11 Tabellen verbinden

Daten für Auswertungen verbinden.

- Grundkurs
  - Kartesisches Produkt, CROSS JOIN
  - INNER JOIN mit zwei und mehr Tabellen
  - LEFT und RIGHT OUTER JOIN
- Vertiefendes
  - Abkürzung mit USING
  - NATURAL JOIN
  - JOIN als Datenquelle
  - Verknüpfung über Nichtschlüsselspalten
  - EQUI JOIN
  - OUTER JOIN und referenzielle Integrität
  - SELF JOIN
  - Common Table Expression (WITH)
  - Einfluss von Indizes auf JOIN

Spätestens jetzt sind wir im nichttrivialen Bereich von `SELECT` angekommen. Daten aus mehreren Tabellen zusammenzuführen, ist Alltagsgeschäft und wird von vielen DBMS-Tools unterstützt. Trotzdem müssen Sie das Handwerk dahinter verstehen, denn jeder DB-Designer muss wissen, wie die Daten später wieder zusammengebastelt werden müssen. Ohne den Aufwand zu kennen, lässt sich kein seriöses ER-Modell erstellen.[1]

Die Quelltexte dieses Kapitels stehen in der Datei `listing08.sql` (siehe Listing 28.8 auf Seite 477, Listing 28.47 auf Seite 601 und Listing 28.32 auf Seite 550).

[1] So, wie wir es in den ersten Kapiteln getan haben ;-).

11.1 Heiße Liebe: Primär-Fremdschlüsselpaare

SQL:2016, **MySQL/MariaDB**, **PostgreSQL**, **T-SQL**

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM tabellenname[, tabellenname]*
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
;
```

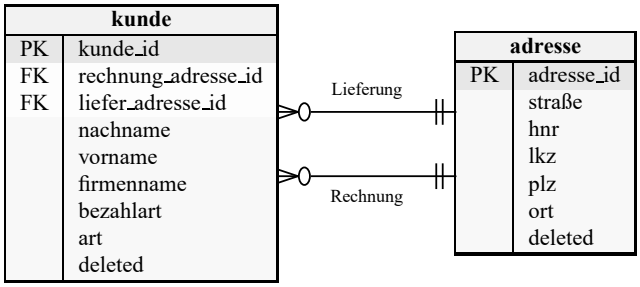


Bild 11.1 ER-Modell: `kunde` und `adresse`

Die bisher durchgeführten Auswertungen fanden immer auf *einer* Tabelle statt. Wenn wir uns die ER-Modelle für den Online-Shop anschauen, erkennen wir, dass inhaltlich zusammenhängende Informationen oft auf mehrere Tabellen verteilt sind.[2]

So stehen beispielsweise die Adressdaten eines Kunden in einer anderen Tabelle als sein Name. Für ein Rechnungsschreiben werden beide Informationen wieder miteinander zu verknüpfen sein. Eine Variante des `SELECT` erlaubt die Verwendung mehrerer Tabellen in einem `SELECT`. Wollen wir mal schauen, was dabei herauskommt:

```
mysql> SELECT
    ->  kunde_id, nachname, vorname, rechnung_adresse_id, adresse_id
    ->  FROM
    ->   kunde, adresse
    -> ;
+----------+-------------+---------+--------------------+------------+
| kunde_id | nachname    | vorname | rechnung_adresse_id | adresse_id |
+----------+-------------+---------+--------------------+------------+
|        1 | Gamdschie   | Samweis |                  1 |          1 |
|        2 | Beutlin     | Frodo   |                  2 |          1 |
|        3 | Beutlin     | Bilbo   |                  2 |          1 |
|        4 | Telcontar   | Elessar |                  3 |          1 |
|        5 | Earendilionn | Elrond  |                  4 |          1 |
|        6 | Eichenschild | Thorin  |               NULL |          1 |
|        1 | Gamdschie   | Samweis |                  1 |          2 |
|        2 | Beutlin     | Frodo   |                  2 |          2 |
```

[2] Dies ist gerade der entscheidende Unterschied zu OO-Datenbanken oder NoSQL-Ansätzen.

```
|         3 | Beutlin      | Bilbo   |                   2 |          2 |
|         4 | Telcontar    | Elessar |                   3 |          2 |
|         5 | Earendilionn | Elrond  |                   4 |          2 |
|         6 | Eichenschild | Thorin  |                NULL |          2 |
|         1 | Gamdschie    | Samweis |                   1 |          3 |
|         2 | Beutlin      | Frodo   |                   2 |          3 |
|         3 | Beutlin      | Bilbo   |                   2 |          3 |
|         4 | Telcontar    | Elessar |                   3 |          3 |
|         5 | Earendilionn | Elrond  |                   4 |          3 |
|         6 | Eichenschild | Thorin  |                NULL |          3 |
|         1 | Gamdschie    | Samweis |                   1 |          4 |
|         2 | Beutlin      | Frodo   |                   2 |          4 |
|         3 | Beutlin      | Bilbo   |                   2 |          4 |
|         4 | Telcontar    | Elessar |                   3 |          4 |
|         5 | Earendilionn | Elrond  |                   4 |          4 |
|         6 | Eichenschild | Thorin  |                NULL |          4 |
|         1 | Gamdschie    | Samweis |                   1 |          5 |
|         2 | Beutlin      | Frodo   |                   2 |          5 |
|         3 | Beutlin      | Bilbo   |                   2 |          5 |
|         4 | Telcontar    | Elessar |                   3 |          5 |
|         5 | Earendilionn | Elrond  |                   4 |          5 |
|         6 | Eichenschild | Thorin  |                NULL |          5 |
|         1 | Gamdschie    | Samweis |                   1 |         10 |
|         2 | Beutlin      | Frodo   |                   2 |         10 |
|         3 | Beutlin      | Bilbo   |                   2 |         10 |
|         4 | Telcontar    | Elessar |                   3 |         10 |
|         5 | Earendilionn | Elrond  |                   4 |         10 |
|         6 | Eichenschild | Thorin  |                NULL |         10 |
|         1 | Gamdschie    | Samweis |                   1 |         11 |
|         2 | Beutlin      | Frodo   |                   2 |         11 |
|         3 | Beutlin      | Bilbo   |                   2 |         11 |
|         4 | Telcontar    | Elessar |                   3 |         11 |
|         5 | Earendilionn | Elrond  |                   4 |         11 |
|         6 | Eichenschild | Thorin  |                NULL |         11 |
+-----------+--------------+---------+---------------------+------------+
42 rows in set (0.00 sec)
```

In Zeile 4 werden zwei Tabellen hinter dem `FROM` angegeben. Das ist neu; bisher stand hier immer nur *die eine* Tabelle.

Im Ergebnis werden mir 42 Zeilen einer *neuen* Tabelle angezeigt. Aber wie sind diese Zeilen gebildet worden? Es fällt auf, dass die `kunde_id` sich nach dem Schema $1,2,3,4,5,6$ wiederholt. Ebenso interessant ist, dass die `adresse_id` mit jeweils sechs gleichen Werten auftaucht.

Betrachten Sie nur die Werte von `kunde_id` und `adresse_id`, so erkennen Sie Datenpaare, die wie folgt aufgebaut sind: Jede Adresse ist mit allen Kunden kombiniert worden. Adresse 1 mit Kunde 1 bis 6, Adresse 2 mit Kunde 1 bis 6 usw. Da es sechs Kunden und sieben Adressen gibt, erhalten wir 42 Kombinationen, das *Kartesische Produkt*.

Definition 40: Kartesisches Produkt
Seien A und B zwei endliche Mengen, $a \in A$ und $b \in B$, dann ist die Menge aller unterschiedlichen Paare (a, b) das *Kartesische Produkt* K der Mengen A und B. ■

Diese Definition lässt $A = B$ zu. Wenn n die Anzahl der Elemente in A ist und m die Anzahl der Elemente in B, dann hat K $n \times m$ viele Elemente.

Definition 41: CROSS JOIN
Das Kartesische Produkt zweier Mengen wird auch *CROSS JOIN* oder *Kreuzprodukt* genannt.

Toll, ein Kartesisches Produkt[3], ja und?

Aufgabe 11.1: Betrachten Sie genau die Zahlenpaare `rechnung_adresse_id` und `adresse_id`. Formulieren Sie eine `WHERE`-Klausel, die genau die Zeilen übrig lässt, die zu einem Kunden die richtige Adressnummer angeben.

In den Zeilen 9, 16, 17, 24 und 31 stehen jeweils die gleichen Werte. Ausformuliert bedeutet dies: Im Fremdschlüssel `rechnung_adresse_id` steht der gleiche Wert wie im Primärschlüssel `adresse_id`. Das sind genau die Elemente des Kartesischen Produkts, die eine gültige Verknüpfung zwischen den beiden Tabellen darstellen. Die `WHERE`-Klausel sollte somit nur diese Zeilen übrig lassen:

```
1  mysql> SELECT
2      ->  kunde_id, nachname, vorname, rechnung_adresse_id, adresse_id
3      ->  FROM
4      ->   kunde, adresse
5      ->  WHERE
6      ->   rechnung_adresse_id = adresse_id
7      -> ;
8  +----------+--------------+---------+--------------------+------------+
9  | kunde_id | nachname     | vorname | rechnung_adresse_id | adresse_id |
10 +----------+--------------+---------+--------------------+------------+
11 |        1 | Gamdschie    | Samweis |                  1 |          1 |
12 |        2 | Beutlin      | Frodo   |                  2 |          2 |
13 |        3 | Beutlin      | Bilbo   |                  2 |          2 |
14 |        4 | Telcontar    | Elessar |                  3 |          3 |
15 |        5 | Earendilionn | Elrond  |                  4 |          4 |
16 +----------+--------------+---------+--------------------+------------+
17 5 rows in set (0.00 sec)
```

Aufgabe 11.2: Was ist mit den Adressen mit den Primärschlüsselwerten 5, 10 und 11 passiert? Wie können Sie das inhaltlich interpretieren? Und was ist mit dem Kunden *Thorin Eichenschild*?

Die Reduzierung des Kartesischen Produkts auf die Zeilen mit passenden Schlüsselpaaren ist schon ein `INNER JOIN`. Was uns nun noch fehlt, ist eine *coole SELECT-Erweiterung* dazu.

[3] Wird von den Azubis gerne *Orgienjoin* genannt.

11.2 INNER JOIN zwischen zwei Tabellen

Definition 42: INNER JOIN
Der *INNER JOIN* zweier Tabellen ist die Teilmenge des kartesischen Produkts, für welche gilt, dass die Fremdschlüsselwerte zu den Primärschlüsselwerten passen.

Die Formulierung über das Kartesische Produkt mit der passenden `WHERE`-Klausel ist für die Programmierung ein wenig sperrig. Stellen Sie sich vor, dass die Ergebnismenge weitere Bedingungen erfüllen muss oder keine echten Tabellen verwendet werden, sondern Unterabfragen[4]. Deshalb gibt es eine eigene Syntax für den INNER JOIN:

SQL:2016, **MySQL/MariaDB**, **PostgreSQL**, **T-SQL**

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabellenname_fk INNER JOIN tabellenname_pk
       ON tabellenname_fk.fk = tabellenname_pk.pk
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
;
```

Der obige Befehl sähe umgebaut so aus:

```
1  SELECT
2   kunde_id, nachname, vorname, rechnung_adresse_id, adresse_id
3   FROM
4    kunde INNER JOIN adresse ON rechnung_adresse_id = adresse_id
5  ;
```

Aufgabe 11.3: Bauen Sie den Befehl so um, dass folgende Ausgabe erzeugt wird:

```
+--------------+---------+-----------------+-----+--------------+
| nachname     | vorname | strasse         | hnr | ort          |
+--------------+---------+-----------------+-----+--------------+
| Gamdschie    | Samweis | Beutelhaldenweg | 5   | Hobbingen    |
| Beutlin      | Frodo   | Beutelhaldenweg | 1   | Hobbingen    |
| Beutlin      | Bilbo   | Beutelhaldenweg | 1   | Hobbingen    |
| Telcontar    | Elessar | Auf der Feste   | 1   | Minas Tirith |
| Earendilionn | Elrond  | Letztes Haus    | 4   | Bruchtal     |
+--------------+---------+-----------------+-----+--------------+
```

11.2.1 Bauanleitung für einen INNER JOIN

Die Programmierung eines `INNER JOIN` fällt meinen Schülerinnen und Schülern oft schwer. Deshalb will ich hier ein wenig ausführlicher beschreiben, wie Sie einen `INNER`

[4] Siehe Kapitel 13 auf Seite 223.

JOIN zusammenbauen können. Die passende Aufgabe dazu: Wir wollen zu einer Bankverbindung die Bankleitzahl und den Banknamen wissen.

1. **Ermitteln der beteiligten Tabellen:** In unserem Fall sind es die Tabellen bankverbindung und bank. Schreiben Sie sich diese Tabellen auf: eine auf die linke Seite vom Blatt und eine auf die rechte Seite.

2. **Ermitteln der Primär- und Fremdschlüssel:** Schreiben Sie unter den Tabellennamen jeweils den Primärschlüssel und alle vorkommenden Fremdschlüssel:

bankverbindung	bank
kunde_id	bank_id
bankverbindung_id	
bank_id	

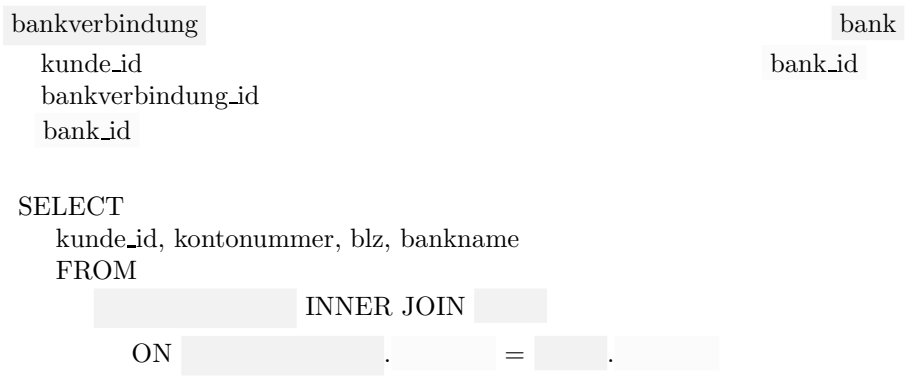
3. **Fremdschlüssel festlegen:** In einer der Tabellen muss ein Fremdschlüssel vorkommen, der auf die andere Tabelle zeigt. Wenn Sie beim Design alles richtig gemacht haben und die Namenskonvention beachten, finden Sie diesen sehr schnell. Es kommen zwei Fremdschlüssel infrage: kunde_id und bank_id. Da eine der Tabellen bank heißt und wir der Namenskonvention gefolgt sind, muss es bank_id sein. Markieren Sie den Fremdschlüssel z.B. mit einem Textmarker:

bankverbindung	bank
kunde_id	bank_id
bankverbindung_id	
bank_id	

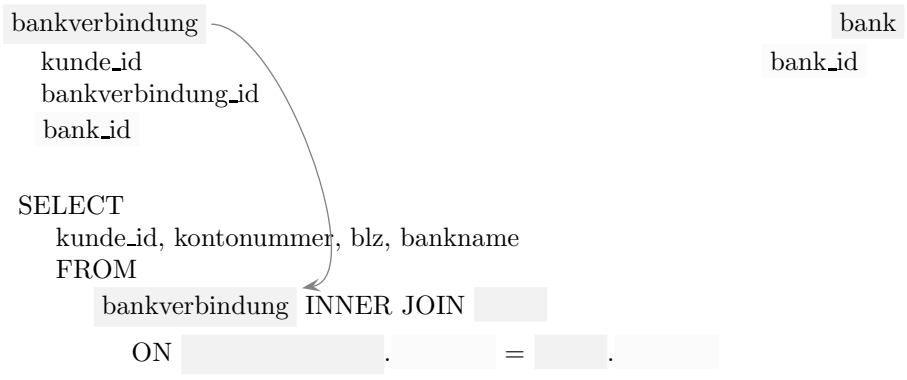
4. **Primärschlüssel festlegen:** Jetzt markieren Sie in der gleichen Farbe in der anderen Tabelle den dazugehörigen Primärschlüssel. Hier ist es die Spalte bank_id:

bankverbindung	bank
kunde_id	bank_id
bankverbindung_id	
bank_id	

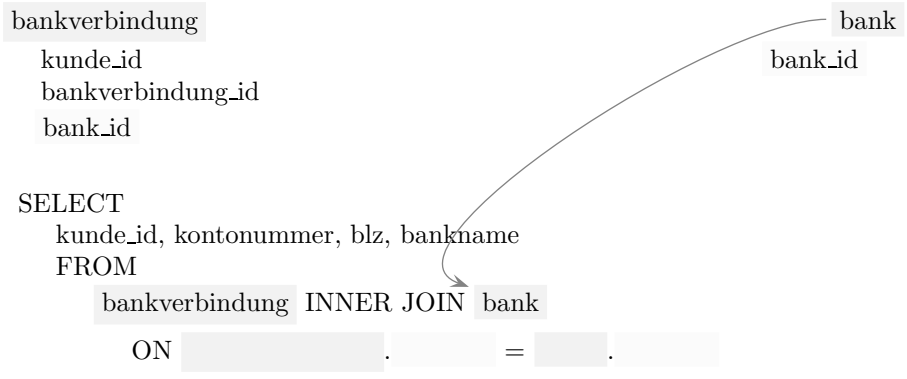
5. **Schablone benutzen:** Jetzt schreiben Sie auf das Blatt die Schablone für den INNER JOIN. Das Ganze sollte jetzt so aussehen:



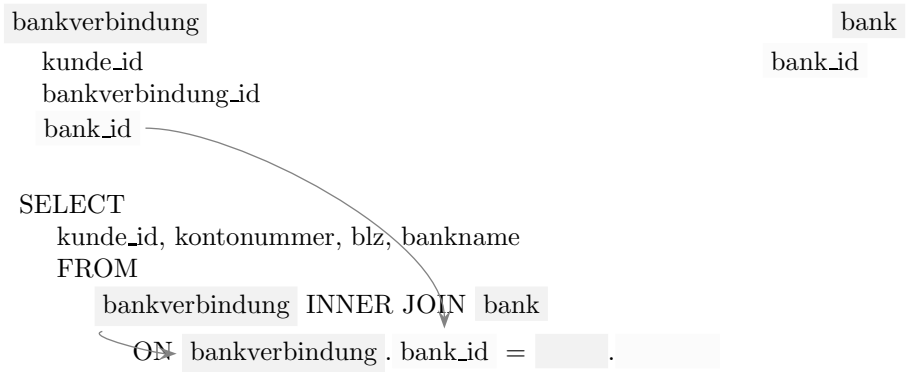
6. **Fremdschlüsseltabelle eintragen:** Fügen Sie links vom `INNER JOIN` den Tabellennamen für die Tabelle mit dem markierten Fremdschlüssel ein:



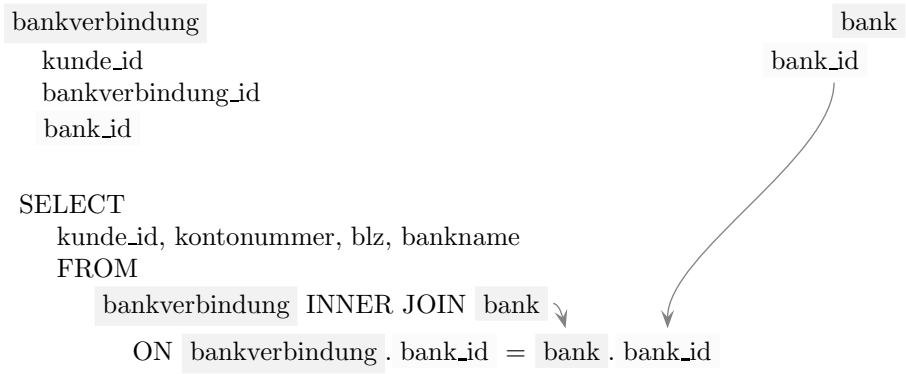
7. **Primärschlüsseltabelle eintragen:** Fügen Sie rechts vom `INNER JOIN` den Tabellennamen für die Tabelle mit dem markierten Primärschlüssel ein:



8. **Fremdschlüssel eintragen:** Fügen Sie zwischen dem `ON` und dem Gleichheitszeichen den Namen der Fremdschlüsseltabelle, einen Punkt und den Namen des Fremdschlüssels ein:



9. **Primärschlüssel eintragen:** Fügen Sie nach dem = den Namen der Primärschlüsseltabelle, einen Punkt und den Namen des Primärschlüssels ein:



10. **Fertig!**

```
mysql> SELECT
    ->  kunde_id, kontonummer, blz, bankname
    ->  FROM
    ->   bankverbindung INNER JOIN bank ON bankverbindung.bank_id = bank.bank_id
    ->  ORDER BY kunde_id, kontonummer;
+----------+-------------+----------+---------------+
| kunde_id | kontonummer | blz      | bankname      |
+----------+-------------+----------+---------------+
|        1 | 1111111111  | 10010010 | Postbank      |
|        1 | 1111111112  | 10010010 | Postbank      |
|        2 | 2222222221  | 10060198 | Pax-Bank      |
|        3 | 3333333331  | 10060198 | Pax-Bank      |
|        4 | 4444444441  | 12070000 | Deutsche Bank |
|        5 | 5555555551  | 12070000 | Deutsche Bank |
+----------+-------------+----------+---------------+
```

Ein zweites Beispiel: Zu einem Kunden werden die Kontonummern ausgegeben.

1. **Ermitteln der beteiligten Tabellen**: Es sind `kunde` und `bankverbindung`.
2. **Ermitteln der Primär- und Fremdschlüssel:** In der Tabelle `bankverbindung` gibt es den zusammengesetzten Primärschlüssel mit `kunde_id` und `bankverbindung_nr` und den Fremdschlüssel `bank_id`. In der Tabelle `kunde` gibt es den Primärschlüssel `kunde_id` und die beiden Fremdschlüssel `rechnung_adresse_id` und `liefer_adresse_id`.

3. **Fremdschlüssel festlegen:** Da wir die Namenskonvention beachtet haben, brauchen wir nur nach einem Fremdschlüssel suchen, der so wie die andere Tabelle heißt. Dies ist in unserem Fall `kunde_id`.
4. **Primärschlüssel festlegen:** Jetzt wird die Spalte `kunde_id` in der Tabelle `kunde` markiert.
5. **Schablone benutzen:** Jetzt schreiben Sie auf das Blatt die Schablone für den `INNER JOIN`.
6. **Fremdschlüsseltabelle eintragen:** Fügen Sie links vom `INNER JOIN` den Tabellennamen für die Tabelle mit dem markierten Fremdschlüssel ein.
7. **Primärschlüsseltabelle eintragen:** Fügen Sie an die passende Stelle den Tabellennamen für die Tabelle mit dem markierten Primärschlüssel ein.
8. **Fremdschlüssel eintragen:** Fügen Sie an die passende Stelle den Spaltennamen des markierten Fremdschlüssels ein.
9. **Primärschlüssel eintragen:** Fügen Sie an die passende Stelle den Spaltennamen des markierten Primärschlüssels ein.
10. **Fertig!**

```
1  mysql> SELECT
2      ->  nachname, vorname, kontonummer
3      ->  FROM
4      ->   bankverbindung b INNER JOIN kunde k ON b.kunde_id = k.kunde_id
5      ->  ORDER BY nachname, vorname, kontonummer;
6  +--------------+---------+-------------+
7  | nachname     | vorname | kontonummer |
8  +--------------+---------+-------------+
9  | Beutlin      | Bilbo   | 3333333331  |
10 | Beutlin      | Frodo   | 2222222221  |
11 | Earendilionn | Elrond  | 5555555551  |
12 | Gamdschie    | Samweis | 1111111111  |
13 | Gamdschie    | Samweis | 1111111112  |
14 | Telcontar    | Elessar | 4444444441  |
15 +--------------+---------+-------------+
```

Haben Sie gesehen, dass in Zeile 4 die Tabellennamen durch Aliase ersetzt wurden?

Aufgabe 11.4: Geben Sie zu allen Kunden Namen und Rechnungsadresse aus.

Aufgabe 11.5: Geben Sie zu allen Kunden den Namen und die Lieferadresse aus. Interpretieren Sie das Ergebnis.

Aufgabe 11.6: Geben Sie zu jedem Kunden den Namen und die Bestellungen nach Kundennamen und Bestelldatum sortiert aus. Die letzte Bestellung des Kunden soll zuerst erscheinen.

Aufgabe 11.7: Geben Sie zu jeder Bestellung die Kundennummer, das Bestelldatum und die Positionen aus. Die Sortierung soll nach Kundennummer, Bestellnummer und Position erfolgen.

Aufgabe 11.8: Geben Sie zu jeder Position den Artikelnamen aus. Es soll nach Artikelname sortiert werden.

11.2.2 Abkürzende Schreibweisen

SQL:2016, MySQL/MariaDB

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabellenname_fk INNER JOIN tabellenname_pk
      [{ON tabellenname_fk.fk = tabellenname_pk.pk|USING(spaltenname)}]
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
;
```

Falls der Name des Fremdschlüssels genau der gleiche ist wie der des Primärschlüssels, klnne Sie die `ON`-Klausel durch eine kürzere Variante mit `USING` ersetzen:

```
1 SELECT
2  kunde_id, kontonummer, blz, bankname
3  FROM
4   bankverbindung INNER JOIN bank USING(bank_id)
5 ;
```

Bei den Fremdschlüsseln, die anders als die Primärschlüssel heißen, wie beispielsweise `rechnung_adresse_id`, ist eine solche Abkürzung nicht möglich.

Aufgabe 11.9: Bauen Sie Ihre Lösungen zu den Aufgaben um, wenn ein `USING` möglich ist.

Sind die Fremdschlüssel-/Primärschlüsselspalten die einzigen Spalten, die in beiden Tabellen gleich sind und die Verknüpfung definieren, spricht man von einem `NATURAL JOIN`.

Definition 43: NATURAL JOIN
Werden zwei Tabellen über die Spalten verknüpft, die den gleichen Namen haben, spricht man von einem *NATURAL JOIN*.

```
1 SELECT
2  kunde_id, kontonummer, blz, bankname
3  FROM
4   bankverbindung NATURAL JOIN bank
5 ;
```

Hinweis: Bitte beachten Sie, dass bei unseren Tabellen in einem `NATURAL JOIN` auch die Spalte `deleted` in die Verknüpfung mit einfließt.

11.2.3 Als Datenquelle für temporäre Tabellen

Wir haben oben die Bankverbindung zu einem Kunden ermittelt. Annahme: Wir wollen jetzt viele Auswertungen der Kunden inklusive der Bankverbindung machen, dann müs-

sen jedes Mal im entsprechenden SELECT die Informationen mit INNER JOIN verknüpft werden.

Obwohl INNER JOIN-Anweisungen durch passend gewählte Indizes sehr beschleunigt werden, entsteht trotzdem eine Rechnerlast auf dem Server, die jedes Mal das gleiche – oder fast das gleiche – Ergebnis liefert.[5]

Nun können wir plausibel annehmen, dass sich die Kundenstammdaten (siehe Definition 36) für einen gewissen Auswertungszeitraum nicht ändern. Daher könnten wir statt dessen das Ergebnis des INNER JOIN in eine (temporäre) Tabelle ablegen und mit dieser weiterarbeiten.

Als Beispiel wollen wir die Kundendaten mit Rechnungsanschrift und allen Bestellungen ausgeben und danach die Kundendaten mit Rechnungsanschrift mit allen Bankverbindungen.

In beiden Fällen werden die Kundendaten mit Rechnungsanschrift benötigt. Das können wir schon:

```
1  SELECT
2   k.kunde_id, k.nachname, k.vorname, a.strasse, a.hnr, a.plz, a.ort
3   FROM
4    kunde k INNER JOIN adresse a
5     ON k.rechnung_adresse_id = a.adresse_id
6  ;
```

Sie bemerken bitte, dass hier abkürzende Alias (Zeile 4) verwendet werden. Das ist gerade bei Verknüpfungen sehr beliebt, da hier oft zwischen den Tabellen unterschieden werden muss. Diese Abfrage wird jetzt in eine Variante des CREATE TABLE eingebaut.

MySQL/MariaDB

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tabellenname
  SELECT auswahl
;
```

```
1  mysql> CREATE TEMPORARY TABLE tmp_kadresse
2      ->  SELECT
3      ->   k.kunde_id, k.nachname, k.vorname, a.strasse, a.hnr, a.plz, a.ort
4      ->   FROM
5      ->    kunde k INNER JOIN adresse a
6      ->     ON k.rechnung_adresse_id = a.adresse_id;
7
8  Query OK, 5 rows affected (0.09 sec)
9  Records: 5  Duplicates: 0  Warnings: 0
10
11 mysql> SELECT kunde_id, nachname, ort FROM tmp_kadresse;
12 +----------+--------------+--------------+
13 | kunde_id | nachname     | ort          |
14 +----------+--------------+--------------+
15 |        1 | Gamdschie    | Hobbingen    |
16 |        2 | Beutlin      | Hobbingen    |
17 |        3 | Beutlin      | Hobbingen    |
18 |        4 | Telcontar    | Minas Tirith |
```

[5] Der Query Cache ist seit der MySQL Version 8.0 abgeschaltet (siehe [MyS21d]).

```
|        5 | Earendilionn | Bruchtal     |
+----------+--------------+--------------+
```

Jetzt zur ersten Auswertung: Alle Kunden mit Adressdaten und ihren Bestellungen:

```
mysql> SELECT
    ->  t.kunde_id, t.nachname, t.ort, b.bestellung_id, DATE(b.datum)
    ->  FROM
    ->   bestellung b INNER JOIN tmp_kadresse t USING(kunde_id);

+----------+-----------+-----------+---------------+---------------+
| kunde_id | nachname  | ort       | bestellung_id | DATE(b.datum) |
+----------+-----------+-----------+---------------+---------------+
|        1 | Gamdschie | Hobbingen |             1 | 2012-03-24    |
|        2 | Beutlin   | Hobbingen |             2 | 2012-03-23    |
+----------+-----------+-----------+---------------+---------------+
```

Aufgabe 11.10: Erzeugen Sie mit der Spaltenliste `t.*, b.bestellung_id, datum` eine neue temporäre Tabelle und verknüpfen Sie diese mit den Positionen der Bestellungen. Achten Sie auf eine sinnvolle Sortierung.

Jetzt zur zweiten Auswertung: Alle Kunden mit Adressdaten und allen Bankverbindungen. Das wollen wir jetzt besonders schön machen. Zuerst eine temporäre Tabelle für die Bankleitzahl und den Banknamen, und dann werden diese beiden temporären Tabellen wieder verknüpft. Und weil wir es jetzt können, wird am Ende noch eine temporäre Tabelle gebaut.

```
mysql> CREATE TEMPORARY TABLE tmp_kbank
    ->  SELECT
    ->   bv.kunde_id, bv.bankverbindung_nr, bv.kontonummer,
    ->   bv.iban, ba.blz, ba.bankname
    ->   FROM
    ->    bankverbindung bv INNER JOIN bank ba USING(bank_id);

mysql> CREATE TEMPORARY TABLE tmp_kbankeinzug
    ->  SELECT
    ->   ka.*, kb.bankverbindung_nr, kb.kontonummer,
    ->   kb.iban, kb.blz, kb.bankname
    ->   FROM
    ->    tmp_kadresse ka INNER JOIN tmp_kbank kb USING(kunde_id);

mysql> SELECT kunde_id, nachname, ort, bankname FROM tmp_kbankeinzug;
+----------+--------------+--------------+---------------+
| kunde_id | nachname     | ort          | bankname      |
+----------+--------------+--------------+---------------+
|        1 | Gamdschie    | Hobbingen    | Postbank      |
|        1 | Gamdschie    | Hobbingen    | Postbank      |
|        2 | Beutlin      | Hobbingen    | Pax-Bank      |
|        3 | Beutlin      | Hobbingen    | Pax-Bank      |
|        4 | Telcontar    | Minas Tirith | Deutsche Bank |
|        5 | Earendilionn | Bruchtal     | Deutsche Bank |
+----------+--------------+--------------+---------------+
```

Das Zusammenspiel der tatsächlichen und temporären Tabellen sei hier in einer Baumstruktur dargestellt. Die Blätter[6] sind die Ursprungstabellen. Zwei davon werden in jeweils

[6] die Knoten ganz links

eine temporäre Tabelle mit `INNER JOIN` zusammengefasst. Die beiden neu erzeugten temporären Tabellen werden wiederum in eine temporäre Tabelle vereinigt.

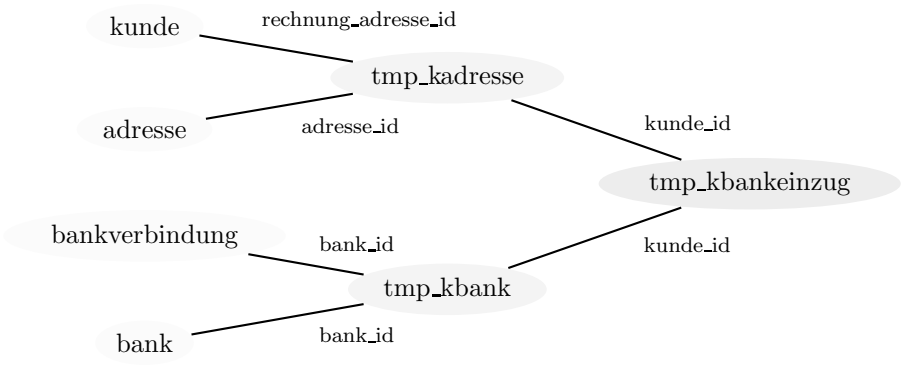


Bild 11.2 Temporäre Tabellen als Datenquelle

Jede der Tabellen kann nun unabhängig voneinander verwendet werden, da die Datensätze in den temporären Tabellen keine Verweise auf die ursprünglichen Datensätze sind, sondern Kopien. Diese *Unabhängigkeit* ist sehr wichtig, wenn Sie mit vielen konkurrierenden Zugriffen auf die Tabellen `kunde` etc. rechnen.

Bei der Betrachtung von Transaktionen (siehe Kapitel 19 auf Seite 313) werden wir noch sehen, dass Tabellen für Operationen gesperrt werden. Durch die temporären Tabellen kann aber auf der Tabelle `kunde` gearbeitet werden, während Auswertungen auf `tmp_kadresse` stattfinden.

Hinweis: Ich habe oben erwähnt, dass es sich um Stammdaten handelt und somit während der Auswertungen eine geringe Änderungswahrscheinlichkeit besteht. Bei Bewegungsdaten ist das Verfahren genau abzuwägen.

11.2.4 Ist Ihnen was aufgefallen?

Beim Bau der letzten temporären Tabelle gab es keinen Primärschlüssel!

```
mysql> DESCRIBE tmp_kadresse;
+----------+---------------+------+-----+---------+-------+
| Field    | Type          | Null | Key | Default | Extra |
+----------+---------------+------+-----+---------+-------+
| kunde_id | int) unsigned | NO   |     | 0       | NULL  |
| nachname | varchar(255)  | NO   |     |         | NULL  |
| vorname  | varchar(255)  | NO   |     |         | NULL  |
| strasse  | varchar(255)  | NO   |     |         | NULL  |
| hnr      | varchar(255)  | NO   |     |         | NULL  |
| plz      | char(9)       | NO   |     |         | NULL  |
| ort      | varchar(255)  | NO   |     |         | NULL  |
+----------+---------------+------+-----+---------+-------+

mysql> DESCRIBE tmp_kbank;
```

```
+-------------------+--------------+------+-----+---------+-------+
| Field             | Type         | Null | Key | Default | Extra |
+-------------------+--------------+------+-----+---------+-------+
| kunde_id          | int unsigned | NO   |     | NULL    | NULL  |
| bankverbindung_nr | int unsigned | NO   |     | NULL    | NULL  |
| kontonummer       | char(25)     | NO   |     |         | NULL  |
| iban              | char(34)     | NO   |     |         | NULL  |
| blz               | char(12)     | NO   |     |         | NULL  |
| bankname          | varchar(255) | NO   |     |         | NULL  |
+-------------------+--------------+------+-----+---------+-------+
```

Und tatsächlich: Für einen `INNER JOIN` ist es nicht nötig, Primär-Fremdschlüsselpaare zu bilden. Dem Befehl ist es völlig egal, was bei einem `INNER JOIN` in der `ON`- oder `USING`-Klausel steht. Es wird nur die Verträglichkeit der Datentypen untersucht.

Natürlich erfolgt die überwiegende Anzahl der Verknüpfungen auf Primär-Fremdschlüsselpaaren. Aber hier haben wir ein Beispiel dafür, dass auch die Verknüpfung über Nichtschlüsselspalten[7] sinnvoll sein kann.

Es geht sogar noch weiter. Bei der `ON`-Klausel ist das Gleichheitszeichen nicht zwingend vorgeschrieben:

```
mysql> SELECT
    ->  k.kunde_id, k.vorname, a.strasse, a.hnr, a.plz, a.ort
    ->  FROM
    ->   kunde k INNER JOIN adresse a
    ->    ON k.rechnung_adresse_id <= a.adresse_id;
+----------+---------+-----------------+-----+-------+--------------+
| kunde_id | vorname | strasse         | hnr | plz   | ort          |
+----------+---------+-----------------+-----+-------+--------------+
|        1 | Samweis | Beutelhaldenweg | 5   | 67676 | Hobbingen    |
|        1 | Samweis | Beutelhaldenweg | 1   | 67676 | Hobbingen    |
|        2 | Frodo   | Beutelhaldenweg | 1   | 67676 | Hobbingen    |
|        3 | Bilbo   | Beutelhaldenweg | 1   | 67676 | Hobbingen    |
|        1 | Samweis | Auf der Feste   | 1   | 54786 | Minas Tirith |
|        2 | Frodo   | Auf der Feste   | 1   | 54786 | Minas Tirith |
|        3 | Bilbo   | Auf der Feste   | 1   | 54786 | Minas Tirith |
|        4 | Elessar | Auf der Feste   | 1   | 54786 | Minas Tirith |
|        1 | Samweis | Letztes Haus    | 4   | 87567 | Bruchtal     |
|        2 | Frodo   | Letztes Haus    | 4   | 87567 | Bruchtal     |
|        3 | Bilbo   | Letztes Haus    | 4   | 87567 | Bruchtal     |
[...]
|        2 | Frodo   | Baradur         | 1   | 62519 | Lugburz      |
|        3 | Bilbo   | Baradur         | 1   | 62519 | Lugburz      |
|        4 | Elessar | Baradur         | 1   | 62519 | Lugburz      |
|        5 | Elrond  | Baradur         | 1   | 62519 | Lugburz      |
|        1 | Samweis | Hochstrasse     | 4a  | 44879 | Bochum       |
|        2 | Frodo   | Hochstrasse     | 4a  | 44879 | Bochum       |
|        3 | Bilbo   | Hochstrasse     | 4a  | 44879 | Bochum       |
|        4 | Elessar | Hochstrasse     | 4a  | 44879 | Bochum       |
|        5 | Elrond  | Hochstrasse     | 4a  | 44879 | Bochum       |
|        1 | Samweis | Industriegebiet | 8   | 44878 | Bochum       |
|        2 | Frodo   | Industriegebiet | 8   | 44878 | Bochum       |
|        3 | Bilbo   | Industriegebiet | 8   | 44878 | Bochum       |
|        4 | Elessar | Industriegebiet | 8   | 44878 | Bochum       |
|        5 | Elrond  | Industriegebiet | 8   | 44878 | Bochum       |
```

[7] Immerhin sind es Fremdschlüssel.

```
35 +----------+---------+-----------------+-----+-------+--------------+
36 28 rows in set (0.00 sec)
```

In Zeile 5 ist anstelle des = ein <= verwendet worden. Ich kann mir zwar keine vernünftige Anwendung dafür ausdenken, will aber nicht bestreiten, dass es sie irgendwo gibt. Wird aber die Verknüpfung über die Gleichheit hergestellt, hat das Ganze einen schönen Namen:

Definition 44: EQUI JOIN
Wird bei INNER JOIN auf die Gleichheit von Fremdschlüsselwert und Primärschlüsselwert getestet, spricht man von einem *EQUI JOIN*.

Sie haben sicher bei der Definition 42 auf Seite 185 bemerkt, dass nur allgemein von *passen* gesprochen wird. Was immer dieses *passen* auch bedeutet. Der Test auf Gleichheit ist aber so gängig, dass der Begriff `INNER JOIN` synonym für `EQUI JOIN` verwendet wird.

Hinweis: Lassen Sie sich nicht verwirren. Erst mal nach Primär-Fremdschlüsselpaaren suchen und diese mit = verknüpfen. In den absolut meisten Fällen sind Sie auf der sicheren Seite. Erst, wenn das so überhaupt nicht klappen sollte, denken Sie über Alternativen nach.

11.3 INNER JOIN über mehr als zwei Tabellen

Eine Möglichkeit, mehr als zwei Tabellen zu verknüpfen, haben Sie oben auf Seite 191 kennengelernt. Die Verwendung von temporären Tabellen bietet sich aber nur bei Stammdaten an oder wenn die Änderungen unerheblich für das Gesamtergebnis sind.

Trotzdem können wir aus der Episode mit den temporären Tabellen eine wichtige Schlussfolgerung ziehen: Das Ergebnis eines `INNER JOIN`s ist wieder eine Tabelle. Wie kann ich damit die Beschränkung umgehen, dass der `INNER JOIN` nur zwei Tabellen verknüpfen kann?

Betrachten wir dazu den Klassiker für die Verknüpfung von mehr als zwei Tabellen: das Auflösen einer *n:m*-Verknüpfung (siehe Abschnitt 2.2.4 auf Seite 28). Wir haben eine *n:m*-Verknüpfung zwischen den Tabellen `artikel` und `warengruppe`.

Unser erster Versuch besteht darin, wie oben beschrieben vorzugehen, indem wir zwei *1:n*-Verknüpfungen erstellen:

1. **Ermitteln der beteiligten Tabellen**: `artikel_nm_warengruppe` und `artikel`.
2. **Ermitteln der Primär- und Fremdschlüssel:** In der Tabelle `artikel_nm_warengruppe` gibt es keine *echten* Primärschlüssel, aber die beiden Fremdschlüssel `warengruppe_id` und `artikel_id`. In der Tabelle `artikel` gibt es nur den Primärschlüssel `artikel_id`.
3. **Fremdschlüssel festlegen:** Laut Namenskonvention muss `artikel_id` in der Tabelle `artikel_nm_warengruppe` der gesuchte Fremdschlüssel sein.
4. **Primärschlüssel festlegen:** Wir markieren `artikel_id` in `artikel`.
5. **Schablone benutzen:** Jetzt schreiben Sie auf das Blatt die Schablone.

6. **Fremdschlüsseltabelle eintragen:** Fügen Sie links vom `INNER JOIN` den Tabellennamen für die Tabelle mit dem markierten Fremdschlüssel ein.
7. **Primärschlüsseltabelle eintragen:** Fügen Sie an die passende Stelle den Tabellennamen für die Tabelle mit dem markierten Primärschlüssel ein.
8. **Fremdschlüssel eintragen:** Fügen Sie an die passende Stelle den Spaltennamen des markierten Fremdschlüssels ein.
9. **Primärschlüssel eintragen:** Fügen Sie an die passende Stelle den Spaltennamen des markierten Primärschlüssels ein.
10. **Fertig!**

```
mysql> SELECT
    ->  a.bezeichnung, nm.warengruppe_id
    ->  FROM
    ->   artikel_nm_warengruppe nm INNER JOIN artikel a USING(artikel_id);
+---------------+----------------+
| bezeichnung   | warengruppe_id |
+---------------+----------------+
| Feder         |              1 |
| Papier (100)  |              1 |
| Schaufel      |              3 |
| Schaufel      |              4 |
| Silberzwiebel |              2 |
| Silberzwiebel |              3 |
| Spaten        |              3 |
| Spaten        |              4 |
| Tinte (blau)  |              1 |
| Tinte (gold)  |              1 |
| Tinte (rot)   |              1 |
| Tulpenzwiebel |              2 |
| Tulpenzwiebel |              3 |
+---------------+----------------+
```

Das ganze Prozedere mit den Tabellen `artikel_nm_warengruppe` und `warengruppe` führt zu einer zweiten Verknüpfung:

```
mysql> SELECT
    ->  w.bezeichnung, nm.artikel_id
    ->  FROM
    ->   artikel_nm_warengruppe nm INNER JOIN warengruppe w USING(warengruppe_id);
+--------------+------------+
| bezeichnung  | artikel_id |
+--------------+------------+
| Bürobedarf   |       3001 |
| Bürobedarf   |       3005 |
| Bürobedarf   |       3006 |
| Bürobedarf   |       3007 |
| Bürobedarf   |       3010 |
| Gartenbedarf |       7856 |
| Gartenbedarf |       7863 |
| Gartenbedarf |       9010 |
| Gartenbedarf |       9015 |
| Pflanzen     |       7856 |
| Pflanzen     |       7863 |
| Werkzeug     |       9010 |
| Werkzeug     |       9015 |
+--------------+------------+
```

Und jetzt kommt's: Wir können sehen, dass die Zeile 4 selbst als Ergebnis eine neue Tabelle erzeugt, denn das Ergebnis besteht aus Zeilen und Spalten (siehe Definition 7 auf Seite 16). Und tatsächlich können wir diese Zeile anstelle einer einzelnen Tabelle in den ersten Befehl oben einsetzen:

```
1  SELECT
2   w.bezeichnung, a.bezeichnung
3   FROM
4    artikel_nm_warengruppe nm INNER JOIN warengruppe w USING(warengruppe_id)
5                              INNER JOIN artikel a USING(artikel_id)
6  ;
```

Wo vorher also nur `artikel_nm_warengruppe` als Datenquelle eines `INNER JOIN` stand, steht nun ein kompletter `INNER JOIN` – also die ganze Zeile 4 – als Datenquelle. Dieser neue `SELECT` liefert das gewünschte Ergebnis:

```
1  +--------------+---------------+
2  | Warengruppe  | Artikel       |
3  +--------------+---------------+
4  | Bürobedarf   | Papier (100)  |
5  | Bürobedarf   | Tinte (gold)  |
6  | Bürobedarf   | Tinte (rot)   |
7  | Bürobedarf   | Tinte (blau)  |
8  | Bürobedarf   | Feder         |
9  | Gartenbedarf | Silberwiebel  |
10 | Gartenbedarf | Tulpenzwiebel |
11 | Gartenbedarf | Schaufel      |
12 | Gartenbedarf | Spaten        |
13 | Pflanzen     | Silberwiebel  |
14 | Pflanzen     | Tulpenzwiebel |
15 | Werkzeug     | Schaufel      |
16 | Werkzeug     | Spaten        |
17 +--------------+---------------+
```

Die Spezifikation des `SELECT` lässt sich somit erweitern:

SQL:2016, **PostgreSQL**, **T-SQL**

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabname [INNER JOIN tabname
      ON tabname.spaltenname = tabname.spaltenname)]*
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
```

MySQL/MariaDB

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabname [INNER JOIN tabname
      {ON tabname.spaltenname = tabname.spaltenname|USING(spaltenname)}]*
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
 ;
```

Der Abschnitt rechts vom ersten *tabname* kann beliebig oft wiederholt werden, wobei beliebig auch bedeuten kann, dass er gar nicht vorkommt. Es wäre dann ein normaler `SELECT`.

Aufgabe 11.11: Warum konnte beim letzten `SELECT` kein `NATURAL JOIN` verwendet werden?

Aufgabe 11.12: Erweitern Sie diesen `SELECT` um die Positionen, in denen der Artikel vorkommt.

Aufgabe 11.13: Erweitern Sie das Ergebnis der letzten Aufgabe um die Daten der Bestellung.

Aufgabe 11.14: Erweitern Sie das Ergebnis der letzten Aufgabe um die Daten des Kunden.

Aufgabe 11.15: Erweitern Sie das Ergebnis der letzten Aufgabe um die Rechnungsadresse des Kunden.

11.4 Es muss nicht immer heiße Liebe sein: OUTER JOIN

Wir wollen die Kunden mit ihren ggf. vorhandenen Bestellungen wissen:

```
mysql> SELECT
    ->  k.kunde_id, k.nachname, k.vorname, b.datum
    ->  FROM
    ->   bestellung b INNER JOIN kunde k USING(kunde_id)
    ->  ORDER BY
    ->   k.nachname, k.vorname;
+----------+-----------+---------+---------------------+
| kunde_id | nachname  | vorname | datum               |
+----------+-----------+---------+---------------------+
|        2 | Beutlin   | Frodo   | 2012-03-23 16:11:00 |
|        1 | Gamdschie | Samweis | 2012-03-24 17:41:00 |
+----------+-----------+---------+---------------------+
```

Aufgabe 11.16: Es werden zwar alle Bestellungen ausgegeben, aber nicht alle Kunden. Warum?

Möchte Sie aber alle Kunden sehen, auch wenn diese keine Bestellungen aufgegeben haben, können Sie keinen `INNER JOIN` verwenden; dazu wird ein `OUTER JOIN`, hier ein `RIGHT OUTER JOIN`, notwendig sein.

```
mysql> SELECT
    ->  k.kunde_id, k.nachname, k.vorname, b.datum
    ->  FROM
    ->   bestellung b RIGHT OUTER JOIN kunde k USING(kunde_id)
    ->  ORDER BY
    ->   k.nachname, k.vorname;
+----------+-------------+---------+---------------------+
```

```
| kunde_id | nachname     | vorname | datum               |
+----------+--------------+---------+---------------------+
|        3 | Beutlin      | Bilbo   | NULL                |
|        2 | Beutlin      | Frodo   | 2012-03-23 16:11:00 |
|        5 | Earendilionn | Elrond  | NULL                |
|        6 | Eichenschild | Thorin  | NULL                |
|        1 | Gamdschie    | Samweis | 2012-03-24 17:41:00 |
|        4 | Telcontar    | Elessar | NULL                |
+----------+--------------+---------+---------------------+
```

Zuerst fällt auf, dass nicht zwei, sondern sechs Zeilen ausgegeben werden, denn jetzt werden auch die Kunden angezeigt, für die keine Bestellungen vorliegen (Zeilen 10, 12, 13 und 15). Da SQL nicht weiß, was es in den entsprechenden Spalten an Werten eintragen soll, wird hier `NULL` verwendet.

Definition 45: OUTER JOIN
Der *OUTER JOIN* zweier Tabellen ist der INNER JOIN dieser beiden Tabellen, der um folgende Zeilen erweitert wird: Zeilen der rechten (*RIGHT OUTER JOIN*) oder linken (*LEFT OUTER JOIN*) Tabelle, für welche keine passenden Paarung gefunden wurde.

Wird der INNER JOIN um Zeilen aus der linken und der rechten Tabelle erweitert, spricht man von einem *FULL OUTER JOIN*.

Keine Sorge, die Sache ist komplizierter zu erklären als zu benutzen. Betrachten wir noch einmal obiges Beispiel. Der `INNER JOIN` liefert uns nur die Zeilen, für welche ein passendes Primär-Fremdschlüsselpaar gefunden wird. Der `RIGHT JOIN` in Zeile 4 erweitert jetzt das Ergebnis des `INNER JOIN`. Um welche Zeilen? Laut Definition 45 um die Zeilen der rechts vom `JOIN` stehenden Tabelle, für die keine passenden Primär-Fremdschlüsselpaare gefunden werden. Und tatsächlich, es sind dies die Kunden ohne Bestellung; für diese Primärschlüsselwerte kann kein passender Fremdschlüsselwert und `bestellung` gefunden werden.

Und das mit dem LEFT und RIGHT ist einfach nur banal. Bei LEFT wird um die Zeilen der Tabelle, die links vom Wort `JOIN` steht, erweitert. Bei RIGHT um die Tabelle, die rechts vom Wort `JOIN` steht. Vertauschen Sie die beiden Tabellennamen und machen aus `RIGHT` ein `LEFT`, kommt genau das gleiche Ergebnis heraus:

```
1  SELECT
2   k.kunde_id, k.nachname, k.vorname, b.datum
3   FROM
4    kunde k LEFT OUTER JOIN bestellung b USING(kunde_id)
5   ORDER BY
6    k.nachname, k.vorname;
```

Ein `LEFT OUTER JOIN` oder `RIGHT OUTER JOIN` wird immer dann verwendet, wenn nicht nur die Zeilen einer Tabelle interessant sind, für die es eine Paarung gibt. Nehmen Sie beispielsweise eine Liste von Vertretern und die von den Vertretern abgeschlossenen Verträge. Wollten Sie nun die Anzahl der abgeschlossenen Verträge pro Vertreter wissen, würde ein `INNER JOIN` alle Vertreter unterdrücken, die noch keinen Vertrag abgeschlossen haben. In einer Übersichtsauswertung wäre dies sicherlich fehlerhaft.

Hurra, wir haben eine neue Variante des `SELECT`:

SQL:2016, PostgreSQL, T-SQL

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabname [[RIGHT OUTER|LEFT OUTER|FULL OUTER|INNER] JOIN tabname
      ON tabname.spaltenname = tabname.spaltenname]*
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
;
```

MySQL und MariaDB kennen keinen `FULL OUTER JOIN`. Bei beiden ist das Schlüsselwort `OUTER` optional, und es gilt: Wird nur `JOIN` angegeben, wird ein `INNER JOIN` verwendet.

MySQL/MariaDB

```
SELECT [DISTINCT]
  {*|spaltenliste|ausdruck}
  FROM
    tabname [[RIGHT [OUTER]|LEFT [OUTER]|INNER] JOIN tabname
      {ON tabname.spaltenname = tabname.spaltenname|USING(spaltenname)}]*
  [WHERE bedingung]
  [ORDER BY spaltenname [ASC|DESC] [,spaltenname [ASC|DESC]]*]
  [LIMIT [offset,] anzahl]
  [INTO OUTFILE 'dateiname' exportoptionen]
;
```

Ein weiteres Beispiel: Wir möchten die Lieferanten[8] und ihre Artikel wissen.

```
mysql> SELECT
    ->  l.lieferant_id, l.firmenname, a.artikel_id, a.bezeichnung
    ->  FROM
    ->   artikel_nm_lieferant nm INNER JOIN artikel a   USING(artikel_id)
    ->                           INNER JOIN lieferant l USING(lieferant_id)
    ->  ORDER BY
    ->   firmenname;
+-------------+------------------------+------------+--------------+
| lieferant_id | firmenname             | artikel_id | bezeichnung  |
+-------------+------------------------+------------+--------------+
|            3 | Bürohengst GmbH        |       3001 | Papier (100) |
|            3 | Bürohengst GmbH        |       3005 | Tinte (gold) |
|            3 | Bürohengst GmbH        |       3006 | Tinte (rot)  |
|            3 | Bürohengst GmbH        |       3007 | Tinte (blau) |
|            3 | Bürohengst GmbH        |       3010 | Feder        |
|            1 | Gartenbedarf AllesGrün |       7856 | Silberwiebel |
|            1 | Gartenbedarf AllesGrün |       7863 | Tulpenzwiebel |
|            1 | Gartenbedarf AllesGrün |       9010 | Schaufel     |
|            1 | Gartenbedarf AllesGrün |       9015 | Spaten       |
+-------------+------------------------+------------+--------------+
```

Altes Problem: Wir sehen nur die Lieferanten, die aktuell Ware liefern. Wir wollen aber alle Lieferanten ausgegeben bekommen:

8 In der Datei `listing08.sql` werden diese angelegt.

```
mysql> SELECT
    ->  l.lieferant_id, l.firmenname, a.artikel_id, a.bezeichnung
    ->  FROM
    ->  artikel_nm_lieferant nm INNER JOIN artikel a   USING(artikel_id)
    ->                          RIGHT JOIN lieferant l USING(lieferant_id)
    ->  ORDER BY
    ->   firmenname;
+--------------+------------------------+------------+---------------+
| lieferant_id | firmenname             | artikel_id | bezeichnung   |
+--------------+------------------------+------------+---------------+
|            3 | Bürohengst GmbH        |       3001 | Papier (100)  |
|            3 | Bürohengst GmbH        |       3005 | Tinte (gold)  |
|            3 | Bürohengst GmbH        |       3006 | Tinte (rot)   |
|            3 | Bürohengst GmbH        |       3007 | Tinte (blau)  |
|            3 | Bürohengst GmbH        |       3010 | Feder         |
|            1 | Gartenbedarf AllesGrün |       7856 | Silberwiebel  |
|            1 | Gartenbedarf AllesGrün |       7863 | Tulpenzwiebel |
|            1 | Gartenbedarf AllesGrün |       9010 | Schaufel      |
|            1 | Gartenbedarf AllesGrün |       9015 | Spaten        |
|            2 | Office International   |       NULL | NULL          |
+--------------+------------------------+------------+---------------+
```

In Zeile 20 wird jetzt die Firma ausgegeben, welche keinen Artikel beliefert.

Umgekehrt kann der `OUTER JOIN` dafür verwendet werden, gerade die Datensätze herauszufiltern, für welche keine passenden Paarungen gefunden werden. Wir wollen alle Lieferanten wissen, die keine Ware liefern:

```
mysql> SELECT l.firmenname
    -> FROM
    ->  artikel_nm_lieferant nm INNER JOIN artikel a USING(artikel_id)
    ->                          RIGHT JOIN lieferant l USING(lieferant_id)
    ->  WHERE artikel_id IS NULL;
+----------------------+
| firmenname           |
+----------------------+
| Office International |
+----------------------+
```

Durch das `IS NULL` in Zeile 5 werden alle Zeilen aus der Ergebnismenge entfernt, die eine Artikelnummer haben. Übrig bleiben die, für welche keine Artikelnummer gefunden wird.

Sie wollen wissen, welche Kunden bisher noch nichts bestellt haben? Kein Problem:

```
mysql> SELECT k.kunde_id, k.nachname, k.vorname
    ->  FROM
    ->   bestellung b RIGHT JOIN kunde k USING(kunde_id)
    ->  WHERE b.bestellung_id IS NULL;
+----------+--------------+---------+
| kunde_id | nachname     | vorname |
+----------+--------------+---------+
|        3 | Beutlin      | Bilbo   |
|        5 | Earendilionn | Elrond  |
|        6 | Eichenschild | Thorin  |
|        4 | Telcontar    | Elessar |
+----------+--------------+---------+
```

Aufgabe 11.17: Welche Kunden haben keine eigene Lieferadresse?

Aufgabe 11.18: Welcher Artikel ist noch nie bestellt worden?

Hinweis: Ist der OUTER JOIN Teil einer Kette von JOINs, müssen Sie darauf achten, dass in der Kette das Ergebnis des OUTER JOINs nicht wieder durch einen INNER JOIN verloren geht.

Aufgabe 11.19: Warum verschwindet hier die Firma `Office International`?

```
SELECT DISTINCT l.firmenname
 FROM
 artikel_nm_lieferant nm INNER JOIN artikel a USING(artikel_id)
                         RIGHT JOIN lieferant l USING(lieferant_id)
                         INNER JOIN bestellung_position USING(artikel_id)
;
```

Aufgabe 11.20: Unter der Annahme, dass Sie keine Constraints verwenden: Wie kann man mit einem OUTER JOIN verletzte referenzielle Integritäten (siehe Definition 22 auf Seite 32) ermitteln?

Zum Schluss noch ein Beispiel für die `FULL OUTER JOIN`-Syntax, wie sie in PostgreSQL angewendet werden kann:

```
oshop=# SELECT
oshop-#  kunde_id, nachname, vorname, bestellung_id, datum
oshop-#  FROM
oshop-#   bestellung FULL OUTER JOIN kunde USING(kunde_id)
oshop-# ;

kunde_id |   nachname   | vorname | bestellung_id |        datum
---------+--------------+---------+---------------+---------------------
       1 | Gamdschie    | Samweis |             1 | 2012-03-24 17:41:00
       2 | Beutlin      | Frodo   |             2 | 2012-03-23 16:11:00
       5 | Earendilionn | Elrond  |               |
       6 | Eichenschild | Thorin  |               |
       4 | Telcontar    | Elessar |               |
       3 | Beutlin      | Bilbo   |               |
```

Mangels fachlich sinnvollem Beispiel ist dies hier das gleiche Ergebnis wie bei einem `RIGHT OUTER JOIN`.

11.5 Narzissmus pur: SELF JOIN

Wir wollen den Kunden die Möglichkeit bieten, in einem Forum Beiträge zu formulieren. Mit unseren bisherigen Tabellen ist das nicht möglich. Ein kurzes Brainstorming zeigt uns, dass wir eine Tabelle mit Anmeldedaten und eine Tabelle mit Beiträgen brauchen (siehe ER-Modell in Bild 11.3 auf der nächsten Seite).

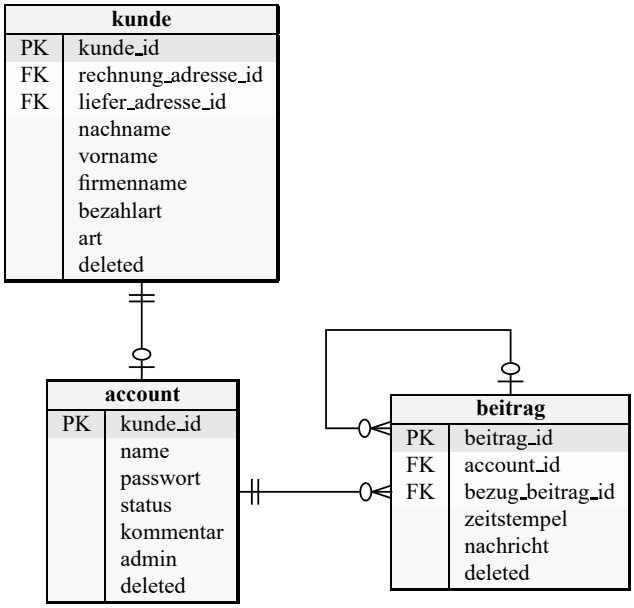


Bild 11.3 ER-Modell: Kundenforum

Zunächst die einfachen Dinge: Jeder Kunde kann einen Account haben, muss er aber nicht. Umgekehrt muss aber ein Account auf einen Kunden zeigen. Wir haben also eine *1:1*-Verknüpfung.

Jeder Account kann mehrere Beiträge erfassen, muss er aber nicht. Ein Beitrag hingegen muss auf genau einen Account verweisen: eine *1:n*-Verknüpfung.

In Foren ist es üblich, dass man auf Beiträge antworten kann. Dazu muss ein Beitrag wissen, auf welchen anderen Beitrag er sich bezieht. Deshalb gibt es in der Tabelle `beitrag` den Fremdschlüssel `bezug_beitrag_id`. In diesen Fremdschlüssel trage ich den Wert von `beitrag_id` ein, den die ursprüngliche Nachricht hat.

Aus Sicht der Tabelle ist der Fremdschlüssel `bezug_beitrag_id` ein Fremdschlüssel, der auf eigene Datensätze verweist. Eine solche Verknüpfung nennt man SELF JOIN.

Definition 46: SELF JOIN
Enthält eine Tabelle einen Fremdschlüssel mit Primärschlüsselwerten der eigenen Tabelle, so nennt man diese Art der Verknüpfung *SELF JOIN*.

Bitte beachten Sie, dass ein SELF JOIN auch ein `INNER JOIN`, `OUTER JOIN` oder `CROSS JOIN` sein kann!

In Bild 11.3 können Sie sehen, wie man einen SELF JOIN sofort erkennt. Er verweist eben auf sich selbst.

Aufgabe 11.21: Erstellen Sie zu den beiden Tabellen `account` und `beitrag` passende `CREATE TABLE`- und ggf. `CREATE INDEX`-Befehle. Füllen Sie die beiden Tabellen mit passenden Inhalten. Beachten Sie dabei folgende Hinweise:

- `account`: Der Primärschlüssel `kunde_id` hat keinen eigenen Zähler.
- `account`: Der Inhalt der Spalte `name` muss ein Schlüssel sein.
- `account`: Der Status kann zwei Werte haben: `aktiv` und `gesperrt`.
- `account`: Der Kommentar muss einen langen Text aufnehmen können.
- `account`: Die Spalte `admin` ist vom Typ `BOOL`.
- `beitrag`: Die Spalte `account_id` ist der Fremdschlüssel auf die Spalte `kunde_id` in der Tabelle `account`.
- `beitrag`: Der Selbstbezug soll den Default 1 haben. Wir werden sicherstellen müssen, dass es eine leere Nachricht mit dem Primärschlüsselwert 1 geben wird. Auf diesen werden alle Nachrichten verweisen, die keine Antworten auf eine andere Nachricht sind.
- `beitrag`: Der Nachrichtentext muss Platz für längere Texte haben.
- `beitrag`: Auf eine Thread-Verwaltung wird hier verzichtet.

Nun stehen uns Testdaten[9] zur Verfügung, deren Nachrichtentexte ich aus Platzmangel auf 20 Stellen in der Ausgabe begrenze:

```
mysql> SELECT
    ->  kunde_id, name, status, admin
    ->  FROM
    ->   account;
+----------+-------+--------+-------+
| kunde_id | name  | status | admin |
+----------+-------+--------+-------+
|        1 | admin | aktiv  |     1 |
|        2 | frodo | aktiv  |     0 |
|        3 | bilbo | aktiv  |     0 |
|        5 | elle  | aktiv  |     0 |
+----------+-------+--------+-------+

mysql> SELECT
    ->  beitrag_id, account_id, bezug_beitrag_id, LEFT(nachricht, 20)
    ->  FROM
    ->   beitrag;
+------------+------------+------------------+----------------------+
| beitrag_id | account_id | bezug_beitrag_id | LEFT(nachricht, 20)  |
+------------+------------+------------------+----------------------+
|          1 |          1 |                1 |                      |
|          2 |          2 |                1 | Der Lieferservice is |
|          3 |          3 |                2 | Das finde ich auch.  |
|          4 |          5 |                2 | Aber ein wenig langs |
|          5 |          2 |                4 | Finde ich nicht.     |
|          6 |          5 |                1 | Angebot könnte besse |
+------------+------------+------------------+----------------------+
```

Jetzt kommt der SELF JOIN: Wir wollen die Antworten auf Nachricht 2 wissen:

```
mysql> SELECT nachricht
    ->  FROM
    ->   beitrag INNER JOIN beitrag ON bezug_beitrag_id = beitrag_id
```

[9] Die entsprechenden Befehle stehen in `listing08.sql`.

```
4      ->   WHERE
5      ->    beitrag_id = 2;
6  ERROR 1066 (42000): Not unique table/alias: 'beitrag'
```

Die Fehlermeldung in Zeile 6 besagt, dass die Tabelle `beitrag` nicht eindeutig ist. Klar, die Tabelle kommt in der Verknüpfung ja auch zweimal vor. Für SQL ist das ein Problem. Dieses wird besonders in Zeile 3 deutlich. Wenn er die beiden Spalteninhalte vergleichen soll, ist unklar, ob mit `beitrag_id` jetzt die Spalte der linken oder rechten Tabelle `beitrag` gemeint ist.

Spätestens jetzt ist die Vergabe eines Alias kein *nice to have* mehr, sondern ein *must be*. Indem man der Tabelle jeweils einen eindeutigen Alias – links `ant` für `beitrag` in der Rolle einer Antwort und rechts `orig` für `beitrag` in der Rolle der Originalnachricht – gibt und diesen bei der Angabe der Spalten auch verwendet, sind alle Unklarheiten beseitigt.

```
1  mysql> SELECT ant.nachricht 'Antwort'
2      ->  FROM
3      ->   beitrag ant INNER JOIN beitrag orig
4      ->    ON ant.bezug_beitrag_id = orig.beitrag_id
5      ->   WHERE
6      ->    orig.beitrag_id = 2;
7  +-------------------------+
8  | Antwort                 |
9  +-------------------------+
10 | Das finde ich auch.     |
11 | Aber ein wenig langsam. |
12 +-------------------------+
```

Es sei bemerkt, dass es keinen eigenen Befehl `SELF JOIN` gibt. Man verwendet dazu einfach einen `INNER JOIN` oder eine andere `JOIN`-Variante, die auf beiden Seiten die gleiche Tabelle stehen hat.

Mithilfe der *common table expression*[10] wird eine virtuelle Ergebnismenge (Tabelle) aufgebaut, die sich auch selbst als Datenquelle verwenden kann. Hier ein Quelltextbeispiel mit `WITH RECURSIVE`:

```
1  mysql> WITH RECURSIVE ant_auf AS
2  -> (
3  ->  -- nicht rekuriver Teil
4  ->  SELECT *
5  ->   FROM beitrag
6  ->   WHERE bezug_beitrag_id = 2
7  ->  UNION ALL
8  ->  -- rekuriver Teil
9  ->  SELECT ant.*
10 ->   FROM
11 ->     beitrag ant INNER JOIN ant_auf orig
12 ->      ON ant.bezug_beitrag_id = orig.beitrag_id
13 -> )
14 -> SELECT beitrag_id, bezug_beitrag_id, nachricht FROM ant_auf;
15 +------------+-----------------+-------------------------+
16 | beitrag_id | bezug_beitrag_id | nachricht               |
17 +------------+-----------------+-------------------------+
18 |          3 |                2 | Das finde ich auch.     |
```

[10] Eine nähere Betrachtung von CTE muss hier leider aus Platzgründen entfallen. Eine gute Einführung finden Sie unter [MyS19].

```
|          4 |                2 | Aber ein wenig langsam. |
|          5 |                4 | Finde ich nicht.        |
+------------+------------------+-------------------------+
```

Ihnen fällt sicherlich auf, dass nun auch der Beitrag 5 als indirekte Antwort auf Beitrag 2 ausgegeben wird.

11.6 Eine Verknüpfung beschleunigen

Sind die Daten auf mehrere Tabellen verteilt, kann eine Verknüpfung mithilfe von Indizes beschleunigt werden. Ein Primärschlüssel hat automatisch einen passenden Index. Das Gleiche gilt für Fremdschlüssel, wenn sie denn durch `FOREIGN KEY ... REFERENCES` deklariert sind.

Anders sieht es bei Verknüpfungen aus, die nicht über indizierte Spalten durchgeführt werden. Denken Sie beispielsweise an den Zusammenbau der letzten temporären Tabelle auf Seite 193. Beide verwendeten die Spalte `kunde_id`, aber diese war in den temporären Tabellen nicht als Primär- oder Fremdschlüssel deklariert. Eine Verknüpfung über diese beiden Spalten kann somit sehr teuer werden.

Grundsätzlich ist aber der Zusammenbau von Daten aus verschiedenen Tabellen teurer als das Auslesen der Daten aus einer Tabelle. Mit temporären Tabellen – wie oben beschrieben – können Sie übliche Verbindungen vorbauen, damit diese nicht jedes Mal neu erstellt werden müssen. Eine weitere Möglichkeit sind redundante Daten.

Redundante Daten sind nach Definition 13 auf Seite 21 Daten, die mehrfach im System abgespeichert sind. Grundsätzlich versucht man, redundante Daten zu vermeiden. Neben dem Speicherplatzverbrauch muss man Daten an vielen Orten aktualisieren, was fehlerträchtig ist.

Aber redundante Daten können auch sinnvoll sein. Wir könnten die Tabelle `kunde` um die Spalten für die Rechnungs- und Lieferadresse erweitern. Ebenso um zwei Spalten, die mir markieren, ob die beiden Adressen noch aktuell sind.

```
ALTER TABLE kunde
 ADD r_strasse VARCHAR(255),
 ADD r_ort VARCHAR(255),
 ADD r_aktuell BOOL NOT NULL DEFAULT TRUE,
 ADD l_strasse VARCHAR(255),
 ADD l_ort VARCHAR(255),
 ADD l_aktuell BOOL NOT NULL DEFAULT TRUE
;
```

Straße und Ort sollen Zusammenbauten der Spalten `strasse` mit `hnr` und `lkz` mit `plz` mit `ort` sein.[11] In periodischen Abständen werden die Daten aus der Adresstabelle in die Kundentabelle kopiert.

```
mysql> UPDATE kunde INNER JOIN adresse ON rechnung_adresse_id = adresse_id
    ->  SET
    ->   r_strasse = CONCAT(strasse, ' ', hnr),
```

[11] Eine von mir willkürlich gefällte Designentscheidung, um eine bessere Übersicht zu haben.

```
    ->   r_ort = CONCAT(lkz, '-', plz, ' ', ort),
    ->   r_aktuell = TRUE;

mysql> SELECT
    ->  kunde_id, r_strasse, r_ort, r_aktuell
    ->  FROM kunde;

+----------+------------------+----------------------+-----------+
| kunde_id | r_strasse        | r_ort                | r_aktuell |
+----------+------------------+----------------------+-----------+
|        1 | Beutelhaldenweg 5 | AL-67676 Hobbingen   |         1 |
|        2 | Beutelhaldenweg 1 | AL-67676 Hobbingen   |         1 |
|        3 | Beutelhaldenweg 1 | AL-67676 Hobbingen   |         1 |
|        4 | Auf der Feste 1   | GO-54786 Minas Tirith |         1 |
|        5 | Letztes Haus 4    | ER-87567 Bruchtal    |         1 |
|        6 | NULL              | NULL                 |         1 |
+----------+------------------+----------------------+-----------+
```

Das ist scharf, oder? Der `INNER JOIN` wird gar nicht in einem `SELECT` verwendet, sondern in einem `UPDATE`! Erinnern Sie sich? Das Ergebnis einer Verknüpfung ist wieder eine Tabelle. Deshalb können Sie an vielen Stellen, wo in der SQL-Referenz der Tabellenname steht, eine Verknüpfung einsetzen.

Aufgabe 11.22: Überlegen Sie mal, lässt sich beim `UPDATE` eine geschickte `WHERE`-Klausel einbauen?

Jetzt wird jedes Mal, wenn sich eine Adresse innerhalb der Periode ändert, die Spalte `r_aktuell` oder `l_aktuell` auf `FALSE` gesetzt. Dies könnten Sie beispielsweise mit einem Trigger[12] erreichen. Mithilfe eines Events[13] wird jetzt in periodischen Abständen nachgeschaut, ob sich die Adressdaten geändert haben. Falls ja, werden die redundanten Daten neu aufgebaut. Falls Sie die Änderung sofort in den redundanten Daten ändern wollen, können Sie dies ebenfalls im Trigger machen.

Aufgabe 11.23: Erweitern Sie das `UPDATE` so, dass gleichzeitig auch die Lieferadresse gesetzt wird. Vorsicht beim zweiten JOIN und der WHERE-Klausel!

[12] Siehe Kapitel 22 auf Seite 357.
[13] Siehe Kapitel 23 auf Seite 365.