

SQL als Datenbankabfragesprache

SQL als Datenbankabfragesprache	1
Möglichkeiten der Datenmanipulationsanweisungen	2
Grundform der SELECT-Anweisung.....	3
Einfache Abfragen	6
Datenauswahl mit Bedingungen	7
Testen auf NULL	8
Der Operator BETWEEN.....	9
Der Operator IN	9
Der Operator LIKE	9
Einfache Unterabfragen	10
Unterabfragen und der IN-Operator	12
Die Operatoren ANY und ALL.....	12
Der Operator EXISTS.....	13
Aggregatfunktionen	15
Gruppierungen (GROUP BY).....	18
Kreuztabellen erstellen	21
Auswahl bestimmter Gruppen (HAVING)	22
Ergebnistabelle ordnen (ORDER BY).....	23
Anzahl der Ergebnisdatensätze einschränken (LIMIT)	24
Datensätze zufällig auswählen (RAND)	25
Mengenoperationen (UNION)	26
Verbund von Tabellen.....	26
Kartesisches Produkt	28
Gleichverbund (Equijoin).....	28
Natürlicher Verbund (Natural Join)	29
Theta-Verbund (Theta-Join).....	30
Der Verbund von mehr als zwei Tabellen	31
Selbstverbund (Selfjoin)	32
Unterabfragen und Verbund	33
Inner- und Outerjoins.....	33
Korrelierte Unterabfragen	35

Inhalt

SQL steht für *Structured Query Language*. Diese Sprache wird in erster Linie dazu verwendet, um Abfragen, Einfüge-, Änderungs- und Löschkommandos an das Datenbanksystem zu formulieren.

Die Anweisungen **SELECT**, **INSERT**, **UPDATE** und **DELETE** gehören zu den *Datenmanipulationsanweisungen* (engl. *Data Manipulation Language DML*).

SQL

SQL enthält aber auch die Anweisungen **CREATE**, **ALTER**, **DROP**, usw. welche dazu dienen Datenbanken, Tabelle und Sichten zu erstellen, ihre Definition zu ändern und diese wiederum zu löschen (engl. *Data Definition Language DDL*).

Zu SQL gehören auch die Anweisungen **GRANT**, **REVOKE**, **COMMIT**, **ROLLBACK**, usw. durch welche Sicherheitsmechanismen eingestellt werden können (engl. *Data Control Language DCL*).

Möglichkeiten der Datenmanipulationsanweisungen

INSERT

Mit **INSERT** können neue Datensätze in eine Tabelle aufgenommen werden:

```
INSERT INTO <Tabelle>(<Spaltenliste>)
VALUES (<werteliste>) [, (<werteliste>)];
```

Nach dem Namen der Tabelle muss zuerst eine Liste der Spaltennamen und danach eine Liste mit den einzufügenden Werten übergeben werden. Spalten mit Standardwerten, Spalten, die **NULL** sein dürfen, sowie autoinkrementierende Spalten müssen nicht angegeben werden.



```
INSERT INTO artikel(hnummer, rbezeichnung, repreis, rvpreis,
rbestand, rsollbestand)
VALUES (10, "NXJ-2389-9", 230.60, 349.90, 150, 150);
```

Wird einer Spalte vom Typ **TIMESTAMP** eine **NULL** übergeben, so speichert das Datenbanksystem in dieser Spalte das aktuelle Datum und die Uhrzeit. Handelt es sich um eine autoinkrementierende Spalte und dieser wird **NULL** übergeben, so bildet das Datenbanksystem für diese Spalte den fortlaufenden numerischen Wert.



Bei den nachfolgenden am Computer durchzuführenden Übungen sollen Sie die Auswirkungen der durchgeführten Befehle und insbesondere die Richtigkeit des erzielten Ergebnisses kontrollieren, indem Sie die Inhalte der Tabellen betrachten und gegebenenfalls weitere Anweisungen formulieren, welche das Ergebnis verifizieren lassen.



1. Importieren Sie in die Datenbank auftragsverwaltung des letzten Kapitels der Reihe nach folgende bereitgestellten CSV-Dateien (**TIPP:**  **Textdatei in Tabelle einfügen**):
kunden.csv, hersteller.csv, mitarbeiter.csv, auftraege.csv, artikel.csv, positionen.csv und arbeitszeiten.csv
2. Fügen Sie für die Mitarbeiterin Frau Dorner folgende zusätzliche Arbeitszeiten ein:
Am 2.5.2010 von 7:34 bis 12:30, am 3.5.2010 von 7:45 bis 13:00 und am 4.5.2010 von 9:00 bis 16:23 Uhr. Verwenden Sie dazu eine einzige SQL-Anweisung.
3. Nehmen Sie Herrn Georg Goller als neuen Mitarbeiter in die Datenbank auf. Welche Mitarbeiternummer erhält er?

UPDATE

Mit **UPDATE** können Sie einzelne Felder eines schon vorhandenen Datensatzes ändern:

```
UPDATE <Tabelle>
SET <Spaltenname>=<wert> [, <Spaltenname>=<wert>]
[WHERE <Bedingung>];
```



```
UPDATE artikel
SET rvpreis = rvpreis * 1.05;
```

Die Verkaufspreise aller (!) Artikel werden um 5% erhöht.

```
UPDATE artikel
SET rbestand = 0
ORDER BY rvpreis DESC
LIMIT 10;
```

Der Lagerbestand der zehn teuersten Artikel wird auf 0 gebracht.

```
UPDATE artikel, hersteller
SET artikel.hnummer = hersteller.hnummer
WHERE artikel.rnummer = 1
AND hersteller.hname = "Magnat";
```

Vom Artikel mit der Nummer 1 wird der Hersteller geändert. Jetzt wird dieser Artikel vom Hersteller mit dem Namen Magnat produziert.

1. Ändern Sie von der Mitarbeiterin Frau Dorner vom 2.5.2010 das Arbeitsende auf 13:30 Uhr ab.
2. Erhöhen Sie den Sollbestand aller Artikel, die vom Hersteller Panasonic stammen, um 50 Stück.
3. Erhöhen Sie den Verkaufspreis der 20 billigsten Artikel um 50%.



Das syntaktisch einfachste Kommando dieses Abschnitts ist **DELETE**:

```
DELETE FROM <Tabelle>
[WHERE <Bedingung>];
```

Alle ausgewählten Datensätze werden gelöscht. Dabei wird die Tabelle selbst nicht gelöscht.

```
DELETE FROM artikel;
```

Alle Artikel aus der Tabelle werden gelöscht.

```
DELETE positionen, auftraege
FROM positionen, auftraege, kunden
WHERE kfname = "Ahnert" AND kname = "Lisa"
AND kunden.knummer = auftraege.knummer
AND auftraege.anummer = positionen.anummer;
```



Alle Aufträge des Kunden Ahnert Lisa mitsamt ihren Auftragspositionen werden gelöscht. Dabei werden Löschungen nur in jenen Tabellen vorgenommen, welche nach **DELETE** stehen. Die Kundin selbst wird nicht gelöscht (**ANMERKUNG**: Diese Operation kann nur dann durchgeführt werden, wenn durch das Löschen die referentiellen Integritäten nicht verletzt werden).

1. Löschen Sie alle Arbeitszeiten der Mitarbeiterin Frau Dorner. Die Mitarbeiterin selbst soll nicht gelöscht werden.
2. Versuchen Sie alle Artikel des Herstellers Revox zu löschen. Wieso funktioniert dies nicht.
3. Löschen Sie dann in den Auftragspositionen jene Artikel die vom Hersteller Revox stammen.
4. Löschen Sie aus der Artikeltabelle die Artikel der Herstellers Revox.



Grundform der SELECT-Anweisung

Mit der **SELECT**-Anweisung werden die Datenwerte aus einer Datenbank ausgewählt. Sie können aus einer oder aus mehreren, miteinander verbundenen Tabellen oder Sichten ausgewählt werden. Das Ergebnis einer solchen Auswahl ist erneut eine Tabelle, die keine, eine oder mehrere Zeilen und eine oder mehrere Spalten hat.

```
SELECT <Klausel>
FROM <Klausel>
[WHERE <Klausel>]
[GROUP BY-<Klausel>]
[HAVING-<Klausel>]
[ORDER BY <Klausel>]
[LIMIT <Klausel>];
```

Grundform

- SELECT-Klausel** In der **SELECT**-Klausel werden die Tabellenspalten, Aggregatfunktionen und Ausdrücke festgelegt, die in die Ergebnistabelle der **SELECT**-Anweisung aufgenommen werden sollen.
- ```
SELECT [ALL|DISTINCT] {*|<Spaltenliste>}
```
- Die in der Spaltenliste aufgeführten Spaltennamen müssen durch Kommas voneinander getrennt sein. Alle Spaltennamen der **SELECT**-Klausel müssen zu Sichten oder Tabellen gehören, die in der **FROM**-Klausel aufgeführt wurden. Spaltennamen dürfen in derselben Anweisung nur einmal vorkommen. Enthalten zwei Tabellen identische Spaltennamen, dann spricht man diese Spalten in der Form <Tabellenname>.<Spaltenname> an.
- Stern \*** Mit einem \* anstelle einer Spaltenliste wird festgelegt, dass alle Spalten der in der **FROM**-Klausel aufgeführten Tabelle in die Ergebnistabelle aufgenommen werden sollen. Der Stern \* kann auch in der Form <Tabellenname>.\* benutzt werden, um alle Spalten der entsprechenden Tabelle in die Ergebnistabelle aufzunehmen.
- Ausdrücke dürfen aus Konstanten, Funktionen (**YEAR**, **MONTH**, **UPPER**, **CONCAT**, **LENGTH**, usw.) und Aggregatfunktionen (**MIN**, **MAX**, **SUM**, **COUNT**, **AVG**) bestehen.
- ALL** Die Angabe **ALL** kann optional benutzt werden, um alle Zeilen auszuwählen (**ALL** ist Standardvorgabe).
- DISTINCT** Mit Hilfe der Option **DISTINCT** wird festgelegt, dass in die Ergebnistabelle von den Zeilen, die dieselben Werte aufweisen, immer nur eine aufgenommen wird. **DISTINCT** kann auch für die Aggregatfunktionen benutzt werden um mehrfache identische Werte zu unterdrücken.
- FROM-Klausel** Die **FROM**-Klausel ist unbedingt erforderlich, denn in ihr werden die Tabellen festgelegt, deren Daten in der Ergebnistabelle zu verwenden sind.
- ```
FROM <Tabellenname>|<Sichtname> [Aliasname]
[,<Tabellenname>|<Sichtname> [Aliasname]...]
```
- Die Tabellenliste besteht aus einer oder mehreren Namen von Tabellen oder Sichten. In der **FROM**-Klausel können auch Aliasnamen für Tabellen oder Sichten, die in der Klausel aufgeführt sind, definiert werden. Ein Aliasname ist ein weiterer Name, der den Tabellen- oder Sichtnamen in einer korrelierten Unterabfrage oder in einem Selbstverbund (engl. Selfjoin) ersetzt (siehe hinten).
- WHERE-Klausel** Die **WHERE**-Klausel ist eine optionale Klausel, mit der eine Auswahlbedingung festgelegt wird. Damit wird bestimmt, welche Zeilen in die Ergebnistabelle aufgenommen werden sollen.
- ```
WHERE <Bedingung>
```
- In die Ergebnistabelle werden nur jene Zeilen einer in der **FROM**-Klausel angeführten Tabelle oder Sicht aufgenommen, welche die Bedingung der **WHERE**-Klausel erfüllen.
- GROUP BY** Die optionale **GROUP BY**-Klausel gruppiert die Zeilen anhand einer oder mehrerer Spalten. Dabei werden in der Ergebnistabelle alle Zeilen zusammengefasst, die in den Gruppiererspalten denselben Wert besitzen. Jede Gruppe wird zu einer Zeile zusammengefasst.
- ```
GROUP BY <Spalte>[,<Spalte>...];
```
- HAVING** Die optionale **HAVING**-Klausel wird mit **GROUP BY** verwendet, um bestimmte Gruppen für die Ergebnistabelle auszuwählen.
- ```
HAVING <Bedingung>
```

**ORDER BY**

Die optionale **ORDER BY**-Klausel dient dazu um die Reihenfolge der Zeilen in der Ergebnistabelle festzulegen. Die Reihenfolge kann aufsteigend (**ASC**) oder absteigend (**DESC**) sein.

```
ORDER BY <Spaltenname|Ganzzahl> [ASC|DESC]
[,<Spaltenname|Ganzzahl> [ASC|DESC]...]
```

Die **ORDER BY**-Klausel ordnet die Zeilen der Ergebnistabelle nach den Werten in den angegebenen Spalten.

Eine Ganzzahl beginnend bei 1 für die Position der Spalte (v.l.n.r.) kann in der **ORDER BY**-Klausel den Namen der Spalte ersetzen.

**LIMIT**

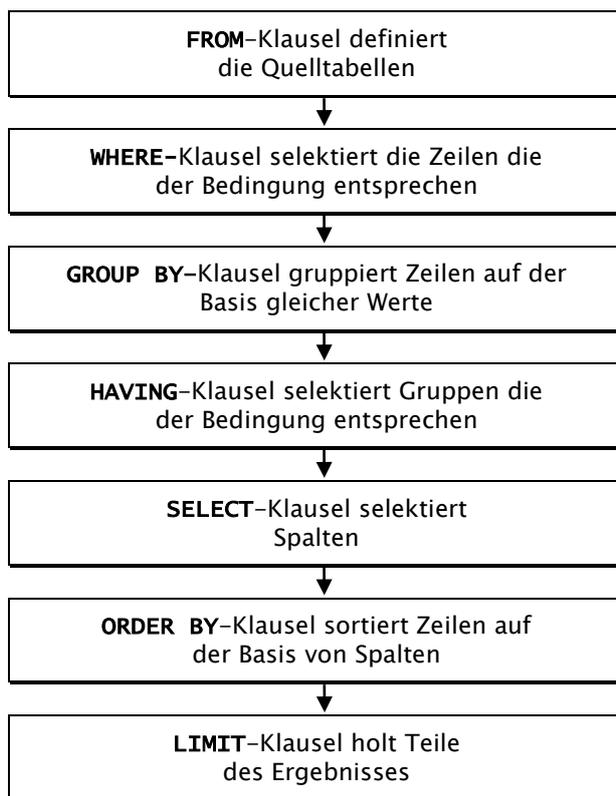
Mit der **LIMIT**-Klausel kann nach einem Teil der tatsächlichen Ergebnismenge gesucht werden.

```
LIMIT [<Nummer Startdatensatz>,<Anzahl Datensätze>
```

Wird nur **<Anzahl Datensätze>** angegeben, so liefert **LIMIT** die ersten **<Anzahl Datensätze>** zurück. **<Nummer Startdatensatz>** gibt jenen Datensatz an, ab welchem das Ergebnis zurückgeliefert wird (**ANMERKUNG:** Zählung beginnt bei 0 für ersten Datensatz). So liefert beispielsweise **LIMIT 5, 10** den sechsten bis fünfzehnten Datensatz zurück).

**Verarbeitung**

Die nachfolgende Abbildung zeigt, in welcher Reihenfolge die einzelnen Klauseln der **SELECT**-Anweisung abgearbeitet werden. Es ist auffallend, dass diese Reihenfolge von der abweicht, in der die Klauseln in der Anweisung erscheinen.



1. Aus mindestens wie vielen Klauseln besteht eine **SELECT**-Anweisung.
2. Kann eine **SELECT**-Anweisung eine **ORDER BY**-Klausel enthalten, ohne gleichzeitig eine **WHERE**-Klausel zu haben?
3. Kann eine **SELECT**-Anweisung eine **HAVING**-Klausel enthalten, ohne gleichzeitig eine **GROUP BY**-Klausel aufzuweisen?



## Einfache Abfragen

**Kalkulationsspalten** Innerhalb einer **SELECT**-Anweisung dürfen vor dem Wortsymbol **FROM** nicht nur Spaltennamen angegeben werden, sondern es lassen sich dort beliebige arithmetische, alphanumerische, Datums- oder Zeit-Ausdrücke aufführen.



Man zeige von der Tabelle `artikel` die Artikelnummern sowie das Produkt von `rvpreis` und `rbestand` an. Der Ergebnisspalte soll der Name `gesamt` gegeben werden. Es soll absteigend nach `gesamt` sortiert werden.

```
SELECT rnummer, rvpreis * rbestand AS gesamt
FROM artikel
ORDER BY gesamt DESC;
```

Man finde die unterschiedlichen Jahre in denen Mitarbeiter im Unternehmen gearbeitet haben.

```
SELECT DISTINCT YEAR(zdatum)
FROM arbeitszeiten;
```

Man finde alle Artikelbezeichnungen und kürze diese auf maximal 13 Zeichen. Dies aber nur dann, wenn die Bezeichnung mehr als 13 Zeichen lang ist. Beim Zusammenkürzen werden die ersten 5 Buchstaben der Bezeichnung, dann drei Punkte und zum Schluss die letzten 5 Buchstaben der Artikelbezeichnung ausgegeben.

```
SELECT
IF(CHARACTER_LENGTH(rbezeichnung) <= 13,
rbezeichnung,
CONCAT(SUBSTRING(rbezeichnung, 1, 5), "...",
SUBSTRING(
rbezeichnung, CHARACTER_LENGTH(rbezeichnung) - 4, 5)))
FROM artikel;
```

Man ermittle für alle Arbeitszeiten die Differenz zwischen Start- und Endezeit.

```
SELECT zvon,zbis,SUBTIME(zbis, zvon)
FROM arbeitszeiten;
```

**ANMERKUNG:** Mit **ADDTIME** können Zeitangaben addiert, mit **ADDDATE** kann zu einem Datum eine gewisse Anzahl von Tagen dazu gezählt werden und mit **DATEDIFF** Differenzen von Datumsangaben gebildet werden.

Man ermittle das Alter der Kunden in Jahren.

```
SELECT TRUNCATE(DATEDIFF(NOW(),kgebdatum) / 366,0)
FROM kunden;
```



Das Werkzeug *phpMyAdmin* interpretiert SQL-Kommandos eigenständig und fügt an manchen Stellen Leerzeichen ein. So geschieht dies beispielsweise bei manchen Funktionen nach dem Funktionsnamen und vor der Klammer. Diese wiederum versteht der MySQL-Server nicht. Deshalb muss mit folgendem Befehl der Server angewiesen werden, diese zu ignorieren:

```
SET GLOBAL sql_mode = "IGNORE_SPACE";
```



1. Löschen Sie die Inhalte aller Tabellen Ihrer Datenbank `auftragsverwaltung`. Achten Sie auf die Reihenfolge, in der Sie diese löschen. Importieren Sie die mitgelieferten CSV-Dateien neuerlich. Achten Sie dabei wiederum auf die Reihenfolge, wie Sie diese importieren.
2. Man finde die unterschiedlichen Monate an denen Mitarbeiter für das Unternehmen gearbeitet haben.
3. Man finde die Namen der ersten 10 Kunden und verbinde den Nach- und Vornamen des Kunden zu einem Datenfeld mit der Bezeichnung `name`. Dabei soll zwischen Nach- und Vorname ein Leerzeichen angezeigt werden.

4. Erweitern Sie die vorige Abfrage, so dass nach dem Kundennamen die Straße mit Hausnummer sowie die Postleitzahl und der Ort im Datenfeld mit dem Namen adresse ausgegeben werden. Dabei sollen wiederum Leerzeichen eingefügt und nach dem Namen und der Hausnummer eine Zeilenschaltung (\n) integriert werden.
5. Geben Sie die Namen der Hersteller und die Länge dieser Namen aus. Geben Sie der Spalte welche die Länge des Namens enthält den Namen laenge. Geben Sie nur jene 4 Namen aus, die am längsten sind.
6. Ermitteln Sie für die aktuellsten 10 Aufträge das Lieferdatum. Geliefert wird 10 Tage nach dem Auftragsdatum. Geben Sie Auftragsnummer, Auftrags- und Lieferdatum aus.

## Datenauswahl mit Bedingungen

In der **WHERE**-Klausel der **SELECT**-Anweisung wird mit Hilfe von Bedingungen angegeben, welche Zeilen in das Endergebnis aufgenommen werden sollen.

Gesucht sind alle unterschiedlichen Kundenorte die größer oder gleich H sind:

```
SELECT DISTINCT kort
FROM kunden
WHERE kort >= "H";
```



Hier werden Orte wie Hamburg, Hannover, Köln usw. ausgegeben. Bonn hingegen wird nicht angezeigt.

Gesucht sind alle Kunden, welche im Jahr 1985 und später geboren sind:

```
SELECT kfname, kname, kgebdatum
FROM kunden
WHERE kgebdatum >= "1985-01-01";
```

Man gebe die Nummern der Kunden an, deren Postleitzahl 80000 beträgt.

```
SELECT knummer
FROM kunden
WHERE kplz = "80000";
```

Die folgende Anweisung findet alle Luxusartikel:

```
SELECT *
FROM artikel
WHERE rluxus = TRUE;
```

Logische Aussagen wie Vergleiche, die logischen Werte wahr und falsch können durch die booleschen Operatoren **AND** (Konjunktion), **OR** (Disjunktion) und **NOT** (Negation) zu neuen logischen Aussagen verknüpft werden. Die Negation hat von allen drei booleschen Operatoren die höchste Priorität, danach folgen die Konjunktion und dann die Disjunktion. Wenn in einem logischen Ausdruck alle drei Operatoren existieren, wird also zuerst die Negation, danach die Konjunktion und erst am Ende die Disjunktion durchgeführt. Diese Verarbeitungsreihenfolge kann durch entsprechende Klammern geändert werden.

## Boolesche Ausdrücke

Gesucht sind alle Artikel deren Sollbestand größer als der Lagerbestand ist und die mehr als 1000 Euro kosten.

```
SELECT *
FROM artikel
WHERE rsollbestand > rbestand AND rvpreis > 1000;
```



Man finde die Nummern und die Namen aller Kunden, die nicht in Stuttgart wohnen.

```
SELECT knummer, kfname, kname
FROM kunden
WHERE NOT kort = "Stuttgart";
```

Gesucht werden die Auftragsnummern derjenigen Aufträge, die einen der Artikel mit den Nummern 22 oder 33 oder beide enthalten.

```
SELECT anummer
FROM positionen
WHERE rnummer = 22 OR rnummer = 33;
```

## DISTINCT

Sollte in der Ergebnistabelle jede Auftragsnummer nur einmal erscheinen, so ist die Angabe **DISTINCT** zu benutzen:

```
SELECT DISTINCT anummer
FROM positionen
WHERE rnummer = 22 OR rnummer = 33;
```



1. Man finde alle Kunden, die in Köln oder in Bozen wohnen.
2. Man finde alle Kunden, die weder in München noch in Bozen wohnen.
3. Man finde alle Kunden aus Stuttgart, die in der Museumstraße wohnen.
4. Man finde jeden Artikel bei dem der Sollbestand unterschritten ist und gebe aus, um wie viel dieser unterschritten wurde. Weiters soll die Artikelnummer, Bezeichnung, Soll- und Lagerbestand ausgegeben werden.
5. Man finde jeden Luxusartikel bei dem der Sollbestand unterschritten ist und gebe aus, um wie viel dieser unterschritten wurde.
6. Man finde jeden Kunden der mindestens 25 Jahre alt ist.

## Testen auf NULL

### NULL

Um Datenwerte von Spalten, die **NULL**-Werte enthalten, abzufragen, dient der **NULL**-Operator. Die allgemeine Form dieses Operators ist:

```
<Spaltenname> IS [NOT] NULL
```

### IS

In der Syntax des **NULL**-Operators befindet sich eine potentielle Fehlerquelle beim Programmieren mit SQL. Der Vergleich auf **NULL**, bzw. **NOT NULL** wird nicht, wie in allen anderen Fällen mit dem Vergleichsoperator = durchgeführt. Das liegt an der Logik des **NULL**-Wertes. Wird nämlich ein Feld mit **NULL** verglichen – beispielsweise rbeschreibung = **NULL** – so liefert dieser Vergleich nicht wahr oder falsch zurück sondern wiederum **NULL**. Wird hingegen rbeschreibung **IS NULL** ausgewertet, so liefert diese Bedingung wahr oder falsch zurück.

Beachten Sie auch dass **NULL** nicht 0 ist. Folglich liefert die Bedingung 0 **IS NULL** falsch zurück.



Die nachfolgende Anweisung liefert ein leeres Ergebnis zurück obwohl alle Beschreibungen der Artikel **NULL** sind.

```
SELECT *
FROM artikel
WHERE rbeschreibung = NULL;
```

Diese Anweisung liefert das korrekte Ergebnis zurück:

```
SELECT *
FROM artikel
WHERE rbeschreibung IS NULL;
```



1. Suchen Sie nach allen Kunden bei denen die Telefonnummer unbekannt ist und die aus München stammen.
2. Gesucht sind jene Kunden deren Telefonnummer und E-Mail-Adresse unbekannt sind und die Ackermann heißen.

## Der Operator BETWEEN

Der Operator **BETWEEN** definiert einen Bereich, in dem die Werte gesucht werden, die die Bedingung erfüllen. Dieser Operator kann für alle Datentypen verwendet werden.

Die Verwendung des Operators **BETWEEN** anstelle von Vergleichsoperatoren erhöht die Transparenz der **SELECT**-Anweisung.

Man finde die Nummern und Bezeichnungen aller Artikel, deren Verkaufspreis zwischen 200 und 500 Euro liegt.

```
SELECT rnummer, rbezeichnung
FROM artikel
WHERE rvpreis BETWEEN 200 AND 500;
```

Man finde alle jene Aufträge, die im Zeitraum zwischen 12.01.2010 und 13.01.2010 eingegangen sind.

```
SELECT *
FROM auftraege
WHERE adatum BETWEEN "2010-01-12" AND "2010-01-13";
```

**BETWEEN**



## Der Operator IN

Mit dem Operator **IN** kann eine Menge von Konstanten angegeben werden, auf die dann die Suche eingeschränkt wird. Es werden lediglich die Zeilen ausgewählt, bei denen der Datenwert der Spalte in der **WHERE**-Klausel eine Konstante der angegebenen Menge beinhaltet.

Der **IN**-Operator kann auch gemeinsam mit dem booleschen Operator **NOT** verwendet werden. In diesem Fall werden nur die Zeilen ausgewählt, für die der Datenwert der Spalte in der **WHERE**-Klausel keine von den angegebenen Konstanten beinhaltet.

Man finde alle Kunden, deren Nummer entweder 1098, 2121 oder 4711 ist.

```
SELECT *
FROM kunden
WHERE knummer IN (1098, 2121, 4711);
```

Man finde alle Hersteller die aus München, Hamburg oder Stuttgart stammen.

```
SELECT *
FROM hersteller
WHERE hort IN ("München", "Hamburg", "Stuttgart");
```

Man finde alle Artikel deren Nummer weder 11, 22 noch 44 ist.

```
SELECT *
FROM artikel
WHERE rnummer NOT IN (11, 22, 44);
```

**IN**

**NOT IN**



## Der Operator LIKE

Der **LIKE**-Operator ist ein Vergleichsoperator, der Datenwerte einer alphanumerischen Spalte mit einem vorgegebenen Suchmuster vergleicht. Hinter dem **LIKE**-Operator steht eine alphanumerische Konstante, das sogenannte Suchmuster. In diesem Muster haben zwei Zeichen eine spezielle Bedeutung. Diese sind % (Prozentzeichen) und \_ (Unterstrich). Das Prozentzeichen steht in einem Muster für kein, ein oder mehrere beliebige Zeichen. Das Unterstreichungszeichen steht für genau ein beliebiges Zeichen. Jedes andere Zeichen im Suchmuster kennzeichnet sich selbst. Weiters gilt:

**LIKE**

Fehlen in einem Suchmuster die Zeichen %, \_ so kann der Operator = verwendet werden. Z. B. ist die Bedingung name **LIKE** "Bäcker" identisch mit der Bedingung name = "Bäcker".



Um im Suchmuster die Zeichen % und \_ zu maskieren, verwendet man \% und \\_:

```
SELECT *
FROM artikel
WHERE rbezeichnung LIKE "%\%%";
```

Die vorige Anweisung sucht alle Artikel, welche in ihrer Bezeichnung an irgendeiner Stelle das %-Zeichen enthalten.

Man finde die Nummern und Namen aller Kunden, deren Familienname mit dem Buchstaben K beginnt.



```
SELECT knummer, kfname, kname
FROM kunden
WHERE kfname LIKE "K%";
```

Man finde die Namen aller Kunden, deren Familienname genau 5 Buchstaben lang ist und als ersten Buchstaben ein K hat.

```
SELECT kfname, kname
FROM kunden
WHERE kfname LIKE "K_____";
```

Man finde alle Kunden, deren Vorname nicht mit er endet.

```
SELECT *
FROM kunden
WHERE kname NOT LIKE "%er";
```



1. Man finde alle Hersteller mit einer Postleitzahl zwischen 20000 und 70000.
2. Man finde alle Kunden die nicht im Jahr 1985 geboren sind.
3. Man finde alle Kunden die eine der Hausnummern 12, 44, 99 besitzen.
4. Man finde alle Kunden die entweder Pelz, Ille oder Blohm heißen.
5. Man finde alle Artikel deren Bezeichnung mit dem Anführungszeichen (") beginnt.
6. Man finde alle Hersteller deren Name mit er endet und deren Name ihres Wohnortes mit B beginnt.
7. Man finde die Nummern und Namen aller Kunden, deren Vorname mindestens 9 Zeichen lang ist.
8. Man finde alle unterschiedlichen Familiennamen der Kunden, welche die Buchstabenkombination nder enthält.
9. Man finde die Namen aller Kunden, deren Vorname genau 6 Buchstaben lang ist und als zweiten Buchstaben ein a hat.

## Einfache Unterabfragen

### Einfache Unterabfragen

Bis jetzt wurde der Vergleich in der **WHERE**-Klausel immer mit einer Konstante, bzw. einem Ausdruck durchgeführt. Zusätzlich dazu ist es möglich, den Vergleich mit dem Ergebnis einer weiteren **SELECT**-Anweisung durchzuführen. Eine solche **SELECT**-Anweisung, die Teil der Bedingung einer **WHERE**-Klausel ist, wird *einfache Unterabfrage* genannt. (Neben den einfachen Unterabfragen gibt es auch die *korrelierten Unterabfragen*, welche in einem späteren Abschnitt behandelt werden.)

### Innere SELECT-Anweisung

Jede Unterabfrage wird häufig als innere **SELECT**-Anweisung bezeichnet, dies im Unterschied zur äußeren **SELECT**-Anweisung, in der die innere eingeschlossen ist. In einer Abfrage mit einfacher Unterabfrage wird stets zunächst die innere **SELECT**-Anweisung ausgewertet und ihr Ergebnis dann an die äußere **SELECT**-Anweisung wei-

tergegeben. Die Definition einer **SELECT**-Anweisung als Unterabfrage unterscheidet sich deutlich von der einer normalen **SELECT**-Anweisung.

```
SELECT [ALL|DISTINCT] {<Spaltenausdruck>|*}
 <FROM-Klausel>
 [<WHERE-Klausel>]
 [<GROUP BY-Klausel>]
 [<HAVING-Klausel>]
```

Es gibt zwei Unterschiede zwischen der Definition der Unterabfrage und der der **SELECT**-Anweisung:

- Die **SELECT**-Klausel darf nur einen Ausdruck oder das Sternzeichen (nur bei Verwendung von **EXISTS**) enthalten.
- Eine **ORDER BY**-Klausel ist nicht erlaubt. Wenn die Werte in eine andere Reihenfolge gebracht werden, ändert dies nichts an der Bedeutung einer Wertemenge.

Gesucht ist der Name des Kunden der den Auftrag mit der Nummer 20 erteilt hat:

```
SELECT kfname, kname
 FROM kunden
 WHERE knummer =
 (SELECT knummer
 FROM auftraege
 WHERE anummer = 20);
```



Jede Unterabfrage kann eine weitere Unterabfrage enthalten. In diesem Fall spricht man von *geschachtelten Unterabfragen*. Bei allen geschachtelten, einfachen Unterabfragen wird zunächst die innerste **SELECT**-Anweisung abgearbeitet und das Ergebnis der nächsthöheren **SELECT**-Anweisung übergeben.

**geschachtelte Unterabfrage**

Werden Unterabfragen im Zusammenhang mit einem Vergleichsoperator (=) verwendet, so darf die innere **SELECT**-Anweisung nur einen einzigen Wert als Ergebnis liefern. Dies ist auch logisch, da ein Vergleich mit mehreren Werten nicht sinnvoll wäre.

**Vergleichsoperatoren**

Welche Kunden wohnen am selben Ort wie der Hersteller mit der Nummer 3?

```
SELECT *
 FROM kunden
 WHERE kort =
 (SELECT hort
 FROM hersteller
 WHERE hnummer = 3);
```



Welche Kunden haben eine größere Postleitzahl als der Kunde mit der Nummer 4711? Dabei wird das Ergebnis nach Postleitzahl aufsteigend sortiert.

```
SELECT *
 FROM kunden
 WHERE CAST(kplz AS SIGNED INTEGER) >
 (SELECT CAST(kplz AS SIGNED INTEGER)
 FROM kunden
 WHERE knummer = 4711)
 ORDER BY kplz;
```

**ANMERKUNG:** Da die Postleitzahl ein alphanumerisches Feld ist, würde ohne die durch **CAST** erfolgte Typumwandlung Zeichen für Zeichen verglichen.

1. Der Artikel mit der Nummer 49 wurde nur einmal bestellt. Wann erfolgte der Auftrag?
2. Der Artikel mit der Nummer 97 wurde nur einmal bestellt. Man finde den Namen des Kunden, der ihn bestellte.
3. Der Artikel ACO Bandabdeckung weiss kommt von einem einzigen Hersteller. Wie heißt sein Name?



4. Man finde alle Aufträge die von einem Kunden eingereicht wurden, dessen Nummer kleiner ist als die des Kunden Kroll Jürgen.
5. Welche anderen Kunden wohnen im gleichen Ort wie der Kunde mit der Nummer 15?
6. Man finde die Nummern und die Namen aller jener Kunden, die älter sind als die Kundin Eckert Vanessa.

## Unterabfragen und der IN-Operator

### IN-Operator

Auf den **IN**-Operator kann nicht nur eine Menge von Konstanten folgen, sondern auch eine Unterabfrage. Das Ergebnis dieser Unterabfrage muss eine Menge von Datenwerten sein.



Man finde die Nummern derjenigen Kunden, die den Artikel mit der Nummer 33 bestellt haben. Da die Unterabfrage unter Umständen mehrere Auftragsnummern zurück liefert, muss hier der **IN**-Operator verwendet werden.

```
SELECT knummer
FROM auftraege
WHERE anummer IN
 (SELECT anummer
 FROM positionen
 WHERE rnummer = 33);
```

Man finde die Nummern und die Namen aller derjenigen Kunden, welche ACO Lichtschachtkörper bestellt haben.

```
SELECT knummer, kfname, kname
FROM kunden
WHERE knummer IN
 (SELECT knummer
 FROM auftraege
 WHERE anummer IN
 (SELECT anummer
 FROM positionen
 WHERE rnummer IN
 (SELECT rnummer
 FROM artikel
 WHERE rbezeichnung LIKE
 "ACO Lichtschachtkörper%"))));
```



1. Man finde die Namen aller Kunden welche im Monat Jänner 2010 den Artikel mit der Nummer 34 bestellt haben.
2. Man finde die Nummern und Namen aller jener Kunden, welche Artikel bestellt haben, deren aktueller Bestand kleiner als der Sollbestand ist.
3. Man finde alle verschiedenen Artikelnummern, die vom Kunden mit der Nummer 1098 bestellt wurden.
4. Es gibt nur einen Artikel mit einem Verkaufspreis von 4.52 Euro. Wie lauten die Namen der Kunden, die diesen Artikel bestellt haben?
5. Man finde die Nummern und Bezeichnungen aller jener Artikel, welche noch nie vom Kunden mit der Nummer 1098 bestellt wurden.
6. Man finde die Nummern und Namen aller jener Kunden, die sowohl den Artikel mit der Nummer 8 als auch den mit der Nummer 10 im selben Auftrag bestellt haben.

## Die Operatoren ANY und ALL

### ANY und ALL

Die Operatoren **ANY** und **ALL** werden immer im Zusammenhang mit einem Vergleichsoperator benutzt. Die allgemeine Form beider Operatoren ist:

```
<Ausdruck> vergleich_op [ANY|ALL] (Unterabfrage)
```

wobei `vergleich_op` einen der Vergleichsoperatoren darstellt.

Beim **ANY**-Operator (dtsch. irgendein) wird die Bedingung als wahr ausgewertet, falls die Unterabfrage wenigstens einen Datensatz als Ergebnis liefert, der dem angegebenen Vergleich entspricht. Insbesondere ist die Bedingung falsch, wenn die Unterabfrage kein Ergebnis liefert.

## ANY-Operator

Beim **ALL**-Operator (dtsch. alle) wird die Bedingung als wahr ausgewertet, wenn alle Ergebnisse der Unterabfrage dem angegebenen Vergleich entsprechen. Insbesondere ist die Bedingung wahr, wenn die Unterabfrage kein Ergebnis liefert.

## ALL-Operator

Die Bedingung  $x = \mathbf{ANY}(s)$  ist gleichwertig mit  $x \mathbf{IN}(s)$ . Ebenso ist die Bedingung  $x <> \mathbf{ALL}(s)$  gleichwertig mit  $x \mathbf{NOT IN}(s)$  und diese Bedingung ist gleichwertig mit  $\mathbf{NOT}(x \mathbf{IN}(s))$ .

Man finde die Nummern und Namen aller derjenigen Kunden, deren Kundennummer nicht die größte ist.

```
SELECT knummer, kfname, kname
FROM kunden
WHERE knummer < ANY
(SELECT knummer
FROM kunden);
```



In diesem Beispiel wird jeder Datenwert der Spalte `knnummer` mit allen Datenwerten derselben Spalte (auch mit sich selbst) verglichen. Für alle Datenwerte, abgesehen von einem, gilt, dass beim Vergleich die Bedingung wenigstens in einem Fall erfüllt ist. Die Zeile mit der größten Kundennummer wird nicht ausgewählt, weil für sie der Vergleich nicht erfüllt ist.

Man finde die Postleitzahl des Wohnortes des Kunden mit der kleinsten Kundennummer.

```
SELECT kp1z
FROM kunden
WHERE knummer <= ALL
(SELECT knummer
FROM kunden);
```

1. Man finde alle diejenigen Artikel, die billiger sind als der (die) teuersten d.h. nicht am teuersten sind. Die Abfrage soll auf den Verkaufspreis des Artikels angewandt werden.
2. Man finde alle diejenigen Artikel, die teurer als irgendeine Gussabdeckplatte sind.
3. Man finde alle diejenigen Artikel, die teurer als alle Artikel des Herstellers mit der Nummer 11 (d.h. teurer als der teuerste Artikel dieses Herstellers) sind.
4. Man finde alle diejenigen Kunden, deren Hausnummer kleiner ist als alle Hausnummern von Kunden aus München.
5. Man finde die Nummern, Namen und Geburtsjahre derjenigen Kunden, die nicht zu den ältesten Kunden gehören (d.h. wenigstens um einen Tag jünger als diese sind).
6. Man finde Nummer, Bezeichnung und Sollbestand derjenigen Artikel, die den zweitgrößten Sollbestand besitzen.
7. Man finde die Nummern derjenigen Aufträge, in denen der Artikel mit der kleinsten Artikelnummer bestellt wurde.



## Der Operator EXISTS

Der **EXISTS**-Operator prüft das Ergebnis einer Unterabfrage; falls die Unterabfrage zumindest einen Datensatz als Ergebnis liefert, wird die Bedingung als wahr ausgewertet. Falls die Unterabfrage keinen Datensatz als Ergebnis liefert, wird die Bedingung in der **WHERE**-Klausel als falsch ausgewertet. Die allgemeine Form des **EXISTS**-Operators ist:

## EXISTS

**[NOT] EXISTS** (Unterabfrage)

Der Operator **EXISTS** wird im Zusammenhang mit korrelierten Unterabfragen häufig verwendet.



Man finde die Namen derjenigen Kunden, die im Monat Jänner mindestens einen Auftrag eingereicht haben. Die Frage in diesem Beispiel kann mit Hilfe des **IN**-Operators beantwortet werden:

```
SELECT kfname, kname
FROM kunden
WHERE knummer IN
 (SELECT knummer
 FROM auftraege
 WHERE MONTH(adatum) = 1);
```

Die Frage kann jedoch auch mit Hilfe des **EXISTS**-Operators beantwortet werden.

```
SELECT kfname, kname
FROM kunden
WHERE EXISTS
 (SELECT *
 FROM auftraege
 WHERE knummer = kunden.knummer
 AND MONTH(adatum) = 1);
```

**Korrelierte Unterabfrage**

Die Unterabfrage in dieser Anweisung enthält etwas, das noch nicht besprochen worden ist. Die Spaltenspezifikation `kunden.knummer` bezieht sich auf die Tabelle, die in der äußeren **SELECT**-Anweisung angegeben ist. Eine solche Unterabfrage wird als *korrelierte Unterabfrage* bezeichnet. Durch die genannte Spaltenspezifikation besteht nämlich eine Verbindung zwischen der Unterabfrage und der äußeren **SELECT**-Anweisung.

In diesem Beispiel werden für jeden Kunden in der Tabelle `kunden` einzeln geprüft, ob die Unterabfrage Zeilen liefert oder nicht, anders gesagt, ob ein Ergebnis vorhanden ist (**EXISTS**). Das bedeutet, dass die Unterabfrage für jede Zeile der Tabelle `kunden` neu ausgeführt wird. Das einzige, was sich für jede Unterabfrage ändert, ist der Wert von `kunden.knummer` in der Bedingung der **WHERE**-Komponente. Für jeden Vertreter der Tabelle `kunden` kann die Unterabfrage ein anderes Zwischenergebnis haben.

Zu welchen Artikeln der Tabelle `artikel` existieren Auftragspositionen?



```
SELECT rnummer, rbezeichnung
FROM artikel
WHERE EXISTS
 (SELECT *
 FROM positionen
 WHERE rnummer = artikel.rnummer);
```

Diese Frage ist berechtigt, denn die Tabelle `artikel` könnte Artikel enthalten, die nicht bestellt wurden.



Lösen Sie die nachfolgenden Aufgaben ausschließlich mit korrelierten Unterabfragen:

1. Man finde die Nummer und das Auftragsdatum aller jener Aufträge, in denen der Artikel mit der Nummer 77 bestellt wurde.
2. Man finde die Nummer und das Auftragsdatum aller jener Aufträge, in denen der Artikel mit der Nummer 77 nicht bestellt wurde.
3. Man finde die Nummern und Namen aller jener Kunden, die keinen von denjenigen Artikeln bestellt haben, der im Auftrag mit der Nummer 12 bestellt wurde.
4. Man finde die Nummern aller derjenigen Kunden, die den Artikel mit der Nummer 11 und den mit der Nummer 55 im selben Auftrag bestellt haben.

5. Man finde für den Kunden mit der Nummer 4 die Nummern aller jener Artikel, die von ihm nicht bestellt worden sind.
6. Man finde die Nummern aller jener Kunden, die einen Artikel bestellt haben, den der Kunde mit der Nummer 4 noch nicht bestellt hat.

## Aggregatfunktionen

Die Aggregatfunktionen werden auf eine Gruppe von Datenwerten aus einer Spalte angewendet, wobei die Gruppe auch alle Zeilen einer Tabelle umfassen kann. Das Ergebnis jeder Aggregatfunktion ist immer ein einziger Wert. SQL kennt die folgenden Aggregatfunktionen:

### Funktion Bedeutung

|              |                                                                                            |
|--------------|--------------------------------------------------------------------------------------------|
| <b>COUNT</b> | Bestimmt die Anzahl der Werte in einer Spalte oder die Anzahl der Zeilen in einer Tabelle. |
| <b>MIN</b>   | Bestimmt den minimalen Wert in einer Spalte.                                               |
| <b>MAX</b>   | Bestimmt den maximalen Wert in einer Spalte.                                               |
| <b>SUM</b>   | Bestimmt die Summe der Werte in einer Spalte.                                              |
| <b>AVG</b>   | Bestimmt das arithmetische Mittel der Werte in einer Spalte.                               |

### Aggregatfunktionen

Die Funktion **COUNT** berechnet die Anzahl der Werte einer Spalte bzw. die Anzahl der Tabellenzeilen einer Tabelle.

**COUNT**(\* | [**DISTINCT**] <Spaltenname>)

### COUNT

Die Angabe **DISTINCT** bewirkt, dass mehrfach vorhandene Werte nicht berücksichtigt werden.

Man bestimme die Anzahl der Zeilen der Tabelle kunden.

```
SELECT COUNT(*)
FROM kunden;
```



Wie viele Kunden wohnen in Bozen?

```
SELECT COUNT(*)
FROM kunden
WHERE kort = "Bozen";
```

Weil die **SELECT**-Klausel erst nach der **WHERE**-Klausel verarbeitet wird, wird die Anzahl der Zeilen gezählt, in denen die Spalte kort den Wert Bozen besitzt.

Wie viele Artikelnummern gibt es?

```
SELECT COUNT(rnummer)
FROM artikel;
```

Man bestimme die Anzahl der verschiedenen Ortsnamen sowie der verschiedenen Nationalitäten der Tabelle kunden.

```
SELECT COUNT(DISTINCT kort), COUNT(DISTINCT knation)
FROM kunden;
```

Die Aggregatfunktionen **MIN** und **MAX** berechnen den minimalen, bzw. den maximalen Datenwert einer Spalte. Die Spalte kann von einem beliebigen Datentyp sein.

### MIN und MAX

```
MAX(<Spaltenname>) oder
MAX(<Spaltenausdruck>)
```

```
MIN(<Spaltenname>) oder
MIN(<Spaltenausdruck>)
```



Man bestimme die kleinste Kundennummer.

```
SELECT MIN(knummer)
FROM kunden;
```

Man ermittle den alphabetisch gesehen letzten Namen aller Kunden.

```
SELECT MAX(kfname)
FROM kunden;
```

Wie hoch ist der Unterschied zwischen dem größten und dem kleinsten Artikelverkaufspreis in Eurocents?

```
SELECT (MAX(rvpreis) - MIN(rvpreis)) * 100
FROM artikel;
```

Wie viele Postleitzahlen in der Tabelle Kunden entsprechen der höchsten Postleitzahl?

```
SELECT COUNT(*)
FROM kunden
WHERE kplz =
(SELECT MAX(kplz)
FROM kunden);
```

## SUM

Die Aggregatfunktion **SUM** berechnet die Summe der Werte einer Spalte. Die Spalte muss einen numerischen Datentyp haben.

```
SUM([ALL|DISTINCT]<Spaltenname>) oder
SUM([ALL]<Spaltenausdruck>)
```

Durch die Angabe **DISTINCT** werden die mehrfach vorhandenen Werte in der betreffenden Spalte vor Verwendung der Funktion **SUM** eliminiert.



Man berechne die Summe aller Werte der Spalte pmenge der Tabelle positionen.

```
SELECT SUM(pmenge)
FROM positionen;
```

Man berechne die Summe aller verschiedenen Werte der Spalte pmenge der Tabelle positionen.

```
SELECT SUM(DISTINCT pmenge)
FROM positionen;
```

Wie groß ist die Summe aller Stückzahlen von bestellten Artikeln für den Kunden mit der Nummer 1098?

```
SELECT SUM(pmenge)
FROM positionen
WHERE anummer IN
(SELECT anummer
FROM auftraege
WHERE knummer = 1098);
```

## AVG

Die Aggregatfunktion **AVG** berechnet das arithmetische Mittel der Datenwerte einer Spalte. Diese Spalte muss einen numerischen Datentyp haben.

```
AVG([ALL|DISTINCT]<Spaltenname>) oder
AVG([ALL]<Spaltenausdruck>)
```

Durch die Angabe **DISTINCT** werden alle mehrfach vorhandenen Werte vor der Berechnung des arithmetischen Mittels aus der betreffenden Spalte eliminiert.



Man berechne das arithmetische Mittel aller Artikelbestände, deren aktueller Bestand größer als der Sollbestand ist.

```
SELECT AVG(rbestand)
FROM artikel
WHERE rbestand > rsollbestand;
```

Man berechne das ungewogene arithmetische Mittel aller Artikelsollbestände. "Ungewogen" bedeutet, dass jeder Wert nur einmal in der Berechnung berücksichtigt wird.

```
SELECT AVG(DISTINCT rsollbestand)
FROM artikel;
```

Man finde diejenigen Artikel, deren aktueller Bestand höher ist, als der mittlere aktuelle Bestand aller Artikel.

```
SELECT *
FROM artikel
WHERE rbestand >
(SELECT AVG(rbestand)
FROM artikel);
```

Bei der Verwendung von Aggregatfunktionen gilt die folgende wichtige Regel:

### Regel für Aggregatfunktion

Falls eine **SELECT**-Anweisung keine **GROUP BY**-Klausel enthält und falls die **SELECT**-Klausel eine oder mehrere Aggregatfunktionen enthält, darf ein in der **SELECT**-Klausel angegebener Spaltenname ausschließlich innerhalb einer Aggregatfunktion vorkommen.

Die folgende Anweisung ist nicht korrekt, da die **SELECT**-Klausel eine Aggregatfunktion als Ausdruck enthält, während der Spaltenname **knummer** außerhalb einer Aggregatfunktion vorkommt.

```
SELECT COUNT(*), knummer
FROM kunden;
```

Der Grund für diese Beschränkung ist der, dass das Ergebnis einer Aggregatfunktion immer aus einem Wert besteht, während das Ergebnis einer Spaltenspezifikation aus einer Wertemenge besteht. Die obige Anweisung würde mehrere Kundennummer zurück liefern. Für SQL sind diese Ergebnisse nicht miteinander vereinbar.

Diese Regel gilt nur für Spaltenspezifikationen und nicht z. B. für Konstanten. Die folgende Anweisung ist korrekt.

```
SELECT "Die Anzahl der Kunden ist ", COUNT(*)
FROM kunden;
```

1. Wie viele Hersteller sind in der Tabelle `hersteller` enthalten?
2. Wie viele unterschiedliche Artikel wurden bestellt?
3. Wie viele Artikel stammen vom Hersteller Panasonic?
4. An wie viel verschiedenen Orten wohnen die Kunden?
5. Man berechne den maximalen und den minimalen Artikelverkaufspreis sowie das arithmetische Mittel aller Artikelverkaufspreise.
6. Wie hoch ist die höchste Stückzahl von bestellten Artikeln in einer Auftragsposition, die von einem Kunden aus Berlin stammt?
7. Man berechne die Summe der Stückanzahlen von bestellten Artikeln vom 12.01.2010.
8. Man finde die Anzahl der Aufträge in denen am 12.01.2010 der Artikel mit der Nummer 10 bestellt wurde.
9. Man berechne das arithmetische Mittel aller Artikelverkaufspreise, die mehr als 500 Euro kosten.
10. Man finde die Nummern und die Bezeichnungen aller derjenigen Artikel, für die mindestens ein Kunde in einer Auftragsposition eine Stückzahl bestellte die größer ist, als die insgesamt bestellte Stückzahl des Artikels mit der Nummer 44.
11. Man finde die Nummer und das Geburtsjahr von jedem Kunden, der mindestens um ein Jahr älter ist als der jüngste Kunde, der in Stuttgart wohnt.
12. Man finde Nummer, Bezeichnung und Verkaufspreis derjenigen Artikel, die den zweitgrößten Verkaufspreis besitzen.



## Gruppierungen (GROUP BY)

### GROUP BY

Die Gruppierung von Tabellenzeilen anhand einer oder mehrerer Spalten erfolgt mit der **GROUP BY**-Klausel.

```
GROUP BY
 <Spaltenspezifikation>[,<Spaltenspezifikation>...]
```

Die **GROUP BY**-Klausel baut für jeden unterschiedlichen Datenwert der genannten Spalte eine Gruppe auf.



Folgende Tabelle sei gegeben:

| lieferanten | (lNr, | lort,     | umsatz) |
|-------------|-------|-----------|---------|
|             | 1     | München   | 500     |
|             | 2     | München   | 800     |
|             | 3     | Stuttgart | 600     |
|             | 4     | München   | 200     |
|             | 5     | Stuttgart | 50      |
|             | 6     | Bamberg   | 1000    |

Die Anweisung

```
SELECT lort, SUM(umsatz)
FROM lieferanten
GROUP BY lort
```

| ergebnis | (lort,    | SUM(umsatz)) |
|----------|-----------|--------------|
|          | München   | 1500         |
|          | Stuttgart | 650          |
|          | Bamberg   | 1000         |

erstellt für jeden unterschiedlichen Lieferantenort eine eigene Gruppe. Es entstehen also die Gruppen München, Stuttgart und Bamberg. Die Aggregatfunktionen – in diesem Fall **SUM** – werden dann separat auf die einzelnen Gruppen durchgeführt und führen ihre Berechnungen in den Gruppen durch.

Dabei müssen folgende Regeln eingehalten werden:



- Falls die **GROUP BY**-Klausel in der **SELECT**-Anweisung angegeben ist, so muss jede Spalte in der Projektion auch in der **GROUP BY**-Klausel erscheinen.
- Zudem darf die Projektion nur Konstanten und Aggregatfunktionen anderer Spalten enthalten, die in der **GROUP BY**-Klausel nicht erscheinen müssen.

In der **GROUP BY**-Klausel können auch mehrere Spalten angegeben werden. In diesem Fall wird die Gruppierung aufgrund aller angegebenen Spalten durchgeführt. Dabei muss die Reihenfolge der Spalten in der **GROUP BY**-Klausel nicht unbedingt der Reihenfolge der Spalten in der Projektion entsprechen.

In welchen Orten wohnen die Kunden?



```
SELECT kort
FROM kunden
GROUP BY kort;
```

Alle Zeilen mit demselben Ort bilden eine Gruppe.

Man gruppiere die Kunden nach Nationalität und Wohnort.

```
SELECT knation, kort
FROM kunden
GROUP BY knation, kort;
```

Man finde für jeden Wohnort die Anzahl der dort wohnenden Kunden.

```
SELECT kort, COUNT(*)
FROM kunden
GROUP BY kort;
```

Die Funktion **COUNT** wird hier auf alle gruppierten Zeilen statt auf alle Zeilen angewandt.

Falls eine **SELECT**-Anweisung eine **GROUP BY**-Klausel enthält, darf ein in der **SELECT**-Klausel angegebener Spaltenname ausschließlich innerhalb einer Aggregatfunktion vorkommen oder muss in der Liste der Spalten der **GROUP BY**-Klausel enthalten sein.

### Regel für Aggregatfunktionen

Die folgende Anweisung ist nicht korrekt, da die Spalte **kort** in der **SELECT**-Klausel vorkommt, während sie innerhalb einer Aggregatfunktion oder in der Liste der Spalten, nach denen gruppiert wird, nicht vorkommt.



```
SELECT kort, COUNT(*)
FROM kunden
GROUP BY knummer;
```

Der Grund für diese Einschränkung ist der, dass das Ergebnis einer Aggregatfunktion für eine Gruppe aus einem einzigen Wert besteht. Das Ergebnis einer Spaltenspezifikation, nach der gruppiert wird, besteht pro Gruppe ebenfalls aus nur einem Wert. Dagegen besteht das Ergebnis einer Spaltenspezifikation, nach der nicht gruppiert wird, aus einer Wertemenge. Für SQL sind diese Ergebnisse nicht miteinander vereinbar.

1. Man gruppiere die Kunden nach ihren Hausnummern und zähle wie viele Hausnummern pro Gruppe existieren. Man sortiere das Ergebnis absteigend nach der Anzahl der Hausnummern.
2. Wie viele Aufträge gingen an den jeweiligen Tagen ein?
3. Wie viele verschiedene Bestellmengen von Artikeln kommen in den jeweiligen Aufträgen vor?
4. Man gruppiere die Artikel nach dem Artikelsollbestand und man ermittle die Anzahl der verschiedenen Bezeichnungen jeder Gruppe.
5. Wie viele verschiedene Artikel wurden in den jeweiligen Aufträgen bestellt.
6. Welche Gesamtanzahlen an Aufträgen haben die einzelnen Kunden erreicht?
7. Man finde für jeden bestellten Artikel die höchste Stückzahl die in einer Auftragsposition erzielt wurde.
8. Man ermittle für jeden Auftrag die Anzahl der zugehörigen Auftragspositionen sowie die Summe aller Stückzahlen von bestellten Artikeln.
9. Man ermittle für jeden Luxusartikel die Anzahl aller im Monat Jänner bestellten Stückzahlen.
10. Man finde für jeden Artikel des Herstellers Panasonic die Anzahl der Auftragspositionen, in denen er bestellt wurde.



Die Aggregatfunktion **GROUP\_CONCAT** ermöglicht es, mehrere Spaltenwerte zu einer Gruppe zusammenzufassen und als ein einziges Datenfeld auszugeben.

### GROUP\_CONCAT

Gesucht sind für jeden Herstellerort die Namen der Hersteller, die aus diesem stammen:

```
SELECT hort,
GROUP_CONCAT(hname ORDER BY hname SEPARATOR ", ") AS namen
FROM hersteller
GROUP BY hort;
```



Das Ergebnis dieser Abfrage hat folgendes Aussehen:

| ergebnis   | (hort,                                        | namen)                          |
|------------|-----------------------------------------------|---------------------------------|
| Altshausen | Loewe, Sony                                   |                                 |
| Bad        | Oldeslohe                                     | ASTRA, Magnat, Pioneer, Quadral |
| Berlin     | aiwa, Blaupunkt, Bose, Hagenuk, Hama, Philips |                                 |
| Bonn       | Eutelsat, Kodak, Phonar, Revox                |                                 |
| Bremen     | Ascom                                         |                                 |

**GROUP BY WITH ROLLUP**

Bei der GROUP BY-Klausel können die Schlüsselwörter WITH ROLLUP angehängt werden. Wenn GROUP BY nur eine Spalte gruppiert, bewirkt WITH ROLLUP lediglich, dass eine zusätzliche Summenzeile am Ende hinzugefügt wird, wobei als Gruppenname NULL verwendet wird.



Gesucht sind für jeden Hersteller die Anzahl der Artikel, die er im Angebot hat:

```
SELECT hnummer, COUNT(*)
FROM artikel
GROUP BY hnummer WITH ROLLUP;
```

Das Ergebnis dieser Abfrage hat folgendes Aussehen:

| ergebnis | (hnummer, | COUNT(*)) |
|----------|-----------|-----------|
| 1        |           | 10        |
| 2        |           | 2         |
| 3        |           | 6         |
| ...      |           | ...       |
| 64       |           | 3         |
| NULL     |           | 400       |

Interessanter wird die Wirkung von WITH ROLLUP bei mehreren Gruppiereten Spalten. In diesem Fall liefert GROUP BY eine Endsumme für die erste Spalte und zusätzlich Zwischensummen für die zweite Spalte.



Gesucht sind für jeden Hersteller wie viele Luxus- und Nicht-Luxus-Artikel er anbietet:

```
SELECT hnummer, rluxus, COUNT(*)
FROM artikel
GROUP BY hnummer, rluxus WITH ROLLUP;
```

Das Ergebnis der Abfrage hat folgendes Aussehen:

| ergebnis | (hnummer, | rluxus, | COUNT(*)) |
|----------|-----------|---------|-----------|
| 1        | 0         |         | 10        |
| 1        | NULL      |         | 10        |
| 2        | 0         |         | 2         |
| 2        | NULL      |         | 2         |
| 3        | 0         |         | 4         |
| 3        | 1         |         | 2         |
| 3        | NULL      |         | 6         |
| 4        | 0         |         | 5         |
| 4        | 1         |         | 1         |
| ...      | ...       |         | ...       |
| NULL     | NULL      |         | 400       |



1. Gesucht ist die Anzahl der Kunden welche in den einzelnen Jahren geboren sind. Zusätzlich sollen in einer eigenen Spalte die Nachnamen der Kunden aufsteigend sortiert ausgegeben werden.
2. Suchen Sie für jeden Mitarbeiter die von ihm im Januar erstellten Aufträge. Ausgegeben werden soll die Nummer des Mitarbeiters, die Anzahl der Aufträge und in einer eigenen Spalte die aufsteigend sortieren Auftragsnummern.
3. Suchen Sie die Anzahl der pro Monat erstellten Aufträge. Weiters sollen Sie in derselben Abfrage die von jedem Mitarbeiter pro Monat durchgeführten Aufträge zählen. Auch soll im Ergebnis die insgesamt Anzahl von Aufträgen ersichtlich sein.

4. Ermitteln Sie die von jedem Mitarbeiter geleisteten Arbeitszeiten. Um die Differenz zwischen Start- und Endzeit zu ermitteln, müssen Sie mit **SUBTIME** arbeiten. Die Differenzen können nicht einfach aufsummiert werden, sondern müssen vorher mit **TIME\_TO\_SEC** in Sekunden umgerechnet werden. Sie erhalten Sekunden, die zum Schluss durch **SEC\_TO\_TIME** wiederum zurück in eine Zeit konvertiert werden müssen.
5. Schreiben Sie obige Abfrage um, so dass für jeden Mitarbeiter ermittelt wird, wie viel er pro Monat gearbeitet hat. Dabei sollen die Monatssummen und die Gesamtsumme aller geleisteten Arbeitsstunden ausgegeben werden.

## Kreuztabellen erstellen

Kreuztabellen sammeln Informationen aus einer Datenquelle und ordnen diese in Gruppen nicht nur vertikal in Zeilen sondern auch horizontal in Spalten an.

Ausgangspunkt sei die Tabelle auftraege:

| auftraege | (anummer, | knummer, | mnummer, | adatum,    | aausgeliefert) |
|-----------|-----------|----------|----------|------------|----------------|
|           | 1         | 271      | 9        | 2010-02-03 | 0              |
|           | 2         | 113      | 3        | 2010-03-01 | 0              |
|           | 3         | 750      | 7        | 2010-03-16 | 0              |
|           | 4         | 703      | 6        | 2010-03-28 | 1              |
|           | 5         | 630      | 8        | 2010-04-20 | 0              |
|           | 6         | 414      | 5        | 2010-03-01 | 0              |
|           | 7         | 691      | 8        | 2010-01-01 | 1              |
|           | 8         | 205      | 2        | 2010-03-16 | 0              |
|           | ...       | ...      | ...      | ...        | ...            |



Es soll für jeden Kunden ermittelt werden, wie viele Aufträge er in den Monaten Januar bis April erteilt hat.

| ergebnis | (knummer, | Januar, | Februar, | Maerz, | April, | COUNT(*)) |
|----------|-----------|---------|----------|--------|--------|-----------|
|          | 4         | 0       | 1        | 0      | 0      | 1         |
|          | 5         | 1       | 0        | 0      | 0      | 1         |
|          | 6         | 0       | 0        | 0      | 1      | 1         |
|          | 7         | 1       | 1        | 0      | 0      | 2         |
|          | 12        | 0       | 1        | 0      | 0      | 1         |
|          | 18        | 0       | 1        | 0      | 1      | 2         |
|          | 24        | 0       | 1        | 0      | 1      | 2         |
|          | 26        | 0       | 0        | 1      | 0      | 1         |
|          | 36        | 1       | 0        | 0      | 0      | 1         |
|          | ...       | ...     | ...      | ...    | ...    | ...       |
|          | NULL      | 54      | 35       | 59     | 52     | 200       |

MySQL stellt nicht wie andere Datenbanksysteme eigene Funktionen zur Erstellung von Kreuztabellen bereit. Hier muss man sich anhand der **GROUP BY**-Klausel und der **IF**-Funktion folgendermaßen behelfen:

```
SELECT knummer,
 SUM(IF(MONTH(adatum) = 1, 1, 0)) AS Januar,
 SUM(IF(MONTH(adatum) = 2, 1, 0)) AS Februar,
 SUM(IF(MONTH(adatum) = 3, 1, 0)) AS Maerz,
 SUM(IF(MONTH(adatum) = 4, 1, 0)) AS April,
 COUNT(*)
FROM auftraege
GROUP BY knummer WITH ROLLUP;
```

Zuerst werden die einzelnen Aufträge pro Kunde gruppiert. Sollte das Datum des Auftrages im Januar liegen, so wird dieser Auftrag zur Januar-Spalte hinzugefügt und dort aufsummiert. Auf dieselbe Art und Weise werden auch für die nächsten Monate die Aufträge analysiert. **COUNT(\*)** am Ende der Projektion bewirkt, dass für den Kunden – also pro Gruppe – die Gesamtanzahl der gemachten Aufträge ermittelt werden.

**WITH ROLLUP** bewirkt, dass am Ende des Abfrageergebnisses die Gesamtsummen über die Monate ermittelt werden.



1. Ermitteln Sie für jeden Mitarbeiter wie viele Tage er in den Monaten Januar bis April gearbeitet hat. Das Ergebnis soll auch die gesamten Arbeitstage pro Mitarbeiter und die Arbeitstage aller Mitarbeiter pro Monat enthalten.
2. Ermitteln Sie für die Orte aus denen die Kunden stammen wie viele Kunden jeweils in den Jahren 1979, 1980, ..., 1989 geboren wurden. Dabei soll für jeden Ort die Gesamtzahl der aus ihm stammenden Kunden und auch für jedes Jahr die Gesamtzahl der Kunden die in diesem Jahr geboren wurden, ermittelt werden. Verwenden Sie als Spaltennamen die Namen j1979, j1980, ..., j1989.

## Auswahl bestimmter Gruppen (HAVING)

### HAVING

Die **HAVING**-Klausel hat dieselbe Funktion für die **GROUP BY**-Klausel wie die **WHERE**-Klausel für die **SELECT**-Anweisung. Mit anderen Worten: Die **HAVING**-Klausel definiert die Bedingung nach der die Gruppen der Zeilen ausgewählt werden.

**HAVING** <Bedingung>

Die Bedingung in der **HAVING**-Klausel darf im Gegensatz zur Bedingung einer **WHERE**-Klausel Aggregatfunktionen und Konstanten enthalten.

Man finde alle Wohnorte, in denen genau zwei Kunden wohnen.



```
SELECT kort
FROM kunden
GROUP BY kort
HAVING COUNT(*) = 2;
```

Alle Zeilen der Tabelle kunden werden zunächst in Bezug auf die Spalte kort gruppiert. Die Aggregatfunktion **COUNT(\*)** zählt alle Zeilen jeder Gruppe, und die Bedingung in der **HAVING**-Klausel wählt die Gruppen aus, die genau zwei Zeilen beinhalten.



Folgende Regeln müssen für Spaltennamen in **HAVING** eingehalten werden:

- Ein in der **HAVING**-Klausel angegebener Spaltenname darf ausschließlich innerhalb einer Aggregatfunktion vorkommen oder
- muss in der Liste der Spalten der **GROUP BY**-Komponente enthalten sein.
- Der Ausdruck in der Bedingung muss je Gruppe stets einen einzigen Wert als Ergebnis liefern.

In der Praxis werden nahezu ausschließlich Vergleiche mit Aggregatfunktionen durchgeführt.

Man ermittle die Nummer und die Summe der Stückzahlen von bestellten Artikeln jedes Auftrags dessen Summe der Stückzahlen von bestellten Artikeln größer als 50 ist.



```
SELECT rnummer, SUM(pmenge)
FROM positionen
GROUP BY rnummer
HAVING SUM(pmenge) > 50;
```

### Allgemeine Regel

Ein in der **HAVING**-Klausel angegebener Spaltenname darf ausschließlich innerhalb einer Aggregatfunktion vorkommen oder muss in der Liste der Spalten der **GROUP BY**-Klausel enthalten sein.

Die folgende Anweisung ist nicht korrekt, da die Spalte `kgebdatum` in der **HAVING**-Klausel vorkommt, während sie innerhalb einer Aggregatfunktion oder in der Liste der Spalten, nach denen gruppiert wird, nicht vorkommt.

```
SELECT kort, COUNT(*)
FROM kunden
GROUP BY kort
HAVING kgebdatum > "1986-11-30";
```

Der Grund für diese Beschränkung entspricht dem der Regel für die **SELECT**-Klausel. Das Ergebnis einer Aggregatfunktion besteht für jede Gruppe immer aus einem Wert. Das Ergebnis einer Spaltenspezifikation, nach der gruppiert wird, besteht pro Gruppe ebenfalls immer nur aus einem Wert. Dagegen besteht das Ergebnis einer Spaltenspezifikation, nach der nicht gruppiert wird, aus einer Wertemenge. Für SQL sind diese Ergebnisse nicht miteinander vereinbar. Mit folgender **WHERE**-Klausel lässt sich obiges Problem aber lösen:

```
SELECT kort, COUNT(*)
FROM kunden
WHERE kgebdatum > "1986-11-30";
GROUP BY kort;
```

1. Man finde alle Artikel die in mindestens 12 verschiedenen Auftragspositionen bestellt wurden.
2. Man finde alle Wohnorte in denen mehr als 30 Kunden wohnen.
3. Von welchen Herstellern kommen genau 5 Artikel? Geben Sie die Namen der Hersteller aus.
4. Man finde die Nummern aller jener Aufträge, deren Auftragspositionen eine durchschnittliche Bestellstückzahl pro Artikel erzielten, die größer ist als die durchschnittliche Bestellmenge des Artikels mit der Nummer 1.
5. Man finde die Nummern aller Kunden, die mindestens zweimal in ein und demselben Auftrag eine Stückzahl größer 95 bestellt haben.
6. Man finde die Nummer von jedem Kunden, der insgesamt genauso viele Aufträge gestellt hat, wie der Kunde mit der Nummer 275.



## Ergebnistabelle ordnen (ORDER BY)

Die **ORDER BY**-Klausel definiert die Reihenfolge der Zeilen in der Ergebnistabelle einer **SELECT**-Anweisung. Diese Klausel ist optional und erscheint immer am Ende einer **SELECT**-Anweisung.

```
ORDER BY {<Spaltenname>|<Ganzzahl>
[ASC|DESC]}, ...
```

Für die Angabe des Ordnungsbegriffes können Spaltennamen und ganze Zahlen benutzt werden. Letztere bezeichnen die Position der jeweiligen Spalte in der Projektion (von links nach rechts gezählt bei 1 beginnend). Als Spaltenname kann auch eine Kalkulationsspalte angegeben werden, welche nur in der **SELECT**-Anweisung durch **AS** definiert wurde.

**ASC** kennzeichnet die aufsteigende und **DESC** die absteigende Sortierfolge. Fehlt diese Angabe, werden die Tabellenzeilen aufgrund der genannten Spalte aufsteigend sortiert.

Der Ordnungsbegriff kann, wie aus der Beschreibung der Syntax ersichtlich ist, mehrere Spalten beinhalten.

**ORDER BY**

**ASC und  
DESC**



Soll die Sortierung der Artikeldaten in numerisch absteigender Reihenfolge erfolgen, so ist das Wortsymbol **DESC** hinter dem Spaltennamen `rnummer` anzugeben:

```
SELECT *
 FROM artikel
 ORDER BY rnummer DESC;
```

Man gebe die Zeilen der Tabelle `auftraege` aus und sortiere dabei zunächst nach der Kundennummer und bei gleichen Kundennummern zusätzlich nach der Auftragsnummer:

```
SELECT *
 FROM auftraege
 ORDER BY knummer, anummer;
```

Man zeige Kundennummer und Kundennamen sowie ihren Wohnort absteigend geordnet nach Wohnort, und bei gleichem Wohnort aufsteigend geordnet nach Kundennummer an:

```
SELECT knummer, kfname, kname, kort
 FROM kunden
 ORDER BY 4 DESC, 1 ASC;
```

Man finde jene 10 Aufträge mit der höchsten Gesamtbestellmenge:

```
SELECT anummer, SUM(pmenge) AS gesamtmenge
 FROM positionen
 GROUP BY anummer
 ORDER BY gesamtmenge DESC
 LIMIT 10;
```



1. Man zeige Kundennummer, Kundenname aufsteigend geordnet nach der Kundennummer an.
2. Man zeige Kundennummer, Kundenname absteigend geordnet nach der Kundennummer an.
3. Man zeige alle Aufträge in chronologisch absteigender Reihenfolge an.
4. Man zeige die Hersteller in alphabetisch aufsteigender Reihenfolge nach dem Namen an.
5. Man zeige die Aufträge vom 12.01.2010 aufsteigend geordnet nach der Kundennummer an.
6. Man zeige für jeden Auftrag, der mindestens 15 Bestellpositionen aufweist, die Gesamtsumme von bestellten Artikelstückzahlen an; man sortiere das Ergebnis in aufsteigender Reihenfolge nach diesen Gesamtsummen.
7. Man ermittle für jeden Kunden sein Alter in Tagen sowie sein abgerundetes Alter in Jahren und man sortiere das Ergebnis in absteigender Reihenfolge nach dem Alter in Tagen (**HINWEIS: FLOOR()**).

## Anzahl der Ergebnisdatensätze einschränken (LIMIT)

### LIMIT

Es können nicht nur die Anzahl der Spalten, sondern auch die Anzahl der Ergebnisdatensätze limitiert werden. Angenommen die Tabelle `kunden` enthielte 100.000 Kunden, aber man bräuchte vorerst nur die ersten zehn Datensätze, um sie beispielsweise in einem HTML-Dokument darzustellen. Da wäre es eine Vergeudung von Rechenzeit, Speicher- und Netzkapazität, auch die verbleibenden 99.990 Datensätze abzurufen.

Um dies zu vermeiden, kann die Anzahl der Ergebnisdatensätze mit **LIMIT n** beschränkt werden.



Man ermittle die ersten 15 Kunden der Kundentabelle aufsteigend sortiert nach Nachname.

```
SELECT *
FROM kunden
ORDER BY kfname
LIMIT 15;
```

Um die nächsten 10 Kunden zu ermitteln, wird **LIMIT** offset, n verwendet. Dabei gibt offset an, mit welchem Datensatz die Ausgabe begonnen werden soll wobei die Zählung der Datensätze mit 0 beginnt.

```
SELECT *
FROM kunden
ORDER BY kfname
LIMIT 15, 10;
```

**LIMIT** 15, 10 beginnt die Ausgabe mit dem sechzehnten Datensatz – also mit dem Datensatz mit der Nummer 15, und 10 gibt die Anzahl der zurückzuliefernden Datensätze an.

Anhand von **SQL\_CALC\_FOUND\_ROWS** und **FOUND\_ROWS()** kann die Anzahl der durch **LIMIT** unterdrückten Datensätze ermittelt werden. Dadurch kann beispielsweise ermittelt werden, wie viele Seiten des Ergebnisses noch angezeigt werden sollen:

```
SELECT SQL_CALC_FOUND_ROWS *
FROM kunden
ORDER BY kfname
LIMIT 15;

SELECT FOUND_ROWS();
```

Der unmittelbar nach der ersten Abfrage folgende Aufruf von **FOUND\_ROWS()** ermittelt die Gesamtzahl der Datensätze, welche die Abfrage ohne **LIMIT** zurückliefern würde.

Beachten Sie, dass das Werkzeug phpMyAdmin bei der Durchführung obiger Abfragen falsche Ergebnisse liefert. Werden diese Anweisungen aber direkt an das Datenbanksystem geschickt – beispielsweise über JDBC – so werden die exakten Ergebnisse geliefert.

**FOUND\_ROWS**

## Datensätze zufällig auswählen (RAND)

Manchmal ist es sinnvoll, Datensätze zufällig auszuwählen – beispielsweise um auf einer Website wechselnde Bilder, Werbeangebote usw. anzuzeigen.

Die nachfolgende Anweisung ermittelt aus der Tabelle `artikel` einen zufällig ausgewählten Artikel:

```
SELECT *
FROM artikel
ORDER BY RAND()
LIMIT 1;
```

1. Ermitteln Sie die ersten 2 Mitarbeiter sortiert nach Namen.
2. Ermitteln Sie die nächsten 2 Mitarbeiter sortiert nach Namen.
3. Wählen Sie aus der Tabelle `mitarbeiter` zwei Mitarbeiter zufällig aus.
4. Wählen Sie einen Artikel zufällig aus, von dem mehr als 300 Stück bestellt wurden.

**RAND**

## Mengenoperationen (UNION)

### UNION

Der Mengenoperator **UNION** verbindet zwei **SELECT**-Anweisungen und liefert als Ergebnis die Zeilen, die entweder von der ersten, oder von der zweiten, oder von beiden **SELECT**-Anweisungen ausgewählt wurden.

```
select_1 UNION select_2
```

`select_1` und `select_2` kennzeichnen zwei **SELECT**-Anweisungen, die der Operator **UNION** verbindet.

### Voraussetzungen

Damit zwei **SELECT**-Anweisungen mit dem **UNION**-Operator verbunden sein können, müssen die folgenden Voraussetzungen erfüllt sein:

- Die Anzahl der Spalten in den beiden Projektionen muss gleich sein.
- Die entsprechenden Spalten müssen denselben Datentyp haben.
- Falls die Ausgabe sortiert sein soll, darf die **ORDER BY**-Klausel nur in der letzten **SELECT**-Anweisung angegeben werden. Sortiert wird erst auf der Grundlage des vollständigen Endergebnisses, also erst nachdem alle Zwischenergebnisse miteinander kombiniert worden sind.



Man finde alle Wohnorte der Kunden und alle Herkunftsorte der Hersteller und sortiere das Ergebnis aufsteigend nach Orten.

```
SELECT kort
 FROM kunden
UNION
SELECT hort
 FROM hersteller
ORDER BY 1;
```



1. Man finde alle unterschiedlichen Kunden- und Herstellernummern.
2. Man finde alle Geburtstage von Kunden sowie alle Datumsangaben an denen Aufträge eingingen. Das Ergebnis soll absteigend nach Datum sortiert werden.
3. Suchen Sie alle Aufträge mit der Anzahl der in ihnen enthaltenen Auftragspositionen. Für jeden Auftrag soll der Text "Auftrag", die Auftragsnummer und die Anzahl der enthaltenen Auftragspositionen ausgegeben werden. Weiters sollen Sie sich zu jedem Artikel ermitteln, in wie vielen Aufträgen er enthalten ist. Ausgegeben werden soll der Text "Artikel", die Artikelnummer und die Anzahl der Aufträge in denen der Artikel enthalten ist. Vereinigen Sie dann die beiden Ergebnismengen, und sortieren Sie nach der zweiten Spalte.

## Verbund von Tabellen

### Join

Der Verbund der Tabellen wird mit Hilfe des relationalen Operators *join* durchgeführt, was im Allgemeinen bedeutet, dass Datenwerte aus zwei oder mehreren Tabellen mittels *einer* **SELECT**-Anweisung ausgewählt werden. Der Operator *join* kann auch zum Verknüpfen einer Tabelle mit sich selbst verwendet werden.

### Unterschied zu UNION

In den vorhergehenden Abschnitten wurden die Mengenoperatoren beschrieben, die im Grunde auch zwei Tabellen verknüpfen. Trotzdem gibt es zwei wesentliche Unterschiede zwischen diesen beiden Arten von Tabellenverknüpfungen:

- Der Mengenoperator **UNION** verknüpft immer zwei **SELECT**-Anweisungen, während der Operator *join* die Verknüpfung mehrerer Tabellen mittels einer **SELECT**-Anweisung durchführt.
- Weiters werden für die Verknüpfung mit dem Operator **UNION** immer die Zeilen der Tabellen verwendet, während der Operator *join*, wie wir noch sehen werden, gewisse Spalten der Tabellen für die Verknüpfung verwendet.

Wenn Tabellen miteinander verbunden werden, werden die Zeilen aus diesen Tabellen aneinandergesetzt. Das Ergebnis ist wiederum eine Tabelle. Eine **SELECT**-Anweisung kann als ein Join bezeichnet werden, wenn die **FROM**-Klausel zwei oder mehr Tabellenspezifikationen enthält und die **WHERE**-Klausel mindestens eine Bedingung umfasst, in der Spalten aus den verschiedenen Tabellen miteinander verglichen werden.

Die Spalten, die in einer **SELECT**-Anweisung für das Zusammenfügen zuständig sind, werden als *Join-Spalten* bezeichnet. Die Join-Spalten zwischen denen ein Vergleichsoperator steht, müssen vom gleichen Datentyp sein.

Die Spaltenpaare, die in einem Vergleich beim Join erscheinen, unterliegen in der Praxis einer weiteren Bedingung, nämlich der, dass die sinnvolle Verknüpfung zweier Tabellen nur über Spaltenpaare durchgeführt wird, die dieselbe logische Bedeutung in der Anwendung haben.

In unserer Beispieldatenbank existieren insgesamt vier solcher Spaltenpaare. Die Tabellen *kunden* und *auftraege* lassen sich durch die Join-Spalten *kunden.knummer* und *auftraege.knummer* verbinden. Genauso lassen sich die Tabellen *hersteller* und *artikel* durch die Join-Spalten *hersteller.hnummer* und *artikel.hnummer*, die Tabellen *auftraege* und *positionen* durch die Join-Spalten *auftraege.anummer* und *positionen.anummer* und die Tabellen *artikel* und *positionen* durch die Join-Spalten *artikel.rnummer* und *positionen.rnummer* verbinden. Diese Spaltenpaare geben die Beziehungen zwischen den Tabellen wieder.

Wie man hieraus erkennt, hat jedes Join-Spaltenpaar der Beispieldatenbank denselben Namen, was im Allgemeinen nicht der Fall sein muss.

Die spezifizierten Namen in der **WHERE**-Klausel, wie z. B. *artikel.rnummer* und *positionen.rnummer*, sind unbedingt anzugeben, falls die Spaltennamen in der **SELECT**-Anweisung nicht eindeutig sind.

Zusätzlich zu Bedingungen mit Join-Spalten können weitere Bedingungen in der **WHERE**-Klausel existieren.

In der folgenden **SELECT**-Anweisung sind *kunden.knummer* und *auftraege.knummer* die Join-Spalten. Diese Anweisung sucht alle Kunden die Aufträge erteilt haben.

```
SELECT kunden.knummer, kfname, kname, anummer, adatum
FROM kunden, auftraege
WHERE kunden.knummer = auftraege.knummer;
```

Eine weitere syntaktische Variante wäre die folgende:

```
SELECT kunden.knummer, kfname, kname, anummer, adatum
FROM kunden INNER JOIN auftraege
ON kunden.knummer = auftraege.knummer;
```

oder

```
SELECT kunden.knummer, kfname, kname, anummer, adatum
FROM kunden INNER JOIN auftraege USING (knummer);
```

Die letzte Variante geht davon aus, dass die Join-Spalten in den zu verknüpfenden Tabellen denselben Namen haben.

**Verknüpfungsbedingung**

**Join-Spalten**

**Sinnvolle Verknüpfung**

**Spaltenkennungen**



**Join-Arten**

Das relationale Datenmodell unterscheidet zwischen verschiedenen Arten von Joins. In den nachfolgenden Abschnitten werden die folgenden Arten von Joins behandelt:

- *Kartesisches Produkt*
- *Gleichverbund* (Equijoin)
- *Natürlicher Verbund* (Natural Join)
- *Thetaverbund*

**Aliasnamen**

Falls mehrere Tabellennamen in der **FROM**-Klausel der **SELECT**-Anweisung vorkommen empfiehlt es sich oft, mit sogenannten Aliasnamen (Pseudonymen) zu arbeiten. Aliasnamen sind temporäre alternative Namen für Tabellennamen.

```
SELECT k.knummer, k.kfname, k.kname, a.anummer, a.adatum
FROM kunden k, auftraege a
WHERE k.knummer = a.knummer;
```

**Kartesisches Produkt****Kartesisches Produkt**

Beim *kartesischen Produkt* zweier Tabellen wird jede Zeile der ersten Tabelle mit jeder Zeile der zweiten Tabelle verkettet. Dadurch entsteht eine Tabelle mit einer Anzahl von Zeilen die dem Produkt der Zeilenanzahlen der ersten und der zweiten Tabelle entspricht.

In der Praxis wird das kartesische Produkt äußerst selten bewusst benutzt. Manchmal kommt es vor, dass der Anwender das kartesische Produkt unbewusst erzeugt, wenn er vergisst, den Vergleich zwischen den Join-Spalten in der **WHERE**-Klausel anzugeben. Dieses Ergebnis entspricht dann nicht dem tatsächlichen Resultat, das der Anwender erhalten wollte.

Man bilde das kartesische Produkt der Tabellen *auftraege* und *positionen*.

```
SELECT *
FROM auftraege, positionen;
```

Das folgende Beispiel stellt ein kartesische Produkt zweier Tabellen dar, in dem die Bedingung in der **WHERE**-Klausel existiert, das aber nicht den Vergleich zwischen Join-Spalten enthält.

```
SELECT *
FROM kunden, auftraege
WHERE anummer = 0012;
```



1. Man bilde das kartesische Produkt der Tabellen *kunden* und *auftraege*. Wie viele Datensätze enthält das Ergebnis? Wie viele Spalten enthält das Ergebnis?
2. (🧐) Man bilde das kartesische Produkt der drei Tabellen *kunden*, *auftraege* und *positionen*.

**Gleichverbund (Equijoin)****Gleichverbund**

Beim *Gleichverbund* ist das Gleichheitszeichen (=) der Vergleichsoperator zwischen den Join-Spalten. Die Projektion umfasst *alle* Spalten beider Tabellen.

Man finde für jeden Artikel, zusätzlich zu seinen Daten (Daten die in der Tabelle *artikel* enthalten sind), die Daten seines Herstellers.



```

SELECT *
 FROM artikel, hersteller
 WHERE artikel.hnummer = hersteller.hnummer;

```

Man bilde den Gleichverbund der Tabellen `artikel` und `positionen` über die Artikelnummer.

```

SELECT *
 FROM artikel, positionen
 WHERE artikel.rnummer = positionen.rnummer;

```

1. Man führe durch einen Gleichverbund die beiden Tabellen `kunden` und `auftraege` zu einer einzigen Tabelle zusammen und gebe nur jene Kunden aus, die aus München stammen.
2. Man übersetze folgende Unterabfrage in einen Gleichverbund:

```

SELECT COUNT(*)
 FROM kunden
 WHERE knummer IN
 (SELECT knummer
 FROM auftraege
 WHERE anummer IN
 (SELECT anummer
 FROM positionen
 WHERE rnummer = 11));

```



## Natürlicher Verbund (Natural Join)

Der *natürliche Verbund* entsteht aus einem Gleichverbund, wenn die doppelte Spalte in der Projektion entfernt wird. Die Projektion eines natürlichen Verbunds muss nicht unbedingt alle unterschiedlichen Spalten beider Tabellen beinhalten. **Natürlicher Verbund**

Der natürliche Verbund wird in der Praxis von allen Verbundarten am häufigsten angewendet. Deswegen impliziert das Kürzel "Verbund" immer einen natürlichen Verbund.

Man bilde den natürlichen Verbund der Tabellen `auftraege` und `positionen` über die Spalte `rnummer`.

```

SELECT auftraege.*, rnummer, pmenge
 FROM auftraege, positionen
 WHERE
 auftraege.anummer = positionen.anummer;

```



Die Projektion in diesem Beispiel beinhaltet alle Spalten der ersten Tabelle und jene Spalten der zweiten Tabelle, die nicht die Join-Spalte darstellen. Die Spalten `anummer` und `pmenge` der zweiten Tabelle sind in der Projektion mit nicht spezifiziertem Namen, d.h. ohne Tabellennamen angegeben. Dies ist möglich, weil diese Namen in den beiden Tabellen eindeutig sind.

Man finde die Kundennummern und Kundennamen derjenigen Kunden, von denen am 12.01.2010 Aufträge eingingen.

```

SELECT DISTINCT kunden.knummer, kfname, kname
 FROM kunden, auftraege
 WHERE adatum = "2010-01-12"
 AND kunden.knummer = auftraege.knummer;

```

Diese Frage kann auch mit Hilfe einer Unterabfrage beantwortet werden.

```

SELECT knummer, kfname, kname
 FROM kunden
 WHERE knummer IN
 (SELECT knummer
 FROM auftraege
 WHERE adatum = "2010-01-12");

```

Der Unterschied zwischen den beiden Anweisungen liegt darin, dass beim Verbund auch Datenfelder der Tabelle auftrage ausgegeben werden könnten, während dies bei der Lösung mittels Unterabfrage nicht möglich ist.



1. Welche Bezeichnung haben die Artikel, die im Auftrag mit der Nummer 12 bestellt wurden? Geben Sie neben der Artikelbezeichnung auch die Auftragsnummer und das Auftragsdatum aus.
2. Man finde für jede Auftragsposition, zusätzlich zur Auftragsnummer, Artikelnummer und Menge auch die Bezeichnung und den Verkaufspreis des Artikels.
3. Man finde alle Aufträge von Kunden aus Bozen.
4. Man finde in der Tabelle positionen alle Auftragspositionen von Abdichtungsflanschen.
5. Man finde zu den Aufträgen mit den Nummern 12, 13 und 14 den Namen, den Wohnort und das Geburtsdatum der entsprechenden Kunden.
6. Man finde die (verschiedenen) Nummern der Artikel, die an den jeweiligen Tagen bestellt wurden. Sortieren Sie das Ergebnis nach Auftragsdatum und Artikelnummer.
7. Man berechne die Summe aller bestellten Mengen von Artikeln vom 12.01.2010.
8. Man finde die Nummern, den Namen und die Anzahl aller Aufträge jener Kunden, die genau 3 Aufträge aufzuweisen haben.
9. Man finde für jeden Artikel der mindestens einmal bestellt wurde, die Anzahl der unterschiedlichen Tage, an denen er bestellt wurde. Sortieren Sie das Ergebnis absteigend nach der Anzahl.
10. Man finde für jeden bestellten Artikel die Bezeichnung und die Anzahl der Aufträge in denen er bestellt wurde.
11. Man zeige für alle an einem Tag bestellten Artikel, die Tagesbestellmenge (Summe aller Bestellmengen an einem Tag für diesen Artikel) an. Man ordne das Ergebnis für jeden Tag in absteigender Reihenfolge nach der Tagesbestellmenge.

## Theta-Verbund (Theta-Join)

### Theta-Verbund

Der *Theta-Verbund* kennzeichnet jene **SELECT**-Anweisung, bei der die Join-Spalten in der **WHERE**-Klausel mit einem der Vergleichsoperatoren verglichen werden.

```
SELECT tabelle_1.spalten, tabelle_2.spalten
FROM tabelle_1, tabelle_2
WHERE join_spalte_1 v_op join_spalte_2;
```

### Operator

v\_op stellt dabei einen beliebigen *Vergleichsoperator* dar. Jeder Gleichverbund ist gleichzeitig auch ein Theta-Verbund, wenn man für v\_op das Gleichheitszeichen benutzt.

Die Verwendung der Form des Theta-Verbunds mit einem anderen Vergleichsoperator als dem Gleichheitszeichen kommt in der Praxis nicht so oft vor. Am häufigsten wird noch der Vergleich auf die Ungleichheit durchgeführt.

Man finde für jeden Kunden alle Hersteller, die ihren Ansitz nicht am Wohnort des Kunden haben.



```
SELECT knummer, kfname, kname, kort, hnummer, hname, hort
FROM kunden, hersteller
WHERE kort <> hort;
```

In diesem Beispiel werden die Spalten kort und hort miteinander verglichen. Der Wohnort des Kunden und der Ansitz des Herstellers sind voneinander verschieden.

Man finde für jeden Kunden alle Orte der Tabelle hersteller, deren Postleitzahl größer ist als die Postleitzahl des Wohnortes des Kunden.

```
SELECT knummer, kfname, kname, kplz, kort, hplz, hort
FROM kunden, hersteller
WHERE kplz < hplz;
```

1. Man finde für alle Kunden diejenigen Hersteller, deren Hausnummer kleiner ist. Man sortiere das Ergebnis nach Kundennummer und nach der Hausnummer des Herstellers.
2. Man finde für alle Kunden diejenigen Artikel, deren Bezeichnung alphabetisch vor dem Familiennamen des Kunden liegt.



## Der Verbund von mehr als zwei Tabellen

Die Anzahl der Tabellen, die miteinander verknüpft sein können, ist theoretisch unbegrenzt. Trotzdem hat jedes System eine implementierungsbedingte Einschränkung, die die Anzahl der möglichen Joins begrenzt hält (derzeit in MySQL über 50).

### Verbund mehrerer Tabellen

Man finde die Nummern, Familiennamen und Namen aller Kunden die den Artikel mit der Nummer 33 bestellt haben.

```
SELECT DISTINCT kunden.knummer, kfname, kname
FROM kunden, auftraege, positionen
WHERE kunden.knummer = auftraege.knummer
AND auftraege.anummer = positionen.anummer
AND rnummer = 33;
```



In diesem Beispiel müssen drei Tabellen, nämlich kunden, auftraege und positionen miteinander verknüpft werden, damit die notwendige Information mittels einer **SELECT**-Anweisung ausgewählt wird. Diese drei Tabellen werden mit Hilfe von zwei Paaren von Join-Spalten verknüpft: (kunden.knummer, auftraege.knummer) und (auftraege.anummer, positionen.anummer).

Man bilde die Tabelle der ersten Normalform, der sieben sich in der dritten Normalform befindenden Tabellen der Beispieldatenbank.

```
SELECT k.knummer, kfname, kname, kstrasse, khnr, kplz, kort,
knation, ktelefon, kemail, kgebdatum, a.rnummer,
adatum, aausgeliefert, m.mnummer, mfname, mname, zdatum, zvon,
zbis, p.rnummer, pmenge, rbezeichnung, repreis, rvpreis,
rbestand, rsollbestand, rimangebot, rluxus, rbeschreibung,
r.hnummer, hname, hstrasse, hhnr, hplz, hort, hnation
FROM kunden k, auftraege a, positionen p, artikel r,
hersteller h, mitarbeiter m, arbeitszeiten z
WHERE k.knummer = a.knummer AND a.anummer = p.anummer
AND p.rnummer = r.rnummer AND r.hnummer = h.hnummer
AND a.mnummer = m.mnummer AND m.mnummer = z.mnummer;
```

In diesem Beispiel werden alle sieben Tabellen der Beispieldatenbank miteinander verknüpft (**ANMERKUNG**: Kunden welche keine Aufträge gemacht haben, werden in das Ergebnis der obigen Anweisung nicht mit aufgenommen).

1. Welche Bezeichnung haben die Artikel, die am 15.01.2010 bestellt wurden?
2. Man finde die Namen aller Kunden aus München, die Abdichtungsflanschen bestellt haben.
3. Man finde die Nummern und Namen aller Kunden, die einen Luxusartikel bestellt haben.
4. Man finde die Nummern und Namen aller Kunden, die Artikel des Herstellers Panasonic bestellt haben.
5. Man berechne die Summe aller bestellten Mengen von Artikeln, die von Kunden aus Deutschland am 15.01.2010 bestellt wurden.



6. Wie viele verschiedene Artikel wurden von den jeweiligen Kunden an den jeweiligen Tagen bestellt.
7. Welche durchschnittliche Bestellmenge haben die einzelnen Kunden für jeden von ihnen bestellten Artikel erzielt.
8. Ermitteln Sie die maximale Bestellmenge eines Kunden (ausgegeben werden soll die Kundennummer und die maximale Bestellmenge, die pro Artikel der Kunde bestellt hat).
9. Man ermittle für jeden Kunden die Summe aller Bestellmengen der von ihm bestellten Artikel. Es sollen aber nur jene Kunden ausgegeben werden, deren Gesamtbestellmenge größer als 1000 ist.
10. Man finde für jeden Artikel des Herstellers Panasonic die Anzahl der Kunden von denen er bestellt wurde.
11. (👁️) Man finde für jeden bestellten Artikel die Artikelnummer, die Artikelbezeichnung sowie die Nummer und den Namen desjenigen (derjenigen) Kunden, die für diesen Artikel die höchste Bestellmenge bei einer Bestellung erzielten.

## Selbstverbund (Selfjoin)

### Selbstverbund

Der Verbund kann nicht nur auf zwei oder mehrere Tabellen angewendet werden, sondern auch auf eine einzige Tabelle. In diesem Fall wird die Tabelle *mit sich selbst verknüpft*, wobei eine einzige Spalte dieser Tabelle gewöhnlich mit sich selbst verglichen wird.

### Aliasnamen

Wird eine Tabelle mit sich selbst verknüpft, so erscheint ihr Name doppelt in der **FROM**-Klausel einer **SELECT**-Anweisung. Damit der Tabellename in beiden Fällen unterschieden werden kann, müssen *Aliasnamen* benutzt werden. Gleichzeitig müssen alle Spalten dieser Tabelle in der **SELECT**-Anweisung spezifiziert sein und zwar mit dem entsprechenden Aliasnamen als Präfix.



Man finde alle Kunden, die in demselben Wohnort wohnen.

```
SELECT DISTINCT k1.knummer, k1.kfname, k1.kname, k1.kort,
FROM kunden k1, kunden k2
WHERE k1.kort = k2.kort
AND k1.knummer <> k2.knummer;
```

In der **FROM**-Klausel dieses Beispiels sind zwei Aliasnamen für die Tabelle kunden eingeführt worden. Die erste Erscheinung der Tabelle hat den Aliasnamen k1 und die zweite den Aliasnamen k2. Die erste Bedingung in der **WHERE**-Klausel definiert die Join-Spalten, während die zweite Bedingung notwendig ist, damit ein Kunde nicht mit sich selbst verknüpft wird, denn der Kunde kommt aus demselben Ort „wie er selbst“.

Man finde alle Paare von Artikeln, deren Verkaufspreis sich dem Betrage nach um weniger als 30 Euro unterscheidet.

```
SELECT r1.rnummer, r1.rbezeichnung, r1.rvpreis,
r2.rnummer, r2.rbezeichnung, r2.rvpreis
FROM artikel r1, artikel r2
WHERE ABS(r1.rvpreis - r2.rvpreis) < 30
AND r1.rnummer <> r2.rnummer;
```



1. Man finde alle Artikel, für die ein anderer Artikel desselben Herstellers mit demselben Einkaufspreis existiert.
2. Für welche Aufträge existiert ein anderer Auftrag mit demselben Auftragsdatum vom selben Kunden?
3. Welche Kunden haben dieselben Artikel bestellt?

4. Man finde alle diejenigen Aufträge, für die ein anderer Auftrag mit derselben Anzahl von Auftragspositionen existiert (verwenden Sie anstelle des Selbstverbundes eine korrelierte Unterabfrage).
5. Man finde alle diejenigen Kunden, die älter sind als der Kunde mit der Nummer 4711 (kann auch ohne Selbstverbund gelöst werden).

## Unterabfragen und Verbund

In diesem Abschnitt werden Beispiele angeführt, wo Unterabfragen und Verbund gemeinsam in einer **SELECT**-Anweisung verwendet werden.

Man finde den (die) Artikel mit maximalem Umsatz (Preis mal Menge) pro Bestellung, und man zeige seine Nummer und Bezeichnung sowie den errechneten Umsatzwert an.

```
SELECT p1.rnummer, rbezeichnung, rvpreis * pmenge
FROM positionen p1, artikel r1
WHERE p1.rnummer = r1.rnummer AND rvpreis * pmenge >= ALL
(SELECT rvpreis * pmenge
FROM positionen p2, artikel r2
WHERE p2.rnummer = r2.rnummer);
```

Durch die Unterabfrage wird über die Artikelnummer ein Verbund der Tabellen `positionen` und `artikel` hergestellt. Als Ergebnis der Unterabfrage wird die Tabelle der Werte ermittelt, die aus dem Produkt der Werte der Spalten von `rvpreis` und `pmenge` errechnet werden. In der äußeren **SELECT**-Anweisung wird in der **WHERE**-Klausel durch den Operator **ALL** auf diese Werte Bezug genommen.

Man finde die Daten der Auftragspositionen aller derjenigen Artikel mit der Artikelnummer 44, 55 oder 66, deren Umsatzwerte betragsmäßig kleiner sind, als der maximale Umsatz in einer Auftragsposition des Artikels mit der Nummer 22.

```
SELECT p1.rnummer, pmenge, pmenge * rvpreis
FROM positionen p1, artikel r1
WHERE p1.rnummer = r1.rnummer
AND p1.rnummer IN (44, 55, 66)
AND pmenge * rvpreis < ANY
(SELECT pmenge * rvpreis
FROM positionen p2, artikel r2
WHERE p2.rnummer = r2.rnummer AND p2.rnummer = 22);
```

Durch die Unterabfrage werden die Umsatzwerte für den Artikel mit der Nummer 22 ermittelt. Anschließend wird ein Verbund von `positionen` und `artikel` über die Artikelnummer (eingeschränkt auf die Artikelnummern 44, 55 und 66) durchgeführt, wobei für die resultierenden Tabellenzeilen der jeweilige (durch Multiplikation von `pmenge` und `rvpreis`) gebildete Umsatzwert kleiner sein muss als jeweils einer der Ergebniswerte der Unterabfrage.

1. Man finde die Umsatzwerte aller Auftragspositionen der Artikel mit den Nummern 44, 55 oder 66, die jeweils betragsmäßig größer sind als alle Einzelumsätze des Artikels mit der Nummer 22.
2. Man finde die Nummern und Namen aller Kunden, die keinen Luxusartikel bestellt haben (also auch die Namen der Kunden die zurzeit nichts bestellt haben).

### Unterabfragen, Verbund



## Inner- und Outerjoins

Wenn ein Join auf zwei Tabellen ausgeführt wird, dann fallen automatisch all jene Datensätze weg und kommen nicht ins Ergebnis, welche mit keinem Datensatz der anderen Tabelle joinen.

### Inner- und Outer- joins



Die folgende Anweisung realisiert einen Verbund zwischen den Tabellen `kunden` und `auftraege`:

```
SELECT *
 FROM kunden k, auftraege a
 WHERE k.knummer = a.knummer;
```

Dabei werden aus dem Ergebnis all jene Kunden entfernt, die keinen Auftrag gemacht haben.

Man finde zu jedem Kunden die Nummern aller von ihm bestellten Artikel (auch die Kunden, welche nichts bestellt haben, sollen ausgegeben werden).



```
SELECT DISTINCT k.knummer, kfname, kname, p.rnummer
 FROM kunden k, auftraege a, positionen p
 WHERE k.knummer = a.knummer AND a.anummer = p.anummer
 ORDER BY 1, 4;
```

### Innerequijoin

Diese `SELECT`-Anweisung gibt aber nicht das wieder, was zu suchen war. Diese `SELECT`-Anweisung ermittelt nur die Kundennummern, die Kundennamen und die Artikelnummern von jedem Kunden, der mindestens einen Artikel bestellt hat. Da nur die Daten über jene Kunden gesucht werden, die einen Auftrag gemacht haben, wird dieses Join als *Innerequijoin* bezeichnet.



Dieser Innerequijoin kann in SQL auch auf folgende Art definiert werden:

```
SELECT DISTINCT k.knummer, kfname, kname, rnummer
 FROM kunden k INNER JOIN auftraege a
 ON k.knummer = a.knummer
 INNER JOIN positionen p
 ON a.anummer = p.anummer
 ORDER BY 1, 4;
```

Die Ausgangsfragestellung aber kann durch Erweitern der `SELECT`-Anweisung beantwortet werden:

```
SELECT DISTINCT k.knummer, kfname, kname, p.rnummer
 FROM kunden k, auftraege a, positionen p
 WHERE k.knummer = a.knummer AND a.anummer = p.anummer
 UNION
 SELECT knummer, kfname, kname, NULL
 FROM kunden
 WHERE knummer NOT IN
 (SELECT knummer
 FROM auftraege)
 ORDER BY 1, 4;
```

In SQL kann diese komplizierte Vereinigung auch durch folgende Anweisung umschrieben werden:

```
SELECT DISTINCT k.knummer, kfname, kname, rnummer
 FROM kunden k LEFT OUTER JOIN auftraege a
 ON k.knummer = a.knummer
 LEFT OUTER JOIN positionen p
 ON a.anummer = p.anummer
 ORDER BY 1, 4;
```

### Left- und Right- Outerequijoin

Diese Form des Equijoins wird als *Left-Outerequijoin* oder kurz *Left-Outerjoin* bezeichnet. Dabei kann nicht nur `LEFT` sondern auch `RIGHT` als Option des Outerjoins angegeben werden:

**LEFT** hier werden alle Zeilen der *linken* am Join beteiligten Tabelle ins Ergebnis mit aufgenommen, auch jene die mit keiner Zeile der rechten am Join beteiligten Tabelle joinen. Wenn man

**RIGHT** verwendet, so werden standardmäßig alle Zeilen der rechts am Join beteiligten Tabelle ins Ergebnis aufgenommen.

Spalten, welche keine Werte erhalten, werden beim Outerjoin mit **NULL** aufgefüllt.

Zwischen den Grundgesamtheiten `kunden.knummer` und `auftraege.knummer` besteht eine Teilmengenrelation: die Grundgesamtheit von `auftraege.knummer` ist eine Teilmenge der Grundgesamtheit `kunden.knummer`.

Wenn wir annehmen, dass es Positionen gäbe, die keinem Auftrag und somit auch keinen Kunden zugeordnet sind – was in unserer Datenbank bereits durch die Definition der Fremdschlüssel unterbunden wird – dann könnte man durch folgende Anweisung alle Bestellpositionen mit ihren Kunden suchen, wobei jene Positionen die keinen Auftrag und/oder Kunden zugewiesen haben die fehlenden Werte automatisch mit **NULL** aufgefüllt bekommen:

```
SELECT DISTINCT k.knummer, kfname, kname, rnummer
FROM kunden k RIGHT OUTER JOIN auftraege a
ON k.knummer = a.knummer
RIGHT OUTER JOIN positionen p
ON a.anummer = p.anummer
ORDER BY 1, 4;
```



Bei der Formulierung eines Joins muss man genau wissen, welche Beziehung zwischen den Join-Spalten besteht. Diese Beziehung sollte vor der Formulierung eines Joins bestimmt werden, um eventuelle Fehler zu vermeiden.



1. Man finde für jeden Kunden die Anzahl der verschiedenen Artikel, die von ihm bestellt worden sind.
2. Man finde für jeden Kunden die Kundennummer, den Kundennamen und die Gesamtsumme aller Stückzahlen der von ihm bestellten Artikel.
3. Man finde für jeden Kunden die Kundennummer, den Kundennamen und die Liste der Artikelnummern derjenigen Artikel, die von ihm bestellt worden sind.
4. Man finde für jeden Kunden die Monate an denen er Bestellungen gemacht hat. Hat ein Kunde überhaupt keine Bestellung gemacht, so soll er auch ausgegeben werden.
5. Formulieren Sie obige Abfrage mit einem Right-Outerjoin.



## Korrelierte Unterabfragen

Eine Unterabfrage wird dann *korreliert* genannt, falls die innere **SELECT**-Anweisung eine Spalte enthält, deren Werte in der äußeren **SELECT**-Anweisung festgelegt sind.

Die beiden folgenden Beispiele zeigen, wie dieselbe Aufgabe mit Hilfe einer einfachen und einer korrelierten Unterabfrage gelöst werden kann.

Man finde die Bezeichnung aller Artikel, die im Auftrag mit der Nummer 13 bestellt wurden.

```
SELECT rbezeichnung
FROM artikel
WHERE rnummer IN
(SELECT rnummer
FROM positionen
WHERE anummer = 13);
```



Dieses Beispiel zeigt eine einfache Unterabfrage, in der zunächst die innere **SELECT**-Anweisung, unabhängig von der äußeren **SELECT**-Anweisung, berechnet wird und als Ergebnis die Werte für den **IN**-Operator geliefert werden. Diese Werte werden dann zur Bildung des endgültigen Ergebnisses benutzt.

## Korrelierte Unterabfragen



Hier wird die Frage vom vorigen Beispiel mit einer korrelierten Unterabfrage beantwortet.

```
SELECT rbezeichnung
FROM artikel
WHERE 13 IN
 (SELECT anummer
 FROM positionen
 WHERE positionen.rnummer = artikel.rnummer);
```

Hier kann die innere **SELECT**-Anweisung nicht in einem Schritt ausgewertet werden, weil sie die Spalte `artikel.rnummer` beinhaltet, die der Tabelle `artikel` aus der äußeren **SELECT**-Anweisung gehört. Die innere **SELECT**-Anweisung ist also von einer Variablen abhängig, die in der äußeren **SELECT**-Anweisung berechnet sein muss.

## Abarbeitung

Im zweiten Beispiel, wie dies bei der korrelierten Unterabfrage grundsätzlich der Fall ist, untersucht das System zunächst die erste Zeile der Tabelle `artikel` und vergleicht die erste Artikelnummer mit der Spalte `positionen.rnummer` in der inneren **SELECT**-Anweisung. Die innere **SELECT**-Anweisung liefert nach diesem Vergleich als Ergebnis eine Wertemenge für die Spalte `anummer`. Enthält diese Wertemenge den Datenwert 13, so wird der erste Artikel in die Ergebnismenge aufgenommen. Dann wird die zweite Zeile der Tabelle `artikel` auf dieselbe Weise untersucht usw.



Man finde alle Kunden, die an demselben Wohnort wohnen (in diesem Beispiel wird eine korrelierte Unterabfrage gezeigt, in der die Tabelle mit sich selbst verknüpft wird).

```
SELECT k1.knummer, k1.kfname, k1.kname, k1.kort
FROM kunden k1
WHERE k1.kort IN
 (SELECT k2.kort
 FROM kunden k2
 WHERE k1.knummer <> k2.knummer);
```

Alle Kunden sind durch ihre Wohnorte in verschiedene Gruppen unterteilt. Man finde Nummer, Namen und Wohnort aller Kunden, die nicht die kleinste Nummer in ihrer Gruppe haben (in diesem Beispiel wird eine korrelierte Unterabfrage mit der Aggregatfunktion **MIN** dargestellt).

```
SELECT k1.knummer, k1.kfname, k1.kname, k1.kort
FROM kunden k1
WHERE k1.knummer >
 (SELECT MIN(k2.knummer)
 FROM kunden k2
 WHERE k1.kort = k2.kort);
```

Wie bereits erwähnt, darf die innere **SELECT**-Anweisung bei der Verwendung eines Vergleichsoperators in der Unterabfrage, wie in einem der vorigen Beispiele, nur einen Datenwert liefern. Falls das Ergebnis der inneren **SELECT**-Anweisung mehrere Datenwerte beinhaltet, muss der **IN**-Operator benutzt werden.



1. Man finde mittels einer korrelierten Unterabfrage alle Wohnorte, in denen mehr als ein Kunde wohnt.
2. Man finde für jeden Artikel diejenigen Auftragspositionen und deren Datum, in denen die bestellte Menge größer ist, als die durchschnittliche Bestellmenge des betreffenden Artikels.
3. Man finde für jeden Kunden diejenigen Aufträge, in denen die Anzahl bestellter Stücke kleiner ist, als die durchschnittliche Anzahl der bestellten Stücke in irgendeinem Auftrag des betreffenden Kunden.