Studiengang: Informatik

Betreuer: Dipl.-Inf. Christoph Mangold

Prüfer: Prof. Dr.-Ing. B. Mitschang

Beginn am: 1.08.2001

Ende am: 31.01.2002

CR-Nummer: H2.8, H3.4

Studienarbeit-Nr. 1824

Portierung eines DB-Modells

Yiping Chen

Abteilung für Anwendungssoftware

Institut für Parallele und Verteilte Höchstleistungsrechner

Universität Stuttgart

Breitwiesenstraße 20-22

D-70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Man betrachtet zuerst die Situation in der Maschinen- und Anlagenbauindustrie in Deutschland. Deutsche Maschinenbauer sehen sich jetzt vor der hohen Konkurrenz mit den aus billigen Lohnländern. Um eine wirtschaftliche Produktion von Maschinen und Anlagen in Deutschland noch weiter sicherzustellen, entwickeln die Maschinen- und Anlagenbauer wiederverwendbare mechatronische Komponenten. Die Konstruktionsunterlagen von Komponenten sollen wieder verwendet werden, damit die Herstellungskosten reduziert werden. Aber mechatronische Komponenten werden in den verschiedenen Abteilungen mit verschiedenen Erstellungssystemen entwickelt. Die Entwurfingenieure benutzen beispielsweise CAD-Systeme, um Komponenten zu konstruieren. Die Fertigungstechniker benutzen SPS-Programmiersysteme um Steuerungsprogramme der Fertigungsmaschinen, die Komponenten verarbeiten, zu schreiben. Bei Dokumentation benutzt man Microsoft-software. In dieser Situation ist es schwierig, die Wissen entwickelter Komponenten wieder zu verwenden, weil die Wissen verteilt in den verschiedenen heterogenen Erstellungssystemen gespeichert sind. Bei der Wiederverwendung von Komponenten entstehen dadurch hohe Kosten. Aus dieser Situation entsteht das föderal Projekt.

In diesem Kapitel wird in Abschnitt 1.1 ein Überblick über das Projekt föderal, zu dem diese Studienarbeit gehört, gegeben. In Abschnitt 1.2 wird die Aufgabenstellung dieser Studienarbeit erläutert. In Abschnitt 1.3 wird Gliederung der vorliegenden Ausarbeitung erläutert. Man beginnt mit dem Überblick des Projektes.

1.1 Überblick des Projektes

In diesem Unterabschnitt werden Zielsetzung und Projektbearbeiter, Informationsarchitektur, Aufgabenstellung und Vorgehensweise des Projektes kurz vorgestellt.

1.1.1 Zielsetzung und Projektbearbeiter

Das Föderal-Projekt hat sich zum Ziel gesetzt, die vorhandenen heterogenen Erstellungssysteme mit einer einheitlichen Informationsarchitektur zu integrieren sowie eine intensive

Erprobung der Architektur und eine anschließende Verbreitung der Ergebnisse des Projektes[Foe01]. Das Projektkonsortium besteht aus Forschungsinstituten der Universität Stuttgart, Maschinenbaufirmen und einer IT-Dienstleistungsfirma. Das Projekt wird durch Mittel des Bundesministeriums für Bildung und Forschung (BMBF) gefördert.

1.1.2 Informationsarchitektur

Ein Lösungsansatz der beschriebenen Probleme liegt darin, eine Modellierungssprache für die Darstellung der Zusammenhänge zwischen heterogenen Erstellungssystemen zu entwickeln. Dies erfolgt durch ein semantisches Netz. Abbildung 1[Lit01] zeigt eine sogenannte föderale Informationsarchitektur. In dieser Struktur wird das semantische Netz in einer zentralen Datenbank beschrieben.

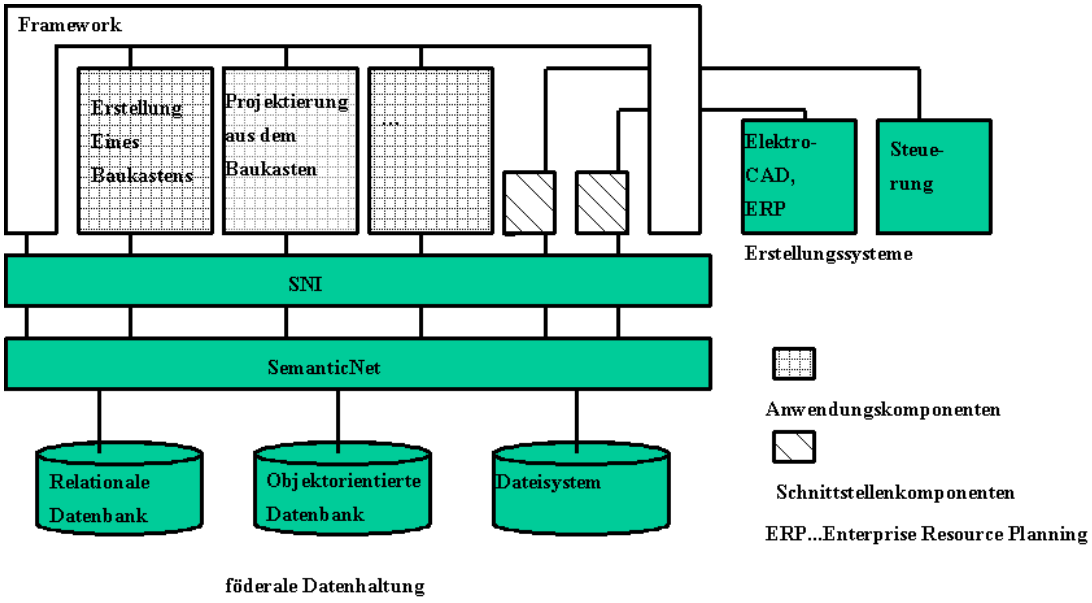

Abbildung 1: fo:deral Informationsarchitektur(FIA)

Die Zentrale Datenbank kann eine relationale Datenbank oder eine objektorientierte Datenbank oder ein Dateisystem sein. Anwendungskomponenten werden über Semantic Net Interface (SNI) ins Netz integriert. Durch SNI kann man das Netz navigieren und abfragen. SNI bietet eine einheitliche Schnittstelle an, die transparent verdeckt, ob das semantische Netz in relationalen Datenbanken oder objektorientierten Datenbanken gespeichert wird. Bei einer Portierung können die Benutzer weiter das System nutzen,

ohne ihre Anwendungsprogramme zu ändern. Erstellungssysteme wie Elektro-CAD-Systeme und Steuerungssysteme werden nach Ausprägung des vorhandenen Systems durch Schnittstellenkomponenten integriert. Die Integration von Anwendungs- und Schnittstellenkomponenten zu einem Erstellungssystem ist durch ein Framework zu erledigen. Siehe Abbildung 1.

1.1.3 Aufgabenschwerpunkte und Vorgehensweise

Im Bezug auf die Ziele, die in Unterabschnitt 1.1.1 erläutert werden, ergeben sich im Projekt folgende Aufgabenschwerpunkte[Foe01]:

1) **Spezifikation**: Eine Schnittstelle zum semantischen Netz wird durch Zusammenarbeit der Forschungsinstitute und Industriepartner des Konsortiums sowie basierend auf Anforderungen externer Firmen spezifiziert.
2) **Entwicklung**: Die methodischen Grundlagen für eine föderale Integration sind auf der Basis von Spezifikation zu erarbeiten. Die Konzeption einer Softwarearchitektur und die prototypische Umsetzung dieser Architektur gehören ebenfalls zur Entwicklung.
3) **Verwertung**: Die Softwarearchitektur sollte möglichst früh in Pilotanwendungen evaluiert werden. Die Verbreitung von Teilergebnissen während des Projektes gehört ebenfalls zur Aufgabe.

Die Vorgehensweise für die Aufgabenschwerpunkte ist möglicherweise das Wasserfallmodell. Nach dem Wasserfallmodell wird ein Projekt etwa nach zeitlicher Sequenz in Analyse-, Design-, Implementierungs- und Testphasen geteilt. Wenn das Projekt föderal nach dem Wasserfallmodell erarbeitet wird, dann wird ein Lösungskonzept nach einer Analyse des Problembereichs erarbeitet, prototypisch entworfen und implementiert und schließlich von den Projektpartnern evaluiert. Aber diese Vorgehensweise hat folgende Nachteile[Foe01]:

1) Im Bereich der Informationstechnik ist die Innovationsgeschwindigkeit sehr schnell. Dieses Projekt dauert mehrere Jahre. Während der Projektlaufzeit entsteht schon neue Technik, die übernommen werden muß. Eine Vorgehensweise nach dem Wasserfallmodell ist in dieser Situation nur bedingt geeignet.

2) Ein Feedback externer Firmen, die nicht in die tägliche Projektarbeit eingeschlossen sind, wird in der Praxis nur durch eine Präsentation erreichter Ergebnisse basierend auf einer prototypischen Umsetzung erreicht.

Wegen dieser Nachteile ist das Projekt nach einen iterativen Entwicklungsprozess zu erarbeiten. Die Phasen vom Wasserfallmodell werden mehrfach durchlaufen. Bei dieser Vorgehensweise sind mehrere Prototypen der Softwarearchitektur zu definieren, die nach jeder Iteration entstanden sind. Der letzte Prototyp ist so zu definieren, daß seine Funktionen für eine anschließende Verwertung der Ergebnisse des Projektes ausreichend ist.

1.2 Aufgabenstellung

Das semantische Netz wurde von einem Projektpartner in einer objektorientierten Datenbank mit Objektstore und Java implementiert. Diese Studienarbeit hat das Ziel, derzeit vorhandene Implementierung in ein objektrelationales und ein relationales Modell umzusetzen. Anschließend wird das objektrelationale und das relationale Modell in DB2 mit JAVA und JDBC implementiert. Schließlich werden die beiden Implementierungen durch Performancemessungen verglichen.

Die beiden in der Arbeit realisierten Implementierungen sollten mit der vorhandenen objektorientierten Implementierung durch Performancemessungen verglichen werden. Da diese Arbeit parallel mit der Arbeit des Projektpartners läuft und die Arbeit des Projektpartners verschoben ist, ist in dieser Arbeit nicht möglich, die Performance aller Implementierungen zu vergleichen.

1.3 Gliederung der Ausarbeitung

Die vorliegende Ausarbeitung gliedert sich in 4 Kapitel. Kapitel 1 ist über Zielsetzung und Projektbearbeiter, Informationsarchitektur, Aufgabenstellung und Vorgehensweise des Projektes und Aufgabenstellung dieser Arbeit. Kapitel 2 behandelt die Grundlagen der objektorientierten Datenbanken, des Entity-Relationship-Modells, der relationalen und objektrelationalen Datenbanken, Entwicklungsumgebung und Klassifizierung der

Datenbank-Management-Systeme. Kapitel 3 ist über Anforderungsanalyse der Arbeit, vorhandener objektorientierter Implementierung, relationaler und objektrelationaler Implementierung, JDBC-Schnittstelle für relationale und objektrelationale Implementierung und Performancemessung von relationaler und objektrelationaler Implementierung. Kapitel 4 ist eine kurze Zusammenfassung dieser Arbeit. Es gibt zwei Anhänge, in denen die in der Arbeit benutzten SQL-Anweisungen beschrieben sind. Schließlich gibt es ein Literaturverzeichnis, in dem die in der Arbeit benutzten Literaturquellen aufgelistet sind.

2. Grundlagen

In diesem Kapitel werden die technischen Grundlagen dieser Studienarbeit vorgestellt. In Abschnitt 2.1wird objektorientiertes Datenmodell kurz vorgestellt. Dieses Modell dient als Grundlage für die objektorientierte Implementierung in diesem Projekt. In Abschnitt 2.2 wird das Entity-Relationship-Modell vorgestellt. Das Entity-Relationship-Modell dient als ein Modellierungswerkzeug für relationale Datenbanken. In Abschnitt 2.3 wird das relationale Datenmodell erläutert. Dieses Modell dient als Grundlage für die relationale Modellierung in der Arbeit. In Abschnitt 2.4 wird das objektrelationale Datenmodell erläutert. Dieses Modell ist die Grundlage für die objektrelationale Implementierung. In Abschnitt 2.5 wird eine Klassifikation von Datenbank-Management-Systemen erläutert. In Abschnitt 2.6 wird Entwicklungsumgebung dieser Arbeit vorgestellt.

2.1 Das objektorientierte Datenmodell

Objektorientierte Datenbank-Management-Systeme (DBMS) finden ihre Anwendungen hauptsächlich bei Nicht-Standard-Anwendungen, wie z.B. bei Computer Aided Design (CAD), Multimediaanwendungen, Architektur etc. Bei dem objektorientierten Datenmodell spielt der Objekttyp (Klasse) eine zentrale Rolle. Der Objekttyp besteht aus Attributen und Methoden. Die Attribute repräsentieren die Struktur eines Objektes und die Methoden zeigen das Verhalten eines Objektes[KE97].

In Unterabschnitt 2.1.1 wird die Definition von Objekt erläutert. Anschließend in Unterabschnitt 2.1.2 werden Entwicklungsansätze von objektorientierten DBMS vorgestellt. In Unterabschnitt 2.1.3 wird ein Ausblick gegeben.

2.1.1 Was ist ein Objekt?

Der zentrale Begriff dieses Modells ist das Objekt. Im objektorientierten Datenmodell besteht ein Objekt aus drei Teilen[KE97]:

Identität: Jedes Objekt hat eine vom System generierten Objektidentität. Die Objektidentität verändert sich nicht während der Lebenszeit des Objekts.

Typ: Jedes Objekt gehört zu einem Typ. Der Typ wird auch Klasse genannt. Die Struktur und das Verhalten des Objektes werden durch seinen Typ festgelegt. Ein einzelnes Objekt wird durch die Instanziierung eines Objekttyps erstellt. Die Attribute und Methoden des Typs werden jedem Objekt zugeordnet.

Wert bzw. Zustand: Jedes Objekt hat zu jedem Zeitpunkt seiner Lebenszeit einen bestimmten Zustand. Der Zustand eines Objektes wird durch die Werte seiner Attribute bestimmt.

2.1.2 Entwicklungsansätze objektorientierter Datenbankmanagementsysteme

Momentan auf dem Markt gibt es viele kommerzielle objektorientierte DBMS. Die Entwicklungsansätze der objektorientierten DBMS kann man in drei wesentliche Gruppen unterteilen[STS97]:

1) Erweiterung von objektorientierten Programmiersprachen um Datenbankkonzepte
2) Erweiterung bestehender DBMS um objektorientierte Konzepte
3) Neu entwickelte objektorientierte DBMS

Die objektorientierte DBMS, die in diesen Gruppen unterteilt werden, haben unterschiedliche Vorteile und Nachteile, die in den folgenden Unterabschnitten kurz vorgestellt werden.

1. Erweiterung von objektorientierten Programmiersprachen um Datenbankkonzepte:

Es gibt viele Datenbankmanagementsysteme, die objektorientierte Programmiersprachen wie u.a. C++ und Smalltalk um Datenbankfunktionalität erweitern. Beispiele sind GemStore, ONTOS, POET, Versant oder ObjectStore. Ein programmiersprachennahes objektorientiertes DBMS entsteht, wenn man die folgenden Datenbankkonzepte zu der Programmiersprache hinzufügt:

- Persistenz und

- Transaktionen (Synchronisation und Recovery)

Der Vorteil dieses Ansatzes ist, daß man nur eine einzige Sprache für Datenbank- und Anwendungsprogrammierung braucht. Der Nachteil ist, daß Datenbankkonzepte wie Anfragesprachen, Sichten, Zugriffspfadunterstützung, Integritätsbedingungen und Zugriffskontrolle in diesem System nicht betrachtet werden.

2. Erweiterung bestehender Datenbankmanagementsysteme um objektorientierte Konzepte:

Man erweitert traditionelle relationale Datenbanksysteme um objektorientierte Konzepte. Bei der Erweiterung werden objektorientierte Strukturierungskonzepte hinzugefügt. Durch Benutzung von Typkonstruktoren können komplex strukturierte Typen erzeugt werden. Objektidentifikatoren und Referenzen werden eingeführt. Die gegenwärtige Entwicklungsrichtung der objektrelationalen DBMS gehört zu diesem Ansatz. Beispiele sind Informix, Oracle und DB2 von IBM, welches in dieser Studienarbeit benutzt wird. Der große Vorteil hier ist, daß man die Kenntnisse der relationalen DBMS weiter benutzen kann und die Anwendungen von relationalen DBMS leicht auf objektrelationale DBMS umgesetzt werden können. Der Nachteil ist, daß die Datenbankprogrammier- und Anfragesprache nicht gleich bleibt.

3. Neu entwickelte objektorientierte Datenbankmanagementsysteme:

Ein Beispiel für neu entwickelte objektorientierte DBMS ist O_2 von O_2 Technology. Die neu entwickelten DBMS basieren weder auf einem Objektmodell einer Programmiersprache noch auf einem vorhandenen Datenbankmodell, sondern auf einem eigenen Datenmodell und vereinen meistens viele objektorientierte und datenbankspezifische Konzepte.

Der Vorteil dieses Ansatzes liegt darin, daß ein Datenmodell entwickelt werden kann, ohne vorgegebene Konzepte zu berücksichtigen. Die beiden anderen Ansätze haben den Nachteil, daß man neue Konzepte in vorhandene Modelle einwandfrei integrieren muß.

Der Nachteil der Neuentwicklung liegt darin, daß bereits vorhandene Erfahrungen mit bestehenden Systemen nur teilweise weiter benutzt werden können.

Abbildung 2 zeigt eine Zusammenfassung der Entwicklungsansätze der objektorientierten DBMS. Basierend auf einer Programmiersprache oder einem relationalen Datenbanksystem, wird ein objektorientiertes DBMS entworfen, indem Datenbankfunktionalitäten oder objektorientierte Konzepte in die Basis integriert werden. Die neu entwickelten objektorientierten DBMS basiert auf einem eigenen Datenmodell.

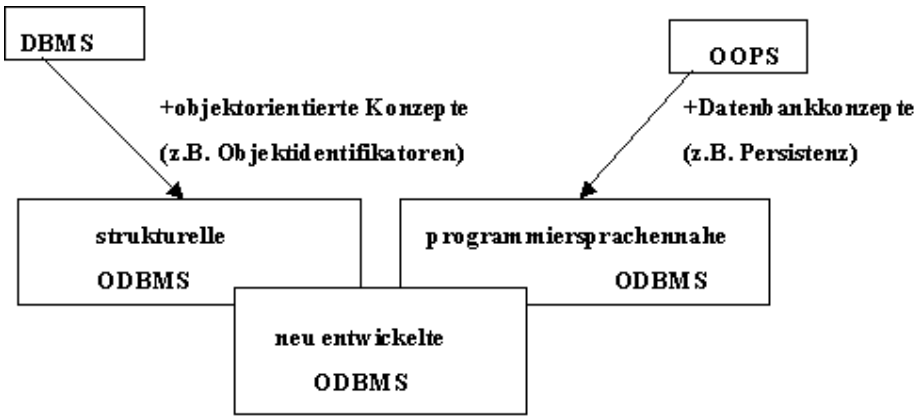

Abbildung 2: Entwicklungsansätze von ODBMS

2.1.3 Ausblick

Die objektorientierten DBMS haben sowohl Vorteile als auch Nachteile, wie folgt: Die objektorientierten DBMS vereinigen die objektorientierten Konzepte mit Datenbankkonzepten. In kommerziellen objektorientierten DBMS werden aber die deskriptiven Anfragesprachen mangelhaft realisiert. Die Anfragen können aufgrund ihres komplexen Modells nur bedingt optimiert werden. Sichten werden nicht implementiert. Objektorientierte DBMS können relationale DBMS nicht ersetzen[STS97].

2.2 Entity-Relationship-Modell

In diesem Abschnitt werden die Grundlagen des Entity-Relationship-Modells kurz vorgestellt. Das Entity-Relationship-Modell ist ein verbreitetes Datenmodell bei Modellierung von Datenbanken und ein semantisches Modell.

In Unterabschnitt 2.2.1 werden Basisbegriffe des Modells vorgestellt. In Unterabschnitt 2.2.2 werden schwache Entities vorgestellt. In Unterabschnitt 2.2.3 werden Erweiterung des Modells vorgestellt.

2.2.1 Basisbegriffe

Das Modell hat fünf Basisbegriffe: Entities, Attribute, Wertmengen von Attributen, Beziehungen und Primärschlüssel.

Entities

Ein Entity stellt ein abstraktes oder in der Realwelt existiertes Objekt dar, das für Datenbankenmodellierung sinnvoll ist. Beispiele dafür sind ein Student, ein Buch, und so weiter. Man kann ähnliche Entities zu einem Entitytyp oder einer Entitymenge abstrahieren. Entitymengen fassen die Menge der Instanzen von einem Entitytyp, also die Objekte von einem gleichen Entitytyp zusammen. Beispiele für Entitytyp oder Entitymengen sind Studenten oder Bücher. Im Entity-Relationship-Diagramm (ER-Diagramm) werden Entitytypen oder Entitymengen durch Rechtecke repräsentiert und der Name des Entitytyps wird innerhalb der Rechtecke angegeben.

Attribute

Ein Attribut beschreibt einen bestimmten Charakter von einem Entitytyp. z.B. ein Entitytyp „Studenten” hat Attribute Immatrikulationsnummer, Name, Geschlecht und Alter. Im ER-Diagramm werden Attribute durch Ovale oder Kreise repräsentiert.

Wertmengen von Attributen

Wertmengen definieren die Werte, die ein Attribut annehmen kann. Beispielsweise liegt der Wert von einem Attribut, Alter von einem Menschen, zwischen 0 und 250. Wertmengen können durch Prädikate und Aufzählungen definiert werden.

Beziehungen

Beziehungen stellen die Verbindung zwischen Entitytypen oder Entitymengen dar. Beispielsweise könnte „lehren" die Beziehung zwischen Entitytyp „Professoren" und Entitytyp „Studenten" sein. Man kann ähnliche Beziehungen zu einem Beziehungstyp abstrahieren. Die Beziehung kann Attribute haben. Es gibt drei Basistypen von Beziehungen, nämlich 1:1, 1:n und n:m. Im ER-Diagramm werden Beziehungsmengen durch Rauten repräsentiert und innerhalb der Raute wird ähnlich wie bei Entitytypen auch der Beziehungsname angegeben. Die Rauten werden mit den beteiligten Entitytypen durch Linien verbunden und die Typen des Beziehungstyps, nämlich 1:1, 1:n oder n:m werden auf der Linie angegeben.

Das Entity-Relationship-Modell unterstützt die 1:1, 1:n und n:m Beziehungen zwischen den Entitymengen. Siehe Abbildung 3.

- Wenn ein Entity einer Entitymenge nur mit einem einzigen Entity einer anderen Entitymenge in Beziehung steht, vice versa, heißt die Beziehung zwischen den beiden Entitys 1:1, beispielsweise hat eine Universität nur einen Rektor und ein Rektor darf nur der Rektor einer Universität sein, dann ist die Beziehung zwischen Entitymenge „Universitäten" und Entitymenge „Rektoren" 1:1.
- Wenn ein Entity einer Entitymenge „A" mit mehreren Entitys einer Entitymenge „B" in Beziehung steht und umgekehrt es für ein Entity aus der Entitymenge „B" ein einziges in Beziehung stehendes Entity in der Entitymenge „A" gibt, heißt die Beziehung zwischen den beiden Entitymengen 1:n, beispielsweise hat eine Firma mehrere Telefonnummern, aber eine Telefonnummer gehört nur zu einer Firma. Die Beziehung zwischen Entitymenge „Firmen" und „Telefonnummer" ist dann 1:n.
- Wenn ein Entity einer Entitymenge mit mehreren Entitys einer anderen Entitymenge in Beziehung steht, heißt die Beziehung zwischen den beiden Entitymengen n:m,

beispielsweise kann ein Student mehrere Vorlesungen hören, und eine Vorlesung kann von mehreren Studenten gehört werden. Die Beziehung zwischen Entitymenge „Vorlesungen" und „Studenten" ist dann m:n.

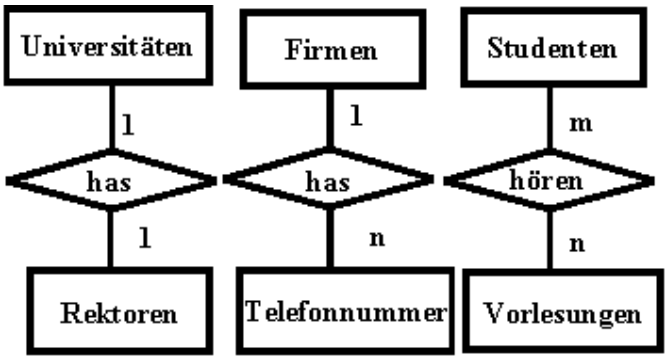

Abbildung 3: Beispiele für 1:1, 1:n und m:n Beziehungen

Im Entity-Relationship-Modell sind ebenfalls rekursive Beziehungen erlaubt. Eine rekursive Beziehung repräsentiert die Beziehung zwischen einer Entitymenge und sich selbst, z.B. die Entitymenge „Mitarbeiter" kann durch Beziehungsmenge „manage" mit sich verbunden werden. Unten den Mitarbeitern gibt es Manager. Die anderen Mitarbeiter außer Managern werden durch einen Manager geleitet. Abbildung 4 zeigt ein Beispiel für rekursive Beziehungen.

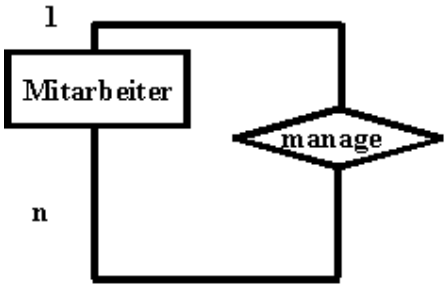

Abbildung 4: rekursive Beziehung

Primärschlüssel

Ein Primärschlüssel ist ein Attribut von einem Entity, das ein Entity eindeutig identifizieren kann. Der Primärschlüssel kann aus einzigem Attribut oder zusammengesetzten Attributen bestehen. Beispiele dafür sind die Immatrikulationsnummer von einem Student oder die Personalnummer eines Mitarbeiters in einer Firma.

2.2.2 Schwache Entities

Wenn die Existenz einer Entitymenge von einer anderen Entitymenge abhängig ist, heißt die erste Entitymenge schwache Entitymenge. Die schwache Entitymenge wird im ER-Diagramm durch doppelt umrandete Rechtecke repräsentiert. Ein konkretes Beispiel dafür ist die Beziehung zwischen „Gebäude" und „Räume". Wenn ein Gebäude abgebrochen wird, existieren auch die Räume in diesem Gebäude nicht mehr. Siehe Abbildung 5.

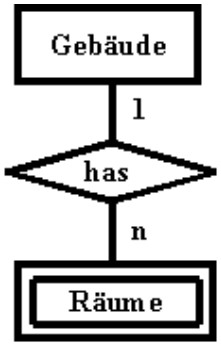

Abbildung 5: Beispiel für schwache Entity

2.2.3 Erweiterung von Entity-Relationship-Modell

Das Entity-Relationship-Modell wurde erweitert, um komplexere semantische Beziehung zwischen Entities darzustellen. Zu den wichtigeren Erweiterungen des Modells zählen das Konzept der Generalisierung/Spezialisierung, Assoziation und Aggregation.

Das Konzept der Generalisierung/Spezialisierung drückt eine Beziehung wie Superklasse/Subklasse in den objektorientierten Programmiersprachen aus. Bei der Generalisierung werden gemeinsame Eigenschaften ähnlicher Entitytypen zu einer Superklasse abstrahiert. Generalisierung drückt die Beziehung „subclass-of" aus. Die Instanzen einer Subklasse sind auch die Instanzen der Superklasse. Die Subklasse erbt von der Superklasse und hat deshalb alle Eigenschaften ihrer Supertypen und zusätzliche eigene Eigenschaften. Spezialisierung ist die inverse Operation von Generalisierung und drückt die Beziehung „instance -of" aus. Bei der Vererbung gibt es das Problem der mehrfachen

Vererbung von Attributen. Eine Instanz erbt von seinen mehreren Superklassen ein jeweils in verschiedenen Superklassen definiertes Attribut mit gleichem Namen. Um dieses Problem zu vermeiden kann man eins der Attribute in einem der Supertypen neu definieren.

Bei dem Konzept der Assoziation gibt es Element-Assoziation und Mengen-Assoziation. Element-Assoziation faßt Elemente zusammen damit die Elemente als ein ganzes Objekt dargestellt werden. Sie drückt die Beziehung von „element-of" aus. Die Mengen-Assoziation ergänzt Element-Assoziation und beschreibt die Beziehungen zwischen zusammengesetzten Mengenobjekten. Sie drückt die Beziehung von „subset-of" aus. Zum Beispiel, Senat ist „subset-of" Universität-Gremien, Herr X ist „element-of" Senat.

Bei dem Konzept von Aggregation handelt es sich um die Zusammensetzung der Objekte aus einfachen Objekten. Sie drückt die Beziehung „part-of" und „component-of" aus, zum Beispiel, Drucker ist „part-of" Peripherie, Peripherie ist eine Komponente eines PC-Systems.

2.3 Das relationale Datenmodell

In diesem Abschnitt wird das relationale Datenmodell vorgestellt. In Unterabschnitt 2.3.1 werden Grundlagen des Modells vorgestellt. In Unterabschnitt 2.3.2 wird die Transformation zwischen Entity-Relationship-Modell (siehe Abschnitt 2.2) und relationalem Datenmodell vorgestellt. In Unterabschnitt 2.3.3 wird die Structured Query Language (SQL) kurz vorgestellt. In Unterabschnitt 2.3.4 wird die referentielle Integrität vorgestellt. In Unterabschnitt 2.3.5 werden Schwachpunkte der relationalen Datenbank erläutert, bevor in Unterabschnitt 2.3.6 ein Ausblick gegeben wird.

2.3.1 Grundlagen des relationalen Datenmodells

Das relationale Datenmodell wurde von E. F. Codd 1970 vorgestellt und hat eine solide mathematische Grundlage. Die auf dem relationalen Datenmodell aufgebauten Datenbanksysteme sind gegenwärtige Marktführer und weltweit verbreitet. Die Basisdatenstruktur des relationalen Datenmodells ist zweidimensionale Tabellen. Die Tabelle heißt auch **Relation**. Jede Zeile der Tabelle heißt ein **Tupel** der Tabelle und repräsentiert ein Entity. Eine Tabelle besteht aus m Tupeln. Die Tupel einer Relation wiederholen sich nicht. Die Ordnung der Tupel ist bedeutungslos. Jede Spalte der Relation heißt **Attribut** (oder Field) der Relation. Die Ordnung der Spalten ist bedeutungslos. Die Attribute wiederholen sich nicht. Jedes Attribut hat einen Wertbereich bzw. eine Domäne (Domain). Die Anzahl der Attribute einer Relation ist konstant. Jede Tabelle hat auch einen Primärschlüssel. Siehe Abbildung 6.

Primär- und Fremdschlüssel

Der Primärschlüssel identifiziert eindeutig ein Tupel in der Tabelle. Der Primärschlüssel besteht aus einem oder mehreren, zusammengesetzten Attributen. Ein Fremdschüssel ist ein Attribut oder besteht aus mehreren, zusammengesetzten Attributen einer Tabelle, deren Wert auf den Wert des Primärschlüssel einer anderen Tabelle verweist. Für jeden Wert eines Fremdschlüssels muß es einen Wert im Primärschlüssel der in Beziehung stehenden Tabelle geben. Es kann mehrere Fremdschlüssel in einer Tabelle geben. Der

Fremdschlüssel spielt eine wichtige Rolle im relationalen Datenmodell. Das Paar Primärschlüssel/Fremdschlüssel drückt die Beziehungen zwischen den Tabellen aus.

Abbildung 6 zeigt ein Beispiel für eine relationale Datenbank. Tabelle `Person` hat die Attribute `PNR`, `Name`, `Vorname` und `Abteilung`. Attribut `PNR` ist Primärschlüssel und Attribut `Abteilung` ist Fremdschlüssel und verweist auf den Primärschlüssel von Tabelle `Abteilung`, die 3 Attribute hat.

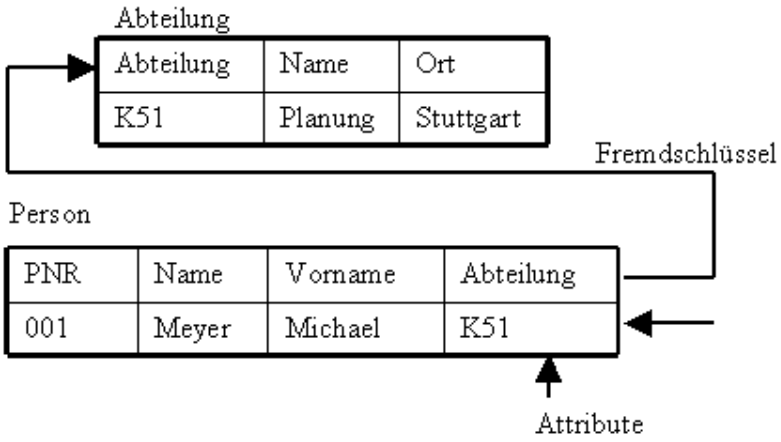

Abbildung 6: Ein Beispiel für relationale Datenbank

2.3.2 Transformation zwischen dem Entity-Relationship-Modell und dem relationalen Modell

Wenn man das Entity-Relationship-Modell (siehe Abschnitt 2.2) in das relationale Modell umwandeln möchte, muß man zuerst die Entitytypen und Beziehungstypen des Entity-Relationship-Modells umwandeln. Es gibt mehrere Möglichkeiten bei der Umwandlung. Im folgenden Unterabschnitt werden die Transformation und Fallbeispiele dazu vorgestellt.

Transformation von Entitytypen

Jeder Entitytyp wird in eine Tabelle umgewandelt. Die Attribute des Entitytyps verwandeln sich in die Attribute der Tabelle. Jedes Entity wird durch ein Tupel der Tabelle

repräsentiert. Der Primärschlüssel des Entitytyps verwandelt sich in den Primärschlüssel der Tabelle.

Transformation von Beziehungen

Jeder Beziehungstyp wird in eine Tabelle umgewandelt. Die Attribute des Beziehungstyps, die nicht Primärschlüssel sind, werden in die Attribute der Tabelle umgewandelt. Die Schlüsselattribute der Entitytypen, die in Beziehung mit diesem Beziehungstyp stehen, werden als Fremdschlüssel in der Tabelle hinzugefügt. In einigen Situationen muß man einige Attribute, die aus den Entitytypen übernommen werden, umbenennen, weil Attribute in unterschiedlichen Entitytypen einen gleichen Namen haben können. Bei der Transformation von N:M Beziehungen besteht der Primärschlüssel der zugehörigen Tabelle aus Zusammensetzung aller Fremdschlüssel.

Die so erzeugten Tabellen müssen noch modifiziert werden, indem einige der Tabellen, die bei der Umsetzung der Beziehungstypen entstehen, gelöscht werden. Aber dies ist nur für die Tabellen sinnvoll, die Beziehungen von 1:1, 1:N, oder N:1 darstellen. Die Tabellen, die Beziehung von N:M repräsentieren, werden normalerweise nicht gelöscht, da ansonsten „Anomalien" entstehen. Diese Modifikation heißt auch Normalisierung der Relationen. Eine Relation kann in eine bestimmte Normalform gebracht werden. Es gibt viele umfangreichen theoretischen Grundlagen, die Normalisierung beschreiben. Hier wird auf detaillierte Erläuterungen verzichtet.

Die Umsetzung von Generalisierung und Aggregation ins relationale Modell wird unzureichend unterstützt, da das relationale Modell diese Konzepte nicht kennt.

Fallbeispiele

Die in den Fallbeispielen gezeigten Lösungen sind nicht die einzig möglichen. Die Primärschlüssel sind unterstrichen und die Fremdschlüssel sind kursiv gedruckt. Bei Transformation der 1:N Beziehung wie Abbildung 7 kann man beispielsweise zwei Tabellen erzeugen. Der Primärschlüssel des Kunden-Entitytyps wird als ein Fremdschlüssel zu der durch den Aufträge-Entitytyp entstehende Tabelle als Attribut hinzugefügt.

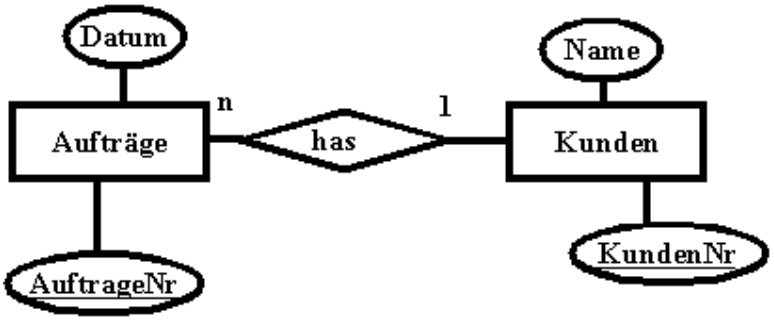

Abbildung 7: Beispiel für Transformation der 1:N Beziehungen

Die beiden Tabellen heißen dann: `Aufträge(AuftrageNr,Datum,`*`KundenNr`*`)` und `Kunden(KundenNr,Name)`.

Bei Transformation der 1:1 Beziehung kann man beispielsweise zwei Tabellen erzeugen. Abbildung 8 zeigt dafür ein Beispiel. Wenn zwei Tabellen benutzt werden, wäre folgende Lösung denkbar: `Hotel(HotelNr,Name,`*`ManagerNr`*`)` und `Manager(ManagerNr,Name)`.

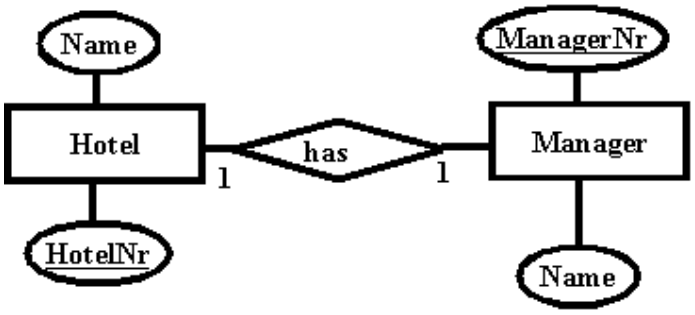

Abbildung 8: Beispiel für Transformation der 1:1 Beziehung

Bei Transformation der N:M Beziehung wäre folgende Lösung denkbar: Zwei Tabellen kommen aus der Umsetzung der zwei Entitytypen und eine Tabelle aus der Umsetzung des Beziehungstyps.

2.3.3 Structured Query Language (SQL)

Die Kurzform SQL steht für Structured Query Language. SQL wird in vielen relationalen Datenbanksystemen wie u.a. DB2, ORACLE, Sybase verwendet. SQL ist eine deklarative Sprache und kann interaktiv zwischen Terminal und Datenbanksystem oder eingebettet in einer Wirtssprache benutzt werden. Der Lernaufwand ist trotzdem nicht aufwendig.

SQL wurde 1986 bzw. 1989 normiert. Eine erweiterte Norm unter Abkürzung SQL2 wurde 1992 herausgegeben. Mitte 1999 ist SQL3 herausgegeben. SQL3 unterstützt objekt-orientierte Erweiterungen.

Zur Definition und Manipulation relationaler Datenbanken bietet SQL Anweisungen für Datendefinition, Datenanfragen, Datenerneuerung und Datenkontroll an. Die Anweisungen der SQL kann man in drei Gruppen unterteilen:

1) Datenmanipulation
2) Datendefinition
3) Datenkontrolle

Diese Gruppen von Anweisungen werden im folgenden kurz erläutert.

Datenmanipulation (DML)

Anweisungen der Data Manipulation Language (DML) dienen zum Hinzufügen, Ändern oder Löschen der Daten in der Datenbank. Es gibt drei Basisbefehle, INSERT(Einfügen), UPDATE (Erneuerung) und DELETE(Löschen). Bei einer Datenanfrage werden existierende Daten nach bestimmten Vorbedingung in der Datenbank durchgesucht. Die Basisstruktur der Anweisungen besteht aus SELECT, FROM und WHERE.

Datendefinition (DDL)

Anweisungen der Data Definition Language (DDL) dienen zum Erstellen, Ändern und Löschen der Datenbankenobjekte der Datenbank wie u.a. Table(Tabelle), View, Sicht,

Index. Ihre Basisanweisungen sind beispielsweise CREATE TABLE, CREATE VIEW, CREATE INDEX.

Datenkontrolle (DCL)

Anweisung der Data Controll Language (DCL) dienen dazu, den Benutzern Berechtigung auf der Datenbank zu geben oder aufzuheben. Ihre Basisanweisungen sind beispielsweise GRANT TO, REVOKE FROM.

2.3.4 Referentielle Integrität

In relationalen Datenbanken gibt es sogenannte referentielle Integrität. Die referentielle Integrität beschreibt die Abhängigkeiten zwischen den Tupeln in verschiedenen Tabellen. Beispielsweise besteht es eine referentielle Integrität zwischen Tabelle `Gebäude` und `Räume`, indem wenn ein Gebäude gelöscht wird, dann werden entsprechend die Zeile in Tabelle Räume gelöscht. Siehe Abbildung 5. Die Tabellen, die solche Abhängigkeiten haben, bezeichnet man entsprechend als Eltern- und Kindtabellen. Hier ist `Gebäude` eine Elterntabelle und `Räume` eine Kindtabelle.

2.3.5 Die Schwachpunkte der relationalen Datenbank

Relationale DBMS sind insbesondere für die sogenannten kaufmännischen Anwendungen, zum Beispiel Bank- und Versicherungswesen angepaßt. Für Nicht-Standard-Anwedungen, z.B. technische Anwendungen und Multimediaanwendungen, haben relationale Datenbank sich jedoch als weniger gut geeignet erwiesen[STS97].

Bei Nicht-Standard-Anwendungen haben die relationale Datenbanken folgende Schwachpunkte:

1. Keine komplexen Datenstrukturen
2. Keine komplexen Integritätsbedingungen
3. Keine effizienten navigierenden Zugriffe

4. Identifikationsprobleme von Anwendungsobjekten
5. Keine Beschreibung des Verhaltens der Objekte
6. Impedance Mismatch

Im folgenden werden diese Punkte jeweils kurz erläutert.

1. Keine komplexen Datenstrukturen

Die Anwendungsobjekte mit komplexen Datenstrukturen werden nicht direkt durch Tupel in flachen Relationen abgebildet, sondern zerlegt in verschiedene Relationen aufgeteilt. Beispielsweise besteht eine Maschinenkomponente aus Teilen, die wieder als eigenständige Datenobjekte betrachtet werden. Dann wird eine Maschinenkomponente in verschiedenen Tabellen abgebildet. Das hat zur Folge, daß bei einem Zugriff auf eine Maschinenkomponente eine Verbundoperationen erforderlich wird.

2. Keine komplexen Integritätsbedingungen

Die von relationalen DBMS unterstützten Integritätsbedingungen reichen nicht aus, um die sehr komplexen Integritätsbedingungen der Nicht-Standard-Anwendungen zu unterstützen. Beispielsweise gibt es technische Randbedingungen wie etwa Vermeidung von Kollisionen während des Fertigungsprozesses in Ingenieursystemen, die nur unzureichend unterstützt werden.

3. Keine effizienten navigierenden Zugriffe

Unter Navigation versteht man, daß der Zugriff von einem gegebenen Datenbankobjekt über Referenzen zu anderen Objekten geht. Bei Navigation zwischen relationalen Tabellen muß man Verbundoperationen durchführen. Diese kosten normalerweise viele Zugriffe.

4. Identifikationsprobleme von Anwendungsobjekten

Man muß einen Primärschlüssel definieren, um die Tupel einer relationalen Datenbank eindeutig zu identifizieren. Doch kann es zu Konflikten führen. Die Benutzer müssen

eindeutig für jedes Tupel einen Schlüssel erzeugen. In DB2 gibt es eine Funktion `GENERATE_UNIQUE()`, die einen eindeutigen Schlüssel systemweit generiert. Dies erleichtert dieses Problem.

5. Keine Beschreibung des Verhaltens der Objekte

In relationalen Datenbanken werden die Daten und das Verhalten der Objekte strikt getrennt. Das DBMS ist zuständig für die Verwaltung der Daten. Das Verhalten der Objekte wird in Anwendungsprogrammen implementiert. Es gibt aber Abläufe, die unabhängig von einzelnen Anwendungen sind. Deswegen sollten sie zentral behandelt werden können. Berechnung der Masse eines Produkts aus der Summe seiner Komponentenmassen ist ein Beispiel dafür.

6. Impedance Mismatch

Die Datenbankanwendungen werden mit einer Programmiersprache wie Java, C++ geschrieben und die Anwendungsprogramme benutzen die SQL-Schnittstelle von relationalen DBMS wie JDBC, um Interaktion mit der Datenbank durchzuführen. Die Datenbanksprache unterscheidet sich oft von der Programmiersprache. Die Unterschiede zwischen den Sprachen muß man durch manuelle Anpassung beseitigen, wenn keine kommerzielle Middleware zur Verfügung steht.

2.3.6 Ausblick

Trotzdem das relationale DBMS nicht geeignet ist für Nicht-Standard-Anwendung, wird es meistens in sogenannten „kaufmännischen Anwendungen" eingesetzt. Relationale DBMS bieten sehr gute Unterstützung für einfache Daten. In der Zukunft werden sie wahrscheinlich von objektrelationalen DBMS ersetzt.

2.4 Das objektrelationale Datenmodell

Das objektrelationale Datenmodell ist die Erweiterung der objektorientierten Funktionalitäten vom relationalen Datenmodell, um komplexe Objekte in der Datenbank zu speichern, damit man Vorteile von beiden Modellen benutzen kann.

Objektrelationale Datenbankmanagementsysteme (DBMS) können in vielen Anwendungsgebieten eingesetzt werden. Ihr Einsatz reicht von Multimediaanwendungen und geographischen Informationssystemen bis kaufmännische Anwendungen wie Rechnungswesen und Versicherungswesen. In Unterabschnitt 2.4.1 werden die Hauptmerkmale eines objektrelationalen DBMS kurz vorgestellt.

2.4.1 Merkmale der objektrelationalen Datenbank

Diese vier wichtige Merkmale eines objektrelationalen DBMS sind [SM99]:

- Unterstützung für die Erweiterung der Basisdatentypen in einem SQL-Kontext
- Unterstützung für komplexe Objekte in einem SQL-Kontext
- Unterstützung für die Vererbung in einem SQL-Kontext
- Unterstützung für ein Regelsystem

Anschließend werden diese vier wichtige Hauptmerkmale in den nächsten vier Unterschnitten vorgestellt.

2.4.1.1. Erweiterung der Basisdatentypen

Das erste Hauptmerkmal eines objektrelationalen DBMS ist die Möglichkeit, neue Datentypen mit entsprechenden Operationen und Funktionen zu erzeugen. Da bei relationalen Datenbanken die Menge der Datentypen und Operationen beschränkt ist, ist es schwierig, einige Probleme zu codieren. Beispielsweise gehört ein zweidimensionaler Punkt nicht zu den Basisdatentypen. Um zwei Punkte addieren zu können, muß man dafür programmieren. Oft führt dies zu schlechter Performance. Mit der Erweiterung der Basisdatentypen wird meistens eine Verbesserung der Performance und der Entwicklungszeit erreicht.

Ein neuer, benutzerdefinierter Datentyp wird erzeugt, indem der Name des Typs, Speicherinformationen über den Typ und zwei Input- und Output-Umwandlungsroutinen angegeben werden. Jeder Typ hat eine interne (als welcher Datentyp wird der neue Datentyp in der Datenbank gespeichert) und eine externe (ASCII) Darstellung. Input- und Output-Umwandlungsroutinen dienen zur Umwandlung der internen Darstellung des Typs in die externe Darstellung oder der externen in die interne Darstellung.

Die typspezifischen Operationen werden für benutzerdefinierte Datentypen hinzugefügt, damit die neuen Daten der Datentypen manipuliert werden können. Diese werden durch Funktionen realisiert. Die Funktionen in einem objektrelationalen DBMS werden entweder in SQL oder C oder JAVA geschrieben und anschließend im System angemeldet. Bei der Anmeldung einer Funktion muß der Name der Funktion, ihre Argumente, ihr Rückgabetyp und der Ausführungscode angegeben werden.

Ein Datenbanksystem muß aber noch mehr bestimmte Eigenschaften unterstützen damit die Typenerweiterung voll objektrelational ist. Dies ist im folgenden vorgestellt.

Dynamische Bindung

Ein DBMS bindet eine benutzerdefinierte Funktion, nur wenn sie benötigt wird, ein. Ansonsten liegen die benutzerdefinierten Funktionen nicht im DBMS-Adressraum. Dies bringt Vorteile gegenüber statischer Bindung. Bei statischer Bindung muß das System heruntergefahren, neu gebunden und neu installiert werden, wenn man einen neuen Datentyp, einen Operator oder eine Funktion hinzufügen möchte. Dies dauert manchmal Tage und ist sehr unpraktisch für die Benutzer.

Client- oder Serveraktivierung

Wo die Funktion aktiviert werden ist die zweite Eigenschaft. Wenn die Funktion im Server im selben Adressraum wie das DBMS aktiviert wird, ist der Aufruf ein lokaler Aufruf und verursacht wenig zusätzlichen Aufwand. Wenn die Funktion in einem eigenen Adressraum oder auf einer anderen Maschine aufgerufen wird, muß RPC (remote

procedure call) benutzt werden und die Argumente von Adressraum des Systems in einen eigenen Adressraum kopiert werden. Dadurch entsteht hoher Aufwand, was sich in der Performance niederschlägt. Aber für die sehr teueren Berechnungen sind die Kosten eines RPC gegenüber den Kosten eines lokalen Aufrufs fast gleich. In diesem Fall ist der Funktionsaufruf in einem eigenen Adressraum möglich.

Ein gutes objektrelationales Datenbanksystem unterstützt beide Möglichkeiten der Aktivierung. In der Realität entscheidet man sich für einen für die Anwendung sinnvollen Funktionsaufruf.

Sicherheit

Bei serverseitiger Aktivierung läuft die Funktion in einem Prozeß mit der Benutzeridentifikation des Systems. Bei clientseitiger Aktivierung läuft die Funktion in einem Prozeß mit der Benutzeridentifikation des Anwenders. Wenn eine fehlerhafte Funktion illegale Anweisungen durchführt oder zu einer nicht existierenden Adresse springt, stützt der Client-Prozeß bei clientseitiger Aktivierung ab, sogar die ganze mit dem System verbundene Benutzergruppe bei serverseitiger Aktivierung.

Um das Problem zu beseitigen gibt es verschiedene Verfahren. Ein Verfahren ist, die serverseitige Funktion in einem vom DBMS getrennt Prozeß zu aktivieren. Dies führt aber zur Einführung eines RPCs und damit zum Performance-Verlust. Ein anderes Verfahren ist die Benutzung von Firewalls (Brandmauern). Das Firewall untersucht den Code der benutzerdefinierten Funktionen während der Laufzeit. Das letzte Verfahren ist, Funktionen mit einer Interpretersprache, wie Java zu schreiben. Trotzdem die Leistung von Interpretersprachen sehr viel schlechter als Compilesprachen wie C ist, haben sie den Vorteil, daß sie die Grenzen der während der Ausführung erzeugten Adressen überprüfen können.

Rückruf

Ein Rückruf ist ein Prozeß, der die Aktivitäten einer Transaktion zurücksetzt. Die von einer benutzerdefinierten Funktion aufgerufene SQL-Anweisung, die einen Rückruf

durchführt, kann wieder eine benutzerdefinierte Funktion enthalten, die ebenfalls Rückrufe durchführt. Deswegen muß ein objektrelationales DBMS die Schachtelung von Rückrufen unterstützen.

Benutzerdefinierte Zugriffsverfahren

Ein objektrelationales DBMS muß auch neue Zugriffsverfahren für benutzerdefinierte Datentypen unterstützen. Beispielsweise für den zweidimensionalen Datentyp Punkte muß man statt einem eindimensionalen Verfahren wie B-Baum ein zweidimensionales Zugriffsverfahren wie R-Baum oder Quad-Baum benutzen. Das Hinzufügen einer Zugriffsmethode muß durch einen erfahrenen Programmierer erfolgen.

Typen beliebiger Länge

Es gibt Datentypen wie ein Bildtyp, die von fester Länge und lang sind. Es gibt auch Datentypen wie eine komprimierte Darstellung eines Bildes, die von variabler Länge sind. Deswegen ist die letzte Anforderung an objektrelationale DBMS im Zusammenhang mit benutzerdefinierten Datentypen, keine Längeeinschränkungen für die benutzerdefinierte Datentypen festzulegen.

2.4.1.2 Komplexe Objekte

Eine zweite Anforderung an objektrelationales DBMS ist die Unterstützung für komplexe Objekte in SQL, die aus mehreren Basis- oder benutzerdefinierten Typen zusammengesetzt werden. Es gibt drei Typkonstruktoren für die Erzeugung komplexer Objekte: Verbunde (Sätze), Mengen und Referenzen.

- **Verbunde als Typkonstruktor**

Ein Verbund ist ein Datentyp, welcher wieder aus einem Satz von Werten besteht. Wenn man ein komplexes Objekt mit Verbund als Konstruktor konstruiert, gibt man den Name des komplexen Objektes zusammen mit dem Namen und Datentyp der einzelnen Teile an. Die Teile eines komplexen Objektes können wieder zusammengesetzte Typen sein.

- **Menge als Typkonstruktor**

Wenn T ein Typ ist, dann muß auch `setof`(T) ein Datentyp sein. Der Typ von T kann Basisdatentyp oder Verbund sein.

- **Referenz als Typkonstruktor**

Eine Referenz ist ein Zeiger. Wenn T ein Typ ist, dann muß ebenfalls `ref`(T) ein Datentyp sein.

Bei Manipulation von Verbunden und Mengen von Verbunden, wird der Standard-SQL in einem gewissen Grad erweitert. Die Benutzerdefinierte Funktion sollte Verbunde und Mengen von Verbunden als Argument übernehmen oder als Ergebnisse der Funktion zurückgeben. Solche Funktionen können in der from-Klausel einer SQL-Anfrage eingebettet werden. Die „Kaskaden-Punkt-Notation" referenziert Attribute von Verbunden oder Mengen von Verbunden, zum Beispiel, **Name des Verbundes oder der Menge von Verbunden . Name des Attributes.** Verbunde und Mengen von Verbunden eignen sich für Daten, die selten geändert werden oder einmal auftreten.

Referenzen sind ähnlich wie die Primärschlüssel-Fremdschlüssel-Beziehung. Eine Referenz ist ein Zeiger, der auf einen Satz eines bestimmten Typs in einer anderen Tabelle verweist. Jede Zeile der Tabelle hat einen eindeutigen Objektbezeichner (OID), der nie geändert wird. Die Referenz speichert den OID, auf den die Referenz verweist. Für die Unterstützung der Referenzen benötigt man eine Erweiterung von SQL. Die Funktion `deref` nimmt eine Referenz als Argument und gibt einen Satz des Typs, auf den die Referenz verweist, zurück. Die Funktion `ref` nimmt ein Argument, das eine Instanz eines Typs ist, und gibt eine Referenz auf die Instanz zurück.

Es gibt vier Gründe dafür, warum ein objektrelationales DBMS zwei Mechanismen, nämlich erweiterbare Basistypen und komplexe Objekte, bieten. Die vier Gründe sind:

- **Natürlichkeit**
- **Kapselung**
- **Objektidentifikation**
- **Datenumwandlung**

Im folgenden werden diese vier Gründe kurz vorgestellt.

Natürlichkeit

Manchmal sind Basistypen für den Benutzer natürlich zu verstehen wie Bilder, positive Zahlen. Manchmal sind Verbunde natürlich zu verstehen wie ein Objekt „Abteilung”.

Kapselung

Basistypen sind vollständig gekapselt. Wenn man die Basistypen manipulieren möchte, kann man nur sie abrufen oder eine Funktion ausführen, die diesen Typ als Parameter benutzt. Umgekehrt sind komplexe Objekte transparent, also sind ihre Felder sichtbar und durch die Anfragesprache manipulierbar. Die interne Repräsentation eines Basistyps kann man beliebig ändern, ohne die Programme, die diesen Basistyp benutzen, zu ändern. Das ist ein Grund für die Kapselung. Gegenteilig kann man die sichtbaren Felder eines komplexen Objektes nicht ändern, weil die Programme, die diesen Typ benutzen, nicht mehr funktionieren. Deswegen eignen sich gekapselte Daten für Basistypen und nicht gekapselte Daten für komplexe Objekte.

Objektidentifikation

Komplexe Objekte haben einen vom System generierten Objektidentifikation (OID) während ein Basistyp keinen hat. Weil eine Referenz ein OID enthält, sind Referenzen nur für komplexe Datentypen verfügbar.

Datenumwandlung

Es existieren viele Datentypen, zum Beispiel integer, die unterschiedliche externe und interne Datenformate erfordern. Die für Basistypen verfügbaren Ein- und Ausgabenfunktionen unterstützen Umwandlung der Datenformate. Für komplexe Objekte muß man Ein- und Ausgabefunktionen sowie Operatoren definieren.

2.4.1.3 Vererbung

Die Unterstützung der Vererbung von Supertypen zu Subtypen ist das dritte Hauptmerkmal des objektrelationalen DBMSs. Zur Vererbung zählen Datenvererbung und Funktionsvererbung. Die beiden werden im folgenden erläutert.

Datenvererbung

Nur auf zusammengesetzten Typen ist Datenvererbung anwendbar. Alle Datenfelder von einem Supertyp werden von seinen Subtypen geerbt. Die zusammengesetzten Typen kann man in eine Vererbungshierarchie gruppieren. Mehrfache Vererbung wird auch teilweise von objektrelationalen DBMS unterstützt. Dies bedeutet, daß ein Subtyp Datenfelder von mehreren Supertypen erben kann. Bei mehrfacher Vererbung existiert das Problem von Mehrdeutigkeit. Mehrdeutigkeit kommt vor, wenn Supertypen ein Feld haben, das einen gleichen Namen mit unterschiedlicher Bedeutung hat. Um dieses Problem zu vermeiden kann man eins der Felder in einem der Supertypen neu definiert.

Den Tabellen müssen die Typen zugeordnet werden, damit die Instanzen von Typen in den Tabellen gespeichert werden können. Vererbung ist nur für Datentypen möglich. Ohne die Zuweisung der Typen an Tabellen können die Tabellen nicht erben. Wie die Typenhierarchie kann man auch die auf einer Typenhierarchie basierenden Tabellen in eine Vererbungshierarchie gruppieren.

Vererbung von Funktionen

Ein Subtyp erbt die Funktionen automatisch von seinen Supertypen. Wenn es bei Aufruf einer Funktion auf einem Subtyp keine Funktion mit gleichem Namen und Parameter für diesen Subtyp gibt, dann sucht das DBMS die ganze Typenhierarchie nach einem Supertyp durch, für den die Funktion mit gleichem Namen und Parameter definiert ist. Wenn es klappt, wird die gefundene Funktion benutzt. Eine Funktion mit gleichem Namen aber mit unterschiedlichen Definitionen kann auf verschiedenen Typen definiert werden. Zur Laufzeit muß das System die richtige Funktion finden. Man nennt dieses Konzept *Polymorphismus*. Ein Subtyp kann auch von seinen Supertypen unterschiedlich definierte

Funktionen mit gleichem Namen erben. Man nennt dieses Problem Mehrdeutigkeit.
Um dieses Problem zu lösen kann man die Funktionen mit unterschiedlichen Namen umbenennen.

2.4.1.4 Regeln

Das letzte Hauptmerkmal für objektrelationales DBMS ist ein Regelsystem. Regeln schützen die Datenintegrität in einer Datenbank. Bei relationalen Systemen werden Regeln durch Trigger unterstützt. Objektrelationale Systeme verlangen nach einem flexiblen Regelsystem. Die übliche Darstellung einer Regel ist
on event do action
Ein Regelsystem erwartet bestimmte Ereignisse. Wenn die Ereignisse vorkommen, dann führt das System die entsprechende Aktion aus. Die Aktion kann vor oder nach der Verarbeitung des Ereignisses ausgeführt werden. Es gibt vier Variationen des Grundparadigmas:

- Aktualisierung-Aktualisierung-Regeln
- Abfrage-Aktualisierung-Regeln
- Aktualisierung-Abfrage-Regeln
- Abfrage-Abfrage-Regeln

Anschließend werden diese Grundparadigma und Nachteile von Regeln im folgenden kurz vorgestellt.

Aktualisierung-Aktualisierung-Regeln

Bei Aktualisierung-Aktualisierung-Regeln ist das Ereignis eine Aktualisierung und die Aktion eine andere Aktualisierung. Diese Art der Regel unterstützt Datenintegrität in einer Datenbank. Wenn ein Ereignis die Datenintegrität verletzt, dann kann die entsprechende Korrektur als ein Teil der Aktualisierungsaktion durchgeführt werden.

Abfrage-Aktualisierung-Regeln

Bei Abfrage-Aktualisierung-Regeln ist das Ereignis eine Abfrage und die entsprechende Aktion eine Aktualisierung in der Datenbank.

Aktualisierung-Abfrage-Regeln

Bei Aktualisierung-Abfrage-Regeln ist das Ereignis eine Aktualisierung und die entsprechende Aktion Erzeugung einer Antwort an den Benutzer. Diese Art der Regel wird üblich als „Alarm" genannt, weil die Aktion einen „alerter" benachrichtigt und der alerter legt fest, wie ein Client von diesem Alarm erfährt. Ein Client kann auf eine oder mehrere Alarmnachrichten warten. Wenn Alarmnachrichten, auf die ein Client wartet, vorkommen, dann startet der Client Abfragen in der Datenbank.

Abfrage-Abfrage-Regeln

Bei Abfrage-Abfrage-Regeln ist das Ereignis eine Abfrage und die entsprechende Aktion ebenfalls eine Abfrage. Durch den Einsatz dieser Art der Regel kann man ein Sicherheitssystem bauen, indem falsche Daten zurückgeliefert werden, wenn ein Benutzer nach ihm nicht autorisierten Daten abfragt.

Nachteile von Regeln

Trotzdem die Regeln sehr leistungsfähig sind, haben sie auch Schwächen, die hier im folgenden beschrieben werden.

1. **Mehrere Regeln, die dasselbe Ereignis auslösen, erzeugen manchmal unvorhersehbare Ergebnisse.**

Eine Anzahl von Regeln kann beliebig definiert werden und verschiedene Regeln von denselben Ereignissen ausgelöst werden. Es existieren Probleme damit. Beispielsweise wird ein Datenelement möglicherweise von zwei Regeln auf zwei unterschiedliche Werte

gesetzt. Die Regeln, die von gleichen Ereignissen ausgelöst werden können, müssen vom System in einer bestimmten Reihenfolge durchgeführt werden.

2. Kettenregeln können Endschlosschleifen verursachen

Man sollte vermeiden, daß endlose Schleifen von Kettenregeln konstruiert werden. Der Aktionsteil einer Regel kann so konstruiert werden, daß er die nächste Regel auslöst. Wenn die ausgelöste Regel im Beginn der Kette steht, dann wird es zu einer Endlosschleifen führen. Ein objektrelationales DBMS muß diese Situation feststellen und darauf reagieren, wie z.B. die Schleife nur einmal auszuführen.

3. Das Beenden des Aktionsteils einer Regel bricht die gesamte Transaktion ab

Normalerweise wird ein Aktionsteil einer Regel in derselben Transaktion ausgeführt, welche die Regel ausgelöst hat. Wenn die Transaktion beendet wird, dann wird der Aktionsteil abgebrochen. Dieser Effekt ist bei Aktualisierung-Aktualisierung-Regel erwartet. Aber es gibt Situationen, in denen man den Effekt nicht erwartet. Bei der Abfrage-Aktualiserung-Regel beispielsweise hat ein Benutzer schon eine Antwort auf seine Anfrage bekommen und im Aktionsteil der Regel wird die Transaktion abgebrochen. Das ist nicht erwünscht. Um das zu vermeiden, sollte der Aktionsteil der Regel in einer unabhängigen Transaktion ausgeführt werden. Diese Transaktion ist unabhängig von dem Ergebnis der Benutzertransaktion.

4. Es ist wichtig zu wissen, wann Regeln ausgelöst werden

Die Regeln werden in den meisten Systemen entweder direkt vor oder nach dem Vorkommen des Ereignisses ausgelöst. Dies heißt sofortige Ausführung. Aber in einigen Situationen werden Regeln nur ausgelöst, wenn die Transaktion, die die Regel auslöst, ausgeführt wird.

2.5 Klassifikation von Datenbankmanagementsystemen

Basierend auf die Diskussion in vorherigen Abschnitten wird in diesem Abschnitt eine Klassifikation von Datenbank-Management-Systemen (DBMS) vorgestellt.

Abbildung 9 zeigt die Klassifikation von DBMS nach ihren Anwendungen. Jeder Quadrant veranschaulicht einen von vier allgemeinen Typen von DBMS. Die vertikale Achse zeigt, ob die Anwendungen Anfragen benötigen. Die horizontale Achse zeigt links einfache Daten und rechts komplexe Daten[SM99].

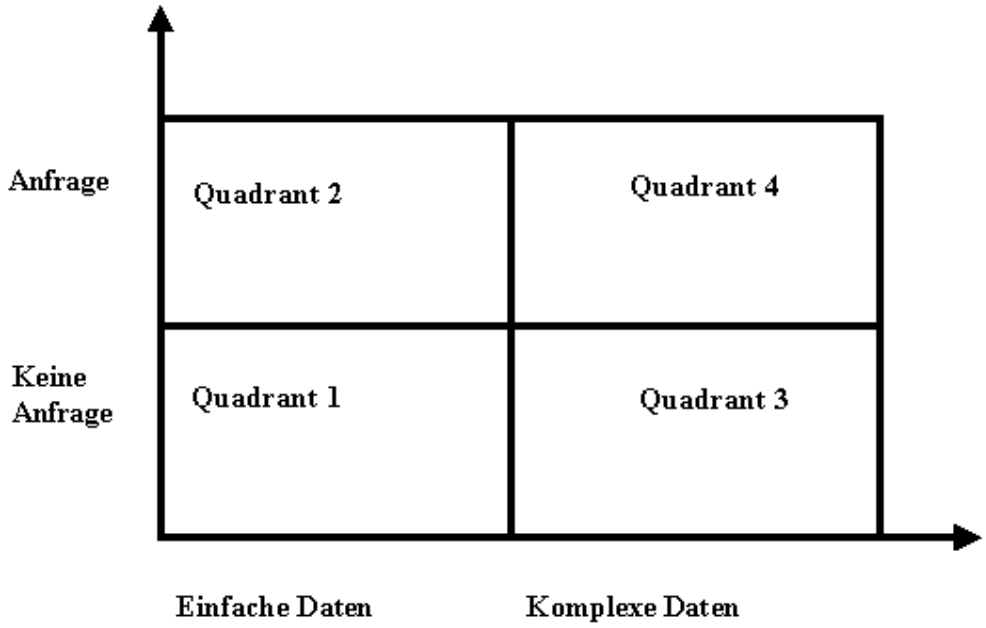

Abbildung 9: Eine Matrix zur Klassifikation von DBMS

Quadrant 1: Einfache Daten ohne Anfragen

Das typische, naheliegende DBMS für einfache Daten ohne Anfragen ist das Dateisystem eines Betriebssystems. Ein Anwendungsbeispiel ist ein Standard-Textverarbeitungssystem wie Word. Man kann eine Datei unter Word öffnen, verarbeiten und wieder schließen. Die einzige „Anfrage" ist „hole Datei" und die einzige „Erneuerung" ist „schreibe Datei". Man benötigt kein SQL. Die Datenstruktur ist auch einfach, da es sich um eine Hierarchie von Dateien handelt. Dateisysteme bieten schon genügende Unterstützung, wenn man keine Anfragen und keine komplexe Datenstrukturen braucht.

Quadrant 2: Einfache Daten mit Anfragen

Das typische DBMS für einfache Daten mit Anfragen ist das relationale Datenbanksystem, z.B. DB2, Oracle. Das DBMS hat eine Anfragesprache wie z.B. SQL2 (oder SQL-92) damit man die Daten in der Datenbank anlegen, erneuern und durchsuchen kann. Die Basisdatenstrukturen sind Tabellen. Die Anwendungen gehören gewöhnlich in die „kaufmännischen Datenverarbeitung" wie Versicherungswesen und Rechnungswesen.

Quadrant 3: Komplexe Daten ohne Anfragen

Die typischen Systeme für komplexe Daten ohne Anfragen sind objektorientierte Datenbanksysteme wie Objectstore, O2, Servio. Die Daten hier sind komplex. Die Basisdatenstrukturen hier sind Objekte und Objekte lassen sich durch Beziehungen verbinden. Wenn ein Anwendungsprogramm, das in einer persistenten und objektorientierten Programmiersprache geschrieben ist, beendet wird, werden die Objekte mit ihren Attributen in der Datenbank gespeichert. Die Anwendungen benötigen keine Anfragesprache.

Quadrant 4: Komplexe Daten mit Anfragen

Die typischen Systeme für komplexe Daten mit Anfragen sind objektrelationale Datenbanksysteme wie Illustra, Omniscience, UniSQL und Hewlett-Packard. Die Anfragesprache ist meistens der neue SQL-Standard (SQL3). Ein typisches Anwendungsbeispiel ist Multimediaanwendung wie „Suche Bilder, die Stuttgart bei Sonnenuntergang betreffen". Die Datenstrukturen sind komplex, z. B. Bilder, Audio und Video.

In der Praxis klassifiziert man das zu lösende Problem in einen der vier Quadranten und benutzt ein für den Quadranten optimiertes DBMS. Es gibt durchaus Anwendungen, die in mehr als einen Quadraten gehören, zur Diskussion dieser Anwendungen sei auf [SM99] verwiesen.

Zum Schluß folgt eine kurze Zusammenfassung der objektorientierten, relationalen und objektrelationalen Datenbanken. Objektorientierte Datenbanksysteme bieten eine sehr

gute Unterstützung komplexer Daten. Sie ergänzen die objektorientierten Programmiersprachen, um komplexe Objekte in der Datenbank zu speichern. Aber es fehlt den meisten objektorientierten Datenbanksystemen eine effiziente Anfragesprache wie SQL von relationalen Datenbanksystemen.

Relationale Datenbanken bieten schon sehr gute Unterstützung für flache Daten und haben eine effiziente Anfragesprache aber schlechte Unterstützung für komplexe Daten wie Objekte.

Bei den objektrelationalen Datenbanken werden die beiden Vorteile von relationalen Datenbanken und objektorientierten Datenbanken kombiniert. Objektrelationale Datenbanken unterstützen komplexe Anfragen und Daten. Mit der schnellen Entwicklung von Webtechnologie und Medientechnik liegen sie im Trend.

2.6 Entwicklungsumgebung

In diesem Abschnitt wird die Entwicklungsumgebung kurz vorgestellt. Wie in Kapitel 1 erwähnt werden JAVA, JDBC und DB2 in dieser Studienarbeit eingesetzt, die jeweils in den nächsten Unterabschnitten vorgestellt werden. In Abschnitt 2.6.1 wird JAVA vorgestellt. In Abschnitt 2.6.2 wird JDBC vorgestellt. In Abschnitt 2.6.3 wird DB2 vorgestellt. In Abschnitt 2.6.4 wird Zusammenspiel von JAVA, JDBC und DB2 diskutiert.

2.6.1 JAVA

Die objektorientierte Programmiersprache Java ist eine wichtige Programmiersprache in der IT-Branche. Java wurde von SUN Microsystems entwickelt. Wichtige Eigenschaften sind:

Einfachheit : Java ist einfach zu lernen. Java hat einen automatischen Mechanismus für Garbbage-collection. Die Programmierer brauchen sich nicht darum zu kümmern.

Objektorientierung: Java ist eine echte objektorientierte Sprache und hat alle Eigenschaften einer objektorientierten Programmiersprache, wie u.a. Einkapselung, Vererbung, Polymorphie.

Plattformunabhängigkeit: Beim Kompilieren von Java-Programme wird Bytecode erzeugt. Die Bytecode wird auf den verschiedenen Plattformen durch JAVA Virtuelle Maschine ausgeführt.

Andere wichtige Eigenschaften von Java sind beispielsweise Unterstützung von Multithreading, und gute Sicherheit. Bei der relationalen Datenbankentwicklung mit Java benutzt man häufig JDBC. Im nächsten Abschnitt wird es einen kurzen Überblick über JDBC gegeben.

2.6.2 JDBC

Die Kurzform JDBC steht für „Java Database Connectivity”. JDBC ist ein geschütztes Warenzeichen. JDBC ist die Schnittstelle zur Verbindung von relationalen Datenbanken

mit Java-Programmen. Die JDBC API hat eine einheitliche Schnittstelle für fast alle üblichen Datenbanksysteme und erlaubt Java-Entwicklern direkt Anwendungsprogramme, die unabhängig vom Datenbanksystem sind, zu schreiben, zum Beispiel, man hat ein Anwendungsprogramm für DB2 geschrieben und kann auch im Prinzip das Programm für Oracle benutzen. Mit JDBC kann man in den Anwendungsprogrammen durch SQL-Anweisungen auf die relationale Datenbank zugreifen. JDBC basiert auf dem Client-Server Mechanismus.

Um mit JDBC arbeiten zu können, braucht man das Package `java.sql` und Treiberklassen. Das Package `java.sql` enthält die Grundklassen von JDBC. Wichtige Klassen in diesem Package sind `java.sql.DriverManager`, `java.sql.Connection`, `java.sql.Statement` und `java.sql.ResultSet`. Die Klasse `DriverManager` sorgt für den Aufbau der Verbindung mit der Datenbank. Die Klasse `Connection` repräsentiert eine Verbindung mit der Datenbank. Die Klasse `Statement` erzeugen die SQL-Anweisungen. Die Klasse `ResultSet` speichert die zurückgelieferten Ergebnisse der SQL-Anweisungen. Die Treiberklassen sind zuständig für die Kommunikation mit dem Datenbanksystem und unterscheiden sich je nach Hersteller.

Die Abbildung 10 zeigt, wie man mit JDBC arbeiten kann. Man muß zuerst den Treiber durch die Klasse `DriverManager` laden. Anschließend baut man mit der Klasse `Connection` eine Verbindung zur Datenbank auf. Jetzt kann man durch Klasse `Statement` eine SQL-Anweisung aufbauen und die Anweisung durchführen. Schließlich liest man das Resultat der Anweisung in das Programm zurück und verarbeitet es weiter.

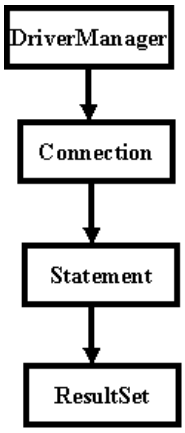

Abbildung 10: **Überblick über dem JDBC-Mechanismus**

2.6.3 DB2 Universal Database

Einleitung

DB2 ist ein relationales DBMS und wurde von IBM 1983 erstmals entwickelt und wird ständig erneuert. Seit Version 7.0 wird es schon um objektrelationale Funktionen erweitert und bietet nicht-traditionelle Datentypen wie BLOB, CLOB und User Defined Type (UDT) an. Mit UDT kann man komplexe Objekttypen definieren. Diese Objekttypen haben sowohl komplexe Datenstruktur als auch User Defined Functions (UDF) zum Verwalten des Verhalten des Objekttyps. Die Version 7.0 von DB2 unterstützt noch nicht alle Datentypen vom neuem SQL-Standard SQL3, wie u.a. Array. DB2 kann auf den üblichen Betriebssystemen, wie u.a. UNIX, WINDOWS, LINUX laufen. Es basiert auf dem Client-Server Mechanismus. Die Server verwalten Datenbanken und die Clients sind zuständig für Anwendungsprogramme, die SQL-Anweisungen erzeugen. DB2 unterstützt auch verschiedene Schnittstellen zwischen Anwendungsprogrammen und Datenbanken.

User Defined Type (Strukturierte Typen)

In diesem Unterabschnitt wird UDT in DB2 kurz vorgestellt. Strukturierte Typen in DB2 haben folgende Merkmale:

- Ein strukturierter Typ kann von einem Vatertyp erben oder als Vatertyp an einen anderen strukturierten Typ vererben.
- Instanzen von strukturierten Typen können entweder als Reihen in einer typisierten Tabelle oder als ein Wert in einer Spalte einer Tabelle gespeichert werden. Die typisierten Tabellen müssen den Typen zugeordnet werden.

Für jeden strukturierten Typ erzeugt DB2 einen *companion type*. Dieser *companion type* wird Referenztyp genannt. Der strukturierte Typ, auf den der Referenztyp zeigt, heißt *referenced type*. Die SQL-Anweisung REF(*type_name*) erstellt einen *referenced type*, dessen Name vom *type_name* repräsentiert wird. In DB2 wird eine Spalte, die die Objektidentifikation speichert, automatisch erzeugt. Die Objektidentifikation identifiziert eindeutig eine Zeile, also eine Instanz eines Typs in der typisierten Tabellenhierarchie. Der Typ

dieser Spalte ist vom Referenztyp (Siehe Abschnitt 2.4.1.2). Der Referenztyp wird benutzt, um die Referenz auf die Zeile der typisierten Tabellen zu speichern. Abbildung 11 zeigt ein Beispiel dafür[IBM00c].

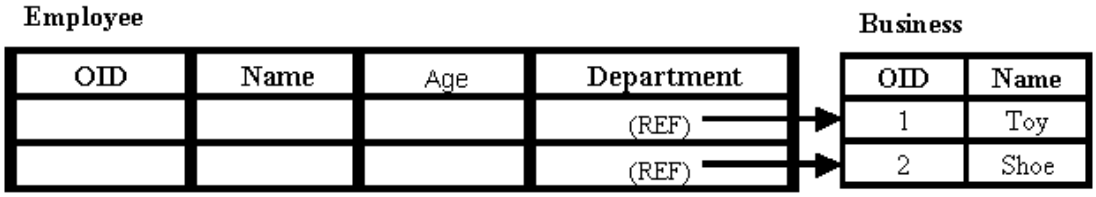

Abbildung 11: Beispiel für Referenztyp

In der Abbildung 11 speichert Tabelle `Employee` die Instanzen vom Typ `Employee_t`. Tabelle `Business` speichert die Instanzen des Typs `Business_t`. Das Attribut `Department` von `Employee` wird als Referenztyp definiert und verweist auf die Instanzen der Tabelle `Business`. Das Attribut `OID` von Tabelle `Employee` und Tabelle `Business` wird von DB2 automatisch erzeugt.

2.6.4 Zusammenspiel von JAVA, JDBC und DB2

Abbildung 12 zeigt das Zusammenspiel von JAVA-Anwendungsprogrammen, JDBC und DB2. Die JDBC-Schnittstelle baut eine Brücke zwischen Anwendungsprogrammen und DB2 auf. Die Datentypen von JAVA werden durch JDBC mit SQL-Datentypen von DB2 abgebildet. DB2 unterstützt nicht das Abspeichern von Java-Objekten direkt in der Datenbank. Die Speicherung von Java-Objekten erfolgt in dieser Studienarbeit durch manuelle Abbildung.

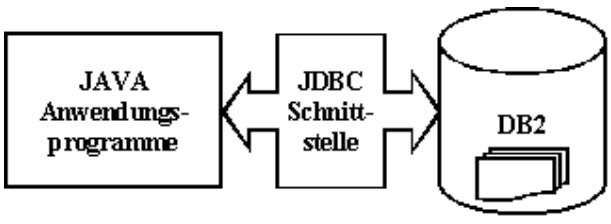

Abbildung 12: Rolle von JDBC

3. Implementierung

In diesem Kapitel werden Implementierungen, Modellierung und Performancetest erläutert. Dieses Kapitel wird in sechs Abschnitte unterteilt. Abschnitt 3.1 beschreibt die Anforderungsanalyse. In Abschnitt 3.2 wird die vorhandene objektorientierte Implementierung vorgestellt. In Abschnitt 3.3 wird die relationale Modellierung erläutert. In Abschnitt 3.4 wird die objektrelationale Modellierung erläutert. In Abschnitt 3.5 wird die Implementierung der JDBC-Schnittstelle erläutert. Schließlich in Abschnitt 3.6 werden die Performancetests erläutert. Wir beginnen mit der Anforderungsanalyse.

3.1 Anforderungsanalyse

Die Anforderungsanalyse ist der Startpunkt der Implementierung. In der Anforderungsanalyse dieser Studienarbeit werden die Gespräche mit den Projektpartnern und die Dokumentation der vorhandenen objektorientierten Implementierung mit JAVA und Objectstore benutzt. Es ergeben sich folgende Punkte:

- Die JDBC-Schnittstelle sollte die grundlegenden Datenmanipulationen, nämlich INSERT, DELETE und UPDATE leisten.
- Das Semantic Net Interface (siehe 3.2) sollte realisiert und möglichst nicht geändert werden. Die Vererbungshierarchie der objektorientierten Implementierung sollte gleich bleiben.
- Die Objektidentifikation(OID) sollte von `GENERATE_UNIQUE()`erzeugt werden.

Nach [Lit01] ist die Definition des semantischen Netzes wie folgt:

1. **Nodes**: Sie enthalten unterschiedliche Datentypen oder wiederum ein Netz. Beispielsweise eine XML-Datei, etc. Jeder Node gehört zu einem Nodetyp. Nodetypen können hierarchisiert werden.
2. **Relationen**: Relationen sind im Gegensatz zu Nodes nicht eigenständig existenzfähig. Relationen definieren die Semantik eines semantischen Netzes, in dem sie die Verbindung zwischen zwei Knoten beschreiben. Die Relation als Beschreibungsmittel muß so allgemein anwendbar sein, daß sie zwei beliebige Elemente eines Netzes

verbinden kann. Das bedeutet, daß Relationen zwei Knoten, einen Knoten und eine Relation oder zwei Relationen miteinander verbinden können.

3. **Relationen**: Unterschiedliche Relationen können die gleiche Semantik haben und damit zu einem Relationstyp abstrahiert werden. Also wird eine Relation einem Relationstyp zugeordnet.
4. **Relationstypen** können hierarchisiert werden. Beispielsweise ist mechanische Aggregation eine spezialisierte Aggregation.
5. Von einem Relationstyp kann zwischen zwei Nodes nur eine Relation hergestellt werden.
6. Jede Relation hat eine Richtung. In der Graphik wird die Richtung veranschaulicht, indem ein Pfeil, der von einem Node oder einer Relation ausgeht, auf einen Node oder eine Relation zeigt. Siehe Abbildung 13. Zu Relationen können wiederum Relationen hergestellt werden. Siehe Abbildung 13.
7. Jeder Node ist genau einmal **Layer** sowie **Package** zugeordnet. Es gibt beliebig viele Layer und Packages. Layer ist Abstraktionsmechanismus von Datenmodell, mit dem die Nodes und Relationen in Klassen und Instanzen klassifiziert werden. Die Bedeutung von Package hier ist ähnlich wie der Begriff von Package der Java-Programmiersprache. Ein Package ist Strukturierung von Nodes. Nodes, die miteinander im Zusammenhang stehen, werden in einem Package gepackt.

Wir betrachten zuerst ein Beispiel eines semantischen Netzes. In der Abbildung 13 sind Node1, Node2 und Node3 flache Nodes. Ein flacher Node enthält keine weitere Netze. Node4 ist ein Covernode, also enthält wiederum ein Netz. Die Nodes können auch Layer und Package zugeordnet werden. Die Relationen verbinden die Nodes und die Relationen. Relation1 verbindet eine Relation mit einem Node. Relation3 verbindet eine Relation mit einer Relation.

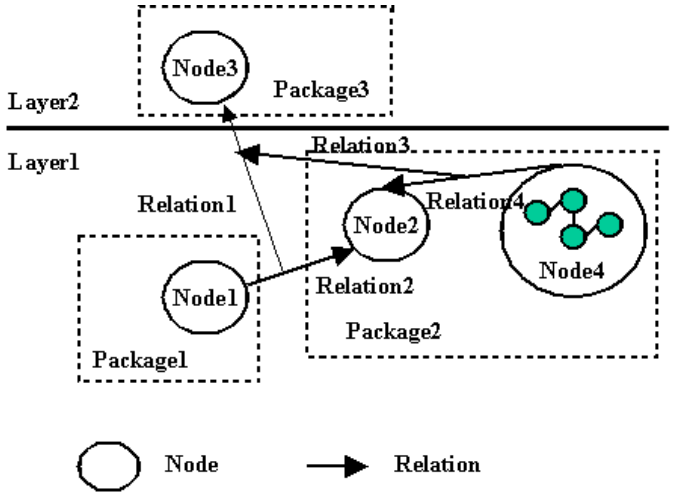

Abbildung 13: Beispiel für semantisches Netz

Layer und Package werden in dieser Arbeit nicht implementiert, weil die beiden nicht in der vorhandenen objektorientierten Implementierung implementiert werden (siehe 3.2).

3.2 Die vorhandene objektorientierte Implementierung

In diesem Abschnitt wird ein kurzer Überblick über die objektorientierte Implementierung gegeben.

Wie bereits in Abschnitt 1.2 und 3.1 erwähnt existiert das Semantic Net Interface (SNI), das Schnittstelle zu Anwendungsprogrammen festlegt, damit die Anwendungsprogramme weiter benutzt werden können, selbst wenn sich das Datenbanksystem verändert wird. Das SNI ist hierarchisch und läßt sich durch Abbildung 14 veranschaulichen.

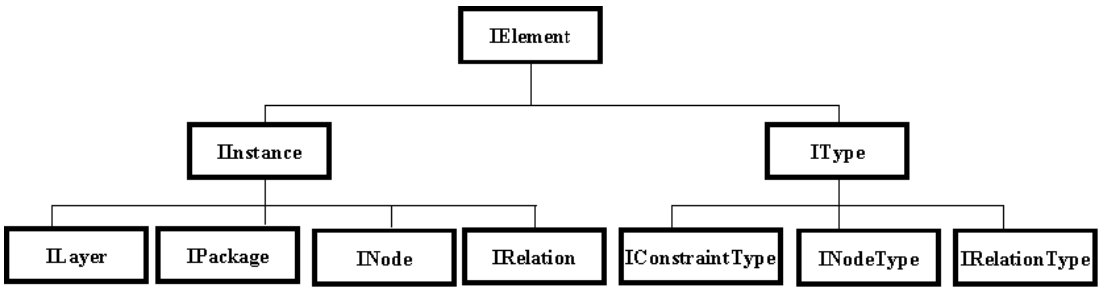

Abbildung 14: Hierarchie von Semantic Net Interface (SNI)

Das Interface `IElement` ist die Wurzel der Hierarchie. Interface `IInstance` erbt von Interface `IElement` und wird von Interface `ILayer`, `IPackage`, `INode`, `IRelation` vererbt. Interface `IType` erbt von Interface `IElement` und wird abgeleitet zu Interface `IConstaintType`, `INodeType` und `IRelationType`. Das Interface `IConstraintType` ist vorhanden aber wird nicht implementiert. Deswegen kommt die Definition von `ConstraintType` in der Anforderungsanalyse (siehe 3.1) nicht vor. Das SNI wird entsprechend objektorientiert, relational und objektrelational implementiert.

Es wird eine Klasse `NetKernel` implementiert. Die Klasse `NetKernel` ist zuständig für die Verbindung mit der Datenbank.

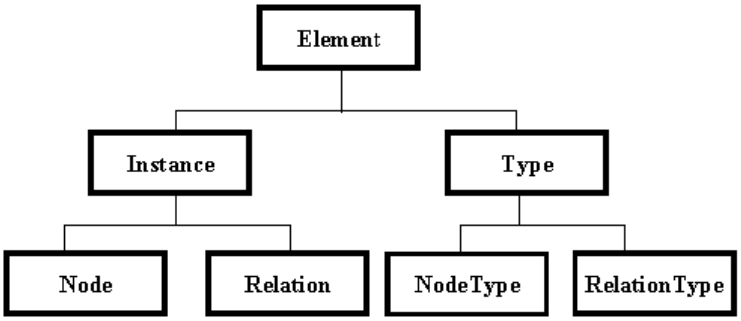

Abbildung 15: Klassenhierarchie der objektorientierte Implementierung

Abbildung 15 zeigt die Klassenhierarchie, die in der objektorientierten Implementierung implementiert wird. Wie bereits in der Anforderungsanalyse (siehe 3.1) erwähnt werden die Elemente des semantischen Netzes Package und Layer nicht in der objektorientierten Implementierung implementiert, deswegen kommen sie in Abbildung 15 nicht vor. Nodes und Relation des semantischen Netzes werden direkt als Java-Klassen implementiert. Typenelemente des semantischen Netzes wie Nodetypen und Relationstypen werden ebenfalls als Java-Klassen implementiert. Alle diese Klassen sind Unterklassen von Klasse Element.

Wie schon in 2.1 erwähnt basiert die objektorientierte Implementierung auf dem DBMS ObjektStore, das die objektorientierte Programmiersprache Java um Datenbankkonzepte erweitert. Um die Instanzen der Klasse `Node`, `Relation`, `RelationType`, `NodeType`, `Instance`, `Type` und `Element` dauerhaft zu speichern, wird eine Klasse `Entity` implementiert, die den Speicherungsmechanismus, der von DBMS Objectstore verlangt wird, realisiert. Dann werden alle Elemente in der objektorientierten Implementierung in der Datenbank durch eine Instanz der Klasse `Entity` repräsentiert.

3.3 Die relationale Modellierung

In diesem Abschnitt beschäftigen wir uns mit der relationalen Modellierung. Dieser Abschnitt wird in Konzeption, logischen und physischen Datenentwurf, und ein Fallbeispiel unterteilt.

3.3.1 Konzeption

Bei (objekt)relationaler Modellierung fängt man mit dem Entity-Relationship-Diagramm (siehe Abschnitt 2.2) an. Man entscheidet, welche in der Anforderungsanalyse (Siehe Abschnitt3.1) identifizierten Objekte, also Node, Relation, Nodetyp und Relationstyp als Entitytyp oder als Beziehungstyp abgebildet werden. Mittlerweile betrachten wir die Vererbungshierarchie der objektorientierte Implementierung (siehe Abschnitt 3.2). Wenn man das Konzept von Generalisierung/Spezialisierung (siehe Abschnitt 2.2.3) benutzt, dann läßt sich die objektorientierte Modellierung leicht in relationale Modellierung umsetzen. Diese Modellierung hat aber Nachteile, weil wie bereits in Abschnitt 2.3.2 erwähnt das Konzept von Generalisierung/Spezialisierung von relationalen DBMS mangelhaft unterstützt wird. Bei der objektrelationalen Modellierung ist dies durch strukturierte Typen einfach umzusetzen (siehe 3.4).

Die Lösung bei relationaler Modellierung liegt darin, daß man Relation und Node nicht getrennt als zwei Entitytypen, sondern als ein Entitytyp betrachtet, weil Relationen zu Relationen wiederum hergestellt werden und Relation zwei beliebige Element eines Netzes verbinden kann (siehe Abschnitt 3.1). Man definiert hier einen Entitytyp `RelationElement`, der Nodes und Relationen repräsentiert. Weil jeder Node zu einem Nodetyp und jede Relation zu einem Relationstyp gehört, definiert man einen Entitytyp `Type`, der Nodetyp und Relationstyp repräsentiert. Dann existiert es eine 1:N Beziehung zwischen `RelationElement` und `Type`. Weil Relationstypen und Nodetyp hierarchisiert werden, existiert es eine 1:N rekursive Beziehung (siehe 2.2.1) auf Entitytyp `Type`. In der Abbildung 16 ist das Entity-Relationship-Diagramm (ER-Diagramm) für relationale Modellierung dargestellt. Auf die genaue Beschreibung der jeweiligen Attribute wird hier verzichtet und in Unterabschnitt 3.3.2 beschrieben.

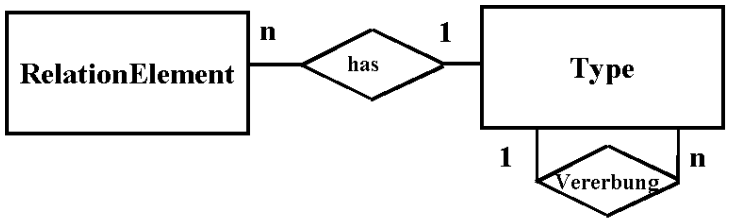

Abbildung 16: ER-Diagram der relationalen Modellierung

3.3.2 Logischer und Physischer Datenentwurf

Der Entitytyp und der Beziehungenstyp, die in Abbildung 16 dargestellt werden, werden nach in Abschnitt 2.3.2 beschriebenen Methoden auf Relationen abgebildet. Der Entitytyp `RelationElement` wird in eine Tabelle `RelationElement` abgebildet. Der Entitytyp `Type` wird in eine Tabelle `rType` abgebildet. Wegen der 1:N Beziehung zwischen Entitytyp `RelationElement` und `Type` wird der Primärschlüssel von Tabelle `rType` als Fremdschlüssel `Type` der Tabelle `RelationElement` hinzugefügt. Wegen der rekursive 1:N Beziehung auf Entitytyp `Type` wird der Primärschlüssel von Tabelle `rType` als Fremdschlüssel `Father` der Tabelle `rType` hinzugefügt. Die Tabellen werden im folgenden gezeigt. Wie bereits erwähnt sind die Primärschlüssel unterstrichen. Die Fremdschlüssel sind kursiv gedruckt. Attribut `OID` bedeutet Objektidentifikation und sein Wert wird durch die in DB2 integrierte Funktion `GENERATE_UNIQUE()`erzeugt.

`RelationElement (`<u>`OID`</u>`, Name, `*`Start`*`, `*`End`*`, `*`Type`*`, Remark)`
`rType(`<u>`OID`</u>`, Name, `*`Father`*`, Type, Remark )`

Tabelle `RelationElement` speichert die Instanz der Relationen und Nodes und man kann letztere durch das Attribut `Remark` unterscheiden. Der Primärschlüssel ist `OID`. Das Attribut `Name` speichert Name von Nodes und Relationen. Das Attribut `Type` speichert den Typ eines Nodes oder einer Relation. Wie bereits erwähnt ist es auch ein Fremdschlüssel, der auf Primärschlüssel von Tabelle `rType` verweist. Weil wenn ein Relationstyp oder Nodetyp gelöscht wird, müssen die entsprechenden Relationen und Nodes gelöscht werden, wird `Type` als `ON DELETE CASCADE` definiert. `ON DELETE CASCADE` bedeutet wenn eine Zeile in der Vatertabelle gelöscht wird, dann werden auf diese Zeile verweisende Zeilen der Kindtabelle gelöscht. In dem Fall von Relationen speichert das Attribut `Start` die OID eines Startelements (Node oder Relation) einer Relation und das Attribut `End` die OID eines Endelements (Node oder Relation) einer Relation, weil Relation eine Richtung hat. In dem Fall von Nodes speichert sowohl `Start` als auch `End`

die OID von Nodes. Die beiden Attribute sind auch Fremdschlüssel, die auf den Primärschlüssel der Tabelle `RelationElement` verweisen. Weil wenn ein Node oder eine Relation gelöscht wird, müssen entsprechend darauf basierende Relationen gelöscht werden, d.h. werden `Start` und `End` als `ON DELETE CASCADE` definiert.

Tabelle `rType` speichert die Instanzen von Typen (Nodetypen oder Relationstypen). Ihr Primärschlüssel ist `OID`. Das Attribut `Name` speichert Name eines Typs. Das Attribut `Father` speichert die OID des Vatertyps, weil Typen hierarchisiert sind. Wie bereits erwähnt ist `Father` auch Fremdschlüssel, der auf den Primärschlüssel von Tabelle `rType` verweist. Weil wenn ein Vatertyp gelöscht wird, müssen die entsprechenden Kindtypen gelöscht werden. Daher wird `Father` als `ON DELETE CASCADE` definiert. Das Attribut `Remark` zeigt, ob es sich beim Einfügen eines Typs um einen Vatertyp oder einen Kindtyp handelt (siehe Anhang 1.4 für genaue Beschreibung). Das Attribut `Type` zeigt, ob es sich um einen Nodetyp oder einen Relationstyp handelt. Siehe Abschnitt 3.3.3 für Fallbeispiel der beiden Tabelle `RelationElement` und `rType`.

Das logische Schema der relationalen Modellierung läßt sich durch Abbildung 17 veranschaulichen. Wie bereits erwähnt sind die Primärschlüssel unterstrichen. Die Fremdschlüssel sind kursiv. Die Pfeile stellen Fremdschlüssel-Beziehungen dar.

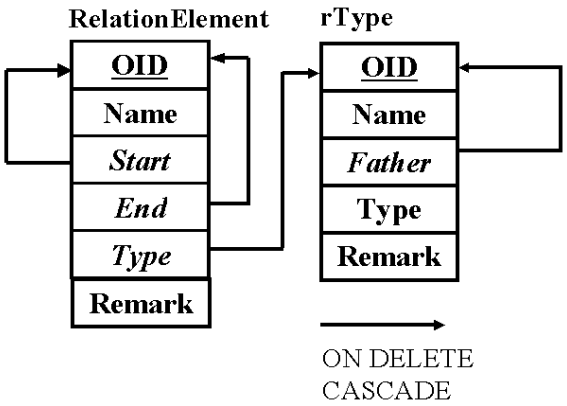

Abbildung 17: Das logische Schema der relationalen Modellierung

Die im folgenden beschriebenen Integritätsbedingungen müssen außerdem noch bewahrt werden.

1. Von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden. Dies kann realisiert werden, indem man Trigger

definiert. Die Trigger überwachen, ob es schon eine Relation von einem Relationstyp zwischen Relationselemente existiert. Wenn dies der Fall ist, dann werden die Trigger ausgelöst (siehe Anhang 1.2).

2. Der Name eines Typs muß eindeutig sein. Dies kann realisiert werden, indem Trigger definiert werden (siehe Anhang 1.3).

Auf die genauen physischen Tabellen- und Triggerdefinitionen mit SQL wird hier verzichtet und sie werden in Anhang 1.1 gezeigt. Statt dessen werden die Definitionen der SQL-Datentypen der Tabellen im folgenden vorgestellt. Zugunsten der Leser wird die Bedeutung der Attribute noch einmal beschrieben. Siehe Tabelle 1 und Tabelle 2.

Tabelle 1: Physisches Schema(relationale Modellierung): rType

Attribute	SQL-Datentypen	Beschreibung
OID	CHAR(13) FOR BIT DATA	Identifikator des Typs (Nodetyp oder Relationstyp)
Name	VARCHAR(30)	Name des NodeTyps oder des Relationstyps
Father	CHAR(13) FOR BIT DATA	OID des Vatertyps
Remark	VARCHAR(30)	Remark zeigt, ob die Zeile Vatertyp oder Kindtyp ist, wird bei Einfügen von Vatertyp gebraucht.
Type	VARCHAR(30)	Type zeigt, ob die Zeile Nodetyp oder Relationstyp ist.

Tabelle 2: Physisches Schema(relationale Modellierung): RelationElement

Attribute	SQL-Datentypen	Beschreibung
OID	CHAR(13) FOR BIT DATA	Identifikator des Nodes oder der Relation
Name	VARCHAR(30)	Name des Nodes oder der Relation
Start	CHAR(13) FOR BIT DATA	Identifikator des Startelements (Node oder Relation) einer Relation
End	CHAR(13) FOR BIT DATA	Identifikator des Endelements (Node oder Relation) einer Relation
Type	CHAR(13) FOR BIT DATA	Identifikator des Typs (Nodetyp oder Relationstyp) dieser Zeile
Remark	VARCHAR (30)	Unterscheidet zwischen Relation oder Node

3.3.3 Fallbeispiel

Schließlich wird hier ein Anwendungsbeispiel gegeben. In Abbildung 18 wird ein semantisches Netz gezeigt und in Tabelle 3, 4 wie man das Netz in DB2 speichern kann**.**

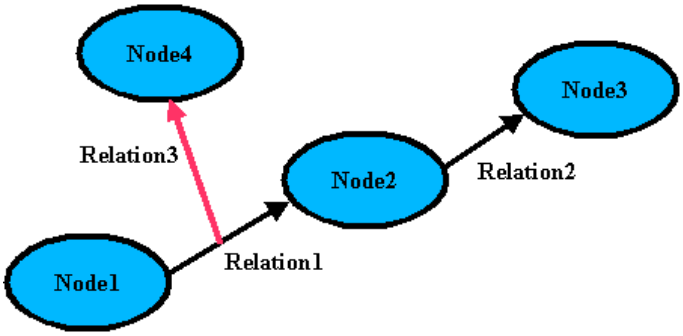

Abbildung 18: Das semantische Netz für Anwendungsbeispiel

Tabelle 3: Fallbeispiel(relationale Modellierung): RelationElement

OID	Name	Start	End	Type	Remark
X'200201171511 06912006000100'	Node1	X'2002011715110 6912006000100'	X'2002011715110 6912006000100'	X'20020117151106 912006000000'	Node
X'200201171511 06912006000200'	Node2	X'2002011715110 6912006000200'	X'2002011715110 6912006000200'	X'20020117151106 912006000000'	Node
X'200201171511 06912006000300'	Node3	X'2002011715110 6912006000300	X'2002011715110 6912006000300	X'20020117151106 912006000000'	Node
X'200201171511 06912006000400'	Node4	X'2002011715110 6912006000400'	X'2002011715110 6912006000400'	X'20020117151106 912006000000'	Node
X'200201171511 06912006000600'	Relation1	X'2002011715110 6912006000100'	X'2002011715110 6912006000200'	X'20020117151106 912006000500'	NN
X'200201171511 06912006000700'	Relation2	X'2002011715110 6912006000200'	X'2002011715110 6912006000300'	X'20020117151106 912006000500'	NN
X'200201171511 06912006000800'	Relation3	X'2002011715110 6912006000600'	X'2002011715110 6912006000400'	X'20020117151106 912006000500'	RN

Bemerkung: Unter Spalte `Remark` gibt es verschiedene Werte, deren Bedeutung wie folgende erläutert werden.

- Node: Diese Zeile repräsentiert eine Instanz von Node.
- NN: Relation geht von einem Node zu einem Node.
- RN: Relation geht von einer Relation zu einem Node.
- NR: Relation geht von einem Node zu einer Relation.
- RR: Relation geht von einer Relation zu einer Relation.

Tabelle 4: Fallbeispiel(relationale Modellierung): rType

OID	Name	Father	Remark	Type
X'20020117151106912006000000'	Flach	X'20020117151106912006000000'	Father	NodeType
X'20020117151106912006000500'	Derivation	X'20020117151106912006000500'	Father	RelationType

Bemerkung:

- Nodetyp `Flach` bedeutet, daß ein Node keine Netze oder keine Datentypen enthält.
- Das Attribut `Remark` hat zwei Werte, nämlich„Father" und „Kind". „Father" bedeutet ein Vatertyp und „Kind" bedeutet ein Kindtyp
- Das Attribut zeigt, ob es sich um einen NodeType oder einen RelationType handelt.

3.4 Die objektrelationale Modellierung

In diesem Abschnitt beschäftigen wir uns mit der objektrelationalen Modellierung. Wie die relationale Modellierung (Abschnitt 3.3) wird dieser Abschnitt in Konzeption, logischen und physischen Datenentwurf, und ein Fallbeispiel unterteilt.

3.4.1 Konzeption

Wie in Abschnitt 3.3 erwähnt kann man das Konzept von Generalisierung/Spezialisierung (siehe 2.2.3) leicht durch strukturierte Typen umsetzen. Basierend auf Definition des semantischen Netzes in der Anforderungsanalyse (Abschnitt 3.1) und die Klassenhierarchie in der objektorientierten Implementierung (Abschnitt 3.2) kann man folgende Entitytypen und Beziehungenstypen definieren. Siehe Abbildung 19.

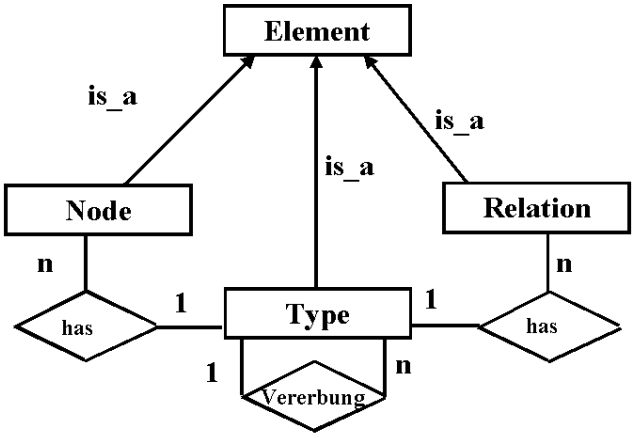

Abbildung 19: ER-Diagramm der objektrelationalen Modellierung

In Abbildung 19 ist das Entitytyp `Element` die Oberklasse der Entitytypen `Node`, `Relation` und `Type`, die jeweils Node, Relation und Nodetyp bzw. Relationstyp repräsentieren. Wie bereits in 3.3.1 erwähnt gehört ein Node zu einem Nodetyp und eine Relation zu einem Relationstyp. Deswegen existiert es zwischen `Node` und `Type` eine 1:N Beziehung. Zwischen `Relation` und `Type` existiert es auch eine 1:N Beziehung. Wie bereits in 3.3.1 erwähnt kann ein Kindtyp von einem Vatertyp erben. Deswegen existiert eine 1:N rekursive Beziehung (siehe 2.2.1) auf Entitytyp `Type`. Bei der Umsetzung dieses ER-Diag-

ramms definiert man zuerst die strukturierte Typenhierarchie. Dann ordnet man den Tabellen die Typen zu. Siehe Unterabschnitt 3.4.2.

3.4.2 Logischer und Physischer Datenentwurf

In Abbildung 20 repräsentiert der Wurzeltyp `Element_t` Entitytyp `Element`. Die Typen, `Node_t`, `Relation_t` und `Type_t`, sind die Untertypen von `Element_t` und repräsentieren entsprechend Entitytypen `Node`, `Relation` und `Type`. Wie in Abschnitt 2.6.3 schon erwähnt erzeugt DB2 automatisch für jeden Typ ein Attribut, OID (Objektidentifikation). Dieses Attribut ist ein Referenztyp (siehe 2.6.3).

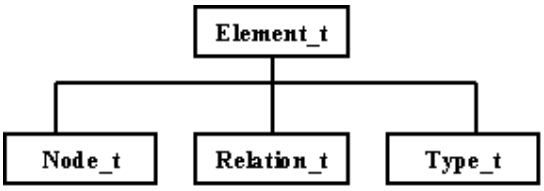

Abbildung 20: Typenhierarchie von Element

`Element_t` ist der Basistyp der Hierarchie und hat ein Attribut `Name`. `Name` wird von allen Untertypen geerbt und speichert Name eines Elements.

Ähnlich wie die Attribute der Tabelle `rType` in der relationalen Modellierung (siehe Abschnitt 3.3) hat `Type_t` eigene Attribute `Father` und `Remark` und `Type`. Das Attribut `Father` ist ein Referenztyp (siehe Abschnitt 2.6.3) und verweist auf Typ `Type_t` wegen der rekursiven 1:N Beziehung auf Entitytyp `Type`. Das Attribut `Remark` zeigt wie in der relationalen Modellierung, ob es sich beim Einfügen eines Typs um einen Vatertyp oder einen Kindtyp handelt (siehe Anhang 2.5). Das Attribut `Type` zeigt, ob es sich um einen Nodetyp oder einen Relationstyp handelt.

`Node_t` hat ein eigenes Attribut `Type`. `Type` ist Referenztyp und verweist auf Typ `Type_t` wegen der 1:N Beziehung zwischen Entitytyp `Node` und `Type`.

Ähnlich wie die Attribute der Tabellen `RelationElement` (siehe Abschnitt 3.3) hat `Relation_t` eigene Attribute `Start`, `End`, `Type` und `Remark`. `Start` und `End` sind ein

Referenztyp und verweisen auf Typ `Element_t`, weil wie bereits in Abschnitt 3.1 erwähnt Relationen zwei beliebige Elemente eines Netzes (Nodes oder Relationen) verbinden können. Attribut `Type` verweist auf Typ `Type_t` wegen der 1:N Beziehung zwischen Entitytypen `Type` und `Relation`. Attribut `Remark` zeigt, ob eine Relation zwei Nodes oder einen Node und eine Relation oder eine Relation und einen Node oder zwei Relationen verbindet.

Die SQL-Anweisungen, die zur Definition der Typenhierarchie benötigt werden, werden hier nicht vorgestellt und sind in Anhang 2.1 nachzuschlagen. Statt dessen werden Definitionen der SQL-Datentypen in Tabellen 5, 6, 7, 8 vorgestellt.

Tabelle 5: Physisches Schema (objektrelationale Modellierung): Element_t

Attribute	SQL-Datentypen
OID	VARCHAR(16) FOR BIT DATA
Name	VARCHAR(30)

Tabelle 6: Physisches Schema (objektrelationale Modellierung): Node_t

Attribute	SQL-Datentypen
OID	VARCHAR(16) FOR BIT DATA
Name	VARCHAR(30)
Type	REF(Type_t)

Tabelle 7: Physisches Schema (objektrelationale Modellierung): Type_t

Attribute	SQL-Datentypen
OID	VARCHAR(16) FOR BIT DATA
Name	VARCHAR(30)
Father	REF(Type_t)
Remark	VARCHAR(30)
Type	VARCHAR(30)

Tabelle 8: Physisches Schema (objektrelationale Modellierung): Relation_t

Attribute	SQL-Datentypen
OID	VARCHAR(16) FOR BIT DATA
Name	VARCHAR(30)
Start	REF(Element_t)
End	REF(Element_t)
Type	REF(Type_t)
Remark	VARCHAR(30)

Aufgrund der Typenhierarchie (siehe Abbildung 20) baut man eine typisierte Tabellenhierarchie (Abbildung 21) auf. Wie bereits erwähnt müssen die typisierten Typen den Tabellen zugeordnet werden, damit die Tabellen die Instanzen der entsprechenden Typen speichern können und automatisch die Attribute der zugeordneten Typen bekommen. Die Instanzen von Node, Relation, Nodetyp und Relationstyp werden entsprechend in den Tabellen `Node`, `Relation` und `Type` gespeichert und automatisch in die Tabelle `Element` eingefügt. In Abschnitt 3.4.3 gibt es Beispiele dazu. Ein Unterschied zu der relationalen Modellierung ist, daß die Instanzen von Nodes und Relationen nicht in einer gemeinsamen Tabelle gespeichert werden.

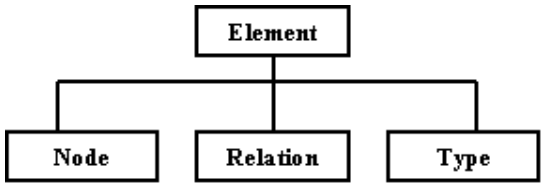

Abbildung 21: Tabellenhierarchie von Element

Die Tabellen werden im folgenden gezeigt. Wieder sind die Primärschlüssel unterstrichen. Die Fremdschlüssel sind kursiv gedruckt. Die Werte von OID werden durch Funktion `GENERATE_UNIQUE()`erzeugt.

`Element (OID, Name)`
`Node(OID, Name, *Type*)`
`Relation (OID, Name, *Start*, *End*, *Type*, Remark)`
`Type(OID, Name, *Father*, Type, Remark )`

Auf die Erläuterung der Attribute der Tabellen werden hier verzichtet, weil die Bedeutung der Attribute der Tabellen wie die Bedeutung der Attribute der Strukturierten Typen sind, die schon vorher erläutert wurden. Wie bereits erwähnt ist das Attribut `Type` von Tabelle `Node` ein Fremdschlüssel, der auf Primärschlüssel von Tabelle `Type` verweist. Weil wenn ein Nodetyp gelöscht wird, müssen entsprechende Nodes gelöscht werden, d.h. es wird als `ON DELETE CASCADE` (siehe Abschnitt 3.3.2) definiert. Die Fremdschlüssel `Start`, `End` und `Type` der Tabelle `Relation` kann man wie die Tabelle `RelationElement` der relationalen Modellierung (Abschnitt 3.3.2) als `ON DELETE CASCADE` definieren.

Der Fremdschlüssel `Father` der Tabelle `Type` kann man wie die Tabelle `type` der relationalen Modellierung (Abschnitt 3.3.2) als `ON DELETE CASCADE` definieren.

Ähnlich wie in der relationalen Modellierung (siehe 3.3.2) läßt sich das logische Schema der objektrelaitonalen Modellierung durch Abbildung 22 veranschaulichen. Wie bereits erwähnt sind die Primärschlüssel unterstrichen. Die Fremdschlüssel sind kursiv gedruckt. Die Pfeile stellen Fremdschlüssel-Beziehungen dar.

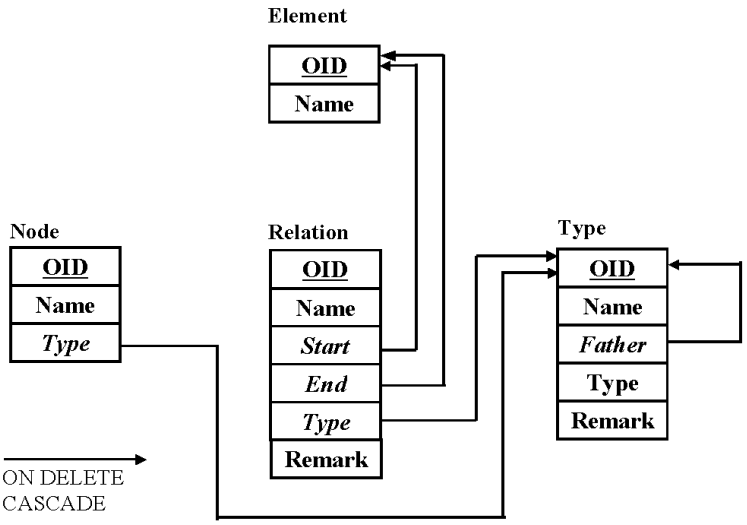

Abbildung 22: Das logische Schema der objektrelationalen Modellierung

Auf die Erläuterung der SQL-Anweisungen, die zu Definition der Tabellenhierarchie benötigt werden, wird hier verzichtet. Sie sind in Anhang 2.2 nachzuschlagen.

Wie in der relationalen Modellierung (Siehe 3.3.2) müssen die im folgenden beschriebenen Integritätsbedingungen außerdem bewahrt werden:

1. Von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden. Dies kann realisiert werden, indem man Trigger definiert(siehe Anhang 2.3).
2. Der Name eines Typs muß eindeutig sein. Dies kann durch Trigger (siehe Anhang 2.4) garantiert werden.

3.4.3 Fallbeispiel

Es wird ein Beispiel gegeben, wie man das semantische Netz in den Tabellen der objektrelationalen Modellierung speichern kann. Das semantisches Netz in Abbildung 18 (siehe Abschnitt 3.3.3) wird in den folgenden Tabellen 9, 10 ,11, 12 gespeichert.

Tabelle 9 : Fallbeispiel(objektrelationale Modellierung): Element

OID	Name
X'20020117151106912006000000'	Flach
X'20020117151106912006000500'	Derivation
X'20020117151106912006000100'	Node1
X'20020117151106912006000200'	Node2
X'20020117151106912006000300'	Node3
X'20020117151106912006000400'	Node4
X'20020117151106912006000600'	Relation1
X'20020117151106912006000700'	Relation2
X'20020117151106912006000800'	Relation3

Bemerkung:

- Die Instanzen der unteren Tabellen, nämlich `Type`, `Node` und `Relation` werden automatisch in Tabelle `Element` hinzugefügt.

Tabelle 10: Fallbeispiel(objektrelationale Modellierung): Type

OID	Name	Father	Remark	Type
X'2002011715110 6912006000000'	Flach	X'2002011715110 6912006000000'	Father	NodeType
X'2002011715110 6912006000500'	Derivation	X'2002011715110 6912006000500'	Father	RelationType

Wie in der relationalen Modellierung (Siehe Abschnitt 3.3) gilt auch hier die folgende Bemerkungen:

- Nodetyp `Flach` bedeutet, daß ein Node keine Netze oder keine Datentypen enthält.
- Das Attribut `Remark` hat zwei Werte, nämlich„Father" und „Kind". „Father" bedeutet ein Vatertyp und „Kind" bedeutet ein Kindtyp
- Das Attribut zeigt, ob es sich um einen NodeType oder einen RelationType handelt.

Tabelle 11: Fallbeispiel(objektrelationale Modellierung): Node

OID	Name	Type
X'2002011715110 6912006000100'	Node1	X'200201171511 06912006000000'
X'2002011715110 6912006000200'	Node2	X'200201171511 06912006000000'
X'2002011715110 6912006000300'	Node3	X'200201171511 06912006000000'
X'2002011715110 6912006000400'	Node4	X'200201171511 06912006000000'

Tabelle 12: Fallbeispiel(objektrelationale Modellierung): Relation

OID	Name	Start	End	Type	Remark
X'2002011715110 6912006000600'	Relation1	X'20020117151106 912006000100'	X'200201171511069 12006000200'	X'2002011715110 6912006000500'	NN
X'2002011715110 6912006000700'	Relation2	X'20020117151106 912006000200'	X'200201171511069 12006000300'	X'2002011715110 6912006000500'	NN
X'2002011715110 6912006000800'	Relation3	X'20020117151106 912006000600'	X'200201171511069 12006000400'	X'2002011715110 6912006000500'	RN

Wie in der relationalen Modellierung (siehe Abschnitt 3.4) gilt auch hier die folgende Bemerkungen:

- NN: Relation geht von einem Node zu einem Node.
- RN: Relation geht von einer Relation zu einem Node
- NR: Relation geht von einem Node zu einer Relation
- RR: Relation geht von einer Relation zu einer Relation

3.5 Implementierung der JDBC-Schnittstelle

In diesem Abschnitt wird die Grundidee der Implementierung der JDBC-Schnittstelle für relationale und objektrelationale Modellierung in der Arbeit erläutert. Die Erläuterung gilt für beide Implementierungen.

Die Objekte wie Nodes, Relationen und Typen des semantischen Netzes, die im Anwendungsprogramm erzeugt werden, werden nach Beenden des Programms gelöscht. Deswegen müssen die Objekte vor dem Beenden in der Datenbank gespeichert werden. Man möchte vom Programm aus die Daten in der Datenbank lesen. Dies erfolgt durch die JDBC-Schnittstelle, die die Abbildung zwischen Anwendungsprogramm und Datenbank leistet.

Es wird mit Java implementiert. Es wird eine Klasse `NetKernel` implementiert. Die Klasse `NetKernel` ist zuständig für die Verbindung mit der Datenbank. Die Methode `openConnection()` stellt eine Verbindung mit der Datenbank bereit. Die Methode `closeConnection()` beendet die Verbindung mit der Datenbank. Es werden die Basisklassen `Element`, `Instance`, `Node`, `Relation`, `Type`, `RelationType` und `NodeType` des SNI (siehe 3.2) implementiert. Die Vererbungshierarchie der Klassen in der objektorientierten Implementierung (siehe 3.2) wird hier übernommen. Die Klasse `Element` ist eine abstrakte Klasse und hat drei abstrakte Methoden, nämlich `Insert()`, `Delete()` und `Update()`. Diese drei Methoden werden entsprechend in den Unterklassen implementiert und sind für die grundlegende Datenmanipulation verantwortlich. Die Abbildung 23 veranschaulicht die Vererbungshierarchie.

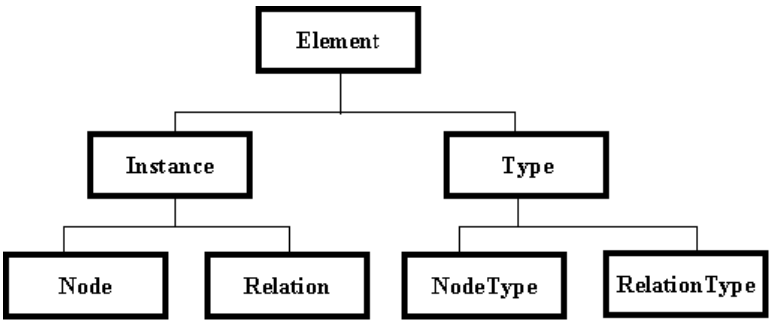

Abbildung 23: Klassenhierarchie der JDBC-Schnittstelle

3.6 Performancetest

Wie bereits in der Aufgabenstellung (Siehe Abschnitt 1.2) erwähnt werden in dieser Studienarbeit jeweils die Performance der objektrelationalen und relationalen Implementierung gemessen. Der Performancevergleich mit der objektorientierten Implementierung konnte wegen der zeitlichen Verschiebung der Arbeit eines Projektpartners nicht durchgeführt werden. In diesem Abschnitt wird der Vergleich zwischen relationaler und objektrelationaler Implementierung vorgestellt.

Bei Performancetest handelt es sich um die Messung der Leistung der Datenbank. Es gibt bestimmte Faktoren, die Performance beeinflussen. Folgende sind einige Beispiele dafür:

- Die Qualität des JDBC-Schnittstellenprogramms
- Die Qualität der entworfenen Datenbanken
- Die Leistung des zuständigen Datenbank-Management-Systems

Ein Index wird jeweils auf Attribute `(Start, End, Type)` der Tabelle `Relation-Element` (siehe 3.3) und der Tabelle `Relation` (siehe 3.4) gesetzt. In der Arbeit wird Performance durch Kopieren eines Teilnetzes, und Navigation im Netz getestet. In Abschnitt 3.6.1 wird Kopieren eines Teilnetzes vorgestellt. In Abschnitt 3.6.2 wird Navigation im Netz vorgestellt. In Abschnitt 3.6.3 wird Interpretation der beiden Tests vorgestellt.

3.6.1 Kopieren eines Teilnetzes

Die Testdaten beruhen auf einem semantischen Netz eines der Industriepartner im föderal-Projekt und werden in einer XML-Datei gespeichert. Das Netz besteht aus 832 Nodes, 1618 Relationen und 24 Relationstypen.

Zuerst werden die Daten durch einen XML-Parser in die Datenbank geschrieben. In diesem Fall wird ein DOM-Parser benutzt. Der Test hier wird jeweils bei jeder Implementierung zehnmal durchgeführt. Nach jedem Test werden die Tabellen neu angelegt. Die Messergebnisse sind in Tabelle 13 veranschaulicht.

Tabelle 13: Testergebnisse: Lesen der XML-Datei

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Durch-schnitt
Relational (Sekunden)	107,8	104,3	104,2	102,2	111,9	113,3	115,2	111,9	111,2	105,7	108,8
Objektrela-tional (Sekunden)	116,5	120,1	119,2	115,4	120,7	114,9	114,3	117,8	113,9	115,9	116,9

Die Daten in dieser Tabelle werden in einer Grafik übertragen. Die Ergebnisse lassen sich dadurch einfach leicht vergleichen.

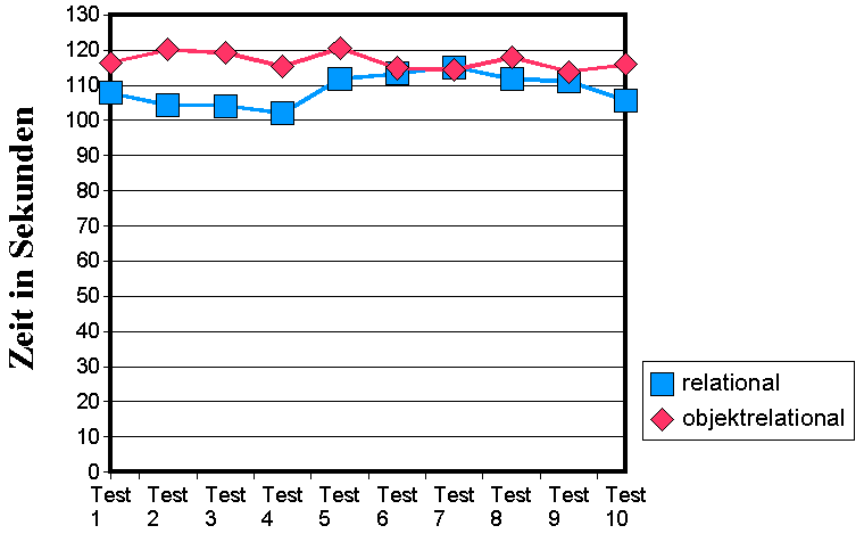

Abbildung 24: Testergebnisse: Lesen der XML-Datei

Abbildung 24 zeigt, daß die relationale Implementierung beim Lesen der XML-Datei in die Datenbank fast gleich schnell wie die objektrelationale Implementierung ist.

Dann wurde die Performance von Kopieren eines Teilnetzes getestet. Ein Teilnetz des gerade in die Datenbank geschriebenen semantischen Netzes wird aus der Datenbank gelesen und wieder in die Datenbank kopiert. Das hier getestete Teilnetz besteht aus einem Wurzelknoten und allen seinen Unterknoten, die rekursiv immer durch den Relationstyp „derivation” verbunden sind. Insgesamt besteht das Teilnetz aus 716 Knoten und

715 Relationen. Es wurde ein Testprogramm mit der Methode der Tiefensuche für Lesen und Kopieren des Teilnetzes geschrieben. Das Testprogramm wird jeweils für jede Implementierung zehnmal ausgeführt. Die Ergebnisse lassen sich jeweils durch Tabellen und Grafik veranschaulichen. Beim Lesen des Teilnetzes ist die relationale Implementierung durchschnittlich ein bißchen schneller als die objektrelationale Implementierung. Siehe Tabelle 14 und Abbildung 25. Beim Kopieren des Teilnetzes in die Datenbank ist die relationale Implementierung durchschnittlich ein bißchen schneller als die objektrelationale Implementierung. Siehe Tabelle 15 und Abbildung 26.

Tabelle 14: Testergebnisse: Lesen des Teilnetzes

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Durchschnitt
Relational (Sekunden)	8,9	8,9	8,5	8,1	7,4	8,4	9,1	7,6	8,1	8,6	8,4
Objektrelational (Sekunden)	9,5	9,5	9,5	10,5	10,7	10,9	10,9	9,2	9,2	8,8	9,9

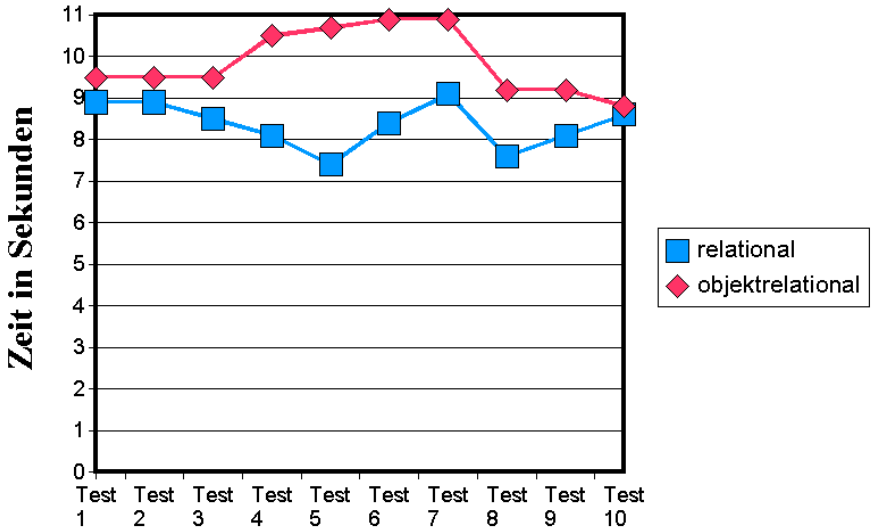

Abbildung 25: Testergebnisse: Lesen des Teilnetzes

Tabelle 15: Testergebnisse: Kopieren des Teilnetzes

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Durch-schnitt
Relational (Sekunden)	88,2	90,8	79,2	83,2	80,2	82,5	86,5	71,1	90	88,2	83,4
Objektrela-tional (Sekunden)	89,7	87,4	89,4	82,7	89,8	90	86,5	88,7	87,5	83,4	87,5

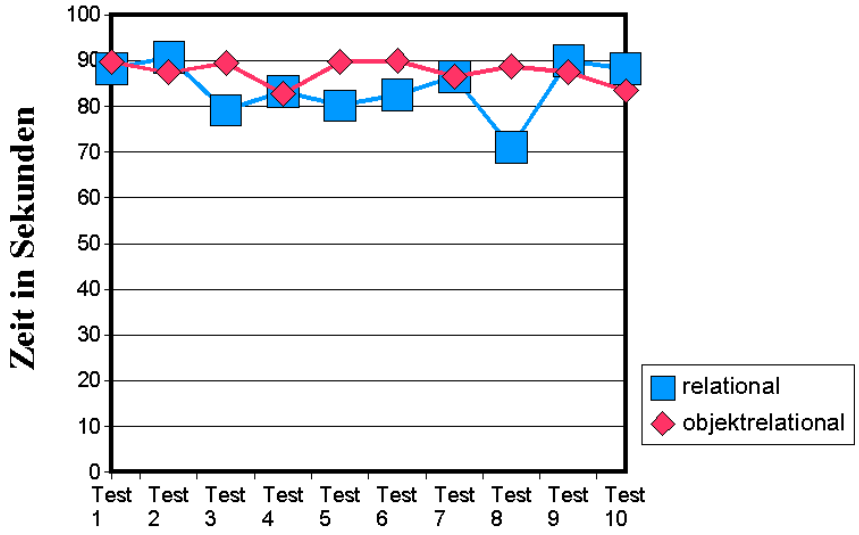

Abbildung 26: Testergebnisse: Kopieren des Teilnetzes

Anschließend wurde ein Test für ein Teilnetz mit variabler Größe durchgeführt. Die Testdaten bleiben unverändert. Das Testprogramm wurde dreimal ohne neues Anlegen der Tabellen gestartet. Zuerst wurde ein Teilnetz von 1430 Nodes und Relationen, die durch einzigen Relationstyp „derivation" mit dem Wurzelknoten verbunden sind, gelesen und wieder gekopiert. Dann von 2860 Nodes und Relationen. Dann von 5720 Nodes und Relationen. Die Ergebnisse lassen sich durch Abbildung 27 und 28 veranschaulichen. Die Einheit der X-Achse ist die Anzahl der Nodes und Relationen und der Y-Achse ist Sekunde.

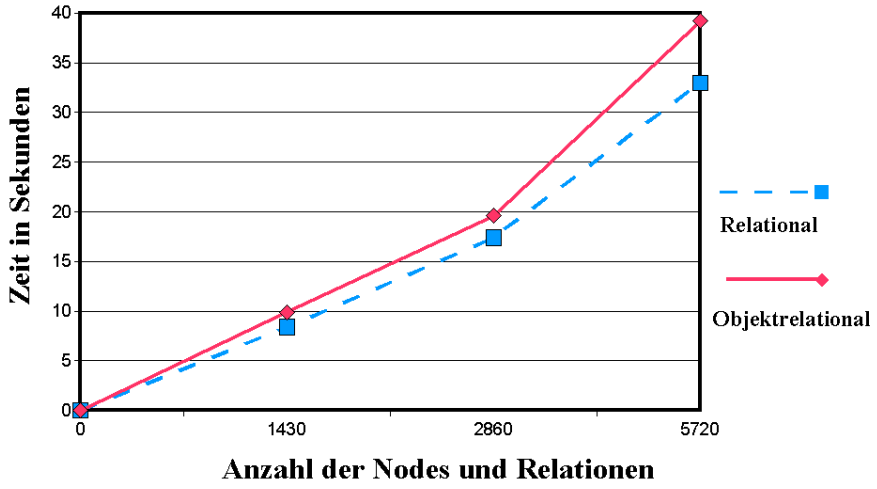

Abbildung 27: Lesen des Teilnetzes (variable Größe)

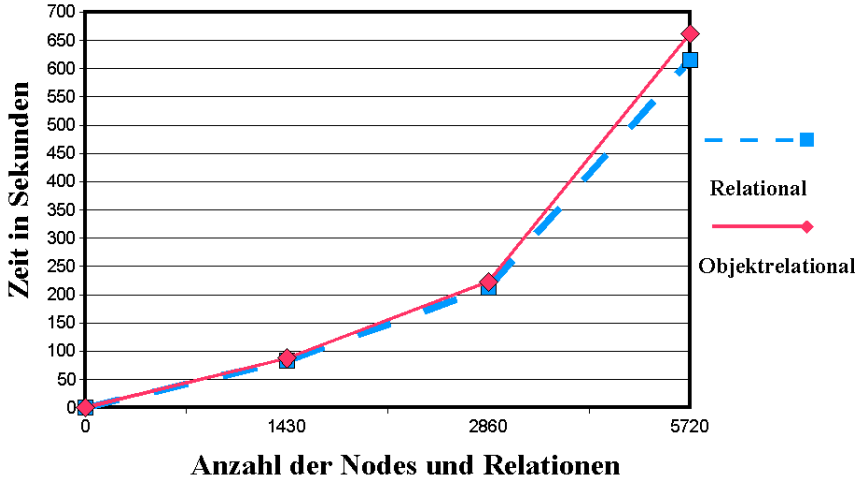

Abbildung 28: Kopien des Teilnetzes(variable Größe)

3.6.2 Navigationstest

In diesem Test wird die Navigation im Netz gemessen, d.h. wie lang es dauert, einen Pfad entlangzugehen. Das hier getestete Netz besteht aus einer Kette von Nodes, die alle über Relation des selben Typs verbunden sind. Getestet werden hier die Pfadlänge von 500, 1000, 2000, 3000, 4000 und 5000. Die Ergebnisse lassen sich durch Tabelle und Grafik veranschaulichen. Siehe Tabelle 16 und Abbildung 29.

Tabelle 16: Testergebnisse: Navigation eines Teilnetzes

	0	500	1000	2000	3000	4000	5000
Relational (Sekunden)	0	7,4	16,9	32,6	51,2	67,2	84,5
Objektrelational (Sekunden)	0	8,1	17,4	33,6	54,9	72,1	97,6

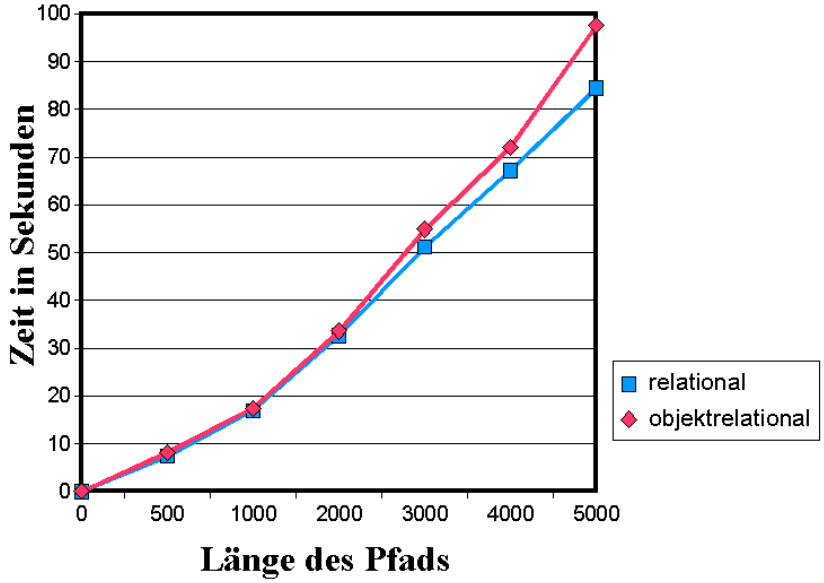

Abbildung 29: Testergebnisse: Navigation eines Teilnetzes

3.6.3 Interpretation der Ergebnisse

Aus den obigen Tests wird es klar, daß die relationale Implementierung bei Performance etwas besser als die objektrelationale Implementierung ist, wenn sich die Größe des Netzes vergrößert oder sich die Pfadlänge verlängert. Die Gründe dafür sind:

1. Bei objektrelationaler Implementierung müssen die Tupel der Untertypen beim Einfügen in die Tabelle `Element` eingefügt werden. Bei relationaler Implementierung muß dies nicht geschehen.

2. Die Methode `selectRelatedElements()`wird jeweils relational und objektrelational implementiert und im Test benutzt. Diese Methode nimmt ein Element (Node oder Relation) und einen Relationstyp als Parameter an und gibt alle Nodes und Relationen, die durch diesen Relationstyp mit diesem Element verbunden sind, zurück. Die Nodes und die Relationen sind bei der relationalen Implementierung in einer gemeinsamen Tabelle gespeichert, während sie bei der objektrelationalen Implementierung getrennt in zwei Tabellen gespeichert sind. Bei der relationalen Implementierung reicht ein Zugriff aus, um alle in Beziehung stehenden Nodes und Relationen zu finden. Bei der objektrelationalen Implementierung wurde es so entworfen, daß die OID der in Beziehung stehenden Elemente zuerst gefunden werden. Dann wird es entsprechend auf die Tabelle `Node` oder `Relation` (siehe 3.4) zugriffen, um die Werte aller Attribute der Relationen oder Nodes zu finden. Deswegen bei der objektrelationalen Implementierung braucht die Methode zwei Zugriffe durch die JDBC-Schnittstelle auf die Datenbank während sie bei der relationalen Implementierung einen Zugriff braucht.

3. Nach [SM99] besitzt das relationale DBMS typspezifische B-Bäume während das objektrelationale DBMS generische B-Bäume besitzt. Weil das objektrelationale DBMS flexibler ist, ist der Code von B-Baum grundsätzlich langsamer. Ein objektrelationales DBMS verbraucht mehr CPU-Zeit als ein relationales DBMS, wenn eine relationale Anfrage durchgeführt wird. Deswegen hat objektrelationales DBMS einen bescheidenen Leistungsverlust.

4. Zusammenfassung

Ausgehend von einer objektorientierten Implementierung des semantischen Netzes unter Objectstore wird eine relationale und eine objektrelationale Implementierung unter DB2 umgesetzt und anschließend wird die Performance der beiden Implementierungen getestet.

In dieser Ausarbeitung werden im Kapitel 2 die Grundlagen der objektorientierten Datenbank, des Entity-Relationship-Modells, der relationalen und objektrelationalen Datenbanken, und die Klassifikation von DBMS erklärt. Dann werden im Kapitel 3 Anforderungsanalyse, vorhandene objektorientierte, relationale und objektrelationale Implementierung und Performancetest erläutert. Die Performancetests zeigen, daß bei Performance die relationale Implementierung etwas besser als die objektrelationale Implementierung ist, wenn sich die Größe des Teilnetzes vergrößert und sich die Pfadlänge verlängert.

Bei den beiden Implementierungen hat man keine kommerzielle Middleware benutzt, welche die Objekte der Anwendungsprogramme direkt in die Datenbank abbildet. Dies wird hier durch manuelle Abbildung realisiert. Vielleicht kann man später kommerzielle Middleware probieren und bessere Performance erzielen.

Die objektrelationale Implementierung hat den Vorteil, daß ihre Datentypen leicht erweiterbar sind. Man kann ihre Performance durch Datenbank-Tuning verbessern.

Anhang 1

In Anhang 1 werden die SQL-Anweisungen, die in der relationalen Implementierung benötigt werden, erläutert. In Unterschnitt 1.1 werden SQL-Anweisungen zur Erzeugung der Tabellen erläutert. In Unterabschnitt 1.2 werden SQL-Anweisungen zur Bewahrung der Integritätsbedingung, „von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden.", erläutert. In Unterabschnitt 1.3 werden SQL-Anweisungen zur Bewahrung der Integritätsbedingung, „Der Name eines Typs muß eindeutig sein.", erläutert. In Unterabschnitt 1.4 werden Bemerkungen zu system-generated Objektsidentifikation bei der relationalen Modellierung erläutert.

1.1 SQL-Anweisung zur Erzeugung der Tabellen

Die folgende SQL-Anweisung erzeugt die Tabelle `rType`.

```
CREATE TABLE rType(
OID CHAR(13) FOR BIT DATA NOT NULL PRIMARY KEY,
Name VARCHAR(30) NOT NULL,
Father CHAR(13) FOR BIT DATA NOT NULL,
Remark VARCHAR(30) NOT NULL,
Type VARCHAR(30) NOT NULL,
CONSTRAINT fk1 FOREIGN KEY(Father)
REFERENCES rType(OID) ON DELETE CASCADE)
```

Die folgende Anweisung erzeugt die Tabelle `RelationElement`.

```
CREATE TABLE RelationElement(
OID CHAR(13) FOR BIT DATA NOT NULL PRIMARY KEY,
Name VARCHAR(30),
Start CHAR(13) FOR BIT DATA NOT NULL,
End CHAR(13) FOR BIT DATA NOT NULL,
Type CHAR(13) FOR BIT DATA NOT NULL,
Remark VARCHAR(30) Not NULL,
CONSTRAINT fk1 FOREIGN KEY(Start)
REFERENCES RelationElement(OID) ON DELETE CASCADE,
CONSTRAINT fk2 FOREIGN KEY(End)
REFERENCES RelationElement(OID) ON DELETE CASCADE,
CONSTRAINT fk3 FOREIGN KEY(Type)
REFERENCES rType(OID) ON DELETE CASCADE)
```

1.2 SQL-Anweisungen zur Bewahrung der Integritätsbedingung: von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden.

Die Trigger `rtrig1,rtrig2` untersucht, ob zwei Relationselemente schon eine Relation von einem Relationstyp haben, wenn eine Relation eingefügt wird. Wenn eine solche existiert, werden die Trigger ausgelöst.

```
CREATE TRIGGER rtrig1
NO CASCADE BEFORE INSERT ON RelationElement
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM RelationElement where Start=neuzeile.Start
and End=neuzeile.End and Type=neuzeile.Type and Remark<>'Node'))
SIGNAL SQLSTATE '70000'('element already exists, can not insert')

CREATE TRIGGER rtrig2
NO CASCADE BEFORE INSERT ON RelationElement
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM RelationElement where Start=neuzeile.End
and End=neuzeile.Start and Type=neuzeile.Type and Remark<>'Node'))
SIGNAL SQLSTATE '70000'('element already exists, can not insert')
END
```

Der Trigger `rtrig3` untersucht, ob zwei Relationselemente(Node oder Relation) schon eine Relation von einem Relationstyp haben, wenn eine Relation erneuert wird. Wenn es existiert, wird der Trigger ausgelöst.

```
CREATE TRIGGER rtrig3
NO CASCADE BEFORE UPDATE ON RelationElement
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM RelationElement where Start=neuzeile.End
and End=neuzeile.Start and Type=neuzeile.Type and Remark<>'Node'))
SIGNAL SQLSTATE '70000'('element already exists, can not update')
END
```

1.3 SQL-Anweisung zur Bewahrung der Integritätsbedingung: Der Name eines Typs muß eindeutig sein.

Der Typenname muß eindeutig sein. Der Trigger `rtrig4` überwacht bei der Einfügung, ob ein Typenname schon existiert.

```
CREATE TRIGGER rtrig4
NO CASCADE BEFORE INSERT ON rType
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM rType where Name=neuzeile.Name ))
SIGNAL SQLSTATE '70000'('name already exists, can not insert')
```

Der Typenname muß eindeutig sein. Der Trigger `rtrig5` überwacht bei der Erneuerung, ob ein Typenname schon existiert.

```
CREATE TRIGGER rtrig5
NO CASCADE BEFORE UPDATE ON rType
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM rType where Name=neuzeile.Name ))
SIGNAL SQLSTATE '70000'('name already exists, can not update')
```

1.4 Bemerkung zu system-generated Objektsidentifikation bei der relationalen Modellierung

Um die Objektidentifikation(OID) direkt vom System zu erzeugen, wird hier Funktion `GENERATE_UNIQUE()` von DB2 benutzt.

Weil Nodes zusammen mit der Relationen in der Tabelle `RelationElement` gespeichert werden, muß man die Attribute `Start` und `End` von Node auch mit einfügen. Hier werden die Werte von `Start` und `End` gleich wie der Wert von Attribut `OID` eingefügt. Weil der Wert von `OID` durch Funktion `GENERATE_UNIQUE()` erzeugt wird, werden die Werte von `Start` und `End` zum Zeitpunkt der Erzeugung der OID eingefügt. Der Trigger `rtrig6` erledigt diese Aufgabe. Die genaue Definition ist im folgenden gezeigt.

```
CREATE TRIGGER rtrig6
NO CASCADE BEFORE INSERT ON RelationElement
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (neuzeile.Remark='Node')
BEGIN ATOMIC
SET neuzeile.OID=GENERATE_UNIQUE();
SET neuzeile.Start=neuzeile.OID;
SET neuzeile.End=neuzeile.OID;
END
```

Für den Vatertyp der Tabelle `rType` wird das Attribut `Father` gleich wie der Wert der `OID` eingefügt. Weil der Wert der `OID` durch Funktion `GENERATE_UNIQUE()` erzeugt wird, werden Werte von `Father` zum Zeitpunkt der Erzeugung der `OID` eingefügt. Das Attribut `Remark` überwacht, ob ein Vatertyp eingefügt wird. Wenn dies der Fall ist, wird der Trigger ausgelöst. Der Trigger `rtrig7` erledigt diese Aufgabe. Die genaue Definition von `rtrig7` ist im folgenden gezeigt.

```
CREATE TRIGGER rtrig7
NO CASCADE BEFORE INSERT ON rType
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (neuzeile.Remark='Father')
BEGIN ATOMIC
SET neuzeile.OID=GENERATE_UNIQUE();
SET neuzeile.Father=neuzeile.OID;
END
```

Anhang 2

In Anhang 2 werden die SQL-Anweisungen, die in der objektrelationalen Implementierung benötigt werden, erläutert. In Unterabschnitt 2.1 werden SQL-Anweisungen zur Erzeugung der strukturierten Typen erläutert. In Unterschnitt 2.2 werden SQL-Anweisungen zur Erzeugung der Tabellenhierarchie erläutert. In Unterabschnitt 2.3 werden SQL-Anweisungen zur Bewahrung der Integritätsbedingung, „von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden.", erläutert. In Unterabschnitt 2.4 werden SQL-Anweisungen zur Bewahrung der Integritätsbedingung, „Der Name eines Typs muß eindeutig sein.", erläutert. In Unterabschnitt 2.5 werden Bemerkungen zu system-generated Objektsidentifikation bei der objektrelationalen Modellierung erläutert.

2.1 SQL-Anweisungen zur Erzeugung der strukturierten Typen

Die strukturierte Typen in DB2 kann man durch SQL-Anweisungen wie folgt erstellen.

Die folgende Anweisung erzeugt den Typ `Element_t`.

```
CREATE TYPE Element_t AS (Name VARCHAR(30))
INSTANTIABLE
REF USING VARCHAR(16) FOR BIT DATA
MODE DB2SQL
```

Die folgende Anweisung erzeugt den Typ `Type_t`.

```
CREATE TYPE Type_t UNDER Element_t AS(
Father Ref(Type_t),
Remark VARCHAR(30),
Type VARCHAR(30))
MODE DB2SQL
```

Die folgende Anweisung erzeugt den Typ `Node_t`.

```
CREATE TYPE Node_t UNDER Element_t AS (
Type Ref(Type_t))
MODE DB2SQL
```

Diese Anweisung erzeugt den Typ `Relation_t`.

```
CREATE TYPE Relation_t UNDER Element_t AS (
Start Ref(Element_t),
End Ref(Element_t),
Type Ref(Type_t),
Remark VARCHAR(30))
MODE DB2SQL
```

2.2 SQL-Anweisung zur Erzeugung der Tabellenhierarchie

Wie in Abschnitt 3.4 schon erwähnt baut man aufgrund der Typenhierarchie eine typisierte Tabellenhierarchie auf. Durch die Hilfe von SQL-Anweisungen kann man die folgenden typisierten Tabellen in DB2 erzeugen.

Die folgende Anweisung erzeugt die Tabelle `Element` vom Typ `Element_t`.

```
CREATE TABLE Element OF Element_t
(REF IS Oid USER GENERATED )
```

Die folgende Anweisung definiert Constraint `uc`, das `OID` eindeutig definiert, weil der Objektidentifikator bei Definition der referentiellen Integrität eindeutig definiert werden muß.

```
ALTER TABLE Element ADD CONSTRAINT uc UNIQUE(Oid)
```

Die folgende Anweisung erzeugt die Tabelle `Type` vom Typ `Type_t`.

```
CREATE TABLE Type OF Type_t UNDER Element
INHERIT SELECT PRIVILEGE
(Father WITH OPTIONS SCOPE TYPE NOT NULL,
 Remark WITH OPTIONS NOT NULL,
 Type WITH OPTIONS NOT NULL)
```

Die folgende Anweisung definiert Constraint `tfk1`. Das Constraint garantiert, wenn ein Vatertyp gelöscht wird, dann werden seine alle Kintypen gelöscht.

```
ALTER TABLE Type ADD CONSTRAINT tfk1 FOREIGN KEY(Father) REFERENCES Type
(Oid) ON DELETE CASCADE
```

Die folgende Anweisung erzeugt die Tabelle Relation vom Typ `Relation_t`.

```
CREATE TABLE Relation OF Relation_t UNDER Element
INHERIT SELECT PRIVILEGES
(Start WITH OPTIONS SCOPE ELEMENT NOT NULL,
 End WITH OPTIONS SCOPE ELEMENT NOT NULL,
 Type WITH OPTIONS SCOPE TYPE NOT NULL,
 Remark WITH OPTIONS NOT NULL,
CONSTRAINT rfk1 FOREIGN KEY(Start) REFERENCES Element(Oid) ON DELETE
CASCADE,
CONSTRAINT rfk2 FOREIGN KEY(End) REFERENCES Element(Oid) ON DELETE
CASCADE,
CONSTRAINT rfk3 FOREIGN KEY(Type) REFERENCES Type (Oid) ON DELETE
CASCADE)
```

Die folgende Anweisung erzeugt die Tabelle `Node` vom Typ `Node_t`.

```
CREATE TABLE Node OF Node_t UNDER Element
INHERIT SELECT PRIVILEGES
( Type WITH OPTIONS SCOPE TYPE NOT NULL,
 CONSTRAINT nfk1 FOREIGN KEY(Type) REFERENCES Type(Oid) ON DELETE
 CASCADE)
```

2.3 SQL-Anweisungen zur Bewahrung der Integritätsbedingung: von einem Relationstyp kann zwischen zwei Relationselemente (Node oder Relation) nur eine Relation hergestellt werden.

Der Trigger `trig1` untersucht bei der Einfügung, ob eine Relation zwischen zwei Relationselementen (Node oder Relation) schon existiert.

```
CREATE TRIGGER trig1
NO CASCADE BEFORE INSERT ON Relation
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM Relation where Start=neuzeile.Start and
End=neuzeile.End and Type=neuzeile.Type))
SIGNAL SQLSTATE '70000'('element already exists,can not insert')
```

Der Trigger `trig2` untersucht bei der Einfügung, ob eine Relation von einem Relationstyp zwischen zwei Relationselementen schon existiert.

```
CREATE TRIGGER trig2
NO CASCADE BEFORE INSERT ON Relation
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM Relation where Start=neuzeile.End and
End=neuzeile.Start and Type=neuzeile.Type))
SIGNAL SQLSTATE '70000'('element already exists,can not insert')
```

Der Trigger `trig3` untersucht bei der Erneuerung, ob eine Relation von einem Relationstyp zwischen zwei Relationselementen schon existiert.

```
CREATE TRIGGER trig3
NO CASCADE BEFORE UPDATE ON Relation
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM Relation where Start=neuzeile.End and
End=neuzeile.Start and Type=neuzeile.Type))
SIGNAL SQLSTATE '70000'('element already exists,can not update')
```

2.4 SQL-Anweisung zur Bewahrung der Integritätsbedingung: Der Name eines Typs muß eindeutig sein.

Der Typenname muß eindeutig sein. Der Trigger `trig4` überwacht bei der Einfügung, ob ein Typenname schon existiert.

```
CREATE TRIGGER trig4
NO CASCADE BEFORE INSERT ON Type
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM Type where Name=neuzeile.Name ))
SIGNAL SQLSTATE '70000'('name already exists, can not insert')
```

Der Typenname muß eindeutig sein, Der Trigger `trig5` überwacht bei der Erneuerung, ob ein Typenname schon existiert.

```
CREATE TRIGGER trig5
NO CASCADE BEFORE UPDATE ON Type
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (EXISTS (SELECT * FROM Type where Name=neuzeile.Name ))
SIGNAL SQLSTATE '70000'('name already exists, can not update')
```

2.5 Bemerkungen zu system-generated Objektsidentifikation bei der objekt-relationalen Modellierung

Ähnlich wie in der relationalen Modellierung wird der Wert des Attributs `Father` der Tabelle `Type` im Fall von Vatertypen gleich wie der Wert der OID eingefügt. Weil der Wert der OID durch Funktion `GENERATE_UNIQUE()` erzeugt wird, werden Werte von `Father` zum Zeitpunkt der Erzeugung der OID eingefügt. Der Trigger `trig6` erledigt diese Aufgabe.

```
CREATE TRIGGER trig6
NO CASCADE BEFORE INSERT ON Type
REFERENCING NEW AS neuzeile
FOR EACH ROW MODE DB2SQL
WHEN (neuzeile.Remark='Father')
BEGIN ATOMIC
SET neuzeile.OID=Type_t(GENERATE_UNIQUE());
SET neuzeile.Father=neuzeile.OID;
END
```

Literatur

[Cha99] Champerlin, Don: DB2 Universal Database der unentbehrliche Begleiter. Bonn: Addison-Wesley-Longmann,1999

[Fis99] Fisher Maydene: The JDBC Tutorial and References, Second Edition: Chapter 3-Advanced Tutorial. Webseite von Firma SUN, 1999

[Foe01] Webseite von Föderal: http://www.foederal.org

[IBM00a] IBM: IBM DB2 Universal Database - SQL Reference, Version 7. IBM, 2000

[IBM00b] IBM: IBM DB2 Universal Database - Command Reference, Version 7. IBM, 2000

[IBM00c] IBM: IBM DB2 Universal Database - Application Development Guide, Version 7. IBM, 2000

[KE97] Kemper A.; Eickler A. : Datenbanksysteme - Eine Einführung. München: R. Oldenbourg Verlag 1997

[Lit01] Litto, Marco: Konzeption der föderale Informationsarchitektur. Technischer Bericht, Firma Mind8, 2001

[Man01] Mangold, Christoph: Ein herbstliches Arbeitspapier. Interner Technischer Bericht, Universität Stuttgart, 2001

[STS97] Saake, Gunter; Türker, Can; Schmitt Ingo: Objektdatenbanken. Bonn; Albany; Attenkirchen: International Thomson Publishing GmbH, 1997

[SM99] Stonebraker, Michael; Moore, Dorothy: Objektrelationale Datenbanken: die nächste große Welle. München; Wien : Hanser, 1999

[SUN] SUN: Java-Webseite von SUN: http://www.java.sun.com

ERKLÄRUNG

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe. Ein Musterexemplar liegt bei der die Arbeiten entgegennehmenden Stelle zur Einsicht aus.

Stuttgart, den 31.01. 2002 Yiping Chen

_____

Unterschrift