Teil 5:

Datenbankdesign, ER-Modell, Normalformen

Generell ist beim Datenbankdesign zwischen *logischem* und *physischem* Design zu unterscheiden. Das logische Design führt zu den Tabellen und Attributen, die in SQL-Anweisungen für ein Anwendungsprogramm genutzt werden Das physische Design bezieht sich auf die Effizienz der Ausführung durch Wahl und Konfiguration eines geeigneten RDBMS, Auswahl und Konfiguration geeigneter Festplatten, der Verteilung der Datenbanktabellen und des Transaktionslogs auf die Festplatten, der Wahl von geeigneten Indices, der Festlegung einer Backup-Strategie und ähnliches. Im Folgenden geht es ausschließlich um das logische Datenbankdesign.

Bisher sind wir beim Entwurf eines Schemas immer *Bottom-Up* vorgegangen. Aus der Aufgabenstellung ergaben sich durch scharfes Hinsehen sofort der Tabellenaufbau und die einzelnen Attribute. Bei kleinen, überschaubaren Aufgaben ist das OK, bei größeren, unübersichtlichen Aufgaben, ist ein anderer Ansatz zweckmäßig, der *Top-Down*, vom Groben zum Feinen, vorgeht. Man beginnt mit der Analyse des Ausschnittes der realen Welt (auch *Miniwelt* genannt), für die man ein Datenmodell entwickeln will. Die Tabellen und Attribute folgen später daraus.

Die reale Welt kann prinzipiell in Einheiten zerlegt werden (Entitäten, Entities) und in Beziehungen zwischen diesen Einheiten (Relationships). Damit kommt man zu einer Top-Down-Designmethode, die sich Entity-Relationship-Modell (ER-Modell) nennt. In einer natürlichsprachlichen Aufgabenstellung ergeben sich die Entities meist aus *Hauptwörtern*, die Relationships meist aus *Zeitwörtern*. Das ER-Modell wird graphisch dargestellt und enthält normalerweise noch keine Attribute.

Es existieren diverse kommerzielle Tools, die Datenbankentwurf nach dem ER-Modell, oder einer Variante davon, unterstützen. Die Notationen variieren etwas, die Konzepte sind aber im Wesentlichen die Gleichen.

Achtung: Die Begriffe 'Relation' und 'Relationship' bedeuten nicht das selbe. Eine Relationship im Sinne des ER-Modells ist eine Beziehung zwischen Entities, eine Relation im Sinne der Relationenalgebra ist eine Beziehung zwischen Attributen. Schlussendlich werden natürlich Entities und Relationships auf Relationen (d.h. Tabellen) abgebildet.

Das Entity-Relationship (ER) Modell

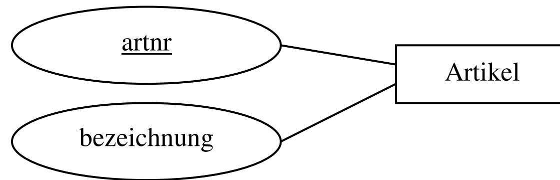
Das ER-Modell für den Top-Down Entwurf von Datenmodellen (entwickelt von Chen, 1972) geht in zwei Schritten vor:

1. Festlegen der Entities und Relationships
2. Abbildung der Entities und Relationships auf Relationen und Hinzufügen von Attributen.

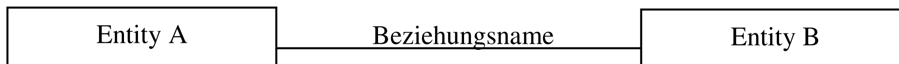
ER-Diagramme

Werden verwendet, um Entities und Relationships graphisch darzustellen. Entities werden in einer rechteckigen Box mit Namen dargestellt. Beispiele für Entities wären etwa *Person*, *Artikel*, *Lieferant*.

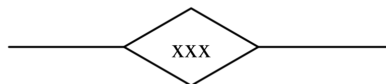
Erweiterte ER-Diagramme erlauben oft auch das Darstellen von Attributen, z.B. durch Ovale, oft auch mit Unterstreichung von Primärschlüsselattributen.



Beziehungen werden als beschriftete Linien zwischen Entities dargestellt. Beispiele: *kauft*, *liefert*.

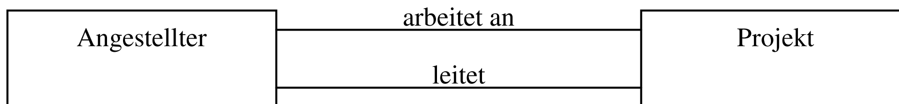


Manchmal werden Beziehungen auch innerhalb von Rauten beschriftet.



Beispiel (H. Sauer, Seite 224)

Es sind auch mehrere Beziehungen zwischen Entities möglich. Zum Beispiel könnte es noch einen Projektleiter geben.



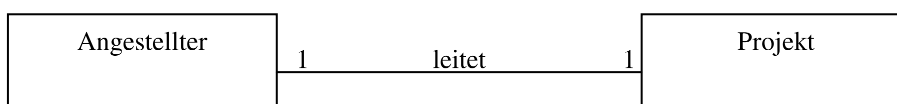
Es besteht nun ein Unterschied, ob ein Projektleiter genau ein Projekt leitet oder auch mehrere Projekte leiten kann. Wir möchten in der Lage sein, diesen Umstand darzustellen. Dazu führt man den *Grad* einer Beziehung ein. Folgende Grade haben große praktische Bedeutung und sind daher im ER-Modell berücksichtigt.

1:1 Eine Eins-zu-eins Beziehung (Beispiel: 'leitet', wenn ein Leiter genau ein Projekt leitet).

1:N Eine Eins-zu-viele Beziehung (Beispiel: 'leitet', wenn ein Leiter mehrere Projekte leiten kann).

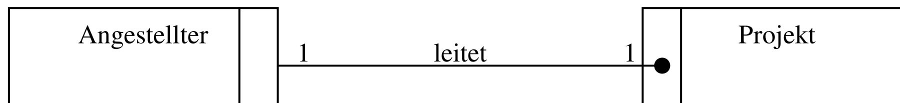
N:M Eine Viele-zu-viele Beziehung (Beispiel: 'arbeitet an', wenn ein Angestellter an mehreren Projekten arbeiten kann und ein Projekt von mehreren Angestellten bearbeitet wird).

1:1 Beziehung



Ein Angestellter leitet maximal ein Projekt und ein Projekt wird von maximal einem Angestellten geleitet. Es kann aber Angestellte geben, die kein Projekt leiten und Projekte, die keinen Leiter haben. Die Beziehung ist *optional*.

Möchte man ausdrücken, dass die Beziehung *obligatorisch* (verpflichtend) ist, wird folgende Notation verwendet.



Man kann sich eine Relationship als 2-spaltige Tabelle vorstellen, die mittels Fremdschlüssel die beiden Entities verknüpft. Wenn ein Entity in der entsprechenden Spalte (links oder rechts) der Relationship-Tabelle vorkommen *muss*, wird ein Kreis gezeichnet. Die Kreise können auch auf beiden Seiten einer Beziehung aufscheinen. In obigem Beispiel wird ausgedrückt, dass jedes Projekt genau einen Leiter haben muss, ein Angestellter muss aber kein Projektleiter sein.

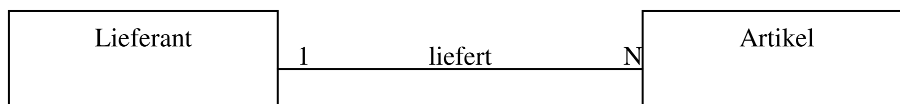
Abbildung auf Tabellen:

beidseitig obligatorische 1:1 Beziehung: beide Entities können zu einer gemeinsamen Tabelle verschmolzen werden.

einseitig obligatorische 1:1 Beziehung: Jeweils eine Tabelle pro Entity, obligatorisches Entity bekommt zusätzliches NOT NULL Attribut mit Fremdschlüssel auf das andere Entity.

optionale 1:1 Beziehung: Jeweils eine Tabelle pro Entity plus eine Tabelle für die Beziehung. Die Beziehungstabelle enthält einen UNIQUE INDEX für beide Fremdschlüssel. Alternative wäre wie bei einseitig obligatorischer 1:1 Beziehung die Verwendung eines Fremdschlüssel-Attributs in einer der beiden Entities, allerdings kann hier dieses Attribut NULL sein, was man zu vermeiden sucht.

1:N Beziehung



Auch hier stellt sich die Frage, ob jeder Lieferant liefert und ob jeder Artikel einen Lieferanten hat. Demgemäß kann man ebenfalls zwischen optional und ein- oder beidseitig obligatorisch unterscheiden. In obigem Beispiel handelt es sich um eine optionale Beziehung. Möchte man ausdrücken, dass jeder Artikel einen Lieferant hat, erhalten wir eine (einseitig) obligatorische 1:N Beziehung wie folgt.

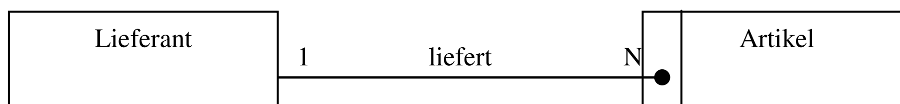


Abbildung auf Tabellen:

Die beteiligten Entitäten bilden in jedem Fall eine eigene Tabelle. Die Beziehung wird abgebildet je nach Art.

obligatorische 1:N Beziehung: N-Entity bekommt zusätzliches NOT NULL Attribut mit Fremdschlüssel auf das andere Entity. Wenn die 1-Entität obligatorisch ist, kann das durch das Datenmodell nicht direkt ausgedrückt werden. Der wichtige Fall ist aber dass die N-Entität obligatorisch ist.

optionale 1:N Beziehung: Eine eigene Tabelle für die Beziehung. Alternative wäre wie bei obligatorischer 1:N Beziehung die Verwendung eines Fremdschlüssel-Attributs in der N-Entity, allerdings kann hier dieses Attribut NULL sein, was man zu vermeiden sucht.

M:N Beziehung

Jeder Student kann mehrere Vorlesungen besuchen und jede Vorlesung kann von mehreren Studenten besucht werden.

Abbildung auf Tabellen:

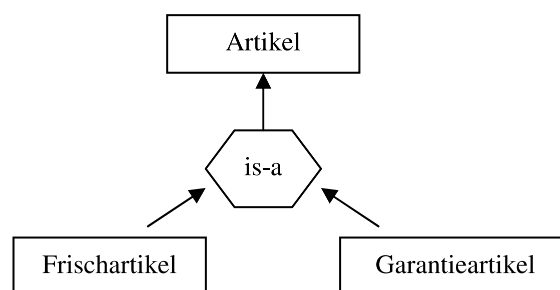
Beide Entitäten und die Relationship werden auf eigene Tabellen abgebildet. Auch hier könnte man prinzipiell zwischen optional und obligatorisch unterscheiden, das hat aber keine große praktische Bedeutung und kann im Datenmodell nicht direkt ausgedrückt werden. Es könnte auch sein, dass man nicht einfach ausdrücken möchte, dass eine Vorlesung von mindestens einem Studenten besucht wird, sondern ein anderes Limit festlegen möchte, z.B. mindestens 3. Es haben sich auch erweiterte Notationen herausgebildet, die diesen Umstand ausdrücken können (zum Beispiel die sogenannte min-max Notation). Da diese Eigenschaften aber keine Entsprechung im Tabellenaufbau haben, sind sie in der Praxis nicht sehr verbreitet und nicht so wichtig wie die Eigenschaften, die auf den Tabellenaufbau durchschlagen.

Vererbung

Erweiterungen des ER-Modells sind vor allem in Richtung Objektorientierung erfolgt. Dabei geht es darum, eine Vererbungsbeziehung wie in objektorientierten Programmiersprachen auf Datenmodellebene auszudrücken. Zum Beispiel könnte es neben den normalen Artikel einer Artikeldatenbank noch spezielle Artikelarten geben, die zusätzliche Eigenschaften und damit zusätzliche Attribute besitzen. In einer OOP-Sprache würde man so etwas durch die Definition einer Unterklasse, die alle Eigenschaften einer Basisklasse erbt, ausdrücken. Die Unterklasse kann aber noch zusätzliche Attribute definieren und ist in gewissem Sinn kompatibel zur Basisklasse.

Bezogen auf das Artikel-Beispiel könnte es Frischprodukte geben, die zusätzlich eine Haltbarkeitsdauer und eine vorgeschriebene Kühltemperatur besitzen. Es könnte auch Artikel geben, für die eine Garantie gegeben wird, dann muss die Garantiedauer und sonstige Garantiemodalitäten gespeichert werden, etc.

In einem erweiterten ER-Modell könnte Vererbung durch eine spezielle Form von Relationship ausgedrückt sein.



Wie kann nun diese Struktur auf Tabellen abgebildet werden?

Für alle beteiligten Entitäten werden eigene Tabellen angelegt. Das allgemeine (Basis-) Entity enthält neben dem Primärschlüssel nur die Attribute, die für alle Artikelarten erforderlich sind. Die speziellen Entitäten enthalten den gleichen Primärschlüssel und die jeweils zusätzlich benötigten Attribute. Sie enthalten aber *nicht* die gemeinsamen Attribute der Basistabelle.

Die Abbildung der Relationship erfolgt damit wie bei einer einseitig obligatorischen 1:1 Beziehung. Ein Artikel kann ein Frischartikel sein, muss es aber nicht. Ein Frischartikel ist sicher ein Artikel, also ist Artikel obligatorisch, woraus folgt dass in der Tabelle für Frischartikel ein Fremdschlüssel auf Artikel erforderlich ist, der NOT NULL ist. Dieser Fremdschlüssel ist gleichzeitig der Primärschlüssel für Frischartikel.

Abfragen, die sich auf Eigenschaften aller Artikel beziehen, werden mit der Tabelle Artikel durchgeführt, Abfragen, die sich auf Eigenschaften von Frischartikel beziehen, mit Frischartikel, allenfalls mit EQUI-JOIN auf Artikel. Wenn tiefe Vererbungsbeziehungen bestehen (mehrere Ebenen) können die Joins aber unangenehm komplex werden, woraus man folgern kann, dass eine Unterstützung durch ein RDBMS hier sinnvoll wäre. Tatsächlich sind solche Erweiterungen des relationalen Modells Gegenstand vieler Produkte und auch Gegenstand von Standardisierungsbemühungen.

In der Praxis findet man häufig noch die Vorgangsweise des *Flachdrückens* von Vererbungsbeziehungen. Es wird nur eine einzige Tabelle für alle Artikelarten angelegt, die alle Attribute enthält. Die speziellen Attribute für einzelne Artikelarten sind NULLable und nur für die jeweiligen Artikelarten mit Werten belegt. Meist wird auch ein zusätzliches Attribut (tag, Marke) eingeführt, dass die Artikelart beschreibt. Mit dieser Vorgangsweise werden Joins vermieden, aber es entstehen sehr komplexe Tabellen, wobei die meisten Attribute optional sind. Eine gute Dokumentation ist unentbehrlich. Bei Einführung einer neuen Artikelart muss die Tabelle erweitert werden, was Anwendungen inkompatibel machen kann. Die Vorgangsweise ist eng verwandt mit sogenannten Variantenrecords (Pascal) oder Unions (C), die schlussendlich durch Vererbung abgelöst wurden. In der Datenbankwelt steht dieser Schritt noch bevor.

Normalformen

Sinn: Vermeidung von unangenehmen Effekten, sogenannten *Anomalien*. Wenn sich *ein* Sachverhalt ändert, soll nur *eine* Datenbankoperation dafür notwendig sein, *nicht mehrere*. Wenn sich zum Beispiel die Artikelbezeichnung ändert, soll es genügen, ein UPDATE auf dem Attribut *bezeichnung* der Tabelle Artikel durchzuführen. Daraus folgt, dass die Bezeichnung eines Artikels nur an einer Stelle gespeichert sein darf. Ansonsten würde man von einer UPDATE-Anomalie sprechen.

Ähnliche Probleme ergeben sich beim Löschen und beim Einfügen. Wenn ein Artikel gelöscht werden soll, soll nach Möglichkeit ein einziges DELETE genügen, ohne Operationen in anderen Tabellen. Sonst würde man von einer DELETE-Anomalie sprechen, die auch darin bestehen kann, dass andere Informationen mit gelöscht werden, die man eigentlich noch braucht.

Normalisierung vermeidet Redundanzen und soll zu einem verständlichen Datenmodell führen, dass auch gut wartbar und erweiterbar ist. Durch Normalisierung sollen Sachverhalte realitätskonform festgehalten werden.

Folgendes Beispiel zeigt welche Probleme sich durch eine schlechte Tabellenstruktur ergeben können. An einer Universität werden sämtliche Professoren (P) und Vorlesungen (V) in einer einzigen gemeinsamen Tabelle PV gespeichert.

sch(PV): {[<u>pNr</u>, pName, pRang, pRaum, <u>vNr</u>, vTitel, vRaum, vSws]}

Kommt nun ein neuer Professor, der noch keine Vorlesung hat, kann dieser nicht eingefügt werden, es sei denn alle Vorlesungsdaten werden mit NULL befüllt. Das geht aber mit Bestandteilen des Primärschlüssels nicht (vNr), also müsste hier der Wert auf z.B. –1 gesetzt werden.

Bekommt dann der Professor eine Vorlesung, so müsste man diesen Dummy-Eintrag ändern, bevor man einen Neuen anlegt. Es ist also eine Fallunterscheidung für die erste und weitere Vorlesungen erforderlich.

Ändert sich das Büro eines Professors (pRaum), müssen sämtliche Tupel, in denen der Professor vorkommt, geändert werden.

Wird eine Vorlesung gelöscht, so muss man aufpassen, dass nicht auch die Daten des Professors mitgelöscht werden, nämlich dann, wenn es das letzte Tupel des Professors ist, usw.

Man sieht hier sehr deutlich, dass eine ungeeignete Tabellenstruktur vorliegt, die nur Schwierigkeiten macht. Die Probleme gründen sich in diesem Fall in der gemeinsamen Speicherung verschiedener Entitäten in einer Tabelle, obwohl keine beidseitig obligatorische 1:1 Beziehung vorliegt.

Saubere versus unsaubere Datenmodelle sind Gegenstand der *Normalisierung*, für die eine ausgefeilte Theorie vorliegt und die im Folgenden gemäß einer Hierarchie von immer strengeren Normalisierungsstufen eingeführt wird.

Hinweis: In einer Datenbankanwendung ist das saubere Datenmodell weitaus wichtiger als die Programme, die darauf operieren.

Damit wir die verschiedenen Normalisierungsstufen definieren können, müssen wir zuerst einige Grundbegriffe einführen.

Definition von Begriffen

funktional abhängig

Definition: In einer Relation R(A, B) ist das Attribut B vom Attribut A funktional abhängig, wenn zu jedem Wert von A genau ein Wert von B gehört.

Aus der Kenntnis von A ergibt sich also der Wert von B. Man sagt deshalb auch "A bestimmt B" oder "A ist *Determinante* von B".

Beispiel: Aus der Kenntnis der Artikelnummer ergibt sich die Artikelbezeichnung. Umgekehrt ist aber die Artikelbezeichnung nicht ausreichend für die Kenntnis der Artikelnummer, wenn mehrere Artikel mit gleicher Bezeichnung existieren. Das Gleiche gilt für den Lieferant.

voll funktional abhängig

Definition: In einer Relation R(S1, S2, A) ist das Attribut A von den Schlüsselattributen S1 und S2 voll funktional abhängig, wenn A vom zusammengesetzten Schlüssel (S1, S2) funktional abhängig ist, nicht aber von S1 oder S2 alleine.

Diese Definition wendet den Begriff der funktionalen Abhängigkeit auf Determinanten an, die mehrstellige Schlüssel sind.

transitiv abhängig

Definition: In einer Relation R(S, A, B) ist das Attribut B vom Schlüssel S (der auch mehrstellig sein kann) transitiv abhängig, wenn A von S funktional abhängig ist, S jedoch nicht von A, und B von A funktional abhängig ist.

Beispiel: sch(Artikel): {[artnr, ..., lifnr, Lieferantenname]}

Hier ist lifnr von artnr funktional abhängig und Lieferantenname von lifnr, somit ist Lieferantenname von artnr transitiv anbhängig.

mehrwertig abhängig

Definition: In einer Relation R(A, B, C) ist das Attribut C mehrwertig abhängig von Attribut A, wenn zu einem A-Wert, für jede Kombination dieses A-Wertes mit einem B-Wert, eine identische Menge von C-Werten existieren kann.

Diese Definition wird nur für höhere Normalformen verwendet.

Erste Normalform

Definition: Eine Relation ist in erster Normalform (1NF), wenn sie keine mengenwertigen Attribute aufweist.

Jedes Attribut darf nur einen (also atomaren) Wert enthalten. Sonst können die relationalen Operatoren (z.B. JOIN) und auch Indices etc. nicht auf diese Werte angewandt werden. Eine Tabelle, die ein VARCHAR-Attribut besitzt und darin zum Beispiel eine Menge von Zahlen speichert, ist nicht in 1NF.

Es gibt aber auch SQL-Erweiterungen, die diese Forderung umgehen und mengenwertige Attribute erlauben und spezielle Operatoren dafür anbieten. Man spricht von Non-First-Normal-Form Datenbanken (NFNF oder kurz NF^2).

Zeite Normalform

Definition: Eine Relation ist in zweiter Normalform (2NF), wenn sie in 1NF ist und alle Nicht-Schlüsselattribute voll funktional vom Gesamtschlüssel abhängig sind, nicht aber von Teilen des Schlüssels.

Diese Regel kann nur dann verletzt werden, wenn ein mehrstelliger Schlüssel (Primär- oder Alternativschlüssel) vorliegt. Attribute, die von einem Teil des Schlüssels abhängig sind, müssen in einer eigenen Tabelle stehen, in der der entsprechende Schlüsselteil der Primärschlüssel ist.

Beispiel: sch(Auftrag): {[kundenr, auftragdat, artnr, bezeichnung, menge]}

ist nicht in 2NF weil *bezeichnung* von *artnr* funktional abhängig ist und *artnr* ein Teil des Primärschlüssels ist. Folgende Modifikation führt aber zu 2NF allerdings kann ein bestimmter Kunde dann nur mehr einmal pro Tag (wenn auftragdat nur auf Tage genau ist) etwas bestellen.

sch(Auftrag2): {[kundenr, auftragdat, artnr, bezeichnung, menge]}

Diese modifizierte Form hat aber immer noch die Bezeichnung redundant gespeichert.

Dritte Normalform

Definition: Eine Relation ist in dritter Normalform (3NF), wenn sie in 2NF ist und keine transitiven Abhängigkeiten aufweist.

Es sind keine funktionalen Abhängigkeiten zwischen Nicht-Schlüsselattributen erlaubt. Eine Aufsplittung in mehrere Tabellen ist erforderlich.

sch(Auftrag3): {[kundenr, auftragdat, artnr, menge]}

sch(Artikel): {[artnr, bezeichnung, ...]}

Hinweis: Falls die Anforderung existiert, dass bei Auftragserfassung die derzeit gültige Artikelbezeichnung eingefroren werden soll, damit auf einer späteren Rechnung die bei Auftragserfassung gültige Bezeichnung verwendet wird und nicht eine allfällig in der Zwischenzeit geänderte, so würde Auftrag2 in 3NF sein. Man sieht daraus, dass die konkreten Anforderungen bei der Beurteilung von funktionalen Abhängigkeiten die zentrale Rolle spielen.

BCNF

Definition: Eine Relation ist in der Boyce/Codd Normalform (BCNF), wenn jede Determinante ein Candidate Key ist.

Attribute dürfen nur vom Primärschlüssel oder einem alternativen Schlüssel funktional abhängig sein. Diese Forderung deckt sich mit dem intuitiven Begriff der Redundanzfreiheit und ist für die Praxis wahrscheinlich die wichtigste. Höhere Normalformen waren lange Zeit ein beliebtes Forschungsthema, haben aber in der Praxis keine vergleichbare Bedeutung erlangt.

Ein Beispiel für eine Relation, die in 3NF ist aber nicht in BCNF, ist folgende Einwohnertabelle für Orte in einem Land, das nach Bundesländern gegliedert ist.

sch(Ort): {[name, bl, lh, ew]}

Unter der Annahme, dass Ortsnamen pro Bundesland (bl) eindeutig sind und alle Landeshauptleute (lh) eindeutige Namen haben ergeben sich zwei alternative Schlüssel:

k1 = {name, bl} oder k2 = {name, lh}

Die Einwohnerzahl (ew) ist funktional abhängig von den Schlüsseln k1 und k2. lh ist funktional abhängig von bl und umgekehrt (im ER-Modell würde man sagen: Bundesland und Landeshauptmann/frau haben eine beidseitg obligatorische 1:1 Beziehung). Es sollte offensichtlich sein, dass hier redundante Information gespeichert wird. Jedes Orts-Tupel enthält sowohl Information über Bundesland als auch über Landeshauptmann/frau, obwohl aus der Kenntnis des einen das jeweils andere ohnehin eindeutig bestimmt ist.

Die Relation ist in 2NF, weil sie in 1NF ist (alle Attribute sind atomar) und weil alle nicht-Schlüsselattribute (hier nur ew) voll funktional abhängig sind vom Schlüssel. Die Abhängigkeit von lh vom Schlüsselteil bl spielt in der 2NF keine Rolle, weil lh ein Schlüsselbestandteil ist.

Die Relation ist in 3NF weil sie nur ein nicht-Schlüsselattribut (ew) aufweist und damit keine transitiven Abhängigkeiten via andere nicht-Schlüsselattribute aufweisen kann.

Die Relation ist nicht in BCNF, weil bl Determinante von lh ist, bl aber kein Candidate Key ist, sondern nur Teil eines Schlüssels.

Somit sehen wir an Hand dieses Beispiels, dass 2NF und 3NF Forderungen darstellen, die an der Idee der redundanzfreien Speicherung von Daten knapp vorbei gehen, nämlich dann, wenn Redundanzen in Schlüsselbestandteilen enthalten sind.

Eine Formulierung für dieses Beispiel, die in BCNF ist, ergibt sich aus der Zerlegung in zwei Tabellen

sch(Ort): {[name, bl, ew]}

sch(Bundesland): {[bl, lh]}

Dadurch wird die redundante Speicherung von lh in jedem Orts-Tupel vermieden. Man bekommt zwar eine neue Tabelle zusätzlich, spart sich aber das Attribut lh in tausenden von Orts-Tupel ohne dass Information verloren geht.

Vierte Normalform

Eine Relation ist in vierter Normalform (4NF), wenn sie in 3NF ist und keine paarweise auftretenden mehrwertigen Abhängigkeiten enthält.

Fünfte Normalform

Eine Relation ist in fünfter Normalform (5NF), wenn sie nicht durch eine Verschmelzung einfacherer (d.h. mit weniger Attributen ausgestatteten) Relationen mit unterschiedlichen Schlüsseln rekonstruiert werden kann.

Zusammenfassung (ohne Beweis):

1NF ⊃ 2NF ⊃ 3NF ⊃ BCNF ⊃ 4NF ⊃ 5NF

Surrogat

Darunter versteht man ein Attribut, das nicht Teil einer Entität ist, sondern zum Zweck der eindeutigen und unveränderlichen Identifizierung einer Entität zusätzlich eingeführt wird (künstlicher Primärschlüssel). Surrogate haben die Eigenschaft, dass sie während der gesamten Lebensdauer einer Entität garantiert *unverändert* bleiben. Dadurch eignen sie sich bestens für die Verwendung als Fremdschlüssel. Würde ein UPDATE auf einem Primärschlüsselattribut erfolgen, so müsste man alle Tupel mit Fremdschlüssel auf den geänderten Primärschlüssel ebenfalls ändern, was man natürlich vermeiden will. Da Surrogate unveränderlich sind, ist die Änderung von Fremdschlüsseln kein Thema.

In unserem Beispiel der Artikel und Lieferanten ist die Artikelnummer und die Lieferantennummer typischerweise ein Surrogat.

Die meisten RDBMS unterstützen Surrogate durch spezielle Attributtypen (IDENTITY). Dabei kann bei INSERT ein künstlicher Schlüsselwert, der durch eine fortlaufende Nummerierung pro Tabelle erzeugt wird, automatisch vergeben werden.

Manche RDBMS unterstützen auch datenbankweit eindeutige Surrogate oder sogar global (weltweit) eindeutige Surrogate, wobei letztere durch Zusammensetzung verschiedener Bestandteile wie Netzwerkkartennummer, Systemzeit, Zufallszahlen und laufende Nummern gebildet werden.

Wenn man RDBMS-Abhängigkeiten vermeiden will, muss man sich eine Hilfstabelle anlegen, die die Zähler für die Surrogate enthält. Die Erzeugung eines neuen Surrogats muss serialisert sein, also in einer Transaktion mit geeignetem Isolationslevel erfolgen, damit Eindeutigkeit garantiert ist.