Relationale Datenbanken

Literatur:

DUBOIS, P.: *MySQL*, New Riders Publishing, 1999.
HEUER, A.; SAAKE, G.; SATTLER, K.-U.: *Datenbanken - kompakt*, Bonn: mitp-Verlag, 2001.
KLINE, K.; KLINE, D.: *SQL in a Nutshell*, Köln: O'Reilly, 2001.
KOFLER, M.: *MySQL. Einführung, Programmierung, Referenz*, Bonn u.a.: Addison-Wesley, 2001.
RRZN/UNIVERSITÄT HANNOVER (Hrsg.): *SQL. Grundlagen und Datenbankdesign*, 2002.
RRZN/UNIVERSITÄT HANNOVER (Hrsg.): *Access 2000. Grundlagen für Anwender*, 2000.

Das relationale Datenmodell ist heute das bei weitem wichtigste. Gängige DBMS (Datenbankmanagementsysteme) unterstützen in der Regel dieses Datenmodell. Die grundlegenden Arbeiten aus dem Jahre 1970 zum relationalen Datenmodell stammen von E.F. CODD.

Im relationalen Datenmodell werden die Daten der Realwelt als eine Menge von **Relationen** abgebildet. Relationen stellen Beziehungen zwischen den Elementen einer Menge oder zwischen den Elementen mehrerer Mengen her.

Mathematische Definition einer Relation:

Sind $D_1, D_2, ..., D_n$ Mengen von Werten, so ist
$R \subseteq D_1 \times D_2 \times ... \times D_n$ eine n-stellige Relation über den Mengen $D_1, D_2, ..., D_n$,
n ist der Grad der Relation.

$D_1 \times D_2 \times ... \times D_n$ steht für das kartesische Produkt.

Ein Element $r = (d_1, d_2, ..., d_n) \in R$ ($d_i \in D_i$, i = 1, ..., n) ist ein **Tupel** der Relation R (n-Tupel).
d_i ist die i-te Komponente des Tupels.

Bsp.: D_1 sei die Menge {Meier, Schmidt, Müller}, D_2 die Menge {m,w}.
Kartesisches Produkt $D_1 \times D_2$ = {(Meier,m), (Meier,w), (Schmidt,m), Schmidt,w), (Müller,m), (Müller,w)}.
Jede Teilmenge dieser Menge ist eine zweistellige Relation, etwa
R = {(Meier,m), (Schmidt,w), (Müller,w)}, aber auch die leere Menge.

Ein **Tupel** einer Relation entspricht einem Objekt (oder Entity, vgl. Entity-Relationship-Modell), d.h. einem "Datensatz" in einer Datenbank. Ein solches Tupel ist zum Beispiel Personalnummer, Name, Vorname, Geschlecht, Beruf, Wohnort und Gehalt des Angestellten Fritz Müller.

Natürlich verändert sich die Anzahl der Tupel in einer Relation über die Zeit. Z.B. kommt in einer Adressdatei der Eintrag eines neuen Mitglieds hinzu. Die Werte innerhalb eines Tupels können sich auch verändern, z.B. durch Adressänderung in einer Datenbank.
Ein **Relationsschema** R(A1, ..., An) definiert eine Relation mit dem Namen R und mit den Attributen A1, ..., An. Jedem Attribut Ai ist ein Wertebereich dom(Ai) zugeordnet. Die zu R gehörenden Relationen sind also sämtliche Relationen des Typs $r \subseteq (dom(A1) \times dom(A2) \times ... \times dom(An))$. In einer Datenbank existiert zu jedem Zeitpunkt genau eine Relation r zum Relationsschema R, nämlich diejenige mit den gerade gültigen Werten (entsprechend den zu diesem Zeitpunkt existierenden Entities).

Ein Relationsschema für das obige Beispiel wäre Personen(Name, Geschlecht), wobei Name der Wertebereich dom(A1) = STRING (Folge von Zeichen) und Geschlecht der Wertebereich dom(A2) = {w,m} zugeordnet ist. Sprachlich wird oft nicht genau differenziert zwischen Relation und Relationsschema.

Eine Relation kann man sich anschaulich als eine **Tabelle** vorstellen. Jede Zeile der Tabelle entspricht einem Tupel der Relation. Die Spalten der Tabelle werden nach den Attributnamen benannt, im Beispiel "Name" und "Geschlecht".

Die folgende Tabelle zeigt eine Relation zum Relationsschema Angestellter(Name, Vorname, Geschlecht, Wohnort, Gehalt).

Name	Vorname	Geschlecht	Wohnort	Gehalt
Müller	Martin	m	Freiburg	60500
Schmidt	Susanne	w	Endigen	35000
Müller	Luise	w	Freiburg	72000

Jedes Attribut einer Relation muss **elementar** sein. D.h. der Wert eines Attributes ist unteilbar und setzt sich nicht aus mehreren anderen Werten zusammen (vgl. 1. Normalform). Z.B. sollte im Feld Vornamen nicht auch das Geschlecht mit aufgenommen werden. Die **Tupel** einer Relation bzw. die Zeilen einer Tabelle sind **eindeutig**, d.h. es dürfen keine gleichen Zeilen auftreten.

Für jede Relation gibt es eine Menge von Attributen, mit deren Hilfe sich eine Zeile einer Tabelle genau identifizieren lässt. Solche Attributmengen nennt man **Schlüssel**. Ein Schlüssel sollte möglichst minimal sein. Beispielsweise ist die Personalnummer ein Schlüssel mit nur einem Attribut. Die Attributsmenge {Name, Vorname, Geburtsdatum} wäre ein Schlüssel mit drei Attributen. Eine Relation kann mehrere Schlüssel enthalten. Z.B. kann der Angestellte über den Schlüssel Personalnummer, aber auch über {Name, Vorname, Geburtsdatum} identifiziert werden. In solchen Fällen wird meist ein Schlüssel ausgezeichnet und zum **Primärschlüssel** erklärt.

Arbeiten mit relationalen Datenbanken (Operationen auf Tabellen)

Ebenso wie alle Daten einer relationalen Datenbank als Relationen dargestellt werden, ist auch das Ergebnis einer **Abfrage** (Query) wieder eine Relation und wird auch wieder als Tabelle dargestellt. DBMS für relationale Datenbanken müssen Operationen zur Verfügung stellen, mit deren Hilfe aus vorhandenen Relationen die gewünschten Relationen abgeleitet werden können. Es müssen Operationen vorhanden sein, mit denen man eine Menge von Tupel aus einer Relation filtern kann. Bei einer solchen **Selektion** werden also im Prinzip bestimmte Zeilen einer Tabelle in eine neue kleinere Tabelle kopiert.

Bei einer **Projektion** werden Spalten gestrichen und es entsteht eine Relation kleineren Grades, d.h. eine Tabelle mit weniger Spalten. Man kann eine Selektion und eine Projektion auch kombinieren.

Machen Sie sich die Operationen anhand der folgenden Tabelle klar:

Mitarbeiter	Straße	PLZ	Ort	Telefon	Durchwahl	Abteilung
AK-0589	Emmendingerstr. 18	79123	Freiburg	(0761)4567	-123	A-1827
DS-0191	Kunzenweg 23	79100	March	(07664)7890	-345	B-271
HM-0188	Hauptstraße 5	79364	Müllheim	(07644)123	-12	A-1021
KE-1189	Kunzenweg 23	79100	March	(07664)7890	-234	A-1723
KK-0888	Emmendingerstr. 18	79123	Freiburg	(0761)4567	-111	C-163
LK-0990	Wechselgasse 3	79115	Freiburg	(0761)321	-13	A-0792
MH-0590	Hauptstraße 5	79364	Müllheim	(07644)123	-23	E-91
ML-1290	Kunzenweg 23	79100	March	(07664)7890	-231	B-16
WE-0589	Emmendingerstr. 18	79123	Freiburg	(0761)4567	-211	B-1341

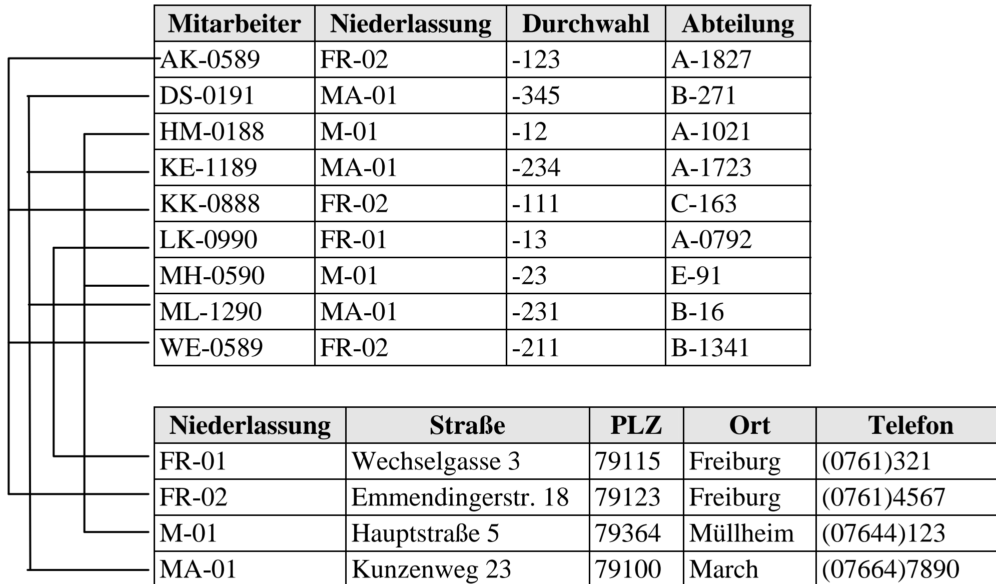
Durch Kombination von Relationen (**Verbund**) werden Relationen größeren Grades erzeugt. Bei einer Verbundoperation werden zwei Tabellen mit mindestens einem gemeinsamen Attribut verbunden.

Beispiel:

Mitarbeiter	Niederlassung	Durchwahl	Abteilung
AK-0589	FR-02	-123	A-1827
DS-0191	MA-01	-345	B-271
HM-0188	M-01	-12	A-1021
KE-1189	MA-01	-234	A-1723
KK-0888	FR-02	-111	C-163
LK-0990	FR-01	-13	A-0792
MH-0590	M-01	-23	E-91
ML-1290	MA-01	-231	B-16
WE-0589	FR-02	-211	B-1341

Niederlassung	Straße	PLZ	Ort	Telefon
FR-01	Wechselgasse 3	79115	Freiburg	(0761)321
FR-02	Emmendingerstr. 18	79123	Freiburg	(0761)4567
M-01	Hauptstraße 5	79364	Müllheim	(07644)123
MA-01	Kunzenweg 23	79100	March	(07664)7890

Im Beispiel werden die Daten der einzelnen Niederlassungen nicht in die Tabelle der Mitarbeiterdaten integriert. Es werden zwei getrennte Tabellen angelegt. Die Tabelle der Mitarbeiter enthält ein Kürzel, das Aufschluss über die Niederlassung gibt in der der Mitarbeiter arbeitet. Das gleiche Kürzel findet sich auch in der Tabelle der Niederlassungen. Die beiden Tabellen werden zu einem **Verbund** verknüpft. Im Beispiel wird jeder Mitarbeiterzeile die Zeile aus der Tabelle der Niederlassungen zugeordnet, die das gleiche Niederlassungskürzel besitzt.



Mitarbeiter	Niederlassung	Durchwahl	Abteilung
AK-0589	FR-02	-123	A-1827
DS-0191	MA-01	-345	B-271
HM-0188	M-01	-12	A-1021
KE-1189	MA-01	-234	A-1723
KK-0888	FR-02	-111	C-163
LK-0990	FR-01	-13	A-0792
MH-0590	M-01	-23	E-91
ML-1290	MA-01	-231	B-16
WE-0589	FR-02	-211	B-1341

Niederlassung	Straße	PLZ	Ort	Telefon
FR-01	Wechselgasse 3	79115	Freiburg	(0761)321
FR-02	Emmendingerstr. 18	79123	Freiburg	(0761)4567
M-01	Hauptstraße 5	79364	Müllheim	(07644)123
MA-01	Kunzenweg 23	79100	March	(07664)7890

Die Möglichkeit relationaler Datenbanken Tabellen zu verknüpfen hat folgende Vorteile:

Keine Redundanz
Bestimmte Informationen müssen nicht mehrfach gespeichert werden.

Weniger Eingabeaufwand
Im Beispiel muss die Adresse einer Niederlassung nur einmal eingegeben werden.

Weniger Speicherplatzbedarf
Im Beispiel benötigt jede neue Erfassung einer Niederlassungsadresse den gleichen Speicherplatzbedarf.

Weniger Pflegeaufwand
Im Beispiel muss bei einer Adressänderung für eine Niederlassung die Änderung nur einmal durchgeführt werden.

Vermeidung eventueller Integritätsverletzungen
Im Beispiel könnten sich bei der Eingabe der Adressen in nur eine Tabelle durch Tippfehler Integritätsverletzungen der Daten einschleichen: Eigentlich identische Informationen, wie zum Beispiel der Straßennamen einer Niederlassung könnten in einzelnen Fällen nicht mehr übereinstimmen.

Datenmodellierung – Verknüpfen, aber wie?

Für die Funktionalität einer relationalen Datenbank ist es ganz entscheidend, welche Daten in welcher Tabelle gespeichert werden und wie die Tabellen untereinander verknüpft werden. Die Aufteilung der Daten in verschiedene Tabellen und die Erstellung der notwendigen Verknüpfungen zwischen den Tabellen einer Datenbank wird als **Datenmodellierung** bezeichnet.

Werden zwei Tabellen miteinander verknüpft, müssen **Schlüssel** (s.o.) verwendet werden, wobei zwei Typen unterschieden werden: Primärschlüssel und Fremdschlüssel.

Primärschlüssel

Der Primärschlüssel einer Tabelle kennzeichnet jede Tabelle eindeutig. Im obigen Beispiel wäre die Nummer der Mitarbeiter ein Primärschlüssel. Beim Anlegen eines Primärschlüssels müssen einige Punkte beachtet werden:

- Ein Primärschlüssel kann sich aus einer oder aus mehreren Spalten zusammensetzen. Er sollte sich in der Regel aus so wenig Spalten wie möglich zusammensetzen und sich so wenig wie möglich ändern.

Im Datenbankprogramm Access werden häufig so genannte „AutoWerte", eine fortlaufende Nummer, für den Primärschlüssel verwendet. AutoWerte haben allerdings den Nachteil, dass sie wenig aussagekräftig für den Nutzer sind.

- Da mit Hilfe des Primärschlüssels jeder Datensatz eindeutig identifiziert werden muss, dürfen die Werte in den Primärschlüsselspalten nicht doppelt vorkommen.
- Jede Tabelle kann nur einen Primärschlüssel besitzen.
- Ein Primärschlüssel darf nicht leer (NULL) sein. Mit NULL-Wert werden undefinierte Feldeinträge bezeichnet, z. B. wenn nichts eingetragen wurde. NULL-Werte sollten nicht mit der numerischen Null (0) bzw. einem Leerstring („") verwechselt werden.

Fremdschlüssel

Fremdschlüssel sind Spalten in einer Tabelle, die auf den Primärschlüssel einer anderen Tabelle verweisen. Im obigen Beispiel ist in der Tabelle der Mitarbeiter die Spalte Niederlassung ein Fremdschlüssel. Die Spalte Niederlassung ist in der Tabelle der Niederlassungen natürlich Primärschlüssel. Für Fremdschlüssel sollten folgende Regeln beachtet werden:

- Eine Tabelle kann mehrere Fremdschlüssel enthalten.
- Auch ein Fremdschlüssel kann sich auf mehrere Spalten einer Tabelle verteilen.
- Der Fremdschlüssel einer Tabelle muss gleich dem Primärschlüssel einer anderen Tabelle sein. Jeder Wert eines Fremdschlüssels muss in der verknüpften Tabelle im Primärschlüssel vorhanden sein.
- Ein Fremdschlüssel darf im Unterschied zu einem Primärschlüssel NULL sein.
- Der Wert eines zusammengesetzten Fremdschlüssels ist NULL, wenn eine der Spalten NULL ist.

Beziehungen

Eine Verknüpfung zwischen Spalten einer Tabelle führt zu einer Beziehung zwischen diesen Tabellen. Tabellen können auf drei unterschiedliche Arten miteinander verknüpft werden: Eins zu viele (1:n), eins zu eins (1:1), und viele zu viele (m:n).

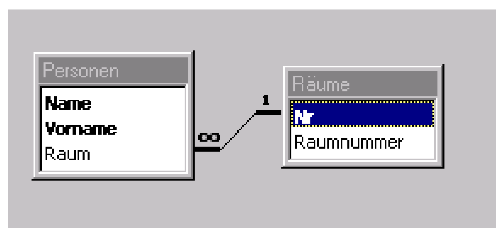
1:n-Beziehung

Tabellen sind mit einer 1:n-Beziehung miteinander verbunden, wenn es zu jeder Zeile der einen Tabelle in der zweiten Tabelle keine, eine oder mehrere Zeilen gibt. Im obigen Beispiel ist die Tabelle der Niederlassungen mit der Tabelle der Mitarbeiter mit einer 1:n-Beziehung verknüpft. Im Beispiel gibt es zu jeder Zeile der Tabelle der Niederlassungen eine oder mehrere zugeordnete Zeilen der Tabelle der Mitarbeiter. Es wäre auch möglich, dass eine neue Niederlassung noch keinem Mitarbeiter zugeordnet ist.

Das Feld der „1"-Seite einer 1:n-Beziehung muss ein Primärschlüssel sein. Das Feld der „n"-Seite ist ein Fremdschlüssel.

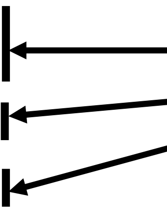
Die meisten Verknüpfungen in relationalen Datenbanken sind 1:n-Beziehungen.

Ein weiteres Beispiel für eine 1:n-Beziehung sind zwei Tabellen in denen die Belegung von Räumen festgehalten wird.



Personen : Tabelle

Name	Vorname	Raum
Baumann	Johanna	1
Maier	Klaus	1
May	Karla	1
Müller	Fritz	2
Müller	Klaus	2
Naumann	Kathrin	3
Schmidt	Markus	3



Räume : Tabelle

Nr	Raumnummer
1	A20
2	A64
3	B17

In Access wird nur dann eine 1:n-Beziehung hergestellt, wenn nur eines der in Beziehung stehenden Felder Primärschlüssel ist oder einen eindeutigen Index (vgl. „Indizieren von Tabellen“, S. 6) besitzt.

1:1-Beziehung

Bei einer 1:1-Beziehung existiert zu jedem Datensatz in der ersten Tabelle genau ein Datensatz in der zweiten Tabelle. Eigentlich könnte eine solche Beziehung auch aufgelöst werden und in nur einer Tabelle gespeichert werden. 1:1-Beziehungen kommen daher auch nicht sehr häufig vor. Es gibt allerdings Situationen, in denen diese Form der Beziehung sinnvoll ist:

- Sicherheitsaspekte (Vertrauliche Daten werden in einer separaten Tabelle gespeichert.)
- Performance (Selten benötigte Daten werden in einer zweiten Tabelle ausgelagert.)

Im Beispiel werden vertrauliche Mitarbeiterdaten in einer Tabelle gespeichert, auf die nicht jeder Mitarbeiter Zugriffsrechte haben soll, der auch auf die Tabelle 1 Zugriff hat.

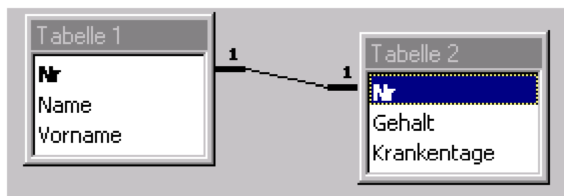


Tabelle 1 : Tabelle

Nr	Name	Vorname
1	Naumann	Kathrin
2	Müller	Fritz
3	Maier	Klaus
4	Schmidt	Markus
5	Müller	Klaus
6	Baumann	Johanna
7	May	Karla

Tabelle 2 : Tabelle

Nr	Gehalt	Krankent
1	5.367,00 DM	12
2	2.345,00 DM	2
3	3.456,00 DM	3
4	2.134,00 DM	0
5	3.456,00 DM	5
6	6.543,00 DM	6
7	4.123,00 DM	1

In Access wird nur dann eine 1:1-Beziehung hergestellt, wenn beide in Beziehung stehenden Felder Primärschlüssel sind oder über eindeutige Indizes verfügen.

n:m-Beziehung

In einer n:m-Beziehung können jedem Datensatz aus der einen Tabelle mehrere passende Datensätze aus der zweiten Tabelle zugeordnet sein und umgekehrt. Dies ist nur möglich, indem eine dritte Tabelle definiert wird, deren Primärschlüssel aus zwei Feldern besteht, und zwar den Fremdschlüsseln aus den beiden anderen Tabellen. Eine solche Tabelle wird Verbindungstabelle genannt. Eine n:m-Beziehung besteht eigentlich aus zwei 1:n-Beziehungen.

Personen : Tabelle

Name	Vorname	Nr
Naumann	Kathrin	1
Müller	Fritz	2
Maier	Klaus	3
Schmidt	Markus	4
Müller	Klaus	5
Baumann	Johanna	6

Hilfstabelle : T

P-Nr	V-Nr
1	1
1	3
2	1
2	2
2	3
3	3
6	1

Projekte : Tabelle

Nr	Name
1	Pro1
2	Orga2
3	Web-Site

Im Beispiel kann eine Person mit Hilfe der Hilfstabelle Mitglied in mehreren Projekten sein.

Referentielle Integrität

Referentielle Integrität ist ein Regelsystem, mit dessen Hilfe ein Datenbankprogramm versucht sicherzustellen, dass Beziehungen zwischen Datensätzen in Tabellen gültig bleiben und dass verknüpfte Daten nicht versehentlich gelöscht oder geändert werden. Bei der Anwendung der referentiellen Integrität hält sich ein Datenbanksystem an folgende Regeln:

1. Es können keine Datensätze auf der „n“-Seite einer Beziehung angelegt werden, wenn nicht auf der „1“-Seite ein entsprechender Datensatz existiert.
   In Access kann man aber auf der „n“-Seite einen NULL-Wert eingeben und damit angeben, dass die Datensätze nicht miteinander verknüpft sind. Dazu gibt man einfach nichts in das Feld ein.
2. Werte im Primärschlüsselfeld einer „1“-Tabelle können nicht geändert werden, wenn in der „n“-Tabelle noch Datensätze vorhanden sind, die mit diesem Primärschlüssel in Beziehung stehen.
3. Versucht der Benutzer Datensätze der „1“-Tabelle zu löschen, wird geprüft, ob sich noch ein zugehöriger Datensatz in der „n“-Tabelle befindet. Nur wenn das nicht der Fall ist, kann der „1“-Datensatz gelöscht werden.

In Access wird referentielle Integrität bei der Vereinbarung von Beziehungen durch Aktivieren des Kontrollkästchens „Mit referentieller Integrität“ aktiviert. Die Einschränkungen beim Löschen bzw. Ändern von Datensätzen können in Access zum Teil außer Kraft gesetzt werden, ohne dass die Regeln der referentiellen Integrität verletzt werden, indem die Kontrollkästchen „Aktualisierungsweitergabe an Detailfeld“ und „Löschweitergabe an Detaildatensatz“ aktiviert werden.

Normalerweise werden Änderungen des Primärschlüssels eines Datensatzes der „1-Tabelle“ durch die referentielle Integrität verhindert (vgl. 2.). Wenn das Kontrollkästchen „Aktualisierungsweitergabe an Detailfeld“ aktiviert ist, wird beim Ändern eines Primärschlüsselwerts auf der „1“-Seite einer Beziehung der damit übereinstimmende Wert in allen Datensätzen auf der „n“-Seite automatisch aktualisiert.

Auch das Löschen von Datensätzen in der „1-Tabelle“ wird standardmäßig verhindert (vgl. 3.). Wird allerdings das Kontrollkästchen „Löschweitergabe an Detaildatensatz“ aktiviert, werden beim Löschen eines Datensatzes auf der „1“-Seite alle Datensätze auf der „n“-Seite automatisch gelöscht.

Indizieren von Tabellen

Man kann Tabellenspalten mit einem **Index** versehen. Dadurch können Such- und Sortieroperationen drastisch beschleunigt werden. Für einen Index legt ein Datenbanksystem eine interne Hilfstabelle an, in der nur die Werte der Indexspalte und ihre Position in der ursprünglichen Tabelle definiert sind. Die Einträge der Tabellenspalten werden in der Hilfstabelle in einem Binärbaum angelegt, der enorm schnelle Suchoperationen auf die entsprechende Spalte der Tabelle erlaubt. Wenn der Benutzer später einen bestimmten Wert dieser Spalte suchen lässt, oder eine Tabelle nach einer indizierten Spalte sortieren lässt, muss das Datenbanksystem nur die Hilfstabelle durchsuchen oder sortieren, nicht die gesamte Tabelle.

Das Anlegen eines Index kann in größeren Tabellen relativ viel Zeit in Anspruch nehmen. Außerdem benötigt ein Index zusätzlichen Speicherplatz in einer Datenbank. Daher sollte man nur für die Spalten einen Index anlegen, deren Inhalt in einer Abfrage sehr häufig als Suchkriterium herangezogen wird.

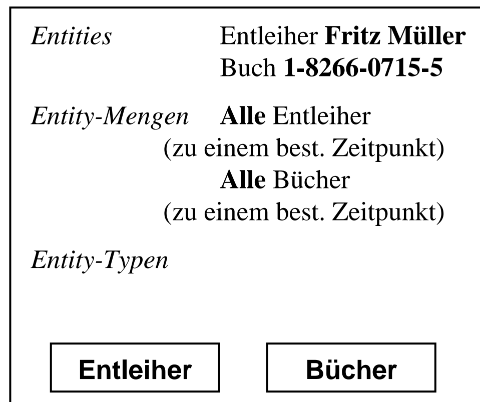
In Access wird für einen Primärschlüssel automatisch ein Index definiert. In der Entwurfsansicht einer Tabelle können Sie auch weitere Indizes vereinbaren. Dabei können Indizes definiert werden, die Duplikate zulassen („Duplikate möglich“) oder nicht („Ohne Duplikate“). Wenn man einen Index ohne Duplikate definiert ist der Index einmalig (unique index) und damit auch ein **Schlüssel**. Solche Indizes werden oft auch als **Sekundärschlüssel** bezeichnet.

Konzeptioneller Entwurf einer Datenbank mit dem Entity-Relationship-Modell

Das **Entity-Relationship-Modell (ER-Modell)** hat sich als Standardmodell für frühe konzeptionelle Entwurfsphasen der Datenbankentwicklung durchgesetzt. Das ER-Modell geht auf einen grundlegenden Artikel von P. CHEN[1] von 1976 zurück. Es bildet alle Elemente einer Datenbank ab und stellt deren Beziehungen untereinander dar.

Das ER-Modell basiert auf den drei grundlegenden Begriffen *Entity*, *Attribut* und *Beziehung*:

1. Entity, Entity-Menge, Entity-Typ



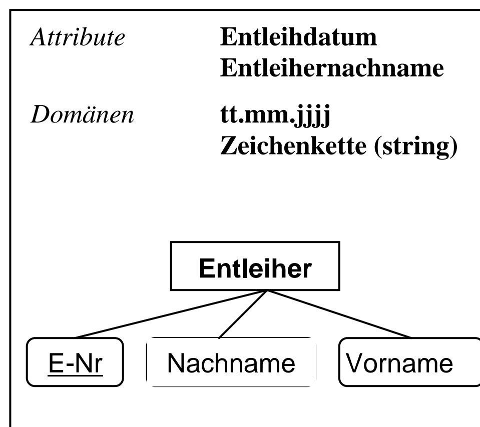
Ein *Entity* ist ein Objekt der realen oder der Vorstellungswelt, über das in einer Datenbank Informationen gespeichert werden sollen, z.B. ein Buch in einer Bibliothek, der Autor eines Buches oder der Entleiher eines Buches. Auch Informationen über Ereignisse, wie z.B. das Entleihen eines Buches, können Objekte im Sinne des ER-Modells sein. Ein Entity entspricht einem Datensatz bzw. einer Zeile in einer Tabelle.

Eine *Entity-Menge* ist eine Sammlung von Entities mit gleichartigen Eigenschaften zu einem bestimmten Zeitpunkt. Die Entity-Menge entspricht einer aktuellen Tabelle einer Datenbank. Das Buch „Die Zahl Pi „ oder die Autorin Sabine Mustermann in einer Bibliotheksdatenbank sind beispielsweise bestimmte Entities. Alle Bücher der Bibliothek und alle Autoren zu einem bestimmten Zeitpunkt sind die zugehörigen Entity-Mengen. Entity-Mengen sind zeitlich veränderlich.

Der *Entity-Typ* kategorisiert gleichartige Entities. Zu einem Entity-Typ gehören Entities, die sich durch die gleichen Eigenschaften (Attribute) charakterisieren lassen.

Entity-Typen werden in Form von Rechtecken grafisch dargestellt.

2. Attribut (Eigenschaft), Domäne, Schlüssel und Primärschlüssel



Ein *Attribut* repräsentiert eine Eigenschaft von Entities oder Beziehungen (Relationships), z.B. den Namen eines Buchautors, den Titel eines Buches oder das Datum eines Entleihvorgangs. Attribute besitzen einen Namen (z.B. Autorenname) und einen Wert (z.B. Müller).

Eine *Domäne* beschreibt den zulässigen Wertebereich eines Attributs. Beispielsweise könnte der Wertebereich des Attributs Erscheinungsjahr eines Buches zwischen 1900 und dem aktuellem Jahr des Systemdatums liegen.

Schlüssel[2] ermöglichen die eindeutige Identifizierung eines Entitys. Dabei darf ein Attribut oder eine Attributskombination eines Schlüssels in einer Entity-Menge (Tabelle) nur ein einziges Mal vorkommen. Falls sich dies mit den vorhandenen Attributen nicht realisieren lässt, wird oft ein künstliches Attribut hinzugefügt, z.B. ein Zählfeld oder eine Identifikationsnummer. Falls es mehr als ein Schlüssel gibt, kann man einen als Primärschlüssel auszeichnen.

Die Attribute von Entities werden als abgerundete mit ihnen verbundene Rechtecke dargestellt. Attribute, die Primärschlüssel sind, werden unterstrichen.

---

[1] CHEN, P. (1976): The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Band 1, Nr. 1, 9-36

[2] vgl. S. 2 und 3, f

3. *Beziehung (Relationship), Beziehungsmenge, Beziehungstyp, Kardinalität, rekursive Beziehung, Part-of-Beziehung, Is-a-Beziehung*

Durch *Beziehungen* werden die Wechselwirkungen und Abhängigkeiten von Entities beschrieben. Z.B. leiht sich der Entleiher Max Schmidt[3] ein bestimmtes Buch.

Eine *Beziehungsmenge* ist eine Sammlung von Beziehungen gleicher Art zur Verknüpfung von Entity-Mengen (Tabellen).

Ein *Beziehungstyp* ist, analog zum Entity-Typ, die Abstraktion gleichartiger Beziehungen, z.B. „Entleiher *hat geliehen* Buch“ oder „Autor *ist Autor von* Buch“.

Beziehungen zwischen Entities werden als Rauten gezeichnet, die durch Striche mit den entsprechenden Entities verbunden sind. In die Raute schreibt man i.d.R. den Namen der Beziehung.

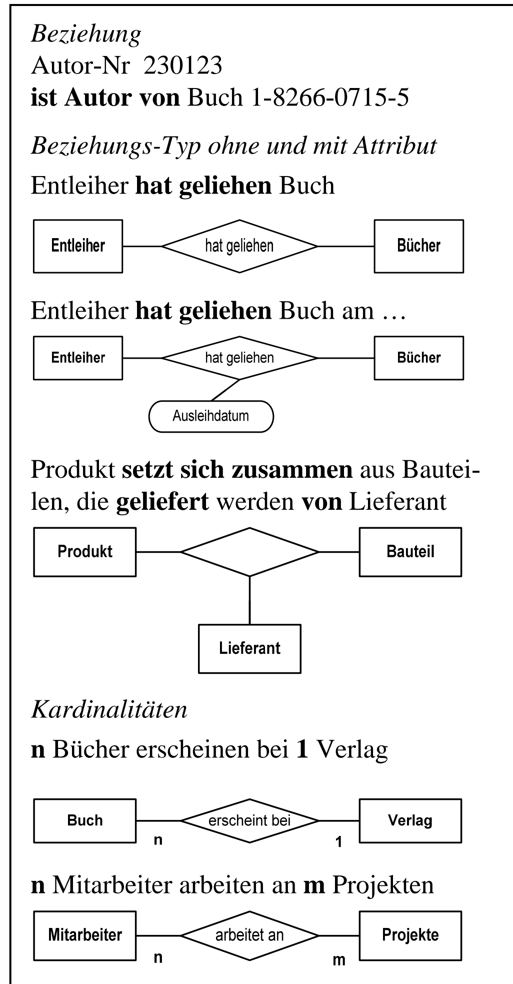
Beziehungen können auch durch Attribute beschrieben werden. Z.B. kann für ein geliehenes Buch das Ausleihdatum mit aufgenommen werden.

In der Regel stehen im ER-Modell zwei Entity-Typen in Beziehung zueinander. Es können aber auch mehrere Entity-Typen zueinander in Beziehung gesetzt werden. Im nebenstehenden Beispiel wird ein Produkt beschrieben, das aus mehreren Bauteilen verschiedener Lieferanten besteht.

Mit Hilfe der *Kardinalität* kann man festlegen, wie viele Entities einer Entity-Menge mit Entities einer anderen Entity-Menge in Beziehung stehen können:

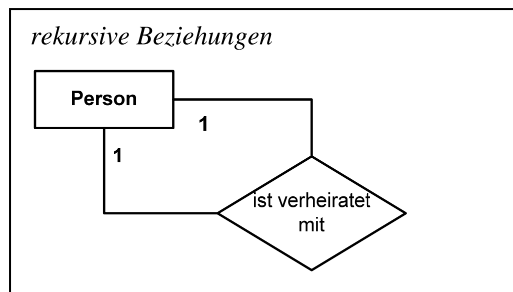
- 1 genau eine Zuordnung
- n, m eine oder mehrere Zuordnungen

Daraus ergeben sich die schon bekannten drei Möglichkeiten für die Darstellung von Beziehungen:



1:1-Beziehung	Jedem Entity einer Entity-Menge ist genau ein Entity einer anderen Entity-Menge zugeordnet.
1:n-Beziehung	Jedem Entity einer Entity-Menge ist ein Entity oder sind mehrere Entities einer anderen Entity-Menge zugeordnet.
n:m-Beziehung	Einem Entity oder mehreren Entities einer Entity-Menge können ein Entity oder mehrere Entities einer anderen Entity-Menge zugeordnet werden.

Für die *Kardinalität* kann auch eine Zahl festgelegt werden, wenn die Anzahl der Entities, die mit einer Entity-Menge in Beziehung stehen, immer gleich ist. Ein Beispiel wäre die Entity-Menge der PKWs, der immer vier Elemente der Menge der Räder zugeordnet sind. (vgl. Abb. Seite 9 zu part-of-Beziehungen)



Bei *rekursiven Beziehungen* besitzt ein Entity-Typ eine Assoziation auf sich selbst. Z.B. kann eine rekursive Beziehung zwischen Personen bestehen, die mit einer anderen Person verheiratet sind. Oder zwischen Bauteilen, die wiederum aus anderen Bauteilen zusammengesetzt sind.

[3] Bei diesem Beispiel darf es in der Datenbank keinen zweiten Max Schmidt geben. Besser wäre es, einen einstelligen Primärschlüssel mit einer Entleiher-Nummer in die Tabelle aufzunehmen.

Das Entity-Relationship-Modell nach CHEN wurde später um die Konzepte der *Aggregation* und der *Generalisierung/Spezialisierung* erweitert.

Eine *Part-of-Beziehung* (Ist-Teil-von-Beziehung) entspricht dem Konzept der *Aggregation*. Beispielsweise hat ein PKW einen Motor, vier Räder, mehrere Sitze usw.

Bei einer *Is-a-Beziehung* (Ist-Beziehung) kann es sich um eine *Generalisierung* (Verallgemeinerung) oder eine *Spezialisierung* handeln. Z.B. können die Entity-Typen **PKW**, **LKW** und **Fahrrad** in einem Entity-Typ **Fahrzeug** zusammengefasst werden. An Stelle dieser *Generalisierung* kann man auch **PKW**, **LKW** und **Fahrrad** als *Spezialisierung* des Entity-Typs **Fahrzeug** auffassen.

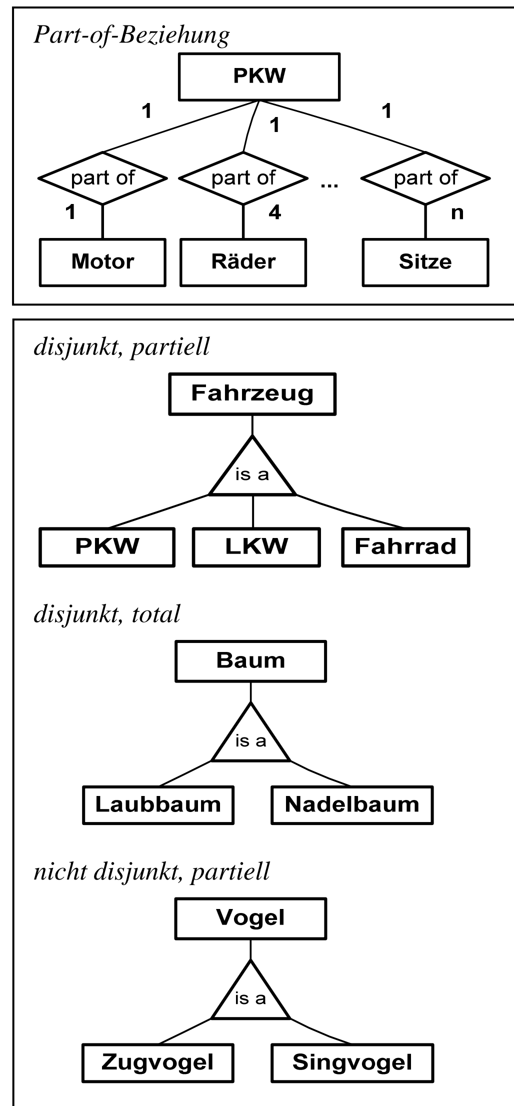
Dabei handelt es sich um eine Teilmengenbeziehung, die man in der Mengenschreibweise wie folgt ausdrücken kann:

PKW $\subseteq$ Fahrzeug
LKW $\subseteq$ Fahrzeug
Fahrrad $\subseteq$ Fahrzeug

Eine *Is-a-Beziehung* ist entweder *disjunkt* oder *nicht disjunkt*. Sie ist entweder *total* oder *partiell*:

disjunkt	Alle Teilmengen sind echte Teilmengen. Kein Element der einen Teilmenge kommt in der anderen vor.
nicht disjunkt	Die Teilmengen können gemeinsame Elemente enthalten.

total	Es gibt keine weiteren Teilmengen.
partiell	Es gibt weitere Teilmengen, die aber nicht aufgeführt sind.



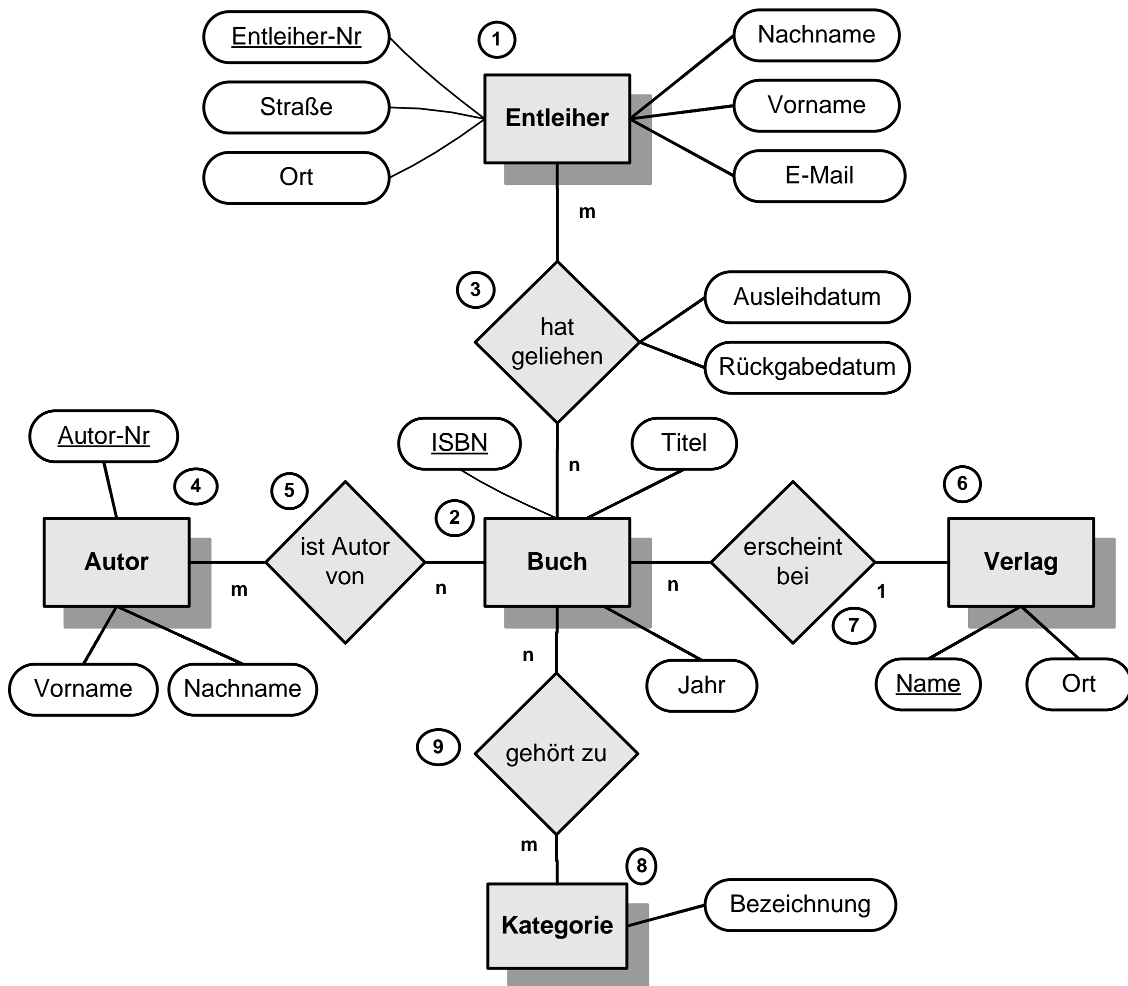
Das folgende einfache Beispiel zeigt das Diagramm eines ER-Modell für eine Bibliothek. Objekte der modellierten Anwendung sind *Entleiher*, *Bücher*, *Autoren, Verlage* und *Kategorien.* In der Regel wird man das ER-Modell nach der Top-Down-Methode entwerfen und das grob entworfene Modell schrittweise verfeinern. Zuerst wird man die Entities *Entleiher*, *Buch*, *Autor, Verlag* und *Kategorie* identifizieren und sich danach ihre Attribute und Beziehungen erarbeiten. Wesentlich ist auch die Festlegung von benötigten Primärschlüsseln für die Entities.

① Im Entity-Typ *Entleiher* sind alle Informationen zu den Entleihern der Bibliothek abgelegt. Für sie wurde ein zusätzliches Attribut Entleiher-Nr aufgenommen, da aus den anderen Attributen nicht mit Sicherheit ein Schlüssel gebildet werden kann. Die E-Mail-Adresse wäre nur dann ein Schlüssel, wenn sicher wäre, dass jeder Entleiher eine Mail-Adresse hat.

② Beim Entity-Typ *Buch* wird als Primärschlüssel die ISBN-Nummer verwendet. Falls es in der Bibliothek Mehrfachexemplare geben würde, müsste die Standnummer des Buches mit aufgenommen werden und diese als Primärschlüssel verwendet werden. Für jedes Buch soll auch der Titel und das Erscheinungsjahr gespeichert werden.

③ Bei der Beziehung *hat geliehen* kann von einer n:m-Beziehung ausgegangen werden. Für einen Ausleihvorgang muss auch das Ausleih- und das Rückgabedatum gespeichert werden.

④ Der Entity-Typ *Autor* hat einen künstlichen Primärschlüssel *Autor-Nr.*



⑤ Da ein Buch mehrere Autoren haben kann, muss die Verknüpfung zwischen den Entity-Typen *Buch* und *Autor* mit einer n:m-Beziehung realisiert werden.

⑥ Beim Entity-Typ Verlag gehen wir davon aus, dass der Name als Primärschlüssel verwendet werden kann.

⑦ Ein *Verlag* veröffentlicht mehrere *Bücher*. Daher wird die Verknüpfung als 1:n-Beziehung realisiert.

⑧ Der Entity-Typ *Kategorie* hat als einziges Attribut eine *Bezeichnung*.

⑨ Ein *Buch* kann mehreren *Kategorien* zugeordnet werden, und es muss daher an dieser Stelle eine n:m-Beziehung verwendet werden.

Angelehnt an die Mengenschreibweise aus der Mathematik ist die folgende kompakte Beschreibung der Entity-Typen, Attribute, Schlüsselfelder und Beziehungen in textueller Form:

Entity-Typen

Entleiher(<u>Entleiher-Nr</u>, Vorname, Nachname, Straße, Ort, E-Mail)
Buch(<u>ISBN</u>, Titel, Jahr)
Autor(<u>Autor-Nr</u>, Vorname, Nachname)
Verlage(Name, Ort)
Kategorie(Bezeichnung)

Beziehungstypen

hat geliehen(*Entleiher, Buch,* Ausleihdatum, Rückgabedatum)
ist Autor von(*Autor, Buch*)
erscheint bei(*Buch, Verlag*)
gehört zu(*Buch, Kategorie*)

Tabellen normalisieren

Das **Normalisieren** von Tabellen ist ein analytisches Verfahren zur Reduktion von Daten. Durch das Zerlegen von Relationen in kleinere Relationen sollen folgende Ziele erreicht werden:

- Vermeiden von Redundanz und Anomalien beim Einfügen, Löschen oder Ändern von Datensätzen. Redundanzen erfordern nicht nur einen höheren Speicherplatzbedarf, sondern sind auch oft die Ursache für verschiedene Fehler.
- Normalisierte Tabellen verringern die Notwendigkeit, Relationen umzustrukturieren.
- Das Datenmodell wird übersichtlicher.

Das folgende Beispiel zeigt einige Probleme, die bei nicht normalisierten Tabellen auftreten können:

Bestell-Nr.	Anzahl	Artikel	Preis	Kundenanschrift
1	40	R	12,80	M. Müller, Hauptstr. 5, 79834 Emmendingen
2	24	B	2,10	Max Fischer, Torstr. 5, 79123 Freiburg
3	34	R	12,80	B. Maier, Rathausplatz 5, 79345 Müllheim
4	50	S	1,20	M. Müller, Hauptstr. 5, 79834 Emmendingen
5	25	F	1,20	R. Bauer, Lindenmatte 5, 79117 Freiburg

1. *Redundante Daten*
   Am deutlichsten ist die Redundanz bei der Kundenanschrift, die bei jeder Bestellung komplett festgehalten wird.

2. *Änderungsanomalie*
   Wenn man die Adresse eines Kunden oder die Bezeichnung für einen Artikel ändern möchte, muss man alle Zeilen der Tabelle durchgehen und in jeder Zeile die gewünschten Änderungen durchführen.

3. *Löschanomalie*
   Wenn man die 3. Bestellung löscht, wird auch die Adresse von B. Maier gelöscht. Bei einer späteren Bestellung dieses Kunden muss die Adresse wieder neu eingegeben werden.

Durch schrittweises Normalisieren kann man diese Probleme lösen. Die Datenbanktheorie unterscheidet mindestens fünf Normalformen. Im Folgenden sollen aber nur die ersten drei Normalformen erläutert werden, da sie für unsere Zwecke ausreichen dürften.

1. Normalform

Definition: Tabellen befinden sich in der 1. Normalform, wenn alle Attribute (Felder) elementar sind.

Rechn.-Nr.	Kunden-Nr.	Datum	Artikel
1	4	01.03.98	2 Papier
2	24	01.03.98	3 Papier, 5 CD, 2 Druckerpatronen
3	3	02.03.98	1 CD
4	5	02.03.98	20 Disketten
5	2	03.03.98	10 Disketten, 1 Druckpatrone
6	7	04.03.98	1 Soundkarte

In der obigen Tabelle findet sich in den Artikelfeldern jeweils eine Liste von Werten. Wenn man im Beispiel die Aufgabe hätte, einen Bericht über alle verkauften Artikel zu erstellen, wäre das sehr schwierig. Auch eine modifizierte Tabelle mit den verkauften Artikeln in mehreren Spalten, wie in der folgenden Tabelle, wäre keine geeignete Lösung, da nicht klar ist, wie viele verschiedene Artikel ein Kunde maximal kaufen können soll und wie viele Spalten für diese Artikel eingerichtet werden sollen.

R.-Nr.	K.-Nr.	Datum	Anz.1	Art.1	Anz.2	Art.2	Anz.3	Art.3
1	4	01.03.98	2	Papier				
2	24	01.03.98	3	Papier	5	CD	2	Druckp.
3	3	02.03.98	1	CD				
4	5	02.03.98	20	Disks	1	Druckp.		
5	2	03.03.98	10	Disks				
6	7	04.03.98	1	Soundk.				

Eine Tabelle, die der 1. Normalform genügt, finden Sie im nächsten Beispiel:

R.-Nr.	Position	K.-Nr.	Datum	Anzahl	Artikel
1	1	4	01.03.98	2	Papier
2	1	24	01.03.98	3	Papier
2	2	24	01.03.98	5	CD
2	3	24	01.03.98	2	Druckerpatrone
3	1	3	02.03.98	1	CD
...	...	...	...	...	...

Durch das Hinzufügen der Spalte Position gibt es nur elementare Werte und keine sich wiederholenden Gruppen.

Allerdings gibt es noch mehrfach Redundanzen in der Tabelle, die durch die folgenden Normalformen eliminiert werden können.

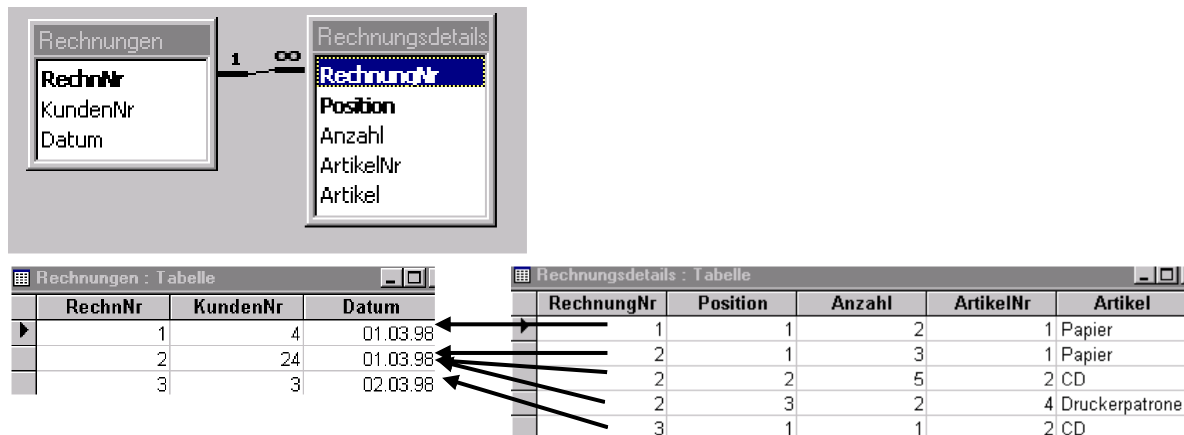
2. Normalform

Definition: Tabellen befinden sich in der 2. Normalform, wenn sie die 1. Normalform haben und darüber hinaus jedes Nichtschlüsselfeld vollständig vom (gesamten) Primärschlüssel abhängig ist.

Die obige Tabelle genügt zwar der 1. Normalform, allerdings sind nicht alle Spalten nur vom Primärschlüssel abhängig. Der Primärschlüssel setzt sich bei der Tabelle aus „R.-Nr.“ **und** „Position“ zusammen. Die Spalten „K.-Nr.“ und „Datum“ sind **allein** von der Rechnungsnummer „R.-Nr.“ und nicht von der Kombination (Primärschlüssel) abhängig. Damit widerspricht diese Tabelle der 2. Normalform.

Das führt dazu, dass z.B. das Datum bei einer Rechnung mit mehreren Positionen mehrfach eingegeben werden muss.

Die 2. Normalform kann erreicht werden, wenn man die Tabelle in zwei neue Tabellen teilt:



RechnNr	KundenNr	Datum
1	4	01.03.98
2	24	01.03.98
3	3	02.03.98

RechnungNr	Position	Anzahl	ArtikelNr	Artikel
1	1	2	1	Papier
2	1	3	1	Papier
2	2	5	2	CD
2	3	2	4	Druckerpatrone
3	1	1	2	CD

In der Tabelle „Rechnungen“ wird als Primärschlüssel „RechnNr“ verwendet. Die Tabelle „Rechnungsdetails“ verwendet den aus „RechnungNr“ und „Position“ zusammengesetzten Primärschlüssel. In beiden Tabellen sind die Nichtschlüsselfelder nur vom jeweiligen Primärschlüssel abhängig. Die Daten zur Kundennummer und zum Datum einer Rechnung werden nicht mehr mehrfach gespeichert, wie es noch in der Tabelle der Fall war, die nicht in der 2. Normalform war.

☞ ***Übung****: Analysieren Sie die Datenbank beispiel.mdb im Ordner „normalisierung“ und bringen Sie min. auf die 2. Normalform.*

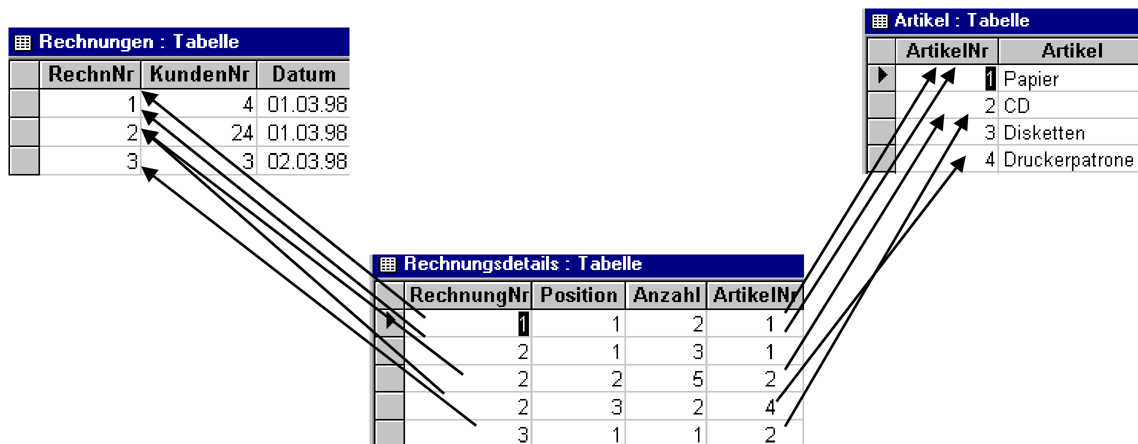
3. Normalform

Definition: Tabellen befinden sich in der 3. Normalform, wenn sie die 2. Normalform haben und darüber hinaus alle Nichtschlüsselfelder voneinander unabhängig sind.

Oder: Außerdem darf keines der Nichtschlüsselfelder transitiv vom Primärschlüssel abhängen.
{RechNr, Position} → {ArtikelNr}
{ArtikelNr} → {Artikel(name)}

Im obigen Beispiel befindet sich die Tabelle „Rechnungsdetails“ nicht in der 3. Normalform, da das Feld „Artikel“ eindeutig vom Feld „ArtikelNr“ abhängig ist. Eine derartige Abhängigkeit könnte zu Schwierigkeiten führen, wenn z.B. in zwei Zeilen zwar dieselbe Artikelbezeichnung eingegeben wird aber nicht die gleiche Artikelnummer (*Änderungsanomalie*). In der Tabelle sind die negativen Konsequenzen auch gut zu erkennen: Die Information, dass zur „Artikelnr„ 1 der Artikel „Papier“ gehört, ist mehrfach und redundant abgelegt. Außerdem kann man die Eingabe der Artikelbezeichnungen in der Tabelle oder in einem Formular viel besser und bequemer gestalten indem man die Bezeichnung z.B. aus einem Kombinationsfeld mit allen Artikelbezeichnungen nachschlagen und eintragen lässt.
Für die 3. Normalform muss die Tabelle erneut aufgeteilt werden:

In der Tabelle „Rechnungsdetails“ wird jetzt nur die Artikelnummer verwendet und die Artikelbezeichnungen werden in einer neuen Tabelle verwaltet. Damit ist die Redundanz bei der Artikelbezeichnung eliminiert, und es ist jetzt beispielsweise sehr viel einfacher und weniger fehleranfällig, die Artikelbezeichnung eines Artikels zu ändern. Die Bezeichnung muss in der Tabelle „Artikel“ nur einmal geändert werden. Zuvor hätte man die Bezeichnung an mehreren Stellen in der Tabelle „Rechnungsdetails“ ändern müssen (Änderungsanomalie).



Rechnungen : Tabelle

RechnNr	KundenNr	Datum
1	4	01.03.98
2	24	01.03.98
3	3	02.03.98

Artikel : Tabelle

ArtikelNr	Artikel
1	Papier
2	CD
3	Disketten
4	Druckerpatrone

Rechnungsdetails : Tabelle

RechnungNr	Position	Anzahl	ArtikelNr
1	1	2	1
2	1	3	1
2	2	5	2
2	3	2	4
3	1	1	2

Tabellen in der 3. Normaform sind weitgehend frei von Redundanzen.

☞ ***Übung**: Analysieren Sie die Datenbank 2_normalform_beispiel.mdb im Ordner „normalisierung“ und bringen Sie min. auf die 3. Normalform.*

Abschließend kann man sagen, dass eine problemgerechte ER-Modellierung in der ersten Phase der Datenbankentwicklung i.d.R. schon zu einem Konzept führt, dass den ersten drei Normalformen genügt.

Structured Query Language (SQL)

In den siebziger Jahren entwickelte IBM ein Produkt namens SEQUEL oder Structured English Query Language, aus dem schließlich SQL bzw. *Structured Query Language (*dt.: strukturierte Abfragesprache) wurde. IBM wollte eine standardisierte Methode für den Zugriff auf Daten und die Bearbeitung von Daten in einer relationalen Datenbank haben. SQL ist leicht zu erlernen und die meisten Datenbank-Applikationen und -produkte bauen heute auf dieser Sprache auf.

SQL ist

1. eine **nichtprozedurale** Sprache. Mit SQL wird beschrieben, *welche* Daten abzurufen, zu löschen oder einzufügen sind. Es wird *nicht* beschrieben, *wie* dies zu geschehen hat. Dabei arbeitet SQL mit logischen Mengen von Daten. Die Daten werden aus Tabellen ausgewählt.
   Beispiel: *SELECT* name, vorname *FROM* mitarbeiter *WHERE* gehalt > 30000
   „Zeige mir Namen und Vornamen aller Mitarbeiter, die mehr als 30000 €verdienen".
2. eine **plattformunabhängige** und **produktübergreifende** Sprache. Sie wurde von ANSI (American National Standards Institute) und ISO (International Standards Organization) als Standard definiert. Der aktuelle Standard 1999 ist SQL99 oder SQL3 aus dem Jahre 1999. Leider sind die Standards nur ein Ideal und nur sehr wenige Hersteller erreichen oder überfüllen die Bedingungen von SQL99.

Die Anweisungen von SQL kann man in vier Befehlsgruppen gliedern:

1. Data Definition Language (**DDL**)
   Mit der DDL kann man neue Datenbanken erstellen, Tabellen, Indizes und weitere Objekte einrichten und die Struktur existierender Datenbanken verändern und löschen.
2. Data Manipulation Language (**DML**)
   Die DML dient der Manipulation von Daten. Es können Datensätze angelegt, geändert oder gelöscht werden. Die wichtigsten Kommandos der DML sind INSERT, UPDATE und DELETE.
3. Data Query Language (**DQL**)
   Mit Hilfe der DQL können Abfragen durchgeführt werden. Das wichtigste Kommando ist SELECT.
4. Data Control Language (**DCL**)
   In der DCL finden sich Kommandos zur Einstellung der Sicherheitsmechanismen von SQL. Mit der DCL kann man Benutzer anlegen und ihnen Zugriffsrechte auf Datenbanken gewähren und entziehen.
   Im weiteren Sinne könnte man die Kommandos der DCL auch der DDL zuordnen.

Von SQL gibt es eine ganze Reihe von herstellerabhängigen Dialekten. I.d.R. enthalten diese Dialekte den größten Teil des Standards SQL99. Darüber hinaus sind oft weitere Features integriert, wie z.B. eine Erweiterung um Java. Einige beliebte SQL-Dialekte sind:

- PL/SQL, das in Oracle-Produkten verwendet wird.
- Transact-SQL des Microsoft SQL Servers
- PostgresSQL (Procedural Language/postgresSQL)
- SQL in InterBase von Borland
- MySQL – ein beliebtes Open Source-Datenbankverwaltungssystem.

Auch in MS-Access kann man SQL-Abfragen erstellen oder mit Hilfe einer Abfrage Tabellen für eine Datenbank erstellen.

Datenbanken erstellen mit *CREATE DATABASE*

Die Anweisung *CREATE DATABASE* gehört zur Data Definition Language (DDL). Mit ihr lassen sich Datenbanken erstellen. Diese Datenbanken enthalten noch keine eigenen Tabellen. I.d.R. sind aber bereits Systemtabellen enthalten, die Metadaten über die Datenbank und die Datenbankobjekte enthalten. Eigene Tabellen können mit der Anweisung *CREATE TABLE* erstellt werden.

Beispiel: *CREATE DATABASE bibliothek;*

Datenbanken löschen mit *DROP DATABASE*

Mit der Anweisung *DROP DATABASE* der DDL lassen sich Datenbanken löschen. Die Datenbank wird nach einer bestätigten Rückfrage komplett mit allen Tabellen und deren Inhalt gelöscht.

Beispiel: *DROP DATABASE bibliothek;*

Alle Datenbanken zeigen lassen mit *SHOW DATABASES* (MySQL)

Die Anweisung *SHOW DATABASES* von MySQL listet alle Datenbanken auf.

Die aktuelle Datenbank wechseln mit *USE DATABASE* (MySQL)

Mit der Anweisung *USE DATABASE* können Sie eine bereits erstellte Datenbank verwenden. Die Anweisung könnte man zur Data Control Language zählen oder zur Data Definition Language.

Diese Anweisung ist typisch für MySQL. In anderen SQL-Dialekten werden für diese Aktion andere Anweisungen verwendet – Z.B. „*CONNECT bibliothek*;" in InterBase.

Beispiel: *USE DATABASE bibliothek;*

Tabellen erzeugen mit *CREATE TABLE*

Dieser Befehl gehört zur Data Definition Language (DDL) von SQL und dient zum Erzeugen von neuen Tabellen und den zugehörigen Datenfeldern. Er besitzt sehr viele zusätzliche Optionen, die z.B. das Definieren des Primärschlüssels, das Erstellen von Indizes, das Einfügen von Gültigkeitsprüfungen und das Definieren von Standardwerten erlauben.

Allgemeine Syntax:

```
CREATE TABLE tabellenname
        (datenfeld1 datentyp1 [DEFAULT standardwert1 | NULL | NOT NULL] [AUTO_INCREMENT],
        (datenfeld2 datentyp2 [DEFAULT standardwert1 | NULL | NOT NULL] [AUTO_INCREMENT],
        …
        (datenfeldn datentypn [DEFAULT standardwertn | NULL | NOT NULL] [AUTO_INCREMENT],
        PRIMARY KEY(datenfeldname));
```

Mit der Anweisung *CREATE TABLE* wird eine neue leere Tabelle erstellt. Danach folgt der gewünschte Tabellenname. In runden Klammern folgen die Vereinbarungen für die einzelnen Datenfelder, für die jeweils ein Name und ein Datentyp angegeben werden muss. Mit der Angabe *PRIMARY KEY* wird ein Datenfeld als **Primärschlüssel** vereinbart.

Beispiel:

```
CRATE TABLE adressen(
        ID INTEGER NOT NULL,
        Vorname VARCHAR(50),
        Nachname VARCHAR(80),
        Geburtstag DATE,
        PRIMARY KEY(ID));
```

Die Beispieltabelle erhält den Namen *adressen*. In Klammern erfolgt die Definition der Datenfelder.
Das erste Datenfeld heißt *ID* und ist vom Typ *INTEGER*. Mit *NOT NULL* wird festgelegt, dass das Wert immer einen Wert enthalten muss. Das ist u.a. sinnvoll, da in der letzten Zeile mit *PRIMARY KEY(ID)* festegelegt wird, dass dieses Feld auch Primärschlüssel ist.
Die Felder *Vorname* und *Nachname* werden mit *VARCHAR* als Zeichenketten mit einer maximalen Länge von 50 und 80 Zeichen definiert. Bei *VARCHAR* werden nur so viele Zeichen gespeichert, wie tatsächlich benötigt werden.
Im Feld *Geburtstag* wird ein Datum in der Form „1995-12-31"gespeichert.

Bei der Definition von Datenfeldern können die folgenden optionalen Parameter verwendet werden:

NOT NULL	Mit diesem Parameter wird eine Eingabe für das entsprechende Datenfeld erzwungen. Für Schlüsselfelder ist die Angabe *NOT NULL* unbedingt anzugeben.
NULL	Mit diesem Parameter wird festgelegt, dass das Datenfeld standardmäßig keinen Wert (auch nicht 0 oder die leere Zeichenkette) enthält. Dies entspricht dem Wert *NULL*.

DEFAULT standardwert	Der Parameter *DEFAULT* definiert einen Standardwert für das Datenfeld.
AUTO_INCREMENT (MySQL)	Der Wert dieses Datenfeldes wird automatisch beim Anlegen eines neuen Datensatzes aus dem Wert des Datenfeldes des vorherigen Datensatzes plus eins gesetzt. Dieser Wert kann vom Benutzer nicht geändert werden. Diese Einstellung ist ideal für Primärschlüsselfelder.

In MySQL können u.a. die folgenden Datentypen eingesetzt werden:

TINYINT	8-Bit-Integer (1 Byte)
SMALLINT	16-Bit-Integer (2 Byte)
MEDIUMINT	24-Bit-Integer (3 Byte)
INT, INTEGER	32-Bit-Integer (4 Byte)
BIGINT	64-Bit-Integer (8 Byte)
FLOAT	Fließkommazahl, 8 Stellen Genauigkeit (4 Byte)
DOUBLE	Fließkommazahl, 16 Stellen Genauigkeit (8 Byte)
REAL	wie *DOUBLE*
DATE	Datum in der Form "2003-12-31", Bereich 1000-01-01 bis 9999-12-31 (3 Byte)
TIME	Zeit in der Form "23:55:05", Bereich +/- 838:59:59 (3 Byte)
DATETIME	Kombination aus DATE und TIME in der Form "2003-12-31 23:55:05" (8 Byte)
YEAR	Jahreszahl 1900-2155 (1 Byte)
TIMESTAMP	Datum und Uhrzeit werden bei jeder Veränderung des Datensatzes mit Systemzeit aktualisiert
CHAR(n)	Zeichenkette mit vorgegebener Länge, max. 255 Zeichen (n Byte)
VARCHAR(n)	Zeichenkette mit variabler Länge, max. n Zeichen (n < 256) Speicherbedarf: 1 Byte pro Zeichen (tatsächliche Länge) + 1
TINYTEXT	Zeichenkette mit variabler Länge, max. 255 Zeichen (n + 1 Byte)
TEXT	Zeichenkette mit variabler Länge, max. 2^{16} Zeichen (n + 2 Byte)
MEDIUMTEXT	Zeichenkette mit variabler Länge, max. 2^{24} Zeichen (n + 3 Byte)
LONGTEXT	Zeichenkette mit variabler Länge, max. 2^{32} Zeichen (n + 4 Byte)
TINYBLOB	Binärdaten (z.B. Grafiken) mit variabler Länge, max. 255 Byte
BLOB	Binärdaten mit variabler Länge, max. 2^{16}-1 Byte
MEDIUMBLOB	Binärdaten mit variabler Länge, max. 2^{24}-1 Byte
LONGBLOB	Binärdaten mit variabler Länge, max. 2^{32}-1 Byte

Mit den folgenden beiden Befehlen werden die beiden Tabellen „Hersteller" und „Produkt" erzeugt, die über die Felder „HerstID" und „HId" über eine 1:n-Beziehung miteinander verknüpft sind (vgl. Abb.):

```
CREATE TABLE Hersteller(
        HerstID INT NOT NULL,
        Name VARCHAR(30) NOT NULL,
        Ort VARCHAR(50) NULL
        PRIMARY KEY (HerstID));
CREATE TABLE Produkt(
        ProdID INT NOT NULL,
        Bezeichnung VARCHAR(50) NOT NULL,
        Preis FLOAT NOT NULL,
        PRIMARY KEY (ProdID));
        FOREIGN KEY (HId) REFERENCES Hersteller (HerstID));
```

Mit Hilfe des Schlüsselworts *PRIMARY KEY* wird der Primärschlüssel festgelegt. In der Tabelle *Hersteller* ist das Feld *HerstID* vom Typ Integer (*INT*) und Primärschlüssel. Die Felder *Name* und *Ort* sind Zeichenketten (*VARCHAR(x)*) der Länge x.

In der Tabelle *Produkt* ist *ProdID* Primärschlüssel. Das Feld *Bezeichnung* ist eine Zeichenkette der Länge 50. Das Feld *Preis* ist vom Typ Fließkommazahl (*FLOAT*). Beide Felder sind vom Typ *NOT NULL*, d.h. in die Tabelle muss immer eine Bezeichnung und ein Preis eines Produktes eingegeben werden. Wenn Nullwerte möglich sein sollen, muss das Schlüsselwort NULL verwendet werden. In manchen Implementierungen (z.B. Access) kann auch einfach gar nichts angegeben werden. Das Feld *HId* ist vom Typ Integer, wie *HerstID* in der Tabelle *Hersteller*.

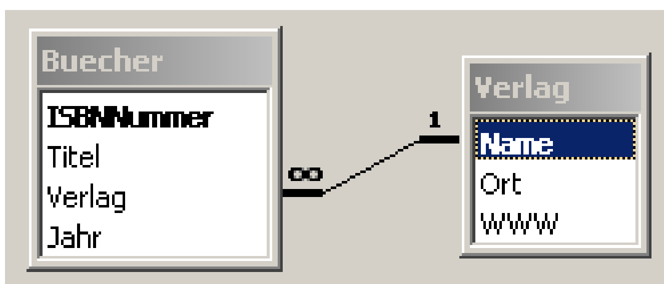
Die Verknüpfung der beiden Tabellen erfolgt mit *FOREIGN KEY (HId) REFERENCES Hersteller (HerstId).* Die beiden verknüpften Felder müssen dabei vom gleichen Datentyp sein. *HID* ist Fremdschlüssel in der Tabelle *Produkt* und wird mit dem Feld *HerstID* der Tabelle *Hersteller* verknüpft. Da *HerstID* in der Tabelle *Produkt* Primärschlüssel ist, wird automatische eine 1:n-Beziehung erzeugt.

In MySQL kann man derzeit aus Kompatibilitätsgründen zwar Fremdschlüsselfelder deklarieren. Diese führen aber nicht dazu, dass MySQL auf die Integrität der Daten achtet. In MySQL müssen Sie also selbst für referentielle Integrität (vgl. Seite 6) sorgen. Es besteht aber berechtigte Hoffnung, dass diese Funktion in einer der nächsten Versionen integriert wird.

Wenn eine MySQL-Tabelle erzeugt wird, werden drei Dateien erzeugt: Eine Tabellendefinitionsdatei mit der Erweiterung *.frm*, eine Datendatei mit der Erweiterung *.myd* und eine Indexdatei mit *.myi* als Erweiterung.

MySQL-Beispiel:

```
CREATE TABLE Verlag (
        Name VARCHAR(60) NOT NULL,
        Ort VARCHAR(50) NULL,
        WWW VARCHAR(60) NULL,
        PRIMARY KEY (Name) );
CREATE TABLE Buecher (
        ISBNNummer VARCHAR(15) NOT NULL,
        Titel VARCHAR(100) NULL,
        Verlag VARCHAR(60) NULL,
        JAHR INT NULL,
        PRIMARY KEY (ISBNNummer) );
```



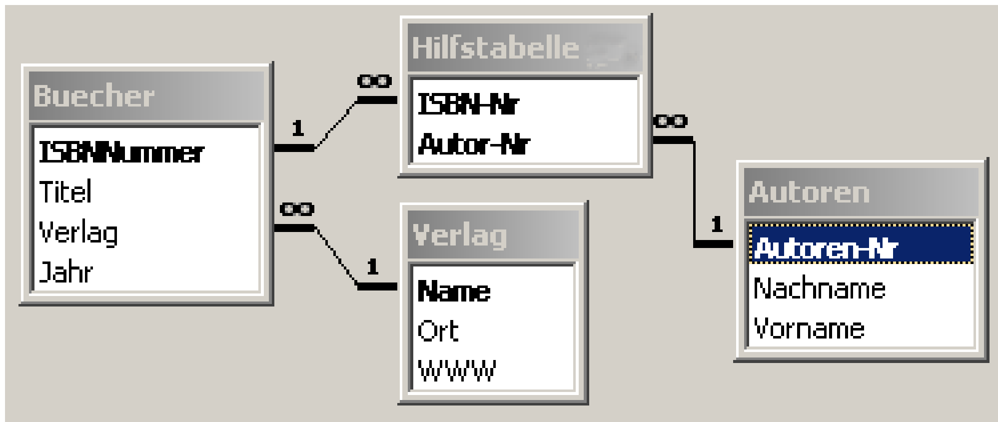
Wichtig ist bei diesem Beispiel, dass das Feld *Name* in der Tabelle *Verlag* und das Feld *Verlag* in der Tabelle *Buecher* die gleiche maximale Länge 60 haben.

Mit einer Hilfstabelle kann man einem Buch die Autorennamen über die ISBN-Nummer zuordnen:

CREATE TABLE Autoren (
Autoren_Nr INT NOT NULL AUTO_INCREMENT,
Nachname VARCHAR(80) NULL,
Vorname VARCHAR(60) NULL,
PRIMARY KEY (Autoren_Nr));

CREATE TABLE Hilfstabelle (
ISBN_Nr VARCHAR(15) NOT NULL,
Autor_Nr INT NOT NULL,
PRIMARY KEY (ISBN_Nr, Autor_Nr));

Es ergibt sich dann die folgende Struktur:



Existierende Tabelle mit *ALTER TABLE* ändern

Mit diesem Befehl kann eine bereits erstellte Tabelle geändert werden. Mit *ALTER TABLE* können Sie:

- Datenfelder hinzufügen oder löschen
  Beispiel:
  ALTER TABLE Verlag
  ADD Strasse VARCHAR(60) DEFAULT „unbekannt“,
  DROP WWW;
  Fügt das Feld *Strasse* hinzu und löscht das Feld *WWW*.
- Datenfelddefinitionen verändern
  Beispiel:
  ALTER TABLE Verlag
  CHANGE Ort VARCHAR(100) NOT NULL;
  Ändert die maximale Länge des Feldes *Ort* auf 100 Zeichen und fordert *NOT NULL*.
- Gültigkeitsprüfungen hinzufügen oder löschen
  Beispiel:
  ALTER TABLE Artikel
  ADD CONSTRAINT Pruef CHECK (Preis > 0);
  Hier wird eine neue Gültigkeitsbedingung definiert. Der Wert für das Feld *Preis* wird bei der Eingabe überprüft. Er muss größer als 0 sein.
- Schlüssel und Indizes hinzufügen oder löschen
  Beispiel:
  ALTER TABLE Verlag
  DROP PRIMARYKEY;
  Löscht den aktuellen Primärschlüssel.

Alle Tabellen einer Datenbank zeigen lassen mit *SHOW TABLES*

Die Anweisung *SHOW TABLES* zeigt eine Liste der vorhandenen Tabellen der aktuell geöffneten Datenbank an.

Beispiel: *SHOW TABLES FROM bibliothek LIKE m%;*

Zeigt alle Tabellen der Datenbank *bibliothek*, die mit dem Buchstaben *m* beginnen. Mit dem Schlüsselwort *FROM* kann eine andere als die aktuelle Datenbank benannt werden.

Daten einfügen, ändern und löschen mit *INSERT, UPDATE, DELETE*

Mit dem Befehl ***INSERT*** kann man einen neuen Datensatz in eine Tabelle einfügen. Nach dem Tabellennamen muss zuerst eine Liste der Spaltennamen und danach eine Liste mit den einzufügenden Werten übergeben werden. Mit dem folgenden Beispiel wird ein neuer Datensatz in die Tabelle *Produkte* gespeichert:

INSERT INTO Produkt (ProdID, Bezeichnung, Preis) VALUES (1, "Appenzeller", 6.7);

Der Datensatz hat die *ProdID* 1, die Bezeichnung „Appenzeller“ und einen Preis von 6,70 Euro. Auf die Nennung der Spaltennamen kann verzichtet werden, falls Werte für alle Spalten übergeben werden und dabei auch die Reihenfolge der Spalten genau beachtet wird.

Mit ***UPDATE*** können Sie einzelne Felder eines vorhandenen Datensatzes verändern. Mit dem folgenden Beispiel wird die Bezeichnung des Datensatzes mit der *ProdID* 1 verändert:

UPDATE Produkt SET Bezeichnung = “Appenzeller Medium“ WHERE ProdID=1;

Vorsicht! Wenn *UPDATE* ohne *WHERE* verwendet wird, gelten die Veränderungen für alle Datensätze der Tabelle.

In *UPDATE*-Anweisungen sind auch Berechnungen möglich. Z.B. kann man die Preise aller Produkte mit der folgenden Anweisung um 10% erhöhen:

*UPDATE Produkt SET Preis = Preis*1.1;*

Beachten Sie, dass *UPDATE* im Beispiel ohne *WHERE* verwendet wird.

Mit dem Kommando ***DELETE*** können Datensätze gelöscht werden. Im folgenden Beispiel werden alle Datensätze gelöscht deren Bezeichnung "Appenzeller" ist.

DELETE FROM Produkt WHERE Bezeichnung = "Appenzeller";

Im nächsten Beispiel werden alle Produkte gelöscht, die teurer als 30 Euro sind:

DELETE FROM Produkt WHERE Preis > 30;

Abfragen mit *SELECT*

Die Select-Anweisungen gehört zur Data Query Language und ist wohl die mächtigste Anweisung von SQL. Mit ihr können Abfragen durchgeführt werden, die die Daten i.d.R. der Datenbank nicht verändern.

Die denkbar einfachste Datenbankabfrage lautet *SELECT * FROM tabelle.* Man erhält so alle Zeilen der Tabelle *tabelle.* Durch die Verwendung des Zeichens * werden alle Spalten der Tabelle ausgegeben.

Wenn man die Anzahl der Spalten einschränken möchte (**Projektion**), gibt man die Namen der gewünschten Spalten explizit an, z.B.: *SELECT name, vorname FROM adressen.*

Die Ergebnisse einer SELECT-Abfrage werden normalerweise unsortiert ausgegeben. Falls eine geordnete Ergebnisliste gewünscht wird, muss dies explizit durch *ORDER BY spalte* angegeben werden, z.B: *SELECT name, vorname FROM adressen ORDER BY name.* Soll die Sortierreihenfolge umgekehrt sein, muss noch das Schlüsselwort DESC (descending (absteigend)) angehängt werden.

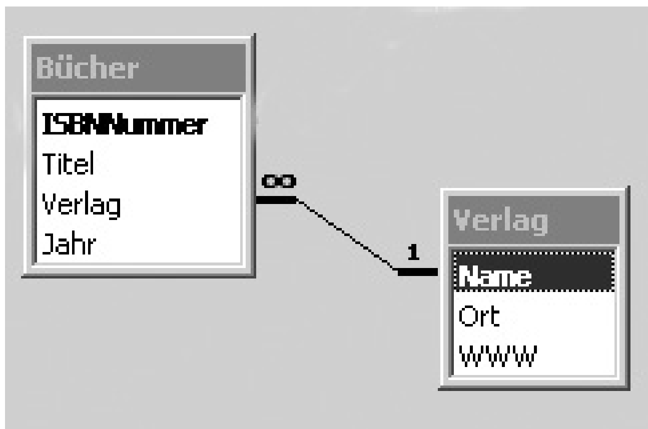
Wenn man eine **Selektion** durchführen möchte, müssen bestimmte Bedingungen in die Anweisung aufgenommen werden. Solche Bedingungen werden mit *WHERE* eingeleitet. Im folgenden Beispiel werden der Nach- und Vorname aus der Tabelle *adressen* der Personen angezeigt, die 18 Jahre und älter sind. Die gefundenen Datensätze werden zusätzlich nach dem Nachnamen sortiert ausgegeben.

SELECT name, vorname, alter FROM adressen WHERE alter >= 18 ORDER BY name

Die folgende Abfrage liefert alle Datensätze deren Nachname mit “L“ beginnt:
*SELECT * FROM adressen WHERE name LIKE ‘L%’*
Vergleiche mit einer Menge von Werten werden am einfachsten mit *IN* durchgeführt:
SELECT name, vorname FROM adressen WHERE alter IN (5, 7, 9)
So werden alle 5-, 7- und 9-jährigen gefunden.

Bis jetzt betrafen alle Beispiele für *SELECT* nur die Daten einer einzigen Tabelle. Oft sind aber in einer relationalen Datenbank zwei oder mehrere Tabellen verknüpft. Bei einer Abfrage mit *SELECT* müssen Daten aus mehreren Tabellen zusammengefügt werden. Dazu wird das Schlüsselwort ***JOIN*** verwendet. Die verknüpften Tabellen müssen eine oder mehrere Spalten haben, die eine gemeinsame Menge von Werten enthalten. Meistens handelt es sich um eine 1:n-Beziehung, bei der der sogenannte *JOIN-Schlüssel* Primärschlüssel der einen und ein Fremdschlüssel in der anderen Tabelle ist (vgl. S. 4 ff).

Ein erstes Beispiel für ein *JOIN* über zwei Tabellen verdeutlicht, dass es verschiedene *JOIN-Typen* geben muss. Betrachten wir die beiden Tabellen „Bücher“ mit den Feldern *ISBNNummer*, *Titel*, *Verlag* und *Jahr* und *Verlag* mit den Feldern *Name*, *Ort* und *WWW*. Die beiden Tabellen sind über die Felder *Verlag* und *Name* mit einer 1:n-Beziehung verknüpft.



Der Versuch aus den Tabellen *Bücher* und *Verlag* eine Liste aller Bücher mit ISBN-Nummer, Titel, Verlag und Ort zu bilden, führt nicht zum gewünschten Ergebnis:
SELECT Bücher.*ISBNNummer,* Bücher.*Titel,* Bücher.*Verlag, Verlag.Ort FROM Bücher, Verlag;*
SQL liefert eine Liste aller möglichen Kombinationen der Daten beider Tabellen – z.B. 20.000 Kombinationen bei einer Datenbank mit 1000 Büchern und 200 Verlagen. Dieser sogenannte ***CROSS JOIN*** entspricht dem kartesischen Produkt (vgl. S. 1).

Zum gewünschten Ergebnis führt eine ***INNER-JOIN***-Abfrage. Dabei werden nicht zusammenpassende Zeilen aus beiden Tabellen verworfen:
SELECT Bücher.ISBNNummer, Bücher.Titel, Bücher.Verlag, Verlag.Ort
FROM Bücher ***INNER JOIN*** *Verlag ON Bücher.Verlag = Verlag.Name;*
Diese Abfrage würde bei 1000 Büchern eine Liste mit 1000 Zeilen liefern, falls bei jedem Buch eine Eintragung im Feld *Verlag* vorhanden wäre. Bücher, die keinen Eintrag (NULL-Wert) haben, werden nicht berücksichtigt. Dieser Join ist der am häufigsten verwendete Verbund.

Bei einem ***LEFT JOIN*** werden alle Datensätze auf der linken Seite der Join-Anweisung zurückgegeben. Wenn es zu einem Datensatz aus der linken Tabelle keinen passenden Datensatz in der rechten Tabelle gibt, wird dieser trotzdem zurückgegeben. Solche linke Inklusionsverknüpfungen schließen alle Datensätze aus der ersten (linken) Tabelle von zwei Tabellen ein, auch wenn keine entsprechenden Werte für Datensätze in der zweiten (rechten) Tabelle vorhanden sind. Die Felder der zweiten (rechten) Tabelle bleiben leer, wenn kein passender Datensatz vorhanden ist.

SELECT Bücher.ISBNNummer, Bücher.Titel, Bücher.Verlag, Verlag.Ort
FROM Bücher ***LEFT JOIN*** *Verlag ON Bücher.Verlag = Verlag.Name;*
Bei dieser Abfrage werden auch dann die Buchdaten *ISBNNummer* und *Titel* eines Buches zurückgegeben, wenn kein Verlag in der Tabelle Bücher eingetragen ist. Die Felder für den Verlagsnamen und –ort werden leer angezeigt.

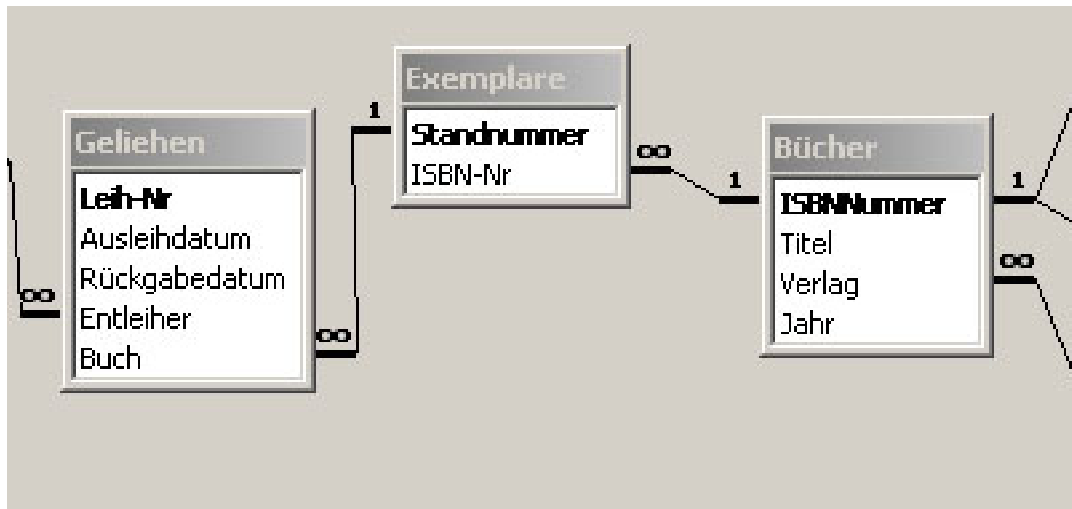
Bei einem ***RIGHT JOIN*** werden alle Datensätze auf der rechten Seite der Join-Anweisung zurückgegeben auch dann wenn, wenn es keinen passenden Datensatz in der Tabelle auf der linken Seite gibt.

Solche rechte Inklusionsverknüpfungen schließen alle Datensätze aus der zweiten (rechten) Tabelle von zwei Tabellen ein, auch wenn keine entsprechenden Werte für Datensätze in der ersten (rechten) Tabelle vorhanden sind. Die Felder der ersten (linken) Tabelle bleiben leer, wenn kein passender Datensatz vorhanden ist.

SELECT Bücher.ISBNNummer, Bücher.Titel, Verlag.Name,, Verlag.Ort
FROM Bücher ***RIGHT JOIN*** *Verlag ON Bücher.Verlag = Verlag.Name;*

Bei dieser Abfrage werden alle Bücher zurückgegeben, für die ein Verlagsnamen eingetragen ist *und* auch die Verlagsdaten *Name* und *Ort* eines Verlages bei dem kein Buch der linken Tabelle in diesem Verlag erschienen ist. Die Felder für die ISBN-Nummer und den Titel eines Buches werden für die Verlage ohne Eintrag in der linken Tabelle leer angezeigt.

Beachten Sie, dass bei *LEFT JOIN* und *RIGHT JOIN* die Reihenfolge in der die Tabellen im SQL-Kommando genannt werden nicht gleichgültig ist.



Die folgende Abfrage ist ein Beispiel für ***JOINs*** **über drei Tabellen**:

SELECT Exemplare.Standnummer, Bücher.ISBNNummer, Bücher.Titel, Bücher.Verlag
FROM (Bücher INNER JOIN Exemplare ON Bücher.ISBNNummer=Exemplare.ISBN-Nr)
INNER JOIN Geliehen ON Exemplare.Standnummer=Geliehen.Buch
WHERE (((Geliehen.Rückgabedatum) Is Null));

Es werden die *Standnummer,* die *ISBNNummer,* der *Titel* und der *Verlag* aus den Tabellen *Exemplare* und *Bücher* für alle geliehenen Bücher ausgegeben. Entliehen sind die Bücher dann, wenn sie in der Tabelle *Geliehen* im Feld *Rückgabedatum* keinen Eintrag (NULL-Wert) haben.

Funktionen

Es gibt zwei verschiedene grundlegende Typen von Funktionen, *Aggregat-Funktionen* und *skalare Funktionen*.

Aggregat-Funktionen in Abfragen mit *SELECT* arbeiten auf der Sammlung von Werten und geben einen einzigen, zusammenfassenden Wert zurück. In SQL gibt es z.B. die Aggregatsfunktionen *AVG*, *MIN*, *MAX* und *SUM*, die den Durchschnitt, das Minimum, das Maximum und die Summe mehrerer Werte berechnen. Die Funktion *COUNT(ausdruck)* zählt die vom Ausdruck definierten Zeilen, *COUNT(*)* zählt alle Zeilen einer angegebenen Tabelle.

Im folgenden Beispiel wird die Anzahl der Bücher ermittelt, die im Verlag Springer erschienen sind:

SELECT COUNT(Verlag) ***AS AnzahlBücher*** *FROM Bücher WHERE Verlag LIKE "Springer";*

Mit dem Schlüsselwort *AS* wird der einzigen Spalte der Ergebnistabelle der Abfrage der Namen „AnzahlBücher“ gegeben.

Wenn zusätzlich zu den Aggregatsfunktionen weitere Ausdrücken in der Elementliste einer *SELECT*-Abfrage verwendet werden, muss die Anweisung eine **GROUP BY-Klausel** haben:

SELECT Bücher.Verlag, COUNT(Verlag) AS AnzahlBücher FROM Bücher GROUP BY Verlag;

Die GROUP BY-Klausel gruppiert die selektierten Zeilen basierend auf dem Wert der Ausdrücke und liefert eine einzelne Zeile mit Informationen für jede Gruppe zurück.

Skalare Funktionen arbeiten auf einem einzelnen Wert und geben auf der Basis dieses Wertes einen einzelnen Wert zurück. Einige skalare Funktionen, wie z.B. *CURRENT_TIME* benötigen überhaupt keine Argumente.

Alle SQL-Dialekte unterstützen eine Vielzahl solcher skalarer Funktionen. Die Liste reicht von der Funktion *ABS(X)*, die den absoluten Wert von X zurückgibt, bis zur Funktion *YEAR(datum)*, die einen Integer-Zahl zurückgibt, die das Jahr des angegeben Datums repräsentiert.