O'REILLY®

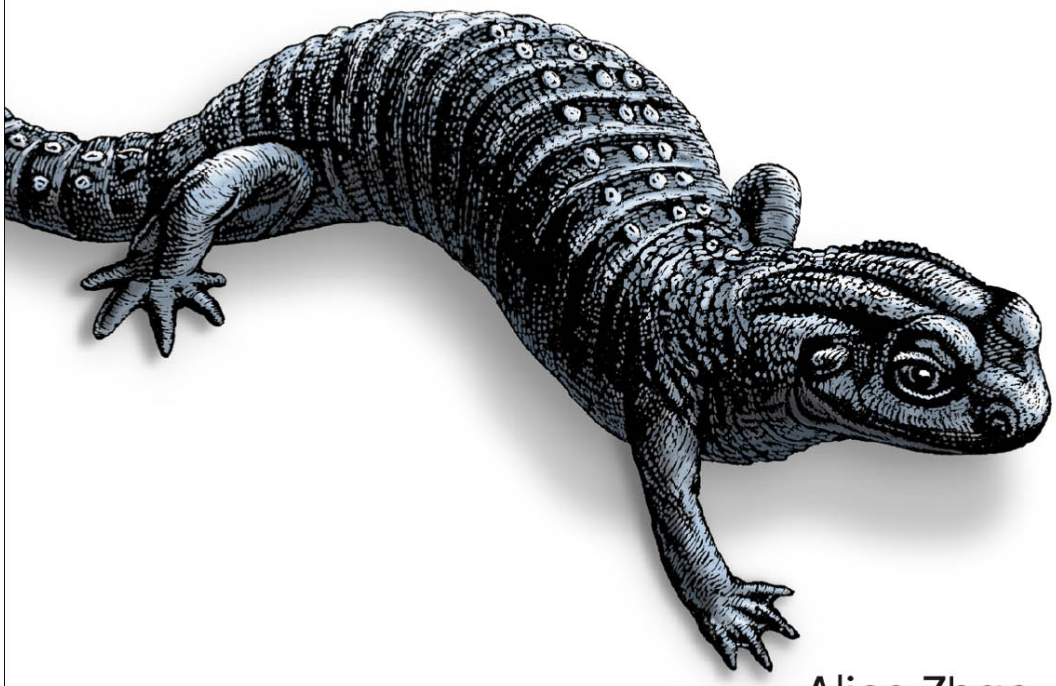
Erweiterte Neuauflage

Jetzt zu MySQL, Oracle, SQLite, PostgreSQL, SQL Server

SQL

kurz & gut

O'Reillys Taschenbibliothek



Alice Zhao

Übersetzung von Kathrin Lichtenberg

Inhalt

KAPITEL 4

SQL-Abfragen: Die Grundlagen

Eine SELECT-Anweisung, die üblicherweise aus sechs Klauseln besteht, bezeichnet man auch als Abfrage. Die einzelnen Klauseln werden jeweils in einem eigenen Abschnitt dieses Kapitels näher behandelt:

1. `SELECT`
2. `FROM`
3. `WHERE`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

Der letzte Abschnitt in diesem Kapitel befasst sich mit der LIMIT – Klausel, die von *MySQL, PostgreSQL* und *SQLite* unterstützt wird.

Die Codebeispiele in diesem Kapitel beziehen sich auf vier Tabellen:

`waterfall`

: Wasserfälle auf der Oberen Halbinsel von Michigan

`owner`

: Besitzer der Wasserfälle

`county`

: Counties, in denen sich die Wasserfälle befinden

`tour`

: Touren, die an mehreren Wasserfällen haltmachen

Hier ist eine Beispielabfrage, die unsere sechs wichtigsten Klauseln verwendet. Sie wird gefolgt von den Abfrageergebnissen, die man auch als *Ergebnismenge* bezeichnet.

```
— Touren mit zwei oder mehr öffentlichen Wasserfällen

SELECT   t.name AS tour_name,

         COUNT(*) AS num_waterfalls

FROM     tour t LEFT JOIN waterfall w

         ON t.stop = w.id

WHERE    w.open_to_public = 'y'

GROUP BY t.name

HAVING   COUNT(*) >= 2

ORDER BY tour_name;

tour_name  num_waterfalls

---- -----

M-28                    6

Munising                6

US-2                    4
```

Eine Datenbank *abzufragen*, bedeutet, Daten aus einer Datenbank, meist aus einer oder mehreren Tabellen, abzurufen oder zu beziehen.

Es ist auch möglich, einen *View* statt einer Tabelle abzufragen. Views sehen wie Tabellen aus und werden aus Tabellen abgeleitet, enthalten selbst aber keine Daten. Mehr zu Views finden Sie in »Views« auf Seite 136 in Kapitel 5.

Die SELECT-Klausel

Die SELECT-Klausel gibt die Spalten an, die eine Anweisung für Sie zurückliefern soll.

In der SELECT-Klausel folgt auf das Schlüsselwort SELECT eine Liste mit Spaltennamen und/oder Ausdrücken, die durch Kommata voneinander getrennt sind. Jeder Spaltenname und/oder Ausdruck wird dann zu einer Spalte in den Ergebnissen.

Spalten auswählen

Die einfachste SELECT-Klausel listet einen oder mehrere Spaltennamen aus den Tabellen in der FROM-Klausel auf:

```
SELECT id, name

FROM owner;

id    name

--- ------

    1 Pictured Rocks

    2 Michigan Nature

    3 AF LLC
```

```
4 MI DNR

5 Horseshoe Falls
```

Alle Spalten auswählen

Um alle Spalten aus einer Tabelle zurückzuliefern, können Sie einen einzelnen Asterisk verwenden, anstatt alle Spaltennamen anzugeben:

```
SELECT *

FROM owner;

id    name             phone         type

__ ______ _____ ___

    1 Pictured Rocks   906.387.2607  public

    2 Michigan Nature  517.655.5655  private

    3 AF LLC                         private

    4 MI DNR           906.228.6561  public

    5 Horseshoe Falls  906.387.2635  private
```

Der Asterisk ist eine hilfreiche Abkürzung, wenn Sie Abfragen testen, weil Sie sich damit eine Menge Tipparbeit ersparen. Allerdings ist es riskant, den Asterisk in Ihrem Produktionscode einzusetzen, weil sich die Spalten in einer Tabelle mit der Zeit ändern können, sodass Ihr Code fehlerhaft wird, wenn es weniger oder mehr Spalten gibt als erwartet.

Ausdrücke auswählen

Sie können nicht nur einfach Spalten auflisten, sondern auch komplexere Ausdrücke in der SELECT-Klausel angeben, um sich in den Ergebnissen bestimmte Spalten ausgeben zu lassen.

Die folgende Anweisung enthält einen Ausdruck zum Berechnen eines Bevölkerungsrückgangs von 10 %, gerundet auf null Dezimalstellen:

```
SELECT name, ROUND(population * 0.9, 0)

FROM county;

name        ROUND(population * 0.9, 0)

----- ----------

Alger                                 8876

Baraga                                7871

Ontonagon                             7036

…
```

Funktionen auswählen

Ausdrücke in der SELECT-Liste beziehen sich üblicherweise auf Spalten in den Tabellen, aus denen Sie die Daten beziehen, es gibt aber auch Ausnahmen. So ist

zum Beispiel eine gebräuchliche Funktion, die sich nicht auf irgendwelche Tabellen bezieht, die Funktion, die das aktuelle Datum zurückgibt:

```
SELECT CURRENT_DATE;

CURRENT_DATE

-----

2021-12-01
```

Der gezeigte Code funktioniert in *MySQL, PostgreSQL* und *SQLite*. Äquivalenten Code für andere RDBMS gibt es in »Datum/Zeit-Funktionen« auf Seite 213 in Kapitel 7.

Die meisten Abfragen enthalten sowohl eine SELECT-als auch eine FROM-Klausel, doch für bestimmte Datenbankfunktionen wie etwa CURRENT_DATE ist nur die SELECT-Klausel erforderlich.

Es ist auch möglich, Ausdrücke in die SELECT-Klausel einzusetzen, die *Unterabfragen* sind (das sind Abfragen, die in andere Abfragen geschachtelt sind). Mehr dazu finden Sie in »Unterabfragen auswählen« auf Seite 72.

Aliasse für Spalten

Der Zweck eines *Spaltenalias* besteht darin, einer Spalte oder einem Ausdruck, die bzw. der in der SELECT-Klausel aufgeführt ist, einen temporären Namen zu geben. Dieser temporäre Name oder Spaltenalias wird dann in den Ergebnissen als Spaltenname angezeigt.

Beachten Sie, dass diese Namensänderung nicht von Bestand ist, da die Spaltennamen in den Originaltabellen unverändert bleiben. Der Alias existiert nur in der Abfrage.

Dieser Code zeigt drei Spalten an.

```
SELECT id, name,
       ROUND(population * 0.9, 0)
FROM county;

id    name       ROUND(population * 0.9, 0)
--- ---- ---------
    2 Alger                            8876
    6 Baraga                           7871
    7 Ontonagon                        7036
…
```

Nehmen wir einmal an, wir wollten die Spaltennamen in den Ergebnissen umbenennen. `id` ist zu uneindeutig, und wir würden der Spalte gern einen aussagekräftigeren Namen geben. `ROUND(population * 0.9, 0)` ist zu lang und der Name soll einfacher sein.

Um einen Spaltenalias zu erzeugen, setzen Sie hinter einen Spaltennamen oder Ausdruck entweder einen Aliasnamen oder das Schlüsselwort `AS` und einen Aliasnamen.

```
— alias_name
SELECT id county_id, name,
       ROUND(population * 0.9, 0) estimated_pop
FROM county;
```

oder:

```
— AS alias_name

SELECT id AS county_id, name,

       ROUND(population * 0.90, 0) AS estimated_pop

FROM county;

county_id  name       estimated_pop

---- ---- -----

         2 Alger                8876

         6 Baraga               7871

         7 Ontonagon            7036

…
```

Beide Möglichkeiten werden in der Praxis eingesetzt, um Aliasse zu erzeugen. In der SELECT-Klausel ist die zweite Option beliebter, weil das Schlüsselwort AS es visuell leichter macht, Spaltennamen und Aliasse in einer langen Liste aus Spaltennamen zu unterscheiden.

Ältere Versionen von *PostgreSQL* verlangen die Verwendung von AS, wenn ein Spaltenalias erzeugt werden soll.

Auch wenn Spaltenaliasse nicht notwendig sind, werden sie beim Arbeiten mit Ausdrücken doch dringend empfohlen, um den Spalten in den Ergebnissen vernünftige Namen zu geben.

Aliasse mit Groß- und Kleinschreibung sowie Interpunktion

Wie die Spaltenaliasse `county_id` und `estimated_pop` zeigen, besteht die Konvention, beim Benennen von Spaltenaliassen Kleinbuchstaben zu verwenden sowie für Leerzeichen Unterstriche zu setzen.

Sie können auch Aliasse erzeugen, die Großbuchstaben, Leerzeichen und Interpunktionszeichen enthalten. Verwenden Sie hierzu doppelte Anführungszeichen, wie dieses Beispiel demonstriert:

```
SELECT id AS "Waterfall #",
  name AS "Waterfall Name"
FROM waterfall;

Waterfall #  Waterfall Name
---- -----
           1 Munising Falls
           2 Tannery Falls
           3 Alger Falls
```

…

Spalten qualifizieren

Nehmen wir einmal an, Sie schrieben eine Abfrage, die Daten aus zwei Tabellen bezieht, die beide eine Spalte namens `name` enthalten. Falls Sie nur `name` in der `SELECT`-Klausel angeben, weiß der Code nicht, welche Tabelle Sie meinen.

Um dieses Problem zu lösen, können Sie einen Spaltennamen durch seinen Tabellennamen *qualifizieren*. Mit anderen Worten, Sie geben einer Spalte mithilfe der *Punktnotation* ein Präfix, um näher zu bestimmen, zu welcher Tabelle sie gehört, wie in `table_name.column_name`.

Im folgenden Beispiel wird eine einzelne Tabelle abgefragt. Es ist also eigentlich nicht nötig, die Spalten hier zu qualifizieren, es dient lediglich der Demonstration. Und so würden Sie eine Spalte durch ihren Tabellennamen qualifizieren:

```
SELECT owner.id, owner.name

FROM owner;
```

Falls Sie in SQL einen Fehler erhalten, wenn Sie einen *mehrdeutigen Spaltennamen* referenzieren, bedeutet dies, dass mehrere Tabellen in Ihrer Abfrage eine Spalte desselben Namens enthalten und Sie nicht angegeben haben, auf welche Tabelle/Spalte Sie sich beziehen. Sie können den Fehler beheben, indem Sie den Spaltennamen qualifizieren.

Tabellen qualifizieren

Wenn Sie einen Spaltennamen anhand seines Tabellennamens qualifizieren, können Sie auch diese Tabelle anhand ihres Datenbank- oder Schema namens qualifizieren. Die folgende Abfrage bezieht Daten speziell aus der `owner`-Tabelle im `sqlbook`-Schema:

```
SELECT sqlbook.owner.id, sqlbook.owner.name
```

```
FROM sqlbook.owner;
```

Der gezeigte Code ist recht lang, da `sqlbook.owner` mehrmals wiederholt wird. Um sich das Tippen zu ersparen, können Sie einen *Tabellenalias* angeben. Das folgende Beispiel gibt der Tabelle `owner` den Alias `o`:

```
SELECT o.id, o.name

FROM sqlbook.owner o;
```

oder:

```
SELECT o.id, o.name

FROM owner o;
```

Spaltenaliasse versus Tabellenaliasse

Spaltenaliasse werden innerhalb der `SELECT`-Klausel definiert, um eine Spalte in den Ergebnissen umzubenennen. Es ist üblich, `AS` zu verwenden, wird aber nicht verlangt.

```
– Spaltenalias

SELECT num AS new_col

FROM my_table;
```

Tabellenaliasse werden innerhalb der `FROM`-Klausel definiert, um einen temporären Namen für eine Tabelle zu erzeugen. Es ist üblich, `AS` wegzulassen, obwohl es auch mit `AS` funktioniert.

```
– Tabellenalias

SELECT *
```

```
FROM my_table mt;
```

Unterabfragen auswählen

Eine *Unterabfrage* ist eine Abfrage, die in eine andere Abfrage geschachtelt ist. Unterabfragen können in verschiedenen Klauseln zu finden sein, einschließlich der SELECT-Klausel.

Im folgenden Beispiel wollen wir zusätzlich zu `id`, `name` und `population` auch die durchschnittliche Bevölkerungszahl aller Counties sehen. Indem wir eine Unterabfrage einfügen, erzeugen wir eine neue Spalte in den Ergebnissen für die durchschnittliche Bevölkerungszahl.

```
SELECT id, name, population,
       (SELECT AVG(population) FROM county)
       AS average_pop
FROM county;

id    name       population  average_pop
--- ---- ---- ----
    2 Alger             9862        18298
    6 Baraga            8746        18298
    7 Ontonagon         7818        18298
…
```

Ein paar Anmerkungen:

- Eine Unterabfrage muss in runde Klammern eingeschlossen sein.
- Wenn man eine Unterabfrage in der SELECT-Klausel schreibt, wird dringend empfohlen, einen Spaltenalias anzugeben, was mit `average_pop` hier geschehen ist. Auf diese Weise trägt die Spalte in den Ergebnissen einen einfachen Namen.
- Es gibt nur einen Wert in der Spalte `average_pop`, der in allen Zeilen wiederholt wird. Wenn man eine Unterabfrage in die SELECT-Klausel einfügt, muss das Ergebnis der Unterabfrage eine einzelne Spalte und entweder keine oder eine Zeile zurückgeben, wie in der folgenden Unterabfrage demonstriert wird, die die durchschnittliche Bevölkerungszahl berechnet.

```
SELECT AVG(population) FROM county;

AVG(population)

------

          18298
```

- Falls die Unterabfrage keine Zeilen zurückliefert, wird die neue Spalte mit NULL-Werten gefüllt.

Nichtkorrelierte versus korrelierte Unterabfragen

Das gezeigte Beispiel ist eine *nichtkorrelierte Unterabfrage*, was bedeutet, dass die Unterabfrage sich nicht auf die äußere Abfrage bezieht. Die Unterabfrage kann für sich allein, also unabhängig von der äußeren Abfrage, ausgeführt werden.

Die andere Art von Unterabfrage wird *korrelierte Unterabfrage* genannt und ist eine, die sich auf die Werte in der äußeren Abfrage bezieht. Sie verlangsamt oft signifikant die Verarbeitungszeit, sodass es am besten ist, die Abfrage umzuschreiben und stattdessen einen `JOIN` zu verwenden. Es folgt ein Beispiel einer korrelierten Unterabfrage mit effizienterem Code.

Leistungsprobleme mit korrelierten Unterabfragen

Die folgende Abfrage gibt die Anzahl der Wasserfälle der einzelnen Besitzer aus. Beachten Sie, dass sich der `o.id = w.owner_id`-Schritt in der Unterabfrage auf die `owner`-Tabelle in der äußeren Abfrage bezieht, wodurch dies zu einer korrelierten Unterabfrage wird.

```
SELECT o.id, o.name,
       (SELECT COUNT(*) FROM waterfall w
       WHERE o.id = w.owner_id) AS num_waterfalls
FROM owner o;

id    name             num_waterfalls
--- ------ -----
    1 Pictured Rocks                 3
    2 Michigan Nature                3
    3 AF LLC                         1
    4 MI DNR                         1
    5 Horseshoe Falls                0
```

Eine bessere Vorgehensweise wäre es, die Abfrage mit einem Join umzuschreiben. Auf diese Weise werden die Tabellen zuerst zusammengefasst, bevor der Rest der Abfrage ausgeführt wird. Das geht viel schneller, als für jede Datenzeile immer wieder eine Unterabfrage auszuführen. Mehr zu Joins finden Sie in »Tabellen mit Joins zusammenbringen« auf Seite 262 in Kapitel 9.

```
SELECT   o.id, o.name,
         COUNT(w.id) AS num_waterfalls
FROM     owner o LEFT JOIN waterfalls w
```

```
          ON o.id = w.owner_id

GROUP BY o.id, o.name

id    name              num_waterfalls

--- ------ -----

    1 Pictured Rocks                3

    2 Michigan Nature               3

    3 AF LLC                        1

    4 MI DNR                        1

    5 Horseshoe Falls               0
```

DISTINCT

Wenn eine Spalte in der `SELECT`-Klausel aufgeführt ist, werden standardmäßig alle Zeilen zurückgeliefert. Um expliziter zu sein, können Sie das Schlüsselwort `ALL` einfügen, aber das ist wirklich optional. Die folgenden Abfragen liefern jede `type`/`open_to_public`-Kombination zurück.

```
SELECT o.type, w.open_to_public

FROM owner o

JOIN waterfall w ON o.id = w.owner_id;
```

oder:

```
SELECT ALL o.type, w.open_to_public

FROM owner o

JOIN waterfall w ON o.id = w.owner_id;

type     open_to_public

---- -----

public   y

public   y

public   y

private  y

private  y

private  y

private  y

public   y
```

Falls Sie doppelt auftretende Zeilen aus den Ergebnissen entfernen wollen, können Sie das Schlüsselwort `DISTINCT` benutzen. Die folgende Abfrage liefert eine Liste der eindeutigen `type`/`open_to_public`-Kombinationen zurück.

```
SELECT DISTINCT o.type, w.open_to_public

FROM owner o
```

```
JOIN waterfall w ON o.id = w.owner_id;

type     open_to_public

---- -----

public   y

private  y
```

COUNT und DISTINCT

Um die Anzahl der eindeutigen Werte in einer *einzigen Spalte* zu zählen, können Sie die Schlüsselwörter COUNT und DISTINCT innerhalb der SELECT-Klausel kombinieren. Die folgende Abfrage liefert die Anzahl der eindeutigen `type`-Werte zurück.

```
SELECT COUNT(DISTINCT type) AS unique

FROM owner;

unique

---

      2
```

Um die Anzahl der eindeutigen Kombinationen aus *mehreren Spalten* zu zählen, können Sie eine DISTINCT-Abfrage als Unterabfrage verpacken und dann einen COUNT auf der Unterabfrage ausführen. Die folgende Abfrage liefert die Anzahl der eindeutigen `type`/`open_to_public`-Kombinationen zurück.

```
SELECT COUNT(*) AS num_unique

FROM (SELECT DISTINCT o.type, w.open_to_public
```

```
        FROM owner o JOIN waterfall w

        ON o.id = w.owner_id) my_subquery;

 num_unique

 ----

          2
```

MySQL und *PostgreSQL* unterstützen die Verwendung der COUNT(DISTINCT)-Syntax auf mehreren Spalten. Die folgenden zwei Abfragen sind äquivalent zur gezeigten Abfrage, brauchen aber keine Unterabfrage:

```
 — MySQL-Äquivalent

 SELECT COUNT(DISTINCT o.type, w.open_to_public)

        AS num_unique

        FROM owner o JOIN waterfall w

             ON o.id = w.owner_id;

 — PostgreSQL-Äquivalent

 SELECT COUNT(DISTINCT (o.type, w.open_to_public))

        AS num_unique

        FROM owner o JOIN waterfall w

             ON o.id = w.owner_id;
```

```
num_unique

____

           2
```

Die FROM-Klausel

Die FROM-Klausel wird verwendet, um die Quelle der Daten festzulegen, die Sie beziehen wollen. Der einfachste Fall ist das Benennen einer einzelnen Tabelle oder eines einzelnen View in der FROM-Klausel der Abfrage.

```
SELECT name

FROM waterfall;
```

Sie können eine Tabelle oder einen View mittels der Punktnotation entweder mit einem Datenbank- oder einem Schema namen qualifizieren. Die folgende Abfrage bezieht Daten speziell aus der `waterfall`-Tabelle im `sqlbook`-Schema:

```
SELECT name

FROM sqlbook.waterfall;
```

Aus mehreren Tabellen

Anstatt uns nur auf eine Tabelle zu beschränken, wollen Sie oft Daten aus mehreren Tabellen holen. Die gebräuchlichste Methode hierfür ist die Verwendung einer `JOIN`-Klausel innerhalb der `FROM`−Klausel. Die folgende Abfrage bezieht Daten aus den Tabellen `waterfall` und `tour` und zeigt sie in einer einzigen Ergebnistabelle an.

```
SELECT *

FROM waterfall w JOIN tour t

     ON w.id = t.stop;

id    name             … name      stop  …

-- -----     --- --

    1 Munising Falls      M-28          1

    1 Munising Falls      Munising      1

    2 Tannery Falls       Munising      2

    3 Alger Falls         M-28          3

    3 Alger Falls         Munising      3

…
```

Schauen wir uns die einzelnen Teile des Codeblocks genauer an.

Tabellenaliasse

```
waterfall w JOIN tour t
```

Den Tabellen `waterfall` und `tour` werden die Tabellenaliasse `w` und `t` gegeben. Dies sind temporäre Namen für die Tabellen in der Abfrage. Tabellenaliasse sind in einer `JOIN`-Klausel nicht erforderlich, helfen aber dabei, Tabellennamen abzukürzen, die in den `ON`- und `SELECT`-Klauseln referenziert werden müssen.

JOIN … ON …

```
waterfall w JOIN tour t

ON w.id = t.stop
```

Diese zwei Tabellen werden mit dem Schlüsselwort `JOIN` zusammengezogen. Einer `JOIN`-Klausel folgt immer eine `ON`-Klausel, die festlegt, wie die Tabellen miteinander verknüpft werden sollen. In diesem Fall muss die `id` des Wasserfalls in der Tabelle `waterfall` dem `stop` bei dem Wasserfall in der `tour`-Tabelle entsprechen.

Es kann sein, dass die `FROM`-, `JOIN`- und `ON`-Klauseln auf unterschiedlichen Zeilen stehen oder eingerückt sind. Das ist nicht erforderlich, verbessert aber die Lesbarkeit – vor allem wenn Sie viele Tabellen miteinander verbinden.

Ergebnistabelle

Das Ergebnis einer Abfrage kommt immer in einer einzigen Tabelle. Die Tabelle `waterfall` hat zwölf Spalten, und die Tabelle `tour` hat drei Spalten. Nachdem diese Tabellen zusammengefasst wurden, hat die Ergebnistabelle 15 Spalten.

```
id    name             … name      stop …

-- -----     --- --

    1 Munising Falls      M-28          1

    1 Munising Falls      Munising      1

    2 Tannery Falls       Munising      2
```

```
    3 Alger Falls          M-28           3

    3 Alger Falls          Munising       3

…
```

Sie sehen vermutlich, dass es in der Ergebnistabelle zwei Spalten mit dem Namen `name` gibt. Die erste stammt aus der `waterfall`-Tabelle, die zweite aus der `tour`-Tabelle. Um sie in der `SELECT`-Klausel referenzieren zu können, müssen Sie die Spaltennamen qualifizieren.

```
SELECT w.name, t.name

FROM waterfall w JOIN tour t

     ON w.id = t.stop;

name             name

-----  ---

Munising Falls   M-28

Munising Falls   Munising

Tannery Falls    Munising

…
```

Damit Sie die beiden Spalten unterscheiden können, sollten Sie für die Spaltennamen auch noch Aliasse vergeben.

```
SELECT w.name AS waterfall_name,
```

```
        t.name AS tour_name

FROM waterfall w JOIN tour t

     ON w.id = t.stop;

waterfall_name  tour_name

_____ ____

Munising Falls  M-28

Munising Falls  Munising

Tannery Falls   Munising

Alger Falls     M-28

Alger Falls     Munising

…
```

JOIN-Variationen

Falls in dem vorhergehenden Beispiel ein Wasserfall nicht in einer der Touren aufgeführt ist, erscheint er auch nicht in der Ergebnistabelle. Um alle Wasserfälle in den Ergebnissen zu erhalten, müssen Sie eine andere Art von Join verwenden.

JOIN ist standardmäßig ein INNER JOIN

Dieses Beispiel verwendet das einfache Schlüsselwort `JOIN`, um die Daten aus den zwei Tabellen zusammenzuziehen. Besser ist es jedoch, ausdrücklich die Art von Join anzugeben, die Sie benutzen. `JOIN` allein bedeutet standardmäßig immer ein `INNER JOIN`. Das bedeutet, dass nur Einträge, die in beiden Tabellen stehen, in den Ergebnissen zurückgeliefert werden.

In SQL kommt eine Vielzahl an Join-Typen zum Einsatz. Diese werden in »Tabellen mit Joins zusammenbringen« auf Seite 262 in Kapitel 9 ausführlicher

behandelt.

Aus Unterabfragen

Eine Unterabfrage ist eine Abfrage, die in eine andere Abfrage geschachtelt ist. Unterabfragen innerhalb der FROM-Klausel sollten unabhängige SELECT-Anweisungen sein. Das bedeutet, dass sie sich überhaupt nicht auf die äußere Abfrage beziehen und allein ausgeführt werden können.

Eine Unterabfrage innerhalb der FROM-Klausel ist auch als *abgeleitete Tabelle* bekannt, weil die Unterabfrage im Prinzip für die Dauer der Abfrage wie eine Tabelle agiert.

Die folgende Abfrage listet alle Wasserfälle auf, die sich in öffentlicher Hand befinden. Der Unterabfrageteil ist fett gedruckt.

```
SELECT w.name AS waterfall_name,

       o.name AS owner_name

FROM (SELECT * FROM owner WHERE type = 'public') o

     JOIN waterfall w

     ON o.id = w.owner_id;

waterfall_name  owner_name

----- -----

Little Miners   Pictured Rocks

Miners Falls    Pictured Rocks
```

```
Munising Falls  Pictured Rocks

Wagner Falls    MI DNR
```

Wichtig ist es, die Reihenfolge zu verstehen, in der die Abfrage ausgeführt wird.

Schritt 1: Ausführen der Unterabfrage

Der Inhalt der Unterabfrage wird zuerst ausgeführt. Sie können sehen, dass dies eine Tabelle ergibt, in der nur die staatlichen Besitzer stehen:

```
SELECT * FROM owner WHERE type = 'public';

id    name            phone         type

--- ----- ----- ---

    1 Pictured Rocks  906.387.2607  public

    4 MI DNR          906.228.6561  public
```

Wenn Sie sich noch einmal die Originalabfrage anschauen, bemerken Sie, dass auf die Unterabfrage sofort der Buchstabe o folgt. Dies ist der temporäre Name oder Alias, den wir den Ergebnissen der Unterabfrage zuweisen.

Aliasse sind für Unterabfragen innerhalb der FROM-Klausel in *MySQL, PostgreSQL* und *SQL Server* notwendig, nicht jedoch in Oracle und SQLite.

Schritt 2: Ausführen der gesamten Abfrage

Als Nächstes können Sie sich das so vorstellen, dass der Buchstabe o den Platz der Unterabfrage einnimmt. Die Abfrage wird nun wie üblich ausgeführt.

```
SELECT w.name AS waterfall_name,

       o.name AS owner_name

FROM o JOIN waterfall w

     ON o.id = w.owner_id;

waterfall_name  owner_name

_____ _____

Little Miners   Pictured Rocks

Miners Falls    Pictured Rocks

Munising Falls  Pictured Rocks

Wagner Falls    MI DNR
```

Unterabfragen versus WITH-Klausel

Eine Alternative zum Schreiben einer Unterabfrage ist es, stattdessen einen allgemeinen Tabellenausdruck (*Common Table Expression*, CTE) mittels einer `WITH`-Klausel zu schreiben. Der Vorteil der `WITH`-Klausel besteht darin, dass die Unterabfrage gleich zu Anfang genannt wird, was saubereren Code ergibt und außerdem die Möglichkeit bietet, die Unterabfrage mehrmals zu referenzieren.

```
WITH o AS (SELECT * FROM owner

           WHERE type = 'public')

SELECT w.name AS waterfall_name,

       o.name AS owner_name

FROM o JOIN waterfall w

     ON o.id = w.owner_id;
```

Die `WITH`-Klausel wird von *MySQL 8.0+* (ab 2018), *PostgreSQL, Oracle, SQL Server* und *SQLite* unterstützt. »Common Table Expressions« auf Seite 282 in Kapitel 9 bietet weitere Beispiele für diese Technik.

Wieso sollte man eine Unterabfrage in der FROM-Klausel benutzen?

Der größte Vorteil der Verwendung von Unterabfragen besteht darin, dass Sie ein großes Problem in kleinere Probleme verwandeln können. Hier sind zwei Beispiele:

Beispiel 1: Mehrere Schritte, um Ergebnisse zu erhalten

Nehmen wir einmal an, Sie wollten die durchschnittliche Anzahl der Halts einer Tour ermitteln. Zuerst müssten Sie feststellen, wie oft die einzelnen Touren jeweils einen Halt einlegen. Anschließend berechnen Sie den Durchschnitt der Ergebnisse.

Die folgende Abfrage ermittelt die Anzahl der Halts der einzelnen Touren:

```
SELECT name, MAX(stop) as num_stops
```

```
FROM tour

GROUP BY name;

name      num_stops

--- ----

M-28              11

Munising           6

US-2              14
```

Sie könnten dann die Abfrage in eine Unterabfrage umwandeln und eine weitere Abfrage um diese herum schreiben, um den Durchschnitt zu ermitteln:

```
SELECT AVG(num_stops) FROM

(SELECT name, MAX(stop) as num_stops

FROM tour

GROUP BY name) tour_stops;

AVG(num_stops)

------

10.3333333333333
```

Beispiel 2: Tabelle in FROM-Klausel ist zu groß

Das ursprüngliche Ziel bestand darin, alle Wasserfälle aufzulisten, die in öffentlicher Hand sind. Das lässt sich tatsächlich ohne Unterabfrage und stattdessen mit einem Join erledigen:

```
SELECT w.name AS waterfall_name,

       o.name AS owner_name

FROM   owner o

       JOIN waterfall w ON o.id = w.owner_id

WHERE  o.type = 'public';

waterfall_name  owner_name

_____ _____

Little Miners   Pictured Rocks

Miners Falls    Pictured Rocks

Munising Falls  Pictured Rocks

Wagner Falls    MI DNR
```

Nehmen wir einmal an, dass die Abfrage sehr lange für ihre Verarbeitung braucht. Das kann passieren, wenn Sie riesige Tabellen zusammenführen (wir sprechen hier von Tabellen mit einer Zeilenanzahl im zweistelligen Millionenbereich). Es gibt mehrere Möglichkeiten, die Abfrage so umzuschreiben, dass sie schneller läuft. Eine dieser Möglichkeiten ist eine Unterabfrage.

Da wir nur an staatlichen Eigentümern interessiert sind, können wir zuerst eine Unterabfrage schreiben, die alle Privatbesitzer herausfiltert. Die kleinere

owner-Tabelle würde dann mit der waterfall-Tabelle zusammengeführt werden, was weniger Zeit beansprucht und dieselben Ergebnisse liefert.

```
SELECT w.name AS waterfall_name,

       o.name AS owner_name

FROM   (SELECT * FROM owner

       WHERE type = 'public') o

       JOIN waterfall w ON o.id = w.owner_id;

waterfall_name  owner_name

_____ _____

Little Miners   Pictured Rocks

Miners Falls    Pictured Rocks

Munising Falls  Pictured Rocks

Wagner Falls    MI DNR
```

Das sind nur zwei Beispiele dafür, wie man eine größere Abfrage mithilfe von Unterabfragen in kleinere Schritte zerlegen kann.

Die WHERE-Klausel

Die WHERE-Klausel dient dazu, die Abfrageergebnisse auf die Zeilen zu beschränken, die uns wirklich interessieren. Oder einfacher gesagt: Dies ist die Stelle, an der wir Daten filtern können. Sie wollen nur ganz selten alle Zeilen einer Tabelle anzeigen. Meist reichen Ihnen die Zeilen, die bestimmte Kriterien erfüllen.

Wenn Sie eine Tabelle mit Millionen von Zeilen untersuchen, sollten Sie niemals ein `SELECT * FROM my_table;` verwenden, weil das unnötig lange für die Ausführung braucht.

Stattdessen ist es eine gute Idee, die Daten vorher zu filtern. Zwei Methoden sind hier gebräuchlich:

Filtern anhand einer Spalte in der `WHERE`*-Klausel*

Besser noch, filtern Sie anhand einer Spalte, die bereits indiziert wurde, um die Datenabfrage noch mehr zu beschleunigen.

```
SELECT *

FROM my_table

WHERE year_id = 2021;
```

Anzeigen der obersten Zeilen der Daten mit der LIMIT-Klausel (oder `WHERE ROWNUM <= 10` *in Oracle oder* `SELECT TOP 10 *` *in SQL Server)*

```
SELECT *

FROM my_table

LIMIT 10;
```

Die folgende Abfrage sucht alle Wasserfälle, die nicht das Wort *Falls* im Namen haben. Mehr zum Schlüsselwort `LIKE` finden Sie in Kapitel 7.

```
SELECT id, name

FROM waterfall

WHERE name NOT LIKE '%Falls%';

id    name

--- ------
```

```
 7 Little Miners

14 Rapid River Fls
```

Der fett dargestellte Abschnitt wird oft als bedingte Anweisung oder Prädikat bezeichnet. Das Prädikat nimmt einen logischen Vergleich für jede Datenzeile vor, der TRUE/FALSE/UNKNOWN ergibt.

Die `waterfall`-Tabelle enthält 16 Zeilen. Für jede Zeile wird geprüft, ob der Wasserfallname *Falls* enthält. Ist *Falls* nicht enthalten, ist das `name NOT LIKE '%Falls%'`-Prädikat TRUE, und die Zeile wird in den Ergebnissen zurückgeliefert. Das war in den zwei gezeigten Zeilen der Fall.

Mehrere Prädikate

Es ist möglich, mehrere Prädikate mit *Operatoren* wie AND oder OR zu kombinieren. Das folgende Beispiel zeigt Wasserfälle ohne *Falls* in ihren Namen, die außerdem keinen Besitzer haben:

```
SELECT id, name

FROM waterfall

WHERE name NOT LIKE '%Falls%'

      AND owner_id IS NULL;

id    name

--- ------

   14 Rapid River Fls
```

Näheres zu Operatoren finden Sie in Operatoren in Kapitel 7.

Auf Unterabfragen filtern

Eine Unterabfrage ist eine Abfrage, die in eine andere Abfrage geschachtelt ist. Häufig findet man sie im Zusammenhang mit der WHERE-Klausel. Das folgende Beispiel ruft öffentlich zugängliche Wasserfälle ab, die sich im County Alger befinden:

```
SELECT w.name

FROM   waterfall w

WHERE  w.open_to_public = 'y'

       AND w.county_id IN (

           SELECT c.id FROM county c

           WHERE c.name = 'Alger');
```

```
name

-----

Munising Falls

Tannery Falls

Alger Falls

…
```

Anders als Unterabfragen in der SELECT-Klausel oder der FROM-Klausel erfordern Unterabfragen in der WHERE-Klausel keinen Alias. Tatsächlich erhalten Sie eine Fehlermeldung, falls Sie einen Alias einsetzen.

Warum sollte man eine Unterabfrage in der WHERE-Klausel benutzen?

Das ursprüngliche Ziel war es, die öffentlich zugänglichen Wasserfälle abzufragen, die im County Alger liegen. Würden Sie diese Abfrage von Grund auf neu schreiben, würden Sie vermutlich folgendermaßen starten:

```
SELECT w.name

FROM   waterfall w

WHERE  w.open_to_public = 'y';
```

An dieser Stelle haben Sie alle Wasserfälle, die öffentlich zugänglich sind. Nun müssen Sie versuchen, diejenigen zu finden, die speziell im County Alger liegen. Sie wissen, dass die `waterfall`-Tabelle keine Spalte mit County-Namen hat, die `county`-Tabelle dagegen schon.

Sie haben zwei Möglichkeiten, um den County-Namen in die Ergebnisse zu bekommen. Entweder Sie schreiben eine Unterabfrage in der WHERE-Klausel, die speziell die Alger-County-Informationen abruft, oder Sie fassen die Tabellen `waterfall` und `county` zusammen:

```
— Unterabfrage in WHERE-Klausel

SELECT w.name

FROM   waterfall w

WHERE  w.open_to_public = 'y'
```

```
        AND w.county_id IN (
            SELECT c.id FROM county c
            WHERE c.name = 'Alger');
```

oder:

```
– JOIN-Klausel
SELECT w.name
FROM   waterfall w INNER JOIN county c
       ON w.county_id = c.id
WHERE  w.open_to_public = 'y'
       AND c.name = 'Alger';
name
-----
Munising Falls
Tannery Falls
Alger Falls
…
```

Die zwei Abfragen liefern die gleichen Ergebnisse. Der Vorteil der ersten Herangehensweise besteht darin, dass Unterabfragen oft leichter zu verstehen sind als Joins. Die zweite Vorgehensweise hat dagegen den Vorteil, dass Joins üblicherweise schneller ausgeführt werden als Unterabfragen.

Funktionierend > Optimieren

Beim Schreiben von SQL-Code gibt es oft mehrere Möglichkeiten, das Gleiche zu erreichen.

Ihre oberste Priorität sollte es immer sein, *funktionierenden* Code zu schreiben. Es ist egal, wenn er sehr lange für die Ausführung braucht oder hässlich ist … er funktioniert!

Falls Sie Zeit haben, besteht der nächste Schritt darin, den Code zu *optimieren*. Sie können seine Leistung verbessern, indem Sie ihn etwa mit einem Join umschreiben, oder Sie machen ihn mit Einrückungen und Großschreibungen lesbarer usw.

Setzen Sie sich nicht unter Druck, von Anfang an den absolut optimalen Code zu schreiben, sondern bemühen Sie sich zunächst um Code, der funktioniert. Eleganter Code kommt dann mit zunehmender Erfahrung.

Andere Möglichkeiten, Daten zu filtern

Die `WHERE`-Klausel ist nicht die einzige Stelle in einer `SELECT`-Anweisung zum Filtern von Datenzeilen.

- `FROM`-Klausel: Wenn man Tabellen miteinander verbindet, gibt die `ON`-Klausel an, wie sie verknüpft werden sollen. Hier können Sie Bedingungen festlegen, mit denen Sie die Datenzeilen einschränken, die durch die Abfrage zurückgeliefert werden (siehe »Tabellen mit Joins zusammenbringen« auf Seite 262 in Kapitel 9 für nähere Informationen).
- `HAVING`-Klausel: Wenn es Aggregationen in der `SELECT`-Anweisung gibt, dann geben Sie in der `HAVING`-Klausel an, wie die Aggregationen gefiltert werden sollen (siehe »Die HAVING-Klausel« auf Seite 92 für weitere Einzelheiten).
- `LIMIT`-Klausel: Um eine bestimmte Anzahl von Zeilen anzuzeigen, können Sie die `LIMIT`-Klausel verwenden. In *Oracle* wird dies mit `WHERE ROWNUM` erledigt und in *SQL Server* mit `SELECT TOP` (siehe »Die LIMIT-Klausel« auf Seite 97 in diesem Kapitel für weitere Einzelheiten).

Die GROUP BY-Klausel

Zweck der `GROUP BY`-Klausel ist es, Zeilen in Gruppen zu sammeln und die Zeilen innerhalb der Gruppen irgendwie zusammenzufassen, sodass am Ende

nur eine Zeile pro Gruppe zurückgegeben wird. Man bezeichnet das manchmal als »Slicing« der Zeilen und als »Rollup« der Zeilen in den einzelnen Gruppen.

Die folgende Abfrage ermittelt die Anzahl der Wasserfälle in den einzelnen Touren:

```
SELECT   t.name AS tour_name,

         COUNT(*) AS num_waterfalls

FROM     waterfall w INNER JOIN tour t

         ON w.id = t.stop

GROUP BY t.name;

tour_name  num_waterfalls

---- -----

M-28                    6

Munising                6

US-2                    4
```

Es gibt hier zwei Teile, die für uns hier interessant sind:

- *Das Sammeln der Zeilen*, das in der GROUP BY-Klausel erfolgt.
- *Das Zusammenfassen der Zeilen* innerhalb der Gruppen, das in der SELECT-Klausel festgelegt wird.

Schritt 1: Das Sammeln der Zeilen

In der GROUP BY-Klausel:

```
GROUP BY t.name
```

geben wir an, dass wir gern alle Datenzeilen anschauen und die Wasserfälle der M-28-Tour in eine Gruppe setzen würden, alle Wasserfälle der Munising-Tour in eine Gruppe und so weiter. Hinter den Kulissen werden die Daten folgendermaßen gruppiert:

```
tour_name  waterfall_name

---- ------

M-28       Munising Falls

M-28       Alger Falls

M-28       Scott Falls

M-28       Canyon Falls

M-28       Agate Falls

M-28       Bond Falls

Munising   Munising Falls

Munising   Tannery Falls

Munising   Alger Falls

Munising   Wagner Falls

Munising   Horseshoe Falls

Munising   Miners Falls
```

```
US-2        Bond Falls

US-2        Fumee Falls

US-2        Kakabika Falls

US-2        Rapid River Fls
```

Schritt 2: Das Zusammenfassen der Zeilen

In der SELECT-Klausel:

```
SELECT t.name AS tour_name,

       COUNT(*) AS num_waterfalls
```

geben wir an, dass wir für jede Gruppe oder für jede Tour die Datenzeilen in der Gruppe zählen wollen. Da jede Zeile einen Wasserfall repräsentiert, würde dies die Gesamtanzahl der Wasserfälle in den jeweiligen Touren repräsentieren.

Die COUNT()-Funktion ist offiziell als *Aggregatfunktion* bekannt oder als eine Funktion, die viele Zeilen mit Daten in einem einzelnen Wert zusammenfasst. Weitere Aggregatfunktionen finden Sie in »Aggregatfunktionen« auf Seite 190 in Kapitel 7.

In diesem Beispiel liefert `COUNT(*)` die Anzahl der Wasserfälle entlang der einzelnen Touren zurück. Das funktioniert hier aber nur, weil jede Datenzeile in den Tabellen `waterfall` und `tour` einen einzelnen Wasserfall repräsentiert.

Wäre einer der Wasserfälle in mehreren Zeilen aufgeführt, würde `COUNT(*)` einen größeren Wert liefern als erwartet. In diesem Fall könnten Sie stattdessen möglicherweise `COUNT(DISTINCT waterfall_name)` verwenden, um die eindeutigen Wasserfälle zu finden. Näheres dazu erfahren Sie in `COUNT` und `DISTINCT`.

Die wichtigste Erkenntnis hier ist, dass Sie die Ergebnisse der Aggregatfunktion unbedingt noch einmal überprüfen müssen, um sicherzugehen, dass sie die Daten auch wirklich so zusammenfasst, wie Sie es vorgesehen haben.

Nachdem nun mit der `GROUP BY`-Klausel die Gruppen erzeugt wurden, wird die Aggregatfunktion einmal auf jede Gruppe angewendet:

```
tour_name  COUNT(*)

---- ---

M-28              6

M-28

M-28

M-28

M-28

M-28

M-28

Munising           6

Munising
```

```
Munising

Munising

Munising

Munising

US-2                4

US-2

US-2

US-2
```

Alle Spalten, auf die die Aggregatfunktion nicht angewendet wurde, in diesem Fall also die Spalte `tour_name`, werden in einem Wert zusammengefasst:

```
tour_name  COUNT(*)

---- ---

M-28              6

Munising          6

US-2              4
```

Dieses Zusammenfassen vieler Zeilen in einer Gesamtzeile bedeutet, dass die SELECT-Klausel beim Anwenden einer GROUP BY-Klausel *nur* dies enthalten sollte:

- Alle Spalten, die in der GROUP BY-Klausel aufgelistet sind: `t.name`
- Aggregationen: `COUNT(*)`

```
SELECT t.name AS tour_name,
       COUNT(*) AS num_waterfalls
…
GROUP BY t.name;
```

Geschieht das nicht, erhalten Sie entweder eine Fehlermeldung, oder es werden unkorrekte Werte zurückgeliefert.

GROUP BY in der Praxis

Folgende Schritte sollten Sie unternehmen, wenn Sie ein GROUP BY nutzen:

1. Stellen Sie fest, welche Spalte(n) Sie verwenden wollen, um Ihre Daten (z. B. den Namen der Tour) herauszulösen oder zu gruppieren.
2. Stellen Sie fest, wie Sie die Daten in den einzelnen Zeilen zusammenfassen wollen (z. B. Zählen der Wasserfälle in den einzelnen Touren).

Wenn Sie dies entschieden haben:

1. Listen Sie in der SELECT-Klausel die Spalte(n) auf, anhand deren Sie gruppieren wollen (z. B. den Namen der Tour), und die Aggregation(en), die Sie in den einzelnen Gruppen berechnen wollen (z. B. Anzahl der Wasserfälle).
2. Listen Sie in der GROUP BY-Klausel alle Spalten auf, die keine Aggregationen sind (z. B. den Namen der Tour).

Komplexere Gruppierungssituationen, einschließlich ROLLUP, CUBE und GROUPING SETS, finden Sie in »Gruppieren und Zusammenfassen« auf Seite 235 in Kapitel 8.

Die HAVING-Klausel

Die HAVING-Klausel setzt Restriktionen auf die Zeilen, die von einer GROUP BY-Abfrage zurückgeliefert werden. Mit anderen Worten, sie erlaubt Ihnen, die

Ergebnisse zu filtern, nachdem ein GROUP BY angewandt wurde.

Eine HAVING-Klausel folgt immer unmittelbar auf eine GROUP BY-Klausel. Ohne GROUP BY-Klausel kann es keine HAVING-Klausel geben.

Dies ist eine Abfrage, die die Anzahl der Wasserfälle auf jeder Tour mittels einer GROUP BY-Klausel auflistet:

```
SELECT   t.name AS tour_name,

         COUNT(*) AS num_waterfalls

FROM     waterfall w INNER JOIN tour t

         ON w.id = t.stop

GROUP BY t.name;

tour_name  num_waterfalls

---- -----

M-28                    6

Munising                6

US-2                    4
```

Nehmen wir einmal an, wir wollten nur die Touren auflisten, die exakt sechs Halts haben. Dazu würden Sie eine HAVING-Klausel hinter der GROUP BY-

Klausel einfügen:

```
SELECT   t.name AS tour_name,

         COUNT(*) AS num_waterfalls

FROM     waterfall w INNER JOIN tour t

         ON w.id = t.stop

GROUP BY t.name

HAVING   COUNT(*) = 6;

tour_name  num_waterfalls

---- -----

M-28                    6

Munising                6
```

WHERE versus HAVING

Aufgabe beider Klauseln ist das Filtern von Daten. Falls Sie versuchen:

- auf bestimmten Spalten zu filtern, schreiben Sie Ihre Bedingung in die `WHERE`-Klausel.
- auf Aggregationen zu filtern, schreiben Sie Ihre Bedingungen in die `HAVING`-Klausel.

Der Inhalt von `WHERE`- und `HAVING`-Klauseln kann nicht vertauscht werden:

- Setzen Sie niemals eine Bedingung mit einer Aggregation in die `WHERE`-Klausel. Sie erhalten einen Fehler.
- Setzen Sie niemals eine Bedingung in die `HAVING`-Klausel, die keine Aggregation betrifft. Solche Bedingungen werden in der `WHERE`-Klausel viel effizienter ausgewertet.

Sie werden bemerken, dass sich die HAVING-Klausel auf die Aggregation COUNT(*) bezieht:

```
SELECT COUNT(*) AS num_waterfalls

…

HAVING COUNT(*) = 6;
```

und nicht den Alias:

```
# Code läuft nicht

SELECT COUNT(*) AS num_waterfalls

…

HAVING num_waterfalls = 6;
```

Das liegt an der Ausführungsreihenfolge der Klauseln. Die SELECT-Klausel steht vor der HAVING-Klausel. Allerdings wird die SELECT-Klausel tatsächlich *nach* der HAVING-Klausel ausgeführt.

Das bedeutet, dass der Alias num_waterfalls in der SELECT-Klausel zu dem Zeitpunkt, an dem die HAVING-Klausel ausgeführt wird, noch nicht existiert. Die HAVING-Klausel muss sich stattdessen auf die reine Aggregation COUNT(*) beziehen.

MySQL und *SQLite* sind Ausnahmen und erlauben Aliasse (`num_waterfalls`) in der `HAVING`-Klausel.

Die ORDER BY-Klausel

Die `ORDER BY`-Klausel wird verwendet, um anzugeben, wie die Ergebnisse einer Abfrage sortiert werden sollen.

Die folgende Abfrage liefert eine Liste mit Eigentümern und Wasserfällen zurück, die unsortiert ist:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,

       w.name AS waterfall_name

FROM   waterfall w

       LEFT JOIN owner o ON w.owner_id = o.id;

owner            waterfall_name

------ -----

Pictured Rocks   Munising Falls

Michigan Nature  Tannery Falls

AF LLC           Alger Falls

MI DNR           Wagner Falls
```

```
Unknown             Horseshoe Falls

…
```

Die COALESCE-Funktion

Die COALESCE-Funktion ersetzt alle NULL-Werte in einer Spalte durch einen anderen Wert. In diesem Fall wandelte sie die NULL-Werte in der Spalte `o.name` in den Text Unknown um.

Würde die COALESCE-Funktion hier nicht verwendet werden, wären Wasserfälle ohne Besitzer nicht in den Ergebnissen aufgetaucht. Stattdessen steht bei ihnen nun, dass sie einen unbekannten Besitzer haben, also Unknown. Sie können daher in die Ergebnisse einsortiert werden.

Mehr dazu finden Sie in Kapitel 7.

Die folgende Abfrage liefert die gleiche Liste zurück, nun jedoch zuerst alphabetisch nach dem Besitzer und dann nach dem Wasserfall sortiert:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY owner, waterfall_name;
```

```
owner            waterfall_name
------ -----
AF LLC           Alger Falls
MI DNR           Wagner Falls
```

```
Michigan Nature  Tannery Falls

Michigan Nature  Twin Falls #1

Michigan Nature  Twin Falls #2

…
```

Die Standardsortierung erfolgt in aufsteigender Reihenfolgt, das heißt, Text verläuft von A nach Z, und Zahlen verlaufen von der niedrigsten zur höchsten. Sie können die Schlüsselwörter ASCENDING und DE SCENDING (die als ASC und DESC abgekürzt werden können) verwenden, um die Sortierung auf den einzelnen Spalten zu kontrollieren.

Das Folgende ist eine Modifizierung der vorherigen Sortierung, allerdings werden dieses Mal die Namen der Besitzer in umgekehrter Reihenfolge sortiert:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,

       w.name AS waterfall_name

…

ORDER BY owner DESC, waterfall_name ASC;

owner            waterfall_name

______ _____

Unknown          Agate Falls

Unknown          Bond Falls

Unknown          Canyon Falls
```

```
…
```

Sie können nach Spalten und Ausdrücken sortieren, die nicht in Ihrer SELECT-Liste stehen:

```
SELECT   COALESCE(o.name, 'Unknown') AS owner,
         w.name AS waterfall_name
FROM     waterfall w
         LEFT JOIN owner o ON w.owner_id = o.id
ORDER BY o.id DESC, w.id;

owner            waterfall_name
------ -----
MI DNR           Wagner Falls
AF LLC           Alger Falls
Michigan Nature  Tannery Falls
…
```

Sie können auch nach numerischer Spaltenposition sortieren:

```
SELECT COALESCE(o.name, 'Unknown') AS owner,
       w.name AS waterfall_name
…
```

```
ORDER BY 1 DESC, 2 ASC;

owner            waterfall_name

------ -----

Unknown          Agate Falls

Unknown          Bond Falls

Unknown          Canyon Falls

…
```

Da die Zeilen einer SQL-Tabelle ungeordnet sind, könnten die Ergebnisse jedes Mal, wenn Sie eine Abfrage ausführen, in einer anderen Reihenfolge angezeigt werden – es sei denn, Sie fügen eine `ORDER BY`-Klausel hinzu.

ORDER BY kann nicht in einer Unterabfrage verwendet werden

Von den sechs Hauptklauseln kann nur `ORDER BY` nicht in einer Unterabfrage verwendet werden. Leider können Sie das Ordnen der Zeilen einer Unterabfrage nicht erzwingen.

Um dieses Problem zu umgehen, müssten Sie Ihre Abfrage mit einer anderen Logik neu schreiben, damit Sie nicht versucht sind, eine `ORDER BY`-Klausel in der Unterabfrage zu verwenden, und eine `ORDER BY`-Klausel nur in die äußere Abfrage einfügen.

Die LIMIT-Klausel

Wenn man sich schnell mal eine Tabelle anschauen möchte, ist es ganz praktisch, die Anzahl der ausgegebenen Zeilen zu beschränken und nicht die ganze Tabelle anzuzeigen.

MySQL, PostgreSQL und *SQLite* unterstützen die `LIMIT`-Klausel. *Oracle* und *SQL Server* verwenden eine andere Syntax mit derselben Funktionalität:

```
— MySQL, PostgreSQL und SQLite

SELECT *

FROM owner

LIMIT 3;

— Oracle

SELECT *

FROM owner

WHERE ROWNUM <= 3;

— SQL Server

SELECT TOP 3 *

FROM owner;

id  name             phone         type

- ------ ----- ---

  1 Pictured Rocks   906.387.2607  public

  2 Michigan Nature  517.655.5655  private

  3 AF LLC                         private
```

Eine andere Möglichkeit, die Anzahl der zurückgegebenen Zeilen zu beschränken, besteht darin, auf einer Spalte in der WHERE-Klausel zu filtern. Die

Filterung wird sogar noch schneller ausgeführt, wenn die Spalte indiziert ist.

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z