# ACCESS Objekt 1: Tabellen

In den Tabellen einer Datenbank werden Daten gespeichert, d.h. nur die 'Rohdaten', keine Berechnungen oder Formatierungen. Nur der Programmierer sieht die Tabellen, nie der Benutzer.

Im Gegensatz zu einer Tabellenkalkulation besteht eine Datenbank nicht aus nur einer oder wenigen Tabellenblättern, sondern es fallen meistens ziemlich viele Tabellen an. Zwischen diesen bestehen **Beziehungen**. Die Verteilung der Daten auf mehrere Tabellen stellt die eigentliche Stärke einer Datenbank dar. Dadurch soll verhindert werden, dass Informationen mehrfach abgespeichert werden bzw. „redundante Daten“ entstehen.

Wofür jeweils Tabellen bzw. Felder gebraucht werden, legt die **Datenbanktheorie** (s.S.4) fest. Dabei handelt es sich um die „Normalisierung der Datenbank“, der eine gute Datenbank entspricht. In der Praxis ist es sinnvoll, die Normalisierung vorab, und zwar am besten auf Papier zu entwerfen. Eine gute Datenbank entsteht zuerst im Kopf!

## 1. Zusammenspiel mehrerer Tabellen

Stellen wir uns vor, wir wollen in einer Datenbank die Namen aller Orte in Deutschland speichern. Dabei interessiert uns auch, in welchem Bundesland ein Ort liegt. Würden wir nun in einer Tabelle ein Feld `txtOrt` und ein Feld `txtLand` anlegen, wäre es nur eine Frage der Zeit, bis verschiedene Schreibweisen ein und desselben Bundeslands in der Tabelle auftauchen. Besser also, wir erstellen eine Tabelle namens `tabOrte` und eine weitere Tabelle `tabLaender`. Hier die Tabelle `tabLaender`:

| tabLaender: Tabelle | |
|---|---|
| **IDLand** | **txtLand** |
| 1 | Baden-Württemberg |
| 2 | Bayern |
| 3 | Berlin |
| ... | ... |
| 16 | Thüringen |

In `tabOrte` tragen wir die Orte und die Nummer des Bundeslands ein:

| tabOrte : Tabelle | | |
|---|---|---|
| **IDOrt** | **txtOrt** | **lngLand** |
| 1 | München | 2 |
| 2 | Bamberg | 2 |
| 3 | Berlin | 3 |
| 4 | Stuttgart | 1 |
| 5 | Bayreuth | 2 |

Die Tabelle `tabLaender` enthält ein Feld `IDLand`, `tabOrte` dagegen ein Feld `lngLand`. Das Nummerieren überlassen wir dabei der Datenbank:

`IDLand` ist vom Typ **Autowert**.

Trägt man im Feld `txtLand` einen neuen Datensatz ein, wird `IDLand` automatisch gefüllt. In der Tabelle `tabOrte` muss dann das Feld, in dem das Bundesland gespeichert wird, den gleichen Zahlenumfang speichern können. **Autowert** entspricht insofern dem Datentyp Long Integer.

Je nach Zweck einer Datenbank können noch viele Tabellen hinzukommen. In unserem Beispiel könnte man z.B. noch Tabellen benötigen, um Straßen zu erfassen. Oder man will nicht nur deutsche Orte abbilden; dann bräuchte man noch eine weitere Tabelle für die Nationen.

Würden wir weitere Angaben zum Bundesland benötigen (z.B. die Landeshauptstadt) würde das ebenfalls in `tabLaender` gespeichert.

**<u>Definition:</u> Eine ACCESS-Tabelle ist eine Informationssammlung gleichartiger Objekte, und jedes Feld speichert eine Eigenschaft des Objekts.**

## 2. Schlüssel

Schauen wir uns die Tabelle `tabOrte` noch genauer an:

Wäre `IDOrt` nicht vom Typ Autowert, sondern Zahl, könnte die gleiche Zahl mehrfach vorkommen! Und in `txtOrt` kann man mehrfach die gleiche Stadt eintragen.

Hier kommen **Schlüssel** oder auch **Indizes** ins Spiel: Ein indiziertes Feld wird schneller durchsucht und sortiert als ein nicht indiziertes Feld. Zudem lässt sich einstellen, ob in einem Index Duplikate erlaubt sind. Wir brauchen also je einen Index ohne Duplikate für die Felder `IDOrt` und `txtOrt`. Jetzt sind Duplikate ausgeschlossen! Will man danach z.B. mehrfach 'Frankfurt' eingeben, muss man zwischen 'Frankfurt am Main' und 'Frankfurt an der Oder' unterscheiden.

Jetzt noch einmal zu `tabLaender`:

`IDLand` enthält Werte, die wir auch in der Tabelle `tabOrte` brauchen. Daher ist dies kein normaler Index, sondern der **Primärschlüssel** der Tabelle. Jede Tabelle sollte einen Primärschlüssel haben. Er bezeichnet eindeutig einen bestimmten Datensatz. Deswegen gibt es auch das Feld `IDOrt` in der Tabelle `tabOrte`: Wir brauchen es in unserem Beispiel zwar nicht, aber mit dem Primärschlüssel wird unsere Datenbank insgesamt schneller.

Der Schlüssel setzt den Primärschlüssel

| | Feldname | Felddatentyp |
|---|---|---|
| 🔑 | IDOrt | AutoWert |
| | txtOrt | Text |
| | lngLand | Zahl |

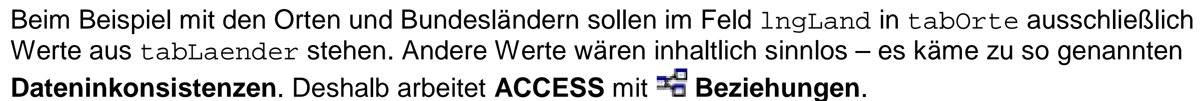
Der Index-Button öffnet das Index-Fenster, in dem man weiterere Indizes setzen kannn

**Indizes: tabOrte**

| | Indexname | Feldname | Sortierreihenfolge |
|---|---|---|---|
| | IDOrt | IDOrt | Aufsteigend |
| 🔑 | PrimaryKey | IDOrt | Aufsteigend |
| | ZweiterSchluessel | txtOrt | Aufsteigend |

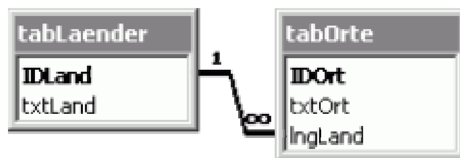
Es sind auch **zusammengesetzte Schlüssel** möglich, also Schlüssel, die sich über mehrere Felder erstrecken. Dazu wird ein Index auf das erste Feld erstellt, und in der nächsten Zeile im Index-Fenster in der Spalte **Feldname** das zweite Feld ausgewählt, ohne die Spalte mit dem Indexnamen auszufüllen.

## 3. Beziehungen

Beim Beispiel mit den Orten und Bundesländern sollen im Feld `lngLand` in `tabOrte` ausschließlich Werte aus `tabLaender` stehen. Andere Werte wären inhaltlich sinnlos – es käme zu so genannten **Dateninkonsistenzen**. Deshalb arbeitet **ACCESS** mit  **Beziehungen**.

Eventuell muss man im sich öffnenden Fenster **Tabellen anzeigen** wählen, um aus einer Liste aller Tabellen `tabOrte` und `tabLaender` auswählen zu können und so im bisher leeren Beziehungsfenster einzufügen.

Dort sieht man dann die beiden Tabellen, wobei die Primärschlüssel **fett** dargestellt sind. Nun ziehen wir aus `tabLaender` das Feld `IDLand` auf `lngLand` in `tabOrte`. Damit wird zwischen diesen Feldern eine Beziehung hergestellt:




Man spricht von einer **1:n-Beziehung**, oder auch 1:∞ (eins zu unendlich)-Beziehung.

Ein Wert darf auf der 1–Seite der Beziehung nur ein einziges Mal vorkommen, auf der n–Seite dagegen beliebig oft.

Für Beziehungen gibt es weitere Einstellungen. Besonders wichtig ist die **referentielle Integrität**, die auf der n–Seite nur Werte erlaubt, die es auch auf der 1–Seite gibt. Inkonsistente Daten sind nicht mehr möglich. Zur referentiellen Integrität kann man auch einstellen, was passiert, wenn man auf der 1–Seite Daten ändert oder löscht: Löscht man z.B. bei erlaubter **Löschweitergabe** einen Datensatz auf der 1–Seite, werden auch alle auf der n–Seite zugehörigen Datensätze gelöscht. Würde man in unserem Beispiel also ein Bundesland löschen, würden zugleich alle zugehörigen Orte gelöscht. Ohne erlaubte Löschweitergabe kann ein Datensatz auf der 1–Seite nur gelöscht werden, wenn es auf der n–Seite keinen zugehörigen Datensatz (mehr) gibt.

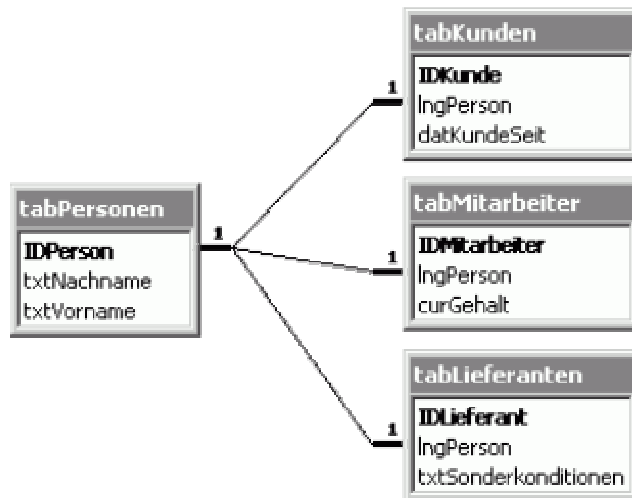
### n:m-Beziehung

Bei einer 1:n-Beziehung zwischen zwei Tabellen steht einen Seite der Beziehung nur ein, auf der anderen Seite dagegen beliebig viele Datensätze. Es gibt aber auch den Fall, dass es auf beiden Seiten beliebig viele Datensätze gibt. Dies nennt man **n:m–Beziehung**. Ein einfaches Beispiel: Ein Unternehmen hat Kunden, die Produkte bestellen. Ein Kunde kann mehrere Produkte bestellen, und ein Produkt kann von mehreren Kunden bestellt worden sein.

Dafür benötigt man neben den Tabellen `tabKunden` und `tabProdukte` noch eine 'Zwischentabelle', die z.B. `tabBestellungen` genannt werden könnte. Damit kann ein Kunde beliebig viele Bestellungen haben, und ebenso kann ein Produkt beliebig oft bestellt worden sein. Die n:m–Beziehung ist somit in zwei 1:n–Beziehungen aufgelöst:

**1:1-Beziehung**

Nehmen wir an, ein Unternehmen hat eine Tabelle mit Adressen aller Personen, mit denen es zu tun hat. Eine Person kann Kunde, Mitarbeiter oder Lieferant sein - oder mehreres davon.




Hier liegen drei 1:1-Beziehungen vor.

Für 1:1-Beziehungen müssen die zu verknüpfenden Felder auf beiden Seiten einen eindeutigen Schlüssel haben. `lngPerson` braucht also in allen Tabellen, in denen es vorkommt, einen Index ohne Duplikate.

Man sieht hier, warum man Personendaten separat speichern sollte:

Wie hoch das Gehalt der Mitarbeiter ist, oder welche Sonderkonditionen Lieferanten haben, geht immer nur einen bestimmten Teil der Mitarbeiter an. Wären umgekehrt die Adressen jeweils in den drei anderen Tabellen gespeichert, hätte man mehrfach gespeicherte (redundante) Daten und die Gefahr unterschiedlicher (inkonsistenter) Angaben. Man könnte nur schwer prüfen, ob ein Mitarbeiter z.B. auch Lieferant ist.

## 4. Normalformen

Bisher haben wir uns mit dem 'Handwerkszeug' einer relationalen Datenbank beschäftigt. Jetzt lernen wir etwas **Datenbanktheorie**, die bisher zu kurz gekommen ist. Das **Normalisieren einer Datenbank** verhindert doppelte (redundante) Daten. Eine Datenbank soll gemäß folgender sog. 'Normalformen' Schritt für Schritt geplant werden. Die Datenbanktheorie kennt noch weitere Normalformen, aber in der Praxis kommen Datenstrukturen, die eine weitergehende Normalisierung erforderlich machen, weniger häufig vor.

### Nullte Normalform

Eine Tabelle soll keine Felder enthalten, die aus anderen Feldern berechnet werden können (dafür sind „Abfragen“ zuständig). Sehen wir uns folgende Tabelle an:

| Verkaeufe | | | | |
|---|---|---|---|---|
| **VerkaufsNr** | **Verkaufsdatum** | **Nettopreis** | **MwSt** | **Bruttopreis** |
| 1 | 19.12.2007 | 100 | 19% | 119 |
| 2 | 05.12.2010 | 50 | 12% | 60 |

Während der erste Datensatz in sich stimmig scheint, ist der Mehrwertsteuersatz des zweiten Datensatzes definitiv falsch, auch der Bruttopreis. MwSt und Bruttopreis gehören nicht in die Tabelle, denn anhand des Datums kann der Mehrwertsteuersatz und damit auch der Bruttopreis errechnet werden.

**Erste Normalform**

Die erste Normalform bestimmt, wofür jeweils ein Feld benötigt wird: Eine Tabelle befindet sich in der ersten Normalform, wenn die Werte in jedem Feld und in jedem Datensatz **atomar** sind, d.h. sich nicht mehr in kleinere Einheiten zerlegen lassen. Hier ein Negativbeispiel:

| Personen | | |
|---|---|---|
| **IDKunde** | **Name** | **Anschrift** |
| 1 | Bilbo Beutlin | Beutelsend 1; 12345 Hobbingen |
| 2 | Petra Roth | Berliner Str. 5a), 60123 Frankfurt am Main |

Name und Anschrift lassen sich noch weiter zerlegen, nämlich in Vorname, Nachname, Straße, HausNr, PLZ und Ort. Wollte man aus dieser Tabelle alle Personen mit einer bestimmten Postleitzahl abfragen, wäre man vor eine unlösbare Aufgabe gestellt: Mal befindet sich vor der Postleitzahl ein Semikolon, mal ein Punkt, und den Benutzern würden bestimmt noch ein paar andere lustige Trennzeichen einfallen - falls sie überhaupt an eine PLZ denken...

**Zweite Normalform**

Eine Tabelle befindet sich in der zweiten Normalform, wenn sie der ersten Normalform entspricht und darüber hinaus jeder Datensatz nur Felder enthält, die sich auf das Objekt beziehen, das durch den **Primärschlüssel** dargestellt wird. In der folgenden Beispieltabelle gibt es zwar ein Primärschlüsselfeld (IDKunde). Dieses hat aber keinen Bezug zu den Autokennzeichen.

| Werkstattkunden | | | |
|---|---|---|---|
| **IDKunde** | **Nachname** | **Auto-KZ1** | **Auto-KZ2** |
| 1 | Junker | F J-2000 | F J-2001 |
| 2 | Becker | SB AF-123 | |

Bei den Kunden haben die Auto-KZ nichts zu suchen. Gemäß der zweiten Normalform müssen sie aus der vorgenannten Tabelle entfernt werden und gehören in eine eigene Tabelle. Nur so können mehr als zwei Kennzeichen pro Kunde dargestellt werden. `Auto-KZ` ist das Primärschlüsselfeld der neuen Tabelle.

| PKWs | |
|---|---|
| **IDKunde** | **Auto-KZ** |
| 1 | F J-2000 |
| 1 | F J-2001 |
| 2 | SB AF-123 |

**Dritte Normalform**

Eine Tabelle befindet sich in der dritten Normalform, wenn sie der zweiten Normalform entspricht und darüber hinaus alle Felder nur einmal vorkommen und alle Felder, die nicht den Primärschlüssel bilden, voneinander unabhängig sind.

Hier ein weiteres Negativbeispiel:

| Orte | | | |
| --- | --- | --- | --- |
| **IDOrt** | **txtOrt** | **lngLand** | **txtLand** |
| 1 | München | 2 | Bayern |
| 2 | Bamberg | 2 | Baiern |
| 3 | Berlin | 3 | Berlin |

Hier wurde nicht nur die Nummer des Bundeslandes, sondern auch noch der Klartext dazu abgelegt. `txtLand` ist aber von `lngLand` abhängig, also wurde gegen die dritte Normalform verstoßen. Deshalb konnte sich auch der Tippfehler bei 'Baiern' einschleichen.

## 5. Tipps & Tricks zu Tabellen

Üblicherweise sollte eine Access-Datenbank aus zwei Dateien bestehen:

1. einer **Backend-Datei** mit den Tabellen und Beziehungen, und
2. einer **Frontend-Datei** mit den übrigen Objekten.

Wenn man die Anwendung später einmal weiterentwickelt, betrifft das selten das Backend. Ist eine neue Version fertig, wird nur die Frontend-Datei ersetzt; die Daten stehen weiter zur Verfügung. In einer Mehrbenutzerumgebung bekommt jeder Benutzer eine Kopie des Frontends, wodurch die Anwendung schneller wird.

**Autowert** ist für interne Zwecke nützlich, aber selten geeignet, um ihn de
man nämlich das Anlegen eines neuen Datensatzes abbricht, so wurde dafür schon eine Nummer angelegt, die nicht ohne weiteres wieder verwendet werden kann. Es kann also zu Lücken in der Nummerierung kommen.

Im Datentyp **OLE-Objekt** kann Access auch beliebige Dateien, wie z.B. Bilder, in einer Tabelle speichern. Das bläht aber die Datenbank enorm auf und ist nicht sehr perfomant. Stattdessen sollte man die Dateien in einem Unterverzeichnis speichern und in der Tabelle nur ein Textfeld mit dem Speicherort anlegen.

Der Datentyp **Nachschlagefeld** ist überflüssig, entspricht nicht der Datenbanklehre und sollte niemals verwendet werden.

Tabellenfelder sollte man **nicht formatieren**. Die Formatierung erfolgt erst später in den Formularen und Berichten. Gerade bei Werten, mit denen später gerechnet werden soll, kann es sonst nämlich beim Betrachten der Werte zu Verwirrung kommen.