

Die Vorgehensweise ist insbesondere zum Test von SQL-Funktionen hilfreich:

```
SELECT 1 + 1
      2

SELECT NOW()
2024-03-02 11:05:

SELECT RIGHT('abcdefg', 3)
      efg
```

Eine Ausnahme ist diesbezüglich Oracle, das derartige Abfragen als syntaktisch falsch ablehnt. Abhilfe schafft die Nennung der virtuellen Tabelle DUAL, die aus einer Spalte mit einem Datensatz besteht:

```
SELECT 1 + 1 FROM DUAL
      2
```

## 10.3 Tabellen verknüpfen (JOIN)

Abfragen, die sich auf eine einzelne Tabelle beziehen, sind in der Praxis eher die Ausnahme. Es liegt in der Natur relationaler Datenbanken, dass die Daten über mehrere Tabellen verteilt und in Abfragen zusammengefügt werden müssen.

### Einführungsbeispiele

SQL sieht zur Verknüpfung von Tabellen viele Syntaxvarianten vor. Bevor ich Ihnen die wichtigsten Varianten der Reihe nach vorstelle, beginne ich hier mit einigen konkreten Beispielen.

In der Tabelle *books* der gleichnamigen Datenbank verweist jeder Datensatz auf einen Verlag (Spalte *publId*). Die WHERE-Bedingung hat hier nur den Zweck, das Ergebnis auf wenige Ergebnisse zu beschränken:

```
SELECT id, title, publId FROM books WHERE id>=167
```

id	title	publId
167	The Relational Model for Database Management	1
168	Time and Relational Theory	42
169	Database Design and Relational Theory	2
170	Python	37

Die Namen der Verlage befinden sich in der Tabelle *publishers*:

```
SELECT id, name FROM publishers WHERE id IN(1, 2, 37, 42)
```

id	name
1	Addison-Wesley
2	Apress
37	Rheinwerk Verlag
42	Morgan Kaufmann Publishers

Um diese Informationen in *einer* Abfrage zusammenzufassen, geben Sie nach `FROM` `table1` eine zweite Tabelle mit `JOIN table2 ON condition` an. Die Bedingung gibt dabei an, *wie* die beiden Tabellen verknüpft werden sollen, d. h. welche Spalte der einen Tabelle (Fremdschlüssel) auf welche Spalte der anderen Tabelle (Primärschlüssel) verweist:

```
SELECT books.id, title, publId, name
FROM books
JOIN publishers ON publId = publishers.id
WHERE books.id >= 167
```

id	title	publId	name
167	The Relational Model for ...	1	Addison-Wesley
168	Time and Relational Theory	42	Morgan Kaufmann ...
169	Database Design and Relat...	2	Apress
170	Python	37	Rheinwerk Verlag

Beachten Sie, dass die Deklaration von Foreign-Key-Regeln keine Voraussetzung für die Anwendung von `JOIN` ist. Sie können beliebige Spalten durch `JOIN` verknüpfen, sofern ihre Datentypen übereinstimmen.

Im obigen SQL-Code ist es von zentraler Bedeutung, dass jedem Spaltennamen, der nicht eindeutig einer Tabelle zugeordnet werden kann, der Tabellename vorangestellt ist. `id` zur Bezeichnung der ID-Spalte reicht nicht aus, weil es diese Spalte ja in zwei Tabellen gibt. Erst die vollständige Bezeichnung (*fully qualified*) der Spalten in der Form `books.id` bzw. `publishers.id` ist eindeutig. (`publId`, `name` und `title` sind auch eindeutig, weil es diese Spalte jeweils nur in einer der beiden Tabellen gibt.)

Um mögliche Namenskonflikte zu vermeiden, ist es üblich, in Abfragen über mehrere Tabellen *jeder* Spalte den Namen der zugeordneten Tabelle voranzustellen. Das führt allerdings zu sehr langen Kommandos; außerdem ist der Tippaufwand erheblich. Deswegen werden normalerweise allen Tabellen mit AS Kurzbezeichnungen (Aliasse) zugeordnet, normalerweise einfach die Anfangsbuchstaben der Tabelle:

```
SELECT b.id, b.title, p.id, p.name
FROM books AS b
JOIN publishers AS p ON b.publId = p.id
WHERE b.id >= 167
```

Das Schlüsselwort AS, das Sie ja schon im vorigen Abschnitt kennengelernt haben, ist auch hier optional. Wird es weggelassen, kommen wir zur Schreibweise, die die meisten Datenbankprofis vorziehen. Die immer noch gleichwertige Abfrage sieht dann wie folgt aus:

```
SELECT b.id, b.title, p.id, p.name
FROM books b
JOIN publishers p ON b.publId = p.id
WHERE b.id >= 167
```

Das Zeichen \* als Platzhalter für alle Spalten funktioniert auch bei Abfragen über mehrere Tabellen. Das Ergebnis enthält dann sämtliche Spalten *aller* Tabellen, was in den meisten Fällen über das Ziel hinausschießt. Bei den meisten DBMS (nicht Oracle) gibt es auch die Variante tabellenname.\* bzw. alias.\*, wodurch nur die Spalten einer bestimmten Tabelle erfasst werden. Die folgende Abfrage liefert aus der Tabelle *books* nur die Spalte *title*, aus *publishers* dagegen alle Spalten:

```
SELECT b.title, p.*
FROM books b
JOIN publishers p ON b.publId = p.id
WHERE b.id >= 167
```

title	id	name	city
The Relational ...	1	Addison-Wesley	München
Time and Relati...	42	Morgan Kaufmann ...	NULL
Database Design...	2	Apress	New York
Python	37	Rheinwerk Verlag	Bonn

*books* ist nicht nur mit *publishers* verknüpft, sondern mit einigen weiteren Tabellen (siehe Abbildung 8.2). Dementsprechend können Sie Abfragen formulieren, bei denen Daten aus drei, vier oder sogar noch mehr Tabellen zusammengeführt werden. Die folgende Abfrage kombiniert Buchtitel, Verlagsinformationen und Sprachen:

```
SELECT b.title,
       p.name AS publisher,
       l.name AS language
FROM books b
JOIN publishers p ON b.publId = p.id
JOIN languages l ON b.langId = l.id
WHERE b.id >= 167
```

title	publisher	language
Database Design ...	Apress	English
Time and Relatio...	Morgan Kaufmann ...	English
The Relational M...	Addison-Wesley	English
Python	Rheinwerk Verlag	Deutsch

Analog können Sie Daten aus noch mehr Tabellen verknüpfen, indem Sie entsprechend öfter JOIN otherTable ON condition angeben.

In der obigen Abfrage gibt es drei Bedingungen: zwei für die richtige Verknüpfung der richtigen Datensätze und eine weitere für die Selektion der Ergebnisdatensätze. Die SQL-Syntax erlaubt es auch, alle drei Bedingungen mit WHERE anzugeben.

```
-- gleichwertig zur vorigen Abfrage
SELECT b.title,
       p.name AS publisher,
       l.name AS language
FROM books b JOIN publishers p JOIN languages l
WHERE b.publId = p.id AND b.langId = l.id AND b.id >= 167
```

In älteren SQL-Versionen gab es das Schlüsselwort JOIN noch gar nicht. Sämtliche Tabellen wurden einfach mit FROM aufgelistet, die Verknüpfungsbedingungen folgten mit WHERE. Daraus ergibt sich der folgende Code, der auch heute noch erlaubt ist:

```
-- ebenfalls gleichwertig
SELECT b.title,
       p.name AS publisher,
       l.name AS language
FROM books b, publishers p, languages l
WHERE b.publId = p.id AND b.langId = l.id AND b.id >= 167
```

Die letzten drei Abfragen liefern alle dasselbe Ergebnis. Die Frage ist aber, ob Sie das Ergebnis auch gleich schnell erhalten. Das hängt davon ab, wie intelligent der Query Optimizer Ihres DBMS vorgeht. Sowohl aus Gründen der besseren Lesbarkeit als auch zur Unterstützung der Abfrageoptimierung durch Ihr DBMS sollten Sie Verknüpfungsbedingungen immer mit JOIN ... ON angeben.

Bei der winzigen Musterdatenbank *books* sind Performanceüberlegungen sowieso hinfällig. Aber bei wirklich großen Datenbanken spielt es durchaus eine Rolle, in welcher Reihenfolge das DBMS die Bedingungen verarbeitet. Im Idealfall sollte dem DBMS die bestmögliche Optimierung unabhängig von der Position der Bedingungen gelingen – aber nicht immer können Sie sich darauf verlassen.

Umgekehrt könnten Sie je nach Join-Typ auch die WHERE-Bedingung in einem der beiden ON-Blöcke unterbringen, z. B. so:

```

SELECT b.title,          -- ebenfalls gleichwertig
       p.name AS publisher,
       l.name AS language
  FROM books b
 JOIN publishers p ON b.publId = p.id AND b.id >= 167
 JOIN languages l ON b.langId = l.id

```

Auch wenn diese Abfrage syntaktisch korrekt ist, wird der Code der Abfrage dadurch unübersichtlich. Die Vorgehensweise ist also nicht empfehlenswert.

Noch ein letzter Hinweis: Dass Sie die SELECT-Bedingungen nahezu frei verschieben können, gilt nur für den in den Beispielen eingesetzten *Inner Join*. Andere Join-Varianten (*Left Join*, *Right Join* usw.) funktionieren dagegen nur dann wie vorgesehen, wenn Sie die Verknüpfungsbedingung mit ON angeben!

### Join-Varianten

Wie ich bereits in Kapitel 9, »Relationale Algebra und SQL«, ausgeführt habe, gibt es aus mathematischer Sicht verschiedene Möglichkeiten, Relationen miteinander zu verknüpfen. Entsprechend kennt die SQL-Syntax diverse Syntaxvarianten rund um JOIN (siehe Tabelle 10.3). Beachten Sie, dass die ergänzenden Schlüsselwörter INNER und OUTER optional sind. Ihre Angabe verändert die Vorgehensweise bei der Verknüpfung nicht.

Syntax	Bedeutung
FROM t1 [CROSS ]JOIN t2 (ohne Bedingung)	kartesisches Produkt
FROM t1 [INNER] JOIN t2 ON cond	Inner Join
FROM t1 LEFT [OUTER] JOIN t2 ON cond	Left Join (erlaubt NULL in t1)
FROM t1 RIGHT [OUTER] JOIN t2 ON cond	Right Join (erlaubt NULL in t2)
FROM t1 FULL [OUTER] JOIN t2 ON cond	Outer Join (erlaubt NULL in t1 und t2)
FROM t1 NATURAL [LEFT] JOIN t2	Natural Join für gleichnamige Verknüpfungsspalten
FROM t1 xxx JOIN t2 USING (column)	Join für die angegebene Spalte

Tabelle 10.3 SQL kennt diverse Join-Varianten.

In der Praxis kommt neben dem Inner Join eigentlich nur der Left Join häufig vor. Trotzdem sollten Sie sämtliche Join-Varianten kennen und ihre Bedeutung verstehen. Da sich einige Varianten in einer »richtigen« Datenbank schwer demonstrieren lassen, gelten als Ausgangspunkt für die folgenden Beispiele die Tabellen t1 und t2

(siehe Abbildung 10.2). Als Verknüpfungsspalte dient *e*. Um Ihnen ein einfaches Ausprobieren der Beispiele zu ermöglichen, befinden sich die beiden Tabellen in der Musterdatenbank *books*, obwohl sie natürlich mit dem eigentlichen Zweck dieser Datenbank nichts zu tun haben.

<b>t1</b>			<b>t2</b>		
<b>a</b>	<b>b</b>	<b>e</b>	<b>c</b>	<b>d</b>	<b>e</b>
1	III	12	33	abc	NULL
2	XI	13	34	efg	12
3	V	NULL	35	opq	27
4	VI	27	36	ijk	NULL
5	II	NULL	37	uvw	8
6	IX	8	38	xyz	9
7	X	15			

Abbildung 10.2 Beispieldaten zur Demonstration der Join-Varianten

### Kartesisches Produkt

Beginnen wir mit einem Join *ohne* Bedingung, also dem kartesischen Produkt. Diese Operation liefert jede Kombination aus Datensätzen aus *t1* und *t2*. Da die erste Tabelle sieben und die zweite Tabelle sechs Datensätze enthält, ergeben sich so 42 Kombinationen. Allein dieser Umstand macht klar, dass ein derartiger Join bei realen Datenbanken riesige Datenmengen produziert und ohne praktische Relevanz ist.

```
SELECT * FROM t1 JOIN t2
```

a	b	e	c	d	e
--	---	---	---	---	---
1	III	12	38	xyz	9
1	III	12	37	uvw	8
1	III	12	36	ijk	NULL
1	III	12	35	opq	27
1	III	12	34	efg	12
1	III	12	33	abc	NULL
2	XI	13	38	xyz	9
...					
7	X	15	33	abc	NULL

### Die »SELECT«-Ergebnisse sind ungeordnet!

Vielleicht irritiert Sie im obigen Beispiel die Reihenfolge der Ergebnisse. Grundsätzlich gilt: Solange Sie nicht ORDER BY verwenden, dürfen Sie nicht erwarten, dass die Ergebnisdatensätze in irgendeiner Weise geordnet sind.

Ja, viele DBMS liefern Datensätze in der gleichen Reihenfolge, in der diese in der Tabelle gespeichert wurden, aber Sie können sich darauf nicht verlassen.

### Inner Join und Natural Join

Der Inner Join berücksichtigt bei der Auswertung der Verknüpfungsbedingung nur Werte ungleich NULL. Dementsprechend liefert die folgende Abfrage nur drei Datensätze – die, bei denen die Werten der Spalte *e* in *t1* mit den Werten der Spalte *e* in *t2* übereinstimmen, aber nicht NULL sind:

```
SELECT * FROM t1 JOIN t2 ON t1.e = t2.e
```

a	b	e	c	d	e
--	---	--	--	----	--
1	III	12	34	efg	12
4	VI	27	35	opq	27
6	IX	8	37	uvw	8

Bei *t1* und *t2* hat die Verknüpfungsspalte *e* in beiden Tabellen den gleichen Namen *e*. In diesem Fall können Sie den Inner Join zu einem Natural Join vereinfachen, das heißt, Sie können sich die Angabe der Verknüpfungsspalte sparen. Ein weiterer Vorteil besteht darin, dass bei der Verwendung von \* für die Spaltenangabe die Verknüpfungsspalte *e* nur noch einmal im Ergebnis auftaucht:

```
SELECT * FROM t1 NATURAL JOIN t2
```

e	a	b	c	d
--	--	---	--	----
12	1	III	34	efg
27	4	VI	35	opq
8	6	IX	37	uvw

Beachten Sie aber, dass die Schlüsselwörter NATURAL JOIN von manchen DBMS nicht unterstützt werden (z. B. Db2 oder SQL Server). Selbst DBMS, die diese Join-Variante unterstützen, scheitern oft bei der Verknüpfung von mehr als zwei Tabellen.

### Join-Spalte mit USING angeben

Wenn die Verknüpfungsspalte in beiden Tabellen die gleichen Namen hat (also *e* in den vorigen Beispielen), können Sie anstelle der mit ON formulierten Bedingung einfach USING colname angeben. Aus ON *t1.e* = *t2.e* wird also das besser lesbare USING *e*. Diese Vorgehensweise funktioniert allerdings nicht, wenn die Primär- und Fremdschlüsselspalten unterschiedliche Namen haben.

```
SELECT * FROM t1 JOIN t2 USING e
```

## Left und Right Join

Joins werden verwendet, um Daten aus verknüpften Tabellen zusammenzuführen. Die vorhin behandelten Inner-Join-Varianten liefern nur Ergebnisse, bei denen die Werte der Verknüpfungsspalten übereinstimmen. Ein Left Join erfasst dagegen *alle* Datensätze der ersten Tabelle. Wenn es passende Zusatzdaten aus einer zweiten Tabelle gibt, werden sie in das Ergebnis eingebaut, wenn nicht, bleiben die entsprechenden Spalten leer. Ein Right Join funktioniert analog, erfasst aber alle Datensätze der zweiten Tabelle und kombiniert diese – soweit möglich – mit passenden Daten aus der ersten Tabelle.

Eine wichtige Anwendung von Left oder Right Joins liegt vor, wenn im Fremdschlüsselfeld NULL zulässig ist. Das bedeutet, dass ein Datensatz nicht zwingend mit Daten einer anderen Tabelle verknüpft ist.

Bei der Auswertung solcher Datensätze stellt sich nun die Frage: Sind Sie nur an Datensätzen interessiert, bei denen das Verknüpfungsfeld einen gültigen Verweis enthält, oder wollen Sie auch solche Datensätze sehen, bei denen das Verknüpfungsfeld NULL ist? Ein Inner Join entspricht der ersten Variante, ein Left Join der zweiten Variante. Beim Left Join tritt aber das Problem auf, dass es für die Ergebnisspalten aus der zweiten Tabelle gar keine Daten gibt. Stattdessen müssen lauter NULL-Werte übergeben werden:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.e = t2.e
```

a	b	e	c	d	e
--	---	----	---	----	----
1	III	12	34	efg	12
2	XI	13	NULL	NULL	NULL
3	V	NULL	NULL	NULL	NULL
4	VI	27	35	opq	27
5	II	NULL	NULL	NULL	NULL
6	IX	8	37	uvw	8
7	X	15	NULL	NULL	NULL

Ein Right Join funktioniert analog wie ein Left Join, berücksichtigt nun aber Ergebnisse, bei denen das Verknüpfungsfeld der rechten Spalte NULL ist:

```
SELECT * FROM t1 RIGHT JOIN t2 ON t1.e = t2.e
```

a	b	e	c	d	e
-----	-----	-----	---	-----	-----
NULL	NULL	NULL	33	abc	NULL
1	III	12	34	efg	12
4	VI	27	35	opq	27
NULL	NULL	NULL	36	ijk	NULL
6	IX	8	37	uvw	8
NULL	NULL	NULL	38	xyz	9

Genau genommen sind ein Left und ein Right Join exakt dieselben Operationen. Der einzige Unterschied besteht in der Reihenfolge, in der die Tabellen ausgewertet werden. Die beiden folgenden Abfragen sind daher gleichwertig:

```
SELECT t1.*, t2.* FROM t1 RIGHT JOIN t2 ON t1.e = t2.e
```

```
SELECT t1.*, t2.* FROM t2 LEFT JOIN t1 ON t1.e = t2.e
```

Da wir in unserem Kulturkreis von links nach rechts lesen, schreiben und gewissermaßen auch denken, erscheint uns ein Left Join meist intuitiver.

In der Praxis haben Left Joins (und, wenn Sie wollen, natürlich auch Right Joins) eine große Relevanz. Nehmen Sie an, Sie wollen in der *books*-Datenbank alle Titel samt Sprache auflisten. Mit einem Inner Join erhalten Sie bei der Musterdatenbank 117 Ergebnisse – und Sie haben vielleicht den Eindruck, das Ergebnis ist vollständig:

```
SELECT b.title, l.name AS language
FROM books b JOIN languages l ON b.langId = l.id
```

title	language
A Guide to the SQL Standard	english
...	(117 Datensätze)

Erst wenn Sie einen Left Join durchführen, bemerken Sie, dass die Tabelle *books* 127 Datensätze enthält. Darunter befinden sich einige, bei denen die Sprache des Buchs nicht gespeichert wurde.

```
SELECT b.title, l.name AS language
FROM books b LEFT JOIN languages l ON b.langId = l.id
```

title	language
A Guide to the SQL Standard	english
The Definitive Guide to Excel VBA	NULL
...	(127 Datensätze)

## Full Join

Ein Full Join kombiniert einen Left und einen Right Join. Das Ergebnis enthält sowohl Datensätze, bei denen das Verknüpfungsfeld der ersten Spalte NULL ist, als auch solche, bei denen das der zweiten Spalte NULL ist. Bei einigen DBMS funktioniert ein Full Join so:

```
SELECT * FROM t1 FULL JOIN t2 ON t1.e = t2.e
```

Es gibt allerdings auch DBMS, die den Full Join gar nicht unterstützen. Dazu zählt auch MySQL. In diesem Fall müssen Sie einen Left Join und einen Right Join kombinieren und die Ergebnisse mit UNION zusammenführen. UNION kombiniert hier die Ergebnisse

von zwei Abfragen und eliminiert gleichzeitig alle Doppelgänger. Der Prozess lässt sich sehr schön veranschaulichen (siehe Abbildung 10.3).

```
(SELECT * FROM t1 LEFT JOIN t2 ON t1.e = t2.e) UNION
(SELECT * FROM t1 RIGHT JOIN t2 ON t1.e = t2.e)
```

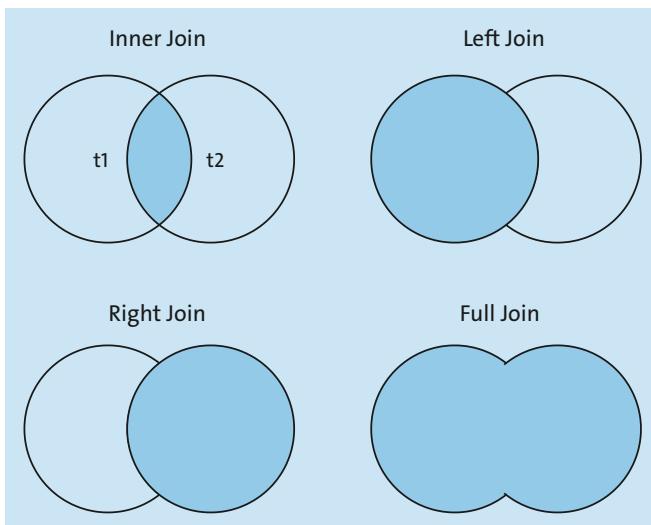


Abbildung 10.3 Grafische Veranschaulung der wichtigsten vier Join-Varianten

### Joins sind keine elementaren Mengenoperatoren

Die Darstellung von Join-Varianten durch ein Venn-Diagramm wie in Abbildung 10.3 ist unter Experten umstritten. Der Grund: Venn-Diagramme helfen bei der Visualisierung der Mengenlehre. Joins sind aber keine elementaren Mengenoperationen. Vielmehr handelt es sich um kartesische Produkte, deren Ergebnisse auf unterschiedliche Art gefiltert werden:

<https://blog.jooq.org/say-no-to-venn-diagrams-when-explaining-joins>

Warum ist Abbildung 10.3 dennoch im Buch? Weil sie perfekt veranschaulicht, wie im einen Fall alle Datensätze von t1, im anderen Fall alle Datensätze von t2 berücksichtigt werden usw. Beachten Sie aber, dass die Größe der Überlappungsbereiche nicht der Anzahl der zurückgegebenen Datensätze entspricht!

Wenn ein DBMS keinen Full Join unterstützt, ist das kein großer Verlust: In der Praxis gibt es so gut wie nie die Notwendigkeit für einen Full Join. In aller Regel erfolgt die Verknüpfung zweier Tabellen über eine Primär- und eine Fremdschlüsselspalte.

Für die Primärschlüsselalte gilt aber immer das Attribut NOT NULL, d. h., dort ist der Wert NULL ohnedies unmöglich. Deswegen kann bei Join-Operationen immer nur auf einer Seite der Wert NULL auftreten, nie auf beiden.

### Self Join

Die Bezeichnung *Self Join* wird manchmal verwendet, wenn eine Tabelle mit sich selbst verknüpft wird. Das erscheint auf den ersten Blick sinnlos zu sein. Tatsächlich kann eine Tabelle aber sehr wohl eine Fremdschlüsselalte enthalten, die auf die eigene Tabelle verweist – z. B. in einer Mitarbeitertabelle auf den übergeordneten Gruppenleiter oder die Chefin.

In der *books*-Datenbank ist dieser Fall in der Tabelle *categories* abgebildet, bei der die Spalte *parent* auf die übergeordnete Kategorie verweist:

```
SELECT * FROM categories WHERE id=2
```

id	name	parent
2	Databases	1

Syntaktisch unterschieden sich Self Joins nicht von anderen Joins. Die einzige Besonderheit ist, dass FROM und JOIN jeweils die gleiche Tabelle nennen. Die folgende Abfrage liefert zu jeder Kategorie die entsprechende Überkategorie. Ein Sonderfall ist dabei der Datensatz mit der ID 11. Er steht an der Spitze der Hierarchie, d. h., es gibt keine Überkategorie.

```
SELECT c.*, pc.name AS parentcategory
FROM categories c
LEFT JOIN categories pc ON c.parent = pc.id
```

id	name	parent	parentcategory
1	Computer books	11	All books
2	Databases	1	Computer books
3	Programming	1	Computer books
4	Relational Databases	2	Databases
5	Object-oriented databases	2	Databases
6	PHP	3	Programming
7	Perl	3	Programming
8	SQL	2	Databases
9	Children's books	11	All books
10	Literature and fiction	11	All books
11	All books	NULL	NULL
...			

## Joins für n:m-Verknüpfungen

Um eine n:m-Verknüpfung in einer Abfrage aufzulösen, benötigen Sie einfach zwei Joins, die die erste Tabelle mit der Verknüpfungstabelle und diese dann mit der zweiten Tabelle verbinden. Bei der *books*-Datenbank liegt eine derartige Verknüpfung zwischen Autoren und Titel vor.

Das folgende Beispiel zeigt den entsprechenden Join. Dank LEFT JOIN liefert das Kommando auch Buchtitel, denen gar kein Autor zugeordnet ist (was hier beim Python-Buch der Fall ist). Beachten Sie, dass Titel mit mehreren Autoren (hier nur »Time and Relational Theory«) im Ergebnis mehrfach vorkommen. Das ist optisch unübersichtlich und erschwert die Verarbeitung. Abhilfe schafft die Gruppierung des Ergebnisses durch GROUP BY, was uns direkt zum nächsten Thema führt.

```
SELECT b.title, a.name
FROM books b
LEFT JOIN writtenBy w ON w.bookId = b.id
JOIN authors a      ON a.id = w.authId
WHERE b.id >= 167
```

title	name
The Relational Model for Database Management	Codd Edgar
Time and Relational Theory	Date Chris
Time and Relational Theory	Darwen Hugh
Time and Relational Theory	Lorentzos Nikos
Database Design and Relational Theory	Date Chris
Python	NULL

Wenn Sie Join-Operationen auf die Spitze treiben möchten, können Sie eine Liste aller Buchtitel mit Verlag, Kategorie, Sprache und Autoren erstellen:

```
SELECT b.title, p.name AS publisher, c.name AS category,
       l.name AS language, a.name AS author
  FROM books b
LEFT JOIN writtenBy wb ON wb.bookId = b.id
JOIN authors a      ON a.id = wb.authId
LEFT JOIN publishers p ON b.publId = p.id
LEFT JOIN categories c ON b.catId = c.id
LEFT JOIN languages l  ON b.langId = l.id
```

title	publisher	category	language	author
A Guide to ...	Addison...	SQL	english	Date Chris
A Guide to ...	Addison...	SQL	english	Darween Hugh
...				

## 10.4 Ergebnisse gruppieren (GROUP BY)

Mit GROUP BY können Sie ein Kriterium nennen, gemäß dem das DBMS die Abfrageergebnisse in Gruppen zusammenfasst. In der Liste der Ergebnisspalten sind nur die folgenden Spalten bzw. Ausdrücke erlaubt:

- ▶ die bei GROUP BY genannte Spalte bzw. der an GROUP BY übergebene Ausdruck
- ▶ sonstige Spalten, wenn darauf eine Aggregatfunktion angewendet wird

Aggregatfunktionen ermitteln aus einer Gruppe von Werten ein zusammenfassendes Ergebnis – z. B. den kleinsten Wert, den größten Wert, die Anzahl oder die Summe der Werte (siehe Tabelle 10.4).

Funktion	Bedeutung
AVG	Bildet den Durchschnittswert.
COUNT	Zählt Datensätze ungleich NULL.
MAX	Ermittelt den größten Wert.
MIN	Ermittelt den kleinsten Wert.
SUM	Bildet die Summe.

**Tabelle 10.4** Die fünf wichtigsten Aggregatfunktionen

Das erste Beispiel ermittelt, wie viele Bücher in welcher Sprache in der Tabelle *books* gespeichert sind. Dank LEFT JOIN erscheinen auch solche Bücher im Ergebnis, zu denen gar keine Sprache gespeichert ist, bei denen *langId* also NULL ist:

```
SELECT l.name, COUNT(*)
FROM books b LEFT JOIN languages l ON l.id = b.langId
GROUP BY l.id

name      COUNT(*)
-----  -----
NULL        10
english     30
deutsch     66
svensk      20
norsk       1
```

Nicht zulässig ist die folgende Abfrage:

```
SELECT b.title, l.name, COUNT(*)
FROM books b LEFT JOIN languages l ON l.id = b.langId
GROUP BY l.id
```

Das DBMS erstellt Gruppen aus Datensätzen, bei denen *langId* übereinstimmt. Das Problem besteht darin, dass in diesem Fall der Buchtitel undefiniert ist. Innerhalb jeder Gruppe gibt es mehrere Titel, das DBMS weiß nicht, welchen es verwenden soll.

### **NULL wird nicht gezählt**

COUNT(column) zählt nur Datensätze, bei denen column nicht NULL ist. Analog addiert SUM(column) nur Werte ungleich NULL und ignoriert alle anderen. Dementsprechend liefern SUM(column)/COUNT(column) ebenso wie AVG(column) den Durchschnitt aller Werte ungleich NULL.

## Aggregatfunktionen ohne GROUP BY anwenden

Aggregatfunktionen können auch ohne GROUP BY direkt auf Tabellen angewendet werden. Die folgenden drei Beispiele ermitteln die Anzahl der englischsprachigen Bücher in *books*, die höchste ID in *books* sowie die Anzahl der unterschiedlichen Werte in der *langId*-Spalte von *books*. Beachten Sie beim letzten Beispiel das Schlüsselwort DISTINCT. Es ist erforderlich, damit das DBMS nicht einfach die Anzahl aller Datensätze zurückgibt, sondern tatsächlich die Anzahl unterschiedlicher Werte zählt:

```
SELECT COUNT(*) FROM books WHERE langId = 1
30
```

```
SELECT MAX(id) FROM books
170
```

```
SELECT COUNT(DISTINCT langId) FROM books
4
```

## Weitere Aggregatfunktionen

Der SQL-Standard sieht eine Menge weiterer Aggregatfunktionen vor allem für statistische Aufgaben vor, z. B. RANK, REGR\_AVGX oder PERCENTILE\_CONT). Allerdings implementieren viele DBMS nur ein Subset der im Standard vorgesehenen Funktionen.

Dafür bieten einige DBMS Aggregatfunktionen außerhalb des SQL-Standards an. Besonders praktisch ist die Möglichkeit, Zeichenketten aus einer Gruppe von Datensätzen zu verbinden. Das gelingt z. B. mit GROUP\_CONCAT in MySQL, mit LISTAGG in Oracle oder mit STRING\_AGG mit SQL Server oder in PostgreSQL. Manche DBMS können Gruppenergebnisse sogar im JSON- oder XML-Format zusammenführen, was bei der Weiterverarbeitung sehr hilfreich sein kann.

Die folgende Abfrage führt zuerst wie im letzten Beispiel von [Abschnitt 10.3, »Tabellen verknüpfen \(JOIN\)«](#), Buchtitel und Autoren zusammen. Anschließend wird für jedes Buch eine Gruppe gebildet (GROUP BY b.id). Für jede Gruppe wird mit COUNT die

Anzahl der Autoren und mit GROUP\_CONCAT eine alphabetisch geordnete Autorenliste erstellt.

```
SELECT MIN(b.title) AS bookTitle,
       COUNT(a.id) AS cnt,
       GROUP_CONCAT(a.name ORDER BY a.name SEPARATOR ', ') AS authors
  FROM books b
 LEFT JOIN writtenBy w ON w.bookId = b.id
 LEFT JOIN authors a   ON a.id = w.authId
 GROUP BY b.id

bookTitle      cnt      authors
-----  -----  -----
Grundkurs P...    4      Ratz Dietmar, Scheffler Jens, Seese ...
Tiroler Sk...    2      Hüttl Franz, Pokos Kurt
Schitourenp...   2      Auferbauer Günter, Auferbauer Luise
Git            2      Kofler Michael, Öggel Bernd
...
...
```

Ein wenig befremdlich ist bei diesem Kommando die MIN-Funktion. Sie wählt aus mehreren Zeichenketten die alphabetisch erste aus, hier also den alphabetisch ersten Titel aus einer Gruppe von Ergebnisdatensätzen. Weil bei diesem Kommando die Gruppierung über die ID der Bücher erfolgt, ist aber der Titel aller Ergebnisdatensätze einer Gruppe ohnedies immer gleich. Insofern ist MIN eigentlich sinnlos: Die Funktion ermittelt den alphabetisch ersten aus lauter gleichen Titeln. Dennoch muss eine Aggregatfunktion angegeben werden, damit das Kommando den SQL-Syntaxregeln entspricht.

Bei MySQL dürfen Sie hier sogar auf die MIN-Funktion verzichten. MySQL erkennt, dass die Gruppierung anhand von *books.id* erfolgt und der Titel innerhalb der Gruppe daher eindeutig ist. Dessen ungeachtet entspricht das verkürzte Kommando SELECT title, ... GROUP BY b.id nicht dem SQL-Standard. Andere DBMS würden es mit einem Fehler zurückweisen.

## Mehrstufige Gruppierung

Es ist zulässig, an GROUP BY mehrere Kriterien zu übergeben. Dann bildet das DBMS für jede auftretende Kombination aller Kriterien Zwischenergebnisse. Das Ergebnis wird häufig ziemlich unübersichtlich und eignet sich nur für eine maschinelle Auswertung, z. B. durch ein Statistikprogramm.

```
SELECT p.name AS publisher, l.name AS language, COUNT(*)
  FROM books b
 LEFT JOIN languages l ON l.id = b.langId
 LEFT JOIN publishers p ON p.id = b.publId
 GROUP BY b.langId, b.publId
```

publisher	language	COUNT(*)
Addison-Wesley	deutsch	26
Addison-Wesley	english	5
Hanser	deutsch	2
NULL	deutsch	8
NULL	NULL	1
O'Reilly & Associates	english	3
NULL	norsk	1
Rheinwerk Verlag	deutsch	5
...		

### Bedingungen für Gruppen (HAVING)

Manchmal ist es zweckmäßig, das Suchergebnis nach Kriterien zu selektieren, deren Werte sich erst durch die Berechnung von Aggregatfunktionen ergeben. Sie suchen nach Filialen, deren Umsätze (SUM) im letzten Monat größer als 100.000 Euro waren, nach Kunden, die im letzten Jahr mehr als 10 Bestellungen durchgeführt haben usw.

Es ist naheliegend, aber falsch, derartige Bedingungen wie bisher mit WHERE zu formulieren. Vielmehr müssen Sie die Bedingungen, die sich auf Gruppeneigenschaften beziehen, mit dem Schlüsselwort HAVING angeben. Das ist deswegen zweckmäßig, weil auf diese Weise zwischen Bedingungen unterschieden wird, die schon vor der Gruppierung berücksichtigt werden sollen (WHERE), und anderen, die erst danach anzuwenden sind (HAVING).

Die folgende Abfrage ermittelt jene Kategorien der books-Datenbank, denen zehn oder mehr Bücher zugeordnet sind:

```
SELECT c.name AS category, COUNT(*)
FROM books b JOIN categories c ON c.id = b.catId
GROUP BY c.id
HAVING COUNT(*) >= 10
```

category	COUNT(*)
Databases	10
Literature and fiction	22
History	10

Wenn bei dieser Auswertung nur englischsprachige Titel berücksichtigt werden sollen, dann müssen Sie in die Abfrage zuerst WHERE, dann GROUP BY und zuletzt HAVING einbauen:

```

SELECT c.name AS category, COUNT(*)
FROM books b JOIN categories c ON c.id = b.catId
WHERE b.langId = 1
GROUP BY c.id
HAVING COUNT(*) >= 5

category          COUNT(*)
-----
MySQL              5
Databases          7
History             5

```

### Die richtige Reihenfolge ist entscheidend!

Die SQL-Syntax ist bei der Reihenfolge der optionalen Schlüsselwörter ausgesprochen pingelig. Wenn Ihr DBMS ein SELECT-Kommando nicht akzeptiert, das absolut korrekt aussieht, sollten Sie unbedingt einen Blick in [Abschnitt 10.7](#), »SELECT-Syntax-Zusammenfassung«, werfen, wo ich die korrekte Reihenfolge aller SELECT-Komponenten zusammengefasst habe.

## Window-Funktionen als Alternative zu GROUP BY

Alle gängigen DBMS stellen mit OVER sogenannte *Window-Funktionen* zur Verfügung, um in SELECT-Ergebnissen nach Gruppen zusammengefasste Zwischenergebnisse einzubauen. Window-Funktionen bieten – vor allem in Kombination mit Subqueries – oft eine gute Alternative zu GROUP-BY-Ausdrücken. Window-Funktionen ermöglichen es insbesondere, den (nach einem bestimmten Kriterium) ersten oder letzten Datensatz jeder Gruppe zu extrahieren – eine Aufgabe, die mit GROUP BY nicht ohne Weiteres lösbar ist. Syntaxdetails und Anwendungsbeispiele folgen in [Abschnitt 13.3](#), »Window-Funktionen (OVER)«.

## 10.5 Ergebnisse sortieren (ORDER BY)

Die ungeordneten Abfrageergebnisse in den vorangegangenen Listings waren zugegebenermaßen irritierend. Es gibt aber natürlich einen Grund, warum ich Ihnen das vergleichsweise simple Schlüsselwort ORDER BY nicht schon längst vorgestellt habe: Ich habe die Abschnitte dieses Kapitels in der Reihenfolge geordnet, in der die einzelnen Schlüsselwörter im SELECT-Kommando vorkommen – und ORDER BY hat seinen Platz ziemlich am Ende nach GROUP BY.

Mit ORDER BY geben Sie einen oder mehrere Spalten bzw. in der Ergebnisliste vorkommende Ausdrücke an, die als Sortierkriterium dienen. Normalerweise werden

die Ergebnisse aufsteigend sortiert, also die kleinen Werte zuerst. Eine abfallende Sortierordnung erreichen Sie mit dem nachgestellten Schlüsselwort DESC (also beispielsweise ORDER BY column DESC).

Das folgende Kommando liefert eine geordnete Liste aller Titel der *books*-Tabelle:

```
SELECT id, title, subtitle FROM books ORDER BY title

id      title                      subtitle
----- -----
11     A Guide to the SQL Standard    A User's Guide ...
52     A Programmer's Introduction to PHP  NULL
164    A Promised Land                NULL
...
...
```

Das zweite Beispiel liefert eine Liste der Autoren englischsprachiger Bücher. Die Liste ist abfallend nach der Anzahl der Bücher geordnet; wenn es mehrere Autoren gibt, die gleich viele Bücher (mit)verfasst haben, sind ihre Namen aufsteigend geordnet.

```
SELECT a.name, COUNT(*)
FROM authors a
JOIN writtenBy w ON w.authId = a.id
JOIN books b ON w.bookId = b.id
WHERE b.langId = 1
GROUP BY a.id
ORDER BY COUNT(*) DESC, a.name
```

name	COUNT(*)
Iggulden Conn	5
Date Chris	4
Coetzee J. M.	2
DuBois Paul	2
...	

## 10.6 Ergebnisse limitieren (LIMIT)

Oft brauchen Sie nicht alle Ergebnisse, sondern es reichen die ersten 10 oder 100. Es hat relativ lange gebraucht, bis im SQL Standard eine derartige Möglichkeit vorgesehen wurde. Aktuelle Versionen von PostgreSQL, Oracle und SQL Server folgen dem Standard. Die Syntax sieht so aus:

```
-- Oracle, PostgreSQL, SQL Server
SELECT ...
[OFFSET start ROWS] FETCH NEXT|FIRST n ROWS ONLY
```

Damit werden zuerst *start* Ergebnisdatensätze übersprungen und dann nur die nächsten bzw. ersten *n* Datensätze ausgeliefert.

Bis sich die SQL-Standardisierungsgremien endlich auf diese Lösung einigten, hatten viele DBMS längst eine andere, syntaktisch viel elegantere Lösung implementiert:

```
-- MySQL, PostgreSQL, SQLite
SELECT ...
LIMIT n [OFFSET start]
```

Bei einigen DBMS ist außerdem die folgende Kurzschreibweisen bzw. Varianten erlaubt:

```
-- MySQL, SQLite
SELECT ... LIMIT start, n
```

Beim SQL Server gibt es schließlich noch diese Variante:

```
-- SQL Server
SELECT TOP n FROM ...
```

Sie sehen also, je nachdem, welches DBMS Sie verwenden, gibt es unterschiedliche Syntaxvarianten, um das gleiche Ziel zu erreichen. Streng genommen ist die Limitierung von SELECT aber nur sinnvoll, wenn das Abfrageergebnis durch ORDER BY geordnet ist. Manche DBMS akzeptieren deswegen LIMIT, OFFSET bzw. FETCH nur, wenn Sie gleichzeitig ORDER BY einsetzen.

Bei allen anderen DBMS hängt es vom Zufall bzw. von der internen Implementierung des DBMS ab, welche Datensätze durch LIMIT oder OFFSET ausgewählt werden. Dessen ungeachtet ist SELECT \* FROM tablename LIMIT 1 ausgesprochen praktisch, um sich mit minimalem Tippaufwand den Aufbau einer Tabelle in Erinnerung zu rufen.

### Vorsicht, ineffizient

Die Limitierung des Ergebnisses wird häufig eingesetzt, um in Client-Programmen oder Weboberflächen Seite für Seite durch ein Ergebnis zu blättern: LIMIT 20 liefert also die ersten 20 Datensätze, LIMIT 20 OFFSET 20 die nächsten 20, LIMIT 20 OFFSET 40 weitere 20 usw. Intern muss dazu aber das gesamte SELECT-Kommando immer wieder vollständig ausgeführt werden, obwohl vom Ergebnis letztlich nur wenige Datensätze benötigt werden. Bei großen Datenbanken ist das ausgesprochen ineffizient.

Markus Winand empfiehlt in seinem E-Book »SQL Performance Explained« andere Wege, das Ergebnis in kleinere Gruppen zu zerlegen. Wenn Sie z. B. die Weboberfläche eines Newstickers programmieren, könnten Sie statt einer vorgegebenen Ergebnisanzahl ein Blättern für Datumsbereiche ermöglichen (Beiträge von heute, von gestern, von vorgestern usw.).

Der Vorteil dieser Vorgehensweise besteht darin, dass nun eine WHERE-Bedingung über die Ergebnisse entscheidet. Das DBMS kann derartige Bedingungen wesentlich besser optimieren. Mehr Details können Sie hier nachlesen:

<https://use-the-index-luke.com/de/sql/partielle-ergebnisse/blättern>

Zuletzt ein konkretes Beispiel: Die folgende Abfrage erstellt zuerst eine Liste, wie viele Titel der *books*-Datenbank in welchem Jahr erschienen sind. Diese Liste wird abfallend sortiert, sodass Jahre mit vielen Titeln zuerst angezeigt werden. Dank LIMIT werden nur die Top-5-Jahre angezeigt. Die Bedingung IS NOT NULL ist ein Vorgriff auf Abschnitt 10.8, »Der Umgang mit NULL«, sollte aber auf Anhieb verständlich sein.

```
SELECT publYear, COUNT(*) FROM books
WHERE publYear IS NOT NULL
GROUP BY publYear
ORDER BY COUNT(*) DESC
LIMIT 5

publYear    COUNT(*)
-----  -----
2012          11
2003           7
2000           6
2011           5
2002           4
```

## 10.7 SELECT-Syntax-Zusammenfassung

Dieser Abschnitt ist ebenso kurz wie wichtig. Er fasst die wichtigsten Komponenten des SELECT-Kommandos in einem kompakten Listing zusammen.

```
SELECT [DISTINCT] column1 [[AS] alias1],
       column2 [[AS] alias2] ...
FROM table1 [[AS] t1]
[, [LEFT|RIGHT] JOIN table2 t2 ON t1.colX = t2.colY]
[, [LEFT|RIGHT] JOIN table3 t3 USING colZ ]
[, NATURAL [LEFT|RIGHT] JOIN table4 t4 ]
[, ... ]
[ WHERE condition1 [AND|OR condition2 ...] ]
[ GROUP BY groupexpr1 [, groupexpr2 ...]
  [ HAVING groupcond1 [AND|OR groupcondition2 ...] ] ]
[ WINDOW mywindow AS (windowexpression) ]
[ ORDER BY col1 [DESC] [, col2 [DESC] ...] ]
[ LIMIT cnt [OFFSET start]]
```

Entscheidend ist dabei die Reihenfolge der Schlüsselwörter: Auch wenn die meisten Bestandteile von SELECT optional sind, so müssen Sie sich doch exakt an die hier vorgegebene Reihenfolge halten!

Anstelle von LIMIT müssen Sie bei einigen DBMS die umständlichere, aber dafür standardkonforme Syntax OFFSET start ROWS FETCH NEXT n ROWS ONLY anwenden (siehe Abschnitt 10.6, »Ergebnisse limitieren (LIMIT)«). Das Schlüsselwort WINDOW, mit dem Sie Ausdrücke zur Partitionierung von Window-Funktionen formulieren können, stelle ich Ihnen in Abschnitt 13.3, »Window-Funktionen (OVER)«, vor.

## 10.8 Der Umgang mit NULL

Soweit die Spalten einer Tabelle nicht explizit mit NOT NULL deklariert sind, ist NULL ein zulässiger Wert. Bei Abfrageergebnissen wird dann einfach NULL angezeigt. Wenn die Ergebnisse durch ein Client-Programm weiterverarbeitet werden, muss der Sonderfall NULL immer berücksichtigt werden.

Dieser Abschnitt fasst zusammen, wie relationale DBMS mit NULL umgehen und welche Funktionen den Umgang mit NULL erleichtern. Grundsätzlich gilt, dass Berechnungen, in die NULL als Argument einfließt, in aller Regel NULL ergeben. (Die SQL-Funktion CONCAT verbindet Zeichenketten.)

```
SELECT 1 + 2 + NULL  
      NULL
```

```
SELECT 3 > NULL  
      NULL
```

```
SELECT CONCAT('abc', NULL)  
      NULL
```

Aggregatfunktionen zeigen ein davon abweichendes Verhalten: Wenn beispielsweise SUM die Summe einer Gruppe von zehn Werten ausrechnen soll und fünf dieser Werte NULL sind, dann berücksichtigt SUM einfach nur die anderen fünf Werte und ermittelt das Ergebnis korrekt (und liefert also nicht NULL als Ergebnis). Analog zählt COUNT nur Datensätze, bei denen die durch COUNT angegebene Spalte ungleich NULL ist.

### Boolesche Logik

Ein Sonderfall ist die Auswertung boolescher Ausdrücke. Hier gilt die dreiwertige Kleene-Logik (siehe Abschnitt 9.1, »Relationale Algebra«). Falls Ihr DBMS die Schlüsselwörter TRUE und FALSE nicht kennt, verwenden Sie stattdessen 1 und 0.

```
SELECT TRUE AND NULL  
NULL
```

```
SELECT NOT NULL  
NULL
```

```
SELECT TRUE OR NULL  
1
```

```
SELECT FALSE AND NULL  
0
```

### Vergleich mit NULL

Um zu testen, ob eine Spalte den Wert NULL enthält bzw. nicht enthält, müssen Sie den Operator `IS NULL` bzw. `IS NOT NULL` verwenden. Manche DBMS kennen alternativ auch die Funktion `ISNULL(column)`, diese entspricht aber nicht dem SQL-Standard. Vorsicht: Der Vergleich `column = NULL` funktioniert nicht erwartungsgemäß, weil dieser Ausdruck immer NULL ergibt.

Die folgende Abfrage ermittelt die Anzahl der Titel in der `books`-Datenbank, bei denen es keine Verlagsinformationen gibt:

```
SELECT COUNT(*) FROM books WHERE publId IS NULL  
34
```

Problematisch ist der Vergleich `column IN (a, b, c)`, wenn die an `IN` übergebene Liste NULL enthält:

```
SELECT 2 IN (1, 2, 3, NULL)          -- erwartungsgemäß True  
1
```

```
SELECT 4 IN (1, 2, 3, NULL)          -- erwartungsgemäß False  
0
```

```
SELECT NULL IN (1, 2, 3, NULL)      -- Vorsicht: NULL!  
NULL
```

Geradezu absurd mutet das Verhalten bei `column NOT IN (a, b, c)` an, wenn die Liste NULL enthält:

```
SELECT 2 NOT IN (1, 2, 3, NULL)     -- erwartungsgemäß False  
0
```

```
SELECT 4 NOT IN (1, 2, 3, NULL)     -- Vorsicht: liefert NULL!  
NULL
```