Hochschule für Technik und Wirtschaft Berlin

University of Applied Sciences

# Datenbanken

Vorlesungsskript für das 3. Semester

Studiengang Internationale Medieninformatik

**6. Datenmodellierung -**

**Entity-Relationship-Modell**

Dozent: M. Sc. Burak Boyaci

Version: 18.11.2025

Wintersemester 25/26



Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

# 9 Entity-Relationship-Modell

## Kapitelübersicht



Wir haben in diesem Skript bis hierher erste Eindrücke von Datenbanken und der Programmierung mit SQL gewonnen. Wir wissen, wie wir per SQL-Datenbanken und Tabellen erstellen und Daten damit speichern und sich anzeigen lassen kann. Aber vielleicht fragen Sie sich jetzt: Brauchen wir dazu wirklich Datenbanken?

Wesentliche Ursache für diesen Eindruck ist zum Einen, dass wir mit unseren Datenbanken nur als Einzelnutzer gearbeitet haben. Die meisten Datenbanksysteme ermöglichen jedoch den gleichzeitigen Zugriff mehrerer Nutzer, eine Eigenschaft, die Excel so nicht bietet. Ein zweiter Grund liegt darin, dass wir bisher alle unsere Daten stets in einer einzigen Tabelle gespeichert haben. Die Konstruktion relationaler Datenbanksysteme ist aber gerade so angelegt, dass sie mehrere Tabellen mit logischen oder inhaltlichen Beziehungen untereinander speichern und manipulieren können. Damit gelingt es, den Datenhaushalt eines komplexen Systems wie zum Beispiel eines Unternehmens mit verschiedenen Beschäftigten, Produkten und Geschäftsprozessen strukturiert zu speichern, so dass zu jedem Zeitpunkt aktuelle Informationen von verschiedenen Nutzern verarbeitet werden können.
Den Entwurf und die Implementierung mehrerer Tabellen und deren Beziehungen untereinander werden wir für den Rest des Skripts behandeln. Betrachten wir aber zunächst einleitend, welche Probleme bei Datenspeicherung in einer einzigen Tabelle auftauchen können.

## 9.1 Probleme mit Daten in einer einzigen Tabelle

Wozu mehrere Tabellen? Das folgende Beispiel soll für einen einigermaßen realistischen Anwendungsfall kurz zusammenfassen, was wir mit einer einzigen Tabelle schaffen können, und was die Nachteile sind.

**Beispiel 9.1.** Ein produzierendes Unternehmen will eine Datenbank nutzen, um seine Umsätze zu speichern. Es setzt dafür eine Tabelle mit folgendem Relationstyp ein:

```
CREATE TABLE umsaetze (
   id            serial,
   datum         date,
   artikel       varchar(50),
   einzelpreis   decimal(10,2),
   anzahl        int,
   kunde         varchar(50),
   wohnort       varchar(50),
   umsatz        decimal(10,2)
);
```

Alles funktioniert wunderbar, das Unternehmen speichert die Umsätze für Oktober und November 2019:

| ID | Datum | Artikel | Einzelpreis | Anzahl | Kunde | Wohnort | Umsatz |
|---|---|---|---|---|---|---|---|
| 1 | 2019-11-06 | Hammer | 9.90 | 2 | Anna | Hagen | 19.80 |
| 2 | 2019-11-02 | Säge | 4.95 | 3 | Bert | Meschede | 14.85 |
| 3 | 2019-10-03 | Hammer | 9.90 | 1 | Otto | Soest | 9.90 |
| 4 | 2019-10-04 | Mottek | 12.95 | 1 | Anna | Hagen | 12.95 |
| 5 | 2019-11-06 | Säge | 4.95 | 2 | Doro | Iserlohn | 9.90 |
| 6 | 2019-10-06 | Zange | 3.90 | 4 | Anna | Hagen | 15.60 |
| 7 | 2019-11-06 | Zange | 3.90 | 3 | Doro | Iserlohn | 11.70 |
| 8 | 2019-10-06 | Hammer | 9.90 | 1 | Otto | Soest | 9.90 |
| 9 | 2019-11-05 | Säge | 4.95 | 4 | Anna | Hagen | 19.80 |
| 10 | 2019-11-06 | Mottek | 12.95 | 2 | Anna | Hagen | 25.90 |
| 11 | 2019-10-04 | Mottek | 12.95 | 3 | Bert | Meschede | 38.85 |
| 12 | 2019-10-06 | Zange | 3.90 | 1 | Bert | Meschede | 3.90 |

Die Geschäftsführung ist sehr zufrieden, sie kann alle ihr wichtigen Auswertungen durchführen lassen, zum Beispiel ...

- Welcher Kunde hat welchen Artikel gekauft?

```
SELECT kunde, artikel FROM umsaetze GROUP BY kunde,  artikel;
```

- Welcher Gesamtumsatz wurde im November 2019 erzielt?

```
SELECT sum (umsatz) FROM umsaetze WHERE datum BETWEEN '2019-11-01' AND '2019-11-30'
```

- In welchen Ort wurde welcher Artikel wie oft verkauft?

```
SELECT wohnort, artikel, sum(anzahl) FROM umsaetze GROUP BY wohnort, artikel
```

- Welcher Umsatz wurde je Artikel erzielt, sortiert nach Gesamtumsatz?

```
SELECT artikel, sum(umsatz) AS gesamt FROM umsaetze GROUP BY artikel
ORDER BY gesamt DESC;
```

- Welcher Artikel wurde im Oktober am meisten verkauft?

```
SELECT artikel, sum(anzahl) AS gesamt FROM umsaetze
WHERE datum BETWEEN '2019-10-01' AND '2019-10-31'
GROUP BY artikel HAVING gesamt >= ALL (
   SELECT sum (anzahl) FROM umsaetze
   WHERE  datum  BETWEEN'2019-10-01'AND'2019-10-31'
   GROUP BY artikel
) ;
```

Alternativ würde hier übrigens auch eine Abfrage mit drei SELECTs funktionieren:

```
SELECT artikel, SUM(anzahl) FROM umsaetze
WHERE datum BETWEEN  '2019-10-01' AND '2019-10-31'
GROUP BY artikel HAVING SUM (anzahl) = (
  SELECT MAX (gesamt) FROM (
    SELECT sum (anzahl) AS gesamt FROM umsaetze
    WHERE datum BETWEEN '2019-10-01'AND'2019-10-31'
    GROUP BY artikel
  )
);
```

Mit der innersten Abfrage wird wie oben eine Liste der Gesamtumsatzzahlen je Artikel ermittelt, für die aber nun in einem zweiten SELECT die Aggregatfunktion **MAX** zur Ermittlung des größten dieser Werte zwischengeschaltet wird, auf den dann in der äußersten Schleife in der **HAVING**-Klausel abgefragt wird. In der innersten Unterabfrage muss dabei allerdings ein Alias für Gesamtumsatzzahl benannt werden.

Aus Sicht der Geschäftsführung scheint also alles gut zu sein. Aber im Lauf der Zeit zeigen sich Probleme:

- Beim Einfügen eines neuen Umsatzes weiß ein Sachbearbeiter nicht den Wohnort des Kunden:

  ```
  INSERT INTO umsaetze (datum, artikel, einzelpreis, anzahl, kunde, umsatz)
  VALUES ('2019-10-21', 'Hammer', 9.90, 1, 'Otto', 9.90) ;
  ```

  Damit wird das Ergebnis unserer obigen Abfrage „In welchen Ort wurde welcher Artikel wie oft verkauft?“ fehlerhaft, obwohl die Datenbank den Ort ja „kennt“.

- Es wird unnötiger Speicherplatz verbraucht, denn dieselbe Information (Artikel-Einzelpreis, Kunde-Wohnort) wird bei *jedem einzelnen* Umsatz gespeichert.

Kann man diese Probleme beheben oder muss man halt damit leben?

Gehen wir den in dem Beispiel auftauchenden Problemen auf den Grund. Können wir unsere Datenspeicherung anders strukturieren, so dass sie möglichst vermieden werden? Von einem abstrakten Standpunkt aus gesehen handelt es sich beidemersten Problem um einen Spezialfall von *Dateninkonsistenz*, also eines Zustands der Datenbank, in der Datensätze sich widersprechende Informationen enthalten. Bei dem zweiten Problem handelt es sich um *Speicherineffizienz*, ein Mangel, der sich bei sehr großen Datenbeständen wie bei einer Unternehmensdatenbank erheblich auswirken kann.

Was ist die tiefere Ursache dieser Mängel? Beide sind bedingt durch *Redundanz* von Daten, also die mehrfache und damit prinzipiell überflüssige Speicherung identischer Information. Wie kann man sie strukturell beschränken? Die Antwort der Theorie relationaler Datenbanken: Durch die Aufteilung zusammenhängender Informationen in mehrere Tabellen. In unserem obigen Beispiel würden wir also nicht alle Informationen eines Umsatzes in einer einzigen Tabelle speichern, sondern die Kundendaten in einer eigenen Tabelle für Kunden, die Artikeldaten in einer eigenen Artikeltabelle, unddie spezifisch umsatzbezogenen Informationen in einer eigenen Umsatztabelle. Wie diese drei Tabellen aussehen könnten, werden wir am Ende dieses Kapitels sehen. Betrachten wir zunächst das Vorgehen, wie wir komplexe Informationsgeflechte in realen Kontexten analysieren und damit besser verstehen kann, nämlich die Modellierung von Daten.

## 9.2 Datenmodellierung

Warum Datenmodellierung? Ein Softwaresystem deckt stets nur einen kleinen Teil der realen Welt ab, d.h. ein „abstrahiertes“ Modell. Zu diesem Modell gehören einerseits Prozesse und

Abläufe, andererseits Daten und Informationen. Die Daten sind in der Informatik hierarchisch organisiert, so dass Computer sie verarbeiten können. Um die Daten der realen Anforderungen in diese Hierarchie zu bringen, müssen wir sie modellieren, d.h. ein Modell der Daten entwerfen.

Datenmodellierung dient im Allgemeinen mindestens drei Zielen:

a) Strukturierung der Anforderungen, um sie für die Entwickler programmierbar zu machen

b) Erleichterung der Kommunikation zwischen Anwendern und Entwicklern des Softwaresystems

c) Strukturierung der Daten, um sie effizient speichern und manipulieren zu können

Die Datenmodellierung ist die Grundlage fürden *logischen Datenbankentwurf* , also den Entwurf der Tabellen mit Attributen und Primärschlüsseln. Das ist die mittlere Ebene der ANSI-SPARC-Architektur. Als Methodik dazu verwendet man üblicherweise zur besseren Anschauung und Übersicht Diagramme, die Entity-Relationship-Diagramme. Zusammen mit einer Beschreibung der darin verwendeten Elemente bilden sie das *Entity-Relationship-Modell (ERM)*. Die grundlegende Idee ist, dass eine *Entity* das abstrahierte Modell einer Tabelle ist, so dass sich aus einem ERM die Tabellen der relationalen Datenbank in der Regel auf eindeutige Weise ableiten lassen.

## 9.3 ER-Diagramme

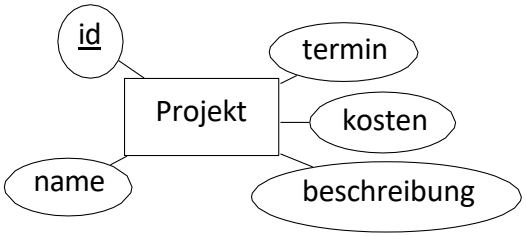
Für den logischen Datenbankentwurf werden die zu speichernden Daten meist mit *Entity-Relationship-Diagrammen (ER-Diagrammen)* grafisch modelliert. Dazu können verschiedene Notationen verwendet werden, die älteste und meistverbreitete ist die *Chen-Notation*.[1] Sie wird auch in diesem Skript verwendet. Es besteht aus drei Grundkomponenten: Dem Entitätstyp, der Beziehung und den Kardinalitäten.

### 9.3.1 Der Entitätstyp

Eine *Entität (entity)* im ERM ist ein individuelles Objekt der zu modellierenden Welt. Das kann ein realer Gegenstand sein wie ein Exemplar *Asterix der Gallier*, der Angestellte Otto Meier oder der Hammer mit der Artikelnummer 4711, aber auch ein immaterieller Vorgang, wie die Bestellung vom 1. April, oder ein Konstrukt wie der Liefervertrag Nr. 123 oder das Unternehmen ABC AG. Der *Entitätstyp (entity type)* ist der Oberbegriff oder die Kategorie einer oder mehrerer gleichartiger Entitäten, also *Buch*, *Angestellter*, *Artikel*, *Bestellung*, *Vertrag*, *Projekt* oder *Unternehmen*. (Die Entität ist daher eng verwandt mit dem Objektbegriff der Objektorientierung, der Entitätstyp entsprechend mit dem Begriff der Klasse.) Ein Entitätstyp wird im ER-Diagramm mit einem Rechteck dargestellt, in dem der Name des Typs steht:


Eine Entität hat in der Regel mehrere *Attribute*. Eines davon ist der Primärschlüssel der Entität, der sie unter allen anderen Entitäten stets eindeutig identifiziert. Häufig werden die Attribute als kleine Ellipsen mit dem Entitätstyp verbunden oder alternativ in dem Rechteck des Entitätstypen aufgelistet, also zum Beispiel:

[1] Chen (1976).



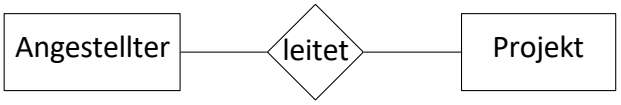

Projekt
- id
- name
- termin
- kosten
- beschreibung

**Bemerkung 9.2.** Wir erkennen hier leicht, dass die Entitätstypen in einem ER-Diagramm die Tabellen einer relationalen Datenbank modellieren werden. Entsprechend sind die Attribute der Entitäten die Spalten der Tabelle, und eine Entität entspricht einem konkreten Datensatz, also einer Zeile der Tabelle. Bei der Bildung der Tabellen aus den Entitätstypen eines ER-Diagramms sind aber einige Regeln zu beachten. Damit werden wir uns im nächsten Kapitel näher beschäftigen.

**Bemerkung 9.3.** Meist wird der Begriff Entitätstyp einfach auch mit„Entität“ bezeichnet. Wir werden das im Folgenden auch oft tun, wenn aus dem Zusammenhang ersichtlich bleibt, was gemeint ist.

### 9.3.2 Die Beziehung

Eine *Beziehung (relationship)* im ERMistder Zusammenhang zwischen zwei Entitäten. Siewird in einem ER-Diagramm mit einem Verb in eine Raute zwischen zwei Entitätstypen dargestellt, z.B.:



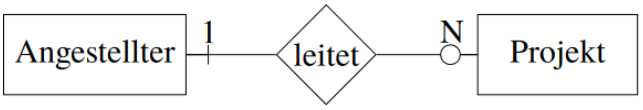

### *9.3.3* Die *Kardinalität*

Eine *Kardinalität (cardinality)* drückt aus, wieviel Entitäten des einen Typs mit wieviel Entitäten des anderen Typs eine gegebene Beziehung maximal haben können und mindestens haben müssen. Dabei wird für die Maximalzahl eine der Möglichkeiten 1 oder N für „eins“ oder „mehrere“ über der Beziehung geschrieben, und ein Kreis oder ein Strich an der Beziehungslinie für die Mindestzahl. Damit gibt es vier mögliche Kardinalitäten:

| **Symbol** | 1 | 1 | N | N |
|---|---|---|---|---|
| **Bezeichnung** | 1 | C | M | CM |

(9.1)

Unter die Kardinalitäten sind die im Englischen üblichen Bezeichnungen 1, C (für “choice”), M (für “many”) und CM dargestellt. Im Deutschen wird bei einem Strich von einer *Muss-Beziehung* gesprochen, bei einem Kreis von einer *Kann-Beziehung*. In einem ER-Diagramm wird für eine konkrete Entität des einen Entitätstyps die Kardinalität an dem jeweils gegenüberliegenden Entitätstypen notiert. Betrachten wir dazu das folgende Beispiel:




Ein*e Angestellte*r *kann* hier mehrere Projekte leiten, ein Projekt aber *muss* von genau einem oder einer Angestellten geleitet werden. Dies ist also eine 1-CM-Beziehung, oder auf Deutsch

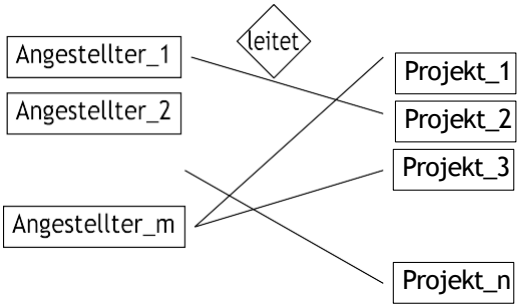
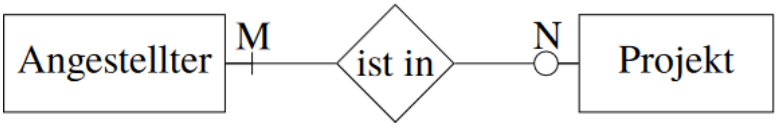



**Abbildung 9.1:** Schema von Beziehungen einzelner Entitäten zueinander bei einer 1-CMBeziehung. Hier kann es auf der linken Seite Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite genau eine Beziehung haben muss.

etwas umständlicher eine Beziehung „1 zu N muss-kann“. Skizzieren wir die einzelnen Entitäten Angestellter_1, Angestellter_2, . . . , Angestellter_m und Projekt_1, Projekt_2, . . . , Projekt_n und ihre konkreten Beziehungen, so erhalten wir die Abbildung 8.1. Bei einer 1-CM-Beziehung kann es also auf der 1-Seite Entitäten geben, die keine Beziehung haben (z.B. Angestellter_2), während jede Entität der N-Seite genau eine Beziehung haben muss. Entsprechend wird für eine Beziehung „mehrere zu mehrere“ üblicherweise das Paar „M : N“ verwendet wie bei folgender M-CM-Beziehung:




Ein*e Angestellte*r kann hier in mehreren Projekten sein, ein Projekt muss mindestens eine oder einen Angestellten haben. (Man verwendet hier „M“ statt „N“ um zu verdeutlichen, dass beide Kardinalitäten unabhängig voneinander sind, also auf beide Seiten nicht die gleich viele Entitäten existieren müssen.) Bei einer M-CM-Beziehung kann es auf der linken Seite also

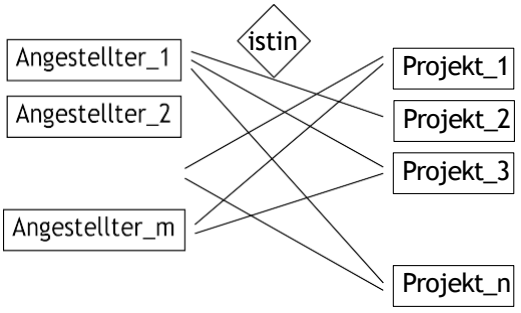



**Abbildung 9.2:** Schema der Beziehungen einzelner Entitäten zueinander bei einer M-CM-Beziehungen. Hier kann es auf der linken Seite Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite mindestens eine Beziehung haben muss.

Entitäten geben, die keine Beziehung haben (Angestellter_2), während jede Entität der rechten Seite mindestens eine Beziehung haben muss.

**Beispiel 9.4.** *(ERM der Umsatzdatenbank aus Beispiel 9.1)*. Wie können wir ein Datenmodell einer Umsatzdatenbank aus unserer ersten Version in Beispiel 9.1 entwerfen? Zunächst müssen wir dazu die Leitfrage beantworten: Welche der relevanten Daten gehören zusammen? Daraus nämlich ergeben sich die Entitätstypen und ihre Attribute. In dem Beispiel können wir leicht erkennen, dass wir drei Entitätstypen haben:

| Kunde |
|---|
| - <u>name</u> |
| - wohnort |

| Artikel |
|---|
| - <u>name</u> |
| - einzelpreis |

| Umsatz |
|---|
| - <u>datum</u> |
| - <u>kunde.name</u> |
| - <u>artikel.name</u> |
| - anzahl |
| - umsatz |

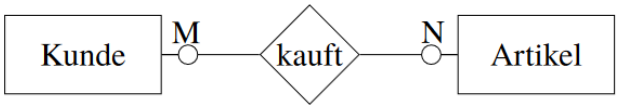
Welche Beziehungen haben wir nun zwischen den Entitäten? Dazu müssen wir zunächst die Frage beantworten: Welche Entitätspaare gibt es? Kombinatorisch gesehen haben wir $\binom{3}{2} = 3$ Paare für drei Entitäten, nämlich Artikel – Umsatz, Kunde – Umsatz und Kunde – Artikel. Die erste Beziehung davon lässt sich darstellen als:


$\Rightarrow$ 1-CM

Mit einem gegebenen Artikel können also mehrere Umsätze erzielt werden, es kann aber auch Artikel geben, die gar keinen Umsatz erzielt. Umgekehrt muss zu einem gegebenen Umsatz genau ein Artikel gehören. Die zweite Beziehung lautet:


$\Rightarrow$ 1-CM

Diese Beziehung bedeutet, dass zu einer Entität „Kunde“ mehrere Entitäten „Umsätze“ gehören können, d.h. ein Kunde kann auch gespeichert sein, wenn er noch keinen Umsatz erzeugt hat. Ist das sinnvoll? Hier zeigt sich schon die gestalterische Wirkung der Datenmodellierung: Ein nur an Speichereffizienz interessierter Informatiker könnte argumentieren, dass für doch nur Kunden mit Umsatz sehen wollen, ein Vertriebler des Unternehmens dagegen würde eher den Aspekt sehen, dass der Kunde vielleicht für einen Artikel interessiert werden könnte und daher gespeichert werden sollte. Die dritte Beziehung ist wieder etwas leichter darzustellen:




$\Rightarrow$ CM-CM

Ein Kunde kann hier mehrere Artikel kaufen (kauft aber vielleicht auch keinen), und ein Artikel kann von mehreren Kunden gekauft werden (vielleicht aber auch gar nicht). Wie aber können wir nun die Beziehungen zwischen diesen Tabellen in SQL programmieren? Das werden wir im nächsten Kapitel sehen.

# 10

# Ableitung von Relationen aus dem ERM

**Kapitelübersicht**



## 10.1 Grundregeln der Implementierung von Beziehungen

Die erste Implementierungsregel ist eine eigentlich eine Konvention und betrifft die Tabellennamen.

**Regel 1.** *Obwohl die Namen von Entitätstypen meist Substantive im Singular sind, sollten die Namen von Tabellen einer relationalen Datenbank im Plural benannt werden.*

Die weiteren Implementierungsregeln beziehen sich auf die einzelnen Kardinalitäten. Dazu beantworten wir zunächst aber die Frage: Wieviel verschiedene Implementierungsregeln benötigen wir eigentlich überhaupt?

**Bemerkung 9.4.** Die ersten Überlegungen in diesem Abschnitt zu einer Umsetzung eines Entity-Relationship-Modells in Datenbanktabellen beruhen im Wesentlichen zunächst nicht auf diesen 10 Fällen, sondern lediglich auf den drei *Hauptbeziehungen* 1:1, 1:N und M:N, als ohne die Mindestzahlen, und hier vor allem die Hauptbeziehungen 1:N und M:N.


Mit diesen beiden Hauptbeziehungen können wir am klarsten die Grundprinzipien der Implementierung von Beziehungen im Allgemeinen erkennen. Die Beziehung 1:1 ist dagegen etwas schwieriger und wird am Ende etwas ausführlicher betrachtet.

□

## 10.2 Die Hauptbeziehung 1:N

Eine 1:N-Beziehung lässt sich wie folgt darstellen:


Hier werden einem einzelnen Datensatz in Tabelle t1 also mehrere Datensätze aus Tabelle t2 zugeordnet, und umgekehrt einem Datensatz in Tabelle t2 ein einziger Datensatz aus t1. Es stellt sich sofort die Frage: In welche der Tabellen muss ein Fremdschlüssel eingefügt werden? In alle beide oder reicht ein einziger Fremdschlüssel in einer der Tabellen?
Mit einem kurzen Gedankenspiel können wir uns die Antwort herleiten: Würden wir gemäß Abbildung 9.1 den Fremdschlüssel eines referenzierten Datensatzes aus t2 in Tabelle t1 speichern, bräuchten wir mehrere Attribute dafür. Wieviel aber genau, hängt vomkonkreten Fall ab; vielleicht brauchen wir in einem Fall nur einen Fremdschlüssel, in einem anderen vielleicht zwei, in wieder einem anderen vielleicht 100. Selbst wenn wir genau wüssten, dass wir maximal 10 Fremdschlüssel bräuchten, wäre es ziemlich ineffizient, wenn wir in den meisten Fällen nur ein oder zwei verwenden müssen. Folgerung: Tabelle t1 ist kein guter Kandidat für die Speicherung von Fremdschlüsseln. Was ist umgekehrt mit Tabelle t2? Hier brauchen wir maximal einen Fremdschlüssel, egal wie groß N sein mag! Die Regel für die Hauptbeziehung 1:N lautet daher:

**Regel 2.** *Bei einer Beziehung 1:N wird nur ein Fremdschlüssel benötigt. Er wird in der N-Seite gespeichert, wie in Abbildung 10.2 dargestellt.*

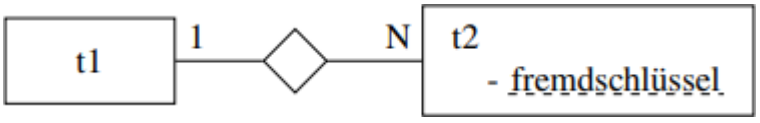



**Abbildung 10.2:** Implementierung einer Beziehung 1:N. Der Fremdschlüssel kommt auf die N-Seite

Es gibt drei Beziehungen der Kategorie 1:N in den Kombinationen „kann“ und „muss“, nämlich C-CM, 1-CM und C-M. Die ersten beiden werden wir jetzt näher untersuchen, C-M erst in Abschnitt 10.6 auf Seite 17.

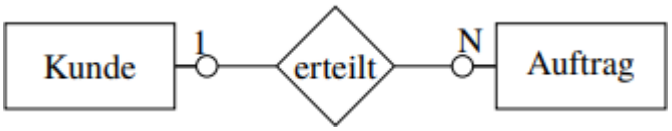
### 10.2.1 C-CM-Beziehungen („1:N kann-kann“)


**Regel 3.** *Bei einer C-CM-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel kann **NULL** sein.*

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY (id)
);
```

```
CREATE TABLE t2 (
  ...,
  fs int,
  FOREIGN KEY (fs) REFERENCES t1(id)
  ON DELETE RESTRICT
);
```

**Beispiel 10.5.** Ein einfaches Beispiel für eine C-CM-Beziehung ist:




Ein Kunde kann hier mehrere Aufträge erteilen, ein Auftrag dagegen kann von höchstens einem Kunden erteilt werden. Es kann aber auch Aufträge geben, die nicht von einem Kunden erteilt werden, beispielsweise betriebsinterne Aufträge. □

### 10.2.2 1-CM-Beziehungen („1:N muss-kann“)


**Regel 4.** *Bei einer 1-CM-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel darf dabei nicht **NULL** sein.*

```
CREATE TABLE t1 (
   id int,
   ...,
 PRIMARY KEY (id)
);
```

```
CREATE TABLE t2 (
   ...,
   fs int NOT NULL ,
   FOREIGN KEY (fs) REFERENCES t1(id)
   ON DELETE RESTRICT
);
```

**Beispiel 10.6.** Ein Beispiel für eine 1-CM-Beziehung ist die folgende:

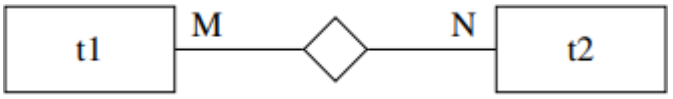

Hier kann eine Abteilung mehrere Angestellte haben, aber jede*r Angestellte muss zu einer Abteilung gehören. In SQL muss also der Fremdschlüssel in der Tabelle angestelltergespeichert werden, z.B. mit

```
CREATE TABLE angestellter (
   ...,
   abteilung_id int NOT NULL   ,
   FOREIGN KEY (abteilung_id) REFERENCES abteilung(id) ON DELETE RESTRICT
);
```

Um die Tabelle angestellter anlegen zu können, muss zuvor die Tabelle abteilung bereits angelegt sein. □

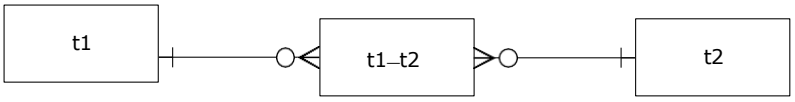
## 10.3 Die Hauptbeziehung M:N

Zur Umsetzung einer M:N-Beziehung müssen wir uns zunächst ein paar Gedanken machen: Mit unseren bisherigen Regeln, die Beziehung mit Fremdschlüsseln zu implementieren, kommen wir hier nicht weiter.



Denn wieviel Fremdschlüssel müssten wir für t1 verwenden und wieviel für t2? Um beliebig viele Datensätze vom Typ t2 zu referenzieren, bräuchten wir für t1 unbegrenzt viele Fremdschlüssel, und umgekehrt – das allein ist ja schon rein logisch nicht realisierbar

Die Lösung besteht darin, die Beziehung mit Hilfe einer dritten „künstlichen“ Tabelle aufzulösen, mit einer sogenannten *Beziehungstabelle*. Jeder Datensatz dieser Tabelle speichert gewissermaßen jedes Datensatzpaar der beiden Grundtabellen, das eine Beziehung bildet. Auf diese Weise erhält man statt der einen M-N-Beziehung zwei 1-N-Beziehungen:




Da die Beziehungstabelle keine Entität nach dem ERM ist, sondern eigentlich ja ein technisches Artefakt zur Implementierung, ist dieses Diagramm der aufgelösten Entitätsbeziehung ein „Tabellendiagramm“. Um Tabellendiagramme von ER-Diagrammenzuunterscheiden, benutzen wir hierfür die neben der Chen-Notation auch gebräuchliche *Krähenfußnotation (crow’s foot notation)*. Hier werden die Kardinalitäten ohne hochstehende Ziffern oder Buchstaben dargestellt, sondern mit Kreisen, Strichen und „Krähenfüßen“ am Ende der Beziehungen:


(10.2)

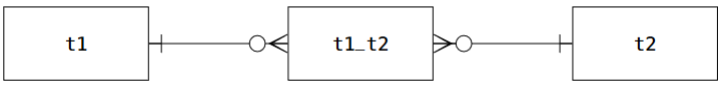
Vgl. Piepmeyer (2011:S. 120) und – im Vergleich – die Chen-Notation (9.1) auf Seite 5.

### 10.3.1 Die Beziehung CM-CM („M:N kann-kann“)

Mit den Überlegungen des vorherigen Abschnitts betrachten wir nun eine CM-CM-Beziehung:


Wir können eine CM-CM-Beziehung durch zwei 1-CM-Beziehungen auflösen, die wir jeweils mit Regel 4 implementieren:

**Regel 5.** *Eine CM-CM-Beziehung wird zu zwei 1-CM-Beziehungen mit Hilfe einer Beziehungstabelle mit zwei Fremdschlüsseln aufgelöst, die die Primärschlüssel der beiden Grundtabellen sind.*



```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY (id)
);
```

```
CREATE TABLE t1_t2 (
  id SERIAL,
  fs_1 int NOT NULL ,
  fs_2 int NOT NULL ,
  ...,
  PRIMARY KEY (id),
  FOREIGN KEY (fs_1) REFERENCES t1(id)
  ON DELETE RESTRICT,
  FOREIGN KEY (fs_2) REFERENCES t2(id)
  ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
  id int,
  ...,
  PRIMARY KEY (id)
);
```

*Ist zusätzlich durch den konkreten Anwendungsfall garantiert, dass zwei einzelne Entitäten* höchstens *eine Beziehung zueinander haben können, so können die beiden Fremdschlüssel den Primärschlüssel der Beziehungstabelle bilden.*

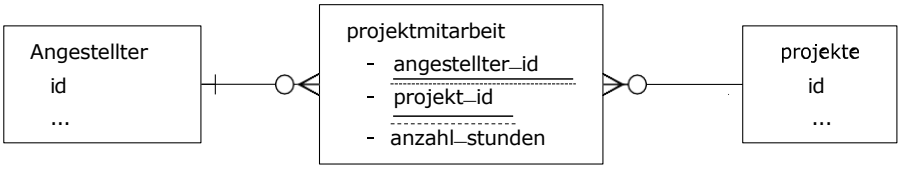
```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY (id)
);
```

```
CREATE TABLE t1_t2 (
  fs_1 int,
  fs_2 int,
  ...,
  PRIMARY KEY (fs_1, fs_2),
  FOREIGN KEY (fs_1) REFERENCES t1(id)
  ON DELETE RESTRICT,
  FOREIGNKEY (fs_2) REFERENCES t2(id)
  ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
id int,
  ...,
PRIMARY KEY (id)
);
```

**Beispiel 10.7.** Ein Beispiel für eine CM-CM-Beziehung ist:

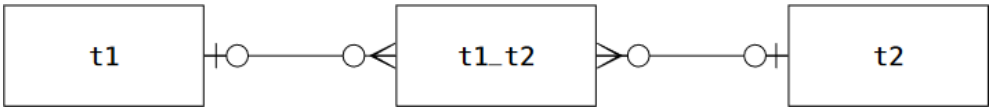

Ein*e Angestellte/r kann also in mehreren Projekten sein, und ein Projekt kann mehrere Angestellte haben. Diese Beziehung muss durch eine Beziehungstabelle aufgelöst werden:




Sie kann neben den beiden Fremdschlüsseln, wie hier, weitere spezielle Attribute enthalten. Durch die „Muss-Seiten“ der Kardinalitäten wird durch die Datenbank gesichert, dass eine CM-CM-Beziehung auch nur dann als Datensatz der Beziehungstabelle gespeichert werden kann, wenn die Datensätze der beiden beteiligten Entitäten bereits existieren. In seltenen Anwendungsfällen muss eine CM-CM-Beziehung jedoch auch in zwei C-CM-Beziehungen aufgelöst werden. Das ist der Fall für sogenannte *gerichtete* Beziehungen, die in dem Sinne „unsymmetrisch“ sind, dass ein Datensatz der einen Tabelle zwar den Datensatz der anderen „kennt“, aber nicht unbedingt umgekehrt. (Beachten wir dabei, dass eine solche Beziehung im

Allgemeinen in einem ER-Diagramm nicht dargestellt werden kann, sondern sich aus dem Kontext des jeweiligen Anwendungsfalls ergibt.)

**Regel 6.** *In seltenen Fällen ist eine CM-CM-Beziehung gerichtet. Sie muss dann in zwei CCM-Beziehungen aufgelöst werden:*




*Damit wird sie mit Regel 4 implementiert durch:*

```
CREATE TABLE t1 (
   id int,
   ...,
   PRIMARY KEY (id)
);
```

```
CREATE TABLE t1_t2 (
   id SERIAL,
   fs_1 int,
   fs_2 int,
   ...,
   PRIMARY KEY (id);
   FOREIGN KEY (fs_1) REFERENCES
   t1(id)
      ON DELETE RESTRICT,
   FOREIGN KEY (fs_2) REFERENCES
   t2(id)
      ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
   id int,
   ...,
    PRIMARY KEY (id)
);
```

Ein Beispiel dafür werden wir weiter unten mit dem Datenmodell für rekursive Beziehungen zur Darstellung gerichteter Graphen in Beispiel 10.12 kennen lernen.

## 10.4 Die Hauptbeziehung 1:1

### 10.4.1 1-C-Beziehungen („1:1 muss-kann“)

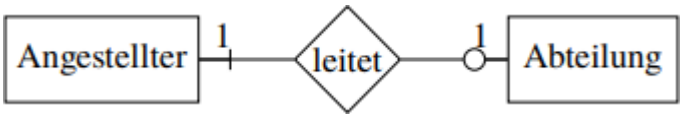

Bei der 1-C-Beziehung muss der Fremdschlüssel auf die C-Seite und darf nicht **NULL** sein, denn es muss ja stets mindestens einen Datensatz der Tabelle t1 geben. Diese Bedingung ist in SQL einfach mit der Integritätsregel **NOT NULL**implementierbar. Wie aber legt man dann in SQL fest, dass auch stets *höchstens* ein Datensatz aus t1 zu einem Datensatz aus t2 gespeichert werden kann? Hier hilft uns das reservierte Wort **UNIQUE**, das verhindert, dass in einer Tabelle zwei Datensätze mit dem gleichen Attributwert gespeichert werden, so dass also keine Dubletten zugelassen werden. Damit erhalten wir die folgende Ableitungsregel:

**Regel 7.** *Bei einer 1-C-Beziehung wird der Primärschlüssel der 1-Seite der Fremdschlüssel der anderen Seite. Der Fremdschlüssel muss dabei **NOT NULL** und **UNIQUE** sein.*

```
CREATE TABLE t1 (
  id int,
  ...,
  PRIMARY KEY (id)
);
```

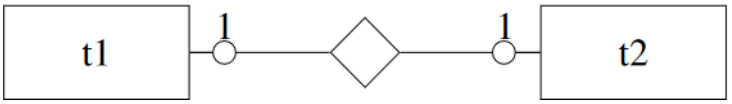
```
CREATE TABLE t2 (
  ...,
  fs int NOT NULL UNIQUE,
  FOREIGN KEY (fs) REFERENCES t1(id)
  ON DELETE RESTRICT
);
```

**Beispiel 10.8.** Eine einfache 1-C-Beziehung ist die folgende:

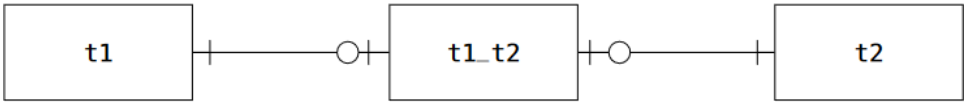



Hier kann ein*e Angestellter höchstens eine Abteilung leiten, aber eine Abteilung muss genau einen Leiter haben.

### 10.4.2 C-C-Beziehungen („1:1 kann-kann“)




Auf den ersten Blick scheint es nicht besonders schwer, diese Beziehung ähnlich wie die 1-C-Beziehung in Regel 7 zu implementieren. Beispielsweise könnte man einfach den Fremdschlüssel wie dort bei t2 vorsehen und ihn lediglich mit **UNIQUE** einschränken, also **NOT NULL** weglassen und so den Wert **NULL** erlauben; Dubletten wären dann trotzdem nicht zugelassen. Leider erfordert bei vielen SQL-Dialekten **UNIQUE** jedoch die Integritätsregel **NOT NULL**. Falls zudemdas tatsächliche Vorkommen der Beziehung eher selten ist, so würde es bei großen Datenbeständen zu ineffizientem Speicherplatzbedarf führen, da die Fremdschlüsselattribute fast überallNULL wären. Wir sollten daher auch in diesem Fall mit einer Beziehungstabelle arbeiten:




Damit wird eine C-C-Beziehung also in zwei 1-C-Beziehungen aufgelöst.

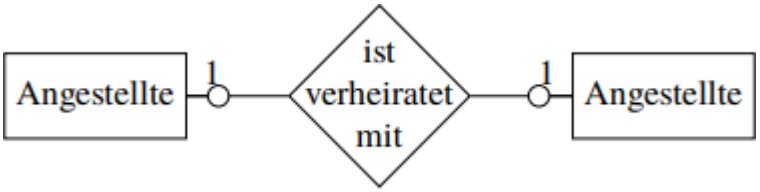
**Regel 8.** *Einer C-C-Beziehung wird durch eine Beziehungstabelle aufgelöst, die die Primärschlüssel jeweils als Fremdschlüssel erhält. Beide Fremdschlüssel müssen dabei **NOT NULL** und **UNIQUE** sein und bilden den Primärschlüssel der Tabelle.*

```
CREATE TABLE t1 (
  id int,
  ...,
 PRIMARY KEY (id)
);
```

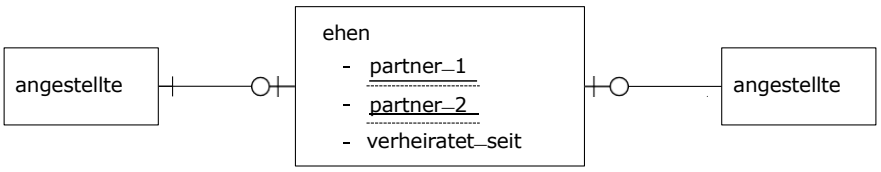
```
CREATETABLE t1_t2 (
  fs_1 int NOT NULL UNIQUE,
  fs_2 int NOT NULL UNIQUE,
  PRIMARY KEY (fs_1, fs_2);
  FOREIGN KEY (fs_1) REFERENCES t1(id)
 ON DELETE RESTRICT ,
  FOREIGN KEY (fs_2) REFERENCES t2(id)
  ON DELETE RESTRICT
);
```

```
CREATE TABLE t2 (
  id int,
  ...,
 PRIMARY KEY (id)
);
```

**Beispiel 10.9.** Eine C-C-Beziehung ist die folgende:

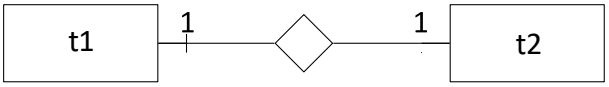



Sie wird aufgelöst durch eine Beziehungstabelle ehen:




In die Beziehungstabelle werden nur Ehen eingetragen, die tatsächlich vorhanden sind. Damit werden diese relativ selten auftretenden Fälle sehr effizient gespeichert.

### 10.4.3 1-1-Beziehungen („1:1 muss-muss“)




Eine 1-1-Beziehung ist eine ganz spezielle Beziehung, die in zwei oder mehr Tabellen gar nicht zu implementieren ist. Zu jeder Entität des Typs t1 müsste es nämlich stets genau eine Entität des Typs t2 geben und umgekehrt, insbesondere müsste es also in Tabelle `t1`stets genauso viele Datensätze geben wie in Tabelle`t2`. Diebeiden Teilnehmereiner solchen Beziehung müssten also *gleichzeitig* entstehen, was technisch nicht möglich ist.[2] Da die beiden beteiligten Entitätstypen ihre Unabhängigkeit vollständig eingebüßt haben, werden sie üblicherweise in einer einzigen Tabelle implementiert:

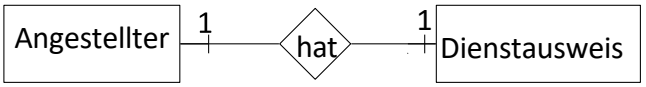
**Regel 9.** *Zwei Entitätstypen mit einer 1-1-Beziehung („1:1 muss-muss“) werden zu einer einzelnen Tabelle zusammengefasst.*

```
CREATE TABLE t (
  attribute_von_t1 <Typ> NOT NULL UNIQUE,
  ...,
  attribute_von_t2 <Typ> NOT NULL UNIQUE,
  ...
);
```

*Falls nur einer der Entitätstypen in dem ER-Diagramm weitere Beziehungen hat, kann man diesen als Tabellennamen verwenden, ansonsten sollte ein neuer Name gesucht werden.*

Im Allgemeinen sollte schon beim Modellieren einer 1-1-Beziehung auf jeden Fallhinterfragt werden, ob sie tatsächlich überhaupt so vorliegt. In der Realität kommt sie eher selten vor.

**Beispiel 10.10.** Ein Beispiel für eine 1-1-Beziehung ist die folgende:




Hier ist es sinnvoll, die Attribute des Dienstausweises einfach in die Tabelle `angestellter` aufzunehmen.

## 10.5 Schlecht oder gar nicht implementierbare Beziehungen

Es fehlen noch die Beziehungen x-M, bei denen mindestens ein Teilnehmer mindestens einmal auftreten *muss* und mehrfach auftreten kann, also

- C-M („1:N kann-muss“),
- 1-M („1:N muss-muss“),
- CM-M („M:N kann-muss“) und
- M-M („M:N muss-muss“).

Zwar könnten wir diese Beziehungen grundsätzlich wie die x-CM-Beziehungen mit einer Beziehungstabelle auflösen, allerdings kann SQL dazu keine ausreichenden Integritätsregeln garantieren.[3] Das relationale Modell stößt hier an seine logischen Grenzen.

[2] Piepmeyer (2011):S. 137.
[3] Piepmeyer (2011):S. 139ff.

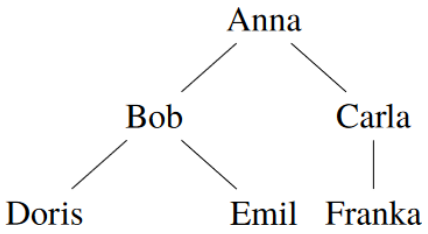
# 10.6 Spezielle Beziehungen

## 10.6.1 Rekursive Beziehungen

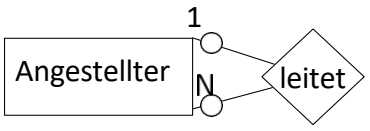
Manchmal gibt es Anwendungsfälle, in denen Entitäten vom selben Entitätstyp Beziehungen miteinander haben. Solche Beziehungen heißen *rekursiv*. Beispiele sind die Knoten von Graphen oder Netzwerken, aber auch die Einheiten einer Hierarchie wie das Organigramm eines Unternehmens. Es gibt zwei wesentliche rekursive Beziehungen, eine C-C-Mund eine CM-CM-Beziehung:


Die rekursive C-CM-Beziehung kann bei Hierarchien und baumartigen Strukturen modelliert werden, die rekursive CM-CM-Beziehung bei Graphen und Netzwerken. Entsprechend können sie mit der Regel 8 bzw. Regel 10 aufgelöst werden.

**Beispiel 10.11.** Gegeben sei die folgende Vorgesetztenstruktur eines Unternehmens:




Um diese Hierarchie zu speichern, kann man die folgende rekursive C-CM-Beziehung modellieren:




D.h. ein Angestellter kann mehrere Angestellte leiten und jeder Angestellte hat höchstens einen Chef. Mit Regel 8 lässt sich diese C-CM-Beziehung mit dem Fremdschlüssel chef(oder leiter) wie folgt implementieren:

```
CREATE TABLE angestellte (
   name VARCHAR(10),
   chef VARCHAR(10),
   PRIMARY KEY (name),
   FOREIGN KEY (chef) REFERENCES angestellte(name) ON DELETE SET NULL
);
```

Mit dem folgenden Einfügebefehl können wir dann die obige Vorgesetztenstruktur speichern:

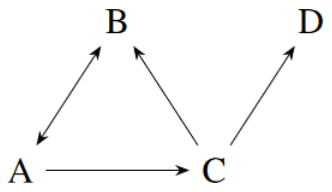
```
INSERT INTO angestellte(name, chef) VALUES
  ('Anna', NULL), ('Bob', 'Anna'), ('Carla', 'Anna'),
  ('Doris', 'Bob'), ('Emil', 'Bob'), ('Franka', 'Carla');
```

Dann ergibt ein SELECT auf diese Tabelle:

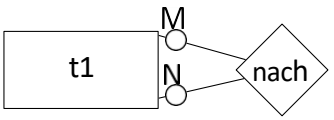
| name | chef |
|---|---|
| Anna | NULL |
| Bob | Anna |
| Carla | Anna |
| Doris | Bob |
| Emil | Bob |
| Franka | Carla |

□

**Beispiel 10.12.** *(Digraph)* Ein *Digraph (directed graph)* ist ein Graph, dessen Knoten durch gerichtete Kanten („Pfeile“) verbunden sind, beispielsweise




für die vier Knoten A, B, C, D. Ein Knoten kann nun auf mehrere Knoten weisen, aber es können auch mehrere Knoten auf ihn weisen. Es handelt sich also um eine rekursiven CM-CM Beziehung (M:N „kann-kann“):



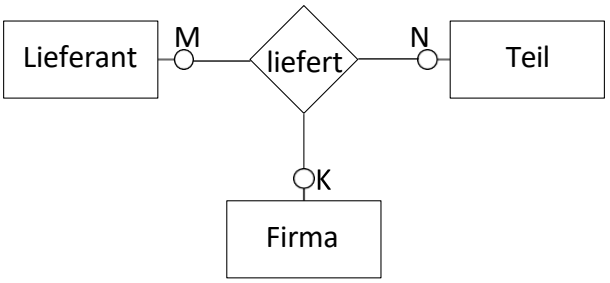

Nach Regel 6 für CM-CM-Beziehungen aufgelöst ergibt sich daraus das Tabellendiagramm


```
CREATE TABLE knoten (
  name varchar(1),
  PRIMARY  KEY (name)
);
  CREATE TABLE
  kanten ( id SERIAL,
  Von varchar(1),
  nach varchar(1),
  PRIMARY KEY (id),
  FOREIGN KEY (von)  REFERENCES knoten(name) ON DELETE CASCADE,
  FOREIGN KEY (nach) REFERENCES knoten(name) ON DELETE CASCADE
);
```

```
--Knoten einfügen:
INSERT INTO knoten(name) VALUES ('A'), ('B'), ('C'), ('D');
--Kanten A<->B,A->C,C->B,C->D einfügen:
INSERT INTO kanten (von, nach)
VALUES ('A','B'), ('B','A'), ('A','C'), ('C', 'B'), ('C','D');
```

### 10.6.2 Mehrwertige Beziehungen

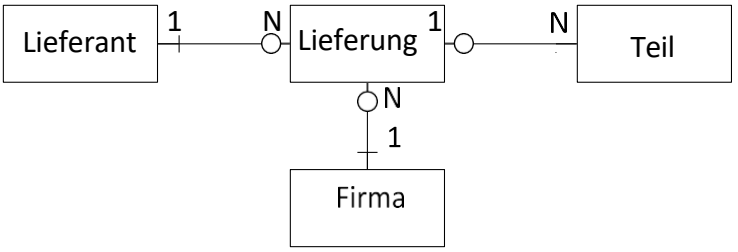
Neben den bisher behandelten 2-wertigen Beziehungen gibt es manchmal auch höherwertige Beziehungen. So ist beispielweise eine dreiwertige Beziehung durch die Entitäten *Lieferant*, *Teil* und *Firma* gegeben:




Diese Beziehung wird durch eine zusätzliche Beziehungstabelle aufgelöst, die jeden Primärschlüssel der Grundtabellen als Fremdschlüssel enthält:

*Lieferung* (lieferant_id, teil_id, firmen_id, anzahl)

In der Regel können die drei Fremdschlüssel den Primärschlüssel der Beziehungstabelle bilden. So entstehen aus der dreiwertigen CM-CM-CM-Beziehung drei zweiwertige C-CM-Beziehungen: Sie wird durch eine Beziehungstabelle in zweiwertige 1:N-Beziehungen aufgelöst:




Entsprechend können wir mit höherwertigen Beziehungen verfahren, so dass wir am Ende immer ein Tabellendiagramm erhalten können, das ausschließlich aus zweiwertigen Beziehungen besteht.
Allerdings bleibt zu bemerken, dass höherwertige Beziehungen in der Praxis eher selten auftreten.[4] Da sie auch eine sehr enge Abhängigkeit der beteiligten Tabellen bewirken, die die Komplexität bei SQL-Anweisungen erhöht, sollte man im Allgemeinen versuchen, sie gleich als zweiwertige Entitäten zu modellieren.

[4] Piepmeyer (2011):S. 130ff.

## 10.7 SQL mit mehreren Tabellen

Wiewirim Zusammenhang mitder Modellierung der Entity-Relationships schon gesehen haben, ist die Information in relationalen Datenbanken für Anwendungen aus der Praxis übermehrere Tabellen verteilt. SQL muss also Mechanismen bereitstellen, Abfragen über mehrere Tabellen in der Ergebnismenge zusammenzuführen. Dabei ist allgemein folgende grundsätzliche Regel zu beachten.

**Regel 10.** *Ein Spaltenname muss in einer SELECT-Abfrage eindeutig sein. Falls ein Spaltenname s in mehreren Tabellenvorkommt, muss ernach einem Punkt an den gewünschten Tabellennamen tabelle angefügt werden, also*

***SELECT*** *..., tabelle.s, ...* ***FROM*** *..., tabelle, ...;*

*Mit dem reservierten Wort* ***AS*** *kann jeder Spalte ein neuer Name (Alias) gegeben werden, der für die Abfrage gültig ist. entsprechend kann auch jeder Tabelle ein Alias vergeben werden.*

### 10.7.1 Kartenspiel

Als Anwendungsbeispiel betrachten wir ein Kartenspiel mit 32 Karten, das die vier Freunde Daniel, Donald, Daisy und Daniel (gleicher Name wie der erste) spielen. Wir möchten die folgenden Blätter in einer Datenbank speichern:

- Daniel hat Herz Ass,
- Daisy hat Pik 7 und Pik Bube,
- Daniel hat Karo Ass.

Unsere Datenbank soll hier generell nur die Blätter für genau ein Spiel speichern können. Bevor wir die Tabellen programmieren, entwerfen wir zunächst das Datenmodell. Offensichtlich gibt es zwei Entitäten, die Spieler und die Karten. Da es sich nur um ein einziges Spiel handelt, kann jede Karte höchstens einem Spieler gehören, aber jeder Spieler kann mehrere Karten haben. Es handelt sich also um eine C-CM-Beziehung (Abbildung 10.3). Sie braucht

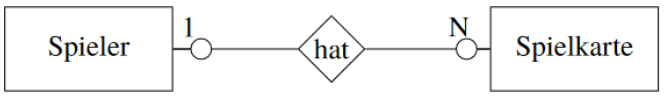
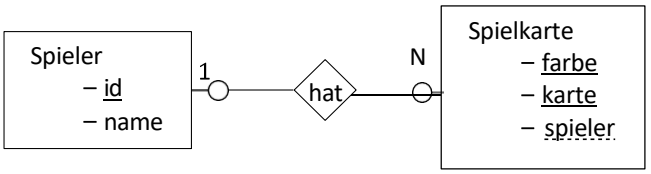



**Abbildung 10.3:** ER-Modell eines Kartenspiels

nicht weiter aufgelöst zu werden, wir können sie direkt als Tabellen implementieren, die die Relationstypen *spieler(id, name)* und *spielkarten(farbe, karte, spieler)* darstellen:




oder etwas kompakter in tabellarischer Form:

| Tabelle | Primärschlüssel | Attribute |
|---|---|---|
| Spieler | id | name |
| Spielkarten | (farbe, karte) | spieler |

Hierbei ist der Primärschlüssel id der Spieler künstlich, da der Name mehrfach auftritt. Entsprechend ist der Fremdschlüssel spielerin dem Relationstyp *spielkarten* der Primärschlüssel desjenigen Spielers, der die Karte hat. Diese Relationstypen können wir in unserer Datenbank wie folgt als Tabellen implementieren:

```
CREATE TABLE spieler (
id SERIAL PRIMARY
   KEY, name
   varchar(20),
);
--
CREATE TABLE spielkarten (
   farbe varchar(5)NOT NULL,
   karte varchar(5)NOT NULL,
   spieler int,
   PRIMARY KEY (farbe, karte),
   FOREIGN KEY (spieler) REFERENCES spieler(id) ON DELETE RESTRICT
);
```

Damit können wir die drei obigen Blätter wie in Tabelle 10.2 speichern. Die folgenden SQL-

**spieler**

| id | name |
|---|---|
| 1 | Daniel |
| 2 | Donald |
| 3 | Daisy |
| 4 | Daniel |

**spielkarten**

| farbe | karte | spieler |
|---|---|---|
| Herz | Ass | 1 |
| Pik | 7 | 3 |
| Pik | Bube | 3 |
| Karo | Ass | 4 |

**Tabelle 10.2:** Daten der Tabellen der Kartenspieldatenbank

Anweisungen speichern die einzelnen Blätter entsprechend in unserer

Datenbank:

```
INSERT INTO spieler (name) VALUES
('Daniel'), ('Donald'), ('Daisy'), ('Daniel');
INSERT INTO spielkarten (farbe, karte) VALUES
('Herz', 'Ass'), ('Pik', '7'), ('Pik', 'Bube'), ('Karo', 'Ass');
```

Um sichnun die Karten von Daisy anzusehen, müssen wir die folgende Abfrage an die Datenbank schicken:

```
SELECT farbe, karte FROM spielkarten WHERE spieler = 2;
```

Das Ergebnis lautet dann:

| farbe | karte |
|---|---|
| Pik | 7 |
| Pik | Bube |

Wie können wir aber das Blatt von Daisy anzeigen lassen, wenn wir ihren Primärschlüssel gar nicht kennen? Dafür müssen wir erst in der Tabelle spielerherausfinden, welche ID sie hat, um dann eine äußere SELECT-Anweisung auf sie auszuführen:

```
SELECT farbe, karte FROM spielkarten WHERE spieler = (
    SELECT id FROM spieler WHERE name='Daisy'
);
```

Diese Abfrage funktioniert allerdings nur, solange der Name Daisy höchstens einmal in der Tabelle spieler vorkommt. Wollen wir entsprechend herausfinden, welche Karten die Spieler namens Daniel haben, so müssen wir auf mehrere Einträge in der Ergebnismenge der Unterabfrage selektieren:

```
SELECT farbe, karte FROM spielkarten WHERE spieler IN (
    SELECT id FROM spieler WHERE name= 'Daniel'
);
```

also IN statt = verwenden.

### 10.7.2 Mehrere Kartenspiele

Betrachten wir nun eine etwas andere Situation, die unser obiges Datenmodell über den Haufen wirft. Nehmen wir an, wir wollen die Kartenverteilungen eines gesamten Spieleabends speichern, also mehrere Spiele. Das heißt, im Gegensatz zu unserem Modell in Abbildung 9.3 kann eine Spielkarte von mehreren Spielern aufgenommen werden. Wir haben also eine CM-CM-

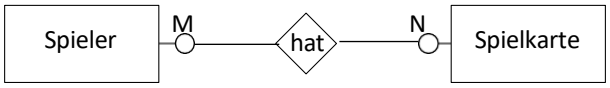



**Abbildung 10.4:** ER-Modell mehrerer Kartenspiele

Beziehung vorliegen. Mit Regel 10 wird sie aufgelöst durch eine Beziehungstabelle und zwei 1-1-Beziehungen (Abbildung 10.5).

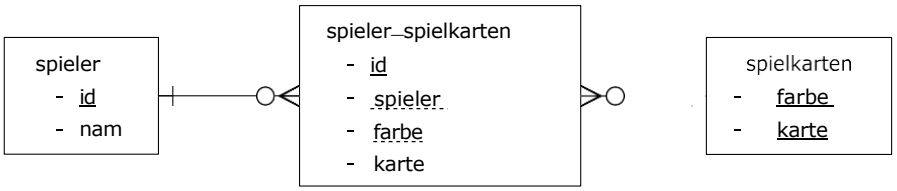



**Abbildung 10.5:** Tabellenmodell zum ER-Modell in Abbildung 9.4

In tabellarischer Form lautet dieses Modell etwas kompakter:

| Tabelle | Primärschlüssel | Attribute |
|---|---|---|
| Spieler | id | name |
| Spielkarten | (farbe, karte) | spieler |
| Spieler_Spielkarten | id | spieler, farbe, karte |

```
CREATE TABLE spieler (
id SERIAL PRIMARY KEY ,
   name varchar(20) NOT NULL
);
--
CREATE TABLE spielkarten (
   farbe varchar(5),
   karte varchar(5) NOT NULL ,
   PRIMARY KEY (farbe, karte),
);
```

Mit den folgenden SQL-Anweisungen werden die vier Spieler*innen, ein vollständiges Skatspiel (32 Karten) und die Beziehungstabelle angelegt:

```
--
CREATE TABLE spieler_spielkarten (
   id SERIAL PRIMARY KEY,
   spieler int NOT NULL,
   farbe varchar(5) NOT NULL ,
   karte varchar(5) NOT NULL ,
   FOREIGN KEY (spieler) REFERENCES spieler(id) ON DELETE RESTRICT ,
   FOREIGN KEY (farbe, karte) REFERENCES spielkarten(farbe, karte) ON DELETE RESTRICT
);
```

Das Einfügen der fünf Karten für Daniel, Daisy und Daniel geschieht durch:

```
INSERT INTO spieler (name) VALUES
 ('Daniel'), ('Donald'), ('Daisy'), ('Daniel');
INSERT INTO spielkarten (farbe, karte) VALUES
 ('Kreuz', '7'), ('Kreuz', '8'), ('Kreuz', '9'), ('Kreuz', '10'),
 ('Kreuz', 'Bube'), ('Kreuz', 'Dame'), ('Kreuz', 'König'), ('Kreuz','Ass'),
 ('Pik', '7'), ('Pik', '8'), ('Pik', '9'), ('Pik', '10'),
 ('Pik', 'Bube'), ('Pik', 'Dame'), ('Pik', 'König'), ('Pik','Ass'),
 ('Herz', '7'), ('Herz', '8'), ('Herz', '9'), ('Herz', '10'),
 ('Herz', 'Bube'), ('Herz', 'Dame'), ('Herz', 'König'), ('Herz', 'Ass'),
 ('Karo', '7'), ('Karo', '8'), ('Karo', '9'), ('Karo', '10'),
 ('Karo', 'Bube'), ('Karo', 'Dame'), ('Karo', 'König'), ('Karo', 'Ass');
 INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES
 (1,  'Herz',   'Ass'),
 (3,  'Pik',    '7'),
 (3,  'Pik',    'Bube'),
 (4,  'Karo',   'Ass');
```

Weitere Karten könnten an die drei verteilt werden mit mehr INSERTs:

```
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (1, 'Herz', 'Ass');
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (2, 'Kreuz', 'Dame');
INSERT INTO spieler_spielkarten (spieler, farbe, karte) VALUES (4, 'Pik','7');
```

Um die Karten anzeigen zu lassen, die Daisy im Laufe des Spieleabends auf der Hand hatte, müssen wir nun aus der Beziehungstabelle selektieren:

```
SELECT farbe, karte FROM spieler_spielkarten WHERE spieler = (
   SELECT id FROM spieler WHERE name='Daisy'
);
```

Entsprechend erhält man die Karten, die die Spieler mit Namen Daniel hatten, mit der Anweisung:

```
SELECT farbe, karte FROM spieler_spielkarten WHERE spieler IN (
   SELECT id FROM spieler WHERE name='Daniel'
);
```

„Nach außen“ unterscheiden sich die Versionen 1-CM und CM-CM also nicht, das „interne“ Tabellenmodell allerdings ist komplexer geworden. Sollten Sie im Ihrer späteren Laufbahn an einem Projekt beteiligt sein, in dem Daten zu modellieren sind — sei es als Anwender oder als Entwickler —, so wissen Sie, worauf Sie zu achten haben: Fragen Sie lieber einmal zu viel nach, ob es sich bei einer konkreten Datenkonstellation wirklich um eine 1:N oder um eine M:N Beziehung handelt. Allein der kleine Austausch von 1 durch M kann im schlimmsten Fall eine ganze Datenbank für den praktischen Einsatz unbrauchbar machen.