Datenbankmodelle und Datenbanksprachen

Florian Matthes Joachim W. Schmidt

Technische Universität Harburg, Harburger Schlo"sstr.
D-22527 Hamburg, Germany
{f.matthes,j.w.schmidt}@sts.tu-harburg.de

24. Juli 1997

Inhaltsverzeichnis

Datenbanken haben während der letzten drei Jahrzehnte ihre Rolle als Schlüsseltechnologie für die effiziente Realisierung komplexer *Informationssysteme* ständig ausgebaut. Hintergrund dieser Entwicklung ist die Tatsache, daß die Fortschritte der *Datenbanksystemtechnologie* es ermöglichen,

- immer größere integrierte Informationsbestände zu verwalten (Megabyte, Gigabyte, Terabyte an Informationen),
- immer neue Informationsstrukturen adäquat zu repräsentieren (kaufmännische, textuelle, wissenschaftliche, statistische, audio-visuelle Informationen) und
- in immer neuen Modalitäten auf Informationsbestände zuzugreifen (Hintergrundverarbeitung, interaktive Recherche, entfernter Datenbankzugriff in Client/Server Architekturen und in Weitverkehrsnetzen).

Gleichzeitig ist die stark zunehmende Popularität von Datenbanken auf kontinuierliche Fortschritte im Verständnis der anwendungsorientierten *Modellierungsanforderungen* von Informationssystemen und bei der Entwicklung leistungsfähiger und benutzerfreundlicher *Datenbanksprachen* zurückzuführen. Diese Fortschritte erlauben insgesamt eine Reduktion der *Komplexität* moderner Informationssysteme, die aufgrund des oben skizzierten Anstiegs des Daten-, Datenstruktur- und Funktionsumfangs dringend erforderlich ist. Das starke Interesse von Anwendern und Systementwicklern an neuartigen objektorientierten und aktivitätsorientierten Datenbankmodellen belegt die Bedeutung problemadäquater Modellierungs- und Sprachkonzepte für die effiziente und effektive Realisierung von Informationssystemen.

Dieses Kapitel bietet eine Einführung in die Grundlagen der Datenbankmodellierung und in die bewährten Konzepte moderner Datenbanksprachen.

Bei der Präsentation des Materials wird der zunehmenden Dynamik des Marktes der Datenbanksysteme und der Reife der Informatikdisziplin „Datenbanken“ Rechnung getragen. Daher wird eine starke Orientierung an konkreten Datenbank*systemen*, wie zum Beispiel IBM IMS, DEC DBMS 10, Ingres QUEL und O_2 vermieden, Bei einer solchen Form der Einführung überlagern die historisch gewachsenen System- und Spracheigenheiten zu stark die Diskussion von Modellierungskonzepten, Datenbanksprachkonstrukten und Alternativen des Datenbankdienstzugriffs.

Obwohl der Schwerpunkt dieses ersten Kapitels also auf *allgemeingültigen* Modell- und Sprachprinzipien liegt, werden ausschließlich *konkrete* Beispiele in der Syntax praxisrelevanter standardisierter Sprachen (zum Beispiel SQL) benutzt, und es werden keine abstrakten Notationen zum Zwecke der Semantikdefinition eingeführt.

Im Vorgriff auf Abschnitt 1.1.2 soll die folgende Analogie mit vertrauten Begriffen der Informatikdisziplin „Programmiersprachen“ die Bedeutung von Datenbankmodellen und Datenbanksprachen für Datenbanksysteme und damit dieses Kapitels für das gesamte Datenbankhandbuch verdeutlichen:

Eine *Datenbanksprache* (Ingres SQL Release 6.0) entspricht einer Programmiersprache (Microsoft C++) mit konkreten syntaktischen Strukturen und implementierungsspezifischen Details. Die Verwendung *standardisierter* Datenbanksprachen (SQL-92, ODMG-93) und standardisierter Programmiersprachen (ISO C++) erhöht die Portabilität, Interoperabilität und Wartbarkeit der mit diesen Sprachen entwickelten Systeme.

Ein *Datenbanksystem* (Ingres System 6.0 für VAX/VMS) oder ein *Datenbankdienst* entspricht einem Compiler für eine Programmiersprache (Microsoft C++ für Intel 80486 Plattformen) und kann als eine generische Implementierung für alle möglichen Deklarationen und Ausdrücke einer Datenbanksprache verstanden werden.

Ein *Datenbankmodell* (Relationales Datenmodell nach Codd) beschreibt die Semantik von Deklarationen und Ausdrücken der Datenbanksprache und entspricht somit einem abstrakten Modell der statischen und dynamischen Semantik einer Programmiersprache, das zum Beispiel durch formale Typ- und Auswertungsregeln beschrieben wird.

Eine *Datenbankmodellfamilie* (Relationale Datenmodelle) charakterisiert die gemeinsamen Eigenschaften verwandter Datenbankmodelle (zum Beispiel assoziative Elementidentifikation, Kopiersemantik, mengenorientierte Verarbeitung) in Analogie zu einer Programmiersprachenfamilie (imperative, funktionale oder deklarative Sprachen).

Dieses erste Kapitel des Datenbankhandbuchs ist wie folgt organisiert:
`Nicht in der endgültigen Version? (zuviel Redundanz?))`

- Abschnitt 1.1 definiert zunächst *Datenbanksysteme als generische Diensterbringer* für die Realisierung von Informationssystemen. Aus den spezifischen Charakteristika von Informationssystemen (Persistenz, Quantität, Reaktivität und Integrität) werden konkrete Anforderungen an die Ausgestaltung von Datenbankmodellen und Datenbanksprachen abgeleitet.

- Abschnitt 1.2 befaßt sich mit bewährten *modellunabhängigen Datenbankkonzepten* (zum Beispiel Schemaschichtung, Typorthogonalität und orthogonale Persistenz) und gibt einen kurzen Abriß der historischen Entwicklung von Datenbanksystemen und ihren Modellen und Sprachen.

- Abschnitt 1.3 stellt die *charakteristischen Strukturen und Operationen* der wichtigsten Datenbankmodellfamilien am Beispiel eines zentralen Datenmodellierungsproblems, der Repräsentation einer n:m Beziehung, vor. Dabei werden relationale, objektorientierte, hierarchische und netzwerkartige Modelle gegenübergestellt.

- Abschnitt 1.4 und 1.5 bilden eine Einführung in die Syntax und Semantik *konkreter Sprachkonstrukte* standardisierter relationaler und objektorientierter Datenbanksprachen.

- Abschnitt 1.6 diskutiert ausgewählte *Datenbankmodell- und Datenbanksprachvarianten*. Neben formalen Datenbanksprachen werden dabei insbesondere auch praxisrelevante und standardisierte Systeme erwähnt und es wird auf weiterführende Literatur verweisen.

- Abschnitt 7 behandelt alternative *Dienst- und Sprachschnittstellen* in modernen Datenbanksystemen (zum Beispiel statische und dynamische Gastspracheneinbettungen sowie bibliotheksbasierte Aufrufschnittstellen).

- Abschnitt 8 erläutert die Problematik der *Datenbanksprachintegration*.

1 Informationssysteme und Datenbanksysteme

In diesem Abschnitt wird zunächst eine wichtige Klasse von Softwaresystemen vorgestellt, die als *Informationssysteme* bezeichnet werden, und die sich durch charakteristische Eigenschaften (Persistenz, Quantität, Reaktivität, Integrität) von anderen Softwaresystemen unterscheiden (Abschnitt 1.1.1).

Darauf aufbauend werden *Datenbanksysteme als generische Diensterbringer* zur effizienten Realisierung von Informationssystemen unter besonderer Berücksichtigung dieser charakteristischen Systemeigenschaften beschrieben (Abschnitt 1.1.2). In diesem Zusammenhang werden *Datenbanksprachen* benötigt, um die generischen Dienste eines Datenbanksystems, die durch ein *Datenbankmodell* beschrieben werden, flexibel an die speziellen Anforderungen eines konkreten Informationssystems anzupassen.

Die Persistenz, Quantität, Reaktivität und Integrität von Informationssystemen besitzt wichtige system- und softwaretechnische Konsequenzen, die zu konkreten Anforderungen an Datenbankmodelle, Datenbanksprachen und Datenbanksprachschnittstellen führen (Abschnitt 1.1.3). Die klassischen Datenbankmodelle machen darüber hinaus weitere implizite spezialisierende Annahmen über die Struktur und Nutzung von Informationssystemen, die auch Hinweise für die künftige Weiterentwicklung von Datenbankmodellen liefern (Abschnitt 1.1.4).

Der Einsatz (standardisierter) Datenbankdienste bei der Realisierung von Informationssystemen führt in der Praxis zu erheblichen Produktivitäts- und Qualitätsverbesserungen über den gesamten Softwarelebenszyklus hinweg, die in Abschnitt 1.1.5 genauer beleuchtet werden.

Dieser Abschnitt schließt mit der Beschreibung eines prototypischen Firmeninformationssystems, das durchgängig als Beispiel im ersten Kapitel verwendet wird (Abschnitt 1.1.6).

1.1 Charakteristika von Informationssystemen

Spätestens Ende der 60er Jahre hat sich recht deutlich ein Wandel im Einsatz von Softwaresystemen vollzogen. Computer sind als Instrumente zur Lösung numerischer Berechnungsaufgaben zunehmend in den Hintergrund getreten und werden allgemein als Systeme zur Repräsentation und Verarbeitung von *Informationen* eingesetzt. Die meisten der heute kommerziell entwickelten Softwaresysteme sind sogenannte *Informationssysteme*, die oft auch als datenintensive Anwendungen oder spezieller als Datenbankanwendungen bezeichnet werden.

Informationssysteme wurden historisch gesehen zunächst in Verwaltungsabteilungen grosser Organisationen, wie Versicherungen, Banken, Telekommunikationsunternehmen oder Versandunternehmen, für kaufmännische informationsverarbeitende Aktivitäten eingesetzt. Durch die weite Verbreitung von Computern als Arbeitsplatz-, Gruppen- und Abteilungsrechner existieren jedoch heutzutage auch zahlreiche dezentralisierte Informationssysteme für Informationsbestände verschiedenster Größe und Komplexität, die von unterschiedlichen Benutzergruppen in verschiedenen Modalitäten (lokal und entfernt, interaktiv und programmgestützt) bearbeitet werden.

Allgemein läßt sich ein *Informationssystem* als ein Softwaresystem mit folgenden charakteristischen Eigenschaften definieren („PQRI-Anforderungen“ , s. auch Abbildung 1):

Persistenz (Langlebigkeit): Ein Informationssystem verwaltet einen Informationsbestand, dessen Lebensdauer nicht durch die Lebensdauer individueller informationsverarbeitender Aktivitäten beschränkt ist.

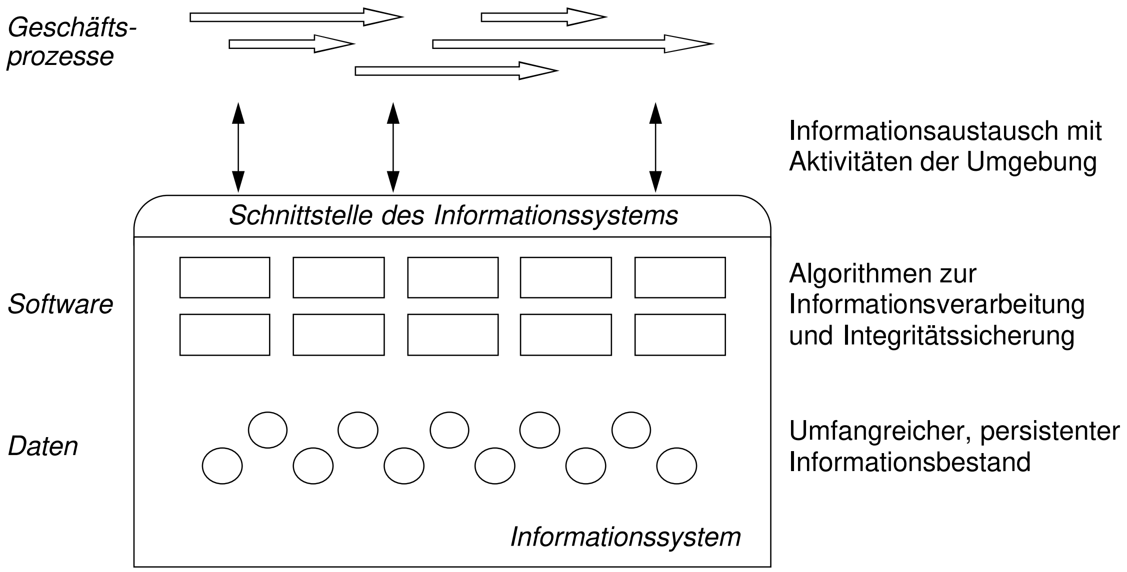

Abbildung 1: Schematische Struktur eines Informationssystems

Quantität: Ein Informationssystem verwaltet einen umfangreichen, regulär strukturierten Informationsbestand, dessen Struktur zum Entwurfszeitpunkt des Informationssystems fixiert wird, dessen Größe jedoch über die gesamte Lebensdauer des Informationssystems hinweg dynamisch variiert.

Reaktivität: Ein Informationssystem ist in die informationsverarbeitenden Aktivitäten seiner Umgebung eingebunden. Es aktualisiert seinen Informationsbestand unmittelbar basierend auf einkommenden Informationen, antwortet auf explizite Informationsanfragen und löst auch selbständig Aktivitäten in der Umgebung aus.

Integrität: Ein Informationssystem wahrt Integritätsbedingungen, die sich sowohl auf seinen persistenten Informationsbestand als auch auf die einkommenden und ausgehenden Informationen beziehen können.

Der Vergleich zwischen Abbildung 1 und Abbildung 2 zeigt, wie *Datenbanksysteme als Diensterbringer* bei der Realisierung von Informationssystemen eingesetzt werden. Dabei werden die Aufgaben der persistenten Datenspeicherung und des effizienten Massendatenzugriffs an ein Datenbanksystem delegiert. Darüber hinaus können Teilaufgaben der Integritätsverwaltung und der Realisierung der Reaktivität ebenfalls durch das Datenbanksystem übernommen werden. Das *Datenbankschema* definiert dabei die Schnittstelle zwischen den verbleibenden informationsverarbeitenden Algorithmen des Informationssystems und den Diensten der Datenbank.

In Abbildung 2 wird, wie auch im folgenden Text, zwischen *Daten* und *Informationen* unterschieden. Daten sind kodierte, maschinenlesbare *Repräsentationen* von Informationen. Umgekehrt werden Informationen aus Daten abgeleitet und zum *Problemlösen* eingesetzt (s. Reaktivität). Diese Unterscheidung hat sich in der Informatik allgemein durchgesetzt und liegt auch zahlreichen Standards, wie ISO, DIN, etc., zugrunde [xx]. An dieser Stelle genügt es darauf hinzuweisen, daß teilweise nur ein gradueller Unterschied zwischen Daten

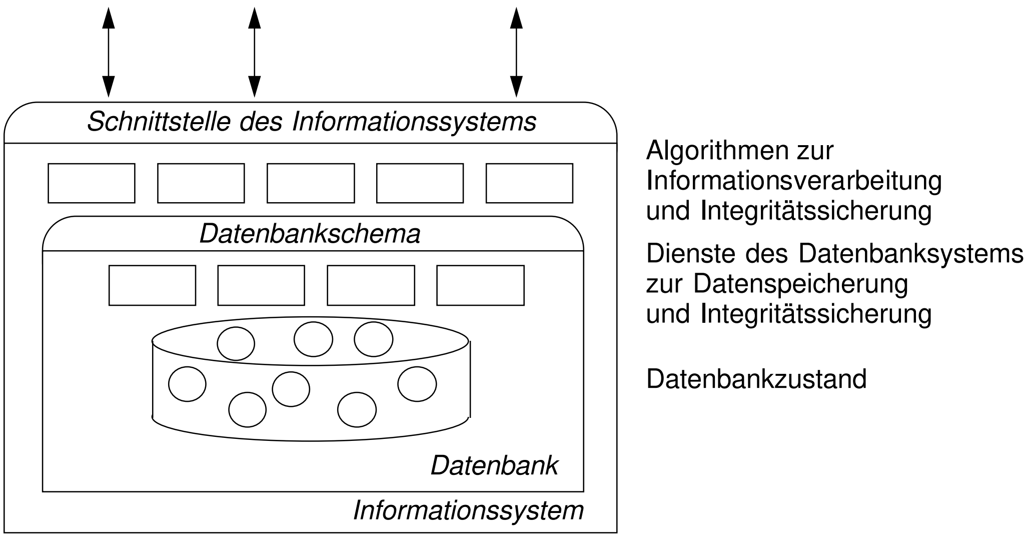

Abbildung 2: Realisierung eines Informationssystems mit einer Datenbank

und Informationen besteht, wobei zum Beispiel der Begriff Datenbestand eine eher *maschinenorientierte* Sicht und der Begriff Informationsbestand eine eher *problemorientierte* Sicht eines Systemzustandes bezeichnet.

Eine *Datenbank* ist also ein Softwaredienst zur Verwaltung eines organisierten, langlebigen, strukturierten Datenbestands, während ein *Informationssystem* zusätzlich zu der Datenbank noch Algorithmen zur Interpretation der Daten als Informationen und umgekehrt zur Repräsentation von Informationen durch Daten umfaßt.

1.2 Dienstgestützte Realisierung von Informationssystemen

Ein wesentliches Ziel der angewandten Informatik ist die Entwicklung von Modellen, Sprachen und *generischen* Diensterbringern oder Werkzeugen, die den Vorgang der Anwendungs- und Systementwicklung effizienter und effektiver gestalten. Im Unterschied zu *spezifischen* Diensten und Softwarewerkzeugen, wie zum Beispiel Druckdiensten, e-mail Diensten, Finanzbuchhaltungssoftware oder Textverarbeitungssoftware, die ein konkretes Anwendungsproblem lösen, leisten generische Dienste und Werkzeuge einen Beitrag für eine gesamte *Problemklasse*. So sind zum Beispiel Programmiersprachen generische Werkzeuge zur algorithmischen Programmierung, GUI-Toolkits sind generische Werkzeuge zur Benutzerschnittstellengestaltung, und Kommunikationssoftware bietet generische Dienste zur Anwendungsentwicklung in Netzwerken.

Voraussetzung für die Entwicklung erfolgreicher generischer Dienste und Werkzeuge ist eine adäquate Konzeptualisierung der Problemklasse durch *Modellbildung*, typischerweise gepaart mit der Entwicklung modellunterstützender *sprachlicher Notationen*. Diese Notationen werden schließlich von den Dienstbenutzern zur *Dienstspezialisierung* bei der Benutzung der generischen Werkzeuge eingesetzt.

So erlauben zum Beispiel eine Programmiersprache die Definition eines konkreten Algorithmus', eine GUI Notation die Definition einer konkreten Benutzerschnittstelle und eine Protokollbeschreibungssprache die Definition einer konkreten Client/Server Verbindung.

In diesem Sinn sind Datenbanksysteme generische Diensterbringer (*general purpose software*), die für die Lösung der charakteristischen, bei der Realisierung von *Informationssystemen* immer wiederkehrenden PQRI-Anforderungen eingesetzt werden, und Datenbanksprachen sind Notationen zur Dienstspezialisierung gemäß eines vorgegebenen Datenbankmodells.

Wie in Abbildung 3 dargestellt, wird also der Implementierungsaufwand für die in Abbildung 2 angedeutete Datenbank im wesentlichen auf die Beschreibung des spezifischen Datenbankschemas reduziert, das eine Spezialisierung des generischen Datenbankdienstes an die Anforderungen des speziellen Informationssystems leistet.

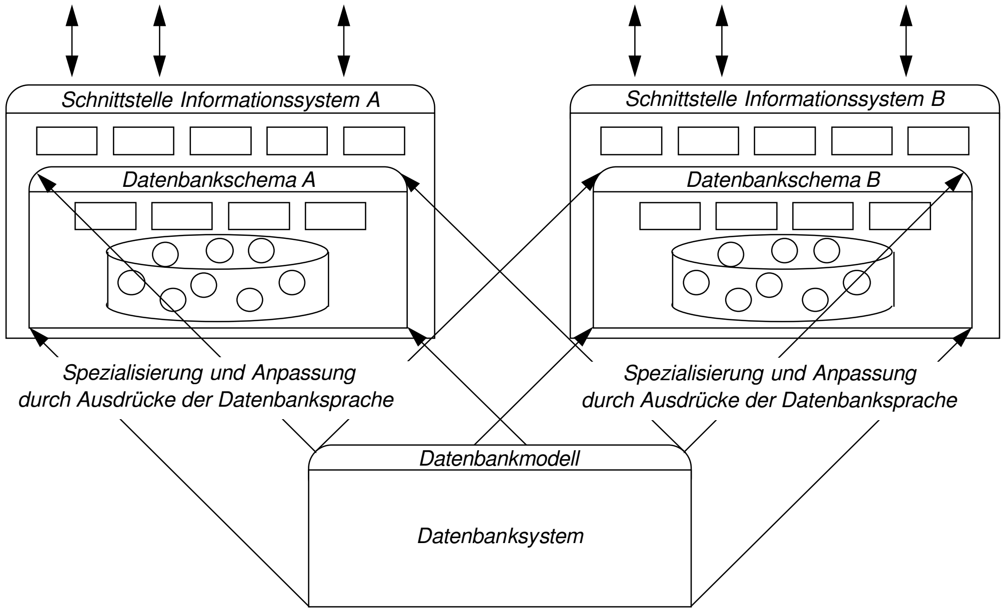

Abbildung 3: Datenbanksysteme als generische Diensterbringer bei der Realisierung von Informationssystemen

Zur Realisierung der Datenbanken zu den Informationssystemen A und B in Abbildung 3 genügt also die Definition von geeigneten Datenbankschemata. Die Dienste zur Speicherung der Daten und zur Integritätsüberwachung werden in generischer Form vom Datenbanksystem zur Verfügung gestellt und durch die Datenbankschemata für die speziellen Anforderungen des Informationssystems A bzw. B spezialisiert. Es ist dabei unerheblich, ob beide Informationssysteme auf dem gleichen physikalischen Datenbanksystem arbeiten oder jedes auf einer eigenen Ausprägung, wichtig ist, daß der Aufwand für die Erstellung dieses Systems nur einmal anfällt.

An dieser Stelle ist es hilfreich, den Zusammenhang zwischen den Begriffen Datenbanksystem, Datenbankmodell, Datenbankschema, Datenbank und Datenbanksprache und ihren Analogien auf dem Gebiet der Programmiersprachen explizit aufzuzeigen.

Ein *Datenbankschema* beschreibt die Menge der Invarianten auf einer speziellen Datenbank. Diese Invarianten umfassen die Namen der Datenbankobjekte, ihre (reguläre) Strukturen, die Beziehungen zwischen Datenbankobjekten, sowie die verfügbaren Datenbankoperationen (Lesen, Einfügen, Löschen, Aktualisieren). Ein Datenbankschema ähnelt daher einer Modulschnittstelle, die Konstanten, Typen, typisierte Variablen und Prozedursignaturen definiert.

Eine *Datenbank* ist ein dynamischer, langlebiger Datenbestand, der den Invarianten seines Datenbankschemas genügt. Eine Datenbank ähnelt daher einer Modulimplementierung, die typkompatible Implementierungen aller in ihrer Schnittstelle genannten Namen exportiert. Die Semantik von Datenbanken unterscheidet sich jedoch wie folgt erheblich von der traditionellen Modulsemantik:

1. Die Zustände der exportierten Objekte einer Datenbank werden *persistent* (langlebig) gespeichert.
2. Mehrere Prozesse, die simultan auf der gleichen Datenbank arbeiten, referenzieren gemeinsam genutzte Objekte (*shared objects*), während im Programmiersprachenkontext mehrere Prozesse, die das gleiche Modul importieren, prozeßlokale Kopien der Modulobjekte (insbesondere der Modulvariablen) erhalten.

Ein *Datenbankmodell* definiert die Strukturierungsmechanismen, Identifikationskonzepte und generischen Operationen, die zur Konstruktion eines Datenbankschemas und zur Beschreibung von Datenbankoperationen zur Verfügung stehen. Ein Datenbankmodell ähnelt daher einem abstrakten Programmiersprachenmodell.

Ein *Datenbanksystem* stellt generische Implementierungen aller Operationen und Datenstrukturen eines Datenbankmodells zur Verfügung. Datenbanksysteme für einfache Datenbankmodelle (zum Beispiel für das Netzwerkdatenmodell) können in Analogie zu generischen Modulen in Programmiersprachen gesehen werden: Ein Klient eines generischen Moduls spezialisiert die Modulfunktionalität durch (Typ-) Parametrisierung für seine speziellen Bedürfnisse. Datenbanksysteme für moderne Datenbankmodelle erfordern hingegen wesentlich leistungsfähigere Parametrisierungskonzepte, um zum Beispiel deklarative Datenbankanfragen, Integritätsverwaltung und Aktivitätssteuerung zu unterstützen.

Eine *Datenbanksprache* bietet diese weitergehenden Parametrisierungskonzepte, die zur Spezialisierung der Datenbanksystemdienste erforderlich sind. Sie definiert insbesondere Benennungsmechanismen, Sichtbarkeitsregeln, Lebensdauerregeln und Kombinationsregeln für benutzerdefinierte Klassen, Beziehungen, Operationen und Integritätsbedingungen. Eine Datenbanksprache ähnelt daher selbst einem spezialisierten Schnittstellenbeschreibungsformalismus (*interface definition language*, IDL), wie er zum Beispiel in modularen Programmiersprachen und in verteilten Objektsystemen zu finden ist.

1.3 Anforderungen an Datenbankmodelle und Datenbanksprachen

Die Persistenz, Quantität, Reaktivität und Integrität von Informationssystemen besitzt konkrete Konsequenzen für die bei ihrer Softwarerealisierung eingesetzten Methoden, Modelle und Werkzeuge, die in den nachfolgenden Unterabschnitten beschrieben werden. Diese Beschreibung liefert gleichzeitig einen Anforderungskatalog für die generischen Dienste von Datenbanksystemen, die, wie im vorangegangenen Abschnitt erläutert wurde, durch Datenbankmodelle und Datenbanksprachen formalisiert werden.

1.3.1 Unterstützung der Persistenz

Ein Informationssystem verwaltet einen Informationsbestand, dessen Lebensdauer nicht durch die Lebensdauer individueller informationsverarbeitender Aktivitäten beschränkt ist.

In der Terminologie von Softwaresystemen werden alle Daten, deren Lebensdauer die Lebensdauer eines einzelnen Betriebssystemprozesses (also einer einzelnen Programmaktivierung) überschreitet, als *persistent* oder langlebig bezeichnet.

Gemäß der klassischen Sicht von Datenbanken ist der persistente Zustand eines Informationssystems ein Modell des *aktuellen* Zustands der Realität, genauer gesagt, der Aspekte der Realität, die für das Informationssystem als relevant erachtet werden. So wird zum Beispiel der Kern eines Reservierungssystems durch die Daten der aktuellen Ressourcenbelegung gebildet.

Die folgenden Beispiele verdeutlichen jedoch, daß innerhalb von Informationssystemen neben dem Datenbankzustand mit quasi unendlicher Lebensdauer auch weitere persistente Datenbestände benötigt werden, deren Lebensdauer in weiten Grenzen variiert:

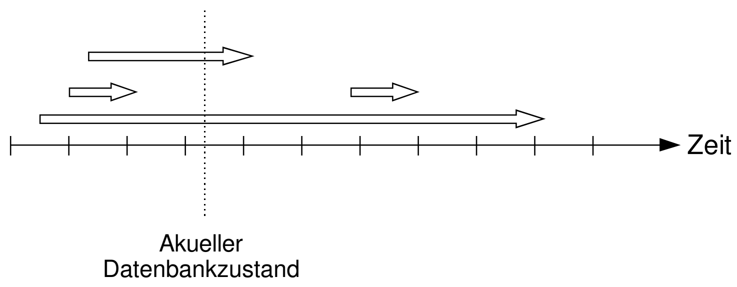

Abbildung 4: Unterstützung der Persistenz

- Speicherung des transienten Zustands einer langandauernden Aktivität (z.B. einer aufwendigen numerischen Berechnung oder einer mehrstufigen Kommunikation mit entfernten Agenten) an einem *Sicherungspunkt*, um eine Fehlererholung im Falle eines Systemausfalls durchzuführen (Minuten).

- Speicherung einkommender Informationsanfragen oder Systemereignisse in *Warteschlangen* bis zu ihrer Bearbeitung durch das Informationssystem (Stunden).

- Speicherung des Zustandes eines *Systemdialogs* (Fensteranordnung, Werkzeugpaletten, Standardeinstellungen) über mehrere interaktive Sitzungen hinweg (Tage).

- Speicherung alternativer *Versionen* von Datenbeständen für experimentelle und vergleichende Studien (Wochen, Monate).

- Speicherung von *Protokollinformationen* über relevante Systemereignisse, die häufig aus juristischen oder betriebswirtschaftlichen Gründen erforderlich sind (Jahre).

- Speicherung von *historischen Datenbanksystemzuständen* zur Archivierung und für temporale Datenbankanfragen (Jahrzehnte).

Der heutige Stand der Technik erlaubt eine persistente Speicherung dynamischer Datenbestände nur auf Sekundärspeichern (magnetische oder optische Festplatten) oder Tertiärspeichern (Bändern, Wechselplatten), wohingegen eine Datenverarbeitung ausschließlich mit Daten im Primärspeicher (RAM) möglich ist. Eine Hauptaufgabe von Datenbanksystemen ist daher die Verwaltung einer *Speicherhierarchie*, d.h. der effiziente Datentransfer zwischen Primär-, Sekundär- und Tertiärspeicher.

Moderne Datenbanksprachen bieten direkte Mechanismen zur Benennung und Manipulation langlebiger Daten und erfordern keine expliziten Lade- und Speicherungsoperationen. Sie erlauben dem Datenbankprogrammierer damit eine *Persistenzabstraktion*. Somit werden die komplexen Details des Datentransfers aus den Algorithmen des Informationssystems eliminiert, und es wird dem Datenbanksystem ein erheblicher Optimierungsspielraum bei der Organisation von Speicherzugriffen gegeben.

Die persistente Speicherung des aktuellen Zustands eines Informationssystems erfordert auch die persistente Speicherung von Beschreibungsinformationen, wie zum Beispiel die Namen, Sichtbarkeitsbereiche und Strukturen der Nutzdaten, damit diese Daten langlebig interpretiert werden können. Die Beschreibungsdaten werden auch als *Metadaten* oder *Schemadaten* bezeichnet, während die *Nutzdaten* den aktuellen Informationssystemzustand beschreiben.

Aufgrund der Langlebigkeit der Informationsbestände sind in praktisch allen Informationssystemen inkrementelle Änderungen der Algorithmen zur Informationsverarbeitung und Integritätssicherung erforderlich, die eventuell auch eine Restrukturierung der Datenbankstrukturen nach sich ziehen, die aber nicht zu einem Verlust der bisher gespeicherten Informationen führen dürfen (*Schemaevolution*).

Bei der Realisierung der informationsverarbeitenden Algorithmen eines Informationssystems benötigt man daher *inkrementelle* und *dynamische Bindungsmechanismen* zur Bindung von Variablennamen in Algorithmen an persistente Datenbankobjekte. Datenbanksprachen bieten daher typischerweise Anweisungen (z.B. connect database, open table, open cursor, etc.), um unter Benutzung der Metadaten zur Programmlaufzeit solche Bindungen inkrementell und dynamisch zu etablieren.

In diesem Zusammenhang bezeichnet der Begriff der *Datenunabhängigkeit* (*data independence*) die (relative) Robustheit der informationsverarbeitenden Aktivitäten gegenüber Änderungen der Datenrepräsentation in der Datenbank. So muß zum Beispiel eine Literaturdatenbank verschiedenartige Anfrageformen, wie z.B. den direkten Zugriff über Signaturen aber auch die Volltextsuche und die Suche über Autoren, unterstützen, die durch sehr unterschiedliche Datenbankzugriffsstrukturen (Streuspeicherung, invertierte Dateien, physische Nachbarschaft auf dem Sekundärspeicher) realisiert werden können. Es ist daher zu erwarten, daß die Datenrepräsentation einer Literaturdatenbank über ihre Lebensdauer hinweg verändert werden muß, zum Beispiel um neuen Benutzeranforderungen gerecht zu werden.

Eine weitere Konsequenz der Persistenz von Informationssystemen ist die Notwendigkeit, den Informationsbestand Ausfälle einzelner Prozesse, Prozessoren oder Kommunikationsverbindungen überdauern zu lassen (*Fehlererholung*). In diesem Zusammenhang wird in Abschnitt 1.4.7 ausschließlich die transaktionale Fehlererholung diskutiert, da sich weitergehende Modell- und Sprachkonzepte zur Fehlererholung (Ausnahmebehandlung, Replikation, partielle Sicherungspunkte) in der Praxis bisher nicht durchgesetzt haben.

1.3.2 Unterstützung der Quantität

Ein Informationssystem verwaltet einen umfangreichen, regulären Informationsbestand, dessen Struktur zum Entwurfszeitpunkt des Informationssystems fixiert werden kann, dessen Größe jedoch über die gesamte Lebensdauer des Informationssystems hinweg dynamisch variiert.

Die *Regularität* der Informationsstrukturen typischer Informationssysteme läßt sich vorteilhaft zur kompakten Datenmodellierung und effizienten Datenmanipulation nutzen. So lassen sich in persistenten Informationsbeständen häufig *Klassen* von anonymen *Objekten* gleicher Struktur unterscheiden. Zwischen den Objekten dieser Klassen bestehen *Beziehungen*, und es können *Invarianten* auf Informationsbeständen formuliert werden, die über die gesamte Lebensdauer des Systems gültig sind.

Zum Beispiel werden die Informationen einer Literaturdatenbank durch Objekte von Klassen, wie *Artikel*, *Journal*, *Report*, *Buch*, *Institution*, *Katalog* und *Standort*, beschrieben, zwischen denen Beziehungen, wie *publiziert-von*, *indiziert-in* und *ausleihbar-an*, existieren. Eine mögliche Invariante einer Literaturdatenbank ist die Forderung, daß jeder Report von genau einer Institution publiziert wird.

Zum Entwurfszeitpunkt eines Informationssystems sind also die Klassen (ihre Namen und Strukturen), die möglichen Beziehungen zwischen den Klassen (ihre Namen und Strukturen) und die Invarianten (Prädikate auf Objekten, Klassen und Beziehungen) bekannt (*statische Typisierung*).Während der gesamten Lebensdauer des Informationssystems werden Objekte dynamisch angelegt und gelöscht, in Beziehung zu anderen Objekten gesetzt, wobei jeweils die Gültigkeit der Invarianten gesichert werden muß (*dynamische Datenspeicherung*). Das *Datenbankschema* eines Informationssystems ist also zeitlich stabil, während sich der *Datenbankzustand* eines Informationssystems dynamisch ändert.

Zur Realisierung von Klassen und Beziehungen werden *Massendatenstrukturen (bulk data types)* eingesetzt. Beispiele für Massendatenstrukturen sind sequentielle Dateien, Tabellen, Listen, Mengen oder Multimengen. Diesen Datenstrukturen ist gemeinsam, daß sie Kollektionen von Elementen mit gleicher Elementstruktur beschreiben, wie z.B. eine Tabelle mit Artikeln oder eine Liste mit Querverweisen, auf denen dynamische Einfüge- und Löschoperationen durchgeführt werden können.

Bei der Realisierung von Algorithmen auf Massendatenstrukturen treten häufig stereotype Programmuster auf, um ähnliche Informationsanfragen zu befriedigen. Beispiele für solche Programmuster sind: Existiert ein Element in der Massendatenstruktur, das eine bestimmte Bedingung erfüllt? Erfüllen alle Elemente der Massendatenstruktur eine bestimmte Bedingung? Erzeuge eine Massendatenstruktur, die alle Elemente einer anderen Massendatenstruktur enthält, die eine bestimmte Bedingung erfüllen, usw.. Es besteht also ein Bedarf nach *Iterationsabstraktion* (existenzielle Quantifizierung, universelle Quantifizierung, Selektion, etc.), die es ermöglicht, stereotype *imperative* Programmuster wie Schleifen, Bedingungen und Rekursionen durch eine *deklarative* Beschreibung der Informationsanfrage zu ersetzen.

Die obige Diskussion von Massendatenstrukturen und Iterationsabstraktionen hebt nur die softwaretechnischen Vorteile in Bezug auf die informationsverarbeitenden Prozesse (geringerer Kodieraufwand, verbesserte Verständlichkeit, Wartbarkeit und Modularität) hervor. Ein in der Praxis mindestens ebenso wichtiges Argument für die Nutzung von Datenbankdiensten für diese Aufgaben ist jedoch die *Effizienz der Datenselektion*, die von Datenbanksystemen mittels Iterationsabstraktionen über Massendatenstrukturen erreicht wird.

Wie bei der Diskussion der Persistenz im vorigen Abschnitt bereits angesprochen, werden die Daten eines Informationssystems langfristig auf Sekundär- und Tertiärspeichern gehalten. Im Gegensatz zu vielen Standardwerkzeugen (Spreadsheets, Textverarbeitungsprogramme, Grafikpakete, CAD Werkzeuge) überschreitet im Regelfall die Gesamtgröße der Informationsstrukturen eines Informationssystems jedoch deutlich die (physische und virtuelle) Hauptspeicherkapazität des zur Informationsverarbeitung eingesetzten Computers. Daher können Informationssysteme nicht wie diese Standardwerkzeuge auf einem einfachen *load / save* Modell aufbauen, bei dem die benötigten persistenten Daten komplett zur Bearbeitung in den Hauptspeicher geladen werden. Andererseits bearbeiten die meisten Aktivitäten in Informationssystemen nur einen kleinen Ausschnitt des gesamten persistenten Datenbestandes. Eine vordringliche Aufgabe bei der Realisierung von Datenbanksystemen ist daher die effiziente Selektion der aktuell benötigten Daten, wobei nur eine möglichst kleine Obermenge der für eine Aktivität benötigten Daten vom Sekundär- oder Tertiärspeicher in den Hauptspeicher transferiert werden darf.

Wie in Kapitel xx (DB-Implementierung) erklärt wird, verfügen Datenbanksysteme über ein reiches Repertoire an Implementierungen für Massendatenstrukturen und Algorithmen zur Realisierung von Iterationsabstraktionen, die auf das spezielle Problem der effizienten Datenselektion auf Sekundärspeichern spezialisiert sind. Daher erreichen Datenbanksysteme generell eine erhebliche Effizienzsteigerung verglichen mit handkodierten Datenselektionen.

Die enorme Bedeutung der effizienten Informationsselektion für die Gesamtperformanz von Datenbanksystemen verdeutlicht der folgende Vergleich der Zugriffszeiten heutiger Primär-, Sekundär- und Tertiärspeicher. Der direkte Zugriff auf eine lokale Variable in einer Programmiersprache läßt sich idealerweise mit einer Maschineninstruktion implementieren und benötigt etwa xx Nanosekunden auf einem Personalcomputer. Der direkte Zugriff auf eine Datenbankvariable erfordert mindestens den Transfer eines Datenblocks von der Festplatte in den Hauptspeicher und benötigt etwa xx Millisekunden. Der direkte Zugriff auf eine bestimmte Bandstelle auf einer VHS Videokassette in einer roboterbedienten Jukebox benötigt etwa xx Sekunden. Setzt man diese Größenverhältnisse in „erfahrbare“ menschliche Zeitintervalle um, so steht ein Hauptspeicherzugriff, der eine Sekunde benötigt, im Verhältnis zu einem Sekundärspeicherzugriff von 15 Minuten und einem Tertiärspeicherzugriff von 11 Tagen und 12 Stunden.

1.3.3 Unterstützung der Reaktivität

Ein Informationssystem ist in die informationsverarbeitenden Aktivitäten seiner Umgebung eingebunden. Es aktualisiert seinen Informationsbestand unmittelbar basierend auf einkommenden Informationen, antwortet auf explizite Informationsanfragen und löst auch selbständig Aktivitäten in der Umgebung aus.

Bei den informationsverarbeitenden Aktivitäten in der Umgebung eines Informationssystems handelt es sich typischerweise um *menschliche Aktivitäten*, wie z.B. die Erledigung von Bankgeschäften, Personalverwaltung, Literaturrecherche, Adreßverwaltung, u.ä., die durch *Agenten* abgewickelt werden. Handelt es sich bei den Agenten um menschliche Nutzer, so wird die Interaktion direkt über (textuelle oder grafische) *Benutzeroberflächen* abgewickelt. Die Interaktion mit computerbasierten Agenten wird über Kommunikationsdienste oder Bibliotheksschnittstellen realisiert (*Dienstschnittstellen*).

Eine zentrale Aufgabe eines Informationssystems ist dabei die implizite oder explizite *Synchronisation* nebenläufiger Aktivitäten, die auf den gleichen Informationsbestand zugreifen. Aus der Sicht des Informationssystems lassen sich lesende *Anfragen* und zustandsverändernde *Anweisungen*, die von Agenten an das Informationssystem gerichtet werden, von den *aktiv* durch das Informationssystem selbst ausgelösten *Aktionen* unterscheiden.

In klassischen Informationssystemen werden „zusammengehörige" Anfragen und Anweisungen eines Agenten zu einer *Transaktion* zusammengefaßt, die atomar, konsistenzerhaltend und isoliert von anderen Transaktionen ausgeführt wird. Das klassische Transaktionsparadigma führt also zu einer engen Verzahnung der Aspekte der Fehlererholung mit den Aspekten der Synchronisation und Integritätssicherung.

Moderne Informationssysteme erfordern häufig jedoch komplexere *Konversationen* mit Agenten, dies sind Folgen von Anfragen, Rückfragen, Bestätigungen, Anweisungen und Aktionen, die nicht adäquat in isolierte Transaktionen zu zerlegen sind.

Das Konzept eines reaktiven Informationssystems geht deutlich über das einfache Paradigma der *stapelorientierten Verarbeitung* hinaus, bei der ein Prozeß sequentiell die drei Phasen Dateneingabe, Datenverarbeitung und Datenausgabe durchläuft. Dieses Konzept liegt immer noch vielen Softwarewerkzeugen wie Compilern, Programmgeneratoren, Satzsystemen (TeX), Simulatoren oder Statistikpaketen zugrunde. Aber auch Datenbanksysteme mit der Fähigkeit zur unmittelbaren Transaktionsverarbeitung (*Online Transaction Processing*, OLTP) können nur als Zwischenstufe auf dem Weg von der stapelorientierten zur voll reaktiven Bearbeitung persistenter Datenbestände betrachtet werden, da jede einzelne Transaktion wiederum dem rigiden sequentiellen Schema Dateneingabe, atomare Datenverarbeitung und Datenausgabe gehorchen muß.

Die Reaktivität von Informationssystemen erfordert den Einsatz algorithmisch vollständiger Sprachen zur *Verhaltensmodellierung*. Die Notwendigkeit der konzeptionellen und sprachlichen Integration der Verhaltensmodellierung mit der Datenmodellierung führt schließlich zu der Problematik der *Programmierspracheneinbettung* von Datenbankdiensten, die in Abschnitt 8 genauer behandelt wird.

1.3.4 Unterstützung der Integrität

Ein Informationssystem wahrt Integritätsbedingungen, die sich sowohl auf seinen persistenten Informationsbestand als auch auf die einkommenden und ausgehenden Informationen beziehen können. Dabei lassen sich zwei komplementäre Aspekte der Integritätssicherung unterscheiden:

Binnenwirkung: Aus der Sicht des Software-Engineering beschreiben und kontrollieren die Integritätsbedingungen eines Informationssystems die Konsistenz eines *gekapselten* langlebigen Systemzustands, bei dem es sich, wie in Abschnitt 1.3.1 angedeutet, um ein Systemmodell eines speziellen Realitätsausschnitts handelt.

Außenwirkung: Aus der Sicht der Agenten in der Umgebung des Informationssystems definieren und erzwingen die Integritätsbedingungen *Informationsstrukturen* und *Geschäftspraktiken*, die von allen informationsverarbeitenden Aktivitäten in der Umgebung des Informationssystems einzuhalten sind.

In diesem ersten Kapitel des Datenbankhandbuchs sind vor allem die Binnenwirkung von Integritätsbedingungen und die Sprachmechanismen zur Integritätsformulierung in verschiedenen Datenbankmodellen von Interesse. Die korrekte und flexible Modellierung von Integritätsbedingungen basierend auf ihrer Außenwirkung ist Gegenstand von Kapitel xx (Konzeptioneller Datenbankentwurf).

Beim Einsatz eines Datenbankmodells zur Realisierung eines Informationssystems lassen sich Integritätsbedingungen zunächst wie folgt klassifizieren:

Modellinhärente Integritätsbedingungen werden bereits implizit durch die Strukturen und Operationen des gewählten Datenbankmodells erzwungen und sind in diesem Sinne in jedem Datenbankzustand und bei jeder Zustandsänderung erfüllt Das einfachste Beispiel einer modellinhärenten Konsistenzbedingung ist die *Typisierung* eines Objektattributs. So läßt sich zum Beispiel im Datenbankschema definieren, daß das Attribut *Alter* eines Objekts der Klasse *Person* nur ganzzahlige Werte annehmen kann. Zuweisungen typinkompatibler Werte an das Attribut *Alter* werden durch das Datenbanksystem bei der Ausführung der Zuweisung (oder bereits bei ihrer Übersetzung) als fehlerhaft zurückgewiesen.

Applikationsspezifische Integritätsbedingungen schränken die Menge der zulässigen Datenbankzustände und -zustandsübergänge weiter ein und müssen im Datenbankschema explizit deklariert werden oder durch die informationsverarbeitenden Algorithmen explizit überprüft werden. Zum Beispiel kann die Bedingung, daß das Alter einer Person nur Werte zwischen 0 und 200 annehmen kann, und daß das Alter jeder Person über die Lebensdauer der Datenbank nur monoton wachsen kann, in keinem kommerziell relevanten Datenmodell direkt durch Datenmodellkonstrukte beschrieben werden.

Weiterhin ist eine Klassifikation von Integritätsbedingungen aufgrund ihrer zeitlichen Ausdehnung möglich:

Statische Integritätsbedingungen müssen in jedem Datenbankzustand S erfüllt sein, unabhängig von der Historie, die zu diesem Zustand führt. Sie lassen sich durch logische Formeln (quantifizierte *Prädikate*) über die Zustandsvariablen von S spezifizieren („Jeder Mitarbeiter arbeitet in genau einer Abteilung").

Dynamische Integritätsbedingungen müssen bei jeder Datenbankzustandsänderung von einen Zustand S_n in einen Zustand S_{n+1} erfüllt sein. Im einfachsten Fall läßt sich eine dynamische Integritätsbedingung durch einen *Prädikattransformer* spezifizieren, dies ist eine logische Formel, die ein Prädikat über S_{n+1} aus einem Prädikat über S_n ableitet („Gehälter von Mitarbeitern nehmen nie ab"). Im allgemeinen können dynamische Integritätsbedingungen sich jedoch auf die gesamte Historie beziehen, die zu einem Zustand S_n führt. In diesem Fall werden logische Formeln von *temporalen Logiken* zur Integritätsdefinition benutzt, die spezielle temporale Quantoren über Variablenhistorien anbieten („Ein Mitarbeiter kehrt niemals in eine Abteilung zurück, die er einmal verlassen hat").

Integritätsbedingungen lassen sich schließlich gemäß ihres Sichtbarkeits- oder Wirkungsbereichs klassifizieren, der sich auf ein einzelnes *Attribut,* ein einzelnes *Objekt*, eine einzelne *Klasse,* eine einzelne *Beziehung* zwischen zwei Klassen oder auf beliebig viele Attribute, Objekte oder Klassen simultan beziehen kann.

Während sich statische, lokale Integritätsbedingungen adäquat durch Konzepte der algorithmischen Programmierung (Attributtypisierung, Rekordtypisierung, Objektkapselung, Datentypabstraktion) unterstützen lassen, erfordern Informationssysteme zusätzlich die Kontrolle komplexer *nicht-lokaler Integritätsbedingungen*, die verschiedene Informationsstrukturen zueinander in Beziehung setzen.

Deklarative Integritätssicherung ist ein über klassische Typisierungskonzepte hinausgehender Dienst von modernen Datenbanksystemen. Dabei werden Integritätsbedingungen über spezielle Schlüsselworte, Klauseln oder quantifizierte boole' sche Prädikate im Datenbankschema definiert. Das Datenbanksystem garantiert die Einhaltung dieser Bedingungen, indem es integritätsverletzende Änderungsoperationen auf der Datenbank erkennt und (evtl. benutzerdefinierte) Schritte zur *Ausnahmebehandlung* (z.B. einen Transaktionsabbruch oder kaskadierte Löschoperationen) einleitet.

Eine deklarative Integritätsbedingung für eine Literaturdatenbank ist zum Beispiel die Forderung, daß zu jedem Element r der Massendatenstruktur *Report* genau ein Element der Massendatenstruktur Institution gehört, das r als Publikation nennt (referentielle Integrität, vgl. Abschnitt 4.5.4).

Prozedurale Integritätssicherung erfordert hingegen explizit programmierte Tests bei Datenbankzustandsänderungen. Zur Sicherung der referentiellen Integrität im obigen Beispiel sind explizite Tests beim Einfügen und Ändern jedes Objekts der Klasse *Report* notwendig sowie beim Einfügen, Löschen und Ändern jedes Objekts der Klasse *Institution.*

Eine deklarative Integritätssicherung besitzt folgende Vorteile:

- Verbesserte Systemwartbarkeit, da eine Integritätsbedingung nur an genau einer Stelle (im Datenbankschema) lokalisiert ist und der Applikationskode nicht verändert werden muß.

- Verbesserte Verstehbarkeit, da deklarative Integritätsbedingungen häufig einfacher als kaskadierte *if*-Anfragen zu lesen sind.

- Optimierbarkeit, da zum Beispiel ein Datenbanksystem bei der Kenntnis referentieller Integritätsbedingungen explizite Zugriffspfadunterstützung für die wiederholte Überprüfung von Integritätsbedingungen und die Verknüpfung (*join*) von Tabellen anbieten kann.

Beim gegenwärtigen Stand der Technik führen deklarative Integritätssicherungsmaßnahmen in komplexen Modellierungssituationen jedoch zu folgenden Problemen:

- Die Ausnahmebehandlungsoperationen verschiedener Integritätsbedingungen (kaskadiertes Löschen, Propagieren, Transaktionsabbruch) sind nur schwer zu synchronisieren und können zu nicht-deterministischem Systemverhalten führen.

- Eine deklarativ spezifizierte Ausnahmebehandlung kann zu kaskadierten Integritätsverletzungen führen, die wiederum Ausnahmebehandlungen auslösen. Eine Terminierung und Konsistenz von Ausnahmebehandlungsoperationen ist nicht garantiert.

- In einem großen Datenbankschema ist es schwierig, alle für eine gegebene Datenbankoperation relevanten Integritätsbedingungen zu lokalisieren.

- Die von Datenbanksystemen angebotenen Sprachmittel zur Formulierung von Ausnahmebehandlungsroutinen sind nicht adäquat (fehlende algorithmische Vollständigkeit).

Die Wahrung der Integrität eines Informationsbestandes erfordert auch die Kontrolle über die *Subjekte* (Agenten, vgl. Abschnitt 1.3.3), die an den informationsverarbeitenden Aktivitäten in der Umgebung des Informationssystems teilnehmen. Die in diesem Zusammenhang auftretenden Probleme der *Authentisierung* von Benutzern gegenüber dem Informationssystem (zum Beispiel durch Passworte) und der *Autorisierung*, also der Vergabe und Kontrolle von Zugriffsrechten von Subjekten auf Datenbankobjekte werden in Kapitel xx näher behandelt.

1.4 Datenbankdienste für moderne Informationssysteme

In diesem Abschnitt wird indirekt die bisher offengebliebene Frage „was ist ein Datenbanksystem" beantwortet, indem diejenigen Dienste aufzählt werden, die von Datenbanksystemen zur Realisierung von Informationssystemen angeboten werden. Detailliertere Erläuterungen zu jedem dieser Dienste finden sich in den vorangegangenen Unterabschnitten. Dabei werden Datenbankdienste in drei Kategorien klassifiziert:

1. *Verpflichtende Datenbankdienste:* Persistente Datenspeicherung, Verwaltung einer Speicherhierarchie mit effizienter Datenselektion, dynamische Massendatenstrukturen, Synchronisationsprimitive, Dienstschnittstellen für den Datenbankzugriff aus Programmiersprachen.

2. *Datenbankkerndienste:* Zusätzlich Persistenzabstraktion, Metadatenverwaltung, inkrementelle und dynamische Bindung, Datenunabhängigkeit durch Schemaschichtung, Iterationsabstraktion, transaktionale Integritätssicherung, Fehlererholung und Parallelitätskontrolle im Mehrbenutzerbetrieb.

3. *Erweiterte Datenbankdienste:* Zusätzlich orthogonale Persistenz, Persistenzunabhängigkeit, deklarative Integritätssicherung, Verhaltensmodellierung, Ausnahmebehandlung, Unterstützung für Konversationen und langandauernde Aktivitäten, Benutzeroberflächen, erweiterte Dienstschnittstellen (4GL, Benutzeroberfläche, Reportgenerator, entfernter Datenbankzugriff).

Abschließend sei bemerkt, daß während der letzten zehn Jahre ein erheblicher Zuwachs des Funktionsumfangs und der Ausdrucksmächtigkeit von Datenbanksprachen und -modellen zu beobachten ist, der aber gleichzeitig auch die Komplexität der Datenbanksystemsoftware deutlich erhöht hat. Analog zu der Entwicklung von CISC zu RISC Hardwarearchitekturen und der Entwicklung von *micro-kernels* in Betriebssystemen kann spekuliert werden, daß eventuell eine Reduktion der Funktionalität der eigentlichen Datenbanksoftware gekoppelt mit leistungsfähigen, problemorientierten Werkzeugen (Applikationsgeneratoren, Bibliotheken, Werkzeugkästen) zu effizienteren und flexibleren Informationssystemen führt, als sie mit monolithischen *general-purpose* Datenbanksystemen zu realisieren sind. Diese Hypothese wird durch das Aufkommen von vergleichsweise primitiven Objektspeichersystemen, Repositories, und Multi-Media-Servern gestützt.

1.5 Vorteile der Nutzung standardisierter Datenbankdienste

Der Einsatz von Datenbanksystemen bei der Realisierung von Informationssystemen führt zunächst zu den bekannten positiven Effekten, die allgemein beim Einsatz generischer Diensterbringer bei der Softwareentwicklung zu beobachten sind:

- Reduzierter Implementierungs- und Wartungsaufwand durch die *Wiederverwendung* kommerziell verfügbarer, vordefinierter und ausgereifter Systemfunktionalität.
- Erhöhte *Portabilität* der entwickelten Software durch die Benutzung standardisierter, plattformunabhängiger Dienste.
- Verbesserte *Systemskalierbarkeit* durch die Möglichkeit zum Austausch von Diensterbringern (Beispiel: Migration eines Informationssystems von einem Einbenutzerdatenbanksystem über ein Mehrbenutzerdatenbanksystem zu einem verteilten Datenbanksystem).

Darüber hinaus leisten Datenbanksysteme weitere wichtige Beiträge zur Qualität der mit ihnen entwickelten Informationssysteme:

Datenabstraktion (data abstraction): Die Verfügbarkeit von ausdrucksmächtigen Datenbankschemainformationen garantiert die Interpretierbarkeit des Datenbankzustandes unabhängig von seiner Repräsentation im Computer (Rekordlayout, Dateilayout, Adressierungsmechanismen, Datenformate, etc.) und unabhängig von speziellen Anwendungsprogrammen. Damit erhöhen Datenbanksysteme entscheidend die Nutzbarkeit des Datenbestandes für „unvorhergesehene" informationsbearbeitende Aktivitäten (*ad hoc queries*).

Gemeinsame Informationsnutzung (information sharing): Im Gegensatz zu applikationsspezifischen Speicherungssystemen erlauben Datenbanksysteme einen gemeinsamen Datenzugriff aus unabhängig entwickelten Anwendungsprogrammen, aus verschiedenen Programmiersprachen und durch parallel exekutierende Agenten.

Integrierte Datenbeschreibung (information integration) Die Evolution von Informationssystemen führt häufig zu der Notwendigkeit, bestehende Informationsbestände zu erweitern oder gar unabhängig entstandene Informationssysteme konsistenzerhaltend zu integrieren. Datenbanksysteme leisten bei diesem Vorgang eine wichtige Hilfestellung, indem sie es gestatten, verschiedene *Sichten* auf ein Datenobjekt zu definieren, und damit eine Replikation von Informationen in verschiedenen Speicherungssystemen zu vermeiden. Außerdem ermöglichen Datenbanksysteme die Definition von Konsistenzbedingungen, die sich über mehrere, ursprünglich unabhängige Objektklassen erstrecken.

Bereits an dieser Stelle sei auf Probleme hingewiesen, die sich durch das Herauslösen der Datenhaltungskomponente aus individuellen Anwendungsprogrammen ergeben können:

- *Redundanz* oder Inkonsistenz zwischen den Informationsstrukturbeschreibungen in der Programmiersprache, in dem Datenbankschema, in der Benutzerschnittstellendefinition und in den Kommunikationsprotokollbeschreibungen.
- *Ineffiziente Datenspeicherung* oder ineffizienter Datenzugriff aufgrund von Modellierungsdiskrepanzen zwischen den Informationsstrukturen des Informationssystems und Datenmodellierungskonzepten der Datenbank (Beispiel: Textdaten, CAD Daten).

Abschließend sei bemerkt, daß generische Diensterbringer wie Datenbanksysteme in der Literatur teilweise unter dem Begriff der *Middleware* zusammengefaßt werden, da sie architekturell und konzeptionell zwischen der anwendungsspezifischen Software und der fest vorgegebenen, plattformspezifischen Betriebssystemsoftware angesiedelt sind [Bernstein 93] (s.a. Abbildung 5).

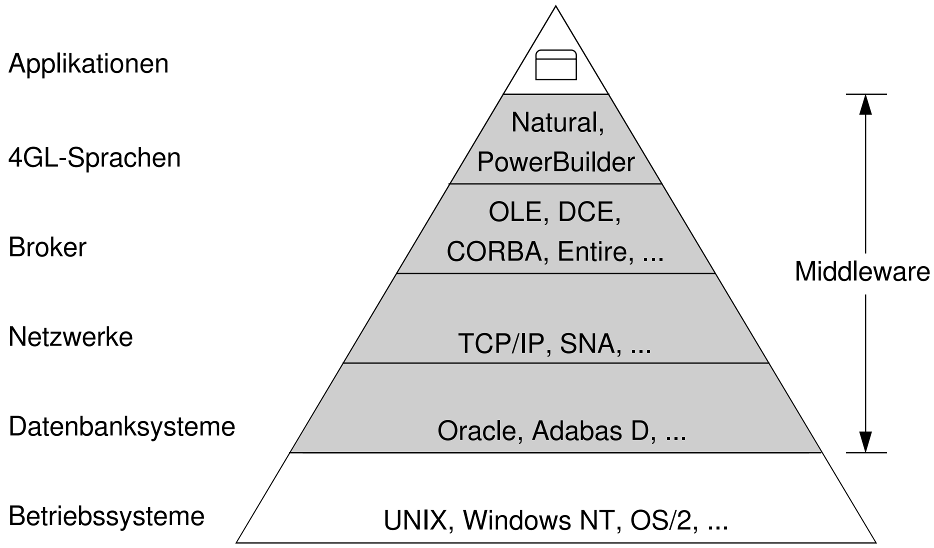

Abbildung 5: Datenbanksysteme als Teil der Middleware

1.6 Beispiel: Ein Firmeninformationssystem

Dieser Abschnitt illustriert die Charakteristika von Informationssystemen an dem konkreten Beispiel eines prototypischen *Firmeninformationssystems* (Abbildung 6). Dieses Beispiel wird durchgängig im gesamten ersten Kapitel verwendet.

1.6.1 Dienstschnittstellen

`Sicht des Systems von außen, Modalitäten des Zugriffs`
`Abbildung: X-11 GUI Schnittstelle, RPC Schnittstelle, C++ Algorithmen, relationales Datenbankschema, Oracle Datenbank`

1.6.2 Informationsstrukturen

Das System verwaltet Informationen über Projekte, Mitarbeiter und Abteilungen, deren reguläre Informationsstrukturen und Integritätsbedingungen sich wie folgt beschreiben lassen:

Jedes *Projekt* besitzt einen Projekttitel, eine eindeutige, maximal fünfstellige Projektnummer sowie ein Budget, das monatlich auf zwei Nachkommastellen genau verwaltet wird und nie überzogen werden kann.

Jeder *Mitarbeiter* besitzt einen Namen. Einige Mitarbeiter sind Werkstudenten, die auf einer Stundenbasis zu einem festen Tarif pro Stunde beschäftigt werden. Die übrigen Mitarbeiter sind Festangestellte und besitzen ein festes Monatsgehalt. Jeder Mitarbeiter kann einen festgelegten prozentualen Anteil seines monatlichen Gehalts aus dem Budget eines oder mehrerer Projekte erhalten. Das übrige Gehalt wird von der Abteilung gezahlt, die den Mitarbeiter beschäftigt. Jeder Mitarbeiter wird von genau einer Abteilung beschäftigt.

Jede *Abteilung* besitzt einen Namen, ein eindeutiges Abteilungskürzel, bestehend aus bis zu vier Buchstaben, und einen Abteilungsleiter, der auch als Mitarbeiter in dieser Abteilung beschäftigt ist. Eine Abteilung ist für die Durchführung der ihr zugeordneten Projekte ver-

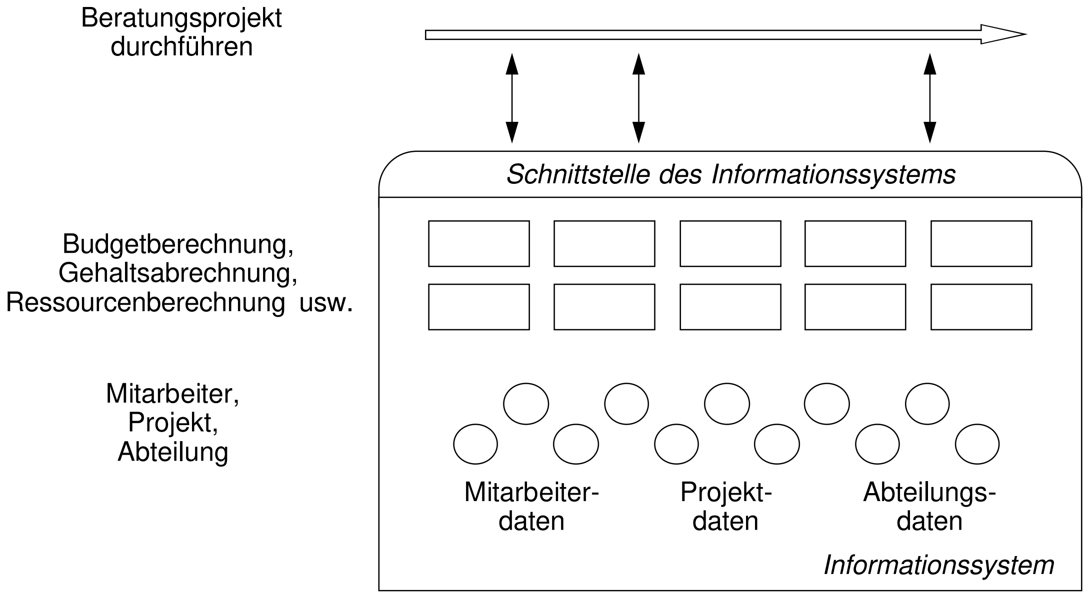

Abbildung 6: Ein Firmeninformationssystem

Abbildung 7: Dienstschnittstellen des Firmeninformationssystems

antwortlich. Die Mitarbeiter einer Abteilung dürfen ihr Gehalt ausschließlich von Projekten beziehen, für deren Durchführung die Abteilung auch verantwortlich ist. Für ein Projekt können mehrere Abteilungen verantwortlich sein. Für jedes Projekt ist mindestens eine Abteilung verantwortlich. Die Abteilungen sind in einer Hierarchie organisiert, da einige Abteilungen eine Oberabteilung besitzen, an die projektbezogene Informationen weitergeleitet werden müssen. Die Informationsweitergabe erfolgt über mehrere Hierarchieebenen hinweg.

Die Informationsstrukturen sind grafisch in OMT-Notation dargestellt. Diese Notation wird erst in Abschnitt 2.1 eingeführt, ist aber auch intuitiv verständlich.

Abbildung 8: Informationsstrukturen des Firmeninformationssystems

Ein Zustand des Informationsbestandes der Projektverwaltung, der die oben genannten Konsistenzbedingungen erfüllt, ist in tabellarischer Form in Abbildung 9 dargestellt.

`Am Schluß erstellen, sobald alle Daten und Attribute`
`klar, evtl Tabelle statt Bild ?`

1.6.3 Informationsverarbeitende Aktivitäten

Das System unterstützt die folgenden informationsverarbeitenden Aktivitäten:

`Wähle Beispiele basierend auf späteren SQL Beispielen im Text`

Abbildung 9: Ein zulässiger Zustand des Informationsbestandes der Projektverwaltung in tabellarischer Form

Elementare Transaktionen

Projekt: Akquisition (Titel, Nummer, Anfangsbudget, verantwortliche Abteilungen, Zuordnung Mitarbeiter), Budgetprognose für Folgemonate. Projektabschluß.

Mitarbeiter: Einstellung. Austritt. Monatsabrechnung. Ausscheiden aus Projekt. Übernahme Projektverantwortung. Abteilungswechsel zum Monatsende.

Abteilung: Gründung, Auflösung. Reorganisation in der Hierarchie. Leiterwechsel.

Anfragen: Anzeige einzelner Mitarbeiter, Projekte und Abteilungen (Attribute und assoziierte Objekte).

Identifikation von M, P, A durch Attribute & Beziehungen (Navigation).

Ausgabe von Listen (geordnet)

Bestimmung aller Projekte, die von Unterabteilung bearbeitet werden (Rekursive Beziehungen)

Konversationen: Geschäftsprozeß: Einstellung neuer Mitarbeiter: Personalabteilung, Anfrage bei Projektleiter, Gehaltsanpassung, Projektbudgetierung.

```
  Zeige Generalisierungen auf:

  Bilder als Attribute [Personaldaten], Handschriftenprobe, Trigger
zum Auslösen von Aktionen [Geburtstag, Budgetüberschreitung],
Projektdokumente im Rechnernetz, e-Mail Korrespondenz im Archiv.
Fax-Versendung, Telefonvermittlung, Digitale Bibliotheken, Multi-Media
Datenbanken, Volltextrecherche
```

2 Grundlagen der Datenbankmodellierung

Dieser Abschnitt führt grundlegende Konzepte ein, die unabhängig vom jeweiligen Datenmodell für die erfolgreiche Realisierung von Informationssystemen relevant sind. Dazu werden in Abschnitt 2.1 zunächst strukturbezogene Abstraktionskonzepte eingeführt und daran anschließend in Abschnitt 2.2 und 2.3 operationsbezogene Konzepte. Basierend auf den Abstraktionsniveaus wird in Abschnitt 2.4 eine Grobklassifikation der Datenmodelle vorgenommen. Sie wird ergänzt durch die Schilderung der Benutzergruppen eines Datenbanksystems und den damit verbundenen Modalitäten in Abschnitt 2.5. Ein historischer Abriß wichtiger Ereignisse in der Forschung der Datenbanktechnik beschließt dieses Kapitel.

2.1 Abstraktionsmechanismen zur Datenstrukturierung

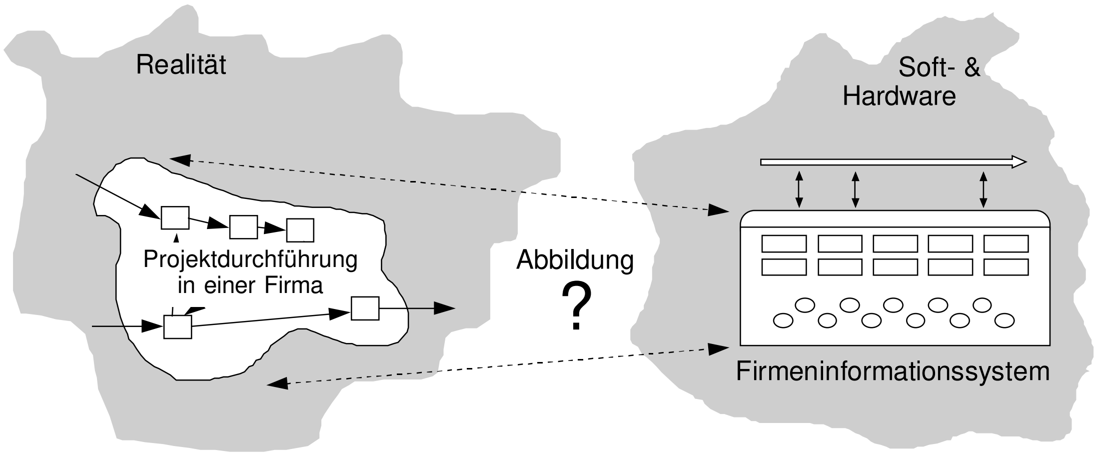

Abbildung 10: Abbildung eines Realitätsausschnitt auf ein Informationssystem

Unabhängig von der Vielfältigkeit der existierenden Datenmodelle gibt es grundlegende Konzepte und Abstraktionsmechanismen, die in der einen oder anderen Form in fast allen Datenmodellen eine zentrale Rolle spielen. Diese werden in diesem Abschnitt als grundlegendes Vokabular für die Vorstellung der klassischen Datenbankmodelle im nächsten Kapitel eingeführt. Gleichzeitig werden die Objektdiagramme nach OMT als eine mögliche graphische Notation für die Datenmodellierung vorgestellt, die bereits im letzten Abschnitt zur Illustration des Beispiels eingesetzt werden.

`Hier nur einfache, minimale Grundlagen. Ziel:`
`Vokabular und grafische`
`Notation Objektdiagramme nach OMT / Rumbaugh für Vorstellung`
`der klassichen Datenbankmodelle im nächsten Kapitel [Brodie, ...]`

In Datenbankmodellen lassen sich zwei grundlegende Konzepte zur Informationsrepräsentation unterscheiden:

Werte (Literale) besitzen eine Semantik unabhängig vom Zustand der Datenbank. Man unterscheidet dabei atomare *Basiswerte,* wie zum Beispiel Zahlen, Zeichen oder Zeichenketten und strukturierte, *zusammengesetzte Werte*. Zusammengesetzte Werte lassen

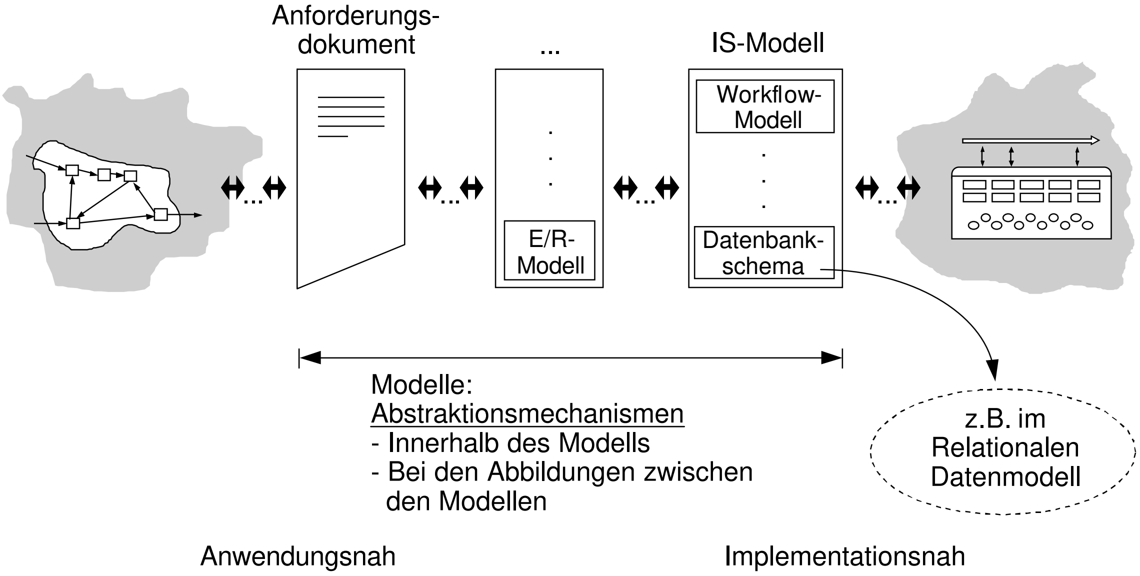

Abbildung 11: Modelle und Abstraktion

sich in *heterogene Strukturen* bestehend aus einer festen Anzahl benannter Attribute, wie z.B. Name, Alter und Telenfonnummer, (Tupel, Rekords, Strukturen) und *homogene Strukturen* (Kollektionen) bestehend aus einer flexiblen Anzahl anonymer Elemente (Arrays, Listen, Mengen, Multimengen) unterscheiden. Operationen auf Werten (Addition, Rekordfeldzugriff, Listenkonkatenation) besitzen *Kopiersemantik*. Als Faustregel gilt, daß sich Werte verlustfrei durch lineare textuelle Repräsentationen darstellen lassen.

Objekte besitzen eine Semantik, die vom Zustand der Datenbank abhängig ist. Der Objektbegriff in Datenbanken ist eng mit dem Begriff der *Zustandsvariablen* imperativer Programmiersprachen verwandt. Wie bei Variablen in imperativen Sprachen kann der Zustand eines Objektes durch destruktive *Zuweisungen* (auf Teilkomponenten oder das gesamte Objekt) über die Lebensdauer einer Datenbank hinweg verändert werden. Unabhängig von seinem Zustand behält ein Objekt jedoch seine *Objektidentität (object identity, OID)* Die Objektidentität eines Objektes kann als Wert in mehreren Datenstrukturen verwendet werden, um das Objekt über verschiedene Pfade zu *referenzieren*. Zum Beispiel kann die Objektidentität eines Objekts zur Beschreibung eines Kindes als Wert des Attributes Kind seiner beiden Eltern und als anonymes Element in einer Menge Schüler auftreten (*sharing*). Änderungen des Objektzustands, z.B. des Alters des Kindes, werden unmittelbar auf allen Pfade sichtbar, in denen das Objekt referenziert wird (Seiteneffeke, *aliasing*). Objekte besitzen also eine Referenzsemantik und eine textuelle Repräsentation von Objektstrukturen erfordert die Einführung von speziellen Notationen, um die (evtl. zyklische) Graphstruktur von Objektreferenzen linear zu repräsentieren.

Datenstrukturen bestehend aus Werten und Objekten werden zur Repräsentation von Informationsstrukturen verwendet, die ihrerseits durch die wiederholte Anwendung von *Ab-*

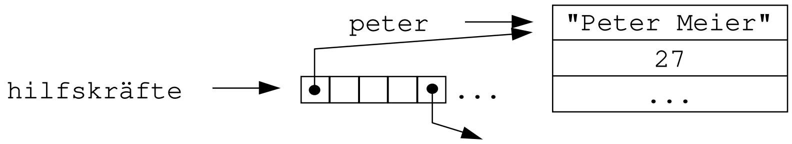

Abbildung 12: Referentielle Identifikation

straktionsmechanismen sogenannter *semantischer Datenmodelle* beschrieben werden können (s. Kapitel xx).

`Formulieren:` Als graphische Notation zur Darstellung von... werden in diesem Kapitel E/R-Diagramme verwendet, Zusammen mit der Einführung der einzelnen Konzepte wird auch deren grafische Dartsaellung in OMT-Notation beschrieben.

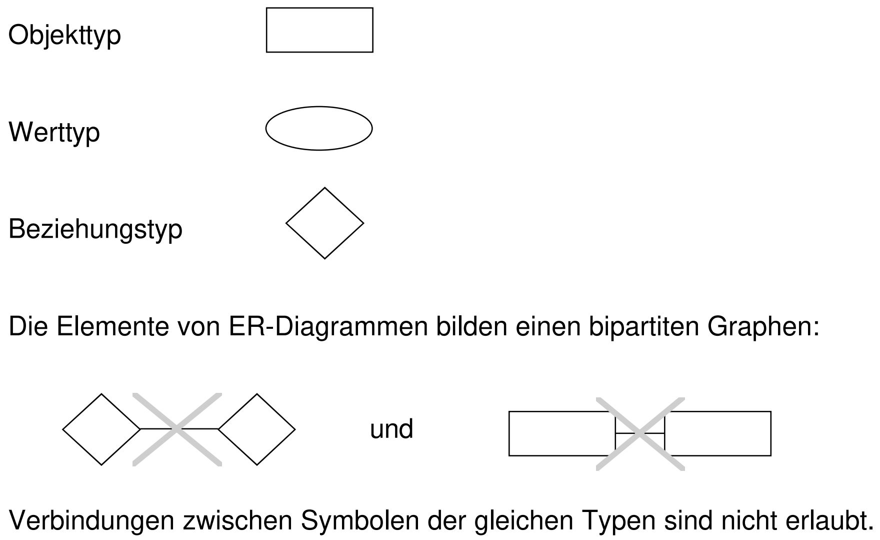

Abbildung 13: Grundlegende Elemente von E/R-Diagrammen

2.1.1 Klassifikation und Instanziierung

Bei der *Klassifikation* werden Objekte mit ähnlichen Eigenschaften zu einer *Klasse* zusammengefaßt. Jedes Objekt wird als *Instanz* seiner Klasse bezeichnet. Instanzen einer Klasse verfügen über die gleiche Struktur, ????

Ein weiterer Aspekt einer Klasse ist die Verwaltung ihres *dynamischen Extension*, die alle Objekte der Klasse umfaßt. (Literale, Domänen, strukturelle Klassifikation & Typen, Klasse, Instanz, statische Typisierung vs. dynamische Extensionsverwaltung, Iteration)

In der E/R-Notation werden Klassen durch Rechtecke dargestellt, Instanzen durch Rechtecke mit abgerundeten Ecken.

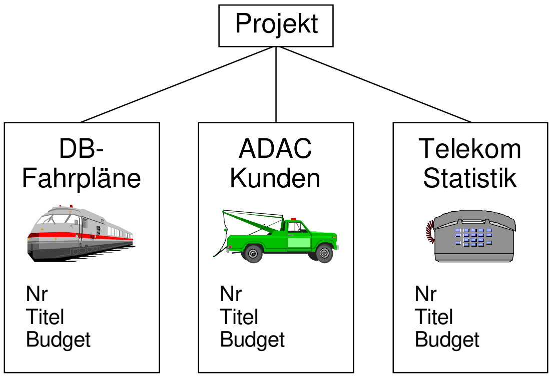

Abbildung 14: Klassifikation und Instantiierung

2.1.2 Aggregation und Dekomposition

Eine Instanz besitzt in der Regel mehrere Eigenschaften. Solche Eigenschaften werden duch Attribute beschrieben, die aus einem *Attributwert* und einem *Attributnamen* bestehen. Diese Attribute einer Instanz werden zu einem *Tupel* oder *Record* zusammengefaßt (*aggregiert*). Auf der Ebene der Klassen besteht diese Aggregation aus der Zusammenfassung von Paaren aus Attributnamen und zugehörigen Typ des Attributs (Domäne).

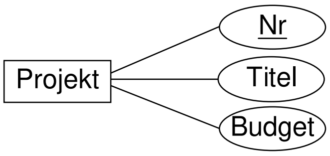

Abbildung 15: Attribute des Objekttyps *Projekt*

Durch *Dekomposition* kann über die Attributnamen auf die einzelnen Attributwerte einer Instanz zugegriffen werden. Sie ist vergleichbar der Selektion von Record-Komponenten in einer Programmiersprache.

Aggregation kann sich aber auch auf die Zusammensetzung von Objekten zu einem übergeordneten Objekt beziehen(vgl.[**?**]), wobei eine *part of*-Beziehung zwischen den Komponenten und dem aggregierten Objekt besteht. Das aggregierte Objekt stellt eine Beziehung zwischen Objekten dar, hat aber den Vorteil, daß man ihm Eigenschaften zuordnen kann und daß es auch selbst wieder an Beziehungen teilnehmen kann. Ein typisches Beispiel sind Stücklisten.

Die strukturellen Komponenten einer Klasse werden in der OMT-Notation durch Auflistung der Attributnamen und der zugehörigen Domänen dargestellt. Die Bildung von aggregierten Objekten wird durch spezielle Beziehungen dargestellt.

(statische Aggregation, Attributname, Attributwert, Records, Tupel, Werte vs. Typen)

2.1.3 Generalisierung und Spezialisierung

Bei der *Spezialisierung* wird eine oder mehrere Klassen definiert, die die ursprüngliche Klasse verfeinern. Die speziellere Klasse wird als *Subklasse*, die allgemeinere als *Superklasse* bezeichnet. Die Klassen bilden eine *Subklassenhierarchie*.Der umgekehrte Vorgang zur Spezialisierung ist die *Generalisierung* bei der gemeinsame Strukturen und Verhalten mehrer Klassen herausfaktorisiert und in einer Superklassen gesammelt werden. Spezialisierung und Generalisierung betonen die Gemeinsamkeiten von Klassen, ohne jedoch die Differenzen zu verwischen.

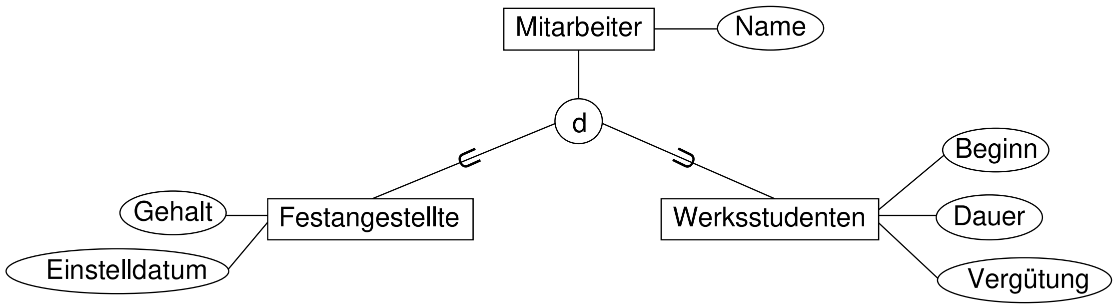

Abbildung 16: Generalisierung und Spezialisirung

Jede Instanz einer Klasse ist auch Instanz all ihrer Superklassen. Eine Spezialisierung kann disjunkt oder überlappend sein, wobei im ersten Fall jedes Objekt einer Klasse nur in höchstens einer ihrer direkten Subklassen enthalten sein darf.

Eine Subklasse *erbt* die Eigenschaften (Attribute und Methoden) ihrer Superklasse und fügt evtl. neue hinzu. An allen Stellen, an denen ein Objekt einer Klasse erwartet wird, wie z.B. bei Operationen kann auch eine beiebige Instanz einer ihrer Subklassen verwendet werden.

`Subtypisierung, Typhierarchie, Substituierbarkeit, Subsumption)`?????

2.1.4 Assoziation und Identifikation

Objekte können miteinander in Beziehung gesetzt -*assoziiert*- werden `Beispiel`. Zwischen zwei Klassen besteht eine Beziehung, wenn deren Instanzen miteinander assoziiert sind. *Binäre* Beziehungen stellen eine Verknüpfung zwischen zwei Klassen bzw. Objekten dar, *ternäre* Beziehungen entsprechend zwischen drei Klassen. Es gibt auch Beziehungen mit noch mehr beteiligten Klassen. Man spricht dann allgemein von n-ären Beziehungen. n wird dabei als *Grad* der Beziehung bezeichnet. In der OMT-Notation werden binäre Beziehungen durch Linien dargestellt, und Beziehungen höheren Grades durch Rauten.

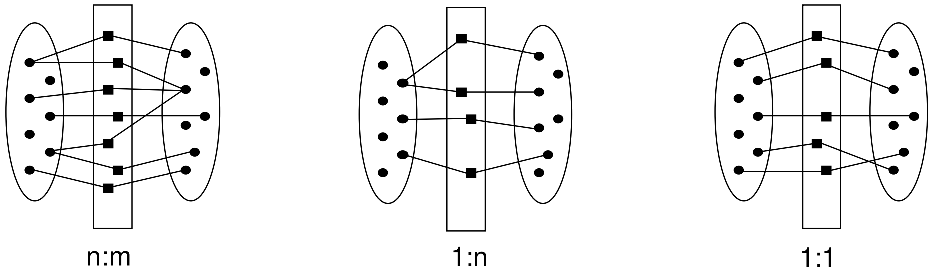

Abbildung 17: Binäre Beziehungen

Im allgemeinsten Fall kann jedes Objekt einer Klasse mit beliebig vielen Objekten der zweiten Klasse in Beziehung stehen und umgekehrt. Man spricht dann von einer n:m-Beziehung. *Kardinalitätsbeschränkungen* dienen dazu genaue Werte oder zulässige Intervalle für die Zahl der Instanzen einer Klasse, die mit einer Instanz der anderen Klasse in Beziehung stehen können, festzulegen. Weiterhin kann zwischen *partiellen* und *totalen* Beziehungen unterschieden werden. Bei einer totalen Beziehung muß jeder Instanz der ersten Klasse mindestens eine Instanz der zweiten Klasse zugeordnet sein, bei einer partiellen Beziehung ist die Zuordnung optional.

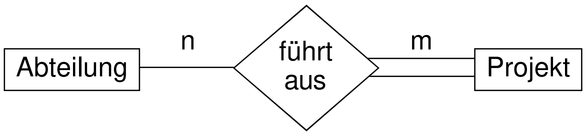

Abbildung 18: n:m-Beziehung zwischen Abteilungen und Projekten

Wenn man die Assoziation zwischen zwei Klassen als Linie betrachtet, so bildet jedes Ende eine *Rolle*, der ein eindeutiger *Rollenname* zugeordnet werden kann. Rollennamen schaffen die Möglichkeit, sich die Menge der Instanzen einer Klasse zu *identifizieren*, die mit einer Instanz der Klasse am entgegengestztzten Ende der Linie assoziiert ist, ohne explizit auf die Beziehung zuzugreifen. Aus der Sicht des Objekts bildet eine Rolle ein abgeleitetes Attribut mit dem Rollennamen als Attributnamen. Der Wert des Attributs ist die Menge der Objekte, die durch die betrachtete Beziehung mit dem Objekt assoziiert sind.

`Beispiel mit Bild`

2.1.5 Identifikation und Schlüssel

Rolle der Identifikation

Die Identifikation einzelner Objekte in den umfangreichen Extensionen einer Datenbank spielt eine zentrale Rolle. Grundsätzlich kann man zwischen zwei Arten von Identifikation unterscheiden. Bei der *referntiellen Identifikation* gibt es zur Identifikation direkte Verweise auf die einzelnen Objekte, die mit Zeigern in der Programmiersprachenwelt vergleichbar sind. Die *assoziative Identifikation* verwendet Werte der Attribute oder von Attributkombination, um sich eindeutig auf ein Objekt zu beziehen.

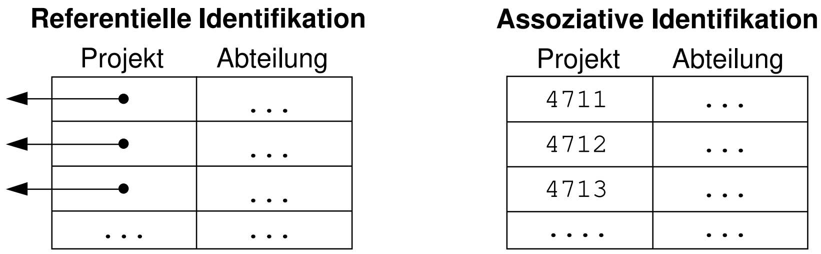

Abbildung 19: Referentielle Identifikation vs. Assoziative Identifikation

Ein Schlüssel ist ein Attribut oder eine Attributkombinationen mit der Eigenschaft, daß keine zwei Instanzen die gleiche Wertkombination in diesen Attributen besitzt. Schlüssel können deshalb zur eindeutigen Identifikation von Instanzen eingestzt werden. Es kann mehrere verschiedene Schlüsselkandidaten geben, von denen einer als *Primärschlüssel* zur Identifikation der Instanzen ausgewählt wird.

Bei der *referentielllen Integrität* geht es grundsätzlich darum, daß ein referenziertes Objekt auch tatsächlch existiert. Zur Darstellung einer Beziehung zu einem Objekt, kann ein Schlüssel als Attributwert verwendet werden (Relationales Modell). Man spricht dann von einem *Fremdschlüssel*. verwendet werden. Dabei ist sicherzustellen, daß das durch den Fremdschlüssel identifizierte Objekt auch tatsächlich existiert (*referentielle Integrität*).

`Erkläre: Orthogonalität und Typvollständigkeit`

2.2 Persistenzabstraktion

Moderne Datenbanksprachen bieten direkte Mechanismen zur Benennung und Manipulation langlebiger Daten und erfordern keine expliziten Lade- und Speicherungsoperationen beim Zugriff auf benannte Datenbankvariablen. Sie erlauben dem Datenbankprogrammierer damit eine *Persistenzabstraktion*. Somit werden die komplexen Details des Datentransfers zwischen Sekundär- und Primärspeicher aus den Algorithmen des Informationssystems eliminiert, und es wird dem Datenbanksystem ein erheblicher Optimierungsspielraum bei der Organisation von Speicherzugriffen gegeben.

In Sprachen, in denen sowohl langlebige als auch kurzlebige Daten bearbeitet werden können, läßt sich die Qualität der Persistenzabstraktion genauer charakterisieren: Ist jede Datenstruktur, die für transiente Daten zur Verfügung steht, auch für persistente Daten verfügbar (Beispiele: Rekords, Vektoren, Rekords mit Varianten, Listen, Zeiger) und umgekehrt jede Datenstruktur für persistente Daten auch für transiente Daten verfügbar (Beispiele: sequentielle Datei, relationale Tabelle), so handelt es sich um eine Sprache mit *orthogonaler Persistenz*. In solchen Sprachen ist der Anwendungsprogrammierer von der Aufgabe befreit, Datenstrukturkonvertierungen beim Übergang zwischen persistenter und transienter Speicherung durchzuführen.

Eine Sprache bietet darüber hinaus *Persistenzunabhängigkeit*, wenn es möglich ist, Algorithmen z.b.— in Form von Prozeduren, Funktionen und Anfragen zu formulieren, die uniform auf transiente und persistente Daten als Argumente angewendet werden können.

In Datenbanksprachen lassen sich drei Ansätze zur *Definition der Datenlebensdauer* unterscheiden.

Persistente Typen: Die Lebensdauer eines Datums wird explizit zusammen mit seiner Struktur definiert. So unterscheiden viele Sprachen zwischen persistenten Datenbanktabellen und transienten Ergebnistabellen. Dieses Modell ist inkompatibel mit dem Konzept der Persistenzunabhängigkeit.

Persistente Sichtbarkeitsbereiche: Die Lebensdauer eines Datums wird implizit aus dem statischen Kontext abgeleitet, in dem das Datum deklariert wird. Eine Variable, die lokal in einer Anfrage definiert wird, ist in diesem Modell ein transientes Datum, während eine global in einem Datenbankschema definierte Variable als persistent betrachtet wird.

Transitive Erreichbarkeit: Dieses Modell ist eine Verfeinerung des Modells der persistenten Sichtbarkeitsbereiche. In diesem Modell gibt es ebenfalls statische persistente Sichtbarkeitsbereiche, die auch als Wurzeln der Persistenz (*roots of persistence*) bezeichnet

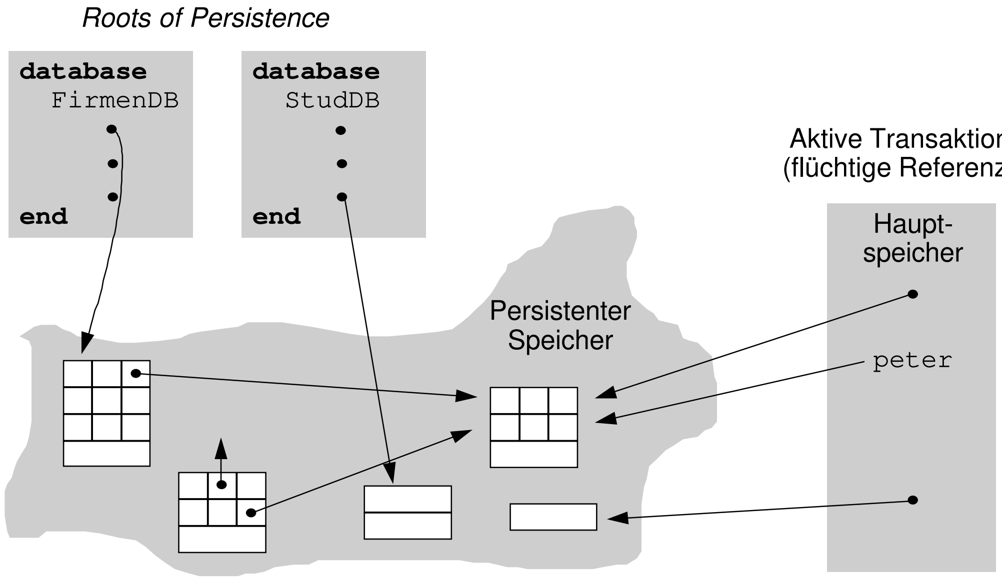

Abbildung 20: Alternativen zur Definition der Datenlebensdauer

werden. Dabei handelt es sich zum Beispiel um die Datenbankschemata eines Datenbanksystems. Zusätzlich werden bei der Lebensdauerdefinition auch (dynamische) Abhängigkeiten zwischen Daten berücksichtigt. Solche Abhängigkeiten entstehen, wenn die Semantik eines Datums A von der Semantik eines Datums B abhängt. Beispiele für solche Abhängigkeiten sind:

- A besitzt B als Rekordfeld oder Mengenelement.
- A ist ein Zeiger auf B.
- A ist ein Fremdschlüssel für B.
- A ist eine Subklasse von B.
- A ist eine Funktion oder Anfrage, die B benutzt.

Zur Sicherstellung der referentiellen Integrität (s. Abschnitt 4.5.4) dieser Abhängigkeiten impliziert in diesem Modell die Persistenz des Datums A automatisch die Persistenz des Datums B. Insgesamt sind also genau die Daten persistent, die transitiv von den Wurzeln der Persistenz aus erreichbar sind.

Ein erheblicher Vorteil des Modells der transitiven Erreichbarkeit ist die Möglichkeit, eine komplexe, transient erzeugte Datenstruktur durch eine einzelne Zuweisung dynamisch von einer persistenten Datenstruktur aus erreichbar zu machen. Außerdem werden subtile Speicherlecks (*memory leaks*) vermieden, die aufgrund fehlender Speicherfreigabeanweisungen für nicht mehr erreichbare persistente Daten in Informationssystemen entstehen.

Das Persistenzkonzept der transitiven Erreichbarkeit kann als Generalisierung des Konzepts der automatischen Freispeicherverwaltung (*garbage collection*) angesehen werden, die in zunehmendem Maße in Programmiersprachen eingesetzt wird. Ein Programm bildet dabei einen transienten Sichtbarkeitsbereich, und seine lokalen Variablen bilden Wurzeln der Erreichbarkeit. Insgesamt lassen sich während der Programmlaufzeit drei Klassen von Daten unterscheiden: Persistente Daten sind von den Wurzeln der Persistenz erreichbar, transiente Daten sind von lokalen Variablen aus erreichbar, tote Daten sind unerreichbar, und der von ihnen belegte Speicherplatz wird automatisch durch das System freigegeben.

2.3 Datenunabhängigkeit durch Schemaarchitekturen

Der bereits in Abschnitt 1.3.1 kurz angesprochene Begriff der Datenunabhängigkeit läßt sich besonders gut anhand der sog. Drei-Schichten-Architektur erläutern. Diese Architektur wurde vom ANSI/SPARC-Kommittee unter anderm mit dem expliziten Ziel der Unabhängigkeit der Daten von Anwendungen und der Unterstützung mehrerer Benutzersichten entworfen.

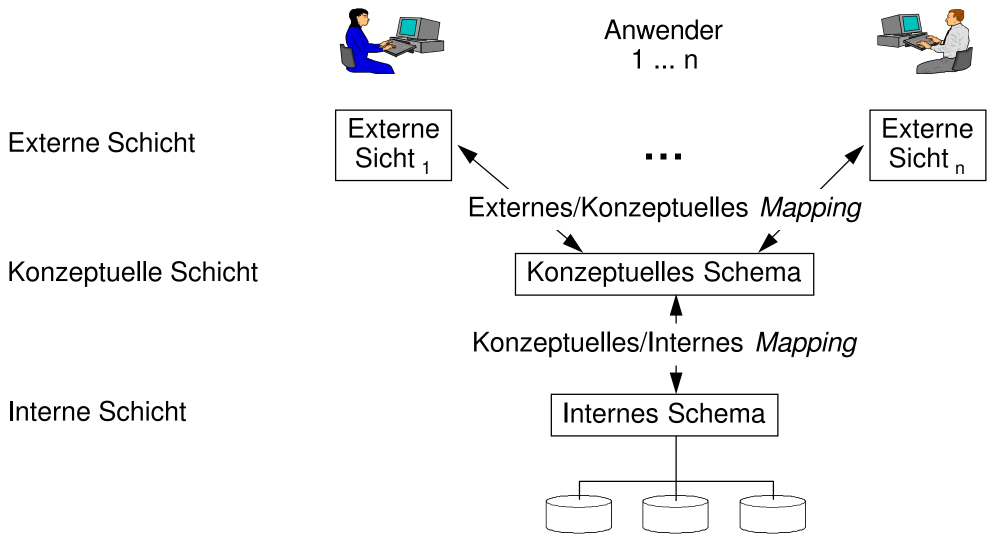

Abbildung 21: Drei-Schichten-Architektur

Wie der Name bereits andeutet, können bei dieser Architektur in drei Schichten verschiedene Schemata definiert werden. Alle Schemata sind im DB-Katalog abgelegt.

1. Die *interne Schicht* besitzt ein *internes Schema*, das die physikalischen Speicherstrukturen der Datenbank beschreibt, wozu die Einzelheiten der Datenspeicherung und der Zugriffspfadunterstützung gehören.

2. In der *konzeptuellen Schicht* legt das *konzeptuelle Schema* die Strukturen der konzeptuellen Sicht der gesamten Datenbank für eine ganze Benutzergemeinde fest. Dabei liegt die Betonung auf den Entitäten, Datetypen, Beziehungen und Integritätsbedingungen, während die physikalischen Speicherstrukturen verborgen werden.

3. Aufgabe der für informationsverarbeitende Aktivitäten sichtbare *externen Schicht* ist die Unterstützung von Benutzersichten. Jedes *externe Schema*, auch Benutzersicht genannt, beschreibt die Sicht eines Benutzers oder einer Benutzergruppe auf die Daten. Es sind nur die für diesen Benutzer relevanten Daten sichtbar, der übrige Teil wird ausgeblendet.

Es lassen sich zwei Arten von Datenunabhängigkeit unterscheiden, die mit Bezug auf die unterschiedlichen Schemata der Drei-Shema-Architektur erläutert werden.

Logischer Datenunabhängigkeit: Das konzeptuelle Schema kann weitgehend ohne Konsequenzen für das externe Schema geändert werden. So soll z.B. nicht nur das Einfügen neuer Datensätze, sondern auch strukturelle Erweiterungen des konzeptuellen Schemas ohne Einfluß auf das externe Schema sein. Beim Löschen von Datensätzen und auch von Strukturen sollen zumindest die externen Schemata, die sich nur auf den verbleibenden Teil der Datenbank beziehen, unbeeinflußt bleiben

Physischer Datenunabhängigkeit: Das interne Schema kann unabhängig von dem konzeptuellen Schema modifiziert werden. Mögliche Modifikationen des internen Schemas sind z.B. die Reorganisation der Daten und die Einrichtung neuer Zugriffspfade. Solange die Datenbank weiterhin dieselben Nutzdaten enthält, sollten solche Veränderungen keine Auswirkungen auf das konzeptuelle Schema besitzen.

Wie bereits an der Drei-Schema-Architektur gesehen, erreicht man Datenunabhängigkeit bei Datenbankmodellen unter anderem durch *Schemaschichtung*. Anfragen und Operationen der informationsverarbeitenden Aktivitäten werden gegen das externe Schema gestellt, während die Daten selbst in der internen Schicht angesiedelt sind. Der notwendige Abbildungsprozeß zwischen den Schichten ist Aufgabe des Datenbanksystems. Er birgt zwar Optionen für die Optimierung ist aber auch mit einem erheblichen Aufwand behaftet und wird daher in manchen Systemen umgangen.

2.4 Grobklassifikation von Datenmodellen

Es gibt viele verschiedene Datenmodelle. Sie lassen sich z.B. klassifizieren nach dem Abstraktionsniveau der Konzepten die sie zur Datenbeschreibung anbieten:

Konzeptuelle Datenmodelle (*high-Level*, *semantische* Datenmodelle, Informationsmodelle) orientieren sich an einer dem Benutzers vertrauten Sicht der Informationsmodellierung. Die verwendeten Konzepte sind Objekte, Entitäten, Attribute, Beziehungen usw.

Implementationsmodelle sehen Konzepte vor, die vom Endbenutzer verstanden werden können, aber trotzdem nicht zu weit weg von der Datenrepräsentation im Rechner sind. Auf diese Weise ist sichergestellt, daß die Konzepte in direkter Weise realisiert werden können. Zu diesen Modellen zählen die in Kapitel 3 vorgestellten relationales Datenmodell, Netzwerkdatenmodell und hierarchisches Datenmodell. Da sie Informationen auch in Rekords speichern werden sie auch als *rekord-basierte* Datenmodell bezeichnet.

low-level (physische) Datenmodelle bieten Konzepte auf einem sehr niedrigen Abstraktionsniveau an. Diese nehmen Bezug auf Details der Datenspeicherung und der Implementierung des Datenbanksystems. Der Einsatz dieser Datenmodelle erfordert einschlägige Erfahrungen im Bereich der

Datenbankmodelle Zugriffspfade Unterscheidung nach den Entitäten die, die Basisstrukturen des Datenmodells bilden:????

Objektbasierte Datenmodelle beschreiben hauptsächlich Objekte und die Beziehungen zwischen diesen Objekten.

Rekordbasierte Datenmodelle verwenden Recordstrukturen für die Repräsentation der Daten.

Klassifikation: Wertbasierte (relationale, funktionale) und objektbasierte (hierarchische, netzwerkartige, objektorientierte) Datenmodelle. Zustandsbegriff jedoch in allen Datenbanken (auf top level). Vorwärtsverweis auf Objektdiskussion in ODMG Kapitel.

Informationsspeicherung auf verschiedenen Abstraktionsstufen mit unterschiedlichen Operationen möglich: [s. Cattell]

Datenbank ...

Objektbank ...

Wissensbank ...

`Information Base: Volltextinfo, Rohdaten`

2.5 Benutzergruppen und Modalitäten

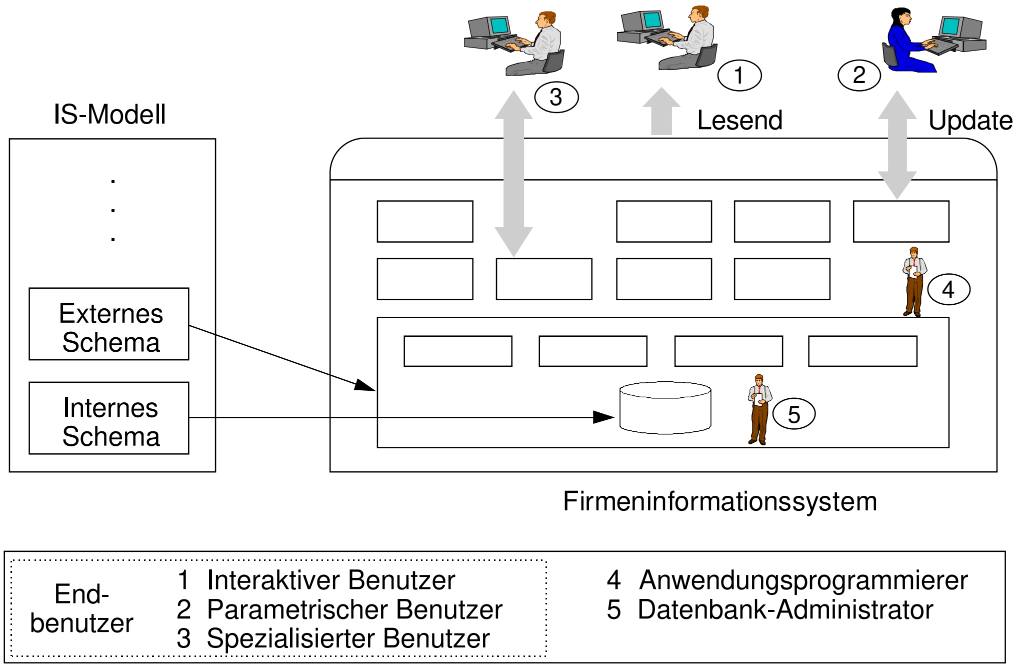

Abbildung 22: Benutzergruppen und Modalitäten

Der *Datenbankadministrator(DBA)* ist für die Kontrolle der internen und der konzeptuellen Schicht der Datenbank verantwortlich. In seinen Aufgabebereich fallen die Definition des Datenbankschemas (Konzeptuelles Schema), die Spezifikation von Integritätsbedingungen, der Entwurf der Speicherungstrukturen und Zugriffspfade (Internes Schema), Modifikationen des Schemas und der physischen Datenorganisation und die Vergabe von Zugriffsrechten an Benutzer.

Der *Anwendungsprogrammierer* gestaltet die externe Schicht der Datenbank verantwortlich. Er definiert die externen Schemata und realisiert Anwendungsprogramme mit eingebetteten Aufrufen an die Datenbank. Diese stehen dem Endbenutzer als vordefinierte Transaktionen zur Verfügung.

Die *Endbenutzer* lassen sich nach der Art der Benutzung und ihrer Kenntnis der Datenbank in mehrere Gruppen unterteilen.

- Der *interaktiver Benutzer* zeichnet sich typischerweise dadurch aus, daß er die DB nur gelegentliche benutzt und meist keine Standardanfragen stellt. Er gehört in der Regel zum Management und schreibt keine Programme, sondern verwendet eine Datenbankanfragesprache.
- Der *parametrische Benutzer* verwendet vordefinierte Standardtransaktionen, die er nur noch mit den geeigneten Parametern versieht. Beipiele dafür sind Buchungen im Reisebüro und Überweisungen in der Bank. Diese Art der Benutzung der Datenbank erfordert keine informatische Ausbildung.
- *Spezialisierte Benutzer* haben komplexe Anforderungen an die Datenbanken, die über den Rahmen der vordefinierten Transaktionen hinausgehen. Sie schreiben Programme für ihre eigenen speziellen Bedürfnisse, was eine gründliche Einarbeitung in die Eigenschaften und Schnittstellen des Datenbanksystems voraussetzt.

Eng verknüpft mit den verschiedenen Arten von Endbenutzern sind die verschiedenen Modalitäten der Benutzung einer Datenbank.

2.6 Datenbanksysteme und ihre Modelle: Ein historischer Abriß

Die Entwicklung von Datenbankmodellen und -sprachen ist im Fluß und starken Einfüssen durch kommerzielle Datenbanksystemprodukte und Standardisierungsbemühungen unterworfen. Dieser Abschnitt gibt einen Überblick über die historische Evolution von Datenbanksystemen und ihren Modellen, soweit sie zum Verständnis der heute in der Praxis und Forschung relevanten Produkte und Konzepte notwendig erscheint. Die Darstellung lehnt sich an [Catell xx, Brodie xx,] und [Navathe xx] an.

2.6.1 Dateisysteme

Die frühen Betriebssysteme für kommerzielle Anwendungen boten bereits die Basisdienste und Modellprimitive, die sich auch in den modernen Datenbanksystemen wiederfinden:

1. Rekords fester Größe aggregieren Attribute verschiedener Datentypen.

2. Dateien fassen gleichartige Rekords zusammen, können dynamisch wachsen und schrumpfen, in ihrem Umfang die Kapazität des Hauptspeicher des Computers überschreiten, und langlebig gespeichert werden.

3.Zugriffsstrukturen, wie Hashtabellen und später ISAM-Dateien ermöglichen es, Rekords basierend auf ihren Attributwerten effizient innerhalb einer Datei zu selektieren.

4.Explizite oder implizite Datei- und Rekordsperren werden eingesetzt, um parallele Zugriffe zu kontrollieren.

2.6.2 Frühe Datenbanksysteme

In den 60er und frühen 70er Jahren entstanden die ersten vollständigen Datenbanksysteme, die das hierarchische und das Netzwerkmodell unterstützen und die Funktionalität von Dateisystemen um weitere Dienste und Abstraktionen erweitern:

5.Rekordidentifikatoren, Adressen oder Verweisattribute ermöglichen es, Rekords für den effizienten direkten Zugriff zu verketten und hierarchische oder netzwerkartige Datenbankstrukturen zu bilden.

6.Mehrere Indexstrukturen können gleichzeitig geöffnet sein und werden vom Datenbanksystem konsistent als eine gemeinsame Datenbank mit Verweisinformationen gewartet.

7.Zugriffskontrollmechanismen beschränken Rekordzugriffe auf authorisierte Agenten (Benutzer und Programme).

8.Es existiert eine Trennung zwischen logischem und physischen Schema.

9. Transaktionen bieten automatische Fehlererholung, Verklemmungserkennung und Integrität beim Mehrbenutzerzugriff .

Nachfolgend sind wichtige Meilensteine dieser Entwicklung aufgeführt:

1959 Die *Conference on Data Systems and Languages* gründet zwei Komitees zur Entwicklung einer Sprache für kommerzielle Anwendungen. Das „kurzfristige“ Komitee entwickelt *COBOL*, wohingegen das „langfirstige“ Komitee unter dem Titel *List Processing Task Group* (später *Database Task Group*) Datenbanksprach- und modellkonzepte untersucht.

1962 Das erste Datenbanksystem *Integrated Data Store* (I-D-S) von *General Electric* wird vertrieben. Spätere Systemversionen werden schrittweise um Rücksetzoperationen, Fehlererholungsmechanismen und Sicherungspunkte (1964), Online Transaktionsverarbeitung (OLTP, 1966) sowie Parallelitätskontrolle und Verklemmungserkennung (1968) erweitert.

1961 Bachmann, der Architekt von I-D-S, stellt Datenstrukturdiagramme basierend auf sogenannten *record types* und *set types* vor. Diese Terminologie ist prägend für die *CODASYL/DBTG* Standardisierung.

1968 Das Produkt *IMS/360 Version 1* basierend auf dem Hierarchischen Datenmodell wird von IBM für das MVS Betriebssystem ausgeliefert. Das Produkt *IMS DB/DC (database/data communication)* entwickelt sich schnell zur Standardplattform zur Realisierung von Informationssystemen. 1994 existieren ca. 8000 IMS Installationen weltweit.

196?-7? xx Partielle Implementierungen des CODASYL/DBTG Datenmodells werden als Produkte vermarktet.

1971 Veröffentlichung des *CODASYL/DBTG Report*, der das Netzwerkdatenmodell und den Dienstzugang zu Netzwerkdatenbanken beschreibt.

1973 Bachmann erhält den *ACM Turing Award*.

2.6.3 Relationale Datenbanksysteme

In den späten 70er und frühen 80er Jahren wurden relationale Datenbanksysteme entwickelt und erreichten den Status kommerzieller Produkte, deren Dienste sich wie folgt von den frühen Datenbanksystemen unterscheiden:

10.Hochsprachliche, deklarative Anfragesprachen ermöglichen eine abstrakte, mengenorientierte Datenmodellierung und -manipulation unabhängig von der Verfügbarkeit gespeicherter Verweisstrukturen.

11.Zusätzliche anwendungsorientierte Dienstschnittstellen unterstützen interaktive (ad-hoc) Anfragen, Formular- und Reportdefinitionen, sowie die integrierte Beschreibung von Algorithmen und Anfragen in Sprachen der Vierten Generation (4GLs).

12.Es existiert eine Trennung zwischen konzeptuellem und logischen Schema.

Wie die folgende historische Übersicht zeigt, ist die Entwicklung relationaler Systeme eng mit der Entwicklung von Datenbanken als einer akademischen Disziplin gekoppelt:

Mit Kalenderdaten vereinheitlichen:

Im Juni 1970 veröffentlichte die *Association for Computing Machinery* einen Text von Dr. E.F. Codd, damals Mitarbeiter am IBM Forschungslabor in San Jose, Kalifornien. Der Text mit dem Titel *A Relational Model of Data for Large Data Bank* stellt einen Meilenstein in der Entwicklung von Datenbankmodellen dar:

- Daten werden natürlich beschrieben, ohne Bezug auf spezielle Implementierungsdetails auf Computern (mengenorientierter Tabellenzugriff vs. navigierender Zugriff auf ISAM Dateien).
- Mathematische Fundierung des Modells (Mengen, Prädikate, funktionale Abhängigkeiten,...)
- Unabhängigkeit der Datenstrukturen von Zugriffspfaden. Uniforme Repräsentation von Objekten und Objektbeziehungen (vs. Tupel und Zeiger).

Direkt nach der Veröffentlichung des Artikels von Dr. Codd begannen Entwicklunsarbeiten am IBM San Jose Research Laboratory für ein Datenbanksystem und eine Sprache, die das relationale Modell unterstützt. Das Datenbankprojekt hieß System R und die Sprache, um Datenbankzugriffe aus Anwendungsprogrammen durchzuführen, wurde SEQUEL (Structured English Query Language) genannt Andere Hersteller waren schneller mit relationalen Produkten auf dem Markt (zum Beispiel Relational Softare Inc. heute Oracle).

IBM 1981 Produkt SQL/DS. 1983 erste Version von DB2 (heutiges Produkt). Anderes Produkt: Ingres (Relational Technology) basierend auf Entwicklungsarbeiten an der Universität Berkeley (Michael Stonebraker, Eugene Wong). Ingres und Digital Equipment Corporation: Rdb/VMS, Rdb/ELN besaßen ursprünglich ander DB-Sprachschnittstellen (QUEL bzw. RDML), jedoch Marktdruck (IBM und Standardisierung) führte zur Verdrängung dieser Sprachen trotz sprachlicher Vorteile. Weitere relationale Datenbanksysteme: Oracle, Sybase, Tandem xx

SQL-89 war unvollständig: Schemaänderungen nicht standardisiert, obwohl Funktionalität in allen DBMS vorhanden. Dynamisches SQL, Szstemtabellen, Mengenoperationen, Fehlerbehandlun Kleinster gemeinsamer Nenner.

Erste Standardisierung 1986, leicht erweitert 1989. 1992 wurde eine entscheidend erweiterte Überarbeitung des Standards abgeschlossen. Implementierungen des vollständigen SQL-92 Sprachumfangs stehen aus (1994), Lösung: Schichtung des Standards. Große Verbreitung von SQL Implementierungen macht SQL auch zum de-facto Standard. Intergalactical data speak (Stonebreaker).

Umfang SQL-92 = 580 Seiten, SQL-89 = 115 Seiten. (5-fache Komplexität?).

Schichtung: Entry SQL, Intermediate SQL, Full SQL. Entry SQL im wesentlichen SQL-89 (Ersatz für SQL-89 Konstrukte, die nicht mehr unterstützt werden, Ersatz für inkompatible Konstrukte, Vorbereitung für Full SQL [Schlüsselwortkonflikte bei Bezeichnern], Fehlerkorrekturen des SQL-89 Standards).

Conformance

Implementation-defined

Implementation-dependent

Interessante Frage: Trennung Funktionalität von Sprache. API. Erklärung ODBC von Microsoft.

1972 Bayer und McCreight veröffentlichen einen Artikel in *Acta Informatica*, in dem sie B-Trees vorstellen, eine Schlüsseltechnologie für die Implementierung wertbasierter, assoziativer Zugriffe.

1972 Die *Special Interest Group on Management of Data* (SIGMOD) der *American Association of Computing Machinery* (ACM) veranstaltet ihre erste internatinale Konferenz.

1975 ANSI SPARC definiert Schichtenarchitektur, Einflußreich auf Verständnis, nicht unbedingt Produkte.

Get more info

1975 Die *Very Large Database Foundation* organisiert die erste internationale Konferenz.

1976 Die *IFIP Working Group 2.6* XX organisiert ihre erste Konferenz.

1976 IBM System R veröffentlicht

19?? IBM DB2 veröffentlicht

1978 Jim Gray veröffentlicht den Artikel *Notes on Database Operating Systems*

Löschen oder erklären warum relevant.

19?? AMADEUS?

1979 Vorschlag: Semantische Datenmodelle und Abstraktionsmechanismen (Codd, Roussopoulos, Brodie,) Frame-Basiert

1981 Codd erhält den *ACM Turing Award*

ER Datenmodell vorgeschlagen (Peter Chen)

Seit 1979 regelmäßige Konferenz

1983 ANSI/SPARC Relational Task Group ??

1983 Härder, Reuter (ACID) in ACM CS.

198? QBE

19x? Minker, Reiter: Deduktive Datenbanken.Sehr aktiv in

19xx Erste Konferenz PODS ??

198? Ingres, Oracle, ...

198? System R,

ADABAS, TH Darmstadt

1989 Erster SQL Standard

198? Fourth-Generation Language: XXX

198? Asthon Tate' s dBase ist erstes DB-Produkt, Borland Paradox, 4th Dimension für den Macintosh... alle basierend auf dem Relationalen Modell

2.6.4 Objektorientierte Datenbanksysteme

Beginn der objektorientierten Datenbanksysteme

13.Programmiersprachenanbindung

14. Persistenzabstraktion

15.Objektidentität

16.Verhaltensmodellierung

17. Vererbung von Struktur und Operationen

18.Versionsverwaltung

Als Meilensteine in der Entwicklung objektorientierter Datenbanksysteme sind zu nennen: 19 Erster Workshop über Datenbankprogrammiersprachen. Alle zwei Jahre im Wechsel mit Persistent Object Stores.

19xx Erste Konferenz OOPSLA

19?? GemStone: Object Identity / Smalltalk

1989 OODBMS Manifesto: Revolution Politsches Pamphlet. Unbedingt notwendige, optionale und offene Bestandteile.

198? Next-Generation Manifesto: Evolution

Dittrich: Verhaltensmäßig objektorientiert, strukturell Objektorientiert

1993 ODMG-93 Standard Release 1. Gefolgt von Release 1.1 in 1994

2.6.5 Datenbanken in offenen Systemen

Evtl. Abschnitt löschen, weil zu spekulativ.

?? Legacy Databases

19?? DCE, CORBA, ...

19?? Object Store

19xx Visual Basic, PowerBuilder

198? Microsoft stellt Access vor und definiert ODBC. Übernahme durch Front-End Werkzeuge Spreadsheets und Back-Ends19?? O2

19xx Volltextsuche mit WAIS

1994 WWW Mosaic erlaubt uniformen Datenzugriff in Weitverkehrsnetzen (Internet). Paradigma: Hypertext.

1993 Gray, Reuter: Transaction Processing, Dienstsicht

Intelligent Information Systems

s. Catell 4.13-4.1.7

Gleichzeitig andere Speicherungssysteme (Volltextdatenbanken, Hypertextdokumente in Filesystemen, Bilddatenbanken. Wissenschaftliche Datenbanken

3 Die etablierten Datenbankmodelle im Überblick

Der Gruppe der wichtigsten etablierten Datenmodelle ist neben den klassischen Vertretern, hierarchisches Datenmodell, Netzwerkdatenmodell und relationales Datenmodell, inzwischen sicherlich auch die Familie der objektorientierten Datenmodelle zuzurechnen. Die praktische Bedeutung des relationalen Modells und der objektorientierten Modelle nimmt dabei stetig zu, während die beiden frühen Modelle zwar aufgrund zahlreicher umfangreicher existierender Systeme noch eine hohe Praxisrelevanz besitzen, für die Realisierung neuer Systeme aber kaum mehr eingesetzt werden.

Ziel dieses Abschnitts ist es, durch eine fokussierte Einführung der charakteristischen Konzepte und deren Erläuterung anhand eines Beispiels ein intuitives Verständnis für die unterschiedlichen Datenbankmodelle zu vermitteln. Vergleichbare Konzepte und Ansätze spielen dabei eine wichtigere Rolle als syntaktische Besonderheiten der einzelnen Modelle. Dieser Betonung der Gemeinsamkeiten und der Erleichterung eines Vergleichs dient die Verwendung einer uniformen grafischen Notation und eines einheitlichen Beispiels bei der Erläuterung der Modelle.

Die grafische Notation und das Beispiel werden in Abschnitt 3.1 vorgestellt. Die folgenden Abschnitte sind den einzelnen Datenbankmodellen gewidmet. Aufgrund ihrer abnehmenden Bedeutung und der Ähnlichkeit der zugrundeliegenden Konzepte werden das Netzwerkdatenmodell und das hierarchische Datenmodell dabei in einem Abschnitt (3.4) zusammengefaßt, während für das relationale Modell und für die Gruppe der objektorientierten Modelle jeweils ein eigener Abschnitt vorgesehen ist. Dieser Teil des Kapitels schließt mit einem zusammenfassenden tabellarischen Vergleich der betrachteten Datenmodelle in Abschnitt 3.5.

3.1 Eine datenmodellunabhängige Metanotation

Die Parallelen und Unterschiede zwischen den betrachteten Datenmodellen können am besten durch die Verwendung einer einheitlichen Notation zur Darstellung der verschiedenen Modelle aufgezeigt werden. Alle Datenbankmodelle basieren auf einem gemeinsamen, relativ kleinen Satz von Datenmodellkonzepten zur Beschreibung von Datenbankzuständen, für die im folgenden eine grafische Notation (s. Abbildung 23) gewählt wird:

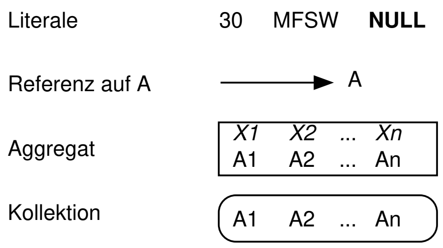

Abbildung 23: Legende für Datenbankzustände

Literale beschreiben die atomare Werte, die als Bausteine in komplexeren Informationsstrukturen verwendet werden können. Beispiele für Literale sind ganze Zahlen oder Zeichenketten. Literale werden ohne Verwendung weiterer Symbole in der Grafik notiert. Ausgezeichnete Werte mit einer speziellen Semantik, wie z.B. der „Nullwert“ **null**, werden durch Fettschrift hervorgehoben.

Aggregate beschreiben zusammengesetzte Werte, die aus einer (ungeordneten) Folge von Paaren aus Attributnamen und zugeordneten Attributwerten bestehen. Die Komponenten eines Aggregats können über ihren Namen angesprochen werden. Die meisten Datenbankmodelle unterstützen selektive Zuweisungen an Wertkomponenten, jedoch nicht das Hinzufügen oder Löschen von Attributen. Aggregate sind mit den *Records* in Pascal und den *Structures* in C vergleichbar. In der grafischen Notation werden Aggregate durch *Rechtecke* dargestellt, die in der ersten Zeile die Attributnamen X_i und in der zweiten Zeile die zugehörigen Attributwerte A_i enthalten.

Referenzen beschreiben (physikalische) Verweise auf andere Objekte. Sie sind mit den aus dem Bereich der Programmiersprachen bekannten Zeigern vergleichbar. Referenzen zwischen Objekten werden durch *Pfeile* dargestellt, die vom referenzierenden zum referenzierten Objekt führen.

Kollektionen beschreiben zusammengesetzte Werte, die aus einer variablen Anzahl unbenannter Komponenten bestehen. Es wird dabei von einer dynamischen Klassifikation der Objekte (Werte) in die Kollektionen ausgegangen. Die Kollektionen werden in der grafischen Notation durch *Rechtecke mit abgerundeten Ecken* dargestellt, die ihre Elemente A_i nebeneinander oder untereinander enthalten.

Bei den meisten Datenbankmodellen wird von einer Typisierung ausgegangen. Typisierung hat für Datenbankmodelle die folgenden Konsequenzen:

- Die Attribute sind typisiert, wodurch Operationen auf den Attributen auf ihre Zulässigkeit überprüft werden können.
- Die Kollektionen sind homogen, d.h. alle Elemente einer Kollektion besitzen den gleichen Typ.

Aufgrund der Typisierung sind auf der *Schemaebene* analoge Modellierungskonzepte zu finden, mit denen die Struktur von zulässigen Datenbankzuständen beschrieben wird:

Basisdatentypen (Domänen) beschreiben Wertebereiche. Sie setzen sich aus Mengen von Literalen zusammen.

Aggregattypen beschreiben die Struktur von Aggregaten. Sie setzen sich aus Paaren von Bezeichnern und Typen für die Attribute des Aggregats zusammen.

Kollektionstypen (Massendatentypen) beschreiben die Struktur von Kollektionen. Dabei wird die Art der Kollektion (Liste, Menge, etc.) und der Typ der Elemente der Kollektion festgelegt.

Mengenwertige Referenzen

Aus diesen Betrachtungen läßt sich bereits schließen, daß es eine gewisse Verwandtschaft zwischen Sprachen zur Schemadefinition und den Konstrukten zur Typdefinition in Programmiersprachen gibt. Eine Konvergenz dieser beiden verwandten Konzepte ist eine der wesentlichen Grundlagen der Datenbankprogrammiersprachen, die in Abschnitt 8.2 behandelt werden.

Um diese Betonung der Gemeinsamkeiten zu erreichen und um einen Vergleich der Modelle zu erleichtern, wird zur Erläuterung der verschiedenen Modelle ein einheitliches Beispiel gewählt. Alle betrachteten Datenmodelle werden anhand einer Projektdatenbank vorgestellt, die einen Ausschnitt des Firmeninformationssystems aus Abschnitt 1.6 bildet. Der Ausschnitt beschränkt sich auf die Darstellung der Projekte und Abteilungen zusammen mit den Beziehungen zwischen diesen Objekttypen (s. Abbildung **??**).

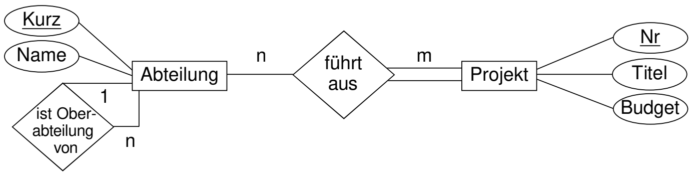

Abbildung 24: Durchgängiges Beispiel für Kapitel 3

3.2 Das relationale Datenmodell

Wegen seiner Beliebtheit und weiten Verbreitung in der Forschungsgemeinde und auch im kommerziellen Bereich spielt das relationale Modell eine wichtige Rolle im Bereich der Datenbanken. Im kommerziellen Bereich ersetzen relationale Systeme mit wachsender Tendenz Systeme, die auf früheren Modellen, wie dem Netzwerkmodell und dem Hierarchischen Modell, beruhen.

Der Erfolg des relationalen Datenmodells ist auf seine Einfachheit zurückzuführen. Es gibt nur eine Art von Datenstrukturen: Tabellen mit Zeilen und Spalten, die Daten von skalaren Datentypen enthalten. Die Anfragesprache basiert auf einigen wenigen, wohldefinierten Operationen auf Tabellen. Das Verständnis der komplexeren Operationen, z.B. zur Definition von Integritätsbedingungen und Sichten, sowie zur Restrukturierung der Daten ist für den Endbenutzer nicht relevant.

Die Einfachheit des Modells resultiert nicht nur in einer hohen Benutzerfreundlichkeit, sondern auch in einer guten mathematischen Handhabbarkeit des Modells.

Nach einem Überblick über die Konzept in Abschnitt 3.2.1 werden Tabellen als grundlegender Modellierungsmechanismus im relationalen Datenmodell vorgestellt. Die Darstellung von Beziehung ist Thema von Abschnitt 3.2.3. Domänen werden in dem darauf folgenden Abschnitt 3.2.4 behandelt. Der Vermeidung von Redundanzen und damit verbundener Anomalien ist Abschnitt 3.2.5 gewidmet. Die restlichen Abschnitte wenden sich der Sprache SQL zu, die sowohl zur Datendefinition als auch zur Datenmanipulation eingesetzt wird.

3.2.1 Überblick über die Konzepte

Die wichtigsten Konzepte des relationalen Datenmodells lassen sich wie folgt zusammenfassen:

- Im relationalen Datenmodell besteht eine Datenbank aus einer Menge von *Relationen*.
- *Relationen* sind Mengen von Elementen
  - deren Struktur durch Attribute definiert,
  - deren Identität durch Schüssel realisiert und

  - deren Werte durch Domänen kontrolliert werden.

  Da Relationen Mengen sind, sind die enthaltenen Elemente unsortiert und es gibt keine Duplikate. Weiterhin ist auch die Reihenfolge der Attribute nicht festgelegt.

- Relationen werden meist durch *Tabellen* dargestellt, wobei jede Tabelle aus Zeilen und Spalten besteht.
- Jede Zeile einer Tabelle repräsentiert ein Element der Relation, das auch als *Tupel* bezeichnet wird.
- Die Zahl der Zeilen (= Anzahl der Tupel in der Relation) ist variabel und wird die *Kardinalität* der Relation genannt.
- Die Spalten der Tabellen enthalten die *Attribute* der Relation.
- Die Zahl der Spalten einer Tabelle (= Anzahl der Attribute der Relation) wird im Schema festgelegt und als *Grad* der Relation bezeichnet.
- Jeder Spalte ist eine *Domäne* zugeordnet, die die zulässigen Werte für das Attribut in allen Reihen festlegt. Im relationalen Modell müssen die Domänen atomare Werte besitzen.
- Jede Tabelle besitzt einen *Primärschlüssel*. Das ist ein ausgezeichnetes Attribut oder eine minimale Attributkombination, die eine eindeutige Identifizierung jedes Tupels der Relation über den Wert der Primärschlüsselattribute gestattet.
- Beziehungen zwischen Datenobjekten werden im relationalem Modell nicht durch Referenzen, sondern durch die Identifikation des referenzierten Objekts über seinen Primärschlüssel repräsentiert (assoziative Identifikation, vgl. Abschnitt 2.1.5).
- Ein Primärschlüssel, der in einer Relation A benutzt wird, um ein Tupel (Objekt) in einer Relation B zu referenzieren, wird als *Fremdschlüssel* von A bezeichnet.

3.2.2 Tabellen und Schlüssel

Zur Modellierung der Projektdatenbank im relationalen Modell werden drei Tabellen benötigt, eine für jede Art von Objekten der Anwendung (*Projekt* und *Abteilung*) und eine Tabelle *Projektdurchführung* für die Darstellung der Beziehung zwischen Abteilungen und Projekten. Abbildung 25 zeigt einen Zustand der Projektdatenbank bei der gewählten Modellierung. Dabei entspricht jede Zeile einem Element der Relation. Die Spalten der Tabellen entsprechen den Eigenschaften der Objekte der Anwendung.

Eine relationale Datenbank ist ein benanntes Aggregat, wobei der Bezeichner für das Aggregat der Schemaname ist. Im Beispiel besitzt das Aggragat drei Komponenten, nämlich die Relationen *Projekte*, *Abteilungen* und *Projektdurchführung*. Jede der Komponenten ist eine Kollektion. Die Elemente der Kollektionen sind die Tupel der Relation also wiederum Aggregate, deren Komponenten nur Literale sein dürfen (flache Tupel).

Schlüssel sind von zentraler Bedeutung für das relationale Datenmodell. Dabei ist zwischen zwei Arten von Schlüsseln zu unterscheiden:

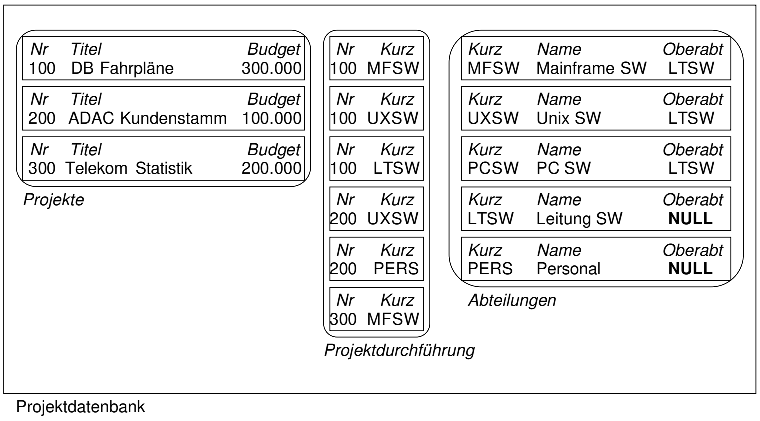

Abbildung 25: Zustand der Firmendatenbank (Relationales Datenmodell)

Primärschlüssel: Ein *Primärschlüssel* ist ein Attribut oder eine minimale Attributkombination, das bzw. die in jeder Reihe der Tabelle eindeutige Werte bzw. Wertkombinationen enthält. In einigen Fällen gibt es mehrere Attribute (Attributkombinationen) mit dieser Eigenschaft. Man spricht dann von mehreren *Schlüsselkandidaten*, von denen einer als Primärschlüssel ausgezeichnet wird. Man kann sich die Primärschlüssel als eindeutige Bezeichner für die durch die Reihen einer Tabelle repräsentierten Objekte der realen Welt vorstellen. Im Beispiel besitzt die Tabelle *Abteilungen* das Attribut *Kurz* als Primärschlüssel und die Tabelle *Projekte* das Attribut *Nr*.

Fremdschlüssel: Ein *Fremdschlüssel* ist ein Attribut oder eine Attributkombination in einer Tabelle, das bzw. die als Werte Primärschlüssel anderer Tabellen annimmt. Ein Fremdschlüssel ist also eine Referenz auf ein anderes Objekt (=eine Reihe einer nicht notwendigerweise verschiedenen Tabelle). In der betrachteten Anwendung ist z.B. das Attribut *AbtKurz* der Tabelle *Projektdurchführung* ein Fremdschlüssel.

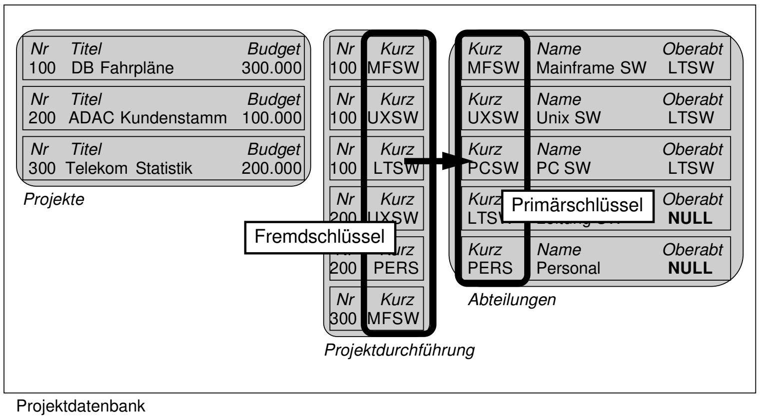

Abbildung 26: Fremdschlüssel

Das relationale Modell erlaubt keine Duplikate in einer Tabelle, d.h. keine zwei Tupel dürfen in allen Attributen übereinstimmen. In diesem Sinne haben alle Tabellen automatisch einen Primärschlüssel, nämlich alle Attribute der Relation. Die Kombination aller Attribute ist jedoch als Primärschlüssel zur Identifikation der Objekte nicht sinnvoll. Für den Fall, daß eine ein- oder zweielementige Teilmenge der Attribute zur Identifikation nicht ausreicht, empfiehlt sich die Einführung eines künstlichen Schlüssel z.B. einer eindeutigen Nummer (ID) für jeden Tupel.

Die Primärschlüssel und Fremdschlüssel der Tabellen spielen eine wichtige Rolle für die Repräsentation von Beziehungen zwischen Objekten im relationalen Modell.

3.2.3 Assoziationen im relationalen Modell

In den meisten modernen Anwendungen des relationalen Modells unterscheiden die Benutzer zwischen zwei Arten von Tabellen. Tabellen, die Objekttypen (Entitäten) der Anwendung darstellen, werden als *Entitätentabellen* bezeichnet. Jede Zeile einer solchen Tabelle repräsentiert ein Objekt der Anwendung.

Die zweite Art von Tabellen wird als *Beziehungstabellen* bezeichnet. Beziehungstabellen werden unter Verwendung von Fremdschlüsseln zur Darstellung von Assoziationen zwischen Objekten der Anwendung eingesetzt.

Dabei wird nicht zur Darstellung jeder Beziehung eine eigene Tabelle benötigt. Binäre 1:1- und 1:n-Beziehungen lassen sich durch Aufnahme eines Fremdschlüsselattributs in eine der Entitätentabellen beteiligten Objekttypen repräsentieren.

Ein Beispiel ist die Abteilungshierarchie, eine optionale 1:n-Beziehung zwischen Abteilungen. Um diese Beziehung darzustellen, wird der Fremdschlüssel *Oberabteilung* als zusätliches Attribut in die Tabelle *Abteilung* eingeführt. Dieser Fremdschlüssel referenziert die Oberabteilung einer Abteilung. Sogar Beziehungen höheren Grades (vgl. Abschnitt 2.1.4) können ohne Beziehungstabellen modelliert werden, solange höchstens einer der beteiligten Objekttypen eine Multiplizität größer eins in der Beziehung besitzt.

Beziehungstabellen sind notwendig, wenn mehr als eine der beteiligten Objekttypen eine höhere Multiplizität hat, wie dies z.B. bei n:m-Beziehungen der Fall ist. Die n:m-Beziehung zwischen *Projekte* und *Abteilungen* muß somit durch eine eigene Beziehungstabelle *Projektdurchfuehrung* dargestellt werden (s. Abbildung 25). Diese Tabelle besitzt zwei Attribute *AbtKurz* und *ProjektNr*, die beide Fremdschlüssel sind: Das erste Attribut referenziert eine Abteilung, das zweite Attribut referenziert ein Projekt. Als Werte für die Fremdschlüssel sind dabei Abteilungskürzel (= Primärschlüssel von *Abteilungen*) und Projektnummern (=Primärschlüssel von *Projekte*) möglich.

Für jedes einzelne Paar Abteilung-Projekt enthält die Beziehungstabelle *Projektdurchführung* eine eigene Zeile. So erscheint z.B. das Abteilungskürzel „UXSW“ zweimal in der Tabelle, da die zugehörige Abteilung an zwei Projekten mitarbeitet und die Prokjektnummer 100 erscheint dreimal in der Tabelle, da drei Abteiungen an dem Projekt mit dieser Nummer arbeiten.

Man beachte, daß Abbildung 25 keine Referenzen zwischen Objekten enthält, für die in Abschnitt 3.1 die Pfeilnotation eingeführt wurde. Die Beziehungen zwischen den Objekten werden im relationalen Modell allein durch gleiche Werte in zusammengehörigen Primärschlüssel/Fremdschlüssel-Paaren dargestellt. Es existieren *keine* Zeiger oder vergleichbare Strukturen zur Verknüpfung von Objekten. Das relationale Modell basiert auf assoziativer Identifikation.

3.2.4 Domänen und referentielle Integrität

Allgemein stellt *referentielle Integrität* sicher, daß Objekte, die referenziert werden, auch tatsächlich in der Datenbank existieren. Beim relationalen Modell ist die referentielle Integrität eng mit der Verwendung der Fremdschlüssel zur Darstellung von Beziehungen zwischen Objekttypen verbunden. Zur Wahrung der referentiellen Integrität muß eine Menge von Integritätsbedingungen definiert werden, die sicherstellen, daß zu jedem in der Datenbank benutzten Fremdschlüssel in der referenzierten Tabelle ein Tupel mit einem entsprechenden Primärschlüsselwert existiert. So muß bei der Projektdatenbank zu jedem Wert k des Attributs *AbtKurz* in der Tabelle *Projektdurchfuehrung* ein Tupel in der Tabelle *Abteilung* mit dem gleichen Wert k für das Attribut *Kurz* existieren.

Ein gutes relationales DBMS sollte die Wahrung der referentiellen Integrität sicherstellen. Eine Überprüfung ist in den folgenden Situationen notwendig:

- Beim Einfügen eines neuen Fremdschlüsselwertes in ein Attribut muß überprüft werden, ob in der referenzierten Tabelle ein Objekt mit diesem Wert als Primärschlüssel existiert. So muß z.B. ein Projekt mit einer entsprechenden Nummer existieren, bevor diese Nummer als Wert für den Fremdschlüssel *ProjektNr* in der Tabelle *Projektdurchfuehrung* verwendet werden kann.

- Beim Löschen eines Tupels aus einer Tabelle muß überprüft werden, ob dieses Tupel noch referenziert wird, d.h. ob sein Primärschlüssel noch irgendwo als Fremdschlüsselwert in Gebrauch ist. Falls noch Referenzen auf den Primärschlüssel existieren, gibt es verschiedene Optionen für das weitere Verfahren:

  1. Es wird eine Fehlermeldung erzeugt.
  2. Die Löschoperation wird propagiert, d.h. das referenzierende Tupel wird ebenfalls gelöscht, was zur Notwendigkeit weiterer Löschungen führen kann (*Kaskadiertes Löschen*).
  3. Die Referenz wird ungültig gemacht, indem der Fremdschlüssel auf einen Nullwert gesetzt wird.

Ein Beipiel für kaskadiertes Löschen sind die Konsequenzen der Entfernung einer Abteilung a aus der Projektdatenbank. Mit der Abteilung müssen alle Tupel in *Projektdurchführung* gelöscht werden, die sich auf diese Abteilung beziehen. Zugleich müssen aber auch alle Abteilungen gelöscht werden, die a über das Attribut *Oberabteilung* referenzieren, d.h. alle Unterabteilungen von a. Dies erfordert dann evtl. wiederum ein Löschen in der Tabelle *Projektdurchfuehrung*, sodaß eine einzelne Löschoperation weitreichende (und auch unerwartete) Folgen haben kann.

Ein weiteres wichtiges Konzept des relationalen Modells sind die *Domänen*. Domänen legen zulässige Wertebereiche für Attribute fest und sind mit Typen für Attribute vergleichbar. Domänen können mit vordefinierten Typen übereinstimmen oder spezielle Mengen von Werten für ein Attribut festlegen.

Das Domänenkonzept erlaubt es, die Zulässigkeit von Operationen auf Attributen zu überprüfen. So kann z.B. der Vergleich von Attributen wie die Nummer und das Budget eines Projekts, die verschiedene unterliegende Domänen besitzen, verhindert werden. Von den meisten relationalen Systemen wird das Domänenkonzept jedoch nicht realisiert, da eine vollständige und flexible Implementierung von Domänen eine erhebliche Komplexität besitzt. Eine ausführlichere Beschreibung des Domänenkonzepts findet sich in [**?**, **?**].

3.2.5 Normalisierung

Viel Forschung im Bereich der Datenbanken besonders in den 80er Jahren hatte die *Normalisierung* relationaler Schemata zum Thema. Normalisierung beschäftigt sich mit der besten konzeptuellen Darstellung von Anwendungsdaten durch relationale Tabellen; dieses Problem kann für umfangreiche Schemata recht komplex werden. Im wesentlichen werden bei der Normalisierung große Tabellen (hoher Grad) in kleinere Tabellen zerlegt mit dem Ziel, nicht zu viel Information in eine Tabelle zu packen.

Mit „zuviel" Information in einer Tabelle können die folgenden Probleme verbunden sein:

Redundanz: Dieselbe Information wird an mehr als einer Stelle gespeichert. Wenn man z.B. die Attribute der Projekte mit in die Relation *Projektdurchführung* aufnimmt, anstatt sie in einer eigenen Relation *Projekt* zu speichern, führt dies zu Redundanz, da dasselbe Projekt mit mehreren Abteilungen in Beziehung stehen kann. Redundanz kann bei Aktualisierungsoperationen zu Inkonsistenzen führen, wenn die Information nicht an allen Stellen geändert wird.

Abhängigkeiten: Bestimmte Informationen können nicht eingefügt werden, ohne daß bestimmte andere Informationen bereits existieren. Bei der unter dem Punkt Redundanz angesprochenen Modellierung kann z.B. ein Projekt nur zusammen mit einem Objekt vom Typ *Projektdurchführung* eingefügt werden.

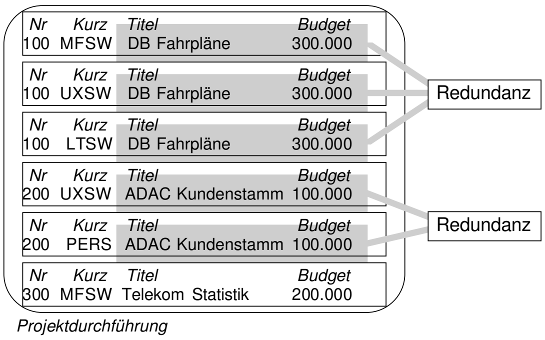

Abbildung 27: Redundanz in der Entitätentabelle *Projektdurchführung*

In der Datenbankliteratur wurden nacheinander eine Reihe von Normalformen entwickelt, die sich mit verschiedenen Anomalien bei der relationalen Schemapräsentation auseinandersetzen. Die *erste Normalform* fordert „flache" Tabellen, d.h. Attribute dürfen nicht mengenwertig sein. Die zweite, dritte und die weiteren Normalformen beschäftigen sich mit den semantischen Abhängigkeiten zwischen den aktuellen Daten in den Tabellen. Diese Normalformen sind besonders für Datenbankadministratoren beim Entwurf relationaler Schemata von Interesse. Für ein mehr konzeptuelles Verständnis des relationalen Modells sind sie jedoch nicht notwendig und werden deshalb hier nicht weiter behandelt. Ein guter Überblick über das Thema findet sich in [**?**].

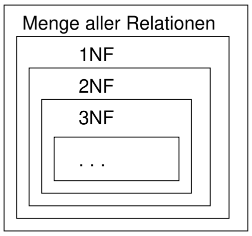

Abbildung 28: Normalformen und ihr Zusammenhang

```
create table Projekte
( Nr integer not null,
  Titel char (30) not null,
  Budget decimal(10,2) not null,
  primary key(Nr) );

create table Abteilungen
( Kurz char(4) not null,
  Name char(30) not null,
  Oberabteilung char(4),
  primary key(Kurz) );

create table Projektdurchfuehrung
( ProjektNr integer not null,
  AbtKurz char(4) not null,
  primary key(Nr, Kurz) );
```

Abbildung 29: SQL-Schemadefinition für die Projektdatenbank

3.2.6 Datendefinition im relationalen Modell

Für relationale Datenbanksysteme hat sich die Datenbanksprache SQL zum Quasi-Standard entwickelt. Sie wird von fast allen relationalen Systemen unterstützt. SQL umfaßt Konstrukte zur Datendefinition, zur deskriptiven Formulierung von Anfragen, zur Modifikation der Datenbank und zur Schemaänderung. Es wird hier nur soweit auf die SQL-Konstrukte eingegangen, wie dies für ein intuitives Verständnis des relationalen Modells notwendig ist. Eine ausführlichere Beschreibung findet sich in Abschnitt 4, der ganz diesem Thema gewidmet ist.

Abbildung 29 zeigt eine SQL-Schemadefinition für die Projektdatenbank. Die Definition eines SQL-Schemas umfaßt eine Reihe von Tabellendefinitionen. Für jede der Tabellen werden der Name der Tabelle, die Attribute (Attributnamen und -typen) und der Primärschlüssel festgelegt. Für Attribute die keine Nullwerte annehmen dürfen, wird die Angabe **not null** gemacht. sehr reguläre Struktur

Aufgrund dieser Schemadefinition ergibt sich das in Abbildung **??** dargestellte Schemadiagramm.

3.2.7 Die Anfragesprache SQL

Ursprünglich wurden für das relationale Modell neben anderen Anfragesprachen die formalen Anfragesprachen relationales Kalkül und relationale Algebra vorgeschlagen (s. Abschnitt

2.6.3). Inzwischen hat sich jedoch die deskriptive, mengenorientierte Sprache SQL allgemein als einheitliche Anfragesprache für relationale Systeme durchgesetzt.

---

select *p.Titel*
from *Projekte p,*
  Projektdurchfuehrung pd,
  Abteilungen a
where *p.Nr = pd.Nr*
and *a.Kurz = pd.Nr*
and *a.Name = 'Mainframe SW'*

Abbildung 30: Beispielanfrage in SQL

Abbildung 30 zeigt eine Beispielanfrage in SQL. Diese Anfrage bestimmt die Titel der Projekte, an denen die Abteilung für die *Mainframe SW* arbeitet. Typischerweise verwenden SQL-Anfrage das select from where-Konstrukt, das zur Iterationsabstraktion dient. Es besteht aus drei Teilen: einer select-Klausel, einer from-Klausel und einer where-Klausel. Dabei legt die from-Klausel die beteiligten Tabellen fest und die where-Klausel ein Selektionsprädikat, mit dessen Hilfe die Ergebnistupel aus dem kartesischen Produkt der beteiligten Tabellen selektiert werden. Die from-Klausel spezifiziert die Projektionsliste für die Ergebnistabelle. Nur durch diese Projektionsliste ausgewählte Spalten treten auch in der Tabelle auf, die die Anfrage als Ergebnis berechnet.

Typisch für eine Anfrage im relationalen Modell ist die Notwendigkeit, einen sog. *Join* über mehrere Tabellen durchzuführen. Die Tabellen werden dabei wertbasiert, z.B. über gleiche Werte in zusammengehörigen Primärschlüssel/Fremdschlüssel-Paaren, miteinander verknüpft. Im Beispiel ist dies ein Join über die drei Tabellen *Projekte*, *Abteilungen* und *Projektdurchführung*.

`Etwas zum deskriptiven Charakter`

3.2.8 Aktualisierungsoperationen in SQL

Unabhängig vom jeweiligen Datenmodell lassen sich Modifikationsoperationen auf Datenbanken in drei grundlegende Klassen einteilen: Neben Operationen zum *Einfügen* neuer Daten gibt es Operationen zum *Ändern* und *Löschen* von Daten, die in der Datenbank existieren. Die Modifikationsoperationen können dabei sowohl Daten betreffen, die Objekte der realen Welt repräsentieren, als auch Daten und Strukturen, die Beziehungen zwischen solchen Objekten repräsentieren.

Da das relationale Modell diese Arten von Information beide durch Tabellen repräsentiert, werden dafür im Gegensatz zu anderen Datenmodellen keine unterschiedlichen Operationen benötigt. Zur Modifikation einer Datenbank unterstützt SQL die generischen Operationen **insert**, **update** und **delete**. Neben einem Parameter zur Selektion einer Tabelle (Tabellenname), den alle drei Operationen benötigen, muß bei den Operationen **update** und **delete** eine Selektion der (des) betroffenen Tupel(s) durchfgeführt werden. Zur Datenselektion können dabei Konstrukte der Anfragesprache eingesetzt werden. Bei der update-Operation werden zusätzlich für ein oder mehrere Attribute neue Werte oder Formeln zu deren Berechnung angegeben, die auf die bisherigen Werte des Attributs zurückgreifen können.

Eine insert-Operation fügt in der Regel nur ein einzelnes Tupel ein, dessen Attributwerte der Operation als Parameter übergeben werden. Abbildung 31 zeigt eine spezielle Form der

insert into *Projektdurchfuehrung (Nr, Kurz)*
select *p.Nr, a.Kurz*
from *Projekte p, Abteilungen a*
where *p.Titel = 'Telekom Statistik'*
and *a.Name = 'Unix SW' ;*

Abbildung 31: SQL-Einfügeoperation

insert-Operation, die eine SQL-Anfrage zur Definition des einzufügenden Tupels benutzt[1]. Die dargestellte Operation definiert eine Beziehung zwischen dem Projekt *Telekom Statistik* und der Abteilung *Unix Software*, indem sie ein entsprechendes Tupel in die Tabelle *Projektdurchführung* einfügt. Die Attributwerte für das einzufügende Tupel (Fremdschlüssel) werden über ein sfw-Konstrukt aus existierenden Tupel der Datenbank selektiert. Im Beispiel werden die Schlüssel des Projekts und der Abteilung, die zur Dartsellung der Beziehung als Fremdschlüssel benötigt werden, über deren Namen ermittelt.

3.2.9 Zusammenfassung

Die Einfachheit des relationalen Modells beruht im wesentlichen auf den folgenden drei Eigenschaften der Tabellen:

- Tabellen sind Mengen im mathematischen Sinne: Duplikate sind bei den Reihen nicht erlaubt, es ist keine Ordnung auf den Reihen definiert;
- Die Attribute sind nicht geordnet, sie werden immer über ihren Namen referenziert, die innerhalb einer Tabelle eindeutig sein müssen;
- Die Werte, die in den Spalten und Reihen der Tabelle gespeichert werden sind atomar, was insbesondere mengenwertige Attribute ausschließt

Diese Einschränkungen stellen zugleich auch Beschränkungen des relationalen Modells dar. Es ist zum Beispiel schwierig Objekte, die in der Realität geordnet sind, wie z.B. die Kapitel eines Buches, effizient durch eine Tabelle zu repräsentieren. evtl bei OO-Modell

`Bewertung RDM, auch in Folien, in OO-Manifesti, Paper über Bewertung von SQL, Bewertung evtl. nicht hier sondern bei OO-System Kritik auch in Heuer Buch + Won-Buch`

3.3 Objektorientierte Datenmodelle

`Alternative Objektorientierte Datenmodelle`

Zahlreiche Aktivitäten in der Entwicklung experimenteller und kommerzieller Systeme Das Feld der objektorientierten Datenbanktechnologie ist momentan geprägt durch:

- Das Fehlen eines *einheitlichen Datenmodells*: Es sind zwar eine Reihe von objektorientierten Datenmodellen entwickelt worden (Literatur xx). Es gibt kein einheitliches Modell, Warum möchte man ein einheitliches Datenmodell Ein einheitliches, standardisiertes Datenmodell erhöht die Portabilität

[1] Mit dieser Form der insert-Operation können auch Mengen von Tupeln eingefügt werden.

- Das Fehlen von *formalen Grundlagen*, wie sie z.B. für das relationale Datenmodell oder die deduktiven Datenbanken existieren. Solche Grundlagen sind insbesondere für die Definition der Semantik der Konzepte und Operatoren eines Datenmodells von Bedeutung.

Zwei wichtige Beispiele für Aktivitäten, die sich um die Untersuchung der zentralen Aspekte und Konzepte objektorientierter Datenmodelle und um die Festlegung eines einheitliche Datenmodells bemühen sind die folgenden:

Anforderungskataloge: Die sogenannten *objektorientierten Manifesti* [] und [] bemühen sich darum die Konzepte herauszukristallisieren, die ein objektorientiertes Datenbanksystem ausmachen. Die präsentierte Sammlungen von Konzepten sind als Diskussionsgrundlage für die Erstellung einer Definition für objektorientierte Systeme gedacht.

Standardisierungvorschläge: Einen wichtigen Beitrag zur Standardisierung liefert die Arbeit der *Object Database Management Group* (*ODMG*), einem Konsortium aus Vertretern vieler namhafter kommerzieller Anbieter von Datenbanksystemen. Die ODMG erarbeitet einen Standardierungsvorschlag ODMG-93 für ein objektorientiertes Datenbankverwaltungssytem (*object database management system*, *ODBMS*), der in Version 1.1 in [] vorliegt, aber zum Zeitpunkt der Entstehung dieses Buches noch weiterentwickelt wird. Der Standardisierungsvorschlag umfaßt ein Objektmodell, eine Datendefinitionsprache, eine Anfragesprache und Sprachanbindungen für C++ und Smalltalk. Dem Standardisierungsvorschlag kommt insofern besondere Bedeutung zu, da alle beteiligten kommerziellen Anbieter zugesagt haben, ein ODMG-93-konformes Produkt auf den Markt zu bringen. In diesem Abschnitt wird ein kurzer Überblick über die Datenmodellierungsaspekte des ODMG-Vorschlags gegeben. Eine ausführlichere Beschreibung des Vorschlags findet sich in Abschnitt 5.

`Alternative Standardisierungsvorschläge`

Im Anschluß werden zunächst Anforderungen erläutert, die an objektorientierte Datenmodelle gestellt werden können. Die nachfolgenden Abschnitte befassen sich dann mit dem Standardisierungsvorschlag der ODMG. In Abschnitt 3.3.2 wird ein Überblick über das Modell anhand markanter Begriffe gegeben, Abschnitt 3.3.3 behandelt Objekttypen und stellt das Firmeninformationssystem in einer objektorientierten Modellierung dar. Grundlegende Mechanismen zur Assoziation und Datendefinition werden in Abschnitt 3.3.4 und 3.3.5 behandelt. Der Überblick über den ODMG-Standard schließt mit der Vorstellung der Anfragesprache OQL (Asbchnitt 3.3.6) und den Aktualisierungsoperationen (Abschnitt 3.3.7).

3.3.1 Anforderungen und Konzepte objektorientierter Datenmodelle

Im folgenden werden die wichtigsten Konzepte objektorientierter Datenmodelle kurz beschrieben. Die Darstellung orientiert sich an den Anforderungen, die in den OO-Manifesti zusammengestellt sind.

Die Anforderungen betreffen z.T. die Modellierungsmächtigkeit und z.T. die Systemfunktionalität. Da die Datenmodellierung Thema dieses Abschnitts ist, werden hier im wesentlichen nur die Anforderungen and as Datenmodell berücksichtigt.

Typkonstruktoren und komplexe Objekte: `Etwas zum Bgriff komplexe Oblekte (eigentlich komplexe Werte ??` Komplexe Objekte werden aus einfacheren Objekten durch die

Verwendung von Konstruktoren aufgebaut. Die Grundbausteine sind dabei die Basistypen wie *Integer* und *Boolean* (einfachste Objekte). Es gibt eine Reihe verschiedener Konstruktoren, die größtenteils aus dem Bereich der Programmiersprachen bekannt sind.

Als Mindestanforderung an ein objektorientiertes Datenmodell kann die Unterstützung eines Konstruktors für Aggregattypen, wie z.B. *Tuple*, eines Mengenkonstruktors, wie z.B. *Set* und eines Konstruktors für geordnete Kollektionen wie *List* oder *Array* betrachtet werden. Weiterhin sind die Konstruktoren eines OO-Datenmodells orthogonal kombinierbar, d.h. jeder Konstruktor kann auf jedes Objekt angewandt werden.

Im Vergleich dazu ist das relationale Modell auf einen Konstruktor für Aggregattypen (Tupel) und einen Konstruktor für Mengen (Relationen) beschränkt. Die Orthogonalität der Konstruktoren ist im relationalen Modell nicht gegeben. Der Konstruktor für Aggregate kann nur auf Basistypen angewandt werden. Auf diese Weise konstruierte Strukturen sind die einzig zulässigen Elementtypen für die Konstruktion von Mengentypen (Relationen), d.h. das relationale Modell ist beschränkt auf Strukturen der folgenden Art:

xx

Objektidentität: Objekte besitzen eine Existenz, die unabhängig von ihren Werten ist. Zu diesem Zweck wird jedem Objekt bei seiner Erzeugung ein Objektidentifikator mit den folgenden Eigenschaften zugeteilt:

- Der Identifikator ist systemweit eindeutig und garantiert die Indentifizierbarkeit.
- Der Identifikator bleibt unabhängig vom Zustand und den Zustandsänderungen des Objekts gleich.
- Die Identifikatoren werden vom System verwaltet und können nicht vom Benutzer geändert werden.

Die Objektidentität erlaubt die Referenzierung eines Objekts durch mehrere andere Objekte (*Object Sharing*). Dadurch lassen sich beliebige Objektgraphen aufbauen, die auch zyklische Strukturen enthalten können.

`Object Update`

Die Unterstützung von Objektidentität ermöglicht verschiedene Äquivalenzbegriffe für Objekte (s. Abbbildung 32):

- Man spricht von der *Identität* zweier Objekte, wenn es sich um das gleiche Objekt handelt d.h. die Identifikatoren der Objekte sind gleich.
- Man spricht von *flacher Gleichheit* (*shallow equality*), wenn auf der obersten Strukturierungsebene beide Objekte die gleichen Werte und gleiche Referenzen auf andere Objekte (Identität der Referenzen) besitzen.
- Man spricht von *tiefer Gleichheit* (*deep equality*), wenn die beiden Objekte auf allen Strukturierungsebenen wertgleich sind. Um tiefe Gleichheit zu testen, ist ein rekursiver Test der flachen Gleichheit für die referenzierten Objekte entlang des Objektgraphen der verglichenen Objekte notwendig.

`persistente Objektidentität, abstraktes Konzept für das es verschiedene Realisierungen gibt.`

Abbildung 32: Gleichheit, Identität und Kopieroperationen im ODMG-Objektmodell

Zu den verschiedenen Varianten des Äquivalenzbegriffs auf Objekten gibt es entsprechende Kopieroperationen (s.a. Abbildung 32). Beim *Gleichsetzen* verweisen die beiden Bezeichner danach auf das gleiche Objekt. Bei einer *flachen Kopie* (*shallow copy*) Wird ein neues Objekt erzeugt, das dieselben Werte und Objektidentifikatoren enthält, wie das kopierte Objekt. Bei einer tiefen Kopie wird ebenfalls ein neues Objekt erzeugt, aber im Gegensatz zur flachen Kopie wird vom ganzen Objektgraph, der vom kopierten Objekt aus erreichbar ist, eine Kopie erzeugt.

`Änderungen`

Kapselung: Das Prinzip der Kapselung ist aus dem Bereich der Programmiersprachen bekannt, wo sie meist durch ein geeignetes Modulkonzept realisiert ist. Die Nützlichkeit eines Modulkonzepts zur Strukturierung und zur Erhöhung der Wartbarkeit hat sich in diesem Bereich erwiesen.

Im Bereich der objektorientierten Datenmodelle ist der Zustand eines Objekts gekapselt. Es kann nur über die Methoden des Objekts auf den Zustand zugegriffen werden.

Kapselung unterstützt eine Art von logischer Datenunabhängigkeit (vgl. Abschnitt 2.3), die Realisierung des Objekttyps und die Implementierung der Methoden kann verändert werden, ohne daß eine Anpassung der Anwendungsprogramme, die Objekte dieses Typs benutzen, notwendig wird.

Durch die Kapselung ist sichergestellt, daß nur über die dafür vorgesehenen Methoden auf den Zustand eines Objekts zugegriffen werden kann. Integritätsüberwachung und Zugriffskontrolle können daher im Rahmen der Methoden durchgeführt werden.

Man spricht von *strikter Kapselung*, wenn ausschließlich über die Methoden auf den Zustand (die Attribute) eines Objekts zugegriffen werden kann. Unter gewissen Umständen ist es sinnvoll, eine Verletzung der Kapselung zuzulassen. Dies ist z.B. im Bereich des rein lesenden Zugriffs (Anfragesprache) der Fall.

Typen und Klassen: Unterscheidung der beiden Begriffe nach Anwendungszeitpunkt

- Ein *Typ* legt die Struktur seiner Elemente und die zulässigen Wertebereiche fest. Die Typinformation wird zur Übersetzungszeit eingesetzt, Zweck: Vermeidung von Laufzeitfehlern, Wahl von Speicherungsstrukturen und Optimierungen
- Extensionsverwaltung

  Die Klassseninformation wird zur Laufzeit verwendet, z.B. zum Überprüfen der aktuell existierenden Objekte

Vererbung: Klasssen können in Subklassenhierarchien angeordnet werden, Die Subklassen erben dabei die Attribute und Methoden ihrer Superklassen. Die Attribute und Methoden können in der Subklasse redefiniert werden.

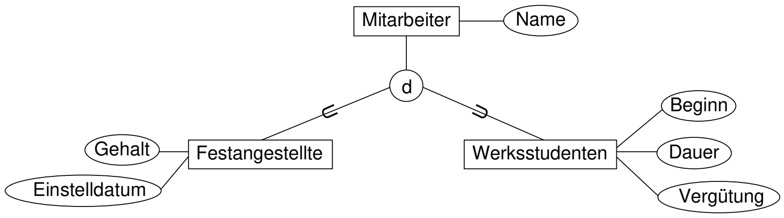

Abbildung 33: Vererbung im E/R-Diagramm

Vererbung ist ein mächtiges Modellierungskonzept zur Beschreibung von Gegebenheiten der realen Welt. In der realen Welt existierende *Is-a*-Beziehungen lassen sich durch Subklassenbeziehungen adäquat darstellen. Im relationalen Modell können solche Beziehungen nur indirekt repräsentiert werden, wodurch ihre eigentliche Semantik weitgehend verloren geht. Weiterhin unterstützt die Vererbung die Wiederverwendung von Definitionen und Implementationen unter Berücksichtigung der Klassenunterschiede.

`Subtypisierung`

Speziellere Elemente (mehr oder verfeinerte Attribut) oder Objekte die eine bestimmte Einschränkung (Prädikat) erfüllen. Im zweiten Fall ist durch Überprüfung des entsprechenden Prädikats eine automatische Verwaltung der Subklassenextension möglich.

Eine Subklassenbeziehung zwischen zwei Klassen impliziert eine Inklusionsbeziehung zwischen ihren Extensionen. Jedes Objekt einer Subklassse ist auch eine Instanz aller Superklassen.

Es ist auch möglich, zu einer Klasse mehrere direkte Superklassen zuzulassen. Man spricht dann von *Mehrfachvererbung*, da Vererbung von allen Superklassen an die betrachtete Klasse stattfindet.

Methodenredefinition und späte Bindung: Für die Methode einer Klasse kann es unterschiedliche Implementierungen geben (*overloading*). In Subklassen können zusätzlich Methoden redefiniert werden (*overriding*) und erhalten so Vorrang vor der ererbten gleichnamigen Methodenimplementierung der entsprechenden (transitiven) Superklasse.

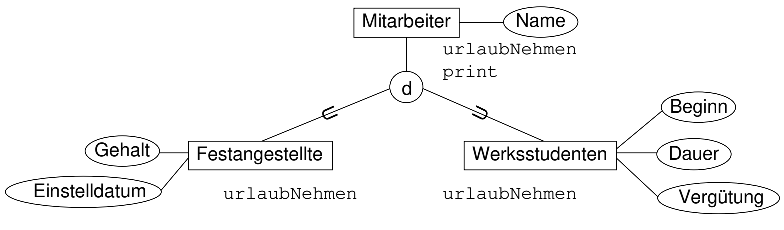

Abbildung 34: Methodenredefinition

In Abhängigkeit von der Klassenzugehörigkeit eines Objektes wird zur Laufzeit eine Implementierung für eine überladene Methode ausgewählt. Dazu muß das Klassenobjekt (Metaobjekt) konsultiert und die Klassenhierarchie travesiert werden. Von der Auswahl einer konkreten Implementierung der Klient einer Methode abstrahieren.

Erweiterbarkeit: Der Benutzer kann neue Typen definieren.

Beziehungen zwischen Klassen: `Subklassenbeziehungen`

Assoziationen zwischen Klassen, Obwohl es auch Ansätze gibt, die ein eigenes Konstrukt für die Dartstellung von Assoziationen vorschlagen [], werden bei den meisten Vorschlägen Assoziationen durch Attribute eines Objekts, die als Attributtyp eine Klasse besitzen, repräsentiert.

Wenn das explizite Löschen von Objekten erlaubt ist, können Probleme mit der referentiellen Integrität auftreten, da noch Referenzen auf das gelöschte Objekt existieren können. Es gibt für diese Situation verschiedenen Techniken, die in Abschnitt 5.5.7 beschrieben werden. Referentielle Integrität.

Hinzu kommen eine Reihe von Anforderungen an die Systemfunktionalität eines objektorientierten Datenbanksystems, die die algorithmische Vollständigkeit, das Persistenzkonzept, die Sekundärspeicherverwaltung den Mehrbenutzerbetrieb (*Concurrency*), die Fehlererholung und die Unterstützung für ad hoc Anfragen betreffen. Diese Aspekte sind jedoch für die Diskussion der objektorientierten Datenmodellierung im Rahmen dieses Abschnitts nicht relevant und werden deshalb hier nicht weiter diskutiert. Einige dieser Aspekte, wie z.B. die Persistenz, werden jedoch im Rahmen der Vorstellung des ODMG-Vorschlags in Abschnitt 5.3 aufgegriffen.

`Wahl des ODMG-Objektmodells Vergleich mit relationalem Modell wichtigste Ideen.`

3.3.2 Überblick über die Konzepte des ODMG-Objektmodells

Das ODMG-Datenmodell basiert auf dem *Core Component* Objektmodell der *Object Management Group* (OMG). Dieser Zusammenhang wird in Abschnitt 5.4 genauer erläutert. Die wichtigsten Konzepte des ODMG-Objektmodells lassen sich wie folgt zusammenfassen:

- Das Datenmodell unterstützt Werte und Objekte.
- *Objekte* sind Instanzen von *Typen* und besitzen eine *Objektidentität*. Ein Objekt ist unmittelbare Instanz genau eines Typs.
- Ein Typ definiert das Verhalten und die möglichen Zustände seiner Instanzen.
- Das Verhalten von Objekten wird durch eine Menge von Operationen beschrieben, die auf Objekten eines Typs ausgeführt werden können. Der Objektzustand ist gekapselt und nur durch die Operationen zugänglich.
- Zusätzlich können bei der Typdefinition *Attribute* für das Objekt und *Beziehungen* zu anderen Objekten festgelegt werden.
- Typen sind in einer *Subtyp-/Supertyphierarchie* organisiert, wobei ein Typ mehrere Supertypen besitzen darf.
- Es gibt eine vordefinierte ODMG-Typhierarchie, die vom Benutzer durch Einführung neuer Subtypen beliebig erweitert werden kann.

- Zu einem Typ kann optional eine *Extension* verwaltet werden, in die Objekte dieses Typs bei ihrer Erzeugung automatisch eingefügt werden.

- Zu einem Typ kann es mehrere Implementierungen geben. Eine Typdefinition zusammen mit einer seiner Implementierungen wird als *Klasse* bezeichnet.

- Es wird ein *orthogonales Persistenzkonzept* unterstützt. Beliebige Objekte können persistent gespeichert werden.

- Das Datenmodell definiert generische, systemweit vorhandene Dienste für Anfragen und Transaktionen.

3.3.3 ODMG-Objekttypen

Im ODMG-Standard wird eine Reihe von Typen vordefiniert, die in einer Subtyphierarchie angeordnet sind (*vordefinierte Typhierarchie*). Diese Typhierarchie besteht aus zwei Subtypbäumen. Neben einem Baum für die benennbaren Objekten (*Denotable_Object*) enthält sie einen weiteren für die ODMG-Datenmodellkonstrukte. Da das ODMG-Objektmodell eine klare Unterscheidung zwischen *Objekten* und *Werten* unterstützt, besteht der Baum der benennbaren Objekte aus einem Teilbaum mit Typen für Objekte und einem mit Typen für Werte[2].

Die Subtypen von *Denotable_Object* umfassen unter anderem die üblichen Basisdatentypen sowie Typen für strukturierte Daten, für verschiedene Arten von Kollektionen, für Iteratoren und für Ausnahmen. Eine ausführlichere Beschreibung der vordefinierten Typhierarchie findet sich in Abschnitt 5.5.1.

Die vordefinierte Typhierarchie kann vom Benutzer durch die Einführung neuer Subtypen erweitert werden. Bei der Typdefinition werden die *Eigenschaften* für die Objekte dieses Typs festgelegt. Das sind die Attribute, die Assoziationen zu anderen Objekttypen und die Signaturen der Methoden für Objekte dieses Typs. Die Einordnung eines Typs in die existierende Typhierarchie geschieht durch die Auswahl eines oder mehrerer Typen der Hierarchie als direkte Supertypen. Von diesen ererbt der neu definierte Typ sämtliche Eigenschaften. Zu einem Typ kann optional eine benannte Menge all seiner Instanzen verwaltet werden. Diese Menge wird als *Extension* des Typs bezeichnet.

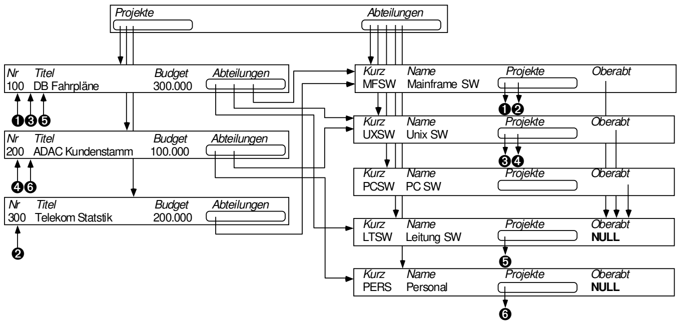

Abbildung 35: Zustand der Firmendatenbank(ODMG-Objektmodell)

[2]Zur Unterscheidung zwischen Objekten und Werten siehe Abschnitt 2.1

Abbildung 35 zeigt einen Zustand der Projektdatenbank im Objektmodell. Für die Projektdatenbank werden die ODMG-Typen *Abteilung* und *Projekt* definiert. Die Datenbank besteht aus einem Aggregat mit zwei benannten Kollektionen *projekte* und *abteilungen* als Komponenten. Diese Kollektionen sind die Extensionen der beiden Objekttypen *Projekt* und *Abteilung*. Die Kollektionen enthalten Referenzen auf die Objekte (Objektidentifikatoren), die als Instanzen des entsprechenden Objekttyps aktuell existieren. Die einzelnen Objekte sind Aggregate, die sich aus den Eigenschaften des Objekts zusammensetzen:

- Die *Attribute* der Objekte können selbst wieder Kollektionen oder Aggregate sein, die sich mit beliebiger Schachtelungstiefe aus den Basistypen zusammensetzen. Im Beispiel treten allerdings nur atomare (flache) Attributwerte auf.

- Die *Assoziationen* sind Komponenten des Aggregats, die eine Referenz oder einer Kollektion von Referenzen auf andere Objekte als Werte besitzen. Dies dient der Darstellung von Beziehungen eines Objekts zu anderen Objekten (s. Abschnitt 3.3.4), einem einzelnen Objekt, wie z.B. einer *Abteilung* zu ihrer *Oberabteilung* bzw. zu einer Kollektion von Objekten wie bei einem Projekt zu den dies Projekt betreuenden Abteilung (*Betreuung*).

- Die *Methoden* der Objekte sind ebenfalls Teil der Datenbank, aber in Abbildung 35 der Übersichtlichkeit halber nicht dargestellt.

Ein Objekttyp ist mit der Schnittstelle eines Abstrakten Datentyps oder einer Modulschnittstelle vergleichbar. Er legt den extern sichtbaren Teil seiner Objekte fest. Die Repräsentation der Objekte und die Realisierung der Methoden werden in der Implementierung des Objekttyps definiert, zu deren Realisierung eine objektorientierte Programmiersprache wie C++ oder Smalltalk verwendet wird (vgl. C++- und Smalltalk-Sprachanbindung, Abschnitt 7.3.1). Zu einem Objekttyp kann es dabei eine oder mehrere Implementierungen geben. Eine Typdefinition zusammen mit einer Implementierung wird im ODMG-Modell als *Klasse* bezeichnet.

3.3.4 Assoziationen im ODMG-Objektmodell

Objektorientierte Modelle verwenden referentielle Indentifikatoren. Im Gegensatz zum relationalen Modell geschieht daher die Darstellung von Beziehungen zu anderen Objekten nicht wertbasiert durch die Verwendung von Fremdschlüsseln (s. Abschnitt 3.2.3), sondern durch Referenzen auf die Objekte. Insbesondere werden bei objektorientierten Ansätzen zu diesem Zweck die eindeutige Objektidentifikatoren verwendet, die vom Zustand des Objekts unabhängig sind.

Die Existenz von Objektidentitäten und die Verwendung von Objektreferenzen zur Repräsentation von Beziehungen zwischen Objekten prägen die Struktur objektorientierter Datenbanken und sind für deren erhebliche Abweichung von der Struktur relationaler Datenbanken, die ein Vergleich der Abbildungen 25 und 35 zeigt, verantwortlich.

Assoziationen werden ähnlich wie Attribute definiert. Es wird ein Bezeichner zum Traversieren der Beziehung und der Typ der assoziierten Objekte festgelegt. Im Gegensatz zu den Wertattributen besitzen diese speziellen Attribute Referenzen auf andere Objekte als Werte. Dabei ist sowohl eine einzelne Referenz als auch eine Kollektion solcher Referenzen möglich.

Zur Darstellung einer binären 1:1-Beziehungen zwischen zwei Objekttypen A und B wird zu einem Objekt des Typs A der Identifikator des assoziierten Objekts vom Typ B angegeben und umgekehrt. Als Beispiel sei hier die Beziehung einer Abteilung zu ihrem Leiter betrachtet: Bei Darstellung beider Richtungen der Beziehung besitzt jede Abteilung eine Referenz auf ihren Leiter und jeder Leiter eine Referenz auf die von ihm geleitete Abteilung.

Bei 1:n-Beziehungen hängt die Modellierung von der betrachteten Richtung ab. Wenn nur eine Richtung einer Beziehung dargestellt wird, verwendet man je nach gewählter Richtung eine einzelne Referenz oder eine Kollektion von Referenzen. So kann man zur Darstellung der Abteilungshierarchie entweder zu jeder Abteilung ihre Oberabteilung oder die Kollektion (Liste) ihrer Unterabteilungen angeben. Zur Modellierung einer bidirektionalen 1:n-Beziehung zwischen dem Objekttyp A und B gibt man entsprechend zu einem Objekt vom Typ A die n Identifikatoren der assoziierten Objekte des Typs B an und zu einem Objekt vom Typ B den Identifikator des assoziierten Objekts vom Typ A.

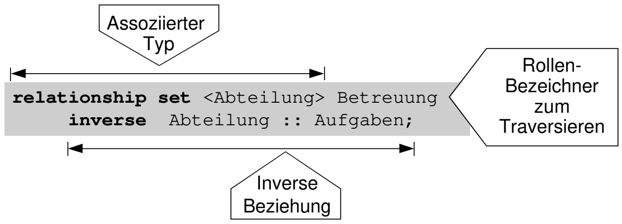

Abbildung 36: Assoziationen im ODMG-Objektmodell

Wie am Beispiel der Beziehung zwischen Projekten und Abteilungen zu sehen ist, wird zur bidirektionalen Modellierung einer n:m-Beziehung in beiden Richtungen eine Kollektion von Objektidentifikatoren benötigt (s. Abbilldung 37). Für jedes Projekt kann man über den Bezeichner *Abteilungen* auf die Kollektion der betreuenden Abteilungen zugreifen. In umgekehrter Richtung kann bei einer Abteilung über den Bezeichner *Projekte* die Kollektion der betreuten Projekte referenziert werden. Wenn nur eine Richtung der Assoziation unterstützt werden soll, z.B. für den Zugriff von den Projekten auf die Abteilungen, genügt es zu jedem Projekt die Identifikatoren der assoziierten Abteilungen in einer Kollektion anzugeben. Der Identifikator einer Abteilung a kann dabei in mehreren Kollektionen auftauchen, wie dies für die Darstellung einer n:m-Beziehung notwendig ist.

Zur Definition von Assoziationen werden im ODMG-Objektmodell spezielle Attribute verwendet, die im Gegensatz zu den Wertattributen Referenzen auf Objekte enthalten. Ein solches Attribut kann eine Objektreferenz oder eine Kollektion solcher Referenzen als Wert besitzen. Um ein Traversieren einer Beziehung in beide Richtungen zu ermöglichen, muß sie in beiden Richtungen definiert werden. Die Beschränkung auf eine der Richtungen der Beziehungen hat Konsequenzen für den Zugriff (Traversieren).

Bei n:m-Beziehungen werden zur Darstellung jeder der beiden Richtungen der Assoziation Kollektionen von Identifikatoren benötigt. Ein Beispiel für eine n:m-Beziehung ist die Assoziation zwischen den Projekten und den Abteilungen, die an ihnen beteiligt sind. Wenn nur eine Richtung dargestellt werden soll, z.B. für den Zugriff von den Projekten auf die Abteilungen, genügt es, zu einer Abteilung die Kollektion der Identifikatoren der beteiligten Objekte anzugeben. Der Identifikator einer Abteilung a kann dabei in mehreren Abteilungen auftauchen, wie dies für die Darstellung einer n:m-Beziehung notwendig ist.

Eine genauere Beschreibung der Definition von Beziehungen findet sich in Abschnitt 5.5.7.

`evtl hier noch etwas zu Subtypbeziehungen`

```
interface Projekt
( extent projekte
  keys Nr)
{ attribute Unsigned Short Nr;
  attribute String Titel;
  attribute Signed Long Budget;
  relationship Set<Abteilung> Betreuung
    inverse Abteilung::Projekte;
};

interface Abteilung
( extent abteilungen
  keys Kurz)
{ attribute String Kurz;
  attribute String Name;
  relationship Set<Projekt> Projekte
    inverse Projekt::Betreuung;
  relationship Abteilung Oberabteilung;
  Boolean neuesProjekt(in Projekt p) raises (...)
};
```

Abbildung 37: ODL-Schemadefinition der Projektdatenbank

3.3.5 Datendefinition im ODMG-Objektmodell

Der ODMG-Standard spezifiziert eine Sprache ODL zur Definition von Schemata, die mit dem ODMG-Objektmodell konform sind. Unter Verwendung dieser Sprache kann die Schemadefinition für die Projektdatenbank wie in Abbildung 37 dargestellt aussehen.

3.3.6 Die Anfragesprache OQL

Mit OQL steht im ODMG-Objektmodell wie im relationalen Modell mit SQL eine mengenorientierte, deskriptive Anfragesprache zur Verfügung. Die Anfragesprache OQL bildet eine Obermenge des Anfragesprachenanteils von SQL. Die wichtigsten Erweiterungen von OQL gegenüber SQL lassen sich wie folgt zusammenfassen:

- OQL berücksichtigt *Werte und Objekte*. Es können Anfragen gegen Kollektionen von Objekten und Werten gestellt werden, und es sind auch Objekte und Werte als Anfrageergebnisse möglich.
- Die Anfragesprache ist nicht so sehr auf eine Kollektionsart fokussiert, wie dies bei SQL mit den Relationen der Fall ist. So sind in OQL z.B. Anfragen auf *verschiedene Arten von Kollektionen* (Mengen, Multimengen, Listen, Felder) möglich.
- OQL weist eine *höhere Sprachorthogonalität* auf. Die Konstrukte der Anfragesprache, deren Kombination in SQL einer Reihe von Einschränkungen unterliegt, sind in OQL frei kombinierbar.
- Das ODMG-Objektmodell ist im Gegensatz zum relationalen Modell nicht auf flache Tupel beschränkt, sondern unterstützt *beliebig tief geschachtelte Strukturen*. Der Zugriff auf Komponenten tiefer geschachtelter Strukturen erfordert *längere Pfadnamen* in

OQL-Anfragen. So wird über den Ausdruck *a.Oberabteilung.Name* auf den Namen der Oberabteilung einer Abteilung a zugegriffen.

- In OQL-Anfragen können neben einem Satz vordefinierter Funktionen beliebige Methoden aufgerufen werden. Dies führt zu einer erheblichen Ausdrucksmächtigkeit der Sprache.

Die folgende OQL-Anfrage ermittelt, wie die entsprechende SQL-Anfrage in Abschnitt 3.2.7, die Titel aller Projekte, an denen die Abteilung *Mainframe SW* arbeitet.

select *p.Titel*
from *projekte p*
where exists *a* **in** *p.Betreuung : a.Name = 'Mainframe SW'*

Wegen der direkten Darstellung der Beziehung zwischen den Projekten und den Abteilungen, von denen es betreut wird, ist hier im Gegensatz zur SQL-Anfrage keine Join-Operation notwendig. Auf die Abteilungen, die das Projekt *p* betreuen, kann einfach über den Ausdruck *p.Betreuung* zugegriffen werden. Dies führt zu einer erheblichen Vereinfachnug der Anfrage.

3.3.7 Aktualisierungsoperationen im ODMG-Objektmodell

In Abschnitt 3.2.8 werden grundlegende Klassen von Modifikationsoperationen auf Datenbanken diskutiert, die in datenmodellspezifischer Form in jedem Datenmodell zu finden sind. Dies sind Operationen zum Einfügen, Löschen und Modifizieren von Daten der Datenbank. Im ODMG-Ansatz treten diese Operationen sowohl für Objekte als auch für Assoziationen zwischen Objekten auf. Die Operationen auf Beziehungsstrukturen sind dabei stets einem konkreten Objekt zugeordnet und manipulieren die Assoziationen, an denen dies Objekt beteiligt ist. Bei bidirektionalen Beziehungen[3] bewirken die Operationen unabhängig von ihrer Zuordnung zu einem konkreten Objekt eine Aktualisierung beider Richtungen der Assoziation. Im ODMG-Objektmodell sind die folgenden Operationen definiert:

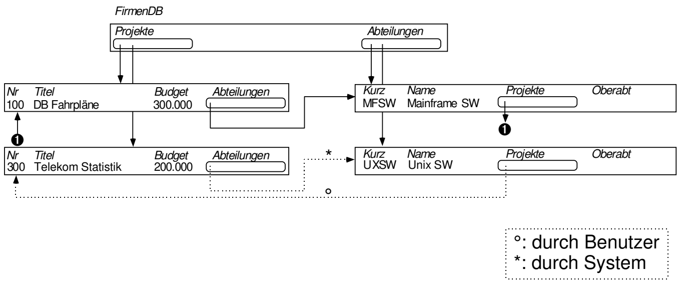

Abbildung 38: Zustand der Firmendatenbank nach einer Aktualisierungsoperation

Einfügen: Die Extensionen der Typen repräsentieren die Massendatenstrukturen im ODMG-Objektmodell, in denen die Objekte des Typs verwaltet werden. Das Einfügen von

[3] Bidirektionale Beziehungen müssen nicht nur in beide Richtungen definiert, sondern auch durch die inverse-Klausel miteinander verbunden sein.

Objekten in Extensionen geschieht automatisch bei der Objekterzeugung. Die Einfügeoperation tritt in diesem Fall also als Teilfunktionalität der Operation zur Objektgenerierung auf.

Beim relationalen Modell ist die Datenbank mit den Relationen auf die persistente Speicherung solcher Massendatenstrukturen beschränkt. Aufgrund des orthogonalen Persistenzkonzepts können beim ODMG-Objektmodell beliebige Objekte persistent gemacht und damit in die Datenbank eingefügt werden.

Zum Einfügen neuer Beziehungen existieren Operationen die ein Objekt mit einem einzelnen Objekt oder einer Menge von Objekten des assoziierten Typs in Beziehung setzen.

Löschen: Jedes Objekt besitzt eine Methode *delete*, die vom Supertyp *Object* aller Objekttypen ererbt wird. Diese Methode löscht das Objekt, für das sie aufgerufen wird, und macht noch existierende Referenzen auf das Objekt ungültig.

Assoziationen zwischen Objekten können ebenfalls gelöscht werden. Für ein Objekt ist sowohl das Lösen der Beziehung zu einzelnen Objekten (*selektives* Löschen) als auch der Beziehungen zu allen assoziierten Objekten möglich (*vollständiges* Löschen).

Modifizieren: Anders als im relationalen Datenmodell gibt es hier keine allgemeine Funktion *update* für die Änderung beliebiger Attribute. Zum Ändern des Objektzustands existiert zum einen für jedes Attribut eines Objekttyps eine vordefinierte Methode *setValue*, die redefiniert werden kann. Zum anderen kann der Objektzustand durch benutzerdefinierte Methoden verändert werden.

Ausgehend von einem konkreten Objekt kann eine Beziehung modifiziert werden, wenn dem Objekt nicht nur ein einzelnes Objekt, sondern eine Menge von Objekten zugeordnet werden kann (s. Abschnitt 3.3.4). Die Operationen zum Erweitern von Beziehungen, die im Objektmodell unterstützt werden, und auch das im vorigen Punkt beschriebene selektive Löschen bilden Modifikationsoperation auf einer solchen Beziehung.

```
Ref<Abteilung> aOID = Abteilungen.select_element(
  “select a in Abteilungen where a.Name = \“Unix SW\” ”);
Ref<Project> pOID = Projekte.select_element(
  “select p in Projekte where p.Titel = \“Telekom Statistik\” ”);
aOID.Projekte.insert_element(pOID);
(* impliziert / äquivalent zu:
pOID.Abteilungen.insert_element(aOID);*)
```

3.4 Überblick über die prärelationale Ära

Die erste Definition des Netzwerkdatenmodells (NDM) wurden etwa zeitgleich mit den Arbeiten über das relationale Modell vorgelegt [](DBH). Da sich das Netzwerkdatenmodell jedoch früher in der Praxis durchgesetzt hat, wird es hier zusammen mit dem hierarchischen Modell unter dem Begriff der *prärelationalen Ära* zusammengefaßt. Das hierarchische Modell unterscheidet sich in seiner Entwicklungsgeschichte deutlich von den anderen Modellen, da es gleichzeitig mit der Implementierung eines konkreten Systems entwickelt worden ist. Es fand also nicht wie bei den anderen Modellen die Entwicklung eines konzeptuellen Datenmodells statt, daß als Grundlage für spätere Systemimplementierungen benutzt wird. Das auf dem hierarchischen Modell basierende System IDS fand eine schnelle Verbreitung. Neben dem Multi-Access Retrieval System (MARS VI) der Control Data Corporation und dem System-2000 von MRI gibt es eine Reihe weiterer System, die nach dem hierarchischen Modell implementiert sind.

Zur Realisierung neuer Systeme werden die beiden prärelationalen Datenbankmodelle kaum mehr eingesetzt, sodaß ihnen in zunehmendem Maße nur noch historische Bedeutung zukommt. Es gibt jedoch in der Praxis eine große Anzahl sehr umfangreicher Implementierungen, die auf diesen Modellen basieren und erhebliche Investitionen darstellen, was noch für einige Zeit für eine hohe Praxisrelevanz dieser Modelle sorgt.

Die restlichen Abschnitte dieses Teils des Kapitels konzentrieren sich auf das Netzwerkdatenmodell beginnend mit einem zusammenfassenden Überblick über die wichtigsten Konzepte in Abschnitt 3.4.1. Das zentrale Konzept des Netzwerkdatenmodells zur Datenstrukturierung, die *DBTG-Mengen*, werden im darauffolgenden Abschnitt beschrieben. Analog zu den übrigen Datenmodellen werden in den weiteren Abschnitten die Darstellung von Assoziationen und die Datendefinition im DBTG-Modell sowie die Unterstützung für Anfrage- und Aktualisierungsoperationen behandelt. Der letzte Abschnitt 3.4.7 befaßt sich mit dem hierarchischen Datenmodell. Dieses ist dem Netzwerkmodell sehr ähnlich, wenn auch mit zusätzlichen Einschränkungen versehen. Seine Beschreibung ist deshalb sehr kurz gehalten und beschränkt sich auf die wichtigsten Unterschiede und Einschränkungen gegenüber dem Netzwerkdatenmodell.

3.4.1 Überblick über die Konzepte des Netzwerkdatenmodells

Die Umstände seiner Entstehung prägen entscheidend die Natur des Netzwerkdatenmodells (NDM). Die Definition des auch als DBTG-Modell (s.u.) bezeichneten NDM ist im Rahmen einer Gruppenaktivität entstanden, und beruht nicht, wie das relationale Modell, im wesentlichen auf der Arbeit eines Einzelnen. In die NDM-Definition fanden zahlreiche Einzelvorschläge Eingang, deren Integration nicht immer als gelungen bezeichnet werden kann. Als Folge davon sind an der NDM-Definition im Laufe der Zeit zahlreiche wesentliche Änderungen vorgenommen worden. Das ursprüngliche Modell wurde 1971 von der *Database Task Group* (*DBTG*) des *CODASYL Systems Committee* vorgeschlagen. Zahlreiche Änderungen und Verbesserungen fanden in die Berichte aus den Jahren 1978 und 1981 eingang.

Syntaktisch ist die NDM-Definition aufgrund ihrer Erstellung durch die *Database Task Group* (*DBTG*) des *CODASYL Systems Committee* stark von COBOL beeinflußt. Der „wortreiche“ Stil dieser Sprache mit seinen vielen Schlüsselworten, Klauseln, und Phrasen hat die bei der NDM-Definition vorherrschende Tendenz, möglichst viele Einzelanforderungen zu befriedigen, sicher noch unterstützt.

Wie das ODMG-Objektmodell beruht auch das NDM auf referentieller Identifikation. Es gibt auf konzeptueller Ebene eine gewisse Ähnlichkeit zwischen den beiden Modellen, aber auch wichtige Unterschiede und Einschränkungen. So fehlt im DBTG-Modell eine deskriptive, mengenorientierte Anfragesprache, wodurch eine aufwendige prozedurale, satzorientierte Verarbeitung von Anfragen notwendig ist. Ein weiterer Punkt sind die zahlreichen implementierungssbedingten Einschränkungen, die die DBTG bei ihrer Modelldefinition eingeführt hat. Diese bedingen sowohl im Bereich der Strukturen als auch der Operationen eine unnötige Verkomplizierung. Ein Beispiel dafür ist die indirekte Darstellung von binären n:m-Beziehungen. Weiterhin fehlt im DBTG-Modell das Konzept der Objektidentität und der orthogonalen Persistenz. Die Sprachanbindung geschieht durch die Verwendung eines Kommunikationsbereichs und weist durch die implizite Manipulation von Aktualitätszeigern (s. Abschnnitt 3.4.5) eine oft schwer durchschaubare Semantik auf.

Im folgenden werden die wichtigsten Konzepte des Netzwerkmodells zusammengefaßt:

- Eine Netzwerkdatenbank besteht aus einer Menge von Rekords, die durch Verweise (Zeiger) miteinander verbunden sind.
- Die Rekords sind mit den Tupeln des relationalen Datenmodells vergleichbar. Als Attributwerte sind jedoch auch strukturierte Werte und *Wiederholgruppen* zur Darstellung mengenwertiger Attribute möglich.
- Die Rekords sind in sogenannten *DBTG-Mengen* organisiert. Eine DBTG-Menge enthält ein ausgezeichnetes Element, das als *Eigner* (*owner*) der Menge bezeichnet wird, und beliebig viele *Mitglieder* (*member*).
- Die Definition eines *DBTG-Mengentyps* legt einen Rekordtyp A für den Eigner der Menge und einen Rekordtyp B für die Mitglieder der DBTG-Menge fest.
- In allen DBTG-Mengen, die den gleichen DBTG-Mengentyp besitzen, kann ein Rekord nur maximal einmal vorkommen. Ein Rekord kann jedoch in DBTG-Mengen verschiedenen Typs vertreten sein.
- DBTG-Mengen werden sowohl zur Darstellung von Objektmengen als auch zur Assoziation von Datenobjekten benutzt.

3.4.2 DBTG-Mengen

Eine DBTG-Menge ist ein Aggregat mit zwei Komponenten. Die erste Komponente enthält einen Verweis auf den Eigner (*owner*) der DBTG-Menge und die zweite Komponente ist eine Kollektion, die eine beliebige Anzahl von Verweisen auf Mitglieder (*member*) enthält. DBTG-Mengen unterscheiden sich in folgenden Punkten von mathematischen Mengen:

- Eine DBTG-Menge ist nicht homogen. Sie enthält Elemente von zwei verschiedenen Typen.
- Es gibt ein ausgezeichnetes Element, den Eigner der DBTG-Menge.
- Die Mitglieder einer DBTG-Menge sind geordnet.

Ein DBTG-Mengentyp ist ein benannter Aggregattyp mit zwei Komponenten. Bei der Definition eines DBTG-Mengentyps werden ein Rekordtyp für den Eigner und ein Rekordtyp für die Mitglieder festgelegt. Zu einem Mengentyp kann es beliebig viele DBTG-Mengen als konkrete Ausprägungen geben. Neben dieser allgemeinen Form der DBTG-Mengen gibt es noch spezielle DBTG-Mengen, bei denen das *System*[4] als Eigner gewählt wird (*system owned sets*). Von dieser Art der DBTG-Mengen gibt es jeweils nur eine Ausprägung. Sie werden dazu verwendet, alle Rekords eines bestimmten Typs in einer Menge zu verwalten. Insbesondere ermöglicht die Definition solcher Mengen für einen Rekordtyp die Angabe einer Sortierordnung auf den Rekords dieses Typs und liefert außerdem Einstiegspunkte für Anfragen an die Datenbank.

[4] Mit *System* ist hier die DBMS Software gemeint.

Die DBTG-Mengen assoziieren ein Datenobjekt (den Eigner) eines Rekordtyp A mit einer Menge von Datenobjekten (den Mitgliedern) eines Rekordtyps B. Dabei gelten die folgenden Einschränkungen:

1. Ein Rekord kann innerhalb *eines* DBTG-Mengentyps nur einmal als Mitglied oder Eigner auftreten. Derselbe Rekord kann jedoch in Ausprägungen verschiedener Mengentypen vertreten sein, und zwar sowohl als Eigner als auch als Mitglied.

2. Ein Rekordtyp kann nicht zugleich Eigner- und Mitgliedstyp derselben DBTG-Menge sein. Dies erschwert insbesondere die Definition rekursiver Beziehungen.

Diese Einschränkungen haben weitreichende Konsequenzen für die Datenmodellierung und Datenmanipulation im NDM. Zum einen kann eine DBTG-Menge durch ihren Eigner und auch durch eines ihrer Mitglieder identifiziert werden, was beim Navigieren durch eine Netzwerkdatenbank ausgenutzt wird (s.Abschnitt 3.4.5). Zum anderen bedingen die Einschränkungen die Einführung spezieller Rekords, der sog. *Verbindungsrekords* bei der Darstellung von rekursiven und von n:m-Beziehungen (s. Abschnitt 3.4.3).

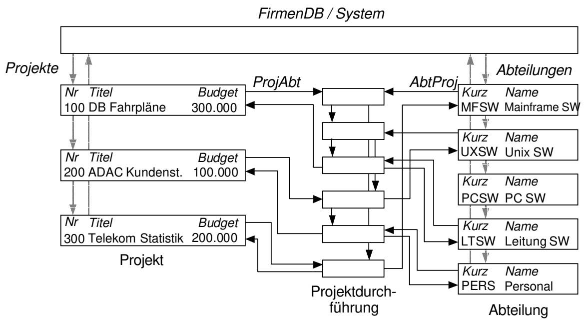

Abbildung 39: Zustand der Firmendatenbank (DBTG-Netzwerkmodell)

Eine Netzwerkdatenbank besteht aus Ausprägungen der verschiedenen im Schema definierten Rekordtypen, die in DBTG-Mengen organisiert sind. Sie ist ein benanntes Aggregat (Name der Datenbank) mit einzelnen DBTG-Mengen, den *system owned sets* [5], und Kollektionen von DBTG-Mengen als Komponenten. Abbildung 39 zeigt einen Zustand der Projektdatenbank bei einer Realisierung als Netzwerkdatenbank. Die Beispieldatenbank enthält die *system owned sets Projekte* und *Abteilungen* und die DBTG-Mengen *ProjAbt*, *AbtProj*, *Oberabteilung* und *Unterabteilungen*.

Wie oben bereits erwähnt sind die einzelnen DBTG-Mengen selbst wiederum Aggregate mit zwei Komponenten, einer Referenz auf den Eigner der Menge und einer Kollektion von Referenzen auf die Mitglieder. Die Datenbank ist also ein Aggregat mit Aggregaten und Kollektionen von Aggregaten als Komponenten. Die DBTG-Mengen *PrpjAbt* und *AbtProj* besitzen z.B. Elemente vom Typ *Projekt* bzw. Abteilung als Eigner und Elemente vom Typ *Projektdurchführung* als Mitglieder. Sie dienen der Darstellung der n:m-Beziehung zwischen Projekten und Abteilungen und werden im Nächsten Abschnitt genauere behandelt.

Die Mitglieder einer DBTG-Menge sind geordnet und in der Regel durch eine listenähnliche Zeigerstruktur miteinander verknüpft. Die zweite Komponente der DBTG-Menge enthält

[5]Von den *system owned sets* gibt es jeweils nur eine Ausprägung

daher eine Referenz auf das erste Mitglied, das wiederum über einen Verweis mit dem zweiten Mitglied der DBTG-Menge verknüpft ist. Das letzte Element besitzt eine Referenz auf den Eigner der Menge. Man kann eine DBTG-Menge also auch als ein Aggregat aus zwei Referenzen betrachten, eine auf den Eigner und eine auf das erste Mitglied.

3.4.3 Assoziationen im Netzwerkdatenmodell

Assoziationen zwischen Datenobjekten werden im Netzwerkdatenmodell durch die Definition von geeigneten DBTG-Mengen modelliert. Das Netzwerkdatenmodell ist auf die Darstellung von binären 1:n-Beziehungen ausgerichtet, ohne jedoch wie das hierarchische Datenmodell die Darstellung anderer Beziehungen unnötig zu erschweren. Die 1:n-Beziehungen werden durch die Zuordnung einer Menge von Rekords zu einem einzelnen Rekord (dem Eigner), wie dies in den DBTG-Mengen geschieht, optimal unterstützt. Binäre 1:1-Beziehungen lassen sich ebenfalls direkt durch DBTG-Mengen modellieren, die neben dem Eigner jeweils genau ein Mitglied besitzen. Die Darstellung der 1:1- und 1:n-Beziehungen entspricht also im wesentlichen der im ODMG-Objektmodell, nur daß die Definition einer Beziehung beim objektorientierten Ansatz als Teil des Typs eines Objekts geschieht (s. Abschnitt 3.3.3), während im NDM ein eigenes Konstrukt, die DBTG-Mengen, verwendet wird.

Zur direkten Darstellung von n:m-Beziehungen zwischen zwei Objekttypen A und B durch einen DBTG-Mengetyp AB mit Eignertyp A und Mitgliedstyp B wäre es notwendig, daß Rekords vom Typ B in mehr als einer der Mengen vom Typ AB als Mitglied auftreten könnten (s. Abbildung 39). Im ODMG-Objektmodell ist diese Art der Darstellung von n:m-Beziehungen möglich (vgl. Abschnitt 3.3.4), im Netzwerkmodell wird sie durch die erste der im vorigen Abschnitt genannten Einschränkungen verhindert.

Zur Darstellung einer binären n:m-Beziehungen werden deshalb sogenannte *Verbindungsrekords*, für die ein zusätzlicher Rekordtyp definiert werden muß und eine zweite DBTG-Menge benötigt. Im Fall der Beziehung zwischen Projekten und Abteilungen wird für die Verbindungsrekords der Rekordtyp *Projektdurchfuehrung* eingeführt und die beiden DBTG-Mengen *AbtProj* und *ProjAbt* definiert (s. Abbildung 39). Um die Beziehung zwischen einem Projekt p und einer Abteilung a darzustellen, wird eine Ausprägung pd des Typs *Projektdurchfuehrung* als Mitglied sowohl in eine Menge vom Typ *ProjAbt* mit dem Projekt p als Eigner als auch in eine Menge vom Typ *AbtProj* mit der Abteilung a als Eigner eingefügt.

Es exitiert für jedes Paar Abteilung/Projekt ein eigenes Verbindungsrekord, so daß sich die Betreuung eines Projekts durch mehrere Abteilungen und die Beteiligung einer Abteilung an mehreren Projekten ohne Verletzung der Einschränkungen des Modells darstellen lassen. Abbildung xx zeigt diesen Ansatz schematisch für die Objekttypen A und B. Mithilfe der Verbindungsrekords lassen sich auch n-äre Beziehungen (n>2) modellieren, da ein Verbindungsrekord in verschiedenen DBTG-Mengen (verschiedener Mengentyp) eingefügt werden kann. Dies ermöglicht die Verknüpfung von mehr als zwei Rekordtypen in einer Beziehung. Eine einfache Erweiterung auf Assoziationen mit Attributen ist ebenfalls möglich, indem das Verbindungsrekord entsprechende Attribute erhält.

Die zweite Einschränkung für die DBTG-Mengen führt dazu, daß rekursive Beziehungen nicht direkt dargestellt werden können, da dies denselben Rekordtyp für den Eigner und die Mitglieder einer DBTG-Menge erfordert. Ein Beispiel dafür ist die Abteilungshierarchie. Analog zur Modellierung im ODMG-Objektmodell bietet es sich an, DBTG-Mengen mit einer Abteilung als Eigner und ihren Unterabteilungen als Mitgliedern zu definieren. Aufgrund der Einschränkung ist aber auch hier die Einführung eines zusätzlichen Rekordtyps notwendig. Für die Projektdatenbank wird dafür der Rekordtyp *Abteilungshierarchie* definiert. Man benötigt dann zwei DBTG-Mengen: eine Menge *Oberabteilung*, mit einer Abteilung als Eigner und einem einzelnen Rekord vom Typ *Abteilungshierarchie* als Mitglied, daß die Verbindung zur Oberabteilung herstellt und einer DBTG-Menge *Unterabteilungen* ebenfalls mit einer Abteilung als Eigner und einer Kollektion von Rekords des Typs *Abteilungshierarchie* als Mitglieder, die die Verbindung zu den Unterabteilungen der Abteilung herstellt. Wenn eine Abteilung *a1* Oberabteilung einer Abteilung *a2* ist, wird dies durch ein Verbindungsrekord *ah* vom Typ *Abteilungshierarchie* dargestellt, das sowohl in eine Menge vom Typ *Unterabteilungen* als auch in eine DBTG-Menge vom Typ *Oberabteilung* als Mitglied eingefügt wird.

3.4.4 Datendefinition im Netzwerkdatenmodell

Abbildung 40 zeigt eine mögliche Definition des Schemas für die Projektdatenbank. Die Notation orientiert sich an der Datendefinitionsprache des netzwerkmodellbasierten Datenbanksystems IDMS (*Integrated Database Management System*, s. z.B. []), das auf dem DBTG-Vorschlag beruht. Da das Gewicht dieses Kapitels mehr auf einem prinzipiellen Verständnis der Konzepte liegt, werden die syntaktischen Einzelheiten der Datendefinition hier nicht näher erläutert.

Bei der Schemadefinition werden zunächst die Rekordtypen *Projekt* und *Abteilung* mit den entsprechenden Attributen zur Darstellung der Objekte der Anwendung definiert. Die Rekordtypen *Projektdurchfuehrung* und *Abteilungshierarchie* sind die Typen für die Verbindungsrekords, die für die Modellierung der Beziehungen benötigt werden. Sie besitzen keine Attribute. Der Definition der Rekordtypen folgt die Definition der DBTG-Mengentypen für dieses Beispiel. Dies sind die Typen *ProjAbt* und *AbtProj* zur Darstellung der n:m-Beziehung zwischen Projekten und Abteilungen sowie die Typen *Unterabteilungen* und *Oberabteilung*, die, wie bereits in Abschnitt 3.4.3 erläutert, zur Modellierung der Abteilungshierarchie benötigt werden. Die letzen beiden DBTG-Mengentypen der Schemadefintion, *Projekte* und *Abteilungen* legen *system owned sets* fest. Sie dienen der Verwaltung aller in der Datenbank vorhandenen Projekte bzw. Abteilungen.

Ein wichtige Rolle spielen im Netzwerkmodell die unterschiedlichen Optionen, die bei der Datendefinition für Rekordtypen und DBTG-Mengentypen festgelegt werden können. Diese Angaben umfassen Sortierordnungen, wie z.B. **order is sorted** und **ascending key is**, Eindeutigkeitsbedingungen (**duplicates not allowed**) und Optionen für das Vorgehen beim Einfügen und Löschen von Datenobjekten, wie z.B. **optional** und **automatic** und haben unter anderem Auswirkungen auf den Effekt bestimmter Aktualisierungsoperationen auf der Datenbank. Außerdem ist es auch möglich bei der Datendefinition die Anordnung von Datenobjekten im Speicher zu beeinflussen (**location mode**)[6]. Einige der möglichen Optionen sind bei der Schemadefinition in Abbildung 40 zu sehen. Auf die Optionen für das Einfügen und Löschen wird in Abschnitt 3.4.6 kurz eingegangen, die übrigen werden hier nicht näher behandelt. Der interessierte Leser sei auf entsprechende Darstellungen z.B. in [**?**] verwiesen.

Aus der Schemadefinition ergibt sich das in Abbildung **??** dargestellte Schemadiagramm. Das Datenbankschema besteht aus den Aggregattypen *Projekt*, *Abteilung*, *Projektdurchfuehrung* und *Abteilungshierarchie*. Die beiden letztgenannten Typen besitzen dabei keine Komponenten. Sie definieren reine Verbindungsrekords. Die Aggregattypen *Projekt* und

[6] Dies widerspricht der in Abschnitt 1.3.1 beschriebenen Forderung nach Datenunabhängigkeit.

schema name is *ProjektDB*.

record name is *Projekt*.
location mode is calc using *Nr*
  duplicates not allowed.
  02 Nr **pic** *X(5)*.
  02 Titel **pic** *X(30)*.
  02 Budget **pic** *9999999.99*.

record name is *Abteilung*.
location mode is calc using *Kurz*
  duplicates not allowed.
  02 Kurz **pic** *X(4)*.
  02 Name **pic** *X(30)*.

record name is *Projektdurchfuehrung*.
location mode is via *ProjAbt* **set**.

record name is *Abteilungshierarchie*.
location mode is via *Oberabteilung* **set**.

set name is *ProjAbt*.
order is next.
owner is *Projekt*.
member is *Projektdurchfuehrung* **optional manual**.

set name is *AbtProj*.
order is next.
owner is *Abteilung*.
member is *Projektdurchfuehrung* **optional manual**.

set name is *Unterabteilungen*.
order is next.
owner is *Abteilung*.
member is *Abteilungshierarchie* **optional manual.**

set name is *Oberabteilung*.
order is next.
owner is *Abteilung*.
member is *Abteilungshierarchie* **optional manual.**

set name is *Projekte*.
order is sorted.
owner is system.
member is *Projekt* **mandatory automatic**
  ascending key is *Nr*.

set name is *Abteilungen*.
order is sorted.
owner is system.
member is *Abteilung* **mandatory automatic**
  ascending key is *Kurz*.

Abbildung 40: Schemadefinition der Projektdatenbank im DBTG-Netzwerkmodell

Projektdurchfuehrung, sowie *Abteilung* und *Projektdurchfuehrung* sind durch mengenwertige Referenzen miteinander verbunden. Diese Struktur wird durch die DBTG-Mengentypen *ProjAbt* und *AbtProj* festgelegt. Weiterhin ist der Typ *Abteilung* über zwei mengenwertige Referenzen mit dem Typ *Abteilugshierarchie* verbunden. Diese Beziehungen werden durch die DBTG-Mengentypen *Unterabteilungen* und *Oberabteilung* spezifiziert. Von der Anwendung her sollte die Beziehung *Oberabteilung* eigentlich einwertig sein. Diese Unterscheidungsmöglichkeit wird jedoch durch die Konstrukte des Netzwerkmodells nicht unterstützt.

Als Eigner der *system owned sets* spielt *System* eine besondere Rolle im Schema einer Netzwerkdatenbank. Es wird bei der Schemadefinition wie ein vordefinierter Rekordtyp verwendet, der allerdings nur als Eigner von DBTG-Mengen auftreten kann. In Abbildung **??** wird dieser Sonderrolle durch die spezielle Darstellung Rechnung getragen. *System* ist durch mengenwertige Referenzen mit den Typen *Projekt* und *Abteilung* verbunden.

3.4.5 DBTG-Anfrageoperationen

Im Gegensatz zum relationalen Modell und zum ODMG-Objektmodell mit ihren deskriptiven, mengenorientierten Anfragesprachen SQL und OQL besitzt das Netzwerdatenbankmodell eine satzorientierte, prozedurale Anfragesprache.

Zur Formulierung von Anfragen (und für die Datenmanipulation) steht ein Satz von Kommandos zur Verfügung, die in eine Gastsprache (*host language*) eingebettet werden. Als Gastsprache sind dabei verschiedene Programmiersprachen möglich. Im folgenden wird die Programmiersprache Pascal als Gastsprache angenommen. Die Anfrageoperationen unterstützen das Auffinden einzelner Rekords (satzorientierte Sprache), die Iteration über Ergebnismengen geschieht explizit durch die Programmierung von Schleifen in der Gastsprache.

Zur Kommunikation zwischen der Anwendung und dem Datenbanksystem besitzt jedes Anwedungsprogramm (*run unit*) eine *user work area* (*UWA*). Dies ist ein Bereich mit Puffer- und Zeigervariablen, auf die sowohl das Datenbanksystem als auch das Anwendungsprogramm zugreifen können. Im einzelnen enthält die UWA eines Anwendungsprogramms die folgenden Strukturen:

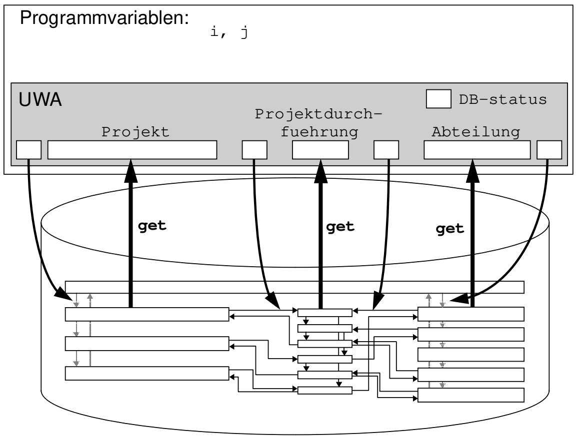

Abbildung 41: Kommunikation zwischen Anwendung und DBMS über UWA

Kommunikationsrekords: Die UWA enthält zu jeden im Schema definierten Rekordtyp einen *Kommunikationsrekord*. Über diese Rekords werden Daten zwischen dem Anwendungsprogramm und dem Datenbanksystem ausgetauscht. Der Datenfluß ist dabei in beiden Richtungen möglich. So kann das Anwendungsprogramm über die Kommunikationsrekords Datenwerte für Anfragen spezifizieren; zugleich stellt das Datenbanksystem aber auch die Elemente des Anfrageergebnisses nacheinander in diesen Rekords für das Anwendungsprogramm bereit.

Bei den meisten DBTG-Systemen wird für die Kommunikationsrekords derselbe Bezeichner wie für den zugehörigen Typ verwendet. Dies führt zu einer unschönen Vermischung unterschiedlicher Konzepte. In den Beispielen in diesem Buch werden deshalb für Kommunikationsrekords und Typen zwar der gleiche Bezeichner verwendet, zur Unterscheidung jedoch für Typen mit großem Anfangsbuchstaben und für die Kommunikationsrekods mit kleinem.

Aktualitätszeiger: Neben den Kommunikationsrekords enthält die *UWA* eine Reihe von *Aktualitätszeigern*, die auf ausgezeichnete Rekords verweisen. Die Aufgabe der Aktualitätszeiger ist vergleichbar mit der von Hilfszeigern beim Durchlaufen einer Liste in Programmiersprachen. Die Aktualitätszeiger dienen der Kennzeichnung der beim *Navigieren* in der Netzwerkdatenbank erreichten Position, die als Ausgangspunkt für Folgeoperationen benutzt wird.

In der *UWA* wird zu jedem Rekordtyp des Schemas ein Aktualitätszeiger verwaltet, der auf das zuletzt zugegriffene Rekord dieses Typs verweist. Weiterhin gibt es zu jedem DBTG-Mengentyp, der im Schema definiert ist, einen Aktualitätszeiger. Dieser verweist auf den Eigner oder eines der Mitglieder einer DBTG-Menge, wodurch diese Menge als die aktuelle dieses Typs ausgezeichnet wird[7]. Zusätzlich gibt es einen Aktualitätszeiger des Programms, der zu jedem Zeitpunkt genau ein Rekord der gesamten Datenbank als das aktuelle des Programms auszeichnet. Dieses Rekord spielt eine wichtige Rolle bei den Aktualisierungsoperationen und wird im folgenden als *aktuelles* Rekord bezeichnet.

Die Aktualitätszeiger werden nicht explizit manipuliert, sondern implizit durch die weiter unten beschriebenen Suchoperationen (**find**) umgesetzt. Sie sind von zentraler Bedeutung für die Navigation durch die DBTG-Mengen der Datenbank. Die meisten Operationen der DBTG-DML hängen von der aktuellen Position der Aktualitätszeiger ab und/oder verändern diese.

Statusvariablen: Über die Statusvariablen, die ebenfalls Teil der UWA sind, kann das Datenbanksystem dem Anwendungsprogramm Statusinformation übermitteln. Dazu gehört unter anderem die Information, ob die letzte Datenbankoperation erfogreich abgeschlossen wurde. In den folgenden Beispielen wird für diesen Zweck die Variable *DB-Status* benutzt, die für den Fall eines erfolgreichen Abschlußes der letzten Operation den Wert 0 enthält. Wie in Abbildung 42 zu sehen ist, kann diese Variable zur Formulierung der Abbruchbedingung bei der Iteration über die Elemente eines Anfrageergenisses verwendet werden.

Anfragen gegen Netzwerkdatenbanken sind geprägt durch den prozeduralen, satzorientierten Charakter der Anfragesprache und die explizite Benutzung des *UWA* zum Austausch von

[7]Aufgrund der in Abschnitt 3.4.2 beschriebenen Einschränkungen kann eine DBTG-Menge durch ihren Eigner und auch durch eines ihrer Mitglieder identifiziert werden.

```
abteilung.Name := 'Mainframe SW'
find any Abteilung using Name

find first Projektdurchfuehrung within AbtProj
  while DB-status = 0 do
  begin
    find owner within ProjAbt
    get projekt
    print(projekt.Titel)
    find next Projektdurchfuehrung within AbtProj
  end;
```

Abbildung 42: DBTG-Beispielanfrage

Daten und Statusinformation. Eine solche Anfrage setzt sich konzeptuell aus zwei Schritten zusammen:

1. Der erste Schritt dient dem *Auffinden der Information*. Dazu werden mit einer oder mehreren find-Operationen die Aktualitätszeiger geeignet positioniert. Ziel dieser Positionierungen ist es, ein Element des Anfrageergebnisses (oder einen Kandidaten dafür) zum aktuellen Rekord zu machen.

   Für diese Aufgaben stehen verschiedene Varianten der find-Operation zur Verfügung, die in zwei Gruppen eingeteilt werden können:

   - Die Operationen der ersten Gruppe dienen zum Suchen innerhalb aller Ausprägungen eines Rekordtyps (**find any**, **find duplicate**).
   - Die Operationen der zweiten Gruppe unterstützen das Navigieren innerhalb von DBTG-Mengen. Zu dieser Gruppe gehören u.a. Operationen für einen Zugriff auf den Eigner (**find owner within**) und für den sequentiellen Zugriff auf die Mitglieder einer DBTG-Menge (**find first within**).

   Eine find-Operation macht ein neues Rekord zum aktuellen Rekord, d.h. der Aktualitätszeiger des Programms wird umgesetzt. Zugleich wird der Aktualitätszeiger des zugehörigen Rekordtyps auf dieses Rekord gesetzt und alle DBTG-Mengen, die dieses Rekord enthalten, werden zur aktuellen Menge ihres Mengentyps. Die Aktualitätszeiger dieser DBTG-Mengentypen werden nämlich auch auf das neue aktuelle Rekord gesetzt.

2. Der zweite Schritt besteht im *Kopieren des Inhalts des aktuellen Rekords in das entsprechende Kommunikationsrekord*, wo er dem Anwendungsprogramm zur Verfügung steht. Zu diesem Zweck wird die get-Operation benutzt.

Abbildung 42 zeigt den typischen Aufbau einer Anfrage an eine Netzwerkdatenbank. Als Beispiel wird die gleiche Anfrage wie bei der Betrachtung von SQL in Abschnitt 3.2.7 und von OQL in Abschnitt 3.3.6 verwendet: Es werden die Titel der Projekte, an denen die Abteilung für die Mainframe Software arbeitet, bestimmt.

Zum Auffinden eines einzelnen Ergebniselements werden eine oder mehrere find-Operationen benutzt, die den Aktualitätszeiger des Programms auf das Ergebniselement positionieren, gefolgt von einer get-Operation, die dieses Element in das Kommunikationsrekord kopiert. Zur

Iteration über die Menge der Ergebniselemente muß in der Gastsprache eine Schleife programmiert werden. Die erste find-Operation der Beispielanfrage lokalisiert die Abteilung mit dem Namen, der dem Datenbanksystem im Kommunikationsrekord *abteilung* zur Verfügung gestellt wird. Zugleich macht diese find-Operation die DBTG-Menge vom Typ *AbtProj*, in der die lokalisierte Abteilung Eigner ist, zur aktuellen ihres Typs. Die zweite find-Operation lokalisiert das erste Mitglied dieser DBTG-Menge. In der while-Schleife werden nacheinander die weiteren Mitglieder dieser Menge abgearbeitet.

Charakteristisch für DBTG-Anfragen ist auch die Navigation in DBTG-Mengen. In diesem Beispiel wird vom Eigner einer DBTG-Menge vom Typ *AbtProj* zu einem der Mitglieder dieser DBTG-Menge gegangen[8]. Mit der Operation **find owner within** *ProjAbt* wird in die DBTG-Menge vom Typ *ProjAbt* gewechselt, in der die betrachtete *Projektdurchfuehrung* Mitglied ist, und auf deren Eigner zugegriffen. Dieser Eigner ist ein Projekt, an dem die betrachtete Abteilung arbeitet, d.h. eines der gesuchten Ergebniselemente. Die Navigationsschritte erfüllen den gleichen Zweck wie die Join-Operation bei SQL und das Verfolgen von definierten Assoziationen bei OQL.

3.4.6 Aktualisierungsoperationen im Netzwerkmodell

Die im vorigen Abschnitt beschriebenen Operationen **find** und **get** dienen dem Auffinden von Information in einer Netzwerkdatenbank. Dieser Vorgang spielt auch bei den Aktualisierungsoperationen eine Rolle, da z.B. ein Datenobjekt lokalisiert werden muß, bevor es gelöscht oder geändert werden kann. Analog zu den Operationen auf den Objekten und Assoziationen beim ODMG-Objektmodell (vgl. Abschnitt 3.3.7), wird beim Netzwerkmodell zwischen Aktualisierungsoperationen auf Rekordebene und auf den DBTG-Mengen unterschieden. Zur Aktualisierung von Datenobjekten (Rekords) gibt es wiederum Operationen zum Einfügen, Löschen und Modifizieren:

Einfügen: Das einzufügende Rekord wird im entsprechenden Kommunikationsrekord zusammengestellt und durch eine store-Operation in die Datenbank eingefügt.

Löschen: Zum Löschen eines Rekords wird dieses zunächst in der Datenbank aufgesucht und zum aktuellen Rekord gemacht. Mit der dafür vorgesehenen Operation **erase** wird das aktuelle Rekord aus der Datenbank entfernt.

Modifizieren: Wie beim Löschen muß auch beim Modifizieren das betroffene Rekord zunächst zum aktuellen Rekord gemacht werden. In beiden Fällen wird dafür eine spezielle Aufsuchoperation (**find for update**) verwendet. Mit der get-Operation wird das zu modifizierende Rekord in das entsprechende Kommunikationsrekord kopiert, wo es durch Zuweisungsoperationen der Gastsprache modifiziert werden kann. Das aktualisierte Rekord wird schließlich durch einen Aufruf der Operation **modify** in die Datenbank zurückgeschrieben.

Hinzu kommen die Operationen auf den DBTG-Mengen. Dazu gehören Operationen zum Einhängen eines Rekords in eine DBTG-Menge (**connect**), zum Aushängen eines Rekords aus einer DBTG-Menge (**disconnect**) und zum Umhängen eines Rekords von einer DBTG-Menge in eine andere des gleichen Typs (**reconnect**). Vor Ausführung dieser Operationen werden

[8]Dieses ist vom Typ *Projektdurchfuehrung*.

```
store projektdurchfuehrung
abteilung.Name := 'Unix SW'
find any Abteilung using Name
connect projektdurchfuehrung to AbtProj
projekt.Titel := 'Telekom Statistik'
find any projekt using Titel
connect projektdurchfuehrung to ProjAbt
```

Abbildung 43: DBTG-Aktualisierungsoperation

die Aktualitätszeiger durch geeignete find-Operationen auf die entsprechenden Rekords und DBTG-Mengen positioniert.

Bei Aktualisierungsoperationen auf Netzwerkdatenbanken spielen die in Abschnitt 3.4.4 erwähnten Optionen eine wichtige Rolle. Bei der Definition einer DBTG-Menge kann z.B. festgelegt werden, daß Rekords ihres Mitgliedstyps bei deren Erzeugung (**store**) automatisch in eine DBTG-Menge des definierten Typs eingehängt werden (**automatic**). Weiterhin kann das Aus- und Umhängen von Rekords zwischen DBTG-Mengen eingeschränkt werden. Dabei ist es möglich das Aus- und Umhängen ganz zu verbieten (**fixed**) oder nur ein Umhängen innerhalb des gleichen DBTG-Mengentyps, aber kein Aushängen zu erlauben (**mandatory**), sodaß ein Mitgliedsrekord stets in einer der DBTG-Mengen des definierten Typs enthalten ist.

Abbildung 43 zeigt die Realisierung der bereits in den Abschnitten 3.2.8 und 3.3.7 betrachteten Aktualisierungsoperationen im Rahmen des Netzwerkmodells. Ziel der Operation ist es, eine Beziehung zwischen der Abteilung *Unix SW* und dem Projekt *Telekom Statistik* zu etablieren. Dazu wird zunächst ein Rekord *pd* vom Typ *Projektdurchfuehrung* in die Datenbank eingefügt (**store**). Die Vorbereitung des Rekords im Kommunikationsrekord *projektdurchfuehrung* durch die Zuweisung von Attributwerten entfällt hier, da es sich um ein reines Verbindungsrekord ohne eigene Komponenten handelt. Danach muß das Rekord noch in die entsprechenden DBTG-Mengen eingefügt werden. Dies geschieht jeweils in zwei Schritten.

Im ersten Schritt wir der Eigner der betroffenen Menge, die Abteilung *a* mit dem Namen „Unix SW“ aufgesucht. Zu diesem Zweck wird dem Attribut *Name* des Kommunikationsrekords *abteilung* der Name der Abteilung zugewiesen und nachfolgend eine entsprechende find-Operation durchgeführt. Diese Operation macht die DBTG-Menge des Typs *AbtProj*, in der die Abteilung *a* Eigner ist, zur aktuellen ihres Typs. Im zweiten Schritt kann dann das Rekord *pd* mit einer connect-Anweisung in diese DBTG-Menge eingefügt werden. Das Einhängen des Rekords *pd* in die DBTG-Menge vom Typ *ProjAbt* geschieht analog.

3.4.7 Das Hierarchische Datenmodell

Hierarchische Strukturierungen und Benennungsschemata erfreuen sich in vielen Bereichen des Lebens großer Beliebtheit. Unter anderem sind z.B. Organisationen aber auch Klassifikationsschemen wie z.B. für Planzen und Tiere häufig hierarchisch organisiert. Auf der Grundlage dieser Tatsache wurde das hierarchische Modell speziell auf hierarchisch organisierte Informationsstrukturen ausgerichtet. Solche Strukturen lassen sich in diesem Datenmodell besonders einfach modellieren und effizient bearbeiten. Probleme ergeben sich jedoch bei der Modellierung von Zusammenhängen, die von einer streng hierarchischen Organisation abweichen.

Da das hierarchische Datenmodell dem Netzwerkdatenmodell in vielen Aspekten sehr ähnlich ist, beschränkt sich dieser Abschnitt auf die wichtigsten Unterschiede zu diesem Modell. Ausführlichere Beschreibungen der Konzepte und Konstrukte des hierarchischen Datenmodells finden sich z.B. in [].

Die Strukturierungsprimitiven des hierarchischen Modells sind vergleichbar mit denen des Netzwerkdatenmodells. In beiden Fällen besteht die Datenbank aus Rekords die durch Referenzen miteinander verknüpft sind. Die starke Ausrichtung des HDM auf Hierarchien führt jedoch zu einigen wichtigen Einschränkungen. Während im NDM (mit gewissen Einschränkungen) beliebige azyklische Graphen aufgebaut werden können, sind beim HDM nur hierarchische Strukturen, d.h. Baumstrukturen, zulässig.

Eine hierarchische Datenbank besteht aus einer Menge von speziellen Bäumen (Wald), den *HDM-Bäumen*. Bäume treten als Datenstrukturen in den verschiedensten Bereichen der Informatik auf. Meist handelt es sich dabei um Strukturen, bei denen alle Knoten einem oder höchstens zwei unterschiedlichen Typen (für innere Knoten und Blätter) angehören, was eine rekursive Definition der zugehörigen Typen erlaubt. Bei den *HDM-Bäumen* handelt es sich dagegen nicht um homogene Strukturen. Vielmehr treten auf jeder Ebene neue Rekordtypen für die Knoten auf. Es ist nicht zulässig, den gleichen Rekordtyp auf mehreren Ebenen zu verwenden.

Die Bäume, aus denen sich eine hierarchische Datenbank zusammensetzt sind alle Ausprägungen desselben *HDM-Baumtyps*, der im Rahmen der Schemadefinition spezifiziert wird. Beginnend mit der Wurzel legt der HDM-Baumtyp die Rekordtypen für die Knoten jeder Ebene fest. Dabei sind auch auf einer Ebene verschiedene Knotentypen zulässig, d.h. ein Knoten kann eine heterogene Nachfolgermenge besitzen. Ein HDM-Baumtyp kann beliebig viele Ebenen für seine Ausprägungen festlegen. Anders als bei rekursiven Defintionen ist hier jedoch die Zahl der Ebenen der Ausprägungen durch die Zahl der in der Typdefinition spezifizierten Ebenen beschränkt.

Zu dem im Schema definierten HDM-Baumtyp kann es beliebig viele Ausprägungen geben. Dabei muß gelten, daß jedes Rekord nur einmal in den Baumstrukturen vorkommt. Dies verbietet jedoch keine zwei wertgleichen Rekords. Solche Duplikationen von Information lassen sich sogar in vielen Fällen, wie z.B. bei der Darstellung von n:m-Beziehungen, gar nicht vermeiden. Auf der anderen Seite muß aber auch jedes der Rekords in einem der Bäume enthalten sein, da es sonst nicht Teil der Datenbank ist. Beim HDM ist die Persistenz also an die Mitgliedschaft in einem der HDM-Bäume gekoppelt.

Mit der in Abschnittt 3.1 eingeführten, einheitlichen Notation kann eine hierarchische Datenbank wie folgt beschrieben werden: Sie besteht aus einer benannten Kollektion (Name der Datenbank) von Aggregaten, den Wurzelelementen der Bäume. In Abbildung 44, die einen Zustand der Beispielsdatenbank (ohne Abteilungshierarchie) bei einer hierarchischen Modellierung zeigt, bilden die Projekte die Wurzeln der Bäume. Jedes dieser Aggregate besitzt Verweise auf eine oder mehrere Kollektionen (für jeden Nachfolgerknotentyp eine Kollektion) von Aggregaten, die die Nachfolgerknoten bilden. Im Beispiel existiert jeweils nur eine Kollektion von Nachfolgerknoten, nämlich die Kollektion aller Abteilungen, die an diesem Projekt arbeiten. Von den Aggregaten in diesen Kollektionen kann wiederum jedes auf eine oder mehrere weitere Kollektion von Aggregaten verweisen usw., wodurch die nächste Baumebene erreicht wird. Im gewählten Beispiel besitzen die Bäume allerdings neben der Wurzel nur eine einzige Ebene. In der konkreten Implementierung sind die erwähnten Kollektionen in der Regel Listen, sodaß eine Reihenfolge auf den Elementen der Kollektion definiert ist.

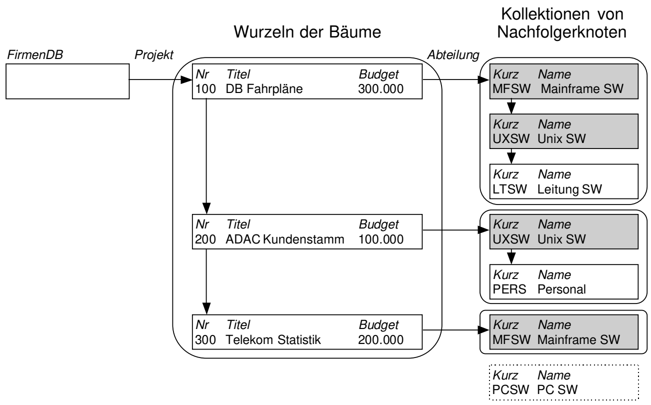

Abbildung 44: Zustand der Firmendatenbank (HDM-Modellierungsvariante A)

Augrund der oben beschriebenen Einschränkungen ist das HDM bei der Modellierung von Assoziationen nur für binäre 1:1- und 1:n-Beziehungen gut geeignet. Binäre 1:n-Beziehungen werden vom HDM direkt unterstützt, da in den Baumstrukturen n Nachfolgerknoten einem Elternknoten zugeordnet werden. Indem man sich auf einen Nachfolgerknoten beschränkt, lassen sich auch binäre 1:1-Beziehungen darstellen. Bei der Modellierung von n:m-Beziehungen, von n-ären Beziehungen und auch bereits dann, wenn ein Knotentyp als Nachfolgerknoten verschiedener Knoten auftritt, ergeben sich jedoch größere Probleme als beim NDM.

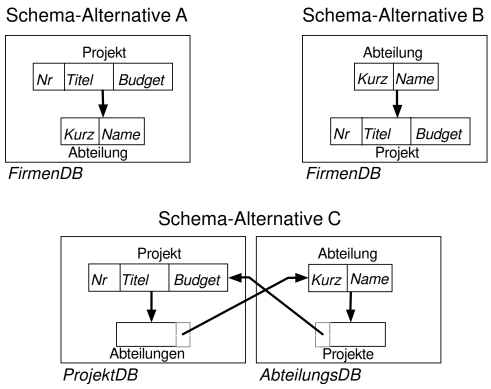

Abbildung 45: Schemadiagramme für drei Modellierungsvarianten der Projektdatenbank (Hierarchisches Datenmodell)

Insbesondere muß für die Darstellung einer n:m-Beziehunge zwischen verschiedenen Modellierungsvarianten gewählt werden, die sich aus der relativen Anordnung der beteiligten Rekordtypen in der Baumstruktur ergeben. Abbildung 45 zeigt die verschiedenen Varianten für die Modellierung der n:m-Beziehung zwischen Projekten und Abteilungen. Bei Variante A ist *Abteilung* Nachfolgerknoten von *Projekt* (vgl. Abbildung 44), während bei der Variante B *Projekt* im Baum unterhalb von *Abteilung* angeordnet ist. Der hierarchische Ansatz führt zu einer Asymmetrie bei der Darstellung von n:m-Beziehungen, die bei den übrigen betrachteten Datenmodellen nicht auftritt.

Unabhängig von der gewählten Modellierungsvariante erfordert die Darstellung von n:m-Beziehungen die Duplizierung von Information, da jedes Rekord nur höchstens einen Elternknoten besitzen darf (Hierarchie!). Die Wahl der Variante entscheidet jedoch darüber, Rekords welchen Typs dupliziert werden müssen und welche Zugriffe auf Daten besonders gut unterstützt werden. So ist z.B. bei der Modellierungsvariante A der Zugriff auf alle Abteilungen, die an einem Projekt arbeiten, besonders einfach möglich. Ein Zugriff auf alle Projekte, an denen eine Abteilung arbeitet, erfordert dagegen ein Durchsuchen der gesamten Baumebene (aller Projekte mit den zugehörigen Abteilungen). Die Wahl der Modellierungsvariante muß daher anwendungsabhängig erfolgen und setzt eine Kenntnis der bevorzugten Zugriffsarten voraus.

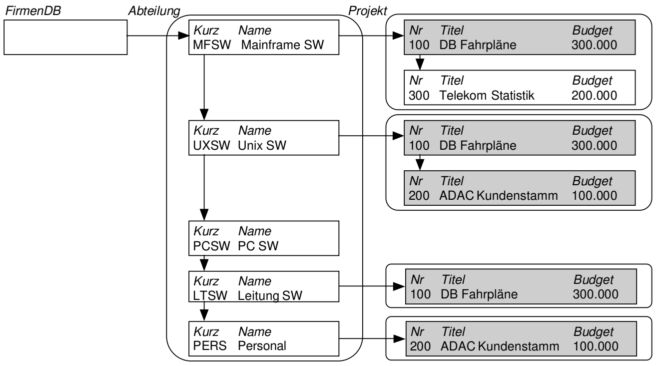

Abbildung 46: Zustand der Projektdatenbank (HDM-Modellierungsvariante B)

Die Notwendigkeit zur Duplizierung von Information bei der Darstellung von n:m-Beziehungen ist in Abbildung 44 zu sehen, die einen Zustand der Projektdatenbank bei Modellierungsvariante A darstellt. Die Informationen zu den Abteilungen *Leitung SW* und *Mainframe SW* kommen jeweils doppelt in der Datenbank vor, da jede dieser Abteilungen an zwei Projekten arbeitet. Allgemein müssen die Ausprägungen des als Nachfolgerknoten gewählten Rekordtyps dupliziert werden, und zwar genau dann, wenn sie mit mehr als einem Rekord des Elternknotentyps in Beziehung stehen. Abbildung 46 zeigt einen Zustand der Beispieldatenbank bei Modellierungsvariante B. Bei dieser Variante kommen entsprechend Informationen über gewisse Projekte mehrfach in der Datenbank vor.

In Abbildung 44 ist noch ein weiteres Problem hierarchischer Datenbanken zu erkennen. Wenn eine Abteilung, wie hier die Abteilung *PC Software*, aktuell nicht an einem Projekt arbeitet, kann sie auch nicht in der Datenbank dargestellt werden. Es ergeben sich also Abhängigkeiten, die in der Realität so nicht existieren: Eine Abteilung kann auch dann weiter bestehen, wenn sie aktuell nicht an einem Projekt arbeitet.

Neben dem Speicherplatzbedarf erhöht die Duplizierung von Information auch die Gefahr von Inkonsistenzen. Bei Aktualisierungsoperationen auf der Datenbank ist sicherzustellen, daß mit einem Datenobjekt zugleich auch alle evtl. existierenden Kopien geändert werden, da die Datenbank sonst inkonsistente Information enthält.

Um die Probleme der Informationsduplizierung und der Asymmetrie bei der Darstellung von Beziehungen zu umgehen, wurde das Konzept der sog. *virtuellen Rekords* eingeführt []. Idee ist es dabei, daß mehr als ein hierarchisches Schema für eine Datenbank unterstützt wird, das heißt, daß der gleiche Datenbestand scheinbar gleichzeitig gemäß verschiedener

HDM-Baumtypen organisiert ist.

Die Umgehung einiger der Einschränkungen des hierarchischen Datenmodells wird dabei durch eine zusätzliche Indirektion erreicht. Anstatt ein Rekord r zu duplizieren wird ein *virtueller Rekord* eingeführt, der einen Verweis auf den Rekord r enthält. Der tatsächliche Rekord tritt nur in einem Baum auf, an allen anderen Stellen werden nur virtuelle Rekords verwendet. von denen auch mehrere auf dasselbe Datenobjekt verweisen können. Jedes virtuelle Rekord kommt wiederum nur einmal in der Datenbank vor. Es kann neben den Verweisen zusätzliche Komponenten enthalten, wodurch sich weitere Optionen für die Modellierung ergeben.

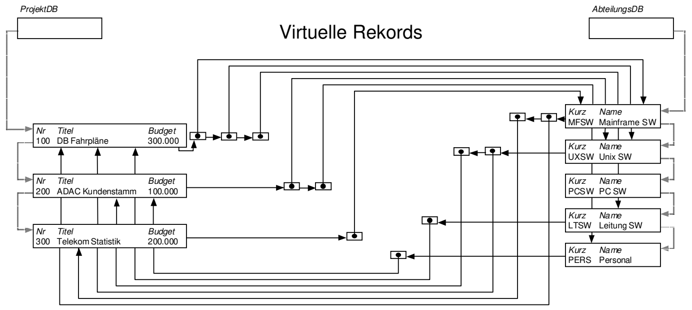

Abbildung 47: Datenbankzustand der Projektdatenbank (HDM-Modellierungsvariante C)

Abbildung 47 zeigt die Modellierung der Projektdatenbank unter Verwendung von virtuellen Rekords (Modellierungsvariante C). Es werden zwei Hierarchien (HDM-Baumtypen) unterstützt: *ProjektDB* und *AbteilungsDB*, die den vorher untersuchten Modellierungsvarianten A und B entsprechen. Die Nachfolgerknoten sind in beiden Hierarchien durch virtuelle Rekords ersetzt, die auf die eigentliche Datenobjekte verweisen. Wie man leicht erkennt, vermeidet diese Art der Modellierung sowohl die Duplizierung von Information als auch die starke Ausrichtung auf bestimmte Zugriffsarten, die bei den Modellierungsvarianten A und B vorhanden sind.

Wie ein Vergleich der Abbildung 47 mit dem entsprechenden Ausschnitt der Abbildung 39 zeigt, führt die Verwendung virtueller Rekords zu einer Annäherung des hierarchischen Datenmodells an das Netzwerkdatenmodell. In einer gewissen Weise sind die virtuellen Rekords mit den Verbindungsrekords des Netzwerkmodells vergleichbar. Sie dienen ebenfalls alleine der Darstellung von Beziehungen und brauchen keine Datenkomponenten zu besitzen.

Die Formulierung von Anfragen geschieht im HDM satzorientiert und prozedural wie im NDM. Es stehen entsprechende Operationen zur Verfügung, die in eine Gastsprache eingebettet werden. Die Kommunikation zwischen der Datenbank und der Applikation findet ebenfalls wie im Netzwerkmodell über eine UWA (*user work area*. vgl. Abschnitt 3.4.5) statt.

Wegen der einfacheren Struktur einer hierarchischen Datenbank (Einschränkung auf Bäume) genügen weniger mächtige Operationen zur Navigation durch die Datenbank. Die Funktionalität der find- und der get-Operation des NDM (vgl. Abschnitt 3.4.5) ist in einer Funktion **get** zusammengefaßt, von der es mehrere Varianten gibt. Insbesondere werden Operationen zum Durchsuchen von Teilbäumen unterstützt. Anders als beim Netzwerkdatenmodell können die Suchoperationen Prädikate als Parameter erhalten, die den Suchraum einschränken. Dabei ist es möglich, Bedingungen für die Knoten auf dem Pfad zu dem gesuchten Rekord festzulegen.

Bei den Aktualisierungoperationen kann wie bei den anderen betrachteteten Datenmodellen zwischen Operationen zum Einfügen, Löschen und Modifizieren unterschieden werden.

Neue Bäume werden durch Einfügen eines neuen Elements (Rekords) des Wurzelknotentyps erzeugt. Beim Einfügen eines neuen Rekords in eine Hierarchie wird dessen Position durch Umsetzen der entsprechenden Aktualitätszeiger festgelegt. Ein Rekord kann dabei nur nach seinem Vorgängerknoten eingefügt werden, da die Existenz eines Rekords in der Datenbank direkt an seine Einordnung in einen der HDM-Bäume gebunden ist (s.o.). Diese Einschränkung erzwingt bestimmte Integritätsbedingungen.

Das Löschen und Ändern von Elementen der Datenbank geschieht wie beim NDM in zwei Schritten. Im ersten Schritt wird das betroffenen Rekord lokalisiert. Dazu wird eine spezielle Variante der oben erwähnten get-Operation benutzt. Im zweiten Schritt wird eine Lösch- bzw. Modifikationsoperation auf dem aufgesuchten Rekord ausgeführt. Eine Löschoperation löscht dabei nicht nur den lokalisierten Rekord, sondern den gesamten Teilbaum, dessen Wurzel dieser Rekord ist. Die Modifikation von Attributwerten geschieht wie beim NDM im Kommunikationsrekord durch eine geeignete Zuweisungsoperation der Gastprache und nachfolgendes Einbringen der Änderung in die Datenbank.

3.5 Vergleich der Modelle

Im direkten Vergleich der betrachteten Modelle schneiden das relationale und die objektorientierten Modelle deutlich besser ab, als die beiden älteren Modelle. Dieser Vorteil spiegelt sich auch in der rasch wachsenden Praxisverbreitung dieser beiden moderneren Modelle wieder.

Dabei zeichnet sich das relationale Modell besonders durch seine Einfachheit und die guten formalen Grundlagen aus, während beim objektorientierten Datenmodell besonders die hohe Modellierungsmächtigkeit und die explizite Unterstützung des Objektbegriffs, der bei objektorientierten Ansätzen allgemein zum anwendungsfreundlichen Charakter beiträgt, hervorzuheben sind. Sowohl beim relationalen Modell als auch beim Objektmodell führt die mengenorientierte, deskriptive Anfragesprache zu einer hohen Benutzerfreundlichkeit. Beide Modelle verfügen außerdem über ein hohes Maß an Datenunabhängigkeit.

Netzwerkmodell und hierarchisches Modell entsprechen nicht mehr den modernen Anforderungen an ein Datenmodell (Datenbanksystem). Dabei sind besonders die mangelnde Datenunabhängigkeit, die komplexe, schwer durchschaubare Semantik der Navigationsoperationen und deren Abhängigkeit von den Aktualitätszeigern zu nennen sowie die prozeduralen, satzorientierten Anfragesprachen, die dem Benutzer die Kontrolle der Iteration aufbürden. Beim hierarchischen Modell kommt erschwerend die eingeschränkte Modellierungsmächtigkeit hinzu.

Um einen direkten Vergleich der unterschiedlichen Modelle zu erleichtern, werden in Tabelle 1 noch einmal die wichtigsten Aspekte der einzelnen Modelle in tabellarischer Form gegenübergestellt.

4 Die relationale Datenbanksprache SQL

Dieser Abschnitt stellt die Sprachkonstrukte der Datenbanksprache SQL vor, die aufgrund ihrer weiten Verbreitung und hohen Akzeptanz in der Praxis auch als *intergalactical dataspeak* (Michael Stonebraker []) bezeichnet wird. Obwohl SQL ursprünglich für relationale Datenbanksysteme entwickelt wurde (s. Abschnitt 2.6.3), haben viele Konstrukte der Sprache (insbesondere die select from where-Iterationsabstraktion und das grundlegende Format der DDL-Anweisungen) Eingang in andere, funktionale und objektorientierte Datenbanksprachen gefunden, und es existieren auch Werkzeuge zur Übersetzung von SQL Anweisungen in Anweisungssequenzen für hierarchische und Netzwerkdatenbanken.

Die nachfolgenden Ausführungen decken einen Großteil des im SQL-92 Standard definierten Sprachumfangs ab, streben jedoch keine enzyklopädische Vollständigkeit an. Im Gegensatz zu SQL-Lehrbüchern [] wird versucht, bei der Vorstellung eines konkreten SQL-Konstrukts auch die zugrundeliegende Datenbankmodellierungsaufgabe und evtl. alternative Lösungsansätze herauszuarbeiten.

4.1 Lexikalische und syntaktische Regeln

Wie in modernen Programmiersprachen üblich, wird ein SQL Quelltext vor der Syntaxanalyse zunächst in eine Folge von *Symbolen* (Lexeme, Token) zerlegt. Bei dieser Zerlegung werden nicht-druckbaren Steuerzeichen (Zeilenvorschub, Seitenwechsel, Tabulator, ...) und *Kommentare* wie Leerzeichen behandelt, so daß die Formatierung des Quelltextes keinen Einfluß auf die Semantik von SQL Anweisungen besitzt. Ein Kommentar ist eine beliebige Zeichenfolge, die mit „–“ beginnt und dem Zeilenende endet:

select * – *dies ist ein Kommentar*
from *Tabelle* – *mit Umlauten äöü bzw. ß*

Außerdem werden Kleinbuchstaben im Quelltext sofort in Großbuchstaben umgewandelt, falls sie nicht in einem speziellen Kontext auftreten (s.u.). SQL unterscheidet folgende Symbole:

- Ein *Name* (Bezeichner) beginnt mit einem Buchstaben, dem eine (evtl. leere) Kette aus Buchstaben, Ziffern und dem Zeichen „_“ folgt. Jede SQL Implementierung definiert eine Zahl n, so daß Bezeichner, die in den ersten n Buchstaben übereinstimmen, als identisch betrachtet werden.

- Der SQL-92 Standard definiert über 210 Namen als *Schlüsselworte*, die nicht zur direkten Benennung benutzerdefinierter Objekte eingesetzt werden könnnen (**absolute**, **action**, **all**, **and**, ..., **from**, ... **select**, **year**, **zone**). In diesem Text werden Schlüsselworte durch Fettdruck hervorgehoben. Schlüsselworte sind in SQL *nicht kontextsensitiv*. Zum Beispiel ist das Schlüsselwort **create**, das nie im Kontext von Anfragen auftreten kann, trotzdem kein zulässiger Name für eine Variable innerhalb einer Anfrage.

- Zusätzlich zu den oben beschriebenen sogenannten *regulären Namen* können auch sogenannte *begrenzte Namen* als Symbole auftreten. Ein begrenzter Name ist eine Zeichenkette innerhalb doppelter Anführungszeichen. Diese Zeichenkette kann beliebige

	Relationales Modell	**ODMG-Objektmodell**	**Netzwerkdatenmodell**	**Hierarchisches Datenmodell**
	Identifikation			
	assoziativ	referentiell	referentiell	referentiell
	Attribute			
Aggregation von Attributen	Tupel	Attribute des Objekttyps	Rekords	Rekords
zusammengesetzte Attribute	keine Modellierungsunterstützung	Strukturierter Attributtyp	Wiederholgruppen	-
mengenwertige Attribute	eigene Relation	Kollektionstyp als Attributtyp	Wiederholgruppen	
	Darstellung von Assoziationen			
generell	Fremdschlüssel	Referenzen auf Objekte	DBTG-Mengen	HDM-Baumstrukturen
1:1	als Attribut einer beteiligten Relation	einzelne Referenzen	DBTG-Menge (ein Mitglied[9])	HDM-Baum (ein Nachfolger)
1:n		Kollektion von Referenzen	DBTG-Menge	HDM-Baum
n:m	eigene Relation 2 Fremdschlüssel	Kollektionen von Refrenzen	Verbindungsrekord + 2 DBTG-Mengen	Duplizierung von Daten bzw. virt. Rekords
n-äre ($n > 2$)	Relation mit n Fremdschlüsseln	eigener Objekttyp	Verbindungsrekord + n DBTG-Mengen	Duplizierung von Daten bzw. virt. Rekords
	Anfragesprachen			
	deskriptiv mengenorientiert	deskriptiv mengenorientiert	prozedural satzorientiert	prozedural satzorientiert
	Strukturen für Massendaten			
	Relationen	Extensionen von Objekttypen	DBTG-Mengen	HDM-Baumstrukturen

Tabelle 1: Vergleich der Modelle

Literal	**SQL Datentyp**
'abc'	**character**(3)
N' äöüß'	**national character**(4)
123	**smallint**
123.456	**decimal**
B' 101010'	**bit**(6)
X' 12EF'	**bit**(16)
date '12-31-94'	**date**
time '12:34:56.78'	**time**(2)
time '12:34:56.789-01:00'	**time**(3) **with time zone**
timestamp '12-31-94 12:34:56'	**timestamp**(0)
interval '12' day	**interval day**

Tabelle 2: SQL Datentypen

Sonderzeichen (auch Umlaute) und Kleinbuchstaben enthalten, die nicht in Großbuchstaben umgewandelt werden. Da begrenzte Namen von Schlüsselworten lexikalisch unterschieden sind, kann durch den Einsatz begrenzter Namen vermieden werden, daß ein einmal erstelltes SQL Programm beim Umstieg auf ein neueres SQL System mit einem reicheren Satz an Schlüsselworten ungültig wird (*Syntaxerweiterungsproblematik*).

- SQL besitzt weiterhin umfangreiche lexikalische Regeln für *Literale*, dies sind Symbole zur Benennung von Werten der SQL Basisdatentypen (s. Tabelle unten). Diese Regeln erläutern insbesondere die Behandlung nationaler Zeichensätze, die Definition von Stringliteralen (Zeichenkettenkonstanten), die länger als eine Eingabezeile sind, oder die Steuerzeichen enthalten. Die Typisierungsregeln für Literale werden in Abschnitt 4.2.3 erläutert.

- Weitere SQL Symbole (z.B. *Operatoren*) werden durch Sonderzeichen des ASCII Zeichensatzes (>, <, =, %, &, (,), *, +, ...) gebildet.

Aufgrund der zahlreichen *Modalitäten*, in denen SQL eingesetzt wird, sind im Einzelfall zusätzliche lexikalische Regeln zu berücksichtigen. Bei der interaktiven Eingabe von SQL-Ausdrücken ist es zum Beispiel erforderlich, die Eingabe einer Anweisungssequenz explizit abzuschließen (z.B. Eingabe von CTRL-D oder „\g“), um die Auswertung der Sequenz durch das Datenbanksystem auszulösen. Bei der Definition von SQL Anfragen innerhalb von Stringliteralen einer Gastsprache (z.B. C) werden zunächst die lexikalischen Regeln der Gastsprache und dann die lexikalischen Regeln von SQL berücksichtigt. Dies führt insbesondere bei geschachtelten Stringliteralen zu unübersichtlichen Konstruktionen:

char* sqlQuery = ” **select** * **from** Tabelle **where** x 2697 = \’ \ \ n \’ ”

SQL besitzt eine sehr umfangreiche *Syntax*, die sich durch eine hohe Anzahl optionaler Klauseln und schlüsselwortbasierter Operatoren auszeichnet. In dieser Einführung stellen wir nur eine Auswahl der SQL Konstrukte vor und verzichten auf eine formale Definition der SQL-Syntax durch Grammatiken oder Syntaxdiagramme, da diese in jeder Systemdokumentation von SQL Datenbanksystemen zu finden sind.

Datenbankmodell-Terminologie	**SQL-Terminologie**	**Englische Übersetzung**
Relation	Tabelle	*table*
Tupel	Reihe	*row*
Attribut	Spalte	*column*
Attributname	Spaltenname	*column name*
Sicht	Tabellensicht	*view table*
Domäne	Basisdatentyp	*base data type*

Tabelle 3: SQL-Terminologie

4.2 Datendefinition in SQL

In diesem Abschnitt werden die wichtigsten SQL-Konstrukte zur Datendefinition vorgestellt. Dabei wird der SQL-Terminologie gefolgt, die von der in Abschnitt 3.2 eingeführten Terminologie des relationalen Datenmodells abweicht (s. Tabelle 3):

4.2.1 Dynamische DDL Anweisungen

Im Gegensatz zu anderen Datenbankmodellen findet in SQL keine strikte Trennung in eine *data definition language* (DDL) zur Schemadefinition und eine separate *data manipulation language* (DML) zur Datenbankmanipulation statt.

Die *Deklaration* einer Tabelle (**create table**) wird in SQL ebenso wie z.B. die Tabellenoperation **update** als eine *Anweisung* betrachtet, die erst dynamisch zur Laufzeit ausgewertet wird. Dieser Ansatz besitzt folgende Vorteile:

- Operationen und Deklarationen können innerhalb von Transaktionen zu einer atomaren Operation *zusammengefaßt* werden.
- Deklarationen können zur Laufzeit durch Werte einer Gastsprache *parametrisiert* werden (Tabellen- und Spaltennamen, Typen, Integritätsbedingungen,...).
- Deklarationen können innerhalb *zusammengesetzter Anweisungen* und Schleifen (**if**, **switch**, **loop**) einer Gastsprache auftreten.

Diese Flexibilität bei der Behandlung von Deklarationen führt aber auch zu erheblichen Nachteilen bei der Arbeit mit statischen Datenbankschemata, wie sie in typischen Informationssystemen anzutreffen sind:

- *Dynamische Bindung*: Alle globalen Namen, die in SQL-Operationen verwendet werden (Tabellen, Sichten, ...) müssen zur Laufzeit dynamisch (in Schemata und Katalogen) identifiziert werden. Auch ein einfacher Tippfehler in einer Anfrage (**select** * **from** *Mitrabeiter*) wird erst zur Laufzeit erkannt.
- *Dynamische Typisierung*: Die Kompatibilität zwischen Anfragen und der Struktur der von ihnen benutzten globalen SQL-Objekte kann ebenfalls erst zur Laufzeit getestet werden. Zur Übersetzungszeit der Anfrage **select** * **from** *Mitarbeiter* **where** *Salary*>*Size*

ist also unbekannt, ob die Tabelle Mitarbeiter tatsächlich die Spalten Salary und Size besitzt, ob diese einen kompatiblen Basisdatentyp besitzen, welche Vergleichsoperation (exakte Ganzzahlarithmetik, Stringvergleich) zu verwenden ist, und welche Spalten das Ergebnis besitzt. Gerade bei Gastspracheneinbettungen ist es daher erforderlich, zusätzliche statische Typinformationen in der Programmiersprache zu verwalten, um zum Beispiel auf das Anfrageergebnis der obigen Anfrage typsicher zuzugreifen.

Moderne Datenbankmodelle (zum Beispiel das objektorientierte Datenbanksystem O_2) lösen dieses Dilemma zwischen Flexibilität und statischer Konsistenzkontrolle beim Datenbankzugriff, indem sie eine *Trennung* zwischen der statischen *Definition* von Schemainformationen (analog zu Modulschnittstellen und Typinformationen) und der dynamischen *Instantiierung* von Datenbankobjekten (analog zu dynamisch gebundenen Modulen, lokalen Prozedurvariablen oder anonymen Pointervariablen) einführen (s. Abschnitt 1.5.3). Im obigen Beispiel ist also die Struktur der Tabelle *Mitarbeiter* zur Übersetzungszeit der eingebetteten Anfrage bekannt, auch wenn die Tabelle evtl. erst nach der Übersetzung dynamisch angelegt wird.

`Vorwärtsverweis:`

polymorphe Typisierung `und` ***Typinferenz*** `(select* from X as x where x.age>30)` `und` ***reflektive Programmierung*** `(Konstruktion von Anfragen basierend auf Schemainformationen).`

4.2.2 Schemata und Kataloge

Ein SQL-Schema ist ein *dynamischer Sichtbarkeitsbereich* für die Namen geschachtelter (lokaler) *SQL-Objekte* (Tabellen, Sichten, Regeln, ...). Im Gegensatz zu statischen Sichtbarkeitsbereichen in blockstrukturierten Programmiersprachen können die *Bindungen* von Namen an Objekte in dynamischen Sichtbarkeitsbereichen durch Anweisungen explizit erzeugt und gelöscht werden:

create schema *FirmenDB;*
  create table *Mitarbeiter ... ;*
  create table *Produkte ... ;*

create schema *ProjektDB;*
  create table *Mitarbeiter ... ;*
  create view *Leiter ... ;*
  create table *Projekte ... ;*
  create table *T ... ;*
  drop table *T;*

drop schema *FirmenDB*

Klassische Datenbankanwendungen arbeiten mit SQL-Objekten genau eines Schemas. Die Integration separat entwickelter Datenbankschemata und die Arbeit in verteilten und föderativen Datenbanken erfordert jedoch auch den simultanen Zugriff auf SQL-Objekte mehrerer Schemata (*multibases*).

Der Name eines mit **create schema** erzeugten SQL-Schemas kann daher verwendet werden, um einen Namen, der in mehr als einem SQL-Schema auftritt, eindeutig mit Hilfe der

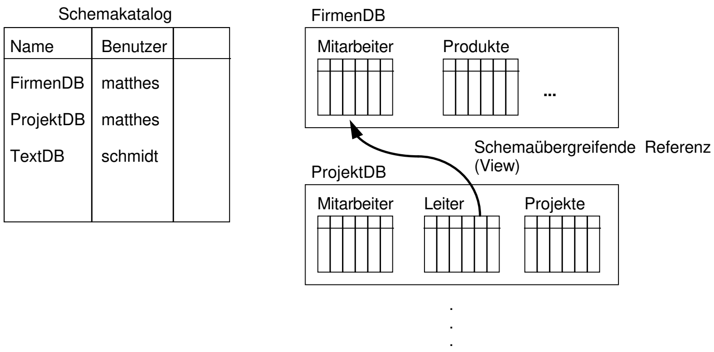

Abbildung 48: SQL-Kataloge und SQL-Schemata als Sichtbarkeitsbereiche (s. Text)

Punktnotation zu *qualifizieren* (*FirmenDB.Mitarbeiter*, *ProjektDB.Mitarbeiter*). Da andererseits jede SQL-Anweisung immer im Kontext eines *aktuellen Schemas* interpretiert wird, kann bei der Arbeit mit nur einem Schema auf den Einsatz qualifizierter Namen verzichtet werden. Das aktuelle Schema, also die Menge der in einer Anweisung global sichtbaren Namen, wird entweder statisch (z.B. bei der Übersetzung von SQL-Modulen) oder dynamisch als Seiteneffekt von Anweisungen (**create schema** *FirmenDB*, **connect** *FirmenDB*) definiert. Damit kommt SQL der in Abschnitt 1.3.1 beschriebenen Forderung nach *dynamischer Bindung* zwischen informationsverarbeitenden Algorithmen und persistenten Informationsbeständen nach.

Ein SQL-Schema ist *persistent*. In SQL impliziert das Anlegen und Löschen eines Schemas gleichzeitig das Anlegen und Löschen einer *Datenbank*, die das Schema implementiert, und die einen persistenten Container für die lokal deklarierten SQL-Objekte (z.B. Tabellen) darstellt. Die Lebensdauer geschachtelter SQL-Objekte ist durch die Lebensdauer ihres Schemas begrenzt. Im obigen Beispiel löscht die Operation **drop schema** *FirmenDB* also ebenfalls die lokale Tabelle *FirmenDB.Mitarbeiter*.

Über qualifizierte Namen ist es auch möglich, in einem Schema Referenzen auf SQL-Objekte eines anderen Schemas zu erzeugen (**create view** *Leiter* **as select** * **from** *FirmenDB.Mitarbeiter* **where** ..., s. Abschnitt 4.6.1). Dies führt zu *Schemaabhängigkeiten*, die beim Löschen eines Schemas berücksichtigt werden müssen:

drop schema *FirmenDB* **cascade***:*

Das Schlüsselwort **cascade** erzwingt in diesem Beispiel das (transitive) Löschen der abhängigen SQL-Objekte in anderen Schemata, insbesondere der Sicht *Leiter* in dem Schema *ProjektDB*.

Schließlich sei darauf hingewiesen, daß Schemanamen ihrerseits in Sichtbarkeitsbereichen enthalten sind, die in der Terminologie von SQL-92 als *Kataloge* bezeichnet werden. Wie in Abbildung 48 dargestellt, werden in Katalogen weitere Informationen zu einem Schema (Zugriffsrechte, Speichermedium, Datum des letzten Backup, ...) persistent gespeichert. Die Namen von Katalogen sind wiederum in Katalogen abgelegt, von denen einer als *Wurzelkatalog* ausgezeichnet ist. Im allgemeinsten Fall wird daher ein SQL-Objekt über eine mehrstufige Qualifizierung von Katalognamen, einen Schemanamen und einen Objektnamen ausgehend von dem Wurzelkatalog eindeutig identifiziert. Die aufwendige Katalogverwaltung wird dabei von Datenbanksystemen teilweise an standardisierte Netzwerknamensdienste (*name services*) delegiert `zitat?`.

4.2.3 Basisdatentypen und Typkompatibilität

Die *formale* Definition des relationales Datenmodell basiert auf einer Menge von *Domänen* (D1, D2, ...,Dn), der die *atomaren* Werte der Attribute von Tupeln in Relationen entstammen. Dabei werden nur schwache Anforderungen an die *algebraische Struktur* einer Domäne D gestellt:

- Existenz einer Äquivalenzrelation (=) auf D zur Definition der Relationensemantik (Duplikatelimination) und des Begriffs der funktionalen Abhängigkeit (s. Abschnitt 3.2.5).
- Existenz weiterer Boole' scher Prädikate (>, <, >=, **substring**, **odd**, ...) auf D zur Formulierung von Selektions- und Joinausdrücken über Attribute (optional).

Eine naheliegende Idee, die von modernen erweiterbaren Datenbankmodellen auch unterstützt wird, besteht darin, beliebige *benutzerdefinierte Domänen* (Datentypen) in Datenbankschemadefinitionen zuzulassen, wobei der Programmierer dem Datenbanksystem unter anderem auch Funktionen zur Implementierung der (Gleichheits-)prädikate bekanntmacht.

Datenbankmodelle der Praxis, wie zum Beispiel SQL-92, besitzen jedoch auch die Aufgabe, den Datenbankzustand und die Semantik von Anfragen unabhängig von speziellen Programmen und Hardwareumgebungen interpretierbar zu halten. Sie definieren daher ein festes Repertoire an anwendungsorientierten *vordefinierten Basisdatentypen*. Die Definition jedes Basisdatentyps umfaßt dabei:

- Lexikalische Regeln für *Literale*, also Notationen zur Benennung von Werten der Basisdatentypen (4, 4.0, „4“),
- *Evaluationsregeln* für unäre, binäre und allgemein n-äre Operatoren (Wertebereich, Ausnahmebehandlung, Behandlung von Nullwerten),
- *Typkompatibilitätsregeln* für gemischte Ausdrücke,
- *Wertkonvertierungsregeln* für den bidirektionalen Datenaustausch mit typisierten Programmiersprachenvariablen bei der Einbettung in Gastsprachen.
- Spezifikationen des *Speicherbedarfs* (minimal, maximal) für Werte des Typs.

Dabei bietet SQL-92 verglichen mit Programmiersprachen eine erheblich feinere Kontrolle über die Datenrepräsentation der Basisdatentypen (Stringlänge, Dezimalstellenanzahl, ...), und erlaubt damit bei der Datendefinition eine anwendungsgerechte Abwägung zwischen *Modellierungspräzision* und *Speicherbedarf*. Außerdem bietet SQL-92 zahlreiche standardisierte Operatoren auf diesen Basisdatentypen (z.B. Suchoperationen in Strings, Datumsfunktionen) und leistet damit ebenfalls einen wichtigen Beitrag zur *Portabilität* der mit SQL entwickelten Programme.

Konkret lassen sich die Basisdatentypen von SQL-92 wie folgt klassifizieren (s. Tabelle in Abschnitt 4.1):

Exact numerics bieten exakte Arithmetik und gestatten teilweise die Angabe einer Gesamtstellenanzahl (**scale**,*s*) und einer Nachkommastellenanzahl (**precision**, *p*) und umfassen die Typen **integer**, **smallint**, **numeric**(*p,s*), **decimal**(*p,s*).

Approximate numerics bieten aufgrund ihrer Fließkommadarstellung einen flexibleren Wertebereich, sind jedoch wegen der Rundungsproblematik nicht für kaufmännische Anwendungen geeignet (**real**, **double precision**, **float**(*p*)),

Character strings beschreiben mit Leerzeichen aufgefüllte Zeichenketten fester Länge *n* (**character**(*n*)) oder variabel lange Zeichenketten mit fester Maximallänge *n* (**character varying**(*n*)).

Analog beschreiben *bit strings* mit Null aufgefüllte Bitmuster fester Länge *n* (**bit**(*n*)) oder variabel lange Bitmuster mit fester Maximallänge *n* (**bit varying**(*n*)).

Die Klasse der *datetime Basistypen* beschreibt Zeit(punkt)werte vorgegebener Granularität (**date**, **time**(*p*), **timestamp**(*p*), **time**(*p*) **with time zone**, **timestamp**(*p*) **with time zone**).

Schließlich beschreiben *time intervals* Zeitintervalle vorgegebener Dimension und Granularität. Zum Beispiel umfaßt der Basisdatentyp **interval year**(*2*) **to month** als Werte folgende Intervallangaben: 0 Jahre und 0 Monate, 0 Jahre und 1 Monat, ..., 1 Jahr und 0 Monate, 1 Jahr und 1 Monat, ..., 99 Jahre und 12 Monate.

Bedenkt man, daß ein Datenbankstandard für jeden Operator (z.B. Zeitintervallberechnung) detaillierte Evaluations- und Typkompatibilitätsregeln definiert, aus denen insbesondere auch die Dimension und Granularität des Berechnungsergebnisses hervorgeht, so wird die in einem Datenbanksystem *versteckte algorithmische Komlexität* deutlich.

Schließlich ist zu erwähnen, daß SQL wie viele Programmiersprachen sowohl die implizite Typanpassung (*coercion*), zum Beispiel zur Umwandlung von Datenwerten geringer Genauigkeit in Datenwerte höherer Genauigkeit, als auch die explizite Typanpassung (*casting*), zum Beispiel zur Umwandlung von Zeichenketten, die nur Ziffern enthalten, in ganzzahlige Werte (**cast**(*'1234'* **as integer**)), unterstützt.

4.2.4 Nullwerte und Wahrheitswerte

Bei der Datenbankmodellierung und -programmierung können Situationen auftreten, in denen anstelle eines Wertes eines Basisdatentyps ein ausgezeichneter *Nullwert* benötigt wird. Folgende Beispiele illustrieren, daß die Semantik (Interpretation) des Nullwerts oft *situationsabhängig* ist:

- Ein Tabellenschema definiert, daß in jeder Reihe der Tabelle *Mitarbeiter* die Spalte *Alter* einen Wert des Typs **integer** besitzt. Ist das Alter eines Mitarbeiters *unbekannt*, so kann dies mit dem Wert **null** gekennzeichnet werden.

- Ein Tabellenschema definiert, daß in jeder Reihe der Tabelle *Abteilung* die Spalte *Oberabteilung* einen Wert des Typs **string** besitzt. Ist *bekannt*, daß eine Abteilung *keine* Oberabteilung besitzt, so kann diese Information mit dem Wert **null** repräsentiert werden.

- Die Typregeln von SQL definieren, daß die Berechnung des Tabellenmaximums oder des Tabellenmittelwerts eines Attributs des Typs **float** ein Ergebnis des selben Typs liefert. Da das Maximum und der Mittelwert einer leeren Menge jedoch mathematisch *undefiniert* ist, kann das Auftreten dieser Ausnahmesituation zur Laufzeit mit dem Ergebniswert **null** signalisiert werden.

Zur Unterstützung solcher Modellierungssituationen ist jeder SQL-Basisdatentyp um den ausgezeichneten Wert **null** erweitert, der von jedem anderen Wert des Basisdatentyps verschieden ist. Fügt man zu SQL-Typspezifikationen die Integritätsbedingung **not null** hinzu (s. z.B. *Name* **char**(40) **not null**), so wird das Auftreten von Nullwerten in Attributen oder Variablen dieses Typs durch statische und dynamische Tests ausgeschlossen.

Ein Nullwert in dem Attribut x kann in Datenbankanfragen über die *Prädikate* x **is null** und x **is not null** behandelt werden.

Als *Vorteil*e der expliziten Unterstützung von Nullwerten durch Datenbankmodelle sind zu nennen:

- Explizite und konsistente Behandlung von Nullwerten durch alle Applikationen (im Gegensatz zu *ad hoc* Lösungen, bei denen z.B. der Wert *-1*, *-MaxInt* oder die leere Zeichenkette als Nullwert eingesetzt wird).

- Exakte Definition der Semantik von Datenbankoperatoren (Zuweisung, Vergleich, Arithmetik) auf Nullwerten.

Diesen Vorteilen stehen jedoch auch einige *Nachteile* gegenüber:

- Eine Erweiterung eines Datentyps um Nullwerte steht oft im Konflikt mit den *algebraischen Eigenschaften* (Existenz von Nullelementen, Assoziativität, Kommutativität, Ordnung, ...) des nicht-erweiterten Datentyps. Es ist zum Beispiel unklar, an welcher Stelle der Sortierordnung der Wert **null** bei den ganzen Zahlen einzuordnen ist (Gilt $\ldots < -2 < -1 < 0 < null < 1 < 2 < \ldots$?). Ähnliches gilt für die Generalisierung der Arithmetik (Gilt $1 +$ **null** = **null** und **null** - **null** = **null** ?). Da die im SQL-Standard (willkürlich) getroffenen Annahmen teilweise nicht mit der speziellen Nullwertsemantik einer Anwendung übereinstimmen, ist es in der Praxis häufig erforderlich, Nullwerte in SQL-Anweisungen aufwendig explizit zu behandeln.

- Wie in Kapitel xx (Anfrageoptimierung) dargestellt, werden algebraische Eigenschaften häufig auch zur *Anfrageoptimierung* ausgenutzt. Eine Anfrage, die Werte eines Datentyps T verwendet, bietet daher im Regelfall weniger Optimierungsspielraum als eine Anfrage mit Werten des Datentyps T **not null**.

- Die Einführung von Nullwerten für Variablen führt zu einem *Schneeballeffekt*, indem Nullwerte auch als (Zwischen-)ergebnisse von beliebigen Ausdrücken auftreten können und separat behandelt werden müssen.

- Beim Datenaustausch zwischen SQL-Datenbanken und Programmiersprachen werden SQL-Attribute eines Typs T, der auch Nullwerte annehmen kann, durch ein Paar bestehend aus einer Programmiersprachenvariablen des Typs T und einer (Boole' schen) *Indikatorvariable* dargestellt, die angibt, ob der SQL-Wert **null** ist. Diese Darstellung führt zu einer hohen *Komplexität der Anwendungsprogramme*.

or	**true**	**false**	**null**
true	**true**	**true**	**true**
false	**true**	**false**	**null**
null	**true**	**null**	**null**

and	**true**	**false**	**null**
true	**true**	**false**	**null**
false	**false**	**false**	**false**
null	**null**	**false**	**null**

x	**not x**	**x is null**	**x is not null**
true	**false**	**false**	**true**
false	**true**	**false**	**true**
null	**null**	**true**	**false**

Tabelle 4: Wahrheitstabellen der dreiwertigen SQL92-Logik

Speziell für den (in SQL-92 nur implizit vorhandenen) Datentyp der *Wahrheitswerte* (**boolean**) führen Nullwerte zu einer *dreiwertigen Logik*(**true**, **false**, **null**), in der die Restriktion aller Operatoren (**and**, **or**, **not**) auf die Argumente **true** und **false** die übliche Boole' sche Semantik besitzt. Die nachfolgenden Wahrheitstabellen zeigen die spezielle dreiwertige Logik, die dem SQL-92 Standard zugrunde liegt. Wie aus den Tabellen hervorgeht, unterstützt SQL-92 also die Interpretation des Wertes **null** als *unbekannt* („vielleicht **true** oder vielleicht **false**"). So ist zum Beispiel das Ergebnis von **true or null** als **true** definiert, da sowohl **true or false** als auch **true or true** zu **true** evaluiert.

Die oben genannte Schwierigkeit einer konsistenten Erweiterung einer Domäne um Nullwerte wird bereits am einfachen Beispiel der Boole' schen Werte und der grundlegenden logischen Äquivalenz x **and not** $x =$ **false** deutlich, die bei der Erweiterung der Domäne um einen Nullwerte verletzt wird (**null and not null** = **null**).

Die hohe Bedeutung, die Nullwerte in der Theorie und Praxis relationaler Modelle erlangt haben, ist zum Teil auch auf die *eingeschränkte Datenstrukturflexibilität* des relationalen Modells (homogene Mengen flacher Tupel) zurückzuführen. Einige Modellierungsaufgaben, die im relationalenn Datenmodell durch Nullwerte gelöst werden, können in anderen Modellen durch speziellere Konzepte wie Vereinigungstypen (*union types, variant records*), Subtypisierung oder Vererbungsbeziehungen gelöst werden.

Abschließend sei angemerkt, daß in der Forschung zahlreiche alternative Modelle für Nullwerte und *mehrwertige Logiken* vorgeschlagen wurden. Ein extremes Beispiel stellt eine Studie der *DataBase Systems Study Group* dar, die 29 alternative Bedeutungen für Nullwerte unterscheidet[].

Bei der Entwicklung des Nachfogers des SQL-92 Standards sind daher zusätzliche Nullwerte und sogar benutzerdefinierte Nullwerte in der Diskussion.

Operation	**Schlüsselwort für Schemamanipulation**	**Schlüsselwort für Datenmanipulation**
Einfügen	**create**	**insert**
Modifizieren	**alter**	**update**
Löschen	**drop**	**delete**

Tabelle 5: Schlüsselworte für die Schema- und Datenmanipulation

4.2.5 Tabellendefinitionen

Tabellen sind die wichtigsten SQL-Objekte, die in einem Schema definiert werden. Eine SQL-Tabellendefinition faßt die folgenden Teilschritte in *einem* syntaktischen Konstrukt zusammen:

- *Typdefinition*: Definition einer Tabellenstruktur (Spaltennamen, Spaltentypen).
- *Variablendefinition*: Definition eines Namens für eine Tabelle dieser Struktur im aktuellen Schema.
- *Variableninstantiierung*: Dynamisches Anlegen einer Variablen dieser Struktur in der aktuellen Datenbank. Eine Variable wird in SQL mit der leeren Tabelle initialisiert.

Moderne Datenbanksprachen bieten separate syntaktische Konstrukte für jeden dieser Schritte, die es zum Beispiel ermöglichen, benannte Tabellentypen zu definieren (Vorteile: Wiederverwendung, statische Schematyisierung).

Das folgende Beispiel illustriert die Syntax einer SQL-Tabellendefinition:

create table *Mitarbeiter(*
Name **char***(29),*
Gehalt **integer**,
Urlaub **smallint***);*

Die Anzahl der Spalten einer Tabelle ist statisch fixiert wird auch als der *Grad* der Tabelle bezeichnet. In SQL ist die Reihenfolge der Spalten einer Tabelle signifiant und es ist nicht erlaubt, Tabellen mit Grad 0 zu definieren.

Die Anzahl der Reihen einer Tabelle variiert dynamisch und wird auch als die *Kardinalität* der Tabelle bezeichnet. Eine Tabelle mit Kardinalität 0 heißt *leer*. Im Gegensatz zur Definition des relationalen Modells kann eine Tabelle *Duplikate* enthalten. Eine Reihe ist ein Duplikat einer anderen Reihe, wenn beide in allen Spalten gemäß der Äquivalenzrelation der jeweiligen Spaltentypen übereinstimmen. In den meisten Kontexten ist die *Reihenfolge* der Reihen einer Tabelle unspezifiziert (siehe aber Abschnitt Abschnitt 4.3.2 und Abschnitt 4.3.9).

Tabellendefinitionen können wie die meisten Objekte eines SQL Schemas dynamisch modifiziert werden. SQL verwendet unterschiedliche Schlüselworte für die Manipulation von Daten und die Manipulation von Metadaten:

Das folgende Beispiel zeigt *Schemaänderungen* für die oben definierte Tabelle *Mitarbeiter*.

alter table *Mitarbeiter*
  add column *Adresse* **varchar***(40)*
  alter column *Name* **unique**
  drop column *Urlaub;*
drop table *Mitarbeiter;*

Dabei ist zu beachten, daß die SQL-DDL-Anweisung **drop table** *T* sich von der SQL-DML-Anweisung **delete from** *T* (s. Abschnitt 4.4.3) darin unterscheidet, daß nicht nur der gesamte Inhalt der Tabelleninstanz sondern auch die Tabellentypinformation und die Tabellenvariableninformation gelöscht werden.

4.2.6 Lebensdauer, Sichtbarkeit und gemeinsame Nutzung von Tabellen

Wie in Abbildung 49 dargestellt, können mehrere informationsverarbeitende *Prozesse* (Sitzungen, *sessions*) die gleiche Datenbank simultan oder sequentiell nacheinander benutzen. Ein Prozeß führt eine Folge von *Transaktionen* (s. Abschnitt 4.7) durch, innerhalb derer Datenbankzugriffe stattfinden. Prozesse und Transaktionen benutzen *Namen* zur Identifikation von Datenbankobjekten.

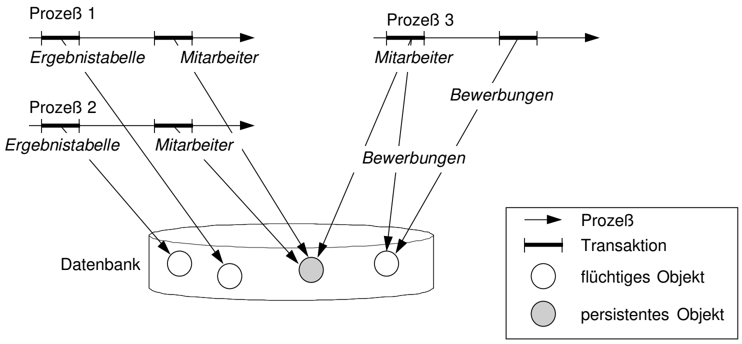

Abbildung 49: Sichtbarkeit, Lebensdauer, gemeinsame Nutzung

Bei der Deklaration von Datenbankobjekten wie SQL-Tabellen sind daher drei Objekteigenschaften zu definieren:

Lebensdauer (extent): Der Zustand eines Objektes kann *persistent* oder nur *flüchtig* gespeichert werden. Persistente SQL-Objekte (*Mitarbeiter*) existieren solange bis sie oder die Datenbank, in der sie deklariert wurden, explizit durch eine SQL Anweisung (**drop**, **delete** ...) gelöscht werden. Eine flüchtige Lebensdauer kann sich über einen gesamten Prozeß (Bewerbungen), eine einzelne Transaktion oder nur eine einzelne SQL-Prozedur erstrecken.

Sichtbarkeit (scope): Der Name eines Objekts kann *global* für alle Prozesse, die eine Datenbank benutzen (*Mitarbeiter*), oder nur *lokal* für einen Prozeß (Bewerbungen, Ergebnistabelle) sichtbar sein. Der Sichtbarkeitsbereich eines Prozesses kann durch SQL-Module noch weiter partitioniert werden.

Gemeinsame Nutzung (sharing): Ein Name kann entweder eine *Referenz* auf ein für mehrere Prozesse zugreifbares Objekt (*Mitarbeiter) oder eine prozeßlokale Kopie* (Instanz) eines

Objekts (Ergebnistabelle) bezeichnen. Referenzen und Kopien unterscheiden sich in der Wirkung von Seiteneffekten (**insert**, **delete**, **update**).

Es ist zu beachten, daß diese Objekteigenschaften *nicht* vollständig *orthogonal* zueinander sind, da zum Beispiel eine Referenz auf ein gemeinsam genutztes Objekt nur erzeugt werden kann, wenn ein Namen in einem globalen Sichtbarkeitsbereich zu seiner Identifikation zur Verfügung steht.

Problematisch ist weiterhin die Situation, in der ein Name mit globaler Sichtbarkeit ein Objekt mit flüchtiger Lebensdauer bezeichnet, da am Ende der Lebensdauer das an den Namen gebundene Objekt undefiniert ist. Dieses Problem der sogenannten *dangling references* ist auch aus Programmiersprachen bekannt.

Da SQL-Datenbanksysteme eine Persistenzabstraktion durchführen, ist es nicht erforderlich, als vierte Objekteigenschaft den *Adreßraum* (im Primär- oder Sekundärspeicher) zu definieren, in dem ein (lokales oder globales) SQL-Objekt anzulegen ist.

Historisch gesehen haben sich Datenbanksysteme auf persistente und globale Datenobjekte konzentriert. Wie in Abschnitt 1.3.1 aufgezählt, existieren jedoch in Informationssystemen häufig auch lokale und flüchtige Informationsstrukturen, bei deren Realisierung Datenbankdienste wie Massedatenstrukturierung und Iterationabstraktion erforderlich sind. Eine explizte Unterstützung flüchtiger und lokaler Objekte durch die Datenbanksoftware bietet dabei folgende *Vorteile*:

- Vermeidung von *Namenskonflikten* im globalen Sichtbarkeitsbereich der Datenbank.
- Automatische *Speicherfreigabe* durch das Datenbanksystem am Prozedur-, Transaktions- bzw. Prozeßende (kein explizites **drop table** erforderlich).
- Effizienzgewinn durch die Möglichkeit zur *prozeßlokalen Speicherung* (z.B. im Hauptspeicher).
- Effizienzgewinn durch die *Vermeidung von Synchronisationsoperationen* (Sperren, Nachrichten, ...) zwischen Prozessen.

Tabelle 6 faßt die in SQL-92 unterstützten Kombinationen von Tabellenlebensdauer, Tabellensichtbarkeit und gemeinsamer Nutzung sowie die jeweils zugehörige SQL-DDL-Anweisung zusammen.

In SQL läßt sich für temporäre Tabellen mit den Klauseln **on commit preserver rows** und **on commit delete rows**. die Lebensdauer flüchtiger Tabellen auf einen gesamten Prozeß oder nur eine einzelne Transaktion einschränken. Für eine Tabelle T, die mit **create ... temporary table** *T(...)* **on commit delete rows** definiert wird, muß das Datenbanksystem keine aufwendigen *Fehlererholungsinformationen* zum Rücksetzen des Tabellenzustands im Falle von Transaktionsabbrüchen speichern.

4.2.7 Standardwerte für Spalten

In SQL ist es möglich, beim Einfügen von Reihen in eine Tabelle Werte für einzelne Spalten unspezifiziert zu lassen. So läßt zum Beispiel die folgende Einfügeoperation den Wert der Spalte *Urlaub* unspezifiziert.

insert into *Mitarbeiter (Name, Gehalt, Urlaub)*
values *('Peter', 3000)*

DDL Anweisung	Tabellen-Lebensdauer	Namens-Sichtbarkeit	Gemeinsame Nutzung
create table T	persistent (Datenbank)	global (Datenbank)	ja
create global temporary table T	flüchtig (Prozeß, Transaktion)	lokal (Prozeß)	nein
create local temporary table T	flüchtig (Prozeß, Transaktion)	lokal (Modul)	nein
declare local temporary table T	flüchtig (Prozeß, Transaktion)	lokal (Prozedur)	nein

Tabelle 6: Lebensdauer, Sichtbarkeit und gemeinsame Nutzung von Tabellen

Die fehlenden Spaltenwerte werden in solchen Fällen durch den Nullwert **null** ersetzt. Es ist jedoch auch möglich, bei der Tabellenerzeugung (**create table**) einen anderen Standardwert für eine Spalte mit der Klausel **default** *Standardwert* festzulegen. Dieser Wert ist entweder ein Literal eines Basistyps oder eine *parameterlose* SQL-Funktion, die zum Zeitpunkt der Einfügeoperation ausgewertet wird und einen Datumswert (**current_date**, **current_time**, **current_timestamp**) oder eine Benutzeridentifikation (**current_user**, **session_user**, **system_user**) liefert.

Spaltenstandardwerte leisten einen nicht zu unterschätzenden Beitrag zur *Datenunabhängigkeit* und *Schemaevolution*, da existierende Anwendungsprogramme auch nach dem Erweitern einer Relation um eine neue Spalte (**alter table** *Mitarbeiter* **add column** Parkplatz **not null**) konsistent mit neu erstellten Anwendungsprogrammen interagieren können.

`Vorwärtsverweis auf Sichten. Wirklich neue Funktionalität?`

4.3 Anfragen in SQL

Dieser Abschnitt zeigt am Beispiel SQL wie Datenbanksprachen *Iterationsabstraktion* durch deklarative Anfrage- und Änderungsoperationen erreichen. SQL erzielt Iterationsabstaktion durch

- *Mengen-orientierte*, *kalkülbasierte* select from where-Anfragen
- Algebraische Mengenoperationen
- Aggregatfunktionen für Mengen
- Quantifizierte Prädikate

Das grundlegende Sprachkonstrukt von SQL ist die *mengenorientierte* select from where-Anfrage, die aus einer Projektionsliste, einer Liste von Bereichstabellen und einem Selektionsprädikat besteht, und die eine Tabelle als Anfrageergebnis liefert:

select *Projektionsliste*
from *Bereichstabellen*
where *Selektionsprädikat;*

Die vollständige Syntax und Semantik des select from where-Ausdrucks ist recht kompliziert und wird in den nachfolenden Abschnitten schrittweise eingeführt.

4.3.1 Monorelationale Select From Where-Anfragen

Monorelationale Anfragen sind Anfragen, die sich nur auf eine Bereichstabelle beziehen:

select *Projektionsliste*
from *Bereichstabelle*
where *Selektionsprädikat;*

Das Ergebnis einer monorelationalen Anfrage ist eine flüchtige, anonyme Tabelle, deren Spaltenstruktur durch die Projektionsliste bestimmt wird. Die Projektionsliste besteht aus einer durch Kommata getrennten Liste von Ausdrücken, die Werte der SQL-Basisdatentypen liefern müssen. Das *Selektionsprädikat* ist ein beliebig komplexer Boole' scher Ausdruck, der zu den Werten **true**, **false** oder **null** evaluieren kann. Für jede Reihe der Bereichstabelle, die das Selektionsprädikat erfüllt (*Selektionsprädikat* = **true**), wird die Projektionsliste ausgewertet und eine neue Reihe mit den berechneten Spaltenwerten in die Ergebnistabelle eingefügt. Die *Reihenfolge* der Reihen in der Ergebnistabelle ist nicht definiert und ist insbesondere nicht durch die Reihenfolge der Reihen in der Bereichstabelle bestimmt.

Um zum Beispiel die Namen und das Kürzel aller Abteilungen zu bestimmen, die der Abteilung „Leitung Software“ mit dem Kürzel „LTSW“ untergeordnet sind, kann die folgende SQL Anfrage verwendet werden:

select *Name, Kurz*
from *Abteilungen*
where *Oberabteilung = 'LTSW';*

Die Anfrage berechnet eine Tabelle mit den Spalten *Name, Kurz* als Ergebnis. Für jede Reihe der Bereichstabelle *Abteilungen*, für die das Prädikat *Oberabteilung='LTSW'* wahr ist, wird eine Reihe in der Ergebnistabelle mit den Werten der Spalten *Name* und *Kurz* angelegt.

Name	*Kurz*
Mainframe SW	MFSW
Unix SW	UXSW
PC SW	PCSW

Abbildung 50: Ergenisrelation einer monorelationalen Anfrage

Ein Stern (*****) in der Projektionsliste einer select from where-Anfrage entspricht der Aufzählung aller Spalten der Bereichstabelle. Die Reihenfolge der Spalten bleibt dabei erhalten, und es wird ausschließlich eine *Tabellenselektion* durchgeführt:

select *
from *Abteilungen*
where *Oberabteilung = 'LTSW';*

Das Weglassen der where-Klausel entspricht der Angabe des Selektionsprädikats **true**, sodaß für jede Reihe der Bereichstabelle eine Reihe im Anfrageergebnis auftaucht und ausschließlich eine *Tabellenprojektion* durchgeführt wird (s. Abbildung **??**):

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW

Abbildung 51: Ergenisrelation einer monorelationalen Anfrage

Oberabt
LTSW
LTSW
LTSW
NULL
NULL

Abbildung 52: Ergebnis einer monorelationalen SQL-Anfrage bei Weglassen der where-Klausel

select *Oberabteilung*
from *Abteilungen*

Eine anonyme flüchtige Kopie einer Datenbanktabelle T erhält man also durch Anfragen der folgenden Form:

select * from *T;*

Die Spaltennamen der Ergebnistabelle können bei Bedarf in der Projektionsliste explizit definiert werden:

select *Kurz* **as** *Unter, Oberabteilung* **as** *Ober*
from *Abteilungen;*

Unter	*Ober*
MFSW	LTSW
UXSW	LTSW
PCSW	LTSW
LTSW	**NULL**
PERS	**NULL**

Abbildung 53: Ergebnisrelation einer Anfrage mit expliziter Projektionsliste

4.3.2 Duplikatelimination und Sortierordnung

Das Anfrageergebnis, das in Abbildung **??** dargestellt ist, enthält dreimal den Wert „LTSW“, da die Bereichstabelle drei Reihen mit diesem Wert in der Spalte *Oberabteilung* besitzt. Verwendet man in der zugehörigen Anfrage das Schlüsselwort **distinct**, so werden im Anfrageergebnis *Duplikate* eliminiert (Zur Erinnerung: Eine Reihe ist ein Duplikat einer anderen Reihe, wenn beide in allen Spalten gemäß der Äquivalenzrelation der jeweiligen Spaltentypen übereinstimmen):

select distinct *Oberabteilung* **from** *Abteilungen*

Nach der Duplikateliminierung enthält im obigen Beispiel die Ergebnistabelle genau zwei Zeilen (*LTSW* und **null**). Allgemeiner gesagt wird durch das Schlüsselwort **distinct** eine Ergebnistabelle in eine *Ergebnismenge* umgewandelt.

Oberabt
LTSW
NULL

Abbildung 54: Ergebnisrelation bei Duplikatelimination

Wie in Abschnitt 4.2.4 erklärt wurde, lassen sich Nullwerte in Spalten durch das Prädikat **is null** erkennen und damit auch in der Ergebnistabelle oder -menge vermeiden:

select distinct *Oberabteilung*
from *Abteilungen*
where *Oberabteilung* **is not null**;

Oberabt
LTSW

Abbildung 55: Ergebnisrelation bei Duplikatelimination und Elimination von nullwertigen Ergebnissen

In manchen Fällen ist es erforderlich, Anfrageergebnisse *sortiert* darzustellen. Dies ist über die **order by** Klausel mit den Optionen **asc** (*ascending*, aufsteigend) und **desc** (*descending*, absteigend) in SQL möglich:

select * from *Abteilungen* **order by** *Kurz* **asc**;

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
PCSW	PC SW	LTSW
UXSW	Unix SW	LTSW

Abbildung 56: Ergebnisrelation bei Sortierung

Die Sortierordnung für eine Spalte wird dabei durch die Sortierordnung der zugrundeliegenden Domäne (s. Abschnitt 4.2.3) definiert. Zum Beispiel werden Zeichenketten lexikographisch sortiert, während Zahlen numerisch sortiert werden. Dabei wird der Nullwert **null** als der minimale Wert jeder Domäne betrachtet.

Die Sortierung kann mehrere Spalten umfassen. Im folgenden Beispiel werden alle Abteilungen gemäß des Kürzels ihrer Oberabteilung aufsteigend sortiert. Innerhalb einer Oberabteilung werden die Abteilungen dann absteigend nach ihrem Kürzel sortiert:

select * from *Abteilungen* **order by** *Oberabteilung* **asc**, *Kurz* **desc**;

4.3.3 Multirelationale Select From Where-Anfragen

Die Leistungsfähigkeit von select from where-Anfragen wird in *multirelationalen* Anfragen deutlich. Dies sind Anfragen, in denen Spalten und Reihen mehrerer Bereichstabellen verknüpft werden. Solche Verknüpfungen sind zum Beispiel notwendig, um Anfragen über *Objektbeziehungen* im Relationalen Modell zu formulieren.

select *Projektionsliste*
from $T_1, \ T_2, \ldots \ T_n$
where *Selektionsprädikat;*

Die Typisierungs- und Auswertungsregeln (Selektionsprädikat, Projektionsliste, Reihenfolge) für multirelationale Anfragen entsprechen den Regeln für monorelationale Anfragen (s. 4.3.1) mit dem Unterschied, daß das Selektionsprädikat und die Projektionsliste für alle möglichen Kombinationen der Reihen der n Bereichstabellen ausgewertet werden. Konzeptionell findet also eine Selektion und Projektion über das *Kartesische Produkt* der Bereichstabellen statt.

Zur Verdeutlichung ist nachfolgend das Ergebnis der Berechnung des Kartesischen Produkts T = R x S der Tabellen R und S mit der SQL-Anfrage **select * from** *R, S* dargestellt:

R
1
2
3

S
A
B
C
null

T	
1	A
2	A
3	A
1	B
2	B
3	B
1	C
2	C
3	C
1	**null**
2	**null**
3	**null**

Beispiel für Firmendatenbank:

select *Name, Kurz*
from *Mitarbeiter, Abteilung*
where *Gehalt* > *3000*

Duplikatelimination und Sortierordnung können auch bei multirelaltionalen Anfragen wie in Abschnitt 4.3.2 beschrieben spezifiziert werden.

Verwendet man einen Stern (*) in der Projektionsliste einer multirelationalen Anfrage, so besitzt die Ergebnistabelle alle Spalten der Bereichstabellen in der Reihenfolge, in der die Bereichstabellen in der from-Klausel aufgelistet wurden. Insbesondere können dabei *Spaltennamensduplikate* in der Ergebnistabelle auftreten, die eine Weiterbearbeitung der Ergebnistabelle erschweren.

Anfragen über n≥2 Bereichstabellen werden auch als n-Weg-Joins (*n way joins*) bezeichnet.

Die bisher vorgestellten Beispiele multirelationaler Anfragen benutzen als Selektionsprädikat einen Gleichheitstest (=) zwischen Spaltenwerten und werden daher auch als *Equi-Joins* bezeichnet. Dabei handelt es sich bei den Spalten in den Beispielen um je einen Fremdschlüssel und einen Primärschlüssel, so daß das Anfrageergebnis Informationen über die *Beziehungen* zwischen Objekten in der Datenbank tabellenförmig darstellt.

Wesentlich seltener werden Equi-Joins zwischen Nicht-Schlüsselspalten durchgeführt. Ein Beispiel dafür ist die folgende Anfrage, die die Namen der Personen, die in der Stadt ihrer Firma wohnen, bestimmt:

select *p.Nr p.Titel p.Budget q.Kurz*
from *Projekte* **as** *p*, *Projektdurchführung* **as** *q*
where *p.Nr* = *q.Nr*

Projekte

Nr	*Titel*	*Budget*
100	DB Fahrpläne	300.000
200	ADAC Kundenstamm	100.000
300	Telekom Statistik	200.000

Projektdurchführung (Ausschnitt)

Nr	*Kurz*
100	MFSW
200	PERS
300	MFSW

Ergebnisrelation

Nr	*Titel*	*Budget*	*Nr2*	*Kurz*
100	DB Fahrpläne	300.000	100	MFSW
200	ADAC Kundenstamm	100.000	200	PERS
300	Telekom Statistik	200.000	300	MFSW

Abbildung 57: Ergebnisrelation einer Equi-Joinoperation

Joinoperationen, die anstelle eines Gleichheitstests (=) ein anderes zweistelliges Boole'sches Prädikat θ (<, >, >=, <=, <>, like, ...) zum Vergleich der Spaltenwerte zweier Tabellen verwenden, werden zusammenfassend auch als *Theta-Joins* (θ*-joins* bezeichnet.

Die folgende Anfrage bestimmt zum Beispiel die Namen der Personen, deren Monatsgehalt das Budget ihrer Abteilung übersteigt:

select *Name*
from *Mitarbeiter* **as** *m*, *Abteilung* **as** *a*
where *m.gehalt* > *a.budget*

Projekte

Nr	*Titel*	*Budget*
100	DB Fahrpläne	300.000
200	ADAC Kundenstamm	100.000
300	Telekom Statistik	200.000

Projektdurchführung (Ausschnitt)

Nr	*Kurz*
100	MFSW
200	PERS
300	MFSW

Ergebnisrelation Projekte × Projektdurchführung

Nr	*Titel*	*Budget*	*Nr2*	*Kurz*
100	DB Fahrpläne	300.000	100	MFSW
100	DB Fahrpläne	300.000	200	PERS
100	DB Fahrpläne	300.000	300	MFSW
200	ADAC Kundenstamm	100.000	100	MFSW
200	ADAC Kundenstamm	100.000	200	PERS
200	ADAC Kundenstamm	100.000	300	MFSW
300	Telekom Statistik	200.000	100	MFSW
300	Telekom Statistik	200.000	200	PERS
300	Telekom Statistik	200.000	300	MFSW

Abbildung 58: Ergebnisrelation einer Joinoperation

4.3.4 Sichtbarkeitsregeln und Spaltennamenkonflikte

Abschnitt 4.2.2 behandelt ausführlich die Sichtbarkeit der *globalen* Namen von SQL-Objekten, die in einem SQL-Schema definiert werden. In diesem Abschnitt werden die Sichtbarkeitsregeln für *lokale* Namen (Spaltennamen, Bereichsvariablennamen) innerhalb von kalkülorientierten SQL-Anfragen erklärt.

Man betrachte hierzu die allgemeine Form einer select from where-Anfrage:

select P
from T_1, T_2,..., T_n
where S;

In den Teilausdrücken P, S, T_1, T_2,..., T_n sind alle globalen Namen von SQL-Objekten (Tabellen, Sichten, Schemata, Kataloge) sichtbar. Im Selektionsprädikat S und in der Projektionsliste P sind zusätzlich die *lokalen* Namen aller Spalten aller Bereichstabellen T_i sichtbar. Dabei *überdeckt* ein lokaler Name einen globalen Namen.

Bei der Evolution großer Datenbankschemata treten unweigerlich Namenskonflikte zwischen den Spaltennamen verschiedener Tabellen sowie zwischen Spaltennamen und globalen Namen auf. Daher existiert in SQL-Anfragen die Möglichkeit zur Definition lokaler *Bereichsvariablen* (*correlation names, alias names*), die zur Qualifizierung von Spaltennamen mittels Punktnotation im Selektionsprädikat und der Projektionsliste eingesetzt werden können:

select P
from T_1 **as** X_1, T_2 **as** X_2,..., T_n **as** X_n
where S;

Bei dieser Form von SQL-Anfragen sind im Selektionsprädikat S und in der Projektionsliste P zusätzlich zu den globalen Namen nur die lokalen Bereichsvariablen X_i sichtbar. Wiederum überdeckt ein lokaler Bereichsvariablenname einen globalen Namen.

select *Mitarbeiter.**
from *Angestellte* **as** *Mitarbeiter*
where *Mitarbeiter.gehalt* > *3000;*

Der Bezeichner *Mitarbeiter* der obigen Anfrage überdeckt in diesem Fall den Namen der gleichlautenden Tabelle *Mitarbeiter*, die im Schema des Firmeninformationssystems definiert ist.

Mithilfe der Notation *Bereichsvariable.** ist es möglich, in der Projektionsliste alle Attribute einer Bereichstabelle zu bezeichnen. In folgendem Beispiel werden durch *m.** die Attribute *m.Name*, *m.Alter* und *m.Gehalt* spezifiziert.

select *m.**
from *Mitarbeiter* **as** *m*
where *m.gehalt* > *3000;*

Abschließend sei erwähnt, daß es sich in der Praxis aus mehreren Gründen als sinnvoll erwiesen hat, in SQL-Anfragen *immer* Bereichsvariablen zu verwenden:

- *Lesbarkeit*: Es ist unmittelbar (ohne Nachschlagen im DB-Schema) zu erkennen, zu welcher Bereichstabelle ein Spaltenname gehört.
- *Wartbarkeit*: Probleme durch Spaltennamenkonflite und das unerwartete Verdecken von Tabellennamen bei Schemaänderungen (Anfügen neuer Spalten oder beim Umbenennen von Spalten) werden für die gesamte Lebensdauer der Datenbank ausgeschlossen.
- *Ausdrucksmächtigkeit*: Es wird möglich, *reflexive* Anfragen zu formulieren. Dies sind Anfragen, in denen *mehrere* Reihen *einer* Tabelle verknüpft werden, zum Beispiel, um mit rekursiven Beziehungen zwischen Datenbankobjekten zu arbeiten.

  Zum Beispiel liefert die folgende Anfrage die Namen von Ober- und Unterabteilungen in tabellarischer Form.

select *o.Name* **as** *Oberabteilung, u.Name* **as** *Unterabteilung*
from *Abteilung* **as** *o, Abteilung* **as** *u*
where *u.Oberabteilung* = *o.Kurz*

Die genannten Vorteile von Bereichsvariablen haben dazu geführt, daß praktisch alle modernen kalkülorientierten Datenbanksprachen (SQL, QUEL, QBE) die Benutzung von Bereichsvariablen erzwingen.

4.3.5 Einfache und quantifizierte Prädikate

`Formulierung Todo !!!!` SQL-Prädikate entsprechen Boole' schen Ausdrücken in Programmiersprachen. Leistungsmächtiger weil wertorientiert und Iterationsabstraktion.

Stark schlüsselwortorientiert, Bindungsstärke nicht hier erläutert.

Zu den meisten der im folgenden betrachteten Operatoren gibt es ein negiertes Gegenstück, deren Semantik in Tabelle 7 betrachtet wird.

Prädikat	x **is null**	x **is not null**	**not** x **is null**	**not** x **is not null**
x ist gleich dem Nullwert	**true**	**false**	**false**	**true**
x ist nicht der Nullwert	**false**	**true**	**true**	**false**
$x = (x_1, x_2, x_3)$, kein Wert x_i ist gleich dem Nullwert	**false**	**true**	**true**	**false**
$x = (x_1, x_2, x_3)$, mindestens ein Wert x_i ist gleich dem Nullwert	**false**	**false**	**true**	**true**
$x = (x_1, x_2, x_3)$, alle Werte x_i sind gleich dem Nullwert	**true**	**false**	**false**	**true**

Tabelle 7: Semantik des Nullwerttests

Vergleichsoperatoren In SQL gibt es die üblichen Operatoren für den Vergleich $a \ \theta \ b$ zweier Werte a und b. Dabei können für θ die allgemein bekannten Operatoren $<$, $>$, $>=$, $<=$, und $<>$ eingesetzt werden.

Eine wichtige Rolle spielen in SQL die Nullwerte. Es gibt daher spezielle Vergleichsoperatoren für den Test auf Nullwerte.

x **is null**
x **is not null**

Die Operatoren **null** und **not null** können auch auf Werte mit mehreren Spalten angewandt werden. Die resultierende Semantik ist dabei allerdings nicht ganz selbstverständlich und wird deshalb in Tabelle 7 erläutert.

Neben diesen allgemeinen Vergleichsoperatoren werden auch solche unterstützt, die auf spezielle Datentypen ausgerichtet sind. Der wichtigste Operator dieser Art ist **like**, das einen partiellen Vergleich von Zeichenketten erlaubt. Es wird wie die anderen Operatoren in Infixnotation benutzt. Die mit *a* **like** *b* verglichenen Zeichenketten können die *wildcards* % und _ enthalten, wobei _ für einen beliebigen Buchstaben und % für keinen, einen oder beliebig viele beliebige Buchstaben steht. In einem Zeichenstring können mehrere dieser Zeichen vorkommen, wodurch sich eine hohe Flexibilität in der Formulierung partieller Suchanfragen ergibt.

x **like** *'%Software Version __'*

Mit dem Operator **between** kann überprüft werden, ob ein Wert *x* in einem durch zwei Werte *a* und *b* spezifizierten geschlossenen Intervall liegt:

x **between** a **and** b

Dieses Prädikat ist äquivalent zu

$a <= x$ **and** $x <= b$

Der between-Operator existiert auch in negierter Form. Syntax und Semantik sind in Tabelle 8 aufgeführt.

Ebenfalls mit Intervallen befaßt sich der Operator **overlaps**.

a **overlaps** b

Er überprüft, ob sich zwei Zeitintervalle a und b überlappen und ist damit besonders für den Bereich der Terminplanung geeignet. Die verglichenen Zeitintervalle können dabei entweder durch einen Start- und einen Endzeitpunkt oder durch einen Startzeitpunkt und die Dauer des Intervalls festgelegt werden, wobei a und b nicht auf die gleiche Weise spezifiziert sein müssen. Zwei Intervalle überlappen nicht, wenn sie sich nur berühren, d.h. wenn der Endzeitpunkt des früheren Intervalls der Startzeitpunkt des späteren Intervalls ist.

Enthaltenseinstest: Der in-Operator überprüft, ob ein Wert x in einer Menge m von anderen Werten enthalten ist. Die Menge m kann dabei wie im folgenden Beispiel durch die Aufzählung ihrer Elemente angegeben werden.

x **in** *(1, 2, 4, 8, 16, 32)*

Es ist aber auch möglich, eine Unteranfrage (*subquery*) anzugeben, deren Ergebnis die untersuchte Menge m bildet.

x **in** *(***select** *name* **from** *Mitarbeiter* **where** *gehalt > 3000)*

Quantifizierte Prädikate: SQL unterstützt auch Operatoren, die (mit einigen Einschränkungen) die Formulierung von quantifizierten Prädikaten erlaubt. Ein Prädikat $x\,\theta$ **all** s das den Operator *all* enthält[10], entspricht in etwa dem folgenden universell quantifizierten Prädikat mit einer impliziten Bereichsvariablen y, die mit dem universellen Quantor an das Anfrageergebnis gebunden ist.

$\forall\, y \in s : x\,\theta\, y$

Das Prädikat ist also nur erfüllt, wenn für alle Elemente y des Anfrageergebnisses x θ y gilt.

Analog dazu ist auch die Formulierung existentiell quantifizierter Prädikate in SQL möglich. Wie der Tabelle 8 zu entnehmen ist, kann dafür wahlweise der Operator **some** oder der Operator **any** benutzt werden. Das Prädikat $x\ \theta$ **some** s ergibt also den Wahrheitswert *true*, wenn für mindestens ein Ergebniselement y zur Anfrage s die Bedingung $x\ \theta\ y$ erfüllt ist.

[10] Für θ sind dabei wieder die bereits weiter oben aufgelisteten Vergleichsoperatoren zulässig.

Prädikat	**Äquivalentes Prädikat**	**Bemerkung**
x **is between** a **and** b x **is not between** a **and** b	a <= x **and** x <= b **not**(x **between** a **and** b)	
a in m a **not in** m	a = m_1 **or** a = m_2 **or** ... a= m_n mit m = $\{m_1, m_2, \ldots, m_n\}$ not(a **in** m)	
(s_1, e_1) **overlaps** (s_2, e_2)	($s_1 < s_2$ **and** $e_1 > s_2$) **or** ($s_2 < s_1$ **and** $e_2 > s_1$)	
is not null		
x θ **all** s	$\forall\, y \in s : x\, \theta\, y$	mit Einschränkung
x θ **some** s x θ **any** s	$\exists\, y \in s : x\, \theta\, y$ $\exists\, t \in s : x\, \theta\, t$	

Tabelle 8: Äquivalente Prädikate

Test auf Leerheit und Eindeutigkeit: Der Operator **exists** suggeriert durch seinen Namen zwar eine Verbindung zur existentiellen Quantizierung, dient jedoch in SQL dazu, die Leerheit eines Anfrageergebnisses zu testen. Er kann auf Unteranfragen (*subqueries*) angewandt werden und liefert den Wahrheitswert *true*, falls das Ergebnis dieser Anfrage mindestens ein Element enthält. So überprüft die folgende Anfrage, ob es einen Mitarbeiter gibt, der mehr als 10000 DM verdient.

exists (**select** * **from** *Mitarbeiter* **where** *gehalt > 10000*

Der Präfixoperator **unique** wird ebenfalls auf Unteranfragen angewandt und überprüft, ob das Anfrageergebnis Duplikate enthält. Er ist nicht zu verwechseln mit dem Operator **distinct**, der eine Duplikatelimination durchführt (s. Abschnitt 4.3.2).

Aus den beschriebenen Prädikaten können durch konjunktive (**and**) und disjunktive (**or**) Verknüpfung komplexere Prädikate aufgebaut werden. Außerdem kann jedes Prädikat durch Voranstellen des not-Operators negiert werden. Wie in Abschnitt 4.2.4 erläutert, ist dabei zu beachten, daß es sich hier wegen der Existenz der Nullwerte um eine dreiwertige Logik handelt. Die entsprechenden Wahrheitstabellen für die Operatoren **and**, **or** und **not** sind in Abschnitt 4.2.4 zu finden.

aussagenlogisch, prädikatenlogik 1. Stufe.

Generelles Problem: Sichtbarkeitsbereich gebundener Variablen. Analogie: Blockstrukturierung in Programmiersprachen. Wie in## erklärt. Menge der globalen Variablen enthält auch lokale Variablen des umfassenden Blocks.

4.3.6 Algebraische Tabellenoperationen

Die aus der Mengenalgebra bekannten Operationen Mengenvereinigung, Schnittmengenbildung und Mengendifferenz werden in SQL-92 auf Tabellen übertragen. Mit den Operatoren **union**, **intersect** und **except** können Tabellen in Anfragen miteinander verknüpft werden.

In der folgenden Anfrage werden die Mengen der Mitarbeiter zweier Abteilungen miteinander verknüpft.

select *
from *MitarbeiterAbt1*
union
select *
from *MitarbeiterAbt2*

In Anlehnung an die Mengenvereinigung führt die Operation **union** automatisch eine Duplikatelimination durch. Es existiert jedoch die zusätzliche Option **all**, um diese Duplikatelimination zu unterdrücken. Die Option **all** kann auch im Zusammenhang mit dem Operator **intersect** verwendet werden.

Voraussetzung für die Verknüpfung mit einer der algebraischen Tabellenoperationen ist die Kompatibilität der Spaltenstruktur der beteiligten Tabellen. Dies erfordert unter anderem, daß die Spalten der Tabellen die gleichen Namen und Datentypen besitzen müssen. Die Forderung nach der Kompatibiltät kann auf einen Teil der Spalten eingeschränkt werden, indem man den jeweiligen Operator um eine corresponding-Klausel erweitert. Es werden dann nur die kompatiblen Spalten der beiden Tabellen berücksichtigt und die Ergebnistabelle enthält ebenfalls nur die kompatiblen Spalten.

x

Eine weitere Einschränkung ist durch die explizite Nennung von Spaltennamen in der corresponding-Klausel möglich. Sowohl die Kompatibilitätsprüfung als auch das Ergebnis beziehen sich in diesem Fall ausschließlich auf die genannten Spalten.

Eine weitere wichtige Klasse algebraischer Tabellenoperationen sind die bereits in Abschnitt 3.2.7 erwähnten Join-Operationen. Wegen der zentralen Bedeutung dieser Operationen für das relationale Modell und der zahlreichen Varianten, die in SQL-92 dafür vorgesehen sind, ist diesen Operationen ein eigener Abschnitt (4.3.3) gewidmet.

4.3.7 Join-Operationen

`Außerdem SQL-92:  (152-162) Komplexe Sichtbarkeitsregeln fuer Zielausdruck`

Die Grundlage von *Join-Operationen* ist die Bildung des kartesischen Produkts über zwei oder mehr Tabellen. In den meisten Fällen wird dieses Produkt durch explizit spezifizierte oder implizit im speziellen Operator enthaltenen *Join-Bedingungen* auf einen Ausschnitt eingeschränkt. Join-Operationen werden in der from-Klausel verwendet (??). `Vielleicht noch etwas mehr grundsätzliches ??`

SQL-92 führt eine explizite Behandlung von Join-Operationen mit einer Vielzahl verschiedener Varianten ein. Die Möglichkeit Join-Operationen auszudrücken ist jedoch zentral für das relationale Modell und existiert bereits in früheren SQL-Versionen. Wie in Abschnitt 4.3.3 erwähnt, genügt es, in der from-Klausel mehrere Tabellen aufzuführen, was der Bildung des kartesischen Produkts über diese Tabellen entspricht. Die Join-Bedingung kann in der where-Klausel der Anfrage festgelegt werden.

Die unterschiedlichen Varianten der Join-Operationen in SQL-92 lassen sich grundsätzlich in zwei Klassen einteilen.

- Das Ergebnis eines *inner Join* enthält nur den Teil des kartesischen Produkts, der die Join-Bedingungen erfüllt.

- Beim *outer Join* wird noch weitere Information aus den beteiligten Tabellen in das Ergebnis aufgenommen.

Die Varianten des *inner* Join, von denen die wichtigsten im folgenden kurz beschrieben werden, unterscheiden sich hauptsächlich in der Handhabung der Join-Bedingung (z.B. explizite oder implizite Angabe von Join-Bedingungen).

Kartesisches Produkt: Der einfachste Join-Operator ist der *cross join*. Er dient zur Bildung des kartesisches Produkt aus den beteiligten Tabellen. Die Verwendung dieses Operators ist äquivalent zur einfachen Auflistung mehrerer Tabellen in der from-Klausel, macht jedoch die Durchführung des Joins explizit. Einschränkende Join-Bedingungen werden in beiden Fällen durch die where-Klausel festgelegt.

Join mit Bedingung: Dieser Operator enthält eine on-Klausel, in der eine beliebige Join-Bedingung festgelegt werden kann. Wie im folgenden Beispiel zu sehen ist, werden die beteiligten Tabellen durch den Operator *join* miteinander verbunden.

`Beispiel mit Gleichsetzen bestimmter Spalten Beachte Scopes`

Die Join-Bedingung kann äquivalent auch über die where-Klausel festgelegt werden. Die Verwendung der on-Klausel gestattet jedoch eine explizite Zuordnung der Bedingung zur Join-Operation, während weitere Einschränkungen der durch die Join-Operation entstandenen Tabelle in der where-Klausel zusammengefaßt werden können.

Join mit Bedingung ist der allgemeinste Join-Operator in der Klasse der *inner joins*; alle übrigen Operatoren lassen sich mit Hilfe dieses Operators ausdrücken, wenn auch nicht in generischer Form.

Natural Join: Der *natural Join*, auch *natural equi-join* genannt, und der *Join mit Spaltennamen* (s.u.) besitzen implizite Join-Bedingungen als Teil der Operatorensemantik. Sie basieren auf der Annahme, daß Spalten die den gleichen Namen besitzen, auch vergleichbare Inhalte enthalten. Bei der Ausführung eines *natural* Join werden nämlich alle Spalten mit den gleichen Spaltennamen verglichen. Alle Paare von Reihen der ersten und zweiten Tabelle, die in all diesen Spalten die gleiche Werte besitzen, werden in die Ergebnistabelle aufgenommen. Zusätzlich wird auf der Ergebnistabelle eine Projektion durchgeführt, die verhindert, daß die gleichnamigen Spalten im Ergebnis doppelt auftreten.

`Hauptanwendung??  Beipiel??`

Äquivalentes Attribut, evtl. in Tabelle ???

t1 natural join t2 select from t1 as x t2 as y where x.a = y.a and x.b =y.b

wobei a und b Attribute sowohl von t1 als auch von t2 sind.

Beispiel

Join mit Spaltennamen Diese Variante der Join-Operation läßt eine Einschränkung des Natural Joins auf einen Teil der gleichnamigen Spalten zu. Die implizite Join-Bedingung

Join-Operation	Bezeichnung	Äquivalente Operation	Bemerkung
select P(t_1, t_2) **from** t_1 **natural join** t_2	*natural* Join **from** t_1, t_2	**select**	 **where** XXX

Tabelle 9: Join-Operationen in SQL-92

vergleicht nicht wie beim Natural Join alle gleichnamigen Spalten, sondern nur die, deren Namen explizit in der using-Klausel des Operators aufgeführt wird.

t1 join t2 using (a,b)

`Beachte Nutzung der Spatenbezeichner im where-Prädikat !!  s.S. 160`

Beim *inner* Join ist die Information aus Zeilen der verbundenen Tabellen, die nicht die Join-Bedingung erfüllen, nicht in der Ergebnistabelle enthalten. Idee des *outer* Joins ist es, diese Information (oder zumindest ein Teil davon) mit in die Ergebnistabelle aufzunehmen. Die Join-Bedingung wird beim *outer* Join ebenfalls in einer on-Klausel festgelegt. (vgl. Join mit Bedingung).

Zur Erläuterung des *outer* Join wird von zwei Tabellen t_1 und t_2 mit n bzw. m Spalten ausgegangen. Wenn keine Projektion durchgeführt wird, hat die Ergebnistabelle n+m Spalten. Es wird dabei zwischen dem *left outer Join*, dem *right outer Join* und dem *full outer Join* unterschieden.

Die Ergebnistabelle eines *left outer* Joins kann man sich als aus zwei Teilen a und b zusammengesetzt vorstellen. Der erste Teil (a) entspricht dem Ergebnis eines *inner* Joins mit der gleichen Join-Bedingung. Teil b enthält all die Reihen der Tabelle t_1, die in Teil a nicht auftreten. Diese Information füllt aber nur die ersten n Spalten dieses Teils der Ergebnistabelle, die übrigen m Spalten werden mit Nullwerten aufgefüllt.

Das Ergenis eines *right outer* Joins setzt sich analog zusammen, nur daß der Teil b der Ergebnistabelle die nicht berücksichtigten Zeilen der Tabelle t_2 enthält, d.h. in diesem Fall werden die ersten n Spalten mit Nullwerten aufgefüllt.

Beim *full outer* Join setzt sich das Ergebnis aus drei Teilen zusammen: den beiden Teilen, die beim *left outer* Join auftreten plus dem zweiten Teil des Ergebnisses des *right outer* Joins.

Im Gegensatz zum *inner* Join sind die *outer* Join-Operatoren nicht assoziativ.

Die folgende Tabelle faßt noch einmal die einzelnen Varianten der Join-Operationen zusammen. Es wird dabei von einer Tabelle t_1 mit den Spalte a, b, c und einer Tabelle t_2 mit den Spalten b, c, d, e ausgegangen.

t1 natural join t2 select from t1 as x t2 as y where x.a = y.a and x.b =y.b

bestimmte Join-Operationen sind besonders wichtig z.B. die Verbindung von Tabellen die Beziehungen darstellen, oder die Verbindung von Information, die aufgrund von Normalisierung in mehrere Tabellen verteilt worden sind.

4.3.8 Aggregatfunktionen

In der Projektionsliste (select-Klausel) einer SQL-Anfrage können sogenannte *Aggregatfunktionen* benutzt werden. Diese Funktionen dienen der Berechnung aggregierter Werte wie z.B.

der Summe über alle Werte einer Spalte in einer Tabelle. Die folgende Anfrage berechnet z.B. die Summe und das Maximum der Budgets aller Projekte.

select **sum**(*p.Budget*), **max**(*p.Budget*)
from *Projekte* **as** *p*

sum	max
600.000	300.000

Abbildung 59: Ergebnisrelation

SQL unterstützt neben Funktionen zum Berechnen der Summe (**sum**), des Maximums (**max**), des Minimums (**min**) und des Durchschnitts (**avg**) auch eine Funktion **count** zum Zählen der Werte in einer Tabelle. Die Funktion **count** kann anders als die anderen Funktionen nicht nur auf eine einzelne Spalte sondern durch den Aufruf **count**(*) auch auf gesamte Tupel angewandt werden. So ermittelt die folgende Anfrage die Anzahl der Tupel in der Relation *Abteilungen*.

select **count**(*)
from *Abteilungen*

count(*)
5

Abbildung 60: Ergebnisrelation

In manchen Situationen ist es notwendig beim Zählen von Werten Duplikate auszuschließen. So ist z.B. die folgende Anfrage zur Berechnung der Anzahl aller Oberabteilungen ungeeignet, da sie Abteilungen mit mehreren Unterabteilungen mehrfach zählt.

select **count**(*a.Oberabteilung*)
from *Abteilungen* **as** *a*
where *a.Oberabteilung* **is not null**

Um dies zu verhindern, bietet SQL das Schlüsselwort **distict** an.

select **count**(**distinct** *a.Oberabteilung*)
from *Abteilungen* **as** *a*
where *a.Oberabteilung* **is not null**

In diesem Fall werden mehrfach auftretende Spaltenwerte nur einfach gezählt (vgl. Duplikatelimination in Abschnitt 4.3.2). **distinct** kann mit der entsprechenden Semantik auch mit den anderen Aggregatfunktionen kombiniert werden.

4.3.9 Gruppierung

Zeilen einer Tabelle werden abhängig von den Werten in bestimmten Spalten, den sogenannten *Gruppierungsspalten*, zu Gruppen zusammengefaßt. Alle Zeilen einer Gruppe enthalten in dieser Spalte bzw. in diesen Spalten den gleichen Wert. Man erhält auf diese Weise eine Tabelle von Gruppen als Zwischenergebnis.

Wenn eine Anfrage eine group by-Klausel enthält, gibt es einige Einschränkungen. Für die select-Klausel sind in diesem Fall nur Ausdrücke der folgenden Art zulässig:

- Spalten aus der group by-Klausel,
- Aufrufe von Aggregatfunktionen und
- Konstanten;

Aggregatfunktionen werden bei gruppierten Tabellen auf die einzelnen Gruppen angewandt. Die Ergebnistabelle enthält also einen aggregierten Wert pro Gruppe.

select *Oberabt*, **count***(Kurz)*
from *Abteilungen*
group by *Oberabt;*

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	**NULL**
PERS	Personal	**NULL**

Ergebnisrelation

Oberabt	*count(Kurz)*
LTSW	3
NULL	2

Abbildung 61: Ergenisrelation

Zusätzlich ist es auch möglich, Bedingungen für die Gruppen einer gruppierten Tabellen festzulegen. Zu diesem Zweck wird die having-Klausel verwendet. Sie definiert einen Filter für die Gruppen. Nur Gruppen, die die Filterbedingung erfüllen, werden im Ergebnis der Anfrage berücksichtigt. Die mit der having-Klausel eingeführte Bedingung wird nicht auf einzelne Zeilen, sondern auf Gruppen angewandt. Es können deshalb in der Bedingung nur Gruppierungsspalten direkt referenziert werden[11]. Auf andere Spalten kann nur als Parameter einer Aggregatfunktion zugegriffen werden.

Wenn eine SQL-Anfrage zwar eine having-Klausel, aber keine group by-Klausel enthält, wird die ganze Tabelle als eine Gruppe betrachtet.

4.4 Änderungsoperationen in SQL

Die folgenden Abschnitte behandeln Operationen, die den Inhalt von Tabellen ändern. Sie beziehen sich jeweils auf eine einzelne Relation.

[11]Diese haben für alle Reihen einer Gruppe denselben Wert.

4.4.1 update-Anweisung

Die update-Anweisung dient zur Modifikation von Attributwerten. Der Effekt einer update-Anweisung ist dabei nicht notwendigerweise auf ein einzelnes Tupel beschränkt. Vielmehr kann die Menge der betroffenen Tupel ähnlich wie bei einer SQL-Anfrage durch ein beliebiges, in einer where-Klausel spezifiziertes Prädikat festgelegt werden.

Eine update-Anweisung besteht aus drei Teilen. Im ersten Teil wird die betroffene Tabelle festgelegt. Der zweite Teil wird durch das Schlüsselwort **set** eingeleitet. In ihm findet die Zuweisung der neuen Attributwerte statt.

Alle Tupel, die das Prädikat, welches mit einer where-Klausel formuliert wird, erfüllen, werden modifiziert.

Im einfachsten Fall wird einem Attribut unabhängig von seinem bisherigen Wert ein neuer Wert zugeordnet. So erhält durch die folgende Anweisung das Projekt „DB Fahrplaene“ ein neues Budget von DM 500.000.

update *Projekte*
set *Budget = 500000*
where *Titel = 'DB Fahrplaene';*

Es ist aber auch möglich den neuen Attributwert in Abhängigkeit vom alten Attributwert festzulegen, wie dies im folgenden Beispiel geschieht.

update *Mitarbeiter*
set *Gehalt = Gehalt + 100*
where *Gehalt* $\leq$ *3000*

Der dritte, optionale Teil einer update-Operation ist die bereits erwähnte where-Klausel zur Selektion der betroffenen Tupel. Wenn die where-Klausel fehlt werden alle Tupel der Tabelle modifiziert.

Eine Verwendung von Unterabfragen in der where-Klausel ist möglich. Diese Unterliegen in diesem Fall aber einigen Einschränkungen.

4.4.2 insert-Anweisung

Die Position des eingefügten Tupels in der Tabelle ist unspezifiziert. Es kann z.B. nicht davon ausgegangen werden, daß neue Tupel hinten an die Tabelle angehängt werden, oder daß eine Sortierung nach den Primärschlüsseln aufrechterhalten wird. Die Positionierung neuer Tupel ist allein Sache der Implementierung des DBMS.

Wenn Attributnamen angegeben werden, ist es nicht notwendig, Rücksicht auf Reihenfolge der Spalten in der Tabelle zu nehmen. Wird für alle Attribute ein Wert festgelegt und geschieht die Nennung der Werte exakt in der Reihenfolge, in der die zugehörigen Spalten in der Tabelle auftreten, kann auf eine Angabe der Attributnamen verzichtet werden. Die explizite Nennung der Attributnamen in der insert-Anweisung besitzt jedoch die folgenden Vorteile.

- Vermeidung von Fehlern in der Datenbank durch die irrtümliche Vertauschung von Spalten, die zufällig den gleichen Datentyp besitzen.

- Verbesserung der Robustheit gegen Schemamodifikationen wie z.B. die Umordnung von Reihen.
- Erhöhung der *Lesbarkeit* der Operationen, der Benutzer braucht sich nicht die Reihenfolge der Spalten zu merken oder nachzusehen.

Im folgenden Beispiel wird ein neues Element in die Tabelle der Projekte eingefügt:

insert into *Projekte*
(Nr, Titel, Budget)
values *(471, 'Mensaabrechnung', 90000);*

4.4.3 delete-Anweisung

Die delete-Anweisung zum Löschen von Datenelementen ist in ihrem strukturellen Aufbau der select from where-Anweisung sehr ähnlich. Sie besitzt ebenfalls eine from-Klausel zur Festlegung der betroffenen Tabelle und eine where-Klausel zur Angabe eines Selektionsprädikates. Da alle Modifikationsoperationen auf eine einzelne Tabelle beschränkt sind, ist die Angabe mehrerer Tabellen in der from-Klausel anders als bei den Anfragen bei dieser Operation nicht zulässig.

Das folgende Beispiel zeigt, wie alle Tupel der Tabelle, die das Selektionsprädikat erfüllen, durch die delete-Anweisung gelöscht werden. Das zuvor eingefügte Projekt „Mensaabrechnung“ wird aus der Tabelle *Projekte* gelöscht.

delete
from *Projekte*
where *Titel = 'Mensaabrechnung';*

Die Operation **delete** dient zum Löschen einzelner Tupel aus einer Tabelle. In SQL können jedoch Objekte sehr unterschiedlicher Granularität gelöscht werden. Die meisten der entsprechenden Operationen sind bereits in den vorherigen Abschnitten beschrieben worden, werden aber hier noch einmal zusammengefaßt.

- Zum Löschen einzelner *Attributwerte* können diese Attribute mit der update-Operation auf den Wert *null* gesetzt werden (s. Abschnitt 4.4.1).
- Das Löschen eines oder mehrerer ganzer *Tupel* geschieht mit der in diesem Abschnitt beschriebenen delete-Anweisung. Welche Tupel gelöscht werden, wird durch die where-Klausel der Anweisung festgelegt.
- Löschen aller *Werte einer Spalte*.
- Ganze *Spalten* können ebenfalls aus einer Tabelle entfernt werden. Dies entspricht einer Schemaänderung und wird deshalb alter table drop column (s.Abschnitt 4.2.2) genannt.
- Löschen des gesamten *Inhalts einer Tabelle* entspricht einer delete-Anweisung ohne where-Klausel.
- Löschen einer ganzen *Tabelle* mitsamt der Datenstruktur wird durch drop table (s. Abschnitt 4.2.2) erreicht.

4.5 Integritätssicherung in SQL

Bereits bei der Übersetzung jeder SQL-Anweisungen erzwingt SQL die folgenden *modellinhärenten Integritätsbedingungen* durch eine textuelle Analyse der Anweisung unter Benutzung von Schemainformationen analog zur *statischen Typisierung* in Programmiersprachen:

Typisierung der Spalten: In einer Spalte können nur typkompatible Werte gespeichert werden.

Homogenität der Reihen: Alle Reihen einer Tabelle besitzen eine identische Spaltenstruktur.

Zur Definition *applikationsspezifischer Integritätsbedingungen* bietet SQL zwei syntaktische Konstrukte, die beide Boole' sche Prädikate zur *deklarativen* Integritätssicherung benutzen, und die durch Laufzeittests erzwungen werden:

Tabellenzusicherungen werden syntaktisch in Tabellendefinitionen geschachtelt spezifiziert:

create table *Tabellenname*
  (...
  constraint *Zusicherungsname* **check***(Prädikat))*

Eine Tabellenzusicherung garantiert, daß in jedem Datenbankzustand die Auswertung des Prädikats für *alle* Reihen der Tabelle den Wert **true** liefert. Man sagt daher auch, daß das Prädikat *universell quantifiziert* ist. Das Prädikat kann sich auf die Spaltennamen der Tabelle und auf die globalen Namen des aktuellen SQL-Schemas beziehen (z.B. Tabellennamen).

Schemazusicherungen sind SQL-Objekte, die dynamisch dem aktuellen SQL-Schema hinzugefügt werden können:

create assertion *Zusicherungsname*
  check*(Prädikat);*

Eine Schemazusicherung garantiert, daß in jedem Datenbankzustand die Auswertung des Prädikats den Wert **true** liefert. Das Prädikat kann sich nur auf die globalen Namen des aktuellen SQL-Schemas beziehen.

Ein Datenbankzustand heißt *konsistent*, wenn er alle im Schema deklarierten Zusicherungen erfült. Logisch gesehen sind daher alle Tabellen- und Schemazusicherungen konjunktiv (mit **and**) verknüpft. Das Aufspalten einer komplexen Zusicherung der Form **check***(Prädikat*$_1$ **and** *Prädikat* $_2$) in zwei Zusicherungen ermöglicht jedoch die separate *Benennung* von Zusicherungen:

constraint *Name*$_1$ **check** *(Prädikat*$_1$*)*
constraint *Name*$_2$ **check** *(Prädikat*$_2$*)*

Zusicherungsnamen werden einerseits benutzt, um Zusicherungen explizit zu löschen (**drop constraint** ...) und zu modifizieren (**set constraint** ... **deferred**); andererseits können Anwendungsprogramme beim Auftreten einer Integritätsverletzung den Namen der aktuell verletzten Zusicherung bestimmen und anwendungsspezifische Ausnahemebehandlungsoperationen auslösen.

Auf den ersten Blick erscheint die Unterscheidung zwischen Tabellen- und Schemazusicherungen überflüssig, da syntaktisch gesehen jede Schemazusicherung direkt in eine Tabellenzusicherung für eine Tabelle T umgewandelt werden kann. Jedoch besitzt die universelle Quantifizierung für Tabellen die subtile Eigenschaft, für leere Tabellen immer den Wert **true** zu liefern, so daß im Fall, daß T leer ist, die Tabellenzusicherungen von T nicht überprüft werden.

Als Beispiel betrachte man die folgende Schemazusicherung, die in keinem Datenbankzustand erfüllt ist:

create assertion *unerfuellbar* **check***(***false***)*

Die entsprechende Tabellenzusicherung

create table *Mitarbeiter*
(...
constraint *unerfuellbar* **check***(***false***))*

ist jedoch immer dann erfüllt, wenn die Tabelle *Mitarbeiter* leer ist. In den folgenden Abschnitten wird die Definition von Zusicherungen mit zunehmendem Wirkungsbereich beschrieben (Spaltenwert, Reihe, Tabelle, mehrere Tabellen).

4.5.1 Spaltenwertintegrität und Domänendeklarationen

Eine Tabellenzusicherung, deren Prädikat sich auf nur einen Spaltennamen bezieht, garantiert die Spaltenintegrität und wird in folgenden Modellierungssituationen eingesetzt:

- Vermeidung von *Nullwerten*, zum Beispiel **check** *(Alter* **is not null***)*,
- Definition von *Unterbereichstypen*, zum Beispiel **check***(Alter >= 0* **and** *Alter <= 150)*,
- Definition von *Formatinformationen* durch Stringvergleiche, zum Beispiel **check***(Postleitzahl* **like** *'D-_____')*
- Definition von *Aufzählungstypen*, zum Beispiel **check***(Note* **in** *(1, 2, 3, 4, 5, 6))* oder **check***(hatVordiplom* **in** *('Ja' , 'Nein'))*

Häufig ist es sinnvoll, eine immer wiederkehrende Kombination eines Basistypen mit zugehörigen Zusicherungen in Form einer benannten *SQL-Domäne* im aktuellen Schema zu definieren:

create domain *Schulnote* **integer**
constraint *NoteDefiniert* **check***(***value is not null***)*
constraint *NoteZwischen1und6* **check***(***value in** *(1, 2, 3, 4, 5, 6));*

Der Domänenname kann nachfolgend überall dort verwendet werden, wo ein SQL-Basistypname erwartet wird.

4.5.2 Reihenintegrität

Eine Tabellenzusicherung, deren Prädikat sich auf mehrere Spaltennamen bezieht, definiert eine Reihenintegritätsbedingung, die von jeder Reihe einer Tabelle erfüllt sein muß, zum Beispiel:

check(*Ausgaben* <= *Einnahmen*)
check((*HatVordiplom*, *HatDiplom*) **in values**(
    ('nein', 'nein')
    ('ja', 'nein')
    ('ja', 'ja')))

Die zweite Zusicherung garantiert, daß das Paar bestehend aus den Spaltenwerten *HatVordiplom* und *HatDiplom* nur Werte aus der Menge der aufgezählten legalen Kombinationen annimmt.

4.5.3 Tabellenintegrität

Während die in den vorangegangenen Abschnitten beschriebenen Zusicherungen in Datenbanksystemen mit geringem Aufwand (ohne zusätzliche Sekundärspeicherzugriffe) überprüft werden können, erfordert die Überprüfung von *quantifizierten Prädikaten* (s. Abschnitt 4.3.5) im schlimmsten Fall als Folge einer einfachen elementorientierten Änderungsoperation die Auswertung einer kompletten mengenorientierter Anfrage zur Integritätsprüfung.

check((**select sum**(*Budget*) **from** *Projekte*) >= 0)

oder

check(**exists** (**select** * **from** *Abteilung* **where** *Oberabt* = 'LTSW')

Einige in der Praxis häufig auftretende quantifizierte Zusicherungen können in Datenbanksystemen effizient mittels *Indexstrukturen* überprüft werden. Während üblicherweise die Integritätsüberprüfung das Laufzeitverhalten von Datenbanksystemen verschlechtert, können in diesen Situationen Zusicherungen sogar zu einem *Effizienzgewinn* bei Anfragen und Änderungsoperationen führen.

SQL-92 bietet spezielle syntaktische Konstrukte für häufig auftretende Muster solcher quantifizierter Zusicherungen. Dies erhöht einerseits die *Lesbarkeit* von Schemadeklarationen und ermöglicht andererseits der Datenbanksystemsoftware eine *optimierende Implementierung* von Zusicherungen.

- *Spaltenwerteindeutigkeit*: Die Eindeutigkeit von Spaltenwertkombinationen in einer Tabelle gestattet eine wertbasierte Identifikation von Tabellenelementen. Eine solche Spaltenkombination wird als *Schlüsselkandidat* bezeichnet.

  Die SQL-Zusicherung

create table *T* (... **unique**(*Spaltenname*$_1$, *Spaltenname*$_2$, ..., *Spaltenname*$_n$))

ist semantisch äquivalent zu der folgenden quantifizierten Zusicherun:

create table *T* (... **check**(**all** *x*, **all** *y* :...
((*x.Spaltenname*$_1$ <> *y.Spaltenname*$_1$) **or** (*x.Spaltenname*$_2$ <> *y.Spaltenname*$_2$) **or** ...
(*x.Spaltenname*$_n$ <> *y.Spaltenname*$_n$))
or $x = y$))

Diese Langform definiert auch die Behandlung von Nullwerten in den Spalten des Schlüsselkandidaten.

Eine Tabelle kann mehrere Schlüsselkandidaten besitzen, die durch separate unique-Klauseln beschrieben werden.

- *Primärschlüsselintegrität:* Ein Schlüsselkandidat, in dessen Spalten keine Nullwerte auftreten dürfen, kann als *Primärschlüssel* ausgezeichnet werden. Eine Tabelle kann nur einen Primärschlüssel besitzen. Die SQL-Zusicherung

create table *T*
(...
primary key(*Spaltenname*$_1$, *Spaltenname*$_2$, ..., *Spaltenname*$_n$))

ist äquivalent zu der SQL-Zusicherung:

create table *T*
(...
unique *Spaltenname*$_1$, *Spaltenname*$_2$, ..., *Spaltenname*$_n$),
check(*Spaltenname*$_1$ **is not null**),
check(*Spaltenname*$_2$ **is not null**),
...,
check(*Spaltenname*$_n$ **is not null**))

- *Referentielle Integrität*(Fremdschlüsselintegrität): Diese Zusicherung bezieht sich auf den Zustand *zweier* Tabellen und wird deshalb im nächsten Abschnitt separat behandelt.

4.5.4 Referentielle Integrität

Die Werte eines Schlüsselkandidaten einer Tabelle T_{Ziel} können als Spaltenwerte in einer Zeile einer Tabelle T_{Quelle} verwendet werden, um eine *Beziehung* (wertbasierte Assoziation) zwischen dieser Zeile und einer zugehörigen Zeile in Tabelle T_{Ziel} zu beschreiben (s. Abschnitt 3.2.3). Die entsprechenden Spalten in T_{Quelle} werden als *Fremdschlüssel* bezeichnet.

Referentielle Integrität (Fremdschlüsselintegrität) ist eine Zusicherung über den Zustand zweier Tabellen, die dann erfüllt ist, wenn zu jeder Reihe in Tabelle T_{Quelle} eine zugehörige Reihe in Tabelle T_{Ziel} existiert, die den Fremdschlüsselwert von T_{Quelle} als Wert ihres Schlüsselkandidaten besitzt.

Zum Beispiel sichert referentielle Integrität zu, daß es zu den Werten des Attributs *Abteilung* der Tabelle *Mitarbeiter* (Fremdschlüsselattribut) entsprechende Werte in der Spalte *Nr* der Tabelle *Abteilungen* (Primärschlüssel) gibt. Abbildung **??** verdeutlicht die wertbasierte

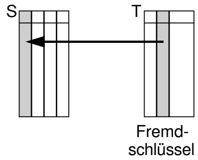

Abbildung 62: Referentielle Integrität

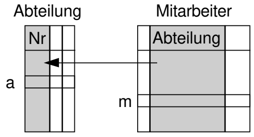

Abbildung 63: Referentielle Integrität

Assoziation zwischen Zeilen der Quelltabelle *Mitarbeiter* und Reihen der Zieltabelle *Abteilungen* über Pfeile, die jeweils vom Fremdschlüsselwert der Spalte *Abteilung* zum zugehörigen Primärschlüsselwert der Spalte *Nr* führen.

In SQL-92 wird die Fremdschlüsselintegrität syntaktisch als eine *Tabellenzusicherung* für T_{Quelle} formuliert. Für unser obiges Beispiel lautet die Tabellendefinition wie folgt:

create table *Mitarbeiter*
(...
constraint *MitarbeiterHatAbteilung*
foreign key*(Abteilung)* **references** *Abteilung(Nr))*

Diese Zusicherung ist äquivalent zu der folgenden SQL-Schemazusicherung:

create constraint *MitarbeiterHatAbteilung* **check***(*
not exists *(***select * from** *Mitarbeiter m* **where**
not exists *(***select * from** *Abteilung a* **where** *m.Abteilung = a.Nr)))*

Im allgemeinen besteht ein Fremdschlüssel aus einer Liste von Spalten, der eine typkompatible Liste von Spalten in T_{Ziel} entspricht:

create table T_{Quelle}
(...
constraint *Name*
foreign key$(A_1, A_2, \ldots, A_n)$ **references** $T_{Ziel}(B_1, B_2, \ldots, B_n))$

In dem häufig auftretenden Fall, daß es sich bei B_1, B_2, ..., B_n um die Primärschlüsselspalten von T_{Ziel} handelt, kann die Angabe der B_i entfallen.

Schließlich sei erwähnt, daß rekursive Beziehungen (z.B. *Abteilung : Oberabteilung*) zu reflexiven Fremdschlüsseldeklarationen ($T_{Quelle} = T_{Ziel}$) führen.

4.5.5 Behandlung von Integritätsverletzungen

Ergreift ein SQL-Benutzer keine speziellen Maßnahmen, so werden SQL-Anweisungen, die eine (modellinhärente oder anwendungsspezifische) Zusicherung verletzen, vom Datenbanksystem *ignoriert*, und es wird über eine Statusvariable signalisiert, welche Zusicherung verletzt wurde.

Eine Sonderrolle nimmt dabei die commit-Anweisung ein, da im Falle des Scheiterns verzögerter Integritätsbedingungen in jedem Fall ein *Transaktionsabbruch*(**rollback**) ausgelöst wird. In der Praxis ist es daher sinnvoll, vor dem Transaktionsende mit der SQL-Anweisung **set constraints all immediate** eine unmittelbare Überprüfung aller verzögerbaren Zusicherungen zu erzwingen und Integritätsverletzungen programmgesteuert explizit zu behandeln.

Gerade im Zusammenhang mit referentiellen Integritätsbedingungen ist es wünschenswert, *automatisch* auf Integritätsverletzungen zu reagieren, und die Integrität nicht durch das Ignorieren von Anweisungen, sondern durch das Auslösen geeigenter *Folgeaktionen* wiederherzustellen.

Die Fremdschlüsselintegrität zwischen zwei Tabellen T_{Quelle} und T_{Ziel} kann durch folgende vier Operationen verletzt werden:

1. **insert into** T_{Quelle}
2. **update**T_{Quelle} **set** ...
3. **delete from** T_{Ziel} ...
4. **update** T_{Ziel} **set** ...

Tritt bei der Anweisung (1) oder (2) ein Fremdschlüsselwert in T_{Quelle} auf, der nicht als Schlüsselwert in T_{Ziel} definiert ist, können in SQL-92 keine Folgeaktionen ausgelöst werden, und die Anweisung wird (mit Fehlerkode) ignoriert.

Wird bei Anweisung (3) oder (4) der Versuch unternommen, eine Reihe zu löschen, deren Schlüsselwert noch als Fremdschlüsselwert in einer oder mehreren Reihen von T_{Quelle} auftritt, so wird eine Folgeaktion ausgeführt, die am Ende der references-Klausel der zugehörigen Tabellenzusicherung spezifiziert wurde (s. Abschnitt 4.5.4). SQL-92 unterstützt zu diesem Zweck folgende Aktionen:

- **set null**: Der Fremdschlüsselwert aller betroffenen Reihen von T_{Quelle} wird durch **null** ersetzt.
- **set default:** Der Fremdschlüsselwert aller betroffenen Reihen von T_{Quelle} wird durch den Standardwert der Fremdschlüsselspalte (**default**, s. Abschnitt 4.2.7) ersetzt.
- **cascade**: In Fall (3, **delete**) werden alle betroffenen Reihen von T_{Quelle} gelöscht. In Fall (4, **update**) werden die Fremdschlüsselwerte aller betroffenen Reihen von T_{Quelle} durch die neuen Schlüsselwerte der korrespondierenden Reihen in T_{Ziel} ersetzt.
- **no action**: Es wird keine Folgeaktion ausgelöst, sondern es greift die Standard-SQL Fehlerbehandlung – die Anweisung (3) bzw. (4) wird ignoriert.

Folgeaktionen können für Fall (3) und (4) separat spezifiziert werden:

create table T
  (...
  constraint xx **foreign key**$(\ldots)$ **references** ...
    on delete set null
    on update cascade)

```
   Reihenfolgesensitivität vermieden durch Phasenansatz & (s.S.227)
   Markierung zum Löschen.
   Verallgemeinerung wird in aktiven Datenbanksystemen verfolgt.  DBMS löst benutzerdef.
Aktionen aus
```

4.5.6 Zeitpunkt der Integritätsüberprüfung

Erlaubt ein Datenbankmodell die Formulierung von Zusicherungen, die sich auf *mehrere* Zustandsvariablen erstrecken, bietet jedoch keine Operationen an, um diese Variablen *simultan* zu aktualisieren, so wird es notwendig, den Zeitpunkt der Integritätsüberprüfung genauer zu spezifizieren. Zum Beispiel ist es in SQL-92 möglich, Zusicherungen über mehrere Tabellen zu formulieren, andererseits kann eine SQL-Anweisung nur Reihen einer Tabelle simultan modifizieren.

SQL-92 löst den Zielkonflikt zwischen früher Fehlerverletzungserkennung und komplexen multi-relationalen Integritätsbedingungen, indem der SQL-Benutzer für jede Tabellen- und Schemazusicherung über Schlüsselworte explizit einen von zwei Modi der Integritätsüberprüfung wählen kann:

- **not deferrable** kennzeichnet eine nicht verzögerbare Zusicherung, die *unmittelbar* nach jeder SQL-Anweisung überprüft wird.

- **deferrable** kennzeichnet eine verzögerbare Zusicherung. Zu einer solchen Zusicherung wird eine zweiwertige Indikatorvariable (*flag*) gespeichert, die angibt, ob nach der nächsten SQL-Anweisung diese Zusicherung geprüft werden soll (flag = **immediate**), oder ob die Prüfung aufgeschoben werden soll (flag = **deferred**). Die Zusicherung wird außerdem immer beim Umschalten der Indikatorvariable auf den Wert **immediate** sowie am Transaktionsende überprüft.

Der Anfangswert der Indikatorvariablen, der am Beginn jeder Transaktion gilt, wird bei der Definition der Zusicherung mit der Klausel **initially immediate** oder **initially deferred** festgelegt. Fehlt die Angabe der Klausel, so wird die Definition **initially immediate** angenommen.

Während der Ausführung einer Transaktion kann der Wert der Indikatorvariablen mit der Anweisung **set constraint** *Zusicherungsname* **immediate** (bzw. **deferred**) verändert werden.

Natürlich wird ein Datenbanksystem Optimierungen durchführen, um redundante Überprüfungen von Zusicherungen zu vermeiden, so daß zum Beispiel eine nicht-verzögerbare Zusicherung nicht tatsächlich nach jeder SQL-Anweisung erneut überprüft wird.

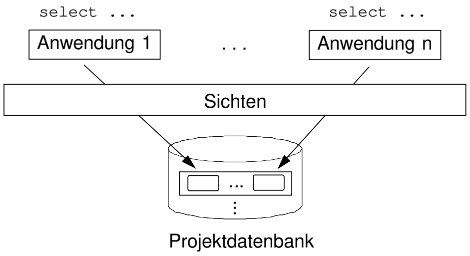

Abbildung 64: Sichten in SQL

4.6 Sichten in SQL

4.6.1 Sichten auf eine Tabelle

Motivation: DB vereinigt zahlreiche Unternehmenssichten auf das gleiche Objekt. Anwendungen sollen jedoch vor der Komplexität geschützt werden.

2. Motivation: Bewußte Kapselung

3 Möglichkeiten der Sichtendefinition

Strukturelle Einschränkung (Projektion)

Prädikative Einschränkung (Selektion)

WITH CHECK LOCAL / CASCADED (S. 82)

Operationale Einschränkung (Zugriffsrechte, INSERT, DELETE, ...)

Hier weg?

s. Updatable Query Specifications auf S. 184 in SQL Buch

4.6.2 Sichten auf mehrere Tabellen

4.7 Transaktionale Datenbankprogrammierung

Commit, Rollback , Abort. Sperrkonflikte, Notification

Abbildung: Persistente DB-Historie

5 Objektorientierte Datenbanksprachen

Ziel dieses Abschnittes ist es, dem Leser ein tieferes Verständnis für objektorientierte Datenbanken zu geben, wobei sich die Betrachtungen auf den ODMG-93 Standard beschränken. Nach der Darlegung allgemeiner charakteristischer Eigenschaften objektorientierter Datenbanken in Abschnitt 5.1 wird in Abschnitt 5.2 zunächst die Notwendigkeit eines weiteren Datenmodells motiviert. Abschnitt 5.3 verschafft dann einen Überblick über den ODMG-93 Standard, dessen Objektmodell in Abschnitt 5.4 in Beziehung zum Objektmodell der Object Management Group gesetzt wird. Analog zu SQL werden anschließend die Datendefinitionssprache ODL (5.5) und die Anfragesprache OQL (5.6) erläutert.

5.1 Charakteristische Eigenschaften objektorientierter Datenbanken

`OO Manifesti wiedergeben oder in 1.2.6 integrieren und hier nur Verweis.`

5.2 Zur Nutzung objektorientierter Datenbanksprachen

Im Vergleich zu SQL streben objektorientierte Datenbanksprachen eine wesentlich weitergehende *Modellierungsmächtigkeit* an, die neben der Beschreibung von persistenten Datenstrukturen auch die Beschreibung von Verhalten (prozeduralem Code) mit einschließt. Informationssysteme, die mit objektorientierten Datenbanken entwickelt werden, besitzen daher auch eine substantiell andere *Architektur* als Datenbanksysteme, die mit relationalen Datenbanksystemen realisiert werden.

- Abbildung 65 zeigt, daß bei der Nutzung von SQL eine *explizite Kommunikation* zwischen Anwendungsprogrammen (geschrieben in konventionellen Sprachen der dritten Generation wie C, Pascal, COBOL, FORTRAN, ...) und dem Datenbanksystem über SQL-Kommandos und -Anfragen stattfindet. Die Argumente der Kommandos und die Ergebnisse der Anfragen werden über spezielle Mechanismen zwischen dem Hauptspeicher und dem externen persistenten Speicher *kopiert*. Dabei muß eine explizite Umwandlung zwischen den typisierten Datenstrukturen der Programmiersprache (Arrays, verkettete Listen, Records) und den SQL-Datenstrukturen (Tabellen, Zeilen) stattfinden.

- Abbildung 66 zeigt, daß bei der Nutzung eines objektorientierten Systems eine *implizite Kommunikation* zwischen den Anwendungsprogrammen (geschrieben in einer speziellen objektorientierten Datenbanksprache, z.B. O_2C oder Persistent C++) und dem Objektspeicherungssystem stattfindet. Der Anwendungsprogrammierer kann *direkt* auf persistenten Datenstrukturen arbeiten (Objektinstanziierung, Attributzugriff, Zuweisung, Methodenaufruf, ...), die dem gleichen Typsystem wie die flüchtigen Datenstrukturen gehorchen, so daß keine für den Programmierer sichtbare Konvertierung zwischen Hauptspeicherstrukturen und persistenten Datenstrukturen stattfinden muß (Persistenzabstraktion, vgl. Abschnitt 2.2). Insbesondere können Programmvariablen direkt persistente Objekte *referenzieren*.

Neben der Vermeidung des *impedance mismatch* bieten objektorientierte Datenbanksprachen folgende *Vorteile* gegenüber SQL-Lösungen bei der Entwicklung von Informationssystemen:

Fig:xx ähnlich zu ODMG S. 3

Abbildung 65: Architektur eines Informationssystems bei der Realisierung mit SQL-Datenbanken

Fig:xx ähnlich zu ODMG S. 3

Abbildung 66: Architektur eines Informationssystems bei der Realisierung mit objektorientierten Datenbanken

Datenmodellierungsmächtigkeit: Wie in den folgenden Abschnitten 5.3 – 5.5.8 näher beschrieben wird, können Programmierer die Strukturen und das Verhalten des Informationssystems durch anwendungsnahe Konzepte wie Objekte, Klassen und Nachrichten direkt modellieren, es ist keine Abbildung in flache Reihen innerhalb von Tabellen erforderlich.

Uniformität: Programmierer müssen nur *eine* Sprache zur Datendefinition und Datenmanipulation (DDL und DML) erlernen, die uniform für flüchtige und persistente, lokale und globale Daten kleinen und großen Umfangs gilt.

Kompaktheit: Datenbankprogramme beschränken sich auf die eigentlichen, anwendungsrelevanten Algorithmen und enthalten keine aufwendigen Transfer- und Konvertierungsroutinen zum Datenaustausch mit der SQL Datenbank.

Wartbarkeit: Typinformationen für persistente globale und flüchtige lokale Daten sind in einem einheitlichen Format gespeichert, so daß Werzeuge die Konsistenz zwischen Datenbankschema und Variablendeklarationen in Anwendungsprogrammen überprüfen können.

Laufzeiteffizienz: Für komplexe Zugriffe läßt sich die Schnittstelle zwischen Programmen und Datenbank wesentlich effizienter gestalten, als dies bei rein mengenorientierten SQL-Schnittstellen möglich ist. (z.B. durch Cache-Techniken, Compiler-Optimierungen und durch Ausnutzen der Referenzsemantik objektorientierter Modelle).

Diesen Vorteilen stehen folgende eher pragmatische *Nachteile* gegenüber.

Fehlende Standardisierung: Gegenwärtig bietet praktisch jeder Hersteller objektorientierter Datenbanken eine eigene integrierte Datenbanksprache an. Entwickler von Informationssystemen sind jedoch daran interessiert, daß ihre Kunden das System auf Hardware- und Softwareplattformen verschiedener Hersteller einsetzen können. Die fehlende Standardisierung verhindert auch, daß Drittanbieter in großem Umfang Werkzeuge zur Unterstützung der Anwendungsentwicklung anbieten (vgl. generische SQL Datenbankbrowser). Schließlich ist der Datenaustausch zwischen unabhängig entwickelten objektorientierten Datenbanksystemen sehr schwierig zu realisieren.

Aufwendige Sprachrealisierungstechniken: Die enge Integration von Sprachprozessoren (Parser, Typchecker, Codegenerator, Speicherverwaltung) und Datenbanksystemen (Datenwörterbuch, Cache-Manager, Anfrageoptimierer) führt dazu, daß Fortschritte bei der Sprachprozessortechnologie (z.B. Optimierer, Debugger, Bibliotheksfunktionalität) nur mit Verzögerung Eingang in Datenbanksprachprozessoren finden.

Beim Zugriff auf relationale Datenbanksysteme aus Sprachen der dritten Generation ergibt sich das folgende Problem, für das sich in Anlehnung an den Begriff des Wellenwiderstands aus der Nachrichtentechnik in der Literatur der Begriff des *impedance mismatch* eingebürgert hat (s. Abbildung 67): Während Sprachen der dritten Generation in der Lage sind, mit sehr hoher Frequenz Operationen auf individuellen Datensätzen durchzuführen, sind relationale Datenbanken darauf optimiert, mit relativ geringer Frequenz hochkomplexe Operationen auf großen Datenmengen (z.B. Mehrweg-Verbundoperationen) durchzuführen. Die Inkompatibilität dieser beiden Wellenwiderstände an der Datenbank-Programmierschnittstelle führt insgesamt zu einer geringen Kommunikationseffizienz (siehe dazu auch Abschnitt 7.2)

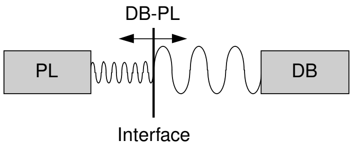

Abbildung 67: *Impedance mismatch* an der Datenbank-Programmierschnittstelle relationaler Datenbanksysteme

5.3 Überblick über den ODMG-93 Standard

Die im vorangegangenen Abschnitt genannten Nachteile unabhängig entwickelter Objektdatenbanksprachen und -modelle wurden von den Herstellern dieser Datenbanksysteme erkannt und haben zur Entwicklung eines *Object Database Standard* durch die sogenannte Object Database Management Group (ODMG) geführt. Der sogenannte ODMG-93 Standard soll langfristig eine ähnliche Rolle wie SQL für relationale Datenbanksysteme bei der Entwicklung *portabler Informationssysteme* einnehmen.

Es lassen sich drei Arten der Portabilität unterschieden:

Quellkode-Portabilität erlaubt den Austausch des Datenbankssystems ohne Modifikation oder erneute Implementierung der Applikation. Der Quellkode der Applikationen muß allerdings für die Verwendung mit dem neuen Datenbanksystem in der Regel neu übersetzt werden.

Binärkode-Portabilität stellt die Lauffähigkeit eines compilierten Anwendungsprogramms auf verschiedenen Systemen sicher, ohne daß ein separater Übersetzungsvorgang für die unterschiedlichen Systeme notwendig ist. Diese Art der Portabilität wird z.B. durch den ODBC-Standard von Microsoft für relationale Systeme sichergestellt.

Datenportabilität bezieht sich auf die Wiederverwendbarkeit von Daten. Durch eine Applikation erzeugte Daten sind auch für Applikationen zugreifbar, die in anderen Sprachen realisiert worden sind.

Ziel des ODMG-Standards ist Quellkode-Portabilität.

Zum Zeitpunkt der Erstellung dieses Textes liegt der ODMG-93 Standard in seiner zweiten Version (Release 1.1) als Buch vor []. Alle am Standardisierungsprozeß beteiligten Firmen haben sich verpflichtet, ODMG-gerechte Schnittstellen für ihre Produkte anzubieten, Der Standard definiert folgende für den Anwendungsprogrammierer sichtbare Komponenten eines Objektdatenbanksystems:

Datenmodell: Das ODMG-Modell basiert auf dem abstrakten *Core Component* Objektmodell der *Object Management Group* (OMG), die ebenfalls ein Zusammenschluß von Softwareherstellern, wie HP, Sun und anderen, ist [xx] und die vor allem durch die Spezifikation der *Common Object Request Broker Architecture* (CORBA) bekannt ist. Das ODMG-Datenmodell definiert Konzepte wie z.B. Objekt, Typ, Attribut, Beziehung, Operation, Schnittstelle, Implementierung und Supertyp.

Datendefinitionssprache: Die *Object Definition Language* (ODL) definiert eine (C++ ähnliche) Syntax zur Beschreibung von ODMG-Typen, die es erlaubt Namen, Attributen, Operationssignaturen, usw. für solche Typen festzulegen. ODL ist eine aufwärtskompatible Erweiterung der *Interface Definition Language* (IDL) der OMG. ODL ist programmiersprachenunabhängig und gestattet die Definition aller Strukturen des ODMG-Modells mit Ausnahme von Operationsimplementierungen. Damit ODL auch zur Schemaintegration und zum Datenaustausch zwischen verschiedenen objektorientierten und relationalen Datenmodellen eingesetzt werden kann, ist die Syntax von ODL erweiterbar ausgelegt.

Anfragesprache: Die *Object Query Language* (OQL) definiert eine (SQL-ähnliche) konkrete Syntax für eine deklarative (nicht-prozedurale) Sprache zur Formulierung von Anfragen auf Kollektionen und auf Datenbanken, die zu dem ODMG-Objektmodell kompatibel sind. OQL ist nicht algorithmisch vollständig, jedoch können OQL Ausdrücke Operationen aufrufen, die in prozeduralen Sprachen implementiert sind, und die ihrerseits wieder OQL Ausdrücke enthalten können. OQL kann sowohl aus Anwendungskode heraus als auch interaktiv von (erfahrenen) Datenbankanwendern benutzt werden.

C++ und Smalltalk Sprachanbindung: Da die ODMG von der in Abbildung 66 skizzierten Integration von Programmiersprache und Datenbanksystem ausgeht, werden keine speziellen Notationen oder „eingebettete DML Anweisungen“ für die Manipulationen von persistenten Objekten benötigt. Die sogenannten C++ und Smalltalk *language bindings* definieren daher nur die Details der Abbildung des ODMG-Objektmodells und der ODMG-Anfragesprache in die bestehende Syntax und Semantik der jeweiligen Programmiersprache (C++ bzw. Smalltalk). Die Sprachanbindung wird entweder durch einen Präprozessor oder durch einen erweiterten Compiler realisiert. Die Sprachanbindung definiert auch die konkreten Namens- und Typkonventionen für die ODMG-*Standardtyphierarchie*.

Bei der Vortstellung des ODMG-93-Standards in den folgenden Abschnitten ist zu beachten, daß sich die Arbeiten zum Zeitpunkt der Erstellung dieses Buches noch im Fluß befinden. Viele Konzepte und Konstrukte, die in der Bschreibung des Standards [] vorgestellt und beschrieben werden sind noch unterspezifiziert und teilweise auch widersprüchlich.

```
ODMG-93 ist noch unterspezifiziert und teilweise
widersprüchlich (Bsp:  in, out, inout in Operationssignaturen
Persistenzmodell) (s.  ODMG 6.9), Sperrprotokolle, parallele
Vererbungshierarchien, Persistenzkonzept (Erreichbarkeit, ...).  Sprachanbindung.
```

5.4 Vergleich des ODMG-93 und des OMG Objektmodells

Wie bereits im vorangegangenen Abschnitt beschrieben, basiert das ODMG-Datenmodell auf dem *Core Component* Objektmodell der *Object Management Group* (OMG), das sich wie folgt zusammenfassen läßt:

- Objekte sind Instanzen von Typen und besitzen eine systemweit eindeutige Objektidentität.
- Ein Typ definiert das Verhalten und die möglichen Zustände seiner Instanzen.
- Das Verhalten von Objekten wird durch eine Menge von Operationen beschrieben, die auf Objekten eines Typs ausgeführt werden können.
- Ein Objekt ist unmittelbare Instanz genau eines Typs.
- Der Typ eines Objekts wird einmalig bei der Objekterzeugung festgelegt. Objekte können ihren Typ nicht dynamisch ändern.
- Typen sind in einer Subtyp/Supertyphierarchie organisiert, wobei ein Typ mehrere Supertypen besitzen darf.
- Subtypbeziehungen werden explizit definiert und werden nicht implizit von der Kompatibilität zwischen den Signaturen des Super- und Subtyps abgeleitet (vgl. strukturelle Subtypisierung [] FM).
- Operationen besitzen Signaturen, die den Namen der Operation, Typen und Namen der Argumente der Operation und den Ergebnistyp der Operation definieren.
- Operationen werden für einen einzelnen Typ definiert, dies ist der Typ des ausgezeichneten ersten Arguments einer Operation.
- Operationen nehmen Werte oder Objekte als Argumente und liefern einen Wert oder ein Objekt als Ergebnis. Die Parameter- und Ergebnisübergabe besitzt Referenzsemantik.
- Operationen werden explizit auf ein Objekt angewandt und dürfen Seiteneffekte besitzen.
- Operationen werden im Implementationsteil einer Typdefinition definiert.

Das ODMG-Modell erweitert das OMG *Core Component* Objektmodell um folgende datenbankspezifische Konzepte:

- Objekte können persistent gespeichert werden.
- Operationen können Ausnahmen auslösen, die im Typ definiert werden.
- Objekte besitzen Attribute und Beziehungen zusätzlich zu dem gekapselten Objektzustand, der nur durch Operationen zugänglich ist.
- Das Datenmodell definiert generische, systemweit vorhandene Dienste für Anfragen und Transaktionen.

Die ODMG folgt damit den Vorstellungen der OMG, die vorsieht, daß das Kernmodell für spezielle Anwendungsdomänen, wie sie z.B. die Vermittlung in verteilten Netzen und die persistente Datenhaltung darstellen, aufwärtskompatibel erweitert werden kann.

5.5 Datendefinition in ODL

`text`

5.5.1 Überblick über die vordefinierte ODMG-Typhierarchie

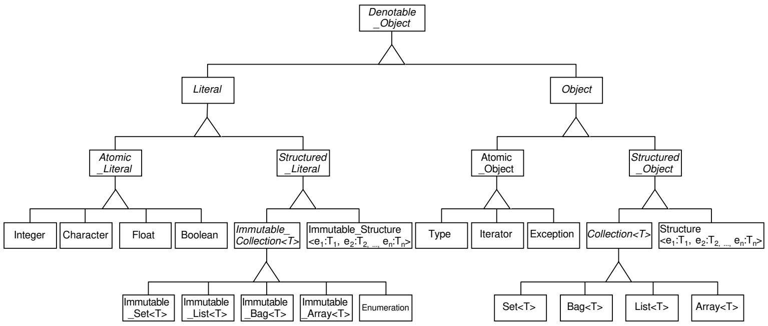

Abbildung 68: Die vordefinierte ODMG-93 Typhierarchie (I)

Der ODMG-Standard legt eine Reihe von vordefinierten Typen fest, die in einer Typhierarchie angeordnet sind. In Abbildung 68 ist ein Teil dieser Typhierarchie in OMT-Notation (s. Abschnitt 5.5.1) dargestellt. Die Typen sind dabei durch das Symbol für die Klassen dargestellt, da sie mit diesen eng verwandt sind[12]. Die Supertypen sind dabei oberhalb ihrer Subtypen angeordnet, an die sie ihre Eigenschaften und Operationen vererben. Die vordefinierte Typhierarchie kann um benutzerdefinierte Subtypen erweitert werden, die ihre Supertypen erweitern und verfeinern können.

Die Hierarchie enthält zwei spezielle Arten von Typen:

Abstrakte Typen definieren nur Eigenschaften, die von ihren Subtypen geerbt werden. Da sie keine `vollständige?` Implementierung besitzen, können abstrakte Typen selbst nicht instantiiert werden. Sie definieren jedoch Signaturen (Protokolle), die für ihre Subtypen gelten. Abstrakte Typen können auch Operationen definieren (`und implementieren`), die auf Instanzen aller ihrer Subtypen anwendbar sind. `Dies unterstützt die Erstellung von generischem Kode und die Wiederverwenbarkeit von Kode??. Bsp has_name (Object)(Gibt es Implementierungsvererbung ???)` In Abbildung 68 und den folgenden Abbildungen werden abstrakte Typen durch Schrägschrift hervorgehoben.

Parametrisierte Typen besitzen einen (oder mehrere) Typparameter. Sie sind keine Typen sondern Typgeneratoren. Erst mit einem konkreten Typ als Parameter versehen, stellen sie selbst konkrete Typen dar. Parametrisierte Typen sind an dem nachgestellten in spitzen Klammern eingeschlossenen Typparameter $<T>$ zu erkennen. Sie werden unter anderem für generische Kollektionen wie z.B. **Set**$<T>$ verwendet. Diese können beliebige Subtypen von *Denotable_Object* als Parameter erhalten, wodurch instantiierbare Kollektionstypen wie **Set**$<$*Mitarbeiter*$>$ oder **Set**$<$*Abteilung*$>$ entstehen.

Die vordefinierte Typhierarchie besitzt zwei separate Wurzelelemente, die abstrakten Typen *Denotable_Object* und *Characteristic*, und bildet ansonsten eine strikte Baumstruktur.

[12]Der genaue Zusammenhang zwischen Typen und Klassen wird im Abschnitt 5.5.8 beschrieben

Allgemein sind jedoch für die Subtyphierarchien beliebige azyklische Graphen zulässig, da ein Typ direkter Subtyp mehrerer anderer Typen sein kann

Anwendungsprogramme können nur Subtypen von *Denotable_Object* direkt benennen, während die Subtypen von *Characteristic* nur die vordefinierten ODMG-Datenmodellkonstrukte beschreiben. Dies bedeutet, daß das ODMG-Modell *keine Konzepte höherer Ordnung* anbietet, da weder Operationen noch Typen Sprachobjekte erster Klasse sind. So ist es zum Beispiel nicht möglich, Operationen als Argumente oder Ergebnisse von Operationen zu verwenden oder Typen dynamisch zur Laufzeit zu manipulieren. Solche Mechanismen sind bei der Realisierung *generischer* Datenbankwerkzeuge, also von Werkzeugen, die für verschiedene Datenbankschemata einzusetzen sind, von Bedeutung.

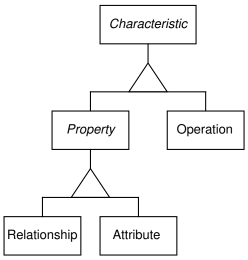

Abbildung 69: Die vordefinierte ODMG-93 Typhierarchie (II)

Der Typ *Characteristic* besitzt die Subtypen *Property* für Eigenschaften und *Operation* für Operationen (s. Abbildung 69). Die Eigenschaften (*properties*) eines Typs umfassen sowohl dessen Attribute als auch dessen Beziehungen zu anderen Typen, die durch die Subtypen *Attribute* und *Relationship* von *Property* definiert werden. Diese Typen werden beim Zugriff auf Metadaten einer ODMG-Datenbank benötigt (s. Abschnitt 5.5.11).

Neben der Integration der Metadatendefinition in das Objektmodell und des Metadatenzugriffs in die Anfragesprache OQL hat die Definition des Typs *Charakteristic* und dessen Subtypen als Teil der vordefinierten Typhierarchie einen weiteren nicht zu unterschätzenden Vorteil. Es wird explizit ein Begriffsapparat eingeführt, der eine präzise und konsistente Beschreibung der Sprachkonstrukte von ODMG unterstützt. Im folgenden wird unter Verwendung dieses Begriffsapparats *Charakteristik* als gemeinsame Bezeichnung für die Eigenschaften und Operationen eines Typs benutzt, wenn es nicht notwendig ist, zwischen den beiden zu unterscheiden. Mit dem Begriff *Eigenschaften* werden die Attribute und Beziehungen eines Typs gemeinsam angesprochen.

5.5.2 Literale und Objekte

Der Typ *Denotable_Object* besitzt zwei Subtypen, nämlich *Literal* und *Object*. Diese Spezialisierung entspricht der in Abschnitt 2.1 erläuterten Unterscheidung zwischen Werten (unveränderlich) und Objekten (veränderbarer Zustand). Sowohl die Literale als auch die Objekte werden weiter in strukturierte und atomare Ausprägungen unterteilt, woraus sich die Subtypen *Atomic_Literal* und *Structured_Literal*, *Atomic_Object* und *Structured_Object* ergeben.

Atomic_Object besitzt die Subtypen *Type*, *Exception* und *Iterator*. Der Typ *Exception* wird bei der Definition von Ausnahmen verwendet, während *Type* alle Typen der vordefinierten Typhierachie als Instanzen besitzt und damit den Zugriff auf Metadaten ermöglicht (s. Abschitt 5.5.11). *Iterator* spielt eine wichtige Rolle bei der Definition von Kollektionen (s.u.).

Atomic_Literal ist Supertyp der üblichen Basistypen *Integer*, *Float*, *Character* und *Boolean*. Für die atomaren Literale gibt es keine create-Operation, alle Instanzen dieser Typen existieren jederzeit implizit. Der Benutzer kann neue Typen für Literale (Subtypen von *Literal*) definieren. Die Operationen auf den vordefinierten Literalen dürfen jedoch nicht überschrieben werden.

Sowohl die strukturierten Objekte als auch die strukturierten Literale sind weiter unterteilt in Kollektionen (*Collection* und *Immutable_Collection*) und Strukturen (*Structure* und *Immutable_Structure*).

5.5.3 Kollektionen und Iteratoren

Kollektionen können über einem beliebigen instantiierbaren Subtyp des Typs *Denotable_Object* definiert werden, also sowohl über Literale als auch über Objekte. Die Typen *Collection* und *Immutable_Collection* selbst sind abstrakte, parametrisierte Typen, wobei der Typparameter den Typ der Elemente festlegt. Der Typ *Collection* unterstützt Operationen zum Erzeugen, Kopieren und Löschen einer Kollektion, zum Einfügen, Ändern und Löschen von Elementen, zur Elementselektion und zum Test auf Enthaltensein.

Zur Iteration über die Elemente einer Kollektion muß ein Iterator erzeugt werden. Iteratoren sind Instanzen des vordefinierten parametrisierten Typs *Iterator*, der den Elementtyp der jeweiligen Kollektion als Parameter besitzt. Ein Iterator verwaltet eine aktuelle Position in einer Kollektion, und mit Hilfe von sukzessiven Aufrufen der auf diesem Typ definierten Funktion *next* können nacheinander alle Elemente der Kollektion besucht werden. Zu einer Kollektion kann mehr als ein Iterator definiert werden, sodaß simultan mehrere Positionen innerhalb einer Kollektion verwaltet werden können.

Die Funktion *next* positioniert den Iterator auf das nächste Element und liefert das neue aktuelle Element zurück. Bei geordneten Kollektionen geschieht die Iteration in der durch die Ordnung der Elemente festgelegte Reihenfolge, wobei bei der Erzeugung des Iterators zwischen den Richtungen *forward* und *backward* gewählt werden kann. Für ungeordnete Kollektionen ist eine beliebige Iterationsreihenfolge möglich.

Weiterhin sind auf Iteratoren Operationen zum Zurücksetzen des Iterators, zum Positionieren auf das erste bzw. letzte Element, für Statusabfragen sowie zum Löschen des Iterators definiert. Für eine C++ Sprachanbindung könnte die Schnittstelle zur Definition des Typs *Iterator* etwa wie folgt aussehen:

```
interface Iterator <T> ()
{attribute Boolean stable
 attribute Enumeration(forward, backward)
 T next()
 T first()
 T last()
 Boolean more()
 void reset()
 void delete() };
```

Das Konzept der Iteratoren ist dem Konzept der Cursor in SQL ähnlich, besitzt jedoch den Vorteil das Iteratoren im Gegensatz zu Cursorn Objekte erster Klasse sind. Dies erlaubt es dem Anwender typisierten, generischen Code zur Iterationsabstraktion zu schreiben, der unabhängig von der Art der Kollektion und dem Elementtyp verwendet werden kann. Man kann z.B. eine Operation *choose* realisieren, die eine Liste der Elemente einer beliebigen Kollektion am Bildschirm anzeigt und ein vom Benutzer selektiertes Element zurückliefert. Einer solchen Funktion wird ein Iterator für die anzuzeigende Kollektion als Parameter übergeben.

Alle Kollektionstypen sind parametrisierte Typen, sodaß sie Kollektionen mit beliebigem, aber innerhalb einer Kollektion festen Elementtyp repräsentieren. Die vordefinierte Typhierarchie enthält vier Typen für Kollektionen mit veränderbarem Zusatand:

Set ist der Typ ungeordneter Kollektionen, die keine Duplikate enthalten (Mengen): Neben den üblichen Mengenoperationen Durchschnitt, Vereinigung und Mengendifferenz definiert der Typ *Set* auch Operationen zum Kopieren von Mengen und Teilmengentests.

Bag ist der Typ ungeordneter Kollektionen, bei denen Duplikate erlaubt sind (Multimengen): Auf den Multimengen sind bis auf die Teilmengentests dieselben Operationen wie auf den Mengen definiert.

List ist der Typ geordneter Kollektionen, bei denen Duplikate erlaubt sind (Listen): Dieser Typ stellt Operationen für das positionsabhängige Einfügen, Löschen und Auffinden von Elementen zur Verfügung. Weiterhin besitzen Objekte dieses Typs eine Eigenschaft *current_position*, die den Index der aktuellen Position für Lese- und Änderungsoperationen enthält. Diese Position ist unabhängig von der durch einen Iterator verwalteten Position.

Array ist der Typ eindimensionaler Felder veränderlicher Länge: Für diesen Kollektionstyp sind Operationen zum Löschen, Ersetzen, Einfügen und Auffinden von Elementen an einer durch einen Index festgelegten Position vorgesehen. Die Länge des Feldes wird bei der Erzeugung initialisiert und kann entweder explizit durch eine resize-Operation oder implizit durch Zuweisung eines Elements an einer Position außerhalb der bisherigen Indexgrenzen verändert werden.

Zu jedem dieser Kollektionstypen existiert ein unveränderliches Gegenstück, das als Subtyp des Typs *Literal* deklariert ist (*Immutable_Set Immutable_Bag Immutable_List Immutable_Array*). Objekte dieser Typen zeigen dasselbe Verhalten, außer daß sie einen unveränderlichen Zustand besitzen. *Immutable_Collection* besitzt weiterhin den Subtyp *Enumeration*. Dieser entspricht dem Enumerationstyp, wie er aus vielen Programmiersprachen bekannt ist. Ein Enumerationstyp führt eine Reihe von Bezeichnern ein, die die einzigen Werte dieses Typs repräsentieren.

Sowohl Zeichenketten als auch *bit strings* werden im ODMG-Objektmodell als spezielle Listen betrachtet. Sie sind daher als Subtypen des jeweiligen Listentyps in der vordefinierten Typhierarchie enthalten (*String*, *Bit_String*, *Immutable_String* und *Immutable_Bit_String*).

5.5.4 Strukturen

Strukturen besitzen im Gegensatz zu Kollektionen, die sich durch eine variable Anzahl unbenannter Komponenten, den Elementen der Kollektion, auszeichnen, eine feste Anzahl benannter Komponenten. Für Strukturen sind folgende parametrisierte Typen vordefiniert:

$\mathit{Structure}{<}e_1\colon T_1\ e_2\colon T_2\ \ldots\ e_n\colon T_n{>}$

und

Immutable_Structure$<e_1$: T_1 e_2: T_2 $\ldots$ e_n: T_n $>$

Sie erhalten eine Reihe von Paaren e_i: T_i als Parameter jeweils bestehend aus einem Komponentennamen e_i und dem zugehörigen Typ T_i der Komponente. Zur Beschreibung von Namen kann z.B. der folgende Typ definiert werden:

Structure< *Vorname* :**String** *Nachname:* **String**>

Der Zugriff auf die Komponenten geschieht über die Komponentennamen durch die zu diesem Zweck allgemein übliche Punktnotation.

Für den Typ *Immutable_Structure* der unveränderlichen Strukturen sind die Subtypen *Date*, *Time*, *Timestamp*, und *Interval* vordefiniert, die dieselbe Semantik wie die entsprechenden Typen in ANSI SQL besitzen (s. Abschnitt 4.2.3). `Beispiel, verwende Datumstyp aus SQL`

5.5.5 Signaturen für Eigenschaften und Operationen

Ein ODMG-Datenbankschema besteht aus einer Menge von benannten Schnittstellen, wobei jede Schnittstelle **interface** einen Objekttyp definiert. Dabei werden sowohl Charakteristiken für die Instanzen dieses Typ als auch Eigenschaften für den Typ selbst festgelegt. Die ODMG-Typdefinitionen für das Beispiel des Firmeninformationssystems sieht wie folgt aus:

Im einzelnen werden in der Schnittstelle Signaturen für die folgenden Charakteristiken spezifiziert:

Attribute: Es werden Namen und Typen für die Attribute der Instanzen des Typs spezifiziert. Die Spezifikation jedes Attributs wird durch das Schlüsselwort **attribute** gekennzeichnet. Im Beispiel sind dies die Attribute *Name* und *PID*.

Als Werte für ein Attribut sind Literale und beliebige Strukturen zulässig, die als Komponenten nur Literale enthalten.

Für jedes Attribut sind die Operationen *get_value* (zum Zugriff auf den Attributwert) und *set_value* (zum Modifizieren des Attributwerts) mit den folgenden Signaturen vordefiniert:

get_value() $\rightarrow$ *exsting_value :Literal*
set_value(new_value :Literal)

Diese Operationen können vom Benutzer überschrieben werden.

Die Attribute sind als abstrakte Komponenten des Zustands zu verstehen. Sie brauchen nicht notwendigerweise auch als Komponente der Repräsentation (Implementierung) zu existieren. Es kann sich vielmehr auch um abgeleitete Attribute handeln, deren Wert bei Bedarf durch eine geeignete get_value-Routine aus anderen Werten berechnet wird. So kann z.B. für einen Typ *Person* ein Attribut *Alter* existieren, dessen Wert jedoch jeweils aus dem Attribut *Geburtsdatum* und dem aktuellen Datum bestimmt wird.

Dieser Zusammenhang spielt auch eine wichtige Rolle für die vom relationalen Datenmodell bekannten Update-Problematik für Sichten (*view updates*). View-Updates sind

```
interface Projekt (extent projekte  key (Nr, Titel))
{ attribute Unsigned Short Nr;
  attribute String Titel;
  attribute Long Budget;
  relationship Set<Mitarbeiter> Beteiligte
    inverse Mitarbeiter :: Projekte;
  relationship List<Abteilung>  Betreuungen
    inverse Abteilung :: Projekte };

interface Mitarbeiter (extent mitarbeiter  key PID)
{ attribute Strucure<Vorname :String Nachname :String> Name
  attribute Unsigned Short PID
  relationship Abteilung  Arbeitgeber
    inverse Abteilung :: Angestellte
  relationship List<Projekt> Projekte
    inverse Projekt :: Beteiligte }

interface Festangestellte: Mitarbeiter (extent festangestellte)
{ attribute Gehalt;
  attribute Einstellungsdatum }

interface Werkstudent: Mitarbeiter (extent werkstudenten)
{ attribute (xxx Datum) Beginn
  attribute Unsigned Short Dauer
  attribute (Preis) Verguetung }

interface Abteilung (extent abteilungen keys Name, Kurz)
{ attribute String Kurz;
  attribute String Name;
  relationship List<Projekt>  Projekte
    inverse Projekt :: Betreuung;
  relationship Leiter Chef
     inverse Leiter :: Bereich;
  relationship Set< Mitarbeiter > Angestellte
     inverse Mitarbeiter :: Arbeitgeber;
  relationship List<Abteilung> Unterabteilungen
     inverse Abteilung :: Oberabteilung
    {order_by Name}
  relationship Abteilung Oberabteilung
     inverse Abteilung :: Unterabteilungen;
  Boolean neuesProjekt in (Projekt) raises () xxx };

interface Leiter: Festangestellte (extent leiter)
{ relationship Abteilung Bereich
     inverse Abteilung :: Chef }
```

bei relationalen Systemen Sache des Datenbanksystems. Wegen inhärenter Mehrdeutigkeiten sind solche Updates nicht in allen Fällen möglich.

Im ODMG-Objektmodell ist die korrekte Propagierung von Updates auf abgeleiteten Attributen (Sichten) Sache des Programmierers der Typimplementierungen. Er muß bei der Realisierung der zugehörigen set_value-Methoden entscheiden, welche Konsequenzen die Veränderung eines abgeleiteten Attributs durch den Benutzer hat. Allgemein sollten zusammengehörige Paare von set_value/get_value-Methoden die Invariante $((get\_value(set\_value(x)) = x$ erfüllen.

Beziehungen: Eingeleitet durch das Schlüsselwort **relationship** wird der Typ der referenzierten Objekte, ein Bezeichner zum Traversieren der Beziehung und gegebenenfalls auch die inverse Beziehung angegeben. Beziehungen im ODMG-Datenmodell werden in Abschnitt 5.5.7 genauer beschrieben.

Operationen: Die Signatur für eine Operation umfaßt den Name der Operation, Namen und Typen der Parameter, den Typ des Rückgabewertes sowie eine Liste der Ausnahmen, die zur Laufzeit von der Operation ausgelöst werden können. Jede Operation ist genau einem Typ zugeordnet und besitzt ein Objekt dieses Typs als impliziten, ausgezeichneten ersten Parameter. `ToDo:` Beispiel zur Erläuterung des ersten Parameters, Es kann bei der Implementierung durch Mechanismen der Zielsprache darauf zugegriffen werden (z.b. durch *self*)s.Abschnitt Implementierung

Als weitere Argumente für eine Operation sind beliebige Objekte und Literale möglich, die Instanzen eines Subtyps von *Denotable_Object* sind. Eine Operation besitzt stets einen Rückgabewert (ist also eine Funktion). Operationen, die nur Seiteneffekte haben, liefern den Wert **nil** zurück.

Im Beispiel werden die Operationen ... auf den Objekten des Typs ... spezifiziert.

Typen können selbst auch als Objekte betrachtet werden (s. Abschnitt 5.5.11). Ein solches Objekt vom Typ *Type* besitzt einige Eigenschaften, denen bei der Schnittstellendefinition Werte zugeordnet werden können:

Supertypen: Ein Objekttyp kann als Subtyp einer oder mehrerer anderer Objekttypen definiert werden. Dazu werden bei seiner Definition in der Schnittstelle diese Typen als Supertypen angegeben (s. Abschnitt 5.5.6).

Extension: Zu einem Typ kann optional die Menge all seiner Instanzen, die sog. *Extension*, verwaltet werden. Sie wird nur angelegt, wenn in der Schnittstelle eine extent-Klausel angegeben wird, mit der gleichzeitig auch ein Name für die Extension festgelegt wird. Beim Erzeugen neuer Instanzen des Typs und beim Löschen von Instanzen wird die Extension automatisch aktualisiert.

Für einen Objekttyp A ist die Extension des Typs vom Typ **Set**$< A >$.

`Parallelen zu SQL`

Schlüssel: Zur Identifikation von Instanzen eines Typs in ihrer Klassenextension kann für diesen Typ in der Schnittstelle durch die key-Klausel ein Schlüssel festgelegt werden. Dabei sind sowohl einfache als auch zusammengesetzte Schlüssel möglich. Ein einfacher Schlüssel besteht aus einer Eigenschaft des Typs, deren Wert jede Instanz des

Werkstudent als Subtyp von Mitarbeiter und Student

Abbildung 70: Ein Beispiel für Mehrfachvererbung

Typs eindeutig identifiziert. Für Mitarbeiter ist dies die Personalnummer *PID*.Dabei ist zu beachten, daß nicht nur Attribute, sondern auch Beziehungen als Schlüssel und Schlüsselbestandteile zulässig sind.

Ein zusammengesetzter Schlüssel besteht aus einer Kombination mehrerer Eigenschaften, die zusammen eindeutig für jede Instanz sind. Zusammengesetzte Schlüssel werden in Klammern eingeschlossen. Im Firmeninfomationssytem besitzt der Typ *Projekt* einen zusammengesetzten Schlüssel:

interface *Projekt(***extent** *projekte* **key** *(Nr, Titel))*
{ ... };

Zu einem Objekttyp kann es auch mehrere alternative Schlüssel geben. In diesem Fall wird eine Liste von mehreren Schlüsseln bei der Schnittstellendefinition angegeben. Abteilungen können z.B. sowohl über ihren Namen als auch über das Kürzel *Kurz* identifiziert werden:

interface *Abteilung (***extent** *abteilungen* **keys** *Name, Kurz)*
{ ... }

Wichtig ist es hierbei, die unterschiedliche Rolle der Schlüssel im ODMG-Objektmodell und in SQL zu sehen. Im ODMG-Modell dienen die Schlüssel allein der Objektselektion und der Modellierung in der Realität existierender Integritätsbedingungen (z.B.„Jeder Mitarbeiter hat eine eindeutige Personalnummer“). In SQL werden Schlüssel zusätzlich für die Modellierung von Beziehungen benötigt (s. Abschnitt 3.2.3).

5.5.6 Subtypdefinitionen

Das ODMG-Objektmodell unterstützt strikte Ererbung von Attributen, Beziehungen und Operationen. Ein Subtyp erbt alle Eigenschaften (Attribute und Relationen) und Operationen von seinen Supertypen. Er kann außerdem die ererbten Charakteristiken verfeinern und durch die Definition weiterer Attribute, Beziehungen und Operationen erweitern.

Da die Subtyphierarchien nicht auf Baumstrukturen beschränkt sind, kann ein Typ mehr als einen direkten Supertyp besitzen. In Erweiterung des Firmeninformationssystems kann man sich einen weiteren Supertyp *Student* für *Werkstudent* vorstellen, wodurch sich die in Abbildung 70 gezeigte Struktur ergibt.

In diesem Fall findet Mehrfachererbung (*multiple inheritance*) statt: Ein Typ erbt die Charakteristiken aller seiner direkten Supertypen. Dabei kann es zu Namenskonflikten kommen, wenn mehrere Supertypen eine Charakteristik des gleichen Namens besitzen. Solche Namenskonflikte müssen im Subtyp durch explizites Umbenennen einer der ererbten Charakteristiken aufgelöst werden.

Die Supertypen eines Typs werden bei der Typdefinition in der Schnittstelle festgelegt. Im erweiterten Beispiel besitzt der Typ *Werkstudent* die Supertypen *Mitarbeiter* und *Student*.

interface *Werkstudent: Mitarbeiter, Student*
(**extent** *werkstudenten)*
{ **attribute** *(xxDatum) Beginn*
attribute Unsigned Short *Dauer*
attribute *(xxPreis) Verguetung* }

Eine Instanz eines Subtyps kann als Instanz jeder seiner Supertypen betrachtet werden. Sie verfügt über das Verhalten und die Zustände, die von einer Instanz der Supertypen erwartet wird. Zusätzlich besitzt sie spezialisierte Zustandsinformation und/oder Verhaltensmuster, die sie als Instanz der Spezialisierung auszeichnen.

Jedes Element eines Subtyps ist auch in den Extensionen all seiner Supertypen enthalten. Wenn für einen Typ A und seinen Subtyp B Extensionen verwaltet werden, ist also die Extension des Typs B eine Teilmenge der Extension des Typs A.

In manchen Fällen erfordert es die Modellierung, daß die Extensionen der direkten Subtypen eines Typs paarweise disjunkt sein sollen, d.h. kein Element des Supertyps gehört zu mehr als einem seiner direkten Subtypen[13]. Eine solche Disjunktheit kann mit den Sprachmitteln des ODMG-Objektmodells nicht erzwungen werden.

Implementierungsvererbung wird durch den ODMG-93-Standard nicht spezifiziert. `nachlesen`

5.5.7 Beziehungen zwischen Typen

Beziehungen zwischen Objekten spielen in der Datenmodellierung eine zentrale Rolle. Im ODMG-Standard werden Beziehungen zwischen Objekten nicht durch ein eigenes übergeordnetes Konstrukt auf der Typebene ausgedrückt, sondern ähnlich wie Attribute als Teil der Strukturdefinition eines Typs definiert. Zur Unterscheidung von den rein wertbasierten Attributen werden Beziehungsdefinitionen durch das Schlüsselwort **relationship** eingeleitet. Im ODMG-Objektmodell können nur binäre, keine n-ären Beziehungen definiert werden. Beziehungen sind nur zwischen Objekten des Typs *Denotable_Object* und damit auch all seiner Subtypen möglich.

Analog zu ein- und mehrwertigen Attributen kann auch bei der Definition einer Beziehung ein einzelnes Objekt oder eine Kollektion von Objekten angegeben werden. Damit lassen sich 1:1-, 1:n- und n:m-Beziehungen darstellen. Es handelt sich dabei in allen Fällen um optionale Beziehungen. Um darzustellen, daß bei einer 1:1-Beziehung einem Objekt kein Objekt zugeordnet wird, verwendet man den ausgezeichneten Wert **nil** anstelle des assoziierten Objekts. Bei 1:n- und n:m-Beziehungen verwendet man bei nicht an der Beziehung beteiligten Objekten die leere Kollektion. anstelle der Spezifikation der assoziierten Objekte. Nicht-optionale Beziehungen wie z.B. jedes Objekt aus A ist mit *genau* einem Objekt aus B assoziiert können mit den Mitteln des ODMG-Objektmodells nicht erzwungen werden (vgl. dazu die Angabe **not null** bei SQL, Abschnitt 4.2.4).

Zur Definition einer 1:1-Beziehungen zwischen zwei Objekttypen A und B wird in der Signatur des Typs A eingeleitet durch das Schlüsselwort **relationship** der Objekttyp B als zweiter an der Beziehung beteiligter Typ angegeben. Als Beispiel wird die Beziehung zwischen den Abteilungen und ihren Leitern betrachtet.

interface *Abteilung (...)*

[13]In OMT wird dies durch ein leeres Dreieck ausgedrückt gegenüber einem gefüllten Dreieck für überlappende Subklassen

{ ...
  relationship *Leiter Chef*
    inverse *Leiter :: Bereich;*
  ... };

interface *Leiter: Festangestellte (...)*
{ ...
  relationship *Abteilung Bereich*
    inverse *Abteilung :: Chef*
  ... }

Beziehungen haben keinen Objektidenifikation und keinen Namen. Es wird jedoch bei der Definition der Beziehung ein Bezeichner eingeführt (*traversal path*), mit dessen Hilfe ein Traversieren der Beziehung möglich ist. Im Beispiel ist dies der Bezeichner *Chef* für das Traversieren der Beziehung von *Abteilungen* zum zugehörigen *Leiter* und der Bezeichner *Bereich* für die umgekehrte Richtung der Beziehung.

Durch Angabe der inverse-Klauseln werden die beiden Richtungen der Beziehung miteinander in Verbindung gebracht. Wenn zu einer Beziehung die inverse Richtung nicht angeben wird, kann die Beziehung nur in der definierten Richtung traversiert werden. Dies kann vom System als Hinweis angesehen werden, daß die Unterstützung zum Traversieren in entgegengesetzter Richtung eingespart werden kann.

Die referentielle Integrität wird durch Beziehungen gewahrt: Das System löst eine Ausnahme aus, falls ein zuvor gelöschtes Objekt über eine Beziehung dereferenziert wird. Aus Anwendersicht ist diese Lösung nicht optimal. Für ihn ist es konsequenter und komfortabler, wenn noch bestehende Referenzen bereits beim Löschen des Objekts vom System erkannt und geeignet behandelt werden. Ein Beispiel für einen solchen Ansatz ist das Kaskadieren von Löschoperationen in SQL (s. Abschnitt 3.2.4). Solche Ansätze sind jedoch aufwendiger in der Realisierung, besonders für objektorientierte Sprachen mit ihrer extensiven Verwendung von Objektreferenzen. Kaskadierendes Löschen kann außerdem auch zu schwer überschaubaren (und auch unerwarteten) Konsequenzen einer einzelnen Löschoperation führen.

Zur Definition einer 1:n-Beziehung zwischen zwei Typen A und B kann zu jedem Objekt aus A eine Kollektion von Objekten aus B angegeben werden (vgl. mehrwertige Attribute). Als Beispiel wird hier die Beziehung zwischen Mitarbeitern und Abteilungen betrachtet: Jeder Mitarbeiter gehört zu einer Abteilung und jede Abteilung hat beliebig viele Mitarbeiter(**Set**< *Mitarbeiter* >).

interface *Abteilung (...)*
{ ...
  relationship **Set**<*Mitarbeiter*> *Angestellte*
    inverse *Mitarbeiter :: Arbeitgeber;*
  ... };

interface *Mitarbeiter(...)*
{ ...
  relationship *Abteilung Arbeitgeber*
    inverse *Abteilung :: Angestellte*
  ...};

Neben **Set** sind auch andere Kollektionstypen möglich. Dies wird im folgenden Beispiel der Definition einer n:m-Beziehung zwischen Projekten und Mitarbeitern illustriert. Bei der n:m-Beziehung wird für beide Richtungen der Beziehung eine Kollektion angegeben.

```
interface Projekt (...)
{ ...
  relationship Set<Mitarbeiter> Beteiligte
    inverse Mitarbeiter :: Projekte;
  ...};

interface Mitarbeiter (...)
{ ...
  relationship List<Projekt> Projekte
    inverse Projekt :: Beteiligte
}
```

Die an einer Beziehung beteiligten Typen müssen nicht unterschiedlich sein, wie an den rekursiven Beziehungen *Oberabteilug* und *Unterabteilungen* zur Modellierung der Abteilungshierarchie zu sehen ist:

```
interface Abteilung (...)
  relationship List<Abteilung> Unterabteilungen
     inverse Abteilung :: Oberabteilung;
     { order_by Name }
  relationship Abteilung Oberabteilung
     inverse Abteilung :: Unterabteilungen;
  ...
 };
```

Für geordnete Kollektionen kann eine Ordnung auf den Objekten angegeben werden. Die Definition einer Ordnung geschieht durch eine order_by-Klausel. In obigem Beispiel werden die Unterabteilungen nach ihrem Namen sortiert.

Auf den Beziehungen sind eine Reihe von Operationen definiert, die sich in vier Gruppen einteilen lassen:

Erzeugung: Mit diesen Operationen können Beziehungen zwischen konkreten Objekten erzeugt werden. Für eine 1:1-Beziehungen können zwei konkreten Objekten assoziiert werden und für eine 1:n-Beziehung ein Objekt mit einer Menge von Objekten.

Erweiterung: Durch diese Operationen können 1:n- und n:m-Beziehungen erweitert werden. Mit einem konkreten Objekt kann ein weiteres Objekt und oder eine Menge weiterer Objekte assoziiert werden. 1:1-Beziehungen können für ein konkretes Objekt nicht erweitert werden.

Löschen: Die Operationen dieser Gruppe erlauben es, die für ein konkretes Objekt bestehenden Beziehungen selektiv oder insgesamt zu löschen. Für n:m-Beziehungen können alle Assoziationen an denen ein bestimmtes Objekt beteiligt ist, gelöscht werden. Die Löschoperationen beziehen sich stets auf eine ausgezeichnete Beziehung, nicht auf alle Beziehungen an denen der Objekttyp beteiligt ist.

Traversieren: Diese Gruppe enthält Funktionen, mit denen von einem Objekt aus auf die mit ihm assoziierten Objekte zugegriffen werden kann (Traversieren der Beziehung). Gemäß der Definition der Beziehungsarten liefern die Operationen zum Traversieren bei einer 1:1-Beziehung ein einzelnes Objekt, während sich bei einer 1:n-Beziehung eine Kollektion von Objekten ergibt. Da sich durch die Art der Definition im ODMG-Standard eine n:m-Beziehung aus zwei 1:n-Beziehungen zusammensetzt, kann die für 1:n-Beziehungen vorgesehene Funktion zum Traversieren der einzelnen Richtungen einer n:m-Beziehung verwendet werden.

Die konkrete Syntax der einzelnen Operationen hängt von der gewählten Sprachanbindung ab.

5.5.8 Klassen als Implementierungen von Typen

Wie in Abschnitt 5.5.5 beschrieben, werden Objekttypen durch Schnittstellen definiert. Die Implementierung der Typen, die textuell von der Schnittstelle getrennt ist, erfolgt in ODMG-konformen Sprachen (`Beispiel`). Zu einer Schnittstelle kann es dabei mehrere Implementierungen geben. `Vorteile??` Eine Schnittstelle zusammen mit einer Implementierung dieser Schnittstelle entspricht dem Konzept der *Klasse*, wie es in Smalltalk oder C++ vorhanden ist.

Das Problem der Auswahl einer der vorhandenen Implementierungen spielt nur bei der Erzeugung von Objekten eine Rolle, sie kann nicht dynamisch geändert werden. Wenn man die Objektidentität einmal besitzt, findet eine automatische Auswahl der „passenden" Implementierung durch das System statt. Dies ermöglicht multi-linguale Systeme, da in einer Applikation vorkommende Objekte in verschiedenen Sprachen geschriebene Implementierungen besitzen können.

Die Implementierung darf mehr Operationen enthalten als die Schnittstelle (private Attribute und Operationen).

`Operationsimplementierung kann immer auf aktuelles Objekt zugreifen (self, this, ...). Methodenaufruf. Ausnahmebehandlung nicht weiter diskutiert.`

5.5.9 Späte Bindung

Namen von Operationen müssen nur innerhalb einer Schnittstellendefinition eindeutig sein. Es können also in unterschiedlichen Typdefinitionen Operationen mit dem gleichen Namen definiert werden. Man spricht dann von *überladenen* Operationen (*operation overloading*).

Beim Aufruf einer Operation mit einem überladenen Namen tritt das Problem der Auswahl der zugehörigen Operation auf. Dieser Auswahlvorgang wird als Namensauflösung oder *operation dispatching* bezeichnet. Bei der Auswahl der Operation richtet sich das System nach dem Typ des ausgezeichneten ersten Parameters der Operation. Jede Operation eines Typs kann auch auf die Objekte all seiner Subtypen angewandt werden. Es kann deshalb mehrere Operationen des gleichen Namens geben, die aufgrund der Subtyphierarchie auf ein Objekt anwendbar sind. Grundsätzlich wird beim *operation dispatching* die Operation ausgewählt, die für den spezifischsten Typ des Objekts definiert ist.

Abbildung 71: *Operation dispatching*

Abbildung 71 zeigt zur Illustration einen Teil der Typhierarchie des Firmeninformationssystems. Es wird angenommen, daß sowohl der Typ *Mitarbeiter* als auch der Typ *Festangestellter* eine Operation *Jahresgehalt* definiert. Auf ein Objekt vom Typ *Festangestellter* können beide Operationen ausgewählt werden, beim *operation dispatching* wird die Operation des Typs *Festangestellter* ausgewählt (spezifischster Typ). Es werden zwei Arten des *operation dispatching* unterschieden:

- Beim *static dispatching* stellt der Compiler zum Übersetzungszeitpunkt den Typ des Objekts m fest und wählt in Abhängigkeit davon die Operation aus. Diese Art des *dispatchings* ist z.B. in Ada realisiert.

- Beim *dynamic dispatching* verzögert der Compiler die Auswahl bis zur Laufzeit. Die Auswahl hängt vom dynamischen Typ des an die Variable m gebundenen Objekts ab. Diese Art des *operation dispatching* ist in den meisten objektorientierten Sprachen wie z.B. in Eiffel und in C++ realisiert.

Der Unterschied zwischen den beiden Arten des *operation dispatchings* läßt sich am besten an einem Beispiel illustrieren. Man betrachtet dazu das folgende Beispiel einer OQL-Anfrage[14], die das durchschnittliche Jahresgehalt berechnet:

avg(**select** *m.Jahresgehalt()*
    from *mitarbeiter m*)

Die Anfrage führt eine Variable m vom Typ *Mitarbeiter* ein, die an die Extension dieses Typs gebunden ist. Für diese Variable. wird die Operation *Jahresgehalt* aufgerufen. Wenn die Anfrage für ein Objekt vom Typ *Festangestellter* ausgewertet wird wählt der Compiler bei *static dispatching* zur Übersetzungszeit die Operation *Jahresgehalt* des Typs *Mitarbeiter* aus. Bei *dynamic dispatching* hingegen wird zur Laufzeit festgestellt, daß das betrachtete Objekt vom Typ *Festangestellter* ist, und es wird die Operation *Jahresgehalt* dieses Typs gewählt.

Der ODMG-Standard geht von der Realisierung der zweiten Art des *operation dispatchings* aus.

Späte Bindung erhöht die Wartbarkeit und Erweiterbarkeit von Softwaresystemen. Es können auch noch nach der Realisierung von Applikationen neue Subklassen eingeführt werden. Die bisherigen Prozeduren können unverändert weiterverwendet werden, tragen aber der erweiterten Subklassenhierarchie Rechnung, indem sie gegebenenfalls den Code der neuen Subklasse verwenden. Es kann also in einer Prozedur auf Implementierungen von Subklassen zugegrifffen werden, die zum Zeitpunkt der Erstellung der Prozedur noch gar nicht geplant waren. Meyer [**?**] spricht in diesem Zusammenhang von geschlossenen und offenen Systemen: Eine Klassenhierarchie ist geschlossen, da man schon mit ihr arbeiten kann; sie ist aber zugleich auch offen für spätere Erweiterungen.

Im Bereich der Datenbanksysteme führt die späte Bindung wegen der dynamischen Zuordnung der Implementierungen zu einer Erschwerung der Anfrageoptimierung (vgl. dazu [] `Paper zur Optimierung von OO-Anfragen`).

[14]Die Syntax von OQL ist der von SQL sehr ähnlich. Sie wird in Abschnitt 5.6 beschrieben

5.5.10 Sichtbarkeits- und Lebensdauerregeln

Jedes Objekt einer ODMG-Datenbank besitzt eine innerhalb dieser Datenbank gültige und eindeutige, unveränderliche Objektidentität, die ihm bei seiner Erzeugung zugeordnet wird. Der Gleichheitstest auf Objekten basiert demgemäß nicht auf der Gleichheit der Objektzustände sondern auf der Indentität der Objekte. Die Operation *x.same_as?(y)*, die für ein Objekt x aufgerufen werden kann, vergleicht die Objektidentitäten der Objekte x und y.

Bei der Erzeugung eines Objekts wird eine Objektidentität zurückgeliefert. Wenn keine Bindung an einen Bezeichner stattfindet, erlaubt diese jedoch keinen Zugriff auf das Objekt. Objekte sind deshalb zunächst anonym und erfordern zu ihrer weiteren Verwendung eine Bindung an einen Bezeichner (Benennung). Zum Zugriff auf Objekte innerhalb einer Sitzung können Objekte an Programmvariablen gebunden werden, die jedoch nur bis zum Ende der Sitzung gültig sind. Eine Identifizierung von Objekten unabhängig von Sitzungen ist über eine Benennung der Objekte auf der Ebene der Schemadefinition möglich. Zusammen mit den Namen der Extensionen bilden diese Objektnamen Eintrittspunkte in die Datenbank (*roots of persistence*).

Im ODMG-Objektmodell können Objekte auf der Schemaebene benannt werden, wobei mehrere Namen für daselbe Objekt zulässig sind. Jeder Name darf jedoch innerhalb seines Sichtbarkeitsbereichs nur einmal vergeben werden. Es kann zur Laufzeit festgestellt werden, ob ein Objekt benannt ist und gegebenenfalls auch, welches der bzw. die Namen sind. Zu diesem Zweck gibt es die vordefinierten Eigenschaften *has_name?* und *names* des Typs *Object*, die an alle Subtypen von *Object* vererbt werden. Für ein benanntes Objekt gilt *has_name?* = **true** und das Attribut *names* besitzt die Menge der Namen des Objekts als Wert.

Es gibt also im ODMG-Objektmodell verschiedene sitzungsunabhängige Zugriffsmöglichkeiten auf ein Objekt:

- Über die benannte Extension des zugehörigen Typs kann das Objekt durch eine (wertbasierte) Selektion z.B. über die Konstrukte der Anfragesprache OQL (s. Abschnitt 5.6) angesprochen werden; vgl. OQL,...??
- Über die ihm zugeordneten Namen, kann das Objekt direkt angesprochen werden.
- Über einen Pfad ausgehend von einem benannten Objekt kann ein Objekt evtl. indirekt angesprochen werden.

Abbildung 72: Optionen für den Zugriff auf Objekte

`Bild siehe Block, anpassen an Text`

Abbildung 72 illustriert die verschiedenen Optionen für den Zugriff auf Objekte. Objekt A kann innerhalb der Sitzung 1 über die Variable x und innerhalb der Sitzung 2 über die Variable y sowie sitzungsunabhängig über seinen Namen *nameA* und als Mitglied der Extension seines Typs *Typ1* angesprochen werden. Objekt B kann indirekt über das benannte Objekt A und damit auch indirekt über die Variablen x und y und auch als Mitglied der Extension seines Typs *Typ2* angesprochen werden.

Der Sichtbarkeitsbereich von Schnittstellennamen und *benannten Wurzelobjekten* ist die gesamte Datenbank. Schnittstellendefinitionen und Typimplementierungen bilden lokale Sichtbarkeitsbereiche. Die Namen von Attributen und Operationen brauchen nur innerhalb dieser

Bereiche eindeutig zu sein. Im Abschnitt 5.5.6 werden die Sichtbarkeitsregeln für ererbte Attribute beschrieben.

Implementierungen werden relativ zu ihren Schnittstellen benannt. Die Namen von Implementierungen brauchen daher nur in diesem Sichtbarkeitsbereich eindeutig zu sein.

Die Lebensdauer eines Objekts ist orthogonal zu seinem Typ (*orthotogonal persistence*). Sie beginnt mit seiner (dynamischen) Erzeugung. Im Gegensatz zu den übrigen Operationen benötigt man zur Objekterzeugung eine Operation, die nicht für eine Instanz des Typs (ein Objekt), sondern für den Typ selbst aufgerufen wird, da die zu erzeugende Instanz noch nicht existiert. Die Operation *create_instance* ist deshalb für den Typ *Type* vordefiniert, der alle übrigen Objekttypen als *Instanzen* enthält (s.Abschnitt 5.5.11). Damit kann diese Operation für all diese Typen aufgerufen werden. Sie besitzt die folgende Signatur:

create_instance($e_1 = v_1$, $e_2 = v_2$, $\ldots$ $e_n = v_n$): *Denotable_Object*

Bei der Objekterzeugung können für alle oder einige der Objektattribute Werte festgelegt werden. Dazu werden Paare aus Attributnamen e_i und Attributwert v_i als Parameter an die create_instance-Operation übergeben. Nicht explizit initialisierte Attributwerte erhalten den Wert **nil**. Als Ergebnis liefert die Operation die Objektidentität des neu erzeugten Objekts. Nach seiner Erzeugung kann der Typ eines Objekts jederzeit ermittelt werden. Diesem Zweck dient die für den Typ *Object* vordefinierte Eigenschaft *type*, die einen Verweis auf den zugehörigen Objekttyp (ein Objekt vom Typ *Type*) enthält.

Die Lebensdauer eines Objekts wird bei der Erzeugung des Objekts festgelegt und kann danach nicht mehr geändert werden. Das ODMG-Objektmodell bietet dafür folgende drei Optionen:

conterminus_with_procedure: Die Lebensdauer des Objekts ist beschränkt auf die Prozedur, in der es erzeugt wurde. Es wird bei der Rückkehr aus der Prozedur automatisch gelöscht.

conterminus_with_process: Die Lebensdauer des Objekts ist beschränkt auf den Prozeß, innerhalb dessen es erzeugt wurde. Es wird am Ende der jeweiligen Sitzung automatisch gelöscht.

conterminus_with_database: Die Lebensdauer des Objekts ist beschränkt auf die Lebensdauer der Datenbank, zu der es gehört. Um die Lebensdauer des Objekts zu beenden, muß es explizit gelöscht werden. Zum expliziten Löschen eines Objekts definiert der Typ *Object* die generische Operation **delete**, die an alle Subtypen von *Object* vererbt werden. Diese Operation entfernt das zu löschende Objekt aus der Datenbank und markiert es in allen Beziehungen, an denen es beteiligt ist, als ungültig (siehe auch Abschnitt 5.5.7). Der Versuch, auf ein explizit gelöschtes Objekt über einen Namen oder einen Pfad zuzugreifen, führt zu einer Ausnahme.

Es ist zulässig, daß sich ein Objekt auf ein anderes mit einer kürzeren Lebensdauer bezieht, allerdings nur bis zum Ende von dessen Lebensdauer.

5.5.11 Zugriff auf Metadaten über Objekte des Typs „Type“

Typen sind selbst durch Objekte repräsentiert und können deshalb auch Instanzen eines Typs sein. Alle Typen der vordefinierten Typhierarchie (s. Abschnitt 5.5.1) sind Instanzen des Typs *Type*, der Subtyp des Typs *Atomic_Object* ist. Insbesondere ist der Typ *Type* selbst auch durch ein Objekt des Typs *Type* und damit auch des Typs *Atomic_Object* repräsentiert. Daraus ergeben sich die folgenden Konsequenzen:

- Jeder Typ der Typhierarchie und jeder benutzerdefinierte Typ, der in diese Hierarchie eingefügt wird, besitzt die Eigenschaften, die durch *Type* definiert werden (siehe Tabelle).
- Als Instanz des Typs *Atomic_Object* besitzt *Type* die Eigenschaften, die durch den Typ *Atomic_Object* definiert sind.
- Da *Type* auch Subtyp von *Atomic_Object* ist, erben alle Instanzen von *Type*, d.h. alle Typen der Typhierarchie und die benutzerdefinierten Typen (s.o.) die Eigenschaften von *Atomic_Object*.

Die Eigenschaften des Typs *Type* sind in der folgenden Tabelle zusammengefaßt.

Name der Eigenschaft	**Typ der Eigenschaft**	**Erklärung**
has_operations	**Set**< Operation >	Menge der Operationen
has_properties	**Set**< Property >	Menge der Eigenschaften
has_subtypes	**Set**< Type >	Menge der Subtypen
has_supertypes	**Set**< Type >	Menge der Supertypen
name	**String**	Name des Typs
extent	**Set**< Atomic_Object >	Extension des Typs
name_of_extent	**String**	Name der Extension

Weiterhin definiert *Type* die Operationen *create_instance* (s. Abschnitt 5.5.10) und *create_extent*, die die implizit verwaltete Extension initialisiert.

Ähnlich wie in relationalen Systemen, bei denen die Metadaten in Tabellen abgelegt sind, können auch im hier beschriebenen Ansatz Metadaten über die ODL beschrieben und mit der Sprache OQL abgefragt werden.

5.6 Anfragen in OQL

OQL ist eine deklarative Anfragesprache mit einer abstrakten und einer konkreten, SQL-ähnlichen Syntax, die im folgenden betrachtet wird. Im Gegensatz zur Sprache SQL, bei der Anfragen im wesentlichen aus select from where-Konstrukten bestehen müssen, bietet OQL ein weites Spektrum von Ausdrücken an, die als Anfragen zugelassen sind.

OQL wurde als eine Obermenge von SQL entworfen. Jede korrekte select from where-Anweisung in SQL ist ebenso gültig in OQL. Die Erweiterungen von OQL betreffen komplexe Objekte, Objektidentität, Pfadausdrücke, Polymorphismus, Funktionsaufrufe und späte Bindung.

OQL unterstützt sowohl Werte (unveränderlich) als auch Objekte (veränderlich). Weiterhin ist es weniger auf eine bestimmte Art von Kollektionen wie etwa die Relationen in SQL (=Multimengen) fokussiert. Neben Anfragen auf Mengen und Mulimengen sind auch Anfragen auf geordnete Kollektionen wie Listen und Felder möglich.

Eine OQL-Anfrage besteht aus einem Ausdruck, der zur Berechnung des Anfrageergebnisses ausgewertet wird. Neben einfachen Ausdrücken wie Bezeichnern von Werten oder

Objekten und Komponentenselektionen spielen dabei Funktionen eine wichtige Rolle. Dabei bestimmt der für die Funktionsparameter berechnete Rückgabewert das Ergebnis der Anfrage. Zu diesem Zweck ist eine Reihe von Funktionen in OQL vordefiniert, von denen sich viele in Infix-Notation anwenden lassen.

Das Spektrum der zulässigen OQL-Anfrageausdrücke läßt sich in fünf Gruppen aufteilen:

- Elementare Anfragen (Abschnitt 5.6.3)
- Konstruktoren für Objekte und Werte (Abschnitt 5.6.4)
- Funktionale Anfragen (Abschnitt 5.6.5)
- Quantifizierte Mengenanfragen (Abschnitt5.6.6)
- Anfragen auf geordnete Kollektionen (Abschnitt5.6.7)

Nach einer Behandlung der Aspekte Sprachorthogonalität und Benennung von Anfragen in den Abschnitten 5.6.1 und 5.6.2 werden diese fünf Gruppen von Ausdrücken beschrieben.

5.6.1 Sprachorthogonalität

Ein deutlicher Vorteil von OQL gegenüber von SQL ist die erhöhte Sprachorthogonalität. Im Gegensatz zu den Einschränkungen in SQL (vgl. Abschnitt 3.5) können die Konstrukte der Anfragesprache bei OQL beliebig miteinander kombiniert werden. Vgl. mit SQL. Reguläre Semantik. Einzelner Name ist bereits eine Anfrage. (Zwischen-)ergebnisse beliebiger Typen.

Der Vorteil der hohen Sprachorthogonalität läßt sich gut am select from where-Konstrukt illustrieren, das im wesentlichen mit dem entsprechenden SQL-Konstrukt übereinstimmt. Die mehrmalige Verwendung dieses Konstrukts innerhalb einer Anfrage führt zu geschachtelten Anfragen. In OQL ist ein geschachteltes select from where-Konstrukt in folgenden Teilen der Anfrage möglich:

select-Klausel: Ein Beipiel dafür ist die Anfrage

„Angestellte der Abteilung PC-Software, die am Projekt DB-Fahrpläne mitarbeiten.“

select select *ang*
  from *abt.Angetellte ang*
  where *dbFahrplaeneProjekt* **in** *ang.Projekte*
from *abteilungen abt*
where *abt.Kurz = 'PC SW';*

Das äußere select from where-Konstrukt liefert hier die Abteilung PC-Software und damit deren Angestellte (*abt.Angestellte*). Das innere Konstrukt selektiert aus dieser Menge von Angestellten die für das Anfrageergebnis relevanten.

from-Klausel: Ein Beipiel dafür ist die Anfrage

„Die Leiter aller Oberabteilungen, die weniger als DM 10.000 verdienen.“

select oAbt.Chef
from select *abt*
  from *abteilungen abt*
  where not *a.Unterabteilungen.empty? oAbt*
where *oAbt.Chef.Gehalt* < *10000*

Das innere select from where-Konstrukt selektiert hier die Menge der Oberabteilungen (=Abteilungen mit mindestens einer Unterabteilung).

where-Klausel: Ein Beipiel dafür ist die Anfrage

„Abteilungen mit mehr als zehn Mitarbeitern, die am Projekt DB-Fahrpläne arbeiten.“

select *abt*
from *abteilungen abt*
where *10* < *count(***select** *ang*
  from *abt.Angetellte ang*
  where *dbFahrplaeneProjekt* **in** *ang.Projekte)*

Das innere select from where-Konstrukt selektiert nacheinander für jede Abteilung die Menge der Angestellten, die an dem gewüschten Projekt arbeiten.

Hohe Flexibilität zeigt die Anfragesprache OQL auch bei der Behandlung von Objekten und Werten. Anfragen können existierende Objekte, atomare Literale, strukturierte Werte und Kollektionen der selbigen zurückliefern, aber auch selbst neue Objekte erzeugen und Objekte und Werte zu beliebig strukturierten Konstrukten kombinieren.

OQL selbst ist nicht algorithmisch vollständig (*computationally incomplete*), aber es ist möglich, in einer Anfrage Methoden aufzurufen und umgekehrt auch Anfragen in Methoden zu verwenden. Dadurch ergibt sich eine hohe Ausdrucksmächtigkeit. `Siehe dazu neuer Text !!`

Wie SQL ist auch OQL eine deklarative Anfragesprache. Dies erleichtert die Formulierung von Anfragen und eröffnet gute Optionen für die Optimierung.

5.6.2 Benannte Anfragen

OQL unterstützt ein Konstrukt für die Benennung von Anfragen. Für eine Anfrage kann durch das define as-Konstrukt ein Bezeichner eingeführt werden. Die folgende Definition bindet den Bezeichner *Oberabteilungen* an die Anfrage, die die Menge der Abteilungen mit mindestens einer Unterabteilung bestimmt.

define *Oberabteilungen* **as**
 select *abt*
 from *abteilungen abt*
 where not *abt.Unterabteilungen.empty?*

Das select from where-Konstrukt wird in Abschnitt 5.6.6 erläutert.

Ein auf diese Weise gebundener Bezeichner kann unter anderem in anderen Anfrageausdrücken innerhalb des Sichtbarkeitsbereichs des Namens anstelle des Anfrageausdrucks benutzt werden. So kann z.B. obige Anfrage unter Verwendung des Bezeichners wie folgt vereinfacht werden:

select *oAbt.Chef*
from *Oberabteilungen oAbt*
where *oAbt.Chef.Gehalt* < 10000

OQL unterstützt keine rekursiven Anfragen. Da Anfragen jedoch Methodenaufrufe und umgekehrt Metoden auch Anfragen enthalten können, kann eine Anfrage eine Operation enthalten, die die Anfrage rekursiv aufruft (indirekte Rekursion).

5.6.3 Elementare Anfragen

Literale und Bezeichner von Objekten und Typen sind bereits zulässige Anfragen in OQL. Als Eintrittspunkte für die Anfragesprachen dienen die Bezeichner der Wurzelobjekte. Dazu gehören die Namen der Extensionen der im Datenbankschema definierten Objekttypen und benannte Objekte (vgl. Abschnitt 5.5.10). Die Extensionen spielen dabei dieselbe Rolle wie die Relationen in SQL.

Für die Anwendung einer Operation (Methode) f auf eine Objekt o gibt es zwei verschiedene Notationen in OQL:

$o \rightarrow f$
$o.f$

Weiterhin gibt es die Möglichkeit ein Objekt x explizit zu typisieren. Dazu wird dem Objekt der Bezeichner T seines Typs eingeklammert vorangestellt.

(T) x

Die Korrektheit der Typangabe wird vom System getestet. Falls die Typangabe nicht korrekt ist, wird zur Laufzeit eine Ausnahme erzeugt. Dabei ist zu beachten, daß auch jeder Supertyp des Typs, für den das Objekt erzeugt worden ist eine korrekte Typangabe ist (s Abschnitt 5.5.6). Die explizite Typangabe eröffnet außerdem die Möglichkeit, daß ein Objekt auf eine Eigenschaft einer seiner Supertypen zugreift (die in dem eigenen Objektyp überschrieben wurde??), indem nicht der Objekttyp sondern der Bezeichner des entsprechenden Supertyps angegeben wird.

xx

Wenn die explizite Typisierung zu diesem Zweck verwendet wird, stellt sie eine Art Typkonvertierung (*Typ-Cast*) dar.

5.6.4 Konstruktoren für Objekte und Werte

Entsprechend der klaren Unterscheidung zwischen Werten und Objekten im ODMG-Standard (s. Abschnitt 5.5.2) gibt es auch verschiedene Konstrukte für deren Erzeugung. Zur Erzeugung eines neuen Objekts wird der Name seines Typs als Konstruktor verwendet. Als Parameter erhält der Konstruktor Initialierungswerte für Attribute des Objekts. Dabei wird für jedes Attribut auch der Name angegeben, d.h. es werden Paare Attributname/Attributwert als Parameter übergeben. Attribute, die nicht explizit initialisiert werden, erhalten den Standardwert **nil**. Mit dem folgenden Ausdruck wird ein neues Objekt vom Typ *Abteilung* erzeugt.

Abteilung(Kurz: 'LTSW', Name: 'Leitung Software')

(siehe Seiteneffekte S. 70 ODMG-Standard) `Mengenwertige Objekte??`

Die Erzeugung von strukturierten Werten hingegen wird durch das Schlüßselwort **struct** eingeleitet. Die Werte der Eigenschaften werden wiederum wie bei den Objekten durch Paare aus Namen und Werten angegeben.

struct*(p1: e1, p2: e2, ... pn: en)*

Es ist insbesondere zu beachten, daß die beiden Ausdrücke

Person(Name: 'Peter', Alter: 20)
struct*(Name: 'Peter', Alter: 20)*

eine ganz unterschiedliche Bedeutung besitzen. Der erste Ausdruck erzeugt ein Objekt vom Typ *Person* mit den veränderlichen Eigenschaften *Name* und *Alter*. Der zweite Asdruck erzeugt einen strukturierten Wert mit den unveränderlichen Komponenten *Name* und *Alter*. Insbesondere liegt den beiden Konstrukten ein unterschiedlicher Gleichheitsbegriff zugrunde. Die Ergebnisse einer zweimaligen Auswertung des ersten Ausdrucks besitzen unterschiedliche Objektidentitäten und werden deshalb als ungleich betrachtet. Es handelt sich um zwei Objekte, die zufällig die gleichen Werte besitzen. Beim zweiten Ausdruck ergeben sich jedoch bei zweimaliger Auswertung identische, nicht-unterscheidbare strukturierte Werte.

Die Konstruktion einer Kollektion erfolgt durch einen entsprechenden Konstruktor (**set**, **bag**, **list**, **array**) gefolgt von der Enumeration der Ausdrücke, die die Elemente der Kollektion bilden.

Die folgende Tabelle faßt die Konstruktoren in abstrakter Form zusammen. Dabei sind die e_i Ausdrücke, die p_i Namen von Eigenschaften und t der Name eines Typs.

Konstruktor	**Parameter**	**zur Erzeugung ...**
t	p_1: e_1, p_2: e_2, $\ldots p_n$: e_n	eines Objektes vom Typ t
struct	p_1: e_1, p_2: e_2, $\ldots p_n$: e_n	eines strukurierten Wertes
set	e_1, e_2, $\ldots e_n$	einer Menge
bag	e_1, e_2, $\ldots e_n$	einer Multimenge
list	e_1, e_2, $\ldots e_n$	einer Liste
array	e_1, e_2, $\ldots e_n$	eines Feldes

5.6.5 Funktionale Anfragen

Zur Formulierung von funktionalen Anfragen stehen in OQL eine Reihe von Operatoren (Funktionen) zur Verfügung. Neben den üblichen unären (z.B. **not**, -) und binären Boolschen (**and**[15], **or**) und arithmetischen Operatoren (-, +, etc.) und Vergleichsoperatoren (+, < , etc.) enthält OQL die unären Operatoren **min**, **max**, **count**, **sum**and **avg**, die auf Kollektionen angewandt werden können. Sie sind vergleichbar mit den Aggregatfunktionen bei SQL (vgl. Abschnitt 4.3.8). So kann man z.B. mit der Anfrage

count*(abteilungen)*

[15] *A* **and** *B* wird von links nach rechts ausgewertet.Dies spielt wegen der möglichen Dereferenzierung von **nil** eine Rolle.

die Anzahl aller Abteilungen ermitteln.

Ein weiterer binärer Infixoperator von OQL ist **in**. Dieser Operator überprüft, ob der als erster Parameter übergebene Ausdruck in der Kollektion, die als zweiter Parameter angegeben wird, enthalten ist. Ein solcher Elementtest kann auf beliebige Kollektionsarten angewandt werden. Für zwei konkrete Abteilungen *a1* und *a2* kann man so z.B. testen, ob *a2* eine Unterabteilung von *a1* ist:

a2 **in** *a1.Unterabteilungen*

Darüberhinaus bietet OQL Operatoren an, mit denen sich Ausdrücke umformen (konvertieren) lassen:

- Häufig liefern Anfragen einelementige Kollektionen als Ergebnis. Mit dem Operator **element** läßt sich das (einzige) Element zur Weiterverarbeitung aus einer solchen Kollektion extrahieren. Wenn die Kollektion mehr als einElement enthält wird eine Ausnahme erzeugt. Dieser Operator ist insbesondere deshalb von Bedeutung, da das für die Anfragesprache wichtige select from where-Konstrukt stets Kollektionen (Mulimengen oder Mengen) als Ergebnis liefert. Das Beipiel zeigt einen wichtigen Spezialfall, bei dem ein Element über den Schlüssel (eindeutig!) aus einer Extension, in diesem Fall der Menge der Mitarbeiter selektiert wird:

element(**select** *m*
  from *mitarbeiter m*
  where *m.PID = 22)*

  Das Ergebnis der select from where-Anfrage ist vom Typ **Bag**<Mitarbeiter>, das Ergebnis der Gesamtanfrage ist vom Typ *Mitarbeiter*.

- Der Operator **listtoset** formt eine Liste in eine Menge um, d.h. es wird eine Duplikatelimination durchgeführt und das Ergebnis ist eine ungeordnete Kollektion (Beispiel s.u.).

- Der Operator **flatten** vermindert die Schachtelungstiefe von ineinander geschachtelten Kollektionen. Er macht aus einer Kollektion von Kollektionen von Ausdrücken von Typ *T* eine Kollektion von Ausdrücken vom Typ *T*, indem er alle Ausdrücke in einer Kollektion vereinigt. Es kann sich dabei in den verschiedenen Schachtelungstiefen um verschiedene Kollektionsarten handeln. Die Art der Ergebniskollektion hängt von den beteiligten Kollektionsarten ab.

flatten(**select** *a.Angestellte*
  from *a* **in** *Abteilungen*
  where *ProjektDBFahrplaene* **in** *a.Projekte)*

  Die select from where-Anfrage liefert die Menge der Mengen von Angestellten, deren Abteilungen am Projekt *ProjektDBFahrplaene* beteiligt ist. Der Operator **flatten** macht daraus die Menge aller Angestellten, deren Abteilungen am Projekt *ProjektDBFahrplaene* beteiligt ist.

Die binären Mengenoperatoren **union** (Vereinigung), **intersect** (Durchschnitt) und **except** (Differenz) sind anwendbar auf ungeordnete Kollektionen mit und ohne Duplikate, also auf Objekte vom Typ *Set* und vom Typ *Bag*. Alle drei Operatoren werden in Infixnotation verwendet und erwarten zwei Kollektionen als Operatoren. Unter Verwendung der oben definierten *Oberabteilungen* und einer entsprechend definierten Menge von *Unterabteilungen* kann man mit dem Infixoperator **intersect** z.B. die Menge aller Abteilungen, die sowohl Ober- als auch Unterabteilung sind, berechnen:

Oberabteilungen **intersect** *Unterabteilungen,*

Für geordnete Kollektionen wie Listen und Felder gibt es den Infixoperator +, der zwei Kollektionen der gleichen Art konkateniert. Damit läßt sich die Menge der Projekte an denen Abteilung *a1* oder *a2* arbeitet, bestimmen:

listtoset*(a1.Projekte + a2.Projekte)*

Hierbei wird der oben erwähnte Operator **listtoset** eingestzt, um Duplikate zu eliminieren, die evtl. aufgrund der Beteiligung beider Abteilungen an gleichen Projekten auftreten.

Alternativ kann diese Menge auch unter Verwendung des Operators **union** zur Mengenvereinigung berechnet werden:

listtoset(a1.Projekte) **union** *listtoset(a2.Projekte)*

5.6.6 Quantifizierte Mengenanfragen

Die quantifizierten Mengenanfragen sind den funktionalen Anfragen, die im vorigen Abschnitt vorgestellt werden, sehr ähnlich. Sie führen jedoch zusätzlich eine Bereichsvariable mit lokalem Sichtbartkeitsbereich ein.

Eine Bereichsvariable x kann in OQL universell und existentiell quantifiziert werden. Dazu stehen die folgenden Konstrukte zur Verfügung:

for all x **in** e_1: e_2
exists x **in** e_1: e_2

Das for all-Konstrukt (universelle Quantifizierung) liefert den Wahrheitswert **true**, wenn das Prädikat e_2 für alle Elemente, die in der durch den Ausdruck e_1 beschriebenen Ausdruck enthalten sind, erfüllt ist. Bei der existentiellen Quantifizierung (**exists**) genügt es, wenn das Prädikat e_2 für eines der durch e_1 beschriebenen Elemente erfüllt ist. Man beachte, daß die Syntax und Semantik des exists-Konstrukts von der des gleichnamigen SQL-Konstrukts abweicht: Das SQL-Konstrukt realisiert einen Test auf leere Menge und keine existentielle Quantifizierung und wird syntaktisch wie eine Aggregatfunktion auf eine Menge angewandt.

Neben der universellen und der existentiellen Quantifizierung ist das **select from where** - Konstrukt von zentraler Bedeutung. Dieses entspricht im wesentlichen dem gleichnamigen SQL-Konstrukt, zeichnet sich jedoch durch höhere Orthogonalität (s. Abschnitt 5.6.1) und die Inkorporation objektorientierter Aspekte aus. Das select from where-Konstrukt ist in OQL von der Form

select e
from $e_1\ x_1,\ e_2\ x_2,\ \ldots,\ e_n\ x_n$
where e'

Dabei wird für jeden der an der Anfrage beteiligten Kollektionen, die durch die Ausdrücke e_i beschrieben werden eine Bereichsvariable x_i eingeführt. Jede der Bereichsvariablen x_i ist an die Kollektion gebunden, die durch den zugehörigen Ausdruck e_i beschrieben wird. Wie bei SQL umfaßt der Sichtbarkeitbereich dieser Variablen die gesamte Anfrage (vgl. Abschnitt 4.3.4). Sie können also in den Ausdrücken *e* und *e'* auftreten. Eine Besonderheit des objektorientierten Ansatzes ist die Verwendung einer Bereichsvariablen x_i in einem nachfolgenden Ausdruck e_j ($j>i$). Dies ermöglicht Anfragen auf mengenwertige Attribute und Beziehungen eines Objekts. Als Beispiel wird noch einmal die Anfrage „Angestellte der Abteilung PC-Software, die am Projekt DB-Fahrpläne mitarbeiten" aus Abschnitt 5.6.1 betrachtet. Sie kann auch wie folgt formuliert werden:

select *ang*
where *abt.Kurz = 'PC SW'* **and** *dbFahrplaeneProjekt* **in** *ang.Projekte*

Weiterhin sind in OQL wie an allen anderen Stellen auch für *e* beliebige OQL-Ausdrücke möglich. Dies eröffnet Möglichkeiten , die weit über die bei SQL unterstützte Selektion und Umbenennung von Spalten hinausgeht. Es können unter anderem beliebig strukturierte Werte konstruiert werden und sogar neue Objekte erzeugt werden. Diese hohe Sprachmächtigkeit ist eine klare Konsequnez der Orthogonalität der Sprache.

`Beispiel`

Das Ergebnis einer solchen Anfrage ist eine Multimenge, d.h. es kann Duplikate enthalten. Zur Vermeidung von Duplikaten kann das Konstrukt **select distinct from where** benutzt werden. Es wird dann eine Duplikatelimination durchgeführt, und das Anfrageergebnis ist eine Menge.

Wenn die Ausdrücke $\mathrm{e}_1, \mathrm{e}_2, \ldots, \mathrm{e}_n$ unterschiedliche Kollektionstypen und insbesondere auch geordnete Kollektionen liefern, werden zur Bestimmung des Anfrageergebnisses gewisse Konvertierungen zwischen den Kollektionsarten durchgeführt, auf die hier aber nicht weiter eingegangen wird.

In SQL und in OQL kann man für den Ausdruck *e* in der select-Klausel einen Stern angegeben. In SQL erhält man dann die Spalten aller in der from-Klausel in der Ergebnistabelle. (vgl. Abschnitt 4.3.1). In OQL werden für das Ergebnis Strukturen konstruiert mit den Namen ... der Bereichsvariablen als Komponentennamen und den Elementtypen der durch die Ausdrücke e_i als Komponententypen. Die Elemente des Ergebnisses sind dann also vom Typ:

Structure< x_1: *Elementtyp*(e_1), x_2: *Elementtyp*(e_2), ..., x_n: *Elementtyp*(e_n)>

Wie in SQL gibt es auch in OQL optionale Erweiterungen des select from where-Konstrukts. Zur Sortierung des Anfrageergebnisses können mit Hilfe der order by-Klausel Sortierkriterien angegeben werden. In der Klausel können die in der Anfrage eingeführten Bereichsvariablen referenziert werden. Die Sortierklausel ist von der Form

order by $\mathrm{e}_1, \mathrm{e}_2, \ldots, e_n$

Dabei geben die e_i die Sortierkriterien an, wobei für jedes Sortierkriterium optional festgelegt werden kann, ob in aufsteigender oder absteigender Reihenfolge sortiert werden soll. Der Operator kann auch zur Sortierung von ungeordneten Kollektionen wie Mengen und Multimengen benutzt werden, das Ergebnis ist auf jeden Fall eine Liste. Der folgende Ausdruck sortiert die Menge der Oberabteilungen absteigend nach der Anzahl ihrer Mitarbeiter und bei gleicher Mitarbeiterzahl alphabetisch nach ihrem Namen und liefert eine sortierte Liste:

select *abt*
from *abteilungen abt*
wherenot *abt.Unterabteilungen.empty?*
order by *(count(abt.Angestellte)* **desc**,
abt.Name **asc**

Zusästzlich kann eine sfw- um eine group by-KLausel erweitert werden die es erlaubt, die Elemente eine Kollektion nach bestimmten Kriterien in mehrere Partitionen zu gruppieren.

group by $(p_1\colon e_1,\ p_2\colon e_2,\ \ldots\ \ p_n\colon e_n)$

Dabei ist x die Bereichsvariable, die an die durch e beschriebene Kollektion gebunden ist. Die group by-Klausel legt das Kriterium für die Partitionierung fest. Das Kriterium setzt sich aus den Ausdrücken e_i zusammen, für die die Bezeichner p_i eingeführt werden. Alle Elemente, bei denen die Auswertung der Ausdrücke e_i dieselbe Wertekombination liefert, werden in eine Partition gruppiert. Wenn für die e_i sich ausschließende Prädikate gewählt werden, stellen diese Prädikate Charakterisierungen der einzelnen Partitionen dar. Dies läßt sich am besten an einem Beispiel veranschaulichen. Durch den folgenden Ausdruck werden die *Mitarbeiter* in Gehaltsklassen aufgeteilt (partitioniert):

select *
from *festangestellte f*
group by *hoch: f.Gehalt > 8000,*
mittel: f.Gehalt < = 8000 and f.Gehalt > = 3000,
niedrig: f.Gehalt < 3000)

Das Ergebnis der group by-Operation ist eine Menge von Strukturen, wobei jede der Strukturen eine der Partitionen repräsentiert. Die Strukturen besitzen Komponenten p_i mit dem jeweiligen Wert von e_i und eine mengenwertige Komponente mit dem vordefinierten Namen *partition*, die die Elemente der jeweiligen Partition enthält. Es ist vom Typ:

Set< **Structure**
< *hoch:* **Boolean**, *mittel:* **Boolean** *niedrig:* **Boolean**
partition: **Bag** < *Festangestellte* > >

Jedes der drei Elemente des Ergebnisses repräsentiert eine der Partitionen und ist durch die Werte der Boole'schen Variablen *hoch*, *mittel* und *niedrig* gekennzeichnet.

Durch Angabe eines entsprechenden Ausdrucks in der select-Klausel ist es auch möglich eine Aggregatfunktionen auf die erzeugten Partitionen anzuwenden. Damit läßt sich z.B. das durchschnittliche Gehalt der Mitarbeiter zu jeder Abteilung berechnen und zusammen mit dem jeweiligen Namen der Abteilung ausgeben.

select *structure(Abteilung avg_Gehalt: avg(***select** *x.Gehalt* **from** *partition x))*
from *festangestellte f*
group by *Abteilung: f.Arbeitgeber.name*

Für das Bespiel ist das Ergebnis vom Typ:

Bag< **Structure** < *Abteilung:* **String**, *avg_Gehalt:* **Integer**> >

Wie in SQL kann auch in OQL die group by-Klausel durch eine having-Klausel erweitert werden, die es erlaubt, einschränkende Prädikate auf den Partitionen zu definieren

5.6.7 Anfragen auf geordnete Kollektionen

Speziell für geordnete Kollektionen bietet OQL Anfragekonstrukte zur Selektion von einzelnen Elementen und Teilkollektionen an:

- Da dem ersten bzw. letzten Elements einer geordneten Kollektion in vielen Situationen, wie z.B. bei der Verwendung einer Liste als Warteschlange eine besondere Bedeutung zukommt, existieren spezielle Operatoren **first** und **last** zur Selektion des ersten bzw. letzten Elements einer geordneten Kollektion. Wenn man davon ausgeht, daß die Projekte zu jeder Abteilung nach Priorität geordnet sind, läßt sich mit der folgenden Anfrage das wichtigste Projekt der Abteilung *a1* ermitteln:

first*(abt1.Projekte)*

- Das i-te Element einer geordneten Kollektion kann durch die Angabe des Index *i* in eckigen Klammern selektiert werden[16]. Obiger Ausdruck läßt sich damit auch wie folgt definieren:

abt1.Projekte[0]

Es ist eine gute Übung sich die Formulierung der entsprechenden Anfrage unter Verwendung der Aggregatfunktion **min** zu überlegen.

- Die positionsabhängige Selektion einer Teilkollektion einer geordneten Kollektion geschieht durch die Angabe des Index (der Position) des ersten und des letzten der auszuwählenden Elemente, eingefaßt in eckige Klammern und getrennt durch einen Doppelpunkt. Analog zum vorigen Beipiel ergibt die folgende Anfrage die *Top Ten*-Liste der bestverdienenden Festangestellten

list*(***select**
  from *festangestellte f*
  order by *f.Gehalt* **desc***) [0:9]*

[16]Es ist zu beachten, daß das erste Element den Index 0 besitzt.

6 Modellierungs- und Sprachvarianten

Ziel: Stelle Syntax grob vor. Biete Orientierung. Zeige sprachliche Vor- und Nachteile auf. Biete Verweise in weiterführende Literatur.

Algebraische Sprachen

s. Tabelle N, 6.1 und Kapitel 6

Relationale Algebra

Objektalgebren. Imperativer Charakter. Bedeutung für interne Repräsentationen (Auswertungspläne)

Kalkülorientierte Sprachen

(Tupelkalkül, Domänenkalkül, Comprehensions)

Rekursive SQL Anfragen

Logikorientierte Sprachen

Prolog, Datalog, LDL. Kurz am Standardbeispiel skizzieren und sonst nur Verweise

Funktionale Sprachen

Sprachen semantischer Datenmodelle

Skripte, Constraints

Vergleich der sprachlichen Mächtigkeit von Datenbanksprachen

Hierarchie der Datenbanksprachen. (Abiteboul, Hull,)

7 Modalitäten des Datenbankzugriffs und ihre Sprachschnittstellen

7.1 Werkzeuge zum interaktiven Datenbankzugriff

Reportgeneratoren, QBE, (s.a. UNIX Tools). Benutzen standardisierte Dienstzugangsschnittstellen.

7.2 Statischer Datenbankzugriff aus monomorphen Programmiersprachen

`Embedded SQL`

`Architekturabbildung. Beispiel mit Code`
Probleme: Debugger
Vorteil: Integrierte Sicht

7.2.1 Anfrage- und Cursordefinition in Embedded SQL

`Updates, Stabilität (S. 265)`

7.2.2 Typisierung und Parametrisierung in Embedded SQL

Syntax für Gastsprachenvariable (host variable).
Abbildung: Bindung?
Wertparameterübergabe in Ausdrücken, Bsp. INSERT ... VALUES
Variablenparameterübergabe in SELECT INTO
Indikatorvariablen:
Lesender Zugriff auf Nullwerte
Schreiben von Nullwerten
Gastsprachentypkompatibilität. Informationsverlust beim Lesen INDICATOR (Strings einfach, andere Typen komplexer)
Korrespondenztabelle C?

7.2.3 Programmierte SQL Fehlerbehandlung

`Beachte:` Auch Datendefinitionen sind Anweisungen (Änderungen auf dem Katalog). Integritätsbedingungen können verletzt werden (doppelte Namen) oder Syntax falsch oder Zugriffsrechte falsch.
Statusparameter: SQLCODE, SQLSTATE

7.3 Statischer Datenbankzugriff aus polymorphen Programmiersprachen

(ODMG-93 C++ Bindings)

7.3.1 Überblick über die ODMG-93 C++ Sprachanbindung

ODL: Minimal erweiterte Syntax für Klassendefinitionen (*inverse)* , *Modifikation der Semantik (?), vordefinierte Klassenbibliothek*
OML: Keine Spracherweiterung, jedoch objektabhängig neue Semantik für Operatoren.
Physikalische Pragmas.

Abbildung 5-1 aus ODMG Report (?)

Persistenzfähige Klassen. Ref<T>. Indirektion über handle. Dereferenzieren von gelöschten Objekten erzeugt eine Ausnahme.

Referenzen auf transiente Werte

Queries & Transaktionen arbeiten nur konsistent auf persistenten Objekten

7.3.2 Korrespondenz zwischen ODMG Datenmodell und C++-Typsystem

Literale auf Literale. Booleans abgebildet auf int.

Strings auf spezielle Klasse String. Analog Date, Time, ...

Strukturen auf Structs

Genau eine C++ Klasse pro Typspezifikation

Parametrisierte Kollektionen abgebildet auf Template Classes

Attribute abgebildet auf data members. Kein Überschreben der get_value und set_value Methoden möglich.

Beziehungen abgebildet auf Objektreferenzen oder Kollektionen von Objektreferenzen als Objektattribute.

Referentielle Integrität bidirektionaler Beziehungen wird durch Überschreiben der set_value Methoden erreicht..

Keine extents und keys

Umgekehrt

Unions, bit fields und references nur eingeschränkt erlaubt.

7.3.3 Objektmanipulation in C++ Syntax

new überladen. Spezifikation des Speicherortes (flüchtig, in Datenbank, nahe an anderem Objekt). e weitere Argumente

Explizite Operationen ("mark modified", "delete"), um Kommunikation mit Speicherverwaltung des DBMS durchzuführen.

7.3.4 Lose Einbettung von OQL-Anfragen in C++

Eng gekoppelt. Grammatik erweitert. Typregeln erweitert. Optimierbar zum Übersetzungszeitpunkt.

Lose gekoppelt: Anfrageausdrücke als Strings in schwach typisierte Funktion eingebunden. Parsing, Übersetzung, Optimierung zur Laufzeit. Generik über Stringmanipulationen.

a) query Methode für Kollektionen

b) oql Funktion mit variabler Parameteranzahl.

7.4 Verbesserte Datenbankanbindung durch sprachliche Erweiterbarkeit

Ausblick auf ODMG-93 Bindungen

7.5 Dynamischer Datenbankzugriff aus Programmiersprachen

(ODBC API)

`Dynamic SQL, ODBC. Beispiel. Literaturverweis (??).`

`Typisierungsproblematik.`

8 Integrierte Datenbanksprachen

Integrierte Umgebungen zur Entwicklung vollständiger Informationssysteme. Vgl. mit Bild xx. Überwindung inkompatibler Annahmen klassischer Compilertechnologie.

KURZ. Beschränkung auf Verweise

8.1 Sprachen der Vierten Generation

Windows 4GL, Informix 4GL, Power Builder

8.2 Datenbankprogrammiersprachen

GemStone, O2C, ObjectStore?

8.3 Persistente Programmiersprachen

Napier-88, Tycoon

9 Transaktionen und ihre Realisierung

Transaktionen als Kooperationsmodell, Eigenschaften von Transaktionen Isolation von Transaktionen:

- Terminologie und Formalisierung
- Klassifikation der Probleme durch Verletzung der Isolation
- Isolation durch Sperrprotokolle
- Diskussion: Vor- und Nachteile niedriger Isolationsgrade

Datenmodelle und Kooperationsmodelle
Datenmodelle:

- Datenbanksprachen unterscheiden sich hauptsächlich in ihrer Mächtigkeit bei der Beschreibung der Struktur und des Zustandes der Datenbanken.
- Ziel: (vgl. Kapitel 2-5): Problemangepaßte Datenstrukturen und deklarative Anfragen für Nutzer der Datenbank

Kooperationsmodelle:

- Weitgehend orthogonal zur Wahl des Datenmodells ist das Kooperationsmodell.
- Ziel: Problemangepaßte Mechanismen zur Synchronisation und Fehlererholung von nebenläufigen Aktivitäten auf dem gemeinsamen Datenbankzustand
- Transaktionen sind das allgemein in Theorie und Praxis akzeptierte Kooperationsmodell.
- Andere Ansätze: Private und öffentliche Arbeitsumgebungen (workspaces bei CASE), Message Passing (nebenläufige Programmierung), Replizierte Datenbanken mit manueller Konfliktauflösung (Groupware, sehr gut geeignet für autonome mobile Aktoren).

Transaktionen für kooperierende Aktivitäten

- Kooperation mehrerer Geschäftsprozesse über Seiteneffekte auf gemeinsamen Daten
- Idee: Der gemeinsame Datenbankzustand ist ein Abbild des aktuellen Zustands der gemeinsamen Realität der nebenläufigen Geschäftsprozesse.

  Beispiele: Buchhaltung, Lagerwirtschaft, Logistik, ...
- Das Datenbanksystem sorgt implizit für Synchronisation und Fehlererholung der Zustandsänderungen, die durch die Geschäftsprozesse ausgelöst werden.
- Der Programmierer ergreift keine expliziten Maßnahmen zur Synchronisation und Fehlererholung.

Grundsätzlich sind Transaktionen auch als Bausteine für komplexere, anwendungsbezogene Kooperationsmodelle geeignet (s. Beispiel "betriebswirtschafltiche Transaktionen" in SAP R/3).

Definition: Transaktion Eine Transaktion ist eine Folge von ßusammengehörigen" Operationen auf dem Systemzustand, die die folgenden Kriterien erfüllt (ACID-Eigenschaften):

- Atomarität: Die Änderungen des Systemzustands sind atomar, d.h. entweder finden alle oder keine statt. Diese Änderungen umfassen Datenbankzustandsänderungen, Nachrichten und Aktionen auf externen Geräten.
- Konsistenz: Eine Transaktion stellt eine konsistente Transformation des Datenbankzustands dar, wenn die Gesamtheit ihrer Aktionen keine der Integritätsbedingungen, die für den Datenbankzustand definiert sind, verletzt.
- Isolation: Obwohl Transaktionen nebenläufig ausgeführt werden, erscheint es für jede Transaktion T, als ob jede einzelne der nebenläufigen Transaktionen entweder als ganzes vor oder als ganzes nach T ausgeführt wird. (Serialisierbarkeit)
- Dauerhaftigkeit: Wenn eine Transaktion erfolgreich abgeschlossen ist, überleben ihre Änderungen des Datenbankzustands auch alle späteren Systemausfälle.

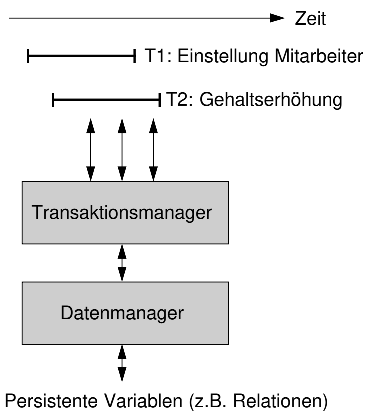

Abbildung 73: Beispiel: Projektdatenbank

Einstellung Mitarbeiter:

begin work
  SQL-Anfragen
  SQL-Änderungsoperationen
end work

9.1 Atomarität

Vorteile:

- Klare Fehlersemantik in komplexen Systemen.

- Schutz des gesamten Datenbestands, der meist wesentlich wertvoller als die einzelnen Zustandsänderungen einer Transaktion ist.
- Möglichkeit zum Rücksetzen von Anwendungstransaktionen im Falle von Verklemmungen, Überlastsituationen, ...

Probleme:

- Rücksetzbarkeit von realen" Operationen (Geld auszahlen, Rakete abschießen, ...)
- Programmgesteuerte Reaktion auf ROLLBACK
  - Restaurieren des Bildschirminhalts, des Zustands von Dateien, ...
  - Benachrichtigung des Benutzers, ...

9.2 Konsistenz

- In der Praxis ein rein syntaktisches Kriterium: Jede sequentielle Anweisungsfolge zwischen **begin work** und **commit work** erzeugt im Einbenutzerbetrieb aus einem konsistenten Datenbankzustand einen konsistenten Datenbankzustand.
- Alternativ: Test statischer semantischer Konsistenzbedingungen durch den Datenmanager (Schlüsselintegrität, ...)
- Außerdem: Transaktionen wahren dynamische Konsistenzbedingungen (z.B. "Gehälter fallen nie").
- Das Rücksetzen von Transaktionen führt immer zu einem konsistenten Zustand.

Probleme:

- Wer sichert die Konsistenz zwischen Datenbankzustand und Realität ("Korrektheit von Transaktionen")?

9.3 Isolation

Durch die ungeschützte parallele Ausführung der Transaktionensanweisungen können Inkonsistenzen entstehen, auch wenn die einzelnen Transaktionen konsistent sind.

Idee:

- Transaktionen enthalten keine expliziten Synchronisationsbefehle (Zugriff auf Semaphore, Mutexe, Sperren, Nachrichtenschlangen, ...)
- Es existieren keine relevanten Reihenfolgeabhängigkeiten zwischen nebenläufigen Transaktionen (s. Beispiel: Einstellung / Gehaltserhöhung), d.h. jede serielle Ausführungsreihenfolge ist ërlaubt".

Überblick:

- Modellbildung (Transaktionsabhängigkeiten)
- Klassifikation der Probleme aufgrund fehlender Isolation (Anomalien)
- Formalisierung des Begriffs der Isolation (Serialisierbarkeit)
- Techniken zur Isolation von Transaktionen (Sperren)

9.4 Dauerhaftigkeit und Fehlererholung

Rücksetzen der Transaktion (**rollback**, **undo**), falls

- Systemfehler oder Integritätsverletzung während der Transaktionsausführung oder während **commit work**
- rollback-Befehl

Wiederherstellen der Transaktionseffekte (**redo**), falls

- Systemfehler nach erfolgreichem **commit work** basierend auf Logdateien, Archivkopien, dem Zustand von Spiegelsystemen, ...

Betrifft eine Transaktion mehr als einen Datenmanager, so muß am Transaktionsende ein Zwei-Phasen-Commit-Protokoll stattfinden:

Phase 1: Feststellen der Commitfähigkeit: Der Transaktionsmanager schickt allen Datenmanagern eine *prepare to commit* Nachricht. Nur wenn alle Datenmanager mit "JA" antworten, wird die Commitfähigkeit festgestellt.

Phase 2: Propagieren der getroffenen Entscheidung: Der Transaktionsmanager schickt allen Datenmanagern eine *commit* bzw. eine *abort* Nachricht.

9.5 Transaktionen

Transaktionen sind Folgen von Einzelaktionen, die zwischen begin und **commit** oder **rollback** stattfinden.

T1	begin	
	slock	A
	xlock	B
	read	A
	write	B
	commit	

T2	begin	
	slock	A
	read	A
	xlock	B
	write	B
	rollback	

Abbildung 74: Beipiel zweier Transaktionen

Symbolische Repräsentation einer Transaktion: $<< t, a_i, o_k > | i = 1, ..., n; k = 1, ..., m >$
Der i-te Schritt der Transaktion t führt die Einzelaktion ai auf dem Objekt ok durch.
Isolation von Transaktionen
Ziel:

- Datenbankhistorien: Beschreibung der nebenläufigen ("verzahnten") Ausführung von Transaktionen.

Hilfsbegriffe:

- Datenbankzustand
- Operation auf Objekt (Komponente des Datenbankzustands)

Generische Aktion	Beschreibung
begin	Legt den Anfang einer Transaktion fest.
commit	Beendet eine erfolgreich ausgeführte Transaktion.
rollback	Auf Objekten innerhalb einer Transaktion durchgeführte Manipulationen werden rückgängig gemacht, anschließend wird die Transaktion beendet.

Tabelle 10:

Aktionen auf Objekten	Beschreibung
read A	Liefert den Wert des benannten Objekts A.
write B	Weist dem benannten Objekt B einen Wert zu und erzeugt neue Objektversion.
xlock B	Setzt X-Sperre (exklusiv) für Manipulation des Objekts B.
slock A	Setzt S-Sperre (shared) für Lesen des Objekts A.
unlock B	Gibt Sperre für Objekt B frei.

Tabelle 11:

- Abhängigkeiten zwischen Operationen
- Transaktion (mit impliziter und expliziter Fehlererholung)

Parallel wird das Konzept der Sperren auf Objekten eingeführt.
Anschließend wird gezeigt, daß durch geeignete Sperrprotokolle (= Sperralgorithmen) nur ërwünschte" Datenbankhistorien zugelassen werden.
Einzelaktionen
Generische Aktionen:
Sperrkompatibilitätsmatrix
Ziel von Sperrprotokollen:

- Erzeugung von serialisierbaren Historien
- Dazu müssen Lese- und Schreibaktionen in einer bestimmten Systematik verwendet werden (Zweiphasen-Sperrprotokoll).

- Ist eine Sperranforderung gewährt worden, wird die Exekution fortgesetzt.
- Ist eine Sperranforderung nicht gewährt worden, wird die Exekution vorerst angehalten.

Notation

- Folgen S, T
- Konkatenation S —— T

Sperranforderung	**Sperrmodus**	**Kompatibilität**
shared	shared	kompatibel
exklusiv	shared	Konflikt
shared	exklusiv	Konflikt
exklusiv	exklusiv	Konflikt

Tabelle 12:

- i-tes Element einer Folge S
- Teilfolge S' von S
- Geordnetes Paar als zweielementige Folge
- Zustand Z eines Systems

9.6 Transaktionsvereinfachung

Transaktionen können dadurch vereinfacht werden, daß die Aktionen **begin**, **commit** und **rollback** durch die Aktionen **read**, **write**, **lock** und **unlock** substituiert sind.

Transaktionsvereinfachung:

- Verzichte auf **begin**.
- Ersetze **commit** durch: < **unlock A** — falls **slock A** oder **xlock A** in T vorkommt >
- Ersetze **rollback** durch: < **write A** — falls **write A** in T vorkommt > —— < **unlock A** — falls **slock A** oder **xlock A** in T vorkommt >
- Beispiel: T1' und T2' auf der nächsten Folie

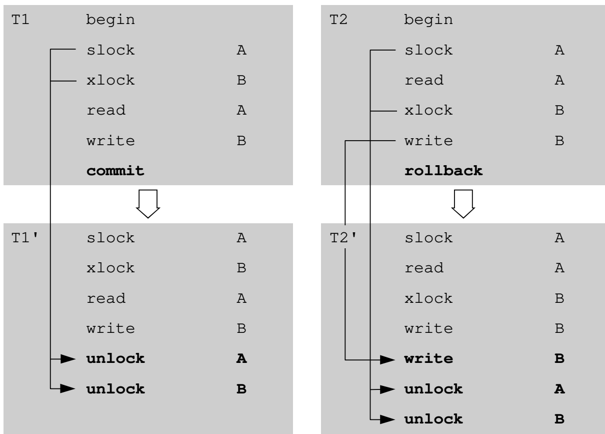

Abbildung 75: Beipiel einer Transaktionsvereinfachung

9.7 Zweiphasige und wohlgeformte Transaktionen

Wohlgeformte Transaktion:

- Jede lock Aktion korrespondiert in systematischer Weise mit den Lese- und Schreibaktionen der Transaktion.
  - Vor Leseaktionen erfolgt ein **slock**.
  - Vor Schreibaktionen erfolgt ein **xlock**.
- Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
- Sperren anderer Transaktionen müssen beachtet werden (Sperrkompatibilitätsmatrix).
- Jede **read**, **write** und **unlock** Aktion korrespondiert mit einer **lock** Aktion.
- Auf jede lock Aktion erfolgt eine korrespondierende unlock Aktion.
- Bei EOT muß eine Transaktion spätestens alle ihre Sperren zurückgeben.

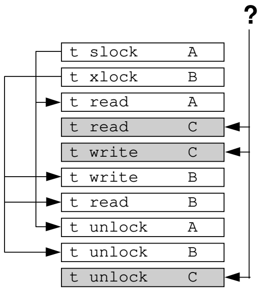

Abbildung 76: Beipiel einer nicht wohlgeformten Transaktion

Zweiphasen-Sperrprotokoll:

- Alle lock Aktionen gehen ihren unlock Aktionen voraus.
- In der Wachstumsphase $T[1], \ldots, T[j]$ der Transaktion T werden Sperren angefordert (lock).
- In der Schrumpfungsphase $T[j+1], \ldots, T[n]$ werden die Sperren zurückgegeben (unlock).
- Wohlgeformte, zweiphasige Transaktionen genügen dem Zweiphasen-Sperrprotokoll.

Striktes Zweiphasen-Sperrprotokoll:

- Spezielle Zusatzanforderungen an den Verlauf der Phasen (s. nächste Folie) lösen Probleme, die über die Anforderungen der Isolation von Transaktionen hinausgehen (Dauerhaftigkeit, Verklemmung, ...).

Historien

- Ziel: Modellierung der parallelen Ausführung von Transaktionen

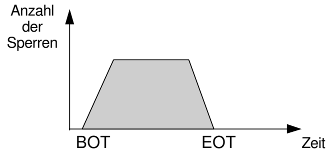

Abbildung 77: Zweiphasen-Sperrprotokoll

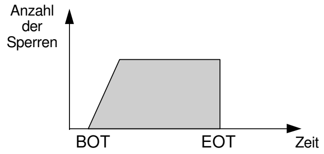

Abbildung 78: Striktes Zweiphasen-Sperrprotokoll

- Folge von Aktionen bestehend aus einer Menge von Transaktionen unter Beibehaltung der Teilfolgen innerhalb der einzelnen Transaktionen
- Notation: $H =<< t, a_i, o_k > | i = 1, ..., n; k = 1, ..., m >$ Ein Schritt einer Historie entspricht einer Aktion a_i der Transaktion t auf dem Objekt o_k.
- Serielle Historie: Die Transaktionen werden vollständig nacheinander ausgeführt. Es entstehen keine Inkonsistenzen aufgrund von Nebenläufigkeit.
- Serialisierbare Historie: Eine Historie, die äquivalent zu einer seriellen Historie ist. (Äquivalenz noch zu definieren!)
- Sperr-Bedingungen schränken die Menge der erlaubten Historien ein. Zum Beispiel kann eine Exklusivsperre auf ein Objekt für eine Aktion nur dann genehmigt werden, wenn für dieses Objekt nicht schon eine Sperre im Rahmen einer anderen Transaktion besteht (s. Kompatibilität zwischen Anforderungsmodus und Sperrmodus auf Folie Sperrkompatibilitätsmatrix).
- Legale Historie: Sperren werden berücksichtigt (s. Sperrkompatibilitätsmatrix).

Abhängigkeiten zwischen Transaktionen in Historien

- Ziel: Definition des Datenflusses zwischen Transaktionen und Vergleich zweier Historien miteinander
- Eine Transaktion T hängt von einer Transaktion T' innerhalb einer Historie H ab, falls

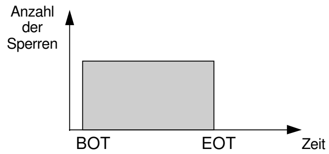

Abbildung 79: Striktes Zweiphasen-Sperrprotokoll mit preclaiming

			T2	slock	A
T1	slock	A	T2	read	A
T1	xlock	B	T2	xlock	B
T1	read	A	T2	write	B
T1	write	B	T2	write	B
T1	unlock	A	T2	unlock	A
T1	unlock	B	T2	unlock	B

Abbildung 80: Beispiel einer legalen und seriellen Historie

			T2	unlock	A
T2	slock	A	T2	unlock	B
T1	slock	A	T1	xlock	B
T2	read	A	T1	read	A
T2	xlock	B	T1	write	B
T2	write	B	T1	unlock	A
T2	write	B	T1	unlock	B

Abbildung 81: Beispiel einer legalen und nicht seriellen Historie

  - T' Objekte beschrieben hat, die anschließend von T gelesen oder beschrieben werden oder
  - T' Objekte gelesen hat, die anschließend von T beschrieben werden.

- Äquivalente Historie: Zwei Historien besitzen die selben Abhängigkeiten, d.h. ihre Transaktionen greifen lesend und schreibend in beiden Historien in der gleichen Reihenfolge auf die gleichen Objekte zu.

- Abhängigkeitsgraphen beschreiben die Abhängigkeiten zwischen Transaktionen in Historien (s. nächste Folie).

- Schmutzige Daten sind Daten, auf die Transaktion T zugreifen kann, die aber potentiell von nebenläufigen Transaktionen T' geändert werden können.

9.8 Abhängigkeitsgraphen

Kritische Elementarsituationen

			T2	write	B
T1	slock	A	T2	unlock	A
T1	**xlock**	**B**	T2	unlock	B
T2	slock	A	T1	read	A
T2	read	A	T1	write	B
T2	**xlock**	**B**	T1	unlock	A
T2	write	B	T1	unlock	B

Abbildung 82: Beispiel einer nicht legale und nicht seriellen Historie

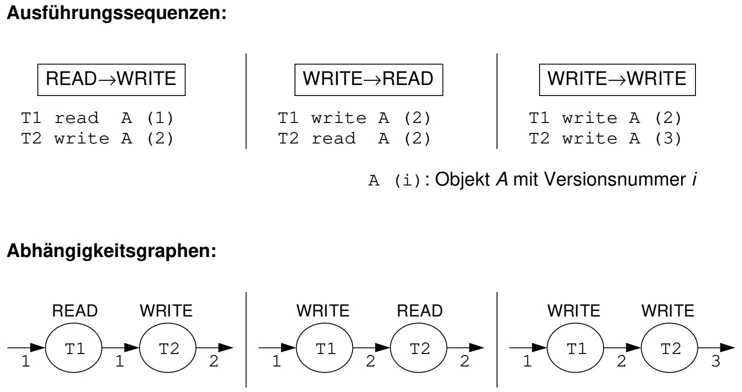

Abbildung 83: Ausführungssequenzen

Abgehende Leseabhängigkeiten (READREAD) werden nicht berücksichtigt.

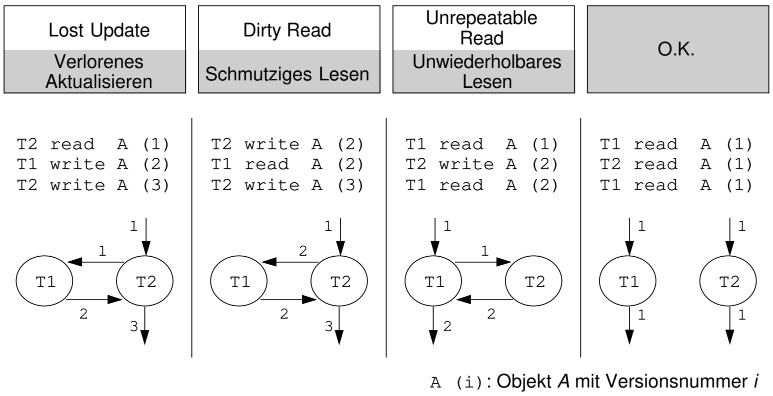

Abbildung 84: Anormale T1/T2 -Abhängigkeiten (Zyklen)

9.9 Abhängigkeitszyklen

Für die linke Seite (s. vorige Folie) gilt:

- $VOR(T4) = \{T1, T3\}$ und $NACH(T4) = \{T6\}$. T2 und T5 sind deshalb weder vor noch nach T4 und können deshalb nebenläufig zu T4 ausgeführt werden.

  T2, T4 und T5 können in beliebiger Reihenfolge und beliebig überlappend ausgeführt werden.

- Die Relation <<< hat keine Zyklen und definiert eine partielle Ordnung der Transaktionen.

Für die rechte Seite (s. vorige Folie) gilt:

- $VOR(T4) = \{T1, T3, T4, T5, T6\}$

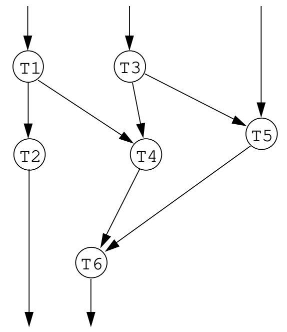

Abbildung 85: Abhängigkeitsgraph mit möglicher serieller Historie (T1, T2, T3, T4, T5, T6)

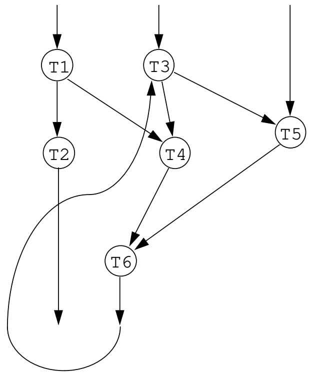

Abbildung 86: Abhängigkeitsgraph ohne mögliche serielle Historie

- Die Relation <<< hat Zyklen (keine Serialisierbarkeit)

Folgende Aussagen sind äquivalent (ohne Beweis):

- Wenn für zwei Transaktionen T und T' gilt, daß Teile von T vor T' und Teile von T' vor T ausgeführt werden, dann liegt eine Nebenläufigkeitsanomalie vor, anderenfalls gibt es keine Nebenläufigkeitsanomalie.
- Hat der Abhängigkeitsgraph Zyklen, gibt es keine äquivalente serialisierbare Historie; ist er zyklenfrei, so existiert mindestens eine serialisierbare Historie.
- Eine serialisierbare Historie entsteht (genau) dann, wenn alle nebenläufigen Transaktionen dem Zweiphasen-Sperrprotokoll genügen.

9.10 Niedrige Grade der Isolation

- Isolation wird von den meisten Systemen aus Performanzgründen nicht in vollem Umfang geboten. Es werden nur WRITE→WRITE und WRITE→READ Abhängigkeiten berücksichtigt (*cursor stability*, *repeatable reads*). READ→WRITE Abhängigkeiten werden meist ignoriert.
- Der ANSI SQL-Standard schreibt volle Isolation vor (Grad 3).

Grad 0: Eine 0 Transaktion überschreibt keine schmutzigen Daten einer anderen Transaktion, wenn diese mindestens Grad 1 hat.

Grad 1: 1 Transaktionen haben keine verlorengegangenen Änderungen.

Grad 2: 2 Transaktionen haben keine verlorengegangenen Änderungen und keine schmutzige Leseoperationen.

Grad 3: 3 Transaktionen haben keine verlorengegangenen Änderungen und nur wiederholbare Leseoperationen.

Sperrprotokolle der Isolationsgrade haben die folgenden Eigenschaften:

Grad 0: Wohlgeformt in bezug auf Schreiboperationen (anarchy)

Grad 1: Zweiphasig in bezug auf exklusive Sperren und wohlgeformt in bezug auf Schreiboperationen (browse)

Grad 2: Zweiphasig in bezug auf exklusive Sperren und wohlgeformt (cursor stability)

Grad 3: Zweiphasig und wohlgeformt (isolated, serializable, repeatable reads)

Die Isolationsgrade entsprechen den Abhängigkeiten:
1 Ö WRITE→WRITE 2 Ö WRITE→WRITE, WRITE→READ 3 Ö WRITE→WRITE, WRITE→READ, READ→WRITE
Isolationstheorem:
Erfüllt eine Transaktion das 0, 1, 2 oder 3 Sperrprotokoll, so erreicht sie Isolationsgrad 1, 2 oder 3 in jeder legalen Historie, wenn alle anderen Transaktioen mindestens vom Grad 1 sind.

- In der Praxis bedeutet dies: Der Grad der eigenen Transaktion kann unabhängig von anderen Transaktionen gewählt werden.
- Ausnahme: Eingabedaten von 1 Transaktionen sind möglicherweise schmutzig. Werden diese Daten zum Aktualisieren der Datenbank verwendet, können andere Transaktionen inkonsistente Daten bekommen. Das Theorem geht deshalb davon aus, daß 1 Transaktionen "wissen, was sie tun".
- Konklusion: Auch wenn eine Transaktion schmutzig Daten liest, nimmt man an, daß sie die Datenbank in einen konsistenten Zustand transformiert. Viele Systeme lassen 1 Isolationverhalten nur für Lese-Transaktionen zu.

9.11 Grade der Isolation in SQL

- Transaktionen mit niedrigen Isolationsgraden benötigen weniger Sperren, halten diese für eine kürzere Zeit und sorgen somit für bessere Performanz.
- 2 Isolation erlaubt es Transaktionen, die gesamte Datenbank abzusuchen und dabei wenig Sperren zu halten und somit andere Transaktionen nicht zu stören.
- Volle Isolation kann nur bei Grad 3 erreicht werden.
- SQL stellt die folgenden Optionen bereit: **set transaction isolation level**

Kommando	**Isolationsgrad**
read uncommitted	Grad 1
read committed	Grad 2
repeatable read	Grad 2,9999
serializable	Grad 3

Tabelle 13: Isolationsgrade in SQL

9.12 Cursor stability

- In SQL wird üblicherweise eine Islolation implementiert, die mehr Konsistenz als Grad 2 bietet (cursor stability).

```
exec sql declare cursor c for
    set gehalt
    from mitarbeiter
    where no = nr;
exec sql open c;
exec sql fetch c into :gehalt;
gehalt = gehalt * 1.5;
exec sql update mitarbeiter
    set gehalt = :gehalt;
    where no = nr;
exec sql close c;
```

Abbildung 87: Cursor stability

```
exec sql select gehalt
    into :gehalt
    from mitarbeiter
    where no = nr;
gehalt = gehalt * 1.5
exec sql update mitarbeiter
    set gehalt = :gehalt
    where no = nr;
```

Abbildung 88: Verlorengegangene Änderungen mit 2 Isolation

Zusammenfassung: Niedrige Isolationsgrade
Vorteile:

- Daten sind nur kurze Zeit gesperrt.
- Weniger Daten werden gesperrt.
- Potentiell höhere Nebenläufigkeit

Nachteile:

```
exec sql update mitarbeiter
    set gehalt = gehalt * 1.5
    where no = nr;
```

Abbildung 89: Searched Update für 1 oder 2

	Grad 0	**Grad 1**	**Grad 2**	**Grad 3**
Name	Chaos	Browse	Cursor Stability	Repeatable Read (Serialisierbar
Schutz	Stört keine Trasaktionen höheren Grades	0 und keine verlorengegangenen Änderungen	Keine verlorengegangenen Änderungen, keine schmutzigen Leseoperationen	Keine verlorengegangenen Änderungen, keine schmutzigen Leseoperationen, wiederholbare Leseoperationen

Tabelle 14:

- Möglicherweise inkonsistente Eingabe- und Ausgabedaten
- Eine **rollback** Operation muß für ihre Aktionen Sperren setzen.
- 0 Transaktionen unterstützen kein ROLLBACK.
- Für reine Lesetransaktionen gilt $0 = 1$.

Die Eigenschaften niedriger Isolationsgrade sind in den folgenden Tabellen nochmals gegenübergestellt.