

Data Engineer

Datenbanken: **SQL**

..

SQL – WH – Übungen



SQL – WH - Übungen



- ToDo's:
 - Lesen / durcharbeiten einer der beiden folgenden pdf's
 - Structured Query Language.pdf
 - SQL-Manual.pdf
 - MS Sql Server und SQL Managementstudio installieren
 - NORDWIND-DB – Beschreibung lesen:
 - Er-Nordwind_komplett.png
 - Nordwind-pdf
 - “Installieren” Nordwind – Beschreibung in Übungsaufgabe
 - Nordwind.bak
 - Nordwind.sql
 - Übungsaufgabe “SQL Uebungen Select NORDWIND.pdf” durcharbeiten
-
- Good luck und viel Vergnügen

Structured Query Language - SQL

SQL Sprachelemente

Kategorien von SQL Befehlen

- DML ... Data Manipulation Language: Befehle zur Datenmanipulation (Ändern, Einfügen, Löschen) und lesendem Zugriff (CRUD)
- DDL ... Data Definition Language: Befehle zur Definition des Datenbankschemas
- DCL ... Data Control Language: Befehle für die Rechteverwaltung und Transaktionskontrolle

SQL DML – Data Manipulation Language



- Befehle zur Datenmanipulation (Ändern, Einfügen, Löschen) und lesendem Zugriff (CRUD)
- Hauptfunktionen von RDBMS
 - (C)reate – Erstellen; in SQL: `INSERT`
 - (R)ead – Lesen; in SQL: `SELECT`
 - (U)pdate – Aktualisieren; in SQL: `UPDATE`
 - (D)elete – Löschen; in SQL: `DELETE`

SQL DDL – Data Definition Language



- Befehle zur Definition des Datenbankschemas
- Beispiele f. Ausprägungen:
 - SQL – in Form englischer Befehlsklauseln
 - XML Schema zur Beschreibung der Struktur v. XML Dokumenten
- Abgrenzung zu DML schwimmend
- In SQL `CREATE` und `INSERT` Statements zum Erstellen von Tabellen und Anlegen von Einträgen

SQL DCL – Data Control Language

- Befehle für die Rechteverwaltung und Transaktionskontrolle
- Datenüberwachungssprache einer DB
- Teil einer Datenbanksprache, der verwendet wird, um Berechtigungen zu vergeben oder zu entziehen
- SQL Kommandos:
 - GRANT
 - REVOKE

SQL Werkzeuge für Datenbanken



- Microsoft SQL Server Management Studio (MS-SSMS)
- Beispiel Datenbanken
 - Northwind
 - Custom ArztverwaltungDB
 - Beispiel-DB KursDB

MS-SSMS Details



SQL Beispiele

- Einfache select Anweisungen
- Joins
- Views
- create, delete, ... – kommen am Ende der Modellierung, wenn die konkrete DB aufgesetzt werden soll
- Beispiele größtenteils aus <https://www.webucator.com/tutorial/learn-sql/subqueries-joins-unions/using-joins-exercise.cfm#tutorial>

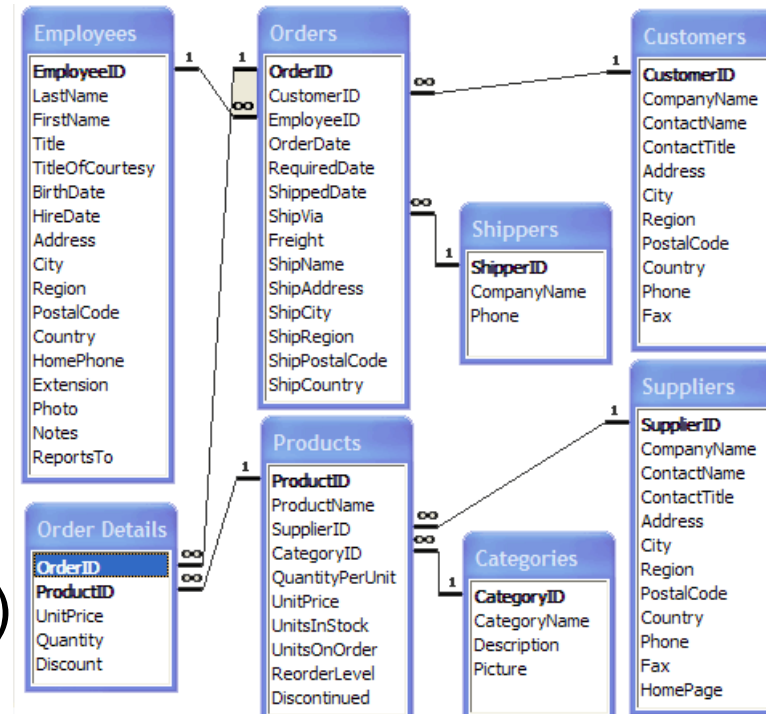
Northwind DB

Beinhaltet
Geschäftstransaktionen
zwischen Anbietern von
Produkten und
Northwind, sowie Kunden
von Northwind. Die Arbeit
der Mitarbeiter von
Northwind an einzelnen
Bestellungen wird auch
erfasst.



Northwind DB – Schema

Hier nur Ausschnitt als
Beispiel dargestellt.
Sollte für folgende
Beispiele ausreichend sein.
Automatische Generierung
des Diagramms über
MSSQL.
Wird in weiterer Folge
(wenn nicht anders
angegeben) für (praktische)
Beispiele verwendet.



AdventureWorks DB



- Modelliert Geschäftstätigkeit für Abenteuerurlaubsanbieter.

SQL Open Issues

- noch zu behandelnde Keywords:
 - `HAVING`
 - `LIMIT <n>` ähnlich zu `TOP (n)`, ... Beschreibungen suchen
 - `ALTER TABLE`
 - `PRINT` Statement
 - Kontrollstrukturen
 - `CASCADE`
 - Index
 - Cluster

SQL – Einfache Abfragen

- Für erste Beispiele wird die Northwind DB verwendet.
- Zeilenschaltungen in Abfragen dienen der besseren Lesbarkeit.
- Um alle Einträge einer Tabelle zu erhalten:

```
SELECT * FROM Employees;
```

- Um nur bestimmte Spalten auszugeben:

```
SELECT FirstName, LastName FROM Employees;
```

- Um nach Spalte zu sortieren (für auf- oder absteigend ASC od. DESC nachstellen)

```
SELECT FirstName, LastName FROM Employees ORDER BY LastName;
```


Die Anatomie einer einfachen SELECT Abfrage



```
SELECT [DISTINCT] [TOP(n)] Spalte(n) [AS Alias]
FROM Tabelle(n) [AS Alias] [[INNER|OUTER] JOIN Tabelle [AS
Alias] ON Felder]
[WHERE Bedingung(en)]
[GROUP BY Spalte(n) [HAVING Bedingung(en)]]
[ORDER BY Spalte(n) [ASC|DESC];
```

- Die Reihenfolge ist wichtig und unveränderbar
- Die Großkleinschreibung der SQL Keywords ist irrelevant, Spalten und Tabellennamen müssen genau den tatsächlichen Namen entsprechen
- Ein Befehlsblock schließt mit einem Semikolon (;) ab

Operatoren

Operator	Beschreibung
=	Gleichheit (a = b); Achtung, nicht == wie in den meisten Sprachen
<> bzw. !=	Ungleichheit (a <> b); Äquivalent in C# ist !=
>	Größer als (a > b)
<	Kleiner als (a < b)
>=	Größer als oder gleich (a >= b)
<=	Kleiner als oder gleich (a <= b)
BETWEEN	Zwischen (inklusive; a BETWEEN x AND y)
LIKE	Sucht nach bestimmten Muster (a LIKE muster), Muster sind Platzhalter
IN	Prüft auf Übereinstimmung mit Werten aus einer Liste (a IN (wert1, wert2, ...))

SQL – Einfache Abfragen

- Einschränkungen aufgrund von Bedingungen – `WHERE` Klausel
- Mögliche Operatoren: `=`, `<>` bzw. `!=`, `>`, `<`, `>=`, `<=`; können auf Texte wie auf Nummern angewandt werden
- Beispiel: Finde Verkaufsmitarbeiter

```
SELECT Title, FirstName, LastName FROM Employees WHERE Title = 'Sales Representative';
```

- `NULL` Abfrage: nimm Eintrag in Ergebnis auf wenn Bedingung (un-)gleich `NULL`
- Beispiel: Mitarbeiter die keiner Region zugeordnet sind

```
SELECT FirstName, LastName FROM Employees WHERE Region IS NULL;
```

SQL – Einfache Abfragen

- Kombination von WHERE und ORDER BY
- Finde alle Kunden die ein Fax haben (nach Firmenname sortiert)

```
SELECT CompanyName, ContactName, Fax
FROM Customers
WHERE Fax IS NOT NULL
ORDER BY CompanyName;
```

- Finde alle Kunden die in einer Stadt wohnen die mit 'A' od. 'B' beginnt

```
SELECT City, CompanyName, ContactName
FROM Customers
WHERE City < ,C`
ORDER BY ContactName DESC;
```

SQL Keywords in einer SELECT Abfrage (1)



SQL Klausel	Aktion	Erf.
SELECT	Gibt die Felder mit den gewünschten Daten an oder wählt mittles * alle Spalten aus	Ja
DISTINCT	Eliminiert redundante Datensätze (basierend auf ALLEN ausgewählten Spalten)	Nein
TOP (n)	Limitiert das Ergebnis auf n Datensätze, je nach Sortierung die ersten oder die letzten n	Nein
AS	Definiert ein Alias für das Vorangegangene Element (Tabelle oder Spalte)	Nein
FROM	Enthält die Tabelle(n) mit den Spalten, die in der SELECT-Klausel angegeben sind	Ja

SQL Keywords in einer SELECT Abfrage (2)



SQL Klausel	Aktion	Erf.
JOIN ... ON	Gibt die Art des Joins, zu verknüpfende Tabelle und aufgrund welches Kriteriums verknüpft wird an – siehe Abschnitt Joins für weitere Details	Nein
WHERE	Gibt Kriterien an, die ein Datensatz erfüllen muss, um in die Ergebnisse einbezogen zu werden	Nein
ORDER BY	Gibt die Sortierung der Ergebnisse an	Nein
GROUP BY	Gibt in einer SQL-Anweisung mit Aggregatfunktionen Felder an, die in der SELECT-Klausel nicht zusammengefasst werden (sondern danach gruppiert)	Nein
HAVING	(nur in Verbindung mit GROUP BY) Gibt Kriterien an, die eine GRUPPE erfüllen muss um in die Ergebnisse einbezogen zu werden	Nein

Achtung (1) !

- Die Syntax muss der Reihenfolge wie vorhin angegeben (SELECT FROM WHERE etc) folgen
- Trotz der Reihenfolge der Syntax ist die Reihenfolge der Evaluierung am Server anders (siehe nächste Seite)
 - Daher können gesetzte Aliase nicht immer durchgehend verwendet werden
- Jede Abfrage wird durch den DB Server optimiert, dh. selbst dieser Ablauf ist nicht notwendigerweise immer der Selbe

Achtung (2) !

- Die Ergebnisse die zurückgeliefert werden sind meistens nach dem PK (falls vorhanden) oder der Spalte ganz links sortiert
- Dies ist nicht notwendigerweise immer der Fall, da, wenn das Ergebnis nicht sortiert wird, die Reihenfolge davon abhängt wie der SQL Server die Abfrage optimiert

Der Ablauf einer SQL Abfrage

SELECT [DISTINCT] [TOP(n)] Spalte(n) [AS Alias]

7

FROM Tabelle(n) [AS Alias] [[INNER|OUTER] JOIN Tabelle [AS
Alias] ON Felder]

1

3

2

[WHERE Bedingung(en)]

4

[GROUP BY Spalte(n) [HAVING Bedingung(en)]]

5

6

[ORDER BY Spalte(n) [ASC|DESC]

8

Komplexere SELECT Abfragen

- Abfragen können auch geschachtelt werden, sodass im `FROM` Teil einer Abfrage keine Tabelle sondern eine andere `SELECT` Abfrage (welche als Ergebnis wiederum eine Tabelle ist) eingetragen wird.
- Jede Abfrage für sich muss nach dem korrekten Schema aufgebaut worden sein.
- Abfragen niedriger Hierarchie können Aliase von Abfragen höherer Hierarchie verwenden.
- Siehe Abschnitt „Verschachtelte Abfragen“ für weitere Details

SQL System-Funktionen

- Mathematische Funktionen

- `round(zahl)` – Rundet angegebene Dezimalzahl auf eine ganze Zahl
- `pi()` – Wert von pi
- `sin(zahl), cos(zahl), tan(zahl)` – Winkelfunktion für angegebenen Wert

- Funktionen für Zeichenketten (Strings)

- `length(string)` – Ermittelt die Länge der Zeichenkette
- `lower(string), upper(string)` – Wandelt angegebene Zeichenkette in Klein- bzw. Großbuchstaben um
- `substring(string, anfang, länge)` – Extrahiert Teilstück der angegebenen Zeichenkette, definiert durch Anfangsposition und Länge

Datumsfelder

- SQL Server kennt 6 Datums- und Uhrzeittypen
- Ein Datum kann immer mit einem Vereinheitlichten String im Muster 'JJJJMMTT' beschrieben werden, zB

```
SELECT *  
FROM Orders  
WHERE OrderDate = '19970403'
```

- Gibt alle Bestellungen vom 3. April 1997 zurück

Datumsfunktionen

- Die Verwendung von einem ganzen Datumsstring ist zwar intuitiv aber nicht immer notwendig oder praktisch
- Es gibt vordefinierte Funktionen um Teile eines datetime Feldes zu bekommen
- Speziell für ein Datum gibt es `DAY ()` , `MONTH ()` , und `YEAR ()`
- Geben jeweils den Teil des Datums zurück der dem Funktionsnamen entspricht
- Mit `GETDATE ()` kann man sich das aktuelle Datum geben lassen

Datumsfunktionen, Beispiel

Dadurch kann auf individuelle Tage oder Monate gefiltert werden

```
SELECT *  
FROM Orders  
WHERE YEAR(OrderDate) = 1997 AND (DAY(OrderDate)  
BETWEEN 1 AND 15)
```

Gibt alle Bestellungen aus dem Jahr 1997 zurück die in der ersten Hälfte jedes Monats getätigt wurden

SQL Aggregierung

Wird verwendet, um Ausgaben zu berechnen, die aus der Ergebnismenge mehrerer Felder stammt. Die fünf gebräuchlichsten Aggregierungsfunktionen sind:

- `COUNT()`
- `SUM()`
- `AVG()`
- `MAX()`
- `MIN()`

Beispiele:

- `SELECT COUNT(*) AS NumEmployees FROM Employees;`
- `SELECT AVG(UnitPrice) AS AveragePrice FROM Products;`

Im SQL Standard sind nur Aggregierungs-Funktionen festgelegt. Andere Funktionen sind Teil des DBMS und nicht des SQL Standards.

SQL Gruppierung

Mit der GROUP BY Klausel können Aggregierungsfunktionen auf Gruppen von Ergebniseinheiten basierend auf Spaltenwerten angewandt werden.

Beispiel: Finde Anzahl der Mitarbeiter pro Stadt:

```
SELECT City, COUNT(EmployeeID) AS NumEmployees  
FROM Employees  
GROUP BY City;
```


SQL Variablen

- Variablennamen sind immer mit vorangestelltem @ zu kennzeichnen.
- Variablen können über das `DECLARE` Keyword angelegt werden.
- Für das Belegen von Variablen wird das Keyword `SET` verwendet.

Beispiel:

```
declare @aVar integer  
set @aVar = 5
```

- Die Variable kann dann grundsätzlich überall verwendet werden.
- Es ist auch möglich ganze (Ergebnis-)Tabellen in Variablen zu speichern.

PRINT Statement

Mit dem Statement PRINT können Ausgaben auf der Kommandozeile erstellt werden. Beispiel:

```
PRINT 'Hello World,
```

Ähnlich wie in prozeduralen Programmiersprachen können durch Verkettung mehrere Variablen u.ä. ausgegeben werden. Dabei muss beachtet werden, dass Datentypen nicht automatisch in Character-Arrays gecastet werden. Beispiel

```
declare @aVar integer
```

```
set @aVar = 5
```

```
print 'A value ' + CAST(@aVar AS VARCHAR)
```

PRINT Statement

Eine andere Möglichkeit ist die Verwendung der Funktion `raiseerror()` (siehe auch unter Fehlerbehandlung). Dabei ist die Verwendung von Formatmodifizierern möglich, womit das Casting entfällt:

```
declare @aVar integer
set @aVar = 5
RAISEERROR('A value %d', 10, 1, @aVar)
```

SQL

Data Definition Language

(1)

Erstellen von Tabellen, Definieren von Constraints

Erstellen von Tabellen

- Beim Erstellen von Tabellen muss festgelegt werden wie die Tabelle heißt – alles andere ist Optional, wird aber meistens auch gleich beim Erstellen der Tabelle festgelegt
 - Spalten und deren Datentyp
 - Standardwerte und ob NULL-Werte erlaubt sind
 - Autowerte
 - Constraints
- TSQL weicht hier teilweise deutlich von ANSI SQL ab!

Tabellenerstellung (1)

- Tabellen werden mit `CREATE TABLE tabellenname` erstellt
- Es können auch direkt die Spalten definiert werden, hierbei muss zumindest der Datentyp angegeben werden,
- Zusätzlich können bestimmte Constraints (Bedingungen/Einschränkungen) vergeben werden

Constraints – Allgemein

- Constraints definieren Regeln für die Daten in Tabellen
- Sollen sicherstellen, dass keine falschen oder inkonsistenten Eingaben gespeichert werden können
- Aktionen die eine Constraint verletzen werden abgebrochen
- Constraints sind meistens Spaltenweise definiert, manchmal auch für ganze Tabellen

“Normale” Constraints

- NOT NULL – Die Spalte darf keine NULL Werte beinhalten
- UNIQUE – Werte in der Spalte dürfen sich NICHT wiederholen
- CHECK – Stellt sicher, dass alle Werte in einer Spalte eine bestimmte Bedingung erfüllen
- DEFAULT – Setzt einen Standardwert für eine Spalte wenn kein Wert angegeben wurde

“Spezielle” Constraints

- `PRIMARY KEY` – Legt den/die Primärschlüssel einer Tabelle fest; im Endeffekt eine Kombination aus `NOT NULL` und `UNIQUE`
- `FOREIGN KEY` – Fremdschlüssel; Spalte die einen oder mehrere Datensätze einer anderen Tabelle identifiziert
- `IDENTITY[(seed, increment)]` – Festlegen eines Autowerts; `seed`: Wert ab dem einzufügen begonnen wird; `increment`: Wert, um den letzter Eintrag in der Spalte erhöht wird
Bei `INSERT` Statements muss dieses Feld nicht mitgegeben werden
- `INDEX` – Indizes erlauben ein schnelleres Einfügen und Auslesen von Daten.
`CREATE [UNIQUE] INDEX index_name ON table_name (column1, column2, ...)`
Ein Index kann über mehrere Attribute erstellt werden.

Tabellenerstellung (2)

- Die grundlegende Syntax zur Definition der Spalten ist

```
spaltenname DATATYPE [IDENTITY(s,i)] [DEFAULT standardwert]  
[NULL | NOT NULL] [UNIQUE] [PRIMARY KEY] [CHECK (bedingung)]
```

```
CREATE TABLE Mitarbeiter (  
    maid int IDENTITY(10000,1) PRIMARY KEY,  
    vorname varchar(50) NOT NULL,  
    nachname varchar(100) NOT NULL,  
    geburtsdatum datetime NOT NULL CHECK (geburtsdatum >= '19000101'),  
    wohnort varchar(25) DEFAULT 'Wien'  
)
```

Datentypen am SQL Server 2012

- 7 Kategorien
 - Zahlen (exakt und ungefähr)
 - Zeichenfolgen (ASCII und Unicode)
 - Datum/Uhrzeit
 - Binäre Zeichenfolge
 - Andere Typen

Zahlen (Exakt)

Genaue Auflistung unter:
[https://technet.microsoft.com/en-us/library/ms172424\(v=sql.110\).aspx](https://technet.microsoft.com/en-us/library/ms172424(v=sql.110).aspx)

Datentyp	Wertebereich	Größe
Bigint	$-2^{63} - 2^{63}-1$	8 Bytes
Int	-2,147,483,648 – + 2,147,483,647	4 Bytes
Smallint	-32,768 – + 32,767	2 Bytes
Tinyint	0 – 255	1 Byte
Bit	0 – 1	1 Byte/8 Bits
Decimal	$-10^{38} - +10^{38}-1$	5 – 17 Bytes
Numeric	Same as Decimal	Same as Decimal
Money	$-2^{63} / 10000 - +2^{63} / 10000$	8 Bytes
Smallmoney	-214,748.3648 - +214,747.3647	4 Bytes

Zeichenfolgen (ASCII und Unicode)

Datentyp	Anzahl Zeichen	Größe
Char	0 – 8000	1 Byte / char
Varchar	0 – 8000	1 Byte / char * 2
Varchar(max)	0 – 2 ³¹ -1	1 Byte / char * 2
Nchar	0 – 4000	2 Byte / char
Nvarchar	0 – 4000	2 Byte / char * 2
Nvarchar(max)	0 – 2 ³⁰ -1	2 Byte / char * 2
Text		
NText		

Binärzeichenfolgen

Datentyp	Anzahl Zeichen	Größe
Binary	0 – 8000 Bytes	1 Byte / Byte
Varbinary	0 – 8000 Bytes	1 Byte / Byte + 2 Bytes
Varbinary(max)	0 – 2 ³¹ -1	1 Byte / Byte + 2 Bytes
Image		2 ³⁰ -1 Bytes

Datum und Uhrzeit

Datentyp	Anzahl Zeichen	Größe
Date	0001-01-01 – 9999-12-31	3 Bytes
Time	00:00:00.0000000 – 23:59:59.9999999	5 Bytes
Datetime	1753-01-01 00:00:00.000 – 9999-12-31 23:59:59.999	8 Bytes
Smalldatetime	1900-01-01 00:00 – 2079-06-06 23:59	4 Bytes
Datetime2(n)	0000-01-01 00:00:00.0000000 – 9999-12-31 23:59:59.9999999	6-8 Bytes
Datetimeoffset	=Datetime2 + UTC Offset	8-10 Bytes

Benannte Constraints

- Prinzipiell bekommen alle Constraints einen automatisch generierten Namen – der kann mehr oder weniger aussagekräftig sein
- Constraint-Namen dürfen pro Datenbank nur einmal vorkommen
- Wird in der Fehlermeldung verwendet wenn ein Verstoß vorliegt
- Best Practice: Constraints selbst benennen, Schlüsselwort **CONSTRAINT**

```
spalte DATATYPE CONSTRAINT constraintname EIGENTLCONSTR const
```


Benannte Constraints, Beispiel



```
CREATE TABLE Mitarbeiter (  
    maid int IDENTITY(10000,1) CONSTRAINT PK_Mitarbeiter PRIMARY KEY,  
    vorname varchar(50) NOT NULL,  
    nachname varchar(100) NOT NULL,  
    geburtsdatum datetime NOT NULL  
        CONSTRAINT CK_Mitarbeiter_geburtsdatum CHECK (geburtsdatum >= '19000101'),  
    wohnort varchar(25)  
        CONSTRAINT DF_Mitarbeiter_wohnort DEFAULT 'Wien',  
    position varchar(25)  
        CONSTRAINT CK_Mitarbeiter_position CHECK (position <> 'Chef')  
        CONSTRAINT DF_Mitarbeiter_position DEFAULT 'Einsteiger',  
    gehalt decimal(7,2) NOT NULL  
        CONSTRAINT CK_Mitarbeiter_gehalt CHECK (gehalt > 0)  
        CONSTRAINT DF_Mitarbeiter_gehalt DEFAULT 0.00  
)
```

Das Primärschlüssel Constraint

- Wie vorhin gezeigt, reicht es das Schlüsselwort `PRIMARY KEY` bei der entsprechenden Spalte zu verwenden
- Alternativ: Festlegen des PK Constraints am Ende der Tabellendefinition mittels `PRIMARY KEY (spalte1, spalte2, spalteN, ...)`
 - Erlaubt es auch Composite Keys zu definieren
 - Bei Composite Keys werden alle Spalten automatisch `UNIQUE NOT NULL`
 - Kann wie jedes andere Constraint auch benannt werden
`CONSTRAINT name PRIMARY KEY (...)`

Das Primärschlüssel Constraint, Beispiele



```
CREATE TABLE pktest (  
    pktestid int,  
    pktestdat varchar(10),  
    PRIMARY KEY (pktestid)  
)
```

```
CREATE TABLE pktest (  
    pktestid int,  
    pktestdat varchar(10),  
    CONSTRAINT PK_pktest PRIMARY KEY (pktestid, pktestdat)  
)
```

Das NULL/NOT NULL Constraint

- Wird dieses Constraint nicht angegeben, so können Werte in dieser Spalte NULL sein (also nicht vorhanden sein) – kann mit NULL auch explizit angegeben werden
- NOT NULL erlaubt daher nicht, dass Werte fehlen bzw. NULL sind
- Betrifft sowohl neu eingefügte, als auch geänderte existierende Werte
- Wird als einziges Constraint NICHT extra benannt

Das UNIQUE Constraint

- Stellt sicher, dass alle Werte der Spalte unterschiedlich sind
- **UNIQUE** Spalten erlauben **EINEN EINZIGEN NULL Wert** (SQL Server)
aber NICHT MEHRERE
 - Kann über Umwege und abhängig von der Version (des Servers) umgangen werden
- Im Gegensatz zum PK können mehrere Spalten **UNIQUE** sein
- Konvention Constraintbenennung: `UQ_Tabelle_Spalte`

Das DEFAULT Constraint

- Falls beim Einfügen eines Datensatzes der Wert für die Spalte fehlt, nimmt diese den angegebenen Standardwert an

```
stadt varchar(50) DEFAULT 'Wien'  
wert int DEFAULT 100
```

- Unabhängig von der NULLbarkeit der Spalte
- Konvention Constraintbenennung: DF_Tabelle_Spalte
- Kann nur Inline angelegt werden, oder mit ALTER TABLE

Das CHECK Constraint (1)

- Schränkt die gültigen Werte einer Spalte auf die angegebenen ein

```
alter int CHECK (alter >= 18 OR alter <= 150)
```

```
land varchar(15) CHECK (land  
    NOT IN ('Nordkorea', 'Auenland'))
```

- Constraintbenennung: CK_Tabelle_Spalte

Das CHECK Constraint (2)

- Kann nicht nur für einzelne Spalten, sondern auch auf mehrere Spalten angewandt werden

```
CONSTRAINT CK_Person CHECK  
    (alter >= 18 AND beruf <> 'Schüler')
```

- Konvention Constraintbenennung: CK_Tabelle

Das Fremdschlüssel Constraint

- Fremdschlüssel können ähnlich wie Primärschlüssel vergeben werden
- Müssen allerdings zusätzlich beinhalten, welche Spalte aus welcher Tabelle referenziert wird
- Syntax in der Spalte:
`Spaltenname datentyp FOREIGN KEY REFERENCES tabelle(spalte)`
- Syntax am Ende:
`FOREIGN KEY (spalte) REFERENCES tabelle(spalte)`

Das Fremdschlüssel Constraint, Beispiele



```
CREATE TABLE student (  
    sid int,  
    vname varchar(25),  
    nname varchar(25),  
    abschluss int,  
    PRIMARY KEY (sid),  
    FOREIGN KEY (abschluss) REFERENCES abschluss(aid)  
)  
  
CREATE TABLE abschluss (  
    aid int,  
    bez varchar(50),  
    PRIMARY KEY (aid)  
)
```

Benennen der PK und FK Constraints (1)



- PK Constraints beinhalten einfach nur den Namen der Tabelle, im Fall eines Composite Keys können die Teilnehmenden Spalten eingefügt werden
 - `PK_Mitarbeiter`
 - `PK_Mitarbeiter_SVN_GebDat`
- FK Constraints beinhalten normalerweise die Namen der FK Tabelle und der PK Tabelle, in dieser Reihenfolge
 - `FK_student_abschluss`
- Falls ein FK keinen PK referenziert sondern ein `UNIQUE` Feld, dann können auch die Spaltennamen mit eingebaut werden
 - `FK_fktabelle_fkspalte_reftabelle_refspalte`

Benennen der PK und FK Constraints (2)



- Prinzipiell: Constraints abkürzen, zumindest Tabellennamen miteinbeziehen
- Bei FK Constraints nur die Spaltennamen angeben wenn PK und FK Namen nicht übereinstimmen
- Es gibt natürlich Alternativen, eine beliebte ist:
Tabelle_CT_Spalten
Bzw. für FK
Tabelle_Spalte_FK_Tabelle_Spalte

Nachträgliches Hinzufügen von Constraints



- Ein Primary Key wird nachträglich an eine Tabelle folgendermaßen angehängt:

```
ALTER TABLE <Tabellenname> ADD PRIMARY KEY(<attribute>)
```

Anmerkung: die Tabelle darf keine Einträge enthalten, die Constraint verletzen, sonst funktioniert das Statement nicht.

- Ein Check Constraint wird folgendermassen hinzugefügt:

```
ALTER TABLE <Tabellenname> ADD CHECK (<Check-Cond>)
```

SQL

Data Definition Language

(2)

Constraints nach Spalten, Modifizieren von Tabellen und Schemas

SQL Data Definition Language (DDL)



Ist der Teil von SQL der dazu verwendet wird, um eine Datenbank zu erstellen und aktuell zu halten.

Befehle in dieser Klasse von SQL Statements:

CREATE

Zum Erstellen von Tabellen. Wird auch für Funktionen, Prozeduren usw. verwendet.

INSERT

Um Daten in entsprechende Tabellen einzufügen.

UPDATE

Um bereits bestehende Datensätze mit neuen Werten zu belegen.

DELETE

Um bestehende Datensätze zu löschen.

DROP

Um existierende Tabellen zu löschen.

SQL Data Definition Language (DDL)



Ist der Teil von SQL der dazu verwendet wird, um eine Datenbank zu erstellen und aktuell zu halten.

Befehle in dieser Klasse von SQL Statements:

`CREATE`

Zum Erstellen von Tabellen. Wird auch für Funktionen, Prozeduren usw. verwendet.

`ALTER`

Ändern von bestehenden Tabellen, Funktionen, Prozeduren, Trigger, ...

Seit SQL Server 2016 SP1 existiert der zusammengesetzte Befehl `CREATE OR ALTER`, mit dem Anpassungen vorgenommen werden können, ohne dass vorher bestehende Einheiten gelöscht werden müssen.

Wo Constraints definiert werden können

- Wie vorhin gezeigt, können Constraints mit der jeweiligen Spalte definiert werden
 - Ausnahme 1: Ein `CHECK` Constraint das sich auf mehrere Spalten bezieht
 - Ausnahme 2: Zusammengesetzter PK
- Constraints können auch definiert werden, nachdem alle Spalten angegeben wurden
 - Ausnahme: `DEFAULT` kann nur mit der Spalte erstellt werden, oder erst als Modifizierung einer Tabelle hinzugefügt werden

Die Unterschiede

- Die Syntax bei ein paar Constraints ist leicht abgeändert wenn sie nicht direkt mit der Spalte selbst definiert werden
- Bei `PRIMARY KEY` muss die Spalte bzw. müssen die Spalten explizit angegeben werden
- Bei `FOREIGN KEY` muss zwischen `FOREIGN KEY` und `REFERENCES` die Fremdschlüsselspalte der Tabelle angegeben werden

Löschen und Leeren von Tabellen

- **Tabelle löschen:** `DROP TABLE tabelle1, tabelle2`
- Funktioniert NICHT, wenn über FOREIGN KEY Constraint andere Tabellen referenziert werden. Damit auch Referenzen gelöscht werden können, muss das Constraint KASKADIEREND aufgesetzt werden; Bsp.:
`FOREIGN KEY (abschluss) REFERENCES abschluss(aid)
ON DELETE CASCADE`
- **Tabelle leeren:** `TRUNCATE TABLE tabelle`
 - Entfernt alle Datensätze
 - Funktioniert nicht bei Tabellen die referenziert werden!

Bearbeiten von Tabellen/Spalten

- **Hinzufügen von Spalten:**

```
ALTER TABLE tabelle ADD  
    spalte datentyp [CONSTRAINT] [,  
    spalte2 datentyp, ...]
```

- **Ändern des Datentyps und/oder des NULL Constraints:**

```
ALTER TABLE tabelle  
    ALTER COLUMN spalte datentyp [NULL | NOT NULL]
```

- **Löschen einer Spalte:**

```
ALTER TABLE tabelle  
    DROP COLUMN spalte1[, spalte2, ...]
```

Bearbeiten von Constraints

- Hinzufügen von Constraints:

```
ALTER TABLE tabelle
```

```
ADD CONSTRAINT name CONST constraint
```

- Entfernen von Constraints:

```
ALTER TABLE tabelle
```

```
DROP CONSTRAINT constraint [, constraint2, ...]
```

Achtung, bei DEFAULT Constraints



- Hinzufügen von Default:

```
ALTER TABLE tabelle
```

```
ADD CONSTRAINT name DEFAULT wert FOR spalte
```

Was es zu beachten gilt

- Daten, Spalten, Tabellen, können nicht gelöscht oder bearbeitet werden wenn sie dadurch gegen Constraints verstoßen!
- Beim Hinzufügen von Constraints wird standardmäßig geprüft ob existierende Daten gegen die neuen Constraints verstoßen
 - Kann auch explizit mit `WITH CHECK` nach `ALTER TABLE tabelle` angegeben werden
 - Kann mit `NOCHECK` nach `ALTER TABLE tabelle` deaktiviert werden

Schemas (1)

- Ein Schema, ist eine Art Gruppierung für Datenbankobjekte
 - zB Tabellen, Views, Stored Procedures
- Vereinfacht die Rechtevergabe, hilft die Datenbank logisch zu organisieren
- Default-Schema: dbo, muss grundsätzlich nicht zwingend angegeben werden, bei manchen Konstrukten trotzdem nötig, z.B.: bei user-defined functions, da sonst im System Scope nach Funktion gesucht wird, wo sie nicht vorhanden ist.

Schemas (2)

- Wird ein eigenes Schema vergeben, dann muss es immer mit angegeben werden, zB eigenesSchema.Tabelle
- Ein Schema wird erstellt mit
`CREATE SCHEMA name`
- Ein Objekt kann einem Schema bei der Erstellung zugeordnet werden, z.B.: `CREATE TABLE schema.Tabelle (...)`

Schemas (3)

- Objekte die bereits existieren können das Schema wechseln
`ALTER SCHEMA neuesSchema`
`TRANSFER altesSchema.Tabelle`
- Schemas können mit `DROP SCHEMA` gelöscht werden, nur wenn diese keine Objekte beinhalten!
- Schemas sind keine Tabellen, keine Benutzer, keine Datenbanken – Schemas sind Schemas bzw. ähnlich zu Namespaces in Programmiersprachen (siehe `dbo.*`)!

Tabellen verknüpfen

Geschachtelte Abfragen

Finde über einen bekannten Wert aus einer Tabelle einen abhängigen Wert aus einer anderen Tabelle, z.B. welcher Kunde hat eine bestimmte Bestellnummer aufgegeben:

```
SELECT CompanyName
FROM Customers
WHERE CustomerID = (
    SELECT CustomerID
    FROM Orders
    WHERE OrderID = 10290
)
```

Geschachtelte Abfragen

Finde alle Kunden, die 1997 eine Bestellung aufgegeben haben.

```
SELECT CompanyName
FROM Customers
WHERE CustomerID IN (SELECT CustomerID
                     FROM Orders
                     WHERE OrderDate BETWEEN '1-Jan-1997' AND '31-Dec-1997');
```

Das Datum kann auch folgendermaßen angegeben werden:

```
'1997-01-01' AND '1997-12-31'
```

Union (Vereinigung)

Verbinde mehrere Tabellen zu einem Ergebnis.

```
SELECT CompanyName, Phone
FROM Shippers
      UNION
SELECT CompanyName, Phone
FROM Customers
UNION SELECT CompanyName, Phone
FROM Suppliers
ORDER BY CompanyName;
```

- Duplikate werden entfernt, bei Verwendung von `UNION ALL` bleiben Duplikate erhalten.
- Es müssen in jedem Teil-Query die zu selektierenden Attribute gleich sein!

Joins (Verbund)

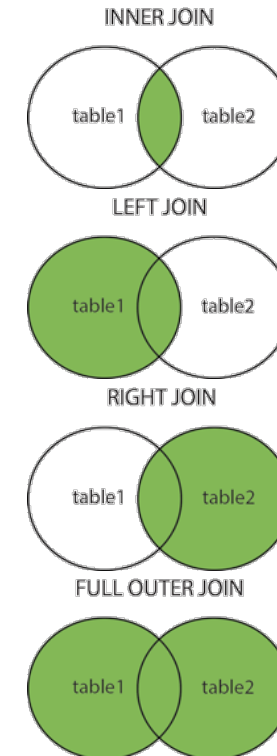
- Über Joins kann z.B., extrahiert werden
 - Welche Produkte von welchen Anbietern verkauft werden.
 - Welche Kunden welche Bestellungen getätigt haben.
 - Welche Kunden welche Produkte kaufen.

Dafür werden die Daten mehrerer Tabellen benötigt, die mit Join bereitgestellt werden können. Generelle Syntax:

```
SELECT table1.column, table2.column  
FROM table1 JOIN table2  
      ON (table1.column=table2.column)  
WHERE conditions
```

Joins (Verbund)

- *Inner Join*: Im Ergebnis werden nur Einträge berücksichtigt, bei denen die gegebenen Bedingungen von Einträgen beider Seiten erfüllt werden.
- *Left Join*: Datensatz aus linker Tabelle kommt auf jeden Fall in Ergebnis, rechte Seite nur wenn es Kriterien entspricht, sonst Einträge leer
- *Right Join*: wie Left Join, nur in die andere Richtung
- *Full (Outer) Join*: Im Ergebnis werden Einträge berücksichtigt, auch wenn die gegebenen Bedingungen von jeweils nur einer Seite erfüllt werden. Die jeweils leere Seite wird mit null gefüllt.



Joins (Verbund) – weitere Arten

- *Cross Join*: kartesisches Produkt aller Zeilen von Tabellen, d.h. es wird jedes Tupel einer Tabelle mit jedem Tupel einer anderen Tabelle verknüpft und als Tupel in die Ergebnismenge aufgenommen. Erzeugt sehr große Tabellen; für Checks von Serverleistungen. Beispiele:

`SELECT * FROM Tabelle1, Tabelle2;` ist dasselbe wie

`SELECT * FROM Tabelle1 CROSS JOIN Tabelle2;`

Anzahl der Elemente in der Ergebnismenge = Anzahl der Elemente in Tabelle1 MAL Anzahl der Elemente in Tabelle2.

Die Tabellen müssen keine übereinstimmenden Attribute haben!

- *Inner Join = Equi Join*: erlaubt nur Gleichheitsvergleich im Join-Prädikat. Andere Vergleichsoperatoren sind nicht möglich. Per default Standard Join Typ.

SQL Inner Join

Inner Join selektiert Einträge, bei denen Entsprechungen in beiden Tabellen vorhanden sind.

- **Syntax**

```
SELECT column_name(s)
FROM table1
INNER JOIN table2 ON table1.column_name =
table2.column_name
```

- **Beispiel**

```
SELECT o.OrderID, c.CompanyName
FROM Orders o
INNER JOIN Customers c
ON o.CustomerID = c.CustomerID
```

SQL Left Join

- **Syntax**

```
SELECT column_name (s)  
FROM table1  
LEFT JOIN table2 ON table1.column_name =  
table2.column_name
```

- **Beispiel**

```
SELECT c.CompanyName, o.OrderID  
FROM Customers c  
LEFT JOIN Orders o  
ON c.CustomerID = o.CustomerID  
ORDER BY c.CompanyName
```

SQL Right Join

- **Syntax**

```
SELECT column_name(s)  
FROM table1  
RIGHT JOIN table2 ON table1.column_name =  
table2.column_name;
```

- **Beispiel: Retourniere alle Beschäftigten und Bestellungen die sie bearbeitet haben:**

```
SELECT o.OrderID, e.LastName, e.FirstName  
FROM Orders o  
RIGHT JOIN Employees e  
ON o.EmployeeID = e.EmployeeID  
ORDER BY o.OrderID;
```

SQL (Full) Outer Join

- **Syntax**

```
SELECT column_name(s)  
FROM table1  
FULL OUTER JOIN table2  
ON table1.column_name = table2.column_name;
```

- **Beispiel: Erhalte alle Kunden und deren Bestellungen:**

```
SELECT c.CompanyName, o.OrderID  
FROM Customers c  
FULL OUTER JOIN Orders o  
ON c.CustomerID = o.CustomerID  
ORDER BY c.CompanyName;
```

SQL Self Join

Ergibt eine Verknüpfung einer Tabelle auf sich selbst. Wird bei reflexiven Relationen benötigt. Syntax wie bei Cross Join ohne speziellem Keyword.

- **Syntax**

```
SELECT column name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

- **Beispiel**

```
SELECT A.CompanyName AS CompanyName1, B.CompanyName AS
CompanyName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

Joins (Verbund)

Welcher Mitarbeiter hat welche Bestellung bearbeitet? Abfrage von Orders alleine gibt nur EmployeeID, von der man nicht weiss welcher Mitarbeiter das ist:

```
SELECT EmployeeID, OrderID FROM Orders;
```

Über Join wird Mitarbeiter-Tabelle verbunden, um Details zum Mitarbeiter mit jeweiliger ID zu erhalten.

```
SELECT e.EmployeeID, e.FirstName, e.LastName, o.OrderID, o.OrderDate  
FROM Employees e JOIN Orders o ON  
      (e.EmployeeID = o.EmployeeID)  
ORDER BY o.OrderDate;
```

Exkurs – Tabellen Alias

Die Verwendung der vollständigen Tabellennamen wird bei komplexen Abfragen oft unübersichtlich. Deshalb können für Tabellen Aliase vergeben werden; aus dem Beispiel von vorher:

```
SELECT e.EmployeeID, e.FirstName, e.LastName,  
       o.OrderID, o.OrderDate  
FROM Employees e JOIN Orders o ON  
     (e.EmployeeID = o.EmployeeID)  
ORDER BY o.OrderDate;
```

Bei der Verwendung von Aliases können die Originalnamen der Tabellen in der Abfrage nicht mehr verwendet werden.

Multi-Tabellen Joins

Es ist auch möglich mehrere Tabelle in einer Abfrage zu joinen. Abfrage kann sehr komplex werden und auch lange zur Ausführung benötigen.

Bericht zeigt OrderId, Name der Firma die bestellt hat und Vor- und Nachname des bearbeitenden Mitarbeiters. Nur Einträge nach dem ersten Jänner 1998 welche nach dem Bedarfsdatum versandt wurden, werden erfasst. Nach Firmennamen sortieren.

```
SELECT o.OrderID, c.CompanyName, e.FirstName, e.LastName
FROM Orders o
      JOIN Employees e ON (e.EmployeeID = o.EmployeeID)
      JOIN Customers c ON (c.CustomerID = o.CustomerID)
WHERE o.ShippedDate > o.RequiredDate AND o.OrderDate > '1-Jan-1998'
ORDER BY c.CompanyName;
```

Datenbank – Programmierung

SQL – Fortgeschrittene Konzepte

- Views
- Custom Functions
- Scalar valued functions (svf), Table valued functions (tvf)
- Stored Procedures
- Transactions (ACID Paradigma)
- Trigger
- Error Handling

Datenbank Programmierung



Effizientes Arbeiten mit Datenbanken



- Wir wollen nicht immer “manuell” Abfragen schreiben müssen
 - Kostet Zeit (einarbeiten in eine DB, formulieren der Abfrage, ...)
 - Tippfehler, semantische Fehler
- Automatisierung und Abstrahierung stehen im Vordergrund
- Wie wird in Programmiersprachen Automatisiert und Abstrahiert?

Abstrahierung und Automatisierung

- Wie wird in Programmiersprachen Automatisiert und Abstrahiert?
 - Bibliotheken, APIs
- Allgemeiner: Klassen, Methoden, Schleifen
- Warum? Wiederverwendbarkeit!
- Nicht nur in anderen Applikationen, sondern auch in der eigenen

Abstrahierung und Automatisierung in DBs



- Abstrahierung: Views (Sichten)
- Automatisierung (und Abstrahierung):
 - Functions (Funktionen)
 - Stored Procedures (gespeicherte Prozeduren)
- Welches Wissen fehlt uns dazu? Variablen/Parameter!

Datenbank – Programmierung

VIEWS

Abstrahierung durch Views (Sichten)



- Eine View kann man sich als virtuelle Tabelle oder gespeicherte Abfrage vorstellen
- Kann das Arbeiten mit der Datenbank vereinfachen
 - Verstecken von Komplexität
 - Wiederverwendung von Abfragen
 - Einfachere Rechtevergabe
 - Anbieten einer sich nicht-ändernden Schnittstelle, selbst wenn sich darunterliegende Tabellen ändern
- Wird im Endeffekt verwendet wie eine Tabelle

Arten von Views

- Vordefinierte Views, vom SQL Server erzeugt
 - Enthält Informationen über ihre Datenbank!
 - Erinnerung an die Tatsache, dass in einer relationalen Datenbank ALLE Informationen in Tabellen stehen müssen
- Userdefinierte Views
 - Simple Views (Views von einer Tabelle)
 - Complex Views (Views auf mehrere Tabellen)

Erstellen, Updaten und Löschen von Views



```
CREATE VIEW name AS  
SELECT...
```

```
ALTER VIEW name AS  
SELECT...
```

```
DROP VIEW name
```

Views können mittels einer SPROC auch umbenannt werden (sp_rename)

Views (Sichten)

- Virtuelle Relation bzw. virtuelle Tabelle
- Nicht alle Benutzer einer DB benötigen Zugriff auf gesamten Datenbestand.
- Über eine im DBMS gespeicherte Abfrage definiert
- Wenn Abfrage Sicht benutzt, wird diese zuvor durch das DBMS berechnet.
- Alias für eine Abfrage, nicht 1:1 dasselbe wie Alias für Tabellennamen.
- **Generelle Syntax:**

```
CREATE VIEW view_name AS  
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

Views (Sichten) – Beispiele

- View für alle Produkte die noch nicht abgesetzt wurden:

```
CREATE OR ALTER VIEW [Current Product List] AS  
SELECT ProductID, ProductName  
FROM Products  
WHERE Discontinued = 0;
```

Anzeigen: `SELECT * FROM [Current Product List];`

- View mit Produkten bei denen der Preis über dem Durchschnitt liegt:

```
CREATE OR ALTER VIEW [Products Above Average Price] AS  
SELECT ProductName, UnitPrice  
FROM Products  
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products);
```

Anzeigen: `SELECT * FROM [Products Above Average Price];`

Simple Views (1)

- Beziehen sich auf EINE EINZIGE Tabelle

```
CREATE VIEW uv_MitarbeiterAktiv
AS
SELECT MitarbeiterId, Vorname, Nachname
FROM Mitarbeiter
WHERE Aktiv = 1
```

- Achtung: Darf kein ORDER BY enthalten, wenn nicht TOP definiert ist!!!

Simple Views (2)

- Können wie eine Tabelle behandelt werden

```
SELECT * FROM uv_AktiveMitarbeiter
```

```
INSERT INTO uv_AktiveMitarbeiter  
VALUES ( 'Max', 'Mustermann' )
```

```
DELETE FROM uv_AktiveMitarbeiter  
WHERE Nachname = 'Mustermann' AND Vorname = 'Max'
```

Simple Views (3)

- Alle DML und DQL Statements funktionieren im Zusammenhang mit simplen Views
 - SELECT, INSERT*, UPDATE, DELETE, JOIN, ...
- Bezgl. INSERT gibt es folgende Bedingungen
 - Die View muss den PK beinhalten wenn dieser kein Autowert ist
 - Die View muss alle NOT NULL Spalten beinhalten
 - Dem PK muss je nachdem (ob identity oder nicht) auch ein Wert zugewiesen werden

Complex Views (1)

- Beziehen sich auf mehrere Tabellen

```
CREATE VIEW uv_MitarbeiterBuero
AS
SELECT MitarbeiterId, Vorname, Nachname, BueroId,
BueroName, Ort
FROM Mitarbeiter
INNER JOIN Buero ON Mitarbeiter.BueroId =
Buero.BueroId
```

Complex Views (2)

- Kann mit DQL wie eine Tabelle behandelt werden, NICHT mit DML
- Vereinfacht gesagt sind keine INSERT, UPDATE, DELETES möglich
- Genauer hingesehen: INSERT und UPDATE sind erlaubt, solange sie nur einzelne Tabellen betreffen, DELETES gar nicht

Verschachtelte Views

- Es ist möglich innerhalb einer View auf eine andere zu verweisen
- Dabei muss darauf geachtet werden, dass sich Spaltennamen nicht wiederholen!
- Es muss natürlich sichergestellt werden, dass alle Views jeweils selbst gültig sind!

Views Allgemein

- Normalerweise nur eingesetzt um mit ihnen via DQL zu interagieren
- Bezgl. DML gibt es weitere kleine Einschränkungen die hier nicht erwähnt wurden (für die meisten Zwecke auch nicht relevant!)
- Sind NICHT Parametrisierbar, man müsste also zB eine View für alle Verkäufe aus 2008, eine für alle aus 2009, eine für alle aus 2010, usw. erstellen
- Dafür gibt es andere Hilfsmittel: Funktionen und Prozeduren

Datenbank – Programmierung

VARIABLEN

Variablen

- Werden verwendet um für „beliebige“ Werte zu stehen
 - Einmal kann die Variable x den Wert 5 haben, ein anderes mal 10
 - Das Programmstatement Write(x) kann dabei unverändert bleiben!
- Dadurch können Sie als Parameter in Funktionen und Stored Procedures verwendet werden
- Funktionsweise in Datenbanken ist im Endeffekt genau die selbe
- Im Rahmen von Stored Procedures und Funktionen sprechen wir auch von Parametern (params)

Deklaration von Variablen

- Variablen müssen deklariert werden

```
DECLARE @varName DATATYPE  
DECLARE @stadt VARCHAR(50)
```

- Es können auch mehrere Variablen auf einmal deklariert werden

```
DECLARE @varName1 DATATYPE, @varName2 DATATYPE (  
    ...)  
DECLARE @stadt VARCHAR(50), @gehalt DECIMAL(8,2)
```

Wertezuweisung

- Werte können mit SET zugewiesen werden

```
SET @varName = WERT
```

```
SET @stadt = 'Wien'
```

- Deklarierte Variablen haben als Default-Wert NULL
- Variablen sollten daher immer gleich nach der Deklaration einen Wert zugewiesen bekommen

Skalare und Tabellenvariablen (1)

- Werte können auch durch SELECTs zugewiesen werden
- Achtung: eine Variable mit regulärem Datentyp (also einer der auch in einer Spalte vorkommen kann) kann nur einen skalaren Wert speichern

```
SELECT @MaIdHoechstesGehalt = MAX(EmployeeID)
FROM Mitarbeiter
ORDER BY Gehalt DESC
```

Skalare und Tabellenvariablen (2)

- Es gibt keine Arrays per se, aber es gibt Variablen vom Typ table
- Diese können, theoretisch, eine ganze Tabelle speichern, oder auch nur eine Liste an zB INTs; eher selten in Verwendung, werden auch nicht weiter besprochen

```
DECLARE @IdsMitarbeiterLondon TABLE (maid INT)
INSERT INTO @IdsMitarbeiterLondon
SELECT maid FROM Mitarbeiter WHERE stadt = 'London'
```

Einsatz von skalaren Variablen

- Können überall dort eingesetzt werden, wo wir entsprechende Werte erwarten

zB statt `SELECT * FROM Kunde WHERE Stadt = 'Wien'`

```
DECLARE @stadt VARCHAR(50)
```

```
SET @stadt = 'Wien'
```

```
SELECT * FROM Kunde WHERE Stadt = @stadt
```

➔ Wir benötigen keine neue Abfrage, wenn wir die Kunden aus einer anderen Stadt wissen wollen – man muss nur den Parameter ändern

Gültigkeit von Variablen

- Variablen werden NICHT zwischen unterschiedlichen Abfragen gespeichert!
- Variablen müssen also mit jeder Abfrage „neu“ deklariert und definiert werden
- Also bitte bedenken, den Code der die Variable deklariert und definiert, immer mit auszuführen!
- Ansonsten leben Variablen innerhalb
 - Des „Batches“ in dem sie deklariert wurden
 - Der Prozedur/Funktion in der sie deklariert wurden

SQL Server Batches

- Ein(e Gruppe von) Statement(s) die als Gruppe zum SQL Server geschickt werden
- Man kann Gruppen von Statements trennen, indem man das Schlüsselwort GO verwendet
- GO ist KEIN T-SQL Schlüsselwort, sondern ein Befehl an den SQL Server!

SQL Server Batches und Variablen

- Gewisse Statements müssen ALLEINE in einem Batch stehen (Wir kommen später darauf zurück)
- Oft ist es Usus, nach jedem kompletten Statement ein GO zu schreiben, auch wenn es nicht notwendig ist, um die Lesbarkeit des Codes zu erhöhen
- Variablen sind, unter anderem, nur bis zum nächsten GO gültig.

Datenbank – Programmierung Kontrollstrukturen

DB Programmierung – Kontrollstrukturen



- IF-ELSE
- CASE Bedingungen
- Schleifen

<https://stackoverflow.com/questions/4487546/do-while-loop-in-sql-server-2008>

IF-ELSE

Generelle Syntax

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE
    { sql_statement | statement_block } ]
```

Bemerkungen

- Aufgrund der überprüften Bedingungen werden die Kommandos in dem einen bzw. dem anderen Zweig ausgeführt.
- ELSE ist optional, je nach Anforderung des booleschen Ausdrucks bzw. der Bedingung.
- Wenn statement mehr als eine Zeile umfasst, muss der Block mit BEGIN und END zu Begin und Ende gekennzeichnet sein. Ansonsten wird nur die erste Zeile des Blocks dem IF (als auch einem ELSE) zugerechnet und es entstehen unerwartete Nebeneffekte.

IF-ELSE – Beispiel

Auf der AdventureWorks DB

```
DECLARE @maxWeight float, @productKey integer
SET @maxWeight = 1000.00
--SET @productKey = 727
SET @productKey = 828
IF @maxWeight <= (SELECT Weight from SalesLT.Product
                  WHERE ProductID = @productKey)
BEGIN
    (SELECT @productKey AS ProductKey, Weight,
    'This product is too heavy to ship and is only available for pickup.'
    AS ShippingStatus
    FROM SalesLT.Product WHERE ProductID = @productKey);
END
ELSE
BEGIN
    (SELECT @productKey AS ProductKey, Weight,
    'This product is available for shipping or pickup.'
    AS ShippingStatus
    FROM SalesLT.Product WHERE ProductID = @productKey);
END
```

IF-ELSE – Beispiel

```
CREATE PROCEDURE uspCalcVelocity @distance float, @time float, @velocity float
OUTPUT
AS
IF (@time = 0.00)
BEGIN
    -- we can't divide by zero, so assume time is 1 hour
    Select @time = 1;
    SELECT @velocity = @distance / @time;
END
ELSE
BEGIN
    SELECT @velocity = @distance / @time;
END
```

Aufruf über:

```
Declare @v float
EXECUTE uspCalcVelocity
        240.00, 0.00, @v OUTPUT
SELECT @v
```

CASE Bedingungen

Syntax allgemein:

```
CASE input_expression
    WHEN when_expression THEN result_expression [
...n ]
    [ELSE else_result_expression]
END
```

Siehe <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/case-transact-sql?view=sql-server-2017>

CASE Beispiel (1)



Auf der AdventureWorks DB

```
SELECT ProductNumber, Category =  
    CASE ProductCategoryID  
        WHEN '6' THEN 'Road'  
        WHEN '5' THEN 'Mountain'  
        WHEN '7' THEN 'Touring'  
        WHEN '2' THEN 'Other sale items'  
        ELSE 'Not for sale'  
    END,  
    Name  
FROM SalesLT.Product  
ORDER BY ProductNumber;  
GO
```

CASE Beispiel (2)



```
USE AdventureWorks2012
GO
SELECT ProductNumber, Name, „Price Range“ =
    CASE
        WHEN ListPrice = 0 THEN ,NA item - not for sale`
        WHEN ListPrice < 50 THEN ,Under $50`
        WHEN ListPrice >= 50 AND ListPrice < 250 THEN ,Under $250`
        WHEN ListPrice >= 250 AND ListPrice < 1000 THEN ,Under $1000`
        ELSE ,Over $1000`
    END
FROM SalesLT.Product
ORDER BY ProductNumber
GO
```

Kontrollstrukturen – while

Generelle Syntax

```
WHILE condition
BEGIN
    {...statements...}
END;
```

Beispiel

```
DECLARE @niter INT;
SET @niter = 0;
WHILE @niter <= 10
BEGIN
    PRINT concat('while loop iteration number', @niter);
    SET @niter = @niter + 1;
END;
PRINT concat('while loop terminated with iteration number', @niter);
GO
```

Kontrollstrukturen – while



Mit break

```
DECLARE @intFlag INT
SET @intFlag = 1
WHILE (@intFlag <=5)
BEGIN
    PRINT @intFlag
    SET @intFlag = @intFlag + 1
    IF @intFlag = 4
        BREAK;
END
GO
```

Mit continue und break

```
DECLARE @intFlag INT
SET @intFlag = 1
WHILE (@intFlag <=5)
BEGIN
    PRINT @intFlag
    SET @intFlag = @intFlag + 1
    CONTINUE;
    IF @intFlag = 4 -- will never execute
        BREAK;
END
GO
```


Kontrollstrukturen – while in einem Cursor



```
DECLARE contacts_cursor CURSOR FOR
SELECT contact_id, website_id
FROM contacts;
OPEN contacts_cursor;
FETCH NEXT FROM contacts_cursor;
WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM contacts_cursor;
        PRINT 'Inside WHILE LOOP on TechOnTheNet.com';
    END;
PRINT 'Done WHILE LOOP on TechOnTheNet.com';
CLOSE contacts_cursor;
DEALLOCATE contacts_cursor;
GO
```

Siehe auch Cursor in einem späteren Abschnitt.

SQL Kontrollstrukturen

- Können ähnlich wie in Programmiersprachen verwendet werden um den Weg von prozeduralen Abläufen zu verändern.
- Mögliche Kontrollstrukturen
 - IF-THEN-ELSE
 - WHILE
 - CASE

SQL Kontrollstrukturen – CASE

Generelle Syntax

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Folgende und weitere Beispiele unter

<https://docs.microsoft.com/en-us/sql/t-sql/language-elements/case-transact-sql?view=sql-server-2017>

SQL Kontrollstrukturen – CASE

Beispiel für einfache Fallunterscheidung

```
SELECT CASE
    WHEN Discontinued = 0 or UnitsInStock =
0
        THEN 1
        ELSE 0
    END as Saleable, *
FROM Products;
```

SQL Kontrollstrukturen – CASE

Beispiel für die AdventureWorks2008 DB

```
USE AdventureWorks2012;
GO
SELECT ProductNumber, Category =
    CASE ProductLine
        WHEN 'R' THEN 'Road'
        WHEN 'M' THEN 'Mountain'
        WHEN 'T' THEN 'Touring'
        WHEN 'S' THEN 'Other sale items'
        ELSE 'Not for sale'
    END,
    Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

SQL Kontrollstrukturen – CASE

Beispiel für die AdventureWorks2008 DB

```
USE AdventureWorks2012;
GO SELECT ProductNumber, Name, "Price Range" =
    CASE
        WHEN ListPrice = 0 THEN 'Mfg item - not for resale'
        WHEN ListPrice < 50 THEN 'Under $50'
        WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
        WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under
$1000'
        ELSE 'Over $1000'
    END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

Datenbank – Programmierung

FUNKTIONEN

Funktionen und Prozeduren

- Weiteres Werkzeug zwecks Abstraktion, Automatisierung, Modularisierung
- Können im Gegensatz zu Views mit Variablen arbeiten und sind daher auch Parametrisierbar!
- Funktionieren (fast) genau so wie Methoden in C#

Arten von Funktionen

- Vordefinierte Funktionen, vom SQL Server bereitgestellt
 - Aggregatfunktionen (zB MAX, AVG, COUNT, ...)
 - Skalare Funktionen (zB RAND, ROUND, ISNULL, ...)
- Userdefinierte Funktionen
 - Skalare Funktionen
 - Inline Table-Valued Funktionen
 - Multi-Statement Table-Valued Funktionen
- Definitionen siehe <https://docs.microsoft.com/en-us/sql/t-sql/statements/create-function-transact-sql?view=sql-server-2017>

Funktionen allgemein

- Geben einen einzigen Wert zurück
- Müssen genau einen Rückgabewert haben
- Können (fast) beliebig viele Parameter haben
- Können intern Variablen verwenden

Bekannte Programmierkonzepte

- Auch am SQL Server sind Konzepte die aus der C# Programmierung bekannt sein sollten verfügbar
 - Verzweigungen mit if/else
 - Verzweigung mit Switch/Case (hier: CASE/WHEN/THEN)
 - While-Schleifen while (for Schleifen nicht direkt verfügbar)
 - Inklusive break und continue
 - Theoretisch auch goto (goto = böse!)
 - Theoretisch auch Exceptionhandling mit Try/Catch (für Fortgeschrittene)
- Diese Konzepte können auch außerhalb von Funktionen und Prozeduren verwendet werden

Unterschiede zwischen den Funktionsarten

- Skalare Funktionen geben tatsächlich genau einen (skalaren) Wert zurück
- Table-Valued Funktionen geben eine Tabelle zurück (und daher indirekt mehr Werte)
- Die Inline Variante kann direkt in Abfragen wie eine Tabelle oder View angesprochen werden (kann aber im Gegensatz zur View parametrisiert werden)
- Die Multi-Statement Variante erlaubt komplexere Syntax beim Erzeugen der Tabelle (wird hier nicht weiter behandelt)

Erstellen & Aufrufen einer skalaren Funktion



--erstellen

```
CREATE FUNCTION name (@param1 DATENTYP, @param2
DATENTYP)
RETURNS datentyp
AS
BEGIN
    ...
END
```

--aufrufen

```
SELECT name (param1,param2)
```

Beispiel – Mitarbeiter ID mittels Name finden



```
ALTER FUNCTION uf_GetMitarbeitId(@vn VARCHAR(100), @nn VARCHAR(100))
RETURNS INT
AS
BEGIN
    DECLARE @returnWert INT;
    SELECT @returnWert = EmployeeId
    FROM Employees
    WHERE firstname = @vn AND lastname = @nn;
    RETURN ISNULL(@returnWert,-1);
END
GO
```

Was es zu beachten gilt

- Das letzte Statement vor dem END muss ein RETURN Statement sein
- Daher empfiehlt es sich, Werte nicht direkt zurückzugeben sondern immer den Wert in eine Variable zu speichern und diese dann am Ende zurückzugeben
- Vor allem wenn man mehrere Rückgabewerte haben kann durch zB If/Else

Inline Table-Valued Functions (ITVFs)



- Gibt keinen skalaren Wert sondern eine Tabelle zurück
- Ist also quasi eine parametrisierte View
- Indirekt also mehr als ein Rückgabewert, als ganzes gesehen dennoch nur ein Rückgabewert (EINE Tabelle!)
- Erlaubt ist nur EIN SELECT Statement, sonst nichts (JOINS, WHERE, GROUP BY etc sind natürlich erlaubt)

Erstellen & Aufrufen einer skalaren Funktion



--erstellen

```
CREATE FUNCTION name (@param1 DATENTYP, @param2
DATENTYP)
RETURNS datentyp
AS
BEGIN
    ...
END
```

--aufrufen

```
SELECT name (param1,param2)
```

Erstellen & Aufrufen einer ITV Funktion (1)



--erstellen

```
CREATE FUNCTION name (@param1 DATENTYP, @param2 DATENTYP)
RETURNS TABLE
AS
RETURN (...)
BEGIN -- nicht bei tvf
    ...
END - nicht bei tvf
```

--aufrufen

```
SELECT * FROM name(param1,param2)
```

Erstellen & Aufrufen einer ITV Funktion (2)



- Der Rückgabebetyp einer ITV ist immer TABLE
- Da außer einem SELECT Statement nichts anderes erlaubt ist, benötigt man kein BEGIN und END
- Da außer einem SELECT Statement nichts anderes erlaubt ist, und die Tabelle direkt zurückgegeben wird, kommt das RETURN Statement direkt nach dem AS
- Aufgerufen/Verwendet kann/wird eine ITVF wie eine View

Erstellen & Aufrufen einer table-valued Funktion



--Definition

```
CREATE (OR ALTER) FUNCTION tabEx(@id nchar(5), @city nvarchar(15))
RETURNS TABLE
AS
RETURN
SELECT *
FROM Customers
WHERE CustomerID = @id OR City = @city;
GO
```

--Aufruf

```
select * from dbo.tabEx('chops', 'Sao Paulo');
```

Multi Statement Table Valued Functions (MSTVF)

- Gibt wie eine Table Valued Function ebenfalls eine Tabelle zurück.
- Kann eines oder mehrere T-SQL Statements beinhalten.
- Die Struktur der zurückgegebenen Tabelle muss innerhalb des CREATE Teils spezifiziert werden.

Unterschiede ITVFs und MSTVFs

- Beide produzieren virtuelle Tabellen
- Inline Table-Valued Functions werden im großen und ganzen wie Views behandelt und sind ähnlich performant
- Multi-Statement Table-Valued Functions erlauben es dafür komplexere Statements zu verwenden um virtuelle Tabellen zu erstellen
- MSTVFs sind syntaktisch auch anders aufgebaut (benötigen eine Rückgabeveriable, benötigen BEGIN/END, ...)

Systemfunktionen – String Handling

- `numeric()`, `decimal()` – Formatiert die Ausgabe numerischer Werte durch Angabe der Nachkommastellen bei nicht ganzzahligen Werte.
- `concat()` – verknüpft übergebene Parameter zu einem String; einzelne Parameter können von unterschiedlichen Typen sein.
- `upper()` – ändert alle im Parameter übergebenen Zeichen in Großbuchstaben.
- `lower()` – ändert alle im Parameter übergebenen Zeichen in Kleinbuchstaben.
- `substring()` – extrahiert einen Unterstring aus dem übergebenen Parameter.
- `len()` – gibt die Anzahl der Zeichen des als Parameter übergebenen Strings zurück.

Systemfunktionen – Date Handling

- `dateadd(interval, number, date)` – addiert number im gegebenen Intervall (z.B. Jahr, Monat, ...) zum gegebenen date und gibt das Ergebnisdatum zurück.
- `datepart(interval, date)` – gibt den als Interval spezifizierten Teil eines Datums als int zurück.
- `datetime(interval, date)` – gibt den als Interval spezifizierten Teil eines Datums als int zurück
- `day(<somedate>)`
`month(<somedate>)`
`year(<somedate>)` – extrahiert den Teil eines Datums der durch den Funktionsnamen gegeben ist.

Systemfunktionen – diverse Funktionen



- `convert(data_type(length), expression, style)` – Konvertiert einen Ausdruck von einem Datentyp in einen anderen; für Datumswerte gibt es spezielle Typmodifizierer, siehe https://www.w3schools.com/sql/func_sqlserver_convert.asp
- `cast(expression AS data_type(length))` – konvertiert einen Ausdruck von einem Datentyp in einen anderen; ähnlich zu `convert` von vorher, siehe https://www.w3schools.com/sql/func_sqlserver_cast.asp
- `isNull(value, alternative-value)` – gibt im Fall, dass `value` null ist den als alternativen Wert angegeben `value` zurück

Beispiele



Erstelle Funktionen für folgende Angaben unter Verwendung der Northwind DB.

- Auflisten aller Produkte, die ein Lieferant anbietet. Die Id (der Name) des Lieferanten wird als Parameter mitgegeben.
- Auflisten aller Produkte einer Kategorie, wobei die Kategorie als Parameter verwendet wird (einmal mit Id, einmal mit Name).
- Auflisten von Produkten mit Preisen deren Wert \geq Wert1 und \leq Wert2 (als anzugebende Parameter) ist.
- Auflisten aller Bestellungen, die von einem bestimmten Mitarbeiter in einem bestimmten Jahr bearbeitet wurden (MA und Jahr sind Parameter).
- Geben Sie die Anzahl (also Summe) der Produkte aus voriger Funktion zurück (Hinweis: Umbauen in eine SVF).
- Erstellen Sie Funktionen wie in den letzten beiden Punkten, nur für die jeweils einzelnen Produkte.
- Anzahl der Orders pro Customer.

Beispiele

- Erstellen Sie eine Funktion, die quantity, product name and suppliername für die top 5 best-selling products innerhalb eines Jahres ausgibt (Jahr als Parameter).
- Durchschnittspreis pro Bestellung und die Bestellnummer pro Kunde (KundenId als Parameter).

Datenbank – Programmierung

STORED PROCEDURES

Stored Procedures

- Erhöht Softwaremodularität und reduziert Wiederholung von Code.
- Reduzieren Kommunikations- und Datenübertragungsaufwand zwischen Client und Server: Anstatt des mehrmaligen Absetzens längerer SQL Statements werden nur die Parameter übertragen.
- Sicherer als alleinstehende SQL Statements: Zugriff kann auf einzelne SPs gewährt werden, ohne Rechte auf die zugrundeliegenden Tabellen zu vergeben.
- Können extern (durch Programmiersprachen) aufgerufen werden, um Anweisungen, die effizienter auf der Datenbank zu lösen sind, auszuführen. Abhängig von konkreter Aufgabe.
- <https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-2017>
- Zu Rückgabewerten: http://www.sommarskog.se/share_data.html

Stored Procedures



"Hello World" Beispiel!

```
CREATE OR ALTER  
PROCEDURE dbo.sp_Hallo  
-- Parameter  
AS  
BEGIN  
PRINT 'Hallo Welt';  
END  
GO
```

```
-- Aufruf  
EXECUTE dbo.sp_Hallo;  
-- Ohne Klammer !!!  
GO  
  
-- oder  
EXEC dbo.sp_Hallo;  
GO
```

Stored Procedures Rückgabewerte

- Rückgabewerte werden ähnlich zu Eingabeparametern angelegt, sie müssen mit dem Schlüsselwort `OUTPUT` deklariert werden, um sie als Rückgabeparameter auszuzeichnen.
- Es können mehrere Rückgabeparameter angegeben werden (durch Komma getrennt hintereinander angeführt, `OUTPUT` Schlüsselwort muss zu jedem Parameter hinzugefügt werden).
- Wird ein Rückgabeparameter ohne das Schlüsselwort `OUTPUT` benutzt, kann der Rückgabewert nicht von der aufrufenden Logik verwendet werden.

Stored Procedures Rückgabewerte

- Eine Stored Procedure kann auch selbst einen Wert (ähnlich wie bei skalaren Funktionen) zurückliefern. Dieser Wert gibt über den Ausführungserfolg der Prozedur Auskunft.
- In der Prozdur (am Ende):
`RETURN integer_value`
ist normalerweise ein int
- Verwendung beim Aufruf:
`EXEC @return_variable = stored_procedure_name`

Stored Procedures – Bemerkungen

- @@ROWCOUNT enthält die Anzahl der Reihen, die im letzten Ergebnis-Set enthalten waren.
- SET NOCOUNT ON (| OFF) – unterdrückt (bzw. erlaubt explizit) die Angabe, wieviele Zeilen von einer Abfrage betroffen sind. Wird diese Funktion ausgeschaltet, verringern sich die Daten, die über das Netzwerk übertragen werden.

Unterschiede Prozeduren vs. Funktionen



Stored Procedure (SP)	Function (UDF - User Defined Function)
SP kann keine, einen oder mehrere Werte zurückgeben	Eine Funktion muss einen einzelnen Wert zurückliefern (welcher ein Skalar oder eine Tabelle sein kann)
In SP können Transaktionen verwendet werden	Es können keine Transaktionen in einer UDF verwendet werden
SP kann Eingabe/Ausgabe Parameter haben	Nur Eingabe Parameter
Aus einer SP können Funktionen aufgerufen werden	Eine SP kann nicht aus einer Funktion aufgerufen werden
Eine SP kann nicht in SELECT/WHERE/HAVING Statements verwendet werden.	Eine UDF kann in in SELECT/WHERE/HAVING Statements verwendet werden
Ausnahmebehandlung mit Try-Catch Blöcken kann verwendet werden	Es kann kein Try-Catch Block in einer UDF verwendet werden

Vgl. <https://stackoverflow.com/questions/1179758/function-vs-stored-procedure-in-sql-server>

Beispiele

- Überprüfen, ob es für ein bestimmtes Jahr und Monat Bestellungen gibt und als Rückgabewert die Anzahl der Bestellungen ausgeben; ist das als Parameter übergebene Jahr grösser dem aktuellen Jahr, soll eine Fehlermeldung ausgegeben werden.
- Ermitteln der Anzahl der verkauften Produkte (Summe der OrderQty) für eine bestimmte Bestellung (Input Parameter OrderId) und zurückgeben des Ergebnisses als OUTPUT; wenn Eingabeparameter OrderId ≤ 0 --> Ausgabe einer Fehlermeldung (FM) mittels print
- überprüfen ob es für eine Kategorie Produkte gibt und ausgeben des Ergebnisses (Anzahl) als OUTPUT. Wenn es keine Produkte gibt --> Ausgabe einer FM und -1 als Rückgabewert

Datenbank – Programmierung

TRIGGER

Trigger



- Ähnlich zu Stored Procedures, die bei Eintritt eines bestimmten Ereignisses ausgelöst werden

-- SQL Server Syntax

--Trigger on an INSERT, UPDATE, ---or DELETE
statement to a table --or view (DML Trigger)

- Generelle Syntax
(siehe:
<https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-2017>)

```
CREATE [ OR ALTER ] TRIGGER [ schema_name .  
]trigger_name  
ON { table | view }  
{ FOR | AFTER | INSTEAD OF }  
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }  
[ WITH APPEND ]  
[ NOT FOR REPLICATION ]  
AS sql_statement [ ; ]
```

Trigger

Beispiel zur Northwind DB

```
Create Trigger trg_employees  
on Employees  
For insert, update  
As  
Raiserror('%d rows modified', 0, 1, @@rowcount)
```

Ausführen des Triggers

Wird bei INSERT oder UPDATE in der Tabelle Employees aufgerufen:

```
Update Employees  
Set city = 'New York'  
where city='Seattle'
```

Um den Trigger wieder zu löschen

```
DROP TRIGGER trigger_name
```

Trigger – Beispiele

```
Create Trigger Test1
On Customers
For Insert, Update
As
Select customerid, city, country from Inserted
print '**** Deleted Table Info ****'
Select customerid, city, country from Deleted
```

Beispielaufruf

```
update customers
set city = ' New York'
where city = 'Paris'
```

Die Tabellen Inserted und Deleted sind temporäre Tabellen, die die betroffenen Reihen des letzten Insert oder Delete Statements beinhalten.

Trigger – Tabellen *Inserted* und *Deleted*



- Automatisch generierte und verwaltete Tabellen, welche eingefügte, gelöschte und aktualisierte Einträge enthält, die während DML Operationen (Insert, Update, Delete) auf einer Tabelle betroffen waren.
- Sie werden bei Triggern für folgende Fälle verwendet (außerhalb von Triggern sind die Tabellen nicht vorhanden):
 - Um Fehler in der Datenmanipulation zu testen und abhängig von den Fehlern entsprechende Maßnahmen zu treffen.
 - Um den Unterschied im Zustand einer Tabelle vor und nach einer Datenmanipulation zu bestimmen und Maßnahmen abhängig von diesem Unterschied vorzunehmen.

Fehlerbehandlung

- Fehlerquellen im System
 - Benutzerfehler
 - Software-Fehler
 - Hardware-Fehler
 - Ressourcenfehler

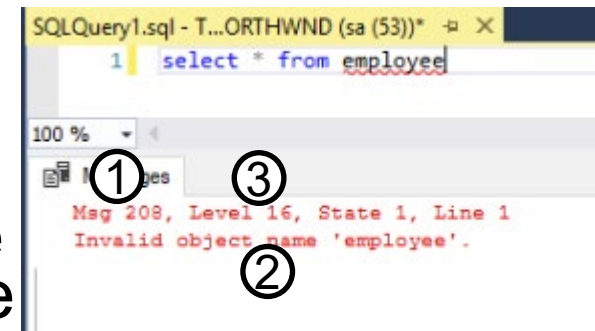
In der Tabelle `sys.messages` sind alle Systemmeldungen innerhalb des SQL-Servers gespeichert.

Abfrage aller im System bekannten/verwendeten Nachrichten:

```
select * from sys.messages
```

Fehlerbehandlung

- Eine Fehlermeldung besteht aus drei Bestandteilen:
 1. Fehlernummer
 2. Fehlerbeschreibung
 3. Schweregrad des Fehlers
- Die Nachricht mit der Message-ID 208 kann direkt in der Tabelle `sys.messages` gefunden werden (je nach Fehler wird eine andere ID zurückgegeben).



```
SQLQuery1.sql - T...ORTHWND (sa (53)) * -p X
1 select * from employee
```

100 %

① ③

Msg 208, Level 16, State 1, Line 1
Invalid object name 'employee'.
②

Fehlerbehandlung – Schweregrad

Schweregrad	Beschreibung
0-10	Status- und Informationsmeldung oder Fehler, die nicht schwerwiegend sind.
11-16	Durch Benutzer behebbare Fehler.
17	Systemressourcen sind nicht ausreichend.
19	Interner Grenzwert von SQL wurde überschritten.
21	Fataler Fehler in allen DB-Prozessen. DB des Systems sind jedoch vermutlich intakt.
22	Fataler Fehler in Tabelle oder Index. Wenn durch Neustart nicht behebbar, evtl. Festplattendefekt.
23	Gesamte DB ist von fatalem Fehler betroffen. Datenbestand muss evtl. wiederhergestellt werden.
24	Fataler Medienfehler. Deutet auf Defekt einer Festplatte hin.

Fehlerbehandlung – TRY/CATCH

- TRY/CATCH Seit SQL Server 2005
- Einem TRY Block muss ein CATCH Block folgen.
- TRY...CATCH Konstrukte können verschachtelt werden, d.h. dieses Konstrukt kann in anderen TRY...CATCH Blöcken platziert werden. Im Fehlerfall wird das zum eingebetteten TRY gehörige CATCH aufgerufen/ausgeführt.
- Syntaktische Fehler werden z.B. nicht erkannt, da der zugehörige Code von vornherein nicht verarbeitet werden kann.
- Fehler (auch syntaktische) die von Intellisense während der Entwicklungszeit nicht erkannt werden führen zu Fehlern, die mit TRY/CATCH behandelt werden können.
- Von der Anwendung her ähnlich wie in Programmiersprachen.
- zusätzliche Beispiele u. Erklärungen siehe <http://sqlhints.com/tag/try-catch/>

Fehlerbehandlung – TRY/CATCH

Generelle Syntax

```
BEGIN TRY
    { sql_statement | statement_block }
END TRY
BEGIN CATCH
    [ { sql_statement | statement_block } ]
END CATCH
[ ; ]
```

Siehe Beispiel in SQL Skript!

TRY/CATCH – Beispiel mit Transaction



```
PRINT 'BEFORE TRY'
BEGIN TRY
    BEGIN TRAN
        PRINT 'First Statement in the TRY block'
        INSERT INTO dbo.Account(AccountId, Name , Balance) VALUES(1, 'Account1', 10000)
        UPDATE dbo.Account SET Balance = Balance + CAST('TEN THOUSAND' AS MONEY) WHERE AccountId = 1
        INSERT INTO dbo.Account(AccountId, Name , Balance) VALUES(2, 'Account2', 20000)
        PRINT 'Last Statement in the TRY block'
    COMMIT TRAN
END TRY
BEGIN CATCH
    PRINT 'In CATCH Block'
    IF(@@TRANCOUNT > 0)
        ROLLBACK TRAN
END CATCH
PRINT 'After END CATCH'
SELECT * FROM dbo.Account WITH(NOLOCK)
GO
```

@@TRANCOUNT gibt an, ob Transaktionen offen sind. In diesem Fall, wenn ja, werden sie zurückgerollt.

Error (Fehler) Funktionen

Innerhalb des Catch-Blockes sind folgende Funktionen verfügbar, um genauere Informationen über die Art des Fehlers zu erhalten (auch außerhalb verfügbar, dort aber nicht belegt):

- `ERROR_NUMBER()` Fehler-Nummer.
- `ERROR_MESSAGE()` Kompletter Text der Fehlermeldung. Es werden auch die substituierten Werte für Parameter wie Längen, Objektnamen oder Zeiten ausgegeben.
- `ERROR_SEVERITY()` Schweregrad des Fehlers.
- `ERROR_STATE()` Statusnummer des Fehlers.
- `ERROR_LINE()` Zeilennummer innerhalb der Routine, die den Fehler hervorgerufen hat.
- `ERROR_PROCEDURE()` Name der gespeicherten Prozedur oder Funktion innerhalb der der Fehler aufgetreten ist.

Error Funktionen Beispiel



<pre>-- Verify that the stored procedure does not already exist. IF OBJECT_ID ('usp_GetErrorInfo', 'P') IS NOT NULL DROP PROCEDURE usp_GetErrorInfo; GO -- Create procedure to retrieve error information. CREATE PROCEDURE usp_GetErrorInfo AS SELECT ERROR_NUMBER() AS ErrorNumber ,ERROR_SEVERITY() AS ErrorSeverity ,ERROR_STATE() AS ErrorState</pre>	<pre>,ERROR_PROCEDURE() AS ErrorProcedure ,ERROR_LINE() AS ErrorLine ,ERROR_MESSAGE() AS ErrorMessage; GO BEGIN TRY -- Generate divide-by-zero error. SELECT 1/0; END TRY BEGIN CATCH -- Execute error retrieval routine. EXECUTE usp_GetErrorInfo; END CATCH;</pre>
--	---

Fehlerbehandlung RAISERROR

Erzeugt eine Ausnahme mit einer benutzerdefinierten Nachricht, die im `sys.messages` Katalog gespeichert ist bzw. dynamisch beim Aufruf der Funktion generiert wird.

Generelle Syntax

```
RAISERROR ( {msg_id | msg_str | @local_variable}
           { ,severity ,state }
           [ ,argument [ ,...n ] ] )
           [ WITH option [ ,...n ] ]
```

Parameter:

<code>msg_id</code>	Vom Benutzer vergebene Id
<code>severity</code>	gibt an, wie wichtig der Fehler ist
<code>state</code>	gibt einen Systemzustand an, der aufgrund des Fehlers auftritt
<code>@@rowcount</code>	gibt die Anzahl der Zeilen/Einträge die von einer DML Operation betroffen sind zurück

Neue Applikationen sollten `THROW` statt `RAISERROR` verwenden.

RAISERROR Beispiel

Aufruf:

```
RAISERROR (15600,-1,-1, 'mysp_CreateCustomer')
```

Rückgabe:

```
Msg 15600, Level 15, State 1, Line 1
```

```
An invalid parameter or option was specified for procedure  
'mysp_CreateCustomer'.
```

Fehlerbehandlung THROW

Seit SQLServer 2012; im Wesentlichen selbe Funktion aber einfachere Syntax als raiserror; in neuen Applikationen sollte nur mehr throw verwendet werden.

Generelle Syntax

```
THROW [ { error_number | @local_variable },  
        { message | @local_variable },  
        { state | @local_variable } ]  
  
[ ; ]
```

Parameter

error_number	Konstante oder Variable, die die Ausnahme repräsentiert.
message	String zur Beschreibung der Ausnahme (nvarchar(2048)).
state	Wert von 0 bis 255 der den Nachrichtenstatus wiedergibt.

Fehlerbehandlung THROW

- Kein expliziter Schweregrad Parameter. Der Schweregrad Parameter ist standardmäßig immer auf 16 gesetzt (Außer wenn in einem CATCH-Block "re-throwed" wird).
- Das Kommando/Statement vor THROW muss mit einem Semikolon abgeschlossen sein.
- Kann eine Error MessageId außerhalb des vordefinierten Bereichs (in sys.messages) verwenden.

THROW Beispiel

Aufruf:

```
THROW 51000, 'The record does not exist.', 1
```

Rückgabe:

```
Msg 51000, Level 16, State 1, Line 1
```

```
The record does not exist.
```

Unterschiede zwischen raiserror und throw unter <http://sqlhints.com/2013/06/30/differences-between-raiserror-and-throw-in-sql-server/>

Datenbank – Programmierung

TRANSAKTIONEN

Transaktionen

- Bieten Zugriffssicherung bei gleichzeitigem Datenzugriff durch mehrere Benutzer.
- Erhaltung von Datenkonsistenz bei mehrfachem Zugriff (auf geteilten Ressourcen) und Fehlern (Programm(ier)fehler od. Hardwareausfälle).
- Im Falle von Problemen muss es möglich sein auf einen konsistenten Vorzustand zurückzuspringen.
- Es müssen bei der Ausführung von Transaktionen die ACID-Eigenschaften immer eingehalten werden.

Transaktionen



Transaktionen müssen dem ACID Paradigma genügen

- | | | | |
|----------|-----|-------------|---|
| A | ... | ATOMICITY | Operation/Transaktion wird ganz oder gar nicht ausgeführt |
| C | ... | CONSISTENCY | darf keine inkonsistenten Zustände hervorrufen |
| I | ... | ISOLATION | muss unabhängig von anderen Transaktionen laufen |
| D | ... | DURABILITY | nachdem Transaktion erfolgreich abgeschlossen, muss Zustand nach Transaktion erhalten bleiben (auch nach Server-Neustarts oder Ausfällen) |

Zu beachtende Konzepte: Locking

Zusätzliches Material (in Englisch):

http://www.sommarskog.se/error_handling/Part2.html

Transaktionen

- SQL Server verwendet generell Transaktionen auch bekannt unter implizite Transaktionen oder `AUTO-COMMIT`.
- Explizite Transaktionen starten mit einem `BEGIN TRANSACTION` Statement und enden mit einem `COMMIT` oder `ROLLBACK` Statement.
- Erfolgreiche Durchführung einer Transaktion wird mit `COMMIT` beendet, bei Fehler während der Transaktionsausführung wird die Transaktion mit dem Kommando `ROLLBACK` in den Anfangszustand zurückversetzt (vor Beginn der Transaktion)
- Bei einem Systemabsturz werden nach einem Neustart alle gestarteten Transaktionen mit einem `ROLLBACK` zurückgesetzt.
- Generell: Um verteilten, gleichzeitigen Zugriff auf Ressourcen zu organisieren, z.B.: was passiert, wenn ein Produkt gelöscht wird, wenn an anderer Stelle eine Bestellung für dieses Produkt angelegt wird?

Transaktionen – Isolationsebenen

- Transaction Isolation Level
- Steuerung des Grades der Parallelität von Transaktionen
- Höherer Level bedeutet
 - + Daten sind aktuell
 - + weniger Konsistenzprobleme
 - – geringerer Durchsatz bei großer Zahl gleichzeitiger Zugriffe weil mehr Sperren (Locking) notwendig sind.

Isolationsebenen:

0 READ UNCOMMITTED	es werden keine Sperren gesetzt, dirty read möglich
1 READ COMMITTED	nur modifizierte Datensätze werden gesperrt, inkonsistentes Lesen und Phantom-Probleme können auftreten
2 REPEATABLE READ	alle Datensätze werden gesperrt, Phantom-Probleme können noch auftreten
3 SERIALIZABLE	alle gelesenen Daten sind bis zum Ende der Transaktion gültig, Phantom-Problem wird verhindert, volle Serialisierbarkeit gegeben

Transaktionen

- Beispiele

- Mit COMMIT

```
BEGIN TRANSACTION;  
DELETE FROM HumanResources.JobCandidate  
WHERE JobCandidateID = 13;  
COMMIT;
```

- Mit ROLLBACK

```
CREATE TABLE ValueTable (id int);  
BEGIN TRANSACTION;  
INSERT INTO ValueTable VALUES (1);  
INSERT INTO ValueTable VALUES (2);  
ROLLBACK;
```

Transaktionen

Spezielle Einstellungen bei der Erstellung von Transaktionen

- `SET NOCOUNT (ON/OFF)`
 - `OFF` – default Einstellung; gibt bei DML Kommandos die Anzahl der betroffenen Zeilen zurück; kann bei großen Datenmengen zu Performanzeinbußen führen, da für die Ausgabe wieder eine Abfrage (automatisch) erzeugt wird.
 - `ON` – schaltet die Ausgabe der Anzahl von betroffenen Zeilen bei DML Kommandos aus.
- `SET XACT_ABORT (ON/OFF)`
 - `OFF` – default Einstellung; Führt innerhalb einer Transaktion alle positiv abschließbaren Statements aus und rollt nur fehlerhafte zurück.
 - `ON` – wenn innerhalb des Transaktionsblock ein Statement fehlschlägt, werden auch alle anderen (auch fehlerfreie) Statements zurückgerollt.

Datenbank – Programmierung

CURSOR

Cursor

- Bieten eine Methode, um über die Einträge in einem Ergebnisdatensatz zu iterieren.
- Jeweils Reihe für Reihe.
- Vergleichbar mit Iteratoren in Programmiersprachen, z.B. for oder foreach.
- Erlaubt die prozedurale Abarbeitung von Datensätzen.
- Es kann dadurch zusätzliche Logik, abhängig von bestimmten Bedingungen die erfüllt werden oder nicht, hinzugefügt werden.
- Ergebnisse können dadurch feingranular extrahiert werden.

Cursor

Die Verwendung von Cursor findet in mehreren Schritten statt:

1. Anlegen von Variablen, die die Rückgabewerte des Cursor speichern.
2. Anlegen des Cursor Objekts.
3. Zuweisen der Abfrage an den Cursor.
4. Öffnen des Cursor.
5. Abfragen der ersten Reihe.
6. Iterieren bis keine Ergebnissätze mehr erhalten werden.
7. Schliessen des Cursor.

- Beispiel für die AdventureWorks DB

```
USE ADVENTUREWORKS2008

GO
DECLARE @ProductID as INT;
DECLARE @ProductName as NVARCHAR(50);

DECLARE @ProductCursor as CURSOR;

SET @ProductCursor = CURSOR FOR
SELECT ProductID, Name
FROM SalesLT.Product;

OPEN @ProductCursor;
FETCH NEXT FROM @ProductCursor INTO @ProductID, @ProductName;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT cast(@ProductID as VARCHAR (50)) + ' ' + @ProductName;
    FETCH NEXT FROM @ProductCursor INTO @ProductID, @ProductName;
END

CLOSE @ProductCursor;
DEALLOCATE @ProductCursor;
```

Datenbank – Programmierung

INDEX

Index

- Ziel: Extraktion von Ergebnismengen von Abfragen effizienter gestalten.
- Zugriffe auf oft benötigte Spalten sollen rascher durchgeführt werden können als für den Rest der Datenbank.
- Vergleichbar mit einem Index am Ende eines Fachbuches.
- SELECT und WHERE Statements werden durch Verwendung eines Index beschleunigt, UPDATE und INSERT Statements verzögert (Wegen Neuerstellung des Index).

Index

- Für einen Index auf einer einzelnen Spalte wird folgendes Kommando verwendet:

```
CREATE INDEX index_name ON table_name (column_name)
```

geht nicht auf gesamte Tabelle ohne Angabe eines Attributs

- Wenn ein Index als UNIQUE gekennzeichnet wird, kann ein Wert immer nur einmal vorkommen:

```
CREATE UNIQUE INDEX index_name on table_name (column_name)
```

- Um einen Index umzubenennen wird folgendes Kommando verwendet:

```
sp_rename 'table_name.index_name', 'new_index_name', 'INDEX'
```

Index

- **Kombinierter Index, allgemein:**

```
CREATE INDEX index_name on table_name (col1, col2)
```

Beispiel:

```
CREATE INDEX IX_Pers_LastN_FirstN  
ON Employees (LastName, FirstName)
```

- Sortierkriterien werden in der Reihenfolge ihrer Definition berücksichtigt
- Für Primärschlüssel und Unique Einschränkungen wird immer ein impliziter Index angelegt.
- Löschen eines Index:

```
DROP INDEX index_name
```

Index

- **CLUSTERED**

- Daten im Index werden nach ihren Werten sortiert.
- Pro Tabelle kann es nur einen Clustered Index geben (weil Daten nur nach einer Reihenfolge sortiert werden können).
- Sortiert werden Daten auch nur dann abgespeichert, wenn ein Clustered Index festgelegt wurde.

- **NONCLUSTERED**

- Dieser Index wird in einer Struktur, die unabhängig von den Datenwerten ist, die indiziert werden, gespeichert.

- **Index in Views**

```
CREATE INDEX index_name ON view_name
```

- Siehe <https://db-berater.blogspot.com/2013/01/verwendung-von-indexed-views-zur.html>

Index

Fälle, unter denen von der Verwendung eines Index abzuraten ist:

- Bei relativ kleinen Tabellen, also solche, die wenige Tupel enthalten.
- Tabellen, bei denen oftmalige Batch-Updates oder Einfüge-Operationen durchgeführt werden.
- Bei Spalten (Attributen) die eine große Zahl an NULL-Werten beinhalten.
- Spalten, die oft geändert werden, sollen nicht indiziert werden. Es wird sonst immer wieder die Neu-Erzeugung des Index angestoßen, was bei einer großen Menge an Einträgen in-performant werden kann.

Datenbank Administration

Backups erstellen und rückspielen

- Backup (Sicherung) erstellen – 3 Arten von Backups
 - Vollständig: Ein komplettes Abbild der Datenbank wird gespeichert.
 - Differentiell: Die Unterschiede seit dem letzten vollständigen bzw. differentiellen Backup werden gespeichert.
 - Transaktionsprotokoll: Backup der (offenen sowie beendeten) Transaktionen wird erstellt.
- Backup rückspielen
 - Als *.sql Datei
 - Als *.bak Datei
 - Als *.mdf Datei
- Behandlung von CSV Dateien

Export einer Datenbank

- Mehrere Möglichkeiten des Exports:
 - als SQL Skript
 - als CSV Datei
 - in ein Format das direkt von anderen Windows Tools (z.B. Excel od. Access) weiterverarbeitet werden kann.

Sicherungen erstellen

- Als *.sql Datei: Über den Import/Export-Dialog
- Als *.mdf Datei:
- Als *.bak Datei:
 - Im Objekt-Explorer Rechtsklick auf die zu sichernde Datenbank
 - Im Kontext Menu Unterpunkt Tasks -> Sichern... auswählen
 - Im erscheinenden Dialog Parameter setzen, vor allem Speicherort an dem Backup abgelegt werden soll.
 - Dann OK klicken.

Datenbank-Sicherung wiedereinspielen



- Als *.sql Datei:
Doppelklick auf *.sql Datei öffnet diese automatisch im MSSMS. Ausführen des SQL Skripts erstellt dann die Datenbank.
- Als *.mdf Datei:
 - Im Object Explorer Rechtsklick auf Databases und im Kontextmenü „Attach...“ auswählen.
 - Über „Hinzufügen...“ entsprechend *.mdf Datei auswählen und OK klicken. Nochmal OK fügt die Datenbank hinzu.
 - Bei “Access Denied”/“Zugriff verweigert” stimmen die Zugriffsrechte bei der Datei nicht. – REVISE
- Als *.bak Datei:
 - Im MSSQL-Studio: Rechtsklick auf Datenbanken links im Objekt-Explorer
 - Datenbank wiederherstellen klicken...
 - Medium Radiobutton selektieren, ... klicken und .bak Datei auswählen
 - Dann 2x OK klicken.

Benutzerverwaltung

- 2 grundlegende Möglichkeiten um Authentifizierung durchzuführen:
 - Windows Konto: Es wird ein vorhandenes Konto vom lokalen Windows-Host bzw. einer Windows-Domäne wieder-/weiterverwendet.
 - SQL Benutzerkonto: Ein Benutzerkonto wird explizit innerhalb einer DB-Tabelle (in einer der Systemtabellen) erstellt.
- Wichtigste SQL Kommandos
 - GRANT
 - REVOKE
 - DENY

Verschiedene Tools

- SQL auf der Kommandozeile über sqlcmd
- Einstellen des SQL Server für Multiuser über Netzwerk:
<https://imron.com/support/knowledgebase/configuring-a-sql-server-for-remote-connections/>

Microsoft SQL Server

- Relationales DBMS von Microsoft
- Versionen: 2000, 2005, 2008, 2012, 2014, 2016, 2017
- Unterschiede zwischen Versionen größtenteils im Bereich von Optimierungen.
- Grundlegende Bedienung größtenteils kaum Änderungen bzw. SQL Basics unterscheiden sich seit 2000 bis heute nur minimal.
- Verschiedene Tools für Datenspeicherung, Datenmanipulation usw.
- Hier: Microsoft SQL Server Manager (MSSMS)

Konzeptuelle Modellierung - Beispiele



- Modellieren Sie ein Zugauskunftssystem, in dem die wichtigsten Züge (z.B. die Intercity- und Eurocity-Züge) repräsentiert werden. Aus dem System sollen die Start- und Zielbahnhöfe und die durch den Zug verbundenen Bahnhöfe einschließlich Ankunfts- und Abfahrtszeiten ersichtlich sein. Geben Sie die Funktionalitäten der Beziehungstypen an.
- Modellieren Sie die Grundlagen eines Krankenhausverwaltungssystems. Insbesondere sollten die Patienten, deren Stationen, deren Zimmer, die behandelnden Ärzte und die betreuenden Pfleger modelliert werden. Verwenden Sie wiederum die Generalisierung zur Strukturierung Ihrer Entitytypen.
- Elmar Fuchs – SQL Grundlagen und DB-Design, Beispiel 3.5.6

Konzeptuelle Modellierung - Beispiele



In Büroräumen (charakterisiert durch eine Zimmernummer) sitzen zu einem Zeitpunkt Mitarbeiter (Personalnummer, Name, Titel, Status) an einem bestimmten Platz. In den Zimmern sind Telefone (besitzen eine eindeutige Telefonnummer) aufgestellt, die als Hausapparat oder Amtsapparat geschaltet sind.

Konzeptuelle Modellierung - Beispiele



In einer Abteilung (besitzt Abteilungsnummer, Name und Ort) arbeiten Mitarbeiter (charakterisiert durch Personalnummer, Name, Geburtsdatum, Privatadresse, Gehalt, Tätigkeitsbezeichnung), die an Projekten (eindeutige Projektnummer, Projektbeginn und -ende) arbeiten. Jeder Mitarbeiter arbeitet mit einer bestimmten Wochenstundenzahl an einem bestimmten Projekt und kann an mehreren Projekten gleichzeitig arbeiten. Jedes Projekt wird von einem anderen Mitarbeiter geleitet.

Konzeptuelle Modellierung – Beispiele



- Zeichnen Sie ein ER-Diagramm der Entitäten Postleitzahl und Ort mit der Beziehung "identifiziert,,."
- Schüler (Vorname, Name) erhalten Zeugnisse. Die Zeugnisse enthalten eine Bemerkung über Mitarbeit und Verhalten und die Fachnoten.
- Zu einer gespeicherten Sammlung von Digitalfotos, deren Datum und Auflösung bekannt ist, soll ein Stichwortverzeichnis angelegt werden.
- CDs (Titel, ISBN-Nummer) sind von bestimmten Interpreten (Name) und enthalten Songs (Titel).
- Entwickler programmieren Komponenten, die selbst wiederum aus Komponenten bestehen können. Um Probleme zu vermeiden, darf eine Komponente nicht von mehreren Entwicklern programmiert werden.

Beispiele unter: <http://www.roro-seiten.de/info/db/04ERModell2/ERModell2.html#0>

SQL – Beispiele

Für die Northwind Datenbank:

- Selektiere alle Einträge der Tabellen Categories, Customers, Employees, Orders, Products, Shippers, Suppliers in einem jeweils eigenem Statement.
- Selektiere CategoryName und Description aus der Tabelle Categories (inkl. Sortierung nach CategoryName).
- Selektiere ContactName, CompanyName, ContactTitle und Phone aus der Tabelle Customers (inkl. Sortierung nach Phone).
- Selektiere EmployeeID, Title, FirstName, LastName und Region aus der Tabelle Employees.
- Selektiere RegionID und RegionDescription aus der Tabelle Region.
- Selektiere CompanyName, Fax, Phone und HomePage aus der Tabelle Suppliers.

SQL – Beispiele

Fuer die Northwind Datenbank:

- Erstelle eine Abfrage für Vor-, Zuname und Einstellungsdatum von Mitarbeitern, sortiert nach dem Einstellungsdatum (neueste zuerst).
- Erstelle eine Abfrage nach OrderID, OrderDate, ShippedDate, CustomerID und Freight sortiert nach Freight, vom teuersten zum billigsten.
- Selektiere CompanyName, Fax, Phone, HomePage und Country aus der Tabelle Suppliers sortiert nach Country in absteigender Reihenfolge und dann nach CompanyName in aufsteigender Reihenfolge.
- Erstelle eine Liste von Mitarbeitern mit Titel, Vor- und Zunamen. Sortiere nach Job-Beschreibung in aufsteigender Reihenfolge und dann nach Zuname in absteigender Reihenfolge.
- Zeige alle Firmen- und Kontaktnamen von Kunden in Buenos Aires.
- Zeige Produktname, Preis und Anzahl aller Produkte die nicht auf Lager sind.

SQL – Beispiele

Für die Northwind Datenbank:

- Zeige Bestelldatum, Versanddatum, KundenId und Fracht aller Bestellungen vom 19. Mai 1997.
- Zeige Vor- und Zuname und Land aller Mitarbeiter die nicht in US sind.
- Zeige Vor- und Zunamen aller Mitarbeiter die an niemanden berichten (ReportsTo).
- Zeige Vor- und Zunamen und Geburtsdatum aller Mitarbeiter, die in den 1950ern geboren sind.
- Zeige VersandPLZ, BestellId und Bestelldatum für alle Bestellungen deren PLZ mit "02389" beginnt.
- Zeige Kontaktnamen, Titel und Firmennamen für alle Kunden deren Kontakttitel nicht mit "Sales" beginnt.

SQL – Beispiele

Für die Northwind Datenbank:

- Zeige Vor-, Zunamen und Stadt von Mitarbeitern im Region 'WA' die nicht in der Stadt Seattle zuhause sind.
- Zeige Firmennamen, Kontakttitel, Stadt und Land aller Kunden in Mexiko oder in irgendeiner Stadt in Spanien ausser Madrid.
- Durchschnittspreis aller Produkte sowie von "continued" und "dis-continued" Produkten.
- Ausgabe von Produkten, die über dem Durchschnittspreis aller Produkte liegen (subquery verwenden).
- Rechnen Sie zu allen Bestellungen den Diskont hinzu und geben Sie den Preis pro Bestellung mit und ohne Diskont aus.
- Berechnen des Alters von Bestellungen
- Berechnen des Alters von Mitarbeitern und wie lange sie schon im Unternehmen sind.

SQL - Beispiele

Für die AdventureWorks Datenbank:

- Anzahl der Datensätze pro Tabelle bestimmen
- Produkte deren Listenpreis zwischen 500 und 1000 liegt
- Produkte, die (k)ein SellEndDate haben
- Ausgabe von Kunden deren Nachname mit einem „G“ beginnt, sortiert nach ihrem Vornamen
- Ausgabe von Kunden inkl. deren Adressen (Hinweis: Verknüpfung nötig)
- Gesamtbetrag pro Bestellung berechnen
- Ausgabe der Produktkategorie und der Elternkategorien (jeweils Name)

SQL Aggregation/Gruppierung – Beispiele



- Zeige ProduktId und Gesamteinheiten aus der “Order Details” Tabelle bei denen weniger als 200 Gesamteinheiten gelistet sind.
- Zeige ProduktId und Durchschnittspreis aus der Produkte Tabelle deren Durchschnittspreis größer als 70 ist.
- Zeige die Anzahl der Bestellungen aus der Bestellung Tabelle bei denen die Anzahl größer als 15 ist.

Geschachtelte Abfragen – Beispiele

- Erstelle einen Report, der den Produktnamen und AnbieterID (supplierId) für alle Produkte, die von Exotic Liquids, Grandma Kelly's Homestead und Tokyo Traders angeboten werden.
- Erstelle einen Report, der alle Produkte aus der Kategorie Seafood beinhaltet.
- Erstelle einen Report, der alle Firmennamen auflistet, die Produkte aus Kategorie 8 anbieten.
- Erstelle einen Report, der alle Firmennamen auflistet, die Produkte aus der Kategorie Seafood anbieten.

Joins – Beispiele

Extrahiere folgende Informationen aus der Northwind DB:

- Umsatz pro Bestellung (Sales_per_Order)
 - Umsatz pro Kunde (Sales_per_Customer)
 - Umsatz pro Mitarbeiter (Sales_per_Employee)
 - Umsatz pro Region und Territory (Sales_per_Region/Territory)
 - Bestellungen pro Region und Territory (Orders_per_Region/Territory)
- bei letzten zwei Abfragen Verwenden der Region Tabelle!

Joins – Beispiele

Auf der Northwind Datenbank:

- Erstelle einen Bericht, der die Order Id und den zugehörigen Mitarbeiternamen für Bestellungen welche nach dem Bedarfsdatum versendet wurden.
- Erstelle einen Bericht, der die Gesamtanzahl der Produkte (aus der „Order Details“ Tabelle) sortiert. Zeige nur Tupel für Produkte von denen weniger als 200 bestellt wurden.
- Erstelle einen Bericht, der die Gesamtanzahl an Bestellungen pro Kunde seit 31. Dezember 1996 zeigt. Der Bericht soll nur Reihen beinhalten für die NumOrders größer als 15 ist.
- Erstelle einen Bericht, der den Firmennamen, OrderId und Gesamtpreis aller Produkte von denen Northwind in Summe mehr als 10.000\$ verkauft hat. In dieser Abfrage ist keine GROUP BY Klausel nötig.

Glossar – Cheat Sheet

- SQL ... Structured Query Language
- ERM ... Entity Relationship Model
- DB ... Datenbank
- DBMS ... Datenbank-Managementsystem
- UML ... Unified Modelling Language
- BLOB... Binary Large Object – speichert Daten in einem Binärformat, evtl. Bilder
- DDL ... Data Definition Language
- DML ... Data Manipulation Language
- DQL ... Data Query Language

Glossar – Cheat Sheet

- Normalformen
 - 1NF
 - 2NF
 - 3NF
- Funktionale Abhängigkeit
- Referentielle Integrität
- Anomalien
 - Im Einbenutzerbetrieb
 - Im Mehrbenutzerbetrieb

Glossar – Cheat Sheet



- Integritätsbedingungen
 - Wertebereiche
 - Schlüsseleigenschaften
 - Fremdschlüssel
 - usw.

Glossar

Quellenhinweise:

- Klemper/Eickler, Datenbanksysteme, 4. Auflage
- enthält Teile aus Foliensätzen von M.Palkovicz/M. Nichterl
- Elmar Fuchs, SQL – Grundlagen und Datenbankdesign, 4. Ausgabe, 2015
- de.wikipedia.org

Fragen?