Datenbanken

Inhalt

Einleitung

Eine **Datenbank (DB),** auch **Datenbanksystem (DBS)** genannt, ist ein System zur elektronischen Datenverwaltung. Die wesentliche Aufgabe eines DBS ist es, große Datenmengen effizient, eindeutig und dauerhaft zu speichern und benötigte Teilmengen in unterschiedlichen, bedarfsgerechten Darstellungsformen für Benutzer und Programme bereitzustellen.

Ein DBS besteht aus zwei Teilen:

- Der Verwaltungssoftware - auch **Datenbankmanagementsystem (DBMS)** genannt
- der **Datenbank** im engeren Sinn – auch Datenbasis genannt (Menge der zu verwaltenden Daten)

Die Verwaltungssoftware organisiert intern die strukturierte Speicherung der Daten und kontrolliert alle lesenden und schreibenden Zugriffe auf die Datenbank. Zur Abfrage und Verwaltung der Daten dient eine **DB-Sprache** (z.B. SQL – Structured Query Language).

Datenbanksysteme gibt es in verschiedenen Formen. Die Art und Weise, wie ein solches System Daten speichert und verwaltet, wird durch das **Datenbankmodell** festgelegt. Die heute gebräuchlichste Form eines Datenbanksystems ist das **relationale DBS**.

Datenbanken werden heute für viele unterschiedliche Aufgaben verwendet. Hier ein paar Beispiele, um den Umfang der Anwendungen zu zeigen:

- Login-Daten, Foren, Gästebücher, … sowie Content Management Systeme (z.B. Wiki) für Webseiten
- Datenlogging von Sensoren (z.B. auf Raspberry)
- Produktion: Ressourcenplanung mit ERP-Systemen (Enterprise Ressource Planning), Dokumentation
- Auftragsverwaltung und Bestellwesen, Kunden- und Adressverwaltung, Rechnungserstellung
- Banken und Versicherungen: Kunden- und Kontoinformationen, Buchungen, …

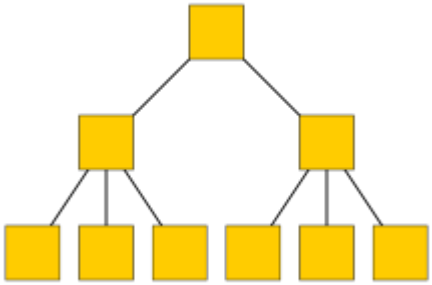
Datenbanksysteme sind heute ein zentraler Bestandteil der Unternehmenssoftware. Damit stellen sie einen kritischen Teil vieler Unternehmen und Behörden dar. Von der Verfügbarkeit, Vollständigkeit und Richtigkeit der Daten hängt die Aktionsfähigkeit eines Unternehmens ab. Die Datensicherheit ist daher ein wichtiger und sogar gesetzlich vorgeschriebener Bestandteil der IT eines Unternehmens oder einer Behörde.

Geschichte und Überblick der Systeme

Bereits in den 1960er Jahren wurde das Konzept eingeführt, Daten durch eine separate Softwareschicht zwischen Betriebssystem (Dateiverwaltung) und Anwendungsprogramm zu verwalten. Auslöser für diese Entwicklung waren Probleme bei der Verarbeitung von Daten in einfachen Dateien: diese sind meist für eine spezielle Anwendung entworfen. Im Laufe der Zeit wird dann oft viel Aufwand durch Umkopieren, Mischen und Restrukturieren der Dateien verwendet.

Hierarchische Datenbanken

Die ersten Datenbanken waren **hierarchisch** strukturiert:



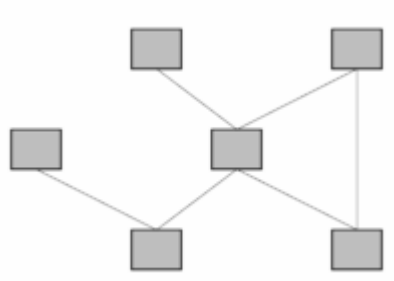
Dies entspricht einer Baumstruktur: Jeder Satz (Record) hat *genau* einen Vorgänger (mit Ausnahme der *Wurzel*).

Der Nachteil von hierarchischen Datenbanken ist, dass sie nur mit einem solchen Baum umgehen können. Verknüpfungen zwischen verschiedenen Bäumen oder über mehrere Ebenen innerhalb eines Baumes sind nicht möglich.

Anmerkung: eine XML-Datei (eXtended Markup Language) entspricht auch einer hierarchischen Struktur!

Netzwerkdatenbanken

Aufgrund der Einschränkungen von hierarchisch strukturierten Datenbanken hat sich Anfang 1970 auch das Konzept einer **Netzwerkdatenbank** entwickelt. Das Netzwerk-Modell fordert keine strenge Hierarchie und kann deswegen auch m:n-Beziehungen abbilden, d. h. ein Datensatz kann mehrere Vorgänger haben. Auch können mehrere Datensätze an oberster Stelle stehen. Es existieren meist unterschiedliche Suchwege, um zu einem bestimmten Datensatz zu kommen.



Das Konzept einer Netzwerkdatenbank wurde von einer amerikanischen Arbeitsgruppe, die auch für die Definition der Programmiersprache COBOL verantwortlich war, entworfen und enthält Vorschläge für drei verschiedene Datenbanksprachen:

- **Schema Data Description Language** (Schema-Datenbeschreibungssprache)
- **Subschema Data Description Language** (Subschema-Datenbeschreibungssprache)
- **Data Manipulation Language** (Datenmanipulationssprache).

Relationale Datenbanken und moderne Systeme

Bereits in den 1970er wurden auch die Grundlagen für **relationale** DBS gelegt (dazu später mehr). Damit wurden Firmen wie Oracle und IBM groß.

Diese relationalen DBS verdrängten in den 1980er Jahren die hierarchischen und netzwerkartigen Systeme. Während in den 1990er Jahren wenige kommerzielle Hersteller von Datenbank-Software faktisch den Markt beherrschten (neben IBM, Oracle, dBASE auch Microsoft mit dem SQL Server), erlangten seit den 2000ern die Open-Source-Datenbankmanagementsysteme eine immer größere Bedeutung.

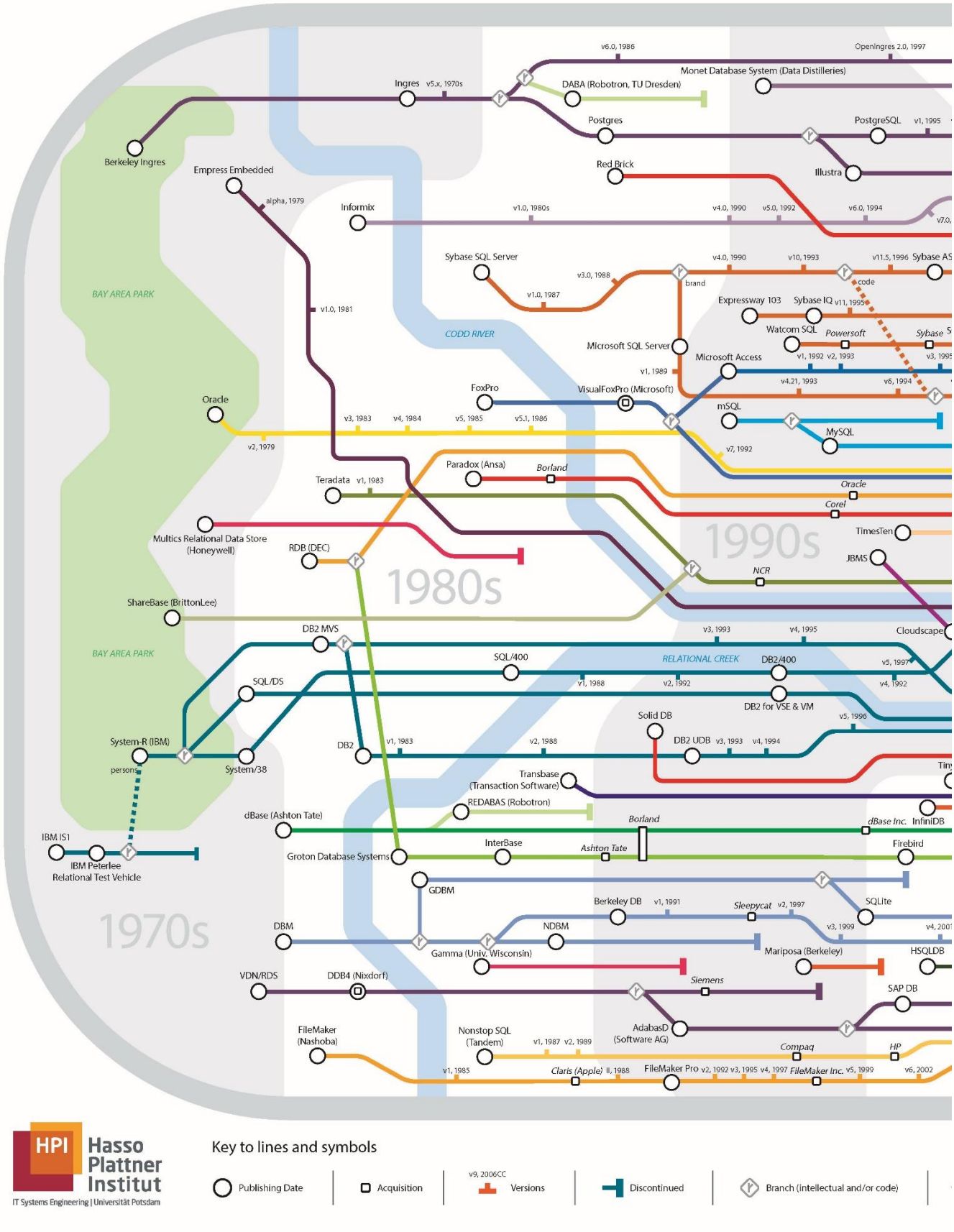
In der folgenden Übersicht der historischen Entwicklung und Zusammenhänge sei auf die Begriffe „Relational Creek“ und „CODD-River“ hingewiesen.

„Relational Creek“ (bedeutet „der kleine Fluss der relationalen Datenbanken“) trennt die alten, hierarchischen bzw. Netzwerk-Datenbanken von den modernen relationalen Datenbanken.

Der CODD-Fluss (benannt nach Edgar F. Codd, einem wichtigen Entwickler im Bereich relationaler Datenbanken) trennt vollständige Datenbanksysteme (oben) von Speziallösungen (unten).

Eine dieser Speziallösungen ist SQLite, welches das derzeit meistverwendete Datenbanksystem der Welt ist (Browser Firefox, Chrome, Safari für Lesezeichen, Cookies und Benutzerdaten, Skype für Kontakte und Chatprotokolle, etc.)

SQLite ist nur eine (wenige hundert Kilobyte große) Bibliothek, die sich direkt in entsprechende Anwendungen integrieren lässt, sodass keine weitere Server-Software benötigt wird. Dies ist der entscheidende Unterschied zu anderen Datenbanksystemen. Die SQLite-Datenbank besteht aus einer einzigen Datei, die alle Tabellen, Indizes, Views, Trigger usw. enthält. Dies vereinfacht den Austausch zwischen verschiedenen Systemen, sogar zwischen Systemen mit unterschiedlichen Byte-Reihenfolgen. Jede Spalte kann Daten beliebiger Typen enthalten, erst zur Laufzeit wird nötigenfalls konvertiert.


1970s
1980s
1990s
BAY AREA PARK
CODD RIVER
RELATIONAL CREEK
Ingres v5.x, 1970s
Berkeley Ingres
Empress Embedded
alpha, 1979
v1.0, 1981
Informix
v1.0, 1980s
v4.0, 1990
v5.0, 1992
v6.0, 1994
v6.0, 1986
DABA (Robotron, TU Dresden)
Postgres
Monet Database System (Data Distilleries)
Openingres 2.0, 1997
PostgreSQL v1, 1995
Illustra
Red Brick
Sybase SQL Server
v1.0, 1987
v3.0, 1988
v4.0, 1990
v10, 1993
v11.5, 1996
Sybase AS
brand
code
Expressway 103
Sybase IQ v11, 1995
Watcom SQL
Powersoft
Sybase
Microsoft SQL Server
v1, 1989
Microsoft Access
v1, 1992
v2, 1993
v4.21, 1993
v6, 1994
FoxPro
VisualFoxPro (Microsoft)
Oracle
v2, 1979
v3, 1983
v4, 1984
v5, 1985
v5.1, 1986
v7, 1992
mSQL
MySQL
Paradox (Ansa)
Borland
Teradata v1, 1983
Oracle
Corel
Multics Relational Data Store (Honeywell)
RDB (DEC)
TimesTen
JBMS
NCR
ShareBase (BrittonLee)
Cloudscape
DB2 MVS
v3, 1993
v4, 1995
SQL/400
v1, 1988
v2, 1992
DB2/400
v5, 1997
v4, 1992
SQL/DS
DB2 for VSE & VM
Solid DB
v5, 1996
System-R (IBM)
persons
System/38
DB2
v1, 1983
v2, 1988
DB2 UDB v3, 1993
v4, 1994
Transbase (Transaction Software)
REDABAS (Robotron)
dBase (Ashton Tate)
Borland
dBase Inc.
InfiniDB
IBM IS1
IBM Peterlee Relational Test Vehicle
InterBase
Ashton Tate
Firebird
Groton Database Systems
GDBM
Berkeley DB v1, 1991
Sleepycat v2, 1997
SQLite
v3, 1999
v4, 2001
DBM
NDBM
Gamma (Univ. Wisconsin)
Mariposa (Berkeley)
HSQLDB
VDN/RDS
DDB4 (Nixdorf)
Siemens
SAP DB
FileMaker (Nashoba)
Nonstop SQL (Tandem)
v1, 1987
v2, 1989
AdabasD (Software AG)
Compaq
HP
v1, 1985
Claris (Apple) II, 1988
FileMaker Pro v2, 1992 v3, 1995 v4, 1997
FileMaker Inc. v5, 1999
v6, 2002
Key to lines and symbols
Publishing Date
Acquisition
v9, 2006CC
Versions
Discontinued
Branch (intellectual and/or code)

Genealogy of Relational Database Management Systems

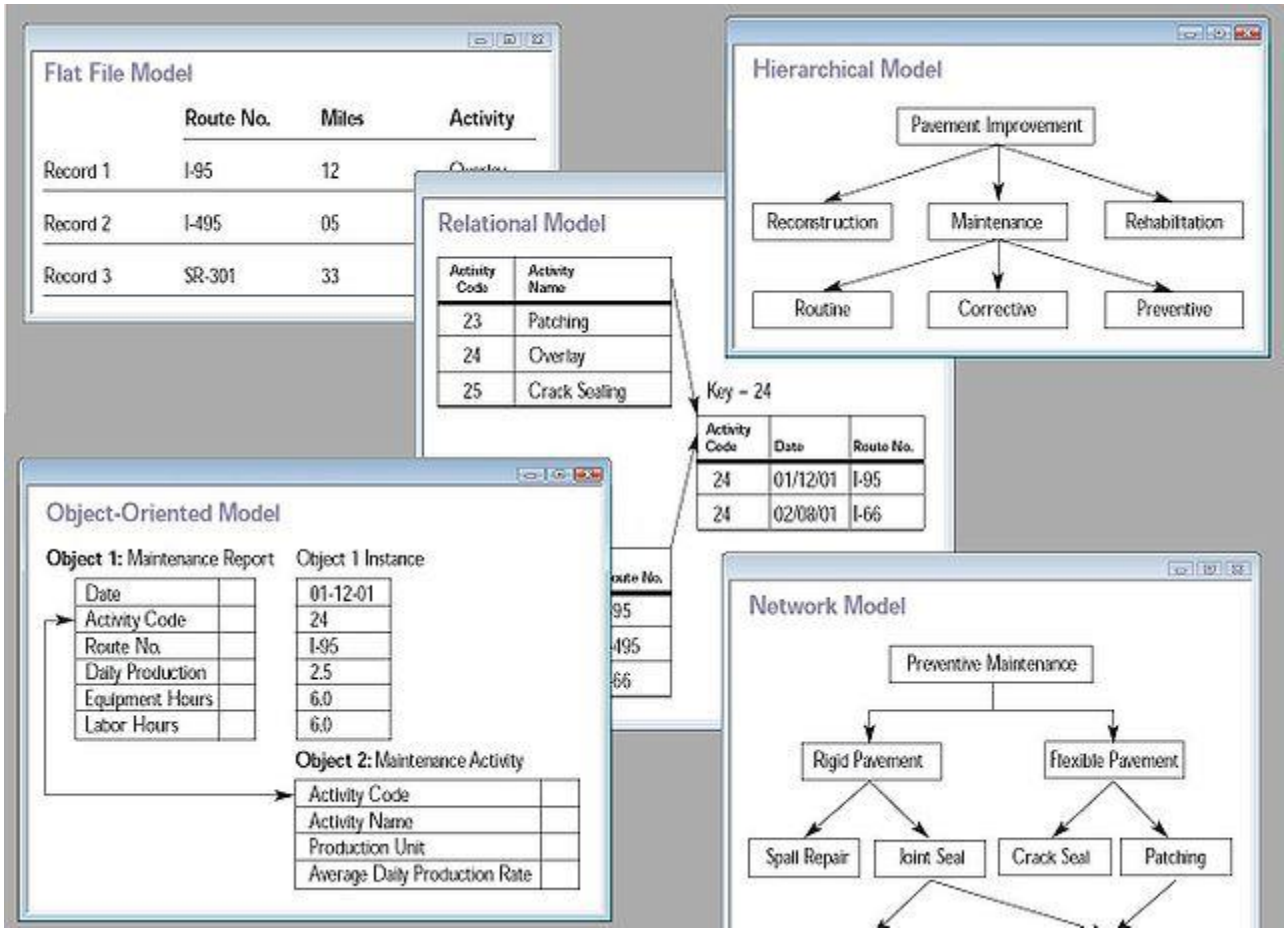
Crossing lines have no special semantics

Felix Naumann, Jana Bauckmann, Claudia Exeler, Jan-Peer Rudolph, Fabian Tschirschnitz
Contact - Hasso Plattner Institut, Potsdam, Felix.Naumann@hpi.uni-potsdam.de
Design - Alexander Sandt Grafik-Design, Hamburg
Version 5.0 - October 2015
http://www.hpi.uni-potsdam.de/naumann/projekte/rdbms_genealogy.html

Datenbankmodell

Ein Datenbankmodell legt die Infrastruktur fest, die ein bestimmtes DBS anbietet. Das bekannteste Beispiel für ein Datenbankmodell ist das relationale Datenbankmodell.

Weitere Beispiele:



Von Marcel Douwe Dekker - Eigenes Werk, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=5679857

Nach Edgar F. Codd definiert sich ein Datenbankmodell aus drei Eigenschaften:

1. **Generische Datenstrukturen**. Beispiel: eine relationale DB besteht aus Relationen mit eindeutigen Namen und jede Relation ist eine Menge von Datensätzen gleichen Typs. Die Struktur ist insofern generisch (allgemein nutzbar), als die Relationen und ihre Attribute (Spalten) beliebig gewählt werden können bzw. beim Einrichten der DB angegeben werden müssen.
2. **Generische Operatoren**, die man auf die Datenstrukturen anwenden kann (Daten eintragen, ändern, abfragen oder ableiten).
3. **Integritätsbedingungen**, mit denen man die zulässigen Datenbankinhalte über die Grundstrukturen (siehe 1.) hinaus weiter einschränken kann. Beim relationalen Datenbankmodell kann z. B. jedes Attribut einer Relation als eindeutig bestimmt werden; dann dürfen nicht zwei Datensätze dieser Relation den gleichen Wert in diesem Attribut haben (bei Eingabe des zweiten, gleichen Werts kommt eine Fehlermeldung).

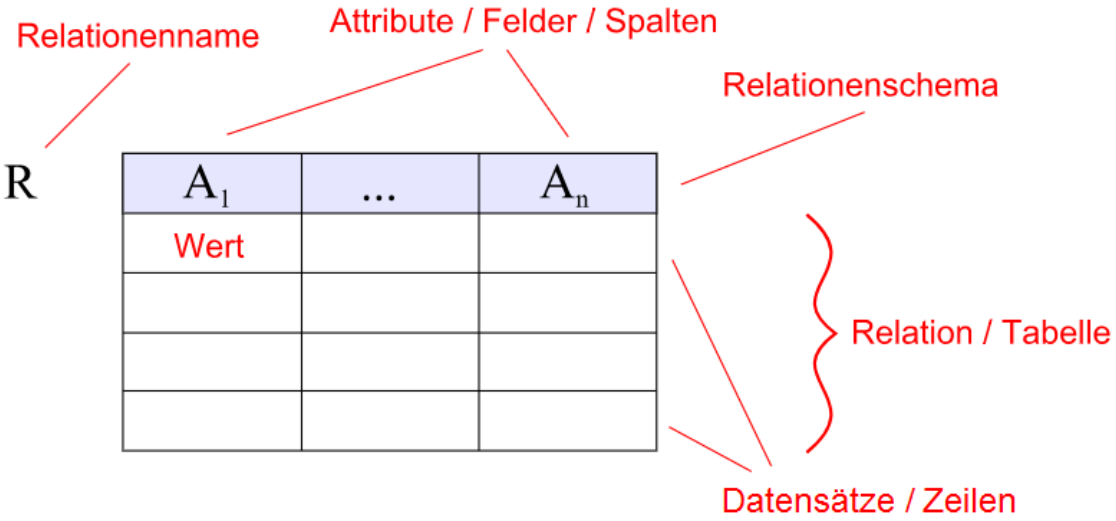
Relationale Datenbank

Eine **relationale Datenbank** beruht auf einem tabellenbasierten relationalen Datenbankmodell. Die Relation ist eine mathematische Beschreibung einer Tabelle.

Das zugehörige DBMS wird als **relationales Datenbankmanagementsystem** oder **RDBMS** (Relational Database Management System) bezeichnet. Zum Abfragen und Manipulieren der Daten wird überwiegend die Datenbanksprache SQL (Structured Query Language) eingesetzt, deren theoretische Grundlage die relationale Algebra ist.

Grundlagen

Eine relationale Datenbank kann man sich als eine Sammlung von Tabellen (den Relationen) vorstellen, in welchen Datensätze abgespeichert sind. Jede Zeile in einer Tabelle ist ein Datensatz (*record*). Jeder Datensatz besteht aus einer Reihe von Eigenschaftswerten (den Spalten der Tabelle).



Das Relationenschema legt dabei die Anzahl und den Typ der Eigenschaftswerte für eine Relation fest. Das Bild illustriert die Relation *R* mit Attributen A_1 bis A_n in den Spalten.

Beispiel

Zum Beispiel wird ein Buch in einer Bibliothek durch den Datensatz *(Buch-ID, Autor, Verlag, Verlagsjahr, Titel, Datum der Aufnahme)* beschrieben. Ein Datensatz muss eindeutig identifizierbar sein. Das geschieht über einen oder mehrere Schlüssel (oder auch *key*). In diesem Fall enthält *Buch-ID* die Schlüssel. Ein Schlüssel darf sich niemals ändern. Er bezieht sich auf den Datensatz und nicht auf die Position in der Tabelle.

Beispiel einer Relation „Buch“:

Buch-ID	Autor	Verlag	Verlagsjahr	Titel	Datum
1	Hans Vielschreiber	Musterverlag	2007	Wir lernen SQL	13.01.2007
2	J. Gutenberg	Gutenberg und Co.	1452	Drucken leicht gemacht	01.01.1452
3	G. I. Caesar	Handschriftverlag	−44	Mein Leben mit Asterix	16.03.−44
5	Galileo Galilei	Inquisition International	1640	Eppur si muove	1641
6	Charles Darwin	Vatikan-Verlag	1860	Adam und Eva	1862

Beziehungen zwischen Tabellen

Man kann nun Verknüpfungen nutzen, um die Beziehungen zwischen Tabellen auszudrücken. Eine Bibliothekdatenbank könnte mit drei Tabellen implementiert werden:

Tabelle *Buch* (siehe vorige Seite), die für jedes Buch eine Zeile enthält:

- Jede Zeile besteht aus den Spalten der Tabelle (Attributen): ***Buch-ID**, Autor, Verlag, Verlagsjahr, Titel, Datum der Aufnahme*.
- Als Schlüssel dient die ***Buch-ID***, da sie jedes Buch eindeutig identifiziert.

Tabelle *Nutzer*, die die Daten von allen registrierten Bibliotheksnutzern enthält:

- Die Attribute wären zum Beispiel: Nutzer-ID, Vorname, Nachname.

Relation „Nutzer“

Nutzer-ID	Vorname	Nachname
10	Hans	Vielschreiber
11	Jens	Mittelleser
12	Erich	Wenigleser

Außerdem braucht man eine dritte Tabelle *Entliehen*, die Informationen über die Verfügbarkeit des Buches enthält. Sie würde die Attribute *Nutzer-ID* und *Buch-ID* enthalten. Jede Zeile dieser Tabelle *Entliehen* ordnet einer Nutzer-ID eine Buch-ID zu.

Relation „Entliehen“

Nutzer-ID	Buch-ID
10	1
10	2
10	3
12	5
12	6

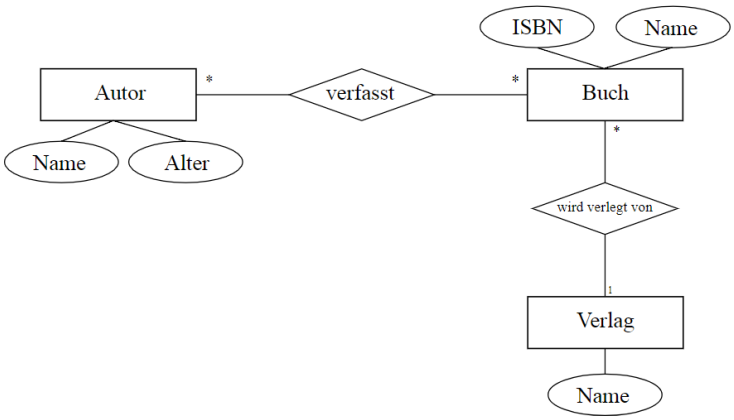
Der Eintrag *(10,3)* bedeutet, dass der Nutzer mit der ID *10* („Hans Vielschreiber“) das Buch mit der ID *3* („Mein Leben mit Asterix“) entliehen hat. Derselbe Nutzer hat auch das Buch „Drucken leicht gemacht“ entliehen (Tabelleneintrag *(10,2)*).

Als Schlüssel nimmt man hier die **Attributmenge** *(Nutzer-ID, Buch-ID)*. Gleichzeitig verbindet die *Nutzer-ID* jeden Eintrag der Tabelle *Entliehen* mit einem Eintrag der Tabelle *Nutzer*, sowie die Buch-ID jeden Eintrag von *Entliehen* mit einem Eintrag der Tabelle *Buch* verbindet. Deswegen heißen diese Attribute in diesem Zusammenhang Fremdschlüssel (engl. *foreign key*). Tabellen ohne Fremdschlüssel nennt man flache Tabellen.

Datenbankschema und Modellierung

Wichtiger Bestandteil einer relationalen Datenbank ist ihr *Schema*. Das Schema legt fest, welche Daten in der Datenbank gespeichert werden und wie diese Daten in Beziehung zueinanderstehen. Der Vorgang zum Erstellen eines Schemas nennt sich **Datenmodellierung**.

Zur Modellierung von relationalen DBs wird auch das Entity-Relationship-Modell verwendet. Es dient zum Entwurf eines konzeptuellen Schemas, welches unter Verwendung eines DBMS implementiert werden kann. Dieser Schritt wird als *logischer Entwurf* bzw. *Datenmodellabbildung* bezeichnet und hat als Ergebnis ein Datenbankschema im Implementierungsdatenmodell des DBMS.



Ein wichtiger Schritt des Modellierungsprozesses ist die **Normalisierung**. Diese soll Redundanzen verringern und Anomalien verhindern, um so die Wartung einer Datenbank zu vereinfachen, sowie die Konsistenz der Daten zu

gewährleisten. Edgar F. Codd hat dazu vier Normalformen vorgeschlagen, die seitdem bei dem relationalen Datenbankentwurf zum Einsatz kommen und um weitere ergänzt wurden.

Normalisierung

Unter Normalisierung einer relationalen DB (der Tabellenstruktur) versteht man die Aufteilung von Attributen (Tabellenspalten) in mehrere Relationen (Tabellen) gemäß den Normalisierungsregeln (siehe unten), so dass eine Form entsteht, die keine vermeidbaren Redundanzen (mehrfach vorhandene, gleiche Daten) mehr enthält.

Datenredundanzen führen dazu, dass bei Änderungen die mehrfach enthaltenen Daten nicht konsistent, sondern nur teilweise und unvollständig geändert werden (womit sie widersprüchlich werden, man spricht von auftretenden Anomalien). Außerdem konsumiert dies unnötig Speicherplatz.

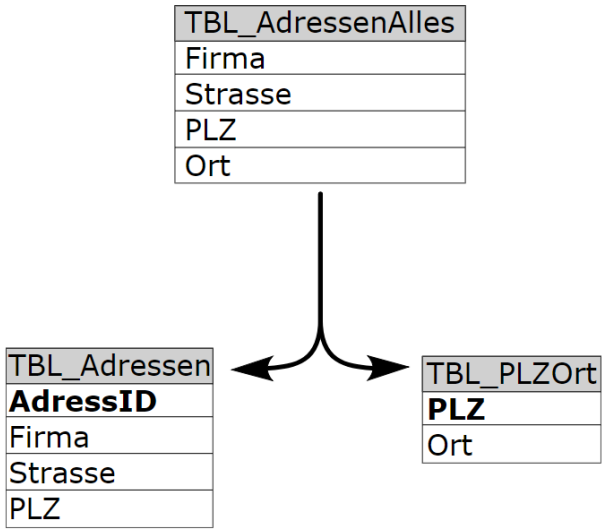
Es gibt verschiedene Ausmaße, in denen ein Datenbankschema gegen Anomalien gefeit sein kann. Je nachdem spricht man davon, dass es in erster, zweiter, dritter usw. Normalform vorliegt. Diese Normalformen sind durch bestimmte (formale) Anforderungen an das Schema definiert.

Normalisierung erfolgt, indem man die Relationen in einfachere zerlegt, bis keine weitere Zerlegung mehr möglich ist. Dabei dürfen jedoch auf keinen Fall Daten verloren gehen.

Normalisiert wird vor allem in der Phase des Entwurfs einer relationalen DB. Für die Normalisierung gibt es Algorithmen (Synthesealgorithmus (3NF), Zerlegungsalgorithmus (BCNF) usw.), die automatisiert werden können.

Beispiel

Eine DB enthält Kunden und deren Adressen sowie Aufträge, die den Kunden zugeordnet sind. Da es mehrere Aufträge vom selben Kunden geben kann, führt eine Erfassung der Kundendaten in der Auftragstabelle dazu, dass sie dort mehrfach vorkommen, obwohl der Kunde immer nur einen Satz gültiger Daten hat (Redundanz).



Beispielsweise kann es dazu kommen, dass in einem Auftrag fehlerhafte Adressdaten zum Kunden eingegeben werden, im nächsten Auftrag werden die korrekten Daten erfasst. So kann es – in dieser Tabelle oder auch gegenüber anderen Tabellen – zu widersprüchlichen Daten kommen. Die Daten wären dann nicht konsistent, man wüsste nicht, welche Daten korrekt sind.

Bei einer normalisierten Datenbank gibt es für die Kundendaten nur einen einzigen Eintrag in der Kundentabelle, mit der jeder Auftrag dieses Kunden verknüpft wird (üblicherweise über die Kundennummer).

Nachteile bei Normalisierung (Denormalisierung)

Die bei der Normalisierung angestrebte Redundanzfreiheit steht allerdings in speziellen Anwendungsfällen in Konkurrenz zur Verarbeitungsgeschwindigkeit oder zu anderen Zielen. Es kann daher sinnvoll sein, auf eine Normalisierung zu verzichten oder diese durch eine Denormalisierung rückgängig zu machen, um die Verarbeitungsgeschwindigkeit (Performanz) zu erhöhen oder Anfragen zu vereinfachen und damit die Fehleranfälligkeit zu verringern oder Besonderheiten von Prozessen (zum Beispiel Geschäftsprozessen) abzubilden.

In diesen Fällen müssen regelmäßig automatische Abgleichroutinen implementiert werden, um Inkonsistenzen zu vermeiden. Alternativ können die betreffenden Daten auch für Änderungen gesperrt werden.

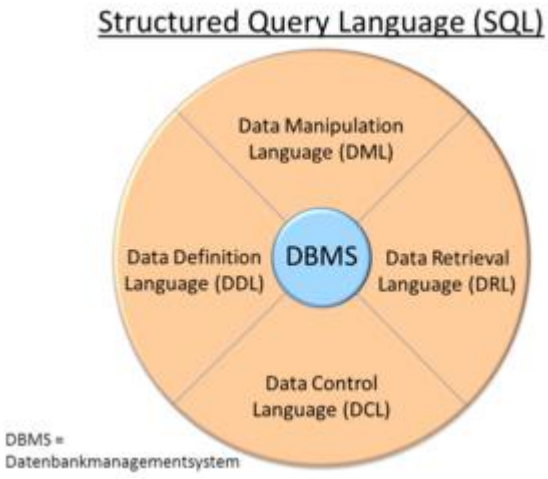
Datenbanksprachen

Mit Hilfe der Datenbanksprache kommuniziert ein Benutzer oder auch ein Programm mit der Datenbank, bzw. dem DBMS. Da ein wichtiger Teil der Arbeit mit Datenbanksystemen die Formulierung von Abfragen ist, gehört zum Sprachumfang in der Regel auch die (Datenbank-)Abfragesprache. Datenbanksprachen sind speziell auf die Anforderungen in diesem Umfeld (Datenbankerstellung, -pflege und -abfrage) zugeschnitten. Es handelt sich nicht um Programmiersprachen im heute geläufigen Sinne – es kann keine Anwendungssoftware damit geschrieben werden.

Es gibt viele DB-Sprachen, die oft auf bestimmte DBMS zugeschnitten sind. Eine normierte Sprache für die weit verbreiteten relationalen Datenbanksysteme ist SQL, das gleichzeitig die Obermenge vieler, proprietärer implementierter SQL-Dialekte ist. SQL selbst schreibt nicht vor, wie die Befehle implementiert werden, sondern lediglich, wie sich die Datenbank bei bestimmten Operationen nach außen verhält.

Man kann die Befehle in folgende Kategorien einteilen:

- Data Manipulation Language (**DML**, deutsch „Datenverarbeitungssprache“): für das Abfragen, Einfügen, Ändern oder Löschen von Nutzdaten
- Data Definition Language (**DDL**, deutsch „Datenbeschreibungssprache“): für das Anlegen, Ändern und Löschen von Datenstrukturen
- Data Control Language (**DCL**, deutsch „Datenaufsichtssprache“): für die Zugriffskontrolle



In SQL sind alle Elemente in einer Sprache (durch unterschiedliche Anweisungen) vereinigt. Bei alten Datenbanksystemen gab es für DML und DDL eigene Sprachen (z.B. DL/I und Assembler-Makros), die DCL wurde mit Betriebssystemmitteln realisiert.

Man nennt SQL auch eine „Sprache der vierten Generation“, weil nicht gesagt wird, wie etwas gemacht werden soll, sondern dass nur mitgeteilt wird, was zu tun ist.

1. Sprachen der ersten Generation oder Maschinensprachen: Programmiersprachen, die in Binärform, also als Bit-Folge von Nullen und Einsen geschrieben werden.
2. Sprachen der zweiten Generation oder Assemblersprachen: Sprachen der ersten Generation sind schlecht lesbar und damit fehleranfällig. Deshalb werden die rohen Maschinenbefehle in wiederkehrenden Gruppen zusammengefasst und mit englischsprachigen Begriffen benannt. Hinzu kommen Register, die feststehende Namen für Speicheradressen darstellen:

```
mov eax, 100          Wert 100 nach eax schreiben
add eax, 10           zum Wert in eax 10 addieren
```

3. Sprachen der dritten Generation oder höhere Programmiersprachen: Sprachen wie C, C++ oder Java, deren Quelltext von einem Interpreter direkt interpretiert oder einmalig von einem Compiler in eine ausführbare Datei übersetzt wird. Es stehen komplexe und selbst definierbare Datentypen sowie diverse Schleifenkonstrukte zur Verfügung. Der Quellcode ist in der Regel unabhängig von der Maschine, auf welcher er später ausgeführt wird. Sprachen der dritten Generation sind prozedurorientiert insofern, da das Programm sowohl Datenstrukturen als auch Zugriffsprozeduren selbst definiert und festlegt, wie etwas gemacht werden soll. Damit liegt die Verantwortung bsp. für effiziente Sortier- und Filteralgorithmen oder für die Erstellung eines leistungsfähigen Index beim Programmierer.

4. Sprachen der vierten Generation oder datenorientierte Programmiersprachen: sind dadurch gekennzeichnet, dass sie nicht mehr prozess-, sondern datenorientiert sind und dass beim Programmieren nicht mehr Schritt für Schritt festgelegt wird, wie etwas gemacht werden soll. Dies bedeutet eine Reduktion gegenüber den Sprachen der 3. Generation, da viele Details außerhalb des Einflussbereichs des eigenen Quellcodes liegen und beim Programmieren nur verwendet, jedoch nicht mehr beeinflusst werden können. Es führt auch zu einer Entlastung von wiederkehrenden Routineaufgaben wie dem Entwerfen einer Datenstruktur, dem Speichern und Laden aus der Datei oder der Entwicklung eines effizienten Sortieralgorithmus. Diese Aufgaben bleiben den Entwicklern der Programmierumgebung überlassen, sodass diese bei großen Datenmengen leistungskritischen Aufgaben durch das hierfür notwendige Spezialwissen abgedeckt sind.
5. Sprachen der fünften Generation oder Sprachen der Künstlichen Intelligenz: Im Gegensatz zu den Sprachen von der ersten bis zur vierten Generation, die sich an dem Aufbau von Rechenmaschinen orientieren und imperativ Probleme Befehl um Befehl abarbeiten, verarbeiten die Sprachen der fünften Generation Eingaben deklarativ entlang eines Systems aus Regeln und Schlussfolgerungen.

SQL - Auswahl und Aggregation vorhandener Daten (Data Manipulation Language - DML-I)

SELECT

Die SELECT-Anweisung ist fundamental für jedes Auswählen von Daten und stellt diese in Form einer virtuellen Tabelle zur Verfügung. Diese virtuelle Tabelle, auch Recordset genannt, also eine Menge (= Set) von Records (= Datenzeilen, Datensätzen), existiert zunächst nur temporär im Arbeitsspeicher und wird nach dem Ende der Befehlsausführung verworfen. Wird SELECT ohne weitere Ergänzungen verwendet, so werden die Daten angezeigt. Die Ausgabe kann auch mit SELECT ... INTO ... FROM in eine neue Tabelle kopiert oder mit INSERT INTO ... SELECT ... zu einer bestehenden Tabelle hinzugefügt werden.

Syntax

```
SELECT  [DISTINCT]
        <Name einer Spalte>
        <Konstante>
        <Berechnung>
        <einer der obigen Ausdrücke> As Spaltenalias
        [, weitere der obigen Ausdrücke]

FROM <Ausdruck, der eine Tabelle zurückgibt> As Tabellenalias

[WHERE ...]

[GROUP BY ...]

[HAVING ...]

[UNION [ALL]]

[Weitere SELECT-Anweisung, welche dieselbe Zahl von
        Spalten und Datentypen liefert]

[ORDER BY [Order-By-Ausdruck] ASC | DESC]]
        [, weitere Sortierungen]
```

Dies ist die grundlegende Syntax des SELECT-Befehls. Details zu den Ausdrücken nach FROM, WHERE, GROUP BY und HAVING finden Sie in den Abschnitten über JOIN, WHERE und GROUP BY. Die folgenden Beispiele behandeln nur den Abschnitt zwischen SELECT und FROM.

```
SELECT <Name einer Spalte - wie oben >

INTO <neue Tabelle>

FROM <Ausdruck, der eine Tabelle zurückgibt> As Tabellenalias
```

Beispiele

Einfache, kommagetrennte Auflistung der gewünschten Spalten, die im Tabellen-Ausdruck vorkommen. Ohne Alias für die Tabelle.

```
SELECT A_NR,
       A_NAME,
       A_PREIS
FROM ARTIKEL
```

Dasselbe wie im ersten Beispiel, aber mit Aliasname A für die Tabelle und aufsteigender Sortierung nach der Spalte A_PREIS.

```
SELECT A.A_NR,
       A.A_NAME,
       A.A_PREIS
FROM ARTIKEL As A
ORDER BY A.A_PREIS
```

Hier werden Alias-Ausdrücke für die Spalten verwendet - das Ergebnis kennt die drei Spalten Netto, MwSt und Brutto. MwSt ist ein konstanter Wert, in der Spalte Brutto wird der Inhalt von A_PREIS multipliziert mit einer Konstanten. Da ein solches Ergebnis keinen Spaltennamen hat, sollte dieser anschließend festgelegt werden. Das Ergebnis wird nach A_PREIS absteigend sortiert.

```
SELECT A_NR,
       A_PREIS  As Netto,
       0.19 As MwSt,
       A_PREIS * 1.19 As Brutto
FROM ARTIKEL
ORDER BY A_PREIS DESC
```

Hier wird für die Tabelle ein Alias A verwendet und mit * sämtliche Spalten ausgewählt. Das Ergebnis wird aufsteigend nach den Artikel-Namen, absteigend nach den Artikel-Preisen sortiert.

```
SELECT A.* FROM ARTIKEL As A
ORDER BY A.A_NAME ASC,
       A.A_PREIS DESC
```

Das Schlüsselwort DISTINCT entfernt alle mehrfach vorkommenden Zeilen mit Ausnahme einer. Diese Abfrage liefert deshalb nicht vier Zeilen mit doppeltem 'Oberhemd', sondern nur drei Zeilen zurück, eine Zelle mit dem Wert 'Oberhemd' wurde entfernt.

```
SELECT DISTINCT A.A_NAME
FROM ARTIKEL As A
```

Diese inhaltlich merkwürdige Abfrage liefert alle Artikel- und alle Vertreter-Namen in einer einzigen Liste aus. Da ALL fehlt, werden sechs Zeilen ausgegeben, das doppelte 'Oberhemd' wird nur einfach in das Resultset übernommen. Beachten Sie, dass die Datentypen übereinstimmen müssen und dass jede einzelne SELECT-Abfrage dieselbe Zahl von Spalten zurückliefern muss. Die Spaltennamen müssen allerdings nicht übereinstimmen, bei den SELECT-Anweisungen ab der zweiten Abfrage kann auf ALIAS-Namen verzichtet werden.

```
SELECT A.A_NAME
FROM ARTIKEL As A
UNION
SELECT B.V_NAME
FROM VERTRETER As B
```

Dies transferiert die virtuelle Tabelle in ein reales neues Tabellenobjekt. Zunächst wird die neue Tabelle 'Kopie-von-Vertreter' erstellt und anschließend mit den Zeilen gefüllt, die von der SELECT-Anweisung zurückgegeben wurden. Ergänzt man diese Abfrage um eine WHERE-Klausel 0 = 1, so werden keine Zeilen kopiert, es wird jedoch eine neue leere Tabelle erstellt.

```
SELECT V.*
        INTO [Kopie-von-Vertreter]
FROM VERTRETER As V
```

Wenn der Spalten-, Alias- oder Tabellenname Sonderzeichen, etwa ein Leerzeichen oder Minus (-) enthält, so schließen Sie den Ausdruck in eckige Klammern ein.

```
Select A.A_PREIS * 1.19 As [Aktueller Bruttopreis]
FROM ARTIKEL As A
```

Soll ein konstanter Text angegeben werden, so setzen Sie diesen in einfache Hochkommata. Dies liefert zwei Spalten, die erste Spalte heißt 'A_NAME', die zweite 'Info'. Das Ergebnis enthält vier Zeilen, in der ersten Spalte stehen die Werte aus der Tabelle, die zweite Spalte enthält viermal den Text 'aktuell'.

```
SELECT A.A_NAME, 'aktuell' As Info
FROM ARTIKEL As A
```

Benötigen Sie eine zusätzliche Zahl, etwa beim Vereinigen zweier Tabellen mit UNION, so können Sie diese einfach notieren. Eine solche Technik kann nützlich sein, wenn mit UNION die Ausgaben mehrerer SELECT-Anweisungen zusammengefügt werden und eine Information benötigt wird, aus welchem SELECT-Abschnitt die einzelne Zeile stammt.

```
SELECT A.A_NAME, 1 As [Index]
FROM ARTIKEL As A
UNION
SELECT B.V_NAME, 2
FROM VERTRETER As B
```

Verwendet man die Technik 'SELECT ... INTO ... FROM ...', um eine neue Tabelle zu erzeugen und enthält die ursprüngliche Tabelle eine automatisch hochzählende ID (Identity(1,1)), so wird dieser spezielle Spaltentyp auch in der Zieltabelle erzeugt. Soll dies vermieden werden, kann zur Identitätsspalte 0 hinzugefügt und der Spaltenname als Alias notiert werden. Nun wird die Spalte Umsatz_Nr der neuen Tabelle als Integer definiert.

```
SELECT U.Umsatz_Nr + 0 As [Umsatz_Nr], U.A_Nr
        Into [Umsatz-Kopie]
FROM UMSATZ As U
```

WHERE

Sollen von einer Tabelle nicht sämtliche, sondern nur wenige Zeilen zurückgegeben werden, so reduzieren Sie die Zahl der ausgegebenen Zeilen mit einer WHERE-Klausel. Die SELECT-Anweisung liefert eine virtuelle Tabelle, bestehend aus Zeilen und Spalten zurück. Der nach FROM folgende WHERE-Abschnitt kann Spaltennamen verwenden, um Bedingungen festzulegen. Für jede Zeile wird geprüft, ob die durch den Spaltennamen festgelegte Zelle die Bedingung erfüllt. Falls dies der Fall ist, wird die Zeile zur Ausgabe hinzugefügt, ansonsten wird diese Zeile nicht zum Resultset ergänzt.

Syntax

```
SELECT ...
FROM ...
WHERE <Bedingung 1> [<logischer Operator> <Bedingung 2>]
```

Es kann entweder nur eine Bedingung geben. Oder es werden mehrere Bedingungen angegeben, welche mit logischen Operatoren (NOT, AND, OR) miteinander verknüpft sind.

Zulässige Vergleichsoperatoren zwischen Spaltennamen und Ausdrücken, Konstanten und weiteren Spaltennamen:

```
=           Gleichheit
<>          verschieden
<           kleiner
>           größer
<=          kleiner oder gleich
>=          größer oder gleich
IS NULL     prüft, ob die Zelle leer ist
IS NOT NULL prüft, ob die Zelle einen Wert enthält
Between     Zwischen zwei Werten liegend
In          prüft, ob der linke Ausdruck in einem der rechten vorkommt
Like        Vergleich auf Textmuster
Exists      prüft, ob die folgende Unterabfrage mindestens eine Zeile zurückliefert
```

Zulässige logische Operatoren, die Ausdrücke mit Vergleichsoperatoren verknüpfen:

```
NOT         der folgende Ausdruck darf nicht erfüllt sein
AND         beide Bedingungen müssen erfüllt sein
OR          mindestens eine Bedingung muss erfüllt sein
```

Beispiele

Wählt die Zeile aus, bei welcher die Zelle A_NR den Wert 11 hat. Hier ist eine Bedingung und kein logischer Operator angegeben:

```
Select A.*
FROM ARTIKEL As A
WHERE A.A_NR = 11
```

Wählt alle Zeilen aus, deren A_NR verschieden von 11 und nicht leer (Not Null) ist:

```
WHERE NOT A.A_NR = 11
```

```
WHERE A.A_NR <> 11
```

Wählt jene Zeilen aus, bei welchen derzeit kein Preis definiert ist, bei denen die Zelle A_PREIS also leer ist. Beachten Sie, dass eine Zelle mit dem Wert 0 nicht leer ist, sondern den Wert 0 enthält. Ebenso ist bei Zellen mit Text die Belegung mit einer leeren Zeichenfolge (A_NAME = "" bzw. '') verschieden von der Zuweisung A_NAME = NULL bzw. der Abfrage A_NAME IS NULL:

```
WHERE A_PREIS IS NULL
```

Dies wählt nur jene Zeilen aus, bei denen sowohl der Wert in A_NAME gleich 'Oberhemd' und der Wert in A_PREIS kleiner als 40.00 ist. Es wird nur die Zeile mit A_NR = 12 ausgewählt:

```
WHERE A.A_NAME = 'Oberhemd' And A.A_PREIS < 40.00
```

Dies wählt jene Zeilen aus, bei denen der Wert in A_NAME gleich 'Oberhemd' oder der Wert in A_PREIS größer als 40.00 ist. Die einzige Zeile, die einen Preis < 40.00 hat, beschreibt ein 'Oberhemd', sodass von dieser Abfrage alle vier Zeilen zurückgegeben werden:

```
WHERE A.A_NAME = 'Oberhemd' Or A.A_PREIS > 40.00
```

Listet die beiden Zeilen auf, deren Preis zwischen 39.8 und 100.00 liegt. Die Randwerte werden mitgezählt, deshalb ist die Zeile A_NR = 12 im Ergebnis enthalten. Beachten Sie, dass das AND zum BETWEEN gehört und hier keine logische Bedeutung hat:

```
WHERE A.A_PREIS BETWEEN 39.8 AND 100.00
```
Bzw.: `WHERE 39.8 <= A.A_PREIS And A.A_PREIS <= 100`

Vergleicht den Wert von A_NAME mit jedem der in der Klammer angegebenen Werten. Stimmt er mit einem dieser überein, so wird die betreffende Zeile ausgegeben:

```
WHERE A.A_NAME IN ('Hose', 'Mantel', 'Strümpfe')
```
Der Ausdruck ist gleichwertig zu:

```
WHERE A.A_NAME = 'Hose' OR
       A.A_NAME = 'Mantel' OR
       A.A_NAME = 'Strümpfe'
```
Rechts kann, wie hier, eine Liste von Konstanten, berechneten Werten oder eine Unterabfrage notiert werden. Die Unterabfrage muss eine Spalte zurückliefern.

Dieser Ausdruck liefert immer False zurück, es werden also keine Zeilen ausgewählt. Dies kann verwendet werden, falls nur die Spaltennamen gewünscht sind. Analog liefert 0 = 0 oder TRUE alle Zeilen zurück, sodass eine solche WHERE-Klausel redundant ist:

`WHERE 0 = 1` oder `WHERE FALSE`

Mit LIKE können Zellen gegen Textmuster geprüft werden, ohne dass eine vollständige Übereinstimmung notwendig ist. Der Unterstrich (_) fungiert als Platzhalter für ein Zeichen, sodass der obige Ausdruck sowohl 'Meyer, Franz' als auch 'Meier, Franz' findet:

```
SELECT V.* FROM VERTRETER
WHERE V.V_NAME LIKE 'Me_er, Franz'
```

Das Prozentzeichen (%) schließt 0 bis mehrere Zeichen ein, sodass aus der Beispieldatenbank sowohl 'Meier, Franz' als auch 'Meyer, Emil' gefunden wird:

```
WHERE A.V_NAME LIKE 'Me%'
```

Suche nach den Sonderzeichen '_' und '%': Wenn Sie nach diesen Sonderzeichen selbst suchen möchten, dann setzen Sie diese in eckige Klammern ([]). Damit wird die eckige Klammer selbst zum Sonderzeichen, sodass auch eine Suche nach einer eckigen Klammer den Einschluss erfordert. Ansonsten erlauben eckige Klammern das Angeben eines Bereiches. Dieses Beispiel findet 'Hier gibt es Tulpen_und_Zwiebeln', nicht jedoch 'Hier gibt es Tulpen-und-Zwiebeln':

```
WHERE <Spaltenname> LIKE '%Tulpen[_]und[_]Zwiebeln%'
```

Entfernt man die Eckklammern, werden beide Einträge ausgegeben. Das folgende Beispiel findet 'Mayer 5', 'Mayer 6', 'Mayer 7', nicht jedoch 'Mayer 8':

```
WHERE <Spaltenname> LIKE 'Mayer [5-7]'
WHERE <Spaltenname> LIKE 'Mayer [567]'
```

Dies findet 'Mayer [5]':

```
WHERE <Spaltenname> LIKE 'Mayer [[]5]'
```

Mit Exists prüfen, ob eine Unterabfrage Werte enthält. Diese Abfrage liefert jene Artikel einmal (!) zurück, für die es Einträge in der Tabelle 'Umsatz' gibt:

```
SELECT A.A_NR
FROM ARTIKEL As A
WHERE EXISTS
        (SELECT B.UMSATZ_NR
        FROM UMSATZ As B
        WHERE B.A_NR = A.A_NR)
```

Oder:

```
SELECT DISTINCT A.A_NR
FROM ARTIKEL As A INNER JOIN UMSATZ As U
ON A.A_NR = U.A_NR
```

Die zweite Abfrage wird länger benötigen, da zunächst alle passenden Zeilen gesucht und mehrfache Einträge anschließend mit DISTINCT entfernt werden. EXISTS bricht dagegen ab, falls bereits eine einzige Zeile gefunden wurde.