Tabellen

Relationale Datenbanken

- Relationale Datenbanken verwenden Tabellen, um Daten zu strukturieren.
- Tabellen werden mit der create-table-Anweisung erzeugt.
- In diesem Kapitel wird diese SQL-Anweisung detailliert diskutiert.

Einfache Tabellen

Es wird eine Tabelle namens `spielkarten` mit den Spalten `farbe` und `karte` erzeugt.

In beiden Spalten stehen Texte mit maximal 20 Buchstaben.

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20)
)
```


Der Datentyp ist wichtig

Die Wahl des geeigneten Datentyps kann erheblichen Einfluss auf die Qualität der Daten haben:

```
create table personen(
  name varchar(20),
  telefon int
)
```

Die Spalte `telefon` ist für die Telefonnummer einer Person vorgesehen. Als Datentyp wurde `int` gewählt. Beliebige Texte sind so – anders als bei `varchar` - nicht möglich.

Eine gute Wahl?

Es werden Datensätze für drei Personen eingefügt:

- Donald mit einer Rufnummer im lokalen Ortsnetz, also ohne führende 0,
- Daisy, die in einer Stadt mit Vorwahl 04711 lebt, und
- Dagobert, der in ein Land mit der internationalen Vorwahl 0047 umgezogen ist.

Der Datentyp ist wichtig

Die Wahl des geeigneten Datentyps kann erheblichen Einfluss auf die Qualität der Daten haben:

```
insert into personen values('Donald', 471123815);
insert into personen values('Daisy', 0471123815);
insert into personen values('Dagobert', 00471123815)

select *
from personen
```

name	telefon
Donald	471123815
Daisy	471123815
Dagobert	471123815

Der Datenbestand nicht korrekt! Der Typ `int` war die falsche Wahl.

Welcher Typ ist geeignet?

- Im vorliegenden Fall wäre ein Datentyp wie `varchar` besser geeignet.
- Nachteil: Die folgende Anweisung wäre möglich:

```
insert into personen values('Donald', 'ich bin keine Nummer');
```


Konsistenz

- Logisch korrekte Daten werden auch als *konsistent* bezeichnet.
- Egal welchen der beiden Typen `int` und `varchar` wir für die Spalte `telefon` verwenden: Inkonsistenzen sind möglich.
- Der Datentyp *alleine* reicht offenbar nicht, um die Konsistenz der Daten zu gewährleisten.
- Wünschenswert sind zusätzliche Regeln, die die Konsistenz sicherstellen.
- Solche Regeln werden *Integritätsregeln* genannt.
- Mit SQL können Integritätsregeln definiert werden.
- *Das DBMS überwacht und garantiert die Einhaltung der Regeln!*

Integritätsregeln

Mit einer Integritätsregel könnte man formulieren, dass nur gültige Telefonnummern (etwa Ziffern, Leerzeichen, Bindestriche) zulässig sind.

Bei Daten über Mitarbeiter könnte man sicherstellen, dass

- Negative Gehälter nicht zulässig sind
- das Datum ihres Austritts aus der Firma zeitlich hinter dem Datum der Einstellung liegt.

Die Syntax dazu lernen wir später.

Dubletten

In Tabellen sind doppelte Datensätze möglich. Für die Tabelle `spielkarten` werden die beiden folgenden Anweisungen fehlerfrei ausgeführt:

```
insert into spielkarten values('Karo', 'Ass');
insert into spielkarten values('Karo', 'Ass');
```

In einem Spiel, in dem es jede Karte nur einmal gibt, wäre der Datenbestand inkonsistent.

Unsere erste Integritätsregel

Mit der Regel `unique` kann in SQL spezifiziert werden, dass in einer Spalte *keine Dubletten* auftreten dürfen.

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  unique(farbe)
)
```

Warum schlägt die zweite der folgenden Anweisungen fehl?

```
insert into spielkarten values('Karo', 'Ass');
insert into spielkarten values('Karo', '7');
```


Integritätsregeln

- Wir haben die Spalte `farbe` als `unique` definiert.
- Ab jetzt *garantiert* das DBMS, dass keine Farbe mehr als einmal in der Tabelle auftaucht.
- Verstöße gegen Integritätsregeln werden vom DBMS mit einer Fehlermeldung quittiert.

Spaltenkombinationen in Integritätsregel

Doppelte Karten können wie folgt vermieden werden:

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  unique(farbe, karte)
)
```

Mit `unique` können also Dubletten in Spalten oder *Spaltenkombinationen* verboten werden.

Constraints

- Das Schlüsselwort `unique` ist eine der vielen Möglichkeiten eine Integritätsregel zu definieren.
- Im Zusammenhang mit SQL werden die Integritätsregeln auch als *Constraints* bezeichnet.

Varianten

Die create table-Anweisung hat einen enormen Variantenreichtum. Was könnte sich hinter dieser Anweisung verbergen?

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  constraint farbe_eindeutig unique(farbe)
)
```


Integritätsregeln löschen

Mit Hilfe der Anweisung `alter table` kann die Struktur von Tabellen geändert werden.

Es können etwa Spalten oder Constraints hinzugefügt oder gelöscht werden:

```
alter table spielkarten drop constraint farbe_eindeutig ;
```

Um die Regel zu löschen, benötigen wir Ihren Namen. Was ist, wenn wir keinen Namen vergeben haben?

Der Systemkatalog

Jede Constraint hat einen Namen. Wenn der Anwender keinen Namen vergibt, vergibt das DBMS implizit einen Namen.

Die Namen findet man im Systemkatalog:

```
select *
from
information_schema.table_constraints
```


Integritätsregeln hinzufügen

Oft zeigt sich, dass bestimmte Constraints nötig sind, *nachdem* die Tabelle erzeugt ist:

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20)
);
alter table spielkarten add constraint ukarten unique(farbe);
```

Vorsicht!

Was passiert wenn man versucht, die folgenden vier SQL-Anweisungen auszuführen?

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20)
);
insert into spielkarten values('Karo', 'Ass');
insert into spielkarten values('Karo', '7');
alter table spielkarten add constraint ukarten unique(farbe);
```


Eine einfache Variante

Wenn Dubletten nur für *eine* Spalte - und nicht für eine Kombination – vermieden werden sollen, bietet SQL eine einfache Variante:

```
create table spielkarten(
  farbe varchar(20) unique,
  karte varchar(20)
)
```

Aufgabe

Was ist eigentlich der Unterschied zwischen den beiden folgenden Anweisungen?

```
create table spielkarten(
  farbe varchar(20) unique,
  karte varchar(20) unique
)

create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  unique(farbe, karte)
)
```


Datensätze finden

Wenn eine Tabelle wie folgt definiert ist:

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  unique(farbe, karte)
)
```

Kann man einen Datensatz *eindeutig* in der Tabelle finden, egal wie viele Datensätze sie enthält:

```
select *
from spielkarten
Where farbe='Karo' and karte='Ass'
```

Schlüssel

- Unique-Constraints verbieten also nicht nur Dubletten für Spalten oder Spaltenkombinationen, sie sind auch ein Instrument um Datensätze *eindeutig zu identifizieren*.
- Spalten oder Kombinationen von Spalten, die Datensätze eindeutig identifizieren, werden auch *Schlüssel* genannt.
- In Tabellen kann es mehrere Schlüssel geben.

Beispiel

- Die folgende Tabelle enthält die Sitzordnung für einen Hörsaal während einer Klausur.
- Welche Schlüssel gibt es?

name	matrikel	reihe	platz
Daniel	4711	1	4
Donald	0815	1	9
Daniel	2342	5	4
...			

Schlüssel

Es ist oft nicht einfach, Schlüssel zu finden.

Meistens benötigt man zusätzliche Informationen.

Im Beispiel setzen wir voraus, dass

- es *niemals* doppelte Matrikelnummern gibt
- ein Sitzplatz durch die Kombination `reihe` und `platz` identifiziert ist
- ein Sitzplatz *niemals* von mehreren Personen besetzt ist.

Schlüssel werden durch die *Anwender* definiert, da das DBMS diese Informationen nicht hat.

Primärschlüssel

Unter allen möglichen Schlüsseln wird einer ausgewählt und zum Primärschlüssel (`primary key`) befördert.

```
create table spielkarten(
  farbe varchar(20),
  karte varchar(20),
  primary key(farbe, karte)
)
```

Eine einfache Variante

Wenn der Primärschlüssel nur eine Spalte enthält, bietet SQL eine einfache Variante:

```
create table spielkarten(
  farbe varchar(20) primary key,
  karte varchar(20)
)
```


Höchstens Einer!

Es kann je Tabelle nur einen Primärschlüssel geben. Die folgende Anweisung schlägt fehl:

```
create table spielkarten(
  farbe varchar(20) primary key,
  karte varchar(20) primary key
)
```

Primärschlüssel – Immer!

- SQL erzwingt nicht die Definition eines Primärschlüssels.
- Da Schlüssel aber die einzige Möglichkeit sind, um Datensätze zu identifizieren, gilt es als Designfehler, Tabellen ohne Primärschlüssel zu definieren.
- Definieren Sie immer einen Primärschlüssel!
- Von dieser Regel gibt es in *fortgeschrittenen* Anwendungen Ausnahmen.

Aufgabe

spielkarten		
farbe	karte	spieler_id
Herz	Ass	0
Pik	7	2
Pik	B	2
Karo	Ass	3

spieler	
id	name
0	Daniel
1	Donald
2	Daisy
3	Daniel

In den Tabellen sind die Primärschlüssel unterstrichen. Verstehen Sie die Bedeutung der Tabellen? Wie sehen die zugehörigen SQL-Anweisungen aus?

Definition der Tabellen

```
create table spielkarten(
       farbe varchar(20),
       karte varchar(20),
       spieler_id int,
       primary key(farbe, karte)
);
create table spieler(
       id int primary key,
       name varchar(20)
);
```

Die Tabelle `spielkarten` soll sich auf die Tabelle `spieler` beziehen. Zu jeder Spielkarte soll es einen Spieler geben.

Daten einfügen

```
insert into spielkarten values('Herz', 'Ass', 0);
insert into spielkarten values('Pik', '7', 2);
insert into spielkarten values('Pik', 'B', 2);
insert into spielkarten values('Karo', 'Ass', 3);
insert into spieler values(0,'Daniel');
insert into spieler values(1,'Donald');
insert into spieler values(2,'Daisy');
insert into spieler values(3,'Daniel');
```

Aufgabe

spielkarten		
farbe	karte	spieler_id
Herz	Ass	0
Pik	7	2
Pik	B	2
Karo	Ass	3

spieler	
id	name
0	Daniel
1	Donald
2	Daisy
3	Daniel

Was passiert, wenn wir versuchen die folgende Anweisung auszuführen?

```
insert into spielkarten values('Herz', '8', 4711);
```


So nicht...

- Die `insert`-Anweisung wird ausgeführt, obwohl es keinen Spieler mit der id 4711 gibt!
- Gibt es eine Regel, mit der man sicherstellen kann, dass es immer einen passenden Spieler gibt?

Die Definition der Regel

```
create table spieler(
       id int primary key,
       name varchar(20)
);
create table spielkarten(
       farbe varchar(20),
       karte varchar(20),
       spieler_id int,
       primary key(farbe, karte),
       foreign key(spieler_id) references spieler(id)
);
```


Die Regel wirkt

```
insert into spieler values(0,'Daniel');
insert into spieler values(1,'Donald');
insert into spieler values(2,'Daisy');
insert into spieler values(3,'Daniel');
insert into spielkarten values('Herz', 'Ass', 4711);
```

Die letzte `insert`-Anweisung wird *nicht* ausgeführt. Das DBMS erkennt, dass es keinen Spieler mit der id 4711 in der Tabelle `spieler` gibt.

Referenzielle Integrität

- Die Tabelle `spielkarten` referenziert den Primärschlüssel der Tabelle `spieler`.
- Diese Art der Integrität wird auch als *referenzielle Integrität* bezeichnet.
- Die Spalte `spieler_id` heißt *Fremdschlüssel*.

Referenzielle Integrität

- In der Regel referenziert der Fremdschlüssel den *Primärschlüssel* einer anderen Tabelle.
- Es reicht aber, wenn die referenzierte Spalte ein Schlüssel (also `unique`) ist.
- Selbstverständlich *kann* ein Fremdschlüssel auch aus mehreren Spalten zusammengesetzt sein.

Varianten

Wenn der Fremdschlüssel einen Primärschlüssel referenziert, muss die Primärschlüsselspalte nicht explizit angegeben werden:

```
create table spielkarten(
       farbe varchar(20),
       karte varchar(20),
       spieler_id int,
       primary key(farbe, karte),
       foreign key(spieler_id) references spieler
);
```


Noch einfacher

Wenn der Fremdschlüssel zusammen mit der Spalte definiert wird, können die Schlüsselworte `foreign key` entfallen:

```
create table spielkarten(
       farbe varchar(20),
       karte varchar(20),
       spieler_id int references spieler,
       primary key(farbe, karte)
);
```

Die Reihenfolge

Die Reihenfolge der `create table`-Anweisungen ist wichtig: Die referenzierte Tabelle muss existieren, *bevor* sie referenziert wird.

Referenzielle Integrität

Schlüssel-Fremdschlüssel-Beziehungen stellen sicher, dass es zu jedem Fremdschlüssel einen Schlüssel in der referenzierten Tabelle gibt.

Die Definition von Schlüssel-Fremdschlüssel-Beziehungen liefert also einen weiteren Beitrag zur Konsistenz der Daten und stellt somit eine Integritätsregel dar.

Diese Form der Integrität wird als *referenzielle Integrität* bezeichnet.

Referenzielle Integrität – Noch ein Beispiel

Machen Sie sich mit den beiden folgenden Tabellendefinitionen vertraut.

```
create table farben(
  name varchar(20) primary key
);
create table spielkarten(
  farbe  varchar(20) references farben,
  karte varchar(20),
  primary key(farbe, karte)
)
```


Referenzielle Integrität – Noch ein Beispiel

Fügen wir einige Datensätze ein:

```
insert into farben values('Karo');
insert into farben values('Herz');
insert into farben values('Pik');
insert into farben values('Kreuz');
```

Wegen der referenziellen Integrität schlägt die zweite der beiden folgenden Anweisungen – wie erwartet - fehl:

```
insert into spielkarten values('Karo', '7');
insert into spielkarten values('Blau', 'Ass');
```

Änderungen

Welches Problem ergibt sich, wenn die Werte in der Tabelle `farben` übersetzt werden sollen?

Aus

- Karo wird Diamonds
- Herz wird Hearts
- Pik wird Spades
- Kreuz wird Clubs

Änderungen

Warum schlägt die folgende Anweisung fehl?

```
update farben
set name = 'Diamonds'
where name ='Karo'
```

Warum schlägt die folgende Anweisung fehl?

```
update spielkarten
set farbe='Diamonds'
where farbe='Karo'
```

Änderungen

Auch weitere Ideen, die das Problem der Übersetzung scheinbar lösen sind nicht zufriedenstellend.

Was ist die Ursache?

Änderungen

```
create table farben(
  farbe varchar(10) primary key
);
```

Die Spalte `farbe` ist Primärschlüssel.

Weil Primärschlüssel durch Fremdschlüssel referenziert werden können, ziehen Änderungen an Primärschlüsseln oft weitere Änderungen nach sich.

Natürliche Primärschlüssel

- Primärschlüssel sollten *unveränderbar* sein.
- Alles was Teil der realen Welt ist, ist potenziell Änderungen unterworfen - *auch wenn es zunächst oft nicht so scheint.*
- Primärschlüssel, die eine Bedeutung in der Realität haben, werden auch als *natürliche Primärschlüssel* bezeichnet.

Künstliche Schlüssel

```
create table farben(
  id int primary key,
  farbe varchar(10) unique
);
create table spielkarten(
  farb_id int references farben,
  karte varchar(20),
  primary key(farbe, karte)
)
```

Die aufgetretenen Probleme wären durch einen bedeutungsfreien Schlüssel - einen *künstlichen Schlüssel* – vermieden worden.

Die Spalte `id` ist ein künstlicher Schlüssel.

Vorsicht!

```
create table farben(
  id int primary key,
  farbe varchar(10) unique
);
```

Warum reicht es nicht, einen künstlichen Schlüssel einzuführen? Was ist noch bemerkenswert an der Tabelle?

Werte, nicht Primärschlüssel ändern

Fügen wir einige Datensätze ein:

```
insert into farben values(0, 'Karo');
insert into farben values(1, 'Herz');
insert into farben values(2, 'Pik');
insert into farben values(3, 'Kreuz');
insert into farben values(4, 'Karo');
```

Die Zahlen 0,1,2,3,4 sind die künstlichen Schlüssel. Warum schlägt die letzte Anweisung fehl?

Künstliche Schlüssel anwenden

Das Pik-Ass kann jetzt unter Ausnutzung der referenziellen Integrität eingefügt werden:

```
insert into spielkarten values(2, 'Ass');
```


Die Lösung

Die Übersetzung der Farbwerte kann einfach gelöst werden:

```
update farben set farbe= 'Diamonds' where id=0;
update farben set farbe= 'Hearts'   where id=1;
update farben set farbe= 'Spades'   where id=2;
update farben set farbe= 'Clubs'    where id=3;
```

Regel

Meiden Sie natürliche Schlüssel, nutzen Sie künstliche Schlüssel!

Weitere Probleme

Will man einen neuen Datensatz einfügen, ist es nicht ganz einfach den nächsten möglichen Wert für den künstlichen Schlüssel zu finden.

```
insert into farben values(???, 'Kreuz');
```

Hilfe für künstliche Schlüssel

Moderne RDBMS bieten Möglichkeiten, um künstliche Schlüssel automatisch zu erzeugen

```
create table farben(
  id int generated always as identity primary key,
  farbe varchar(10) unique
)
```

- Man muss/kann den Wert von `id` nicht mehr *explizit* vergeben.
- Der Wert von `id` wird *implizit* vom RDBMS vergeben.

Hilfe für künstliche Schlüssel

- Beachten Sie, dass die folgenden Anweisungen keinen Wert mehr für `id` enthalten.
- Die Spalte(n) für die Werte *explizit* vergeben werden, werden dazu nach dem Tabellennamen aufgeführt.
- Für die *nicht* genannten Spalten werden Werte *implizit* vom System vergeben.

```
insert into farben(farbe) values('Karo');
insert into farben(farbe) values('Herz');
```

Hilfe für künstliche Schlüssel

Die Werte für `id` wurden korrekt vergeben:

id	farbe
1	Karo
2	Herz

Nehmen Sie sich Zeit für die Suche nach geeigneten Integritätsregeln. Ein gutes Regelwerk wird sich schnell bezahlt machen und Ihnen die Reparatur inkonsistenter Daten ersparen.

Beispiel

Welche Maßnahmen verbessern die Konsistenz der folgenden Tabelle?

```
create table mitarbeiter(
  name varchar(20),
  geschlecht varchar(1),
  email varchar(30),
  zugehoerigkeit int,
  gehalt int
)
```


1. Maßnahme: Künstlicher Primärschlüssel

Künstliche Primärschlüssel sind Pflicht.

```
create table mitarbeiter(
  id int generated always as identity primary key,
  name varchar(20),
  geschlecht varchar(1),
  email varchar(30),
  zugehoerigkeit int,
  gehalt int
)
```

2. Maßnahme: Weitere Schlüssel

E-Mail Adressen sollen eindeutig sein.

```
create table mitarbeiter(
  id int generated always as identity primary key,
  name varchar(20),
  geschlecht varchar(1),
  email varchar(30) unique,
  zugehoerigkeit int,
  gehalt int
)
```


3. Maßnahme: Wertebereich

- Für die Firmenzugehörigkeit und das Gehalt sind ganze Zahlen vorgesehen.
- Hier ist es sinnvoll, keine negative Zahlen zu zulassen.
- Dazu bietet SQL das Schlüsselwort check, das wie folgt verwendet wird.

3. Maßnahme: Wertebereich

Keine negativen ganzen Zahlen verwenden:

```
create table mitarbeiter(
  id int generated always as identity primary key,
  name varchar(20),
  geschlecht varchar(1),
  email varchar(30) unique,
  zugehoerigkeit int check (zugehoerigkeit >=0),
  gehalt int check (gehalt>=0)
)
```


3. Maßnahme: Wertebereich

Für das Geschlecht sind alle Buchstaben möglich. Das erscheint nicht sinnvoll. Wir beschränken den Wertebereich auf 'M', 'W' und 'D'.

3. Maßnahme: Wertebereich

```
create table mitarbeiter(
  id int generated always as identity primary key,
  name varchar(20),
  geschlecht varchar(1)
      check(geschlecht = 'W' or geschlecht = 'M' or geschlecht = 'D'),
  email varchar(30) unique,
  zugehoerigkeit int check (zugehoerigkeit >=0),
  gehalt int check (gehalt>=0)
)
```


Varianten

Alternativ zu

```
check(geschlecht = 'W' or geschlecht = 'M' or geschlecht = 'D‘)
```

wäre auch knapper und klarer möglich gewesen:

```
check(geschlecht in ( 'W', 'M', 'D'))
```

Statische Integritätsregeln

Mit Hilfe von check definierte Regeln werden auch als statische Integritätsregeln bezeichnet.

Statisch oder nicht statisch

Wie können wir die statische Regel

```
check(geschlecht in ( 'W', 'M', 'D'))
```

mit einer Regel formulieren, die nicht statisch ist?

Referenzielle Integrität

```
create table geschlecht(
  id int generated always as identity primary key,
  name varchar(1) unique
);
create table mitarbeiter(
  id int generated always as identity primary key,
  name varchar(20),
  geschlecht_id int references geschlecht,
  email varchar(30) unique,
  zugehoerigkeit int check (zugehoerigkeit >=0),
  gehalt int check (gehalt>=0)
)
```


Statisch oder referenziell?

Die Konsistenz der Spalte `geschlecht` kann über

- eine statische Integritätsregel oder
- eine referenzielle Integritätsregel

gewährleistet werden. Beide Varianten haben Vor- und Nachteile.

Die referenzielle Variante ist unempfindlicher gegenüber Änderungen, während die statische Variante geringfügig schneller sein kann.

Eine klare Empfehlung kann nicht gegeben werden.