

Datenbankadministration

Microsoft SQL Server 2017

Florian Weidinger

1. Ausgabe: 05.03.2020
Letzte Änderung: 26. März 2020

Inhaltsverzeichnis

| | |
|---|------------|
| I. SQL | 1 |
| 1. Einstieg in SQL | 1-1 |
| 1.1. Einführung | 1-2 |
| 1.2. Grundlegende Operationen in SQL | 1-4 |
| 1.2.1. Mengenoperationen | 1-4 |
| 1.2.2. Die Projektion | 1-6 |
| 1.2.3. Die Selektion | 1-6 |
| 1.2.4. Kartesisches Produkt | 1-7 |
| 1.2.5. Die Join-Operation im Allgemeinen | 1-7 |
| 1.2.6. Inner-Joins | 1-8 |
| 1.2.7. Outer-Joins | 1-10 |
| 1.3. Auswahlabfragen mit SQL | 1-10 |
| 1.3.1. Schema einer Auswahlabfrage | 1-10 |
| 1.3.2. Der * als Joker | 1-12 |
| 1.3.3. Arithmetische Ausdrücke | 1-13 |
| 1.3.4. NULL Werte | 1-14 |
| 1.3.5. Verkettung von Zeichenketten | 1-16 |
| 1.3.6. Spaltenaliasnamen | 1-17 |
| 1.4. Einige Konventionen zu SQL | 1-18 |
| 2. Selektieren und Sortieren | 2-1 |
| 2.1. Selektieren von Zeilen: Die WHERE-Klausel | 2-2 |
| 2.1.1. Relationale Operatoren | 2-3 |
| 2.1.2. Logische Verknüpfung von Ausdrücken | 2-8 |
| 2.2. Festlegen einer Sortierung | 2-10 |
| 2.2.1. Die ORDER BY Klausel | 2-10 |
| 2.2.2. Auf- und absteigendes Sortieren | 2-12 |
| 2.3. Übungen - Selektieren und Sortieren | 2-13 |
| 2.4. Lösungen - Selektieren und sortieren | 2-17 |
| 3. Funktionen | 3-1 |
| 3.1. Der Begriff der Funktion | 3-2 |
| 3.2. Zeichenkettenfunktionen | 3-3 |
| 3.2.1. Groß- oder Kleinschreibung - UPPER, LOWER, INITCAP | 3-3 |
| 3.2.2. Zeichenketten bearbeiten | 3-5 |
| 3.3. Arithmetische Funktionen | 3-11 |
| 3.3.1. FROM oder nicht FROM, das ist hier die Frage | 3-11 |
| 3.3.2. Arithmetische Funktionen anwenden | 3-12 |
| 3.4. Datumsfunktionen | 3-15 |
| 3.4.1. Datumswerte | 3-15 |
| 3.4.2. Datumsarithmetik in Oracle | 3-16 |

| | | |
|-----------|---|------------|
| 3.4.3. | Datumsarithmetik in MS SQL Server | 3-19 |
| 3.5. | Sonstige Funktionen | 3-20 |
| 3.6. | Datentypen | 3-22 |
| 3.6.1. | Numerische Datentypen | 3-23 |
| 3.6.2. | Zeichendatentypen | 3-24 |
| 3.6.3. | Datums- und Zeittypen | 3-24 |
| 3.7. | Konvertierung von Datentypen | 3-25 |
| 3.7.1. | Implizite Datentypkonvertierung | 3-25 |
| 3.7.2. | Explizite Datentypkonvertierung | 3-28 |
| 3.8. | Übungen - Funktionen | 3-33 |
| 3.9. | Lösungen - Funktionen | 3-36 |
| 4. | Erweiterte Datenselektion | 4-1 |
| 4.1. | Der Inner Join | 4-2 |
| 4.1.1. | Die ON-Klausel | 4-2 |
| 4.1.2. | Tabellenaliasnamen | 4-3 |
| 4.1.3. | Die USING-Klausel (Nur Oracle) | 4-4 |
| 4.1.4. | Der Natural-Join (Nur Oracle) | 4-5 |
| 4.1.5. | Die Theta-Style Syntax | 4-6 |
| 4.1.6. | Mehr als zwei Tabellen verknüpfen | 4-7 |
| 4.2. | Outer Joins | 4-8 |
| 4.2.1. | Left- und Right-Outer-Join | 4-8 |
| 4.2.2. | Der Full Outer Join | 4-12 |
| 4.3. | Spezielle Joins | 4-12 |
| 4.3.1. | Der Self-Join | 4-12 |
| 4.3.2. | Non-Equi-Joins | 4-16 |
| 4.4. | Mengenoperationen | 4-16 |
| 4.4.1. | Voraussetzungen zur Nutzung der SET-Operatoren | 4-17 |
| 4.4.2. | Die SET-Operatoren | 4-17 |
| 4.5. | Übungen - Erweiterte Datenselektion | 4-22 |
| 4.6. | Lösungen - Erweiterte Datenselektion | 4-27 |
| 5. | Gruppenfunktionen | 5-1 |
| 5.1. | Die GROUP BY-Klausel | 5-2 |
| 5.2. | Die Aggregatfunktionen | 5-3 |
| 5.2.1. | Die Funktion COUNT | 5-4 |
| 5.2.2. | Die Funktion SUM | 5-5 |
| 5.2.3. | Die Funktion AVG | 5-5 |
| 5.2.4. | Die Funktionen MIN und MAX | 5-7 |
| 5.2.5. | Gruppierungen mit mehreren Ebenen | 5-8 |
| 5.3. | Gruppierte Abfragen filtern | 5-8 |
| 5.3.1. | Die WHERE-Klausel | 5-8 |
| 5.3.2. | Die HAVING-Klausel | 5-9 |
| 5.4. | Die Abarbeitungsreihenfolge des SELECT-Statements | 5-11 |
| 5.5. | Übungen - Gruppenfunktionen | 5-12 |
| 5.6. | Lösungen - Gruppenfunktionen | 5-15 |
| 6. | Unterabfragen (Subqueries) | 6-1 |
| 6.1. | Grundsätzliches zu Unterabfragen | 6-2 |
| 6.1.1. | Was sind Unterabfragen? | 6-2 |

| | | |
|-----------|--|------------|
| 6.1.2. | Wann sind Unterabfragen notwendig? | 6-2 |
| 6.1.3. | Regeln für Unterabfragen | 6-4 |
| 6.1.4. | Arten von Unterabfragen | 6-4 |
| 6.2. | Skalare Unterabfragen (Scalar Subqueries) | 6-5 |
| 6.2.1. | Wo können skalare Unterabfragen stehen? | 6-5 |
| 6.2.2. | Fehlerquellen in skalaren Unterabfragen | 6-6 |
| 6.3. | Einspaltige Unterabfragen | 6-7 |
| 6.3.1. | Einspaltige Unterabfragen in WHERE- und HAVING-Klausel | 6-7 |
| 6.3.2. | Existenzprüfungen | 6-8 |
| 6.4. | Inlineviews / Derived Tables | 6-10 |
| 6.5. | Top N Analysen | 6-11 |
| 6.5.1. | Die Top N Analyse in Oracle | 6-11 |
| 6.5.2. | Die Top N Analyse in MS SQL Server | 6-14 |
| 6.6. | Pivot-Tabellen | 6-14 |
| 6.6.1. | Der PIVOT-Operator (Oracle) | 6-14 |
| 6.6.2. | Der PIVOT-Operator (MS SQL Server) | 6-18 |
| 6.7. | Übungen - Unterabfragen | 6-21 |
| 6.8. | Lösungen - Unterabfragen | 6-25 |
| 7. | Data Manipulation Language (DML) | 7-1 |
| 7.1. | Die DML-Anweisungen | 7-2 |
| 7.1.1. | Datensätze einfügen - Die INSERT-Anweisung | 7-2 |
| 7.1.2. | Datensätze ändern - Die UPDATE-Anweisung | 7-5 |
| 7.1.3. | Datensätze löschen - Die DELETE-Anweisung | 7-8 |
| 7.2. | Das Transaktionskonzept - COMMIT und ROLLBACK | 7-9 |
| 7.2.1. | Beginn und Ende einer Transaktion | 7-10 |
| 7.2.2. | Eine Transaktion erfolgreich abschließen | 7-11 |
| 7.2.3. | Eine Transaktion rückgängig machen | 7-14 |
| 8. | Data Definition Language | 8-1 |
| 8.1. | Tabellen erstellen und verwalten | 8-2 |
| 8.1.1. | Namenskonventionen und Einschränkungen | 8-2 |
| 8.1.2. | CREATE TABLE - Tabellen erstellen | 8-4 |
| 8.1.3. | CREATE TABLE AS... (CTAS) | 8-5 |
| 8.1.4. | ALTER TABLE - Tabellen verändern | 8-6 |
| 8.1.5. | DROP TABLE - Tabellen löschen | 8-11 |
| 8.1.6. | TRUNCATE TABLE - Tabellen leeren | 8-11 |
| 8.2. | Views erstellen verwalten | 8-13 |
| 8.2.1. | Was sind Views? | 8-13 |
| 8.2.2. | Views erstellen | 8-13 |
| 8.2.3. | Views und DML | 8-18 |
| 8.2.4. | Views ändern | 8-21 |
| 8.2.5. | Views löschen | 8-22 |
| 8.3. | Übungen - Erstellen von Views | 8-23 |
| 8.4. | Lösungen - Erstellen von Views | 8-25 |
| 9. | Constraints | 9-1 |
| 9.1. | Was sind Constraints | 9-2 |
| 9.2. | Die Constraints | 9-2 |
| 9.2.1. | Das CHECK-Constraint | 9-3 |

| | | |
|--------|---|------|
| 9.2.2. | Das NOT NULL-Constraint | 9-4 |
| 9.2.3. | Das UNIQUE-Constraint | 9-5 |
| 9.2.4. | Das PRIMARY KEY-Constraint | 9-6 |
| 9.2.5. | Das FOREIGN KEY-Constraint | 9-7 |
| 9.2.6. | Das SQL Server DEFAULT-Constraint | 9-10 |
| 9.3. | Constraints umbenennen und löschen | 9-10 |
| 9.3.1. | Constraints umbenennen | 9-10 |
| 9.3.2. | Constraints löschen | 9-11 |
| 9.3.3. | Standardwerte in SQL Server löschen | 9-11 |

Teil I.

SQL

1. Einstieg in SQL

Inhaltsangabe

| | | |
|------------|--|-------------|
| 1.1 | Einführung | 1-2 |
| 1.2 | Grundlegende Operationen in SQL | 1-4 |
| 1.2.1 | Mengenoperationen | 1-4 |
| 1.2.2 | Die Projektion | 1-6 |
| 1.2.3 | Die Selektion | 1-6 |
| 1.2.4 | Kartesisches Produkt | 1-7 |
| 1.2.5 | Die Join-Operation im Allgemeinen | 1-7 |
| 1.2.6 | Inner-Joins | 1-8 |
| 1.2.7 | Outer-Joins | 1-10 |
| 1.3 | Auswahlabfragen mit SQL | 1-10 |
| 1.3.1 | Schema einer Auswahlabfrage | 1-10 |
| 1.3.2 | Der * als Joker | 1-12 |
| 1.3.3 | Arithmetische Ausdrücke | 1-13 |
| 1.3.4 | NULL Werte | 1-14 |
| 1.3.5 | Verkettung von Zeichenketten | 1-16 |
| 1.3.6 | Spaltenaliasnamen | 1-17 |
| 1.4 | Einige Konventionen zu SQL | 1-18 |

1.1. Einführung

SQL¹ ist die auf dem Markt etablierte Manipulations- und Abfragesprache für relationale Datenbankmanagementsysteme. Es kam erstmals mit Oracle V2 1979 auf den Markt und wurde durch die beiden Institute ANSI (1986) und ISO (1987) standardisiert. 1989 wurden die Arbeitsergebnisse der beiden Gremien im „SQL-89“ bzw. „SQL-1“ Standard zusammengefasst. Eine grundlegende Überarbeitung erfolgte 1992 mit dem „SQL-2“ Standard, der auch als „SQL-92“ Standard bezeichnet wird, welcher im Wesentlichen auch heute noch Anwendung findet. Der aktuell gültige SQL-Standard ist der „SQL-2003“ Standard (SQL:2003 ISO/IEC 9075).

- **1989 - SQL-89-Standard, SQL-1-Standard**

- DML (Data Manipulation Language): SELECT, INSERT, UPDATE, DELETE
- DDL (Data Definition Language): Tabellen, Indizes, Sichten, GRANT/REVOKE
- Transaktionen: BEGIN, COMMIT, ROLLBACK
- Cursor

- **1992 - SQL-92-Standard, SQL-2-Standard**

- DDL (Data Definition Language): BLOB, VARCHAR, DATE, TIME, TIMESTAMP, BOOLEAN
- DML (Data Manipulation Language): INNER- und OUTER-Join, SET-Operatoren
- Transaktionen: SET TRANSACTION
- Cursor
- Constraints
- Systemtabellen

- **1999 - SQL-99-Standard, SQL-3-Standard**

- DML (Data Manipulation Language): Benutzerdefinierte Datentypen, Rollen
- DDL (Data Definition Language): Rekursive Abfragen
 - * Intermediate Level: CASCADE DELETE
 - * Full Level: CASCADE UPDATE
- Constraints: Trigger
- Call Level Interface: ODBC, JDBC, OLE DB
- Persistent Storage Modules: Stored Procedures

¹SQL = Structured Query Language

- **2003 - SQL-2003-Standard**

- DML (Data Manipulation Language): MERGE
- DDL (Data Definition Language): Identitätsattribute
- Schemata: Informations- und Definitions Schema
- SQL/XML: Einbettung von XML in die Datenbank
- Mediums: Zugriff auf externe Daten

Seit Ende der 80er-Jahr wird SQL, in Teilen, von nahezu allen relationalen Datenbankmanagementsystemen unterstützt, wobei jeder Hersteller seine eigenen Erweiterungen, zusätzlich zum Standard einfügt. Dies führt dazu, dass es eine Vielzahl von SQL-Dialekten gibt, welche sehr unterschiedlich sein können.

Die Syntax von SQL ist stark an die englische Sprache angelehnt und somit relativ einfach zu erlernen. Es stellt eine Reihe von Befehlen zur Verfügung, welche sich in vier Kategorien einteilen lassen:

- Abfragesprache (Query-Language)
- Datenmanipulationssprache (DML Data Manipulation Language)
- Datendefinitionssprache (DDL Data Definition Language)
- Datenkontrollsprache (DCL Data Control Language)

Durch die weitreichende Standardisierung von SQL, ist es möglich Anwendungsprogramme zu erstellen, welche eingeschränkt unabhängig vom verwendeten DBMS sind. Dies gilt zumindest für die Teile des SQL-Standards, die von allen Anbietern implementiert werden.

Im Gegensatz zu Programmiersprachen, wie z. B. Turbo Pascal, C++ oder Java, handelt es sich bei SQL um eine „Deklarative Programmiersprache“. Dies bedeutet, dass während in einer Sprache, wie C++, der *genaue Weg zur Lösung eines Problems* beschrieben wird, in SQL der Programmierer das *gewünschte Ergebnis so genau wie möglich* beschreiben muss. Den Weg dorthin findet SQL selbst, so dass sich der Programmierer nicht darum kümmern braucht, wie SQL zu seinem Ergebnis kommt.

Problematisch an diesem Ansatz kann sein, dass SQL immer ein Ergebnis liefert, solange die Syntax der Anweisung korrekt ist. Das gefundene Ergebnis muss jedoch nicht unbedingt das Gewollte sein und kann sich, in sehr geringen Details, vom Richtigen unterscheiden, was eine Fehlersuche bzw. das Bemerkens eines Fehlers sehr schwierig gestaltet.

In diesem Skript werden Teile der beiden SQL-Dialekte, von Oracle 11g R2 und Microsoft SQL-Server 2008 R2 bzw. SQL Server 2012, anhand von praktischen Beispielen, näher erleutert.

1.2. Grundlegende Operationen in SQL

SQL basiert auf einer Relationalen Algebra. Im Sinne der Datenbanktheorie, ist dies eine formale Sprache, die es ermöglicht, Suchoperationen auf einem relationalen Schema durchzuführen. Mit ihrer Hilfe hat man die Möglichkeit, Relationen zu verknüpfen, zu reduzieren und komplexe Informationen zu ermitteln.

Die Relationale Algebra stellt verschiedene Operationen bereit, welche, mit Hilfe von SQL, umgesetzt werden können.

- Mengenoperationen
 - Vereinigung (UNION)
 - Schnittmenge (INTERSECT)
 - Differenz (MINUS)
- Projektion
- Selektion
- Kreuzprodukt (Kartesisches Produkt)
- Join

Für diese Unterrichtsunterlage hat die Relationale Algebra nur insofern eine Bedeutung, dass sie die theoretische Grundlage für die Sprache SQL darstellt.

1.2.1. Mengenoperationen

Die Vereinigung

Die Vereinigung ist eine Operation, bei der alle Zeilen zweier Relationen zu einer neuen Relation zusammengeführt werden. Dargestellt wird die Vereinigung in der Relationalen Algebra mit: $R_1 \cup R_2$ (R_1 vereinigt mit R_2).

Das folgende Beispiel veranschaulicht die Funktionsweise dieser Mengenoperation. Die beiden Relationen R_1 und R_2 werden miteinander vereinigt. Als Ergebnis entsteht eine neue Relation, die alle Zeilen der beiden zugrunde gelegten Relationen enthält, mit Ausnahme der Zeilen, welche redundant vorkommen.

Dies betrifft hier die Zeile „1 2 3 4“, in der Relation R_2 . Eine gleiche Zeile existiert bereits in der Relation R_1 . Somit wird diese Zeile nur einmal im Ergebnis erscheinen.

Tabelle 1.1.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

Tabelle 1.2.: R_2

| A | B | C | D |
|---|---|---|---|
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |
| 1 | 2 | 3 | 4 |

Tabelle 1.3.: $R_1 \cup R_2$

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |

Redundante Zeilen werden bei der Vereinigung zweier Relationen R_1 und R_2 aus dem Ergebnis entfernt!



Die Schnittmenge

Die Schnittmenge enthält als Ergebnis nur die Zeilen, die in den beiden Relationen R_1 und R_2 identisch sind. Die Darstellung erfolgt in der Relationalen Algebra mit: $R_1 \cap R_2$ (R_1 geschnitten mit R_2).

Tabelle 1.4.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

Tabelle 1.5.: R_2

| A | B | C | D |
|---|---|---|---|
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |
| 1 | 2 | 3 | 4 |

Tabelle 1.6.: $R_1 \cap R_2$

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |

Da in diesem Beispiel nur die Zeile „1 2 3 4“ in beiden Relationen R_1 und R_2 vorhanden ist, wird nur diese in der Ergebnisrelation $R_1 \cap R_2$ abgebildet.

Die Differenz

Die Differenz zweier Relationen R_1 und R_2 entspricht den Zeilen, die nur in der linken der beiden Relationen vorkommen. D. h., die Differenz ist, wie in der Arithmetik auch, nicht kommutativ ($R_1 \setminus R_2 \neq R_2 \setminus R_1$).

Tabelle 1.7.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

Tabelle 1.8.: R_2

| A | B | C | D |
|---|---|---|---|
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |
| 1 | 2 | 3 | 4 |

Tabelle 1.9.: $R_1 \setminus R_2$

| A | B | C | D |
|---|---|---|---|
| 5 | 6 | 7 | 8 |

Tabelle 1.10.: $R_2 \setminus R_1$

| A | B | C | D |
|---|---|---|---|
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |

1.2.2. Die Projektion

Die Projektion wählt Attribute aus einer Relation (Spalten aus einer Tabelle) aus, weshalb sie auch als *Attributbeschränkung* bezeichnet wird. Es ist dabei egal, ob alle Attribute oder nur eine Teilmenge der Attributmenge ausgewählt wird. In der Relationalen Algebra wird die Projektion mit $\pi_\beta(R_1)$ ausgedrückt, wobei β die Liste der gewählten Attribute bezeichnet. Hierzu ein paar Beispiele:

Tabelle 1.11.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

Tabelle 1.12.: $\pi_{(A,B)}(R_1)$

| A | B |
|---|---|
| 1 | 2 |
| 5 | 6 |

Tabelle 1.13.: $\pi_A(R_1)$

| A |
|---|
| 1 |
| 5 |

Tabelle 1.14.: $\pi_*(R_1)$

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |

1.2.3. Die Selektion

Die Selektion ermöglicht es ausgewählte Zeilen aus einer Relation in das Ergebnis aufzunehmen. Nicht erwünschte Zeilen werden einfach ausgeblendet. Dies geschieht mit einem „Selektionsausdruck“, einer einfachen Bedingung, die für jede einzelne Zeile geprüft wird. Mathematisch wird die Selektion als $\sigma_{\text{Ausdruck}}(R)$ dargestellt. σ (sigma) steht für Selektion.

Tabelle 1.15.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 0 | 1 | 2 |
| 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 0 |

Tabelle 1.16.: $\sigma_{(A=3)}\pi_{(A,B)}(R_1)$

| A | B |
|---|---|
| 3 | 4 |

Tabelle 1.17.: $\sigma_{(B \geq 4)}\pi_{(A,B,C)}(R_1)$

| A | B | C |
|---|---|---|
| 5 | 6 | 7 |
| 3 | 4 | 5 |
| 7 | 8 | 9 |

Tabelle 1.18.: $\sigma_{(A > 2 \wedge B \geq 4)}\pi_*(R_1)$

| A | B | C | D |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 0 |

Es ist möglich, einen Selektionsausdruck zu definieren, der eine leere Menge \emptyset zum Ergebnis hat. Dies wäre z. B. bei folgendem Ausdruck der Fall: $\sigma_{(A=4)}(\pi_*(R_1))$. Da in der Spalte A in keiner Zeile der Wert 4 vorkommt, ist das Ergebnis eine leere Menge.

1.2.4. Kartesisches Produkt

Das Kartesische Produkt $R_1 \times R_2$ entspricht der Multiplikation aller Zeilen zweier Relationen R_1 und R_2 , sofern alle Attribute unterschiedlich sind. Daraus folgt, dass die Resultatrelation die Kombination aller Zeilen aus R_1 und R_2 umfasst.

Tabelle 1.19.: R_1

| A | B |
|---|---|
| 1 | 2 |
| 5 | 6 |

Tabelle 1.20.: R_2

| C | D | E | F |
|---|---|---|---|
| 0 | 9 | 8 | 7 |
| 6 | 5 | 4 | 3 |
| 1 | 2 | 3 | 4 |

Tabelle 1.21.: $R_1 \times R_2$

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | 2 | 0 | 9 | 8 | 7 |
| 1 | 2 | 6 | 5 | 4 | 3 |
| 1 | 2 | 1 | 2 | 3 | 4 |
| 5 | 6 | 0 | 9 | 8 | 7 |
| 5 | 6 | 6 | 5 | 4 | 3 |
| 5 | 6 | 1 | 2 | 3 | 4 |

1.2.5. Die Join-Operation im Allgemeinen

Das Wort Join bezeichnet zu deutsch einen Verbund. Im Sinne der Relationalen Algebra ist das der Verbund der beiden hintereinander ausgeführten Operationen „Kartesisches Produkt“ und „Selektion“. Der Selektionsausdruck hat dabei die Form: $R_1 \theta R_2$, wobei θ (theta) einen geeigneten Vergleichsoperator ($=, \neq, >, <$ u. ä.) darstellt.

Die Relationale Algebra definiert den Join wie folgt:

$$R_1 \bowtie_{\text{Ausdruck}} R_2 := \{r_1 \cup r_2 \mid r_1 \in R_1 \wedge r_2 \in R_2 \wedge \text{Ausdruck}\}$$

Bedeutung:

- $R_1 \bowtie_{\text{Ausdruck}} R_2$: Verbund der beiden Relation R_1 und R_2
- $r_1 \cup r_2$: Vereinigung aller Zeilen aus R_1 und R_2 (Kreuzprodukt)
- $r_1 \in R_1$: r_1 ist eine Zeile aus der Relation R_1
- $r_2 \in R_2$: r_2 ist eine Zeile aus der Relation R_2
- \wedge : UND-Verknüpfung

Aus diesem Ausdruck lässt sich die folgende Formel ableiten:

$$R_1 \bowtie_{\text{Ausdruck}} R_2 := \sigma_{\text{Ausdruck}}(R_1 \times R_2)$$

1.2.6. Inner-Joins

Inner-Joins stellen die am häufigsten benötigte Form der Joins dar, so dass man hier von der „Standardform eines Joins“ sprechen könnte. Bei dieser Form des Joins wird das Kartesische Produkt zweier Relationen R_1 und R_2 gebildet und anschließend werden alle Zeilen eliminiert, die nicht der Join-Bedingung, auch Join-Prädikat genannt, genügen.

Es gibt zwei Unterarten des Inner-Join, den „Equi-Join“ und den „Non-Equi-Join“.

Equi-Join

Der Equi-Join ist eine spezielle Form des Inner-Join, der dadurch zustande kommt, dass im Join-Prädikat der Operator $=$ als Vergleichsoperator genutzt wird. Nur dann handelt es sich um einen Equi-Join.

$$\pi_{(R_1.A, R_1.B, R_2.C)}(R_1 \bowtie_{(R_1.A=R_2.A)} R_2 := \sigma_{(R_1.A=R_2.A)}(R_1 \times R_2))$$

Tabelle 1.22.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 0 | 1 | 2 |
| 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 0 |

Tabelle 1.23.: R_2

| A | B | C | D |
|---|---|---|---|
| 9 | 8 | 7 | 6 |
| 4 | 5 | 2 | 3 |
| 1 | 0 | 9 | 8 |
| 6 | 4 | 5 | 7 |
| 0 | 1 | 2 | 3 |

Tabelle 1.24.: $R_1 \bowtie R_2$

| R1.A | R1.B | R2.C |
|------|------|------|
| 1 | 2 | 9 |
| 9 | 0 | 7 |

In diesem Beispiel werden die Attributwerte der Spalten $R_1.A$ und $R_2.A$ miteinander verglichen und nur dort, wo eine Übereinstimmung gefunden wird, wird diese Zeile in die Ergebnisrelation aufgenommen.

Non-Equi-Join

Beim Non-Equi-Join handelt es sich um das Gegenteil zum Equi-Join. Sobald im Join-Prädikat nicht der $=$ -Operator genutzt wird, handelt es sich um einen Non-Equi-Join.

$$\pi_{R_2.*}(R_1 \bowtie_{(R_1.A > R_2.A \wedge R_1.B < R_2.B)} R_2)$$

Tabelle 1.25.: R_1

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 0 | 1 | 2 |
| 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 0 |

Tabelle 1.26.: R_2

| A | B | C | D |
|---|---|---|---|
| 9 | 8 | 7 | 6 |
| 5 | 4 | 3 | 2 |
| 1 | 0 | 9 | 8 |
| 7 | 6 | 5 | 4 |
| 3 | 2 | 1 | 0 |

Tabelle 1.27.: $R_1 \bowtie R_2$

| A | B | C | D |
|---|---|---|---|
| 7 | 6 | 5 | 4 |
| 3 | 2 | 1 | 0 |

1.2.7. Outer-Joins

Outer-Joins unterscheiden sich dadurch von Inner-Joins, dass bei ihnen alle Zeilen, entweder der linken oder der rechten Join-Seite, in die Ergebnisrelation aufgenommen werden. Es gibt:

- Left-Outer-Join: Alle Zeilen der linken Join-Seite werden in die Ergebnisrelation aufgenommen.
- Right-Outer-Join: Alle Zeilen der rechten Join-Seite werden in die Ergebnisrelation aufgenommen.
- Outer-Join / Full-Outer-Join: Alle Zeilen beider Seiten werden in die Ergebnisrelation aufgenommen.

Nicht vorhandene Werte werden dabei durch sogenannte NULL-Werte aufgefüllt.

1.3. Auswahlabfragen mit SQL

1.3.1. Schema einer Auswahlabfrage

Die Struktur einer SQL-Abfrage ist sehr simpel und besteht aus maximal 6 Klauseln, die in der hier gezeigten Reihenfolge angegeben werden müssen.

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```



Die kleinstmögliche SQL-Abfrage besteht aus den beiden Klausen **SELECT** und **FROM**.

Die **SELECT**-Klausel ist obligatorisch ² und immer die erste in der Reihenfolge. Ihr folgt eine Liste von Attributen und Ausdrücken, sowie die **FROM**-Klausel. Letztere ist dafür zuständig, die „Datenquellen“ festzulegen, also die Tabellen, auf die sich die Abfrage bezieht.

Listing 1.1: Eine einfache Auswahlabfrage in Oracle

```
SELECT Vorname , Nachname
FROM Mitarbeiter;
```

²obligatorisch = Zwingend notwendig

In **Beispiel 1.1** werden die Klauseln **SELECT** und **FROM** dazu benutzt, um die beiden Tabellenspalten **VORNAME** und **NACHNAME**, aus der Tabelle **MITARBEITER** abzufragen.

Das Ergebnis könnte sich so darstellen:

| VORNAME | NACHNAME |
|------------------------------|-----------------|
| Max | Winter |
| Sarah | Werner |
| Finn | Seifert |
| Sebastian | Schwarz |
| Tim | Sindermann |
| Peter | Müller |
| Emily | Meier |
| Dirk | Peters |
| ... | ... |
| 100 Zeilen ausgewählt | |



Die Symbole, an der Seite der Ergebnistabellen, haben folgende Bedeutung:



So wird das Ergebnis in einer Oracle 11g R2 Datenbank dargestellt.



So wird das Ergebnis in einem Microsoft SQL Server 2008 R2 dargestellt.



Oracle und Microsoft SQL Server stellen das Ergebnis auf die gleiche Art und Weise dar.

Eine solche Abfrage lässt sich nun um beliebige Spalten erweitern.

Listing 1.2: Eine einfache Auswahlabfrage in Oracle

```
SELECT Vorname , Nachname , Geburtsdatum , SozVersNr
FROM Mitarbeiter;
```



| VORNAME | NACHNAME | GEBURTSDATUM | SOZVERS NR |
|-----------|------------|--------------|----------------------|
| Max | Winter | 31.08.88 | D370941F-6CD-6C07977 |
| Sarah | Werner | 03.11.77 | 18FE2247-C53-7EAD1ED |
| Finn | Seifert | 17.10.85 | C973A840-B92-C5B97CD |
| Sebastian | Schwarz | 27.06.92 | E8F8587D-CBA-0F28A80 |
| Tim | Sindermann | 11.01.80 | 703838BD-E9C-BD118A6 |

100 Zeilen ausgewählt

1.3.2. Der * als Joker

In verschiedenen Kartenspielen gibt es Karten, die als Joker dienen. Solche „Universalkarten“ können sich, in der richtigen Situation ausgespielt, als sehr nützlich erweisen. Auch in SQL gibt es einen Joker, den Stern *. Er ist immer dann dienlich, wenn man die Spaltenstruktur einer Tabelle nicht kennt oder einfach alle Spalten ausgeben möchte.

Listing 1.3: Der * als Joker

```
SELECT *
FROM Bankfiliale;
```



| BANKFILIALE_ID | STRASSE | HAUSNUMMER | PLZ | ORT |
|----------------|----------------|------------|-------|--------------|
| 1 | Poststraße | 1 | 06449 | Aschersleben |
| 2 | Markt | 5 | 06449 | Aschersleben |
| 3 | Goethestraße | 4 | 39240 | Calbe |
| 4 | Lessingstraße | 1 | 06406 | Bernburg |
| 5 | Schillerstraße | 7 | 39240 | Barby |
| 6 | Kirchstraße | 8 | 39444 | Hecklingen |
| 7 | Ringstraße | 10 | 06420 | Könnern |

21 Zeilen ausgewählt



Der Stern * dient, in der SELECT-Liste einer Auswahlabfrage, als Platzhalter, stellvertretend für alle Spalten einer Tabelle.

Sollen alle Spalten, bis auf eine einzige angezeigt werden, kann der Stern leider nicht weiterhelfen. Eine Formulierung wie „* -1“ ist nicht möglich. In einem solchen Fall müssen alle Spalten, bis auf die betreffende angegeben werden.

1.3.3. Arithmetische Ausdrücke

SQL ist, wie viele andere Programmiersprachen auch, in der Lage arithmetische Ausdrücke zu berechnen. Diese können nicht nur in der SELECT-Liste des SQL-Statements, sondern auch in verschiedenen anderen Klauseln, vorkommen. Die Syntax solcher Ausdrücke unterscheidet sich in Oracle und SQL Server nicht.

Tabelle 1.28.: Arithmetische Operatoren in Oracle und SQL Server

| (Operator) | (Bedeutung) |
|------------|-------------------------|
| + | Addition |
| - | Subtraktion |
| * | Multiplikation |
| / | Division |
| % | Modulo (Nur SQL-Server) |
| . | Dezimaltrennzeichen |

Beispiel 1.4 zeigt ein einfaches Anwendungsbeispiel. Für die Entscheidung über die Gehaltserhöhung der Mitarbeiter, ist es notwendig, im Vorfeld eine Auswertung zu erstellen, die zeigt, wie sich die Erhöhung von 2,5 % auf die aktuellen Gehälter auswirkt.

Listing 1.4: Arithmetische Ausdrücke in SQL

```
SELECT Mitarbeiter_ID, Nachname, Gehalt, Gehalt * 1.025
FROM Mitarbeiter;
```

| MITARBEITER_ID | NACHNAME | GEHALT | GEHALT*1.025 |
|----------------|------------|--------|--------------|
| 1 | Winter | 88000 | 90200 |
| 2 | Werner | 50000 | 51250 |
| 3 | Seifert | 50000 | 51250 |
| 4 | Schwarz | 30000 | 30750 |
| 5 | Sindermann | 30000 | 30750 |
| 6 | Müller | 30000 | 30750 |

100 Zeilen ausgewählt





| MITARBEITER_ID | NACHNAME | GEHALT | GEHALT*1.025 |
|----------------|------------|--------|--------------|
| 1 | Winter | 88000 | 90200 |
| 2 | Werner | 50000 | 51250 |
| 3 | Seifert | 50000 | 51250 |
| 4 | Schwarz | 30000 | 30750 |
| 5 | Sindermann | 30000 | 30750 |
| 6 | Müller | 30000 | 30750 |

100 Zeilen ausgewählt



Oracle und SQL Server unterscheiden sich in der Anzeige von numerischen Werten. Oracle stellt Zahlen immer rechtsbündig dar. SQL Server zeigt dagegen alle Werte linksbündig an.

1.3.4. NULL Werte

Es kommt vor, dass nicht immer alle Attribute eines Datensatzes befüllt sind, d. h. einige Attribute haben keinen Attributwert. Dies kann aus zwei Gründen der Fall sein:

- Der Wert eines Attributes ist zum Zeitpunkt der Eingabe des Datensatzes nicht bekannt (z. B. der Vorname einer Person ist nicht bekannt).
- Der Wert eines Attributs steht zum Zeitpunkt der Eingabe des Datensatzes noch nicht fest (z. B. das Sterbedatum einer Person bei der Erstellung der Geburtsurkunde).



Ein NULL Wert steht immer für einen unbekannten Wert und ist nicht mit der natürlichen Zahl 0 zu verwechseln.

NULL Werte haben insbesondere bei der Verwendung arithmetischer Ausdrücke eine große Bedeutung. Um dies zu demonstrieren, soll für alle Mitarbeiter deren Monatsgehalt berechnet werden. Dieses setzt sich aus dem Grundgehalt (Spalte GEHALT) und einer Provision (Spalten PROVISION) zusammen. Da nicht jeder Mitarbeiter eine Provision erhält, existieren NULL Werte in der Tabelle MITARBEITER.

Listing 1.5: NULL in arithmetischen Ausdrücken

```
SELECT Vorname, Nachname, Gehalt + Gehalt / 100 * Provision
FROM Mitarbeiter;
```

| VORNAME | NACHNAME | GEHALT+GEHALT/100*PROVISION |
|-----------------------|----------|-----------------------------|
| Max | Winter | |
| Sarah | Werner | |
| ... | ... | ... |
| Johannes | Lehmann | 2400 |
| Louis | Schmitz | 2500 |
| Martin | Schacke | 1200 |
| 100 Zeilen ausgewählt | | |



Einige Mitarbeiter erhalten keine Provision. Oracle und SQL Server unterscheiden sich bei der Anzeige der NULL Werte. Oracle zeigt für NULL Werte nichts an. SQL Server zeigt die Zeichenkette NULL an.



Und hier das Ergebnis für den MS SQL Server.

| VORNAME | NACHNAME | (Kein Spaltenname) |
|-----------------------|----------|--------------------|
| Max | Winter | NULL |
| Sarah | Werner | NULL |
| Finn | Seifert | NULL |
| ... | ... | ... |
| Johannes | Lehmann | 2400 |
| Louis | Schmitz | 2500 |
| Marie | Kipp | NULL |
| Amelie | Krüger | NULL |
| Martin | Schacke | 1200 |
| ... | ... | ... |
| 100 Zeilen ausgewählt | | |



Jeder arithmetische Ausdruck, in dem ein NULL Wert als Operand vorkommt, hat als Ergebnis den Wert NULL.



1.3.5. Verkettung von Zeichenketten

In manchen Fällen ist es notwendig, die Ausgabe einzelner Spalten miteinander zu verbinden. Dieser Vorgang wird als Konkatination³ bezeichnet. Hierfür kennt Oracle den Operator `||` und SQL Server den Operator `+`.

Beispiel 1.4 wird nun dahingehend erweitert, dass die Gehaltserhöhung als Bericht angezeigt wird. Für jeden Mitarbeiter muss eine Zeile der Form „AAA hat ein Gehalt von XXX EUR und soll YYY EUR erhalten.“

Listing 1.6: Verkettung von Zeichenketten in Oracle

```
SELECT Nachname || ' hat ein Gehalt von ' || Gehalt ||
       ' EUR und soll ' || (Gehalt * 1.025) ||
       ' EUR erhalten.'
FROM   Mitarbeiter;
```



NACHNAME || ' HATEINGEHALT VON ' || GEHALT || ' EUR UND SOL ... '

Winter hat ein Gehalt von 88000 EUR und soll 90200 EUR erhalten.

...

Lehmann hat ein Gehalt von 2000 EUR und soll 2050 EUR erhalten.

Schmitz hat ein Gehalt von 2000 EUR und soll 2050 EUR erhalten.

Kipp hat ein Gehalt von 2000 EUR und soll 2050 EUR erhalten.

...

Listing 1.7: Verkettung von Zeichenketten in SQL Server

```
SELECT Nachname + ' hat ein Gehalt von ' + CAST(Gehalt AS VARCHAR(15)) +
       ' EUR und soll ' +
       CAST(Gehalt * 1.025 AS VARCHAR(15)) + ' erhalten.'
FROM   Mitarbeiter;
```



(Kein Spaltenname)

Winter hat ein Gehalt von 88000 EUR und soll 90200 EUR erhalten.

...

Lehmann hat ein Gehalt von 2000.00 EUR und soll ...

Schmitz hat ein Gehalt von 2000.00 EUR und soll ...

...

Schacke hat ein Gehalt von 1000.00 EUR und soll 1025 EUR erhalten.

...

100 Zeilen ausgewählt

100 Zeilen ausgewählt

³Verkettung von Zeichen oder Zeichenketten

An [Beispiel 1.6](#) und [Beispiel 1.7](#) sieht man sehr gut die unterschiedliche Reaktionsweise beider Datenbank Management Systeme. Oracle ist ohne weiteres in der Lage, Daten unterschiedlichen Typs (Zeichenketten, Zahl, Datumswerte, usw.) miteinander zu verknüpfen.

Bei Microsoft SQL Server ist dies nicht der Fall. Hier muss ein Vorgriff auf spätere Kapitel erfolgen. Der Zahlenwert der aus der Berechnung $gehalt * 1.025$ resultiert, muss explizit in eine Zeichenkette umgewandelt werden, bevor die Verkettung funktioniert.

1.3.6. Spaltenaliasnamen

Bei der Anzeige des Ergebnisses einer Auswahlabfrage wird für jede Spalte eine Überschrift erzeugt. Oracle und Microsoft SQL Server sind sich dabei in sofern einig, als dass für die Spaltenüberschriften die Spaltenbezeichner der Tabellenspalten genutzt werden. Wird aber in der **SELECT**-Liste eine Konstante, eine Funktion oder ein anderer Ausdruck genutzt, scheiden sich die Geister. SQL Server zeigt in so einem Falle einfach gar keine Überschrift an, während Oracle einen Teil des Ausdrucks als Überschrift nutzt. Dieses Verhalten ist in den vorangegangenen Beispielen an einigen Stellen zu sehen.

Der SQL-Standard erlaubt es dieses Verhalten zu beeinflussen und manuell eine Spaltenüberschrift, einen sogenannten „Spaltenaliasnamen“, zu vergeben. Dies funktioniert im Falle von Oracle und SQL Server auf die gleiche Art und Weise, da sich hier beide an den Standard halten.

Listing 1.8: Vergabe eines Spaltenaliasnamen

```
SELECT Vorname, Nachname, Gehalt + Gehalt / 100 * Provision AS "Neues Gehalt"
FROM Mitarbeiter;
```

| VORNAME | NACHNAME | Neues Gehalt |
|----------|----------|--------------|
| ... | ... | ... |
| Johannes | Lehmann | 2400 |
| Louis | Schmitz | 2500 |
| Marie | Kipp | |
| Amelie | Krüger | |
| Stefan | Beck | |
| Martin | Schacke | 1200 |
| ... | ... | ... |

100 Zeilen ausgewählt



Der ANSI SQL-Standard sieht vier verschiedene Möglichkeiten zur Angabe eines Spaltenaliasnamen vor:

- [Ausdruck] **AS** "Spaltenaliasname"
- [Ausdruck] "Spaltenaliasname"
- [Ausdruck] Spaltenaliasname
- [Ausdruck] **AS** Spaltenaliasname

Oracle und MS SQL Server unterstützen alle vier Varianten. Die Angabe von Anführungszeichen ist nur dann notwendig, wenn im Spaltenaliasnamen Sonderzeichen oder Leerzeichen vorkommen.



Oracle wandelt einen Spaltenaliasnamen automatisch in Großbuchstaben um, es sei den, er wird in Anführungszeichen eingeschlossen.

1.4. Einige Konventionen zu SQL

Um die Lesbarkeit von SQL-Statements zu verbessern werden an dieser Stelle einige Konventionen genannt, die im praktischen Umgang mit SQL eingehalten werden sollten.

- Da SQL-Anweisungen mehrzeilig sein können erhält jede Klausel (SELECT, FROM, usw.) eine eigene Zeile.
- SQL ist nicht casesensitiv, d. h. Groß- und Kleinschreibung ist nicht relevant. Zur Verbesserung der Lesbarkeit sollten Schlüsselwörter groß geschrieben werden. Beispiele hierfür sind im gesamten Skript zu finden.
- Verwenden Sie Einrückungen zur Verbesserung der Lesbarkeit.

Listing 1.9: So nicht!

```
SELECT Nachname + ' hat ein Gehalt von ' +  
CAST(Gehalt AS VARCHAR(15)) +  
' EUR und soll ' +  
CAST(Gehalt + Gehalt / 100 * Provision AS VARCHAR(15)) + ' erhalten.'  
FROM Mitarbeiter;
```

Listing 1.10: Viel besser lesbar!

```
SELECT Nachname + ' hat ein Gehalt von ' + CAST(Gehalt AS VARCHAR(15)) +  
      ' EUR und soll ' +  
      CAST(Gehalt + Gehalt / 100 * Provision AS VARCHAR(15)) + ' erhalten.'  
FROM Mitarbeiter;
```

- Gemäß SQL-Standard muss jedes SQL-Statement mit einem Semikolon (;) abgeschlossen werden. Anwendungen wie der SQL*Developer, der JDeveloper oder das Management Studio verbergen diesen Sachverhalt jedoch.

In SQL*Plus muss jedes SQL-Statement mit ; oder / abgeschlossen werden!



2. Selektieren und Sortieren

Inhaltsangabe

| | | |
|------------|--|-------------|
| 2.1 | Selektieren von Zeilen: Die WHERE-Klausel | 2-2 |
| 2.1.1 | Relationale Operatoren | 2-3 |
| 2.1.2 | Logische Verknüpfung von Ausdrücken | 2-8 |
| 2.2 | Festlegen einer Sortierung | 2-10 |
| 2.2.1 | Die ORDER BY Klausel | 2-10 |
| 2.2.2 | Auf- und absteigendes Sortieren | 2-12 |
| 2.3 | Übungen - Selektieren und Sortieren | 2-13 |
| 2.4 | Lösungen - Selektieren und sortieren | 2-17 |

2.1. Selektieren von Zeilen: Die WHERE-Klausel

Im vorangegangenen Kapitel wurde gezeigt, wie mit den beiden SQL-Klauseln **SELECT** und **FROM** der gesamte Inhalt einer Tabelle angezeigt werden kann. Zusätzlich zu diesen beiden Klauseln wird nun die optionale **WHERE**-Klausel eingeführt, die eine Selektion der Datensätze ermöglicht. Diese kann einen beliebig komplexen Ausdruck enthalten, der dann das „Auswahlkriterium“ für die Datensätze darstellt. Die Syntax der **WHERE**-Klausel lautet wie folgt:

Listing 2.1: Die WHERE-Klausel

```
WHERE <Ausdruck1> <Relationaler Operator> <Ausdruck2>
```



Der Begriff „Ausdruck“ steht in der Programmierung für ein auf einen Kontext bezogenes, auswertbares Gebilde. Bei *Ausdruck1* und *Ausdruck2* kann es sich beispielsweise um Spaltenbezeichner, Funktionsaufrufe, arithmetische Berechnungen, Konstanten usw. handeln.

Beispiel 2.1 zeigt insgesamt drei Ausdrücke:

- *<Ausdruck1>*
- *<Ausdruck2>*
- *<Ausdruck1> <Relationaler Operator> <Ausdruck2>*

Nicht nur *Ausdruck1* und *Ausdruck2* sind Ausdrücke, sondern auch die Verbindung beider, mittels eines Operators, wird als Ausdruck betrachtet.



Ein Operator ist ein mit einer Semantik belegtes Zeichen, dass eine genau definierte Operation darstellt. Operatoren werden meist in Gruppen eingeteilt, z. B. arithmetische Operatoren (+, -, *, /), relationale Operatoren, logische Operatoren, usw.

Tabelle 2.1 listet die in Oracle und MS SQL Server vorhandenen relationalen Operatoren auf.

2.1.1. Relationale Operatoren

Tabelle 2.1.: Relationale Operatoren in Oracle und MS SQL Server

| (Operator) | (Bedeutung) |
|-----------------|---|
| = | Gleichheit |
| != | Ungleichheit |
| < | Kleiner als |
| <= | Kleiner oder gleich |
| > | Größer als |
| >= | Größer oder gleich |
| LIKE | Ähnlichkeit zweier Zeichenketten |
| IN | Der linke Ausdruck befindet sich in einer Liste von Werten, die der rechte Ausdruck erzeugt. |
| IS NULL | Der linke Ausdruck liefert den Wert NULL zurück. |
| BETWEEN A AND B | Der Wert des linken Ausdrucks liegt zwischen den Wertgrenzen A und B. Die Wertgrenzen A und B werden in das Intervall mit einbezogen. |

Numerische Werte vergleichen

Der Vergleich von numerischen Werten gestaltet sich sowohl in Oracle als auch im MS SQL Server sehr einfach.

Listing 2.2: Gleichheit zweier numerischer Werte

```
SELECT Vorname, Nachname
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID = 5;
```

```
VORNAME  NACHNAME
-----
Tim       Sindermann
1 Zeile ausgewählt
```



Listing 2.3: Wert A größer oder gleich Wert B

```
SELECT Vorname, Nachname, Gehalt
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID >= 50;
```



| VORNAME | NACHNAME | GEHALT |
|-----------|----------|--------|
| Emilia | Köhler | 2500 |
| Karolin | Klingner | 2000 |
| Chris | Roggatz | 3000 |
| Christian | Haas | 2000 |
| Jessica | Winkler | 2000 |
| Anna | Keller | 2500 |
| Johannes | Klingner | 2500 |
| Emma | Krüger | 3500 |

51 Zeilen gewählt

Listing 2.4: Prüfen eines Intervalls

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID BETWEEN 5 AND 9;
```



| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|------------|
| 5 | Tim | Sindermann |
| 6 | Peter | Müller |
| 7 | Emily | Meier |
| 8 | Dirk | Peters |
| 9 | Louis | Winter |

5 Zeilen gewählt

Listing 2.5: Alle Zeilen aus einer Wertemenge anzeigen

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID IN (5, 7, 9);
```



| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|------------|
| 5 | Tim | Sindermann |
| 7 | Emily | Meier |
| 9 | Louis | Winter |

3 Zeilen gewählt

Zeichenketten vergleichen

Der Vergleich zweier Zeichenketten bringt, im Gegensatz zum Vergleich numerischer Werte, eine Schwierigkeit mit sich. Abhängig vom benutzten RDBMS¹ werden Zeichenkettenvergleiche casesensitiv oder incasesensitiv durchgeführt. In Oracle beispielsweise ist „Oracle“ ungleich „oracle“ oder „ORACLE“ ungleich „OrAcLe“. Der MS SQL Server hingegen verhält sich nicht casesensitiv. Für ihn sind alle vier Werte gleich.

Listing 2.6: Ein einfacher Zeichenkettenvergleich

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  Nachname = 'Scholz';
```

| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|----------|
| 96 | Johanna | Scholz |

1 Zeile ausgewählt



Im nächsten Beispiel wird eine ähnliche **WHERE**-Klausel verwendet, wie in [Beispiel 2.6](#), sie führt jedoch zu einem ganz anderen Ergebnis.

In SQL müssen Zeichenketten in Hochkommas ' eingeschlossen werden! Diese dürfen nicht mit den Akzent-Zeichen verwechselt werden!



Listing 2.7: Ein einfacher Zeichenkettenvergleich

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  Nachname = 'SCHOLZ';
```

Keine Zeilen ausgewählt!



Da die Oracle-Datenbank casesensitiv arbeitet, ist „SCHOLZ“ ungleich „Scholz“. Somit werden keine Datensätze gefunden. Der MS SQL Server hat hier keine Schwierigkeiten. Ihn stört die unterschiedliche Schreibweise der Zeichenketten nicht, weshalb er das gewünschte Ergebnis anzeigt.

¹RDBMS = Relationales Datenbank Management System



| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|----------|
| 96 | Johanna | Scholz |

1 Zeile ausgewählt

Zeichenketten vergleichen mit LIKE

Ist es notwendig nach einem Zeichenmuster zu suchen, wie z. B. *Alle Mitarbeiter, deren Nachname mit „Sch“ beginnt*, so kann dies mit dem **LIKE**-Operator geschehen.

Listing 2.8: Zeichenkettensuche mit einem Suchmuster

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM Mitarbeiter
WHERE Nachname LIKE 'Sch%';
```



| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|-----------|------------|
| 4 | Sebastian | Schwarz |
| 11 | Sophie | Schwarz |
| 25 | Elias | Schreiber |
| 29 | Louis | Schmitz |
| 33 | Martin | Schacke |
| 36 | Hans | Schumacher |

10 Zeilen ausgewählt

Der **LIKE**-Operator nutzt zwei Wildcards, um Suchmuster für Zeichenketten zu erstellen.

Tabelle 2.2.: Wildcards des LIKE-Operators

| (Wildcard) | (Bedeutung) |
|------------|--|
| % | (Prozentzeichen) Null, eines oder beliebig viele Zeichen |
| _ | (Unterstrich) Genau ein Zeichen |

Für **Beispiel 2.8** bedeutet dies: *Die ersten drei Zeichen des Suchmusters sind S, c und h. Nach dem h können null, eines oder beliebig viele andere Zeichen stehen.* Im nächsten Beispiel wird die **_**-Wildcard benutzt, um alle Mitarbeiter zu suchen, deren Nachname an der dritten Stelle ein kleines g trägt.

Listing 2.9: Zeichenkettensuche mit einem etwas komplexeren Suchmuster

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM Mitarbeiter
WHERE Nachname LIKE '__g%';
```

| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|----------|
| 37 | Louis | Wagner |
| 52 | Chris | Roggatz |
| 83 | Peter | Roggatz |
| 88 | Joachim | Wagner |

4 Zeilen ausgewählt



Die ersten beiden Zeichen des Suchmusters sind `_` Unterstriche, d. h. an der ersten und zweiten Stelle der gesuchten Zeichenketten **muss** sich jeweils genau ein Zeichen befinden. Das dritte Zeichen ist mit dem `g` genau definiert. Anschließend können wieder null, eines oder beliebig viele andere Zeichen stehen.

Der LIKE-Operator verwendet die beiden Wildcards `%` und `_`. `%` Steht für null, eines oder beliebig viele Zeichen. `_` steht für genau ein Zeichen.



Vergleiche mit NULL-Werten

Sowohl Oracle, als auch der MS SQL Server kennen beide den Operator `IS NULL`. Mit seiner Hilfe können Spalten auf NULL-Werte hin überprüft werden. Sollen z. B. alle Mitarbeiter, die keinen Vorgesetzten haben, angezeigt werden, wird ein Vergleich mit dem `IS NULL`-Operator angestellt.

Listing 2.10: Der IS NULL Operator

```
SELECT Mitarbeiter_ID, Vorname, Nachname, Vorgesetzter_ID
FROM   Mitarbeiter
WHERE  Vorgesetzter_ID IS NULL;
```

| MITARBEITER_ID | VORNAME | NACHNAME | VORGESETZTER_ID |
|----------------|---------|----------|-----------------|
| 1 | Max | Winter | |

1 Zeile ausgewählt



Da es in diesem Beispiel keinen wesentlichen Unterschied bei der Ergebnisanzeige zwischen Oracle und SQL Server gibt (SQL Server zeigt das Wort NULL für NULL-Werte und alle Spaltenwerte linksbündig an), wurde hier auf ein getrenntes Abdrucken der Ergebnisse verzichtet.

Das Gegenstück zum `IS NULL`-Operator, ist der `IS NOT NULL`-Operator.



Wird ein Ausdruck, mit Hilfe des Gleichheitsoperators (=), mit dem Wert NULL verglichen, ist das Ergebnis des Vergleichs immer NULL!

2.1.2. Logische Verknüpfung von Ausdrücken

In vielen Fällen ist es notwendig komplexe Ausdrücke zu formulieren, indem mehrere Ausdrücke miteinander verknüpft werden. Eine solche Verknüpfung geschieht unter Zuhilfenahme der logischen Operatoren *AND*, *OR* und *NOT*.

Logische Verknüpfungen mit AND

Der logische Operator *AND* verknüpft zwei Bedingungen miteinander und liefert ein wahres Ergebnis, sobald beide Ausdrücke ein wahres Ergebnis haben. Die Logiktablette [Tabelle 2.3](#) zeigt die möglichen Ergebnisse einer AND-Verknüpfung.

Tabelle 2.3.: Der logische Operator AND

| Aussagen | (Wahr) | (Falsch) |
|----------|--------|----------|
| Wahr | w | f |
| Falsch | f | f |

In [Beispiel 2.11](#) wird gezeigt, wie der *AND*-Operator dazu genutzt werden kann, um zwei Bedingungen miteinander zu verknüpfen. Es sollen alle Mitarbeiter angezeigt werden, deren Gehalt unter 1.500 EUR liegt und die in der Bankfiliale Nummer zwei arbeiten.

Listing 2.11: Der AND Operator

```
SELECT Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM Mitarbeiter
WHERE Gehalt < 1500
      AND Bankfiliale_ID = 2;
```



| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|---------|----------|--------|----------------|
| Martin | Schacke | 1000 | 2 |
| Oliver | Wolf | 1000 | 2 |

2 Zeilen ausgewählt

Logische Verknüpfungen mit OR

Der logische Operator *OR* liefert, im Unterschied zu *AND*, ein wahres Ergebnis, sobald mindestens einer der beiden Ausdrücke ein wahres Ergebnis hat.

Tabelle 2.4.: Der logische Operator OR

| Aussagen | (Wahr) | (Falsch) |
|----------|--------|----------|
| Wahr | w | w |
| Falsch | w | f |

Wird in [Beispiel 2.11](#) der Operator *AND* durch ein *OR* ersetzt, verändert sich die Ergebnismenge. Es werden jetzt alle Mitarbeiter angezeigt, die entweder ein Gehalt unter 1.500 EUR haben oder die in Bankfiliale Nummer zwei arbeiten.

Listing 2.12: Der OR Operator

```
SELECT Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM Mitarbeiter
WHERE Gehalt < 1500
      OR Bankfiliale_ID = 2;
```

| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|---------|------------|--------|----------------|
| Louis | Winter | 12000 | 2 |
| Stefan | Beck | 1500 | 2 |
| Martin | Schacke | 1000 | 2 |
| Max | Oswald | 1500 | 2 |
| Oliver | Wolf | 1000 | 2 |
| Hans | Schumacher | 1000 | 3 |
| Maja | Keller | 1000 | 5 |
| Elias | Sindermann | 1000 | 8 |
| Jonas | Meier | 1000 | 12 |

9 Zeilen ausgewählt



Aussagen mit NOT umkehren

Die Bedeutung des Operators *NOT* ist sehr einfach zu umschreiben. Er kehrt ein Ergebnis um. Aus einem wahren Ergebnis wird ein falsches und umgekehrt. Dieser Effekt ist auch mit *IS NULL* und *IS NOT NULL* zu sehen. In [Beispiel 2.13](#) werden alle Mitarbeiter angezeigt, deren Gehalt kleiner als 1.500 EUR ist und die nicht in der Bankfiliale Nummer zwei arbeiten.

Listing 2.13: Der NOT Operator

```

SELECT Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM Mitarbeiter
WHERE Gehalt < 1500
      AND NOT Bankfiliale_ID = 2;

```



| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|---------|------------|--------|----------------|
| Hans | Schumacher | 1000 | 3 |
| Maja | Keller | 1000 | 5 |
| Elias | Sindermann | 1000 | 8 |
| Jonas | Meier | 1000 | 12 |

4 Zeilen ausgewählt



Die Klammern (und) haben Einfluss auf die Bedeutung von Ausdrücken. Werden mehrere logische Operatoren kombiniert, kann so die Lesbarkeit von Ausdrücken verbessert oder deren Bedeutung verändert werden.

2.2. Festlegen einer Sortierung

In allen vorangegangenen Beispielen war die Reihenfolge der Ausgabe der Datensätze unbestimmt. Sowohl Oracle als auch Microsoft SQL Server geben die Datensätze immer in der Reihenfolge aus, in der sie in der Quelltable vorliegen. Soll eine sortierte Ausgabe erfolgen, muss dies mit Hilfe der in [Beispiel 1.3.1](#) gezeigten **ORDER BY**-Klausel geschehen. Dazu muss diese, mit Sortierbegriffen versehen, am Ende des SQL-Statements angegeben werden.

2.2.1. Die ORDER BY Klausel

Listing 2.14: Die ORDER BY Klausel

```

ORDER BY <Sortierbegriff 1> [asc|desc],
        <Sortierbegriff 2> [asc|desc],
        <Sortierbegriff n> [asc|desc] ...

```

Als Sortierbegriffe können Spaltenbezeichner, Spaltenaliasnamen, berechnete Ausdrücke und auch Spaltenpositionsangaben, bezogen auf die Reihenfolge der Spaltennamen in der SELECT-Liste, genutzt werden. [Beispiel 2.15](#) und [Beispiel 2.16](#) zeigen die Anwendung der **ORDER BY**-Klausel.



Werden mehrere Sortierbegriffe angegeben, wird die Sortierung von links nach rechts durchgeführt. Das bedeutet, dass zuerst nach dem äußerst linken Sortierbegriff sortiert wird und anschließend wird, innerhalb dieser Sortierung, jeder weitere Sortierbegriff angewandt. Die Sortierungen werden also ineinander geschachtelt.

Listing 2.15: Die ORDER BY Klausel mit Spaltenbezeichnern

```
SELECT  Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM    Mitarbeiter
WHERE   Gehalt <= 1500
ORDER BY Gehalt;
```

| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|---------|------------|--------|----------------|
| Oliver | Wolf | 1000 | 2 |
| Hans | Schumacher | 1000 | 3 |
| Maja | Wolf | 1000 | 5 |
| Elias | Sindermann | 1000 | 8 |
| Jonas | Meier | 1000 | 12 |
| Martin | Schacke | 1000 | 2 |
| Max | Oswald | 1500 | 2 |
| Stefan | Beck | 1500 | 2 |

18 Zeilen ausgewählt



Listing 2.16: Die ORDER BY Klausel mit Positionsangaben

```
SELECT  Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM    Mitarbeiter
WHERE   Gehalt <= 1500
ORDER BY 3, 2;
```

| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|---------|------------|--------|----------------|
| Maja | Keller | 1000 | 5 |
| Jonas | Meier | 1000 | 12 |
| Martin | Schacke | 1000 | 2 |
| Hans | Schumacher | 1000 | 3 |
| Elias | Sindermann | 1000 | 8 |
| Oliver | Wolf | 1000 | 2 |
| Stefan | Beck | 1500 | 2 |
| Georg | Dühning | 1500 | 20 |

18 Zeilen ausgewählt





Bei der Benutzung von Positionsangaben (siehe [Beispiel 2.16](#)) muss darauf geachtet werden, dass sich diese auf die Reihenfolge der Spaltenbezeichner in der SELECT-Liste beziehen. Wird die SELECT-Liste später verändert, müssen unter Umständen auch die Positionsangaben angepasst werden.

2.2.2. Auf- und absteigendes Sortieren

Zu jedem Suchbegriff können die beiden Kürzel **ASC** und **DESC** mit angegeben werden. **ASC**² bewirkt aufsteigende Sortierung (Standard) und **DESC**³ absteigende Sortierung. [Beispiel 2.17](#) zeigt, wie sich das Ergebnis durch die absteigende Sortierung der Spalte GEHALT verändert.

Listing 2.17: Die ORDER BY Klausel mit absteigender Sortierung

```
SELECT  Vorname, Nachname, Gehalt, Bankfiliale_ID
FROM    Mitarbeiter
WHERE   Gehalt <= 1500
ORDER BY Gehalt DESC, 2 ASC;
```



| VORNAME | NACHNAME | GEHALT | BANKFILIALE_ID |
|------------|-----------|--------|----------------|
| Stefen | Beck | 1500 | 2 |
| Georg | Dühning | 1500 | 20 |
| Tom | Fischer | 1500 | 17 |
| Jannis | Friedrich | 1500 | 14 |
| Maximilian | Hahn | 1500 | 13 |
| Lena | Hermann | 1500 | 4 |
| Anne | Huber | 1500 | 10 |

18 Zeilen ausgewählt



Eine in der **ORDER BY**-Klausel als Sortierbegriff genutzte Spalte, muss nicht in der SELECT-Liste der Abfrage vorhanden sein.

²engl. Ascending = aufsteigend

³engl. Descending = absteigend

2.3. Übungen - Selektieren und Sortieren

1. Erstellen Sie eine Abfrage, die die Konto_ID und das aktuelle Guthaben des Girokontos der Bankkunden anzeigt, die weniger als 1000 EUR Guthaben besitzen.

| KONTO_ID | GUTHABEN |
|----------|-----------|
| 15 | -10496,4 |
| 16 | -54593,2 |
| 43 | -42144,1 |
| 48 | -140505,1 |
| 57 | -1088,4 |
| 59 | 760,1 |
| 83 | 336,2 |
| 87 | -9009,1 |
| 99 | -69705,6 |

55 Zeilen ausgewählt



2. Erstellen Sie eine Abfrage, die die Mitarbeiter_ID und den Nachnamen der Mitarbeiter mit der Vorgesetzter_ID „2“ anzeigt.

| MITARBEITER_ID | NACHNAME |
|----------------|------------|
| 4 | Schwarz |
| 5 | Sindermann |

2 Zeilen ausgewählt



3. Erstellen Sie eine Abfrage, die die Konto_ID und das aktuelle Guthaben des Girokontos der Bankkunden anzeigt, deren Guthaben nicht zwischen 1000 EUR und 1500 EUR liegt.

| KONTO_ID | GUTHABEN |
|----------|----------|
| 1 | 111316,9 |
| 2 | 96340,2 |
| 3 | 59633 |
| 5 | 98449 |
| 6 | 26130,7 |
| 7 | 23128,7 |
| 9 | 8857,6 |
| 10 | 68001,3 |

428 Zeilen ausgewählt



4. Lassen Sie sich die Kunden_ID und das Geburtsdatum aller Eigenkunden anzeigen, die zwischen dem 20. Februar 1980 und dem 02. März 1988 geboren sind. Zusätzlich soll die Abfrage nach dem Geburtsdatum in aufsteigender Reihenfolge sortiert werden.



| KUNDEN_ID | GEBURTSDATUM |
|-----------|--------------|
| 391 | 01.03.80 |
| 387 | 03.03.80 |
| 339 | 22.03.80 |
| 75 | 22.03.80 |
| 458 | 07.05.80 |
| 50 | 21.05.80 |

124 Zeilen ausgewählt

5. Zeigen Sie, in alphabetischer Reihenfolge, die Mitarbeiter_ID und den Nachnamen der Mitarbeiter an, die die Vorgesetzter_ID „5“ oder „6“ haben.



| MITARBEITER_ID | NACHNAME |
|----------------|----------|
| 17 | Becker |
| 16 | Berger |
| 20 | Große |
| 13 | Kaiser |
| 18 | Köhler |
| 14 | Lorenz |
| 22 | Rollert |

10 Zeilen ausgewählt

6. Erstellen Sie eine Abfrage, die den Nachnamen und die Bankfiliale_ID der Mitarbeiter ausgibt, die die Vorgesetzten_ID „5“ oder „6“ haben und deren Bankfiliale_ID zwischen „10“ und „20“ ist. Die Spalten sollen mit „Mitarbeiter“ und „Bankfiliale“ benannt werden.



| MITARBEITER | BANKFILIALE |
|-------------|-------------|
| Becker | 10 |
| Köhler | 11 |
| Weber | 12 |
| Große | 13 |
| Walther | 14 |

6 Zeilen ausgewählt

7. Zeigen Sie die Mitarbeiter_ID und den Nachnamen des Mitarbeiters an, der keinen Vorgesetzten hat.

| MITARBEITER_ID | NACHNAME |
|----------------|----------|
| 1 | Winter |

1 Zeile ausgewählt



8. Zeigen Sie die Kunden_ID und das Geburtsdatum derjenigen Eigenkunden an, die im Jahre 1980 geboren sind.

| KUNDEN_ID | GEBURTSDATUM |
|-----------|--------------|
| 387 | 03.03.80 |
| 538 | 22.01.80 |
| 161 | 17.08.80 |
| 254 | 12.09.80 |

14 Zeilen ausgewählt



9. Erstellen Sie eine Abfrage, die den Nachnamen, das Gehalt und die Provision für alle Mitarbeiter anzeigt, die eine Provision erhalten. Sortieren Sie die Ausgabe in absteigender Reihenfolge nach dem Gehalt.

| NACHNAME | GEHALT | PROVISION |
|------------|--------|-----------|
| Hartmann | 4000 | 30 |
| Roth | 3500 | 20 |
| Walther | 3500 | 20 |
| Wagner | 3500 | 20 |
| Zimmermann | 3500 | 30 |

33 Zeilen ausgewählt



10. Zeigen Sie die Nachnamen aller Mitarbeiter an, in deren Nachname an dritter Stelle ein „a“ vorkommt.

| NACHNAME |
|----------|
| Haas |
| Haas |
| Krause |
| Krause |

4 Zeilen ausgewählt



11. Zeigen Sie die Nachnamen aller Mitarbeiter an, deren Nachname ein kleines „a“ und ein kleines „e“ enthält.



NACHNAME

Sindermann

Kaiser

Zimmermann

Walther

Neumann

Lehmann

24 Zeilen ausgewählt

2.4. Lösungen - Selektieren und sortieren

1. Erstellen Sie eine Abfrage, die die Konto_ID und das aktuelle Guthaben des Girokontos der Bankkunden anzeigt, die weniger als 1000 EUR Guthaben besitzen.

```
SELECT Konto_ID, Guthaben
FROM Girokonto
WHERE Guthaben < 1000;
```



2. Erstellen Sie eine Abfrage, die die Mitarbeiter_ID und den Nachnamen der Mitarbeiter mit der Vorgesetzter_ID „2“ anzeigt.

```
SELECT Mitarbeiter_ID, Nachname
FROM Mitarbeiter
WHERE Vorgesetzter_ID = 2;
```



3. Erstellen Sie eine Abfrage, die die Konto_ID und das aktuelle Guthaben des Girokontos der Bankkunden anzeigt, deren Guthaben nicht zwischen 1000 EUR und 1500 EUR liegt.

```
SELECT Konto_ID, Guthaben
FROM Girokonto
WHERE Guthaben NOT BETWEEN 1000 AND 1500;
```



4. Lassen Sie sich die Kunden_ID und das Geburtsdatum aller Eigenkunden anzeigen, die zwischen dem 20. Februar 1980 und dem 02. März 1988 geboren sind. Zusätzlich soll die Abfrage nach dem Geburtsdatum in aufsteigender Reihenfolge sortiert werden.

```
SELECT Kunden_ID, Geburtsdatum
FROM Eigenkunde
WHERE Geburtsdatum BETWEEN '20.02.1980' AND '02.03.1988'
ORDER BY Geburtsdatum;
```



5. Zeigen Sie, in alphabetischer Reihenfolge, die Mitarbeiter_ID und den Nachnamen der Mitarbeiter an, die die Vorgesetzter_ID „5“ oder „6“ haben.



```
SELECT  Mitarbeiter_ID, Nachname
FROM    Mitarbeiter
WHERE   Vorgesetzter_ID IN (5, 6)
ORDER BY Nachname;
```

6. Erstellen Sie eine Abfrage, die den Nachnamen und die Bankfiliale_ID der Mitarbeiter ausgibt, die die Vorgesetzten_ID „5“ oder „6“ haben und deren Bankfiliale_ID zwischen „10“ und „20“ ist. Die Spalten sollen mit „Mitarbeiter“ und „Bankfiliale“ benannt werden.



```
SELECT Nachname AS "Mitarbeiter", Bankfiliale_ID AS "Bankfiliale"
FROM    Mitarbeiter
WHERE   Vorgesetzter_ID IN (5, 6)
        AND Bankfiliale_ID BETWEEN 10 AND 20;
```

7. Zeigen Sie die Mitarbeiter_ID und den Nachnamen des Mitarbeiters an, der keinen Vorgesetzten hat.



```
SELECT  Mitarbeiter_ID, Nachname
FROM    Mitarbeiter
WHERE   Vorgesetzter_ID IS NULL;
```

8. Zeigen Sie die Kunden_ID und das Geburtsdatum derjenigen Eigenkunden an, die im Jahre 1980 geboren sind.



```
SELECT  Kunden_ID, Geburtsdatum
FROM    Eigenkunde
WHERE   Geburtsdatum BETWEEN '01.01.1980' AND '31.12.1980';
```

9. Erstellen Sie eine Abfrage, die den Nachnamen, das Gehalt und die Provision für alle Mitarbeiter anzeigt, die eine Provision erhalten. Sortieren Sie die Ausgabe in absteigender Reihenfolge nach dem Gehalt.

```
SELECT  Nachname, Gehalt, Provision
FROM    Mitarbeiter
WHERE   Provision IS NOT NULL
ORDER BY Gehalt DESC;
```



10. Zeigen Sie die Nachnamen aller Mitarbeiter an, in deren Nachname an dritter Stelle ein „a“ vorkommt.

```
SELECT Nachname
FROM    Mitarbeiter
WHERE   Nachname LIKE '__a%';
```



11. Zeigen Sie die Nachnamen aller Mitarbeiter an, deren Nachname ein kleines „a“ und ein kleines „e“ enthält.

```
SELECT Nachname
FROM    Mitarbeiter
WHERE   Nachname LIKE '%a%'
       AND Nachname LIKE '%e%';
```



3. Funktionen

Inhaltsangabe

| | | |
|------------|--|-------------|
| 3.1 | Der Begriff der Funktion | 3-2 |
| 3.2 | Zeichenkettenfunktionen | 3-3 |
| 3.2.1 | Groß- oder Kleinschreibung - UPPER, LOWER, INITCAP | 3-3 |
| 3.2.2 | Zeichenketten bearbeiten | 3-5 |
| 3.3 | Arithmetische Funktionen | 3-11 |
| 3.3.1 | FROM oder nicht FROM, das ist hier die Frage | 3-11 |
| 3.3.2 | Arithmetische Funktionen anwenden | 3-12 |
| 3.4 | Datumsfunktionen | 3-15 |
| 3.4.1 | Datumswerte | 3-15 |
| 3.4.2 | Datumsarithmetik in Oracle | 3-16 |
| 3.4.3 | Datumsarithmetik in MS SQL Server | 3-19 |
| 3.5 | Sonstige Funktionen | 3-20 |
| 3.6 | Datentypen..... | 3-22 |
| 3.6.1 | Numerische Datentypen | 3-23 |
| 3.6.2 | Zeichendatentypen | 3-24 |
| 3.6.3 | Datums- und Zeittypen | 3-24 |
| 3.7 | Konvertierung von Datentypen | 3-25 |
| 3.7.1 | Implizite Datentypkonvertierung | 3-25 |
| 3.7.2 | Explizite Datentypkonvertierung | 3-28 |
| 3.8 | Übungen - Funktionen | 3-33 |
| 3.9 | Lösungen - Funktionen..... | 3-36 |

3.1. Der Begriff der Funktion



Eine Funktion ist eine Rechenvorschrift (ein kleines Programm), welches zu einer Menge von Eingabewerten eine Ergebnismenge (Ausgabewert) erzeugt. Die Eingabewerte werden als sogenannte Parameter/Argumente an die Funktion übergeben.

Bildlich kann man sich eine Funktion vorstellen, wie eine Maschine, an deren einem Ende etwas zugeführt wird und am Anderen entsteht ein Produkt (Ergebnis). SQL kennt zwei unterschiedliche Arten von Funktionen:

- **Single Row Functions / Skalare Funktionen:** Sie werden immer nur auf einen einzelnen Attributwert angewandt und müssen somit für jede Ergebniszeile einmal ausgeführt werden.
- **Grouping Functions / Aggregatfunktionen:** Diese Art von Funktionen wird pro Abfrage nur einmal ausgeführt und bezieht sich immer auf eine Gruppe von Werten (siehe [Abschnitt 5](#)).

Die Gruppe der Single Row Functions (in SQL Server werden diese Funktionen als „Skalare Funktionen“ bezeichnet) wird wiederum in mehrere Arten unterteilt.

- **Zeichenkettenfunktionen:** Dieser Funktionstyp findet Anwendung auf Zeichenketten. Mit ihm kann in Zeichenketten gesucht, Teile aus Zeichenketten herausgeschnitten und noch vieles mehr gemacht werden.
- **Arithmetische Funktionen:** Die Klasse der arithmetischen Funktionen führt Rechenoperationen auf den Operanden durch (z. B. Runden, Radizieren, Logarithmieren, Potenzieren, Modulo, usw.).
- **Datumsfunktionen:** Datumsfunktionen stehen in Zusammenhang mit Datumswerten. Sie können z. B. den Wochentag zu einem Datum oder einfach nur das aktuelle Systemdatum anzeigen.
- **Sonstige Funktionen:** Alles was sich nicht in die oben stehenden drei Kategorien einteilen lässt, zählt zu den sonstigen Funktionen.



Mit Ausnahme der **FROM**-Klausel, können Single Row Functions in allen anderen Klauseln genutzt werden!

3.2. Zeichenkettenfunktionen

Die Kategorie der Zeichenkettenfunktionen stellt nützliche Werkzeuge zur Auswertung und Modifikation von Zeichenketten¹ zur Verfügung. Mit ihrer Hilfe kann man:

- die Schreibweise eines Strings (Groß- / Kleinschreibung) verändern,
- die Länge einer Zeichenkette ermitteln,
- Leerzeichen abschneiden,
- Teilzeichenketten (Substrings) ausschneiden

und noch vieles mehr. An dieser Stelle sollen einige Beispiele für Zeichenkettenfunktionen in Oracle und SQL Server gezeigt werden.

Das wesentliche Ziel dieses Abschnittes ist es, dem Teilnehmer die Anwendung von Funktionen im Allgemeinen näher zu bringen. Spezielle Kenntnisse über einzelne Funktionen stehen dabei im Hintergrund.



3.2.1. Groß- oder Kleinschreibung - UPPER, LOWER, INITCAP

In [Abschnitt 2.1.1](#) wurde bereits auf die Problematik der Casesensitivität hingewiesen. Oracle ist standardmäßig casesensitiv, SQL Server nicht. Soll in Oracle nach einer bestimmten Zeichenkette, z. B. einem Nachnamen gesucht werden und die korrekte Schreibweise ist nicht bekannt, kann es vorkommen, dass das gewünschte Ergebnis nicht erreicht wird. Hierfür gibt es eine Lösung: Die Funktionen **UPPER**, **LOWER** und **INITCAP**.

Tabelle 3.1.: Zeichenkettenfunktionen



| Funktionsbezeichnung | | Bedeutung |
|----------------------|-------|--|
| UPPER | UPPER | Wandelt die gesamte Zeichenkette in Großbuchstaben um. |
| LOWER | LOWER | Wandelt die gesamte Zeichenkette in Kleinbuchstaben um. |
| INITCAP | n. a. | Wandelt das erste Zeichen jedes Wortes in einen Großbuchstaben um. |

¹Zeichenkette = engl. String



Die Funktion **INITCAP** existiert in MS SQL Server nicht!

Beispiel 3.1 zeigt die Anwendung der drei Funktionen **UPPER**, **LOWER** und **INITCAP** in Oracle.

Listing 3.1: UPPER, LOWER und INITCAP

```
SELECT UPPER(Vorname) AS GROSS, LOWER(Nachname) AS Klein,
       INITCAP(Vorname || ' ' || Nachname) AS NORMAL
FROM   Mitarbeiter;
```



| GROSS | KLEIN | NORMAL |
|-----------|---------|-------------------|
| MAX | winter | Max Winter |
| SARAH | werner | Sarah Werner |
| FINN | seifert | Finn Seifert |
| SEBASTIAN | schwarz | Sebastian Schwarz |

100 Zeilen ausgewählt



Die Anwendung der Funktionen **UPPER** und **LOWER** ist in Oracle und MS SQL Server identisch!

Eingangs wurde erwähnt, dass Single Row Functions nicht nur in der **SELECT**-Klausel genutzt werden können, sondern, z. B. auch in der **WHERE**-Klausel. Dadurch kann das beschriebene Problem der Casesensitivität gelöst werden.

Listing 3.2: Das Problem der Casesensitivität

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  Nachname LIKE 'winter';
```



| MITARBEITER_ID | VORNAME | NACHNAME |
|-------------------------|---------|----------|
| Keine Zeilen ausgewählt | | |

Der Mitarbeiter „Winter“ wird von Oracle nicht gefunden, da er in der Datenbank mit einem großen W am Namensanfang gespeichert ist. Für Oracle sind „winter“ und „Winter“ zwei unterschiedliche Zeichenketten. Hier kann die **LOWER**-Funktion Abhilfe schaffen.

Listing 3.3: LOWER - Die Lösung des Problems

```
SELECT Mitarbeiter_ID, Vorname, Nachname
FROM   Mitarbeiter
WHERE  LOWER(Nachname) LIKE 'winter';
```



| MITARBEITER_ID | VORNAME | NACHNAME |
|----------------|---------|----------|
| 1 | Max | Winter |
| 9 | Louis | Winter |

2 Zeilen ausgewählt

Die **LOWER**-Funktion stellt in [Beispiel 3.3](#) sicher, dass alle Werte der Spalte NACHNAME in Kleinbuchstaben ausgegeben werden. Somit ist der Vergleich mit „winter“ unproblematisch.

3.2.2. Zeichenketten bearbeiten

Eine weitere Anwendung von Zeichenkettenfunktionen besteht darin, Teile aus Zeichenketten herauszutrennen oder deren Länge festzustellen. Hierzu kennen Oracle und SQL Server unterschiedliche Funktionen, die in [Tabelle 3.2](#) beschrieben werden. Sie stellt jedoch nur einen Ausschnitt aus der Menge der Zeichenkettenfunktionen dar.

Tabelle 3.2.: Zeichenkettenfunktionen



| Funktionsbezeichnung | | Bedeutung |
|----------------------|-----------|--|
| SUBSTR | SUBSTRING | Schneidet einen Teil einer Zeichenkette aus und liefert ihn zurück. |
| LENGTH | LEN | Gibt die Länge einer Zeichenkette zurück. |
| INSTR | CHARINDEX | Diese Funktionen suchen nach Zeichenkette A in einer Zeichenkette B und liefern die Position von A zurück. Ist A nicht in B erhält man 0 als Ergebnis. |

SUBSTR, LENGTH und INSTR in Oracle

Die Funktion **LENGTH** ermittelt, aus wie vielen Zeichen ein String besteht. Sie wird meist in Zusammenhang mit den beiden anderen Funktionen **SUBSTR** und **INSTR** genutzt. [Beispiel 3.4](#) zeigt die Anwendung von **LENGTH** auf sehr einfache Art und Weise.

Listing 3.4: Die **LENGTH**-Funktion

```
SELECT LENGTH (IBAN), IBAN
FROM Konto
WHERE Konto_ID = 1281;
```



| LENGTH (IBAN) | IBAN |
|---------------|------------------------|
| 22 | DE23465387306148533897 |

1 Zeile ausgewählt

Alleine für sich, ist die Information „22“ auf den ersten Blick nutzlos, wenn man aber bedenkt, dass z. B. eine IBAN eine feste Länge von 22 Zeichen hat, kann man mit Hilfe von **LENGTH** verifizieren, ob es sich um eine IBAN mit gültiger Länge handelt.

SUBSTR ist dabei behilflich, einen Teil aus einer Zeichenkette auszuschneiden. Eine solche Vorgehensweise ist z. B. dann notwendig, wenn eine Information, in der Form „Winter, Max“, in einer Tabellenspalte abgelegt ist oder, wenn wie im Falle der IBAN, mehrere Informationen einfach verkettet wurden (Länderkennung, Prüfziffer, BLZ und KtoNr).



Das Ergebnis einer solchen Operation wird als „Teilzeichenkette“ oder „Substring“ bezeichnet, wobei „Substring“ die geläufigere Variante darstellt.

Beispiel 3.5 zeigt die Anwendung der Funktion **SUBSTR**, um die IBAN eines Kontos in ihre Bestandteile zu zerlegen.

Listing 3.5: Die Anwendung der Funktion **SUBSTR**

```
SELECT SUBSTR (IBAN, 1, 2) AS Laenderkuerzel, SUBSTR (IBAN, 5, 8) AS BLZ,
       SUBSTR (IBAN, 14) AS KtoNr
FROM   Konto
WHERE  Konto_ID = 1281;
```



| LAENDE | BLZ | KTONR |
|--------|----------|------------|
| DE | 46538730 | 6148533897 |

1 Zeile ausgewählt

Tabelle 3.3.: Funktionsargumente von SUBSTR

| Argument | Beispiel | Erläuterung |
|----------|----------|---|
| 1 | IBAN | Beliebige Zeichenkette (Literal, Spaltenbezeichner oder eine andere Funktion). |
| 2 | 11 | An welcher Stelle im ersten Argument soll das Ausschneiden begonnen werden?. Hier kann eine beliebige Integerzahl stehen, die kleiner ist, als die Gesamtlänge von Argument 1. |
| 3 | 16 | Mit dem dritten und letzten Argument wird angegeben, wie viele Zeichen ausgeschnitten werden sollen. <ul style="list-style-type: none"> • $n > 1$: Es werden n Zeichen ausgeschnitten. • n nicht angegeben: Es werden alle Zeichen, bis zum Ende der Zeichenkette angezeigt. • $n < 1$: NULL-Wert als Ergebnis |

Mit **INSTR** kann die Position eines Zeichens oder einer Zeichenkette in einer Zeichenkette bestimmt werden. Eine typische Aufgabenstellung könnte sein: „Bestimme ob das Länderkürzel DE in der IBAN vorkommt“. In SQL ausgedrückt sieht dies so aus:

Listing 3.6: Automatische Positionsbestimmung

```
SELECT INSTR (IBAN, 'DE') AS Position
FROM Konto
WHERE Konto_ID = 1281;
```

POSITION

1

1 Zeile ausgewählt



Beispiel 3.6 zeigt, wie mit Hilfe der **INSTR**-Funktion die Position eines einzelnen Zeichens bzw. mehrere Zeichen in einer Zeichenkette ermittelt werden kann.

Tabelle 3.4.: Funktionsargumente von INSTR

| Argument | Beispiel | Erläuterung |
|----------|----------|---|
| 1 | IBAN | Beliebige Zeichenkette (Literal, Spaltenbezeichner oder eine andere Funktion). |
| 2 | 'DE' | Das zweite Argument gibt an, nach welchem Zeichen / welcher Zeichenkette gesucht werden soll. |



Die Funktion **INSTR** kennt noch zwei weitere Parameter, welche hier nicht näher erläutert werden. Weitere Informationen zu dieser Funktion können aus der Online-Dokumentation entnommen werden.

Die folgenden Links verweisen auf die Online-Dokumentation der Oracle-Datenbank.



- [\[i77725\]](#)
- [\[i87066\]](#)
- [\[i77598\]](#)

SUBSTRING, LEN und CHARINDEX in MS SQL Server

Die Funktion **LEN** ermittelt, aus wie vielen Zeichen ein String besteht. Sie wird meist in Zusammenhang mit den beiden anderen Funktionen **SUBSTRING** und **CHARINDEX** genutzt. [Beispiel 3.7](#) zeigt die Anwendung von **LEN** auf sehr einfache Art und Weise.

Listing 3.7: Die **LEN**-Funktion

```
SELECT LEN(IBAN), IBAN
FROM Konto
WHERE Konto_ID = 1281;
```



| (Kein Spaltenname) | Nachname |
|--------------------|------------------------|
| 22 | DE23465387306148533897 |

1 Zeile ausgewählt

Alleine für sich, ist die Information „22“ auf den ersten Blick nutzlos, wenn man aber bedenkt, dass z. B. eine IBAN eine feste Länge von 22 Zeichen hat, kann man mit Hilfe von **LEN** verifizieren, ob es sich um eine IBAN mit gültiger Länge handelt.

SUBSTRING ist dabei behilflich, einen Teil aus einer Zeichenkette auszuschneiden. Eine solche Vorgehensweise ist z. B. dann notwendig, wenn eine Information, in der Form „Winter, Max“, in einer Tabellenspalte abgelegt ist oder, wenn wie im Falle der IBAN, mehrere Informationen einfach verkettet wurden (Länderkennung, Prüfziffer, BLZ und KtoNr).

Das Ergebnis einer solchen Operation wird als „Teilzeichenkette“ oder „Substring“ bezeichnet, wobei „Substring“ die geläufigere Variante darstellt.



Beispiel 3.8 zeigt die Anwendung der Funktion **SUBSTRING**, um die IBAN eines Kontos in ihre Bestandteile zu zerlegen.

Listing 3.8: Die Anwendung der Funktion **SUBSTRING**

```
SELECT SUBSTRING(IBAN, 1, 2) AS Laenderkuerzel, SUBSTRING(IBAN, 5, 8) AS BLZ,
       SUBSTRING(IBAN, 14) AS KtoNr
FROM   Konto
WHERE  Konto_ID = 1281;
```

| LAENDE | BLZ | KTONR |
|--------|----------|------------|
| DE | 46538730 | 6148533897 |

1 Zeile ausgewählt



Tabelle 3.5.: Funktionsargumente von SUBSTR

| Argument | Beispiel | Erläuterung |
|----------|----------|---|
| 1 | IBAN | Beliebige Zeichenkette (Literal, Spaltenbezeichner oder eine andere Funktion). |
| 2 | 11 | An welcher Stelle im ersten Argument soll das Ausschneiden begonnen werden?. Hier kann eine beliebige Integerzahl stehen, die kleiner ist, als die Gesamtlänge von Argument 1. |
| 3 | 16 | Mit dem dritten und letzten Argument wird angegeben, wie viele Zeichen ausgeschnitten werden sollen. <ul style="list-style-type: none"> • $n > 1$: Es werden n Zeichen ausgeschnitten. • n nicht angegeben: Es werden alle Zeichen, bis zum Ende der Zeichenkette angezeigt. • $n < 1$: NULL-Wert als Ergebnis |

Mit **CHARINDEX** kann die Position eines Zeichens oder einer Zeichenkette in einer Zeichenkette bestimmt werden. Eine typische Aufgabenstellung könnte sein: „Bestimme ob das Länderkürzel DE in der IBAN vorkommt“. In SQL ausgedrückt sieht dies so aus:

Listing 3.9: Automatische Positionsbestimmung

```
SELECT CHARINDEX('DE', IBAN) AS Position
FROM Konto
WHERE Konto_ID = 1281;
```



POSITION

1

1 Zeile ausgewählt

Beispiel 3.9 zeigt, wie mit Hilfe der **CHARINDEX**-Funktion die Position eines einzelnen Zeichens in einer Zeichenkette ermittelt werden kann.

Tabelle 3.6.: Funktionsargumente von CHARINDEX

| Argument | Beispiel | Erläuterung |
|----------|---------------------------|---|
| 1 | 'DE' | Das erste Argument gibt an, nach welchem Zeichen / welcher Zeichenkette gesucht werden soll. |
| 2 | Nachname + ', ' + Vorname | Das zweite Argument ist eine beliebige Zeichenkette. An dieser Stelle kann ein Literal, ein Spaltenbezeichner oder eine andere Funktion stehen. |



Die Funktion **CHARINDEX** kennt noch einen weiteren Parameter, welcher hier nicht näher erläutert wird. Weitere Informationen zu dieser Funktion können aus der Online-Dokumentation, der MSDN, entnommen werden.

Die folgenden Links verweisen auf die Online-Dokumentation des MS SQL Server, im Microsoft Developer Network (MSDN).



- [\[ms190329\]](#)
- [\[ms187748\]](#)
- [\[ms186323\]](#)

3.3. Arithmetische Funktionen

Arithmetische Funktionen dienen dazu Berechnungen anzustellen. Dies kann beispielsweise sein:

- Runden,
- Radizieren (Wurzelziehen),
- Potenzieren,
- Logarithmieren

und vieles mehr. In dieser Unterrichtsunterlage werden nur einige Beispiele gezeigt.

3.3.1. FROM oder nicht FROM, das ist hier die Frage

In [Beispiel 3.10](#) und [Beispiel 3.11](#) wird die Berechnung der Quadratwurzel, der Zahl 4, in Oracle und MS SQL Server gezeigt. Die Anwendung der Funktion **SQRT** ist in beiden Systemen gleich. Trotzdem existiert ein gravierender Unterschied zwischen beiden Beispielen.

Listing 3.10: Berechnung der Quadratwurzel, der Zahl 4, in Oracle

```
SELECT SQRT(4) AS "Wurzel 4"
FROM dual;
```

Listing 3.11: Berechnung der Quadratwurzel, der Zahl 4, in MS SQL Server

```
SELECT SQRT(4) As "Wurzel 4";
```

| Wurzel 4 |
|----------|
| 2 |

1 Zeile ausgewählt



Eingangs wurde behauptet, dass jedes **SELECT**-Statement immer eine **FROM**-Klausel benötigen würde. Dies ist gemäß SQL-Standard auch richtig. Das DBMS Oracle setzt an dieser Stelle den Standard konsequent um, der MS SQL Server nicht.

Im DBMS Oracle muss jedes **SELECT**-Statement immer eine **FROM**-Klausel aufweisen, in MS SQL Server nicht.



Um den SQL-Standard einhalten zu können, gibt es in Oracle eine „Dummy-Tabelle“ namens DUAL. Diese enthält nur eine Spalte, mit einer Zeile, wie in [Beispiel 3.12](#) zu sehen ist. Sie kommt immer dann zum Einsatz, wenn das Ergebnis einer Funktion, unabhängig von irgendwelchen Datensätzen, abgerufen werden muss.

Listing 3.12: Die Tabelle DUAL in Oracle

```
SELECT *
FROM dual;
```



DUMMY

X

1 Zeile ausgewählt

Der MS SQL Server kommt ohne diese Tabelle aus, da bei ihm die **FROM**-Klausel einfach weggelassen werden darf.

3.3.2. Arithmetische Funktionen anwenden

Oracle kennt ca. 30 und MS SQL Server ca. 20 arithmetische Funktionen. Ziel dieser Unterrichtsunterlage ist es, einige wenige davon herauszugreifen und deren Anwendung zu erläutern. Hierbei handelt es sich um die folgenden Funktionen:

Tabelle 3.7.: Arithmetische Funktionen



Funktionsbezeichnung

Bedeutung

| | | |
|-------|---------|--|
| CEIL | CEILING | Gibt immer die kleinste ganze Zahl aus, die größer oder gleich n ist (Ganzzahliges Aufrunden). |
| FLOOR | FLOOR | Gibt immer die größte ganze Zahl aus, die kleiner oder gleich n ist (Ganzzahliges Abrunden). |
| LOG | n. a. | Berechnet den Logarithmus der Zahl x zur Basis n. |
| MOD | % | Gibt den Rest einer ganzzahligen Division zurück. |
| POWER | POWER | Potenziert die Zahl x mit n. |
| ROUND | ROUND | Auf- bzw. Abrunden einer Zahl nach dem kaufmännischen Rundungsverfahren ($x < 0,5$ = Abrunden, $x \geq 0,5$ = Aufrunden). |
| TRUNC | n. a. | Schneidet die Nachkommastellen einer Zahl ab und gibt den ganzzahligen Anteil zurück. |

In den beiden folgenden Beispielen wird die Anwendung von Rundungsfunktionen in Oracle und MS SQL Server gezeigt.

Listing 3.13: Rundungsfunktionen in Oracle

```
SELECT SQRT(3) AS "Wurzel 3", CEIL(SQRT(3)) AS "Aufrunden",
       FLOOR(SQRT(3)) AS "Abrunden",
       ROUND(SQRT(3), 2) AS "Kaufm. runden"
FROM   dual;
```

| Wurzel 3 | Aufrunden | Abrunden | Kaufm. runden |
|-----------|-----------|----------|---------------|
| 1,7320508 | 2 | 1 | 1,73 |

1 Zeile ausgewählt



Die Funktionen aus [Beispiel 3.13](#) sind weitestgehend selbsterklärend. Die mit `SQRT(3)` erzeugte Zahl 1,7320508 wird durch `CEIL` aufgerundet auf 2, von `FLOOR` abgerundet auf 1 und mit Hilfe von `ROUND` kaufmännisch, auf zwei Nachkommastellen, aufgerundet.

Die Funktion `ROUND` rundet kaufmännisch. Das zweite Argument gibt an, auf welche Nachkommastelle gerundet werden soll. Durch die Angabe von 0 wird auf die nächste ganze Zahl gerundet.



Die Anwendung dieser Funktionen ist im MS SQL Server identisch, mit der Ausnahme, dass `CEIL` dort `CEILING` heißt.

Listing 3.14: Rundungsfunktionen in MS SQL

```
SELECT SQRT(3) AS "Wurzel 3", CEILING(SQRT(3)) AS "Aufrunden",
       FLOOR(SQRT(3)) AS "Abrunden",
       ROUND(SQRT(3), 2) AS "Kaufm. runden"
FROM   dual;
```

| Wurzel 3 | Aufrunden | Abrunden | Kaufm. runden |
|-----------|-----------|----------|---------------|
| 1,7320508 | 2 | 1 | 1,73 |

1 Zeile ausgewählt



In [Beispiel 3.13](#) bzw. [Beispiel 3.14](#) ist zu sehen, dass es möglich ist, Funktionen in einander zu verschachteln. Mit Hilfe des Ausdrucks `SQRT(3)` wird die Quadratwurzel der Zahl 3 errechnet (1,73205080756888). Dieser Wert soll anschließend auf 2 Nachkommastellen gerundet werden. Hierzu kann auf beiden Systemen die Funktion `ROUND` herangezogen werden.

Bei der Abarbeitung des Ausdrucks `ROUND(SQRT(3), 2)` halten sich Oracle und SQL Server an die Gesetze der Arithmetik, d. h. es wird zuerst der innerste Ausdruck (in diesem Falle `SQRT(3)`) aufgelöst und anschließend wird das Ergebnis dieses Ausdrucks durch die Funktion `ROUND` auf zwei Nachkommastellen gerundet.

Die zweite Gruppe der arithmetischen Funktionen, die hier vorgestellt werden sollen, bilden die sogenannten „höheren Rechenarten“ ab. Dies sind: Radizieren, Potenzieren, Logarithmieren. Zusätzlich ist hier noch die Modulo-Operation dargestellt, die den Rest einer ganzzahligen Division ausgibt.

Listing 3.15: Höhere Rechenarten in Oracle

```
SELECT MOD(9, 2) AS Modulo, POWER(10, 2) AS Power,
       LOG(10, 1000) AS Log, SQRT(16) AS Quadratwurzel,
       SQRT(27) / SQRT(3) AS "3. Wurzel von 27",
       LOG(5, 8) AS "Log 8 Basis 5"
FROM dual;
```



| Modulo | Power | Log | Quadratwurzel | 3. Wurzel von 27 | Log 8 Basis 5 |
|--------|-------|-----|---------------|------------------|---------------|
| 1 | 100 | 3 | 4 | 3 | 1,292029 |

1 Zeile ausgewählt

In MS SQL Server können die gleichen Berechnungen durchgeführt werden, jedoch mit dem Unterschied, dass:

- Die Modulo-Operation durch den %-Operator dargestellt wird und nicht durch eine Funktion.
- Es gibt in MS SQL Server keine Entsprechung zur LOG-Funktion.

In MS SQL Server gibt es nur die Funktion LOG10 (Dekadischer Logarithmus zur Basis 10). Um nun die LOG-Funktion von Oracle nachzustellen muss hier $\log_5 8 = \log_{10} 8 / \log_{10} 5$ gerechnet werden.

Listing 3.16: Höhere Rechenarten in MS SQL Server

```
SELECT 9 % 2 AS Modulo, POWER(10, 2) AS Power,
       LOG10(1000) AS Log, SQRT(16) AS Quadratwurzel,
       SQRT(27) / SQRT(3) AS "3. Wurzel von 27",
       LOG10(8) / LOG10(5) AS "Log 8 Basis 5"
FROM dual;
```



| Modulo | Power | Log | Quadratwurzel | 3. Wurzel von 27 | Log 8 Basis 5 |
|--------|-------|-----|---------------|------------------|---------------|
| 1 | 100 | 3 | 4 | 3 | 1,292029 |

1 Zeile ausgewählt





- [i97801]
- [i77449]
- [i84140]
- [i77996]
- [i78493]
- [i78633]
- [ms189818]
- [ms178531]
- [ms175121]
- [ms174276]
- [ms175003]

3.4. Datumsfunktionen

3.4.1. Datumswerte

Der Umgang mit Datumswerten in einer Datenbank ist meist nicht einfach. Jedes RDBMS speichert Datumsangaben anders und behandelt diese auch anders. Aus diesem Grund soll an dieser Stelle erst einmal ein Überblick darüber gegeben werden, wie Oracle und MS SQL Server mit Datumswerten umgehen.

Tabelle 3.8.: Behandlung von Datumswerten

| Eigenschaft | | |
|----------------------|---|---|
| |  |  |
| Standarddatumsformat | DD-MON-YY (z. B. 12-NOV-08) | yyyy-mm-ddThh:mm:ss[.mmm] (z. B. 2004-05-23T14:25:10.487) |
| Speicherung | internes numerisches Format | internes numerisches Format |
| Wertebereich | Von 4713 vor Christus bis Dezember 9999. | Zwischen dem 1. Januar 1753 und dem 31. Dezember 9999. |
| Systemdatum anzeigen | SYSDATE / SYSTIMESTAMP | getdate() |

3.4.2. Datumsarithmetik in Oracle

Unter dem Begriff „Datumsarithmetik“ versteht man das Rechnen mit Datumswerten. In Oracle gibt es zwei Möglichkeiten:

- Addition oder Subtraktion von Zahlen zu einem Datumswert
- Verwendung von Interval-Literalen

Führt man Datumsarithmetik durch Addition oder Subtraktion von Zahlen durch gelten folgende Regeln:

- Ganze Zahlen sind Tage, z. B. 1 ist ein Tag, 15 sind fünfzehn Tage
- Fraktale (Nachkommastellen) sind Stunden, Minuten und Sekunden, z. B. $\frac{1}{24} = 0,041666$ ist eine Stunde oder $\frac{1}{60} = 0,000694444$ ist eine Minute
- Es darf nur addiert oder subtrahiert werden, alle anderen Rechenoperationen sind verboten.

Listing 3.17: Einfache Datumsarithmetik in Oracle

```
SELECT SYSDATE AS "Datum/Uhrzeit", SYSDATE + 2 AS "2 Tage",
       SYSDATE - 3 / 24 AS "3 Stunden"
FROM   dual;
```



| Datum/Uhrzeit | 2 Tage | 3 Stunden |
|---------------------|---------------------|---------------------|
| 30.04.2013 14:36:24 | 02.05.2013 14:36:24 | 30.04.2013 11:36:24 |

1 Zeile ausgewählt

Intervall-Literale (Oracle)

Intervall-Literale sind dazu da, um Zeiträume anzugeben. Diese können in Form von Jahren, Monaten, Tagen, Stunden, Minuten oder Sekunden ausgedrückt werden. Es gibt zwei grundsätzlich unterschiedliche Arten von Intervallen:

- YEAR TO MONTH
- DAY TO SECOND

Das YEAR TO MONTH Intervall kann aus bis zu zwei Feldern bestehen, wobei das erste die Jahre und das zweite die Monate angibt. [Tabelle 3.9](#) zeigt hierzu einige Beispiele.

Tabelle 3.9.: Das YEAR TO MONTH Intervall

| Beispiel | Bedeutung |
|-------------------------------|---|
| INTERVAL '10' YEAR | Ein Intervall von 10 Jahren (und 0 Monaten) |
| INTERVAL '101' YEAR(3) | Ein Intervall von 101 Jahren |
| INTERVAL '10' YEAR TO YEAR | Das gleiche wie INTERVAL '10' YEAR |
| INTERVAL '10-3' YEAR TO MONTH | Ein Intervall von 10 Jahren und 3 Monaten |
| INTERVAL '27' MONTH | Ein Intervall von 27 Monaten |

Bei der Angabe von Intervall-Literalen gibt es wichtige Dinge zu beachten:

- Die Zeitangabe wird immer in Hochkommas gesetzt,
- zu jedem Intervall muss angegeben werden, ob es sich um Jahre (YEAR) oder Monate (MONTH) handelt,
- die Standardpräzision eines YEAR TO MONTH Intervalls ist immer 2-stellig. Bei drei- oder mehrstelligen Jahresangaben, muss die Präzision angegeben werden. In [Beispiel 3.18](#) wird dieses Problem gezeigt.

Listing 3.18: Richtiger Umgang mit YEAR TO MONTH Intervallen

```
SELECT SYSDATE - INTERVAL '101' YEAR
FROM dual;

ORA-01873: the leading precision of the interval is too small
01873. 00000 - "the leading precision of the interval is too small"
*Cause:      The leading precision of the interval is too small to store the
              specified interval.
*Action:     Increase the leading precision of the interval or specify an
              interval with a smaller leading precision.

SELECT SYSDATE - INTERVAL '101' YEAR(3)
FROM dual;
```

SYSDATE-INTERVAL'101(3)' YEAR

02.05.03

1 Zeile ausgewählt



Mit der Präzision wird angegeben, wie viele Stellen die Jahresangabe haben darf. Der Standardwert ist 2.

Beispiel 3.18 zeigt, dass die Angaben `INTERVAL '101' YEAR` mit dem Oracle-Fehler ORA-01873 scheitert, da die Standardpräzision nur 2-stellig ist und daher bei einer dreistelligen Jahresangabe die Präzision durch die Angaben `YEAR(3)` auf drei Stellen erhöht werden muss.

Bei der zweiten Art von Intervall-Literal, dem DAY TO SECOND Intervall verhält es sich mit der Syntax genauso, wie beim YEAR TO MONTH Intervall. Tabelle 3.10 zeigt Beispiele für solche Intervalle.

Tabelle 3.10.: Das DAY TO SECOND Intervall

| Beispiel | Bedeutung |
|---|---|
| <code>INTERVAL '8' DAY</code> | Ein Intervall von 8 Tagen |
| <code>INTERVAL '8 4' DAY TO HOUR</code> | Ein Intervall von 8 Tagen und 4 Stunden |
| <code>INTERVAL '8 4:25' DAY TO MINUTE</code> | Ein Intervall von 8 Tagen, 4 Stunden und 25 Minuten |
| <code>INTERVAL '8 4:25:10' DAY TO SECOND</code> | 8 Tage, 4 Stunden, 25 Minuten und 10 Sekunden |
| <code>INTERVAL '120' DAY(3)</code> | 120 Tage (Präzision 3!!!) |
| <code>INTERVAL '3' HOUR</code> | 3 Stunden |
| <code>INTERVAL '3:18' HOUR TO MINUTE</code> | 3 Stunden und 18 Minuten |
| <code>INTERVAL '3:18:10' HOUR TO SECOND</code> | 3 Stunden, 18 Minuten und 10 Sekunden |
| <code>INTERVAL '210' HOUR</code> | 210 Stunden |
| <code>INTERVAL '18' MINUTE</code> | 18 Minuten |
| <code>INTERVAL '18:10' MINUTE TO SECOND</code> | 18 Minuten und 10 Sekunden |
| <code>INTERVAL '120' MINUTE</code> | 120 Minuten |
| <code>INTERVAL '10' SECOND</code> | 10 Sekunden |
| <code>INTERVAL '180' SECOND</code> | 180 Sekunden |



Für das DAY TO SECOND Intervall gelten, bezüglich der Präzision, die gleichen Regeln, wie beim YEAR TO MONTH Intervall.

Manchmal ist es notwendig Zeitintervalle zu formulieren, die zu keinem der beiden Typen passen. Dies könnte z. B. 4 Jahre, 10 Monate, 8 Tage und 5 Stunden sein. Auch das ist möglich, wie Beispiel 3.19 zeigt.

Listing 3.19: Ein komplexe Zeitintervall

```
SELECT SYSDATE - INTERVAL '4-10' YEAR TO MONTH -
        INTERVAL '8 5' DAY TO HOUR AS Interval
FROM dual;
```



INTERVAL

24.06.08

1 Zeile ausgewählt

3.4.3. Datumsarithmetik in MS SQL Server

Unter dem Begriff „Datumsarithmetik“ versteht man das Rechnen mit Datumswerten. In SQL Server stehen hierfür die Funktionen:

- **DATEADD**
- **DATEDIFF**
- **DATEPART**
- **DATENAME**

zur Verfügung. Sie verarbeiten Datumswerte und Zeitintervalle.

Die verschiedenen Intervalle, die für die genannten Funktionen zur Verfügung stehen sind: NANOSECOND, MICROSECOND, MILLISECOND, SECOND, MINUTE, HOUR, WEEKDAY, WEEK, DAY, DAYOFYEAR, MONTH, QUARTER, YEAR.



Die Funktion DATEADD - Datumswerte addieren

Beispiel 3.20 zeigt, die Anwendung der Funktion **DATEADD**

Listing 3.20: Die Funktion **DATEADD** in SQL Server

```
SELECT GETDATE(), DATEADD(DAY, 1, GETDATE()), DATEADD(HOUR, 1, GETDATE())
       DATEADD(MINUTE, 1, GETDATE());
```

| (Kein Spaltenname) | (Kein Spaltenname) |
|-------------------------|-------------------------|
| 2013-05-02 12:00:36.047 | 2013-05-03 12:00:36.047 |
| (Kein Spaltenname) | (Kein Spaltenname) |
| 2013-05-02 13:00:36.047 | 2013-05-02 12:01:36.050 |



Diese Funktion ermöglicht es, ein Zeitintervall zu einem Datum zu addieren.

Die Funktion DATEDIFF - Eine Differenz bilden

Um die Möglichkeit zu schaffen, die Differenz zwischen zwei Datumswerten zu bilden, wurde die Funktion **DATEDIFF** in MS SQL Server integriert.

Listing 3.21: Die Funktion **DATEDIFF** in SQL Server

```
SELECT GETDATE(),
       DATEDIFF(DAY, CONVERT(DATETIME2, '01.05.2010', 104), GETDATE()),
       DATEDIFF(YEAR, CONVERT(DATETIME2, '01.05.2010', 104), GETDATE());
```



| (Kein Spaltenname) | (Kein Spaltenname) | (Kein Spaltenname) |
|-------------------------|--------------------|--------------------|
| 2013-05-13 11:49:34.730 | 1108 | 3 |

Das Ergebnis dieser Funktion ist die Differenz zwischen dem Startdatum und dem Enddatum, in dem angegebenen Intervall.

Die Funktionen DATEPART/DATENAME - Teile eines Datums extrahieren

Mit Hilfe der Funktionen **DATEPART** und **DATENAME** können Teile eines Datums, als Zeichenkette extrahiert werden.

Listing 3.22: **DATEPART** und **DATENAME** in SQL Server

```
SELECT GETDATE(), DATEPART(YEAR, GETDATE()),
       DATEPART(MONTH, GETDATE()) AS "DATEPART",
       DATENAME(MONTH, GETDATE()) AS "DATENAME";
```



| (Kein Spaltenname) | (Kein Spaltenname) |
|-------------------------|--------------------|
| 2013-05-13 11:49:34.730 | 2013 |
| (DATEPART) | (DATENAME) |
| 5 | Mai |

3.5. Sonstige Funktionen

Das NULL-Werte etwas besonderes darstellen wurde in [Abschnitt 1.3.4](#) bereits erläutert. Welche Effekte durch diese Besonderheit auftreten, war in [Beispiel 1.5](#) zu sehen. Dieser Abschnitt zeigt nun, wie man mit der Besonderheit der NULL-Werte umgeht.

Oracle und SQL Server kennen Funktionen, um dieser Problematik Herr zu werden. In Oracle ist es die **NVL**, in SQL Server die **ISNULL** (nicht zu verwechseln mit **IS NULL**) Funktion. Beide Funktionen ersetzen NULL-Werte durch einen nahezu frei wählbaren Ersatzwert. **Beispiel 3.23** zeigt die Funktion **NVL** in Oracle.

Listing 3.23: Die Funktion **NVL**

```
SELECT Gehalt + (Gehalt / 100 * Provision) AS "Mit NULL",
       Gehalt + (Gehalt / 100 * NVL(Provision, 0)) AS "Ohne NULL"
FROM   Mitarbeiter;
```

| Mit NULL | Ohne NULL |
|------------------------------|-----------|
| | 30000 |
| | 30000 |
| ... | ... |
| 2400 | 2400 |
| 2500 | 2500 |
| | 2000 |
| | 1500 |
| 1200 | 1200 |
| 101 Zeilen ausgewählt | |



In **Beispiel 3.22** geschieht folgendes:

- Bei der Berechnung von $Gehalt + (Gehalt / 100 * Provision)$ kommt die Problematik mit den NULL-Werten zum Tragen und es wird, in einigen Zeilen, der Wert NULL angezeigt.
- Bei der Berechnung von $Gehalt + (Gehalt / 100 * NVL(Provision, 0))$ werden die in der Spalte PROVISION auftretenden NULL-Werte von NVL durch den Wert 0 ersetzt, so dass die Berechnung ein gültiges Ergebnis liefern kann.

Gleiches geschieht beim MS SQL Server durch die Funktion **ISNULL**, wie in **Beispiel 3.24** zu sehen ist.

Listing 3.24: Die Funktion **ISNULL**

```
SELECT Gehalt + (Gehalt / 100 * Provision) AS "Mit NULL",
       Gehalt + (Gehalt / 100 * ISNULL(Provision, 0)) AS "Ohne NULL"
FROM   Mitarbeiter;
```





| Mit NULL | Ohne NULL |
|------------------------------|--------------|
| | 30000.000000 |
| | 30000.000000 |
| ... | ... |
| 2400.000000 | 2400.000000 |
| 2500.000000 | 2500.000000 |
| 101 Zeilen ausgewählt | |

3.6. Datentypen

Datentypen helfen in der Programmierung konkrete Wertebereiche und darauf definierte Operationen festzulegen. Ist eine Tabellenspalte beispielsweise so erstellt worden, dass sie nur Datumswerte akzeptiert, können dort keine anderen Werte, wie z. B. Zahlen oder Zeichenketten, gespeichert werden. Auch die für Datumswerte definierten Operationen sind exakt begrenzt. Während Addition oder Subtraktion von Zahlen zu einem Datumswert erlaubt sind, ist die Addition zweier Datumswerte nicht möglich. Ebenso verhält es sich mit Zahlen und Zeichenketten. Auf Zahlen können die Rechenoperationen der Arithmetik (+, -, * und /) angewandt werden, auf Zeichenketten nicht.

Tabelle 3.11 liefert einen kleinen Ausschnitt aus der Menge der Datentypen, die Oracle und SQL Server kennen und erläutert deren Bedeutung.

Tabelle 3.11.: Datentypen



|  |  | Bedeutung |
|---|---|--|
| NUMBER | NUMERIC | Numerische Datentypen mit fester Genauigkeit und fester Anzahl von Dezimalstellen. |
| DATE | DATETIME2 | Datums- und Zeitdatentypen zum Darstellen von Datum und Tageszeit. |
| TIMESTAMP | DATETIME2 | Ein Datums- und Zeitdatentyp mit höherer Genauigkeit als DATE. |
| VARCHAR2 | VARCHAR | Nicht-Unicode-Zeichendaten variabler Länge. |
| CHAR | CHAR | Nicht-Unicode-Zeichendaten fester Länge |

3.6.1. Numerische Datentypen

Aufbau

Die beiden Datentypen NUMBER (Oracle) und NUMERIC (SQL Server) sind dazu da, um positive oder negative numerische Werte, mit oder ohne Nachkommastellen, aufzunehmen. Die Wertebereiche, die beide Datentypen aufnehmen können, unterscheiden sich.

Tabelle 3.12.: Datentypen

| | | Wertebereich |
|---|---------|--|
|  | NUMBER | $\pm 1.0 * 10^{-130}$ bis $\pm 1.0 * 10^{126} - 1$ |
|  | NUMERIC | $-1.0 * 10^{-38}$ bis $1.0 * 10^{38} - 1$ |

Zahlen die größer oder kleiner als die angegebenen Wertebereiche sind, können nicht aufgenommen werden.



Präzision und Skalarität

Unter der Präzision versteht man die Angabe, bei wie vielen Stellen insgesamt noch ein rundungsfehlerfreies Ergebnis angezeigt werden kann. Die maximale Präzision beider Typen beschränkt sich auf Zahlen, die kleiner oder gleich 10^{38} sind. Daraus folgt, dass solange sich eine Zahl in diesem Wertebereich befindet, sie frei von Rundungsfehlern ist. Ist sie größer, können Rundungsfehler auftreten. Beim Microsoft SQL Server stellt sich diese Problematik nicht, da bei NUMERIC der Wertebereich auf 10^{38} beschränkt ist.

Um die benötigte Präzision einstellen zu können, ist es auf beiden Systemen möglich, die maximale Anzahl Stellen, die eine Tabellenspalte aufnehmen können soll, auf einen Wert zwischen 1 und 38 einzuschränken. Ist eine Spalte, z. B. auf neun Stellen begrenzt, ist die größte Zahl, die in diese Spalte eingefügt werden kann, die 999.999.999.

Mit Hilfe der Skalarität kann manuell festgelegt werden, wie viele der durch die Präzision angegebenen Stellen rechts vom Komma verwendet werden. Wird eine Spalte mit einer Präzision von neun und einer Skalarität von zwei definiert, kann sie maximal sieben Stellen links und zwei Stellen rechts vom Komma enthalten. Die größte Zahl, die in eine solche Spalte eingefügt werden kann, ist somit die 9.999.999,99.

3.6.2. Zeichendatentypen

Typen fester Länge

Bei den Zeichendatentypen wird nach Typen fester Länge und Typen variabler Länge unterschieden. Der Datentyp zur Aufnahme von Zeichenketten fester Länge, heißt in beiden Systemen CHAR. Datentypen fester Länge haben ihren Namen daher, dass bei der Definition einer Tabellenspalte, mit einem solchen Typ, die Länge der Spalte fest angegeben werden muss.



Der Speicherplatzverbrauch einer solchen Spalte richtet sich nicht nach ihrem Inhalt (Anzahl der enthaltenen Zeichen), sondern nach der vorgegebenen Größe. Wird eine Spalte mit einer Größe von 20 definiert, beträgt ihr Speicherplatzverbrauch 20, 40 oder 80 Byte^a. Dies trifft auch dann zu, wenn die Spalte nur ein einziges Zeichen enthält.

^aJe nach dem, welcher Zeichensatz verwendet wird, werden pro Zeichen 1, 2 oder 4 Byte Speicherplatz benötigt.

Typen variabler Länge

Die Zeichendatentypen variabler Länge unterscheiden sich in Oracle und SQL Server nur in ihrem Namen. SQL Server verwendet die SQL-92-Standard gemäße Bezeichnung VARCHAR, während Oracle die Bezeichnung VARCHAR2, als Synonym für VARCHAR verwendet. Die Nutzung von VARCHAR ist aber auch in Oracle zulässig.



Im Unterschied zu den Typen fester Länge, ergibt sich der Speicherplatzverbrauch bei Typen variabler Länge nicht durch eine fest definierte Größe, sondern anhand ihres Inhalts. Es kann eine maximale Größe angegeben werden. Die Spalte kann dann maximal so viele Zeichen aufnehmen, wie angegeben.

3.6.3. Datums- und Zeittypen

Zur Speicherung von Datums- und Zeitwerten kennt Oracle die beiden Datentypen DATE und TIME-STAMP. Microsoft SQL Server verwendet DATETIME2.

In MS SQL Server gibt es auch einen Datentyp **TIMESTAMP**. Dieser ist jedoch, abweichend vom SQL-2003-Standard, nicht zur Speicherung von Datums- und Zeitwerten vorgesehen und seit SQL Server 2008 als veraltet eingestuft.



Oracle - DATE und TIMESTAMP

Der Datentyp **DATE** speichert seine Datumswerte in einem internen, numerischen Format. Er berücksichtigt dabei: Jahrzehnt, Jahr, Monat, Tag, Stunde, Minute, Sekunde. Der Typ **TIMESTAMP** ist eine Erweiterung. Er kann Datums- und Zeitangaben auf bis zu 9 Stellen (Nanosekunde) genau, im Sekundenbereich speichern.

SQL Server - DATETIME2

Werte des **DATETIME2**-Datentyps werden von der Microsoft SQL Server 2008 R2 Database Engine (Datenbankmodul) intern, als zwei 4 Byte lange, ganze Zahlen, gespeichert. Die ersten 4 Byte enthalten die Anzahl von Tagen, vor oder nach dem Referenzdatum, dem 1. Januar 1900. Die anderen 4 Byte speichern die Tageszeit, die als Anzahl von Millisekunden seit Mitternacht dargestellt wird.

3.7. Konvertierung von Datentypen

Unter der Konvertierung von Datentypen versteht man das Umwandeln eines Wertes, eines bestimmten Typs, in einen anderen Typ. Beispielsweise kann eine Zeichenkette „1234“ in die gleichlautende Zahl „1234“ umgewandelt werden. Intern wird nur der Datentyp von **VARCHAR/VARCHAR2** auf **NUMERIC/NUMBER** geändert. Dies ist z. B. notwendig, um arithmetische Operationen durchzuführen.

3.7.1. Implizite Datentypkonvertierung

Unter der impliziten Konvertierung versteht man, dass automatische Umwandeln eines Datentyp durch das DBMS in einen Anderen.



In [Beispiel 3.25](#) wird die implizite Konvertierung der Gehälter in Zeichenketten gezeigt.

Listing 3.25: Implizite Konvertierung von NUMBER zu VARCHAR2

```
SELECT 'Das Gehalt von ' || Nachname || ' betraegt: ' || gehalt
      AS "Implizite Konvertierung"
FROM   Mitarbeiter;
```



Implizite Konvertierung

```
Das Gehalt von Winter betraegt: 88000
Das Gehalt von Werner betraegt: 50000
Das Gehalt von Seifert betraegt: 50000
Das Gehalt von Schwarz betraegt: 30000
101 Zeilen ausgewählt
```

Damit Oracle eine Ausgabe wie „Das Gehalt von Winter betraegt: 88000“ darstellen kann, muss ein einheitlicher Datentyp geschaffen werden. Hierzu wird ein Ausdruck in zwei Teile zerlegt, einen linken und einen rechten. Der rechte Teil wird dann, sofern möglich, in den Datentyp des Linken konvertiert. Für [Beispiel 3.25](#) bedeutet dies konkret:


- Linke Seite: 'Das Gehalt von ' || Nachname || 'betraegt: '
- Rechte Seite: Gehalt
- Datentyp der linken Seite: VARCHAR2
- Datentyp der rechten Seite: NUMBER
- Daraus folgt: Konvertiere NUMBER nach VARCHAR2

Nicht immer ist das Konvertieren eines Datentyps in einen anderen möglich. Für die implizite Konvertierung gibt es einige Einschränkungen.

- Der Wert selbst muss in den neuen Typ konvertierbar sein. Beispielsweise kann die Zeichenkette ABCDE nicht in eine Zahl oder ein Datum umgewandelt werden, da sie kein sinnvolles Datum darstellt.
- Der Datentyp selbst muss in den neuen Datentyp konvertierbar sein. Zum Beispiel kann Oracle einen Wert des Datentyps NUMBER nicht direkt in einen Wert des Typs DATE konvertieren, da hierzu ein Referenzdatum benötigt wird.


Die beiden folgenden Tabellen zeigen, welche impliziten Typkonvertierung in Oracle und SQL Server möglich sind. Dabei wird immer zugrunde gelegt, dass der betreffende Wert auch konvertierbar ist.

Tabelle 3.13.: Implizite Datentypkonvertierung in Oracle



| | NUMBER | VARCHAR2 | CHAR | DATE | TIMESTAMP |
|-----------|--------|----------|------|------|-----------|
| NUMBER | – | X | X | – | – |
| VARCHAR2 | X | – | X | X | X |
| CHAR | X | X | – | X | X |
| DATE | – | X | X | – | – |
| TIMESTAMP | X | X | X | – | – |

Tabelle 3.14.: Implizite Datentypkonvertierung in Microsoft SQL Server



| | NUMERIC | VARCHAR | CHAR | DATETIME2 |
|----------|---------|---------|------|-----------|
| NUMERIC | – | X | X | X |
| VARCHAR | X | – | X | X |
| CHAR | X | X | – | X |
| DATETIME | – | X | X | – |

Der SQL Server besitzt intern eine Rangfolge seiner Datentypen. Dadurch wird bei der impliziten Konvertierung der Datentyp mit der niedrigeren Rangfolge in den Datentyp mit der höheren Rangfolge umgewandelt.

- [autoId8]
- [ms191530]
- [ms190309]



3.7.2. Explizite Datentypkonvertierung

Explizite Datentypkonvertierung in Oracle




Bei der expliziten Datentypkonvertierung werden Werte, mit Hilfe von Funktionen, von einem Datentyp in einen anderen konvertiert.

Oracle kennt für das explizite Konvertieren von Daten verschiedene Funktionen. Diese sind:

- `TO_CHAR`
- `TO_DATE`
- `TO_TIMESTAMP`
- `TO_NUMBER`

Es existieren noch weitere Funktionen, die an dieser Stelle jedoch ungenannt bleiben. Die folgende Tabelle zeigt eine Übersicht, welche Datentypen, mit Hilfe der expliziten Datentypkonvertierung umgewandelt werden können.

Tabelle 3.15.: Explizite Datentypkonvertierung in Oracle



| | NUMBER | VARCHAR2 | CHAR | DATE | TIMESTAMP |
|-----------|-----------|----------|---------|---------|--------------|
| NUMBER | – | TO_CHAR | TO_CHAR | | – |
| VARCHAR2 | TO_NUMBER | – | – | TO_DATE | TO_TIMESTAMP |
| CHAR | TO_NUMBER | – | – | TO_DATE | TO_TIMESTAMP |
| DATE | – | TO_CHAR | TO_CHAR | – | TO_TIMESTAMP |
| TIMESTAMP | – | TO_CHAR | TO_CHAR | – | – |

Beispiel 3.26 zeigt die Umwandlung des Systemdatums in eine Zeichenkette.

Listing 3.26: Explizite Datentypkonvertierung in Oracle

```
SELECT TO_CHAR(SYSDATE)
FROM   dual;
```

TO_CHAR(SYSDATE)

02.05.13

1 Zeile ausgewählt



Tatsächlich geschieht in [Beispiel 3.26](#) nichts sichtbares, dennoch hat eine Umwandlung stattgefunden. Um diese sichtbar zu machen, kann während der Konvertierung eine Formatierung des Ergebniswertes durchgeführt werden.

Listing 3.27: Konvertierung und Formatierung

```
SELECT TO_CHAR(SYSDATE, 'DD.MM.YYYY')  
FROM dual;
```

TO_CHAR(SYSDATE, 'DD.MM.YYYY')

02.05.2013

1 Zeile ausgewählt



Das zweite Argument der **TO_CHAR**-Funktion, 'DD.MM.YYYY', wird als „Formatmodell“ bezeichnet. Mit Hilfe dieser Buchstabenkombination wird das Ausgabeformat für die Zeichenfolge festgelegt. Die Bedeutung der Buchstaben ist:

- DD: Tag 2-stellig
- MM: Monat 2-stellig
- YYYY: Jahr 4-stellig

[Beispiel 3.28](#) zeigt ein weiteres, individuelles Formatmodell für einen Datumswert.

Listing 3.28: Ein anderes Formatmodell

```
SELECT TO_CHAR(SYSDATE, 'DD. MON YYYY')  
FROM dual;
```

TO_CHAR(SYSDATE, 'DD. MONYYYY')

02. MAI 2013

1 Zeile ausgewählt

1 Zeile ausgewählt



Die Buchstabenkombination `MON` sorgt dafür, dass der Monat als Wort angezeigt wird. Ob „Mai“ oder „May“ angezeigt wird, ist abhängig von den Ländereinstellungen der Datenbank.



Ein Formatmodell ist eine Zeichenfolge, die das Format eines Datums oder eines numerischen Wertes beschreibt. Ein Formatmodell ändert nicht die interne Darstellung eines Wertes in der Datenbank, sondern formatiert lediglich die Ausgabe. Es setzt sich aus mehreren Formatelementen zusammen, z. B. `DD` oder `MM` oder `YYYY`.

Welche Formatmodelle erstellt werden können, hängt davon ab, welche Formatelemente die einzelnen Funktionen kennen. Die folgenden Literaturhinweise führen zu den Tabellen in der Oracle Online-Dokumentation, die alle existierenden Formatelemente enthält.



- [\[i34570\]](#)
- [\[i34924\]](#)



Für die Funktion `TO_CHAR` existiert keine eigenständige Zusammenstellung von Formatelementen, da sie sowohl Datumswerte in Text, als auch Zahlen in Text konvertiert und dafür die Formatelemente von `TO_NUMBER` und `TO_DATE` nutzt.

Explizite Datentypkonvertierung in Microsoft SQL Server



Bei der expliziten Datentypkonvertierung werden Werte, mit Hilfe von Funktionen, von einem Datentyp in einen anderen konvertiert.

MS SQL Server kennt die Funktion `CONVERT` zur Konvertierung von Datentypen ². Die Syntax dieser Funktion lautet:

Listing 3.29: Die Syntax der `CONVERT`-Funktion in MS SQL Server

```
CONVERT ( <Datentyp>[(Laenge)], <Ausdruck>[, Style])
```

Tabelle 3.16.: Funktionsargumente von `CONVERT`

²Es gibt zusätzlich die Funktion `CAST`. Jedoch wird seitens Microsoft empfohlen, `CONVERT` zu nutzen, obgleich die Verwendung von `CAST` dem ISO-Standard entsprechen würde.

| Argument | Beispiel | Erläuterung |
|------------|--------------|---|
| <Datentyp> | DATETIME | Dies ist die Angabe des Datentyps, in den <Ausdruck> konvertiert werden soll. Für jeden Datentyp kann optional eine Länge mit angegeben werden. |
| <Ausdruck> | '16.01.2009' | <Ausdruck> ist eine Zeichenkette, Zahl, ein Datums- Zeitwert oder eine Funktion. |
| [Style] | 104 | Optional kann bei der CONVERT-Funktion ein Formatmodell angegeben werden. |

Ein Formatmodell legt fest, wie ein Eingabewert interpretiert oder ein Ausgabewert formatiert werden soll. Alle Formatmodelle sind durch Zahlen kodiert.

Die Bedeutung der Formatmodelle soll konkret anhand von [Beispiel 3.30](#) erläutert werden.

Listing 3.30: CONVERT mit Formatmodell

```
SELECT CONVERT(DATETIME2, '02.05.2013', 104)
```

(Kein Spaltenname)

```
2013-05-02 00:00:00.0000000
```



[Beispiel 3.30](#) zeigt die Konvertierung der Zeichenkette „02.05.2013“ in ein Datum. Damit dies korrekt ablaufen kann, muss die Datenbank wissen, wie die einzelnen Teile der Zeichenkette zu verstehen sind. Die Formatmodellnummer 104 besagt, dass der angegebene Ausdruck im Format „DD.MM.YYYY“ zu interpretieren ist.

Was passiert, wenn ein zum Ausdruck inkompatibles Formatmodell angegeben wird, zeigt [Beispiel 3.30](#). Das Formatmodell „4“ liest den Ausdruck „02.05.2013“ als „DD.MM.YY“ (2-stellige Jahreszahl). Da der Ausdruck aber mit 4-stelliger Jahreszahl angegeben ist, führt dieses Beispiel zu einer Fehlermeldung.

Listing 3.31: CONVERT mit falschem Formatmodell

```
SELECT CONVERT(DATETIME2, '02.05.2013', 4)
```

```
Meldung 241, Ebene 16, Status 1, Zeile 1
```

```
Fehler beim Konvertieren einer Zeichenfolge in einen datetime-Wert.
```

In [Beispiel 3.30](#) diente **CONVERT** dazu, um die Zeichenfolge „02.05.2013“ korrekt zu interpretieren (Eingabeformat). Die gleiche Funktion kann aber auch das Ausgabeformat eines Ausdrucks bestimmen. [Beispiel 3.32](#) zeigt, wie das aktuelle Systemdatum in eine Zeichenkette konvertiert wird. Dabei wird die 2-stellige Jahresangabe in eine 4-stellige umformatiert.

Listing 3.32: Formatieren den Ausgabe mit **CONVERT**

```
SELECT GETDATE(), CONVERT(VARCHAR, GETDATE(), 104)
```

(Kein Spaltenname)

(Kein Spaltenname)

2013-05-02 17:18:24.763 02.05.2013

Welche Formatmodelle SQL Server kennt ist unter dem folgenden Literaturhinweis nachlesbar.



- [ms191530]

3.8. Übungen - Funktionen

1. Lassen Sie das aktuelle Datum auf dem Bildschirm ausgeben und benennen Sie die Spalte mit „Datum“.

```
Datum
-----
12.05.13
1 Zeile ausgewählt
```



2. Lassen Sie das aktuelle Datum mit Uhrzeit auf dem Bildschirm ausgeben und benennen Sie die Spalte mit „Datum/Uhrzeit“.

```
Datum/Uhrzeit
-----
12.05.13 10:58:45,439419 +02:00
1 Zeile ausgewählt
```



3. Schreiben Sie eine Abfrage, welche die Mitarbeiternummer, den Nachnamen, das Gehalt und ein um 3,5 % erhöhtes Gehalt für jeden Mitarbeiter anzeigt. Das erhöhte Gehalt soll als ganze Zahl und mit dem Spaltenalias „Neues Gehalt“ ausgegeben werden!

| MITARBEITER_ID | NACHNAME | GEHALT | Neues Gehalt |
|----------------|----------|--------|--------------|
| 1 | Winter | 88000 | 91080 |
| 2 | Werner | 50000 | 51750 |
| 3 | Seifert | 50000 | 51750 |
| 4 | Schwarz | 30000 | 31050 |

100 Zeilen ausgewählt



4. Verändern Sie die Abfrage, aus der vorangegangenen Aufgabe so, dass eine zusätzliche Spalte hinzugefügt wird, die die Differenz zwischen dem alten und dem erhöhten Gehalt anzeigt. Benennen Sie die Spalte mit „Gehaltserhoehung“.

| MITARBEITER_ID | NACHNAME | GEHALT | Neues Gehalt | Gehaltserhoehung |
|----------------|----------|--------|--------------|------------------|
| 1 | Winter | 88000 | 91080 | 3080 |
| 2 | Werner | 50000 | 51750 | 1750 |
| 3 | Seifert | 50000 | 51750 | 1750 |
| 4 | Schwarz | 30000 | 31050 | 1050 |

100 Zeilen ausgewählt



5. Zeigen Sie die Nachnamen und die Länge der Nachnamen aller Mitarbeiter an, deren Nachname mit einem der Buchstaben „J“, „M“ oder „S“ beginnt. Die Spalten sollen, wie in der Lösung zu sehen ist, beschriftet sein. Die Nachnamen müssen in Großbuchstaben ausgegeben werden. Sortieren Sie die Abfrage in absteigender Reihenfolge nach den Nachnamen!



| Nachname | Laenge |
|-----------|--------|
| SINDERMAN | 10 |
| SINDERMAN | 10 |
| SIMON | 5 |
| SIMON | 5 |
| SIMON | 5 |
| SEIFERT | 7 |
| SEIFERT | 7 |
| SCHWARZ | 7 |
| SCHWARZ | 7 |

23 Zeilen ausgewählt

6. Zeigen Sie für jeden Mitarbeiter den Nachnamen an, sein Geburtsdatum und seit wie vielen Monaten dieser bereits 18 Jahre alt ist (gerundet auf zwei Stellen, nach dem Komma). Benennen Sie die Spalte mit den Monaten: „Alter in Monaten“. Sortieren Sie die Abfrage in aufsteigender Reihenfolge nach der Spalte „Alter in Monaten“.



Zur Lösung dieser Aufgabe mit Oracle soll die Funktion **MONTHS_BETWEEN** herangezogen werden, deren Syntax der Oracle Onlinedokumentation entnommen werden kann.



| NACHNAME | GEBURTSDATUM | Alter in Monaten |
|----------|--------------|------------------|
| Krüger | 31.05.93 | 23,4 |
| Walther | 07.01.93 | 28,18 |
| Lehmann | 07.11.92 | 30,18 |
| Keller | 04.11.92 | 30,27 |
| Schwarz | 27.06.92 | 34,53 |
| Weber | 10.06.92 | 35,08 |
| Peters | 13.05.92 | 35,98 |
| Köhler | 05.05.92 | 36,24 |
| Lorenz | 13.12.91 | 40,98 |

100 Zeilen ausgewählt

7. Ermitteln Sie Vorname, Nachname und Geburtsdatum der Mitarbeiter, die mindestens 1 Jahr und 4 Monate nach dem „07.05.1978“ geboren sind. Sortieren Sie die Abfrage in absteigender Reihenfolge nach dem Geburtsdatum.

| VORNAME | NACHNAME | GEBURTSDATUM |
|----------|----------|--------------|
| Emma | Krüger | 31.05.93 |
| Lina | Walther | 07.01.93 |
| Johannes | Lehmann | 07.11.92 |

81 Zeilen ausgewählt



8. Zeigen Sie für jeden Mitarbeiter, der zum Zeitpunkt der Ausführung dieser Abfrage mindestens 35 Jahre alt ist, dessen Mitarbeiter_ID, das Geburtsdatum und den Wochentag seiner Geburt an. Beschriften Sie die Spalten, wie in der Lösung vorgegeben. Ordnen Sie die Abfrage in aufsteigender Reihenfolge nach dem Wochentag, beginnend beim ersten Tag der Woche!

| MITARBEITER_ID | GEBURTSDATUM | Wochentag |
|----------------|--------------|------------|
| 42 | 31.01.77 | MONTAG |
| 90 | 14.12.76 | DIENSTAG |
| 36 | 14.02.78 | DIENSTAG |
| 2 | 03.11.77 | DONNERSTAG |
| 51 | 19.02.76 | DONNERSTAG |

11 Zeilen ausgewählt



9. Schreiben Sie eine Abfrage, die für alle Mitarbeiter deren Nachnamen und die Bankfiliale_ID anzeigt. Wenn ein Mitarbeiter in keiner Bankfiliale tätig ist, soll „Keine Bankfiliale“ angezeigt werden.

| NACHNAME | BANKFILIALE |
|------------|-------------------|
| Möller | Keine Bankfiliale |
| Winter | Keine Bankfiliale |
| Meier | Keine Bankfiliale |
| Sindermann | Keine Bankfiliale |
| Schwarz | Keine Bankfiliale |
| Werner | Keine Bankfiliale |
| Krüger | 1 |
| Peters | 1 |
| Kipp | 1 |

100 Zeilen ausgewählt



3.9. Lösungen - Funktionen

1. Lassen Sie das aktuelle Datum auf dem Bildschirm ausgeben und benennen Sie die Spalte mit „Datum“.



```
SELECT SYSDATE AS "Datum"  
FROM dual;
```



```
SELECT GETDATE() AS "Datum";
```

2. Lassen Sie das aktuelle Datum mit Uhrzeit auf dem Bildschirm ausgeben und benennen Sie die Spalte mit „Datum/Uhrzeit“.



```
SELECT SYSTIMESTAMP AS "Datum/Uhrzeit"  
FROM dual;
```



```
SELECT GETDATE() AS "Datum/Uhrzeit";
```

3. Schreiben Sie eine Abfrage, welche die Mitarbeiternummer, den Nachnamen, das Gehalt und ein um 3,5 % erhöhtes Gehalt für jeden Mitarbeiter anzeigt. Das erhöhte Gehalt soll als ganze Zahl und mit dem Spaltenalias „Neues Gehalt“ ausgegeben werden!



```
SELECT Mitarbeiter_ID, Nachname, Gehalt,  
       ROUND(Gehalt * 1.035, 0) AS "Neues Gehalt"  
FROM Mitarbeiter;
```



```
SELECT Mitarbeiter_ID, Nachname, Gehalt,  
       CEILING(ROUND(Gehalt * 1.035, 0)) AS "Neues Gehalt"  
FROM Mitarbeiter;
```

4. Verändern Sie die Abfrage, aus der vorangegangenen Aufgabe so, dass eine zusätzliche Spalte hinzugefügt wird, die die Differenz zwischen dem alten und dem erhöhten Gehalt anzeigt. Benennen Sie die Spalte mit „Gehaltserhoehung“.

```
SELECT Mitarbeiter_ID, Nachname, Gehalt,
       ROUND(Gehalt * 1.035, 0) AS "Neues Gehalt",
       ROUND(Gehalt * 1.035, 0) - Gehalt AS "Gehaltserhoehung"
FROM   Mitarbeiter;
```



```
SELECT Mitarbeiter_ID, Nachname, Gehalt,
       CEILING(ROUND(Gehalt * 1.035, 0)) AS "Neues Gehalt",
       CEILING(ROUND(Gehalt * 1.035, 0)) - Gehalt AS "Gehaltserhoehung"
FROM   Mitarbeiter;
```



5. Zeigen Sie die Nachnamen und die Länge der Nachnamen aller Mitarbeiter an, deren Nachname mit einem der Buchstaben „J“, „M“ oder „S“ beginnt. Die Spalten sollen, wie in der Lösung zu sehen ist, beschriftet sein. Die Nachnamen müssen in Großbuchstaben ausgegeben werden. Sortieren Sie die Abfrage in absteigender Reihenfolge nach den Nachnamen!

```
SELECT UPPER(Nachname) AS "Nachname",
       LENGTH(Nachname) AS "Laenge"
FROM   Mitarbeiter
WHERE  (UPPER(Nachname) LIKE 'J%'
       OR UPPER(Nachname) LIKE 'M%'
       OR UPPER(Nachname) LIKE 'S%')
ORDER BY Nachname DESC;
```



```
SELECT UPPER(Nachname) AS "Nachname",
       LEN(Nachname) AS "Laenge"
FROM   Mitarbeiter
WHERE  (Nachname LIKE 'J%'
       OR Nachname LIKE 'M%'
       OR Nachname LIKE 'S%')
ORDER BY Nachname DESC;
```



6. Zeigen Sie für jeden Mitarbeiter den Nachnamen an, sein Geburtsdatum und seit wie vielen Monaten dieser bereits 18 Jahre alt ist (gerundet auf zwei Stellen, nach dem Komma). Benennen Sie die Spalte

mit den Monaten: „Alter in Monaten“. Sortieren Sie die Abfrage in aufsteigender Reihenfolge nach der Spalte „Alter in Monaten“.



```
SELECT  Nachname, Geburtsdatum,
        ROUND(MONTHS_BETWEEN(
            SYSDATE, Geburtsdatum +
            INTERVAL '18' YEAR), 2) AS "Alter in Monaten"
FROM      Mitarbeiter
ORDER BY 3;
```



```
SELECT  Nachname, Geburtsdatum,
        ROUND(DATEDIFF(MONTH, DATEADD(YEAR, 18, Geburtsdatum),
            GETDATE()), 2) AS "Alter in Monaten"
FROM      Mitarbeiter
ORDER BY 3;
```

7. Ermitteln Sie Vorname, Nachname und Geburtsdatum der Mitarbeiter, die mindestens 1 Jahr und 4 Monate nach dem „07.05.1978“ geboren sind. Sortieren Sie die Abfrage in absteigender Reihenfolge nach dem Geburtsdatum.



```
SELECT  Vorname, Nachname, Geburtsdatum
FROM      Mitarbeiter
WHERE      Geburtsdatum >
            TO_DATE('07.05.1978', 'DD.MM.YYYY') +
            INTERVAL '1-4' YEAR TO MONTH
ORDER BY 3 DESC;
```



```
SELECT  Vorname, Nachname, Geburtsdatum
FROM      Mitarbeiter
WHERE      Geburtsdatum > DATEADD(MONTH, 4, DATEADD(YEAR, 1, '07.05.1978'))
ORDER BY 3 DESC;
```

8. Zeigen Sie für jeden Mitarbeiter, der zum Zeitpunkt der Ausführung dieser Abfrage mindestens 35 Jahre alt ist, dessen Mitarbeiter_ID, das Geburtsdatum und den Wochentag seiner Geburt an. Beschriften Sie die Spalten, wie in der Lösung vorgegeben. Ordnen Sie die Abfrage in aufsteigender Reihenfolge nach dem Wochentag, beginnend beim ersten Tag der Woche!



```
SELECT  Mitarbeiter_ID, Geburtsdatum,
        TO_CHAR(Geburtsdatum, 'DAY') AS "Wochentag"
FROM    Mitarbeiter
WHERE   SYSDATE > Geburtsdatum + INTERVAL '35' YEAR
ORDER BY TO_CHAR(Geburtsdatum, 'D');
```



```
SELECT  Mitarbeiter_ID, Geburtsdatum,
        DATENAME(WEEKDAY, Geburtsdatum) AS "Wochentag"
FROM    Mitarbeiter
WHERE   GETDATE() > DATEADD(YEAR, 35, Geburtsdatum)
ORDER BY DATEPART(WEEKDAY, Geburtsdatum);
```

9. Schreiben Sie eine Abfrage, die für alle Mitarbeiter deren Nachnamen und die Bankfiliale_ID anzeigt. Wenn ein Mitarbeiter in keiner Bankfiliale tätig ist, soll „Keine Bankfiliale“ angezeigt werden.



```
SELECT  Nachname, NVL(
        TO_CHAR(Bankfiliale_ID), 'Keine Bankfiliale') AS Bankfiliale
FROM    Mitarbeiter
ORDER BY Bankfiliale_ID;
```



```
SELECT  Nachname,
        ISNULL(CONVERT(VARCHAR, Bankfiliale_ID),
        'Keine Bankfiliale') AS Bankfiliale
FROM    Mitarbeiter
ORDER BY Bankfiliale_ID;
```


4. Erweiterte Datenselektion

Inhaltsangabe

| | | |
|------------|--|-------------|
| 4.1 | Der Inner Join | 4-2 |
| 4.1.1 | Die ON-Klausel | 4-2 |
| 4.1.2 | Tabellenaliasnamen | 4-3 |
| 4.1.3 | Die USING-Klausel (Nur Oracle) | 4-4 |
| 4.1.4 | Der Natural-Join (Nur Oracle) | 4-5 |
| 4.1.5 | Die Theta-Style Syntax | 4-6 |
| 4.1.6 | Mehr als zwei Tabellen verknüpfen | 4-7 |
| 4.2 | Outer Joins | 4-8 |
| 4.2.1 | Left- und Right-Outer-Join | 4-8 |
| 4.2.2 | Der Full Outer Join | 4-12 |
| 4.3 | Spezielle Joins | 4-12 |
| 4.3.1 | Der Self-Join | 4-12 |
| 4.3.2 | Non-Equi-Joins | 4-16 |
| 4.4 | Mengenoperationen | 4-16 |
| 4.4.1 | Voraussetzungen zur Nutzung der SET-Operatoren | 4-17 |
| 4.4.2 | Die SET-Operatoren | 4-17 |
| 4.5 | Übungen - Erweiterte Datenselektion | 4-22 |
| 4.6 | Lösungen - Erweiterte Datenselektion | 4-27 |

Werden zwei Relationen R_1 und R_2 in einer Abfrage miteinander verknüpft, entsteht ein kartesisches Kreuzprodukt. Die Anzahl der Zeilen in diesem Produkt entspricht $R_1 * R_2$. Es bildet die Grundlage für eine Join-Operation, bei der aus einem Kreuzprodukt, mit Hilfe eines Selektionsausdruckes, gezielt die nicht benötigten Zeilen eliminiert werden.

4.1. Der Inner Join

Beim Inner Join werden, im Ergebnis der Abfrage, nur die Zeilen angezeigt, die der Join-Bedingung genügen.

4.1.1. Die ON-Klausel

Die **ON**-Klausel stellt die flexibelste und am Häufigsten genutzte Möglichkeit dar, um zwei Tabellen, in einer Join-Operation, miteinander zu verknüpfen. Dafür werden zwei Spaltenbezeichner und ein Operator benötigt. [Beispiel 4.1](#) zeigt einen Inner Join zwischen den beiden Tabellen KUNDE und EIGENKUNDE.

Listing 4.1: Ein Join zwischen Kunde und Eigenkunde

```
SELECT Vorname, Nachname, PLZ, Ort
FROM Kunde INNER JOIN Eigenkunde
      ON (Kunde.Kunden_ID = Eigenkunde.Kunden_ID);
```



| VORNAME | NACHNAME | PLZ | ORT |
|------------------------------|-----------|-------|--------------|
| Sophie | Junge | 39435 | Bördeaue |
| Hanna | Beck | 39439 | Güsten |
| Noah | Bunzel | 39435 | Egeln |
| Sebastian | Peters | 39240 | Staßfurt |
| Leni | Braun | 06425 | Alsleben |
| Jannis | Schreiber | 06406 | Bernburg |
| Noah | Rollert | 39435 | Wolmirsleben |
| Amelie | Becker | 06425 | Plötzkau |
| Christian | Keller | 06449 | Giersleben |
| 400 Zeilen ausgewählt | | | |

Im Vergleich zu allen Beispielen, die in den vorangegangenen Kapiteln zu sehen waren, ändert sich in [Beispiel 4.1](#) nur die **FROM**-Klausel. Hier werden zwei Tabellen, KUNDE und EIGENKUNDE, getrennt durch die beiden Schlüsselworte **INNER JOIN** angegeben. Diese Syntax stammt aus dem SQL-99-Standard und ist selbsterklärend.

In der **ON**-Klausel werden die beiden Spalten angegeben, mit deren Hilfe die Verknüpfung zwischen den Tabellen hergestellt wird. Wichtig für diese beiden Spalten ist, dass sie beide miteinander vergleichbare Werte enthalten. Eine Namensgleichheit beider Spalten ist jedoch nicht notwendig.

Da beide Spalten den Bezeichner **KUNDEN_ID** haben, ist es notwendig die Spaltenbezeichner voll zu qualifizieren. Ein voll qualifizierter Spaltenbezeichner wird immer in der Form **TABELLENBEZEICHNER.SPALTENBEZEICHNER** angegeben.



Es wird empfohlen Spaltenbezeichner immer zu qualifizieren, da dies der Datenbank das Auffinden der Spalten erleichtert und somit die Performance des SQL-Statements steigt. Ohne die Qualifizierung der Spaltenbezeichner in der **ON**-Klausel antworten sowohl Oracle, als auch der MS SQL Server mit einer Fehlermeldung.

Listing 4.2: Eine fehlerhafte ON-Klausel in Oracle

```
SELECT Vorname, Nachname, PLZ, Ort
FROM   Kunde INNER JOIN Eigenkunde
       ON (Kunden_ID = Kunden_ID);
```

Fehler bei Befehlszeile:3 Spalte:24
Fehlerbericht:
SQL-Fehler: ORA-00918: column ambiguously defined
00918. 00000 - "column ambiguously defined"
*Cause:
*Action:

Listing 4.3: Eine fehlerhafte ON-Klausel in MS SQL Server

```
SELECT Vorname, Nachname, PLZ, Ort
FROM   Kunde INNER JOIN Eigenkunde
       ON (Kunden_ID = Kunden_ID);
```

Meldung 209, Ebene 16, Status 1, Zeile 3
Mehrdeutiger Spaltenname 'Kunden_ID'.
Meldung 209, Ebene 16, Status 1, Zeile 3
Mehrdeutiger Spaltenname 'Kunden_ID'.

4.1.2. Tabellenaliasnamen

Genau wie bei Spaltenbezeichnern existiert auch für Tabellenbezeichner die Möglichkeit, Aliasnamen festzulegen. Der Vorteil solcher Tabellenaliasnamen liegt darin, dass die Länge eines SQL-Statements, durch die Vergabe von sehr kurzen Aliasnamen, stark reduziert werden kann. [Beispiel 4.4](#) produziert das gleiche Ergebnis, wie [Beispiel 4.1](#), nutzt jedoch Aliasnamen für die beiden Tabellen.

Listing 4.4: Die Benutzung von Tabellenaliasnamen

```

SELECT Vorname, Nachname, PLZ, Ort
FROM Kunde k INNER JOIN Eigenkunde ek
      ON (k.Kunden_ID = ek.Kunden_ID);

```



| VORNAME | NACHNAME | PLZ | ORT |
|-----------|----------|-------|----------|
| Sophie | Junge | 39435 | Bördeaue |
| Hanna | Beck | 39439 | Güsten |
| Noah | Bunzel | 39435 | Egeln |
| Sebastian | Peters | 39240 | Staßfurt |

400 Zeilen ausgewählt



Tabellenaliasnamen gelten nur innerhalb eines Statements und beeinflussen die Struktur der Datenbank nicht. Wird ein Tabellenaliasname vergeben, so muss er im gesamten SQL-Statement genutzt werden!

Die bereits bekannten Klauseln **WHERE** und **ORDER BY** können auch in einer Join-Abfrage genutzt werden. In [Beispiel 4.5](#) wird das Ergebnis auf die Kunden mit Wohnort „Egeln“ reduziert und eine aufsteigende Sortierung nach dem Feld **NACHNAME** eingerichtet.

Listing 4.5: Join mit einschränkender WHERE-Klausel und Sortierung

```

SELECT Vorname, Nachname, PLZ, Ort
FROM Kunde k INNER JOIN Eigenkunde ek
      ON (k.Kunden_ID = ek.Kunden_ID)
WHERE Ort LIKE 'Egeln'
ORDER BY Nachname;

```



| VORNAME | NACHNAME | PLZ | ORT |
|---------|----------|-------|-------|
| Alina | Braun | 39435 | Egeln |
| Noah | Bunzel | 39435 | Egeln |
| Hanna | Bunzel | 39435 | Egeln |
| Paul | Koch | 39435 | Egeln |

18 Zeilen ausgewählt

4.1.3. Die USING-Klausel (Nur Oracle)

Die **USING**-Klausel stellt eine weitere Möglichkeit dar, eine Join-Operation durchzuführen. Sie ist eine Kurzschreibweise für **ON R1.Spalte = R2.Spalte**.

Listing 4.6: Die USING-Klausel

```

SELECT  Vorname, Nachname, PLZ, Ort
FROM    Kunde k INNER JOIN Eigenkunde ek
        USING (Kunden_ID)
WHERE   Ort LIKE 'Egeln'
ORDER BY Nachname;

```

| VORNAME | NACHNAME | PLZ | ORT |
|---------|----------|-------|-------|
| Alina | Braun | 39435 | Egeln |
| Noah | Bunzel | 39435 | Egeln |
| Hanna | Bunzel | 39435 | Egeln |
| Paul | Koch | 39435 | Egeln |
| Karolin | Lange | 39435 | Egeln |
| Marie | Lehmann | 39435 | Egeln |

18 Zeilen ausgewählt



Die Nutzung der **USING**-Klausel unterliegt auch einigen Einschränkungen.

- Die in der **USING**-Klausel genutzte Spalte darf nicht qualifiziert werden.
- Die in der **USING**-Klausel genutzte Spalte muss in den beiden, an der Join-Operation teilnehmenden Tabellen den gleichen Namen tragen.

Listing 4.7: Fehlerhafte Nutzung der USING-Klausel in Oracle

```

SELECT  Vorname, Nachname, PLZ, Ort
FROM    Kunde k INNER JOIN Eigenkunde ek USING (Kunden_ID)
WHERE   k.Kunden_ID = 200
ORDER BY Nachname;

```

Fehler bei Befehlszeile:4 Spalte:10
Fehlerbericht:
SQL-Fehler: ORA-00904: "D"."KUNDEN_ID": invalid identifier
00904. 00000 - "%s: invalid identifier"
*Cause:
*Action:

4.1.4. Der Natural-Join (Nur Oracle)

Die Natural-Join-Syntax stellt die dritte Variante zur Realisierung von Inner Joins dar.

Listing 4.8: Die Natural-Join-Syntax

```

SELECT  Vorname, Nachname, PLZ, Ort
FROM    Kunde k NATURAL JOIN Eigenkunde ek
WHERE   Ort LIKE 'Egeln'
ORDER BY Nachname;

```



| VORNAME | NACHNAME | PLZ | ORT |
|---------|----------|-------|-------|
| Alina | Braun | 39435 | Egeln |
| Noah | Bunzel | 39435 | Egeln |
| Hanna | Bunzel | 39435 | Egeln |

18 Zeilen ausgewählt

Beispiel 4.8 zeigt, dass bei dieser Syntax sowohl die **ON**-Klausel, als auch die **USING**-Klausel überflüssig sind. Dies rührt daher, dass Oracle automatisch die Spalten in den beiden Tabellen sucht, die den gleichen Namen und den gleichen Datentyp aufweisen. Es werden dabei so vielen Spalten einbezogen wie möglich.



Da Oracle immer alle Spalten mit gleichem Namen und gleichem Datentyp in den Natural-Join einbezieht, sollte diese Syntax mit bedacht genutzt werden!

4.1.5. Die Theta-Style Syntax



Sowohl in Oracle, als auch in MS SQL Server kann die Theta-Style-Syntax nur noch zur Realisierung von Inner Joins genutzt werden. Bis auf wenige Ausnahmen ist daher die ANSI-Style-Syntax, mit dem Schlüsselwort **INNER JOIN**, vorzuziehen!

Die Theta-Style-Syntax stellt die Urvariante der Join-Syntax dar, die auch schon vor dem SQL-99-Standard existierte. Bei dieser Form der Syntax wird in der **FROM**-Klausel nur eine kommaseparierte Liste von Tabellen angegeben, während die Verknüpfungsbedingung in der **WHERE**-Klausel formuliert wird.

Listing 4.9: Ein Inner Join mit Theta-Style-Syntax

```

SELECT  Vorname, Nachname, PLZ, Ort
FROM    Kunde k, Eigenkunde ek
WHERE   k.Kunden_ID = ek.Kunden_ID
        AND Ort LIKE 'Egeln'
ORDER BY Nachname;

```

| VORNAME | NACHNAME | PLZ | ORT |
|---------|----------|-------|-------|
| Alina | Braun | 39435 | Egeln |
| Noah | Bunzel | 39435 | Egeln |
| Hanna | Bunzel | 39435 | Egeln |



4.1.6. Mehr als zwei Tabellen verknüpfen

Bei komplexeren Abfragen ist es oft notwendig, auf die Daten von mehr als nur zwei Tabellen zuzugreifen. Dies kann mit allen bisher gezeigten Syntax-Varianten geschehen.

Da der MS SQL Server sowohl die **USING**-Klausel, als auch die **NATURAL JOIN**-Klausel nicht kennt, kann dieser nur die ANSI-Style-Syntax und den Theta-Style nutzen!



Listing 4.10: Vier Tabellen, verbunden durch Inner Joins

```
SELECT  Vorname, Nachname, IBAN
FROM    Kunde k INNER JOIN Eigenkunde ek
        ON (k.Kunden_ID = ek.Kunden_ID)
        INNER JOIN EigenkundeKonto ekk
        ON (ek.Kunden_ID = ekk.Kunden_ID)
        INNER JOIN Konto ko
        ON (ekk.Konto_ID = ko.Konto_ID)
WHERE   Ort LIKE 'Egeln'
ORDER BY Nachname, IBAN;
```

| VORNAME | NACHNAME | IBAN |
|---------|----------|------------------------|
| Alina | Braun | DE2327682878309669110 |
| Alina | Braun | DE23582034208834002588 |
| Hanna | Bunzel | DE23343859500956216053 |
| Noah | Bunzel | DE23419162344850780394 |
| Noah | Bunzel | DE23506210719641227144 |

46 Zeilen ausgewählt



Für die Ausführung des SQL-Statements ist die Reihenfolge, in der die Tabellen miteinander verbunden werden, nicht wichtig.



Das gleiche Ergebnis lässt sich auch mit der Theta-Style-Syntax erzielen, wie [Beispiel 4.11](#) zeigt.

Listing 4.11: Ein komplexer Join in der Theta-Style-Syntax

```

SELECT  Vorname, Nachname, IBAN
FROM    Kunde k, Eigenkunde ek, EigenkundeKonto ekk, Konto ko
WHERE   k.Kunden_ID = ek.Kunden_ID
        AND ek.Kunden_ID = ekk.Kunden_ID
        AND ekk.Konto_ID = ko.Konto_ID
        AND Ort LIKE 'Egeln'
ORDER BY Nachname, IBAN;

```



| VORNAME | NACHNAME | IBAN |
|---------|----------|------------------------|
| Alina | Braun | DE2327682878309669110 |
| Alina | Braun | DE23582034208834002588 |
| Hanna | Bunzel | DE23343859500956216053 |
| Noah | Bunzel | DE23419162344850780394 |
| Noah | Bunzel | DE23506210719641227144 |
| Hanna | Bunzel | DE23916870475976982996 |
| Paul | Koch | DE23337659559291799957 |
| Paul | Koch | DE23747825550493162192 |
| Karolin | Lange | DE2338135354878273969 |
| Karolin | Lange | DE23657965268917709598 |
| Marie | Lehmann | DE23311656553298147754 |

46 Zeilen ausgewählt

4.2. Outer Joins

Während bei den Inner Joins nur solche Zeilen im Ergebnis angezeigt werden, die der Join-Bedingung genügen, ist dieses Verhalten bei den Outer-Joins anders, da auch Datensätze sichtbar werden, die der Join-Bedingung nicht entsprechen.



Wie bereits erwähnt, können Outer-Joins nicht mehr, mit Hilfe der Theta-Style-Syntax, dargestellt werden!

4.2.1. Left- und Right-Outer-Join

Bei Left- bzw. Right-Outer-Joins wird eine der beiden teilnehmenden Tabellen vollständig angezeigt. Die Schlüsselworte **LEFT** und **RIGHT** geben dabei an, welche der beiden Seiten komplett angezeigt werden soll.

Der Left-Outer-Join

Beim Left-Outer-Join wird die Tabelle, die auf der linken Seite der Join-Klausel steht vollständig angezeigt. Von der Tabelle auf der rechten Seite werden nur solche Datensätze angezeigt, die der Join-Bedingung genügen.

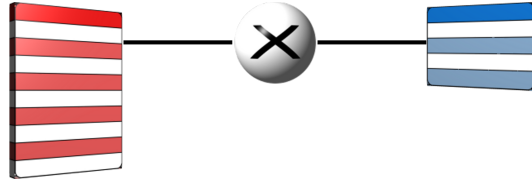


Abb. 4.1.:
Left-Outer-Join

Beispiel 4.12 zeigt einen Left-Outer-Join zwischen den beiden Tabellen MITARBEITER und BANKFILIALE. Die Auswirkungen des Left-Outer-Joins zeigen sich im Ergebnis nur in den letzten sieben Zeilen. Dort werden Mitarbeiter angezeigt, die in keiner Bankfiliale arbeiten und somit nicht der Join-Bedingung genügen.

Listing 4.12: Ein Left-Outer-Join in Oracle

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m LEFT OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID);
```

| VORNAME | NACHNAME | B.PLZ | B.ORT |
|-----------|------------|-------|--------------|
| Amelie | Krüger | 06449 | Aschersleben |
| Marie | Kipp | 06449 | Aschersleben |
| ... | ... | ... | ... |
| Emily | Meier | | |
| Peter | Müller | | |
| Tim | Sindermann | | |
| Sebastian | Schwarz | | |
| Finn | Seifert | | |
| Sarah | Werner | | |
| Max | Winter | | |

100 Zeilen ausgewählt



Da in **Beispiel 4.12** keine Sortierung vorgegeben wurde zeigt Oracle die Zeilen mit den NULL-Werten, in den Spalten PLZ und ORT, automatisch ganz zuletzt an! Dieses Verhalten kann mit dem **NULLS FIRST**-Schlüsselwort, in der **ORDER BY**-Klausel geändert werden.

Listing 4.13: NULL-Werte nach oben sortieren, NULLS FIRST

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m LEFT OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
ORDER BY PLZ NULLS FIRST;
```



| VORNAME | NACHNAME | B.PLZ | B.ORT |
|-----------|------------|-------|----------|
| Emily | Meier | | |
| Peter | Möller | | |
| Tim | Sindermann | | |
| Sebastian | Schwarz | | |
| Max | Winter | | |
| Sarah | Werner | | |
| Finn | Seifert | | |
| Sophie | Schwarz | 06406 | Bernburg |

100 Zeilen ausgewählt

Der MS SQL Server unterstützt die gleiche Syntax wie Oracle, kennt jedoch das **NULLS FIRST**-Schlüsselwort nicht, da er NULL-Werte bei Angabe einer **ORDER BY**-Klausel automatisch oben anzeigt.

Listing 4.14: Der Left-Outer-Join im MS SQL Server

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m LEFT OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
ORDER BY PLZ;
```



| VORNAME | NACHNAME | PLZ | ORT |
|-----------|------------|-------|----------|
| Emily | Meier | NULL | NULL |
| Peter | Möller | NULL | NULL |
| Tim | Sindermann | NULL | NULL |
| Sebastian | Schwarz | NULL | NULL |
| Max | Winter | NULL | NULL |
| Sarah | Werner | NULL | NULL |
| Finn | Seifert | NULL | NULL |
| Sophie | Schwarz | 06406 | Bernburg |

100 Zeilen ausgewählt

Der Right-Outer-Join

Der Right-Outer-Join ist das Komplement zum Left-Outer-Join. Er zeigt alle Datensätze der Tabelle an, die sich auf der rechten Seite befindet. Aus der Tabelle auf der linken Join-Seite werden wiederum nur jene Zeilen angezeigt, die der Join-Bedingung genügen.

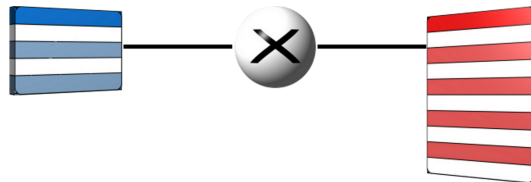


Abb. 4.2.:
Der
Right-Outer-Join

Listing 4.15: Ein Right-Outer-Join in Oracle

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m RIGHT OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
ORDER BY Nachname NULLS FIRST, PLZ;
```

| VORNAME | NACHNAME | B.PLZ | B.ORT |
|---------|----------|-------|------------|
| | | 06425 | Alsleben |
| Finn | Bauer | 06425 | Plötzkau |
| Leonie | Bauer | 39444 | Hecklingen |

94 Zeilen ausgewählt



Der erste Datensatz aus [Beispiel 4.15](#) zeigt, dass es eine Bankfiliale gibt, in der noch keine Mitarbeiter arbeiten. Das gleiche Beispiel lässt sich auch in MS SQL Server abarbeiten.

Listing 4.16: Der gleiche Right-Outer-Join in MS SQL Server

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m RIGHT OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
ORDER BY PLZ, Nachname;
```

| VORNAME | NACHNAME | PLZ | ORT |
|---------|----------|-------|------------|
| NULL | NULL | 06425 | Alsleben |
| Finn | Bauer | 06425 | Plötzkau |
| Leonie | Bauer | 39444 | Hecklingen |

94 Zeilen ausgewählt



4.2.2. Der Full Outer Join

Der Full-Outer-Join stellt die logische Ergänzung zu Left-Outer-Join und Right-Outer-Join dar. Er verküpft zwei Tabellen miteinander und zeigt auf beiden Seiten jeweils alle Tabellenzeilen an. Er ist in beiden DBMS, Oracle und MS SQL Server bekannt und syntaktisch gleich.

Listing 4.17: Ein Full-Outer-Join in Oracle

```
SELECT  Vorname, Nachname, b.PLZ, b.Ort
FROM    Mitarbeiter m FULL OUTER JOIN Bankfiliale b
        ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
ORDER BY PLZ NULLS FIRST, Nachname NULLS FIRST;
```



| VORNAME | NACHNAME | B.PLZ | B.ORT |
|-----------|------------|-------|------------|
| Emily | Meier | | |
| Peter | Möller | | |
| Sebastian | Schwarz | | |
| Finn | Seifert | | |
| ... | ... | ... | ... |
| Anne | Zimmermann | 06406 | Bernburg |
| Franz | Berger | 06408 | Ilberstedt |
| ... | ... | ... | ... |
| | | 06425 | Alsleben |
| Finn | Bauer | 06425 | Plötzkau |

101 Zeilen ausgewählt

4.3. Spezielle Joins

4.3.1. Der Self-Join

Ein Self-Join ist eine besondere Form des Inner Join. Er kommt immer dann zum Einsatz wenn der Primary Key einer Tabelle auf einen Foreign Key in der gleichen Tabelle zeigt, also bei rekursiven Beziehungstypen. Ein solcher rekursiver Beziehungstyp existiert in der Tabelle MITARBEITER. Er stellt das Vorgesetztenverhältnis zwischen den Mitarbeitern dar.

Wenn als Ergebnis einer Abfrage zu jedem Mitarbeiter sein Vorgesetzter angezeigt werden soll, so geht dies nur mittels Self-Join. In der folgenden Tabelle wird das Ergebnis eines solchen Self-Joins dargestellt. Es zeigt, wie zu jedem Vor- und Nachnamen eines Mitarbeiters, der Vor- und Nachname von dessen Vorgesetzten ermittelt werden kann.



| M# | MVORNAME | MNACHNAME | V# | VVORNAME | VNACHNAME |
|----|-----------|------------|----|-----------|-----------|
| 2 | Sarah | Werner | 1 | Max | Winter |
| 3 | Finn | Seifert | 1 | Max | Winter |
| 4 | Sebastian | Schwarz | 2 | Sarah | Werner |
| 5 | Tim | Sindermann | 2 | Sarah | Werner |
| 6 | Peter | Möller | 3 | Finn | Seifert |
| 7 | Emily | Meier | 3 | Finn | Seifert |
| 8 | Dirk | Peters | 4 | Sebastian | Schwarz |
| 9 | Louis | Winter | 4 | Sebastian | Schwarz |

Die Quelltabelle aufspalten

Grundsätzlich ist die Aufgabe einer Join-Operation zwei Tabellen zu einer Ergebnisrelation zu verknüpfen. Im besonderen Falle eines rekursiven Beziehungstyps existiert jedoch nur eine Tabelle. Wie kann der Join stattfinden? Die Antwort auf diese Frage liegt in der Nutzung von Tabellenaliasnamen.

Durch die Vergabe von Tabellenaliasnamen kann mehrfach auf ein und die selbe Tabelle, innerhalb eines SQL-Statements, zugegriffen werden!

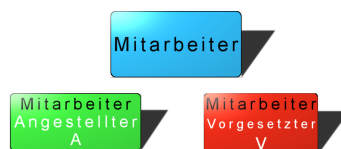


Abb. 4.3.:
Gesplante
Persönlichkeit -
Eine Tabelle,
zwei Aliase

Abbildung 4.3 zeigt, dass für die Tabelle MITARBEITER zwei Tabellenaliasnamen vergeben werden, nämlich „A“ für Angestellter und „V“ für Vorgesetzter. In SQL ausgedrückt bedeutet dies:

Listing 4.18: Eine Tabelle - zwei Aliasnamen

```

SELECT m.*
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
...
  
```

Die richtige Join-Bedingung finden

Die eigentliche Leistung, bei der Erstellung eines Self-Join, liegt darin, die korrekte Join-Bedingung zu finden. Fest steht, dass die beiden Spalten MITARBEITER_ID und VORGESETZTER_ID am Join beteiligt sein werden, aber es gibt insgesamt vier verschiedene Möglichkeiten, diese beiden Spalten zu kombinieren:

- **ON** (m.Mitarbeiter_ID = v.Mitarbeiter_ID)
- **ON** (m.Mitarbeiter_ID = v.Vorgesetzter_ID)
- **ON** (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
- **ON** (m.Vorgesetzter_ID = v.Vorgesetzter_ID)

Nun gilt es herauszufinden, welche die richtige Variante ist. Dies geht am Einfachsten, in dem man sich Beispieldaten schafft.



| MIT_M# | MIT_NACHNAME | MIT_V# | VOR_M# | VOR_NACHNAME | VOR_V# |
|--------|--------------|--------|--------|--------------|--------|
| 3 | Seifert | 1 | 1 | Winter | |
| 2 | Werner | 1 | 1 | Winter | |
| 5 | Sindermann | 2 | 2 | Werner | 1 |
| 4 | Schwarz | 2 | 2 | Werner | 1 |
| 7 | Meier | 3 | 3 | Seifert | 1 |
| 6 | Möller | 3 | 3 | Seifert | 1 |
| 12 | Weber | 4 | 4 | Schwarz | 2 |
| 11 | Schwarz | 4 | 4 | Schwarz | 2 |

Betrachtet man nun diese vier Join-Bedingungen, im Zusammenhang mit den Beispieldaten, lassen sich zwei davon direkt ausschließen.

- **ON** (m.Mitarbeiter_ID = v.Mitarbeiter_ID)
- **ON** (m.Vorgesetzter_ID = v.Vorgesetzter_ID)

Die Bedingung **ON** (m.Mitarbeiter_ID = v.Mitarbeiter_ID) verknüpft den Mitarbeiter aus der „Tabelle A“ mit dem gleichen Mitarbeiter aus der „Tabelle V“. Das bedeutet, dass alle Mitarbeiter mit sich selbst verknüpft werden, aber nicht mit Ihrem Vorgesetzten.

Die zweite Bedingung **ON** (`m.Vorgesetzter_ID = v.Vorgesetzter_ID`) erzeugt „logisches Chaos“. Der Mitarbeiter Seifert liefert hierzu ein gutes Beispiel:

| MIT_M# | MIT_NACHNAME | MIT_V# | VOR_M# | VOR_NACHNAME | VOR_V# |
|--------|--------------|--------|--------|--------------|--------|
| 3 | Seifert | 1 | 1 | Werner | 1 |
| 3 | Seifert | 1 | 1 | Seifert | 1 |



Es zeigt sich, dass der Mitarbeiter Seifert mit sich selbst und mit seinem Kollegen Werner verknüpft wird. Beide haben eines gemeinsam: Sie haben den gleichen Vorgesetzten. Somit verbleiben nur noch zwei Bedingungen:

- **ON** (`m.Mitarbeiter_ID = v.Vorgesetzter_ID`)
- **ON** (`m.Vorgesetzter_ID = v.Mitarbeiter_ID`)

Interessant sind nur die beiden verbleibenden Bedingungen, denn sie liefern beide ein sinnvolles Ergebnis. Verwendet man die erste von beiden, **ON** (`m.Mitarbeiter_ID = v.Vorgesetzter_ID`), zeigt sich folgendes Ergebnis:

| MIT_M# | MIT_NACHNAME | MIT_V# | VOR_M# | VOR_NACHNAME | VOR_V# |
|--------|--------------|--------|--------|--------------|--------|
| 3 | Seifert | 1 | 6 | Möller | 3 |
| 3 | Seifert | 1 | 7 | Meier | 3 |



Bei beiden Mitarbeitern, Möller und Meier, steht in der Spalte `VORGESETZTER_ID` der Wert 3. Daraus folgt, beide haben den Mitarbeiter Nummer drei als Vorgesetzten. Mitarbeiter Nummer drei ist Seifert. Mit Hilfe dieser Join-Bedingung werden zu jedem Vorgesetzten die Untergebenen angezeigt. Gesucht ist aber etwas anderes:

Zu jedem Angestellten soll der Vorgesetzte angezeigt werden. Die aktuelle Join-Bedingung zeigt die Informationen also nur aus der falschen Sichtweise an.

Was bleibt, ist nur noch die Bedingung **ON** (`m.Vorgesetzter_ID = v.Mitarbeiter_ID`). Diese zeigt das korrekte, gewünschte Ergebnis an.

| MIT_M# | MIT_NACHNAME | MIT_V# | VOR_M# | VOR_NACHNAME | VOR_V# |
|--------|--------------|--------|--------|--------------|--------|
| 3 | Seifert | 1 | 1 | Winter | |



Das komplette SQL-Statement zu dieser Problemstellung lautet:

Listing 4.19: Ein Self-Join

```

SELECT m.Mitarbeiter_ID AS MIT_M#, m.Vorname AS MIT_Vorname,
       m.Nachname AS MIT_Nachname, m.Vorgesetzter_ID AS MIT_V#,
       v.Mitarbeiter_ID AS VOR_M#, v.Vorname AS VOR_Vorname,
       v.Nachname AS VOR_Nachname, v.Vorgesetzter_ID AS VOR_V#
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);

```

4.3.2. Non-Equi-Joins

Kurzgesagt ist ein Non-Equi-Join ein Join, der nicht den Gleichheitsoperator (=) verwendet, sondern einen beliebigen anderen. Meist ist dies dann der **BETWEEN**-Operator. Da diese Art von Join in der Praxis jedoch äußerst selten ist, soll an dieser Stelle nicht weiter darauf eingegangen werden.

4.4. Mengenoperationen

In den vorangegangenen Abschnitten wurde gezeigt, wie zwei Tabellen durch eine Join-Operation miteinander verknüpft werden können. Dies bedingt immer, dass in beiden Tabellen eine Spalte vorhanden ist, die als Join-Attribut genutzt werden kann. Zusätzlich dazu, gibt es noch eine weitere Methode Datensätze unterschiedlicher Tabellen miteinander zu verknüpfen, die *SET-Operatoren*. Sie ermöglichen es, die Operationen der Mengenlehre in einer Datenbank durchzuführen. [Tabelle 4.1](#) zeigt die Operationen und die dazu gehörenden Operatoren:

Tabelle 4.1.: Die SET-Operatoren

| Mengenoperation | SET-Operator | Erläuterung |
|---------------------------|--------------|---|
| Vereinigung | UNION ALL | Zeigt die Vereinigungsmenge der beiden beteiligten Tabellen an. Duplikatzeilen bleiben erhalten. |
| Vollständige Vereinigung | UNION | Zeigt die Vereinigungsmenge der beiden beteiligten Tabellen, ohne Duplikatzeilen an. |
| Differenz (Oracle) | MINUS | Zeigt nur die Datensätze an, die in der linken der beiden Tabellen vorkommen und keine Entsprechung in der rechten Tabelle haben. |
| Differenz (MS SQL Server) | EXCEPT | Zeigt nur die Datensätze an, die in der linken der beiden Tabellen vorkommen und keine Entsprechung in der rechten Tabelle haben. |
| Durchschnitt | INTERSECT | Zeigt nur die Schnittmenge beider Tabellen an. |

4.4.1. Voraussetzungen zur Nutzung der SET-Operatoren

Um Mengenoperationen, auf zwei Relationen R und S, anwenden zu können, müssen beide miteinander kompatibel sein. Diese Form der Kompatibilität wird *Typenkompatibilität* oder auch *Vereinigungsverträglichkeit* genannt. Damit zwei Tabellen zueinander Typenkompatibel sind, müssen folgende Bedingungen gegeben sein:

- R und S müssen die gleiche Anzahl Attribute aufweisen.
- Der Wertebereich/Datentyp der Attribute von R und S muss identisch sein.

Das bedeutet zum einen, dass nur solche Abfragen mit Hilfe von SET-Operatoren kombiniert werden können, die die gleiche Anzahl Spalten in der **SELECT**-Klausel haben. Zum anderen müssen die verknüpften Spalten den gleichen Datentyp aufweisen.

4.4.2. Die SET-Operatoren

UNION und UNION ALL

Der **UNION ALL**-Operator verbindet die Ergebnisse zweier **SELECT**-Statements (Vereinigungsmenge). Sollte es Datensätze geben, die in beiden Abfragen ausgewählt werden (redundante Zeilen), werden diese angezeigt.

Der **UNION**-Operator verbindet, genau wie der **UNION ALL**-Operator, die Ergebnisse zweier SQL-Statements. Der Unterschied zwischen beiden liegt darin, dass der **UNION**-Operator redundante Zeilen ausschließt.

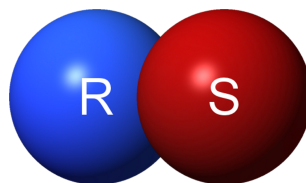


Abb. 4.4.:
Vereinigungs-
menge mit
UNION ALL

In einem einfachen Beispiel zum **UNION ALL**-Operator sollen alle Orte angezeigt werden, in denen Kunden oder Mitarbeiter leben. Um diese Aufgabe zu lösen, müssen zwei Abfragen ausgeführt werden.

Listing 4.20: Orte, an denen Kunden leben

```
SELECT Ort  
FROM   Eigenkunde;
```

Listing 4.21: Orte, an denen Mitarbeiter leben

```
SELECT Ort
FROM Mitarbeiter;
```

Zur Lösung der Aufgabe, müssen die Ergebnisse beider Abfragen kombiniert werden. Dies wird im ersten Anlauf durch den **UNION ALL**-Operator erledigt.

Listing 4.22: Orte, an denen Kunden oder Mitarbeiter leben

```
SELECT Ort
FROM Mitarbeiter
UNION ALL
SELECT Ort
FROM Eigenkunde;
```



ORT

Aschersleben
Bördeaue
Borne
Schönebeck
Alsleben
Hamburg
Borne
Egel
Schönebeck

500 Zeilen ausgewählt

An einigen Orten, wie z. B. Aschersleben, Borne, Egel und Schönebeck, ist zu erkennen, dass der **UNION ALL**-Operator keine redundanten Zeilen ausblendet. Soll das Ergebnis reduziert werden, so dass jeder Ort genau einmal angezeigt wird, kommt der **UNION**-Operator zum Einsatz.

Listing 4.23: Orte, an denen Kunden oder Mitarbeiter leben (reduziert)

```
SELECT Ort
FROM Mitarbeiter
UNION
SELECT Ort
FROM Eigenkunde;
```

ORT

Alsleben
 Aschersleben
 Barby
 Berlin
 Bernburg
 Borne
 Bördeaue
 Calbe

30 Zeilen ausgewählt



Durch die Anwendung des **UNION**-Operators, statt des **UNION ALL**-Operators verkürzt sich das Ergebnis von 500 Zeilen auf 30.

Der **UNION ALL**-Operator sollte nur dann zum Einsatz kommen, wenn dies zwingend notwendig ist!



In einem weiteren Beispiel soll gezeigt werden, wie Datensätze aus unterschiedlichen Tabellen im Ergebnis gekennzeichnet werden können. In einer Abfrage sollen alle Mitarbeiter und alle Kunden mit den Attributen VORNAME, NACHNAME, PLZ und ORT angezeigt werden. Für die Kunden muss in einer extra Spalte der Buchstabe „K“ und für alle Mitarbeiter der Buchstabe „M“ angezeigt werden.

Listing 4.24: Spalten mit konstanten Werten und UNION

```

SELECT 'M' AS Personentyp, Vorname, Nachname, PLZ, Ort
FROM   Mitarbeiter
UNION
SELECT 'K', Vorname, Nachname, PLZ, Ort
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);

```

| PERSONENTYP | VORNAME | NACHNAME | PLZ | ORT |
|-------------|-----------|----------|-------|--------------|
| K | Alexander | Huber | 22043 | Hamburg |
| K | Alexander | Lorenz | 06408 | Ilberstedt |
| K | Alina | Baumann | 07545 | Gera |
| ... | ... | ... | ... | ... |
| M | Alexander | Weber | 06449 | Aschersleben |
| M | Amelie | Krüger | 03042 | Cottbus |

500 Zeilen ausgewählt



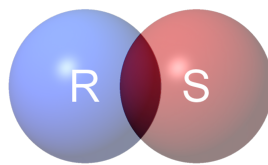
In der ersten Abfrage wird eine Spalte, mit Aliasnamen PERSONENTYP eingefügt. Sie bezieht ihren Wert nicht aus einer Tabelle, sondern sie enthält einfach nur den Buchstaben „K“ für Kunde. Die gleiche Spalte muss nun auch in der zweiten Abfrage eingeführt werden, da beide Abfragen, wie bereits erwähnt, die gleiche Anzahl Spalten, mit den gleichen Datentypen haben müssen. In der zweiten Abfrage kann jedoch der Aliasname entfallen, da dieser nur in der ersten Abfrage registriert/genutzt wird.

INTERSECT

Mit Hilfe des **INTERSECT**-Operators kann der Durchschnitt zweier Ergebnisse angezeigt werden. Das bedeutet, es werden nur die Zeilen angezeigt, die in beiden Relationen, R und S, gleichermaßen vorkommen.

Um die Wirkungsweise dieses Operators zu demonstrieren, wird [Beispiel 4.23](#) abgewandelt. Der **UNION**-Operator wird durch den **INTERSECT**-Operator ausgetauscht.

Abb. 4.5.:
Schnittmenge mit
INTERSECT



Listing 4.25: Orte, an denen sowohl Kunden als auch Mitarbeiter leben

```
SELECT Ort
FROM   Mitarbeiter
INTERSECT
SELECT Ort
FROM   Eigenkunde;
```



ORT

Alsleben
Aschersleben
Bernburg
Borne
Bördeau

25 Zeilen ausgewählt

Das Ergebnis dieser Abfrage liefert nur noch die Orte, an denen sowohl Kunden als auch Mitarbeiter leben.

MINUS / EXCEPT

Dieser Operator zeigt den Inhalt der linken Relation, ohne den Inhalt der Rechten an. Korrekt ausgedrückt bedeutet dies: $t1 \text{ MINUS } t2 = t1 \setminus t2$. Für SQL Server muss anstatt **MINUS** der Operator **EXCEPT** genutzt werden.

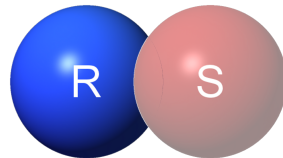


Abb. 4.6.:
Der MINUS /
EXCEPT
Operator

Für das kommende Beispiel werden die beiden Tabellen MITARBEITER und EIGENKUNDE vertauscht. Der **INTERSECT**-Operator wird gegen den **MINUS**-Operator ausgetauscht.

Listing 4.26: Orte, an denen nur Kunden, aber keine Mitarbeiter leben

```
SELECT Ort
FROM   Eigenkunde
MINUS
SELECT Ort
FROM   Mitarbeiter;
```

ORT

Barby
Berlin
Leipzig
Staßfurt
Wolmirsleben
5 Zeilen ausgewählt



Es gibt 30 verschiedene Orte, an denen Kunden leben und 25 verschiedene Orte, an denen Mitarbeiter leben. In 5 Orten leben nur Kunden, aber keine Mitarbeiter. Der **MINUS**-Operation bzw. der **EXCEPT**-Operator ist dabei behilflich, diese Orte herauszufiltern.

4.5. Übungen - Erweiterte Datenselektion

- Schreiben Sie eine Abfrage, die für jeden Mitarbeiter den Vornamen, den Nachnamen, die Bankfiliale_ID und den Ort anzeigt, an dem sich seine Filiale befindet.



| VORNAME | NACHNAME | BANKFILIALE_ID | ORT |
|----------|----------|----------------|--------------|
| Marie | Kipp | 1 | Aschersleben |
| Louis | Schmitz | 1 | Aschersleben |
| Johannes | Lehmann | 1 | Aschersleben |
| Dirk | Peters | 1 | Aschersleben |
| Amelie | Krüger | 1 | Aschersleben |
| Martin | Schacke | 2 | Aschersleben |

93 Zeilen ausgewählt

- Schreiben Sie eine Abfrage, welche die Mitarbeiternummer, den Nachnamen, das Gehalt und ein um 3,5 % erhöhtes Gehalt für alle Mitarbeiter anzeigt, die in einer Filiale in „Aschersleben“ arbeiten. Das erhöhte Gehalt soll als ganze Zahl und mit dem Spaltenalias „Neues Gehalt“ ausgegeben werden.



| MITARBEITER_ID | NACHNAME | GEHALT | Neues Gehalt |
|----------------|----------|--------|--------------|
| 8 | Peters | 12000 | 12420 |
| 9 | Winter | 12000 | 12420 |
| 28 | Lehmann | 2000 | 2070 |
| 29 | Schmitz | 2000 | 2070 |
| 30 | Kipp | 2000 | 2070 |
| 31 | Krüger | 2500 | 2588 |
| 32 | Beck | 1500 | 1553 |
| 33 | Schacke | 1000 | 1035 |
| 34 | Oswald | 1500 | 1553 |
| 35 | Wolf | 1000 | 1035 |

10 Zeilen ausgewählt

- Erstellen Sie eine Abfrage, die zu jedem Eigenkunden, der ein Depot besitzt, seinen Vor- und Nachnamen, die Strasse mit der Hausnummer, sowie PLZ und Ort anzeigt.



| VORNAME | NACHNAME | STRASSE | PLZ | ORT |
|-----------|----------|-------------------|-------|----------|
| Sophie | Junge | Plutoweg 3 | 39435 | Bördeau |
| Hanna | Beck | Beimsstraße 9 | 39439 | Güsten |
| Sebastian | Peters | Steinigstraße 3 | 39240 | Staßfurt |
| Tina | Berger | Bundschuhstraße 1 | 04177 | Leipzig |

239 Zeilen ausgewählt

4. Schreiben Sie eine Abfrage, die für alle Eigenkunden deren Vor- und Nachnamen anzeigt, sowie den Vor- und den Nachnamen ihres persönlichen Finanzberaters (Tabelle EIGENKUNDEMITARBEITER). Sortieren Sie die Abfrage nach den Nachnamen der Finanzberater.

| Vorname Kunde | Nachname Kunde | Vorname Berater | Nachname Berater |
|---------------|----------------|-----------------|------------------|
| Amelie | Fuchs | Leonie | Bauer |
| Sarah | Becker | Leonie | Bauer |
| Pia | Zimmermann | Leonie | Bauer |
| Hanna | Schreiber | Leonie | Bauer |
| Frank | Zimmermann | Leonie | Bauer |
| Chris | Wagner | Leonie | Bauer |
| Petra | Berger | Leonie | Bauer |
| Maximilian | Junge | Leonie | Bauer |

384 Zeilen ausgewählt



5. Schreiben Sie eine Abfrage, die für alle Eigenkunden, die keinen Berater haben (die nicht in der Tabelle EIGENKUNDEMITARBEITER enthalten sind), den Vor- und den Nachnamen anzeigt.

| VORNAME | NACHNAME |
|-----------|------------|
| Sebastian | Schröder |
| Udo | Schumacher |
| Mia | Huber |
| Simon | Witte |
| Max | Bunzel |
| Finn | Fischer |
| Lara | Meierhöfer |
| Jannis | Meier |

16 Zeilen ausgewählt



6. Schreiben Sie eine Abfrage, die zu jedem Mitarbeiter (Vorname, Nachname) den Vor- und den Nachnamen seines Vorgesetzten anzeigt.

| VORNAME_M | NACHNAME_M | VORNAME_V | NACHNAME_V |
|-----------|------------|-----------|------------|
| Finn | Seifert | Max | Winter |
| Sarah | Werner | Max | Winter |
| Tim | Sindermann | Sarah | Werner |
| Sebastian | Schwarz | Sarah | Werner |
| Emily | Meier | Finn | Seifert |
| Peter | Möller | Finn | Seifert |

99 Zeilen ausgewählt



7. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass alle Mitarbeiter, einschließlich des Mitarbeiters „Winter“, der keinen Vorgesetzten hat, angezeigt werden. Sortieren Sie das Ergebnis aufsteigend nach der Vorgesetzten_ID. Der Mitarbeiter „Winter“ soll ganz oben auf der Liste stehen.



| VORNAME | NACHNAME | VORNAME | NACHNAME |
|-----------|------------|---------|----------|
| Max | Winter | | |
| Finn | Seifert | Max | Winter |
| Sarah | Werner | Max | Winter |
| Tim | Sindermann | Sarah | Werner |
| Sebastian | Schwarz | Sarah | Werner |
| Emily | Meier | Finn | Seifert |

100 Zeilen ausgewählt

8. Erstellen Sie eine Abfrage, die ermittelt, ob es Mitarbeiter gibt, die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten).



| VORNAME | NACHNAME |
|---------|----------|
| Finn | Bauer |
| Stefan | Beck |
| Lina | Becker |
| Emma | Berger |
| Udo | Bosse |
| Georg | Dühning |
| Tom | Fischer |

60 Zeilen ausgewählt

9. Schreiben Sie eine Abfrage, die für alle Mitarbeiter, die höchstens 3 Jahre älter, aber keinesfalls jünger sind als ihr Vorgesetzter, den Vornamen, den Nachnamen, das Geburtsdatum und das Geburtsdatum des Vorgesetzten anzeigt.



| VORNAME | NACHNAME | GEBURTSDATUM | Geburtstag Chef |
|---------|----------|--------------|-----------------|
| Finn | Seifert | 17.10.85 | 31.08.88 |
| Jessica | Weber | 10.06.92 | 27.06.92 |
| Dirk | Peters | 16.09.91 | 27.06.92 |
| Chris | Lang | 08.10.86 | 30.01.89 |
| Marie | Kipp | 27.09.90 | 16.09.91 |

20 Zeilen ausgewählt

10. Schreiben Sie eine Abfrage, die für alle Mitarbeiter, die am gleichen Ort arbeiten, an dem sie auch wohnen, deren Vorname, Nachname den Wohnort und den Arbeitsort anzeigt. Beschriften Sie die Spalten, wie es in der Lösung zu sehen ist. Sortieren Sie die Abfragen in absteigender Reihenfolge nach dem Wohnort.

| VORNAME | NACHNAME | Wohnort | Arbeitsort |
|---------|------------|----------|------------|
| Emily | Günther | Plötzkau | Plötzkau |
| Jannis | Friedrich | Güsten | Güsten |
| Tim | Zimmermann | Egeln | Egeln |

3 Zeilen ausgewählt



11. Erstellen Sie eine Abfrage, die ermittelt, ob es Mitarbeiter gibt (Vorname und Nachname), die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten) und Filialleiter.

| VORNAME | NACHNAME |
|-----------|------------|
| Amelie | Krüger |
| Anna | Schneider |
| Chris | Simon |
| Christian | Haas |
| Elias | Sindermann |
| Emilia | Köhler |
| Emma | Krüger |

40 Zeilen ausgewählt



12. Erstellen Sie eine Abfrage, die alle Eigenkunden anzeigt, die nur Girokonten aber keine anderen Konten besitzen.

| VORNAME | NACHNAME |
|---------|----------|
| Amelie | Becker |
| Amelie | Richter |
| Chris | Walther |
| Emilia | Keller |
| Georg | Keller |
| Johanna | Schäfer |

21 Zeilen ausgewählt



13. Erstellen Sie mit Hilfe einer Abfrage eine Liste, die den Vor- und den Nachnamen aller Kunden enthält, die sowohl ein Sparbuch, als auch ein Depot besitzen. Ob die Kunden ein Girokonto haben oder nicht ist irrelevant.



| VORNAME | NACHNAME |
|------------------------------|------------|
| Alexander | Lorenz |
| Alina | Baumann |
| Alina | Huber |
| Alina | Peters |
| Alina | Schumacher |
| Alina | Schütz |
| Amelie | Fuchs |
| Amelie | Günther |
| 176 Zeilen ausgewählt | |

14. Schreiben Sie eine Abfrage, die eine Liste aller Eigenkunden ausgibt, die ein Girokonto und ein Sparbuch besitzen, aber kein Depot.



| VORNAME | NACHNAME |
|------------------------------|------------|
| Alina | Braun |
| Andy | Klingner |
| Anna | Schubert |
| Anna | Sindermann |
| Anna | Wagner |
| Bea | Witte |
| Ben | Lehmann |
| Chris | Beck |
| Chris | Weber |
| 134 Zeilen ausgewählt | |

4.6. Lösungen - Erweiterte Datenselektion

1. Schreiben Sie eine Abfrage, die für jeden Mitarbeiter den Vornamen, den Nachnamen, die Bankfiliale_ID und den Ort anzeigt, an dem sich seine Filiale befindet.

```
SELECT m.Vorname, m.Nachname, m.Bankfiliale_ID, b.Ort
FROM   Mitarbeiter m INNER JOIN Bankfiliale b
      ON (b.Bankfiliale_ID = m.Bankfiliale_ID);
```



2. Schreiben Sie eine Abfrage, welche die Mitarbeiternummer, den Nachnamen, das Gehalt und ein um 3,5 % erhöhtes Gehalt für alle Mitarbeiter anzeigt, die in einer Filiale in „Aschersleben“ arbeiten. Das erhöhte Gehalt soll als ganze Zahl und mit dem Spaltenalias „Neues Gehalt“ ausgegeben werden.

```
SELECT m.Mitarbeiter_ID, m.Nachname, m.Gehalt,
      ROUND(m.Gehalt * 1.035, 0) AS "Neues Gehalt"
FROM   Mitarbeiter m INNER JOIN Bankfiliale b
      ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE  LOWER(b.Ort) LIKE 'aschersleben';
```



3. Erstellen Sie eine Abfrage, die zu jedem Eigenkunden, der ein Depot besitzt, seinen Vor- und Nachnamen, die Strasse mit der Hausnummer, sowie PLZ und Ort anzeigt.

```
SELECT k.Vorname, k.Nachname, ek.Strasse || ' ' || ek.Hausnummer AS Strasse,
      ek.PLZ, ek.Ort
FROM   Kunde k INNER JOIN Eigenkunde ek
      ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk
      ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Depot d
      ON (ekk.Konto_ID = d.Konto_ID);
```





```
SELECT k.Vorname, k.Nachname, ek.Strasse + ' ' + ek.Hausnummer AS Strasse,
       ek.PLZ, ek.Ort
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID)
       INNER JOIN EigenkundeKonto ekk
       ON (ek.Kunden_ID = ekk.Kunden_ID)
       INNER JOIN Depot d
       ON (ekk.Konto_ID = d.Konto_ID);
```

4. Schreiben Sie eine Abfrage, die für alle Eigenkunden deren Vor- und Nachnamen anzeigt, sowie den Vor- und den Nachnamen ihres persönlichen Finanzberaters (Tabelle EIGENKUNDEMITARBEITER). Sortieren Sie die Abfrage nach den Nachnamen der Finanzberater.



```
SELECT k.Vorname AS "Vorname Kunde", k.Nachname AS "Nachname Kunde",
       m.Vorname AS "Vorname Berater", m.Nachname AS "Nachname Berater"
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID)
       INNER JOIN EigenkundeMitarbeiter ekm
       ON (ek.Kunden_ID = ekm.Kunden_ID)
       INNER JOIN Mitarbeiter m
       ON (ekm.Mitarbeiter_ID = m.Mitarbeiter_ID)
ORDER BY m.Nachname;
```

5. Schreiben Sie eine Abfrage, die für alle Eigenkunden, die keinen Berater haben (die nicht in der Tabelle EIGENKUNDEMITARBEITER enthalten sind), den Vor- und den Nachnamen anzeigt.



```
SELECT k.Vorname, k.Nachname
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID)
       LEFT OUTER JOIN EigenkundeMitarbeiter ekm
       ON (ek.Kunden_ID = ekm.Kunden_ID)
WHERE  ekm.Kunden_ID IS NULL;
```

6. Schreiben Sie eine Abfrage, die zu jedem Mitarbeiter (Vorname, Nachname) den Vor- und den Nachnamen seines Vorgesetzten anzeigt.



```
SELECT m.Vorname AS Vorname_M, m.Nachname AS Nachname_M,
       v.Vorname AS Vorname_V, v.Nachname AS Nachname_V
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);
```

7. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass alle Mitarbeiter, einschließlich des Mitarbeiters „Winter“, der keinen Vorgesetzten hat, angezeigt werden. Sortieren Sie das Ergebnis aufsteigend nach der Vorgesetzten_ID. Der Mitarbeiter „Winter“ soll ganz oben auf der Liste stehen.



```
SELECT m.Vorname AS Vorname_M, m.Nachname AS Nachname_M,
       v.Vorname AS Vorname_V, v.Nachname AS Nachname_V
FROM   Mitarbeiter m LEFT OUTER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
ORDER BY m.Vorgesetzter_ID NULLS FIRST;
```



```
SELECT m.Vorname AS Vorname_M, m.Nachname AS Nachname_M,
       v.Vorname AS Vorname_V, v.Nachname AS Nachname_V
FROM   Mitarbeiter m LEFT OUTER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
ORDER BY m.Vorgesetzter_ID;
```

8. Erstellen Sie eine Abfrage, die ermittelt, ob es Mitarbeiter gibt, die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten).



```
SELECT m.Vorname, m.Nachname
FROM   Mitarbeiter m LEFT OUTER JOIN EigenkundeMitarbeiter ekm
       ON (m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
WHERE  ekm.Mitarbeiter_ID IS NULL
       AND m.Bankfiliale_ID IS NOT NULL;
```

9. Schreiben Sie eine Abfrage, die für alle Mitarbeiter, die höchstens 3 Jahre älter, aber keinesfalls jünger sind als ihr Vorgesetzter, den Vornamen, den Nachnamen, das Geburtsdatum und das Geburtsdatum des Vorgesetzten anzeigt.



```
SELECT m.Vorname, m.Nachname, m.Geburtsdatum,
       v.Geburtsdatum AS "Geburtstag Chef"
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
WHERE  m.Geburtsdatum BETWEEN v.Geburtsdatum - INTERVAL '3' YEAR AND
       v.Geburtsdatum;
```



```
SELECT m.Vorname, m.Nachname, m.Geburtsdatum,
       v.Geburtsdatum AS "Geburtstag Chef"
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
WHERE  m.Geburtsdatum BETWEEN DATEADD(YEAR, -3, v.Geburtsdatum) AND
       v.Geburtsdatum;
```

10. Schreiben Sie eine Abfrage, die für alle Mitarbeiter, die am gleichen Ort arbeiten, an dem sie auch wohnen, deren Vorname, Nachname den Wohnort und den Arbeitsort anzeigt. Beschriften Sie die Spalten, wie es in der Lösung zu sehen ist. Sortieren Sie die Abfragen in absteigender Reihenfolge nach dem Wohnort.



```
SELECT m.Vorname, m.Nachname, m.Ort AS "Wohnort", b.Ort AS "Arbeitsort"
FROM   Mitarbeiter m INNER JOIN Bankfiliale b
       ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE  m.Ort = b.Ort
ORDER BY m.Ort DESC;
```

11. Erstellen Sie eine Abfrage, die ermittelt, ob es Mitarbeiter gibt (Vorname und Nachname), die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten) und Filialleiter.

```
SELECT m.Vorname, m.Nachname
FROM   Mitarbeiter m LEFT OUTER JOIN EigenkundeMitarbeiter ekm
       ON (m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
WHERE  ekm.Mitarbeiter_ID IS NULL
MINUS
SELECT DISTINCT v.Vorname, v.Nachname
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);
```



```
SELECT m.Vorname, m.Nachname
FROM   Mitarbeiter m LEFT OUTER JOIN EigenkundeMitarbeiter ekm
       ON (m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
WHERE  ekm.Mitarbeiter_ID IS NULL
EXCEPT
SELECT DISTINCT v.Vorname, v.Nachname
FROM   Mitarbeiter m INNER JOIN Mitarbeiter v
       ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);
```



12. Erstellen Sie eine Abfrage, die alle Eigenkunden anzeigt, die nur Girokonten aber keine anderen Konten besitzen.

```
SELECT k.Vorname, k.Nachname
FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
       INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
       INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
MINUS
SELECT k.Vorname, k.Nachname
FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
       INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
       INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
MINUS
SELECT k.Vorname, k.Nachname
FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
       INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
       INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);
```





```

SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
EXCEPT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
EXCEPT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);

```

13. Erstellen Sie mit Hilfe einer Abfrage eine Liste, die den Vor- und den Nachnamen aller Kunden enthält, die sowohl ein Sparbuch, als auch ein Depot besitzen. Ob die Kunden ein Girokonto haben oder nicht ist irrelevant.



```

SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
INTERSECT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);

```


14. Schreiben Sie eine Abfrage, die eine Liste aller Eigenkunden ausgibt, die ein Girokonto und ein Sparbuch besitzen, aber kein Depot.



```

SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
INTERSECT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
MINUS
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);

```



```

SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
INTERSECT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
EXCEPT
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
      INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);

```


5. Gruppenfunktionen

Inhaltsangabe

| | | |
|------------|--|-------------|
| 5.1 | Die GROUP BY-Klausel | 5-2 |
| 5.2 | Die Aggregatfunktionen | 5-3 |
| 5.2.1 | Die Funktion COUNT | 5-4 |
| 5.2.2 | Die Funktion SUM | 5-5 |
| 5.2.3 | Die Funktion AVG | 5-5 |
| 5.2.4 | Die Funktionen MIN und MAX | 5-7 |
| 5.2.5 | Gruppierungen mit mehreren Ebenen | 5-8 |
| 5.3 | Gruppierte Abfragen filtern | 5-8 |
| 5.3.1 | Die WHERE-Klausel | 5-8 |
| 5.3.2 | Die HAVING-Klausel | 5-9 |
| 5.4 | Die Abarbeitungsreihenfolge des SELECT-Statements | 5-11 |
| 5.5 | Übungen - Gruppenfunktionen | 5-12 |
| 5.6 | Lösungen - Gruppenfunktionen | 5-15 |

In vielen Fällen ist es notwendig, die aus einer Abfrage resultierenden Datensätze nicht einzeln anzuzeigen, sondern sie nach bestimmten Kriterien zusammenzufassen. Dieser Vorgang wird als „gruppieren“ bezeichnet und mittels der **GROUP BY**-Klausel umgesetzt. Sie wird zwischen die beiden Klauseln **WHERE** und **ORDER BY** eingefügt.

5.1. Die GROUP BY-Klausel

In einem ersten Beispiel werden aus der Tabelle MITARBEITER die IDs aller Vorgesetzten angezeigt, so dass eine „Liste der Vorgesetzten“ entsteht.

Listing 5.1: Die „Liste der Vorgesetzten“

```
SELECT  Vorgesetzter_ID
FROM    Mitarbeiter
GROUP BY Vorgesetzter_ID
ORDER BY 1;
```

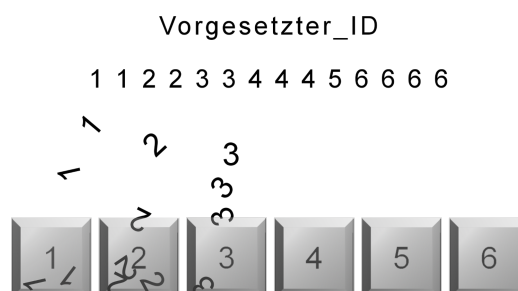


VORGESETZTER_ID

1
2
3
...
27
NULL

28 Zeilen ausgewählt

Abb. 5.1.:
Gruppieren von
Datensätzen



Mit Hilfe der **GROUP BY**-Klausel werden die einzelnen IDs in Gruppen eingeteilt. Anschließend wird für jede Gruppe die ID genau einmal angezeigt.

Statt einer einfachen Gruppierung, wie sie in [Beispiel 5.1](#) erzeugt wurde, kann auch eine mehrfache Gruppierung erzeugt werden. Diese sind aber meist nur in Verbindung mit Aggregatfunktionen, die im folgenden Abschnitt behandelt werden, sinnvoll.

5.2. Die Aggregatfunktionen

Im Gegensatz zu den Single Row Functions, die sich immer nur auf eine Zeile auswirken und deshalb pro Zeile einmal ausgeführt werden müssen, beziehen sich Aggregatfunktionen immer auf eine Gruppierung. Dies können alle Werte einer Spalte oder mehrere getrennte Bereiche sein. Sinn und Zweck dieser Funktionen ist es, den Anwender dabei zu unterstützen, vordefinierte Berechnungen durchzuführen. [Tabelle 5.1](#) zeigt einen Überblick, über die wichtigsten Aggregatfunktionen.

Tabelle 5.1.: Aggregatfunktionen

| Aggregatfunktion | Bedeutung | Wertebereich |
|------------------|---|--------------|
| AVG | Berechnet für den übergebenen Bereich den Durchschnitt aller Werte. NULL-Werte werden bei der Berechnung nicht berücksichtigt. | Numerisch |
| COUNT | Zählt die zur Gruppierung gehörenden Datensätze. Als Funktionsargument kann ein beliebiger Ausdruck übergeben werden. Wird ein Spaltenbezeichner verwendet, zählt die Funktion die Anzahl der Werte in dieser Spalte. NULL-Werte werden von dieser Funktion nicht berücksichtigt. | Universell |
| MAX | Liefert den größten Wert eines Bereiches zurück. | Universell |
| MIN | Liefert den kleinsten Wert eines Bereiches zurück. | Universell |
| SUM | Berechnet die Summe, für den übergebenen Bereich. NULL-Werte werden bei der Berechnung nicht berücksichtigt. | Numerisch |

[Beispiel 5.2](#) zeigt die Anwendung der Summen-Funktion **SUM**, um die Gehälter aller Mitarbeiter pro Filiale zu ermitteln.

Listing 5.2: Fehler in der Gruppierung

```
SELECT Bankfiliale_ID , SUM(Gehalt)
FROM   Mitarbeiter;
```

Auch wenn auf den ersten Blick an dieser Abfrage nichts falsches zu bemerken ist, antwortet das DBMS mit einer Fehlermeldung, was daran liegt, dass die Funktion **SUM** automatisch eine Gruppierung bildet, in die die Spalte **BANKFILIALE_ID** nicht mit einbezogen wird.

Listing 5.3: Die Fehlermeldung in Oracle

```
Fehler beim Start in Zeile 1 in Befehl:
SELECT Bankfiliale_ID , SUM(Gehalt)
FROM   Mitarbeiter
Fehler bei Befehlszeile:1 Spalte:7
Fehlerbericht:
SQL-Fehler: ORA-00937: keine Gruppenfunktion fuer Einzelgruppe
00937. 00000 - "not a single-group group function"
```

Listing 5.4: Die Fehlermeldung in SQL Server

Meldung 8120, Ebene 16, Status 1, Zeile 1
Die 'Bank.dbo.Bankfiliale_ID'-Spalte
ist in der Auswahlliste ungültig, da sie nicht in einer
Aggregatfunktion und nicht in der **GROUP BY**-Klausel enthalten ist.



Sobald eine Gruppenfunktion zum Einsatz kommt, müssen alle in der **SELECT**-Klausel gelisteten Attribute gruppiert werden. Dies kann durch die Anwendung weiterer Gruppenfunktionen oder durch die **GROUP BY**-Klausel geschehen.

Um diesen Fehler zu beheben, muss das Statement aus [Beispiel 5.2](#) um eine **GROUP BY**-Klausel erweitert werden.

Listing 5.5: Der korrekte Einsatz der **SUM**-Funktion

```
SELECT  Bankfiliale_ID, SUM(Gehalt)
FROM    Mitarbeiter
GROUP BY Bankfiliale_ID;
```



| BANKFILIALE_ID | SUM(GEHALT) |
|----------------|-------------|
| | 308000 |
| 1 | 20500 |
| 6 | 21000 |
| 11 | 23000 |
| 13 | 20000 |
| 2 | 17000 |
| 14 | 19500 |
| 20 | 19500 |

21 Zeilen ausgewählt

5.2.1. Die Funktion COUNT

Wie in [Tabelle 5.1](#) beschrieben, wird **COUNT** zum Zählen von Werten genutzt. In [Beispiel 5.6](#) wird ermittelt, wie viele Mitarbeiter in der Tabelle MITARBEITER gespeichert sind.

Listing 5.6: Das Zählen von Datensätzen

```
SELECT COUNT(*) AS "Mitarbeiter"
FROM    Mitarbeiter;
```

```

MITARBEITER
-----
                100
1 Zeile ausgewählt

```



Der Stern * kann in der **COUNT**-Funktion als „Joker“ genutzt werden. **COUNT(*)** zählt alle Zeilen einer Tabelle und hat somit den gleichen Effekt, wie das Zählen der Einträge der Primärschlüsselspalte einer Tabelle.

Ein weiteres Beispiel zeigt, dass die **COUNT**-Funktion NULL-Werte nicht berücksichtigt.

Listing 5.7: NULL-Werte werden nicht gezählt!

```

SELECT COUNT (Vorgesetzter_ID)
FROM    Mitarbeiter;

```

```

COUNT (VORGESETZTER_ID)
-----
                99
1 Zeile ausgewählt

```



Da der Mitarbeiter Winter (MITARBEITER_ID = 1) keinen Vorgesetzten hat, ist bei ihm ein NULL-Wert in der Spalte VORGESETZTER_ID, was dazu führt, dass **COUNT** nur 99 Werte zählt.

5.2.2. Die Funktion SUM

Mit Hilfe von **SUM** kann die Summe aller Werte eines Bereichs gebildet werden. Ein Beispiel zu dieser Funktion ist in [Beispiel 5.5](#) zu sehen. Im Gegensatz zur **COUNT**-Funktion, spielen NULL-Werte keine Rolle, in Bezug auf die **SUM**-Funktion. Ein NULL-Wert wird durch die **SUM**-Funktion einfach ignoriert und verfälscht das Ergebnis dadurch nicht.

5.2.3. Die Funktion AVG

Die Abkürzung „AVG“ steht für das englische Wort „average“ = Durchschnitt (arithmetisches Mittel). Die Funktion **AVG** berechnet den Durchschnitt der Werte einer Gruppierung.

In **Beispiel 5.8** wird die **AVG**-Funktion genutzt, um das Durchschnittsgehalt für jede Filiale zu berechnen.

Listing 5.8: Die AVG-Funktion

```
SELECT  Bankfiliale_ID, AVG(Gehalt)
FROM    Mitarbeiter
GROUP BY Bankfiliale_ID;
```



| BANKFILIALE_ID | AVG (GEHALT) |
|----------------|--------------|
| | 44000 |
| 1 | 4100 |
| 6 | 4200 |
| 11 | 4600 |
| 13 | 5000 |
| 2 | 3400 |
| 14 | 4875 |
| 20 | 4875 |
| 4 | 4100 |

21 Zeilen ausgewählt

Das nächste Beispiel erläutert den Zusammenhang zwischen **AVG** und **NULL**-Werten. Im Versuch soll die durchschnittliche Provision, die ein Mitarbeiter erhält, berechnet werden. Wichtig für diesen Versuch ist, dass nur ein Teil der Mitarbeiter eine Provision erhält.

Listing 5.9: AVG und NULL-Werte in Oracle

```
SELECT COUNT(Provision) AS Anzahl, ROUND(AVG(Provision), 2) AS "AVG",
       COUNT(NVL(Provision, 0)) AS "Anzahl NVL",
       AVG(NVL(Provision, 0)) AS "AVG NVL"
FROM    Mitarbeiter;
```



| ANZAHL | AVG | ANZAHL NVL | AVG NVL |
|--------|-------|------------|---------|
| 33 | 22,58 | 100 | 7,45 |

1 Zeile ausgewählt

Listing 5.10: AVG und NULL-Werte im MS SQL Server

```
SELECT COUNT(Provision) AS Anzahl, ROUND(AVG(Provision), 2) AS "AVG",
       COUNT(ISNULL(Provision, 0)) AS "Anzahl ISNULL",
       AVG(ISNULL(Provision, 0)) AS "AVG ISNULL"
FROM    Mitarbeiter;
```



| ANZAHL | AVG | ANZAHL NVL | AVG NVL |
|--------|-------|------------|---------|
| 33 | 22,58 | 100 | 7,45 |

1 Zeile ausgewählt

Je nach dem, ob NULL-Werte durch die **NVL**-Funktion/**ISNULL**-Funktion bereinigt werden oder nicht, wird ein unterschiedliches Ergebnis, durch die **AVG**-Funktion, errechnet.

5.2.4. Die Funktionen MIN und MAX

Die Funktionen **MIN** und **MAX** ermitteln den größten bzw. kleinsten Wert aus einer Menge und können auf nahezu jeden Datentyp angewendet werden. **Beispiel 5.11** zeigt die Anwendung von **MAX** auf Spalten verschiedener Datentypen.

Listing 5.11: Anwendung von MAX auf verschiedene Datentypen

```
SELECT MAX(Nachname), MAX(Geburtsdatum), MAX(PLZ), MAX(Provision)
FROM Mitarbeiter;
```

| MAX (NACHNAME) | MAX (GEBURTSDATUM) | MAX (PLZ) | MAX (PROVISION) |
|----------------|--------------------|-----------|-----------------|
| Zimmermann | 31.05.93 | 80995 | 30 |

1 Zeile ausgewählt



Zu beachten ist, dass die angezeigten Daten in keiner Beziehung zueinander stehen. Es handelt sich um die Maximalwerte aus den einzelnen Spalten. Der Angestellte Zimmermann hat keinesfalls das Geburtsdatum 31.05.1993 und er bekommt auch keine Provision.

Bezüglich NULL-Werte gibt es, sowohl in Oracle, als auch in MS SQL Server, keine Probleme mit **MIN** oder **MAX**, wie das folgende **Beispiel 5.12** zeigt.

Listing 5.12: Die MAX-Funktion und NULL-Werte (Oracle)

```
SELECT MAX(Provision), MAX(NVL(Provision,0))
FROM Mitarbeiter;
```

| MAX (PROVISION) | MAX (NVL (PROVISION, 0)) |
|-----------------|---------------------------|
| 30 | 30 |

1 Zeile ausgewählt



Das gleiche Beispiel kann in MS SQL Server, mit Hilfe der **ISNULL**-Funktion reproduziert werden.

Alle Eigenschaften, die für die **MAX**-Funktion gelten, gelten uneingeschränkt auch für die **MIN**-Funktion.



5.2.5. Gruppierungen mit mehreren Ebenen

Wie bereits angekündigt, ist es möglich, mit Hilfe der **GROUP BY**-Klausel, eine Abfrage mehrfach zu gruppieren. Eine Mehrfachgruppierung ist immer dann notwendig, wenn innerhalb einer Gruppe weitere Gruppen gebildet werden müssen. [Beispiel 5.13](#) listet alle Kunden auf, die nach dem 01.01.1995 geboren wurden, gruppiert nach Ort und Strasse.

Listing 5.13: Eine Gruppierung mit mehreren Ebenen

```
SELECT Ort, Strasse, COUNT(*) AS Anzahl
FROM Kunde k INNER JOIN Eigenkunde ek
      ON (k.Kunden_ID = ek.Kunden_ID)
WHERE Geburtsdatum > '01.01.1995'
GROUP BY Ort, Strasse
ORDER BY Ort;
```



| ORT | STRASSE | ANZAHL |
|-----------------------------|------------------|--------|
| Aschersleben | Am Markt | 1 |
| Bördeaue | Plutoweg | 1 |
| Bördeaue | Okerstraße | 1 |
| ... | ... | ... |
| Hecklingen | Turmstraße | 1 |
| Hecklingen | Pestalozzistraße | 1 |
| Hecklingen | Seestraße | 1 |
| ... | ... | ... |
| Steißfurt | Wielandstraße | 2 |
| 21 Zeilen ausgewählt | | |

5.3. Gruppierte Abfragen filtern

5.3.1. Die WHERE-Klausel

Das die **WHERE**-Klausel auch auf gruppierte Abfragen angewandt werden kann, ist bereits in [Beispiel 5.13](#) zu sehen. Wesentlich dabei ist, dass sie vor dem Gruppieren abgearbeitet wird, d. h. es wird die Menge der Zeilen eingeschränkt, die noch gruppiert werden muss. Hierzu ein Beispiel: Mit Hilfe einer Abfrage soll ermittelt werden, wer der jüngste Kunde ist.

Listing 5.14: Wer ist der jüngste Kunde

```
SELECT MAX(Geburtsdatum)
FROM Eigenkunde;
```

MAX (GEBURTSDATUM)

07.04.97

1 Zeile ausgewählt



Im Folgenden wird [Beispiel 5.14](#) durch eine **WHERE**-Klausel eingeschränkt.

Listing 5.15: Der jüngste Kunde aus Alsleben

```
SELECT MAX(Geburtsdatum)
FROM   Eigenkunde
WHERE  Ort LIKE 'Alsleben';
```

MAX (GEBURTSDATUM)

21.05.93

1 Zeile ausgewählt



Die angefügte **WHERE**-Klausel sorgt dafür, dass die Gruppierung, die durch die **MAX**-Funktion entsteht, nur auf die Kunden angewandt wird, die in Alsleben wohnen.

Die **WHERE**-Klausel wird immer vor dem Gruppieren abgearbeitet!



5.3.2. Die HAVING-Klausel

Gerade eben wurde gezeigt, dass mit Hilfe der **WHERE**-Klausel eine Selektion vor der Gruppierung der Datensätze erreicht werden kann. Was aber ist, wenn eine Auswahl auf gruppierten Datensätzen erfolgen soll? Im folgenden Versuch soll für jede Bankfiliale das niedrigste Gehalt aufgelistet werden, aber nur dann, wenn es größer als 1.500 EUR ist.

Listing 5.16: Ein Versuch... mit Oracle

```
SELECT  Bankfiliale_ID, MIN(Gehalt)
FROM    Mitarbeiter
WHERE   MIN(Gehalt) > 1500
GROUP BY Bankfiliale_ID;

ORA-00934: group function is not allowed here
00934. 00000 - "group function is not allowed here"
```

Listing 5.17: Der gleiche Versuch... mit MS SQL Server

```
SELECT  Bankfiliale_ID, MIN(Gehalt)
FROM    Mitarbeiter
WHERE   MIN(Gehalt) > 1500
GROUP BY Bankfiliale_ID;
```

Meldung 147, Ebene 15, Status 1, Zeile 3

An aggregate may not appear in the **WHERE** clause unless it is in a subquery contained in a **HAVING** clause or a select list, and the column being aggregated is an outer reference.

Beispiel 5.16 und Beispiel 5.17 zeigen, dass die Verarbeitung einer Aggregatfunktion in der **WHERE**-Klausel nicht möglich ist. Dies liegt daran, dass, wie bereits erwähnt, die **WHERE**-Klausel schon vor der Gruppierungsphase abgearbeitet wird.

Um das gewünschte Ziel erreichen zu können, muss eine neue Klausel, die **HAVING**-Klausel, eingeführt werden. Sie ermöglicht es, Selektionen auf gruppierten Zeilen durchzuführen. Beispiel 5.16 muss korrekt lauten:

Listing 5.18: Die HAVING-Klausel

```
SELECT  Bankfiliale_ID, MIN(Gehalt)
FROM    Mitarbeiter
GROUP BY Bankfiliale_ID
HAVING  MIN(Gehalt) > 1500;
```



| BANKFILIALE_ID | MIN(GEHALT) |
|---------------------|-------------|
| | 30000 |
| 1 | 2000 |
| 6 | 2000 |
| 11 | 2000 |
| 7 | 2000 |
| 18 | 2500 |
| 15 | 2000 |
| 16 | 3000 |
| 19 | 2000 |
| 9 Zeilen ausgewählt | |

Die **HAVING**-Klausel eliminiert, nach dem Gruppieren, alle Datensätze, auf die die Bedingung zutrifft: **MIN(Gehalt) <= 1500**.



Die **HAVING**-Klausel wird auf gruppierte Zeilen angewandt und kann deshalb nur in Verbindung mit der **GROUP BY**-Klausel stehen.

5.4. Die Abarbeitungsreihenfolge des SELECT-Statements

Nachdem nun in den vorangegangenen Kapiteln alle standardisierten Klauseln des **SELECT**-Kommandos behandelt wurden, stellt sich noch immer die Frage: „In welcher Reihenfolge werden die Klauseln des **SELECT**-Kommandos abgearbeitet?“. Die korrekte Antwort lautet:

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

Zuerst wird mit Hilfe der **FROM**-Klausel ermittelt, auf welche Tabellen sich das **SELECT**-Statement bezieht. Im zweiten Schritt filtert die **WHERE**-Klausel alle Zeilen aus den Quelltabellen, die für das Statement nicht mehr relevant sind. Die beiden Klauseln **GROUP BY** und **HAVING** sorgen für Gruppierungen und das Filtern von gruppierten Zeilen. Zu guter letzt werden die **SELECT**- und die **ORDER BY**-Klausel abgearbeitet, so dass das Ergebnis auf dem Bildschirm ausgegeben werden kann.

5.5. Übungen - Gruppenfunktionen

- Schreiben Sie eine Abfrage, die das höchste und das niedrigste Gehalt, das Durchschnittsgehalt und die Summe aller Gehälter ausgibt. Beschriften Sie die Spalten, wie es in der Lösung zu sehen ist.



| Maximum | Minimum | Mittelwert | Summe |
|---------|---------|------------|--------|
| 88000 | 1000 | 7255 | 725500 |

1 Zeile ausgewählt

- Verändern Sie die Abfrage aus der vorangegangenen Abfrage so, dass die Informationen für jede einzelne Bankfiliale angezeigt werden. Sortieren sie das Ergebnis nach den IDs der Bankfilialen.



| BANKFILIALE_ID | Maximum | Minimum | Mittelwert | Summe |
|----------------|---------|---------|------------|-------|
| 1 | 12000 | 2000 | 4100 | 20500 |
| 2 | 12000 | 1000 | 3400 | 17000 |
| 3 | 12000 | 1000 | 3900 | 19500 |
| 4 | 12000 | 1500 | 4100 | 20500 |
| 5 | 12000 | 1000 | 4200 | 21000 |
| 6 | 12000 | 2000 | 4200 | 21000 |

20 Zeilen ausgewählt

- Schreiben Sie eine Abfrage, die die Anzahl der Mitarbeiter pro Bankfiliale ausgibt. Beschriften Sie die Spalten so, wie es in der Lösung zu sehen ist und sortieren Sie das Ergebnis nach den IDs der Filialen.



| BANKFILIALE_ID | Anzahl |
|----------------|--------|
| 1 | 5 |
| 2 | 5 |
| 3 | 5 |
| 4 | 5 |
| 5 | 5 |
| 6 | 5 |

20 Zeilen ausgewählt

4. Schreiben Sie eine Abfrage, die für jeden Ort einzeln, die Anzahl der Eigenkunden zählt, die vor dem „01.01.1990“ 18 Jahre alt waren.

| ORT | Anzahl |
|-----------------------------|--------|
| Nienburg | 5 |
| Calbe | 3 |
| Hecklingen | 3 |
| Dresden | 1 |
| Berlin | 2 |
| Schönebeck | 1 |
| Leipzig | 1 |
| 15 Zeilen ausgewählt | |



5. Erstellen Sie eine Abfrage, die für alle bankeigenen Kunden die Buchungen auf deren Girokonten zählt. Interessant sind nur Buchungen mit einem Betrag >10.000 EUR. Sortieren Sie die Abfrage nach der Spalte KONTO_ID.

| KONTO_ID | COUNT (*) |
|------------------------------|-----------|
| 1 | 8 |
| 2 | 11 |
| 3 | 10 |
| 5 | 7 |
| 6 | 8 |
| 7 | 9 |
| 9 | 8 |
| 367 Zeilen ausgewählt | |



6. Schreiben Sie eine Abfrage, die alle Mitarbeiter anzeigt, deren Gehalt um mehr als 4.000 EUR niedriger ist, als das Durchschnittsgehalt aller Mitarbeiter.

| VORNAME | NACHNAME | GEHALT |
|-----------------------------|------------|--------|
| Louis | Wagner | 1500 |
| Lukas | Weiß | 2000 |
| Maja | Keller | 1000 |
| Karolin | Klingner | 2000 |
| Elias | Sindermann | 1000 |
| 59 Zeilen ausgewählt | | |



7. Schreiben Sie eine Abfrage, die alle Mitarbeiter anzeigt, die höchstens zwei Jahre älter sind, als der jüngste Mitarbeiter in deren Bankfiliale!



| VORNAME | NACHNAME | GEBURTSDATUM | Juengster Mitarbeiter |
|-----------|------------|--------------|-----------------------|
| Johannes | Lehmann | 1992-11-07 | 1992-11-07 |
| Dirk | Peters | 1991-09-16 | 1992-11-07 |
| Stefan | Beck | 1983-12-21 | 1984-11-16 |
| Martin | Schacke | 1984-11-16 | 1984-11-16 |
| Lukas | Weiß | 1989-03-23 | 1989-03-23 |
| Alexander | Weber | 1987-11-05 | 1989-03-23 |
| Anne | Zimmermann | 1991-01-28 | 1991-01-28 |

32 Zeilen ausgewählt

8. Schreiben Sie eine Abfrage, die zu jedem Filialleiter, das Gehalt seines am schlechtesten bezahlten Mitarbeiters anzeigt. Sortieren Sie die Abfrage nach den Bankfilial-IDs der Filialleiter.



| VORNAME | NACHNAME | GEHALT | Kleinstes Gehalt |
|-----------|----------|--------|------------------|
| Dirk | Peters | 12000 | 2000 |
| Louis | Winter | 12000 | 1000 |
| Alexander | Weber | 12000 | 1000 |
| Sophie | Schwarz | 12000 | 1500 |
| Jessica | Weber | 12000 | 1000 |

20 Zeilen ausgewählt

5.6. Lösungen - Gruppenfunktionen

1. Schreiben Sie eine Abfrage, die das höchste und das niedrigste Gehalt, das Durchschnittsgehalt und die Summe aller Gehälter ausgibt. Beschriften Sie die Spalten, wie es in der Lösung zu sehen ist.

```
SELECT MAX(Gehalt) AS "Maximum", MIN(Gehalt) AS "Minimum",
       AVG(Gehalt) AS "Mittelwert", SUM(Gehalt) AS "Summe"
FROM   Mitarbeiter;
```



2. Verändern Sie die Abfrage aus der vorangegangenen Abfrage so, dass die Informationen für jede einzelne Bankfiliale angezeigt werden. Sortieren sie das Ergebnis nach den IDs der Bankfilialen.

```
SELECT Bankfiliale_ID, MAX(Gehalt) AS "Maximum", MIN(Gehalt) AS "Minimum",
       AVG(Gehalt) AS "Mittelwert", SUM(Gehalt) AS "Summe"
FROM   Mitarbeiter
WHERE  Bankfiliale_ID IS NOT NULL
GROUP BY Bankfiliale_ID
ORDER BY Bankfiliale_ID;
```



3. Schreiben Sie eine Abfrage, die die Anzahl der Mitarbeiter pro Bankfiliale ausgibt. Beschriften Sie die Spalten so, wie es in der Lösung zu sehen ist und sortieren Sie das Ergebnis nach den IDs der Filialen.

```
SELECT Bankfiliale_ID, COUNT(*) AS "Anzahl"
FROM   Mitarbeiter
GROUP BY Bankfiliale_ID
ORDER BY Bankfiliale_ID;
```



4. Schreiben Sie eine Abfrage, die für jeden Ort einzeln, die Anzahl der Eigenkunden zählt, die vor dem „01.01.1990“ 18 Jahre alt waren.

```
SELECT Ort, COUNT(*) AS "Anzahl"
FROM   Eigenkunde ek
WHERE  Geburtsdatum + INTERVAL '18' YEAR <
       TO_DATE('01.01.1990', 'DD.MM.YYYY')
GROUP BY Ort;
```





```
SELECT Ort, COUNT(*) AS "Anzahl"
FROM Eigenkunde ek
WHERE DATEADD(YEAR, 18, Geburtsdatum) <
      CONVERT(DATETIME2, '01.01.1990', 104)
GROUP BY Ort;
```

5. Erstellen Sie eine Abfrage, die für alle bankeigenen Kunden die Buchungen auf deren Girokonten zählt. Interessant sind nur Buchungen mit einem Betrag >10.000 EUR. Sortieren Sie die Abfrage nach der Spalte KONTO_ID.



```
SELECT ekk.Konto_ID, COUNT(*)
FROM EigenkundeKonto ekk INNER JOIN Girokonto g
      ON (ekk.Konto_ID = g.Konto_ID)
      INNER JOIN Buchung b ON (g.Konto_ID = b.Konto_ID)
WHERE b.Betrag > 10000
GROUP BY ekk.Konto_ID
ORDER BY 1;
```

6. Schreiben Sie eine Abfrage, die alle Mitarbeiter anzeigt, deren Gehalt um mehr als 4.000 EUR niedriger ist, als das Durchschnittsgehalt aller Mitarbeiter.



```
SELECT m.Vorname, m.Nachname, m.Gehalt
FROM Mitarbeiter m, Mitarbeiter v
GROUP BY m.Mitarbeiter_ID, m.Vorname, m.Nachname, m.Gehalt
HAVING (m.Gehalt + 4000) < AVG(v.Gehalt);
```

7. Schreiben Sie eine Abfrage, die alle Mitarbeiter anzeigt, die höchstens zwei Jahre älter sind, als der jüngste Mitarbeiter in deren Bankfiliale!



```
SELECT m.Vorname, m.Nachname, m.Geburtsdatum,
      MAX(a.Geburtsdatum) AS "JUENGSTER MITARBEITER"
FROM Mitarbeiter m INNER JOIN Mitarbeiter a
      ON (m.Bankfiliale_ID = a.Bankfiliale_ID)
GROUP BY m.Mitarbeiter_ID, m.Vorname, m.Nachname, m.Geburtsdatum
HAVING m.Geburtsdatum BETWEEN MAX(a.Geburtsdatum) - INTERVAL '2' YEAR AND
      MAX(a.Geburtsdatum);
```



```
SELECT    m.Vorname, m.Nachname, m.Geburtsdatum,
          MAX(a.Geburtsdatum) AS "JUENGSTER MITARBEITER"
FROM      Mitarbeiter m INNER JOIN Mitarbeiter a
          ON (m.Bankfiliale_ID = a.Bankfiliale_ID)
GROUP BY  m.Mitarbeiter_ID, m.Vorname, m.Nachname, m.Geburtsdatum
HAVING    m.Geburtsdatum BETWEEN DATEADD(YEAR, -2, MAX(a.Geburtsdatum)) AND
          MAX(a.Geburtsdatum);
```

8. Schreiben Sie eine Abfrage, die zu jedem Filialleiter, das Gehalt seines am schlechtesten bezahlten Mitarbeiters anzeigt. Sortieren Sie die Abfrage nach den Bankfilial-IDs der Filialleiter.



```
SELECT    v.Vorname, v.Nachname, v.Gehalt,
          MIN(m.Gehalt) AS "Kleinstes Gehalt"
FROM      Mitarbeiter m INNER JOIN Mitarbeiter v
          ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
WHERE     v.Bankfiliale_ID IS NOT NULL
GROUP BY  v.Mitarbeiter_ID, v.Vorname, v.Nachname, v.Gehalt, v.Bankfiliale_ID
ORDER BY  v.Bankfiliale_ID;
```


6. Unterabfragen (Subqueries)

Inhaltsangabe

| | |
|--|-------------|
| 6.1 Grundsätzliches zu Unterabfragen | 6-2 |
| 6.1.1 Was sind Unterabfragen? | 6-2 |
| 6.1.2 Wann sind Unterabfragen notwendig? | 6-2 |
| 6.1.3 Regeln für Unterabfragen | 6-4 |
| 6.1.4 Arten von Unterabfragen | 6-4 |
| 6.2 Skalare Unterabfragen (Scalar Subqueries) | 6-5 |
| 6.2.1 Wo können skalare Unterabfragen stehen? | 6-5 |
| 6.2.2 Fehlerquellen in skalaren Unterabfragen | 6-6 |
| 6.3 Einspaltige Unterabfragen | 6-7 |
| 6.3.1 Einspaltige Unterabfragen in WHERE- und HAVING-Klausel | 6-7 |
| 6.3.2 Existenzprüfungen | 6-8 |
| 6.4 Inlineviews / Derived Tables | 6-10 |
| 6.5 Top N Analysen | 6-11 |
| 6.5.1 Die Top N Analyse in Oracle | 6-11 |
| 6.5.2 Die Top N Analyse in MS SQL Server | 6-14 |
| 6.6 Pivot-Tabellen | 6-14 |
| 6.6.1 Der PIVOT-Operator (Oracle) | 6-14 |
| 6.6.2 Der PIVOT-Operator (MS SQL Server) | 6-18 |
| 6.7 Übungen - Unterabfragen | 6-21 |
| 6.8 Lösungen - Unterabfragen | 6-25 |

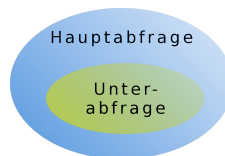
6.1. Grundsätzliches zu Unterabfragen

6.1.1. Was sind Unterabfragen?

Unterabfragen sind Abfragen, die in eine andere Abfrage, die Hauptabfrage oder „Mainquery“, eingebettet werden. Dies kann an mehreren Stellen geschehen.

- **SELECT**-Klausel
- **FROM**-Klausel (Inlineview)
- **WHERE**-Klausel
- **HAVING**-Klausel

Abb. 6.1.:
Unterabfragen



Für Unterabfragen gibt es die unterschiedlichsten Bezeichnungen.

- Subquery
- Inner query
- Nested query

6.1.2. Wann sind Unterabfragen notwendig?

Mit Hilfe von SQL können zwei verschiedene Arten von Problemstellungen gelöst werden:

- Einschrittige Problemstellungen
- Mehrschrittige Problemstellungen

Unter einer einschrittigen Problemstellung versteht man die Art von Fragestellung, die mit einer einzigen Abfrage (einem einzigen Arbeitsschritt) gelöst werden kann, so wie dies in den vorangegangenen Kapiteln der Fall war.

Mehrschrittige Problemstellungen erfordern, wie der Name es sagt, mehrere Abfragen, die aufeinander aufbauen (die eine Abfrage benötigt das Ergebnis der anderen), um zu einer Lösung zu kommen. Eine solche Problemstellung könnte z. B. so lauten: „Wie hoch ist das Gehalt des Vorgesetzten der Mitarbeiterin *Lena Große*?“

Diese Frage lässt sich in zwei Fragen teilen:

1. Wer ist der Vorgesetzte von Lena Große?
2. Wie hoch ist dessen Gehalt?

Die Antworten zu beiden Fragen lassen sich sehr einfach als SQL-Statements formulieren.

Listing 6.1: Wer ist der Vorgesetzte von Lena Grosse

```
SELECT Vorgesetzter_ID
FROM   Mitarbeiter
WHERE  Vorname LIKE 'Lena' AND Nachname LIKE 'Grosse';
```

VORGESETZTER_ID

6

1 Zeile ausgewählt

Listing 6.2: Wie hoch ist dessen Gehalt

```
SELECT Gehalt
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID = 6;
```

GEHALT

30000

1 Zeile ausgewählt

Mit Hilfe der beiden Abfragen wurde die Antwort ermittelt: „Der Vorgesetzte von Lena Große hat ein Gehalt von 30.000 EUR.“. Durch das Kombinieren beider Queries, lässt sich diese Aufgabe viel eleganter lösen. [Beispiel 6.3](#) zeigt einen möglichen Lösungsansatz.

Listing 6.3: Wie hoch ist das Gehalt des Vorgesetzten der Mitarbeiterin *Lena Große*?

```
SELECT Gehalt
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID = (SELECT Vorgesetzter_ID
                           FROM   Mitarbeiter
                           WHERE  Vorname LIKE 'Lena'
                           AND    Nachname LIKE 'Grosse');
```



| GEHALT |
|--------|
| 30000 |

1 Zeile ausgewählt



Das DBMS arbeitet bei einer solchen Auswahlabfrage immer zuerst die Unterabfrage(n) ab!

6.1.3. Regeln für Unterabfragen

- Unterabfragen stehen immer in Klammern!
- Es können alle ihnen bisher bekannten Operatoren eingesetzt werden!
- Unterabfragen **sollten immer ohne ORDER BY-Klausel** erstellt werden!

Die Aussage, dass Unterabfragen immer ohne **ORDER BY** verwendet werden sollten, rührt daher, dass falls eine Sortierung in der Hauptabfrage stattfindet, zuerst in der Unterabfrage sortiert wird und anschließend nochmals in der Hauptabfrage. Dies führt zu unnötiger Sortierarbeit, die die Datenbank belastet.



In MS SQL Server darf eine Unterabfrage kein **ORDER BY** enthalten. Das DBMS antwortet sonst mit einer Fehlermeldung (Meldung 1033, Ebene 15).

6.1.4. Arten von Unterabfragen

Grundsätzlich gibt es vier unterschiedliche Arten von Unterabfragen:

- Skalare Unterabfragen: Eine solche Abfrage liefert exakt einen Wert zurück.
- Einspaltige Unterabfragen: Dieser Abfragetyp liefert mehrere Werte aus einer Spalte zurück.
- Mehrspaltige Unterabfragen: Die Abfrage liefert Werte mehrerer Spalten zurückgeliefert.
- Korrelierte Unterabfragen: Ihre Ausführung ist von der Hauptabfrage abhängig.

6.2. Skalare Unterabfragen (Scalar Subqueries)

**Skalar:**

Größe aus der Mathematik, die durch die Angabe eines einzelnen Wertes genau definiert werden kann.

Beispiele für Skalare sind: Gehalt, Provision, ...

Skalare Unterabfragen zeichnen sich dadurch aus, dass sie genau einen einzigen Wert zurückliefern. Dies wird mit Hilfe einer entsprechenden **WHERE**-Klausel innerhalb der Unterabfrage erreicht. Ergibt die Abfrage kein Ergebnis, wird NULL zurückgeliefert. Ein erstes Beispiel für diese Art von Unterabfrage war in [Beispiel 6.3](#) zu sehen. Es wird nur das GEHALT eines einzigen Angestellten angezeigt.

6.2.1. Wo können skalare Unterabfragen stehen?

Skalare Unterabfragen können in allen in [Abschnitt 6.1.1](#) erwähnten Klauseln stehen.

Skalare Unterabfragen in der SELECT-Klausel

Skalare Unterabfragen sind die einzigen, die in der **SELECT**-Klausel eines SQL-Statements stehen dürfen. Sie können beispielsweise dazu dienen, um einen Outer-Join zu vermeiden, meist ist jedoch die Join-Variante sehr viel performanter. Aus diesem Grund sollten skalare Unterabfragen in der **SELECT**-Klausel absolut vermieden werden.



Skalare Unterabfragen in der **SELECT**-Klausel sollten unter allen Umständen vermieden werden!

Skalare Unterabfragen in der WHERE-Klausel

Die **WHERE**-Klausel ist der Ort, an dem skalare Unterabfragen am häufigsten anzutreffen sind. Sie dienen zur Berechnung von Werten, mit deren Hilfe das Resultat der Hauptabfrage eingeschränkt wird (siehe [Beispiel 6.3](#)).

Skalare Unterabfragen in der Having-Klausel

Hier gelten die gleichen Grundsätze, wie in der **WHERE**-Klausel. Der einzige Unterschied ist, dass hier ein Aggregat mit dem Resultat einer skalaren Unterabfrage verglichen werden kann.

6.2.2. Fehlerquellen in skalaren Unterabfragen

Die häufigste Fehlerquelle, im Umgang mit skalaren Unterabfragen, ist eine falsche **WHERE**-Klausel. Schränkt sie das Ergebnis der Unterabfrage nicht genügend ein, wird mehr als ein Datensatz/Wert zurückgeliefert und die Datenbank antwortet mit einer Fehlermeldung.

Listing 6.4: Mehr als eine Zeile: Fehlermeldung in Oracle

```
ORA-01427: Unterabfrage fuer eine Zeile liefert mehr als eine Zeile
```

Listing 6.5: Mehr als eine Zeile: Fehlermeldung in SQL Server

```
Meldung 512, Ebene 16, Status 1, Zeile 1
Die Unterabfrage hat mehr als einen Wert zurueckgegeben. Das ist nicht
zulaessig, wenn die Unterabfrage auf =, !=, <, <=, > oder >= folgt oder
als Ausdruck verwendet wird.
```

Hier ein Beispiel zu diesen Fehlermeldungen: Es soll das Geburtsdatum des Vorgesetzten der Mitarbeiterin „Große“ ermittelt werden.

Listing 6.6: Eine Single Row Unterabfrage mit Problemen!

```
SELECT Geburtsdatum
FROM Mitarbeiter
WHERE Mitarbeiter_ID = (SELECT Vorgesetzter_ID
                        FROM Mitarbeiter
                        WHERE LOWER(Nachname) LIKE 'grosse');
```

Das Problem bei dieser Abfrage ist, dass die Tabelle MITARBEITER zwei Angestellte mit dem Namen „Große“ enthält. Das bedeutet, die Unterabfrage liefert mehr als einen Wert zurück, so dass der Vergleich mit einem Single Row Operator scheitert.



Als Single Row Operatoren werden relationale Operatoren bezeichnet, die einen Wert auf ihrer linken Seite mit genau einem Wert auf ihrer rechten Seite vergleichen können. Hierzu zählen: = >= <= < > != LIKE

6.3. Einspaltige Unterabfragen

Diese Kategorie der Unterabfragen unterscheidet sich von den skalaren dahingehend, dass sie eine einspaltige Liste von mehreren Werten (Vektor) zurückliefern und dass sie nicht in der **SELECT**-Klausel eines SQL-Statements vorkommen dürfen.

6.3.1. Einspaltige Unterabfragen in WHERE- und HAVING-Klausel

IN (bekannt aus [Abschnitt 2.1](#)) ist der einzige Operator, der auf seiner rechten Seite nicht nur einen einzelnen Wert, sondern eine ganze Wertemenge verarbeiten kann. Dies kann eine konstante Menge sein, so wie dies bisher der Fall war, aber es kann auch eine, durch eine Query dynamisch generierte Menge sein. [Beispiel 6.7](#) zeigt den Einsatz des **IN**-Operators. Es muss eine Liste aller Kunden ermittelt werden, die vor dem „01.01.1980“ ein Konto bei der Bank eröffnet haben.

Listing 6.7: **IN** mit Unterabfrage

```
SELECT Vorname , Nachname
FROM Kunde
WHERE Kunden_ID IN (SELECT Kunden_ID
                     FROM EigenkundeKonto
                     WHERE Eröffnungsdatum < TO_DATE('01.01.1980'));
```

| VORNAME | NACHNAME |
|---------|----------|
| Jan | Weiß |
| Petra | Berger |
| Karolin | Lange |
| Tom | Hartmann |

28 Zeilen ausgewählt



Auf die gleiche Art und Weise, wie in [Beispiel 6.7](#) gezeigt, können einspaltige Unterabfragen auch in einer **HAVING**-Klausel eingesetzt werden, was jedoch nur sehr selten vorkommt.

6.3.2. Existenzprüfungen

Der EXISTS-Operator

Der Name *Existenzprüfung* sagt ohne Umschweife aus, worum es geht. Mit Hilfe des Operators **EXISTS** kann die Existenz bestimmter Daten geprüft werden. [Beispiel 6.8](#) zeigt auf worum es sich hierbei handelt. Es soll eine Liste der Bankfilialen ermittelt werden, in denen Mitarbeiter eingesetzt sind.

Listing 6.8: Der **EXISTS**-Operator

```
SELECT Strasse, Hausnummer, PLZ, Ort
FROM   Bankfiliale b
WHERE  EXISTS (SELECT 1
               FROM   Mitarbeiter m
               WHERE  b.Bankfiliale_ID = m.Bankfiliale_ID);
```



| STRASSE | HAUSNUMMER | PLZ | ORT |
|----------------|------------|-------|--------------|
| Poststraße | 1 | 06449 | Aschersleben |
| Markt | 5 | 06449 | Aschersleben |
| Goethestraße | 4 | 39240 | Calbe |
| Lessingstraße | 1 | 06406 | Bernburg |
| Schillerstraße | 7 | 39240 | Barby |

20 Zeilen ausgewählt

Das Ergebnis dieser Auswahlabfrage sind alle Bankfilialen, in denen Mitarbeiter arbeiten. Es verbleibt eine Filiale ohne Mitarbeiter.

Was geschieht in dieser Abfrage nun in welcher Reihenfolge?

1. Die **FROM**-Klausel der Hauptabfrage wird ausgewertet und die erforderlichen Daten werden ermittelt.
2. Die **FROM**-Klausel der Unterabfrage wird ausgewertet und die erforderlichen Daten werden ermittelt.
3. Die **WHERE**-Klausel der Unterabfrage wird ausgeführt. Der Join zwischen BANKFILIALE und MITARBEITER wird gebildet.
4. Die **WHERE**-Klausel der Hauptabfrage wird ausgeführt.
5. Die **SELECT**-Klausel der Hauptabfrage liefert die benötigten Daten.

Das Besondere an dieser Form der Abfrage ist die **WHERE**-Klausel der Unterabfrage. Dort wird die Tabelle BANKFILIALE (Hauptabfrage) mit der Tabelle MITARBEITER (Unterabfrage) verknüpft. Die Unterabfrage kann somit auf die Datensätze der Hauptabfrage zugreifen.

Werden die Tabellen einer Unterabfrage mit einer Tabelle der Hauptabfrage verknüpft, spricht man von einer „korrelierten Unterabfrage“.



Für die Ausführung des gesamten Statements bedeutet dies, dass die Unterabfrage nicht nur einmal, sondern mehrfach ausgeführt werden muss. Genauer gesagt wird die Unterabfrage für jede Zeile der Hauptabfrage einmal ausgeführt. Bezogen auf [Beispiel 6.8](#) bedeutet dies, dass die Unterabfrage 21 mal ausgeführt wird, da die Tabelle Bankfiliale 21 Datensätze hat. Die Mehrfachausführung der Unterabfrage ist notwendig, da für jede Bankfiliale einzeln geprüft werden muss, ob es dort Mitarbeiter gibt oder nicht.

Eine weitere Besonderheit dieser Art von Abfrage ist die **SELECT**-Klausel der Unterabfrage. Dort stehen keine Spaltenbezeichner und auch kein *. Statt dessen wird hier ein Literal, eine 1 (eins) verwendet. Der Hintergrund hierfür ist, dass die **SELECT**-Klausel der Unterabfrage für die Ausführung des gesamten Statements keine Bedeutung hat. Es wird nur geprüft, ob für jeden Datensatz der Hauptabfrage ein Datensatz in der Unterabfrage existiert. Das bedeutet, dass sobald die Unterabfrage eine Zeile zurückliefert die Bedingung erfüllt ist und der Datensatz der Hauptabfrage angezeigt wird.

Der NOT EXISTS-Operator

Der **NOT EXISTS**-Operator stellt das Pendant zum **EXISTS**-Operator dar. Müssen beispielsweise alle Filialen ermittelt werden, in denen keine Mitarbeiter arbeiten kommt **NOT EXISTS** zum Einsatz.

Listing 6.9: Der **NOT EXISTS**-Operator

```
SELECT Strasse, Hausnummer, PLZ, Ort
FROM   Bankfiliale b
WHERE  NOT EXISTS (SELECT 1
                   FROM   Mitarbeiter m
                   WHERE  b.Bankfiliale_ID = m.Bankfiliale_ID);
```

| STRASSE | HAUSNUMMER | PLZ | ORT |
|-------------|------------|-------|----------|
| Kurze Gasse | 47 | 06425 | Alsleben |

1 Zeile ausgewählt



6.4. Inlineviews / Derived Tables

In [Abschnitt 6.1.1](#) wurde bereits erwähnt, dass eine Unterabfrage auch in der **FROM**-Klausel eines SQL-Statements stehen kann.



Eine Unterabfrage in der **FROM**-Klausel wird in Oracle als „Inlineview“ und in MS SQL Server als „Derived Table“ bezeichnet.

[Beispiel 6.10](#) zeigt ein SQL-Statement, welches eine Inlineview nutzt.

Listing 6.10: Eine Inlineview

```
SELECT Vorname, Nachname, MinGehalt
FROM   (SELECT   Bankfiliale_ID, MIN(Gehalt) MinGehalt
        FROM     Mitarbeiter
        GROUP BY Bankfiliale_ID) m1
       INNER JOIN Mitarbeiter m
       ON (m1.Bankfiliale_ID = m.Bankfiliale_ID)
WHERE  m.Gehalt = m1.MinGehalt;
```



| VORNAME | NACHNAME | MINGEHALT |
|----------|------------|-----------|
| Johannes | Lehmann | 2000 |
| Louis | Schmitz | 2000 |
| Marie | Kipp | 2000 |
| Martin | Schacke | 1000 |
| Oliver | Wolf | 1000 |
| Hans | Schumacher | 1000 |
| Lena | Herrmann | 1500 |

29 Zeilen ausgewählt

In [Beispiel 6.10](#) wird die Inlineview dazu benutzt, um das kleinste Gehalt je Abteilung zu berechnen. Mit Hilfe des Joins wird sie mit der Tabelle MITARBEITER verknüpft, so dass die Attribute VORNAME und NACHNAME angezeigt werden können, ohne in Konflikt mit der **GROUP BY**-Klausel zu kommen.



Inlineviews bieten eine gute Möglichkeit, um gruppierte und ungruppierte Informationen in einer Abfrage gemeinsam anzeigen zu können.

6.5. Top N Analysen

Die Top N Analyse ist ein Verfahren, bei dem Datensätze in ein Ranking eingeordnet werden. Hiermit werden Fragestellungen geklärt wie z. B.:

- Die 3 reichsten Kunden anzeigen
- Die 5 Mitarbeiter mit den höchsten Gehältern auflisten
- Die beiden größten Schuldner der Bank ermitteln

Beide Datenbankmanagementsysteme beherrschen diese Technik, gehen dabei aber unterschiedliche Wege.

6.5.1. Die Top N Analyse in Oracle

Die Top N Analyse funktioniert in Oracle mit Hilfe einer sortierten Inlineview und einer Pseudospalte Namens ROWNUM.

Die Pseudospalte Rownum

Mit der Bezeichnung „Pseudospalte“ ist gemeint, dass die ROWNUM keine tatsächlich vorhandene Spalte ist, obwohl sie in jeder Abfrage verwendet werden kann. Sie bietet die Möglichkeit, die Ergebniszeilen einer Abfrage fortlaufend zu nummerieren (1, 2, 3, ..., N). Zu beachten ist dabei, dass eine Zeile in einer Oracle-Datenbank keine feste Nummerierung hat. Diese wird erst im Ergebnis einer Abfrage zugeordnet.

Listing 6.11: Ein einfaches Beispiel für die Rownum

```
SELECT  Rownum, Vorname, Nachname
FROM    Mitarbeiter
WHERE   Ort LIKE 'Aschersleben';
```

| ROWNUM | VORNAME | NACHNAME |
|--------|-----------|----------|
| 1 | Max | Winter |
| 2 | Alexander | Weber |
| 3 | Leni | Dühning |

3 Zeilen ausgewählt



Eine Tabellenzeile hat keine feste Nummerierung. Die Rownum wird während der Abarbeitung einer Abfrage zugewiesen.

Eine weitere, entscheidende Tatsache ist, dass die ROWNUM erst nach der Abarbeitung der **WHERE**-Klausel zugeordnet wird, aber noch bevor Gruppierungen oder Sortierungen ausgeführt werden. Aus diesem Grund, wird die Abfrage in [Beispiel 6.12](#) ein falsches Ergebnis liefern, da die Sortierung hätte zuerst stattfinden müssen. Hier werden höchstwahrscheinlich nicht die beiden größten Guthaben, sondern zwei beliebige Guthaben angezeigt. Welche Zeilen gelistet werden hängt davon ab, welche die Abfrage zuerst ermittelt.

Listing 6.12: Falsche Anwendung der Rownum-Pseudospalte

```
SELECT Konto_ID, Guthaben
FROM Girokonto
WHERE Rownum < 3
ORDER BY Guthaben DESC;
```



| KONTO_ID | GUTHABEN |
|----------|----------|
| 1 | 111316,9 |
| 2 | 96340,2 |

2 Zeilen ausgewählt



Die Rownum wird erst nach Abarbeitung der **WHERE**-Klausel, aber noch vor allen Gruppierungen und Sortierungen hinzugefügt.

Ein dritter „Stolperstein“, in Zusammenhang mit der ROWNUM ist, dass die ROWNUM erst inkrementiert wird, wenn sie zugewiesen wurde. Das soll heißen, dass die **WHERE**-Klausel in [Beispiel 6.13](#) ebenfalls fehlschlägt, da nach allen Rownums größer eins gefragt wird, ohne das Rownum eins jemals zugewiesen worden wäre (ohne 1 keine 2).

Listing 6.13: Erneut eine falsche Anwendung der Rownum

```
SELECT Konto_ID, Guthaben
FROM Girokonto
WHERE Rownum > 3
ORDER BY Guthaben DESC;
```



| KONTO_ID | GUTHABEN |
|----------|----------|
|----------|----------|

0 Zeilen ausgewählt

Die Lösung für diese Probleme besteht nun darin,

1. dass niemals einer der beiden Operatoren > oder >= in Zusammenhang mit der ROWNUM verwendet werden sollte und
2. dass die Abfrage aus [Beispiel 6.12](#) in eine Inlineview geschachtelt wird.

Durchführung der Top N Analyse

Die korrekte Form der Top N Analyse sieht in Oracle wie folgt aus:

Listing 6.14: Eine korrekt funktionierende Top N Analyse in Oracle

```
SELECT *
FROM (SELECT Konto_ID, Guthaben
      FROM Girokonto
      ORDER BY Guthaben DESC)
WHERE Rownum < 3;
```

| KONTO_ID | GUTHABEN |
|----------|----------|
| 362 | 147670,3 |
| 198 | 147264 |

2 Zeilen ausgewählt



Im Gegensatz zu [Beispiel 6.12](#) werden hier wirklich die beiden größten Gehälter angezeigt. Warum dies so ist, kann durch die Abarbeitungsreihenfolge der Abfrage aus [Beispiel 6.14](#) erklärt werden.

1. FROM-Klausel der Inlineview
2. SELECT- und ORDER BY-Klausel der Inlineview
3. FROM-Klausel der Hauptabfrage
4. Zuweisung der ROWNUM
5. Ausführung der WHERE-Klausel der Hauptabfrage
6. SELECT-Klausel der Hauptabfrage

In [Beispiel 6.14](#) wird also zuerst nummeriert und dann selektiert.

6.5.2. Die Top N Analyse in MS SQL Server

In Microsoft SQL Server existiert eigens der Operator **TOP** zur Durchführung von Top N Analysen. Er wird in der **SELECT**-Klausel eingesetzt und legt fest, wie viele Zeilen angezeigt werden.

Listing 6.15: Top N Analyse in MS SQL Server

```
SELECT TOP (2) Konto_ID, Guthaben
FROM Girokonto
ORDER BY Guthaben DESC;
```



| KONTO_ID | GUTHABEN |
|----------|----------|
| 362 | 147670,3 |
| 198 | 147264 |

2 Zeilen ausgewählt

Durch die Angabe von **TOP** (2) werden nur die ersten zwei Zeilen der Ergebnismenge angezeigt.

6.6. Pivot-Tabellen

Mit MS SQL Server 2005 bzw. Oracle 11g R1 wurden der **PIVOT** und der **UNPIVOT**-Operator eingeführt. Diese ermöglichen die einfache Erstellung von Pivottabellen.



In einer Pivottabelle werden Daten, die im Zeilenformat vorliegen, im Spaltenformat angezeigt oder umgekehrt. Das „Drehen“ der Daten wird als „Pivoting“ bezeichnet, woraus sich der Name für diese Tabellen ableitet.

- **PIVOT:** Dreht Daten die zeilenweise vorliegen so, dass eine spaltenweise Darstellung möglich ist.
- **UNPIVOT:** Dreht Daten die spaltenweise vorliegen so, dass eine zeilenweise Darstellung möglich ist.

6.6.1. Der PIVOT-Operator (Oracle)

Die Möglichkeiten, die der **PIVOT**-Operator bietet, werden anhand des folgenden Beispiels verdeutlicht. Für die Filialen 1 bis 3 sollen die jeweils kleinsten Gehälter angezeigt werden.

Listing 6.16: Die niedrigsten Gehälter in den Filialen 1 bis 3

```
SELECT  Bankfiliale_ID, MIN(Gehalt)
FROM    Mitarbeiter
WHERE   Bankfiliale_ID IN (1, 2, 3)
GROUP BY Bankfiliale_ID;
```

| BANKFILIALE_ID | MIN(GEHALT) |
|----------------|-------------|
| 1 | 2000 |
| 2 | 1000 |
| 3 | 1000 |

3 Zeilen ausgewählt



In [Beispiel 6.16](#) werden die gewünschten Zahlen ermittelt. Die Darstellung der Gehälter erfolgt zeilenweise. Sollen die gleichen Zahlen spaltenweise dargestellt werden, wird der **PIVOT**-Operator benötigt. [Beispiel 6.17](#) zeigt dessen Einsatz.

Listing 6.17: Das Ergebnis als Pivottabelle

```
SELECT *
FROM    (SELECT Gehalt, Bankfiliale_ID
        FROM    Mitarbeiter)
PIVOT   (MIN(Gehalt) AS Gehalt FOR Bankfiliale_ID IN (1, 2, 3));
```

| 1_GEHALT | 2_GEHALT | 3_GEHALT |
|----------|----------|----------|
| 2000 | 1000 | 1000 |

1 Zeile ausgewählt



Die Syntax des PIVOT-Operators

Da das SQL-Statement aus [Beispiel 6.17](#) auf den ersten Blick sehr komplex wirkt, ist es notwendig, es an dieser Stelle im Detail zu betrachten.

Für die Ausführung des Pivoting wird in [Beispiel 6.17](#) eine Inlineview verwendet.

Listing 6.18: Die Inlineview

```
(SELECT Gehalt, Bankfiliale_ID
FROM    Mitarbeiter)
```

Diese Inlineview legt fest, welche Spalten im Endergebnis der Abfrage zu sehen sein werden. Sie kann beliebig komplex sein. Der **PIVOT**-Operator verarbeitet im zweiten Schritt die Spalten dieser View weiter.

Listing 6.19: Der **PIVOT**-Operator

```
PIVOT  (MIN(Gehalt) AS Gehalt FOR Bankfiliale_ID IN (1, 2, 3));
```

Die Bedeutung dieses Operators ist:

- Gruppiere nach der Spalte **BANKFILIALE_ID**.
- Zeige **MIN**(Gehalt) für Bankfiliale_ID = 1.
- Zeige **MIN**(Gehalt) für Bankfiliale_ID = 2.
- Zeige **MIN**(Gehalt) für Bankfiliale_ID = 3.
- Benutzte den Alias „Gehalt“ für den Ausdruck **MIN**(Gehalt).

Spaltenalias in der FOR-Klausel

Die Spaltenbezeichnungen im Ergebnis von [Beispiel 6.17](#) werden gebildet, in dem der Name der aggregierten Spalte (hier **GEHALT**) mit den Werten der **FOR**-Klausel kombiniert werden. Dadurch entstehen die Namen **1_GEHALT**, **2_GEHALT** und **3_GEHALT**. Auch an dieser Stelle sind Aliasnamen möglich.

Listing 6.20: Die **FOR**-Klausel mit Aliasnamen

```
SELECT *
FROM   (SELECT Gehalt, Bankfiliale_ID
         FROM   Mitarbeiter)
PIVOT  (MIN(Gehalt) AS Gehalt FOR Bankfiliale_ID
         IN (1 AS "Filiale 1", 2 AS "Filiale 2", 3 AS "Filiale 3"));
```



| Filiale 1_GEHALT | Filiale 2_GEHALT | Filiale 3_GEHALT |
|-------------------------|-------------------------|-------------------------|
| 2000 | 1000 | 1000 |

1 Zeile ausgewählt

Zusätzliche Spalten zum Pivoting

In einer Pivot-Abfrage können noch weitere Spalten enthalten sein, die nicht aggregiert oder in der **FOR**-Klausel genutzt werden. Diese Spalten werden als zusätzliche Gruppierungsmerkmale genutzt.

Oracle führt eine implizite Gruppierung der Ergebnismenge durch. Diese basiert auf allen nicht gruppierten Spalten, inklusive der Spalten, die in der **FOR**-Klausel genutzt werden.



In **Beispiel 6.21** wird im ersten Schritt nach dem Geburtsjahr, von 1987 bis 1989 gruppiert. Da diese Spalte in der **FOR**-Klausel verwendet wird, wird diese Information in Spaltenform dargestellt.

Die Spalte Ort hingegen, wird in Zeilenform angezeigt, da sie nicht in der **FOR**-Klausel angegeben wurde.

Ob eine Information in Spalten- oder Zeilenform dargestellt wird, hängt davon ab, ob die betreffende Spalte in der **FOR**-Klausel gelistet wurde oder nicht.



Listing 6.21: Zusätzliche Gruppierungen in einer Pivot-Abfrage

```
SELECT *
FROM (SELECT Gehalt, TO_CHAR(Geburtsdatum, 'YYYY') AS Geburtsdatum, Ort
      FROM Mitarbeiter)
PIVOT (MIN(Gehalt) AS Gehalt FOR Geburtsdatum IN ('1987', '1988', '1989'));
```

| ORT | '1987'_GEHALT | '1988'_GEHALT | '1989'_GEHALT |
|---------------------------|---------------|---------------|---------------|
| Calbe | | | |
| Plötzkau | 2500 | | |
| Nienburg | | | |
| Bernburg | | | |
| Dresden | | | |
| Hecklingen | | | 3000 |
| Borne | | | 30000 |
| Schönebeck | | | |
| Giersleben | | | |
| Gera | | | 3500 |
| ... | | | |
| Hamburg | 12000 | | |
| Alsleben | | | |
| Schwerin | | | |
| Dessau | 2500 | | |
| Könnern | | | |
| Cottbus | | | |
| Potsdam | 3500 | 2000 | 2000 |
| Aschersleben | 12000 | 88000 | 2000 |
| 1 Zeile ausgewählt | | | |
| Magdeburg | 3000 | | |
| 1 Zeile ausgewählt | | | |



Die vorangegangenen Beispiele stellen nur einen Einstieg in das Thema „Pivottabellen“ dar. Tatsächlich ist der **PIVOT**-Operator noch weitaus mächtiger.

6.6.2. Der PIVOT-Operator (MS SQL Server)

Die Möglichkeiten, welche der **PIVOT**-Operator bietet, werden anhand des folgenden Beispiels verdeutlicht. Für die Bankfilialen 1 bis 3 sollen die jeweils kleinsten Gehälter angezeigt werden.

Listing 6.22: Die niedrigsten Gehälter in den Filialen 1 bis 3

```
SELECT Bankfiliale_ID, MIN(Gehalt)
FROM Mitarbeiter
WHERE Bankfiliale_ID IN (1, 2, 3)
GROUP BY Bankfiliale_ID;
```



| Bankfiliale_ID | (Kein Spaltenname) |
|----------------|--------------------|
| 1 | 2000 |
| 2 | 1000 |
| 3 | 1000 |

3 Zeilen ausgewählt

In [Beispiel 6.22](#) werden die gewünschten Zahlen ermittelt. Die Darstellung der Gehälter erfolgt zeilenweise. Sollen die gleichen Zahlen spaltenweise dargestellt werden, wird der **PIVOT**-Operator benötigt. [Beispiel 6.23](#) zeigt dessen Einsatz.

Listing 6.23: Das Ergebnis als Pivottabelle

```
SELECT *
FROM (SELECT Gehalt, Bankfiliale_ID
      FROM Mitarbeiter) AS Sourcetable
PIVOT (MIN(Gehalt)
       FOR Bankfiliale_ID IN ([1], [2], [3])
       ) AS Pivottable;
```



| 1 | 2 | 3 |
|------|------|------|
| 2000 | 1000 | 1000 |

1 Zeile ausgewählt

Die Syntax des PIVOT-Operators

Da das SQL-Statement aus [Beispiel 6.23](#) auf den ersten Blick sehr komplex wirkt, ist es notwendig, es an dieser Stelle im Detail zu betrachten.

Für die Ausführung des Pivotings wird in [Beispiel 6.23](#) eine Inlineview verwendet.

Listing 6.24: Die Inlineview

```
(SELECT Gehalt, Bankfiliale_ID  
FROM Mitarbeiter) AS Sourcetable
```

Diese Inlineview legt fest, welche Spalten im Endergebnis der Abfrage zu sehen sein werden. Sie kann beliebig komplex sein. Der **PIVOT**-Operator verarbeitet im zweiten Schritt die Spalten dieser View weiter.

Listing 6.25: Der **PIVOT**-Operator

```
PIVOT (MIN(Gehalt) FOR Bankfiliale_ID IN ([1], [2], [3])) AS Pivottable;
```

Die Bedeutung dieses Operators ist:

- Gruppiere nach der Spalte **BANKFILIALE_ID**.
- Zeige **MIN**(Gehalt) für **Bankfiliale_ID** = 1.
- Zeige **MIN**(Gehalt) für **Bankfiliale_ID** = 2.
- Zeige **MIN**(Gehalt) für **Bankfiliale_ID** = 3.

Für eine Pivotabfrage gelten in MS SQL Server folgende Syntaxregeln:

- Für die Quell-View muss zwingend ein Aliasname vergeben werden. In [Beispiel 6.23](#) ist dies „Sourcetable“
- Für die Pivottabelle muss zwingend ein Aliasname vergeben werden. In [Beispiel 6.23](#) ist dies „Pivottable“
- Es dürfen in der Pivottabelle keine Aliasnamen vergeben werden.
- Die Werte in der **FOR**-Klausel müssen in eckigen Klammern stehen.

Zusätzliche Spalten zum Pivoting

In einer Pivot-Abfrage können noch weitere Spalten enthalten sein, die nicht aggregiert oder in der **FOR**-Klausel genutzt werden. Diese Spalten werden als zusätzliche Gruppierungsmerkmale genutzt.



MS SQL Server führt eine implizite Gruppierung der Ergebnismenge durch. Diese basiert auf allen nicht gruppierten Spalten, inklusive der Spalten, die in der **FOR**-Klausel genutzt werden.

In **Beispiel 6.26** wird im ersten Schritt nach dem Geburtsjahr, von 1987 bis 1989 gruppiert. Da diese Spalte in der **FOR**-Klausel verwendet wird, wird diese Information in Spaltenform dargestellt. Die Spalte Ort hingegen, wird in Zeilenform angezeigt, da sie nicht in der **FOR**-Klausel angegeben wurde.



Ob eine Information in Spalten- oder Zeilenform dargestellt wird, hängt davon ab, ob die betreffende Spalte in der **FOR**-Klausel gelistet wurde oder nicht.

Listing 6.26: Zusätzliche Gruppierungen in einer Pivot-Abfrage

```
SELECT *
FROM (SELECT Gehalt, DATEPART(YEAR, Geburtsdatum) AS Geburtsdatum, Ort
      FROM Mitarbeiter) AS Sourcetable
PIVOT (MIN(Gehalt)
      FOR Geburtsdatum IN ([1987], [1988], [1989])) AS Pivottable;
```



| Ort | 1987 | 1988 | 1989 |
|-----------------------------|-------|-------|-------|
| Calbe | | | |
| Plötzkau | 2500 | | |
| Nienburg | | | |
| Bernburg | | | |
| Dresden | | | |
| Hecklingen | | | 3000 |
| Borne | | | 30000 |
| Schönebeck | | | |
| Börde | | | |
| ... | | | |
| Cottbus | | | |
| Potsdam | 3500 | 2000 | 2000 |
| Aschersleben | 12000 | 88000 | 2000 |
| Magdeburg | | | 3000 |
| 25 Zeilen ausgewählt | | | |
| 25 Zeilen ausgewählt | | | |

6.7. Übungen - Unterabfragen

- Schreiben Sie eine Abfrage, die für alle Eigenkunden, die keinen Berater haben (die nicht in der Tabelle EIGENKUNDEMITARBEITER enthalten sind), den Vor- und den Nachnamen anzeigt.

- Lösen Sie die Aufgabe mit Hilfe des **EXISTS**-Operators!
- Lösen Sie die Aufgabe mit Hilfe des **IN**-Operators!

| VORNAME | NACHNAME |
|-----------|------------|
| Sebastian | Schröder |
| Udo | Schumacher |
| Mia | Huber |
| Simon | Witte |
| Max | Bunzel |
| Finn | Fischer |
| Lara | Meierhöfer |
| Jannis | Meier |

16 Zeilen ausgewählt



- Erstellen Sie eine Abfrage, die ermittelt, ob es Mitarbeiter gibt (Vorname und Nachname), die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten) und Filialleiter.

- Lösen Sie die Aufgabe mit Hilfe des **EXISTS**-Operators!
- Lösen Sie die Aufgabe mit Hilfe des **IN**-Operators!

| VORNAME | NACHNAME |
|-----------|------------|
| Amelie | Krüger |
| Anna | Schneider |
| Chris | Simon |
| Christian | Haas |
| Elias | Sindermann |
| Emilia | Köhler |
| Emma | Krüger |

40 Zeilen ausgewählt



3. Schreiben Sie eine Abfrage, die den häufigsten Vornamen der Bankmitarbeiter anzeigt und wie oft dieser in der Tabelle MITARBEITER vorkommt.



| VORNAME | ANZAHL |
|---------|--------|
| Chris | 5 |

1 Zeile ausgewählt

4. Schreiben Sie eine Abfrage, welche die drei Eigenkunden mit den niedrigsten Guthaben auf den Girokonten anzeigt.



| VORNAME | NACHNAME | GUTHABEN |
|---------|----------|-----------|
| Franz | Walther | -140505,1 |
| Jan | Simon | -98218,6 |
| Philipp | Hartmann | -69705,6 |

3 Zeilen ausgewählt

5. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass die drei Eigenkunden mit dem niedrigsten Guthaben (Girokonto + Sparbuch) angezeigt werden. Es müssen auch diejenigen Kunden angezeigt werden, die nur ein Girokonto oder nur ein Sparbuch haben!



| VORNAME | NACHNAME | SUM (GUTHABEN) |
|---------|----------|----------------|
| Franz | Walther | -139154,4 |
| Jan | Simon | -98218,6 |
| Philipp | Hartmann | -69065,9 |

3 Zeilen ausgewählt

6. Schreiben Sie eine Abfrage, die alle Eigenkunden anzeigt, welche im Jahr 1985 keine Buchungen verursacht haben.



| VORNAME | NACHNAME |
|---------|----------|
| Sarah | Bauer |
| Sofia | Bauer |
| Tom | Bauer |
| Alina | Baumann |

285 Zeilen ausgewählt

7. Schreiben Sie eine Abfrage, die für jede Bankfiliale den Mitarbeiter mit dem höchsten Gehalt ausgibt.

| BANKFILIALE | VORNAME | NACHNAME | GEHALT |
|---------------------------------|---------|----------|--------|
| Poststraße 1 06449 Aschersleben | Dirk | Peters | 12000 |
| Kirchstraße 8 39444 Hecklingen | Leonie | Kaiser | 12000 |
| Schmiedestraße 3 39240 Staßfurt | Finn | Köhler | 12000 |
| Am Dom 11 06449 Giersleben | Lena | Große | 12000 |

20 Zeilen ausgewählt



8. Schreiben Sie eine Abfrage, die für jeden Wohnort (EIGENKUNDE.ORT) den Kunden anzeigt, der im Jahr 1987 das höchste Einkommen hatte (Das Einkommen ist die Summe aller Beträge eines Kunden, in der Tabelle BUCHUNG). Sortieren Sie die Abfrage nach den Wohnorten.

| ORT | VORNAME | NACHNAME | BETRAG |
|--------------|---------|----------|---------|
| Alsleben | Peter | Koch | 57855,4 |
| Aschersleben | Lara | Dühning | 2395,7 |
| Barby | Chris | Beck | -6817,8 |

30 Zeilen ausgewählt



9. Erstellen Sie eine Abfrage, die die Umsätze der Bank (SUM(Buchung.Betrag)) für die Jahre 1985 bis einschließlich 1989 als Pivottabelle anzeigt.

| '1985' | '1986' | '1987' | '1988' | '1989' |
|----------|----------|------------|---------|-----------|
| 559132,5 | 539497,2 | -2036841,3 | 1081361 | 1027003,1 |

1 Zeile ausgewählt



10. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass die Beträge innerhalb der einzelnen Jahre nach Quartalen aufgeteilt werden.

| QUARTAL | '1985' | '1986' | '1987' | '1988' | '1989' |
|---------|----------|----------|------------|----------|----------|
| 1 | 32204,8 | 985,2 | 2981,1 | 176852 | 9777,1 |
| 3 | -11792,8 | -71935,3 | 191697,3 | 282848 | 681185,9 |
| 2 | 151841,1 | 53654,8 | -2174503,9 | 430097,2 | 223402,7 |
| 4 | 386879,4 | 556792,5 | -57015,8 | 191563,8 | 112637,4 |

4 Zeilen ausgewählt



11. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass eine Summenzeile, unterhalb der Pivottabelle angezeigt wird.



| QUARTAL | '1985' | '1986' | '1987' | '1988' | '1989' |
|---------|----------|----------|------------|----------|-----------|
| 1 | 32204,8 | 985,2 | 2981,1 | 176852 | 9777,1 |
| 2 | 151841,1 | 53654,8 | -2174503,9 | 430097,2 | 223402,7 |
| 3 | -11792,8 | -71935,3 | 191697,3 | 282848 | 681185,9 |
| 4 | 386879,4 | 556792,5 | -57015,8 | 191563,8 | 112637,4 |
| Summe | 559132,5 | 539497,2 | -2036841,3 | 1081361 | 1027003,1 |

5 Zeilen ausgewählt

6.8. Lösungen - Unterabfragen

1. Schreiben Sie eine Abfrage, die für alle Eigenkunden, die keinen Berater haben (die nicht in der Tabelle EIGENKUNDEMITARBEITER enthalten sind), den Vor- und den Nachnamen anzeigt.

- Lösen Sie die Aufgabe mit Hilfe des **EXISTS**-Operators!
- Lösen Sie die Aufgabe mit Hilfe des **IN**-Operators!

```
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
WHERE NOT EXISTS (SELECT 1
                   FROM EigenkundeMitarbeiter ekm
                   WHERE ekm.Kunden_ID = ek.Kunden_ID);

SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
WHERE ek.Kunden_ID NOT IN (SELECT Kunden_ID
                           FROM EigenkundeMitarbeiter);
```



2. Schreiben Sie eine Abfrage, die anzeigt, ob es Mitarbeiter gibt, die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (arbeiten in keiner Filiale) und Filialleiter.

- Lösen Sie die Aufgabe mit Hilfe des **EXISTS**-Operators!
- Lösen Sie die Aufgabe mit Hilfe des **IN**-Operators!

```
SELECT m.Vorname, m.Nachname
FROM Mitarbeiter m
INNER JOIN Mitarbeiter m1 ON (m.Vorgesetzter_ID = m1.Mitarbeiter_ID)
INNER JOIN Bankfiliale b ON (m1.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE NOT EXISTS (SELECT 1
                   FROM EigenkundeMitarbeiter ekm
                   WHERE m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
ORDER BY m.Vorname;

SELECT m.Vorname, m.Nachname
FROM Mitarbeiter m
INNER JOIN Mitarbeiter m1 ON (m.Vorgesetzter_ID = m1.Mitarbeiter_ID)
INNER JOIN Bankfiliale b ON (m1.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE m.Mitarbeiter_ID NOT IN (SELECT Mitarbeiter_ID
                               FROM EigenkundeMitarbeiter ekm)
ORDER BY m.Vorname;
```



3. Schreiben Sie eine Abfrage, die den häufigsten Vornamen der Bankmitarbeiter anzeigt und wie oft dieser in der Tabelle MITARBEITER vorkommt.



```
SELECT Vorname, Anzahl
FROM (SELECT Vorname, COUNT(Vorname) AS Anzahl
      FROM Mitarbeiter
      GROUP BY Vorname
      ORDER BY COUNT(Vorname) DESC
     )
WHERE rownum = 1;
```



```
SELECT TOP 1 Vorname, COUNT(Vorname) AS Anzahl
FROM Mitarbeiter
GROUP BY Vorname
ORDER BY COUNT(Vorname) DESC;
```

4. Schreiben Sie eine Abfrage, welche die drei Eigenkunden mit den niedrigsten Guthaben auf den Girokonten anzeigt.



```
SELECT *
FROM (SELECT Vorname, Nachname, Guthaben
      FROM Kunde k INNER JOIN EigenkundeKonto ekk
            ON (k.Kunden_ID = ekk.Kunden_ID)
            INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
      ORDER BY Guthaben)
WHERE rownum <= 3;
```



```
SELECT TOP 3 Vorname, Nachname, Guthaben
FROM Kunde k INNER JOIN EigenkundeKonto ekk
      ON (k.Kunden_ID = ekk.Kunden_ID)
      INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
ORDER BY Guthaben;
```

5. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass die drei Eigenkunden mit dem niedrigsten Guthaben (Girokonto + Sparbuch) angezeigt werden. Es müssen auch diejenigen Kunden angezeigt werden, die nur ein Girokonto oder nur ein Sparbuch haben!



```
SELECT *
FROM (SELECT k.Vorname, k.Nachname, SUM(Guthaben)
      FROM (SELECT Kunden_ID, Guthaben
            FROM EigenkundeKonto ekk INNER JOIN Girokonto g
              ON (ekk.Konto_ID = g.Konto_ID)
            UNION
            SELECT Kunden_ID, Guthaben
            FROM EigenkundeKonto ekk INNER JOIN Sparbuch s
              ON (ekk.Konto_ID = s.Konto_ID)) gut
      INNER JOIN Kunde k
        ON (k.Kunden_ID = gut.Kunden_ID)
      GROUP BY k.Kunden_ID, k.Vorname, k.Nachname
      ORDER BY 3)
WHERE rownum <= 3;
```



```
SELECT TOP 3 k.Vorname, k.Nachname, SUM(Guthaben)
FROM (SELECT Kunden_ID, Guthaben
      FROM EigenkundeKonto ekk INNER JOIN Girokonto g
        ON (ekk.Konto_ID = g.Konto_ID)
      UNION
      SELECT Kunden_ID, Guthaben
      FROM EigenkundeKonto ekk INNER JOIN Sparbuch s
        ON (ekk.Konto_ID = s.Konto_ID)) gut
      INNER JOIN Kunde k ON (k.Kunden_ID = gut.Kunden_ID)
GROUP BY k.Kunden_ID, k.Vorname, k.Nachname
ORDER BY 3;
```

6. Schreiben Sie eine Abfrage, die alle Eigenkunden anzeigt, welche im Jahr 1985 keine Buchungen verursacht haben.



```
SELECT Vorname, Nachname
FROM Kunde k INNER JOIN EigenkundeKonto ekk
      ON (k.Kunden_ID = ekk.Kunden_ID)
WHERE NOT EXISTS (SELECT 1
                  FROM Buchung b INNER JOIN EigenkundeKonto ekk1
                        ON (b.Konto_ID = ekk1.Konto_ID)
                  WHERE ekk1.Kunden_ID = ekk.Kunden_ID
                  AND Buchungsdatum BETWEEN
                        TO_DATE('01.01.1985') AND
                        TO_DATE('31.12.1985'))
GROUP BY k.Kunden_ID, Vorname, Nachname
ORDER BY Nachname, Vorname;
```



```
SELECT DISTINCT Vorname, Nachname
FROM Kunde k INNER JOIN EigenkundeKonto ekk
      ON (k.Kunden_ID = ekk.Kunden_ID)
WHERE k.Kunden_ID NOT IN (SELECT ek.kunden_id
                        FROM Buchung b1 INNER JOIN EigenkundeKonto ek
                              ON (b1.Konto_ID = ek.Konto_ID)
                        WHERE b1.Buchungsdatum BETWEEN
                              CONVERT(DATETIME2, '01.01.1985', 104) AND
                              CONVERT(DATETIME2, '31.12.1985', 104))
ORDER BY Nachname, Vorname;
```

7. Schreiben Sie eine Abfrage, die für jede Bankfiliale den Mitarbeiter mit dem höchsten Gehalt ausgibt.



```
SELECT b.Strasse || ' ' || b.Hausnummer || ' ' || b.PLZ || ' ' ||
      b.Ort AS Bankfiliale,
      m.Vorname, m.Nachname, m.Gehalt
FROM Mitarbeiter m INNER JOIN Bankfiliale b
      ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE Gehalt = (SELECT MAX(Gehalt)
                FROM Mitarbeiter m1
                WHERE m.Bankfiliale_ID = m1.Bankfiliale_ID);
```




```

SELECT b.Strasse + ' ' + b.Hausnummer + ' ' + b.PLZ + ' ' +
       b.Ort AS Bankfiliale,
       m.Vorname, m.Nachname, m.Gehalt
FROM   Mitarbeiter m INNER JOIN Bankfiliale b
       ON (m.Bankfiliale_ID = b.Bankfiliale_ID)
WHERE  Gehalt = (SELECT MAX(Gehalt)
                FROM   Mitarbeiter m1
                WHERE  m.Bankfiliale_ID = m1.Bankfiliale_ID);

```

8. Schreiben Sie eine Abfrage, die für jeden Wohnort (EIGENKUNDE.ORT) den Kunden anzeigt, der im Jahr 1987 das höchste Einkommen hatte (Das Einkommen ist die Summe aller Beträge eines Kunden, in der Tabelle BUCHUNG). Sortieren Sie die Abfrage nach den Wohnorten.



```

SELECT b1.Ort, b1.Vorname, b1.Nachname, b1.betrag
FROM   (SELECT ek.Kunden_ID, ek.Ort, k.Vorname, k.Nachname,
              SUM(Betrag) AS betrag
        FROM   EigenkundeKonto ekk INNER JOIN Buchung b
              ON (ekk.Konto_ID = b.Konto_ID)
        INNER JOIN Eigenkunde ek
              ON (ekk.Kunden_ID = ek.Kunden_ID)
        INNER JOIN Kunde k
              ON (k.Kunden_ID = ek.Kunden_ID)
        WHERE  Buchungsdatum BETWEEN
              TO_DATE('01.01.1987', 'DD.MM.YYYY') AND
              TO_DATE('31.12.1987', 'DD.MM.YYYY')
        GROUP BY ek.Kunden_ID, ek.Ort, k.Vorname, k.Nachname) b1
WHERE  betrag = (SELECT MAX(betrag)
                FROM   (SELECT ek.Kunden_ID, ek.Ort, SUM(Betrag) AS betrag
                        FROM   EigenkundeKonto ekk INNER JOIN Buchung b
                              ON (ekk.Konto_ID = b.Konto_ID)
                        INNER JOIN Eigenkunde ek
                              ON (ekk.Kunden_ID = ek.Kunden_ID)
                        WHERE  Buchungsdatum BETWEEN
                              TO_DATE('01.01.1987', 'DD.MM.YYYY') AND
                              TO_DATE('31.12.1987', 'DD.MM.YYYY')
                        GROUP BY ek.Kunden_ID, ek.Ort) b2
                WHERE  b1.Ort = b2.Ort)
ORDER BY b1.Ort;

```



```

SELECT b1.Ort, b1.Vorname, b1.Nachname, b1.betrag
FROM (SELECT ek.Kunden_ID, ek.Ort, k.Vorname, k.Nachname,
           SUM(Betrag) AS betrag
      FROM EigenkundeKonto ekk INNER JOIN Buchung b
           ON (ekk.Konto_ID = b.Konto_ID)
      INNER JOIN Eigenkunde ek
           ON (ekk.Kunden_ID = ek.Kunden_ID)
      INNER JOIN Kunde k
           ON (k.Kunden_ID = ek.Kunden_ID)
     WHERE Buchungsdatum BETWEEN
           CONVERT(DATETIME2, '01.01.1987', 104) AND
           CONVERT(DATETIME2, '31.12.1987', 104)
     GROUP BY ek.Kunden_ID, ek.Ort, k.Vorname, k.Nachname) b1
WHERE betrag = (SELECT MAX(betrag)
               FROM (SELECT ek.Kunden_ID, ek.Ort, SUM(Betrag) AS betrag
                     FROM EigenkundeKonto ekk INNER JOIN Buchung b
                          ON (ekk.Konto_ID = b.Konto_ID)
                     INNER JOIN Eigenkunde ek
                          ON (ekk.Kunden_ID = ek.Kunden_ID)
                     WHERE Buchungsdatum BETWEEN
                           CONVERT(DATETIME2, '01.01.1987', 104) AND
                           CONVERT(DATETIME2, '31.12.1987', 104)
                     GROUP BY ek.Kunden_ID, ek.Ort) b2
              WHERE b1.Ort = b2.Ort)
ORDER BY b1.Ort;

```

9. Erstellen Sie eine Abfrage, die die Umsätze der Bank (SUM(Buchung.Betrag)) für die Jahre 1985 bis einschließlich 1989 als Pivottabelle anzeigt.



```

SELECT *
FROM (SELECT TO_CHAR(Buchungsdatum, 'YYYY') AS Datum, Betrag
      FROM Buchung)
PIVOT (SUM(Betrag) FOR Datum IN ('1985', '1986', '1987', '1988', '1989'));

```



```

SELECT *
FROM (SELECT DATEPART(YEAR, Buchungsdatum) AS Datum, Betrag
      FROM Buchung) AS Sourcetable
PIVOT (SUM(Betrag)
      FOR Datum IN ([1985], [1986], [1987], [1988], [1989])) AS Pivottable;

```

10. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass die Beträge innerhalb der einzelnen Jahre nach Quartalen aufgeteilt werden.

```
SELECT *
FROM (SELECT TO_CHAR(Buchungsdatum, 'Q') AS Quartal,
              TO_CHAR(Buchungsdatum, 'YYYY') AS Datum, Betrag
      FROM Buchung)
PIVOT (SUM(Betrag) FOR Datum IN ('1985', '1986', '1987', '1988', '1989'));
```



```
SELECT *
FROM (SELECT DATENAME(QUARTER, Buchungsdatum) AS Quartal,
              DATENAME(YEAR, Buchungsdatum) AS Datum, Betrag
      FROM Buchung) AS Sourcetable
PIVOT (SUM(Betrag)
      FOR Datum IN ([1985], [1986], [1987], [1988], [1989])) AS Pivottable;
```



11. Verändern Sie die Abfrage aus der vorangegangenen Aufgabe so, dass eine Summenzeile, unterhalb der Pivottabelle angezeigt wird.

```
SELECT *
FROM (SELECT TO_CHAR(Buchungsdatum, 'Q') AS Quartal,
              TO_CHAR(Buchungsdatum, 'YYYY') AS Datum, Betrag
      FROM Buchung)
PIVOT (SUM(Betrag) FOR Datum IN ('1985', '1986', '1987', '1988', '1989'))
UNION ALL
SELECT *
FROM (SELECT 'Summe' AS Quartal, TO_CHAR(Buchungsdatum, 'YYYY') AS Datum,
              Betrag
      FROM Buchung)
PIVOT (SUM(Betrag) FOR Datum IN ('1985', '1986', '1987', '1988', '1989'));
```





```
SELECT *
FROM (SELECT DATENAME(QUARTER, Buchungsdatum) AS Quartal,
              DATENAME(YEAR, Buchungsdatum) AS Datum, Betrag
       FROM Buchung) AS Sourcetable
PIVOT (SUM(Betrag) FOR Datum IN ([1985], [1986], [1987], [1988], [1989]))
      AS Pivottable
UNION ALL
SELECT *
FROM (SELECT 'Summe' AS Quartal, DATENAME(YEAR, Buchungsdatum) AS Datum,
              Betrag
       FROM Buchung) AS Sourcetable
PIVOT (SUM(Betrag) FOR Datum IN ([1985], [1986], [1987], [1988], [1989]))
      AS Pivottable;
```

7. Data Manipulation Language (DML)

Inhaltsangabe

| | |
|--|------------|
| 7.1 Die DML-Anweisungen | 7-2 |
| 7.1.1 Datensätze einfügen - Die INSERT-Anweisung | 7-2 |
| 7.1.2 Datensätze ändern - Die UPDATE-Anweisung | 7-5 |
| 7.1.3 Datensätze löschen - Die DELETE-Anweisung | 7-8 |
| 7.2 Das Transaktionskonzept - COMMIT und ROLLBACK | 7-9 |
| 7.2.1 Beginn und Ende einer Transaktion | 7-10 |
| 7.2.2 Eine Transaktion erfolgreich abschließen | 7-11 |
| 7.2.3 Eine Transaktion rückgängig machen | 7-14 |

In den vergangenen Kapiteln wurde bisher nur der Teil von SQL beschrieben, der als sog. „Query language“ bezeichnet wird. Hier wird jetzt gezeigt, wie vorhandene Daten manipuliert werden können. Der dafür zuständige Teil von SQL heißt: „Data Manipulation Language“ oder kurz „DML“.

Gemäß SQL-Standard besteht DML aus drei Befehlen:

- **INSERT**: Daten einfügen.
- **UPDATE**: Daten ändern.
- **DELETE**: Daten löschen.

7.1. Die DML-Anweisungen

7.1.1. Datensätze einfügen - Die INSERT-Anweisung

Mit Hilfe der **INSERT**-Anweisung werden neue Datensätze an eine Tabelle angefügt. Die Syntax für ein einfaches **INSERT** lautet:

Listing 7.1: Die INSERT Anweisungen

```
INSERT INTO <Tabelle> (<Spalte 1>, <Spalte 2>, ..., <Spalte n>)
VALUES (<Wert 1>, <Wert 2>, ..., <Wert n>);
```

Tabelle 7.1.: Die INSERT-Anweisung

| Ausdruck | Bedeutung |
|------------------------------|---|
| INSERT INTO <Tabelle> | An dieser Stelle steht der Name der Tabelle oder View, in die der Datensatz eingefügt werden soll. |
| <Spalte 1>, <Spalte 2>, ... | Dies ist die Spaltenliste. Hier können alle Spalten angegeben werden, in die Daten eingefügt werden. Die Spaltenliste ist optional. |
| VALUES <Wert 1>, ... | Dies ist die Werteliste. Hier werden alle Werte aufgeführt, die in <Tabelle> eingefügt werden sollen. Statt einem festen Wert, kann an jeder Stelle auch ein Ausdruck stehen, der einen Wert erzeugt (z. B. eine Funktion). |

Beispiel 7.2 demonstriert die einfachste Form eines **INSERT**-Statements: Es wird eine neue Zeile in die Tabelle BANKFILIALE eingefügt.

Listing 7.2: Ein einfaches INSERT

```
INSERT INTO Bankfiliale (Bankfiliale_ID, Strasse, Hausnummer, PLZ, Ort)
VALUES (22, 'Rosenweg', '14a', '06425', 'Ploetzkau');
```

In obigem Beispiel wird der Wert „22“ in die Spalte BANKFILIALE_ID, der Wert „Rosenweg“ in die Spalte STRASSE eingefügt. Die restlichen drei Werte werden in die Spalten HAUSNUMMER, PLZ und ORT geschrieben. Die Spaltenliste der **INSERT**-Anweisung muss die einzelnen Spalten keineswegs in der Reihenfolge enthalten, wie sie in der Tabelle enthalten sind.

Listing 7.3: Ein einfaches INSERT

```
INSERT INTO Bankfiliale (Strasse, Hausnummer, PLZ, Ort, Bankfiliale_ID)
VALUES ('Rosenweg', '14a', '06425', 'Ploetzkau', 22);
```

In der Spaltenliste müssen die Spalten nicht in der Reihenfolge aufgeführt werden, wie sie in der Tabelle vorkommen. Die Reihenfolge in der Spaltenliste ist beliebig!



Wie in [Tabelle 7.1](#) bereits beschrieben, ist die Spaltenliste hinter den **INSERT INTO**-Schlüsselwörtern optional. Daraus folgt, dass sich [Beispiel 7.2](#) auch so schreiben lässt:

Listing 7.4: Ein einfaches INSERT ohne Spaltenliste

```
INSERT INTO Bankfiliale
VALUES (22, 'Rosenweg', '14a', '06425', 'Ploetzkau');
```

Das **INSERT**-Statement in [Beispiel 7.4](#) wird vom DBMS so interpretiert, dass der erste Wert in die erste Spalte, der zweite Wert in die zweite Spalte, der dritte Wert in die dritte Spalte, usw. eingefügt wird.

Die INSERT-Anweisung und NULL-Werte

Soll mit einer **INSERT**-Anweisung ein NULL-Wert in eine Tabellenspalte eingefügt werden, geschieht dies mit Hilfe des Schlüsselwortes **NULL**. In [Beispiel 7.5](#) wird eine neue Zeile in die Tabelle BANK eingefügt. Während des Einfügevorgangs ist der Wert für die Spalte RATING noch nicht bekannt. Die Zeile soll nun ohne diesen Wert eingefügt werden.

Listing 7.5: Ein einfaches INSERT mit NULL-Werten

```
INSERT INTO Bank
VALUES (21, 'KRDCU21SES', 'Lokki Bank of Cyprus', 'Steuerparadies', '42',
        '01067', 'Berlin', NULL);
```

Mit Hilfe des Schlüsselwortes **NULL** kann ein **NULL**-Wert in eine Tabellenspalte eingefügt werden.



Standardwerte

Standardwerte werden meist dann genutzt, wenn in eine Spalte häufig der gleiche Wert eingefügt werden muss. Sie müssen bei der Erstellung einer Tabelle mit definiert werden. Ein Beispiel hierfür könnte die Spalte BUCHUNGSDATUM der Tabelle BUCHUNG sein. Wird eine neue Buchung erfasst, muss immer das aktuelle Tagesdatum eingetragen werden. Diese kann durch die Funktion **SYSDATE** (Oracle) bzw. **GETDATE** (SQL Server) erzeugt werden.

Beispiel 7.6 zeigt, wie in die Spalte BUCHUNGSDATUM das aktuelle Datum, als Standardwert eingefügt wird.

Listing 7.6: Einfügen eines Standardwertes

```
INSERT INTO Buchung (Buchungs_ID, Betrag, Buchungsdatum, Konto_ID,
                     Transaktions_ID)
VALUES (500000, 300.20, DEFAULT, 1, 666666);
```



Soll in eine Tabellenspalte deren Standardwert eingefügt werden, muss das Schlüsselwort **DEFAULT** benutzt werden.

Die INSERT-Anweisung und Unterabfragen

Die **INSERT**-Anweisung ist in der Lage eine Unterabfrage zu verwenden, um den Inhalt einer Tabelle in eine andere Tabelle einzufügen. Dies kann z. B. das Kopieren eines Datensatzes in eine Tabelle gleicher Struktur sein oder das Abfragen einzelner Attribute, um diese für die Berechnung neuer Werte zu nutzen. Die Syntax für die **INSERT**-Anweisung mit Unterabfrage lautet:

Listing 7.7: Die **INSERT**-Anweisung mit Unterabfrage

```
INSERT INTO <Tabelle> (<Spalte 1>, <Spalte 2>, ..., <Spalte n>)
<Unterabfrage>;
```

Das **INSERT**-Statement kann eine beliebig komplexe Unterabfrage, wie in **Abschnitt 6** beschrieben, verwenden. **Beispiel 7.8** zeigt, wie ein Datensatz aus der Tabelle MITARBEITER in die strukturgleiche Tabelle AUSGESCHIEDEN kopiert wird.

Listing 7.8: Die **INSERT**-Anweisung mit Unterabfrage

```
-- Erstellen der Tabelle in Oracle
CREATE TABLE Ausgeschieden
AS
  SELECT *
  FROM   Mitarbeiter
 WHERE  1 = 2;
```



```
-- Erstellen der Tabelle in SQL Server
SELECT *
INTO    Ausgeschieden
FROM    Mitarbeiter
WHERE   1 = 2;

INSERT INTO Ausgeschieden
SELECT *
FROM    Mitarbeiter
WHERE   Mitarbeiter_ID = 70;
```

Der Datensatz des Mitarbeiters Nummer 70 wird in die Tabelle AUSGESCHIEDEN kopiert.

7.1.2. Datensätze ändern - Die UPDATE-Anweisung

Die **UPDATE**-Anweisung repräsentiert den Teil von DML der es ermöglicht, bestehende Datensätze zu verändern. Die Syntax von **UPDATE** lautet:

Listing 7.9: Die Syntax des **UPDATE**-Kommandos

```
UPDATE <Tabelle>
SET    <Spalte 1> = <Wert>,
      <Spalte 2> = <Wert>,
      ...
      <Spalte n> = <Wert>
[WHERE <Where-Klausel>];
```

Tabelle 7.2.: Die UPDATE-Anweisung

| Ausdruck | Bedeutung |
|-------------------------|---|
| UPDATE <Tabelle> | An dieser Stelle steht der Name der Tabelle oder View, in der ein Datensatz verändert werden soll. |
| SET <Spalte 1> = <Wert> | Die SET-Anweisung gibt die Spalten an, deren aktueller Wert durch den neuen Wert <Wert> ersetzt werden soll. Hier können mehrere „Spalte = Wert“-Paare, durch Komma getrennt, stehen. |
| WHERE <Where-Klausel> | Optionale WHERE-Klausel, die den Umfang der Datensätze, die geändert werden sollen, einschränkt. |

Eine genauso einfache, wie auch gefährliche Form der **UPDATE**-Anweisung, ist in [Beispiel 7.10](#) zu sehen.

Listing 7.10: Ein gefährliches **UPDATE**

```
UPDATE Mitarbeiter
SET    Gehalt = Gehalt * 1.035;
```

Die Gefahr bei dieser **UPDATE**-Anweisung besteht darin, dass die Angabe einer einschränkenden **WHERE**-Klausel fehlt. Das DBMS wird in diesem Falle alle Datensätze der Tabelle MITARBEITER verändern und nicht nur eine bestimmte Gruppe.

Soll nur das Gehalt des Mitarbeiters *Max Winter* geändert werden, muss das **UPDATE**-Statement um eine **WHERE**-Klausel erweitert werden:

Listing 7.11: Ein korrektes **UPDATE**

```
UPDATE Mitarbeiter
SET    Gehalt = Gehalt * 1.035
WHERE  Mitarbeiter_ID = 1;
```

Wie in [Tabelle 7.2](#) zu sehen ist, können auch mehrere Spalten eines Datensatzes gleichzeitig geändert werden. In [Beispiel 7.12](#) wird die Mitarbeiterin „Lena Hermann“ (Mitarbeiter_ID 40) von Filiale 4 nach Filiale 8 versetzt und gleichzeitig wird ihre Provision von 20 % auf 30 % erhöht.

Listing 7.12: Ein korrektes **UPDATE** mehrerer Spalten

```
UPDATE Mitarbeiter
SET    Bankfiliale_ID = 8,
        Provision = 30
WHERE  Mitarbeiter_ID = 40;
```

Wo Licht ist, da ist aber immer auch Schatten. Wenn bei einem Mitarbeiter die Provision erhöht wird, muss sie bei einem anderen gekürzt oder gestrichen werden. Der Mitarbeiter „Lukas Weiß“ hat im vergangenen Geschäftsjahr ein sehr schlechtes Ergebnis erzielt, weshalb ihm die Provision gestrichen wird. Dies geschieht, indem die Spalte PROVISION mit einem **NULL**-Wert gefüllt wird.

Listing 7.13: Da geht sie hin, die Provision

```
UPDATE Mitarbeiter
SET    Provision = NULL
WHERE  Mitarbeiter_ID = 38;
```

Nicht nur NULL-Werte, auch Standardwerte können innerhalb eines **UPDATE**-Statements genutzt werden.

Listing 7.14: Ein **UPDATE** mit Standardwert

```
UPDATE Mitarbeiter
SET    Gehalt = DEFAULT
WHERE  Mitarbeiter_ID = 82;
```

[Beispiel 7.14](#) geht davon aus, dass für die Spalte GEHALT ein Standardwert von „1500“ festgelegt worden ist.

UPDATE mit Unterabfrage

Wie bei der **INSERT**-Anweisung, kann auch bei der **UPDATE**-Anweisung eine Unterabfrage genutzt werden. Diese kann an zwei Stellen stehen: In der **SET**-Klausel und in der **WHERE**-Klausel. Hierzu einige Beispiele.

Das Gehalt des Mitarbeiters „Jannis Friedrich“ soll geändert werden. Das neue Gehalt muss 20 % des Gehalts seines unmittelbaren Vorgesetzten betragen.

Listing 7.15: **UPDATE** mit Unterabfrage

```
UPDATE Mitarbeiter
SET    Gehalt = (SELECT v.Gehalt
                  FROM    Mitarbeiter m INNER JOIN Mitarbeiter v
                        ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
                  WHERE   m.Mitarbeiter_ID = 79) * 0.20
WHERE  Mitarbeiter_ID = 79;
```

Mit Hilfe der folgenden **SELECT**-Anweisung kann die Korrektheit des **UPDATE**-Statements aus [Beispiel 7.15](#) nachgewiesen werden.

Listing 7.16: Der Beweis

```
SELECT Mitarbeiter_ID, Vorname, Nachname, Gehalt
FROM   Mitarbeiter
WHERE  Mitarbeiter_ID IN (79, 21);
```

In [Beispiel 7.15](#) wird die Mitarbeiter_ID 79 an zwei Stellen angegeben. Durch eine Veränderung des **UPDATE**-Statements kann dies auf eine Angabe reduziert werden.

Listing 7.17: **UPDATE** mit korrelierter Unterabfrage in Oracle

```
UPDATE Mitarbeiter m1
SET    Gehalt = (SELECT v.Gehalt
                  FROM    Mitarbeiter m INNER JOIN Mitarbeiter v
                        ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
                  WHERE   m.Mitarbeiter_ID = m1.Mitarbeiter_ID)
WHERE  m1.Mitarbeiter_ID = 79;
```

In der **UPDATE**-Klausel wird ein Alias für die Tabelle MITARBEITER festgelegt. Diesen Alias benutzt die Unterabfrage, um auf die Werte des äußeren Statements, des **UPDATE**-Statements, zuzugreifen. Dadurch genügt es, wenn die Mitarbeiter_ID nur einmal gesetzt wird. Im MS SQL Server muss der Alias für die Tabelle MITARBEITER über eine **FROM**-Klausel definiert werden, so dass sich [Beispiel 7.17](#) wie folgt ändert:

Listing 7.18: **UPDATE** mit korrelierter Unterabfrage im MS SQL Server

```
UPDATE m1
SET    Gehalt = (SELECT v.Gehalt
                FROM    Mitarbeiter m INNER JOIN Mitarbeiter v
                        ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
                WHERE   m.Mitarbeiter_ID = m1.Mitarbeiter_ID)
FROM    Mitarbeiter m1
WHERE   m1.Mitarbeiter_ID = 79;
```

„Was des einen Freud ist, ist des andern Leid“. Dieser Grundsatz trifft auch bei der Gehaltserhöhung für Herrn Friedrich zu. Da er nun 400 EUR mehr Gehalt bekommt, müssen bei den anderen Angestellten dementsprechende Einsparungen vorgenommen werden. Für alle Mitarbeiter der Filiale 14, mit Ausnahme von Herrn Friedrich, muss das Gehalt um 2 % gekürzt werden.

Listing 7.19: Gehaltskürzung für eine ganze Filiale

```
UPDATE Mitarbeiter
SET    Gehalt = Gehalt * 0.98
WHERE  Mitarbeiter_ID IN (SELECT Mitarbeiter_ID
                        FROM    Mitarbeiter
                        WHERE   Bankfiliale_ID = 14
                        AND     Mitarbeiter_ID <> 79);
```

7.1.3. Datensätze löschen - Die DELETE-Anweisung

Die dritte und letzte der DML-Anweisungen, ist die **DELETE**-Anweisung. Sie ermöglicht es, Datensätze zu löschen. Die Syntax der **DELETE**-Anweisung lautet wie folgt:

Listing 7.20: Die **DELETE**-Anweisung

```
DELETE FROM <Tabelle>
WHERE <Where-Klausel>;
```

Tabelle 7.3.: Die DELETE-Anweisung

| Ausdruck | Bedeutung |
|-----------------------|---|
| DELETE <Tabelle> | An dieser Stelle steht der Name der Tabelle oder View, aus der Datensätze gelöscht werden sollen. |
| WHERE <Where-Klausel> | Optionale WHERE-Klausel, die den Umfang der Datensätze begrenzt, die gelöscht werden sollen. |

Ähnlich wie bei der **UPDATE**-Anweisung, gibt es auch bei der **DELETE**-Anweisung eine kleine Falle. [Beispiel 7.21](#) zeigt, wie man mit einer sehr einfachen **DELETE**-Anweisung in große Schwierigkeiten geraten kann.

Listing 7.21: Eine tödliche **DELETE**-Anweisung

```
DELETE FROM Buchung;
```

Die Auswirkungen der **DELETE**-Anweisung aus [Beispiel 7.21](#) sind einfach und kurz erklärt. Es werden alle Datensätze aus der Tabelle BUCHUNG gelöscht. Des Rätsels Lösung ist die gleiche wie beim **UPDATE**-Statement: Es fehlt die einschränkende **WHERE**-Klausel. Um beispielsweise nur eine einzelne Buchung zu löschen ist folgende Modifikation notwendig:

Listing 7.22: Schon viel besser!!!

```
DELETE FROM Buchung  
WHERE Transaktions_ID = 345;
```

DELETE mit Unterabfrage

Auch in der **DELETE**-Anweisung kann eine Unterabfrage genutzt werden. Hierzu ein einfaches Beispiel: Da die Bankfiliale, in der Poststraße, in Aschersleben aufgelöst wird, müssen leider auch die dort beschäftigten Mitarbeiter wieder dem Arbeitsmarkt zur Verfügung gestellt werden.

Listing 7.23: **DELETE** mit Unterabfrage

```
DELETE FROM Mitarbeiter  
WHERE Bankfiliale_ID = (SELECT Bankfiliale_ID  
                        FROM Bankfiliale  
                        WHERE LOWER(Strasse) LIKE 'poststrasse'  
                        AND PLZ = '06449');
```

7.2. Das Transaktionskonzept - COMMIT und ROLLBACK

Die Datenbankmanagementsysteme Oracle und SQL Server sind beides transaktionsbasierte DBMS. Das bedeutet, dass alle DML-Anweisungen innerhalb einer Transaktion ablaufen. Die Frage die sich dabei stellt ist: „Was ist eine Transaktion?“ Der Begriff Transaktion ist dem spätlateinischen „transagere = Überführen, Übertragen“ entliehen und den meisten Leuten aus dem Finanzbereich bekannt. Man denke einfach an die Überweisung eines Betrags von Konto A auf Konto B. Der vereinfachte Ablauf einer solchen Finanztransaktion könnte wie folgt aussehen:

1. Kontoinhaber A füllt einen Überweisungsträger aus. Damit beginnt die Transaktion.
2. Die Bank des Kontoinhabers A zieht den Überweisungsbetrag von seinem Konto ab und übermittelt die Informationen bezüglich der Überweisung an Bank B.

3. Bank B schreibt den Betrag auf dem Konto von Kontoinhaber B gut.
4. Der Vorgang wird in einem Journal protokolliert. Damit ist die Überweisung abgeschlossen.

Warum aber der Begriff der Transaktion? Die Antwort auf diese Frage hängt eng mit der Antwort auf eine andere Frage zusammen: „Was wäre wenn, nach der Abbuchung von Konto A der Vorgang unterbrochen würde?“ In so einem Falle ist das gewohnte Verhalten, das alle bisher gemachten Schritte wieder rückgängig gemacht werden, d. h. der abgebuchte Betrag muss wieder auf das Konto von A zurückgebucht werden. Würde dies nicht geschehen, wäre das Geld von A verschwunden.

Das Rückgängigmachen aller bisher gemachten Aktionen ist aber nur dann möglich, wenn

- genau bekannt ist, welche Aktionen zusammengehören und
- in welcher Reihenfolge sie stattgefunden haben.

Deshalb werden alle Aktionen in einer größeren Einheit, der Transaktion, zusammengefaßt. Es muss also im Ernstfall nur ermittelt werden, zu welcher Transaktion die letzte Aktion gehörte um alle Vorgängeraktionen ermitteln zu können.



Definition *Transaktion*: Eine Transaktionen ist eine logische Arbeitseinheit, die einen oder mehrere Arbeitsschritte enthält. Transaktionen sind in sich geschlossene Einheiten. Die Ergebnisse aller Arbeitsschritte einer Transaktion können entweder übernommen oder rückgängig gemacht werden.

Dieses Konzept lässt sich auch auf Datenbanken übertragen. Werden mehrere zusammengehörende SQL-Anweisungen ausgeführt, muss auch gewährleistet werden, dass entweder alle erfolgreich beendet werden oder aber alle rückgängig gemacht werden.

7.2.1. Beginn und Ende einer Transaktion

Wann beginnt eine Transaktion?

In Oracle startet eine Transaktion:

- Implizit bei jedem ersten DML-Statement.
- Explizit durch die Anweisung `SET TRANSACTION`.

In MS SQL Server startet eine Transaktion:

- Wenn der implizite Transaktionsmodus aktiviert wurde, bei jedem ersten DML-Statement.
- Explizit durch die Anweisung **BEGIN TRANSACTION**

Das Standardverhalten in SQL Server ist, dass jedes einzelne DML-Statement als eigene Transaktion abgehandelt wird. Zur Aktivierung des impliziten Transaktionsmodus muss die SQL-Anweisung **SET IMPLICIT_TRANSACTIONS ON** abgesetzt werden.



Wann endet eine Transaktion?

Eine Transaktion kann an zwei verschiedenen Punkten enden:

- Sie wird erfolgreich abgeschlossen.
- Sie wird manuell rückgängig gemacht.

7.2.2. Eine Transaktion erfolgreich abschließen

Das COMMIT-Kommando

Wenn alle Statements einer Transaktion erfolgreich verlaufen sind, muss die Transaktion beendet werden, um die gemachten Änderungen dauerhaft in der Datenbank zu speichern. Dies geschieht in Oracle mit Hilfe des Kommandos **COMMIT**. Wird eine Transaktion nicht mit **COMMIT** abgeschlossen, werden automatisch alle unbestätigten Änderungen rückgängig gemacht. [Beispiel 7.24](#) ff. zeigen dieses Verhalten.

Listing 7.24: Eine Transaktion wird abgebrochen

```
SELECT Bank_ID, BIC, Name
FROM   Bank
WHERE  Bank_ID >= 18;
```



| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |

3 Zeilen ausgewählt

```
INSERT INTO Bank
VALUES (21, 'NOSDEL21SES', 'Lokki Bank of Cyprus',
       'Strasse der Europaeischen Union', '3', '00000', 'Pleitingen',
       'D--');
```

```
SELECT Bank_ID, BIC, Name
FROM   Bank
WHERE  Bank_ID >= 18;
```



| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |
| 21 | NOSDEL21SES | Lokki Bank of Cyprus |

4 Zeilen ausgewählt

```
-- An dieser Stelle findet ein Absturz der Client-Anwendung statt
-- und die Anwendung wird neu gestartet.
```

```
SELECT Bank_ID, BIC, Name
FROM   Bank
WHERE  Bank_ID >= 18;
```



| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |

3 Zeilen ausgewählt

Weil vor dem Absturz der Client-Anwendung die Transaktion nicht mit **COMMIT** abgeschlossen wurde, ist die gemachte Änderung wieder verschwunden. Das gleiche Szenario nun noch einmal, aber mit **COMMIT** am Ende.


```

INSERT INTO Bank
VALUES      (21, 'NOSDEL21SES', 'Lokki Bank of Cyprus',
            'Strasse der Europaeischen Union', '3', '00000', 'Pleitingen',
            'D--');

SELECT Bank_ID, BIC, Name
FROM      Bank
WHERE     Bank_ID >= 18;

COMMIT;

```

| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |
| 21 | NOSDEL21SES | Lokki Bank of Cyprus |

4 Zeilen ausgewählt



```

-- An dieser Stelle wird die Client-Anwendung beendet und
-- neugestartet.

SELECT Bank_ID, BIC, Name
FROM      Bank
WHERE     Bank_ID >= 18;

```

| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |
| 21 | NOSDEL21SES | Lokki Bank of Cyprus |

4 Zeilen ausgewählt



Die **COMMIT**-Anweisung persistiert^a die Aktionen einer Transaktion in der Datenbank. Ohne **COMMIT** werden alle Änderungen wieder zurückgerollt.

^apersistent = dauerhaft



COMMIT in Microsoft SQL Server

In Microsoft SQL Server muss dem **COMMIT**-Kommando noch das Schlüsselwort **TRANSACTION** (oder **TRAN**) hinzugefügt werden. Dies beendet sowohl implizite als auch explizite Transaktionen.

Listing 7.25: Eine implizite Transaktion committen

```
SET IMPLICIT_TRANSACTIONS ON
INSERT INTO Bank
VALUES      (21, 'NOSDEL21SES', 'Lokki Bank of Cyprus',
            'Strasse der Europaeischen Union', '3', '00000', 'Pleitingen',
            'D--');

SELECT Bank_ID, BIC, Name
FROM    Bank
WHERE   Bank_ID >= 18;

COMMIT TRAN;
```

Listing 7.26: Eine explizite Transaktion committen

```
BEGIN TRANSACTION
INSERT INTO Bank
VALUES      (21, 'NOSDEL21SES', 'Lokki Bank of Cyprus',
            'Strasse der Europaeischen Union', '3', '00000', 'Pleitingen',
            'D--');

SELECT Bank_ID, BIC, Name
FROM    Bank
WHERE   Bank_ID >= 18;

COMMIT TRAN;
```

7.2.3. Eine Transaktion rückgängig machen

Das ROLLBACK-Kommando

Das Kommando **ROLLBACK** stellt das Gegenstück zu **COMMIT** dar. Sollen die Aktionen einer Transaktion nicht dauerhaft in der Datenbank gespeichert werden, können sie mit **ROLLBACK** zurückgerollt (rückgängig gemacht) werden.

Listing 7.27: Eine Transaktion wird abgebrochen

```
SELECT Bank_ID, BIC, Name
FROM    Bank
WHERE   Bank_ID >= 18;
```

| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |
| 21 | NOSDEL21SES | Lokki Bank of Cyprus |

4 Zeilen ausgewählt



```
DELETE FROM Bank
WHERE Bank_ID = 21;

SELECT Bank_ID, BIC, Name
FROM Bank
WHERE Bank_ID >= 18;
```

| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |

3 Zeilen ausgewählt



```
ROLLBACK;

SELECT Bank_ID, BIC, Name
FROM Bank
WHERE Bank_ID >= 18;
```

| BANK_ID | BIC | NAME |
|---------|-------------|------------------------------|
| 18 | BVXYDE21SES | Bank der Landwirte |
| 19 | BGIODE21SES | Austrailian Bank Association |
| 20 | DFGHDE21SES | South Africa Bank |
| 21 | NOSDEL21SES | Lokki Bank of Cyprus |

4 Zeilen ausgewählt



Die Anweisung **ROLLBACK** rollt alle Aktionen einer Transaktion zurück und beendet sie.



ROLLBACK in Microsoft SQL Server

Genau wie das **COMMIT**-Kommando, muss auch das **ROLLBACK**-Kommando um das Schlüsselwort **TRANSACTION** ergänzt werden.

8. Data Definition Language

Inhaltsangabe

| | | |
|------------|---|-------------|
| 8.1 | Tabellen erstellen und verwalten | 8-2 |
| 8.1.1 | Namenskonventionen und Einschränkungen | 8-2 |
| 8.1.2 | CREATE TABLE - Tabellen erstellen | 8-4 |
| 8.1.3 | CREATE TABLE AS... (CTAS) | 8-5 |
| 8.1.4 | ALTER TABLE - Tabellen verändern | 8-6 |
| 8.1.5 | DROP TABLE - Tabellen löschen | 8-11 |
| 8.1.6 | TRUNCATE TABLE - Tabellen leeren | 8-11 |
| 8.2 | Views erstellen verwalten | 8-13 |
| 8.2.1 | Was sind Views? | 8-13 |
| 8.2.2 | Views erstellen | 8-13 |
| 8.2.3 | Views und DML | 8-18 |
| 8.2.4 | Views ändern | 8-21 |
| 8.2.5 | Views löschen | 8-22 |
| 8.3 | Übungen - Erstellen von Views | 8-23 |
| 8.4 | Lösungen - Erstellen von Views | 8-25 |

Die Data definition language ist der Teil von SQL, der es ermöglicht, Objekte in der Datenbank zu erstellen und zu verwalten. DDL besteht im wesentlichen aus den vier Befehlen:

- **CREATE:** Erstellen von Objekten
- **ALTER:** Ändern von Objekten
- **DROP:** Objekte löschen
- **TRUNCATE:** Leeren von Tabellen.

Der Begriff des „Objekts“ bezieht sich, je nach DBMS, auf die Unterschiedlichsten Dinge:

- Tabellen
- Views
- Indizes
- Sequenzen
- PL/SQL oder T-SQL Prozeduren und Funktionen

...und vieles mehr. Welche Möglichkeiten dem Anwender bei der Erstellung eines Objekts geboten werden, ist stark abhängig vom jeweiligen DBMS.

8.1. Tabellen erstellen und verwalten

8.1.1. Namenskonventionen und Einschränkungen


Bevor näher auf die Namenskonventionen für Objekte eingegangen wird, müssen an dieser Stelle zuerst einige Fachbegriffe geklärt werden.

- **Bezeichner:** Namen für Objekte (Tabellen, Spalten, Views, usw.) heißen im Fachjargon Bezeichner.
- **Umschlossene Bezeichner:** Sind Bezeichner, die in Anführungszeichen " eingeschlossen sind

- **Reservierte Wörter:** Begriffe die in SQL eine bestimmte Bedeutung haben, z. B. **SELECT**, **WHERE**, usw.
- **Namensraum:** Logische Einteilung für Objektnamen. Bezeichner müssen innerhalb eines Namensraumes eindeutig sein.

Tabelle 8.1 listet die wichtigsten Einschränkungen auf, die für Bezeichner in beiden DBMS gelten.

Tabelle 8.1.: Einschränkungen für Bezeichner

| Bezeichnerlänge | 30 | 128 |
|--------------------|--|---|
| Reservierte Wörter | Bezeichner können keine reservierten Wörter sein, es sei denn, sie sind in Anführungszeichen " eingeschlossen. | Bezeichner können keine reservierten Wörter sein, es sei denn, sie sind in Anführungszeichen " eingeschlossen. |
| Namensgebung | Wenn Bezeichner nicht in Anführungszeichen (") eingeschlossen sind, müssen diese mit einem Buchstaben beginnen. Für umschlossene Bezeichner gilt dies nicht. | Wenn Bezeichner nicht in Anführungszeichen (") oder ([]) eingeschlossen sind, müssen diese mit einem Buchstaben, _, @ oder # beginnen. Für umschlossene Bezeichner gilt dies nicht. |
| Gültige Zeichen | Nicht umschlossene Bezeichner können nur aus den Buchstaben a-z und A-Z, den Ziffern 0-9, sowie _, \$ und # bestehen. Für umschlossene Bezeichner gilt, dass dort alle Zeichen, auch Leerzeichen vorkommen können. | Nicht umschlossene Bezeichner können nur aus den Buchstaben a-z und A-Z, den Ziffern 0-9, sowie @, \$, _ und # bestehen. Für umschlossene Bezeichner gilt, dass dort alle Zeichen, auch Leerzeichen vorkommen können. |
| Namensgleichheit | Zwei Datenbankobjekte im gleichen Namensraum müssen unterschiedliche Namen haben. | Bezeichner müssen innerhalb eines Schemas eindeutig sein. |
| Casesensitivität | Nicht umschlossene Bezeichner sind nicht Casesensitiv. Bezeichner die mit (") oder ([]) umschlossen sind, sind Casesensitiv. | Bezeichner die mit (") oder ([]) umschlossen sind, sind nicht Casesensitiv. |

Damit umschlossene Bezeichner in SQL Server 2008 R2 genutzt werden können, muss die Option **QUOTED_IDENTIFIER** den Wert **ON** haben. Dieser kann nötigenfalls mit **SET QUOTED_IDENTIFIER ON** gesetzt werden.



Die folgenden Internetliteraturhinweise liefern weitere Informationen.



- [i27561]
- [ms187879]

8.1.2. CREATE TABLE - Tabellen erstellen

Sowohl in Oracle als auch in SQL Server werden Tabellen mit Hilfe des Kommandos **CREATE TABLE** erstellt. Die grundlegende, SQL-Standardkonforme Syntax für **CREATE TABLE** lautet:

Listing 8.1: Die Syntax der CREATE TABLE-Anweisung

```
CREATE TABLE <tabellen_name> (
    <spaltenbezeichner 1> <datentyp>,
    <spaltenbezeichner 2> <datentyp>,
    ...,
    <spaltenbezeichner n> <datentyp>
);
```

Tabelle 8.2.: Die CREATE TABLE-Anweisung

| Ausdruck | Bedeutung |
|--------------------------------|--|
| CREATE TABLE <Tabellenname> | Diese Klausel leitet das Erstellen der Tabelle ein. Für den Tabellennamen gelten die in Tabelle 8.1 angegebenen Beschränkungen. |
| <Spaltenbezeichner> <Datentyp> | Jede Tabellenspalte wird durch einen Bezeichner/Name und einen Datentyp repräsentiert. Mit Hilfe des Namens kann die Spalte später angesprochen werden und der Datentyp legt den Wertebereich der Spalte fest. Je nach DBMS gelten auch hier unterschiedliche Einschränkungen. |

[Beispiel 8.2](#) zeigt ein einfaches **CREATE TABLE**-Statement.

Listing 8.2: Eine einfache CREATE TABLE-Anweisung in Oracle

```
CREATE TABLE Aktie (
    Aktie_ID    NUMBER,
    Name        VARCHAR2(25),
    WKN         NUMBER,
    ISIN        VARCHAR2(12)
);
```


Listing 8.3: Das gleiche in MS SQL Server

```
CREATE TABLE Aktie (  
    Aktie_ID    NUMERIC,  
    Name        VARCHAR(25),  
    WKN         NUMERIC,  
    ISIN        VARCHAR(12)  
);
```

Es wird eine Tabelle namens AKTIE, mit den Spalten AKTIE_ID, NAME, WKN und ISIN angelegt.

Zur besseren Umsetzung der Beispiele in den folgenden Abschnitten, werden nun einige Datensätze in die Tabelle AKTIE eingefügt.

Listing 8.4: Beispieldatensätze

```
INSERT INTO Aktie  
VALUES (1, 'Henker Co KG', 1236547, 'DE0006800002');  
  
INSERT INTO Aktie  
VALUES (2, 'AD and D', 43116589, 'DE0002300023');  
  
COMMIT;
```

8.1.3. CREATE TABLE AS... (CTAS)

Die Abkürzung „CTAS“ steht für **CREATE TABLE AS** und meint ein **CREATE TABLE** mit Unterabfrage. Mit Hilfe von CTAS können bestehende Tabellen teilweise oder ganz kopiert werden. [Beispiel 8.5](#) zeigt, wie in Oracle eine vollständige Kopie der Tabelle AKTIE angefertigt wird.

Listing 8.5: Oracle - CREATE TABLE AS (CTAS)

```
CREATE TABLE Aktie_Kopie  
AS  
    SELECT *  
    FROM Aktie;
```

Es wird eine Tabelle namens AKTIE_KOPIE erstellt. Diese erhält die komplette Struktur und den gesamten Inhalt der Tabelle AKTIE.

In Microsoft SQL Server kennt das **CREATE TABLE**-Statement keine Möglichkeit, eine Unterabfrage zu nutzen. Hier muss stattdessen das **SELECT INTO**-Statement genutzt werden.

Listing 8.6: MS SQL Server - SELECT INTO

```
SELECT *  
INTO Aktie_Kopie  
FROM Aktie;
```

Die Auswirkungen bleiben die gleichen, wie unter Oracle mit CTAS.



Microsoft SQL Server kennt das **CREATE TABLE AS**-Statement nicht. Es muss stattdessen das **SELECT INTO**-Statement genutzt werden. Die Auswirkungen von **CREATE TABLE AS** und **SELECT INTO** sind gleich.

8.1.4. ALTER TABLE - Tabellen verändern

Mit Hilfe der **ALTER TABLE**-Anweisung können bestehende Tabellendefinition verändert werden. Dies betrifft z. B.:

- Das Hinzufügen neuer Spalten zu einer Tabelle.
- Das Löschen von Spalten.
- Das Umbenennen von Spalten.
- Das Ändern des Datentyps einer Spalte.
- Ändern der Größe einer Spalte.
- Das Hinzufügen, ändern und löschen eines Standardwerts.
- Das Hinzufügen und Löschen von Constraints (siehe [Abschnitt 9.1](#))

Eine neue Spalte an eine Tabelle anfügen

In beiden DBMS gibt es, zum Hinzufügen einer Spalte zu einer Tabelle, die ADD-Klausel des **ALTER TABLE**-Kommandos. In [Beispiel 8.7](#), wird der Tabelle AKTIE eine neue Spalte namens HERKUNFT hinzugefügt.

Listing 8.7: Oracle - Tabellenspalte hinzufügen

```
ALTER TABLE Aktie  
ADD Herkunft VARCHAR2(25);
```

In SQL Server unterscheidet sich dieses Statement nur durch den Datentyp.

Listing 8.8: MS SQL Server - Tabellenspalte hinzufügen

```
ALTER TABLE Aktie
ADD Herkunft VARCHAR(25);
```

Wird eine neue Spalte an eine Tabelle angefügt, haben alle Zellen dieser Spalte den Wert NULL, es sei den, es wird ein Standardwert für diese Spalte definiert. In diesem Falle füllt Oracle die Spalte mit dem Standardwert auf. SQL Server tut dies nicht.



Beispiel 8.9 und Beispiel 8.10 zeigen, wie sich Oracle und MS SQL Server verhalten, wenn eine neue Spalte, mit einem Standardwert, hinzugefügt wird. Die Spalte HERKUNFT wird mit dem Standardwert „Deutschland“ an die Tabelle AKTIE angefügt. In Oracle werden dann automatisch alle bereits vorhandenen Zeilen mit dem neuen Standardwert aufgefüllt. In SQL Server wird dies nicht der Fall sein.

Listing 8.9: Tabellenspalte mit Standardwert hinzufügen in Oracle

```
ALTER TABLE Aktie
ADD Herkunft VARCHAR2(25) DEFAULT 'Deutschland';

SELECT *
FROM Aktie;
```

| AKTIE_ID | NAME | WKN | ISIN | HERKUNFT |
|----------|--------------|----------|--------------|-------------|
| 1 | Henker Co KG | 1236547 | DE0006800002 | Deutschland |
| 2 | AD and D | 43116589 | DE0002300023 | Deutschland |

2 Zeilen ausgewählt



Wird das gleiche Experiment in MS SQL Server durchgeführt, zeigt sich das die Spalte HERKUNFT nicht automatisch aufgefüllt wird.

Listing 8.10: Tabellenspalte mit Standardwert hinzufügen in SQL Server

```
ALTER TABLE Aktie
ADD Herkunft VARCHAR(25) DEFAULT 'Deutschland';

SELECT *
FROM Aktie;
```



| AKTIE_ID | NAME | WKN | ISIN | HERKUNFT |
|----------|--------------|----------|--------------|----------|
| 1 | Henker Co KG | 1236547 | DE0006800002 | NULL |
| 2 | AD and D | 43116589 | DE0002300023 | NULL |

2 Zeilen ausgewählt

Spalten vergrößern und verkleinern

Es besteht die Möglichkeit, die Definition einer Spalte nachträglich zu verändern. Dabei können verschiedene Dinge, wie z. B. der Spaltentyp oder der Standardwert einer Spalte geändert werden. Um eine solche Änderung durchzuführen, kennt das **ALTER TABLE**-Kommando unter Oracle die **MODIFY**-Klausel und unter SQL Server die **ALTER COLUMN**-Klausel. Hierzu einige Beispiele.

In [Beispiel 8.11](#) wird die Breite der Spalte HERKUNFT in der Tabelle AKTIE verändert.

Listing 8.11: Anpassen der Spaltenlänge in Oracle

```
ALTER TABLE Aktie
MODIFY Herkunft VARCHAR2(30);
```

Eine Vergrößerung stellt prinzipiell niemals ein Problem dar. Schwieriger wird es hingegen, wenn eine Spalte verkleinert werden muss. In Oracle geht das nur dann, wenn die Inhalte der Spalte kleiner sind als die neue Spaltengröße. Anderenfalls antwortet Oracle mit der in [Beispiel 8.12](#) sichtbaren Fehlermeldung:

Listing 8.12: Fehlermeldung beim verkleinern einer Spalte in Oracle

```
MODIFY Herkunft VARCHAR2(15)
*
FEHLER in Zeile 2:
ORA-01441: Spaltenlaenge kann nicht vermindert werden, weil ein Wert zu gross ist
```



Eine Tabellenspalte kann in Oracle nur auf die Größe des größten darin enthaltenen Werts verkleinert werden. In SQL Server kann eine Tabellenspalte auch mit Inhalt verkleinert werden.

Bei SQL Server muss lediglich die **MODIFY**-Klausel durch die **ALTER COLUMN**-Klausel ersetzt werden.

Listing 8.13: Anpassen der Spaltenlänge in SQL Server

```
ALTER TABLE Aktie
ALTER COLUMN Herkunft VARCHAR(30);
```

Ändern des Datentyps

Mit Hilfe der **MODIFY**-Klausel kann nicht nur die Größe einer Spalte verändert werden, sondern auch der Datentyp. In [Beispiel 8.14](#) wird der Datentyp der Spalte WKN von **NUMBER** auf **VARCHAR2**, bzw. von **NUMERIC** auf **VARCHAR** geändert.

Listing 8.14: Ändern des Datentyps

```
ALTER TABLE Aktie  
MODIFY WKN VARCHAR2(10);
```

In SQL Server sieht das Ändern des Datentyps einer Spalte sehr ähnlich aus.

Listing 8.15: Ändern des Datentyps

```
ALTER TABLE Aktie  
ALTER COLUMN WKN VARCHAR(10);
```

Eine Tabellenspalte muss in Oracle leer sein, damit ihr Datentyp verändert werden kann. In SQL Server kann der Datentyp einer Spalte auch mit Inhalt verändert werden.



Einen Defaultvalue hinzufügen

Eine weitere Aktion die mit **MODIFY** bzw. **ALTER COLUMN** möglich ist, ist das Hinzufügen, ändern oder entfernen eines Standardwertes bei einer Tabellenspalte. In [Beispiel 8.16](#) wird in Oracle der Standardwert der Spalte HERKUNFT von „Deutschland“ auf „USA“ geändert.

Listing 8.16: Einen Standardwert ändern

```
ALTER TABLE Aktie  
MODIFY Herkunft DEFAULT 'USA';
```

Mit der gleichen Anweisung kann der Standardwert einer Spalte, unter Oracle, nicht nur geändert sondern auch hinzugefügt werden. Das Löschen des Standardwertes geschieht, indem NULL als Standardwert zugewiesen wird.

Listing 8.17: Standardwert hinzufügen

```
ALTER TABLE Aktie  
MODIFY Herkunft DEFAULT NULL;
```

Bei SQL Server ist das Löschen eines Standardwerts anders als bei Oracle. In SQL Server wird ein Standardwert als sogenanntes Constraint¹ gehandhabt. Deshalb wird diese Aktion zu einem späteren Zeitpunkt in [Abschnitt 9.3.3](#) behandelt.



Wird in Oracle mit Hilfe **ADD**-Klausel eine Spalte mit Standardwert hinzugefügt, werden alle NULL-Werte in der gleichen Spalte mit dem Standardwert aufgefüllt. Wird die Spalte dagegen mit der **MODIFY**-Klausel, nachträglich mit einem Default-Wert ausgestattet, bleiben alle NULL-Werte erhalten!

Tabellenspalten umbenennen

Es ist möglich, bestehende Spalten umzubenennen. Dafür wird in Oracle die **RENAME COLUMN**-Klausel des **ALTER TABLE**-Kommandos verwendet. In Microsoft SQL Server gibt es hierfür eine gespeicherte Hilfsprozedur, welche das Umbenennen übernimmt. Der neue Spaltenname muss innerhalb der Tabelle eindeutig sein und es dürfen keine anderen Operationen zusammen mit dem Umbenennen geschehen.

Listing 8.18: Tabellenspalte umbenennen in Oracle

```
ALTER TABLE Aktie
RENAME COLUMN Name TO Bezeichnung;
```

Listing 8.19: Tabellenspalte umbenennen in SQL Server

```
EXEC sp_rename 'Aktie.Name', 'Bezeichnung', 'COLUMN'
```



Für das Umbenennen von Objekten ist in SQL Server die gespeicherte Hilfsprozedur `sp_rename` zuständig.

Zu beachten ist, dass das Umbenennen einer Spalte Auswirkungen auf abhängige Objekte wie z. B. Views oder Trigger haben kann und deshalb mit größter Vorsicht durchzuführen ist.

Tabellenspalten löschen

Tabellenspalten, die nicht mehr benötigt werden, können jeder Zeit gelöscht werden. Auf diese einfache Art und Weise kann Speicherplatz zu weiteren Nutzung freigegeben werden. Allgemein gilt als Einschränkung beim Löschen einer Tabellenspalte:

¹constraint engl. = Einschränkung

- Die letzte Spalte in einer Tabelle kann nicht gelöscht werden. Es muss dann die gesamte Tabelle gelöscht werden.

Für Oracle gilt zusätzlich:

- Ein normaler Nutzer kann keine Spalten aus einer Tabelle löschen, die dem Nutzer sys gehört.

Beispiel 8.19 zeigt das Löschen einer Tabellenspalte in Oracle und SQL Server.

Listing 8.20: Tabellenspalte löschen

```
ALTER TABLE Aktie  
DROP COLUMN WKN;
```

8.1.5. DROP TABLE - Tabellen löschen

Eine nicht mehr benötigte Tabelle, wird in Oracle und SQL Server mit dem **DROP TABLE**-Kommando gelöscht.

- Alle verknüpften Indizes und Trigger werden mitgelöscht.
- Alle abhängigen Views bleiben bestehen und werden ungültig.

Das folgende Beispiel löscht die Tabelle AKTIE.

Listing 8.21: Eine Tabelle löschen

```
DROP TABLE Aktie;
```

8.1.6. TRUNCATE TABLE - Tabellen leeren

Eine Tabelle kann mit **TRUNCATE TABLE** geleert werden. Die Tabelle selbst bleibt dabei erhalten. Um eine Tabelle zu leeren, gibt es drei Möglichkeiten:

- Das DML-Statement **DELETE**
- Das Löschen der Tabelle mit **DROP TABLE** und neu erstellen mit **CREATE TABLE**
- Das DDL-Statement **TRUNCATE**

Den Tabelleninhalt mit DELETE löschen

Es können alle Zeilen einer Tabelle mit dem DML-Kommando **DELETE** gelöscht werden.

Listing 8.22: Zeilen mit DELETE löschen

```
DELETE FROM Aktie;
```

Bei einer großen Tabelle werden hierfür sehr viele Systemressourcen benötigt (CPU, RAM, usw.). Des Weiteren kann es passieren, dass beim Löschen von Zeilen, Trigger ausgelöst werden.



Der Speicherplatz, der durch die Tabelle vor dem Löschen belegt wurde, bleibt bei der Verwendung von **DELETE** belegt.

Einzigster Vorteil ist, dass mit der **DELETE**-Klausel die Zeilen ausgewählt werden können, die gelöscht werden sollen.

Die Tabelle löschen und neu erstellen

Eine Tabelle kann gelöscht und mit **CREATE TABLE** neu erstellt werden. Dabei gehen alle mit dieser Tabelle verbundenen Indizes, Integritäts Constraints und Trigger verloren und alle von der Tabelle abhängigen Objekte werden ungültig.

Eine Tabelle mit TRUNCATE leeren

Um alle Zeilen einer Tabelle zu löschen kann das **TRUNCATE**-Statement verwendet werden.

Listing 8.23: Zeilen mit TRUNCATE abschneiden

```
TRUNCATE TABLE Aktie;
```



In Oracle produziert das **TRUNCATE**-Statement, wie alle DDL-Statements, automatisch ein **COMMIT**, d. h. es kann nicht rückgängig gemacht werden. In SQL Server ist das Zurückrollen eines **TRUNCATE**-Statements möglich.



Der Speicherplatz, der durch die Tabelle vor dem Löschen belegt wurde, wird bei der Verwendung von **TRUNCATE**, freigegeben.

8.2. Views erstellen verwalten

8.2.1. Was sind Views?

Bei der täglichen Arbeit mit einer Datenbank treten häufig immer wiederkehrende **SELECT**-Statements auf. Dies kann z. B. deshalb sein, weil ein Nutzer immer wieder die gleiche Sicht (gleiche Spalten, gleiche Filterbedingung) auf die Daten einer Tabelle benötigt.

Eine View ist eine genau definierte Sicht auf eine bestimmte Datenmenge.



8.2.2. Views erstellen

Views werden mit dem **CREATE VIEW**-Kommando erstellt. Die Syntax für **CREATE VIEW** sieht wie folgt aus:

Listing 8.24: Die Syntax von CREATE VIEW

```
CREATE VIEW <View_name>
(<Spalten_alias 1, Spalten_alias 2, ..., Spalten_alias n>
AS
<Auswahlabfrage>;
```

Tabelle 8.3.: Die CREATE VIEW-Anweisung

| Ausdruck | Bedeutung |
|-------------------------|---|
| CREATE VIEW <View_name> | Diese Klausel leitet das Erstellen der View ein. Für den Viewname gelten die in Tabelle 8.1 angegebenen Beschränkungen. |
| <Spalten_alias> | Für jeden Spaltenbezeichner, der in der Auswahlabfrage genutzt wird, kann an dieser Stelle ein Aliasname festgelegt werden. |
| <Auswahlabfrage> | Dies ist das SELECT-Statement. |

Ein einfaches Beispiel für das Erstellen einer View ist in [Beispiel 8.24](#) zu sehen.

Listing 8.25: Eine einfache View

```
CREATE VIEW v_Kunde
AS
SELECT Vorname, Nachname
FROM Kunde;
```

Was an dieser Stelle passiert ist, dass das DBMS das **SELECT**-Statement verarbeitet und unter dem Namen **v_KUNDEN** abspeichert. Anschließend kann, wie in [Beispiel 8.26](#), mit SQL auf die View zugegriffen werden.

Listing 8.26: Zugriff auf eine View

```
SELECT Vorname , Nachname
FROM   v_Kunde ;
```



| VORNAME | NACHNAME |
|-----------|------------|
| Niklas | Schneider |
| Mia | Keller |
| Lilli | Beck |
| Emilia | Keller |
| Finn | Junge |
| Marie | Vogel |
| Rudi | Roggatz |
| Leni | Koch |
| Chris | Zimmermann |
| Justin | Gabriel |
| Sebastian | Schröder |

561 Zeilen ausgewählt

Auch wenn in der **SELECT**-Klausel der Auswahlabfrage der * verwendet wird, wird im Hintergrund folgendes Statement, als View gespeichert:

Listing 8.27: Was tatsächlich gespeichert wird

```
-- So wird die View erstellt
CREATE VIEW v_Kunde
AS
    SELECT *
    FROM   Kunde ;

-- Das wird gespeichert
SELECT Kunden_ID , Vorname , Nachname
FROM   Kunde ;
```

Diese Tatsache ist nicht ganz unwichtig, wie folgendes Szenario beweist:

Listing 8.28: Eine Szenario mit Tücke

```
CREATE VIEW v_Aktie
AS
    SELECT *
    FROM   Aktie ;

ALTER TABLE Aktie
ADD Herkunft VARCHAR2(30);

SELECT *
FROM   v_Aktie ;
```

| AKTIE_ID | NAME | WKN | ISIN |
|----------|--------------|----------|--------------|
| 1 | Henker Co KG | 1236547 | DE0006800002 |
| 2 | AD and D | 43116589 | DE0002300023 |

2 Zeilen ausgewählt



Da bei der Erstellung der View v_AKTIE, der *, in die einzelnen Spalten der Tabelle AKTIE aufgelöst wurde, ist die neu hinzugefügte Spalte HERKUNFT, in der View v_AKTIE noch nicht zu sehen. Hierzu müsste die Viewdefinition geändert bzw. die View neu erstellt werden.

Wird in der Auswahlabfrage einer View das * Symbol verwendet, wird dieses interpretiert. D. h. es wird ersetzt durch die tatsächliche Spaltenliste der Quelltable. Änderungen an der Struktur der Tabelle werden somit von der View nicht erkannt.



Wie in [Tabelle 8.3](#) bereits erklärt, kann bei der Erstellung einer View auch eine Liste mit Spaltenaliasnamen angegeben werden. Dies ist in den folgenden Fällen immer notwendig:

- Wenn in der View ein berechneter Ausdruck vorhanden ist
- Wenn in der View mehrere Tabellen mit einem Join verbunden sind und Spalten mit gleichem Namen ausgegeben werden müssen.

[Beispiel 8.29](#) zeigt eine View mit Spaltenaliasliste.

Listing 8.29: Eine einfache View mit Spaltenaliasliste

```
CREATE VIEW v_Kunde
(Vorname, Nachname, Lebensalter)
AS
SELECT k.Vorname, k.Nachname,
       ROUND(MONTHS_BETWEEN(SYSDATE, Geburtsdatum) / 12, 0)
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);

SELECT *
FROM   v_Kunde;
```



| VORNAME | NACHNAME | LEBENSALTER |
|-----------|------------|-------------|
| Mia | Keller | 41 |
| Emilia | Keller | 23 |
| Finn | Junge | 37 |
| Marie | Vogel | 42 |
| Rudi | Roggatz | 26 |
| Leni | Koch | 38 |
| Chris | Zimmermann | 23 |
| Sebastian | Schröder | 24 |
| Justin | Zimmermann | 34 |
| Petra | Krause | 34 |
| Clara | Rollert | 23 |
| Gustav | Witte | 23 |

400 Zeilen ausgewählt

Wie gut zu erkennen ist, ersetzen die Spaltenalias die tatsächlichen Spaltennamen in V_KUNDE. Die gleiche Auswirkung wäre auch mit dem folgenden Statement zu erreichen:

Listing 8.30: Eine einfache View mit Spaltenaliasen

```
CREATE VIEW v_Kunde
AS
SELECT k.Vorname, k.Nachname,
       ROUND(MONTHS_BETWEEN(SYSDATE, Geburtsdatum) / 12, 0) AS Lebensalter
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);
```



Wird eine Spaltenaliasliste genutzt, muss diese genauso viele Aliasnamen umfassen, wie die **SELECT**-Liste der Auswahlabfrage Spaltennamen zurückgibt.

Hierzu ein kleines Beispiel. Im folgenden **CREATE VIEW**-Statement werden zu wenige Spaltenalias angegeben. Oracle und auch SQL Server antworten prompt mit einer Fehlermeldung.

Listing 8.31: Eine einfache View mit fehlerhafter Spaltenaliasliste in Oracle

```

CREATE VIEW v_Kunde
(Vorname, Nachname)
AS
SELECT k.Vorname, k.Nachname,
       ROUND(MONTHS_BETWEEN(SYSDATE, Geburtsdatum) / 12, 0)
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);

(Vorname, Nachname)
*
FEHLER in Zeile 2:
ORA-01730: invalid number of column names specified

```

SQL Server antwortet wie folgt:

Listing 8.32: Eine einfache View mit fehlerhafter Spaltenaliasliste in SQL Server

```

CREATE VIEW v_Kunde
(Vorname, Nachname)
AS
SELECT k.Vorname, k.Nachname, DATEDIFF(YEAR, getDate(), Geburtsdatum)
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);

Meldung 8158, Ebene 16, Status 1, Prozedur v_Kunde, Zeile 4
'v_Kunde' besitzt mehr Spalten, als in der Spaltenliste angegeben sind.

```

Bereits weiter oben in diesem Abschnitt wurde erläutert, dass eine View, in der ein berechneter Ausdruck vorkommt, zwingend mit Spaltenaliasen versehen werden muss. [Beispiel 8.33](#) beweist dies:

Listing 8.33: Eine View mit einer berechneten Spalte in Oracle

```

CREATE VIEW v_Kunde
(Vorname, Nachname)
AS
SELECT k.Vorname, k.Nachname,
       ROUND(MONTHS_BETWEEN(SYSDATE, Geburtsdatum) / 12, 0)
FROM   Kunde k INNER JOIN Eigenkunde ek
       ON (k.Kunden_ID = ek.Kunden_ID);

SELECT k.Vorname, k.Nachname,
       ROUND(MONTHS_BETWEEN(SYSDATE, Geburtsdatum) / 12, 0)
       *
FEHLER in Zeile 3:
ORA-00998: Dieser Ausdruck braucht einen Spalten-Alias

```

Auch SQL Server hat hiermit Probleme:

Listing 8.34: Eine View mit einer berechneten Spalte in SQL Server

```
CREATE VIEW v_Kunde (Vorname, Nachname) AS
SELECT k.Vorname, k.Nachname, DATEDIFF(YEAR, getDate(), Geburtsdatum)
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID);
```

Meldung 4511, Ebene 16, Status 1, Prozedur v_Kunde, Zeile 3
Fehler beim Ausführen von CREATE VIEW oder CREATE FUNCTION, da
für die 3-Spalte kein Spaltenname angegeben wurde.

Das Problem kann mit einer Spaltenaliasliste oder durch direkte Vergabe eines Spaltenalias gelöst werden.









In SQL Server darf die Auswahlabfrage einer View keine ORDER BY-Klausel enthalten.

8.2.3. Views und DML

Views können auch für die Ausführung von DML-Statements verwendet werden. Dabei gibt es jedoch einige Einschränkungen und Regeln die zu beachten sind.

Tabelle 8.4.: Regeln für DML-Operationen auf Views

| Einschränkung |   | |   | |   | |
|--|--|---|---|---|---|---|
| | INSERT | | UPDATE | | DELETE | |
| Es dürfen keine Aggregatfunktionen (COUNT, SUM, MAX, MIN, AVG) in der View genutzt werden. | X | X | X | X | X | X |
| Die View darf keine GROUP BY-Klausel enthalten. | X | X | X | X | X | X |
| Die View darf das DISTINCT-Schlüsselwort nicht benutzen. | X | X | X | X | X | X |
| Die View darf keine berechneten Ausdrücke aufweisen. | X | X | X | X | | |
| Die View darf keine Pseudospalten enthalten | X | | X | | X | |
| Die View darf keinen Join enthalten. | X | X | | | X | X |
| Alle mit NOT NULL markierten Spalten der Basistabelle müssen im INSERT-Statement berücksichtigt werden. | X | X | | | | |
| Der Einfüge- bzw. Änderungsvorgang muss, falls eine CHECK-Option in der View vorhanden ist (siehe Abschnitt 8.2.3), den Vorgaben der WHERE-Klausel der Abfrage genügen. | X | X | X | X | | |
| Die View darf keine READ ONLY-Option (siehe Abschnitt 8.2.3) enthalten. | X | | X | | | |

Die Einschränkung WITH CHECK OPTION

Bei der Erstellung einer View kann eine zusätzliche Einschränkung mit angegeben werden, die sogenannte **CHECK OPTION**. Diese schränkt den Nutzer dahingehend ein, dass nur noch solche Datensätze geändert werden können, die auch in der View zu sehen sind.

Listing 8.35: Ein Experiment mit den CHECK OPTION

```
CREATE VIEW v_Mitarbeiter AS
SELECT *
FROM   Mitarbeiter
WHERE  Bankfiliale_ID = 5;

INSERT INTO v_Mitarbeiter
VALUES (666, 'Florian', 'Weidinger', 12, 8, TO_DATE('01.03.1988', 'DD.MM.YYYY'),
       '38B546C1-CDF-36A7B97', 1500, 'Abendrot Gase', '13', '39444',
       'Hecklingen', 20);

1 row inserted

ROLLBACK;
```

Obwohl die **WHERE**-Klausel der View **v_MITARBEITER** die Anzeige auf die Bankfiliale mit der ID fünf einschränkt, kann trotzdem ein Datensatz in die Bankfiliale Nummer acht eingefügt werden.

Um die DML-Möglichkeiten der View **v_MITARBEITER** einzuschränken, wird im nächsten Beispiel die **CHECK**-Option angewendet.

Listing 8.36: Ein Experiment mit der CHECK OPTION in Oracle

```
CREATE VIEW v_Mitarbeiter AS
SELECT *
FROM   Mitarbeiter
WHERE  Bankfiliale_ID = 5
WITH CHECK OPTION;

INSERT INTO v_Mitarbeiter
VALUES (666, 'Florian', 'Weidinger', 12, 8, TO_DATE('01.03.1988', 'DD.MM.YYYY'),
       '38B546C1-CDF-36A7B97', 1500, 'Abendrot Gase',
       '13', '39444', 'Hecklingen', 20);

INSERT INTO v_Mitarbeiter
      *
FEHLER in Zeile 1:
ORA-01402: Verletzung der where-Klausel einer View with check option
```

Da jetzt die **CHECK**-Option genutzt wurde, reagiert das DBMS mit einer Fehlermeldung auf DML-Statements, die sich auf **v_MITARBEITER** beziehen und nicht der **WHERE**-Klausel der View entsprechen.

Listing 8.37: Ein Experiment mit der CHECK OPTION in SQL Server

```
CREATE VIEW v_Mitarbeiter
AS
  SELECT *
  FROM   Mitarbeiter
  WHERE  Bankfiliale_ID = 5
WITH CHECK OPTION;

INSERT INTO v_Mitarbeiter
VALUES (666, 'Florian', 'Weidinger', 12, 8,
       CONVERT(DATETIME2, '01.03.1988', 104),
       '38B546C1-CDF-36A7B97', 1500, 'Abendrot Gase',
       '13', '39444', 'Hecklingen', 20);
```

Meldung 550, Ebene 16, Status 1, Zeile 1
Fehler beim Einfügen oder Aktualisieren, da die Zielsicht **WITH CHECK OPTION** angibt oder sich auf eine Sicht erstreckt, die **WITH CHECK OPTION** angibt, und mindestens eine Ergebniszeile nicht der **CHECK OPTION**-Einschränkung entsprach.

Die Einschränkung WITH READ ONLY - Oracle

Die **READ ONLY**-Option für Views ermöglicht es, einem Nutzer den Schreibzugriff auf eine View zu verbieten. Die View kann somit nur noch lesend genutzt werden.

Listing 8.38: Eine View mit mit READ ONLY Option erstellen

```
CREATE VIEW v_Mitarbeiter
AS
  SELECT *
  FROM   Mitarbeiter
WITH READ ONLY;
```

Versucht ein Nutzer trotzdem mit einem DML-Statement auf die View zuzugreifen, wird er mit einer Fehlermeldung abgewiesen.

Listing 8.39: Daten in eine READ ONLY View einfügen schlägt fehl

```
INSERT INTO v_Mitarbeiter
VALUES (666, 'Florian', 'Weidinger', 12, 8,
       TO_DATE('01.03.1988', 'DD.MM.YYYY'),
       '38B546C1-CDF-36A7B97', 1500, 'Abendrot Gase',
       '13', '39444', 'Hecklingen', 20);

INSERT INTO v_Mitarbeiter
*
FEHLER in Zeile 1:
ORA-42399: cannot perform a DML operation on a read-only view
```

Um diese Option wieder von der View zu nehmen, muss die View neu erstellt werden (siehe [Abschnitt 8.2.4](#))

8.2.4. Views ändern

Müssen an einer View Veränderungen vorgenommen werden, bedeutet dies immer, dass die View neu erstellt werden muss. Oracle und SQL Server kennen hierzu unterschiedliche Wege:

- In Oracle wird die **CREATE VIEW**-Klausel erweitert: **CREATE OR REPLACE VIEW**.
- SQL Server benutzt hierfür die **ALTER VIEW**-Anweisung.

Die beiden Beispiele [Beispiel 8.40](#) und [Beispiel 8.41](#) zeigen, wie in Oracle und SQL Server eine Viewdefinition geändert werden kann.

Listing 8.40: Eine View ändern in Oracle

```
-- Zuerst wird die View erstellt
CREATE VIEW v_Mitarbeiter
AS
    SELECT *
    FROM   Mitarbeiter
WITH READ ONLY;

-- Dann wird sie geändert
CREATE OR REPLACE VIEW v_Mitarbeiter
AS
    SELECT *
    FROM   Mitarbeiter;
```

Und nun SQL Server.

Listing 8.41: Eine View ändern in SQL Server

```
-- Zuerst wird die View erstellt
CREATE VIEW v_Mitarbeiter
AS
    SELECT *
    FROM Mitarbeiter;

-- Dann wird sie geändert
ALTER VIEW v_Mitarbeiter
AS
    SELECT *
    FROM Mitarbeiter
    WHERE Bankfiliale_ID = 5;
```

8.2.5. Views löschen

Zum Löschen von Views gibt es das Kommando **DROP VIEW**.

Listing 8.42: Eine View löschen

```
DROP VIEW viw_countries;
```

8.3. Übungen - Erstellen von Views

1. Erstellen Sie die View `V_ARBEITSORT`. Diese muss für jeden Mitarbeiter den Vorname, den Nachnamen, die Bankfiliale_ID und den Ort anzeigen, an dem sich die Filiale befindet.

| VORNAME | NACHNAME | BANKFILIALE_ID | ORT |
|----------|----------|----------------|--------------|
| Marie | Kipp | 1 | Aschersleben |
| Louis | Schmitz | 1 | Aschersleben |
| Johannes | Lehmann | 1 | Aschersleben |
| Dirk | Peters | 1 | Aschersleben |
| Amelie | Krüger | 1 | Aschersleben |

93 Zeilen ausgewählt



2. Erstellen Sie die View `V_DEPOTBESITZER`, die zu jedem Eigenkunden, der ein Depot besitzt, seinen Vor- und Nachnamen, die Strasse mit der Hausnummer, sowie PLZ und Ort anzeigt.

| VORNAME | NACHNAME | STRASSE | PLZ | ORT |
|-----------|----------|-------------------|-------|----------|
| Sophie | Junge | Plutoweg 3 | 39435 | Bördeau |
| Hanna | Beck | Beimsstraße 9 | 39439 | Güsten |
| Sebastian | Peters | Steinigstraße 3 | 39240 | Staßfurt |
| Tina | Berger | Bundschuhstraße 1 | 04177 | Leipzig |

239 Zeilen ausgewählt



3. Erstellen Sie die View `V_FINANZBERATER`, die für alle Eigenkunden deren Vor- und Nachnamen anzeigt, sowie den Vor- und den Nachnamen ihres persönlichen Finanzberaters (Tabelle `EIGENKUNDEMITARBEITER`).

| Vorname Kunde | Nachname Kunde | Vorname Berater | Nachname Berater |
|---------------|----------------|-----------------|------------------|
| Mia | Keller | Lena | Herrmann |
| Emilia | Keller | Louis | Wagner |
| Finn | Junge | Leni | Friedrich |
| Marie | Vogel | Finn | Wolf |
| Rudi | Roggatz | Frank | Meierhöfer |
| Leni | Koch | Frank | Hartmann |
| Chris | Zimmermann | Clara | Walther |
| Justin | Zimmermann | Leni | Friedrich |
| Petra | Krause | Chris | Hartmann |
| Clara | Rollert | Franz | Berger |

400 Zeilen ausgewählt



4. Erstellen Sie die View `V_UNTERSTELLUNGSVERHAELTNIS`, die zu jedem Mitarbeiter (Vorname, Nachname) den Vor- und den Nachnamen seines Vorgesetzten anzeigt. Wichtig ist, dass alle Mitarbeiter, auch Herr Max Winter, der keinen Vorgesetzten hat, angezeigt werden.



| VORNAME | NACHNAME | VORNAME | NACHNAME |
|-----------|------------|---------|----------|
| Finn | Seifert | Max | Winter |
| Sarah | Werner | Max | Winter |
| Tim | Sindermann | Sarah | Werner |
| Sebastian | Schwarz | Sarah | Werner |
| Emily | Meier | Finn | Seifert |
| Peter | Möller | Finn | Seifert |

100 Zeilen ausgewählt

5. Erstellen Sie die View `V_INNENDIENSTMITARBEITER`, die ermittelt, ob es Mitarbeiter gibt (Vorname und Nachname), die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten) und Filialleiter.



| VORNAME | NACHNAME |
|-----------|------------|
| Amelie | Krüger |
| Anna | Schneider |
| Chris | Simon |
| Christian | Haas |
| Elias | Sindermann |

40 Zeilen ausgewählt

6. Erstellen Sie die View `V_GIROKONTOINHABER`, die alle Eigenkunden anzeigt, die nur ein Girokonto besitzen.



| VORNAME | NACHNAME |
|---------|------------|
| Amelie | Becker |
| Amelie | Richter |
| Chris | Walther |
| Emilia | Keller |
| Georg | Keller |
| Johanna | Schäfer |
| Justin | Zimmermann |

21 Zeilen ausgewählt

8.4. Lösungen - Erstellen von Views

1. Erstellen Sie die View v_ARBEITSORT. Diese muss für jeden Mitarbeiter den Vorname, den Nachnamen, die Bankfiliale_ID und den Ort anzeigen, an dem sich die Filiale befindet.

```
CREATE VIEW v_Arbeitsort
AS
SELECT m.Vorname, m.Nachname, m.Bankfiliale_ID, b.Ort
FROM Mitarbeiter m INNER JOIN Bankfiliale b
ON (b.Bankfiliale_ID = m.Bankfiliale_ID);
```



2. Erstellen Sie die View v_DEPOTBESITZER, die zu jedem Eigenkunden, der ein Depot besitzt, seinen Vor- und Nachnamen, die Strasse mit der Hausnummer, sowie PLZ und Ort anzeigt.

```
CREATE VIEW v_Depotbesitzer
AS
SELECT k.Vorname, k.Nachname, ek.Strasse || ' ' ||
ek.Hausnummer AS Strasse,
ek.PLZ, ek.Ort
FROM Kunde k INNER JOIN Eigenkunde ek
ON (k.Kunden_ID = ek.Kunden_ID)
INNER JOIN EigenkundeKonto ekk
ON (ek.Kunden_ID = ekk.Kunden_ID)
INNER JOIN Depot d
ON (ekk.Konto_ID = d.Konto_ID);
```



```
CREATE VIEW v_Depotbesitzer
AS
SELECT k.Vorname, k.Nachname, ek.Strasse + ' ' + ek.Hausnummer AS Strasse,
ek.PLZ, ek.Ort
FROM Kunde k INNER JOIN Eigenkunde ek
ON (k.Kunden_ID = ek.Kunden_ID)
INNER JOIN EigenkundeKonto ekk
ON (ek.Kunden_ID = ekk.Kunden_ID)
INNER JOIN Depot d
ON (ekk.Konto_ID = d.Konto_ID);
```



3. Erstellen Sie die View v_FINANZBERATER, die für alle Eigenkunden deren Vor- und Nachnamen anzeigt, sowie den Vor- und den Nachnamen ihres persönlichen Finanzberaters (Tabelle EIGENKUNDENMITARBEITER).



```
CREATE VIEW v_Finanzberater
("Vorname Kunde", "Nachname Kunde", "Vorname Berater", "Nachname Berater")
AS
SELECT k.Vorname, k.Nachname, m.Vorname, m.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
LEFT OUTER JOIN EigenkundeMitarbeiter ekm
ON (ek.Kunden_ID = ekm.Kunden_ID)
LEFT OUTER JOIN Mitarbeiter m
ON (ekm.Mitarbeiter_ID = m.Mitarbeiter_ID);
```

4. Erstellen Sie die View v_UNTERSTELLUNGSVERHAELTNIS, die zu jedem Mitarbeiter (Vorname, Nachname) den Vor- und den Nachnamen seines Vorgesetzten anzeigt. Wichtig ist, dass alle Mitarbeiter, auch Herr Max Winter, der keinen Vorgesetzten hat, angezeigt werden.



```
CREATE VIEW v_Unterstellungsverhaeltnis
AS
SELECT m.Vorname, m.Nachname, v.Vorname, v.Nachname
FROM Mitarbeiter m LEFT OUTER JOIN Mitarbeiter v
ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID)
```

5. Erstellen Sie die View v_INNENDIENSTMITARBEITER, die ermittelt, ob es Mitarbeiter gibt (Vorname und Nachname), die keine Kundenberatung durchführen. Ausgenommen sind leitende Mitarbeiter (Mitarbeiter die in keiner Bankfiliale arbeiten) und Filialleiter.



```
CREATE VIEW v_Innendienstmitarbeiter
AS
SELECT m.Vorname, m.Nachname
FROM Mitarbeiter m LEFT OUTER JOIN EigenkundeMitarbeiter ekm
ON (m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
WHERE ekm.Mitarbeiter_ID IS NULL
MINUS
SELECT DISTINCT v.Vorname, v.Nachname
FROM Mitarbeiter m INNER JOIN Mitarbeiter v
ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);
```



```
CREATE VIEW v_Innendienstmitarbeiter
AS
SELECT m.Vorname, m.Nachname
FROM Mitarbeiter m LEFT OUTER JOIN EigenkundeMitarbeiter ekm
ON (m.Mitarbeiter_ID = ekm.Mitarbeiter_ID)
WHERE ekm.Mitarbeiter_ID IS NULL
EXCEPT
SELECT DISTINCT v.Vorname, v.Nachname
FROM Mitarbeiter m INNER JOIN Mitarbeiter v
ON (m.Vorgesetzter_ID = v.Mitarbeiter_ID);
```

6. Erstellen Sie die View v_GIROKONTOINHABER, die alle Eigenkunden anzeigt, die nur ein Girokonto besitzen.



```
CREATE VIEW v_Girokontoinhaber
AS
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)
MINUS
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)
MINUS
SELECT k.Vorname, k.Nachname
FROM Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);
```



```
CREATE VIEW v_Girokontoinhaber
AS
  SELECT k.Vorname, k.Nachname
  FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
        INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
        INNER JOIN Girokonto g ON (ekk.Konto_ID = g.Konto_ID)

EXCEPT
  SELECT k.Vorname, k.Nachname
  FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
        INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
        INNER JOIN Sparbuch s ON (ekk.Konto_ID = s.Konto_ID)

EXCEPT
  SELECT k.Vorname, k.Nachname
  FROM   Kunde k INNER JOIN Eigenkunde ek ON (k.Kunden_ID = ek.Kunden_ID)
        INNER JOIN EigenkundeKonto ekk ON (ek.Kunden_ID = ekk.Kunden_ID)
        INNER JOIN Depot d ON (ekk.Konto_ID = d.Konto_ID);
```


9. Constraints

Inhaltsangabe

| | | |
|------------|---|-------------|
| 9.1 | Was sind Constraints | 9-2 |
| 9.2 | Die Constraints | 9-2 |
| 9.2.1 | Das CHECK-Constraint | 9-3 |
| 9.2.2 | Das NOT NULL-Constraint | 9-4 |
| 9.2.3 | Das UNIQUE-Constraint | 9-5 |
| 9.2.4 | Das PRIMARY KEY-Constraint | 9-6 |
| 9.2.5 | Das FOREIGN KEY-Constraint | 9-7 |
| 9.2.6 | Das SQL Server DEFAULT-Constraint | 9-10 |
| 9.3 | Constraints umbenennen und löschen | 9-10 |
| 9.3.1 | Constraints umbenennen | 9-10 |
| 9.3.2 | Constraints löschen | 9-11 |
| 9.3.3 | Standardwerte in SQL Server löschen | 9-11 |

9.1. Was sind Constraints

Der englische Begriff „Constraint“ bedeutet übersetzt soviel wie: „Einschränkung“ oder „Zwang“. Constraints werden in Datenbankmanagementsystemen verwendet, um genau definierte Richtlinien für die Erfassung und die Verwaltung der Daten zu schaffen. Sie sorgen z. B. dafür, dass manche Spalten immer zwingend einen Wert ungleich NULL haben müssen oder dass sie nur eindeutige Werte aufnehmen können. Es ist auch möglich einen genauen Wertebereich für eine Spalte zu definieren oder Werte aus Spalten anderer Tabellen zu referenzieren. Oracle und MS SQL Server kennen fünf Constraints für das relationale Datenmodell:

- **CHECK**: Definiert einen exakten Wertebereich für eine Spalte.
- **NOT NULL**: Definiert eine Spalte so, dass sie zwingend immer einen Wert ungleich NULL enthalten muss.
- **UNIQUE**: Legt fest, dass die Werte einer Spalte oder einer Kombination von Spalten eindeutig sein müssen.
- **PRIMARY KEY**: Hat die Aufgabe, ein eindeutiges Identifikationsmerkmal für jede Zeile einer Tabelle darzustellen. Er ist eine Kombination aus dem **NOT NULL**- und dem **UNIQUE**-Constraint und kann sich ebenfalls auf eine Kombination von Spalten beziehen.
- **FOREIGN KEY**: Referenziert eine Spalte einer anderen Tabelle, die mit einem **UNIQUE**- oder **PRIMARY KEY**-Constraint versehen sein muss und stellt somit die referentielle Integrität (siehe [Abschnitt 9.2.5](#)) der Datenbank sicher.

Zusätzlich zu diesen fünf kennt Oracle noch das „REF“-Constraint, das jedoch nur im Rahmen der objektorientierten Anteile von Oracle Bedeutung hat und hier keine weitere Erwähnung findet. SQL Server kennt zusätzlich noch ein weiteres Constraint: das **DEFAULT**-Constraint.

9.2. Die Constraints

Constraints können mit Hilfe der beiden Kommandos **CREATE TABLE** und **ALTER TABLE** angelegt werden. Sie werden durch einen Bezeichner und ihren Typ repräsentiert. Die Bezeichner von Constraints unterliegen ebenfalls den in [Tabelle 8.1](#) beschriebenen Regeln.



Wird für ein Constraint kein Name festgelegt, legt Oracle automatisch einen Namen nach dem Schema „SYS_Cn“ fest, wobei n eine sechstellige Zufallszahl darstellt z. B. SYS_C168349. SQL Server verwendet ein Namensschema mit dem Aufbau „typ__tabelle__spalte__n“ wobei n eine eindeutige hexadezimal Nummer darstellt, z. B. PK__mitarbeiter__mitarbeiter_id__4B561A78.

In einem **CREATE TABLE**-Kommando können Constraints als „Inline Constraint“ und als „Out Of Line Constraint“ angelegt werden.

Listing 9.1: Constraints erstellen

```
CREATE TABLE <Tabellenname>(
  <Spalte 1> <Datentyp> CONSTRAINT <Inline Constraint Name> <Constraint Typ>,
  <Spalte 2> <Datentyp> CONSTRAINT <Inline Constraint Name> <Constraint Typ>,
  ...
  <Spalte n> <Datentyp> CONSTRAINT <Inline Constraint Name> <Constraint Typ>,
  CONSTRAINT <Out Of Line Constraint Name> <Constraint Typ> <Spalte 1, Spalte n>
  CONSTRAINT <Out Of Line Constraint Name> <Constraint Typ> <Spalte>
);
```



Wird ein Constraint direkt mit der Definition einer Spalte angelegt, wird es als Inline Constraint bezeichnet und bezieht sich auf die Spalte mit der es definiert wurde. Wird ein Constraint im Anschluss an die Spaltendefinitionen angelegt, wird es als Out Of Line Constraint bezeichnet und kann sich auf mehrere Spalten beziehen.

9.2.1. Das CHECK-Constraint

Das **CHECK**-Constraint hat die Aufgabe einen genauen Wertebereich für eine Spalte festzulegen. Beispielsweise wäre ein **CHECK**-Constraint auf der Spalte GEHALT der Tabelle MITARBEITER sinnvoll, das definiert, dass Gehälter niemals negativ und niemals über 90000 EUR sein können.

In [Beispiel 9.2](#) wird gezeigt, wie die oben genannte Einschränkung für die GEHALT-Spalte der Tabelle MITARBEITER als Out Of Line Constraint angelegt wird.

Listing 9.2: Ein **CHECK**-Constraint als Out Of Line Constraint

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT gehalt_ck CHECK (Gehalt > 0 AND Gehalt <= 90000);
```

Um ein **CHECK**-Constraint als Inline Constraint anzulegen, muss es direkt bei der Tabellenerstellung mit angelegt werden. [Beispiel 9.3](#) zeigt das gleiche Constraint noch einmal, aber als Inline Constraint.

Listing 9.3: Ein **CHECK**-Constraint als Inline Constraint

```
CREATE TABLE Mitarbeiter (  
...  
    Gehalt          NUMBER(12,2)  
    CONSTRAINT gehalt_ck (Gehalt > 0 AND Gehalt <= 90000),  
...  
);
```



In welchem Format ein **CHECK**-Constraint angelegt wird, ob als Inline oder als Out Of Line Constraint, spielt keine Rolle. Beide Formen sind möglich. Der Unterschied besteht darin, dass sich ein Inline Constraint nur auf die Spalte beziehen kann, mit deren Definition es angelegt wurde. Ein Out Of Line Constraint kann sich auf alle Spalten der Tabelle beziehen, mit der zusammen es angelegt wurde.

Um die Auswirkungen des obigen Merksatzes zu zeigen, wird das GEHALT_CK-Constraint ein wenig modifiziert. Es muss jetzt auch die Spalte PROVISION mit einbezogen werden. Das Gesamtgehalt eines Mitarbeiters darf 90.000 EUR nicht überschreiten, die Provision mit eingerechnet.

Listing 9.4: Ein komplexes **CHECK**-Constraint

```
ALTER TABLE Mitarbeiter  
ADD CONSTRAINT gehalt_ck CHECK (Gehalt > 0  
                                AND (Gehalt + (Gehalt * Provision / 100)) <= 90000);
```

9.2.2. Das NOT NULL-Constraint

Das **NOT NULL**-Constraint ist dafür zuständig sicherzustellen, dass beim Einfügen oder Ändern einer Tabellenzeile bestimmte Spalten immer einen Wert haben müssen.



Das **NOT NULL**-Constraint stellt eine Ausnahme zu allen anderen Constraints dar, denn es kann nur als Inline Constraint angelegt werden.

Beispiel 9.5 zeigt, wie ein **NOT NULL**-Constraint angelegt wird.

Listing 9.5: Ein **NOT NULL**-Constraint anlegen in Oracle

```
ALTER TABLE Mitarbeiter  
MODIFY Gehalt CONSTRAINT gehalt_nn NOT NULL;
```

Um ein solches Constraint wieder rückgängig zu machen, kann die folgende Kurzform verwendet werden:

Listing 9.6: Das Gegenteil von **NOT NULL**

```
ALTER TABLE Mitarbeiter  
MODIFY Gehalt NULL;
```

In den meisten DBMS wird ein **NOT NULL**-Constraint intern als **CHECK**-Constraint umgesetzt, weshalb [Beispiel 9.5](#) und [Beispiel 9.7](#) gleichbedeutend sind.

Listing 9.7: Die alternative Form eines **NOT NULL**-Constraints in Oracle

```
ALTER TABLE Mitarbeiter  
ADD CONSTRAINT gehalt_nn CHECK (Gehalt IS NOT NULL);
```

In beiden Fällen wird intern ein **CHECK**-Constraint, nach dem in [Beispiel 9.7](#) gezeigten Schema, angelegt. Auch in SQL Server ist dies der Fall. Im Gegensatz zu Oracle, muss bei SQL Server immer der Datentyp der Spalte mit angegeben werden, wenn eine Spalte ein **NOT NULL**-Constraint erhält.

Listing 9.8: Ein **NOT NULL** Constraint anlegen in SQL Server

```
ALTER TABLE Mitarbeiter  
ALTER COLUMN Gehalt NUMERIC(12,2) NOT NULL;
```

Listing 9.9: Die alternative Form eines **NOT NULL** Constraints in SQL Server

```
ALTER TABLE Mitarbeiter  
ADD CONSTRAINT gehalt_nn CHECK Gehalt IS NOT NULL;
```

Um in SQL Server eine Spalte mit einem **NOT NULL**-Constraint zu belegen, muss der Datentyp der Spalte mit angegeben werden, auch wenn dieser sich nicht ändern soll!



9.2.3. Das UNIQUE-Constraint

Das **UNIQUE**-Constraint hat die Aufgabe, dafür Sorge zu tragen, dass alle Werte, die in eine Tabellenspalte eingetragen werden, eindeutig sind.

In Oracle sind NULL-Werte eindeutig. Das heißt, in einer mit einem **UNIQUE**-Constraint belegten Spalte können beliebig viele NULL-Werte vorkommen. In SQL Server sind NULL-Werte nicht eindeutig. Somit kann in SQL Server nur ein NULL-Wert pro Tabellenspalte vorkommen, wenn die Spalte mit einem **UNIQUE**-Constraint belegt ist.



[Beispiel 9.10](#) zeigt, wie in Oracle und SQL Server ein **UNIQUE**-Constraint auf die Spalte SOZVERS NR der Tabelle MITARBEITER gelegt wird.

Listing 9.10: Ein UNIQUE-Constraint anlegen

```
ALTER TABLE Mitarbeiter  
ADD CONSTRAINT sozversnr_uk UNIQUE (SozVersNr);
```

Wie bereits beim **CHECK**-Constraint gezeigt, kann auch ein **UNIQUE**-Constraint als Inline Constraint erstellt werden. [Beispiel 9.11](#) zeigt diesen Vorgang. Die Syntax ist in Oracle und SQL Server gleich.

Listing 9.11: Ein **UNIQUE**-Constraint als Inline Constraint anlegen

```
CREATE TABLE Mitarbeiter (  
...  
    SozVersNr          VARCHAR2(20)  
    CONSTRAINT sozversnr_uk UNIQUE,  
...  
);
```

Oftmals genügt es nicht, wenn der Wert einer Spalte eindeutig ist. Es kann auch sein, dass die Kombination mehrerer Werte aus mehreren Spalten eindeutig sein muss. In so einem Fall kann ein **UNIQUE**-Constraint auch auf eine Kombination mehrerer Spalten gelegt werden, wie [Beispiel 9.12](#) zeigt.

Listing 9.12: Ein kombiniertes UNIQUE-Constraint anlegen

```
ALTER TABLE Mitarbeiter  
ADD CONSTRAINT mitarbeiter_uk UNIQUE (Vorname, Nachname, SozVersNr);
```

9.2.4. Das PRIMARY KEY-Constraint

Das **PRIMARY KEY**-Constraint hat eine ganz besondere Aufgabe. Es ist dafür zuständig, ein Attribut oder eine Gruppe von Attributen einer Tabelle als eindeutig zu kennzeichnen, um so ein Identifikationsmerkmal für jede Tabellenzeile einer Tabelle zu schaffen.

Die Nutzung von Primärschlüsseln ist notwendig, da es eine wesentliche Leistung eines relationalen Datenbankmanagementsystems ist, die Datenkonsistenz zu gewährleisten und hierzu gehört auch das Vermeiden von redundanten Datensätzen.



Der Unterschied zwischen einem **UNIQUE**-Constraint und einem **PRIMARY KEY**-Constraint ist, dass ein **PRIMARY KEY**-Constraint keine NULL-Werte zulässt. Ein **PRIMARY KEY**-Constraint ist eine Mischung aus einem **NOT NULL**- und einem **UNIQUE**-Constraint.

Da eine relationale Datenbank nicht ohne **PRIMARY KEY**-Constraints auskommt, werden diese meist schon bei der Erstellung einer Tabelle angelegt.

Listing 9.13: Ein PRIMARY KEY-Constraint als Inline Constraint anlegen

```
CREATE TABLE Mitarbeiter (
    Mitarbeiter_ID      NUMBER CONSTRAINT mitarbeiter_pk PRIMARY KEY,
    ...
);
```

Genau wie bei einem **UNIQUE**-Constraint, kann es notwendig sein, einen Primärschlüssel nicht nur auf eine Spalte, sondern auf eine Gruppe von Spalten zu legen. Dies ist meist in schwachen Entitäten der Fall, da hier die Kombination zweier Primärschlüssel aus den beiden äußeren Entitäten als Primärschlüssel genutzt wird.

Listing 9.14: Ein PRIMARY KEY-Constraint als Out Of Line Constraint auf mehrere Spalten anlegen

```
CREATE TABLE Mitarbeiter (
    Mitarbeiter_ID      NUMBER,
    Vorname             VARCHAR2(30),
    Nachname             VARCHAR2(35),
    ...
    CONSTRAINT mitarbeiter_pk
        PRIMARY KEY (Mitarbeiter_ID, Vorname, Nachname)
    ...
);
```

9.2.5. Das FOREIGN KEY-Constraint

In einem RDBMS steht üblicherweise keine Entität „einzeln im Raum“. Sie steht immer in Zusammenhang mit anderen Entitäten. Diese Zusammenhänge werden durch Foreign Key-Constraints dargestellt und überwacht.

Der Zusammenhang, in dem die Entitäten eines RDBMS stehen, wird als „Referentielle Integrität“ bezeichnet.



Ein Beispiel hierfür stellen die beiden Tabellen MITARBEITER und BANKFILIALE bereit. Sie stehen, durch die Spalte BANKFILIALE_ID, die in beiden Relationen vorkommt, in Zusammenhang zu einander. Dieser Zusammenhang besteht darin, dass jeder Mitarbeiter genau einer Bankfiliale zugeordnet ist. Das heißt, in die Spalte BANKFILIALE_ID der Tabelle MITARBEITER werden die Primärschlüsselwerte der Tabelle BANKFILIALE eingetragen, um so den Zusammenhang herzustellen. [Beispiel 9.15](#) zeigt, wie ein Fremdschlüsselconstraint angelegt wird.

Listing 9.15: Ein Foreign Key-Constraint als Out Of Line Constraint anlegen

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT mitarbeiter_filiale_fk
FOREIGN KEY (Bankfiliale_ID)
REFERENCES Bankfiliale(Bankfiliale_ID);
```

Die Definition eines Fremdschlüssels als Out Of Line Constraint hat zwei Teile:

- Die **FOREIGN KEY**-Klausel: Sie legt fest, welche Spalte die referenzierende Spalte ist.
- die **REFERENCES**-Klausel: Sie legt fest, welche Spalte referenziert wird. Bei dieser Spalte muss es sich um eine Primärschlüssel- oder **UNIQUE**-Spalte handeln.



Wird ein Fremdschlüssel als Inline Constraint bei der Erstellung der Tabelle miterstellt, entfällt die **FOREIGN KEY**-Klausel.



Es gibt zwei Situationen, die in einer relationalen Datenbank keines Falls auftreten dürfen:

- Ein referenzierter Primärschlüsselwert wird gelöscht. Beispiel: Eine Bankfiliale, in der sich noch Mitarbeiter befinden, wird aus der Tabelle BANKFILIALE gelöscht. Dies würde Datensätze in der Tabelle MITARBEITER zurücklassen, die sich auf eine Filiale beziehen, die gar nicht mehr existiert.
- In eine Fremdschlüsselspalte wird ein Wert eingetragen, der in der referenzierten Primärschlüsselspalte nicht vorkommt. Beispiel: Ein Mitarbeiter wird in die Bankfiliale mit der ID 300 aufgenommen, welche gar nicht existiert. Auch hier würde sich ein Angestellter auf eine Abteilung beziehen, welche es nicht gibt.

In beiden Fällen wäre die Referentielle Integrität der Datenbank verletzt, was zu Informationsverlust bzw. fehlerhafter Information führt. Dies zu vermeiden ist die Aufgabe des Foreign Key-Constraints.

Listing 9.16: Ein Foreign Key-Constraint als Inline Constraint anlegen

```
CREATE TABLE Mitarbeiter (
...
Bankfiliale_ID      NUMBER
CONSTRAINT mitarbeiter_filiale_fk
REFERENCES Bankfiliale(Bankfiliale_ID)
...
);
```


Der SQL-Standard kennt zwei Erweiterungen zum **FOREIGN KEY**-Constraint. Dies sind die beiden Klauseln **ON DELETE CASCADE** und **ON DELETE SET NULL**.

- **ON DELETE CASCADE**: Wird ein referenzierter Wert gelöscht, werden automatisch alle referenzierenden Werte mitgelöscht. Beispiel: Wird eine Filiale aus der Tabelle **BANKFILIALE** gelöscht, werden automatisch auch alle Mitarbeiter gelöscht, welche sich in dieser Filiale befinden.
- **ON DELETE SET NULL**: Wird ein referenzierter Wert gelöscht, werden automatisch alle referenzierenden Werte auf **NULL** gesetzt. Beispiel: Wird eine Filiale aus der Tabelle **BANKFILIALE** gelöscht, wird die **BANKFILIALE_ID** eines jeden Angestellten automatisch auf **NULL** gesetzt.

Beide Zusätze können sehr nützlich sein, bergen jedoch auch große Risiken in sich. Wird beispielsweise die **ON DELETE CASCADE**-Klausel zu unvorsichtig angewandt, kann es passieren, dass Daten gelöscht werden, die gar nicht gelöscht werden dürfen.

Die **ON DELETE SET NULL** ist nicht so radikal, wie **ON DELETE CASCADE**, aber auch sie ist nicht ganz ungefährlich. Wird ein referenzierter Wert gelöscht, werden alle referenzierenden Werte kaskadierend auf **NULL** gesetzt. Das hat zur Folge, dass plötzlich Datensätze bestehen, die keinen Bezug mehr zu anderen Datensätzen haben.

Sowohl bei der **ON DELETE CASCADE**- als auch bei der **ON DELETE SET NULL**-Klausel muss mit äußerster Vorsicht gearbeitet werden.



Beispiel 9.17 und Beispiel 9.18 zeigen, wie diese Klauseln angewandt werden.

Listing 9.17: Ein Foreign Key-Constraint mit **ON DELETE CASCADE**-Klausel

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT mitarbeiter_filiale_fk FOREIGN KEY (Bankfiliale_ID)
REFERENCES Bankfiliale(Bankfiliale_ID)
ON DELETE CASCADE;
```

Listing 9.18: Ein Foreign Key-Constraint mit **ON DELETE SET NULL**-Klausel

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT mitarbeiter_filiale_fk FOREIGN KEY (Bankfiliale_ID)
REFERENCES Bankfiliale(Bankfiliale_ID)
ON DELETE SET NULL;
```

9.2.6. Das SQL Server DEFAULT-Constraint

In Microsoft SQL Server werden Standardwerte als Constraints an eine Spalte angefügt. Das Anfügen eines Default-Constraints an eine Spalte während der Tabellenerstellung funktioniert genauso wie in Oracle.

Listing 9.19: Erstellen einer Tabelle mit einem Standardwert

```
CREATE TABLE
  Aktie ( Aktie_ID  NUMERIC ,
  Name      VARCHAR(25) ,
  Herkunft  VARCHAR(25) DEFAULT 'USA' ,
  WKN       NUMERIC ,
  ISIN      VARCHAR(12)
);
```

Der Unterschied zwischen Oracle und MS SQL Server zeigt sich aber, wenn ein Default-Constraint nachträglich hinzugefügt werden soll.

Listing 9.20: Tabellenspalte mit Standardwert hinzufügen in SQL Server

```
ALTER TABLE Aktie
ADD CONSTRAINT herkunft_dv
DEFAULT 'USA'
FOR Herkunft;
```

Anders als in Oracle muss für den SQL Server die **ADD CONSTRAINT**-Klausel benutzt werden.

9.3. Constraints umbenennen und löschen

9.3.1. Constraints umbenennen

Sowohl in Oracle als auch in SQL Server ist es möglich, ein Constraint umzubenennen.

Listing 9.21: Ein Constraint umbenennen in Oracle

```
ALTER TABLE Mitarbeiter
RENAME CONSTRAINT gehalt_ck TO gehalt_provision_ck;
```

Listing 9.22: Ein Constraint umbenennen in SQL Server

```
EXEC sp_rename 'gehalt_ck', 'gehalt_provision_ck', 'OBJECT';
```

9.3.2. Constraints löschen

Soll ein bereits bestehendes Constraint wieder entfernt werden, muss in Oracle und SQL Server die **DROP CONSTRAINT**-Klausel des **ALTER TABLE**-Kommandos genutzt werden.

Listing 9.23: Ein Constraint löschen

```
ALTER TABLE Mitarbeiter  
DROP CONSTRAINT mitarbeiter_filiale_fk;
```

Dies lässt sich auf alle fünf Constraintarten anwenden.

Enthält eine zu löschende Tabelle Primärschlüssel- oder UniqueConstraints, welche durch Fremdschlüssel anderer Tabellen referenziert werden, muss in Oracle zusätzlich die Klausel **CASCADE CONSTRAINTS** verwendet werden. Dadurch werden die Fremdschlüssel der anderen Objekte entfernt. In SQL Server müssen zuerst die referenzierenden Foreign Key Constraints gelöscht werden, ehe die Tabelle gelöscht werden kann.

Listing 9.24: Eine Tabelle mit Fremdschlüsselbeziehungen löschen

```
DROP TABLE Mitarbeiter CASCADE CONSTRAINTS;
```

9.3.3. Standardwerte in SQL Server löschen

Was Standardwerte sind, ist bereits aus dem vorhergehenden Kapitel bekannt. Wie sie in Oracle und in SQL Server angelegt werden ist ebenfalls bekannt. Was bisher noch nicht gezeigt wurde, ist, wie sie in SQL Server wieder gelöscht werden. Da in SQL Server ein Standardwert wie ein Constraint behandelt wird, muss auch die **DROP CONSTRAINT**-Klausel des **ALTER TABLE**-Statements verwendet werden, um einen Standardwert zu löschen.

Listing 9.25: Einen Standardwert in SQL Server löschen

```
ALTER TABLE Aktie  
DROP CONSTRAINT herkunft_dv;
```

