastErrorCode( string& errorMessage );
```

Returns the error code of the last storage management operation. *CheckLastErrorCode* provides the error code without resetting the internal error code while the other functions reset the internal error code. Optionally the accompanying error message is returned.

```
static void SetError( const SmiError smiErr,
                      const int sysErr = 0 );
static void SetError( const SmiError smiErr,
                      const string& errMsg );
static void SetError( const SmiError smiErr,
                      const char* errMsg );
```

Allows to set an SmiError code and a system error code or an error message. (maybe these functions should not be public. Currently messages are generated only for errors occuring in the BERKELEY DB or in ORACLE.)

```
private:
 SmiEnvironment();
```

Creates an instance of the *Storage Management Interface* environment.

```
~SmiEnvironment();
```

Destroys a *Storage Management Interface* environment.

```
SmiEnvironment( SmiEnvironment& );
```

The copy constructor is not implemented.

```
static bool SetDatabaseName( const string& dbname );
```

Checks whether the given database name *dbname* is valid or not, converts the name to all lower case and stores the converted name in a member variable. The function returns `true`, if the name is valid.

A valid database name consists of at most `SMI_MAX_DBNAMELEN` (currently 15) alphanumeric characters or underscores. The first character has to be alphabetic. Database names are **not** case sensitive.

```
static bool InitializeDatabase();
```

Initializes a new database.

```
static bool RegisterDatabase( const string& dbname );
```

Registers the database *dbname* when it is created or opened. The function returns `true`, if the database could be registered successfully or if the application runs in single user mode.

NOTE: The registration is necessary to protect a database from accidental deletion by another user.

```
static bool UnregisterDatabase( const string& dbname );
```

Unregisters the database *dbname*. The function returns `true`, if the database could be unregistered successfully or if the application runs in single user mode.

```
static bool LockDatabase( const string& dbname );
```

Locks the database *dbname*. The function returns `true`, if a lock could be acquired successfully, i.e. no other user accesses the database, or if the application runs in single user mode.

NOTE: Before a database can be erased it has to be locked.

```
static bool UnlockDatabase( const string& dbname );
```

Unlocks the database *dbname*. The function returns `true`, if the lock could be released successfully or if the application runs in single user mode.

```
  static SmiEnvironment instance;    // Instance of environment
  static SmiError       lastError;   // Last error code
  static string         lastMessage; // Last error message
  static bool           smiStarted;  // Flag SMI initialized
  static bool           singleUserMode;
  static bool           useTransactions;
  static string         configFile;  // Name of config file
  static string         uid;         // ID of Secondo user
  static bool           dbOpened;    // Flag database opened
  static string         database;    // Name of current database
  static string         registrar;   // Name of the registrar
  static SmiType        smiType;     // Implementation type

  class Implementation;
  Implementation* impl;
  friend class Implementation;
  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
  friend class SmiRecord;
};
```

E.1.6 Class *SmiKey*

The class *SmiKey* is used to store key values of different types in a consistent manner. Key values are restricted in length to at most `SMI_MAX_KEYLEN` bytes. If the length of the key value is less than `SMI_MAX_KEYLEN_LOCAL` the key value ist stored within the class instance, otherwise memory is allocated.

```
class SMI_EXPORT SmiKey
{
 public:
  enum KeyDataType
    { Unknown, RecNo, Integer, Float, String, Composite };
```

Lists the types of key values supported by the *SmiFiles* for keyed access:

- **Unknown** – not a true type, designates an uninitialized key instance

- **RecNo** – a record number of a *SmiRecordFile*
- **Integer** – signed integer number (base type *long*)
- **Float** – floating point number (base type *double*)
- **String** – character string (base type *string*)
- **Composite** – user-defined key structure, the user has to provide a mapping function which is called to map the key structure to a byte string which can be sorted like a usual string in lexical order. On key retrieval the function is called to unmap the byte string to the user-defined key structure.

```
SmiKey( MapKeyFunc mapKey = 0 );
SmiKey( const SmiRecordId key );
SmiKey( const long key );
SmiKey( const double key );
SmiKey( const string& key );
SmiKey( const void* key, const SmiSize keyLen,
        MapKeyFunc mapKey );
SmiKey( SmiKey& other );
```

Creates a key with a type according to the constructor argument.

```
~SmiKey();
```

Destroys a key.

```
SmiKey& operator=( const SmiKey& other );
const bool operator>( const SmiKey& other );
const KeyDataType GetType() const;
```

Returns the type of the key.

```
bool GetKey( SmiRecordId& key );
bool GetKey( long& key );
bool GetKey( double& key );
bool GetKey( string& key );
bool GetKey( void* key, const SmiSize maxKeyLen,
             SmiSize& keyLen );
```

Returns the value of the key. The argument type must match the type of the key!

```
static void Map( const long   inData, void* outData );
static void Map( const double inData, void* outData );
static void Unmap( const void* inData, long&   outData );
static void Unmap( const void* inData, double& outData );
```

These functions are provided for convenience. They may be used in user-defined mapping functions to map integer and floating-point numbers to lexical byte strings and vice versa.

```
protected:
private:
 void  FreeData();
```

Frees the memory allocated for a key, if memory was previously allocated. The function is called internally when a new key value is assigned.

```
const void* GetAddr() const;
```

Returns the memory address of the key value.

```
void  SetKey( const SmiRecordId key );
void  SetKey( const long key );
void  SetKey( const double key );
void  SetKey( const string& key );
void  SetKey( const void* key, const SmiSize keyLen,
              MapKeyFunc mapKey );
void  SetKey( const KeyDataType kdt,
              const void* key, const SmiSize keyLen,
              MapKeyFunc mapKey = 0 );
```

Sets the internal key value to the passed key value, setting also the key type.

```
  KeyDataType keyType;   // Type of the key value
  SmiSize     keyLength; // Size of the key value in bytes
  MapKeyFunc  mapFunc;   // Address of a mapping function
  union                  // Structure for storing the key
  {
    SmiRecordId recnoKey;
    long        integerKey;
    double      floatKey;
    char        shortKeyData[SMI_MAX_KEYLEN_LOCAL+1];
    char*       longKeyData;
  };

  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
  friend class SmiKeyedFileIterator;
  friend class SmiRecord;
};
```

E.1.7 Class *SmiRecord*

This class provides a record handle for processing data records. A record contains user-defined byte strings of arbitrary length. A record handle is initialized by the appropriate methods of a *SmiFile*. After initialization the handle can be used to access complete or partial records.

```
class SMI_EXPORT SmiRecord
{
 public:
  SmiRecord();
```

Creates a record handle. A handle is passed to *SmiFile* or *SmiFileIterator* methods for initialization and can be used thereafter to access a record.

```
  ~SmiRecord();
```

Destroys a record handle.

```
SmiSize  Read( void* buffer,
               const SmiSize numberOfBytes,
               const SmiSize offset = 0 );
```

Reads a sequence of at most *numberOfBytes* bytes from the record into the *buffer* provided by the user. Optionally a record *offset* can be specified. The amount of bytes actually transfered is being returned.

```
SmiSize  Write( const void*   buffer,
                const SmiSize numberOfBytes,
                const SmiSize offset = 0 );
```

Writes a sequence of at most *numberOfBytes* bytes into the record from the *buffer* provided by the user. Optionally a record *offset* can be specified. The amount of bytes actually transfered is being returned.

```
SmiSize  Size();
```

Retrieves the total amount of bytes stored within the persistent representaion of this record.

```
bool  Truncate( const SmiSize newSize );
```

Truncates the record to a specified length of *newSize* bytes. The function returns `false` if newSize is greater than the current record's length.

```
void Finish();
```

Finishes the operation on the associated record. The record handle may be reused (i.e. reinitialized) afterwards. It is usually not necessary to call this method when the record handle is not reused, since the destructor will call it implicitly.

NOTE: If a transaction would end before the destructor is called it is essential to explicitly call this method.

```
protected:
private:
 class Implementation;
 Implementation* impl;
 SmiFile* smiFile;       // associated SmiFile object
 SmiKey   recordKey;     // Key (or record no.) of this record
 SmiSize  recordSize;    // Total size of the record
 bool     fixedSize;     // Record has fixed length
 bool     initialized;   // Handle is initialized
 bool     writable;      // Record is writable

 friend class SmiFile;
 friend class SmiFileIterator;
 friend class SmiRecordFile;
 friend class SmiRecordFileIterator;
 friend class SmiKeyedFile;
 friend class SmiKeyedFileIterator;
};
```

E.1.8 Class *SmiFile*

This class provides the methods common to both *SmiRecordFiles* and *SmiKeyedFiles*.

```
class SMI_EXPORT SmiFile
{
 public:
  enum FileType   { FixedLength, VariableLength, Keyed };
```

Is an enumeration of possible file types:

- **FixedLength** – Files of this type consist of a set of records all having a fixed size which cannot be changed. A record is filled with binary null characters if not the whole record is written. Depending on the implementation this file type allows for better locking characteristics in multi-user environments. Records are identified by record numbers.
- **VariableLength** – Files of this type consist of a set of records of potentially varying size. Records are identified by record numbers.
- **Keyed** – Files of this type consist of key/data pairs.

```
  enum AccessType { ReadOnly,    Update         };
```

Is an enumeration of possible access types:

- **ReadOnly** – Records are selected for read access only. Operations which change the contents or the size of a record are not permitted.
- **Update** – Records are selected for read and/or write access.

```
  bool Create( const string& context = "Default" );
```

Creates a new anonymous *SmiFile*. Optionally a *context* can be specified.

```
  bool Open( const SmiFileId id,
             const string& context = "Default" );
```

Opens an existing anonymous *SmiFile* using its file identifier *id*. Optionally a *context* can be specified.

```
  bool Open( const string& name,
             const string& context = "Default" );
```

Opens an existing named *SmiFile* or creates a new named *SmiFile* if it does not exist. Optionally a *context* can be specified.

```
  bool Close();
```

Closes an open *SmiFile*.

```
  bool Drop();
```

Erases a *SmiFile*. It is necessary to close any record iterators or record handles before dropping a *SmiFile*.

```
string GetContext();
```

Returns the context of the *SmiFile*.

```
string GetName();
```

Returns the name of a named *SmiFile* or an empty string for an anonymous *SmiFile*.

```
SmiFileId GetFileId();
```

Returns the unique *SmiFile* identifier.

```
bool   IsOpen();
```

Returns whether the *SmiFile* handle is open and can be used to access the records of the *SmiFile*.

```
protected:
 SmiFile();
 SmiFile( SmiFile &smiFile );
 ~SmiFile();
 bool CheckName( const string& name );
```

Checks whether the given name *name* is valid.

```
 bool        opened;               // Open state of SmiFile
 string      fileContext;          // Name of file context
 string      fileName;             // Name of named SmiFile
 SmiFileId   fileId;               // Unique file identifier

 FileType    fileType;             // Type of SmiFile records
 SmiSize     fixedRecordLength;    // Length of records with
                                   //   fixed length
 bool        uniqueKeys;           // Uniqueness of keys
 SmiKey::KeyDataType keyDataType;  // Data type of keys

 class Implementation;
 Implementation* impl;
private:

 friend class SmiFileIterator;
 friend class SmiRecord;
};
```

E.1.9 Class $SmiRecordFile$

The class *SmiRecordFile* allows record oriented access to persistent objects. New records can only be appended to a *SmiRecordFile*, but existing records can be processed in random order using their record numbers.

By means of an iterator it is possible to scan through all records of a *SmiFile*. The records are obtained in the order they were appended to the file.

```
class SmiRecordFileIterator;  // Forward declaration

class SMI_EXPORT SmiRecordFile : public SmiFile
{
 public:
  SmiRecordFile( const bool hasFixedLengthRecords,
                 const SmiSize recordLength = 0 );
```

Creates a handle for an *SmiRecordFile*. The handle is associated with an *SmiFile* by means of the *Create* or *Open* method.

```
  ~SmiRecordFile();
```

Destroys a handle of an *SmiRecordFile*.

```
  bool SelectRecord( const SmiRecordId recno,
                     SmiRecord& record,
                     const SmiFile::AccessType accessType =
                           SmiFile::ReadOnly );
```

Selects the record identified by the record number *recno* for read only or update access. The user has to provide a record handle which is initialized by this method.

```
  bool SelectAll( SmiRecordFileIterator& iterator,
                  const SmiFile::AccessType accessType =
                        SmiFile::ReadOnly );
```

Initializes an iterator for sequentially accessing all records of the *SmiFile*. The requested access type - read only or update - has to be specified.

```
  bool AppendRecord( SmiRecordId& recno,
                     SmiRecord& record );
```

Appends a new record to the *SmiFile* and returns a record handle pointing to the new record.

```
  bool DeleteRecord( const SmiRecordId recno );
```

Deletes the record identified by record number *recno* from the file.

```
 protected:
 private:
};
```

E.1.10 Class $SmiKeyedFile$

The class *SmiKeyedFile* allows key oriented access to persistent objects. Records are defined as key/data pairs. A key may have one of the types Integer, Float, String or Composit. Data may have an arbitrary length.

By means of an iterator it is possible to scan through some or all records of an *SmiKeyedFile*. The records are retrieved in ascending order of the key values. For duplicate records no special ordering is supported.

```
class SmiKeyedFileIterator;  // Forward declaration

class SMI_EXPORT SmiKeyedFile : public SmiFile
{
 public:
  SmiKeyedFile( const SmiKey::KeyDataType keyType,
                const bool hasUniqueKeys = true );
```

Creates a *SmiFile* handle for keyed access. The keys have to be of the specified type *keyType*. If *hasUniqueKeys* is true, then for each key only one record can be stored. Otherwise duplicate records are allowed.

```
  ~SmiKeyedFile();
```

Destroys a file handle.

```
  bool SelectRecord( const SmiKey& key,
                     SmiKeyedFileIterator& iterator,
                     const SmiFile::AccessType accessType =
                           SmiFile::ReadOnly );
```

Selects the data record with the given key. It is required to specify whether the select takes place for read only or update of the record. This method always initializes a record iterator regardless whether keys are unique or not. Usually this method should only be used when duplicate records are supported.

The function returns `true` when at least one record exists for the given key.

```
  bool SelectRecord( const SmiKey& key,
                     SmiRecord& record,
                     const SmiFile::AccessType accessType =
                           SmiFile::ReadOnly );
```

Selects the data record with the given key and initializes a *SmiRecord* handle for processing the record. In is required to specify whether the select takes place for read only or update of the record. In case of duplicates only the first record is returned with access type *ReadOnly*, regardless of the specified access type!

The function returns `true` when a record exists for the given key.

```
  bool SelectRange( const SmiKey& fromKey, const SmiKey& toKey,
                    SmiKeyedFileIterator& iterator,
                    const SmiFile::AccessType accessType =
                          SmiFile::ReadOnly,
                    const bool reportDuplicates = false );
```

Selects a range of records with keys for which the following condition holds $fromKey \leq key \leq toKey$. A record iterator for processing the selected records is initialized on return.

By default an iterator for read only access without reporting duplicates is initialized, but update access and reporting of duplicates may be specified.

The function returns `true` when the iterator was initialized successfully.

```
  bool SelectLeftRange( const SmiKey& toKey,
                        SmiKeyedFileIterator& iterator,
                        const SmiFile::AccessType accessType =
                              SmiFile::ReadOnly,
                        const bool reportDuplicates = false );
```

Selects a range of records with keys for which the following condition holds $key \leq toKey$. A record iterator for processing the selected records is initialized on return.

By default an iterator for read only access without reporting duplicates is initialized, but update access and reporting of duplicates may be specified.

The function returns `true` when the iterator was initialized successfully.

```
bool SelectRightRange( const SmiKey& fromKey,
                       SmiKeyedFileIterator& iterator,
                       const SmiFile::AccessType accessType =
                             SmiFile::ReadOnly,
                       const bool reportDuplicates = false );
```

Selects a range of records with keys for which the following condition holds $fromKey \leq key$. A record iterator for processing the selected records is initialized on return.

By default an iterator for read only access without reporting duplicates is initialized, but update access and reporting of duplicates may be specified.

The function returns `true` when the iterator was initialized successfully.

```
bool SelectAll( SmiKeyedFileIterator& iterator,
                const SmiFile::AccessType accessType =
                      SmiFile::ReadOnly,
                const bool reportDuplicates = false );
```

Selects all records of the associated *SmiKeyedFile*. A record iterator for processing the selected records is initialized on return.

By default an iterator for read only access without reporting duplicates is initialized, but update access and reporting of duplicates may be specified.

The function returns `true` when the iterator was initialized successfully.

```
bool InsertRecord( const SmiKey& key, SmiRecord& record );
```

Inserts a new record for the given key. An *SmiRecord* handle is initialized on return to write the record.

The function returns `true` when the record was created successfully.

```
bool DeleteRecord( const SmiKey& key );
```

Deletes all records having the given key.

The function returns `true` when the records were successfully deleted.

```
 protected:
 private:
};
```

E.1.11 Class $SmiFileIterator$

The class *SmiFileIterator* allows to scan through all records of a SmiFile. The order in which the records are retrieved depends on the type of the *SmiFile*. An *SmiFileIterator* is instantiated through an *SmiRecordFileIterator* or an *SmiKeyedFileIterator*.

```
class SMI_EXPORT SmiFileIterator
{
 public:
  bool DeleteCurrent();
```

Deletes the current record.

```
  bool EndOfScan();
```

Tells whether there are unscanned records left in the selected set of records.

```
  bool Finish();
```

Closes the iterator. The iterator handle may be reused (i.e. reinitialized) afterwards. This method should always be called as soon as access to the record set selected by the iterator is not required anymore, although the destructor will call it implicitly.

```
  bool Restart();
```

Repositions the iterator in front of the first record of the selected set of records.

```
 protected:
  bool Next( SmiRecord& record );
```

Advances the iterator to the next record of the selected set of records. A handle to the current record is returned.

NOTE: This function is never called directly by the user, but by the *Next* function of a derived class, namely *SmiRecordFileIterator* and *SmiKeyedFileIterator*.

```
  SmiFileIterator();
```

Creates a handle for an *SmiFile* iterator. The handle is associated with an *SmiFile* by means of a *Select* method of that file. Initially the iterator is positioned in front of the first record of the selected set of records.

```
  ~SmiFileIterator();
```

Destroys an *SmiFile* iterator handle.

```
  bool solelyDuplicates;
  bool ignoreDuplicates;
```

Flags whether duplicate records for a key may exist and whether those records should be reported. If *reportDuplicates* is `false`, only the first record of a set of duplicate records with equal keys is reported.

```
  SmiFile* smiFile;     // associated SmiFile object
  bool     endOfScan;   // Flag end of scan reached
  bool     opened;      // Flag iterator opened
  bool     writable;    // Flag iterator for update access
  bool     restart;     // Flag for restart
  bool     rangeSearch; // Flag for range search using
  SmiKey*  searchKey;   // Searchkey as start of range
  class Implementation;
  Implementation* impl;
 private:
};
```

E.1.12 Class *SmiRecordFileIterator*

The class *SmiRecordFileIterator* allows to scan through all records of an *SmiRecordFile*. The records are retrieved in the order as they were appended to the file.

```
class SMI_EXPORT SmiRecordFileIterator : public SmiFileIterator
{
 public:
  SmiRecordFileIterator();
```

Creates a handle for an *SmiRecordFile* iterator. The handle is associated with an *SmiFile* by means of a *Select* method of that file. Initially the iterator is positioned in front of the first record of the selected set of records.

```
  ~SmiRecordFileIterator();
```

Destroys an *SmiRecordFileIterator* handle.

```
  bool Next( SmiRecordId& recno, SmiRecord& record );
  bool Next( SmiRecord& record );
```

Advances the iterator to the next record of the selected set of records. A handle to the current record and the record number of the current record are returned.

```
 protected:
 private:
  friend class SmiRecordFile;
};
```

E.1.13 Class *SmiKeyedFileIterator*

The class *SmiKeyedFileIterator* allows to scan through some or all records of an *SmiKeyedFile*. The records are retrieved in ascending order according to the associated key values.

```
class SMI_EXPORT SmiKeyedFileIterator : public SmiFileIterator
{
 public:
  SmiKeyedFileIterator( bool reportDuplicates = false );
```

Creates a handle for a *SmiKeyedFile* iterator. The handle is associated with a *SmiFile* by means of a *Select* method of that file. Initially the iterator is positioned in front of the first record of the selected set of records.

```
  ~SmiKeyedFileIterator();
```

Destroys a *SmiKeyedFile* iterator handle.

```
  bool Next( SmiKey& key, SmiRecord& record );
  bool Next( SmiRecord& record );
```

Advances the iterator to the next record of the selected set of records. A handle to the current record – and the key of the current record, if needed – are returned.

NOTE: If the underlying *SmiKeyedFile* has keys of type *SmiKey::Composite* it is **very important** to initialize the key object *key* with the correct key mapping function, otherwise unmapping of the key does not take place. That is *key* should be created as `SmiKey key( keyMappingFunction );`.

```
 protected:
 private:
  SmiKey firstKey;       // Start of selected key range
  SmiKey lastKey;        // End of selected key range

  friend class SmiKeyedFile;
};

#endif // SECONDO_SMI_H
```

E.2 Header File: Storage Management Interface (Berkeley DB)

January 2002 Ulrich Telle

E.2.1 Overview

The **Storage Management Interface** provides all types and classes needed for dealing with persistent data objects in SECONDO. The interface itself is completely independent of the implementation. To hide the implementation from the user of the interface, the interface objects use implementation objects for representing the implementation specific parts.

The BERKELEY DB implementation of the interface uses several concepts to keep track of the SECONDO databases and their *SmiFiles*.

Since each BERKELEY DB environment needs several control processes (deadlock detection, logging, recovery) the decision was taken to use only one BERKELEY DB environment for managing an arbitrary number of SECONDO databases.

Named *SmiFiles* within each SECONDO database are managed in a simple file catalog.

SECONDO databases

In the root data directory resides a BERKELEY DB file named *databases* which contains entries for each SECONDO database. All files of a SECONDO database are stored in a subdirectory of the BERKELEY DB data directory. For each SECONDO database three BERKELEY DB files hold the information about the *SmiFile* objects:

- **sequences** – provides unique identifiers for *SmiFile* objects.
- **filecatalog** – keeps track of all **named** *SmiFile* objects. The unique identifier is used as the primary key for the entries.
- **fileindex** – represents a secondary index to the file catalog. The name of a *SmiFile* object combined with the context name is used as the primary key for the entries.

File catalog

Each *SmiFile* object - whether named or anonymous - gets a unique identifier when created. Since in a transactional environment objects get visible to other users only when the enclosing transaction completes successfully, the information about named *SmiFile* objects is queued for processing after completion of the transaction. Requests for deleting *SmiFile* objects are queued in a similar way.

File catalog updates and file deletions take place at the time a transaction completes. The actions taken depend on whether the transaction is committed or aborted. In case of a commit catalog updates and file deletions are performed as requested by the user, in case of an abort only those files which where created during the transaction are deleted, not catalog update is performed.

BERKELEY DB handles

Each *SmiFile* object has an associated BERKELEY DB handle. Unfortunately the BERKELEY DB requires that handles are kept open until after the enclosing transaction completes. Since an *SmiFile* object may go out of scope before, the destructor of the object must not close the handle. To solve the problem the storage management environment provides a container for BERKELEY DB handles. The constructor of an *SmiFile* object allocates a handle by means of the environment methods *AllocateDbHandle* and *GetDbHandle*, the destructor returns the handle by means of the environment method *FreeDbHandle*. After completion of the enclosing transaction the environment method *CloseDbHandles* closes and deallocates all BERKELEY DB handles no longer in use.

E.2.2 Implementation methods

The class *SmiEnvironment::Implementation* provides the following methods:

Database handling	Catalog handling	File handling
LookUpDatabase	LookUpCatalog	ConstructFileName
InsertDatabase	InsertIntoCatalog	GetFileId
DeleteDatabase	DeleteFromCatalog	EraseFiles
	UpdateCatalog	AllocateDbHandle
		GetDbHandle
		FreeDbHandle
		CloseDbHandles

All other implementation classes provide only data members.

E.2.3 Imports, Constants, Types

```
#ifndef SMI_BDB_H
#define SMI_BDB_H

#include <errno.h>
#include <queue>
#include <map>
#include <vector>

#include <db_cxx.h>

const u_int32_t CACHE_SIZE_STD = 128;
```

Default cache size is 128 kB

```
const u_int32_t CACHE_SIZE_MAX = 1024*1024;
```

Maximum cache size is 1 GB

These are constants which define the default and maximum cache size for the BERKELEY DB environment.

```
typedef size_t DbHandleIndex;
```

Is the type definition for indices of the BERKELEY DB handle array.

```
const DbHandleIndex DEFAULT_DBHANDLE_ALLOCATION_COUNT = 10;
```

Space for BERKELEY DB handles is reserved in chunks of DEFAULT_DBHANDLE_ALLOCATION_COUNT elements to avoid frequent memory reallocation.

```
const db_recno_t SMI_SEQUENCE_FILEID = 1;
```

Identifies the record number of the *FileId* sequence.

```
struct SmiDbHandleEntry
{
  Db*           handle;
  bool          inUse;
  DbHandleIndex nextFree;
};
```

Defines the structure of the elements of the BERKELEY DB handle array. Handles which are not *inUse* anymore are closed and freed after the completion of a transaction. The free element is put on a free list for later reuse.

```
struct SmiCatalogEntry
{
  SmiFileId fileId;
  char      fileName[2*SMI_MAX_NAMELEN+2];
  bool      isKeyed;
  bool      isFixed;
};
```

Defines the structure of the entries in the file catalog. The identifier *fileId*, the name *fileName* and the type is stored for each named *SmiFile*.

```
struct SmiDropFilesEntry
{
  SmiFileId fileId;
  bool      dropOnCommit;
};
```

Defines the structure of the elements in the queue for file drop requests. Drop requests are fulfilled on successful completion of a transaction if the flag *dropOnCommit* is set or on abortion of a transaction if this flag is **not** set. In all other cases an entry is ignored.

```
struct SmiCatalogFilesEntry
{
  SmiCatalogEntry entry;
  bool            updateOnCommit;
};
```

Defines the structure of the elements in the map for file catalog requests. Catalog requests are fulfilled on successful completion of a transaction if the flag *updateOnCommit* is set or on abortion of a transaction if this flag is **not** set. In all other cases an entry is ignored.

E.2.4 Class $SmiEnvironment :: Implementation$

This class handles all implementation specific aspects of the storage environment hiding the implementation from the user of the *SmiEnvironment* class.

```
class SmiEnvironment::Implementation
{
 public:
  static DbHandleIndex AllocateDbHandle();
```

Allocates a new BERKELEY DB handle and returns the index within the handle array.

```
  static Db*  GetDbHandle( DbHandleIndex idx );
```

Returns the BERKELEY DB handle at the position *idx* of the handle array.

```
  static void FreeDbHandle( DbHandleIndex idx );
```

Marks the BERKELEY DB handle at position *idx* as **not in use**.

```
  static void CloseDbHandles();
```

Closes all handles in the handle array which are not in use anymore.

```
  static SmiFileId GetFileId();
```

Returns a unique file identifier.

```
  static bool LookUpCatalog( const string& fileName,
                             SmiCatalogEntry& entry );
```

Looks up a file named *fileName* in the file catalog. If the file was found, the function returns `true` and the catalog entry *entry* contains information about the file like the file identifier.

```
  static bool LookUpCatalog( const SmiFileId fileId,
                             SmiCatalogEntry& entry );
```

Looks up a file identified by *fileId* in the file catalog. If the file was found, the function returns `true` and the catalog entry *entry* contains information about the file like the file name.

```
  static bool InsertIntoCatalog( const SmiCatalogEntry& entry,
                                 DbTxn* tid );
```

Inserts the catalog entry *entry* into the file catalog.

```
  static bool DeleteFromCatalog( const string& fileName,
                                 DbTxn* tid );
```

Deletes the catalog entry *entry* from the file catalog.

```
  static bool UpdateCatalog( bool onCommit );
```

Updates the file catalog on completion of a transaction by inserting or deleting entries collected during the transaction. The flag *onCommit* tells the function whether the transaction is committed (`true`) or aborted (`false`).

```
  static bool EraseFiles( bool onCommit );
```

Erases all files on completion of a transaction for which drop requests were collected during the transaction. The flag *onCommit* tells the function whether the transaction is committed (`true`) or aborted (`false`).

```
static string ConstructFileName( SmiFileId fileId );
```

Constructs a valid file name using the file identifier *fileId*.

```
static bool LookUpDatabase( const string& dbname );
```

Looks up the Secondo database *dbname* in the database catalog. The function returns `true` if a database with the given name exists.

```
static bool InsertDatabase( const string& dbname );
```

Inserts the name *dbname* of a new SECONDO database into the database catalog. The function returns `true` if the insert was successful.

```
static bool DeleteDatabase( const string& dbname );
```

Deletes the name *dbname* of an existing SECONDO database from the database catalog. The function returns `true` if the deletion was successful.

```
protected:
 Implementation();
 ~Implementation();
private:
 string  bdbHome;         // Home directory
 u_int32_t minutes;       // Time between checkpoints
 DbEnv*  bdbEnv;          // Berkeley DB environment handle
 bool    envClosed;       // Flag if environment is closed
 DbTxn*  usrTxn;          // User transaction handle
 bool    txnStarted;      // User transaction started
 Db*     bdbDatabases;    // Database Catalog handle
 Db*     bdbSeq;          // Sequence handle
 Db*     bdbCatalog;      // Database File Catalog handle
 Db*     bdbCatalogIndex; // Database Catalog Index handle

 bool    listStarted;
 Dbc*    listCursor;
```

Are needed to support listing the names of all existing SECONDO databases.

```
 queue<SmiDropFilesEntry>         bdbFilesToDrop;
 map<string,SmiCatalogFilesEntry> bdbFilesToCatalog;
 vector<SmiDbHandleEntry>         dbHandles;
 DbHandleIndex                    firstFreeDbHandle;

 friend class SmiEnvironment;
 friend class SmiFile;
 friend class SmiRecordFile;
 friend class SmiKeyedFile;
 friend class SmiRecord;
};
```

E.2.5 Class $SmiFile :: Implementation$

This class handles all implementation specific aspects of an *SmiFile* hiding the implementation from the user of the *SmiFile* class.

```
class SmiFile::Implementation
{
  public:
  protected:
    Implementation();
    ~Implementation();
  private:
    DbHandleIndex bdbHandle; // Index in handle array
    Db*           bdbFile;   // Berkeley DB handle
    bool          isSystemCatalogFile;
```

Flags an *SmiFile* as a system catalog file. This distinction is needed, since transactional read operations on system catalog files could lead easily to deadlock situations by the way the transaction and locking mechanism of the BERKELEY DB works. Therefore read operation on system catalog files should not be protected by transactions.

```
  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
  friend class SmiRecord;
};
```

E.2.6 Class $SmiFileIterator :: Implementation$

This class handles all implementation specific aspects of an *SmiFileIterator* hiding the implementation from the user of the *SmiFileIterator* class.

```
class SmiFileIterator::Implementation
{
 public:
 protected:
  Implementation();
  ~Implementation();
 private:
  Dbc* bdbCursor;  // Berkeley DB cursor

  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiRecordFileIterator;
  friend class SmiKeyedFile;
  friend class SmiKeyedFileIterator;
};
```

E.2.7 Class $SmiRecord :: Implementation$

This class handles all implementation specific aspects of an *SmiRecord* hiding the implementation from the user of the *SmiRecord* class.

```
class SmiRecord::Implementation
{
 public:
 protected:
  Implementation();
  ~Implementation();
 private:
  Db*  bdbFile;      // Berkeley DB handle
  Dbc* bdbCursor;    // Berkeley DB cursor
  bool useCursor;    // Flag use cursor in access methods
  bool closeCursor;  // Flag close cursor in destructor

  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
  friend class SmiRecord;
};

#endif
```

E.3 Header File: Storage Management Interface (Oracle DB)

April 2002 Ulrich Telle

E.3.1 Overview

The **Storage Management Interface** provides all types and classes needed for dealing with persistent data objects in SECONDO. The interface itself is completely independent of the implementation. To hide the implementation from the user of the interface, the interface objects use implementation objects for representing the implementation specific parts.

The ORACLE implementation of the interface uses several concepts to keep track of the SECONDO databases and their *SmiFiles*. Named *SmiFiles* within each SECONDO database are managed in a simple file catalog.

SECONDO databases

All SECONDO databases belong to one ORACLE user, the SECONDO manager. The connect information for this user is kept in the SECONDO configuration file. Since the password is stored therein in plain text the configuration file should be protected appropriately. The names of the ORACLE database tables storing *SmiFiles* always have the name of the SECONDO database name as a prefix. For each SECONDO database two ORACLE objects hold the information about the *SmiFile* objects:

- **SEQUENCES** – is an ORACLE sequence providing unique identifiers for *SmiFile* objects within a Secondo database.
- **TABLES** – is an ORACLE table used as a *file catalog* keeping track of all **named** *SmiFile* objects. The unique identifier is used as the primary key for the entries. The name of a *SmiFile* object combined with the context name is used as a secondary unique key for the entries.

File catalog

Each *SmiFile* object - whether named or anonymous - gets a unique identifier when created. Since in a transactional environment objects get visible to other users only when the enclosing transaction completes successfully, the information about named *SmiFile* objects is queued for processing after completion of the transaction. Requests for deleting *SmiFile* objects are queued in a similar way.

File catalog updates and file deletions take place at the time a transaction completes. The actions taken depend on whether the transaction is committed or aborted. In case of a commit catalog updates and file deletions are performed as requested by the user, in case of an abort only those files which where created during the transaction are deleted, not catalog update is performed.

E.3.2 Implementation methods

The class *SmiEnvironment::Implementation* provides the following methods:

Database handling	Catalog handling	File handling
LookUpDatabase	LookUpCatalog	ConstructTableName
DeleteDatabase	InsertIntoCatalog	ConstructSeqName
	DeleteFromCatalog	ConstructIndexName
	UpdateCatalog	GetFileId
		EraseFiles

The class *SmiFile::Implementation* provides the following methods:

Key handling
BindKeyToCursor
GetSeqId

All other implementation classes provide only data members.

E.3.3 Imports, Constants, Types

```
#ifndef SMI_ORA_H
#define SMI_ORA_H

#include <errno.h>
#include <queue>
#include <map>
#include <vector>

#include <ocicpp.h>

struct SmiCatalogEntry
{
  SmiFileId fileId;
  string    fileName;
  bool      isKeyed;
  bool      isFixed;
};
```

Defines the structure of the entries in the file catalog. The identifier *fileId*, the name *fileName* and the type is stored for each named *SmiFile*.

```
struct SmiDropFilesEntry
{
  SmiFileId fileId;
  bool      dropOnCommit;
};
```

Defines the structure of the elements in the queue for file drop requests. Drop requests are fulfilled on successful completion of a transaction if the flag *dropOnCommit* is set or on abortion of a transaction if this flag is **not** set. In all other cases an entry is ignored.

```
struct SmiCatalogFilesEntry
{
  SmiCatalogEntry entry;
  bool            updateOnCommit;
};
```

Defines the structure of the elements in the map for file catalog requests. Catalog requests are fulfilled on successful completion of a transaction if the flag *updateOnCommit* is set or on abortion of a transaction if this flag is **not** set. In all other cases an entry is ignored.

E.3.4 Class $SmiEnvironment :: Implementation$

This class handles all implementation specific aspects of the storage environment hiding the implementation from the user of the *SmiEnvironment* class.

```
class SmiEnvironment::Implementation
{
 public:
  static SmiFileId GetFileId();
```

Returns a unique file identifier.

```
  static bool LookUpCatalog( const string& fileName,
                             SmiCatalogEntry& entry );
```

Looks up a file named *fileName* in the file catalog. If the file was found, the function returns `true` and the catalog entry *entry* contains information about the file like the file identifier.

```
  static bool LookUpCatalog( const SmiFileId fileId,
                             SmiCatalogEntry& entry );
```

Looks up a file identified by *fileId* in the file catalog. If the file was found, the function returns `true` and the catalog entry *entry* contains information about the file like the file name.

```
  static bool InsertIntoCatalog( const SmiCatalogEntry& entry );
```

Inserts the catalog entry *entry* into the file catalog.

```
  static bool DeleteFromCatalog( const string& fileName );
```

Deletes the catalog entry *entry* from the file catalog.

```
  static bool UpdateCatalog( bool onCommit );
```

Updates the file catalog on completion of a transaction by inserting or deleting entries collected during the transaction. The flag *onCommit* tells the function whether the transaction is committed (`true`) or aborted (`false`).

```
  static bool EraseFiles( bool onCommit );
```

Erases all files on completion of a transaction for which drop requests were collected during the transaction. The flag *onCommit* tells the function whether the transaction is committed (`true`) or aborted (`false`).

```
  static string ConstructTableName( SmiFileId fileId );
  static string ConstructSeqName( SmiFileId fileId );
  static string ConstructIndexName( SmiFileId fileId );
```

Construct a valid table, sequence or index name using the file identifier *fileId*.

```
static bool LookUpDatabase( const string& dbname );
```

Looks up the SECONDO database *dbname* in the database catalog. The function returns `true` if a database with the given name exists.

```
static bool DeleteDatabase( const string& dbname );
```

Deletes the name *dbname* of an existing SECONDO database from the database catalog. The function returns `true` if the deletion was successful.

```
protected:
 Implementation();
 ~Implementation();
private:
 OCICPP::Connection usrConnection;
```

Is the connection for user commands. All data manipulation commands are executed using this connection.

```
OCICPP::Connection sysConnection;
```

Is the connection for system commands. All data definition commands are executed using this connection since ORACLE issues implicitly a *commit transaction* immediately before and after data definition commands.

```
ostream* msgStream;
```

Is the output stream for error messages.

```
bool     txnStarted;
```

Is a flag indicating whether a user transaction has been started.

```
bool           listStarted;
OCICPP::Cursor listCursor;
```

Are needed to support listing the names of all existing SECONDO databases.

```
queue<SmiDropFilesEntry>         oraFilesToDrop;
map<string,SmiCatalogFilesEntry> oraFilesToCatalog;

friend class SmiEnvironment;
friend class SmiFile;
friend class SmiFileIterator;
friend class SmiRecordFile;
friend class SmiKeyedFile;
friend class SmiRecord;
};
```

E.3.5 Class $SmiFile :: Implementation$

This class handles all implementation specific aspects of an *SmiFile* hiding the implementation from the user of the *SmiFile* class.

```
class SmiFile::Implementation
{
 public:
 protected:
  Implementation();
  ~Implementation();
  static bool BindKeyToCursor(
                const SmiKey::KeyDataType keyType,
                const void* keyAddr,
                const int keyLength,
                const string& bindname,
                OCICPP::Cursor& csr );
```

Binds the key values according to the key datatype to placeholders in an SQL command for a given database cursor.

```
  int GetSeqId( OCICPP::Connection& con );
```

Returns a unique record sequence number for *SmiFiles* with duplicate keys. Since the current version of the OCI C++ library does not support the `RETURNING` clause, explicit sequence numbers are the only way to identify newly inserted records with duplicate keys. Otherwise it would be possible to return the internal row identification of ORACLE for this purpose.

```
 private:
  string oraTableName;  // complete table name
  string oraSeqName;    // complete sequence name
  string oraIndexName;  // complete index name

  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
};
```

E.3.6 Class $SmiFileIterator :: Implementation$

This class handles all implementation specific aspects of an *SmiFileIterator* hiding the implementation from the user of the *SmiFileIterator* class.

```
class SmiFileIterator::Implementation
{
 public:
 protected:
  Implementation();
  ~Implementation();
 private:
  OCICPP::Cursor oraCursor;  // Oracle DB cursor

  friend class SmiFile;
```

```
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiRecordFileIterator;
  friend class SmiKeyedFile;
  friend class SmiKeyedFileIterator;
};
```

E.3.7 Class $SmiRecord :: Implementation$

This class handles all implementation specific aspects of an *SmiRecord* hiding the implementation from the user of the *SmiRecord* class.

```
class SmiRecord::Implementation
{
 public:
 protected:
  Implementation();
  ~Implementation();
 private:
  OCICPP::Lob     oraLob;      // Oracle Large Object
  OCICPP::Cursor  oraCursor;   // Oracle cursor
  bool            closeCursor; // Request to close the cursor
                               //   in the destructor

  friend class SmiFile;
  friend class SmiFileIterator;
  friend class SmiRecordFile;
  friend class SmiKeyedFile;
  friend class SmiRecord;
};

#endif
```

Anhang F

Programmdokumentation: SECONDO Tools

In diesem Anhang findet sich die Dokumentation der wichtigsten Module der betriebssystemunabhängigen SECONDO-Tools und ihrer Schnittstellen.

Die Dokumentation wurde mit Hilfe des PD-Systems aus den Quelltexten gewonnen. Die Kommentierung der Quelltexte erfolgte durchgehend in Englisch.

F.1 Header File: Compact Table

February 1994 Gerd Westerman

November 1996 RHG Revision

January 2002 Ulrich Telle Port to C++

F.1.1 Concept

A compact table is a sequence of elements of equal size indexed by natural numbers starting from 1. Whereas a list offers only sequential access, a table offers random access to all its elements.

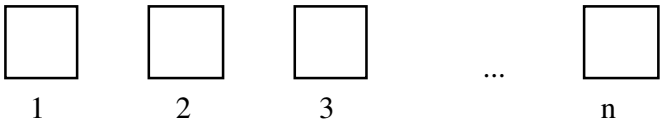

Abbildung F.1: Concept of a compact table

A compact table also provides the storage for its elements. It can be used in several ways: The first is like an array in programming languages by selecting a component by index or assigning a value to a component. The second way of using it is as a container for a set of elements retrievable by index. In that case it doesn't matter under which index an element is kept. For the latter purpose, the table maintains a record of which of its slots are filled or empty, respectively, and is also able to extend its size automatically when all slots are filled. The third way is writing and reading it sequentially, like a list.

Creation/Removal	Size info	Element access	Managing a set
CTable	Size	`[]` const	IsValid
~CTable	NoEntries	`[]`	EmptySlot
			Add
			Remove

Iterator	Scanning	Persistence
Iterator	++	Load
Begin	EndOfScan	Save
End	GetIndex	
operator==	operator*	
operator!=	operator=	

F.1.2 Imports, Types

```
#ifndef CTABLE_H
#define CTABLE_H

#include <vector>

typedef unsigned long Cardinal;
```

F.1.3 Class $CTable$

```
template <typename T>
class CTable
{
public:
```

Construction and destruction

```
CTable( Cardinal const count );
```

Creates a table with *count* slots. If the table needs to grow it does so automatically.

```
~CTable();
```

Destroys *CTable*, releasing its storage.

Size Info

```
Cardinal Size();
```

Returns the size (number of slots) of the table.

```
Cardinal NoEntries();
```

Returns the highest valid index (the largest index of a filled slot). In particular useful when the table is filled sequentially.

Accessing Elements

The following two *Select* operations get the address of a slot for reading, or writing its contents. After writing into a slot it is considered valid and changed. If used as a lvalue the slot is marked valid.

```
const T& operator[]( Cardinal n ) const;
```

"Select for reading". Returns the address of the slot with index *index*.

Precondition: $1 \leq index \leq Size(Table)$ and slot *index* must be valid.

```
T& operator[]( Cardinal n );
```

"Select for writing". Returns the address of the slot with index *index*. Makes slot *index* valid and marks it as changed.

Precondition: $1 \leq Index \leq Size(Table)$.

Managing a Set

```
bool IsValid( Cardinal const index ) const;
```

Determines whether slot *index* is valid.

Precondition: $1 \leq index \leq Size(CTable)$.

```
const Cardinal EmptySlot();
```

Returns the index of an empty slot. If necessary (because the table is full) the table is made larger before returning the index.

NOTE: An initial sequence of *EmptySlot* operations (before any *Remove* operations) returns always the empty slot with the lowest index. Hence, one can fill a table sequentially by a sequence of operations of the form

---

```
i := Empty Slot (t); t[i] := p; ... <fill entry>
```

After *Remove* operations this is not guaranteed any more (in contrast to earlier definitions of compact tables!).

```
const Cardinal Add( const T& element );
```

Copies the element referenced by *element* into some empty slot of *Table*, which may be automatically extended for this purpose, and returns the index where the element was put.

Provided for convenience; is the same as:

```
i := EmptySlot (Table); Table\[i\] := element; RETURN i;
```

An initial sequence of *Add* operations (before any *Remove* operations and without *Remove* or *EmptySlot* in between) is guaranteed to fill the table sequentially and hence maintains the order of insertions.

```
void Remove( Cardinal const index );
```

Makes slot *Index* empty (no more valid).

Precondition: $1 \leq Index \leq Size(Table)$.

F.1.4 Scan Operations

Iterators allow to scan through this *CTable* by enumerating only the valid slots in increasing order.

```
class Iterator;            // Declaration required
friend class Iterator;     // Make it a friend
class Iterator             // Definition
{
 public:
  Iterator();
```

Creates a default iterator. The iterator can't be used before an initialized iterator is assigned.

```
  Iterator( const Iterator& other );
```

Creates a copy of iterator *other*.

```
//    Iterator( const Iterator& other, bool );
      Iterator& operator++();           // Prefix ++
      const Iterator operator++( int ); // Postfix ++
```

Advances the scan to the next element. No effect if *EndOfScan* holds before. *EndOfScan* may become true.

```
      T& operator*() const;
      Iterator& operator=( const Iterator& other );
      bool operator==( const Iterator& other ) const;
```

Compares two iterators and returns *true* if they are equal.

```
      bool operator!=( const Iterator& other ) const;
```

Compares two iterators and returns *true* if they are not equal.

```
      Cardinal GetIndex() const;
```

Returns the index in the table of the current scan element. The element may then be accessed by the table operations.

```
      bool EndOfScan() const;
```

True if the scan is at position *end* (no more elements present).

```
     private:
      Iterator( CTable<T>* ctPtr );
```

Creates an iterator for the CTable referenced by *ctPtr* pointing to the first valid slot.

```
      Iterator( CTable<T>* ctPtr, bool );
```

Creates an iterator for the CTable referenced by *ctPtr* pointing beyond the highest valid slot. Such an iterator can be used to mark the end of a scan.

```
      CTable<T>* ct;          // referenced Compact Table
      Cardinal current;       // current iterator position
      friend class CTable<T>;
    };
    Iterator Begin();
```

Creates an iterator for this *CTable*, pointing to the first valid slot.

```
    Iterator End();
```

Creates an iterator for this *CTable*, pointing beyond the last valid slot.

Persistence

NOTE: The methods *Load* and *Save* for support of persistence are currently not implemented. Since the class *CTable* is based on the C++ template mechanism the type of the slots of compact table could be any C++ class. To support persistency it would be necessary to have a common base class to all classes used as template types. This would be quite limiting.

```
//  bool Load( string const fileName );
```

Loads a table from file *fileName*. If the return value is `false` something went wrong.

```
//  bool Save( string const fileName );
```

Saves the *Compact Table* to file *fileName*.

Private Members

```
private:
  vector<T>    table;      // Array of table elements
  vector<bool> valid;      // Array of table element states
  Cardinal elemCount;      // Size of compact table
  Cardinal leastFree;      // Position of free slot
  Cardinal highestValid;   // Position of highest valid slot
};
```

Inclusion of the implementation of the template class *CTable*

```
#include "CTable.cpp"

#endif
```

F.2 Header File: Display TTY

May 1998 Friedhelm Becker

December 1998 Miguel Rodríguez Luaces Ported for use in the client server version of SECONDO

May 2002 Ulrich Telle Port to new SECONDO version

F.2.1 Overview

There must be exactly one TTY display function of type *DisplayFunction* for every type constructor provided by any of the algebra modules which are loaded by the *AlgebraManager*. The first parameter is the original type expression in nested list format, the second parameter is the type expression in numeric nested list format describing the value which is going to be displayed by the display function. The third parameter is this value in nested list format.

F.2.2 Includes and Defines

```
#ifndef DISPLAY_TTY_H
#define DISPLAY_TTY_H

#include <string>
#include <map>

#include "SecondoInterface.h"
#include "NestedList.h"

typedef void (*DisplayFunction)( ListExpr type,
                                 ListExpr numType,
                                 ListExpr value );
```

Is the type definition for references to display functions.

F.2.3 Class $DisplayTTY$

This class manages the display functions in the SECONDO TTY interface. he map *displayFunctions* holds all existing display functions. It is indexed by a string consisting of the *algebraId* and the *typeId* of the corresponding type constructor.

Display functions are used to transform a nested list value into a pretty printed output in text format. Display functions which are called with a value of compound type usually call recursively the display functions of the subtypes, passing the subtype and subvalue, respectively.

```
class DisplayTTY
{
 public:
  static void Initialize( SecondoInterface* secondoInterface );
```

Initializes the display function mapping for the SECONDO interface given by *secondoInterface*.

```
static void DisplayResult( ListExpr type, ListExpr value );
```

Displays a *value* of *type* using the defined display functions. Both paramaters are given as nested lists.

```
protected:
private:
 DisplayTTY();
 ~DisplayTTY();
 static void InsertDisplayFunction( const string& name,
                                    DisplayFunction df );
```

The method *InsertDisplayFunction* inserts the display function *df* given as the second argument into the map *displayFunctions* at the index which is determined by the type constructor *name* given as first argument.

```
static void CallDisplayFunction( const ListExpr idPair,
                                 ListExpr type,
                                 ListExpr numType,
                                 ListExpr value );
```

The method *CallDisplayFunction* uses its first argument *idPair* — consisting of the two-elem-list <algebraId, typeId> — to find the right display function in the map *displayFunctions*. The arguments are simply passed to this display function.

```
static void DisplayGeneric( ListExpr type,
                            ListExpr numType,
                            ListExpr value );
```

Is a generic display function used as default for types to which not special display function was assigned:

```
static void DisplayRelation( ListExpr type,
                             ListExpr numType,
                             ListExpr value );
```

Is a display function for relations.

```
static int  MaxAttributLength( ListExpr type );
static void DisplayTuple( ListExpr type,
                          ListExpr numType,
                          ListExpr value,
                          const int maxNameLen );
static void DisplayTuples( ListExpr type,
                           ListExpr numType,
                           ListExpr value );
```

Are display functions for tuples.

```
static void DisplayInt( ListExpr type,
                        ListExpr numType,
                        ListExpr value );
static void DisplayReal( ListExpr type,
                         ListExpr numType,
                         ListExpr value );
```

```
  static void DisplayBoolean( ListExpr list,
                              ListExpr numType,
                              ListExpr value );
  static void DisplayString( ListExpr type,
                             ListExpr numType,
                             ListExpr value );
```

Are display functions for the types of the standard algebra.

```
  static void DisplayFun( ListExpr type,
                          ListExpr numType,
                          ListExpr value );
```

Is a display function for functions.

```
  static SecondoInterface* si; // Ref. to Secondo interface
  static NestedList*       nl; // Ref. to nested list container
  static map<string,DisplayFunction> displayFunctions;
};

#endif
```

F.3 Header File: Name Index

October 1995 Michael Endemann

November 1996 RHG Revision

March 2002 Ulrich Telle Port to C++ (using the *map* container of the standard C++ library)

F.3.1 Overview

A name index is conceptually a list of pairs of the form (*name*, *Cardinal*). The order of elements in the list is not specified.

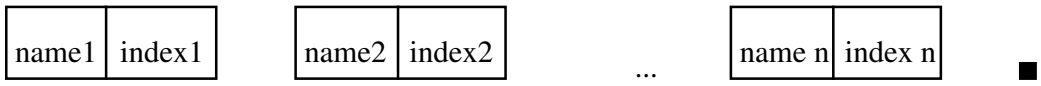

Abbildung F.2: Concept of a name index

The purpose is to allow efficient access to objects by name. The objects themselves will be kept in a (compact) table. In contrast to lists and tables, this one is not a generic structure; all components are known. The name index is implemented as a specialization of the *map* template class of the C++ standard library.

Furthermore a scan over the elements of the name index is offered by *map iterators*.

Dictionary-operations will be done in logarithmic time with this implementation.

F.3.2 Includes, Types

```
#ifndef NAME_INDEX_H
#define NAME_INDEX_H

#include <string>
#include <map>

typedef unsigned long Cardinal;
typedef map<string,Cardinal> NameIndex;

#endif
```

F.4 Header File: Nested List

Copyright (C) 1995 Gral Support Team

November 1995 Ralf Hartmut Güting

May 13, 1996 Carsten Mund

June 10, 1996 RHG Changed result type of procedure *RealValue* back to REAL.

September 24, 1996 RHG Cleaned up PD representation.

October 22, 1996 RHG Made operations *ListLength* and *WriteListExpr* available.

February 2002 Ulrich Telle Port to C++

F.4.1 Overview

A *nested list* can be viewed in two different ways. The first is to consider it as a list, which may be empty, where each element is either an *atom* or a nested list. An example of a nested list (in textual representation) is:

```
(query (select cities (fun (c city) (> (attribute c pop) 500))))
```

The structure of this list can be shown better by writing it indented:

```
(   query
    (   select
        cities
        (   fun
            (c city)
            (> (attribute c pop) 500)
        )
    )
)
```

Hence this is a list with two elements; the first is the atom "query", the second a list again. This list consists of three elements, which are the two atoms "select" and "cities", followed by another list. And so forth. Note that the structure of the list is determined only by parentheses and blanks.

The second, perhaps more implementation-oriented, view of a nested list is that it is a binary tree. Atoms are the leaves of this tree.

The nested list module described here offers a type *ListExpr* to represent such structures. A value of this type can be viewed as a pointer which can point to:

- nothing; in this case it represents an *empty list*,
- a leaf of the binary tree; it represents an *atom*,
- an internal node; it represents a *list*.

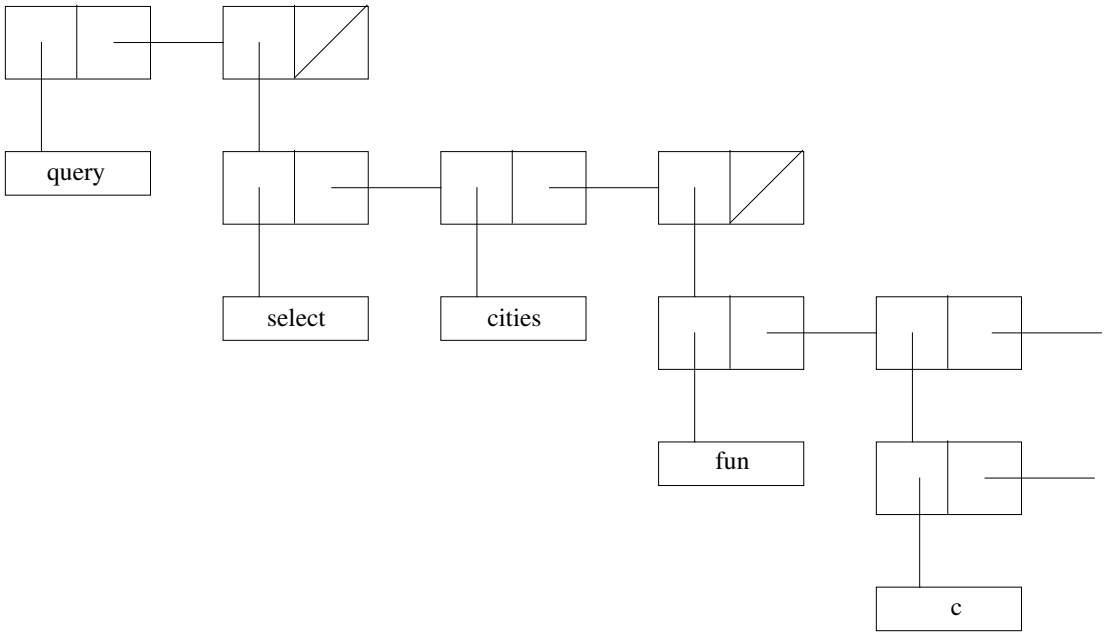

Abbildung F.3: Nested List

Atoms are typed and can be of the following types (on the right hand side example atoms are shown):

```
Integer          12, -371
Real             3.14, 62E05
Boolean          TRUE, FALSE
String           "Hello, World", "Germany"
Symbol           fun, c, <=, type
Text             <text>This is a so-called "text"!</text--->
```

The precise textual format of the various atoms is important for the representation of nested lists in files. Files can be edited by a user; furthermore it should be possible to exchange files and to read them by programs in other languages than C++. The formats are defined as follows:

For integers and reals, the conventions of C++ for the representation of constants (literals) apply. The two boolean values are represented as shown. A string value is a sequence of characters enclosed by double quotes with at most 48 characters which does not contain a double quote. A symbol value is a sequence of at most 48 characters described by the following grammar:

```
<symbol> ::= <letter> {<letter> | <digit> | <underline char>}*
           | <other char> {<other char>}*
```

Here `<other char>` is any character which is not a letter, a digit, underline or blank and is not contained in the following list of “forbidden characters”:

( ) "

Finally, a text value is any sequence of characters of arbitrary length enclosed within brackets of the form <text> and </text--->.

F.4.2 Interface methods

Nested lists are offered in a module *NestedList* offering the type *ListExpr* and the following operations:

Construction	Test	Traversal	Input/Output
TheEmptyList	IsEmpty	First	ReadFromFile
Cons	IsAtom	Rest	WriteToFile
Append	EndOfList		ReadFromString
Destroy	ListLength		WriteToString
	ExprLength		WriteListExpr
OneElemList	Equal		
TwoElemList	IsEqual	Second	
ThreeElemList		Third	
FourElemList		Fourth	
FiveElemList		Fifth	
SixElemList		Sixth	

Construction of atoms	Reading atoms	Atom test
IntAtom	IntValue	AtomType
RealAtom	RealValue	
BoolAtom	BoolValue	
StringAtom	StringValue	
SymbolAtom	SymbolValue	
TextAtom	CreateTextScan	
AppendText	GetText	
	EndOfText	

The operations are defined below.

F.4.3 Includes, Constants and Types

```
#ifndef NESTED_LIST_H
#define NESTED_LIST_H

#include <string>
#include <iostream>
#include "CTable.h"
```

Nested lists are represented by four compact tables called *nodeTable*, *intTable*, *stringTable*, and *textTable*, which are private member variables of the nested list container class *NestedList*.

```
const int INITIAL_ENTRIES = 50;
```

Specifies the default size of the compact tables. This value can be overwritten in the constructor.

```
typedef unsigned long ListExpr;
```

Is the type to represent nested lists.

```
enum NodeType
{
  NoAtom,
  IntType,
  RealType,
  BoolType,
  StringType,
  SymbolType,
  TextType
};
```

Is an enumeration of the different node types of a nested list.

```
typedef Cardinal NodesEntry;
typedef Cardinal IntsEntry;
typedef Cardinal StringsEntry;
typedef Cardinal TextsEntry;
```

Pointers into the various tables are all represented by integers; 0 is interpreted as a nil pointer.

```
struct Constant
{
  union
  {
    bool   boolValue;
    long   intValue;
    double realValue;
  };
};
```

Entries in the *intTable* table are managed by overlaying scalar variables of the appropriate scalar data type of an integer, real, or boolean value.

```
struct TextScanRecord
{
  TextsEntry currentFragment;
  Cardinal   currentPosition;
};
typedef TextScanRecord* TextScan;
```

Text entries can be of arbitrary size and are split across as many nodes of the *textTable* table as necesary. It is possible to iterate over these nodes using a text scan. A *TextScanrecord* is used to hold the state of such a scan. *currentFragment* is a pointer to a (valid) entry in the table *textTable*; *currentPos* is a pointer to a character of the current text fragment.

```
const unsigned int STRINGSIZE = 32;
typedef char StringArray [STRINGSIZE];
struct StringRecord
{
  StringArray field;
};
```

Symbols and strings with a maximum size of $3 \times STRINGSIZE$ characters are represented as at most 3 chunks of $STRINGSIZE$ characters. This approach was chosen to minimize memory consumption.

NOTE: The struct type *StringRecord* is introduced only because the vector templates used in the implementation of compact tables don't allow character arrays as the template data type.

```
const int TEXTSIZE = 64;
const Cardinal MaxFragmentLength = TEXTSIZE - sizeof(TextsEntry);

struct TextRecord
{
  TextsEntry next;
  char       field[MaxFragmentLength];
};
typedef TextRecord* Text;
```

A text entry is represented as a simple linked list of text chunks.

```
struct NodeRecord
{
  NodeType nodeType;
  union
  {
    struct                    // NoAtom
    {
      NodesEntry left;
      NodesEntry right;
      bool       isRoot;
    } n;
    struct                  // IntType, RealType, BoolType
    {
      IntsEntry index;
    } a;
    struct                  // StringType, SymbolType
    {
      StringsEntry first;
      StringsEntry second;
      StringsEntry third;
      Cardinal     strLength;
    } s;
    struct                  // TextType
    {
      TextsEntry start;
      TextsEntry last;
      Cardinal   length;
    } t;
  };
};
typedef NodeRecord* Node;
```

A *NodeRecord* represents all node types of a nested list.

Here only some of the fields need further explanation. *isRoot* is *true* after creation of an internal node; it is set to *false* when the node is used as an argument to *Cons* or as a second argument to *Append* which means this node is made the son of some other node. For a string or symbol atom, *second* and *third* may be 0 (nil pointers). For a text atom, *start* points to the first entry of the linked list of pieces of text and *last* to the last one; *length* is the total number of characters that have been entered into a text atom.

F.4.4 Class $NestedList$

```
class NestedList
{
 public:
  NestedList( int initialEntries = INITIAL_ENTRIES );
```

Creates an instance of a nested list container. The compact tables which store the nodes of nested lists reserve initially memory for holding at least *initialEntries* nodes.

```
  virtual ~NestedList();
```

Destroys a nested list container.

Construction Operations

```
  ListExpr TheEmptyList();
```

Returns a pointer to an empty list (a "nil" pointer).

```
  ListExpr Cons( const ListExpr left, const ListExpr right );
```

Creates a new node and makes *left* its left and *right* its right son. Returns a pointer to the new node.

Precondition: *right* is no atom.

```
  ListExpr Append( const ListExpr lastElem,
                   const ListExpr newSon );
```

Creates a new node *p* and makes *newSon* its left son. Sets the right son of *p* to the empty list. Makes *p* the right son of *lastElem* and returns a pointer to *p*. That means that now *p* is the last element of the list and *lastElem* the second last.... *Append* can now be called with *p* as the first argument. In this way one can build a list by a sequence of *Append* calls.

Precondition: *lastElem* is not the empty list and no atom, but is the last element of a list. That is: `EndOfList( lastElem ) == true`, `IsEmpty( lastElem ) == false`, `IsAtom( lastElem ) = false`.

Note that there are no restrictions on the element *newSon* that is appended; it may also be the empty list.

```
  void Destroy( const ListExpr list );
```

Destroys the complete subtree (including all atoms) below the root *list*.

Precondition: *list* must be the root of a list binary tree. That means, it must not have been used as an argument to *Cons* or as a second argument (*newSon*) to *Append*. This also implies that *list* is no atom and is not empty. Note that a list structure can have several roots. In such a case one has to call *Destroy* for each of the roots (and that is permitted). If *Destroy* is called only for some of the roots, an incorrect structure will result.

Test Operations

```
bool IsEmpty( const ListExpr list );
```

Returns `true` if *list* is the empty list.

```
bool IsAtom( const ListExpr list );
```

Returns `true` if *list* is an atom.

```
bool EndOfList( ListExpr list );
```

Returns `true` if *Right*(*list*) is the empty list. Returns `false` otherwise and if *list* is empty or an atom.

```
int ListLength( ListExpr list );
```

list may be any list expression. Returns the number of elements, if it is a list, and -1, if it is an atom. **Be warned:** unlike most others, this is not a constant time operation; it requires a list traversal and therefore time proportional to the length that it returns.

```
int ExprLength( ListExpr expr );
```

Reads a list expression *expr* and counts the number *length* of subexpressions.

```
bool Equal( const ListExpr list1, const ListExpr list2 );
```

Tests for deep equality of two nested lists. Returns `true` if *list1* is equivalent to *list2*, otherwise `false`.

```
bool IsEqual( const ListExpr atom, const string& str,
              const bool caseSensitive = true );
```

Returns `true` if *atom* is a symbol atom and has the same value as *str*.

Traversal

```
ListExpr First( const ListExpr list );
```

Returns (a pointer to) the left son of *list*. Result can be the empty list.

Precondition: *list* is no atom and is not empty.

```
ListExpr Rest( const ListExpr list );
```

Returns (a pointer to) the right son of *list*. Result can be the empty list.

Precondition: *list* is no atom and is not empty.

Input/Output

```
bool ReadFromFile( const string& fileName,
                   ListExpr& list );
```

Reads a nested list from file *filename* and assigns it to *list*. The format of the file must be as explained above. Returns `true` if reading was successful; otherwise `false`, if the file could not be accessed, or the line number in the file where an error occurred.

```
bool WriteToFile( const string& fileName,
                  const ListExpr list );
```

Writes the nested list *list* to file *filename*. The format of the file will be as explained above. The previous contents of the file will be lost. Returns `true` if writing was successful, `false` if the file could not be written properly.

Precondition: *list* must not be an atom.

```
bool ReadFromString( const string& nlChars,
                     ListExpr& List );
```

Like *ReadFromFile*, but reads a nested list from array *nlChars*. Returns `true` if reading was successful.

```
bool WriteToString( string& nlChars,
                    const ListExpr list );
```

Like *WriteToFile*, but writes to the string *nlChars*. Returns `true` if writing was successful, `false` if the string could not be written properly.

Precondition: *list* must not be an atom.

```
void WriteListExpr( ListExpr list, ostream& ostr );
void WriteListExpr( ListExpr list );
```

Write *list* indented by level to standard output.

Auxiliary Operations for Construction

A number of procedures is offered to construct lists with one, two, three, etc. up to six elements.

```
ListExpr OneElemList( const ListExpr elem1 );
ListExpr TwoElemList( const ListExpr elem1,
                      const ListExpr elem2 );
ListExpr ThreeElemList( const ListExpr elem1,
                        const ListExpr elem2,
                        const ListExpr elem3 );
ListExpr FourElemList( const ListExpr elem1,
                       const ListExpr elem2,
                       const ListExpr elem3,
                       const ListExpr elem4 );
ListExpr FiveElemList( const ListExpr elem1,
                       const ListExpr elem2,
                       const ListExpr elem3,
```

```
                          const ListExpr elem4,
                          const ListExpr elem5 );
ListExpr SixElemList( const ListExpr elem1,
                      const ListExpr elem2,
                      const ListExpr elem3,
                      const ListExpr elem4,
                      const ListExpr elem5,
                      const ListExpr elem6 );
```

A pointer to the new list is returned.

Auxiliary Operations for Traversal

Similarly, there are procedures to access the second, ..., sixth element. Acessing the first element is a basic operation defined above.

```
ListExpr Second( const ListExpr list );
ListExpr Third( const ListExpr list );
ListExpr Fourth( const ListExpr list );
ListExpr Fifth( const ListExpr list );
ListExpr Sixth( const ListExpr list );
```

A pointer to the respective element is returned. Result may be the empty list, of course.

Precondition: *list* must not be an atom and must have at least as many elements.

Construction of Atoms

There is a set of operations to transform each of the basic types into a corresponding atom:

```
ListExpr IntAtom( const long value );
ListExpr RealAtom( const double value );
ListExpr BoolAtom( const bool value );
ListExpr StringAtom( const string& value );
ListExpr SymbolAtom( const string& value );
```

Values of type *Text* may have arbitrary length. To construct *Text* atoms, two operations are offered:

```
ListExpr TextAtom();
void AppendText( const ListExpr atom,
                 const string&  textBuffer );
```

The first operation *TextAtom* creates the atom. Calls of *AppendText* add pieces of text stored in *textBuffer* at the end.

Precondition: *atom* must be of type *Text*.

Reading Atoms

There are corresponding procedures to get typed values from atoms:

```
long IntValue( const ListExpr atom );
```

Precondition: *atom* must be of type *Int*.

```
double RealValue( const ListExpr atom );
```

Precondition: *atom* must be of type *Real*.

```
bool BoolValue( const ListExpr atom);
```

Precondition: *atom* must be of type *Bool*.

```
string StringValue( const ListExpr atom );
```

Precondition: *atom* must be of type *String*.

```
string SymbolValue( const ListExpr atom);
```

Precondition: *atom* must be of type *Symbol*.

Again, the treatment of *Text* values is a little more difficult. To read from a *Text* atom, a *TextScan* is opened.

```
TextScan CreateTextScan( const ListExpr atom );
```

Creates a text scan. Current position is 0 (the first character in the *atom*).

Precondition: *atom* must be of type *Text*.

```
void GetText( TextScan textScan, const Cardinal noChars,
              string& textBuffer );
```

Copies *noChars* characters, starting from the current position in the *scan* and appends them to the string *textBuffer*.

The text behind the current position of the *scan* may be shorter than *noChars*. In this case, all characters behind the current *scan* position are copied.

```
bool EndOfText( const TextScan textScan );
```

Returns `true`, if the current position of the *TextScan* is behind the last character of the text.

```
void DestroyTextScan( TextScan& textScan );
```

Destroys the text scan *textScan* by deallocating the corresponding memory.

```
Cardinal TextLength( const ListExpr textAtom );
```

Returns the number of characters of *textAtom*.

Precondition: *atom* must be of type *Text*.

Atom Test

```
NodeType AtomType( const ListExpr atom );
```

Determines the type of list expression *atom* according to the enumeration type *NodeType*. If the parameter is not an atom, the function returns the value 'NoAtom'.

```
protected:
 void DestroyRecursive ( const ListExpr list );
 void PrintTableTexts();
 string NodeType2Text( NodeType type );
 string BoolToStr( const bool boolValue );
 ListExpr NthElement( const Cardinal n,
                      const Cardinal initialN,
                      const ListExpr list );
 bool WriteList( ListExpr list, const int level,
                 const bool afterList, const bool toScreen );
 void WriteAtom( const ListExpr atom, bool toScreen );
 bool WriteToStringLocal( string& nlChars, ListExpr list );
private:
 CTable<NodeRecord>   nodeTable;   // nodes
 CTable<Constant>     intTable;    // ints;
 CTable<StringRecord> stringTable; // strings
 CTable<TextRecord>   textTable  ; // texts
 ostream*             outStream;
 static bool          doDestroy;
```

The class member *doDestroy* defines whether the *Destroy* method really destroys a nested list. Only if *doDestroy* is `true`, nested lists are destroyed.

As long as the *Nested List* class does not support reference counting it might be necessary to set *doDestroy* to `false` to avoid problems due to deleting parts of nested lists which are still in use elsewhere.

```
};

#endif
```

Anhang G

Programmdokumentation: System Tools

In diesem Anhang findet sich die Dokumentation der wichtigsten Module der betriebssystemabhängigen SECONDO-Tools und ihrer Schnittstellen.

Die Dokumentation wurde mit Hilfe des PD-Systems aus den Quelltexten gewonnen. Die Kommentierung der Quelltexte erfolgte durchgehend in Englisch.

G.1 Header File: Application Management

April 2002 Ulrich Telle

G.1.1 Overview

The module *Application* provides support for managing an application process. Access to the name of the executable and the command line parameters is offered in a portable way.

When used together with the module *Processes* a very simple process communication based on a signal mechanism is available. Currently two user defined signals and a termination signal are supported. The user signals are flagged in member variables of the application and can be queried and reset at any time. For handling a termination signal two different ways are offered:

First, simple flagging is possible and the flag can be queried at any time in the event loop of the application. The flag should be checked regularly and appropriate action should be taken when it is set.

Second, the application could provide overwrite the empty virtual method *AbortOnSignal*. In this method any application specific clean up could take place. If the method returns to the calling signal handler the application is exited with a return code of **-999**.

Additionally a communication socket can be passed from a parent process to its child processes in an operating system independent manner.

Usually the application developer derives his own class from the *Application* base class. Only one instance of an application is allowed. The instantiation of the application should be the very first task in the main program, since the signal mechanism is activated in the constructor and could miss signals when not created immediately after start up of the application.

G.1.2 Interface Methods

This module offers the following routines:

Creation/Removal	Information retrieval	Process Communication
Application	GetArgCount	GetParent
~Application	GetArgValues	HasSocket
	GetApplicationName	GetSocket
	GetApplicationPath	ShouldAbort
	Instance	GetUser1Flag
	GetOwnProcessId	GetUser2Flag
		ResetUser1Flag
	Sleep	ResetUser2Flag

G.1.3 Imports, Constants, Types

```
#ifndef APPLICATION_H
#define APPLICATION_H

#include "SecondoConfig.h"
#include "SocketIO.h"
```

```
#ifndef SECONDO_PID
#define SECONDO_PID
#ifndef SECONDO_WIN32
typedef int   ProcessId;
#define INVALID_PID (-1)
#else
typedef DWORD ProcessId;
#define INVALID_PID ((DWORD)-1)
#endif
#endif
```

G.1.4 Class *Application*

This class provides an application framework.

```
class SDB_EXPORT Application
{
 public:
  Application( int argc, const char **argv );
```

Creates and initializes the *Application* object.

NOTE: The constructor is usually called directly from the program's main() routine. The parameters *argc* and *argv* are usually those passed to the main() function.

NOTE: Only exactly **one** *Application* instance is allowed per process.

```
  virtual ~Application();
```

Destroys the *Application* object.

```
  int GetArgCount() const { return (argCount); };
```

Returns the number of command line arguments.

```
  const char** GetArgValues() const { return (argValues); };
```

Returns the pointer to the array of command line arguments.

```
  const string GetApplicationName() const
  { return (appName); };
```

Returns the name of the executable file.

```
  const string GetApplicationPath() const
  { return (appPath); };
```

Returns the path name where the application was started from.

```
  const ProcessId GetOwnProcessId() { return (ownpid); };
```

Returns the real process identification of the process itself.

```
  const ProcessId GetParent() { return (parent); };
```

Returns the real process identification of the parent process, if available. If it is not available `INVALID_PID` is returned.

NOTE: Unfortunately this information is not available on all platforms. For example the operating system *Microsoft Windows* does not provide it on its own, but for child processes which are created using the *ProcessFactory* class the parent process identification is accessible.

```
static Application* Instance();
```

Returns a reference to the single *Application* instance.

```
bool ShouldAbort() const { return (abortFlag); };
```

Checks whether the abort flag was set by a signal handler. If this method returns `true`, the application should terminate as soon as possible.

```
bool GetUser1Flag() { return (user1Flag); };
bool GetUser2Flag() { return (user2Flag); };
```

Check whether one of the user flags has been set by a remote signal.

```
void ResetUser1Flag() { user1Flag = false; };
void ResetUser2Flag() { user2Flag = false; };
```

Reset the user flags to unsignaled state, thus allowing to receive further user signals. The meaning of these signals is not defined by the *Application* class.

```
bool HasSocket() { return (hasSocket); };
```

Returns `true` if a socket handle was passed to the application through the argument list.

NOTE: This is useful for communication server processes where a listener process starts a child process for servicing client requests.

```
Socket* GetSocket() { return (clientSocket); };
```

Returns a reference to the socket, which might be passed to the application through the argument list.

```
static void Sleep( const int seconds );
```

Causes the application to enter a wait state until a time interval of *seconds* seconds has expired.

The following methodes are only available to derived application classes:

```
protected:
 void SetAbortMode( bool activate ) { abortMode = activate; };
```

Activates or deactivates the abortion mode of the application.

When the abortion mode is **activated**, the method *AbortOnSignal* is invoked, when the application receives a signal on which the application should be terminated. After return from the method *AbortOnSignal* the application is terminated immediately.

When the abortion mode is **not activated**, the method *AbortOnSignal* is **not** invoked, when the application receives a signal on which the application should be terminated. Instead an abort flag is set which should be checked regularly in the event loop of the application.

```
  bool GetAbortMode() { return (abortMode); };
```

Returns the current state of the abort mode. If the abort mode is activated `true` is returned, otherwise `false`.

```
  virtual bool AbortOnSignal ( int sig ) { return (true); };
```

Is called by the application's default signal handler whenever a signal is caught that would have aborted the process anyway. This is the case for most signals like `SIGTERM`, `SIGQUIT` and so on. The pre-installed signal handler ensures proper application shutdown in such circumstances.

```
 private:
  int          argCount;      // number of arguments
  const char** argValues;     // array of arguments
  string       appName;       // name of application
  string       appPath;       // path of application
  ProcessId    ownpid;        // own process id
  ProcessId    parent;        // parent process id
  bool         hasSocket;     // flag
  Socket*      clientSocket;  // reference to client socket
  int          lastSignal;    // last signal received
  bool         abortMode;     // abort mode
  bool         abortFlag;     // abort signal flag
  bool         user1Flag;     // user1 signal flag
  bool         user2Flag;     // user2 signal flag

#ifndef SECONDO_WIN32
  static void AbortOnSignalHandler( int sig );
```

Is the default signal handler for handling signals which usually would terminate the process.

```
  static void UserSignalHandler( int sig );
```

Is the default signal handler for handling user signals (SIGUSR1 and SIGUSR2).

```
#else
  Socket* rshSocket;
  DWORD WINAPI RemoteSignalHandler();
  static DWORD WINAPI RemoteSignalThread( LPVOID app )
  {
    return (((Application*) app)->RemoteSignalHandler());
  }
  static BOOL __stdcall AbortOnSignalHandler( DWORD sig );
```

These methods emulate the signal mechanism for the *Microsoft Windows* platform.

```
#endif

  static Application* appPointer;
};

#endif // APPLICATION_H
```

G.2 Header File: Dynamic Library Management

January 2002 Ulrich Telle

G.2.1 Overview

The module *DynamicLibrary* provides support for dynamic linking, which allows a process to link libraries at run-time. Dynamically loaded libraries are very similar to ordinary shared libraries; they are usually built as standard shared libraries. The main difference is that the libraries aren't automatically loaded at program link time or start-up; instead, there is an application programming interface (API) for opening a library, looking up symbols, handling errors, and closing the library. Often it is a very useful feature for applications to be able to load and execute code from an external source at the runtime. Plugins or modules are good examples.

Unix-like operating systems, i.e. Linux and Solaris, provide this facility by offering the **dl** functions (*dlopen*, *dlsym*, *dlerror* and *dlclose*) as a mechanism for loading libraries at runtime. The Win32 API provides the functions *LoadLibrary*, *GetProcAddr* and *FreeLibrary* for managing dynamic link libraries (DLL).

Unfortunately there are considerable differences in the implementation and functionality of the dynamic loading mechanism on these platforms. While Unix-like systems usually perform a complete linking step resolving unresolved function and data references, Windows resolves only references from the loaded DLL to other DLLs. To make things worse, this is only done, when the DLL was created using appropriate *import libraries*. Otherwise it's up to the DLL itself to resolve references. One has to keep these deficiencies in mind when writing portable programs.

In most cases an application loads a dynamic library, tries to locate a function by using its known name and then to execute the code of the function to perform the requested task. Using a C function interface locating the address of a function is quite straight forward since the names seen in the C source code and seen by the linker are essentially the same. Using C++ things get more complicated since due to the name mangling the C++ compiler performs to make calling functions type-safe and to allow overloading of functions. For example the function name *init* might look like `init_int23myclassqbe_ev_abc` or something similar strange. So finding the function isn't an easy task, but there is a way out of the problem when there exists at least **one** C call protected from name mangling with `extern C` in the dynamic library that returns a pointer to a class instance. Clever as it is this *trick* solves only part of the problem since the calling program wont actually have any knowledge about the object, unless there is a common parent class from which the object class in the dynamic library is inherited. Although one is limited to calling those methods on the returned objects that had definitions in the base class, it still allows a great deal of flexibility through the use of virtual methods.

Implementing C++ classes in a DLL usually only works when used within a C++ application built with the same compiler due to name mangling and heap memory management. Beware of mixing run time libraries since one might end up with heap corruption; application and DLL should both use the same debug versus non debug run time libaries.

G.2.2 Interface methods

This module offers the following routines to manipulate dynamic libraries:

Creation/Removal	Library loading	Information retrieval
DynamicLibrary	Load	GetFunctionAddress
~DynamicLibrary	Unload	GetLibraryName
	IsLoaded	GetLastErrorMessage

G.2.3 Class *DynamicLibrary*

The class *DynamicLibrary* implements a portable wrapper to the operating system specific mechanisms to load libraries at runtime.

```
#ifndef DYNAMIC_LIBRARY_H
#define DYNAMIC_LIBRARY_H

#include "SecondoConfig.h"

class SDB_EXPORT DynamicLibrary
{
 public:
  DynamicLibrary();
```

Initializes the *DynamicLibrary* object. Once the object is constructed, use the method *Load* to dynamically link to a dynamic library while the process is running.

```
  virtual ~DynamicLibrary();
```

Destroys an instance of *DynamicLibrary* The method *Unload* is call implicitly.

```
  bool Load( const string& libraryName );
```

Loads a dynamic library *libraryName* into a process while it is running (dynamic linking). The method returns `true` on success, otherwise `false`. On failure use method *GetLastErrorMessage* to get an error message.

```
  bool Unload();
```

Unloads a dynamic library from a process while it is running. The method returns `true` on success, otherwise `false`. On failure use method *GetLastErrorMessage* to get an error message.

```
  bool IsLoaded() const;
```

Returns `true` if a dynamic library is currently loaded into the process.

```
  string GetLibraryName() const;
```

Returns the name of the currently loaded dynamic library. An empty string is returned when no library is loaded.

```
  void* GetFunctionAddress( const string& functionName );
```

Finds the function named *functionName* and returns a function pointer to it. If the function cannot be found, a null pointer is returned.

```
  string GetLastErrorMessage();
```

Returns the error message text of the last failed class method. An empty string is returned when no error occurred. The internal message buffer is emptied.

```
 protected:
#ifdef SECONDO_WIN32
  HANDLE libraryHandle; // Handle of library
#else
  void* libraryHandle;
#endif
  string libName;       // Name of currently loaded library
  string errorMessage;  // Error message text
 private:
  void SetErrorMessage();
```

Is used to create an error message when one of dynamic library system calls failed. The method *GetLastErrorMessage* returns this message to the user on request.

```
  // Do not use the following functions.
  DynamicLibrary( const DynamicLibrary& other );
  int operator==( const DynamicLibrary& other ) const;
  DynamicLibrary& operator=( const DynamicLibrary& other);
};

#endif // DYNAMIC_LIBRARY_H
```

G.3 Header File: File System Management

May 2002 Ulrich Telle

G.3.1 Overview

The *File System Management* provides several services for handling files and folders (directories) in an operating system independent manner. There are functions for inspecting and manipulating the folder (directory) tree. No functions for file access are provided.

G.3.2 Interface methods

The class *SmiEnvironment* provides the following methods:

Folder Management	Files and Folders	File management
GetCurrentFolder	FileOrFolderExists	CopyFile
SetCurrentFolder	RenameFileOrFolder	GetFileAttributes
CreateFolder	DeleteFileOrFolder	SetFileAttributes
EraseFolder	FileSearch	
	AppendSlash	

G.3.3 Imports, Constants, Types

```
#ifndef FILESYSTEM_H
#define FILESYSTEM_H 1

#include "SecondoConfig.h"
#include <string>
#include <vector>

#ifdef SECONDO_WIN32
typedef DWORD FileAttributes;
#else
typedef uint32_t FileAttributes;
#endif
```

Is the type for the file attributes for a specific file.

NOTE: The values of the file attributes are operating system dependent. One has to keep this in mind when implementing portable applications.

```
typedef vector<string> FilenameList;
```

Is the type for a collection of filenames found by a File Search.

```
typedef bool (*FileSearchCallbackFunc)
          ( const string& absolutePath,
            const string& fileName,
            FileAttributes attribs );
```

Is the type of user-supplied functions for filename filtering. The function arguments are:

- *absolutePath* – Directory where file resides.
- *fileName* – Filename without directory.
- *attribs* – File attributes.

G.3.4 Class $FileSystem$

This class implements all functions for the file system management as static members. Since the constructor is private the class cannot be instantiated.

```
class FileSystem
{
 public:
  static string GetCurrentFolder();
```

Returns the current folder (directory).

```
  static bool SetCurrentFolder( const string& folder );
```

Sets the current folder (directory) to *folder*. The function returns `true`, if the current folder could be set.

```
  static bool CreateFolder( const string& folder );
```

Creates the folder (directory) located at *folder*. The function returns `true`, if the folder could be created.

```
  static bool DeleteFileOrFolder( const string& fileName );
```

Deletes the file or folder (directory) specified in *fileName*. The function returns `true`, if the file or folder could be deleted.

The function fails if the file is protected by file attributes or if the folder to be removed contains one or more files.

```
  static bool EraseFolder( const string& folder,
                           uint16_t maxLevels = 16 );
```

Removes the folder (directory) specified in *folder*. The function returns `true`, if the remove operation succeeded.

This function makes every attempt to delete the folder (directory), such as removing file protection attributes and files contained within folders (directories).

maxLevels controls how many levels of subfolders the function will traverse in order to remove a folder (directory).

NOTE: When *EraseFolder* deletes multiple files, files will be deleted until an error occurs. Any files that were successfully deleted before the error occurred will not be restored. This situation typically occurs if the user does not have permission to remove a file.

```
  static bool RenameFileOrFolder( const string& currentName,
                                  const string& newName );
```

Renames (moves) a file or folder (directory) from *currentName* to *newName*. The function returns `true`, if the copy operation succeeded.

```
static bool CopyFile( const string& source,
                                const string& dest );
```

Copies a file from *source* to *dest*. The function returns `true`, if the copy operation succeeded.

NOTE: On Unix systems this function may be used to copy folders (directories) as well. Keep in mind that this property is not portable.

```
static bool FileOrFolderExists( const string& fileName );
```

Checks for the existence of the file indicated by *fileName*. The function returns `true`, if the file exists.

```
static FileAttributes GetFileAttributes( const string&
                                                fileName );
```

Returns the file attributes for the file *fileName*. In case of an error the function returns 0.

```
static bool SetFileAttributes( const string& fileName,
                                FileAttributes attribs );
```

Sets the file attributes for a file to the values specified in *attribs* . The function returns `true`, if the attributes could be set.

```
static bool FileSearch( const string& folder,
                        FilenameList& filenameList,
                        const string* searchName = 0,
                        uint16_t maxLevels = 1,
                        bool includeFolders = true,
                        bool fullPath = true,
                        FileSearchCallbackFunc
                          fileSearchCallback = 0 );
```

Returns a list of filenames which meet the search criteria in *filenameList*. *folder* indicates the folder (directory) where the search begins.

If *searchName* is specified, the list will only contain the files matching this name. Wildcard searches are currently **not** supported, but the callback function *fileSearchCallback* can be used to filter the filenames.

maxLevels controls how many levels of subdirectories will be searched.

includeFolders specifies whether subfolder names are to be included in the list of filenames.

If *fullPath* is `true`, the complete pathname of each file will be returned.

```
static bool SearchPath( const string& fileName, string& foundFile );
```

Searches the file *fileName* on the path and returns `true` if the file was found, otherweise `false`. If the file was found the complete pathname of the file is returned in *foundFile*.

```
static void AppendSlash( string& pathName );
```

Appends the proper slash character to a pathname. This character will either be a forward or backward slash, depending on the operating system used.

```
 protected:
 private:
#ifdef SECONDO_WIN32
  static void UnprotectFile( const string& fileName );
```

Removes file protection attributes from a file, so that it may be modified or deleted.

NOTE: This function is available in Windows only.

```
#endif
```

The following functions are not implemented and must never be used.

```
  FileSystem() {};
  ~FileSystem() {};
  FileSystem( const FileSystem& other );
  FileSystem& operator=( const FileSystem& other );
  int operator==( const FileSystem& other ) const;
};

#endif // FILESYSTEM_H
```

G.4 Header File: Messenger

May 2002 Ulrich Telle

G.4.1 Overview

The SECONDO server consists of several processes which need to communicate with each other. The communication is based on local sockets. Quite often only short messages need to be exchanged between two processes. Examples are registering access to a SECONDO database, locking a database or writing log messages. In these cases the class *Messenger* provides an easy to use interface to communicate with a message queue server.

G.4.2 Class $Messenger$

This class implements a simple mechanism for interprocess communication with a message queue server. The method *Send* connects to the message queue server, sends a simple string message and waits for an answer. After receiving the answer string, it disconnects from the message queue server.

```
#ifndef MESSENGER_H
#define MESSENGER_H

#include <string>

class Messenger
{
 public:
  Messenger( const string& queueName )
    : msgQueue( queueName ) {};
```

Creates a messenger instance for communication with the message queue *queueName*.

```
  virtual ~Messenger() {};
```

Destroys a messenger instance.

```
  bool Send( const string& message, string& answer );
```

Send the *message* string to the message queue and wait for the *answer* string. If the communication was successful, the method returns `true`.

```
 protected:
 private:
  string msgQueue; // Name of the message queue
};

#endif
```

G.5 Header File: Process Management

April 2002 Ulrich Telle

G.5.1 Overview

The module *Processes* provides support for creating and managing child processes in an operating system independent manner. Processes are created using a process factory which is capable of managing a collection of a user specified number of processes. After spawning child processes the parent process can interact with the child process by a signal mechanism. Currently two user signals and a termination signal are supported for child processes which use the *Application* class for implementing the signal mechanism. The parent process may wait for completion of one or all of his child processes. After termination of a child process its exit code can be accessed. After inspecting the exit code of child process its entry in the process collection may be reused by the process factory. On startup of the process factory the application may allow to reuse entries of terminated processes which exit code was not inspected.

Additionally support for passing a communication socket to a child process is provided. This is useful when a server process listens on a port for client connections (through the method *Accept* of the socket module) and spawns a child process for servicing each client.

The process factory takes care of any necessary clean up actions after termination of a child process, especially when client sockets for network communication are involved.

G.5.2 Interface Methods

This module consists of two classes: the *Process* class and the *ProcessFactory* class. Usually the *Process* class should not be used directly by an application. The class *ProcessFactory* offers the following routines:

Process factory	Process handling	Process information
StartUp	SpawnProcess	IsProcessOk
ShutDown	WaitForProcess	IsProcessTerminated
GetInstance	WaitForAll	GetExitCode
	SignalProcess	GetRealProcessId
	SignalRealProcess	
	Sleep	

G.5.3 Imports, Constants, Types

```
#ifndef PROCESSES_H
#define PROCESSES_H

#include <string>
#include <vector>
#include "SecondoConfig.h"
#include "SocketIO.h"

#ifndef SECONDO_WIN32
#include <cstdlib>
#include <cstring>
```

```
#include <cstdio>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <libgen.h>
#endif

#define DEFAULT_MAX_PROCESSES 10
```

Is the default size of the process collection of the process factory;

```
#ifndef SECONDO_PID
#define SECONDO_PID
#ifndef SECONDO_WIN32
typedef int   ProcessId;
#define INVALID_PID (-1)
#else
typedef DWORD ProcessId;
#define INVALID_PID ((DWORD)-1)
#endif
#endif
```

Is the type definition for process identifiers.

```
enum ProcessSignal { eSIGTERM, eSIGUSR1, eSIGUSR2 };
```

Is an enumeration of supported signals.

The following signals are currently supported:

- *eSIGTERM* – is a request for the signaled process to terminate itself.
- *eSIGUSR1* – is a request for the signaled process to execute the first of two possible user specified actions.
- *eSIGUSR1* – is a request for the signaled process to execute the second of two possible user specified actions.

```
class ProcessFactory;
```

Forward declaration of class *ProcessFactory*

G.5.4 Class $Process$

```
class Process
{
 public:
  Process();
```

Constructs a process administration instance.

```
  ~Process();
```

Destroys a process administration instance.

```
bool SendSignal( const ProcessSignal signo = eSIGTERM );
```

Sends the signal *signo* to the associated process. In case of success `true` is returned, otherwise `false`.

```
bool WaitForTermination();
```

Waits for the termination of the associated process. The methode returns `true`, if a termination signal was received. In case of an error `false` is returned.

```
void Finish();
```

Cleans up a reserved, but terminated process administration instance. This function is used by the process factory to reclaim entries in the process collection of terminated processes, if the application has not checked the exit code of a terminated process but has allowed the reuse of such entries.

```
Process( const Process& other );
Process& operator=( Process const &other );
```

A copy constructor and an assignment operator are only defined since they are required by the standard class *vector* which is used to manage a collection of processes in the process factory. These methods should not be used.

```
 protected:
  bool      reserved;
  int       cycle;
  bool      terminated;
  int       exitStatus;

#ifdef SECONDO_WIN32
  bool      hasSocket;
  Socket*   clientSocket;
  SOCKET    inheritableSocket;
  HANDLE    event;
  bool      notMonitored;
  PROCESS_INFORMATION processInfo;

  void ActivateWaiter();
  DWORD WINAPI Waiter();
  static DWORD WINAPI WaiterThread( LPVOID p )
  { return (((Process*) p)->Waiter()); }
#else
  pid_t         pid;
#endif
  friend class ProcessFactory;
};
```

G.5.5 Class $ProcessFactory$

This class provides methods to manage a collection of subprocesses. After spawning a new process it is possible to wait for completion of the process and to check its exit code. Simple means to communicate with the process are availabe through a signal mechanism.

```
class ProcessFactory
{
 public:
  static bool StartUp( const bool reuseTerminated = true,
                       const int  maxChildProcesses =
                                    DEFAULT_MAX_PROCESSES );
```

Initializes the process factory. The flag *reuseTerminated* controls whether entries in the internal child process table may be reused after termination of the child process but before the parent process has checked the exit code of the child process. If the parent process is not interested in the exit codes, the flag should be set to `true`.

The parameter *maxChildProcesses* defines the size of the internal child process table, i.e. the maximal number of concurrent child processes.

```
  static bool ShutDown();
```

Shuts down the process factory.

```
  static bool SpawnProcess( const string& programpath,
                            const string& arguments,
                            int& processId,
                            const bool hidden = true,
                            Socket* clientSocket = 0 );
```

Spawns a process. The program specified by *programpath* will be started as a separate process and receives the *arguments* string as its command line. The internal process identifier is returned as *processId*.

If the flag *hidden* is set (which it is by default), the process is started as a background process; if the flag is **not** set, the process is started as a foreground process, if that is possible.

To support communication on client sockets across process boundaries a reference to a socket *clientSocket* can be specified and is transfered to the newly created process. Client sockets are usually created by the *Accept* method of the socket module.

```
  static ProcessId GetRealProcessId( const int processId );
```

Returns the operating system dependent process identifier. If the process does not exist or is already terminated, the value `INVALID_PID` is returned.

NOTE: In situations where an application needs to send signals to processes it did not spawn itself access to the real process identifier is necessary. One should **not** use this identifier for manipulating a process directly since this could interfer with this class and could cause unpredictable results.

```
  static bool SignalProcess( const int processId,
                             const ProcessSignal signo =
                                     eSIGTERM );
```

Sends the specified signal *signo* to the process *processId*, if that process is still running. In case of success `true` is returned, otherwise `false`.

```
  static bool SignalRealProcess( const ProcessId processId,
                                 const ProcessSignal signo =
                                         eSIGTERM );
```

Sends the specified signal *signo* to the process *processId*, if that process is still running. In case of success `true` is returned, otherwise `false`.

```
static bool GetExitCode( const int processId, int& status );
```

Provides access to the exit code *status* of the process *processId*. The method returns `true`, if the process has already terminated.

```
static bool IsProcessOk( const int processId );
```

Checks whether the process *processId* exists in the process collection. The method returns `true` if the process exists and is still running or is in terminated state, otherwise `false` is returned.

```
static bool IsProcessTerminated( const int processId );
```

Checks whether the process *processId* is in terminated state. If the process is terminated `true` is returned, otherwise `false`. An application should check both *IsProcessOk* **and** *IsProcessTerminated* to detect an error condition.

```
static bool WaitForProcess( const int processId );
```

Waits for the termination of process *processId*. In case the process terminated `true` is returned, in case of an error `false` is returned.

```
static bool WaitForAll();
```

Waits for the termination **all** processes under control of the process factory. The method returns `true` if all processes have terminated; in case of an error `false` is returned.

```
static void Sleep( const int seconds );
```

Causes the application to enter a wait state until a time interval of *seconds* seconds has expired.

```
ProcessFactory* GetInstance() { return (instance); }
```

Returns a reference to the single instance of the process factory.

```
protected:
 ProcessFactory( const bool reuseTerminated = true,
                 const int maxChildProcesses =
                              DEFAULT_MAX_PROCESSES );
 virtual ~ProcessFactory();
private:
 ProcessFactory( ProcessFactory& );

 static ProcessFactory* instance;
 vector<Process> processList;
 int  maxChilds;
 bool reuseTerminatedEntries;
#ifndef SECONDO_WIN32
 static void ChildTerminationHandler( int sig );
#endif
};

#endif
```

G.6 Header File: Profiles

January 2002 Ulrich Telle

G.6.1 Overview

Applications often need a mechanism for providing configuration parameters. One way to supply those parameters is a profile. A profile is a text file consisting of one ore more named sections which contain one or more named parameters and their values, respectively. Each line in the profile is either a section heading, a key/value pair or a comment. A section heading is enclosed in square brackets; key name and key value are separated by an equal sign (leading and trailing whitespace is removed from both key name and value, but intervening blanks are considered to be part of the name or the value); a comment line starts with semicolon. All Text in a profile has to be left aligned. Example:

```
[section 1]
;   comment
key 1=value 1
key 2=value 2

[section 2]
;   comment
key 1=value 1
....     (and so on)
```

G.6.2 Interface methods

This module offers routines to get or set a parameter:

Routines
GetParameter
SetParameter

G.6.3 Class $SmiProfile$

The class *SmiProfile* implements routines to manipulate profile strings stored in a text file:

```
#ifndef SMI_PROFILES_H
#define SMI_PROFILES_H

#include "SecondoConfig.h"
#include <string>

class SMI_EXPORT SmiProfile
{
 public:
  static string GetParameter( const string& sectionName,
                              const string& keyName,
```

```
                                    const string& defaultValue,
                                    const string& fileName );
```

Searches the profile *fileName* for the key *keyName* under the section heading *sectionName*. If found, the associated string is returned, else the default value is returned.

```
    static long   GetParameter( const string& sectionName,
                                const string& keyName,
                                long          defaultValue,
                                const string& fileName );
```

Searches the profile *fileName* for the key *keyName* under the section heading *sectionName*. The function returns

- *zero* – if the key value is not an integer,
- *actual value* – if it is possible to interpret the key value as an integer (Note that only the beginning of the key value is interpreted, i.e. `key=345abc` returns `345`)
- *default value* – if key or section not found

```
    static bool   SetParameter( const string& sectionName,
                                const string& keyName,
                                const string& keyValue,
                                const string& fileName );
```

Searches the profile *fileName* for the section heading *sectionName* and the key *keyName*. If found the value is changed to *keyValue*, otherwise it will be added.

The first special case is when *keyValue* is an empty string. Then the corresponding line is deleted.

The second special case is when both *keyValue* and *keyName* are empty strings. In that case the whole section *sectionName* is deleted. Lines beginning with ';' are however kept since they are considered comments.

```
};

#endif // SMI_PROFILES_H
```

G.7 Header File: Socket I/O

February 2002 Ulrich Telle

G.7.1 Overview

TCP (Transaction Control Protocol) is one of the most common protocols on the Internet. It has two basic communicators: clients and servers. Servers offer services to the network; clients connect to these services through ports. Sockets are the basis of network communication. Client software uses sockets to connect to servers. Servers use sockets to process network connections. TCP sockets provide very reliable connections using a special open communication flow similar to low-level stream I/O.

Implementing clients and servers are fairly straightforward. Servers build a socket for listening on a port for incoming client connect requests. Clients also build a socket, connect to the server socket and then exchange messages.

The set of classes defined here supports a subset of the TCP/IP protocol suite. Currently only IP version 4 is supported, but support for IP version 6 could be incorporated. The code is partly based on the socket module of the *System Abstraction Layer* C++ library written by Konstantin Knizhnik (http://www.ispras.ru/~knizhnik/), but was completely rewritten to support for instance I/O streams on sockets.

Implementations of sockets are mostly based on socket libraries provided by the operating system. Local domain sockets are directly supported only in Unix. For 32-bit-Windows this socket module provides a very efficient implementation of local sockets using shared memory and semaphore objects. This allows very fast inter-process communication on one computer.

NOTE: Different operating systems require different initialization and deinitialization of the socket interface. This is done transparently by instantiating a static instance of an internal class hidden in the implementation. The constructor is called before the main function is entered, thereby ensuring that the operating systems's socket interfaces is properly installed. The destructor is called on normal termination of the program.

G.7.2 Interface methods

The class *Socket* is the heart of this module. Network communication is done by using methods of this class. Additionally to creating and destroying sockets, reading and writing on these sockets there are methods to identify the client and the server thus allowing a basic form of authentication and access control through the classes *SocketRule* and *SocketRuleSet*. It is possible to specify a list of rules where each rule consists of an IP address, an IP address mask and an access policy allowing or denying access. When an IP address is checked against a set of rules, for each rule a bitwise **and** operation is performed on the given IP address and the IP address mask of a rule. The result is compared with the IP address of the rule. If the result matches, the decision whether access should be allowed or denied is based on the access policy of the rule and the default access policy of the rule set. Finally, the class *SocketAddress* hides details of the internet addresses from the user.

The class *Socket* provides the following methods:

Creation/Removal	Input/Output	Information
Socket	Read	IsLibraryInitialized
~Socket	Write	IsOk
CreateLocal	GetSocketStream	GetErrorText
CreateGlobal		GetSocketAddress
Accept	CancelAccept	GetPeerAddress
CreateClient	Close	GetHostname
Connect	ShutDown	GetIP
GetDescriptor		

The class *SocketRule* provides the following methods:

Creation/Removal	Checking	I/O settings
SocketRule	Match	SetDelimiter
~SocketRule	Allowed	GetDelimiter
	Denied	

The class *SocketRuleSet* provides the following methods:

Creation/Removal	Checking	Persistence
SocketRuleSet	AddRule	LoadFromFile
~SocketRuleSet	Ok	StoreToFile

The class *SocketAddress* provides the following methods:

Creation/Removal	Address handling
SocketAddress	SetAddress
~SocketAddress	GetSocketString
operator=	GetIPAddress
	GetPort
	operator==

The class *SocketBuffer* implements the **streambuf** interface of the C++ standard library for socket streams. This greatly simplifies reading and writing to and from sockets. Since this class is used internally only, the class interface is not described here.

```
#ifndef SOCKET_IO_H
#define SOCKET_IO_H
```

G.7.3 Imports and definitions

```
#include "SecondoConfig.h"
#include <time.h>
#include <iostream>
#include <vector>

#ifdef SECONDO_WIN32
#include <winsock2.h>
#else
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#endif

#define DEFAULT_CONNECT_MAX_ATTEMPTS 100
```

Defines the default for how many attempts are made at most to connect a client socket to a server socket.

```
#define DEFAULT_RECONNECT_TIMEOUT    1  // seconds
```

Defines the default time interval between two connect attempts.

```
#define DEFAULT_LISTEN_QUEUE_SIZE    5
```

Defines the default capacity of the listener queue.

```
#define LINGER_TIME                  10 // seconds
```

Defines how long the kernel should try to send data still waiting in the socket buffer after the socket was closed.

```
#define WAIT_FOREVER                 ((time_t)-1)
```

Specifies an indefinite time period. Specifying this value for a time out period is identical to blocking I/O mode.

```
#define ENABLE_BROADCAST             1
```

Defines the flag for enabling broadcast messages on UDP sockets.

```
#ifdef SECONDO_WIN32
#include <winsock2.h>
typedef SOCKET SocketDescriptor;
#else
typedef int    SocketDescriptor;
#define INVALID_SOCKET (-1)
#endif
```

Is a type definition for a socket handle. If a socket is not valid, the handle value equals `INVALID_SOCKET`.

G.7.4 Class *Socket*

The class *Socket* defines an abstract socket interface. The concrete operating system dependent socket classes are implemented as derived classes.

Since *Socket* is an abstract base class, instances of *Socket* are created through the use of special static *factory* methods.

NOTE: Since signals present portability problems, this code does not support signals (like SIGIO, etc.), which is why there is no interface to the *fcntl* function.

```
class SocketBuffer;
```

Is a forward declaration of the stream buffer class for socket stream support.

```
class SDB_EXPORT Socket {
 public:
  enum SocketDomain
  {
    SockAnyDomain,
    SockLocalDomain,
    SockGlobalDomain
  };
```

Is an enumeration of domain types:

- *SockAnyDomain* – the domain type is chosen automatically
- *SockLocalDomain* – the domain type is local (i.e. Unix domain sockets)
- *SockGlobalDomain* – the domain type is global (i.e. INET sockets)

```
static bool IsLibraryInitialized();
```

Checks whether the operating socket interface was successfully initialized.

```
Socket() { state = SS_CLOSE; }
```

Initializes a *Socket* instance as an invalid socket in closed state.

```
virtual ~Socket() {}
```

Destroys a socket.

```
static Socket*  Connect( const string& address,
                         const string& port,
                         const SocketDomain domain =
                           SockAnyDomain,
                         const int maxAttempts =
                           DEFAULT_CONNECT_MAX_ATTEMPTS,
                         const time_t timeout =
                           DEFAULT_RECONNECT_TIMEOUT );
```

Establishes a connection to a server. This method will do at most *maxAttempts* attempts to connect to the server, with an interval of *timeout* seconds between the attempts. The address of the server is specified by *address* and *port*, both as strings. The type of the connection is specified by *domain*. The following values of this parameter are recognized:

- *SockAnyDomain* – the domain is chosen automatically
- *SockLocalDomain* – local domain (connection with a server on the same host)
- *SockGlobalDomain* – internet domain

If *SockAnyDomain* is specified, a local connection is chosen when either the port was omitted in the specification of the address or hostname is **localhost**; a global connection is used in all other cases.

This method always creates a new socket object and returns a pointer to it. If a connection to the server was not established, this socket contains an error code describing reason of failure. So the returned socket should be first checked by its *IsOk* method.

```
static Socket*  CreateLocal( const string& address,
                             const int listenQueueSize =
                               DEFAULT_LISTEN_QUEUE_SIZE);
```

Creates and opens a socket in the local domain at the server side. The parameter *address* specifies the name to be assigned to the socket. The parameter *listenQueueSize* specifies the size of the listen queue.

This method always creates a new socket object and returns a pointer to it. If a connection to the server was not established, this socket contains an error code describing reason of failure. So the returned socket should be first checked by its *IsOk* method.

```
static Socket*  CreateGlobal( const string& address,
                              const string& port,
                              const int listenQueueSize =
                                DEFAULT_LISTEN_QUEUE_SIZE );
```

Creates and opens a socket in the global (internet) domain at the server side. The parameter *address* specifies the name to be assigned to the socket. The parameter *listenQueueSize* specifies the size of the listen queue.

This method always creates a new socket object and returns a pointer to it. If a connection to the server was not established, this socket contains an error code describing reason of failure. So the returned socket should be first checked by its *IsOk* method.

```
virtual SocketDescriptor GetDescriptor() = 0;
```

Returns the socket descriptor of the socket. This socket descriptor may be inherited by a child process later on.

```
static Socket*  CreateClient( const SocketDescriptor sd );
```

(Re)creates a socket instance for the socket descriptor *sd*, created by the *Accept* method.

This method is provided to allow passing socket descriptors to child processes.

NOTE: While in Unix-like systems socket handles are inherited by child processes automatically, this is not the case in Windows systems. Usually you have to take special measure to make a socket handle inheritable and to make it accessible by a child process. The parent process must not close the socket before the child process has finished using the socket.

```
virtual int     Read( void* buf,
                      size_t minSize, size_t maxSize,
                      time_t timeout = WAIT_FOREVER ) = 0;
```

Receives incoming data, transferring it from the socket into the buffer *buf*. The maximal size of the buffer is given by *maxSize*. The function returns the number of bytes actually read from the socket, which ranges from *minSize* to *maxSize*. The function does not return to the caller before at least *minSize* bytes were received or a time out occurred. When the return value is less than *minSize*, a time out has occurred. When the return value is less than zero, an error has occurred. Usually it means that the socket has disconnected.

```
virtual bool    Read( void* buf, size_t size ) = 0;
```

Receives incoming data, transferring it from the socket into the buffer *buf*. The function does not return to the caller before exactly *size* bytes were received or an error occurred. The function returns `true` when the transfer was successful.

```
virtual bool    Write( void const* buf, size_t size ) = 0;
```

Sends the data contained in buffer *buf* over the socket. The buffer is assumed to contain *size* bytes. The function returns `true` when all bytes could be transferred successfully. In case of an error `false` is returned.

```
virtual bool    IsOk() = 0;
```

Checks whether the socket is correctly initialized and ready for operation.

```
virtual string  GetErrorText() = 0;
```

Returns an error message text for the last error occurred.

```
virtual string  GetSocketAddress() const = 0;
```

Returns the IP address of the socket in string representation.

```
virtual string  GetPeerAddress() const = 0;
```

Returns the IP address of the socket to which this socket is connected in string representation.

```
virtual Socket* Accept() = 0;
```

Is called by a server to establish a pending client connection. When the client executes the *Connect* method and accesses the server's accept port, this method will create a new socket, which can be used for communication with the client.

The function returns a pointer to a new socket that controls the communication between client and server. The new socket must be released by the server once it has finished using it. If the operation failed a `NULL` pointer is returned.

The function *Accept* blocks until a connection will be established and therefore cannot be used to detect activity on multiple sockets.

```
virtual bool    CancelAccept() = 0;
```

Cancels an accept operation and closes the socket.

```
virtual bool    Close() = 0;
```

Closes the socket.

NOTE: The operating system decrements the associated reference counter of the socket by one. The TCP/IP connection is closed when the reference counter reaches zero.

```
virtual bool    ShutDown() = 0;
```

Shuts down the socket. Thereafter read and write operations on the socket are prohibited. All future attempts to read or write data from/to the socket will be refused. But all previously initiated operations are guaranteed to be completed. The function returns `true` if the operation was successfully completed, `false` otherwise.

```
static string GetHostname( const string& ipAddress );
```

Tries to get the fully qualified host name corresponding to the IP address *ipAddress* which is given in string representation. If the method fails the string $< unknown >$ will be returned.

```
static int GetIP( const string& address );
```

Gets the IP address of the host. $address$ parameter should contain either symbolic host name (for example $robinson$), or a string with IP address (for example $195.239.208.225$)

```
iostream& GetSocketStream();
```

Returns a reference to the I/O stream associated with the socket.

An I/O stream is available only for sockets created by the methods *Connect* and *Accept*.

```
protected:
 enum { SS_OPEN, SS_SHUTDOWN, SS_CLOSE } state;
```

Defines the socket state.

```
  SocketBuffer* ioSocketBuffer; // Socket stream buffer
  iostream*     ioSocketStream; // Socket I/O stream
};

extern string GetProcessName();
```

Returns the current host name combined with an identifier of the current process.

G.7.5 Class $SocketAddress$

Class $SocketAddress$ represents a socket address. Socket addresses combine an IP address with a port number. The IP address identifies a host, while the port number identifies a service available on the host. Typically used by TCP/IP clients to indicate a machine and service they are connecting to.

```
class SDB_EXPORT SocketAddress
{
 public:
  SocketAddress();
```

Initializes a socket address, which consists of an IP address and a port number.

The IP address is set to the wildcard address (`INADDR_ANY`). The port number is set to zero.

```
SocketAddress( const SocketAddress& sockAddr );
```

Creates a socket address that is an identical copy of *sockAddr*.

```
SocketAddress( const string& ipAddr, uint16_t portNo = 0 );
```

Initializes a socket address converting the string representation of an IP address *ipAddr* into the internal binary representation. The port number is set to *portNo*.

```
virtual ~SocketAddress();
```

Destroys a socket address.

```
SocketAddress& operator=( const SocketAddress& sockAddr );
```

Changes *self* into an identical copy of the socket address referenced by *sockAddr*.

```
bool operator==( const SocketAddress& sockAddr ) const;
```

Compares *self* with socket address *sockAddr*. The method returns `true` if both objects are equal, otherwise it returns `false`.

```
void SetAddress( const string& ipAddr,
                 uint16_t portNo = 0 );
```

Sets the socket address converting the string representation of an IP address *ipAddr* into the internal binary representation. The port number is set to *portNo*.

```
void SetAddress( const string& ipAddr,
                 const string& portNo );
```

Sets the socket address converting the string representation of an IP address *ipAddr* into the internal binary representation. The string representation *portNo* of the port number is also converted.

```
string GetSocketString() const;
```

Returns the IP address of the socket including the port number as a string. The port number is appended to the IP address after inserting a colon as a delimiter, i.e. *132.176.69.10:1234*.

```
string GetIPAddress() const;
```

Returns the IP address portion of the socket address in string format.

```
uint16_t GetPort() const;
```

Returns the port number portion of the socket address in host byte order.

```
 protected:
  int sa_len;
  union
  {
    struct sockaddr    sock;
    struct sockaddr_in sock_inet;
  } u;
};
```

G.7.6 Class *SocketRule*

```
class SDB_EXPORT SocketRule
{
 public:
  enum Policy { DENY, ALLOW };
```

Is an enumeration of access policies:

- *ALLOW* – specifies that access should be granted when a host address matches the rule.
- *DENY* – specifies that access should be denied when a host address matches the rule.

```
SocketRule();
```

Creates an empty rule.

```
SocketRule( const string& strIpAddr,
            const string& strIpMask,
            const Policy setAllowDeny = ALLOW );
```

Creates a rule initialized by the given IP address *strIpAddr* and IP address mask *strIpMask* and the access policy *setAllowDeny*.

```
virtual ~SocketRule() {};
```

Destroys a rule.

```
bool Match( const SocketAddress& host );
```

Checks whether the IP address *host* matches the rule. The access policy is ignored.

```
bool Allowed( const SocketAddress& host );
```

Checks whether access should be allowed for the IP address *host*.

```
bool Denied( const SocketAddress& host );
```

Checks whether access should be denied for the IP address *host*.

```
static void SetDelimiter( const char newDelimiter = '/' );
```

Sets the delimiter used between IP address, IP address mask and access policy when a rule is sent onto an output stream.

```
static char GetDelimiter();
```

Returns the current delimiter used between IP address, IP address mask and access policy when a rule is sent onto an output stream.

```
friend ostream& operator <<( ostream& os, SocketRule& r );
```

Allows to send a rule to an output stream.

```
protected:
 Policy  allowDeny;     // Access policy
 in_addr ipAddr;        // IP address of the rule
 in_addr ipMask;        // IP address mask
 static char delimiter; // Output delimiter
};
```

G.7.7 Class $SocketRuleSet$

```
class SDB_EXPORT SocketRuleSet
{
 public:
  SocketRuleSet( SocketRule::Policy initDefaultPolicy =
                   SocketRule::DENY );
```

Creates an empty rule set with a default access policy *initDefaultPolicy*.

```
  virtual ~SocketRuleSet() {};
```

Destroys a rule set.

```
  void AddRule( const string& strIpAddr,
                const string& strIpMask,
                SocketRule::Policy allowDeny =
                  SocketRule::ALLOW );
```

Adds a rule consisting of the IP address *strIpAddr*, the IP address mask *strIpMask* and the access policy *allowDeny* to the rule set.

```
  bool Ok( const SocketAddress& host );
```

Checks whether access should be granted for the IP address *host*.

```
  bool LoadFromFile( const string& ruleFileName );
```

Loads a set of rules from the file with name *ruleFileName*. The method returns `true` when the file could be read successfully.

```
  bool StoreToFile( const string& ruleFileName );
```

Stores a set of rules into the file with name *ruleFileName*. The method returns `true` when the file could be written successfully.

```
  friend ostream& operator <<(ostream& os, SocketRuleSet& r);
```

Allows to send a set of rules to an output stream.

```
 protected:
   vector<SocketRule> rules;         // Set of rules
   SocketRule::Policy defaultPolicy; // Access policy for set
};
```

G.7.8 Class $SocketBuffer$

The class $SocketBuffer$ implements the standard **streambuf** protocol for sockets. Separate buffers for reading and writing are implemented.

```
class SDB_EXPORT SocketBuffer : public std::streambuf
{
 public:
  SocketBuffer( Socket& socket );
```

Creates a socket buffer associated with the socket *socket*.

```
~SocketBuffer();
```

Destroys a socket buffer.

```
bool is_open() const { return socketHandle != 0; }
```

Checks whether the stream buffer is ready for operation.

```
SocketBuffer* close();
```

Closes the stream buffer.

```
streampos seekoff( streamoff, ios::seek_dir, int )
  { return EOF; }
```

Disallows seeking in the stream buffer since a TCP stream is strictly sequential.

```
int xsputn( const char* s, const int n );
```

Allows faster writing onto the socket of a string consisting on *n* characters.

```
int xsgetn( char* s, const int n );
```

Allows faster reading from the socket of a string consisting on *n* characters.

```
protected:
 virtual int overflow( int ch = EOF );
```

Writes the data in the output buffer to the associated socket.

```
 virtual int uflow();
 virtual int underflow();
```

Tries to read data from the associated socket, when the input buffer is empty.

```
 virtual int sync();
```

Flushes all output data from the buffer to the associated socket.

```
 virtual int pbackfail( int ch = EOF );
```

Disallows to unget a character.

```
private:
 SocketBuffer( const SocketBuffer& );
 SocketBuffer& operator=( const SocketBuffer& );

 Socket* socketHandle; // Handle of associated socket
 int     bufferSize;   // Size of the I/O buffer
 char*   inBuffer;     // Input buffer
 char*   outBuffer;    // Output buffer
};

#endif
```

Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, den 27. Juni 2002