Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Datenbanken

Vorlesungsskript für das 3. Semester

Studiengang Internationale Medieninformatik

5. Mengenoperationen und Joins in SQL

Dozent: M. Sc. Burak Boyaci

Version: 04.11.2025

Wintersemester 25/26

7

Mengenoperationen

Kapitelübersicht

Mengenoperationen in SQL sind ein Mittel, um die Ergebnismenge mehrerer Datenbankabfragen zu einer einzelnen Ergebnismenge zusammenzufassen. Grundlage dafür ist die Mengenlehre.

7.1 Mengenlehre: Mathematik der Mengen

Eine *Menge* (englisch: *set*) ist eine Zusammenfassung einzelner Elemente mit wohldefinierten Eigenschaften. Die *Mengenlehre* ist das Teilgebiet der Mathematik, das sich mit der Untersuchung von Mengen beschäftigt. Die gesamte moderne Mathematik ist in der Sprache der Mengenlehre formuliert und baut auf ihren Axiomen auf.

Der Begriff der Menge wurde 1895 von dem Mathematiker Georg Cantor formuliert. Zu Beginn des 20. Jahrhunderts wurden durch (heute so genannte) „naive“ Interpretationen des Mengenbegriffs logische Widersprüche entdeckt. Ein Beispiel dafür ist das folgende Paradox, das Russel 1902 in etwas anderer Form für Mengen formulierte undbereits in der Antikebekannt war.[1]

Beispiel 7.1. *(Das Lügnerparadox)* Wenn wir bei der Eingrenzung einer Mengenlehre nicht aufpassen, können wir sehr schnell innere Widersprüche erhalten. Betrachten wir dazu die Menge W aller wahren Aussagen und speziell die Aussage

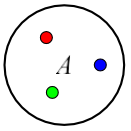
$$a = \text{„Diese Aussage ist falsch“} \tag{7.1}$$

(Beachten Sie, dass die Aussage sich auf sich selbst bezieht.) Ist die Aussage wahr oder nicht? Oder formal: Gilt $a \in W$ oder $a \notin W$? Nehmen wir an, sie ist wahr, so ist ihre Aussage aber falsch: $a \in W \Rightarrow a \notin W$. Nehmen wir umgekehrt an, sie sei falsch, so ist ihre Aussage wahr: $a \notin W \Rightarrow a \in W$. ERROR!

[1] „Ich sagte [...]: Alle Menschen sind Lügner.“ [Altes Testament, Psalm 116,11 (ca. 200 v. Chr.)]

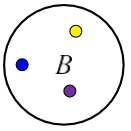
Die Mengenlehre musste historisch (durch das sogenannte „Regularitätsaxiom“) so eingegrenzt werden, dass sie selbstbezügliche Elemente nichterlaubt, also solche, diesich selbst enthalten wie in dem Beispiel. In wesentlichen Teilen gelang dies Ernst Zermelo 1907 und endgültig Abraham Fraenkel 1930, indem die Grundlagen der heute ZFC genannten Mengenlehre präzisiert wurden. Allerdings: Obwohl diese Mengenlehre das logische Fundament der modernen Mathematik bildet, kann man nicht beweisen, dass sie für unendliche Mengen in sich widerspruchsfrei ist. (Das folgt aus dem berühmten Zweiten Unvollständigkeitssatz des Mathematikers Kurt Gödel von 1931.)

In der Informatik beschäftigen wir uns jedoch ausschließlich mit *endlichen* Mengen. Hier bewegen wir uns auf sicherem Boden, denn für diesen Fall ist die Widerspruchsfreiheit der Mengenlehre bewiesen.[2] Eine endliche Menge wird in der Mathematik üblicherweise durch Aufzählen seiner Elemente innerhalb geschweifter Klammern angegeben, also beispielsweise

A = {rot, grün, blau}. 

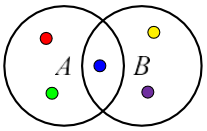
(7.2)

Betrachten wir eine zweite Menge

B = {blau, gelb, lila}, 

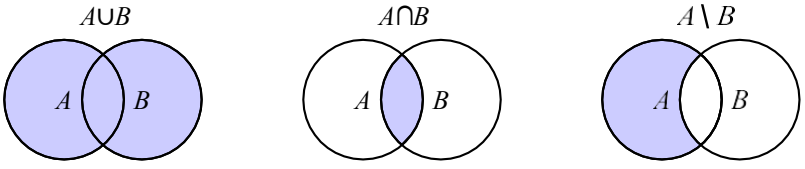
(7.3)

so können wir A und B durch ein Venn-Diagramm darstellen:



(7.4)

Entsprechend können wir die Mengenoperationen Vereinigung (∪), Schnitt (∩) und Differenz (\) visualisieren:



Es gilt also $A \cup B$ = {rot, grün, blau, gelb, lila}, $A \cap B$ = {blau} und $A \setminus B$ = {rot, grün}.

7.2 Mengenoperationen in SQL

In SQL können Ergebnismengen von Datenbankabfragen mit den Mengenoperationen **UNION**, **INTERSECT**, **EXCEPT** zu einer einzigen Ergebnismenge zusammengefügt werden. Bei diesen Mengenoperationen gelten zwei Datensätze als gleich, wenn sie dieselben Spalteneinträge haben. Mit dieser Entsprechung ist **UNION** die Vereinigung zweier Ergebnismengen, **INTERSECT** deren Schnittmenge und **EXCEPT** deren Differenz. Eine weitere Mengenoperation ist **IN**:

```
SELECT ... FROM tabelle WHERE spalte IN (x, y,...)
```

[2] https://en.wikipedia.org/wiki/General_set_theory

Sie prüft in einer **WHERE**-Klausel, ob ein Wert des Attributs spalte in der Menge {*x*, *y*, . . . } enthalten ist.

Beispiel 7.2. Betrachten wir als erstes Beispiel für Mengenoperationen in SQL den Relationstyp mengen(menge, element) mit den folgenden Einträgen:

menge	element
A	rot
A	grün
A	blau

menge	element
B	blau
B	gelb
B	lila

Das ist genau eine Datenbankdarstellung unserer obigen mathematischen Mengen *A* und *B*. In SQL erzeugen wir sie wie folgt:

```
--Tabellenstrukturen
CREATE TABLE
   mengen ( menge
   varchar(1),
   element varchar(4)
) ;
--------
--Daten:
INSERT INTO mengen (menge, element) VALUES
   ('A', 'rot'),
   ('A', 'grün'),
   ('A', 'blau'),
   ('B', 'blau'),
   ('B', 'gelb'),
   ('B', 'lila');
```

Wenden wir nun die Mengenoperationen darauf an:

```
SELECT element AS "A ∪ B" FROM mengen WHERE menge = 'A'
UNION
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT element AS "A ∩ B" FROM mengen WHERE menge = 'A'
INTERSECT
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT element AS "A \ B" FROM mengen WHERE menge = 'A'
EXCEPT
SELECT element FROM mengen WHERE menge = 'B'
--
SELECT menge, element FROM mengen WHERE element IN ('gold', 'lila', 'blau');
```

(Die Zeichen ∩ und ∪ haben die Unicodes 0x2229 und 0x222A). Dann erhalten wir jeweils die Ausgaben

$A \cup B$
lila
gelb
grün
blau
rot

$A \cap B$
blau

$A \setminus B$
grün
rot

menge	element
A	blau
B	lila
B	blau

Das entspricht genau unserem mathematischen Bild in Gleichung (7.4). □

7.3 Verschachtelte SELECT-Anweisungen: SELECT in SELECT

Eine verschachtelte **SELECT**-Anweisung ist eine Abfrage mit einer **WHERE**-Klausel auf eine andere SELECT-Anweisung, die ja auch eine – wenn auch nur temporäre – Tabelle ist. Hier wird also die Abfrage an eine Haupttabelle gestellt, wobei die WHERE-Bedingung eine Nebentabelle zum Vergleich heranzieht. Haupt- und Nebentabelle können dabei identisch sein, dürfen aber durchaus auch verschieden sein. Aus der Nebentabelle können ein oder mehrere Vergleichswerte entnommen werden. Eine solche Anweisung wird auch oft *Unterabfrage (Subquery)* oder *innere Abfrage* genannt.

```
SELECT ... FROM haupttabelle WHERE ... (
  SELECT ... FROM nebentabelle
);
```

Die Reihenfolge der Abarbeitung dieser Anweisung geht von innen nach außen (bzw. hier von unten nach oben): Zunächst wird die Unterabfrage ausgeführt, danach die Hauptabfrage. Verschachtelungen von Abfragen können nur deshalb funktionieren, da jede SELECT-Anweisung stets wieder eine Tabelle liefert.

Wir können grundsätzlich zwei Arten verschachtelter SELECT-Anweisungen unterscheiden: Solche, die auf einen Vergleichswert aus einer Untertabelle abfragen, und solche, die auf einen Wertebereich der Untertabelle abfragen. Erstere können einfach mit den bekannten Vergleichsoperatoren verwendet werden, für Letztere müssen wir spezielle Mengenoperatoren verwenden.

7.3.1 Unterabfragen auf einen einzigen Vergleichswert

Besteht eine Unterabfrage als Ergebnismenge aus genau einen Wert, beispielsweise ein Attributwert eines einzelnen Datensatzes oder dem Ergebnis einer Aggregatfunktion, so kann dieser mit dem Vergleichsoperator = in der **WHERE**-Klausel der Hauptabfrage bearbeitet werden.

```
SELECT spalte_1, ..., spalte_n FROM haupttabelle
WHERE spalte_x = (
  SELECT [spalte_y | aggregatfunktion] FROM nebentabelle ...
) ;
```

Handelt es sich bei dem Wert der Ergebnismenge um einen String, so kann statt des Vergleichsoperators entsprechend auch der **LIKE**-Operator verwendet werden. Handelt es sich bei dem Wert der Ergebnismenge dagegen um eine Zahl, so können auch die numerischen Vergleichsoperatoren >, >=, <, <=oder <>verwendet werden.

Beispiel 7.3. Wollen wir in unserer Comicsammlung wissen, welches die nach Erscheinungsjahr ältesten Alben sind, so können wir das nicht mit der Anweisung

```
SELECT titel, jahr FROM alben WHERE jahr =MIN(jahr) ;
```

denn dies widerspräche der Regel 4. Aber auch wenn wir **WHERE** durch **HAVING** ersetzen würden, kämen wir nur zu einer Fehlermeldung, da wir nicht nach Titel und Preis gruppieren. Wir müssen hier ein verschachteltes SELECT verwenden:

```
SELECT titel, jahr FROM alben WHERE jahr = (
  SELECT MIN (jahr) FROM alben
) ;
```

Hier ermittelt die Unterabfrage zunächst das minimale Jahr, bevordie äußere Abfrage alle Alben findet, die diesem Erscheinungsjahr gleichen. Man könnte schnell auf die Idee kommen, dass das die Abfrage auch

Beispiel 7.4. Wollen wir in unserer Comicsammlung diejenigen Titel finden, deren Preis kleiner als der Durchschnittspreis aller Alben ist, so erreichen wir dies mit der Anweisung:

```
SELECT titel, preis FROM alben WHERE preis < (
   SELECT AVG (preis) FROM alben
) ;
```

Hier ermittelt die Unterabfrage zunächst den Durchschnittspreis (hier 4,74), mit dem die äußere Abfrage die billigeren Alben filtert. Damit erhalten wir

titel	preis
Asterix, der Gallier	2,80
Asterix und Kleopatra	2,80
Gespenster Geschichten	1,20
Die Trabantenstadt	3,80

als Ergebnismenge.

Beispiel 7.5. Wollen wir – ähnlich wie in Beispiel 6.11 – die Titel einer speziellen Reihe mit ihren Preisdifferenzen zum Durchschnittspreis der Reihe sehen, so können wir das mit der folgenden verschachtelten Anweisung erreichen. Mit

```
SELECT titel, preis, preis - (
    SELECT AVG (preis) FROM alben WHERE reihe = 'Asterix'
) AS "Preisdifferenz"
FROM alben
WHERE reihe = 'Asterix' ;
```

erhalten wir beispielsweise die Titel der Asterixreihe mit ihren Differenzen zum Durchschnittspreis der Reihe:

titel	preis	Preisdifferenz
Asterix und Kleopatra	2.80	-0.640000
Asterix, der Gallier	2.80	-0.640000
Der große Graben	5.00	1.560000
Die Trabantenstadt	3.80	0.360000
Asterix als Legionär	2.80	-0.640000

Diese Anweisung ist von der Laufzeit her effizienter als die entsprechende Abfrage mit OVER PARTITION in Beispiel 6.11.

7.3.2 Unterabfragen auf mehrere Vergleichswerte

Besteht die Ergebnismenge der Unterabfrage aus Attributwerten mehrerer Datensätze, so können wir die Vergleichsoperatoren nicht mehr verwenden. Stattdessen müssen wir einen der *Mengenoperatoren* **EXISTS**, **IN**, **ANY** oder **ALL** verwenden. Ihre jeweilige Wirkung auf die Ergebnismenge (...)einer Unterabfrage ist wie folgt:

SQL Operator	Wirkung
EXISTS(...)	Prüft, ob die Unterabfrage überhaupt Daten liefert
x **IN**(...)	Prüft, ob der Wert für *x* in der Ergebnismenge der Unterabfrage vorkommt
x S **ANY**(...)	Prüft, ob *x* S irgendeinem Wert in der Ergebnismenge ist
x S **ALL**(...)	Prüft, ob *x* S jedem Wert in der Ergebnismenge ist

Hierbei bezeichnet S einen der Vergleichsoperatoren <, <=, >, >= oder =. Alle diese Ausdrücke müssen Teil einer **WHERE-** oder **HAVING**-Klausel sein. Die Operatoren **ANY** und **ALL** dienen dazu,
Vergleiche mit dem Minimum oder dem Maximum einer Menge durchzuführen. Wollen wir beispielsweise diejenigen Werte einer Spalte x filtern, die kleiner gleich dem Minimum bzw. Maximum einer Ergebnismenge (...)sind, so erreichen wir dies durch

SQL:	x <= **ALL**(...)	x <= **ANY**(...)
Mathematisch:	x S min (...)	x S max (...)

Umgekehrt filtern wir mit

SQL:	x >= **ALL**(...)	x >= **ANY**(...)
Mathematisch:	*x* S max (...)	*x* S min (...)

diejenigen Werte, die größer gleich dem Maximum, bzw. dem Minimum, von (...)sind.

NULL-Werte: Enthält die Unterabfrage einen **NULL**-Wert, so liefert der **ALL**-Operator *nie* den Wert **TRUE**! Der Grund ist, dass bei einem unbekannten Wert in der Menge ja auch unbekannt ist, ob das Vergleichskriterium erfüllt ist. Bei **ANY** dagegen wird ein **NULL**-Wert einfach ignoriert. Ferner sind strenggenommen die Operatoren = **ANY** bzw. = **ALL** in SQL überflüssig, denn einerseits ist = **ANY** völlig äquivalent zu **IN**, und andererseits liefert = **ALL** bei einer Ergebnismenge mit unterschiedlichen Werten nie **TRUE**.

Beispiel 7.6. Wollenwir in unserer Comic-Datenbank aus Beispiel 3.4 herausfinden, welcheTitel insgesamt in einem Jahr veröffentlicht wurden, in denen ein Asterix-Album erschien, so können wir das mit der Anweisung erreichen:

```
SELECT jahr, titel FROM alben WHERE jahr IN (
    SELECT jahr FROM alben WHERE reihe= 'Asterix'
) ;
```

Das Ergebnis dieser Abfrage lautet:

jahr	titel
1968	Asterix, der Gallier
1968	Asterix und Kleopatra
1974	Gespenster Geschichten
1974	Die Trabantenstadt
1980	Der große Graben

Wollen wir stattdessen herausfinden, welche Alben aus anderen Reihen im selben Jahr wie ein Asterix-Album erschienen, sollten wir die Abfrage etwas modifizieren:

```
SELECT jahr, titel FROM alben WHERE reihe<> 'Asterix' AND jahr IN (
    SELECT jahr FROM alben WHERE reihe= 'Asterix'
) ;
```

Das liefert als Ergebnismenge nur die Gespenster Geschichten aus dem Jahr 1974. Wollen wir die Titel mit den günstigsten Preisen einer Reihe sehen, so können wir die folgende Abfrage senden:

```
SELECT titel, preis FROM alben WHERE preis IN(
SELECT min (preis) FROM alben GROUP BY reihe
) ;
```

Sie liefert uns die Ergebnismenge:

titel	preis
Asterix, der Gallier	2.80
Asterix und Kleopatra	2.80
Gespenster Geschichten	1.20
Der große Graben	5.00
Das Kriminalmuseum	8.80
Das Meisterwerk	8.80
Asterix als Legionär	2.80

Wollen wir die Titel sortiert haben, so können wir **ORDER BY** titel anfügen.
Um schließlich herauszufinden, welche Titel einen Preis kleiner gleich den Durchschnittspreisen aller Reihen haben, müssen wir den **ALL**-Operator anwenden. Die Unterabfrage zur Ermittlung der Durchschnittspreise lautet

```
SELECT AVG (preis) FROM alben GROUP BY reihe HAVING AVG (preis) IS NOT NULL
```

und liefert die Werte

avg(preis)
1.20
3.60
8.80

Mit der **HAVING**-Klausel wird garantiert, dass kein **NULL**-Wert in der Ergebnismenge erscheint. Darauf können wir unsere Abfrage aufbauen:

```
SELECT titel, preis FROM alben WHERE preis <= ALL (
  SELECT AVG (preis) FROM alben GROUP BY reihe HAVING AVG (preis) IS NOT NULL
);
```

die als einzige Ergebniseintrag die Gespenster Geschichten mit dem Preis 1,20 liefert.

8

Joins

Kapitelübersicht

8.1 Fremdschlüssel

Definition 8.1. Ein *Fremdschlüssel* in einer Tabelle ist ein Schlüssel, der nur Werte enthält, die als Primärschlüssel von mindestens einer anderen Tabelle enthalten sind.

In SQL gehört die Deklaration eines Fremdschlüssels zur Definition einer Tabelle und kann mit den reservierten Wörtern

```
FOREIGN KEY (...) REFERENCES ...
```

implementiert werden. In den Klammern hinter **FOREIGN KEY** steht das Tabellenattribut, das den Fremdschlüssel bildet, hinter **REFERENCES** der Name der Tabelle und in Klammern deren Primärschlüssel, auf den verwiesen wird.

```
CREATE TABLE tabelle_1 (
  primärschlüssel <datentyp>,
  ...,
  PRIMARY KEY (primärschlüssel)
);

CREATE TABLE tabelle_2 (
  ...,
  fremdschlüssel <datentyp>,
  FOREIGN KEY (fremdschlüssel) REFERENCES tabelle_1(primärschlüssel)
)
```

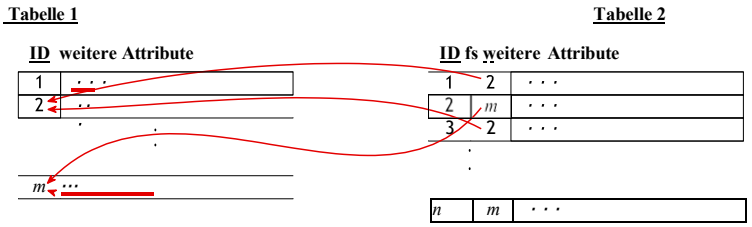


Abbildung 8.1: Die Fremdschlüssel (fs) der Datensätze von Tabelle 2 referenzieren auf die Primärschlüssel von Datensätzen der Tabelle 1.

Bildlich können wir uns vorstellen, dass der Fremdschlüssel jedes einzelnen Datensatzes der einen Tabelleaufdie Primärschlüssel referenziert, wie in Abbildung 8.1. Wirerkennen hier sofort, dass nicht unbedingt jeder Primärschlüssel referenziert wird, aber dass jeder Fremdschlüssel ungleich **NULL**auf einen Primärschlüssel referenzieren muss.

Eine Tabelle kann auch Fremdschlüssel aus mehreren anderen Tabellen enthalten, in SQL wird dann die **FOREIGN KEY**-Anweisung entsprechend wiederholt:

```
CREATE TABLE tabelle (
  ...,
  fremdschlüssel <datentyp>,
  FOREIGN KEY (fremdschlüssel_1) REFERENCES tabelle_1(primärschlüssel_1),
  ...,
  FOREIGN KEY (fremdschlüssel_n) REFERENCES tabelle_n(primärschlüssel_n)
)
```

Definition 8.2. *(Referenzielle Integrität)* Ein Fremdschlüssel einer Tabelle, der einen Wert ungleich **NULL** hat, muss auf den Primärschlüssel eines existierenden Datensatzes verweisen.

Zur Erhaltung der referenziellen Integrität muss ein RDBMS verhindern, dass Datensätze mit referenzierten Primärschlüsseln gelöscht werden. Um die referenzielle Integrität zu gewährleisten, kann man in SQL die Anweisung

```
ON DELETE RESTRICT
```

nach der Deklaration eines Fremdschlüssels geben, also:

```
CREATE TABLE tabelle_2 (
  ...,
  fs <datentyp>,
  FOREIGN KEY (fs) REFERENCES tabelle_1(primärschlüssel) ON DELETE RESTRICT
)
```

Die wichtigsten Anweisungen zur Erhaltung der referenziellen Integrität sind in Tabelle 8.1 aufgelistet. Sie decken alle möglichen Strategien von Löschen verhindern bis auf Ändern der

Anweisung	Wirkung bei Löschung des referenzierten Datensatzes
ON DELETE CASCADE	Löscht alle Datensätze mit, die auf den zu löschenden Datensatz referenzieren
ON DELETE RESTRICT	Datensatz wird nur gelöscht, wenn danach die referenzielle Integrität wiederhergestellt ist (Bei SQL-Server: **ON DELETE NO ACTION**[1])
ON DELETE SET DEFAULT	Setzt den Fremdschlüssel auf seinen Standardwert
ON DELETE SET NULL	Fremdschlüssel wird auf **NULL** gesetzt

Tabelle 8.1: SQL-Anweisungen zur Erhaltung der referenziellen Integrität von Fremdschlüsseln und die Wirkung bei Löschen des referenzierten Datensatzes.

Fremdschlüsselwertes auf **NULL**ab.[2]
Im Allgemeinen werden Fremdschlüssel dazu verwendet, um Beziehungen in ER-Diagrammen zu implementieren. Auf welche Weise das genau geschieht, hängt von den Kardinalitäten der beteiligten Tabellen ab. Wir werden dies im Laufe des Kapitels noch behandeln.

Manchmal begegnet man dem Problem, sich Daten anzeigen zu lassen, für deren Auswahl Informationen aus mehreren Tabellen benötigt werden. In SQL können auf verschiedene Weise Abfragen auf mehrere Tabellen durchgeführt werden. Die allgemeine Variante ist, die zu betrachtenden Tabellen hinter **FROM** einfach mit Komma getrennt aufzulisten und gewünschte Verknüpfungen der Tabellen in der WHERE-Klausel zu schreiben, neben weiteren Filterbedingungen. Viel mächtiger und für meist eleganter und verständlicher ist jedoch ein „Join“, der zwischen Verknüpfungs- und Filterbedingungen sauber trennt. Es gibt zwei Typen von Joins, den „Inner Join“, auch „Equi-Join“ genannt, und den „Outer Join“. Die wichtigsten Varianten beider Join-Typen werden wir im Folgenden behandeln.

8.2 Inner Joins

Um zwei Tabellen zu verknüpfen, kann die JOIN-Klausel verwendet werden, hinter dem ON - Parameter steht die Verknüpfungsbedingung. Eine zusätzliche Filterbedingung kann, wie gewöhnlich, in einer optionalen WHERE-Klausel festgelegt werden. Die Syntax dafür lautet:

```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
INNER JOIN  rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung ;
```

Die Verknüpfungsbedingung wird oft *Join-Bedingung* oder **ON**-Klausel genannt. Der Join-Typ **INNER** vor **JOIN** kann nach SQL-Standard weglassen werden, in MS Access allerdings muss er verwendet werden. Sehr oft besteht die Verknüpfungsbedingung darin, dass der Fremdschlüssel der einen Tabelle mit dem Primärschlüssel id der anderen Tabelle übereinstimmen muss. Die ON-Klausel lautet daher typischerweise:

```
... ON linke_tabelle.fremdschlüssel = rechte_tabelle.id ...
```
(8.1)

Das Ergebnis eines Joins ist eine Tabelle, die die Spalten beider Tabellen enthält und die verknüpften Datensätze zu einem Datensatz zusammenfügt.

Beispiel 8.1. Betrachten wir die folgenden beiden Tabellen:

alben

titel	reihe	band	preis	jahr
Gespenster Geschichten	1	1	1.20	1974
Asterix, der Gallier	2	1	2.80	1968
Asterix und Kleopatra	2	2	2.80	1968
Asterix als Legionär	2	10	3.00	NULL
Die Trabantenstadt	2	17	3.80	1974
Lucky Luke	NULL	1	5.00	1976
Der große Graben	2	25	5.00	1980
Der geheimnisvolle Stern	3	1	NULL	1972
Tim und der Haifischsee	3	23	NULL	1973
Das Kriminalmuseum	4	1	8.80	1985
Das Meisterwerk	4	2	8.80	1986

reihen

id	name
1	Gespenster Geschichten
2	Asterix
3	Tim und Struppi
4	Franka
5	Prinz Eisenherz

Hierbei ist alben die linke Tabelle, die mit der rechten Tabelle reihenüber einen Fremdschlüssel verknüpft ist, d.h. der ON-Parameter lautet entsprechend Gleichung (8.1):

```
...    ON alben.reihe = reihen.id    ...
```

Mit dem Befehl

```
SELECT * FROM alben INNER JOIN reihen ON alben.reihe =reihen.id
```

erhalten wir damit die Ergebnistabelle:

titel	reihe	band	preis	jahr	id	name
Gespenster Geschichten	1	1	1.20	1974	1	Gespenster Geschichten
Asterix als Legionär	2	10	3.00	*null*	2	Asterix
Asterix und Kleopatra	2	2	2.80	1968	2	Asterix
Asterix, der Gallier	2	1	2.80	1968	2	Asterix
Der große Graben	2	25	5.00	1980	2	Asterix
Die Trabantenstadt	2	17	3.80	1974	2	Asterix
Der geheimnisvolle Stern	3	1	*null*	1972	3	Tim und Struppi
Tim und der Haifischsee	3	23	*null*	1973	3	Tim und Struppi
Das Kriminalmuseum	4	1	8.80	1985	4	Franka
Das Meisterwerk	4	2	8.80	1986	4	Franka

Die Gesamtheit der Spalten besteht also aus allen Spalten beider Tabellen. Verwenden wir nun einen Join, um uns nur den Namen der Reihe und den Titel des Albums anzeigen zu lassen, allerdings mit der Bezeichnung „reihe“ für den Reihennamen. Dazu wählen wir nur zwei Spalten aus:

```
SELECT alben.titel, reihen.name AS reihe
FROM alben
INNER JOIN reihen ON alben.reihe = reihen.id;
```

und erhalten so die Ergebnistabelle:

titel	reihe
Asterix als Legionär	Asterix
Asterix und Kleopatra	Asterix
Asterix, der Gallier	Asterix
Das Kriminalmuseum	Franka
Das Meisterwerk	Franka
Der geheimnisvolle Stern	Tim und Struppi
Der große Graben	Asterix
Die Trabantenstadt	Asterix
Gespenster Geschichten	Gespenster Geschichten
Tim und der Haifischsee	Tim und Struppi

Mit dem Ausdruck **AS** reihe erreichen wir, dass in der Ergebnistabelle statt des Attributnamen name eben reihe steht.
Wollen wir uns nun nur die Alben aus der Asterix-Reihe anzeigen lassen, so fügen wir einfach eine geeignete WHERE-Klausel an:

```
SELECT reihen.name AS reihe, alben.titel FROM alben
INNER JOIN   reihen ON alben.reihe = reihen.id
WHERE reihen.name='Asterix' ;
```

reihe	titel
Asterix	Asterix, der Gallier
Asterix	Asterix und Kleopatra
Asterix	Asterix als Legionär
Asterix	Die Trabantenstadt
Asterix	Der große Graben

Wir sehen mit diesem Beispiel, das bei einem Inner Join beide Tabellen passende Einträge haben müssen. Die Reihe „Prinz Eisenherz“ taucht hier nirgendwo auf, ebenso das Album „Lucky Luke“. Daraus leiten wir das folgende allgemeine Merkmal eines Inner Joins ab.

Regel 5 (Merkmal eines Inner Joins) *Durch einen Inner Join können nur Datensätze angezeigt werden, die in den beiden Tabellen miteinander verknüpft sind. Beide Tabellen sind dabei gleichberechtigt. Lediglich bei* ***SELECT*FROM...*** *wird die Reihenfolge der angezeigten Spalten der Ergebnismenge vertauscht, da die Spalten der linken Tabelle hier stets zuerst erscheinen.*

8.3 Left und Right Joins

Im Gegensatz zu einem Inner Join werden bei einem *Outer Join* auch Datensätze ohne Verknüpfung zur verknüpften Tabelle angezeigt. Eine der Tabellen ist dabei jedoch führend in dem Sinne, dass von ihr auch nichtverknüpfte Datensätze angezeigt werden. Je nach Typ des Outer Joins ist das die linke oder die rechte Tabelle. Entsprechend gibt es einen **LEFT JOIN** mit der Syntax

```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
LEFT JOIN rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung;
```

oder einen **RIGHT JOIN**mit der Syntax

```
SELECT spalte_1, ..., spalte_n
FROM linke_tabelle
RIGHT JOIN   rechte_tabelle ON verknüpfungsbedingung
WHERE filterbedingung;
```

(Statt **JOIN** darf man bei den meisten RDBMS jeweils auch **OUTER JOIN**schreiben. Da aber mit einem der Wörter **LEFT** oder **RIGHT**der Typ des Joins schon eindeutig festgelegt ist, werden wir **OUTER**im Folgenden weglassen.)

Beispiel 8.2. Betrachten wir als Anwendungsfall eine kleine Datenbank, in der die Angestellten einer Firma und die Dienstwagen gespeichert sind.

Gegeben seien dazu die beiden folgenden Tabellen.

```
--Tabellenstrukturen:
CREATE TABLE angestellte (
    id int PRIMARY KEY, name
    varchar(10), dienstwagen
    int,
    FOREIGN KEY(dienstwagen) REFERENCES dienstwagen(id) ON DELETE RESTRICT
) ;
CREATE TABLE dienstwagen (
    id int PRIMARY KEY,
    kennzeichen varchar(10)
) ;
--Daten:
INSERT INTO angestellte (id, name, dienstwagen) VALUES
  (1, 'Anna', 2), (2, 'Otto', null), (3, 'Alice', 1), (4, 'Bob', 1) ;
INSERT INTO dienstwagen (id, kennzeichen) VALUES
  (1, 'HA-FH1234' ), (2, 'HA-FH2345' ), (3, 'HA-FH3456' ) ;
```

also

angestellte

id	name	dienstwagen
1	Anna	2
2	Otto	*null*
3	Alice	1
4	Bob	1

dienstwagen

id	kennzeichen
1	HA-FH 1234
2	HA-FH 2345
3	HA-FH 3456

Wie können wir nun alle Angestellten mit ihren Dienstwagen anzeigen lassen, wobei für Angestellte ohne Dienstwagen der Wert NULLerscheinen soll?
Ein Left Join über alle Spalten ergibt dann:

```
SELECT * FROM angestellte LEFT JOIN dienstwagen
  ON angestellte.dienstwagen = dienstwagen.id ;
```

id	name	dienstwagen	id	kennzeichen
1	Anna	2	2	HA-FH 2345
2	Otto	*null*	*null*	*null*
3	Alice	1	1	HA-FH 1234
4	Bob	1	1	HA-FH 1234

Wir sehen, alle Datensätze der linken Tabelle angestelltesind in der Ergebnismenge enthalten sowie die verknüpften Datensatze aus der rechten Tabelle dienstwagen. Alle nicht eingeteilten Dienstwagen werden nicht angezeigt. Ein Right Join über alle Spalten dagegen hat das Resultat:

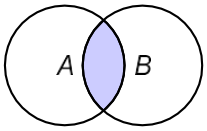
```
SELECT * FROM angestellte RIGHT JOIN  dienstwage
  ON angestellte.dienstwagen = dienstwagen.id;
```

id	name	dienstwagen	id	kennzeichen
1	Anna	2	2	HA-FH 2345
3	Alice	1	1	HA-FH 1234
4	Bob	1	1	HA-FH 1234
null	*null*	*null*	3	HA-FH 3456

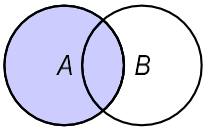
Hier sind alle Datensätze der rechten Tabelle dienstwagen enthalten, sowie die mit ihnen verknüpften aus der linken Tabelle angestellte. Entsprechend werden alle Angestellten ohne Dienstwagen, hier also Otto, nicht angezeigt. □

Bemerkung 8.3. Die Wirkungsweise der verschiedenen Joins ist in Abbildung 8.1 illustriert. Hierbei sind *A* und *B* zwei Tabellen, deren Datensätze als Mengen dargestellt sind. Die Schnittmenge repräsentiert dabei jeweils die durch die ON-Bedingung verknüpften Datensätze. Im

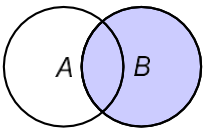
```
FROM A INNER JOIN
B
ON A.fs = B.id
```



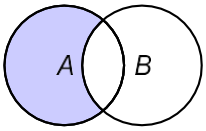

```
FROM A LEFT JOIN
B
ON A.fs = B.id
```




```
FROM A RIGHT JOIN
B
ON A.fs = B.id
```




```
FROM A  LEFT  JOIN
B
ON A.fs = B.id
WHERE  A.fs  IS
NULL
```




```
FROM A RIGHT JOIN
B
ON A.fs = B.id
WHERE A.fs IS NULL
```

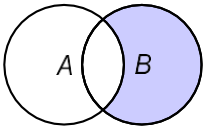


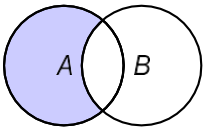
Abbildung 8.1: Wirkungvon Joins aufzwei Tabellen *A* und *B*. Die Schnittmengerepräsentiert die durch die Join-Bedingung verknüpften Datensätze. (*A*.fs ist der Fremdschlüssel von *A*, *B*.id der Primärschlüssel von *B*.)

Umkehrschluss sind die Teile der Mengen außerhalb der Schnittmenge diejenigen Datensätze, die keine Verknüpfung zu der jeweils anderen Tabelle haben. Ein Inner Join hat als Ergebnismenge also nur die miteinander verknüpften Datensätze beider Tabellen, ein Left Join alle Datensätze der Tabelle *A*, und ein Right Join alle der Tabelle *B*. Wollen wir, wie in der zweiten Zeile, die nichtverknüpften Datensätze explizit anzeigen, so muss eine WHERE-Klausel hinzugefügt werden, um die Datensätze mit dem Wert **NULL**ihres Verknüpfungsschlüssels zu filtern,

in der Regel der Fremdschlüssel. □

Bemerkung 8.4. Tatsächlich hätten wir für den vorletzten Fall in Abbildung 8.1,

```
FROM A LEFT JOINB
ON A.fs = B.id
WHERE A.fs IS NULL
```



keinen **LEFT JOIN** gebraucht, da für die Filterbedingung in der **WHERE**-Klausel keine Information aus Tabelle *B* benötigt wird. Dasselbe Ergebnis würde hier also auch durch eine Anweisung nur mit *A* erreicht, d.h. durch

```
FROM A
WHERE A.fs IS NULL
```

```
FROM A LEFT JOIN B
  ON A.fs = B.id
WHERE A.fs IS NULL
```

Wir werden im folgenden Beispiel aber Verknüpfungsbedingungen betrachten, die nicht auf einem Fremdschlüssel A.fsbasieren, sondern auf anderen Informationen aus *B*. □

Beispiel 8.5. *(Wann ist ein Join notwendig?)* Gegeben seien zwei Tabellen, von denen wir annehmen, dass sie unabhängig voneinander sind, also keine direkte Beziehung miteinander haben:

freunde

name	wohnort
Anna	Hagen
Bert	Iserlohn
Cindy	Soest
Dora	Meschede
Ernie	Lüdenscheid

kollegen

name	abteilung
Anna	Controlling
Cindy	IT
Ernie	Vertrieb
Gerd	Marketing

Wie lautet eine SQL-Abfrage, die uns die Namen unserer Freunde anzeigt, die nicht auch Kollegen sind? Gesucht ist also der Fall  aus Abbildung 8.1, d.h. ein Left Join: Wähle alle Datensätze aus freundeaus, deren namenicht in kollegenist,

```
SELECT freunde.name FROM freunde
LEFT JOIN kollegen ON freunde.name = kollegen.name
WHERE kollegen.name IS NULL ;
```

Da in der Tabelle freunde kein Fremdschlüssel oder sonst eine Information aus der Tabelle kollegenvorhanden ist, *müssen* wir hier einen Outer Join verwenden! □

Regel 6 *Ein Join ist immer notwendig, wenn für eine Abfrage Information aus mehreren Tabellen benötigt wird.*

Beispiel 8.6. *(Left und Right Joins)* Gegeben seien zwei Tabellen, von denen wir annehmen, dass sie unabhängig voneinander sind, also keine direkte Beziehung miteinander haben:

freunde

name	wohnort
Anna	Hagen
Bert	Iserlohn
Cindy	Soest
Dora	Meschede
Ernie	Lüdenscheid

kollegen

name	abteilung
Anna	Controlling
Cindy	IT
Ernie	Vertrieb
Gerd	Marketing

Wie lautet eine SQL-Abfrage, die uns die Namen derjenigen anzeigt, die nicht Freunde und Kollegen sind? Überlegen wir uns dazu die Lösung schrittweise:

1. Wähle alle Datensätze aus freundeaus, deren namenicht in kollegenist. 
2. Wähle alle Datensätze aus kollegenaus, deren namenicht in freundeist. 
3. Vereinige beide Ergebnistabellen. 

1. Hier wird ein Left Join benötigt (siehe Beispiel 8.5)

```
SELECT freunde.name FROM freunde
LEFT JOIN kollegen ON freunde.name = kollegen.name
WHERE kollegen.name IS NULL ;
```

name
Bert
Dora

(8.2)

2. Hier verwenden wir analog einen Right Join:

```
SELECT kollegen.name FROM freunde
RIGHT JOIN   kollegen ON freunde.name = kollegen.name
WHERE freunde.name IS NULL ;
```

name
Gerd

(8.3)

3. Vereinigen wir zum Schluss die beiden Abfragen (8.2) und (8.3) mit **UNION**, so erhalten wir unsere Lösung:

```
SELECT freunde.name, freunde.name FROM freunde
LEFT JOIN kollegen ON freunde.name =kollegen.name
  WHERE kollegen.name IS NULL
UNION
SELECT kollegen.name, kollegen.name FROM freunde
RIGHT JOIN   kollegen ON freunde.name =kollegen.name
  WHERE freunde.name IS NULL
```

name
Bert
Dora
Gerd

Übrigens hätten wir das Problem auch mengentheoretisch lösen können, indem wir jeweils alle Datensätze der einen Tabelle selektieren und davon alle Datensätze der anderen als Menge subtrahieren, dasselbe mit vertauschten Tabellen selektieren; und schließlich beides vereinen:

(8.4)

also in SQL:

```
( (SELECT name FROM freunde) EXCEPT (SELECT name FROM kollegen) )
UNION
( (SELECT name FROM kollegen) EXCEPT (SELECT name FROM freunde) );
```

Diese Lösung ist nach meinem Empfinden noch eleganter, allerdings funktioniert sie nicht für alle Datenbanksysteme (insbesondere nicht Access).

8.4 Joins mit mehr als zwei Tabellen

Für die Anzahl der Tabellen eines Joins gibt es keine Obergrenze. Joins bieten sich daher an, bei Beziehungstabellen statt der Fremdschlüssel aussagekräftigere Bezeichnungen der referenzierten Tabelle anzuzeigen. Betrachten wir dazu als Anwendungsbeispiel das Problem, unsere Comic-Datenbank um die Möglichkeit zu erweitern, auch die Autoren der Alben zu speichern.[1] Ein Autor kann dabei mehrere Alben schreiben, umgekehrt kann ein Album mehrere Autoren haben. Da jedes Album mindestens einen Autor haben muss, erhalten wir damit das das Entity-Relationship-Diagramm in Abbildung 8.2. Da zwischen Album und Autor eine CM-CM-Beziehung

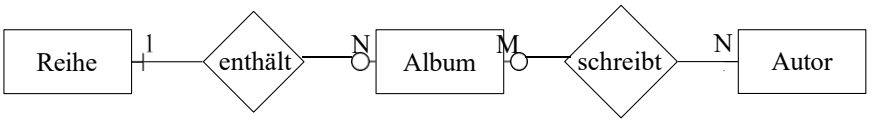


Abbildung 8.2: ER-Diagramm der Comics mit Autoren

vorliegt, müssen wir zwischen ihnen eine Beziehungstabelle einführen. Die Attribute eines Autors sollen nur aus einer ID und seinem Namen bestehen, d.h. die Autoren sind vom Relationstyp *autoren(id, name)*, die Beziehungstabelle vom Typ *albenautoren(autor, titel)*. Die vollständige Tabellenstruktur ist damit wie folgt:

Tabelle	Primärschlüssel	Attribute
reihen	id	name
alben	titel	reihe, band, preis, jahr
autoren	id	name
albenautoren	autor, titel	

In SQL erzeugen wir damit eine Tabelle autoren, und entsprechend eine Tabelle albenautoren:

```
CREATE TABLE autoren (
  id int NOT NULL  ,
  name varchar(20) NOTNULL
  ,
  PRIMARY KEY (id)
```

```
CREATE TABLE albenautoren ( autor
  int NOT NULL  ,
  titel varchar(50) NOT NULL ,
  PRIMARY KEY (autor,titel),
  FOREIGN KEY (autor) REFERENCES autoren(id) ON DELETE RESTRICT ,
  FOREIGN KEY (titel) REFERENCES alben(titel) ON DELETE RESTRICT
);
```

Speichern wir zum Schluss noch Daten in die Tabellen:

```
INSERT INTO autoren (id, name) VALUES
  (1, 'Uderzo'),
  (2, 'Goscinny'),
  (3, 'Hergé'),
  (4, 'Kuijpers'),
  (5, 'Franquin'),
  (6, 'Morris');
```

```
INSERT INTO albenautoren (autor, titel) VALUES
  (1, 'Asterix als Legionär'),
  (1, 'Asterix und Kleopatra'),
  (1, 'Asterix, der Gallier'),
  (1, 'Der große Graben'),
  (1, 'Die Trabantenstadt'),
  (2, 'Asterix als Legionär'),
  (2, 'Asterix und Kleopatra'),
  (2, 'Asterix, der Gallier'),
  (2, 'Die Trabantenstadt'),
  (3, 'Der geheimnisvolle Stern'),
  (3, 'Tim und der Haifischsee'),
  (4, 'Das Kriminalmuseum'),
  (4, 'Das Meisterwerk'),
  (6, 'Lucky Luke');
```

[1] Piepmeyer (2011):S.

Zusammen mit den Tabellen und Daten aus Beispiel 10.1 haben wir die Datenbank von Comic-Alben vervollständigt.
Wie können wir uns nun eine Liste der Autoren und den Reihen, an denen sie mitgewirkt haben, jeweils mit Namen anzeigen? Anhand des Tabellendiagramms in Abbildung 8.2 erkennen wir, dass wir die Namen in den Tabellen autoren und reihen nur über zwei weitere Tabellen verknüpfen können. Wir können das mit einer einfachen Aneinanderreihung von **INNER JOINS** programmieren:

```
SELECT DISTINCT autoren.name AS autor, reihen.name AS reihe
FROM       autoren
INNER JOIN albenautoren ON autoren.id = albenautoren.autor
INNER JOIN alben        ON albenautoren.titel = alben.titel
INNER JOIN reihen       ON reihen.id = alben.reihe
```

Hinweis: Verwenden Sie bei einem Join über mehr als zwei Tabellen zur Unterstützung ein ER-Diagramm!

8.5 Self Joins

Ein Self Join ist ein Join, der eine Tabelle mit sich selber verknüpft. In der Regel ist ein Self Join ein **INNER JOIN**. Solche Joins sind immer dann nötig, wenn Werte einer Spalte aus verschiedenen Datensätzen verknüpft werden sollen. Da der Join in diesem Fall auf beiden Seiten dieselbe Tabelle verknüpft, müssen die beiden „Instanzen“ (also Ergebnistabellen) durch einen Alias unterschieden werden:

```
SELECT spalte_1, ..., spalte_n
FROM tabelle AS t1
INNER JOIN   tabelle AS t2
ON t1.spalte_k = t2.spalte_k [WHERE
filterbedingung]
```

(Oft wird **AS** auch weggelassen.)

Beispiel 8.7. Betrachten wir wieder unsere normalisierte Datenbank von Comic-Alben aus Beispiel 10.1. Eine typische Fragestellung für einen Self Join ist die folgende: In welchen Jahren ist es in einer Reihe jeweils zu einer Preiserhöhung gekommen? Oder etwas anders ausgedrückt: Was sind die Erscheinungsjahre aller Alben, zu denen es in der gleichen Reihe ein günstigeres Vorgängeralbum gab? Ein erster Lösungsansatz ist die folgende Anweisung, mit der wir die Datensätze einer Reihe miteinander verknüpfen und unsalle Spalten der Ergebnistabelle anzeigen lassen, nachdem diejenigen Bände gefiltert, bei denen der Preis gestiegen ist:
Sie ergibt die Ergebnismenge

```
SELECT DISTINCT a2.reihe, a2.jahr
FROM alben AS a1 INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.band < a2.band AND a1.preis < a2.preis
```

titel	reihe	band	preis	jahr	titel	reihe	band	preis	jahr
Asterix, der Gallier	2	1	2.80	1968	Der große Graben	2	25	5.00	1980
Asterix, der Gallier	2	1	2.80	1968	Die Trabantenstadt	2	17	3.80	1974
Asterix, der Gallier	2	1	2.80	1968	Asterix als Legionär	2	10	3.00	*null*
Asterix und Kleopatra	2	2	2.80	1968	Der große Graben	2	25	5.00	1980
Asterix und Kleopatra	2	2	2.80	1968	Die Trabantenstadt	2	17	3.80	1974
Asterix und Kleopatra	2	2	2.80	1968	Asterix als Legionär	2	10	3.00	*null*
Asterix als Legionär	2	10	3.00	*null*	Der große Graben	2	25	5.00	1980
Asterix als Legionär	2	10	3.00	*null*	Die Trabantenstadt	2	17	3.80	1974
Die Trabantenstadt	2	17	3.80	1974	Der große Graben	2	25	5.00	1980

Links sehen wir die Instanz a1unserer Albentabelle, rechts die Instanz a2. In a1sind die Einträge, die uns interessieren. D.h. wir grenzen unsere anzuzeigende Spaltenauswahl ein und filtern die Bände, bei denen der Preis gestiegen ist:

```
SELECT a2.jahr, a2.titel, a2.reihe, a2.band, a2.preis
FROM alben AS a1 INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.band < a2.band AND a1.preis < a2.preis
```

Damit erhalten wir:

jahr	titel	reihe	band	preis
1980	Der große Graben	2	25	5.00
1974	Die Trabantenstadt	2	17	3.80
null	Asterix als Legionär	2	10	3.00
1980	Der große Graben	2	25	5.00
1974	Die Trabantenstadt	2	17	3.80
null	Asterix als Legionär	2	10	3.00
1980	Der große Graben	2	25	5.00
1974	Die Trabantenstadt	2	17	3.80
1980	Der große Graben	2	25	5.00

Die Jahre von Preiserhöhungen innerhalb einer Reihe erhalten wir damit schließlich durch die Anzeige von Reihe, Jahr und Preis sowie die Eliminierung von Dubletten durch **DISTINCT**:

```
SELECT DISTINCT a2.reihe, a2.jahr, a2.preis
FROM alben AS a1
INNER JOIN alben AS a2 ON a1.reihe = a2.reihe
WHERE a1.band < a2.band AND a1.preis <a2.preis
ORDER BY  a2.jahr
```

also:

reihe	jahr	preis
2	*null*	3.00
2	1974	3.80
2	1980	5.00

Definitiv gab es damit bei der Reihe 2 (Asterix) Preiserhöhungen 1974 und 1980. Wenn wir davon ausgehen, dass die Preise im Laufe der Zeit nie kleiner werden, gab es irgendwann vor 1974 (und nach 1968) eine erste Preiserhöhung.