

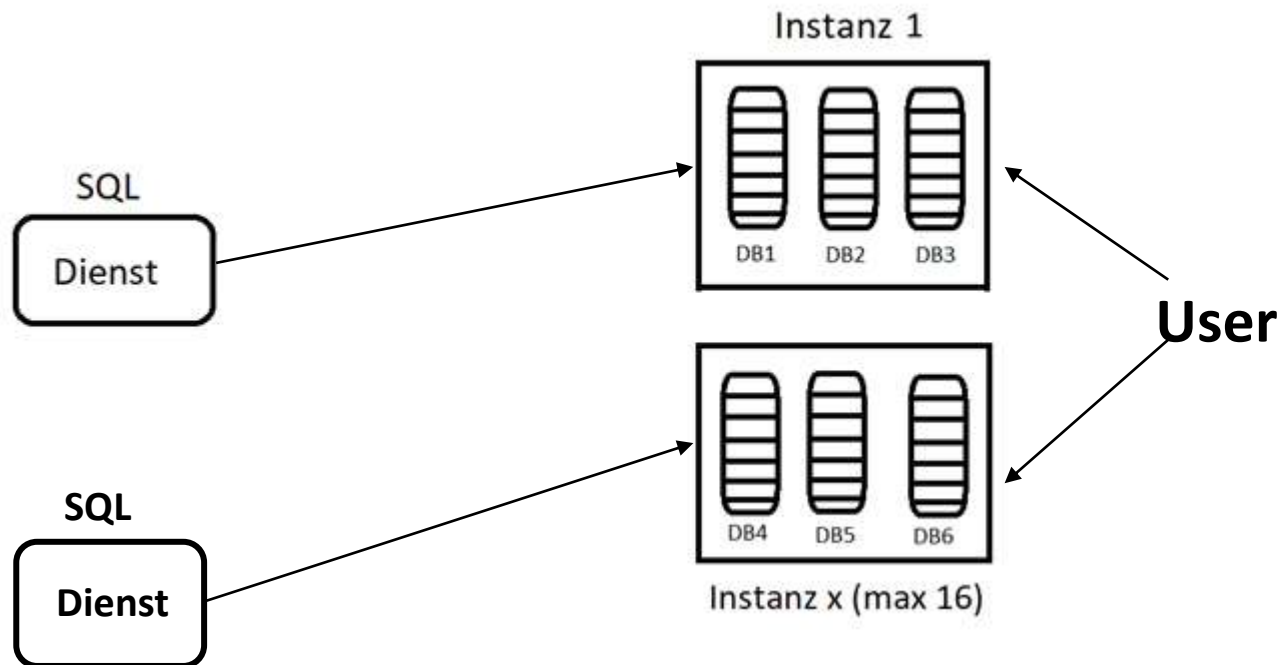


Modul 1 – SQL Server und TSQL

Was ist ein SQL Server?

- Was ist SQL? -> (Structured Query Language)
 - Datenbanksprache zur Erstellung von Datenbankstrukturen
 - Bearbeiten und Abfragen von Daten der darauf basierenden Datenbanken
- Server die auf der SQL Sprache basieren
- Bietet alles notwendige für relationale Datenbanken
- Relationen stellen die Datenbanken dar
- Ein Datensatz (eine Zeile einer Tabelle) ist eine Relation
- Beziehungen zwischen Tabellen sind durch Relationen dargestellt
 - Primärschlüssel → Fremdschlüssel

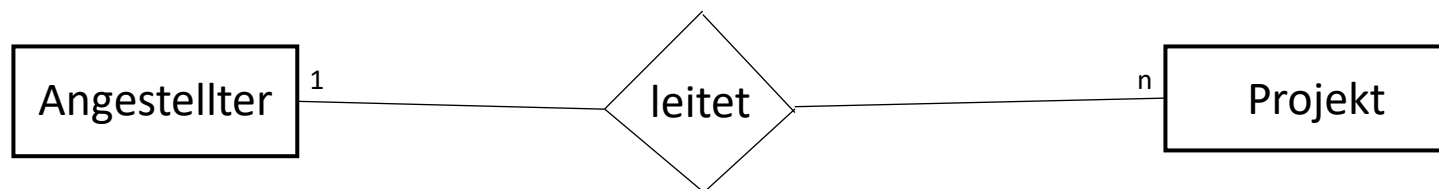
Wie ist ein SQL Server und wie ist er aufgebaut?



Relationale Datenbanken

- Bestandteile:
 - Tabellen mit Tabellenspalten und Datensätzen (Tabellenzeilen)
 - Beziehungen
 - Views
 - Prozeduren
 - Etc.
- Logisches Modell : ERM (Entity-Relationship-Modell)

Beispiel:



SSMS – SQL Server Management Studio

- SSMS (SQL Server Management Studio)
 - Verbindung zur Instanz
 - Zugriff auf Datenbanken (Lesen / Schreiben)
 - Einstellung der Datenbanken
 - Backup / Wiederherstellung
- Verbindungen zu allen vorhandenen Instanzen möglich
 - Programm durchsucht System und Netzwerk nach möglichen Instanzen
- Verschiedene Authentifizierungsmöglichkeiten
 - Windows - Authentifizierung
 - SQL Server - Authentifizierung

SSMS – SQL Server Management Studio #2

- Zwei wichtige Fenster :
 - Objektexplorer
 - Hauptfenster
- Objektexplorer zeigt Instanzeninhalt (Ordner, Datenbanken, Tabellen, etc.)
- Hauptfenster ist Arbeitsfenster
 - Erstellen von Skripten
 - Verwalten der Datenbank/Instanz
- Eigenschaftsfenster ist nützlich → genauere Informationen über Daten/Tabellenspalten
- Unter dem Reiter „Ansicht“ sind alle Fenster aufgelistet und einblendbar



Modul 2 – Einführung in TSQL Abfragen

Was ist TSQL?

- Structured Query Language (SQL)
 - Entwickelt von IBM in den 70ern
 - Verschiedene Implementierungen im Laufe der Jahre durchschritten
 - Häufig genutzt in der Industrie (MySQL , TSQL, etc.)
- Transact –SQL allgemein bezeichnet als TSQL
 - Standardabfrage unseres SQL Servers 2016
 - Basiert auf dem SQL Grundprinzip
 - Einige andere Funktionen / Parameter
- TSQL arbeitet viel mit Logik
 - Beschreibt Fälle durch so genannte Aussagen
 - Logische Operatoren werden verwendet um Aussagen zu bilden

Teilsprachen der Sprache TSQL

Teilsprachen von TSQL	DML*	DDL	DCL
Bedeutung:	Data Manipulation Language	Data Definition Language	Data Control Language
Aufgabe:	Erstellen bzw. Schreiben von Abfragen und das Manipulieren der Daten	Erstellen bzw. Definieren von Datenbankobjekten und deren Eigenschaften	Erstellen bzw. Verwalten der Sicherheitsrichtlinien unseres SQL Servers
Beispiele:	SELECT, INSERT, UPDATE, DELETE	CREATE, ALTER, DROP	GRANT, REVOKE, DENY

* Hier liegt unser Schwerpunkt für den Kurs

TSQL Elemente

- Prädikate und Operatoren
- Funktionen
- Variablen
- Expressions (Ausdrücke)
- Abfragen Separatoren
- Kommentare

TSQL Elemente und ihre Rollen

Element	Form	Aufgabe
SELECT	SELECT (Tabellenspalten)	Definiert welche Spalten zurückgegeben werden soll
FROM	FROM (Tabelle/View)	Definiert die Quellentabelle bzw. Ursprungstabelle
WHERE	WHERE (Bedingung)	Filtert die Ausgabe nach einem oder mehreren Kriterien
GROUP BY	GROUP BY (Tabellenspalten)	Gruppiert gleiche Elemente einer Ausgabe in einer Zeile
HAVING	HAVING (Bedingung mit Prädikat)	Filtert die Ausgabe nach einem oder mehreren Kriterien mit Prädikat
ORDER BY	ORDER BY (Tabellenspalten)	Sortiert die Ausgabe aufsteigend bzw. absteigend

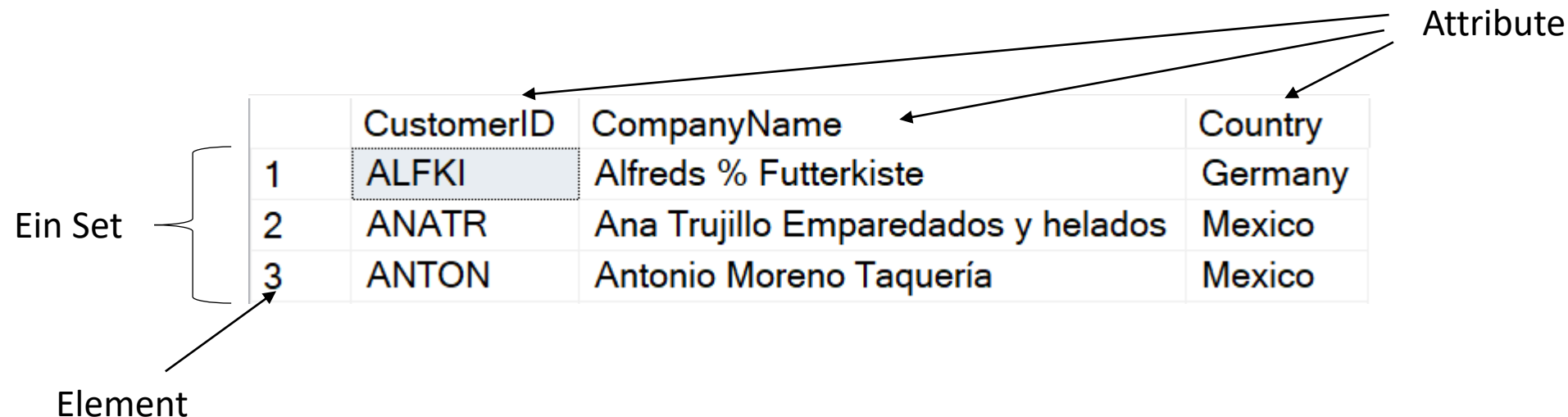
Logische Prozessreihenfolge

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

Die Reihenfolge, in der eine Abfrage geschrieben ist, hat nichts mit der Ausführungsreihenfolge zu tun.

SET Theorie in SQL Servern

- 3 Begriffe: Set, Elemente, Attribute
- Set → Alle Datensätze der Tabelle
- Elemente → ein einzelner Datensatz
 - Elemente müssen einzigartig sein (z.B. eine ID für jeden Datensatz)
- Attribute → einzelne Wert eines Datensatzes (Tabellenspalten)



The diagram shows a table with four columns: an index, CustomerID, CompanyName, and Country. The first three rows are highlighted with a bracket labeled 'Ein Set'. The first row is further highlighted with a dotted border, and an arrow points to it from the label 'Element'. Three arrows point from the label 'Attribute' to the columns CustomerID, CompanyName, and Country.

	CustomerID	CompanyName	Country
1	ALFKI	Alfreds % Futterkiste	Germany
2	ANATR	Ana Trujillo Emparedados y helados	Mexico
3	ANTON	Antonio Moreno Taquería	Mexico

Arbeit mit TSQL Scripts

- Reihe von SQL Abfragen → Skript
- Sollte mit einmaligem Ausführen funktionieren, keine Fehlermeldungen
- WICHTIG: Datenbanken beachten
 - System erkennt nicht automatisch die zugehörigen Datenbanken zu den Abfragen
- Lösung: **USE** Operator
- Beispiel:

USE Northwind

GO

- GO beendet einen Befehl



Modul 3 – Select Abfragen

Elemente einer Select Abfragen

Element	Aussage
SELECT	SELECT (Tabellenspalten)
FROM	FROM (Tabelle oder Sicht)
WHERE	WHERE (Bedingung)
GROUP BY	GROUP BY (Tabellenspalten)
ORDER BY	ORDER BY (Tabellenspalten)

- SELECT bestimmt die auszugebenen Inhalte
- FROM legt die Tabellen fest, aus denen Sie stammen
- WHERE legt die Filterung/Bedingung für die Abfrage fest
- GROUP BY gruppiert gleiche Datensätze der angegebenen Spalten
- ORDER BY sortiert aufsteigend bzw. absteigend nach angegebenen Spalten

Daten aus einer Tabelle abrufen

- SELECT mit einer Liste von Spalten (oder auch einen *) aus gewählter Tabelle
- FROM wird benutzt, um die Tabelle zu spezifizieren
- Datentyp etc. werden aus Ursprungstabelle übernommen
- Benennt die Spalten bei Bedarf durch den Parameter AS

```
SELECT companyname, country  
FROM Sales.Customers;
```

Anzeige von Tabellenspalten

- Anzeigen aller Tabellenspalten:

```
SELECT *  
FROM Sales.Customers;
```

- Anzeigen bestimmter Tabellenspalten:

```
SELECT companyname, country  
FROM Sales.Customers;
```

Berechnungen in der SELECT Klausel

- Berechnungen sind skalar, geben Ergebnis in EINER Zeile zurück

Operator	Beschreibung
+	Addieren oder Verkettung
-	Subtraktion
*	Multiplikation
/	Dividieren
%	Modulo

```
SELECT unitprice, qty, (qty * unitprice)
FROM Sales.OrderDetails;
```

Duplikate in SQL Abfragen

- SQL Abfragen sind nicht immer relational
 - Einzelnen Zeilen haben keine Garantie, dass sie einzigartig sind
 - Sogar einzigartige Daten aus der Ursprungstabelle können Duplikate ausgeben

```
SELECT country  
FROM Sales.Customers;
```

```
country  
-----  
Argentina  
Argentina  
Belgium  
Austria  
Austria
```

Der DISTINCT Parameter

- Spezifiziert die Abfrage, so dass nur noch einzigartige Ergebnisse ausgegeben werden
 - Entfernt Duplikate erst in der Ausgabe und nicht in der Ursprungstabelle
- Garantiert die Eindeutigkeit des gesamten Ausgabesets

```
SELECT DISTINCT companyname, country  
FROM Sales.Customers;
```

companyname -----	country -----
Customer AHPOP	UK
Customer AHXHT	Mexico
Customer AZJED	Germany
Customer BSVAR	France

Aliase um auf Spalten/Tabellen zu verweisen

- Spalten Alias :

```
SELECT orderid, unitprice, qty AS quantity  
FROM Sales.OrderDetails;
```

- Tabellen Alias:

```
SELECT custid, orderdate  
FROM SalesOrders AS SO;
```

- Aliase im Select werden nur in der ORDER BY Klausel erkannt
 - FROM, WHERE werden vor SELECT ausgeführt

Einfach CASE Aussagen in Select

- TSQL CASE Aussagen geben nur EINEN Wert zurück also auch nur eine Spalte
- CASE kann benutzt werden in:
 - Der SELECT Klausel
 - Der WHERE oder HAVING Klausel
 - Der ORDER BY Klausel
- CASE gibt das Ergebnis der hinterlegten Aussage zurück

Beispiel:

```
SELECT CASE  
    WHEN MIN(Value) <= 0 THEN 0  
    WHEN MAX(1/Value) >= 100 THEN 1  
END  
FROM Data;
```



Modul 03 – Arbeiten mit Datentypen

SQL Server Datentypen

- SQL Server ordnet Spalten, Ausdrücke, Variablen und Parameter Datentypen zu
- Datentypen definieren die Art von Werten, die in unseren Spalten/Variablen gelagert werden können
 - Integer, character, dates, decimals, strings, etc.
- SQL Server stellt integrierte Datentypen bereit

SQL Server Datentypen Kategorien	
Exakte Numerische	Unicode Zeichen und Zeichenketten
Ungefähre Numerische	Binäre Zeichenketten
Datum und Zeit	Etc.
Zeichen und Zeichenkette	

n definieren

Numerische Datentypen

- Genaue numerische Datentypen

Datentyp	Range	Speichergröße in Bytes
tinyint	0 – 255	1
Smallint	-32,768 – 32,768	2
int	$-2^{31} - 2^{31}-1$	4
bigint	$-2^{63} - 2^{63}-1$	8
bit	1 , 0 oder NULL	1
decimal/numeric	$-10^{38} + 1 - 10^{38}-1$ (bei maximal genutzter Präzision)	5 - 17
money	-922,337,203,685,477.5808 – 922,337,203,685,477.5807	8
smallmoney	-214,748.3648 – 214,748.3647	4

String Datentypen

- String Datentypen

Datentyp	Range	Speichergröße in Bytes
Binary	1 – 8000 bytes	n bytes
varbinary	1 – 8000 bytes	n bytes + 2
varbinary (max)	1 – 2,1 Billionen bytes	n bytes + 2

- Der **Image** Datentyp ist ebenfalls ein String Datentyp aber wird in zukünftigen Versionen von SQL Servern entfernt, stattdessen sollte **varbinary(max)** benutzt werden.

Andere Datentypen

Data Type	Range	Storage (bytes)	Remarks
xml	0-2 GB	0-2 GB	Stores XML in native hierarchical structure
uniqueidentifier	Auto-generated	16	Globally unique identifier (GUID)
hierarchyid	n/a	Depends on content	Represents position in a hierarchy
rowversion	Auto-generated	8	Previously called timestamp
geometry	0-2 GB	0-2 GB	Shape definitions in Euclidian geometry
geography	0-2 GB	0-2 GB	Shape definitions in round-earth geometry
sql_variant	0-8000 bytes	Depends on content	Can store data of various other data types in the same column
cursor	n/a	n/a	Not a storage datatype—used for cursor operations
table	n/a	n/a	Not a storage data type—used for query operations

Datentypen Priorität

- Priorität des Datentyps legt fest, welcher Datentyp gewählt wird, wenn Ausdrücke verschiedener Typen kombiniert werden
- Standardmäßig wird der Datentyp mit niedrigerer Priorität in den mit höherer Priorität konvertiert
- Es ist wichtig, implizite Umwandlung zu verstehen:
 - CHAR -> VARCHAR -> NVARCHAR -> TINYINT -> INT -> DECIMAL -> TIME -> DATE -> DATETIME2 -> XML
- Nicht alle Datentypen lassen sich in einen anderen konvertieren
 - Nicht implizit oder explizit

Wann werden Datentypen konvertiert?

- Szenarien:
 - Wenn Daten verschoben, mit anderen Daten verglichen oder kombiniert werden
 - Während der Zuweisung von Variablen
- Implizite Konvertierung:
`WHERE <column of smallint type> = <value of int type>` in Datentyp
 - Transparent für den Nutzer:
`CAST(unitprice AS INT)`
- Explizite Konvertierung:
 - Nutzen der CAST oder CONVERT Funktion

Strings Konkatenieren

- Der Operator + (plus) und die Funktion CONCAT können beide zum Verketteten von Strings in SQL 2016 verwendet werden
- Nutzen der CONCAT Funktion

```
SELECT custid, city, region, country,  
       CONCAT(city, ', ' + region, ', ' + country) AS location  
FROM Sales.Customers;
```

LL in eine leere

- ```
SELECT empid, lastname, firstname,
 firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

- Keine Konvertierung von NULL oder den Datentypen

# Datums und Zeit Datentypen

- In SQL Server 2008 wurden die Datentypen date, time, datetime2 und datetimeoffset eingeführt
- SQL Server 2012 hat weitere Funktionen zum Arbeiten mit Datums- und Uhrzeitdatentypen hinzugefügt

| Data Type      | Storage (bytes) | Date Range (Gregorian Calendar)      | Accuracy                                             | Recommended Entry Format                      |
|----------------|-----------------|--------------------------------------|------------------------------------------------------|-----------------------------------------------|
| datetime       | 8               | January 1, 1753 to December 31, 9999 | Rounded to increments of .000, .003, or .007 seconds | YYYYMMDD<br>hh:mm:ss[.mmm]                    |
| smalldatetime  | 4               | January 1, 1900 to June 6, 2079      | 1 minute                                             | YYYYMMDD<br>hh:mm:ss[.mmm]                    |
| datetime2      | 6 to 8          | January 1, 0001 to December 31, 9999 | 100 nanoseconds                                      | YYYYMMDD<br>hh:mm:ss[.nnnnnnn]                |
| date           | 3               | January 1, 0001 to December 31, 9999 | 1 day                                                | YYYY-MM-DD                                    |
| time           | 3 to 5          | n/a – time only                      | 100 nanoseconds                                      | hh:mm:ss[.nnnnnnn]                            |
| datetimeoffset | 8 to 10         | January 1, 0001 to December 31, 9999 | 100 nanoseconds                                      | YYYY-MM-DDThh:mm:ss[.nnnnnnn][<br>{+ -}hh:mm] |



# Modul 04 – Sortieren und Filtern von Daten

# Filtern von Daten in der WHERE-Klausel mit Prädikaten

- WHERE-Klauseln verwenden Prädikate
  - Muss als logische Bedingungen ausgedrückt werden
  - Nur Zeilen, für die das Prädikat TRUE ergibt, werden akzeptiert
  - Werte von FALSE oder UNKNOWN herausgefiltert
- WHERE-Klausel folgt FROM, geht anderen Klauseln voraus
  - In der SELECT-Klausel deklarierte Aliase können nicht angezeigt werden
- Kann von SQL Server für die Verwendung von Indizes optimiert werden

# WHERE Klausel-Syntax

- Kunden aus Spanien herausfiltern:

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

- Zeilen für Bestellungen nach dem 1. Januar 2007 herausfiltern

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

- Zeilen gefiltert nach Zeit zwischen 1. Januar 2007 und 1. Januar 2008

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```



# M005 – Wildcards

Name Trainer

ppedv AG

Datum: xx.yy.zzzz

# Wie werden Wildcards verwendet?

- Wenn man nicht alle Zeichen kennt..
- werden verwendet, um ein oder mehrere Zeichen in einer Zeichenfolge zu ergänzen
- Können nur mit dem LIKE-Operator verwendet werden

# Arten von Wildcards im SQL Server

| Symbol | Beschreibung                                        | Beispiel      |
|--------|-----------------------------------------------------|---------------|
| %      | (k)ein oder mehrere Zeichen                         | ber% oder a%o |
| _      | nur <u>ein</u> Zeichen                              | _onde_n       |
| []     | jedes Zeichen in den Klammern steht für ein Zeichen | [bsp]         |
| ^      | jedes Zeichen <u>nicht</u> in den Klammern          | [^oa]         |
| -      | jedes Zeichen im gewählten Bereich                  | [a-c]         |
|        |                                                     |               |

Hinweis: Wildcards können in Kombination verwendet werden. Bspw: [abc]%\_k[1-4]%\_t

# Beispiele für jede Wildcard

- %

```
SELECT * FROM Customers
WHERE City LIKE 'ber%';
```

→ Berlin, Bern, Bergamo

- \_

```
SELECT * FROM Customers
WHERE City LIKE '_ünchen';
```

→ München

- [] und -

```
SELECT * FROM Customers
WHERE City LIKE ',[bsp]%';
```

→ Ein Zeichen, entweder b oder s oder p

# Beispiele für jede Wildcard

```
SELECT * FROM Customers
WHERE City LIKE ',[a-c]%' ;
```

→ Ein Zeichen von **a** bis **c**

## Sonderbeispiel:

PIN-Bankkarte: [0-9] [0-9] [0-9] [0-9] → jede einzelne PIN-Eingabe (Zeichen) wird mithilfe der []-Wildcard auf Richtigkeit geprüft!

Was wäre, wenn... PIN-Eingabe = 0379 → Prüfung: >0 und <1500?

→ Geht nicht mit ">0 und <1500", wegen 0 zu Beginn!



# Beispiele für jede Wildcard

- ^

```
SELECT * FROM Customers
WHERE City LIKE ,[^abc]%' ;
```

→ ein Zeichen außer a & b & c

# Sonderfälle

- Was wäre, wenn eine Datei '%' oder '' im Name hätte?

Für %:

```
SELECT * FROM Customers
WHERE CompanyName LIKE '%[%]%' ;
```

Für ':

```
SELECT * FROM Customers
WHERE CompanyName LIKE '%[']%' ;
```



# Modul 06 – Abfragen auf mehrere Tabellen

# Die FROM Klausel und virtuelle Tabellen

- FROM Klausel legt die Ursprungstabellen fest, die im SELECT verwendet werden
- FROM Klausel kann sowohl Tabellen als auch Operator beinhalten
- Ergebnis der FROM Klausel ist eine virtuelle Tabelle
  - Nachfolgende Anweisungen verwenden diese virtuelle Tabelle
- FROM Klausel kann Tabellen Aliase beinhalten
  - Nützlich für spätere Phasen der Abfrage

# Überblick über Joins

| Join Type    | Description                                                                                                                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Cross</b> | Kombiniert alle Zeilen beider Tabellen miteinander (Erstellt ein Kartesisches Produkt)                                                                                                                    |
| <b>Inner</b> | Startet mit einem kartesischen Produkt, setzt später aber Filter und filtert nach dem angegebenen Prädikat                                                                                                |
| <b>Outer</b> | Beginnt mit kartesischem Produkt; Alle Zeilen aus der zugewiesenen Tabelle werden beibehalten und die übereinstimmenden Zeilen aus anderen Tabellen abgerufen. Zusätzliche NULL als Platzhalter eingefügt |

# TSQL Join Syntax

- ANSI SQL-92
  - Tabellen werden über JOIN Operator in der FROM Klausel verbunden

```
SELECT ...
FROM Table1 JOIN Table2
 ON <on_predicate>
```

- ANSI SQL-89
  - Tabellen werden über Kommas in FROM Klausel verknüpft
    - Nicht empfohlen!!!

```
SELECT ...
FROM Table1, Table2
WHERE <where_predicate>
```

# Wie funktionieren Inner Joins?

- Rückgabe nur von Zeilen, die in beiden Tabellen einen Wert haben
- Vergleicht die Zeilen basierend auf einem Attribut das als Prädikat mitgegeben wird
  - ON Klausel in SQL-92 Syntax (bevorzugt)
  - WHERE Klausel in SQL-89 Syntax (nicht empfohlen)
- Wieso die Filterung in der ON Klausel?
  - Logische Unterscheidung zwischen der Filterung des Joins (ON) und des Ergebnisses (WHERE)
  - Abfrage wird nicht optimiert

# Inner Join Syntax und Beispiel

- Einzelnen Tabellen werden in FROM Klausel aufgelistet
  - Getrennt durch JOIN Operatoren
- Tabellen Aliase sind empfehlenswert

```
SELECT o.orderid,
 o.orderdate,
 od.productid,
 od.unitprice,
 od.qty
FROM Sales.Orders AS o
 JOIN Sales.OrderDetails AS od
 ON o.orderid = od.orderid;
```



# Wie funktionieren Outer Joins?

- Gibt alle Zeilen aus einer Tabelle und alle übereinstimmenden Zeilen aus der zweiten Tabelle zurück
- Zeilen aus einer Tabelle werden beibehalten
  - Bezeichnet mit LEFT, RIGHT, FULL Schlüsselwort
  - Alle Zeilen aus der beibehaltenen Tabelle werden in der Ergebnismenge ausgegeben
- Passende Zeilen aus anderen Tabellen werden zusätzlich in der Ergebnismenge ausgegeben
- Zusätzliche Zeilen wurden zu Ergebnissen für nicht übereinstimmende Zeilen hinzugefügt
  - NULLs wurden an Stellen hinzugefügt, an denen Attribute nicht übereinstimmen

# Outer Join Syntax und Beispiele

- Gibt alle Zeilen aus der ersten Tabelle zurück, nur Treffer aus der zweiten Tabelle:

```
FROM t1 LEFT OUTER JOIN t2 ON
 t1.col = t2.col
```

- Gibt alle Zeilen aus der zweiten Tabelle zurück, nur die Übereinstimmungen von der ersten Tabelle:

```
FROM t1 RIGHT OUTER JOIN t2 ON
 t1.col = t2.col
```

- Gibt nur Zeilen aus der ersten Tabelle zurück, ohne Übereinstimmung in der zweiten Tabelle:

```
FROM t1 LEFT OUTER JOIN t2 ON
 t1.col = t2.col
WHERE t2.col IS NULL
```

# Wie funktionieren Cross Joins?

- Kombiniert jede Zeile aus der ersten Tabelle mit jeder Zeile aus der zweiten Tabelle
- Alle möglichen Kombinationen werden ausgegeben
- Logische Grundlage für inner und outer Joins
  - Inner Join beginnt mit kartesischem Produkt, fügt Filter hinzu
  - Outer-Join verwendet kartesische Ausgabe, gefiltert, fügt nicht übereinstimmende Zeilen zurück  
(mit NULL-Platzhaltern)
- Aufgrund der kartesischen Produktausgabe ist dies normalerweise keine gewünschte Join-Form

# Cross Join Syntax

- Es wird nicht verglichen, keine ON Klausel benötigt
- Gibt alle Zeilen aus der linken Tabelle zusammen mit jeder Zeile aus der rechten Tabelle zurück (ANSI SQL-92 syntax):

```
SELECT ...
FROM t1 CROSS JOIN t2
```


- Gibt alle Zeilen aus der linken Tabelle zurück, die mit jeder Zeile aus der rechten Tabelle kombiniert wird (ANSI SQL-89-Syntax):

```
SELECT ...
FROM t1, t2
```

# Wie funktionieren Self Joins?

- Wieso Self Joins benutzen?
  - Vergleich von zwei Spalten in gleicher Tabelle möglich
- Erstellt zwei Instanzen derselben Tabelle in der F
  - Es wird mindestens ein Alias benötigt

```
SELECT e.empid, e.lastname,
 e.title, e.mgrid, m.lastname
FROM HR.Employees AS e
JOIN HR.Employees AS m
ON e.mgrid=m.empid;
```



|                 |
|-----------------|
| empid           |
| lastname        |
| firstname       |
| title           |
| titleofcourtesy |
| birthdate       |
| hiredate        |
| address         |
| city            |
| region          |
| postalcode      |
| country         |
| phone           |
| mgrid           |



# Modul 07 – DISTINCT, TOP, ORDER BY

# Die ORDER BY Klausel

- ORDER BY sortiert Zeilen in Ergebnissen
  - Keine garantierte Reihenfolge der Zeilen ohne ORDER BY
  - ORDER BY garantiert die Sortierreihenfolge des Ergebnisses
  - Letzte Klausel, die logisch verarbeitet werden soll
- ORDER BY kann sich beziehen auf:
  - Spalten nach Name oder Spaltenalias
  - Spalten, die nicht zur SELECT-Liste gehören
- Sortierreihenfolge mit ASC oder DESC deklarieren
  - Asc = default

# ORDER BY Syntax

- Schreiben von ORDER BY mit Spaltennamen:

```
SELECT <select list>
FROM <table source>
ORDER BY <column1_name>, <column2_name>;
```

- Writing ORDER BY using column aliases:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <alias1>, <alias2> ASC|DESC;
ORDER BY 1, 3 desc, 5 asc --für die Faulen
Spaltennummer
```



# Die DISTINCT Klausel

- bei SELECT-Anweisung können gleiche Werte erscheinen
- DISTINCT-Anweisung beschränkt sich nur Ergebniszeilen → keine Duplikate → verwirft doppelte Zeilen

# DISTINCT Syntax

- die DISTINCT-Anweisung wird gleich nach der SELECT-Anweisung geschrieben

```
SELECT DISTINCT City FROM Customers;
```

- vor der DISTINCT-Anweisung, kann eine COUNT-Anweisung geschrieben werden, um die Anzahl der DISTINCT-Werte zu erhalten

```
SELECT COUNT(DISTINCT City) FROM Customers;
```

# Die TOP Klausel

- durch die TOP-Anweisung kann eine beliebige Anzahl an Werten zurückgegeben werden
  - Die ersten je nach Sortierung
  - Ohne ORDER BY → fast Zufall
  - Mit WHERE → genauere Suche

Hinweis: Die TOP Klausel wird nicht von jedem Datenbanksystem unterstützt!

# TOP Syntax

- Nur TOP Klausel:

```
SELECT TOP 3 * FROM Customers;
```

- TOP Klausel mit ORDER BY:

```
SELECT TOP 3 * FROM Customers
ORDER BY Country;
```

- TOP Klausel mit WHERE:

```
SELECT TOP 3 * FROM Customers
WHERE Country = 'Germany';
```

# Modul 08 – Gruppieren und Aggregieren von Daten

# Verwendung von Aggregate Funktionen

- Aggregate Funktionen:
  - Geben einen Skalar Wert zurück (ohne Spaltenname)
  - Ignoriert NULL Werte (außer in der COUNT Funktion)
  - Verwendung möglich in:
    - SELECT, HAVING und ORDER BY Klausel
  - Häufig verwendet in Verbindung mit der GROUP BY Klausel

```
SELECT AVG(unitprice) AS avg_price,
MIN(qty) AS min_qty,
MAX(discount) AS max_discount
FROM Sales.OrderDetails;
```

```
avg_price min_qty max_discount

26.2185 1 0.250
```

# Übersicht über die Aggregate Funktionen

| Funktion | Beschreibung                                      |
|----------|---------------------------------------------------|
| SUM      | Summiert alle Werte der Spalten auf               |
| MIN      | Gibt den kleinsten Wert der Spalte zurück         |
| MAX      | Gibt den größten Wert der Spalte zurück           |
| AVG      | Gibt den Durchschnittswert der Spalte zurück      |
| COUNT    | Gibt die Anzahl der vorhandenen Datensätze zurück |

# DISTINCT in Verbindung mit Aggregate Funktionen

- DISTINCT sorgt dafür, dass nur verschiedene Werte von der Funktion betrachtet werden
- DISTINCT in Aggregate Funktionen löscht NUR Werte und keine kompletten Zeilen

Beispiel:

```
SELECT empid, YEAR(orderdate) AS orderyear,
COUNT(custid) AS all_custs,
COUNT(DISTINCT custid) AS unique_custs
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

| empid | orderyear | all_custs | unique_custs |
|-------|-----------|-----------|--------------|
| 1     | 2006      | 26        | 22           |
| 1     | 2007      | 55        | 40           |
| 1     | 2008      | 42        | 32           |
| 2     | 2006      | 16        | 15           |



# Wie benutze ich die GROUP BY Klausel?

- GROUP BY erstellt Gruppen für Ausgabezeilen gemäß einer eindeutigen Kombination von Werten, die in der GROUP BY-Klausel angegeben sind

```
SELECT <select_list>
FROM <table_source>
WHERE <search_condition>
GROUP BY <group_by_list>;
```

- GROUP BY berechnet einen Summenwert für Aggregatfunktionen in nachfolgenden Phasen (für jede einzelne Zeile)

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

# Logische Abfolge von Operationen

| Logical Order | Phase    | Comments           |
|---------------|----------|--------------------|
| 5             | SELECT   |                    |
| 1             | FROM     |                    |
| 2             | WHERE    |                    |
| 3             | GROUP BY | Creates groups     |
| 4             | HAVING   | Operates on groups |
| 6             | ORDER BY |                    |

- Wenn eine Abfrage GROUP BY verwendet, arbeiten alle nachfolgenden Phasen mit den Gruppen und nicht mit den Quellzeilen

# GROUP BY Ablauf

```
SELECT SalesOrderID,
SalesPersonID, CustomerID
FROM Sales.SalesOrderHeader;
```

| SalesOrder ID | SalesPerson ID | CustomerID |
|---------------|----------------|------------|
| 43659         | 279            | 29825      |
| 43660         | 279            | 29672      |
| 43661         | 282            | 29734      |
| 43662         | 282            | 29994      |
| 43663         | 276            | 29565      |
| ...           | ...            | ...        |
| 75123         | NULL           | 18759      |



```
WHERE
CustomerID IN
(30097,
30098)
```

| SalesOrder ID | SalesPerson ID | Customer ID |
|---------------|----------------|-------------|
| 51803         | 290            | 29777       |
| 69427         | 290            | 29777       |
| 44529         | 278            | 30010       |
| 46063         | 278            | 30010       |

```
GROUP BY
SalesPersonID
```



| SalesPersonID | Count(*) |
|---------------|----------|
| 278           | 2        |
| 290           | 2        |

# GROUP BY mit Aggregate Funktionen

- Aggregate Funktionen werden normalerweise in der SELECT Klausel verwendet

```
SELECT custid, COUNT(*) AS count
FROM Sales.Orders
GROUP BY custid;
```

- Aggregate Funktionen beziehen sich auf alle Spalten, nicht nur auf die der SELECT Klausel

# Filtern von Gruppierten Daten

- HAVING Klausel:

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid
HAVING COUNT(*) > 10;
```

- Jeder Gruppe muss die in der HAVING Klausel benannte Bedingung erfüllen
- HAVING Klausel wird nach GROUP BY Klausel ausgeführt



# Modul 09 – Views und Prozeduren

# Views

Grundaufbau:

```
CREATE VIEW v*ViewName*
as
Abfrage an die Datenbank
GO
```

- v als Bezeichner um Views einfacher von anderen Elementen unterscheiden zu können
- Ausgabe der Abfrage wird in Tabellenform gespeichert
- sind gut, um sich nicht mehr verändernde Daten zu speichern
- kann kein **ORDER BY** enthalten

# Schreiben von Abfragen, die Ergebnisse aus Sichten zurückgeben

- Ansichten können in einer SELECT-Anweisung wie eine Tabelle referenziert werden
- Sichten sind benannte Tabellenausdrücke mit Definitionen, die in einer Datenbank gespeichert sind.
- Sichten können eine Sicherheitsschicht (Backup) für eine Datenbank bereitstellen

```
CREATE VIEW HR.EmpPhoneList
AS
SELECT empid, lastname, firstname, phone
FROM HR.Employees;
```



# Prozeduren

Grundaufbau:

```
CREATE PROCEDURE usp*ProzedureName*
(
 @Variablename1 (Datentyp),
 @Variablename2 (Datentyp)
)
as
Abfrage an die Datenbank
GO
```

- usp als Bezeichner um Prozeduren von anderen Elementen unterscheiden zu können
- Parameter nicht notwendig, machen Prozedur aber nützlich um eine spezielle Filterung durchzuführen

# Schreiben von Abfragen, die Ergebnisse aus Prozeduren zurückgeben

- Ergebnisse von Prozeduren werden NICHT über den SELECT Operator aufgerufen
- Einer Prozedur ist die gespeicherte Abfrage hinterlegt, nicht das Ergebnis dieser

```
EXEC uspBeispielProzedur „Parameter1“, „Parameter2“
```

```
GO
```

```
* Ausgabe *
```

be der Prozedur



# Modul 10 – SET-Operatoren

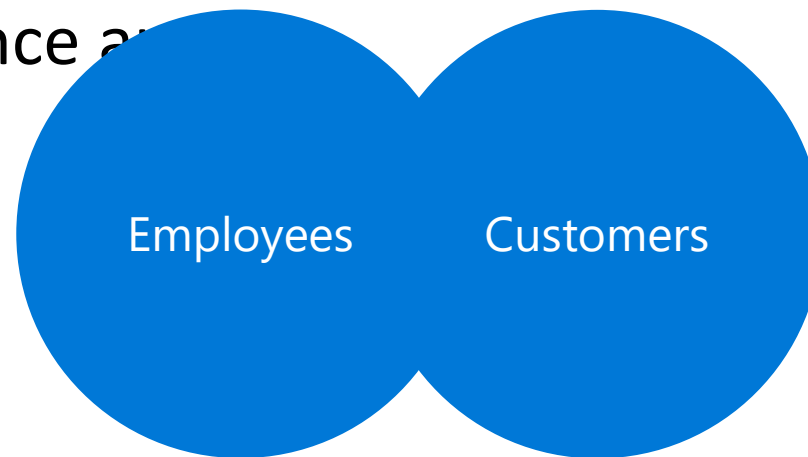
# Interaktionen zwischen Sets (Sätzen)

- Die Ergebnisse von zwei Eingabe Abfragen können weiter manipuliert werden -> Set Operatoren
- Sätze können kombiniert oder verglichen werden
- Beide Sätze müssen die gleiche Anzahl kompatibler Spalten haben
- ORDER BY nicht zulässig in Eingabe Abfragen, kann aber für Ergebnis der Set-Operation verwendet werden
- Nullen als gleich beim Vergleich von Mengen

```
<SELECT query_1>
<set_operator>
<SELECT query_2>
[ORDER BY <sort_list>];
```

# Der UNION Operator

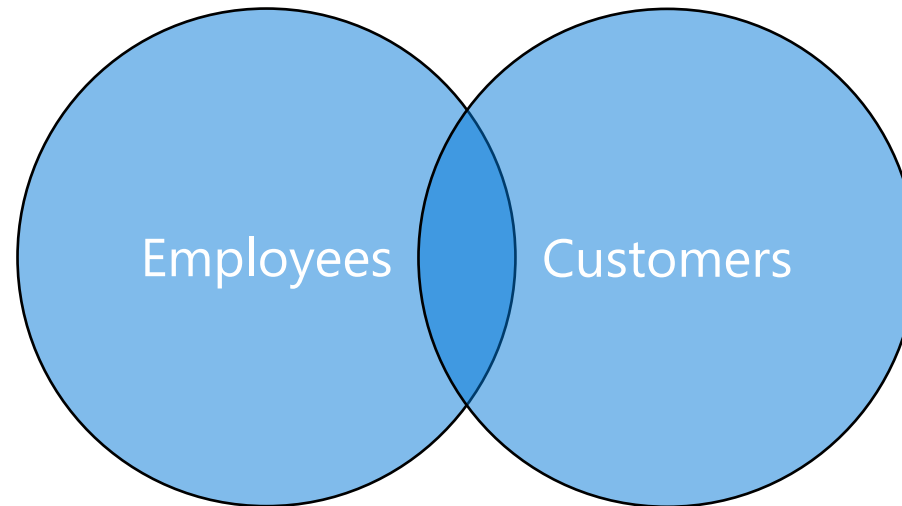
- UNION gibt die kombinierte Variante beider Tabellen zurück
- Doppelte Datensätze werden beim „fusionieren“ entfernt (wirkt sich auf die Performance aus)



```
SELECT country, region, city FROM HR.Employees
UNION
SELECT country, region, city FROM Sales.Customers;
```

# Der INTERSECT Operator

- INTERSECT gibt alle Datensätze zurück, die in beiden Eingabemengen enthalten sind



```
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```



# Modul 11 - Unterabfragen

# Was sind Unterabfragen?

- Unterabfragen sind Abfragen integriert in einer anderen Abfrage
- Ergebnisse der Unterabfrage werden an die Hauptabfrage weitergeleitet
  - Unterabfrage ist wie eine Aussage für Hauptabfrage
- Unterabfragen können in sich abgeschlossen oder korreliert sein
  - Unabhängige Unterabfragen haben keine Abhängigkeit von der äußeren Abfrage (in sich abgeschlossene)
  - Korrelierte Unterabfragen verlassen sich auf Werte aus der Hauptabfrage in der sie stehen
- Unterabfragen können an vielen Stellen auftreten:
  - Als Spalte: dann darf sich nur eine „Zelle“ aus er Abfrage ergeben
  - Als Tabellenergebnis: dann ist ein FROM (Unterabfrage) möglich, wobei die



# Vergleich zwischen Unterabfragetypen

## Abgeschlossene Unterabfrage:

Hauptabfrage:

```
SELECT orderid, productid,
unitprice, qty
FROM Sales.OrderDetails
WHERE orderid = ()
```

Unterabfrage:

```
SELECT MAX(orderid) AS
lastorder
FROM Sales.Orders
```



Hauptabfrage

```
SELECT orderid, empid, orderdate
FROM Sales.Orders AS 01
WHERE
orderdate = ()
```

Unterabfrage:

```
SELECT MAX(orderdate) FROM
Sales.Orders AS 02
WHERE 02.empid = 01.empid
```



# EXISTS Parameter

- Wenn eine Unterabfrage mit dem EXISTS Parameter verwendet wird, dann führt sie einen Existenztest durch
  - Gibt lediglich wahr (true) oder falsch (false) zurück
- Wenn Zeilen als Ergebnis der Unterabfrage zurückgegeben werden würden, dann gibt die Unterabfrage ein true zurück anstatt der Zeilen
- Andererseits gibt die Unterabfrage ein false zurück
- `WHERE [NOT] EXISTS (subquery)`

# Exists Syntax und Beispiele

- EXISTS folgt keine Spalte oder ähnliches sondern immer eine

```
SELECT custid, companyname
FROM Sales.Customers AS c
WHERE EXISTS (
 SELECT *
 FROM Sales.Orders AS o
 WHERE c.custid=o.custid);
```

```
SELECT custid, companyname
FROM Sales.Customers AS c
WHERE NOT EXISTS (
 SELECT *
 FROM Sales.Orders AS o
 WHERE c.custid=o.custid);
```



# Modul 12 - DML

# INSERT zum Daten einfügen

- Die Anweisung INSERT ... VALUES fügt eine neue Zeile in die Tabelle

```
INSERT INTO Sales.OrderDetails
 (orderid, productid, unitprice, qty, discount)
VALUES (10255, 39, 18, 2, 0.05);
```

- ```
INSERT INTO Sales.OrderDetails  
    (orderid, productid, unitprice, qty, discount)  
  
VALUES  
    (10256, 39, 18, 2, 0.05),  
    (10258, 39, 18, 5, 0.10);
```

n
befüllt sein)

INSERT mit Datenanbietern verwenden

- INSERT ... SELECT um Zeilen aus einer anderen Tabelle einzufügen

```
INSERT Sales.OrderDetails  
(orderid, productid, unitprice, qty, discount)  
  
SELECT * FROM NewOrderDetails
```

- INSERT INTO Production.Products
(productID, productname, supplierid, categoryid, unitprice)
EXEC Production.AddNewProducts;

cherten

SELECT INTO

SELECT -> INTO ist genau das gleiche wie INSERT <- SELECT

- Erstellt auch eine Tabelle für die Ausgabe, die auf der Ausgabe selbst basiert
- Die neue Tabelle basiert auf der Spaltenstruktur der Abfrage

```
SELECT * INTO NewProducts FROM PRODUCTION.PRODUCTS  
WHERE ProductID >= 70
```

gen

Verwendung von UPDATE

- UPDATE verändert alle Zeilen in der angegebenen Tabelle oder View
 - Es sei denn, Zeilen werden mit einer WHERE-Klausel gefiltert oder mit einer JOIN-Klausel eingeschränkt

```
• UPDATE Production.Products
  SET    unitprice = (unitprice * 1.04)
WHERE   categoryid = 1 AND discontinued = 0
;
```

```
UPDATE Production.Products
  SET    unitprice *= 1.04
          -- Using compound
          -- assignment operators
WHERE   categoryid = 1 AND discontinued = 0;
```


Datentypen zuweisen

- Tabelle wird mit CREATE erstellt
 - In Klammern kommen Tabellenspalten + Datentyp:

```
CREATE TABLE PERSON (  
    PersonenID int identity primary key,  
    Nachname nvarchar(40) not NULL,  
    Vorname nvarchar(40) notNULL,  
    [Alter] int  
)
```

- Tabelle besteht nun aus 4 Spalten -> PersonenID ist eindeutige Spalte

Verwendung von IDENTITY

Die IDENTITY-Eigenschaft generiert automatische Spaltenwerte

- Optionale Start- und Inkrementwerte können bereitgestellt werden

```
CREATE TABLE Production.Products  
(PID int IDENTITY(1,1) NOT NULL, Name VARCHAR(15),...)
```

- In nur einer Spalte in einer Tabelle kann IDENTITY definiert sein

```
INSERT INTO Production.Products (Name,...)
```

- VALUES ('MOC 2072 Manual',...) Anweisung
weggelassen werden

Verwendung von Sequenzen

Sequenzobjekte wurden zuerst in SQL Server 2012 hinzugefügt

- Unabhängige Objekte in der Datenbank
 - Mehr Flexibilität als die IDENTITY Eigenschaft
 - Kann als Standardwert/eindeutiger Wert einer Spalte verwendet werden
- Mit CREATE / ALTER / DROP-Anweisungen verwalten

- -- Define a sequence

```
CREATE SEQUENCE dbo.InvoiceSeq AS INT START WITH 1  
INCREMENT BY 1;
```

```
-- Retrieve next available value from sequence
```

```
SELECT NEXT VALUE FOR dbo.InvoiceSeq;
```

DELETE zum Löschen von Daten

- um Daten in der Tabelle zu entfernen, wird die DELETE-Anweisung verwendet

Achtung: Bei Verwendung von DELETE unbedingt mithilfe der WHERE Klausel angeben, welche Daten von der Tabelle gelöscht werden sollen. Bei Nicht-Verwendung der WHERE-Klausel werden alle Daten von der Tabelle gelöscht!!!

- Aufbau: `DELETE FROM <table_name> WHERE <condition>;`

Beispiel der DELETE Klausel

```
DELETE FROM Customers WHERE CustomerName = "Around the Horn";
```

- In diesem Beispiel wird der Kundenname (CustomerName) „Around the Horn“ mithilfe der DELETE Klausel aus der Tabelle „Customers“ gelöscht