

Anmerkungen zu Übungsblatt 9 / 10

Netzwerk-Datenbanken / Objektorientierte Datenbanken

Jürgen Kalinski, Jens E. Wolff
Universität Bonn, Institut für Informatik III
Römerstr. 164, 53117 Bonn
cully@cs.uni-bonn.de

31. Januar 1999

Übungsblatt 9, Aufgabe 1

Schema

Im Netzwerkmodell werden Entity-Typen durch *Record-Typen* repräsentiert. Jeder Record-Typ umfaßt eine Menge von Attributen. Ein Record eines Typs ist ein Datensatz mit Werten für jedes dieser Attribute.

Set-Typen modellieren binäre N:1-Beziehungen zwischen Records. Bei der Besprechung der Aufgabe 3 des Übungsblatts 1 ist bereits dargestellt worden, wie mehrstellige Beziehungen und N:M-Beziehungen mittels binärer N:1-Beziehungen „simuliert“ werden können. Betrachten wir als Beispiel die N:M-Beziehung zwischen Personen und Filmen (siehe Abb. 1). Aus dem Beziehungstyp *mitwirken* wird der Entity-Typ *Mitwirkung*. Jedes Entity dieses Typs stellt *eine* Verknüpfung zwischen einer Person (z.B. Steven Spielberg) und einem Film (z.B. „E.T.“) dar.

Die Definition eines Netzwerk-Schemas gibt für jeden Record-Typ dessen Felder und ihre Typen an:

```
RECORD NAME IS Person
  Name          TYPE CHARACTER 40
  Vorname       TYPE CHARACTER 40
```

In der Beziehung bzw. im Set-Typ *PM* wird jeder Person eine Menge von Mitwirkungsdatensätzen zugeordnet. Umgekehrt ist jeder Mitwirkungsdatensatz mit genau einer Person (via *PM*) und genau einem Film (via *FM*) verknüpft. Die Personen bzw. Filme werden als *Owner* der Set-Typen bezeichnet, die Mitwirkungsdatensätze als *Member*:

```
SET NAME IS PM
  OWNER IS Person
  MEMBER IS Mitwirkung
```

In der DBTG Data Definition Language können zudem Angaben zur Speicherung und Verwaltung der Records und Sets gemacht werden. So spezifiziert

```
LOCATION MODE IS CALC USING Name
```

daß der Personennamen mittels einer Hash-Funktion den Speicherort eines Personensatzes festlegt. Mit

```
LOCATION MODE IS VIA PM SET
```

werden die Members des *PM*-Sets mit dem Owner physikalisch geballt gespeichert. Auf diese Weise können schnelle Zugriffe vom Owner auf seine Member unterstützt werden.¹

¹Man vergleiche dies mit dem „verzahnten“ Abspeichern in Clustern bei relationalen Datenbanken (siehe [2]).

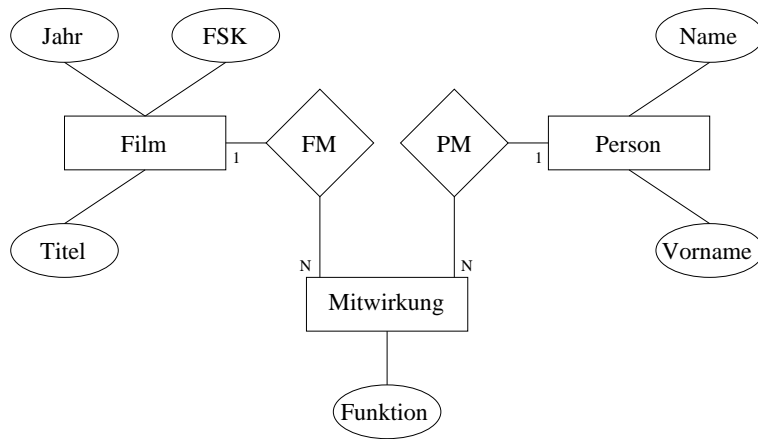
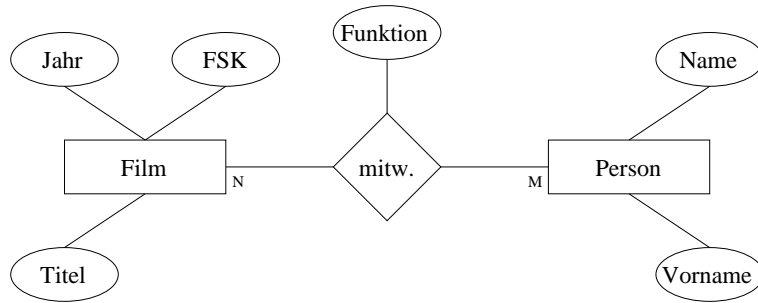


Abbildung 1: Rückführung von N:M- auf N:1-Beziehungen

Film	ID	Titel	Jahr	FSK	Person	ID	Name	Vorname
	29	Indiana Jones III	1989	12		19	Winger	Debra
	30	E.T.	1981	0		32	Spielberg	Steven
	31	Forget Paris	1995	6				

Mitwirkung	Film	Person	Funktion
	29	32	Regisseur
	30	19	Stimme von E.T.
	30	32	Regisseur
	31	19	Schauspielerin

Abbildung 2: Ausprägung des relationalen Schemas

Ausprägung

Bei der Umsetzung des ER-Schemas in ein relationales Schema wird man üblicherweise künstliche Schlüsselattribute einführen. Zwar stellt die Kombination aus Name und Vorname einen Schlüssel der Relation *Person* dar. Aus speicherökonomischen Gründen wird man jedoch für Fremdschlüsselbeziehungen wie in der Relation *Mitwirkung* typischerweise nicht diesen Schlüssel nutzen, sondern den Personen und Filmen eine *ID* zuweisen (siehe Abb. 2).

Man beachte, daß solche „Surrogate“ zur Objektidentifikation in Netzwerkdatenbanken nicht benötigt werden. Es gibt genau einen Record, der die Person Steven Spielberg repräsentiert. Verknüpfungen zwischen diesem und anderen Datensätzen werden direkt über Zeiger realisiert (siehe Abb. 3). In der relationalen Datenbank erfolgen Verknüpfungen zwischen „Objekten“, d.h. Tupeln über die *ID*-Werte.

Anfragen

Anfragen werden im Netzwerk-Modell in der Form gestellt, daß man mit in einer Wirtssprache eingebetteten Kommandos durch das Netzwerk „navigiert“. Wenn wir beispielsweise an allen Regisseuren interessiert sind, mit denen Debra Winger gearbeitet hat, sieht der Ablauf grob skizziert wie folgt aus:

1. Zunächst wird der Winger'sche Personensatz aufgesucht. Im folgenden werden all dessen *PM*-Members durchlaufen. Für jeden Member wird dessen *FM*-Owner ermittelt. Damit finden wir alle Filme, an denen Debra Winger mitgewirkt hat.
2. Zu jedem Film wird nun dessen Regisseur ermittelt. Dazu werden zu jedem Film alle *FM*-Member durchlaufen. Falls ein Member im Funktionsfeld den Wert *Regisseur* trägt, wird dessen *PM*-Owner aufgesucht. Sein Name wird sodann ausgegeben.

Zu jedem Record-Typ gibt es in der *User Work Area* eine Schablone, die dem Datenaustausch zwischen Datenbank und Wirtssprache dient. Zunächst wird die Schablone zu *Person* mit dem gegebenen Namen initialisiert. Der Personen-Datensatz kann bei `LOCATION MODE IS CALC USING Name` mittels Hashing aufgesucht werden:

```
( 1)   Person.Name = 'Winger'
( 2)   FIND Person RECORD BY CALC-KEY
( 3)   FIND FIRST Mitwirkung RECORD WITHIN PM SET
( 4)   WHILE found DO
        ... <siehe unten> ...
(17)   FIND NEXT Mitwirkung RECORD WITHIN PM SET
(18)   OD
```

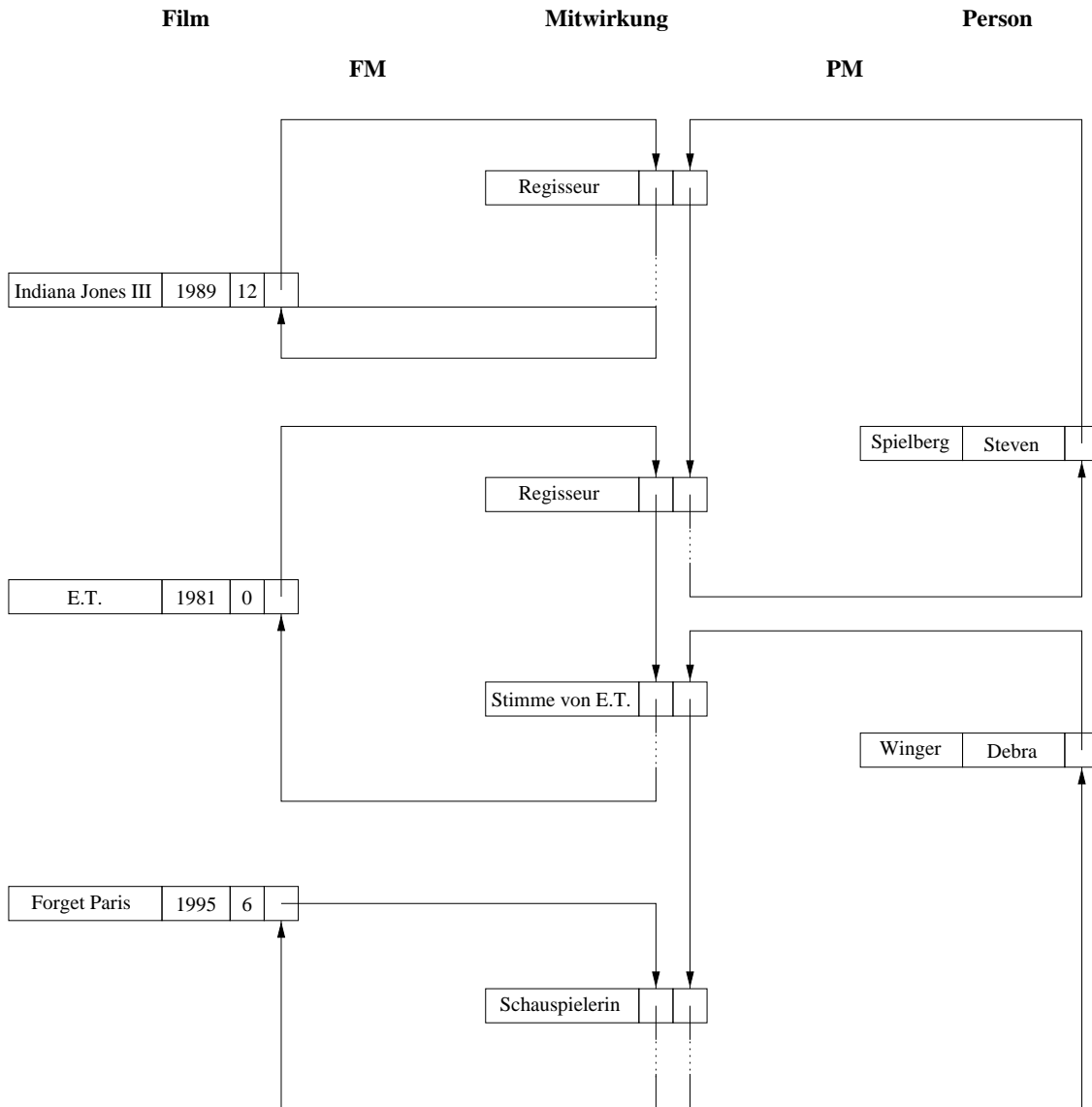


Abbildung 3: Ausprägung des Netzwerk-Schemas

Um beispielsweise eine `FIND NEXT`-Anweisung ausführen zu können, verwaltet eine Netzwerk-Datenbank zu jedem Record-Typ und jedem Set-Typ einen *Currency Pointer*. Der *Current of Record Type* $crt(Person)$ beispielsweise verweist auf denjenigen Personendatensatz, auf den zuletzt zugegriffen wurde. Mit

```
FIND Person RECORD BY CALC-KEY
```

wird der *Current of Record Type* $crt(Person)$ auf den Winger'schen Datensatz gesetzt. Der *Current of Set Type* $cst(PM)$ verweist auf den Personen- oder Mitwirkungsdatensatz, auf den zuletzt zugegriffen wurde. Nach der Ausführung von

```
FIND FIRST Mitwirkung RECORD WITHIN PM SET
```

zeigt $cst(PM)$ auf den ersten Mitwirkungssatz, der mit Debra Winger im *PM*-Set assoziiert ist.

Die restlichen Anweisungen lauten in Analogie zu den bereits diskutierten:

```
( 5)          ... <siehe unten> ...
( 6)          FIND OWNER WITHIN FM SET
( 7)          FIND FIRST Mitwirkung RECORD WITHIN FM SET
( 8)          WHILE found DO
( 9)              GET Mitwirkung.Funktion
(10)              IF Mitwirkung.Funktion='Regisseur' THEN
(11)                  FIND OWNER WITHIN PM SET
(12)                  GET Person.Name
(13)                  PRINT Person.Name
(14)              FI
(15)          OD
(16)          ... <siehe unten> ...
```

Fatal ist nur, daß durch

```
(11)          FIND OWNER WITHIN PM SET
```

der *Currency Pointer* $cst(PM)$ auf den Regisseur von „E.T.“ gesetzt wird. Infolgedessen wird die `FIND NEXT`-Anweisung in Zeile (17) nicht mehr den zweiten Mitwirkungsdatensatz im *PM*-Set von Debra Winger lesen!

Ein möglicher Work-around besteht darin, den Wert des *Current of Set Type* zwischenzuspeichern und $cst(PM)$ vor dem `FIND NEXT` explizit wieder diesen Wert zuzuweisen:

```
( 5)          ptr := cst(PM)
(16)          FIND Mitwirkung RECORD DATABASE KEY IS ptr
```

Es sollte deutlich geworden sein, daß die DBTG-Standardisierung stattgefunden hatte, als relevante informatische Konzepte noch nicht ausreichend analysiert und erkannt worden waren. Unübersehbar ist der im Vergleich zum deklarativen SQL operationale Charakter der Anfragesprache. Die *Currency Pointer* stellen ein sehr einfaches Mittel dar, um Iterationen der Sets zu realisieren, das erhebliche Wartbarkeits- und Verifizierbarkeitsprobleme mit sich bringt.

Nähere Ausführungen zum Netzwerkmodell findet man Leser in C.J. Dates Lehrbuch „*An Introduction to Database Systems*“ [1].

Übungsblatt 9, Aufgabe 3

Wir nehmen an, daß wir die Film-Datenbank in einer Sprache wie C++ mit Klassendeklarationen wie in Abb. 4 skizzierten realisieren.

Diese Strukturen zugrundelegend navigiert man fast wie in der Netzwerkdatenbank durch die Listen. Einer der Unterschiede besteht darin, daß Iterationen über eine Liste zweckmäßigerweise

```

class Person {
public:
    ...
    Iterator<Participation*> get_movies();
    void insert_movie(Movie* m, char* r);
    ...
private:
    List<Participation*> movies;
};

class Movie {
public:
    ...
    Iterator<Participation*> get_persons();
    ...
private:
    List<Participation*> persons;
};

class Participation {
public:
    ...
    Person* get_person();
    Movie* get_movie();
    ...
private:
    Person* person;
    Movie* movie;
    char* role;
};

class PersonDB {
public:
    ...
    Person* find(char* name);
    ...
private:
    Set<Person*> persons;
};

```

Abbildung 4: Klassendeklarationen

```

PersonDB db("MyFile");           // opens database file
Person* p = pm.find("Winger");    // locates Person record
if (p) {
    Iterator<Participation*> it1 = p->get_movies();
    // visit all of Debra Winger's movies
    while (it1->next()) {
        Movie* m = it1->get_value()->get_movie();
        Iterator<Participation*> it2 = p->get_persons();
        while (it2->next()) {
            // Iterate through all participants,
            // find the director and print his name.
        }
    }
}
}

```

Abbildung 5: Navigierende Suche

über Iteratoren realisiert werden, statt Positionsangaben in Currency Pointers zu kodieren (siehe Abb. 5).

Bei der Implementierung von Änderungsfunktionen ist darauf zu achten, daß die Listen konsistent aktualisiert werden. Die Funktion *insert_movie* von *Person* beispielsweise unterstützt die Hinzufügung der Information, daß das Empfängerobjekt *p* (eine Person) am Film *m* mitgewirkt und dabei die Funktion *r* ausgeübt hat. Es wird also ein neues *Participation*-Objekt erzeugt und ein Verweis darauf zur *movies*-Liste von *p* hinzugefügt. Im *Participation*-Objekt wird über *movie* eine Verknüpfung zu *m* hergestellt und über *person* zu *p*. Außerdem muß aber auch die *persons*-Liste in *m* um einen Verweis auf das *Participation*-Objekt erweitert werden!

Übungsblatt 10, Aufgabe 1

Wenn wir die Film-Anwendung mit einem objektorientierten Datenbanksystem realisieren, gelangen wir zu einem ODMG-Schema wie dem in Abb. 6. Wie im Netzwerkschema bilden die Entity-Klassen des ER-Schemas die grundlegenden Bausteine — hier *Klassen* genannt. Klassendeklarationen sind Vereinbarungen über die Struktur einzelner Objekte, der Instanzen der Klassen. Das Schlüsselwort **extent** bewirkt, daß das DBMS die Menge aller Instanzen einer Klasse (auch *Extension* der Klasse genannt) verwaltet und somit Anfragen nach beispielsweise allen Filmen, die gewisse Kriterien erfüllen, unterstützt.

Wie in der C++-Realisierung werden Beziehungen direkt den einzelnen Klassen zugeordnet und nicht separat geführt (wie die Sets im Netzwerkmodell). Jede Person ist über die Beziehung *beteiligt_an* mit einer Menge von Filmen assoziiert; jeder Film über *hat_Mitwirkende* mit einer Menge von Personen. Im Gegensatz zur C++-Realisierung wird die konsistente Verwaltung der beiden Listen jedoch vom DBMS übernommen. Hierzu reicht es aus, daß bei den Klassenvereinbarungen explizit angegeben wird, daß es sich um inverse Beziehungen handelt. Es sei außerdem ausdrücklich darauf hingewiesen, daß die Repräsentation der N:M-Beziehung zwischen Filmen und Personen nicht wie im Netzwerkmodell in N:1-Beziehungen aufgelöst werden muß.

Das wäre die gute Nachricht. Die schlechte Nachricht allerdings ist, daß dieses Verfahren nur bei binären N:M-Beziehungen ohne Attribute zum Erfolg führt. Mehrstellige Beziehungen oder Beziehungen mit Attributen (wie in unserem Beispiel die Funktionsangabe bei der Mitwirkung einer Person in einem Film) führen dazu, daß wie auch beim Netzwerkschema die Beziehung selbst durch einen Entitätstyp bzw. eine Klasse repräsentiert wird. Auf diese Weise gelangen wir zum Schema aus Abb. 7.

Man beachte die Ähnlichkeit mit dem Netzwerkschema. Während in letzterem die Verknüpfungen zwischen *Mitwirkung*- und *Film*-Objekten durch den Set-Typ *FM* erfolgt, wird er im ODMG-

```

class Person (extent Alle_Personen) {
    attribute string Name;
    attribute string Vorname;
    relationship set<Film> beteiligt_an
        inverse Film::hat_Mitwirkende;
};

class Film (extent Alle_Filme) {
    attribute string Titel;
    attribute string Jahr;
    attribute short FSK;
    relationship set<Person> hat_Mitwirkende
        inverse Person::beteiligt_an;
};

```

Abbildung 6: ODMG-Schema I

Schema durch *hat_Mitwirkende* in *Film* und durch *beteiligt_an* in *Mitwirkung* abgebildet. Es werden zwei Beziehungstypen benötigt, um sowohl von *Mitwirkung*- zu *Film*-Objekten wie auch umgekehrt navigieren zu können. Im Netzwerkmodell war die bidirektionale Navigation dadurch gewährleistet, daß *Mitwirkung*- und *Film*-Records innerhalb einer Liste miteinander verknüpft waren und von jedem Member zum Owner, wie auch vom Owner zu den Members gesprungen werden konnte.

Betrachten wir nun zunächst die Anfrage nach allen Filmen, an denen Debra Winger mitgewirkt hat. In OQL wird dies wie folgt ausgedrückt:

```

SELECT m.beteiligt_an.Titel
FROM   p IN Alle_Personen, m IN p.hat_Funktion
WHERE  p.Name='Winger'

```

Die Variable *p* wird an alle Personen, d.h. an die Extension der Klasse *Person*, gebunden und erfährt eine weitere Einschränkung in der **WHERE**-Klausel. Die Variable *m* wird an alle Objekte der Klasse *Mitwirkung* gebunden, die mit *p* assoziiert sind.

Im Grunde entspricht **m IN p.hat_Funktion** dem Durchlaufen des *PM*-Sets von Debra Winger. Das jeweilige Objekt *m* von *Mitwirkung* verweist über *beteiligt_an* auf ein *Film*-Objekt, dessen Titel ausgegeben wird. Im Netzwerkmodell entspricht dies dem Aufsuchen des Owners im *FM*-Set von *m*. Referenzen, die einen Verweis auf genau ein Objekt darstellen (wie *beteiligt_an*), können direkt mittels der Punkt-Notation verfolgt werden, und mehrere solcher „Dereferenzierungen“ können in der Punkt-Notation hintereinander geschrieben werden. Das Verfolgen mengenwertiger Referenzen (wie *hat_Funktion*) erfordert die Einführung einer Variablen im **FROM**-Teil.

Die kompliziertere Anfrage nach allen Regisseuren, mit denen Debra Winger gearbeitet hat, liest sich wie folgt:

```

SELECT m2.besetzt_durch.Name
FROM   p IN Alle_Personen, m1 IN p.hat_Funktion,
        m2 IN m1.beteiligt_an.hat_Mitwirkende
WHERE  p.Name='Winger' AND m2.Funktion='Regisseur'

```

Der Leser möge zur Übung die Analogien zur Anfrage im Netzwerkmodell herstellen.

Literatur

- [1] DATE, C.J.: *An Introduction to Database Systems*. Addison-Wesley, Zweite Auflage, 1977.
- [2] KEMPER, ALFONS und ANDRÉ EICKLER: *Datenbanksysteme: Eine Einführung*. R. Oldenbourg Verlag, München, Wien, 2., aktualisierte und erweiterte Auflage, 1997.

```

class Person (extent Alle_Personen) {
    attribute string Name;
    attribute string Vorname;
    relationship set<Mitwirkung> hat_Funktion
        inverse Mitwirkung::besetzt_durch;
};

class Film (extent Alle_Filme) {
    attribute string Titel;
    attribute string Jahr;
    attribute short FSK;
    relationship set<Mitwirkung> hat_Mitwirkende
        inverse Mitwirkung::beteiligt_an;
};

class Mitwirkung (extent Alle_Mitwirkungen) {
    attribute string Funktion;
    relationship Film beteiligt_an
        inverse Film::hat_Mitwirkende;
    relationship Person besetzt_durch
        inverse Person::hat_Funktion;
};

```

Abbildung 7: ODMG-Schema II