

Leider ist IF nicht Teil des SQL-Standard und steht nur in manchen DBMS zur Verfügung (in SQL Server und in SQLite unter dem Namen IIF). Eine Alternative ist die standardkonforme CASE-Konstruktion. Es gibt zwei Syntaxvarianten:

```
CASE expression
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  ELSE defaultvalue
END

CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE defaultvalue
END
```

CASE führt auch zum Ziel, macht das Kommando aber noch länger:

```
SELECT publYear,
       SUM(CASE WHEN langId=1 THEN 1 ELSE 0 END) AS English,
       SUM(CASE WHEN langId=2 THEN 1 ELSE 0 END) AS Deutsch,
       SUM(CASE WHEN langId=3 THEN 1 ELSE 0 END) AS Svensk,
       SUM(CASE WHEN langId=4 THEN 1 ELSE 0 END) AS Norsk
  FROM books WHERE publYear IS NOT NULL
 GROUP BY publYear ORDER BY publYear;
```

Ein bisschen bewegen wir uns mit diesen Kommandos in die Richtung »Programmieren mit SQL«, das ich in diesem Buch in [Kapitel 18](#), »Stored Procedures und Trigger«, ausführlicher aufgreife.

13.2 Subqueries

Der Begriff *Subquery* bezeichnet den Umstand, dass zwei oder mehrere SQL-Kommandos ineinander verschachtelt sind. Zumeist handelt es sich um lauter SELECT-Kommandos. Viele DBMS bieten auch die Möglichkeit, den Wirkungsbereich eines UPDATE- oder DELETE-Kommandos durch ein eingebettetes SELECT-Kommando zu steuern. Subqueries sind seit SQL-92 Teil des SQL-Standards. Sie werden von fast allen marktüblichen DBMS unterstützt, wenn auch in unterschiedlichem Ausmaß.

Das folgende Einführungsbeispiel bezieht sich auf die Datenbank *books*. Die innere, in runde Klammern gestellte Abfrage ermittelt die IDs von Kategorien, für die es der Datenbank mehr als drei Titel gibt. Die äußere Abfrage wertet dieses Zwischenergebnis aus und liefert eine sortierte Liste aller Kategorienamen zurück:

```

SELECT name FROM categories
WHERE id IN (
    SELECT id, catId FROM books
    GROUP BY catId HAVING COUNT(*) > 3 )
ORDER BY name

id      name
--- -----
 9      Children's books
 1      Computer books
 96     Crime
...

```

Subquery versus Join

Manche Abfragen lassen sich wahlweise durch Join- oder Subquery-Konstruktionen bilden. Oft sind Subqueries für Menschen besser lesbar bzw. verständlich. Andererseits fällt vielen DBMS die Optimierung von Joins leichter als die von Subqueries; es kann also sein, dass ein gleichwertiges Kommando mit JOIN effizienter ausgeführt wird. Der folgende SQL-Code liefert dasselbe Ergebnis wie die vorige Subquery:

```

SELECT c.id, c.name FROM categories c
JOIN books b ON b.catId = c.id
GROUP BY b.catId HAVING COUNT(*) > 3
ORDER BY c.name

```

Syntax

Verschachtelte Abfragen bestehen zumindest aus zwei Teilen: Die Basisabfrage wird *Main Query* oder *Outer Query* genannt, die darin eingebettete Abfrage *Subquery* oder *Inner Query*. In den meisten Fällen wird die innere Abfrage ausgeführt, bevor die äußere ausgewertet wird. (Eine Ausnahme ist die Syntaxvariante mit EXISTS.)

Subqueries müssen in Klammern gestellt werden, damit das DBMS den Code der Outer und der Inner Query abgrenzen kann. Subqueries können an drei Orten in einem SQL-Kommando platziert werden: bei FROM, WHERE und HAVING. Im Folgenden stelle ich Ihnen einige Syntaxvarianten samt Beispiel vor.

- **Die Subquery liefert einen singulären Wert für einen direkten Vergleich:** Im einfachsten Fall ist die Subquery so formuliert, dass sie einen singulären Wert liefert. Dieser wird im äußeren Teil des SQL-Kommandos ausgewertet:

```
SELECT ... WHERE mycol = | < | > | <> (SELECT ...)
```

Im folgenden Beispiel ermittelt die Subquery die ID der Sprache »schwedisch«. Der restliche Teil der Abfrage liefert alle Buchtitel in dieser Sprache:

```
SELECT title, publYear FROM books
WHERE langId =
  (SELECT id FROM languages WHERE name = 'svensk')
```

Sollte die Subquery kein Ergebnis liefern (WHERE name = 'abc'), dann ergibt auch das äußere SELECT-Kommando kein Ergebnis (*empty set*). Liefert die Subquery dagegen mehrere Ergebnisse, kann das Gesamtkommando nicht weiter ausgeführt werden und wird mit einer Fehlermeldung abgebrochen. Diesen Fehler können Sie mit LIMIT 1 in der inneren Abfrage verhindern; allerdings hängt es vom Kontext der Abfrage ab, ob das eine sinnvolle Strategie ist. Wahrscheinlicher ist, dass Sie die innere Abfrage falsch formuliert haben.

- ▶ **Die Subquery liefert mehrere Werte für den Vergleich mit IN oder NOT IN:** Wenn die Subquery so formuliert ist, dass sie mehrere Ergebnisse liefert, erfolgt die Auswertung am einfachsten mit IN oder NOT IN:

```
SELECT ... WHERE mycol [NOT] IN (SELECT ...)
```

Das Einführungsbeispiel zu diesem Abschnitt entspricht diesem Muster. Mit NOT IN können Sie übrigens unkompliziert testen, ob es bei zwei miteinander verknüpften Tabellen Querverweise gibt, die ins Leere führen. (Das ist nur möglich, wenn es für die Tabellen keine Foreign-Key-Regeln gibt.)

Das folgende Beispiel sucht in der Tabelle *books* nach Titeln mit einem ungültigen Wert in der *publId*-Spalte. Bei der Musterdatenbank ist das Ergebnis leer, weil eine Foreign-Key-Regel die Einhaltung der referenziellen Integrität sicherstellt.

```
SELECT id, title FROM books
WHERE publId NOT IN
  (SELECT id FROM publishers);
```

- ▶ **Die Subquery liefert mehrere Werte für den Vergleich mit ANY oder ALL:** Gewissermaßen eine Kombination der beiden obigen Varianten ergibt sich, wenn die Subquery mehrere Einzelwerte liefert (also ein tabellarisches Ergebnis mit *einer* Spalte), die mit ANY oder ALL mit einer Spalte der Haupt-Query verglichen werden:

```
SELECT * WHERE mycol = | < | > | <> ANY (SELECT othercol ...)
```

```
SELECT * WHERE mycol = | < | > | <> ALL (SELECT othercol ...)
```

Wenn dem Vergleichsoperator ANY folgt, dann werden aus der Hauptabfrage alle Datensätze ausgewählt, bei denen es zumindest mit einem Datensatz der Subquery eine Übereinstimmung gibt.

Folgt dem Vergleichsoperator dagegen ALL, dann werden aus der Hauptabfrage nur solche Datensätze ausgewählt, für die der Vergleich mit *allen* Datensätzen der Subquery übereinstimmt. ALL in Kombination mit dem Operator = ist selten

zielführend: Ein Wert der Haupt-Query kann ja nicht gleichzeitig mit unterschiedlichen Werten der Subquery übereinstimmen. ALL lässt sich aber gut mit Kleiner-als-, Größer-als- oder Ungleichheitsoperatoren kombinieren. Am häufigsten ist die Kombination `mycol <> ALL (...)`.

Beispielsweise liefert die folgende Abfrage sämtliche Verlage, die im Jahr 2010 ein Buch veröffentlicht haben:

```
SELECT * FROM publishers
WHERE id = ANY
  (SELECT publId FROM books WHERE publYear = 2010)
```

Umgekehrt liefert die folgende Abfrage mit dem Vergleichsoperator `<> ALL` jene Verlage, die 2010 *kein* Buch veröffentlicht haben:

```
SELECT * FROM publishers
WHERE id <> ALL
  (SELECT publId FROM books WHERE publYear = 2010)
```

- ▶ **Die Subquery liefert eine virtuelle Tabelle, die das äußere Kommando auswertet:** Bei dieser Syntaxvariante wertet das äußere SELECT-Kommando die Ergebnisse aus, die die Subquery liefert. Es ist syntaktisch erforderlich, dass Sie dem temporären Zwischenergebnis einen Namen geben:

```
SELECT ... FROM (SELECT ...) [AS] tempresult WHERE ...
```

Sie können das Zwischenergebnis in weiterer Folge auch mit anderen Tabellen verknüpfen. Die folgende Beispielabfrage ermittelt zuerst in der Subquery Titel, deren Veröffentlichungsjahr unbekannt ist. Das Abfrageergebnis wird dann mit der Tabelle *publishers* verknüpft, sodass zu jedem Titel auch der Name des Verlags im Suchergebnis angezeigt werden kann:

```
SELECT tmp.id, tmp.title, p.name
FROM (SELECT id, title, publId
      FROM books WHERE publYear IS NULL) tmp
JOIN publishers p ON p.id = tmp.publId
```

- ▶ **Die Subquery liefert das Kriterium zur Auswahl von Datensätzen (EXISTS):** Eine eher ungewöhnliche Subquery-Konstruktion ermöglicht das Schlüsselwort EXISTS. Dabei wird, wenn man die vom DBMS durchgeführte Optimierung einmal beiseitelässt, die Subquery für jeden Datensatz der Haupt-Query ausgeführt. Wenn die Subquery ein oder mehrere Ergebnisse liefert, dann landet der zugehörige Datensatz der Haupt-Query im Ergebnis. Anders formuliert: Die Subquery entscheidet, welche Datensätze der Haupt-Query gültig sind.

```
SELECT ... WHERE EXISTS (SELECT ...)
```

Die folgende Abfrage ermittelt Verlage, die Bücher in der zweiten, dritten usw. Auflage veröffentlicht haben:

```
SELECT * FROM publishers p
WHERE EXISTS
  (SELECT * FROM books b
   WHERE b.publId = p.id AND b.edition > 1)
ORDER BY p.name
```

Abschnitt 13.7, »Wiederholungsaufgaben«, enthält weitere Fragestellungen für Subqueries. Den Lösungscode finden Sie wie immer im Anhang.

Subqueries mit HAVING

Subquery-Bedingungen können Sie nicht nur mit WHERE, sondern auch mit HAVING kombinieren. (Sie erinnern sich: HAVING verwenden Sie statt WHERE, um Bedingungen für Datensatzgruppen zu formulieren – siehe Abschnitt 10.4, »Ergebnisse gruppieren (GROUP BY)«.)

Das folgende Beispiel besteht aus gleich drei Ebenen: Die innerste Subquery ermittelt eine Liste mit der Anzahl der Bücher, die pro Verlag erschienen sind. Die Subquery in der mittleren Ebene ermittelt daraus den Mittelwert. Die Haupt-Query liefert eine Liste aller Verlage, die mehr Bücher veröffentlicht haben als der durchschnittliche Verlag:

```
SELECT p.name, COUNT(*)
FROM books b
JOIN publishers p ON b.publId = p.id
GROUP BY p.id HAVING COUNT(*) >
  (SELECT AVG(cnt) FROM
   (SELECT COUNT(*) AS cnt
    FROM books
    GROUP BY publId) AS tmp
  )
ORDER BY p.name
```

name	COUNT(*)
Addison-Wesley	34
Apress	5
Galileo Press	6
HarperCollins	5
Kiepenheuer & Witsch	5
Rheinwerk Verlag	6

Subqueries mit INSERT, UPDATE und DELETE kombinieren

Subqueries sind ein gutes Hilfsmittel, um die Wirkung von UPDATE- oder DELETE-Kommandos auf bestimmte Datensätze einer Tabelle einzuschränken:

```
UPDATE/DELETE ...
WHERE mycol IN (SELECT ...)
```

INSERT kann das Ergebnis eines SELECT-Kommandos in eine Tabelle einfügen. Diese Syntaxvariante habe ich Ihnen bereits in [Abschnitt 11.1, »Daten einfügen \(INSERT\)«](#), vorgestellt. Es ist aber unüblich, in diesem Kontext von »Subqueries« zu sprechen. Beachten Sie auch, dass das SELECT-Kommando in diesem Fall nicht in runde Klammern gestellt wird:

```
INSERT INTO mytable (col1, col2, col3, ...)
SELECT ... FROM othertable WHERE ...
```

Subqueries mit WITH definieren und anwenden

Es ist Ihnen vermutlich schon aufgefallen: Der endlose Code von SQL-Kommandos mit Subqueries ist oft sehr unübersichtlich. Mit dem Schlüsselwort WITH, das im SQL:1999-Standard als optionales Feature eingeführt wurde, können Sie Teilaufgaben quasi in der Einleitung eines SQL-Kommandos vorweg definieren. Das macht den Code einer Abfrage zwar nicht kürzer, aber oft besser zu lesen.

Die mit WITH definierten Teilaufgaben werden im SQL-Standard als *Common Table Expression* bezeichnet. CTEs sind benannte, temporäre Ergebnisse innerhalb eines SELECT-, INSERT-, UPDATE- oder DELETE-Kommandos. Ihre Anwendung ist keineswegs auf verschachtelte Abfragen beschränkt; vielmehr können CTEs bei jeder SQL-Abfrage eingesetzt werden, deren Code sich in mehrere unabhängige Teile zerlegen lässt. Die Grundidee besteht darin, dass der SQL-Code mit der Definition der CTEs beginnt. Die so definierten Ausdrücke können dann im Hauptcode angewendet werden. Die Grundsyntax sieht so aus:

```
WITH cte1 AS (query1),
      cte2 AS (query2)
SELECT col1, col2
FROM cte1 JOIN cte2 ON ...
```

Die Beispieldarstellung aus dem Abschnitt »Subqueries mit HAVING« lässt sich mit CTEs wie folgt neu formulieren:

```
WITH booksPerPublishers AS
  ( SELECT COUNT(*) AS cnt FROM books
    GROUP BY publId ),
avgPubl AS
  ( SELECT AVG(cnt) AS booksAvg FROM booksPerPublishers )
```

```
-- Hier beginnt die Hauptabfrage.  
SELECT p.name, COUNT(*)  
FROM books b  
JOIN publishers p ON b.publId = p.id  
GROUP BY p.id HAVING COUNT(*) > (SELECT booksAvg FROM avgPubl)  
ORDER BY p.name
```

Die meisten gängigen DBMS unterstützen WITH in ihren aktuellen Versionen – MySQL z. B. seit Version 8, MariaDB seit Version 10.2. Allerdings gibt es verschiedene Syntaxvarianten. Einen guten Überblick, welches DBMS mit welcher WITH-Variante zurechtkommt, gibt die folgende Seite:

<https://modern-sql.com/feature/with>

WITH in rekursiven Abfragen

In Abschnitt 13.4, »Rekursion«, werden wir noch einmal auf WITH stoßen. Das Schlüsselwort spielt auch bei der Formulierung rekursiver Abfragen eine wichtige Rolle.

13.3 Window-Funktionen (OVER)

Mit GROUP BY können Sie mehrere Datensätze eines Abfrageergebnisses zu einer Gruppe zusammenfassen und für diese Gruppe Aggregatfunktionen berechnen – z. B. den Auftragswert aller im April durchgeföhrten Bestellungen. Window-Funktionen können ebenfalls Funktionen auf eine Gruppe von Datensätzen anwenden, das Ergebnis aber in jeden einzelnen Ergebnisdatensatz einbauen. Daraus ergeben sich im Vergleich zu GROUP BY bessere Verarbeitungsmöglichkeiten.

Window-Funktionen und das Schlüsselwort OVER wurden zwar erst 2016 als optionales Feature in den SQL-Standard aufgenommen, es steht aber bei einigen DBMS schon wesentlich länger zur Verfügung (in Oracle seit 2007, im SQL Server seit 2008). Welche der gängigen DBMS Window-Funktionen ab welcher Version unterstützen, ist auf dieser Webseite zusammengefasst:

[https://modern-sql.com/caniuse/over_\(empty_over_clause\)](https://modern-sql.com/caniuse/over_(empty_over_clause))

Das folgende Beispiel verdeutlicht die Grundidee von OVER: Wird diese Funktion nach einer Aggregatfunktion angegeben, dann wird diese Aggregatfunktion über alle Datensätze des Ergebnisses berechnet. Mit OVER(PARTITION BY col) wird die Aggregatfunktion für alle Datensätze errechnet, die in die gleiche Partition (in das gleiche »Ergebnisfenster«, daher *Window Function*) fallen wie der aktuelle Datensatz.