

Datenbankabfrage mit SQL



Inhalt

| | | |
|-------|---|----|
| 1 | Relationale Datenbanksysteme | 9 |
| 1.1 | Allgemeines..... | 9 |
| 1.2 | Datenbanksysteme..... | 9 |
| 1.3 | Ebenen in relationalen Datenbanksystemen | 10 |
| 1.4 | Werkzeuge der DBS..... | 12 |
| 1.5 | Was ist ein relationales Datenbankmodell? | 12 |
| 2 | Datenstrukturen..... | 13 |
| 2.1 | Skalare Werte und Basisdatentypen..... | 13 |
| 2.2 | Domänen | 13 |
| 2.3 | Relationen (Tabellen)..... | 15 |
| 2.3.1 | Relationsarten | 15 |
| 2.3.2 | Fundamentalprinzip relationaler Datenbanken | 15 |
| 3 | Integritätsbedingungen | 16 |
| 3.1 | Primärschlüssel | 16 |
| 3.2 | Null- Werte..... | 16 |
| 3.3 | Primärschlüsselintegrität | 17 |
| 3.4 | Fremdschlüssel | 17 |
| 3.5 | Referentielle Integrität..... | 17 |
| 3.6 | Operationsregeln für Fremdschlüssel | 17 |
| 3.6.1 | NUL – Regel..... | 17 |
| 3.6.2 | Löscherregel | 17 |
| 3.6.3 | Änderungsregeln..... | 18 |
| 3.7 | Beispieldatenbank "Standard" | 18 |
| 4 | Methoden zum Datenbankentwurf | 20 |
| 4.1 | ER- Modell..... | 20 |
| 4.1.1 | Die 1:1 Beziehung | 21 |
| 4.1.2 | Die 1:C Beziehung | 21 |
| 4.1.3 | Die 1:N Beziehung | 21 |
| 4.1.4 | Die 1:NC Beziehung | 21 |
| 4.1.5 | Die M:N Beziehung..... | 22 |
| 4.1.6 | Die N:MC Beziehung oder NC:MC Beziehung | 22 |
| 4.2 | Normalisierung | 23 |
| 4.2.1 | Erste Normalform | 23 |
| 4.2.2 | Zweite Normalform | 24 |
| 4.2.3 | Dritte Normalform..... | 25 |
| 4.3 | Zusammenfassung | 25 |
| 5 | Datenbankabfragen mit SQL..... | 27 |
| 5.1 | Geschichtlicher Abriss zu SQL | 27 |
| 5.1.1 | Ein Überblick | 27 |
| 5.1.2 | Was ist Transact- SQL? | 28 |

| | | |
|--------|--|----|
| 6 | SELECT- Anweisung | 28 |
| 6.1 | SELECT- Liste und FROM- Klausel | 29 |
| 6.1.1 | Ermitteln von Daten ohne Datenquelle | 29 |
| 6.1.2 | Spaltenliste festlegen und Basistabelle angeben | 29 |
| 6.2 | Die WHERE – Klausel..... | 30 |
| 6.2.1 | Vergleichsoperatoren | 31 |
| 6.2.2 | Schlüsselwort BETWEEN | 32 |
| 6.2.3 | Schlüsselwort IN..... | 32 |
| 6.2.4 | Schlüsselwort LIKE | 33 |
| 6.2.5 | Arbeiten mit NULL- Werten | 34 |
| 6.2.6 | Festlegen mehrerer Suchbedingungen für mehrere Spalten..... | 35 |
| 6.3 | Formatieren von Ergebnissen | 36 |
| 6.3.1 | Erläuternder Text (Literalen)..... | 37 |
| 6.3.2 | Ändern von Spaltennamen..... | 37 |
| 6.3.3 | Entfernen doppelter Reihen aus dem Ergebnis..... | 37 |
| 6.3.4 | Sortieren der Ergebnismenge..... | 38 |
| 6.3.5 | Auflisten der TOP n – Werte | 39 |
| 6.3.6 | Case Ausdrücke..... | 40 |
| 6.4 | Berechnen der Ergebnismengen..... | 41 |
| 6.4.1 | Arithmetische Operatoren zur Berechnung | 41 |
| 6.4.2 | Operator + für das Verketten von Zeichenfolgen | 41 |
| 6.4.3 | Operator CONCAT für das Verketten von Zeichenfolgen | 42 |
| 6.4.4 | Skalare Funktionen | 42 |
| 6.5 | Gruppieren und Zusammenfassen von Daten..... | 47 |
| 6.5.1 | Aggregatfunktionen | 47 |
| 6.5.2 | Berechnen vollständiger Zusammenfassungswerte für eine Tabelle | 53 |
| 6.5.3 | Rangfolgen für gruppierte Daten..... | 58 |
| 6.6 | SELECT INTO- Klausel | 62 |
| 6.6.1 | Temporäre Tabellen | 62 |
| 6.6.2 | Temporäre Tabellen durch Tabellenvariable | 62 |
| 6.6.3 | Permanente Tabellen | 62 |
| 6.7 | Mengenoperator UNION | 63 |
| 6.8 | Except und Intersect | 64 |
| 6.8.1 | Except Operator | 64 |
| 6.8.2 | Intersect Operator | 64 |
| 6.9 | Die TABLESAMPLE- Klausel..... | 65 |
| 6.10 | Die Operatoren PIVOT und UNPIVOT | 65 |
| 6.10.1 | PIVOT | 65 |
| 6.10.2 | UNPIVOT | 66 |
| 6.11 | Unterabfragen | 67 |
| 6.11.1 | Einfache Unterabfragen..... | 67 |
| 6.11.2 | Korrelierte Unterabfragen..... | 70 |
| 6.11.3 | EXISTS und NOT EXISTS in Unterabfragen | 70 |

| | | |
|--------|---|-----|
| 6.12 | Verknüpfung von Tabellen – JOIN..... | 71 |
| 6.12.1 | Das kartesische Produkt – CROSS JOIN..... | 71 |
| 6.12.2 | Der INNER JOIN | 72 |
| 6.12.3 | OUTER JOIN | 74 |
| 6.12.4 | APPLY- Operator..... | 76 |
| 6.12.5 | Die OVER Klausel..... | 77 |
| 6.13 | Abfragen mit CTE | 79 |
| 6.14 | Volltextdaten abfragen..... | 80 |
| 6.14.1 | Prädikate CONTAINS und FREETEXT | 80 |
| 6.14.2 | Funktionen CONTAINSTABLE und FREETEXTTABLE | 83 |
| 6.15 | Zusammenfassung SELECT- Anweisung | 84 |
| 7 | Datenänderung mit SQL..... | 85 |
| 7.1 | Einfügen von Daten | 85 |
| 7.1.1 | In der Reihenfolge der Tabellenspalten | 85 |
| 7.1.2 | Veränderte Reihenfolge | 85 |
| 7.1.3 | Unbekannte Werte aufnehmen | 86 |
| 7.1.4 | Default Werte aufnehmen | 86 |
| 7.1.5 | Insert mit DEFAULT VALUES..... | 86 |
| 7.1.6 | Einfügen von Daten mit Select und Execute | 87 |
| 7.1.7 | Insert mit ROW CONSTRUCTOR | 88 |
| 7.1.8 | Insert mit der Option BULK der OPENROWSET- Funktion | 88 |
| 7.1.9 | Verwenden von Output bei Insert..... | 88 |
| 7.1.10 | Insert mit TOP n..... | 89 |
| 7.1.11 | Insert mit CTE..... | 89 |
| 7.2 | Ändern von Daten | 90 |
| 7.2.1 | Einfache Update- Anweisung | 90 |
| 7.2.2 | Update mit der TOP- Klausel | 91 |
| 7.2.3 | Update mit der Output- Klausel | 91 |
| 7.2.4 | Update mit CTE | 92 |
| 7.2.5 | Update mit .WRITE- Klausel zum Ändern von Daten | 92 |
| 7.2.6 | Update mit .WRITE- Klausel zum Hinzufügen und Entfernen von Daten | 93 |
| 7.3 | Löschen von Daten | 94 |
| 7.3.1 | Einfache DELETE- Anweisungen | 94 |
| 7.3.2 | DELETE mit der TOP- Klausel..... | 95 |
| 7.3.3 | DELETE mit der OUTPUT-Klausel | 95 |
| 7.4 | MERGE- Statement..... | 96 |
| 8 | Arbeiten mit XML..... | 99 |
| 8.1 | XML- Daten | 99 |
| 8.2 | XML- Schema | 100 |
| 8.2.1 | Aufbau eines XML- Schemas..... | 101 |

| | | |
|--------|--|-----|
| 8.3 | Typisieren von XML- Daten..... | 105 |
| 8.3.1 | XML- Schema erstellen..... | 105 |
| 8.3.2 | XML- Schema ändern | 106 |
| 8.3.3 | XML- Schema löschen | 106 |
| 8.3.4 | XML- Schema verwenden | 107 |
| 8.4 | Relationale Daten in XML konvertieren..... | 107 |
| 8.4.1 | Verwenden von FOR XML RAW | 108 |
| 8.4.2 | Verwenden von FOR XML AUTO | 110 |
| 8.4.3 | Verwenden von FOR XML EXPLICIT | 112 |
| 8.4.4 | TYPE- Direktive | 114 |
| 8.4.5 | PATH- Direktive..... | 115 |
| 8.5 | Arbeit mit XML- Daten vom Datentyp XML..... | 116 |
| 8.5.1 | Welche Speicherform für XML- Daten?..... | 116 |
| 8.5.2 | XPath und XQuery | 117 |
| 8.5.3 | Methoden für Abfragen aus XML- Datentypen..... | 122 |
| 9 | Sichten..... | 127 |
| 9.1 | Erstellen von Sichten..... | 128 |
| 9.2 | Ändern von Sichten | 129 |
| 9.3 | Löschen von Sichten..... | 130 |
| 9.4 | Ändern von Daten über eine Sicht..... | 130 |
| 9.4.1 | Insert..... | 130 |
| 9.4.2 | Update | 131 |
| 9.4.3 | Delete | 132 |
| 9.5 | Indizierte Sichten..... | 132 |
| 9.6 | Partitionierte Sichten..... | 133 |
| 10 | SQL Server Datenbanken | 135 |
| 10.1 | Datenbanktypen | 136 |
| 10.2 | Systemdatenbanken: | 136 |
| 10.3 | Server- und Datenbankinformationen abfragen..... | 138 |
| 10.3.1 | Systemtabellen - Systemsichten | 138 |
| 10.4 | Benutzerdatenbanken..... | 144 |
| 10.4.1 | Art der Datenspeicherung..... | 144 |
| 10.4.2 | Arbeitsweise des Transaktionsprotokolls..... | 146 |
| 10.4.3 | Erstellen von Datenbanken..... | 151 |
| 10.4.4 | Ändern von Datenbanken..... | 161 |
| 10.4.5 | Löschen von Datenbanken..... | 167 |
| 10.4.6 | Datenbanksnapshot..... | 167 |
| 10.4.7 | Datenintegrität pflegen | 168 |
| 11 | SQL Server Tabellen | 171 |
| 11.1 | Erstellen von Datentypen und Tabellen | 172 |
| 11.1.1 | Datentypen | 172 |
| 11.1.2 | Anordnung der Daten in einer Zeile | 176 |
| 11.1.3 | Organisation von großen Datentypen..... | 176 |
| 11.1.4 | Organisation der Daten vom Typ TEXT, NTEXT und IMAGE | 177 |

| | | |
|---------|--|-----|
| 11.1.5 | Spalten mit geringer Dichte | 177 |
| 11.1.6 | Basistabellen erstellen..... | 178 |
| 11.1.7 | Basistabellen löschen | 191 |
| 11.1.8 | Basistabellen ändern..... | 191 |
| 11.2 | Implementieren der Datenintegrität..... | 193 |
| 11.2.1 | Einschränkungen..... | 194 |
| 11.2.2 | Erstellen von Einschränkungen..... | 195 |
| 11.2.3 | Verzögern und Deaktivieren von Einschränkungen | 198 |
| 11.2.4 | Standardwerte und Regeln | 199 |
| 12 | Indizes | 201 |
| 12.1 | Planen von Indizes | 201 |
| 12.2 | Organisationsstruktur von Tabellen und Indizes | 202 |
| 12.2.1 | Organisation von Tabellen | 202 |
| 12.2.2 | Partitionen..... | 202 |
| 12.2.3 | Gruppierte Tabellen, Heaps und Indizes | 202 |
| 12.2.4 | XML-Indizes..... | 203 |
| 12.2.5 | Zuordnungseinheiten | 203 |
| 12.3 | Indexarchitektur | 203 |
| 12.3.1 | Verwenden von Heap | 203 |
| 12.3.2 | Verwenden eines gruppierten Index..... | 205 |
| 12.3.3 | Verwenden eines nicht gruppierten Index | 206 |
| 12.4 | Richtlinien für die Indizierung | 207 |
| 12.5 | Erstellen von Indizes | 208 |
| 12.5.1 | Füllgrad von Indizes..... | 209 |
| 12.5.2 | Eindeutige Indizes | 211 |
| 12.5.3 | Index für Spalten mit variabler Breite | 211 |
| 12.5.4 | Zusammengesetzte Indizes..... | 211 |
| 12.5.5 | Deckende Indizes | 212 |
| 12.5.6 | Index mit eingeschlossenen Spalten | 212 |
| 12.5.7 | Gefilterte Indizes | 212 |
| 12.5.8 | Partitionierte Indizes..... | 213 |
| 12.5.9 | Komprimierter Index..... | 213 |
| 12.5.10 | Indizes für berechnete Spalten..... | 216 |
| 12.5.11 | Indizierte Sichten | 216 |
| 12.6 | Die ALTER INDEX- Anweisung | 222 |
| 12.7 | Indexinformationen abrufen | 223 |
| 12.7.1 | Gespeicherte Systemprozeduren | 223 |
| 12.7.2 | Katalogsichten | 223 |
| 12.7.3 | Systemfunktionen | 223 |
| 12.7.4 | Indexfragmentierung | 224 |
| 12.8 | Index löschen | 225 |

| | | |
|---------|---|-----|
| 12.9 | Volltextindizierung | 225 |
| 12.9.1 | Volltextkataloge | 226 |
| 12.9.2 | Volltextindizes | 227 |
| 13 | Programmieren mit Transact- SQL | 229 |
| 13.1 | Sprachelemente..... | 229 |
| 13.1.1 | Skripte und Batches | 229 |
| 13.1.2 | Anweisungsblöcke | 230 |
| 13.1.3 | Kommentare | 230 |
| 13.1.4 | Meldungen | 231 |
| 13.1.5 | Verwenden von Variablen..... | 234 |
| 13.1.6 | Bedingungen mit IF..Else | 237 |
| 13.1.7 | Bedingungen mit IIF..... | 238 |
| 13.1.8 | Schleifen..... | 238 |
| 13.1.9 | TRY...Catch- Konstrukte | 239 |
| 13.1.10 | RETURN- Anweisung..... | 241 |
| 13.1.11 | CHOOSE- Anweisung | 241 |
| 13.1.12 | Dynamische SQL Anweisungen | 241 |
| 13.2 | GOTO- Anweisung | 243 |
| 13.3 | WAITFOR- Anweisung..... | 243 |
| 13.4 | Synonyme..... | 244 |
| 13.5 | Cursor | 245 |
| 13.5.1 | DECLARE- Anweisung | 246 |
| 13.5.2 | OPEN- Anweisung..... | 247 |
| 13.5.3 | FETCH- Anweisung | 247 |
| 13.5.4 | CLOSE- Anweisung | 248 |
| 13.5.5 | DEALLOCATE- Anweisung | 248 |
| 14 | Gespeicherte Prozeduren..... | 251 |
| 14.1 | Programmieren gespeicherter Prozeduren | 252 |
| 14.2 | Ausführen von gespeicherten Prozeduren..... | 254 |
| 14.3 | Ändern und Löschen von gespeicherten Prozeduren..... | 255 |
| 14.4 | Eingabeparameter in gespeicherten Prozeduren | 256 |
| 14.4.1 | Erstellen gespeicherter Prozeduren mit Eingabeparametern..... | 256 |
| 14.4.2 | Ausführen gespeicherter Prozeduren mit Eingabeparametern..... | 257 |
| 14.4.3 | Zurückgeben von Werten | 258 |
| 14.5 | Erneutes Kompilieren gespeicherter Prozeduren..... | 259 |
| 14.6 | CLR- Prozeduren | 259 |
| 15 | Benutzerdefinierte Funktionen..... | 261 |
| 15.1 | Arten von benutzerdefinierten Funktionen | 262 |
| 15.1.1 | Skalarfunktionen..... | 262 |
| 15.1.2 | Inlinefunktionen mit Tabellenrückgabe..... | 263 |
| 15.1.3 | Funktion mit mehreren Anweisungen und Tabellenrückgabe | 264 |
| 15.2 | Schemagebundene Funktionen | 265 |
| 15.3 | Leistungsaspekte | 265 |
| 15.4 | Funktionen ändern | 266 |

| | | |
|--------|--|-----|
| 15.5 | Funktionen löschen..... | 266 |
| 15.6 | CLR- Funktionen | 266 |
| 16 | Trigger..... | 269 |
| 16.1 | DML- Trigger erstellen | 269 |
| 16.1.1 | Funktionsweise von DML- Triggern | 271 |
| 16.2 | DDL- Trigger erstellen..... | 274 |
| 16.3 | LOGON- Trigger erstellen | 276 |
| 16.4 | Trigger ändern | 277 |
| 16.5 | Trigger löschen | 277 |

1 Relationale Datenbanksysteme

1.1 Allgemeines

Programme zur Verwaltung externer Datenbestände, die nicht auf einer Datenbank gespeichert sind, besitzen eine zu starre Zuordnung zu bestimmten Anwendungsprogrammen. Jedes Programm definiert seine eigenen Datenstrukturen. Dadurch besteht eine starke Abhängigkeit zwischen Verarbeitungslogik und den physischen Datenstrukturen.

Daraus können sich folgende Nachteile für einen schnellen und sicheren Zugriff auf die Daten ergeben:

1. Datenredundanz:

Daten mit gleichem Informationsgehalt sind häufig mehrfach, in Kombination mit jeweils anderen Daten und/oder unterschiedlichen Organisationsformen gespeichert.

2. Dateninkonsistenz:

Ein und dasselbe Faktum ist widersprüchlich d.h. mit unterschiedlichen Werten an verschiedenen Stellen im Datenbestand gespeichert.

3. Programmpflege:

Änderungen an der logischen und physischen Struktur des Datenbestandes erzwingt eine Änderung alle auf diese Daten zugreifender Programme.

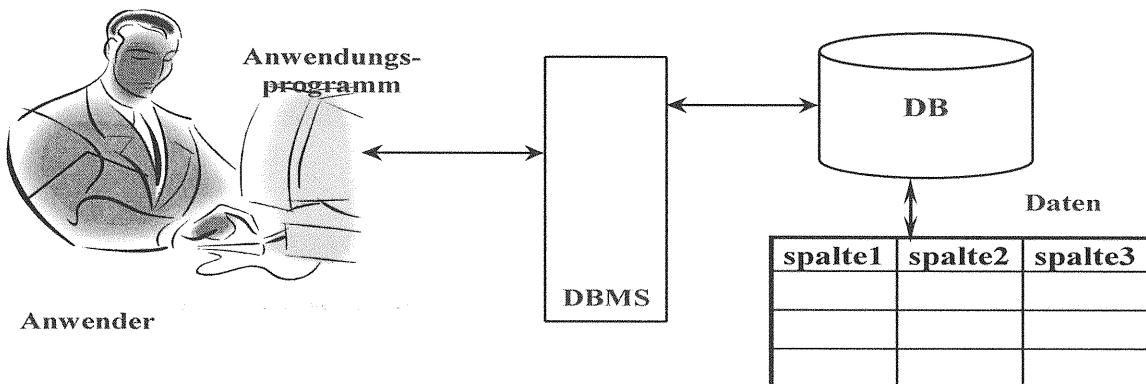
1.2 Datenbanksysteme

Darum werden Datenorganisationsformen benötigt, welche die Anwendung von den gespeicherten Daten trennt. Diese Softwareschicht die zwischen beiden liegt nennt man Datenbankverwaltungssystem (DBMS Database Management System). Es steuert sämtliche Zugriffsanforderungen auf die Daten.

Ein DBMS verwaltet eine oder mehrere Datenbanken (DB) plus zugehörige Hilfs- und Systemdaten.

Mit DBS ist gemeint:

$$\text{DB} + \text{DBMS} = \text{DBS}$$



Das DBMS besteht in der Regel aus einem so genannten Data Dictionary (DD). Hierbei handelt es sich um einen Datenkatalog (Systemkatalog) der sich auf der einen Seite selber verwaltet und auf der anderen Seite die Zugriffsanforderungen auf die Benutzerdaten abwickelt.

Was sollte ein DBMS leisten?

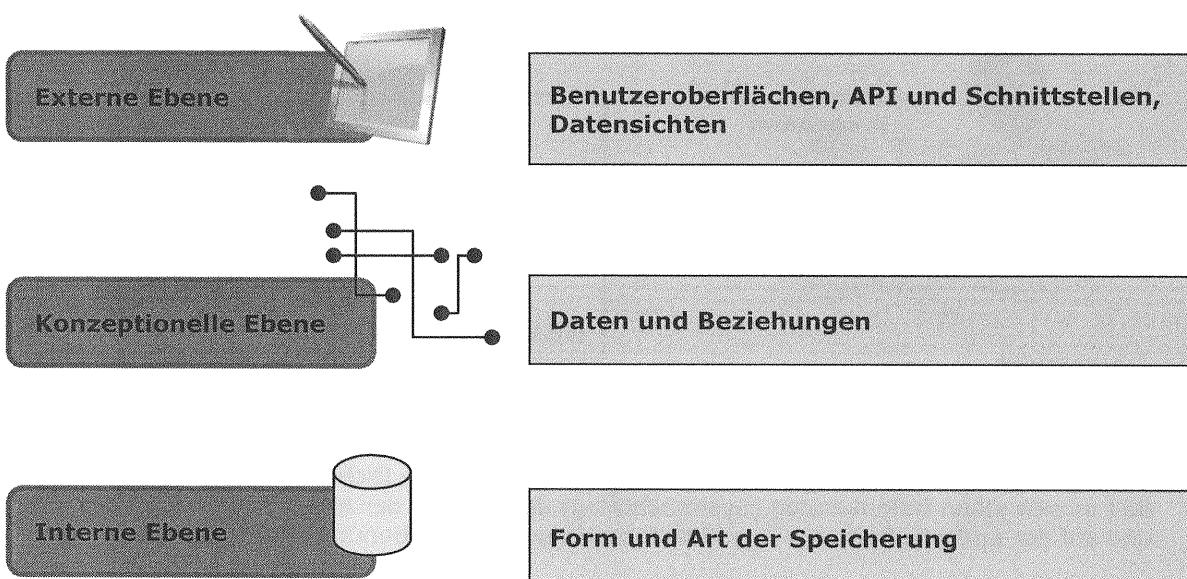
- ♦ zulassen von DDL Anweisungen (Data Definition Language) um Datenbankobjekte zu erstellen.
- ♦ Anforderungen von Benutzern entgegennehmen und entsprechende Aktionen (insert, update, delete, select) als DML Anweisungen (Data Manipulation Language) und DQL Anweisungen (Data Query Language) veranlassen.
- ♦ vor der Anforderung von DDL – und DML Anweisungen muss eine Zugriffskontrolle erfolgen. Die Zugriffskontrolle wird vorher mit entsprechenden DCL Anweisungen (Data Control Language) vereinbart.
- ♦ Sicherstellen von Integritätsbedingungen bei Änderungsoperationen (insert, update und delete).
- ♦ es müssen bei Abfragen die günstigsten (in der Regel die schnellsten) physischen Zugriffspfade ausgewählt werden (Optimierung).
- ♦ gewährleisten von Multiuser – Betrieb, das heißt konkurrierende Benutzeranforderungen so synchronisieren, dass ein optimaler Gesamtdurchsatz an Daten durch TCL Anweisungen (Transactional Control Language) erreicht wird.
- ♦ abgebrochene Änderungsoperationen müssen rückgängig gemacht werden (Roll-back) und um einen konsistenten Zustand der Daten wiederherzustellen. Ebenso müssen Nichtabgeschlossene Transaktionen nach einem Systemabsturz wieder zurückgesetzt werden (Recovery).

1.3 Ebenen in relationalen Datenbanksystemen

Die ANSI- SPARC- Architektur, auch Drei- Ebenen- Architektur genannt, beschreibt den grundlegenden Aufbau eines relationalen Datenbanksystems.

Dabei wird von drei Ebenen gesprochen:

1. Die **konzeptionelle Ebene** (fachliche Sicht), welche auf Basis des semantischen Datenmodells die Sachlogik formal beschreibt.
2. Die **interne Ebene** (technische Sicht), die formal darstellt, **wie** und **wo** die Daten in der Datenbank gespeichert werden.
3. Die **externe Ebene** (individuelle anwendungsorientierte Sicht), die formal beschreibt, wie sich die Datenbank den Benutzern und Anwendungen darstellt.



Was bedeutet DDL, DML, TCL, DQL und DCL?

In diese Hauptgruppen wird die Datenbankabfragesprache SQL (Structure Query Language) eingeteilt:

DDL Datendefinitionssprache (Data Definition Language)

Basiert auf den Befehlen zur Definition von verschiedenen Objekten innerhalb der Datenbank. Dazu zählen alle Anweisungen, mit denen die logischen Strukturen der Datenbank bzw. der Datenbankobjekte erstellt oder verändert werden.

Hierzu gehören folgende Befehle:

| | |
|------------------------|---|
| CREATE objekt_name ... | erstellt ein neues Datenbankobjekt. |
| ALTER objekt_name ... | ändert ein bestehendes Datenbankobjekt. |
| DROP objekt_name ... | löscht ein bestehendes Datenbankobjekt. |

DML Datenmanipulationssprache (Data Manipulation Language)

Besteht aus einer Reihe von Befehlen, die bestimmen, welche Werte innerhalb der einzelnen Datensätze vorkommen. Dazu zählen alle Anweisungen an das Datenbanksystem, die dazu dienen, die Daten zu ändern.

Dazu gehören folgende Befehle:

| | |
|------------|-----------------------------------|
| DELETE ... | Zeilen einer Tabelle löschen. |
| UPDATE ... | Zeilen einer Tabelle ändern. |
| INSERT ... | Zeilen in einer Tabelle einfügen. |

DCL Datenkontrollsprache (Data Control Language)

besteht aus Anweisungen, die sich auf Rechte für Objekte innerhalb der Datenbank beziehen.

Folgende Anweisungen stehen zur Verfügung:

| | |
|------------|-------------------------------|
| GRANT ... | eine Berechtigung erteilt. |
| REVOKE ... | eine Berechtigung entfernt. |
| DENY ... | eine Berechtigung verweigert. |

TCL Transaktionssteuersprache (Transactional Control Language)

besteht aus Anweisungen, die unterschiedliche Transaktionen verwalten.

Folgende Anweisungen stehen zur Verfügung:

| | |
|--------------------------|--------------------------------|
| COMMIT TRANSACTION... | beendet eine Transaktion. |
| BEGIN TRANSACTION... | beginnt eine Transaktion. |
| ROLLBACK TRANSACTION ... | rollt eine Transaktion zurück. |

DQL Datenabfragesprache (Data Query Language)

besteht aus Anweisungen, die Daten aus einer Tabelle abfragen.

Folgende Anweisung steht zur Verfügung:

| | |
|------------|--------------------------------|
| SELECT ... | Daten aus der Datenbank lesen. |
|------------|--------------------------------|

1.4 Werkzeuge der DBS

- ❖ Abfrageschnittstellen, grafische (SSMS, Query Analyzer,...) oder nicht grafisch (SQLCMD, OSQL, SQL Plus,...), für ADHOC- Abfragen.
- ❖ Reportgeneratoren für die einfache Erstellung von Berichten und Listen (bei SQL Server wird das Programm Reporting Service, Access oder WinWord eingesetzt).
- ❖ Schnittstellen zu Spreadsheet-Programmen (Kalkulationstabellen-Programmen). Zum Beispiel OLEDB oder ODBC zu Übernahme von Datenbankdaten in Excel.
- ❖ Präsentationsgrafiktools zur Auswertung betrieblicher Daten. Für SQL Server ist dafür Excel mit Power Pivot oder PowerPoint in Zusammenarbeit mit Analysis-Service (Data Warehouse) vorgesehen.
- ❖ 4GL Tools (Programmiersprachen der 4 Generation) wie Visual Basic.NET, C#, Delphi, Ingress 4GL, Open Road und weitere
- ❖ Datenbankdesign Tools (MS Visio, Erwin, ER/Studio, ...), Werkzeuge für die Planung und Entwicklung eines Datenmodells für die jeweilige reale Welt. Mit diesen Programmen lässt sich solch ein Modell dann in das Datenbanksystem implementieren.

1.5 Was ist ein relationales Datenbankmodell?

Das relationale DB- Model wurde in den Jahren 1968 bis 1972 von E. F. Codd entwickelt und eingeführt. Es ist das mittlerweile am weitesten verbreitete Datenbankmodell.

Es beschreibt einen Formalismus der Repräsentation von Objekten der realen Welt (also das betreffende Unternehmen usw.) und deren Manipulation in einem Datenbanksystem. Es zeichnet sich durch seine einfache Grundstruktur aus. Ziel ist es, die Datenunabhängigkeit zu gewährleisten.

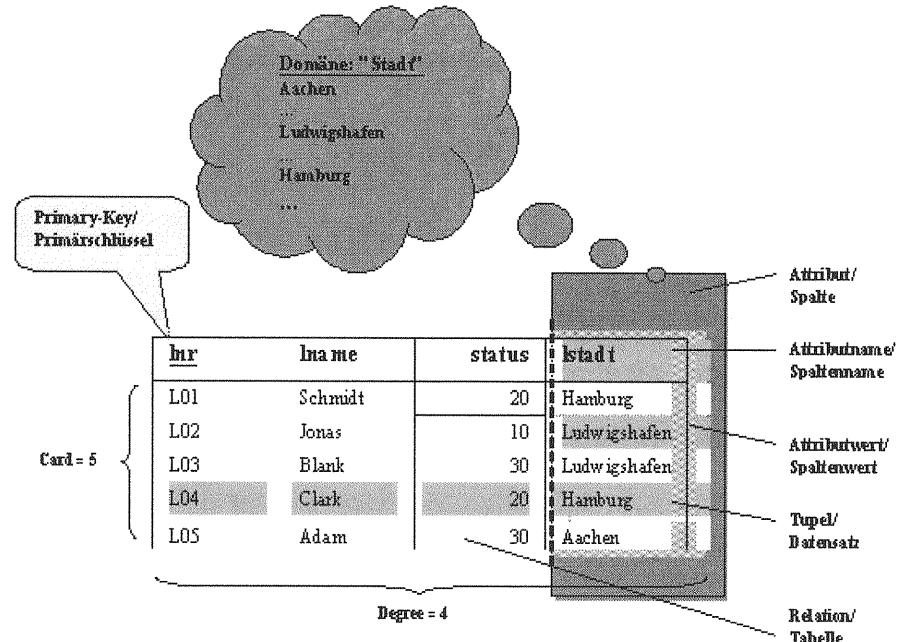
Für das relationale Datenbankmodell sprechen im Besonderen zwei Gründe:

- ❖ **Einfachheit:** Alle Information einer relationalen Datenbank wird einheitlich durch Werte repräsentiert, die mit Hilfe einer Relation (Tabelle) strukturiert sind.
- ❖ **Systematik:** Mathematische Grundlage des Modells ist die Mengentheorie.

Die Grundstruktur des Modells besteht aus drei Komponenten:

- ❖ **Datenobjekttypen:** Basiseinheiten (Tabellen, Sichten, Primärschlüssel, ...) zur Darstellung der logischen Struktur jeder Datenbank.
- ❖ **Integritätsregeln:** Bedingungen die alle auftretenden Objekte erfüllen müssen.
- ❖ **Operatoren:** werden auf Objekte des Datenmodells mit vorhersagbaren Ergebnissen angewandt.

2 Datenstrukturen



- **Card (Cardinalität)**: kennzeichnet die vertikale Ausdehnung einer Relation also die Anzahl der Datensätze in einer Tabelle. Der Wertebereich geht von 0 bis ∞ .
- **Degree**: beschreibt die horizontale Ausdehnung einer Relation also die Anzahl der Spalten einer Tabelle. Der Wertebereich geht von 1 bis ∞ .

2.1 Skalare Werte und Basisdatentypen

Der skalare Wert (atomarer Wert) ist die kleinste semantische Dateneinheit relationaler Systeme. Das heißt, sie können nicht ohne Bedeutungsverlust in weitere (kleinere) Teilwerte zerlegt werden.

Jeder skalare Wert gehört einem Basisdatentyp (integer, char, ...) an.

Beispiel:

Spalte: "Iname"
 skalarer Wert = Schmidt
 Basisdatentyp = varchar(25)

2.2 Domänen

Domänen geben den Wertebereich einer Spalte (Attribut) einer Tabelle (Relation) vor.

Definition: Eine Domäne ist eine Menge skalarer Werte eines Basisdatentyps, die als aktuelle Werte eines oder mehrerer Attribute (Spalten) zulässig sind.

Eine Spalte einer Tabelle welche einer Domäne unterliegt, kann also nur solche Werte aufnehmen die durch die Domäne vorgegeben sind.

Domänen sind Relationen (Tabellen) übergreifend, also können mehrere Spalten einer Domäne unterliegen.

Beispiel:

1. Erstellen einer Domäne:

```
create domain Inummern char(5) check Inummer between 'L01' and 'L99'
```

...

2. Anwenden einer Domäne beim Erstellen einer Tabelle:

```
create table lieferant
  (Inr domain(Innummer) not null,
  ...)
```

Hinweis: Obwohl Domänen Bestandteil des Relationsmodells sind, ist das Domänenkonzept zurzeit in wenigen kommerziell verfügbaren DBMS implementiert. Attribute werden bis jetzt statt auf Domänen direkt auf Basisdatentypen definiert und anschließend mit einer Wertebereichseinschränkung versehen.

Eine **Ausnahme** bildet dabei das Datenbanksystem **INTERBASE** der Firma Borland. Hier wurde das Domänenkonzept durchgesetzt wie im nachfolgenden Beispiel zu sehen ist.

Syntax:

```
create domain domainname as basisdatentyp
  [default {literal | NULL | user}]
  [NOT NULL] [check {domainen_einschränkung}]
  [collate collation]
```

Beispiel:

```
create domain orte as varchar(30) default 'Erfurt'
  check (value in ('Erfurt', 'Jena', 'Weimar', 'Gotha', 'Suhl'))
```

Vorzüge von Domänen:

- einem Attribut können nur solche Werte zugewiesen werden, die Element der dem Attribut zugeordneten Domäne sind. Alle Anderen Transaktionen werden vom DBMS abgewiesen.
- automatische Erkennung von Fehlern in Abfragen in deren Where Klausel Attribute mit unterschiedlichen Domänen verglichen werden.

Beispiel:

```
select *
  from artikel
  where gewicht = amenge
```

Die oben stehende Abfrage ist in den gängigen DBS ohne Fehler durchführbar. Dabei macht der Vergleich zwischen der Lagermenge eines Artikels und seinem Gewicht überhaupt keinen Sinn.

Wäre das Domänenkonzept zur Anwendung gekommen und definiert worden, dass die Spalte "amenge" einer Domäne "mengen" und die Spalte "gewicht" einer Domäne "gewichte" zugeordnet sind, dann würde das DBMS bei obenstehender Abfrage eine Fehlermeldung anzeigen.

2.3 Relationen (Tabellen)

Die Relation (Begriff entstammt der Mengenlehre der Mathematik) bzw. die Tabelle ist der einzige zulässige strukturierte Datentyp des Relationenmodell oder in einer Datenbank.

Merkmale und Eigenschaften:

- besitzen einen innerhalb der Datenbank eindeutigen Namen.
- die Spaltenüberschriften (Attributnamen) innerhalb einer Relation sind eindeutig.
- eine Relation besitzt 0 bis n Datensätze (Tupel).
- eine Relation besitzt 1 bis n Spalten (Attribute).
- die Reihenfolge der Datensätze ist nicht definiert.
- die existiert keine festgelegte Reihenfolge der Spalten (in der Praxis existiert eine Reihenfolge, sie wird durch die Reihenfolge der Spaltennamen bei der Erstellung der Tabelle festgelegt).
- zu keiner Zeit existieren in einer Relation zwei gleiche Datensätze.
- jede Relation besitzt eine Spalte oder eine Spaltenkombination zur genauen Identifizierung der Datensätze, einen so genannten Primärschlüssel.
- in einer Tabelle existieren keine Wiederholungsgruppen.

2.3.1 Relationsarten

1. Basisrelationen: (Basis- oder Benutzertabellen) besitzen einen innerhalb der Datenbank eindeutigen Namen und existieren unabhängig von anderen Relationen. Sie werden auf interner Ebene (auf dem Speichermedium) durch Daten repräsentiert.
2. Views (Sichten): besitzen einen in der Datenbank eindeutigen Namen. Sie werden auf interner Ebene nicht durch Daten repräsentiert sondern nur ihre DML- Definition ist auf dem Medium abgelegt. Diese DML Anweisung ist eine Abfrage auf Basistabellen bzw. auf andere Views. Daher ist ihre Existenz von der Existenz dieser Relationen abhängig.
3. Ergebnisrelationen: sind in der Regel unbenannte Relationen die einen momentanen Zustand des Datenbestandes als Resultat von Datenbankabfragen liefern. Sie enthalten Daten von Basis – oder View- Relationen die bestimmte Abfragekriterien erfüllen.
4. Temporäre Relationen: entstehen durch DDL- bzw. DCL- Anweisungen. Sie sind in der Regel benannt und werden zu einem bestimmten Zeitpunkt automatisch wieder aus der Datenbasis entfernt.
5. Snapshots: sind benannte Relationen. Sie bestehen aus gespeicherten Abfragen die in regelmäßigen Abständen automatisch wiederholt werden.

2.3.2 Fundamentalprinzip relationaler Datenbanken

Eine relationale Datenbank repräsentiert sich als eine Menge von Relationen welche insgesamt eine Gesamtaussage einer realen Welt widerspiegeln. Das wird durch logische Beziehungen zwischen den Relationen erreicht. Diese logische Verknüpfung der Relationen erfolgt ausnahmslos durch den Wertevergleich von Attributen der betreffenden Relationen.

Weder in noch zwischen den Relationen existieren irgendwelche Zeiger, die den Benutzern zugänglich wären.

3 Integritätsbedingungen

Integritätsbedingungen legen fest welche Daten, wo, unter welchen Umständen und wann zulässig sein sollen. Sie sollen die ständige Datenintegrität bei Änderungsoperationen (insert, update, delete) gewährleisten.

3.1 Primärschlüssel

Definition: Der Primärschlüssel ist ein Attribut oder eine Attributkombination einer Relation, dessen Werte- bzw. Wertekombinationen garantiert eindeutig sind in Bezug auf die Tupel dieser Relation.

Über einen Primärschlüssel soll es möglich sein einen einzelnen Datensatz eindeutig anzusprechen.

Als Primärschlüssel kommt jedes Attribut bzw. Attributs Kombination in Betracht, das zwei Bedingungen erfüllt:

1. Eindeutigkeit: Zu keinen Zeitpunkt existieren mehrere gleiche Werte bzw. Wertekombinationen in dem betreffenden Attribut bzw. der betreffenden Attributs Kombination.
2. Minimalität: Handelt es sich um eine Attributs Kombination, so muss diese minimal in dem Sinne sein, dass kein Attribut aus der Kombination entfernt werden kann ohne Verlust der Eindeutigkeit.

Primärschlüssel die aus einer Spalte gebildet werden nennt man „einfacher Primärschlüssel“ und Primärschlüssel aus mehreren Spalten „zusammengesetzter Primärschlüssel“.

3.2 Null- Werte

Eine Spalte, bei der NULL zulässig ist, kann erforderlich sein, wenn benötigte Daten noch nicht zur Verfügung stehen, beispielsweise der zweite Vorname eines Lieferanten. Der Anwender könnte nun dafür ein Leerzeichen (Alphanumerische Werte) oder eine Null (Numerische Werte) eintragen.

Aber diese Vorgehensweise kann Abfragen auf die Datensätze erschweren oder Ergebnisse verfälschen (statt einem sind mehrere Lehrzeichen eingetragen oder Durchschnittsberechnungen mit Nullen).

Ein relationales Datenbanksystem sieht darum für solche Fälle so genannte NULL- Werte vor.

Definition: Ein Null- Wert (Nullmarke) ist ein Bitmuster das vom DBS als unbekannter Wert interpretiert wird. Eine Nullmarke ist keine Numerische Null und kein Alphanumerisches Leerzeichen. Eine Nullmarke besitzt keinen Datentyp und kann damit auf jede Spalte (vorausgesetzt es ist zulässig) angewendet werden. Das Bitmuster kann nicht verändert werden.

Allgemein gilt, dass NULL- Werte zu vermeiden sind. Abfragen und Aktualisierungen werden durch NULL- Werte komplexer, und einige Optionen, beispielsweise Primärschlüssel und die Eigenschaft IDENTITY (AutoWert) können in einer Spalte die NULL- Werte zulässt, nicht verwendet werden.

Hinweis: Es wird empfohlen, die Verwendung von Null-Werten weitgehend einzuschränken, um den Wartungsaufwand und mögliche Auswirkungen auf vorhandene Berichte oder Abfragen so gering wie möglich zu halten. Planen Sie die Abfragen und Datenänderungen so, dass Null-Werte minimale Auswirkungen haben.

3.3 Primärschlüsselintegrität

Ein Primärschlüssel darf keine NULL Marken enthalten. Das bedeutet das auch keine Komponente eines zusammengesetzten Primärschlüssels NULL Marken zulassen darf.

3.4 Fremdschlüssel

Definition: Ein Fremdschlüssel ist ein Attribut bzw. Attributkombination einer Relation, dessen Nichtnullwerte Primärschlüsselwerte einer anderen oder derselben Relation sind.

3.5 Referentielle Integrität

Aus der Definition eines Fremdschlüssels ergibt sich:

Eine relationale Datenbank darf nur referenzierende Fremdschlüsselwerte enthalten.

Praktisch bedeutet das, wenn in eine Fremdschlüsselpalte ein Wert eingetragen wird überprüft das DBMS automatisch ob es diesen Wert auch in der referenzierten Primärschlüsselpalte gibt. Wenn nicht, dann wird die Änderungsoperation (insert, update) abgewiesen.

3.6 Operationsregeln für Fremdschlüssel

Operationsregel für Fremdschlüssel weisen das DBMS an, bestimmte Regel durchzusetzen wenn an den entsprechenden Primärschlüsselwerten Änderungen (update, delete) durchgeführt werden. Das heißt, was soll mit den abhängigen Fremdschlüsselwerten geschehen?

3.6.1 NULL – Regel

Die Null – Regel ist eine Entscheidung ob für die Fremdschlüsselpalten der Tabellen NULL – Marken in Frage kommen oder nicht. Diese Auswahl wird entsprechend den Gegebenheiten der realen Welt getroffen.

3.6.2 Löschregel

Wie soll das DBMS auf die Anweisung reagieren, einen Datensatz zu löschen, dessen Primärschlüsselwert von einem Fremdschlüsselwert referenziert wird.

1. bedingtes Löschen:

Die Löschoperation wird nur ausgeführt, wenn keine dem zu löschenen Primärschlüsselwert entsprechende Fremdschlüsselwerte existieren.

2. kaskadierendes Löschen:

Die Löschoperation wird ausgeführt und alle Datensätze deren Fremdschlüsselwert dem zu löschenen Primärschlüsselwert entsprechen, werden mit gelöscht.

3. Löschen mit NULL-Setzung:

Die Löschoperation wird ausgeführt. Alle Fremdschlüsselwerte die dem zu löschenen Primärschlüsselwert entsprechen, werden auf NULL gesetzt (wenn der Fremdschlüssel NULL erlaubt).

3.6.3 Änderungsregeln

Wie soll das DBMS auf die Anweisung reagieren, einen Primärschlüsselwert zu ändern, wenn er von einem Fremdschlüsselwert referenziert wird.

1. bedingtes Ändern:

Die Änderungsoperation wird nur ausgeführt, wenn keine dem zu ändernden Primärschlüsselwert entsprechende Fremdschlüsselwerte existieren.

2. kaskadierendes Ändern:

Die Änderungsoperation wird ausgeführt und alle Fremdschlüsselwerte die dem zu ändernden Primärschlüsselwert entsprechen, werden mit geändert.

3. Ändern mit NULL-Setzung:

Die Änderungsoperation wird ausgeführt. Alle Fremdschlüsselwerte die dem zu ändernden Primärschlüsselwert entsprechen, werden auf NULL gesetzt (wenn der Fremdschlüssel NULL erlaubt).

Diese eben genannten Integritätsregeln werden beim Erstellen der Tabellen festgelegt (kann auch nachträglich passieren) und werden von diesem Augenblick an, automatisch vom DBMS überwacht und realisiert.

3.7 Beispieldatenbank "Standard"

Alle nachfolgenden Erläuterungen und Beispiele werden mit den Tabellen und Daten der Datenbank "Standard" illustriert.

Sie beschreibt eine reale Welt, in der Lieferanten, einer Firma, Artikel liefern und Artikel von Lieferanten, dieser Firma, geliefert werden.

Zu jeder Entität sind die wichtigsten Merkmale aufgeführt und jeder Datensatz wird durch einen eindeutigen Wert (Primärschlüssel) repräsentiert.

Eine Geschäftsregel des Unternehmens lautet, jeder Lieferant liefert ein und denselben Artikel nur maximal einmal am Tag.

Beispiel:

Tabellen der Datenbank "Standard" (Syntax SQL Server) erstellen

```
CREATE TABLE artikel
  (anr nummern NOT NULL constraint anr_ps primary key,
  aname namen NOT NULL ,
  farbe char (7) NULL,
  gewicht decimal(9,2) NULL,
  astadt staedte NULL,
  amenge mengen NULL)
```

go

```
CREATE TABLE lieferant
  (Inr nummern NOT NULL constraint Inr_ps primary key,
  Iname namen NOT NULL,
  status int NULL,
  istadt staedte NULL)
```

go

```
CREATE TABLE lieferung
  (lnr nummern NOT NULL constraint lnr_fs references lieferant(lnr)
   on update cascade
   on delete no action,
  anr nummern NOT NULL constraint anr_fs references artikel(anr)
   on update cascade
   on delete no action,
  lmenge mengen NOT NULL,
  ldatum datetime NOT NULL,
  constraint lief_ps primary key(lnr, anr, ldatum) )
```

```
go
```

4 Methoden zum Datenbankentwurf

Eine komplexe Aufgabe ist das Entwerfen einer relationalen Datenbank. Sämtliche Informationen der realen Welt müssen gesammelt und in Beziehung zueinander gebracht werden. Dazu werden die Informationen in Tabellen zusammengefasst, Integritätsbedingungen und die Beziehungen zwischen den Tabellen festgelegt sowie die unterschiedlichen Benutzerschichten realisiert.

Nachfolgende Schritte sind für einen korrekten und übersichtlichen Datenbankentwurf notwendig:

Normalisierungsprozess

Ist eine Methode um eine redundanzfreie Speicherung der Daten zu realisieren. Dadurch werden Änderungsanomalien an den Daten vermieden.
Die Daten werden über mehrere Stufen, die sogenannten Normalformen, auf Tabellen verteilt bis möglichst keine Redundanzen und Inkonsistenzen mehr vorhanden sind. Es wird das relationale Datenbankdesign erreicht.

Entity- Relationship- Modell

Mit Hilfe des ER- Models werden die Beziehungen zwischen den Tabellen dargestellt.

Integritätsregeln

Das sind Bestimmungen, welche den logischen Aufbau und die Korrektheit der Daten garantieren. Diese werden dann vom DBMS bei DML-Anweisungen auf die Daten angewendet.

4.1 ER- Modell

Das Entity- Relationship- Modell dient dazu, im Rahmen der Datenmodellierung einen Ausschnitt der realen Welt zu beschreiben. Es besteht aus einer Grafik und einer Beschreibung der darin verwendeten Elemente, wobei Dateninhalte und Datenstrukturen dargestellt werden. Es dient als Grundlage für das Design relationaler Datenbanken.

Grundlage der ER- Modelle ist die Typisierung von Objekten und deren Beziehungen untereinander.

Begriffe:

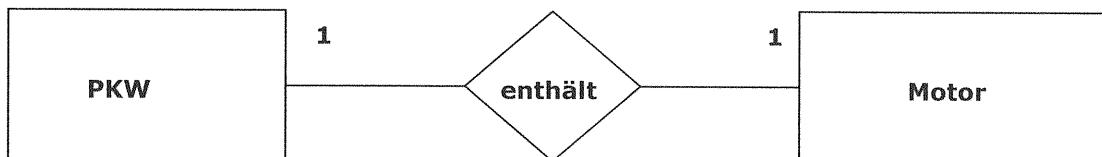
- Entität:** Objekt der Wirklichkeit, materiell oder abstrakt (Beispiel: **Mitarbeiter** „Schulze“, **Abteilung** „3“).
- Entitätstyp:** Sammlung von Entitäten mit gleichen Eigenschaften (Beispiel: alle Mitarbeiter, alle Abteilungen)
- Beziehung:** Verknüpfung zwischen zwei oder mehreren Entitäten (Beispiel: „Mitarbeiter Schulze arbeitet in Abteilung 3“)
- Beziehungstyp:** Beschreibt den numerischen Zusammenhang zwischen den einzelnen Elementen einer Beziehung. Dabei steht 0 für keine Zuordnung, 1 für genau eine Zuordnung und n bzw. m für viele Zuordnungen (Beispiel: 1 Abteilung besteht aus n Mitarbeitern – n Mitarbeiter arbeiten in 1 Abteilung).
- Attribute:** Sind die verschiedenen Felder einer Entität (Beispiel: Vorname, Nachname und Geburtstag eines Mitarbeiters) die einem bestimmten Wertebereich (Domäne) unterliegen. Ein Attribut das die Entität eindeutig beschreibt wird Schlüssel genannt,

4.1.1 Die 1:1 Beziehung

Zwischen zwei Entitäten besteht eine 1:1 Beziehung, wenn folgendes gilt:

Zu jeder Entität A gibt es genau eine Entität B und umgekehrt.

Im Beispiel wird dargestellt, dass ein PKW einen Motor besitzt und ein Motor einem PKW zugeordnet ist.

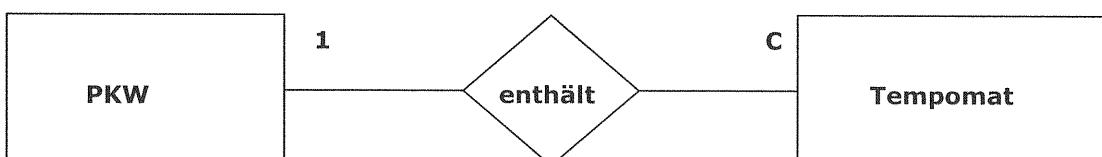


4.1.2 Die 1:C Beziehung

Es gibt auch Entitäten, zwischen denen eine Beziehung stehen **kann**. Für diesen Fall führt man die konditionale Beziehung ein.

Zu jeder Entität A kann es eine oder keine Entität B geben. Umgekehrt gilt, für jede Entität B gibt es genau eine Entität A.

Im Beispiel wird dargestellt, dass ein PKW einen oder keinen Tempomat besitzt aber jeder Tempomat einem PKW zugeordnet ist.



4.1.3 Die 1:N Beziehung

Zwischen zwei Entitäten besteht eine 1:n Beziehung, wenn folgendes gilt:

Zu einer Entität A gibt es eine oder mehrere Entitäten B. Umgekehrt gibt es zu jeder Entität B genau eine Entität A.

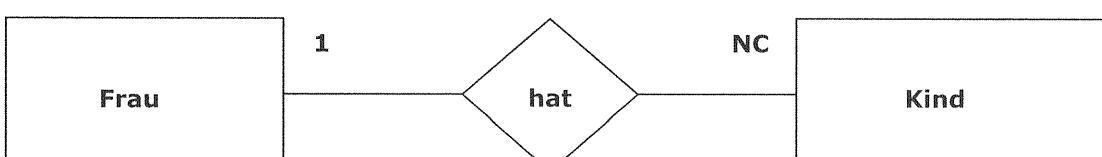
Im Beispiel wird dargestellt, dass in einer Abteilung ein oder mehrere Mitarbeiter arbeiten aber jeder Mitarbeiter nur genau einer Abteilung zugeordnet ist.



4.1.4 Die 1:NC Beziehung

Analog der 1:C Beziehung, ist es darüber hinaus auch zulässig, dass einer Entität A eine, mehrere oder keine Entsprechung in Entität B zugeordnet ist. Umgekehrt gibt es zu jeder Entität B genau eine Entität A.

Im Beispiel wird dargestellt, dass eine Frau ein, mehrere oder kein Kind haben kann aber jedes Kind genau einer Frau zugeordnet ist.



4.1.5 Die M:N Beziehung

Eine M:N Beziehung liegt vor, wenn folgendes gilt:

Zu jeder Entität A gibt es ein oder mehrere Entitäten B und umgekehrt.

Im Beispiel wird dargestellt, dass ein Student eine oder mehrere Prüfungen absolvieren muss und umgekehrt muss eine Prüfung von einem oder mehreren Studenten absolviert werden.



4.1.6 Die N:MC Beziehung oder NC:MC Beziehung

Es kann auch möglich sein, dass einer Entitätsmenge aus der N:M Beziehung kein, ein oder mehrere Entitäten aus einer anderen Entitätsmenge zugeordnet ist.

Im Beispiel wird dargestellt, dass ein Student kein, ein oder mehrere Kurse belegen kann und umgekehrt wird ein Kurs von keinem, einen oder mehreren Studenten belegt.



Im Relationenmodell ist eine M:N Beziehung nicht darstellbar. Sie muss deshalb in 1:N Beziehungen aufgelöst werden. Die Beziehung (in unserem Fall "belegt") wird eine eigene Tabelle (Verbindungs- oder Beziehungstabelle).

Die Tabelle, die die Verbindungsentität darstellt, nimmt den Primärschlüssel der Beteiligten Entitäten als Fremdschlüssel auf. Die beiden Schlüssel ergeben zusammen ergeben den Primärschlüssel in der neuen Tabelle.

Zusammenfassung:

Das Entity- Relationship- Modell wird verwendet um einen Ausschnitt der realen Welt zu beschreiben. Es dient der Verständigung zwischen Anwendern und Entwicklern in der konzeptionellen Phase der Anwendungsentwicklung, und als Grundlage für das Design in der Implementierungsphase

4.2 Normalisierung

In einigen Fällen ist es aber auch sinnvoll, auf eine Normalisierung zu verzichten oder Teile der normalisierten Daten durch Denormalisierung (Views erstellen) rückgängig zu machen, um zum Beispiel die Verarbeitungsgeschwindigkeit zu erhöhen oder Anfragen zu vereinfachen und damit die Fehleranfälligkeit zu verringern.

4.2.1 Erste Normalform

Definition: Eine Relation befindet sich in der ersten Normalform, wenn alle Attribute atomar sind und die Relation frei von Wiederholungsgruppen ist.

4.2.1.1 Was bedeutet atomar?

Gegeben ist eine Tabelle in der Kundeninformationen gespeichert sind.

kunden

| kdnr | name | wohnort |
|------|--------------|----------------------------------|
| 1000 | Otto Büchner | Weimarer Straße 34, 99086 Erfurt |
| 1001 | Maria Krause | Zitronenweg 6, 99189 Urbich |
| 1003 | Susi Drosten | Hinterm Wald 12, 99089 Erfurt |

Jetzt überführen wir diese Tabelle in die erste Normalform

kunden

| kdnr | vname | name | strasse | hnr | plz | ort |
|------|-------|---------|-----------------|-----|-------|--------|
| 1000 | Otto | Büchner | Weimarer Straße | 34 | 99086 | Erfurt |
| 1001 | Maria | Krause | Zitronenweg | 6 | 99189 | Urbich |
| 1003 | Susi | Drosten | Hinterm Wald | 12 | 99089 | Erfurt |

Es ist Ansichtssache ob es Zweckmäßig ist, die Hausnummer als alleinstehendes Attribut zu verwenden. Das sollte man von Fall zu Fall entsprechend den Fakten der realen Welt entscheiden. Im Normalfall wird die Hausnummer mit dem Straßennamen abgespeichert, weil der reale Mensch nur selten in eine andere Hausnummer derselben Straße umzieht, sondern eher in eine andere Straße ziehen wird.

4.2.1.2 Was bedeutet keine Wiederholungsgruppen?

Gegeben ist die oben stehende Kundentabelle in der wir auch die Telefonnummern der Kunden speichern wollen.

kunden

| kdnr | vname | name | ... | telefon1 | telefon2 | telefon3 |
|------|-------|---------|-----|----------------|------------------|----------------|
| 1000 | Otto | Büchner | ... | 0361 435363 | 0172 98707897 | null |
| 1001 | Maria | Krause | ... | 0361 757585 | 0170 12567892 | 0361 757586 |
| 1003 | Susi | Drosten | ... | 0361 234564 | null | null |

Diese Tabelle besitzt Wiederholungsgruppen in den Attributen "telefon", in denen die Gruppen Vorwahl und Telefonnummer stecken.

Bringen wir diese Tabelle in die erste Normalform benötigen wir eine zusätzliche Tabelle.

kunden

| kdnr | vname | name | strasse | hnr | plz | ort |
|------|-------|---------|--------------------|-----|-------|--------|
| 1000 | Otto | Büchner | Weimarische Straße | 34 | 99086 | Erfurt |
| 1001 | Maria | Krause | Zitronenweg | 6 | 99189 | Urbich |
| 1003 | Susi | Drosten | Hinterm Wald | 12 | 99089 | Erfurt |

telefon

| vorwahl | nummer | kdnr |
|---------|----------|------|
| 0361 | 435363 | 1000 |
| 0172 | 98707897 | 1000 |
| 0361 | 757585 | 1001 |
| 0170 | 12567892 | 1001 |
| 0361 | 757586 | 1001 |
| 0361 | 234564 | 1003 |

4.2.2 Zweite Normalform

Definition: Eine Relation befindet sich in der zweiten Normalform, wenn sie sich bereits in der ersten Normalform befindet und jedes Nichtschlüssel Feld funktional vom Primärschlüssel abhängt.

Wenn das oben stehende Beispiel um die Bestellungen der Kunden erweitert wird entsteht eine neue Tabelle "Bestellungen".

bestellungen

| bestnr | kdnr | artname | menge | datum |
|--------|------|---------|-------|------------|
| 1 | 1000 | DVD | 20 | 01.04.2007 |
| 2 | 1000 | CD | 50 | 01.04.2007 |
| 3 | 1003 | DVD | 50 | 04.05.2007 |
| 4 | 1001 | Monitor | 1 | 13.04.2007 |
| 5 | 1000 | Monitor | 2 | 05.04.2007 |

Da es zu erwarten ist, dass es noch mehrere Bestellungen von Kunden geben wird, kann es bei dieser Tabelle zu Problemen mit dem Artikelnamen kommen. Er könnte im Laufe der Zeit öfters falsch geschrieben werden. Dadurch wird das Suchen von Informationen und auch die Korrektur eines falsch geschriebenen Artikelnamen nicht ganz einfach.

Bringen wir die Tabelle in die zweite Normalform.

bestellungen

| bestnr | kdnr | artnr | menge | datum |
|--------|------|-------|-------|------------|
| 1 | 1000 | 100 | 20 | 01.04.2007 |
| 2 | 1000 | 101 | 50 | 01.04.2007 |
| 3 | 1003 | 100 | 50 | 04.05.2007 |
| 4 | 1001 | 102 | 1 | 13.04.2007 |
| 5 | 1000 | 102 | 2 | 05.04.2007 |

artikel

| artnr | artname |
|-------|---------|
| 100 | DVD |
| 101 | CD |
| 102 | Monitor |

Ich bin jetzt auch in der Lage die Tabelle der Artikel um weitere Informationen zu erweitern ohne Datenredundanzen zu erzeugen.

4.2.3 Dritte Normalform

In der Kundentabelle (oben) fällt auf dass die Spalten "plz" und "ort" unmittelbar voneinander abhängen. Man spricht von transitiver Abhängigkeit.

Definition: Eine Relation befindet sich in der dritten Normalform, wenn sie sich bereits in der zweiten Normalform befindet und kein Nichtschlüsselwert von einem anderen Nichtschlüsselwert abhängig ist.

Bringen wird die Tabelle "**kunden**" in die dritte Normalform.

kunden

| kdnr | vname | name | strasse | hnr | plz |
|------|-------|---------|--------------------|-----|-------|
| 1000 | Otto | Büchner | Weimarische Straße | 34 | 99086 |
| 1001 | Maria | Krause | Zitronenweg | 6 | 99189 |
| 1003 | Susi | Drosten | Hinterm Wald | 12 | 99089 |

orte

| plz | ort |
|-------|--------|
| 99086 | Erfurt |
| 99189 | Urbich |
| 99089 | Erfurt |

Wenn eine Datenbank von Anfang an bis zur dritten Stufe normalisiert wird, werden spätere Änderungen und Ergänzungen wesentlich leichter fallen.

Die relationale Theorie kennt noch eine 4. und 5. Normalform. Diese dienen zur Beseitigung von Redundanzen innerhalb eines zusammengesetzten Schlüssels.

4.3 Zusammenfassung

Objekte der realen Welt werden im relationalen Modell durch Tabellen dargestellt. Jede Tabelle setzt sich aus Zeilen und Spalten zusammen. Jede einzelne Zeile einer Tabelle, die auch als Datensatz (Tupel) bezeichnet wird, setzt sich aus Spalten (Attributen) zusammen. Jede Spalte setzt sich aus einem bestimmten Spaltennamen (Attributnamen) und einem Spaltenwert (Attributwert) zusammen. Sie repräsentieren bestimmte Merkmale des entsprechenden Objektes der realen Welt.

Zusätzlich werden sogenannte Schlüsselattribute vergeben, um zum einen die Zuordnung zwischen Objekten (Tabellen) darzustellen und zum anderen den Zugriff auf einen Datensatz eindeutig zu definieren.

Integrität oder Konsistenz ist die Widerspruchsfreiheit von Datenbeständen. Eine Datenbank ist integer oder konsistent, falls die gespeicherten Daten fehlerfrei erfasst sind und den gewünschten Informationsgehalt korrekt wiedergeben

Ein Relationsschema sollte einen Entity-Typ oder einen Beziehungstyp beschreiben. Das heißt eine Relation sollte Informationen beinhalten, die tatsächlich auch logisch zusammengehören. Wird dies nicht berücksichtigt, kann es zu Update- oder Löschanomalien kommen.

Um die Anomalien weitestgehend zu vermeiden, werden verschiedene Normalformen für relationale Datenbankschemas angewendet.

5 Datenbankabfragen mit SQL

5.1 Geschichtlicher Abriss zu SQL

SQL erblickte Ende der 70er Jahre in einem IBM-Labor in San Jose (Kalifornien) das Licht der Welt. Die Entwicklung war ursprünglich für das IBM-Produkt DB2 vorgesehen.

SQL steht für Structured Query Language - zu Deutsch etwa: strukturierte Abfragesprache. Es handelt sich um eine nichtprozedurale Sprache.

Gegenüber einem einfachen DBMS zeichnet sich ein RDBMS durch die Mengenorientierte Datenbanksprache aus, die bei den meisten RDBMS in SQL realisiert ist. Mengenorientiert heißt, dass SQL die Datenmengen in Gruppen verarbeitet.

Die von den Organisationen ANSI (American National Standards Institute) und ISO (International Standards Organisation) propagierten SQL-Standards dienen den Datenbankherstellern zum Teil nur als Richtlinien. 1986 wurde der erste SQL-Standard vom ANSI verabschiedet (welcher dann 1987 von der ISO ratifiziert wurde). 1992 wurde der Standard deutlich überarbeitet und als SQL-92 (oder auch SQL-2) veröffentlicht. Alle aktuellen Datenbanksysteme halten sich im Wesentlichen an diese Standardversion. Die neuere Version SQL-1999 (ISO/IEC 9075:1999, auch SQL-3 genannt) ist noch nicht in allen Datenbanksystemen implementiert. Das gilt auch für die nächste Version SQL-2003. Der aktuelle Standard wurde 2008 unter SQL-2008 verabschiedet.

Viele bekannte Datenbanksysteme wie DB2, Informix, Microsoft SQL Server, Pervasive P.SQL, MaxDB, MySQL, Oracle, PostgreSQL, Borland Interbase, Firebird, Sybase, SQLite und die neueren Versionen von Microsoft Access implementieren Teile des SQL Sprachstandards. Alle Datenbankprodukte unterscheiden sich mehr oder weniger vom ANSI- Standard. Darüber hinaus zielen die meisten Systeme mit proprietären Erweiterungen darauf ab, SQL zu einer echten prozeduralen Sprache zu machen.

Relationale DBS sind heute Stand der Technik. Ihre Vorteile gegenüber nicht-relationalen DBS sind allgemein anerkannt. Fast alle von ihnen verwenden SQL. Die Differenz zwischen den Herstellern von RDBS, was den Dialekt von SQL betrifft, ist so gering, dass es jedem, der mit den Grundlagen der relationalen Technologie und den Grundstrukturen von SQL vertraut ist, in kürzester Zeit gelingt, mit jedem SQL System zurechtzukommen.

Die SQL-Sprache besteht aus einem Satz von ca. 20 Anweisungen. Der Einsatz dieser Anweisungen kann auf zweierlei Wegen erfolgen:

1. im interaktiven Modus, indem die Anweisungen an einem Terminal eingegeben werden und unmittelbar danach vom DBS ausgeführt werden. Das Ergebnis wird auf dem Terminal ausgegeben.
2. im eingebetteten Modus, in dem SQL-Anweisungen aus einem Programm heraus ausgeführt und ihre Ergebnisse im Programm weiterverarbeitet werden.

5.1.1 Ein Überblick

SQL ist die De-facto-Standardsprache, mit der sich Daten aus relationalen Datenbanken abrufen und dort manipulieren lassen.

Mit SQL kann der Programmierer oder Datenbankadministrator folgende Aufgaben ausführen:

- die Struktur einer Datenbank modifizieren.
- die Einstellungen der Systemsicherheit ändern.
- Benutzerberechtigungen für Datenbanken oder Tabellen einrichten.
- eine Datenbank nach Informationen abfragen.
- den Inhalt einer Datenbank aktualisieren.

Die am häufigsten eingesetzte SQL Anweisung ist die SELECT- Anweisung. Diese Anweisung ruft Daten aus der Datenbank ab und gibt sie an den Benutzer zurück. Neben SELECT bietet SQL Anweisungen für das Erstellen neuer Datenbanken, Tabellen, Felder und Indizes sowie das Einfügen und Löschen von Datensätzen. Außerdem können mit SQL Anweisungen Datenbank- und Objektberechtigungen erteilt oder entzogen werden.

5.1.2 Was ist Transact- SQL?

T-SQL ist eine Erweiterung der SQL-Standardsprache (Sybase und Microsoft) und umfasst Fehlerbehandlung, Row- Processing und Variablen- Deklaration. Es ist die Hauptsprache, für die Kommunikation zwischen Anwendung und SQL Server verwendet wird. T-SQL enthält die DML-, DDL- und DCL- Fähigkeiten von Standard- SQL und zusätzlich erweiterte Funktionen welche die Funktionalität und Flexibilität der Programmierung erhöhen.

Alle gängigen Programmiersprachen wie C++, C#, ASP, PHP, Visual Basic und viele weitere sind in der Lage mit Transact- SQL umzugehen. TSQL bietet Beste Möglichkeiten wenn es darum geht kleine Änderungen am SQL Server durchzuführen.

TSQL ist eine prozedurale und keine objektorientierte Programmiersprache.

6 SELECT- Anweisung

Mit dieser Anweisung ist es möglich, Informationen aus den Basistabellen und Viewtabellen abzufragen.

Allgemeine Syntax:

| | |
|--|---|
| [WITH common_table_expression] | Gibt ein temporäres Ergebnis an, das als allgemeiner Tabellenausdruck (CTE, Common Table Expression) bekannt ist. |
| SELECT spaltenliste | Gibt die Ergebnisspalten an |
| [INTO neuer_tabellenname] | Speichert das Ergebnis in eine neue Tabelle |
| FROM {tabellenname viewname} | Informationsquellen angeben |
| [WHERE suchbedingung] | Auswahlbedingungen für Zeilen |
| [GROUP BY gruppenbildungsbedingung] | Gruppenbildung |
| [HAVING suchbedingung für gruppen] | Auswahlbedingungen für Gruppen |
| [ORDER BY spaltennamen [ASC DESC]] | Sortierung des Abfrageergebnisses |

Die Reihenfolge der einzelnen Klauseln der SELECT- Anweisung ist fest definiert. Es können Klauseln weggelassen werden, wenn bestimmte Regeln (die wir in diesem Kapitel erarbeiten) eingehalten werden. Die Schlüsselwörter SELECT und FROM (außer bei Select ohne Datenquelle) müssen bei jeder SELECT- Anweisung angegeben werden.

In einer Select- Anweisung können bis zu 256 Tabellen angesprochen werden. In der Spaltenliste einer Select- Anweisung können bis zu 4096 Spalten angegeben werden.

6.1 SELECT– Liste und FROM- Klausel

6.1.1 Ermitteln von Daten ohne Datenquelle

Bei dieser Art der SQL- Anweisung wird keine FROM- Klausel benötigt, da keine Basistabelle angegeben wird.

Beispiele:

| | |
|---|--|
| select 'abc'; | Ergebnis: abc |
| select 50 * 33; | Ergebnis: 1650 |
| select 'Paul ' + 'geht heute ' + 'früh nach Hause'; | Ergebnis: Paul geht heute früh nach Hause |
| select getdate(); | Ergebnis: das aktuelle Systemdatum |
| select @@version; | Ergebnis: der Eintrag in dieser globalen Variablen |

6.1.2 Spaltenliste festlegen und Basistabelle angeben

In der SELECT- Liste wird angegeben, ob die Abfrage alle Spalten einer oder mehrerer Tabellen zurückgeben soll oder nur bestimmte Spalten. In der FROM- Klausel werden die Basis- und/oder Viewtabellen angegeben, in denen die Daten enthalten sind, die abgefragt werden.

Beachten Sie:

- Die Spalten werden in der Reihenfolge Abgerufen, in der sie in der SELECT- Liste angegeben sind
- Spaltennamen werden mit Komma getrennt.
- Vermeiden Sie den Stern (*) in der SELECT- Liste, da diese Abfragen sehr Zeit intensiv sind und nicht immer alle Informationen benötigt werden.

Beispiele:

Namen, Farbe und Lagerort aller Artikel.

```
select aname, farbe, astadt
from artikel;
```

Alle Informationen über alle Artikel.

```
select *
from artikel;
```

Werden Informationen nur aus einer Datenbank abgerufen und wurden die Datenbankobjekte in dieser Datenbank vom SA oder Mitgliedern der Gruppe SYSADMIN oder DBCREATOR erstellt, ist nur die Angabe des Tabellen- oder Viewnamen in der FROM- Klausel nötig. Das gleiche gilt für DB- Objekte die sich im Standardschema des jeweiligen Benutzers befinden.

Trifft das nicht zu müssen erweiterte Angaben gemacht werden. Standardmäßig werden alle Namen der Datenbankobjekte wie folgt im DBMS abgelegt:

servername.datenbankname.schema.tabellenname.spaltenname

Daraus ergeben sich folgende Anwendungsmöglichkeiten:

Gesucht sind die Namen aller Lieferanten.

```
select lname from lieferant;
```

oder:

```
select lieferant.lname from lieferant;
```

oder:

```
select lieferant.lname from dbo.lieferant;
```

oder:

```
select dbo.lieferant.lname from dbo.lieferant;
```

oder:

```
select dbo.lieferant.lname from standard.dbo.lieferant;
```

oder:

```
select standard.dbo.lieferant.lname from standard.dbo.lieferant;
```

Da Tabellen- oder Viewnamen oft sehr lang sein können, wird sehr oft mit Tabellen- und Spaltenalias gearbeitet.

Beispiele:

Alle Angaben aller Lieferanten.

```
select a.*  
from lieferant a;
```

Die Wohnorte aller Lieferanten.

```
select a.istadt  
from lieferant a;
```

Da der Abfrageoptimierer zuerst die FROM- Klausel auswertet, ist die Tabelle "lieferant" in den gesamten restlichen Klauseln der Abfrage unter ihrem Alias bekannt. Der Spalten Alias wird in der Regel nur dann benötigt, wenn die Abfrage einen JOIN (Tabellenverknüpfung) ausführt und in der SELECT- Liste bzw. WHERE- Klausel (bei logischen Verknüpfungen) Spaltennamen angegeben wurden, die in mehreren der in der FROM- Klausel aufgeführten Tabellen vorkommen.

6.2 Die WHERE – Klausel

Mit der WHERE- Klausel werden die Bedingung für die von einer Abfrage zurückgegebenen Zeilen angegeben. Sie gibt eine Reihe von Suchbedingungen an, so dass nur Zeilen, die den Suchbedingungen entsprechen, zum Erstellen des Resultsets verwendet werden.

Die WHERE- Klausel wird nicht nur für die Anweisung Select verwendet, sondern auch in der UPDATE- und DELETE- Anweisung.

Eine WHERE- Klausel kann eine unbegrenzte Anzahl von Suchbedingungen enthalten. Eine Suchbedingung ist ein Ausdruck, der den Wert TRUE, FALSE oder UNKNOWN zurückgibt. Zeilen, für die die Suchbedingung TRUE ergibt, werden im Resultset angezeigt.

Bei einem Ausdruck kann es sich um einen Spaltennamen, eine Konstante, eine Skalarfunktion, eine Variable, eine skalare Unterabfrage oder um eine Kombination dieser Elemente handeln, die mit Operatoren verknüpft sind.

Die angeführten Suchbedingungen können auf Spalten der in der FROM- Klausel enthaltenen Basis- oder Viewtabellen verweisen, die nicht in der SELECT- Liste enthalten sein müssen. Für die Suchbedingungen können folgende Operatoren bzw. Schlüsselwörter zur Anwendung kommen:

| Typbezeichnung | Beschreibung |
|---------------------------------------|--------------------------|
| Vergleiche | =, >, <, >=, <=, <>, ... |
| Bereiche | [not] between |
| Listen | [not] in |
| Zeichenfolgeübereinstimmung | [not] like |
| unbekannte Spaltenwerte | is [not] null |
| Kombination von Vergleichsbedingungen | and, or |
| Negation von Vergleichsbedingungen | not |

Hinweis: Versuchen Sie die Verwendung von NOT zu vermeiden. Sie beinträchtigen die Leistungsfähigkeit des Abfrageoptimierers erheblich.

6.2.1 Vergleichsoperatoren

Vergleichsoperatoren werden mit Zeichendaten, numerischen Daten oder Datumsdaten verwendet und können in der WHERE- oder HAVING- Klausel einer Abfrage eingesetzt werden. Vergleichsoperatoren ergeben einen Wert vom booleschen Datentyp; sie geben abhängig vom Ergebnis der getesteten Bedingung TRUE oder FALSE zurück.

Syntax:

spaltenausdruck OPERATOR {wert | unterabfrage}

| Operator | Bedeutung |
|----------|--|
| = | Gleich |
| > | Größer als |
| < | Kleiner als |
| >= | Größer gleich |
| <= | Kleiner gleich |
| <> | Ungleich |
| != | Nicht gleich (kein ISO- Standard) |
| !< | Nicht kleiner als (kein ISO- Standard) |
| !> | Nicht größer als (kein ISO- Standard) |

Beispiel:

Die Namen der Hamburger Lieferanten.

```
select lname
from lieferant
where lstadt = 'Hamburg';
```

Hinweis: SQL Server verlangt entweder einfache ("") oder doppelte ("") Anführungszeichen für alphanumerische Werte und Datumswerte. Für numerische Werte werden keine Anführungszeichen benötigt. Der Spaltenausdruck kann über den Vergleichsoperator nur mit einem Wert verglichen werden.

Folgende Abfrage ist also falsch.

Die Namen der Hamburger und Aachener Lieferanten.

```
select lname
from lieferant
where lstadt = 'Hamburg', 'Aachen';
```

6.2.2 Schlüsselwort BETWEEN

Das Schlüsselwort BETWEEN wählt Datensätze innerhalb eines angegebenen Bereiches aus. Die angegebenen Grenzwerte sind dabei in das Ergebnis eingeschlossen.

Das Schlüsselwort BETWEEN wird stets mit AND verwendet und gibt in einer Suchbedingung einen inklusiven Bereich an, auf den hin geprüft werden soll.

Syntax:

```
spaltenausdruck [NOT] BETWEEN anfangs_ausdruck AND ende_ausdruck
```

Beispiel:

Alle Lieferungen zwischen dem 01.07.2010 und dem 20.10.2010.

```
select *
from lieferung
where Idatum between '01.07.2010' and '20.10.2010';
```

Alle Artikelnamen der Artikel mit einem Gewicht von 14 bis 17 Gramm.

```
select aname
from artikel
where gewicht between 14 and 17;
```

Durch die Regeln beim Vergleich von alphanumerischen Zeichen (lt. ASCII- Zeichensatz) ergeben sich für die Verwendung von BETWEEN folgende Möglichkeiten der Datenabfrage.

Beispiel:

Alle Namen und Wohnorte der Lieferanten, die an einem Ort wohnen dessen Name mit H bis L beginnt.

```
select Iname, Istadt
from lieferant
where Istadt between 'H' and 'M';
```

oder:

```
select Iname, Istadt
from lieferant
where Istadt between 'H' and 'Lz';
```

Die letzten beiden Beispiele sind keine üblichen Varianten. Normalerweise werden solche Abfragen mit LIKE durchgeführt.

6.2.3 Schlüsselwort IN

Ermittelt, ob ein bestimmter Spaltenwert mit einem Wert aus einer Unterabfrage oder Liste übereinstimmt.

Syntax:

```
spaltenausdruck [NOT] IN ({unterabfrage | wert [,...n]})
```

Beispiel

Die Namen der Hamburger und Aachener Lieferanten.

```
select Iname
from lieferant
where Istadt in ('Hamburg','Aachen');
```

Alle Lieferungen mit Liefermengen von 100, 200 oder 400 Stück.

```
select *
from lieferung
where lmenge in (100,200,400);
```

6.2.4 Schlüsselwort LIKE

Bestimmt, ob die angegebene Zeichenfolge einem angegebenen Muster entspricht. Ein Muster kann normale Zeichen und Platzhalterzeichen einschließen. Bei einem Mustervergleich müssen normale Zeichen exakt mit den angegebenen Zeichen in der Zeichenfolge übereinstimmen. Platzhalterzeichen können jedoch mit beliebigen Teilen der Zeichenfolge übereinstimmen.

Das Verwenden der Vergleichsoperatoren für Zeichenfolgen "=" und "!=" ist nicht so flexibel wie das Verwenden von Platzhalterzeichen mit dem LIKE- Operator.

Hinweis: Ist eines der Argumente nicht vom Zeichenfolgen-Datentyp, versucht SQL Server, dieses Argument gegebenenfalls in den Zeichenfolgen-Datentyp zu konvertieren.

Syntax:

spaltenausdruck [NOT] LIKE 'zeichenfolge' [ESCAPE maskierungszeichen]

Der LIKE- Operator kann mit folgenden Platzhalterzeichen verwendet werden.

| Platzhalter | Beschreibung | Beispiel |
|-----------------|---|--|
| % | Eine Zeichenfolge aus null oder mehreren beliebigen Zeichen | WHERE aname LIKE '%welle%' findet alle Artikel, die das Wort 'welle' enthalten. |
| _ (Unterstrich) | Ein einzelnes beliebiges Zeichen | WHERE aname LIKE '_ol_n' findet alle Artikelnamen mit sechs Buchstaben, die auf n enden und wo nach dem ersten Buchstaben die Zeichenkette ol als dritter und vierter Buchstabe steht. |
| [] | Beliebiges einzelnes Zeichen im angegebenen Bereich oder in der angegebenen Menge | WHERE aname LIKE '[M-S]%' findet alle Artikelnamen, die mit einem Zeichen zwischen M und S beginnen. |
| [^] | Beliebiges einzelnes Zeichen, das sich nicht im angegebenen Bereich oder in der angegebenen Menge befindet. | WHERE aname LIKE '_[^c]%' findet alle Artikelnamen, deren zweiter Buchstabe nicht c ist. |

Die verwendete Sortierreihenfolge beeinflusst die Arbeitsweise des Schlüsselworts LIKE nicht. Wenn Sie eine andere Sortierreihenfolge (andere Sortierung, andere Festlegung zur Beachtung der Groß-/ Kleinschreibung) angegeben haben, erhalten Sie möglicherweise andere Ergebnisse.

Das Schlüsselwort ESCAPE bietet die Möglichkeit, einen Mustervergleich für die Platzhalterzeichen selbst (^, %, [] und _) durchzuführen. Nach dem Schlüsselwort ESCAPE wird das Zeichen angegeben, das als Maske (ESCAPE- Zeichen) verwendet werden soll.

Dieses Zeichen signalisiert, dass im Zeichenfolgenausdruck nach einer buchstäblichen Entsprechung für das folgende Zeichen gesucht werden soll.

Beispiel:

Zeige alle Artikelnamen mit Unterstrich an.

```
select aname
from artikel
where aname like '%y_%' escape 'y';
```

Um effiziente Abfragen zu erstellen sollte auf vorangestellte Platzhalter wie -like '%en' verzichtet werden, besser sind Platzhalter in der Mitte oder am Ende.

6.2.5 Arbeiten mit NULL- Werten

Ein NULL- Wert weist darauf hin, dass der Wert unbekannt ist. Ein NULL- Wert unterscheidet sich von einem leeren Wert oder dem numerischen Wert Null (0).

Vergleiche zwischen zwei NULL- Werten oder zwischen einem NULL- Wert und einem anderen Wert geben ein nicht definiertes Ergebnis zurück, da der Wert jedes NULL- Wertes unbekannt ist.

NULL- Werte weisen in der Regel auf Daten hin, die unbekannt oder nicht zutreffend sind oder zu einem späteren Zeitpunkt hinzugefügt werden sollen.

Syntax:

spaltenausdruck IS [NOT] NULL

Es gilt für NULL- Werte:

- Um das Vorhandensein von NULL- Werten in einer Abfrage zu testen, verwenden Sie IS NULL oder IS NOT NULL in der WHERE- Klausel.
- Wenn Abfrageergebnisse in SQL Server Management Studio angezeigt werden, werden NULL- Werte im Resultset als NULL dargestellt.
- NULL- Werte können in eine Spalte eingefügt werden, indem NULL explizit in einer INSERT- oder UPDATE- Anweisung angegeben wird, indem eine Spalte aus einer INSERT- Anweisung weggelassen wird oder indem beim Hinzufügen einer neuen Spalte zu einer vorhandenen Tabelle die ALTER TABLE- Anweisung verwendet wird.

Beispiel:

Gesucht sind alle Lieferanten deren Wohnort noch nicht bekannt ist.

```
select *
from lieferant
where Istadt is NULL;
```

Da NULL- Werte unerwartet auftreten können, empfiehlt es sich, diese mit sogenannten NULL- Ersetzungsfunktionen aufzufangen. SQL Server stellt dafür drei Funktionen bereit.

6.2.5.1 ISNULL- Funktion

Gibt beim Auftreten einer NULL- Marke einen Ersatzwert zurück. Sonst wird der entsprechende Spaltenwert zurückgegeben.

Syntax:

isnull(ausdruck, rückgabewert)

Beispiel:

Gesucht sind Lieferanten mit ihren Statuswert. Ist der Status unbekannt soll die Zahl 0 zurückgegeben werden.

```
select lnr, lname, isnull(status, 0) as [status]
from lieferant;
```

6.2.5.2 NULLIF- Funktion

Diese Funktion gibt einen NULL- Wert zurück wenn beide Ausdrücke im Argument dieser Funktion gleich sind. Sind die Werte nicht gleich wird der Wert des ersten Ausdrucks zurückgegeben.

Syntax:

```
nullif(ausdruck1, ausdruck2)
```

Beispiel:

Gesucht sind die Nachnamen von Lieferanten. Wenn der Nachname und der Vorname eines Lieferanten gleich sind, soll eine NULL- Marke zurückgegeben werden.

```
select lnr, nullif(lname,vname) as [namen]
from lieferant;
```

6.2.5.3 COALESCE- Funktion

Diese Funktion gibt den ersten Ausdruck seiner Parameterliste zurück der nicht NULL ist. Sind alle Ausdrücke NULL dann gibt die Funktion NULL zurück.

Syntax:

```
coalesce(ausdruck1, ausdruck2 [,...])
```

Beispiel:

Lieferanten mit ihren Lieferantennamen und dem Spaltenwert einer weiteren Spalte die keine NULL- Marke aufweist.

```
select lnr, lname, coalesce(vname, status, lstadt) as [erster bekannter Wert]
from lieferant;
```

Beispiel:

Gesucht sind Lieferanten mit ihren Statuswert. Ist der Status unbekannt soll die Zahl 0 zurückgegeben werden.

```
select lnr, lname, coalesce(status, 0) as [status]
from lieferant;
```

6.2.6 Festlegen mehrerer Suchbedingungen für mehrere Spalten

Sie können den Bereich der Abfrage erweitern oder begrenzen, indem Sie mehrere Datenspalten als Teil der Suchbedingung einschließen.

Wenn Sie eine Abfrage erstellen möchten, die nach Werten in einer der beiden (oder in mehreren) Spalten sucht, legen Sie eine OR-Bedingung fest.

Wenn Sie eine Abfrage erstellen möchten, die allen Bedingungen in zwei (oder mehreren) Spalten entsprechen muss, legen Sie eine AND-Bedingung fest.

Priorität:

- AND = Abfrageergebnis wird wahr, wenn sämtliche Suchbedingungen erfüllt sind.
 OR = Abfrageergebnis wird wahr, wenn mindestens eine Suchbedingung erfüllt ist.
 NOT = negiert den auf den auf diesen Operator folgenden Ausdruck

Hinweis: Durch Klammersetzung kann die Priorität bei der Anwendung der Oben-stehenden Operatoren manipuliert werden.

Beispiel OR-Bedingung:

Die Namen und Statuswerte der Lieferanten aus Aachen und Hamburg.

```
select lname, status
from lieferant
where lstadt = 'Aachen' or lstadt = 'Hamburg';
```

Beispiel AND-Bedingung:

Die Artikelnamen der Artikel, die in einer Stadt lagern, deren Ortsname mit E bis L beginnen und deren Gewicht größer 15 Gramm ist.

```
select aname
from artikel
where gewicht > 15 and astadt like '[E-L]%' ;
```

Beispiel AND- und OR-Bedingung:

Die Artikelnamen der Artikel, die in einer Stadt lagern, deren Ortsname mit E bis L beginnen und deren Gewicht größer 15 Gramm ist, oder der Artikel, deren Lagermenge größer 700 Stück ist.

```
select aname
from artikel
where gewicht > 15 and astadt like '[E-L]%' or amenge > 700;
```

Beispiel AND- und OR-Bedingung und Klammersetzung:

Die Artikelnamen der Artikel, die in einer Stadt lagern, deren Ortsname mit E bis L beginnen und deren Gewicht größer 15 Gramm ist oder deren Lagermenge größer 700 Stück ist und deren Gewicht größer 15 Gramm ist..

```
select aname
from artikel
where gewicht > 15 and (astadt like '[E-L]%' or amenge > 700);
```

6.3 Formatieren von Ergebnissen

In eingeschränkter Weise kann das Abfrageergebnis einer Ad-Hoc Abfrage übersichtlicher formatiert und sortiert werden.

Es dient dazu die Ergebnisse einer Abfrage für den Leser übersichtlicher zu gestalten.

Es gibt fünf Möglichkeiten:

1. Einfügen von Text (erläuternder Text) in die Ergebnismenge.
2. Ändern von Spaltenüberschriften.
3. Entfernen doppelter Reihen (Datensätze) aus dem Ergebnis.
4. Sortieren der Ergebnismenge.
5. Case

6.3.1 Erläuternder Text (Literalen)

Für die Übersichtlichkeit des Ergebnisses wird so genannter erläuternder Text in die SELECT-Liste als zusätzliche Spalte eingefügt.

Syntax:

```
select {Spaltenname | 'Text'}[, {Spaltenname | 'Text'}[,...]]
from {Tabellenname | Viewname}
```

Beispiel:

```
select anr, aname, gewicht, 'Gramm', anmenge, 'Stück'
from artikel;
```

6.3.2 Ändern von Spaltennamen

Durch so genannte „Spalten- Aliase“ können abstrakte Spaltennamen (vnum, anum, lname,...) in den Abfrageergebnissen in übersichtliche Spaltenüberschriften umgewandelt werden. Standardmäßig wird der Spaltenname als Überschrift verwendet.

Es stehen drei Methoden zur Verfügung, zwischen denen funktional kein Unterschied besteht:

Syntax:

```
select spaltenname as überschrift [, spaltenname as überschrift, ...]
from tabellenname
```

```
select spaltenüberschrift = spaltenname [, überschrift = spaltenname, ...]
from tabellenname
```

```
select spaltenname überschrift [, spaltenname überschrift, ...]
from tabellenname
```

Die erste Methode entspricht dem ANSI Standard.

Spaltenüberschriften, die Leerzeichen enthalten oder nicht den Benennungskonventionen für SQL Server-Objekte entsprechen, müssen in einfache Anführungszeichen gesetzt werden.

Ein Spaltenaliasname kann bis zu 128 Zeichen lang sein. Spaltenaliasnamen können auch für berechnete Spalten erstellt werden.

6.3.3 Entfernen doppelter Reihen aus dem Ergebnis

Wenn im Ergebnis einer Abfrage keine Mehrfachnennungen einzelner Werte benötigt werden, können diese durch das Schlüsselwort DISTINCT aus dem Ergebnis entfernt werden.

Beispiel:

Welcher Lieferant hat bisher geliefert? (ohne distinct)

```
select lnr Ergebnis:      12 Datensätze
      from lieferung;
```

Welcher Lieferant hat bisher geliefert? (mit distinct)

```
select distinct lnr Ergebnis:      4 Datensätze
      from lieferung;
```

DISTINCT wirkt sich auf den ganzen Datensatz der Ergebnismenge aus, darum wird es auch nur einmal unmittelbar hinter dem Schlüsselwort SELECT angegeben. Die Gleichheit wird über alle in der SELECT- Liste angegebenen Spalten überprüft. DISTINCT gibt das Resultset unsortiert zurück.

6.3.4 Sortieren der Ergebnismenge

Um die Ergebnismenge einer Abfrage nach bestimmten Spaltenwerten zu sortieren, wird die ORDER BY- Klausel verwendet.

Syntax:

```
select spaltenname [, spaltenname, ...]
from tabellenname
order by {spaltenname|select-Listennummer|Ausdruck}
[collate collation_name[ASC|DESC] [, ...]]
```

Es finden in relationalen Datenbanksystemen drei Methoden der Sortierung Anwendung:

1. Sortieren nach den Spaltennamen.

```
select aname, farbe, astadt
from artikel
order by aname, astadt ASC;
```

2. Sortieren nach Spaltenaliasnamen.

```
select aname as 'Artikelname', farbe as 'Artikelfarbe', astadt as 'Lagerort'
from artikel
order by 'Artikelname', 'Lagerort' ASC;
```

3. Sortieren nach SELECT- Listennummer.

```
select aname, farbe, astadt
from artikel
order by 1, 3 ASC;
```

4. Sortieren in der deutschen Telefonbuchsortierung.

```
select aname , farbe , astadt
from artikel
order by aname, 'astadt' collate German_Phonebook_CI_AS ASC;
```

Alle drei Methoden können vermischt werden. Das Resultset kann nach Spalten sortiert werden, die nicht in der SELECT- Liste aufgeführt sind (außer bei DISTINCT und UNION).

Für die Anzahl der Elemente in der ORDER BY- Klausel gibt es keine Beschränkung. Allerdings ist die Zeilengröße der für Sortievorgänge benötigten temporären Arbeitstabellen auf 8.060 Byte beschränkt. Hierdurch wird die Gesamtgröße der in einer ORDER BY- Klausel angegebenen Spalten beschränkt.

Die Sortierung ist standardmäßig in aufsteigender Reihenfolge (ASC - ascend). Das Ergebnis kann aber auch absteigend (DESC - descend) sortiert werden. Diese Auswahl ist explizit für jede Spalte in der ORDER BY- Klausel möglich.

6.3.4.1 Sortieren der Ergebnismenge mit OFFSET und FETCH

Es wird empfohlen, die OFFSET- und FETCH- Klausel statt der TOP- Klausel zu verwenden, um eine Abfrageauslagerung zu implementieren und die Anzahl der an einen Client gesendeten Datensätze einzuschränken.

Syntax:

```
select spaltenname [, spaltenname, ...]
from tabellenname
order by {spaltenname|select-Listennummer|Ausdruck}
          [collate collation_name[ASC|DESC] [, ...]]
[
  offset {int_wert | scalar_expr} {row | rows}
  [fetch {first | next} { int_wert | scalar_expr} {row | rows} only]
]
```

Die Option OFFSET gibt an, die Anzahl der Datensätze die übersprungen werden soll bevor ein Resultset zurückgegeben werden soll.

Die Option FETCH gibt die Anzahl der Zeilen an, die zurückgegeben werden sollen nachdem die OFFSET- Klausel verarbeitet wurde.

Beispiel:

Die fünf stärksten Lieferungen nach den ersten drei starken Lieferungen.

```
select a.Inr, Iname, Imenge
from lieferant a join lieferung b on a.Inr = b.Inr
order by Imenge desc
offset 3 rows fetch next 5 rows only;
```

6.3.5 Auflisten der TOP n – Werte

Durch die Verwendung von TOP (n) werden nur die ersten N- Zeilen oder N- Prozent eines Resultsets aufgelistet. Der TOP-Ausdruck kann in SELECT-, INSERT-, UPDATE-, MERGE- und DELETE- Anweisungen verwendet werden.

Syntax:

```
[ TOP ({n | @var}) [PERCENT] [ WITH TIES] ]
```

Richtlinien:

- verwenden sie eine ORDER BY- Klausel und geben sie dort den Bereich (ASC für die Kleinsten, DESC für die Größten) der Werte an. Andernfalls werden die ersten n- zufälligen Datensätze zurückgegeben.
- verwenden Sie hinter dem TOP- Schlüsselwort eine Zahl größer null oder eine Variable vom Datentyp integer, bigint oder float die größer null ist.
- wenn das TOP n PERCENT- Schlüsselwort eine Zeile mit Dezimalstellen ergibt, runden SQL Server auf die nächste ganze Zahl auf.
- Klammern zur Begrenzung von expression in TOP sind in INSERT-, UPDATE- und DELETE-Anweisungen erforderlich. Zum Sicherstellen der Abwärtskompatibilität wird die Verwendung von TOP expression ohne Klammern in SELECT-Anweisungen unterstützt, jedoch nicht empfohlen.
- verwenden Sie die WITH TIES- Klausel, um Übereinstimmungen im Resultset einzuschließen. Übereinstimmungen entstehen wenn, wenn zwei oder mehr Werte mit denen letzten Zeilen identisch sind.
- WITH TIES kann nur in Verbindung mit einer ORDER BY- Klausel verwendet werden.

Beispiel:

Gesucht sind die drei Lieferungen mit den höchsten Liefermengen.

```
select TOP (3) Inr, anr, lmenge, ldatum
from lieferung
order by lmenge DESC;
```

Gesucht sind die Lieferungen mit den drei höchsten Liefermengen.

```
select TOP (3) WITH TIES Inr, anr, lmenge, ldatum
from lieferung
order by lmenge DESC;
```

Gesucht sind die Lieferungen mit den drei höchsten Liefermengen.

```
declare @anz integer
set @anz = 3

select TOP (@anz) WITH TIES Inr, anr, lmenge, ldatum
from lieferung
order by lmenge DESC;
```

6.3.6 Case Ausdrücke

Wertet eine Liste von Bedingungen aus und gibt einen von mehreren möglichen Ergebnisausdrücken zurück. Der Case Ausdruck entspricht dem SQL 92- Standard.

Eine übliche Verwendung der CASE- Funktion ist das Ersetzen von Codes oder Abkürzungen durch besser lesbare Werte oder auch das Kategorisieren von Daten.

6.3.6.1 Die einfache CASE- Funktion

Die einfache CASE- Funktion vergleicht einen Ausdruck mit mehreren einfachen Ausdrücken, um das Ergebnis zu bestimmen.

Syntax:

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

Beispiel:

```
select anr, aname, amenge, case astadt
    when 'Hamburg' then 'lagert in Hamburg'
    when 'Ludwigshafen' then 'lagert in Ludwigshafen'
    else 'lagert an einem anderen Ort'
    end as 'Lagerorte'
from artikel;
```

6.3.6.2 Die komplexe CASE- Funktion

Die komplexe CASE- Funktion wertet eine Menge boolescher Ausdrücke aus, um das Ergebnis zu bestimmen.

Syntax:

```
CASE
    WHEN Boolean_expression THEN result_expression
    [...]
    [ELSE else_result_expression]
END
```

Beispiel:

```
select anr, aname, astadt, case
    when amenge between 0 and 400 then 'Nachbestellen'
    when amenge between 401 and 1000 then 'Ausreichend'
    else 'Überkapazität'
    end as 'Bewertung'
from artikel;
```

6.4 Berechnen der Ergebnismengen

6.4.1 Arithmetische Operatoren zur Berechnung

Arithmetische Operatoren führen mathematische Operationen mit zwei Ausdrücken mit beliebigen Datentypen der numerischen Datentypkategorie aus.

| Operator | Bedeutung |
|------------|---|
| + | Addition |
| - | Subtraktion |
| * | Multiplikation |
| / | Division |
| % (Modulo) | Gibt den ganzzahligen Rest einer Division zurück. Beispiel: 12 % 5 = 2 (der Rest von 12 geteilt durch 5 ist 2). |

Mit Plus (+) und Minus (-) können auch arithmetische Operationen für DATETIME- und SMALLDATETIME- Werte ausgeführt werden.

Beispiel:

```
select amenge * 0.001 as 'Gewicht in Kilogramm'
from artikel;
```

Hinweis: Bei der Division ist der Datentyp des Ergebnisses der in der Rangfolge höhere Datentyp von Dividend oder Divisor (Rangfolge: REAL → DECIMAL → MONEY → INTEGER).

6.4.2 Operator + für das Verketten von Zeichenfolgen

Der Operator für das Verketten von Zeichenfolgen wird durch das Pluszeichen (+) dargestellt und fügt zwei Zeichenfolgen aneinander.

Wenn zwei Zeichenfolgen verkettet werden, wird die Sortierung des Ergebnisausdrucks entsprechend den Regeln zur Sortierungsriorität festgelegt.

Beispiel:

```
select 'Der Lieferant ' + lName + ' wohnt in ' + lCity
from lieferant;
```

6.4.3 Operator CONCAT für das Verketten von Zeichenfolgen

CONCAT gibt eine Zeichenfolge zurück, die das Ergebnis der Verkettung von zwei oder mehr Zeichenfolgewerten darstellt.

Syntax:

```
concat( zeichenfolge_1, zeichenfolge_2 [, zeichenfolge_n]);
```

6.4.4 Skalare Funktionen

Verarbeiten keinen, einen oder mehrere Werte und geben dann einen einzelnen Wert zurück. Diese Funktionen können überall dort angewendet werden, wo in einer SELECT-Abfrage Ausdrücke stehen dürfen.

Skalarfunktionen können in folgende Kategorien aufgeteilt werden.

| Funktionskategorie | Erklärung |
|-------------------------------|---|
| Konfigurationsfunktionen | Geben Informationen über die aktuelle Konfiguration zurück. |
| Cursorfunktionen | Geben Informationen über Cursor zurück. |
| Datums- und Zeitfunktionen | Verarbeiten Datums- und Zeiteingabewerte und geben eine Zeichenfolge, einen Zahlen-, Datums- oder Zeitwert zurück. |
| Mathematische Funktionen | Führen Berechnungen auf der Grundlage von Eingabewerten durch, die als Parameter für die Funktion bereitgestellt werden, und geben einen numerischen Wert zurück. |
| Metadatenfunktionen | Geben Informationen über die Datenbank und Datenbankobjekte zurück. |
| Sicherheitsfunktionen | Geben Informationen über Benutzer und Rollen zurück. |
| Zeichenfolgenfunktionen | Verarbeiten Zeichenfolgen (des Typs char oder varchar) und geben eine Zeichenfolge oder einen numerischen Wert zurück. |
| Systemfunktionen | Führen Operationen bezüglich Werten, Objekten und Einstellungen in Microsoft® SQL Server™ aus und geben Informationen über diese zurück. |
| Statistische Systemfunktionen | Geben statistische Informationen über das System zurück. |
| Text- und Imagefunktionen | Verarbeiten Text- bzw. Image-Eingabewerte oder -Spalten und geben Informationen über diese Werte zurück. |

Alle Funktionen sind deterministisch oder nicht deterministisch:

- Deterministische Funktionen geben stets dasselbe Ergebnis zurück, wenn sie mit bestimmten Eingabewerten aufgerufen werden.
- Nicht deterministische Funktionen können unterschiedliche Ergebnisse zurückgeben, wenn sie mit bestimmten Eingabewerten aufgerufen werden.

Ob eine Funktion deterministisch oder nicht deterministisch ist, wird als Determinismus der Funktion bezeichnet.

Beispielsweise ist die DATEADD- Funktion deterministisch, weil sie für bestimmte Argumentwerte stets dasselbe Ergebnis für die drei Parameter zurückgibt. GETDATE ist nicht deterministisch, weil diese Funktion mit demselben Argument aufgerufen wird, aber der zurückgegebene Wert ist bei jeder Ausführung unterschiedlich.

6.4.4.1 Mathematische (Numerische) Funktionen

Skalaren Funktionen führen, normalerweise mit als Argumente angegebenen Eingabewerten, Berechnungen durch und geben einen numerischen Wert zurück. Alle mathematischen Funktionen, außer RAND, sind deterministisch, bei jedem Aufrufen mit bestimmten Eingabewerten werden immer die gleichen Ergebnisse zurückgegeben. RAND ist nur deterministisch, wenn ein Startwert (seed) angegeben wird.

Beispiel:

Gesucht ist die Artikelnummer, der Artikelname und das Gesamtlagergewicht der Artikel in Kilogramm.

```
select anr, aname, round(gewicht * amenge * 0.001, 2) as 'Gesamtlagergewicht in Kilo'
from artikel;
```

6.4.4.2 Datumsfunktionen

Berechnen die entsprechende Datums- und Zeiteinheit aus einem Ausdruck, oder liefern den Wert aus einem Zeitintervall.

Alle Datumsfunktionen verwenden folgende Datums- und Zeiteinheiten:

| Datumseinheit | Abkürzungen |
|---------------|-------------|
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |

Datums- und Zeitfunktionen können in einer SELECT- Anweisung überall dort stehen, wo Ausdrücke stehen können. Datumsangaben werden in Hochkomma gesetzt.

In nachfolgender Tabelle ist die Determinismus- Eigenschaft einiger dieser Funktionen aufgelistet.

| Funktion | Determinismus |
|------------|-----------------------|
| DATEADD | Deterministisch |
| DATEDIFF | Deterministisch |
| DATENAME | Nicht deterministisch |
| DATEPART | Deterministisch |
| DAY | Deterministisch |
| GETDATE | Nicht deterministisch |
| GETUTCDATE | Nicht deterministisch |
| MONTH | Deterministisch |
| YEAR | Deterministisch |

Beispiel:

Gesucht sind die Artikelnummern, das Lieferdatum und das mögliche Zahlungsziel (30 Tage).

```
select anr, ldatum, dateadd(dd, 30, ldatum) as 'Zahlungsziel'
from lieferung;
```

Bei Zeichenfunktionen muss unbedingt beachtet werden mit welchem Zeichensatz gearbeitet wird. Es kann sonst zu unerwarteten Ergebnissen kommen.

Beispiel:

Die erste Abfrage gibt eine 1 zurück weil als Sprache italienisch eingestellt wurde und die zweite Abfrage mit der englischen Sprache gibt eine 7 zurück.

```
SET LANGUAGE Italian;
SELECT @@DATEFIRST as 'first day to week', datepart(dw, getdate()) as 'day of week';
GO
SET LANGUAGE us_english;
SELECT @@DATEFIRST as 'first day to week', datepart(dw, getdate()) as 'day of week';
```

6.4.4.3 Zeichenfolgenfunktionen

Mit diesen Funktionen können Zeichenfolgen bearbeitet werden. Sie verarbeiten Zeichenfolgen-Eingabewerte und geben einen numerischen Wert oder eine Zeichenfolge zurück.

Alle integrierten Zeichenfolgefunktionen sind mit Ausnahme von CHARINDEX und PATINDEX deterministisch. Sie geben stets denselben Wert zurück, wenn sie mit bestimmten Eingabewerten aufgerufen werden.

Beispiel:

Gesucht ist der numerische Teil der Artikelnummern aller Artikel.

```
select substring(anr, 2, 2)
from artikel;
```

6.4.4.4 Systemfunktionen

Systemfunktionen führen Operationen auf Werten, Objekten und Einstellungen in SQL Server aus und geben Informationen über diese zurück. Es gibt Deterministische und nicht Deterministische Systemfunktionen (siehe Systemfunktionen in der Onlinedokumentation).

Alle weiteren Funktionen sind in der Onlinedokumentation aufgeführt und können dort nachgelesen werden.

Beispiel:

Gesucht ist der Name der Arbeitsstation.

```
select host_name();
```

6.4.4.4.1 Die Konvertierungsfunktionen CAST, CONVERT und PARSE

Konvertiert einen Ausdruck explizit von einem Datentyp in einen anderen. Die Funktionalität von CAST und CONVERT ist ähnlich.

SQL Server benutzt auch eine implizite Konvertierung, wenn zum Beispiel Daten eines Typs mit einem anderen verglichen werden. Sie ist für den Benutzer nicht sichtbar.

Beispiel:

```
...
where smallint_spalte = integer_wert
```

Dabei ist die Datentyprangfolge wichtig, das heißt ein Datentyp mit niedriger Priorität wird in der Regel implizit in einen Datentyp mit höherer Priorität umgewandelt.

Beispiel:

Von niedriger Priorität nach hoher Priorität.

```
CHAR -> VARCHAR -> NVARCHAR -> TINYINT -> INT -> DECIMAL ->
TIME -> DATE -> DATETIME2 -> XML
```

Die Konvertierung von einem Typ mit höherer Priorität in einen Typ mit niedrigerer Priorität erfolgt immer explizit mit der CAST- Funktion.

Beim Verschieben von Daten aus einem Objekt in ein anderes Objekt oder beim Zuweisen eines Abfragewertes an eine Variable kann sich eine explizite Datentypkonvertierung notwendig erweisen.

Syntax:

Verwenden von CAST:

```
cast (expression as data_type)
```

Verwenden von CONVERT:

```
convert ( data_type [ ( length ) ] , expression [ , style ] )
```

Verwenden von PARSE:

```
parse ( string_wert as data_type [ using culture ] )
```

CAST entspricht dem ANSI- Standard, die Funktion CONVERT nicht. Mit der CAST Funktion können keine Datumswerte in verschiedene Formate konvertiert werden. Dafür muss die CONVERT Funktion verwendet werden. Beide Funktionen können in der SELECT- und WHERE- Klausel verwendet werden.

PARSE wandelt einen String in ein angegebenes Datenformat für eine optionale Kultur um. Wenn die Konvertierung fehlschlägt bei oben stehenden Funktionen fehlschlägt wird eine Fehlermeldung erzeugt und der Vorgang abgebrochen. Wollen Sie aber eine Fehlerbehandlung im Stapel durchführen verwenden Sie die Funktion TRY_PARSE. Wenn die Konvertierung hier fehlschlägt gibt sie NULL zurück.

Beispiele:

Gesucht ist der numerische Teil der Artikelnummern aller Artikel als Integer Wert.

```
select cast(substring(anr, 2,2) as integer)
from artikel;
```

Gesucht sind die Artikelnummern und das Lieferdatum dieser Artikel im deutschen Datumsformat mit vierstelliger Jahreszahl und ohne Uhrzeit.

```
select anz, convert(char(10), ldatum, 104) as 'Lieferdatum'
from lieferung;
```

Einsetzen von CAST zur Formatierung von Spalten.

```
select anr, cast(ename as char(15)) as 'Artikel', farbe
from artikel;
```

Umwandeln eines deutschen Datumsstring in einen DATE- Wert.

```
select parse('22. März 2012' as date using 'de-DE');
```

6.4.4.4.2 Die Konvertierungsfunktionen TRY_CONVERT und TRY_PARSE

TRY_CONVERT() gibt einen in den angegebenen Datentyp umgewandelten Wert zurück wenn die Umwandlung erfolgreich war oder der Wert NULL wird zurückgegeben wenn die Konvertierung nicht durchführbar ist.

Syntax:

```
try_convert(data_type , wert [, style])
```

Diese Funktion ist nur ab Kompatibilitätsgrad 110 verfügbar und kann nur auf Servern mit SQL Server 2012 und höher ausgeführt werden.

Beispiel:

Es soll überprüft werden ob sich der Wert "test" in den Datentyp FLOAT umwandeln lässt.

```
select case
    when try_convert(float, 'test') is null then 'Konvertierung fehlgeschlagen'
    else 'Konvertierung erfolgt'
end as [Ergebnis];
```

TRY_PARSE() gibt einen in den angegebenen Datentyp umgewandelten Wert zurück wenn die Umwandlung erfolgreich war oder der Wert NULL wird zurückgegeben wenn die Konvertierung nicht durchführbar ist.

Syntax:

```
try_parse(wert as data_type [using culture])
```

Diese Funktion sollte ausschließlich verwendet werden um Zeichenfolgen in ein Datum oder eine Uhrzeit zu konvertieren. Für die allgemeine Typkonvertierung sollte CAST oder CONVERT verwendet werden.

Diese Funktion ist nur ab Kompatibilitätsgrad 110 verfügbar und kann nur auf Servern mit SQL Server 2012 und höher ausgeführt werden. Es muss .NET Framework Common Language Runtime vorhanden sein.

Beispiel:

Überprüfen ob ein Datumsstring der deutschen Kultureinstellung entspricht.

```
set language german;
select case when try_parse('01/31/2013' as datetime using 'de-DE') is null then 'Nein'
            else 'Ja'
        end as [Ergebnis];
```

6.5 Gruppieren und Zusammenfassen von Daten

Beim Abrufen von Daten kann es sinnvoll sein, die Daten zu gruppieren oder zusammenzufassen.

6.5.1 Aggregatfunktionen

Aggregatfunktionen (auch Mengenfunktionen oder statistische Funktionen) beziehen sich auf jeweils eine Spalte einer Tabelle und fassen die Werte der Spalte in einen Skalar Wert zusammen. Mit Ausnahme von COUNT ignorieren Aggregatfunktionen alle NULL-Werte.

Das Ergebnis einer Aggregatfunktion ist immer nur genau ein Wert.

In der SELECT-Liste können neben einer Aggregatfunktion nur weitere Aggregatfunktionen, eine oder mehrere Gruppierungsspalten aus einer GROUP BY- Klausel und ein oder mehrere konstante Ausdrücke stehen, sonst keine weiteren Elemente

6.5.1.1 Einsatz von Aggregatfunktionen

Aggregatfunktionen sind als Ausdrücke nur in folgenden Elementen zulässig:

- In der Auswahlliste einer SELECT-Anweisung (Unterabfrage oder äußere Abfrage).
- In einer HAVING- Klausel.

Die Transact-SQL-Programmiersprache stellt umfangreiche Aggregatfunktionen zur Verfügung.

Die Transact-SQL-Programmiersprache stellt folgende Aggregatfunktionen zur Verfügung:

| | |
|-----------------|--------|
| AVG | MAX |
| BINARY_CHECKSUM | MIN |
| CHECKSUM | SUM |
| CHECKSUM_AGG | STDEV |
| COUNT | STDEVP |
| COUNT_BIG | VAR |
| GROUPING | VARP |

Die folgenden 5 Aggregatfunktionen sollen in diesem Skript näher betrachtet werden.

- AVG (average) Durchschnitt der (numerischen) Spaltenwerte
- MAX Größter Spaltenwert
- MIN Kleinster Spaltenwert
- SUM Summe der (numerischen) Spaltenwerte
- COUNT Anzahl der Spaltenwerte

Die einzige Aggregatfunktion, die nicht nur auf Spalten angewendet wird, ist die Funktion COUNT(*). Sie zählt die Anzahl der Reihen, die die Bedingungen der Abfrage erfüllen.

Statistische Funktionen werden in der SELECT-Klausel wie Feldnamen verwendet und benutzen – mit einer Ausnahme (COUNT(*)) – Spaltennamen als Argumente. SUM und AVG können nur auf numerische Felder angewendet werden. COUNT, MAX und MIN können mit numerischen sowie mit Zeichenfeldern arbeiten. Werden MIN und MAX bei Zeichenfeldern angewendet, so beziehen sie sich auf deren ASCII-Werte.

Beispiele:

Der nach der alphabetischen Reihenfolge – erste Lieferant.

```
select min(lname)
from lieferant;
```

Größter, kleinster und durchschnittlicher Statuswert aller Lieferanten.

```
select max(status), min(status), avg(status)
from lieferant;
```

Die Prüfsummen der Werte in der Spalte „lname.lieferant“.

```
select checksum(lname)
from lieferant;
```

Die Prüfsummen der Spalten der Artkeltabelle.

```
select checksum(*)
from artikel;
```

Die Prüfsumme der Artkeltabelle.

```
select checksum_agg(checksum(*))
from artikel;
```

Folgende Abfrage liefert einen Fehler:

```
select lnr, max(status)
from lieferant;
```

WARUM? In der SELECT- Liste befindet sich außer der Aggregatfunktion noch eine andere Spalteninformation und die Abfrage enthält keine GROUP BY - Klausel.

Damit ergibt sich für den Optimierer das Problem: Für welche LNR soll er den maximalen Statuswert ermitteln, oder welcher LNR soll er den ermittelten Maximalstatus zuordnen?

6.5.1.2 Spezielle Attribute von COUNT

Die COUNT- Funktion zählt die Werte einer Spalte oder die Zeilen einer Tabelle.

6.5.1.2.1 Aggregatfunktionen mit NULL- Werten

Alle Aggregatfunktionen ignorieren NULL- Werte in Spalten. Eine Ausnahme bildet die COUNT(*)- Funktion. Beim Zählen von Zeilen einer Tabelle ist es unerheblich, ob einzelne Spalten NULL- Werte enthalten. Das kann bei der Anwendung von COUNT() zu unerwarteten Ergebnissen führen, bei den anderen Aggregatfunktionen ist diese Eigenschaft aber von Vorteil.

Beispiel:

1. Sie zählen die Artikel über die Spalte "farbe" der Artikeltabelle. Für diese Spalte sind NULL- Werte erlaubt. In der Tabelle stehen 10 Datensätze davon ist bei zwei Artikeln keine Farbe eingetragen. Sie erwarten im Ergebnis die Zahl 10 aber sie werden nur die Zahl 8 als Ergebnis angezeigt bekommen.
2. Sie ermitteln den durchschnittlichen Statuswert aller Lieferanten. Der Lieferant "L05" hat keinen Statuswert eingetragen da er noch nicht geliefert hat. Die Aggregatfunktion "avg" ignoriert diesen NULL- Wert und ermittelt den richtigen Ergebniswert.

6.5.1.2.2 Einsatz von DISTINCT und ALL

Sollen bei der Ergebnisbildung Mehrfachwerte innerhalb einer Spalte unberücksichtigt bleiben, ist das Schlüsselwort „DISTINCT“ im Funktionsargument anzugeben. Im umgekehrten Fall steht „ALL“ als Standardeinstellung, weshalb sich die explizite Angabe erübrigt.

Beispiele:

Anzahl aller bisherigen Lieferungen, die Lieferanten durchgeführt haben.

| | | |
|-----------------------------------|------|---|
| select count(*) from lieferung | oder | select count(all Inr) from lieferung |
| | oder | select count(lnr) from lieferung |

Anzahl der Lieferanten, die bisher geliefert haben.

```
select count(distinct lnr)
from lieferung;
```

Sinnvoll ist die Anwendung von „DISTINCT“ in der Regel nur bei der COUNT()-Funktion.

Darum dürfte die folgende Abfrage unsinnig sein:

```
select avg(distinct lmenge)
from lieferung;
```

Warum? Gleiche Liefermengen werden eliminiert und bei der Berechnung des Durchschnitts nicht berücksichtigt.

Aggregatfunktionen (außer checksum_agg) dürfen nicht miteinander verschachtelt werden. Jede Abfrage mit einer solchen Verschachtelung bringt deshalb zwingend eine Fehlermeldung hervor.

| | |
|--|---------------------------|
| select count(max(status)) from lieferant; | Diese Abfrage ist falsch! |
|--|---------------------------|

6.5.1.3 Aggregatfunktionen und skalare Ausdrücke

Sie können in statistischen Funktionen auch Argumente anwenden, die aus skalaren Ausdrücken bestehen und sich auf eine oder mehrere Spalten beziehen. (In diesem Fall ist die Verwendung von "DISTINCT" nicht möglich.)

Beispiel:

Nehmen wir an, die Tabelle Lieferung besäße noch eine Spalte „Preis/Stück“, in der die Einzelpreise der in dieser Lieferung gelieferten Artikel gespeichert sind. Um jetzt die Lieferung mit dem größten Gesamtpreis zu finden, könnte man folgende Abfrage schreiben:

```
select max( lmenge * [preis/stück])
from lieferung;
```

6.5.1.4 GROUP BY- Klausel

Definition: Die GROUP BY- Klausel legt die Spalte(n) fest, über die die Gruppenbildung erfolgt.

Die wichtigste Anwendung der GROUP BY- Klausel ist die gruppenbezogene Aggregierung von Werten, also die Anwendung der statistischen Funktionen auf die durch GROUP BY- Klausel gebildeten Gruppen statt auf die gesamte Tabelle.

Alle Spaltennamen in der SELECT- Liste die nicht Parameter der Aggregatfunktion sind, müssen zur Gruppierung verwendet werden. Das heißt, sie müssen in der GROUP BY- Klausel stehen

Teilsyntax nicht ISO Kompatibel:

```
...  
group by [all] spalte [,...] [with {cube | rollup}]
```

Teilsyntax ISO Kompatibel:

```
...  
group by {spalte [,...] | rollup(spalte [,...]) [,...] | cube(spalte [,...]) [,...]  
| grouping sets(spalte [,...] {[ ,rollup()] | [,cube()]})}
```

Die nicht ISO- Kompatible Syntax der GROUP BY- Klausel wird in zukünftigen Versionen von SQL Server so nicht mehr unterstützt, das gilt für die Operatoren „WITH CUBE“ und „WITH ROLLUP“ und „ALL“.

Beispiel:

Gesucht ist die kleinste Liefermenge eines jeden Lieferanten.

```
select lnr, min(lmenga)  
from lieferung  
group by lnr;
```

Ist in der Select-Anweisung eine WHERE- Klausel enthalten, werden alle Zeilen, die diese nicht erfüllen, vor der Gruppenbildung eliminiert.

Beispiel:

Gesucht ist die größte Lieferung eines jeden Lieferanten nach dem 23.07.90.

```
select lnr, max(lmenga)  
from lieferung  
where ldatum > ,23.07.90'  
group by lnr;
```

6.5.1.5 HAVING- Klausel

Definition: Die HAVING- Klausel ist die WHERE- Klausel für Gruppen.
Das heißt, die HAVING- Klausel ist die WHERE- Klausel der GROUP BY- Klausel

Die WHERE- Klausel selektiert Zeilen einer Tabelle, die HAVING- Klausel selektiert Gruppen, die durch die GROUP BY- Klausel definiert sind.

Die Datentypen text, image und ntext können in einer HAVING- Klausel nicht verwendet werden.

In einer WHERE- Klausel dürfen keine Aggregatfunktionen stehen aber in einer HAVING- Klausel dürfen nicht nur Konstanten sondern auch Aggregatfunktionen enthalten sein.

Beispiel:

Gesucht ist der größte Status des jeweiligen Wohnortes von Lieferanten die nicht aus Aachen kommen, wenn der durchschnittliche Statuswert am jeweiligen Ort nicht kleiner als 15 ist.

```
select max(status), lstadt  
from lieferant  
where lstadt <> ,Aachen'  
and avg(status) > 15  
group by lstadt;
```

Diese Abfrage ist falsch!

Richtige Lösung:

```
select max(status), lstadt
from lieferant
where lstadt <> 'Aachen'
group by lstadt
having avg(status) > 15;
```

In der Praxis wird die HAVING- Klausel fast ausschließlich für Vergleiche mit Aggregatfunktionen verwendet.

Für die Argumente von HAVING gilt: sie dürfen nur einen Wert pro auszugebender Gruppe als Ergebnis liefern.

Darum ist folgender Befehl nicht erlaubt:

```
select lnr, max(lmenge)
from lieferung
group by lnr
having anr = 'A02';
```

besser ist deshalb:

```
select lnr, max(lmenge)
from lieferung
where anr = 'A02'
group by lnr;
```

Es gibt keine Beschränkung bezüglich der Anzahl von Bedingungen in den Suchbedingungen einer HAVING- Klausel.

```
select sum(lmenge), lnr
from lieferung
group by lnr
having min(lmenge) > 200 and avg(lmenge) < 1300;
```

Eine HAVING- Klausel kann auch ohne GROUP BY- Klausel verwendet werden. Es darf in der SELECT- Liste kein Spaltenname stehen, da er sonst in einer GROUP BY- Klausel stehen muss.

Darum ist folgende Abfrage richtig:

Berechne nur dann die Summe der Liefermenge wenn die Anzahl der Lieferungen zehn übersteigt.

```
select sum(lmenge)
from lieferung
having count(lnr) > 10;
```

6.5.1.6 Zusammenfassung für Aggregatfunktionen

Eine SELECT- Anweisung, die GROUP BY- und HAVING- Klausel enthält, wird in folgender Reihenfolge bearbeitet:

1. Durch die WHERE- Klausel werden alle Reihen ausgeschlossen, die nicht die Suchbedingungen der WHERE- Klausel erfüllen.
2. Durch die GROUP BY- Klausel werden die Reihen gruppiert, die die mit der WHERE- Klausel festgelegten Suchbedingungen erfüllen, wobei für jeden eindeutigen Wert in der GROUP BY- Klausel eine Gruppe erstellt wird. Durch Weglassen der GROUP BY- Klausel wird für die ganze Tabelle nur eine Gruppe erstellt.
3. Durch die HAVING- Klausel werden die Gruppen, die die Suchbedingungen der HAVING- Klausel nicht erfüllen, ausgeschlossen. Wenn die Abfrage eine GROUP BY- Klausel enthält, werden durch die HAVING- Klausel einzelne Gruppen aus dem Abfrageergebnis ausgeschlossen.

Die in der SELECT- Liste einer HAVING- Klausel angegebenen Aggregatfunktionen berechnen für jede definierte Gruppe einen Zusammenfassungswert.

Durch die Kombination von GROUP BY- Klausel, HAVING- Klausel und Aggregatfunktion wird für jede Datengruppe eine Reihe und ein Zusammenfassungswert zurückgegeben. Für die

Verwendung von GROUP BY und HAVING gelten aus ANSI- Kompatibilitätsgründen einige Einschränkungen:

1. Spalten in einer SELECT- Liste, auf die keine Aggregatfunktion angewendet wird, müssen in der GROUP BY- Klausel definiert sein.
2. Die in einer HAVING- Klausel aufgeführten Spalten dürfen nur einen Wert zurückgeben.
3. Eine Abfrage, die eine HAVING- Klausel verwendet, sollte ebenfalls eine GROUP BY- Klausel enthalten. Ohne GROUP BY- Klausel werden alle nicht durch die WHERE- Klausel ausgeschlossenen Reihen als einzige Gruppe bearbeitet.

6.5.2 Berechnen vollständiger Zusammenfassungswerte für eine Tabelle

Die GROUP BY- Klausel kann zusammen mit den Funktionen ROLLUP und CUBE verwendet werden, um zusätzliche Reihen (Datensätze) mit Zusammenfassungswerten für sämtliche in der SELECT- Klausel definierten Spalten zu erhalten. Dadurch können übergreifende Informationen ermittelt werden, ohne dafür zusätzliche Abfragen zu erstellen.

6.5.2.1 ROLLUP- und CUBE Funktionen

Die Funktionen ROLLUP() und CUBE() werden zusammen mit Aggregatfunktionen verwendet, um in Ergebnismengen zusätzliche Reihen zu erzeugen.

Die in der SELECT- Liste angegebene Aggregatfunktion wird auf diese Reihen angewendet, um Zusammenfassungswerte für zusätzliche Superaggregatzeilen zu erzeugen.

Die Funktionen CUBE() und ROLLUP() unterscheiden sich wie folgt:

- **ROLLUP()** generiert einfache GROUP BY- Aggregatzeilen, zusammen mit Zwischensummen- und Superaggregatzeilen sowie auch einer Gesamtergebnissezeile. Die Anzahl der zurückgegebenen Gruppierungen entspricht der Anzahl der Ausdrücke in der <Liste zusammengesetzter Elemente> plus einer Gruppierung.

```
select a, b, c, sum(spalte)
from tabelle
group by rollup(a,b,c);
```

Erzeugt eine Zeile mit einer Zwischensumme für jede eindeutige Kombination der Werte von (a,b,c), (a,b) und (a). Außerdem wird eine Gesamtergebnissezeile berechnet.

- **CUBE()** generiert einfache GROUP BY- Aggregatzeilen, die ROLLUP- Superaggregatzeilen und Kreuztabellenzeilen. CUBE gibt eine Gruppierung für alle Permutationen der Ausdrücke in der <Liste zusammengesetzter Elemente> aus. Die Anzahl der generierten Gruppierungen entspricht 2^n , wobei n die Anzahl der Ausdrücke in der <Liste zusammengesetzter Elemente> ist.

```
select a, b, c, sum(spalte)
from tabelle
group by cube(a,b,c);
```

Erzeugt für jede eindeutige Kombination der Werte von (a,b,c), (a,b), (a,c), (b,c), (a), (b) und (c) wird eine Zeile generiert, zusammen mit einer Zwischensumme für jede Zeile sowie einer Gesamtergebnissezeile.

Hinweis: Verwenden Sie die Operatoren „WITH ROLLUP“ und „WITH CUBE“ und „ALL“ nicht mehr in den Abfragen mit der GROUP BY- Klausel. Sie werden in zukünftigen Versionen von SQL Server nicht mehr implementiert. Sie dienen zurzeit lediglich aus Gründen der Abwärtskompatibilität.

So enthält die folgende Abfrage mit der ROLLUP- Funktion nachfolgendes Resultset:

```
select lname, DATENAME(mm,ldatum) as 'Monat', DATEPART(yyyy,ldatum) as 'Jahr',
sum(lmenge) as ,Gesamtliefermenge'
from lieferant a join lieferung b on a.lnr = b.lnr
where a.lnr ='L01'
group by rollup(lname, DATENAME(mm,ldatum), DATEPART(yyyy,ldatum))
order by lname, DATENAME(mm,ldatum), DATEPART(yyyy,ldatum);
```

| Iname | Monat | Jahr | Gesamtliefermenge |
|---------|--------|------|-------------------|
| NULL | NULL | NULL | 1300 |
| Schmidt | NULL | NULL | 1300 |
| Schmidt | August | NULL | 100 |
| Schmidt | August | 1990 | 100 |
| Schmidt | Januar | NULL | 400 |
| Schmidt | Januar | 1990 | 400 |
| Schmidt | Juli | NULL | 500 |
| Schmidt | Juli | 1990 | 500 |
| Schmidt | Mai | NULL | 300 |
| Schmidt | Mai | 1990 | 300 |

Setzen wir die Cube- Funktion ein, erhalten wir folgendes Ergebnis.

```
select lname, DATENAME(mm,ldatum) as 'Monat', DATEPART(yyyy,ldatum) as 'Jahr',
sum(lmenge) as ,Gesamtliefermenge'
from lieferant a join lieferung b on a.lnr = b.lnr
where a.lnr = 'L01'
group by cube(lname, DATENAME(mm,ldatum), DATEPART(yyyy,ldatum))
order by lname, DATENAME(mm,ldatum), DATEPART(yyyy,ldatum);
```

| Iname | Monat | Jahr | Gesamtliefermenge |
|---------|--------|------|-------------------|
| NULL | NULL | NULL | 1300 |
| NULL | NULL | 1990 | 1300 |
| NULL | August | NULL | 100 |
| NULL | August | 1990 | 100 |
| NULL | Januar | NULL | 400 |
| NULL | Januar | 1990 | 400 |
| NULL | Juli | NULL | 500 |
| NULL | Juli | 1990 | 500 |
| NULL | Mai | NULL | 300 |
| NULL | Mai | 1990 | 300 |
| Schmidt | NULL | NULL | 1300 |
| Schmidt | NULL | 1990 | 1300 |
| Schmidt | August | NULL | 100 |
| Schmidt | August | 1990 | 100 |
| Schmidt | Januar | NULL | 400 |
| Schmidt | Januar | 1990 | 400 |
| Schmidt | Juli | NULL | 500 |
| Schmidt | Juli | 1990 | 500 |
| Schmidt | Mai | NULL | 300 |
| Schmidt | Mai | 1990 | 300 |

Größenbeschränkung:

- Für eine GROUP BY- Klausel die ROLLUP oder BUBE verwendet, gilt die Maximalzahl von 32 Ausdrücken und es können maximal 4096 Gruppierungssätze generiert werden.

6.5.2.2 GROUPING SETS Funktion

Ist ein Ausdruck, der es möglich macht, mehrere Gruppierungsebenen in einer Anweisung zu formulieren.

Sie gibt mehrere Gruppierungen der Daten in einer Abfrage an. Es werden statt der vollständigen Menge der Aggregationen, die von CUBE oder ROLLUP generiert werden, nur die angegebenen Gruppen aggregiert. Die Ergebnisse entsprechen denen von UNION ALL für die angegebenen Gruppen. GROUPING SETS kann ein einzelnes Element oder eine Liste von Elementen enthalten. GROUPING SETS kann Gruppierungen angeben, die jenen entsprechen, die von ROLLUP oder CUBE zurückgegeben werden. Die Gruppierungssatzelementliste kann auch die ROLLUP- oder CUBE- Funktion enthalten.

Die leere Gruppe GROUP BY GROUPING SETS() generiert eine Gesamtsumme.

Beispiel:

```
select lname, DATENAME(mm,ldatum) as 'Monat', DATEPART(yyyy,ldatum) as 'Jahr',
sum(lmenge) as 'Gesamtliefermenge'
from lieferant a join lieferung b on a.lnr = b.lnr
where a.lnr = 'L01'
group by grouping sets(lname, (DATENAME(mm,ldatum), DATEPART(yyyy,ldatum)))
order by lname, DATENAME(mm,ldatum), DATEPART(yyyy,ldatum);
```

| Iname | Monat | Jahr | Gesamtliefermenge |
|---------|--------|------|-------------------|
| NULL | August | 1990 | 100 |
| NULL | Januar | 1990 | 400 |
| NULL | Juli | 1990 | 500 |
| NULL | Mai | 1990 | 300 |
| Schmidt | NULL | NULL | 1300 |

Syntaxeinschränkungen:

- GROUPING SETS sind in einer GROUP BY Klausel nicht zugelassen, es sei denn, sie sind Teil einer GROUPING SETS- Liste. Zum Beispiel ist GROUP BY sp1, (sp2, sp3) nicht erlaubt, aber GROUP BY GROUPING SETS(sp1, (sp2, sp3)) ist erlaubt.
- GROUPING SETS sind in GROUPING SETS nicht zulässig, sie können also nicht verschachtelt werden.
- Für eine GROUP BY- Klausel die GRIOUING SETS verwendet, gilt die Maximalzahl von 32 Ausdrücken und es können maximal 4096 Gruppierungssätze generiert werden.

6.5.2.3 GROUPING- Funktion

In den oben stehenden Beispielen wurde die GROUPING- Funktion eingesetzt. Verwendet wird die Funktion entweder mit dem ROLLUP- oder mit dem CUBE- Operator. Sie liefert im Resultset den Wert 1 oder 0.

- 1 steht für einen Zusammenfassungswert in der vorstehenden Spalte.
- 0 steht für Detailwerte in der vorangegangenen Spalte.

Mit Hilfe dieser Funktion kann festgestellt werden, ob die im Resultset angegebenen NULL-Werte Spaltenwerte in den Datenbanktabellen sind oder vom ROLLUP- oder CUBE- Operator erzeugt wurden.

Beispiel:

```
select astadt as 'Lagerort', grouping(stadt),
farbe as 'Artikelfarbe', grouping(farbe), sum(amenge) as 'Gesamtlagermenge'
from artikel
group by astadt, farbe WITH CUBE;
```

| Lagerort | | Artikelfarbe | | Gesamtlagermenge |
|--------------|---|--------------|---|------------------|
| Hamburg | 0 | rot | 0 | 2200 |
| Hamburg | 0 | NULL | 1 | 2200 |
| Ludwigshafen | 0 | blau | 0 | 1300 |
| Ludwigshafen | 0 | grün | 0 | 1200 |
| Ludwigshafen | 0 | NULL | 1 | 2500 |
| Mannheim | 0 | blau | 0 | 400 |
| Mannheim | 0 | NULL | 1 | 400 |
| NULL | 1 | NULL | 1 | 5100 |
| NULL | 1 | blau | 0 | 1700 |
| NULL | 1 | grün | 0 | 1200 |
| NULL | 1 | rot | 0 | 2200 |

Um das Ergebnis anschaulicher zu machen, können wir auch die GROUPING- Funktion und den CASE Operator einsetzen.

```
select case
when (grouping(stadt) = 1) then 'Alle Lagerorte'
else isnull(stadt, 'Unbekannt')
end as 'Lagerort',
case when (grouping(farbe) = 1) then 'Alle Farben'
else isnull(farbe, 'Unbekannt')
end as 'Artikelfarbe',
sum(amenge) as 'Gesamtlagermenge'
from artikel
group by rollup(stadt, farbe);
```

| Lagerort | Artikelfarbe | Gesamtlagermenge |
|----------------|--------------|------------------|
| Hamburg | rot | 2200 |
| Hamburg | Alle Farben | 2200 |
| Ludwigshafen | blau | 1300 |
| Ludwigshafen | grün | 1200 |
| Ludwigshafen | Alle Farben | 2500 |
| Mannheim | blau | 400 |
| Mannheim | Alle Farben | 400 |
| Alle Lagerorte | Alle Farben | 5100 |
| Alle Lagerorte | blau | 1700 |
| Alle Lagerorte | grün | 1200 |
| Alle Lagerorte | rot | 2200 |

Hinweis:

- Die Funktion wird wie ein Ausdruck in der SELECT- Liste verwendet.
- Die Funktion gibt den Wert 1 für jede Zeile zurück, die eine ROLLUP- oder CUBE- Zusammenfassung darstellt.
- Die Funktion gibt den Wert 0 für jede Zeile zurück, nicht durch ROLLUP/CUBE gebildet wurde.
- Die Funktion kann nur für Spalten angewendet werden, die in der GROUP BY- Klausel aufgeführt sind.

6.5.2.4 GROUPING_ID- Funktion

Diese Funktion erweitert die GROUPING- Funktion indem sie die Ebene der Gruppierung berechnet. Sie kann nur in der SELECT-, HAVING- oder ORDER BY- Klausel verwendet werden wenn die GROUP BY- Klausel angegeben wird.

Der Spaltenausdruck der GROUPING_ID- Funktion muss genau mit dem Ausdruck der GROUP BY- Liste übereinstimmen.

```
select case
    when grouping_id(aname) = 0 then aname
    when grouping_id(aname) = 1 then 'Artikel- Total'
    else 'Unbekannt' end as [Artikel],
    case
        when grouping_id(aname,lname) = 0 then lname
        when grouping_id(aname,lname) = 1 then aname + '-Total'
        when grouping_id(aname,lname) = 3 then 'Lieferungen-Total'
        else 'Unbekannt' end as [Lieferanten],
        count(b.anr) as [Lieferanzahl]
from lieferant a join lieferung b on a.lnr = b.lnr join artikel c on b.anr = c.anr
group by rollup(aname, lname);
```

| Artikel | Lieferanten | Lieferanzahl |
|----------------|-------------------|--------------|
| Bolzen | Blank | 3 |
| Bolzen | Clark | 1 |
| Bolzen | Jonas | 2 |
| Bolzen | Schmidt | 1 |
| Bolzen | Schulze | 1 |
| Bolzen | Bolzen-Total | 8 |
| Mutter | Blank | 1 |
| Mutter | Jonas | 1 |
| Mutter | Schmidt | 1 |
| Mutter | Mutter-Total | 3 |
| Nockenwelle | Clark | 1 |
| Nockenwelle | Schmidt | 1 |
| Nockenwelle | Nockenwelle-Total | 2 |
| Schraube | Clark | 1 |
| Schraube | Schmidt | 2 |
| Schraube | Schraube-Total | 3 |
| Zahnrad | Schmidt | 1 |
| Zahnrad | Zahnrad-Total | 1 |
| Artikel- Total | Lieferungen-Total | 17 |

6.5.3 Rangfolgen für gruppierte Daten

Bezeichnet das Nummerieren von Zeilen in einem Satz von Daten anhand eines Rangfolgentyps und eines Satzes von Qualifizierern. Rangfolgen können auch auf eine Datenpartition einer partitionierten Tabelle angewendet werden.

Rangfolgefunktionen geben für jede Partitionszeile einen Rangfolgewert zurück. Je nachdem welche Funktion verwendet wird, erhalten einige Zeilen möglicherweise dieselben Werte wie andere Zeilen.

SQL Server stellt vier Rangfolgefunktionen bereit. Sie liefern jeweils eine andere Rangfolgenausgabe.

- RANK
- DENSE_RANK
- NTILE
- ROW_NUMBER

Bezogen auf die Datenpartitionen (Datenfenster), verhalten sich Rangfolgefunktionen ähnlich wie Aggregatfunktionen.

Sie geben je nach Typ einen Rangfolgewert für jede Zeile in einer Partition zurück, und zwar basierend auf ihrer Beziehung zu anderen Zeilen und deren Werten der partitionierten Spalte.

Rangfolgefunktionen sind nicht deterministisch.

6.5.3.1 Rangfolgefunktion RANK

Gibt den Rang jeder Zeile innerhalb der Partition eines Resultsets zurück. Der Rang einer Zeile ergibt sich, indem 1 zur Anzahl von Rängen vor der fraglichen Zeile addiert wird.

Syntax:

RANK() OVER ([< partition_by_klausel >] < order_by_klausel >)

< partition_by_klausel >

Teilt das von der FROM- Klausel erzeugte Ergebnis in Partitionen, auf die die RANK-Funktion angewendet wird.

< order_by_klausel >

Bestimmt die Reihenfolge, in der die RANK- Werte auf die Zeilen in einer Partition sortiert werden.

Sind zwei oder mehr Zeilen gleichwertig, erhält jede gleichwertige Zeile denselben Rang. Wenn zum Beispiel zwei Lieferanten denselben Wert in der Liefermenge aufweisen, erhalten beide den Rang 1. Der Lieferant mit der nächsthohen Liefermenge erhält den Rang 3, da es bereits zwei Zeilen mit einem höheren Rang gibt.

Beispiel:

Die Rangfolge der Lieferanten anhand der Gesamtliefermengen, mit Lücken.

```
select a.Innr, Iname,
rank() over(order by sum(lmenge) desc) as [Rang],
sum(lmenge) as [Gesamtliefermenge]
from lieferant a join lieferung b on a.Innr = b.Innr
group by a.Innr, Iname;
```

| Innr | Iname | Rang | Gesamtliefermenge |
|------|---------|-------|-------------------|
| --- | ----- | ----- | ----- |
| L01 | Schmidt | 1 | 1300 |
| L03 | Blank | 2 | 1100 |
| L04 | Clark | 3 | 900 |
| L02 | Jonas | 3 | 900 |
| L06 | Schulze | 5 | 200 |

6.5.3.2 Rangfolgefunktion DENSE_RANK

Gibt den Rang von Datensätzen innerhalb der Partition ohne Lücken in der Rangfolge an. DENSE_RANK weist dieselbe Funktionsweise wie RANK auf allerdings ohne Lücken in der Rangfolgesequenz.

Syntax:

DENSE_RANK() OVER ([< partition_by_klausel >] < order_by_klausel >)

< partition_by_klausel >

Teilt das von der FROM- Klausel erzeugte Ergebnis in Partitionen, auf die die RANK- Funktion angewendet wird.

< order_by_klausel >

Bestimmt die Reihenfolge, in der die RANK- Werte auf die Zeilen in einer Partition sortiert werden.

Beispiel:

Ranking der Lieferungen anhand der Gesamtliefermengen, ohne Lücken.

```
select a.Innr, Iname,
dense_rank() over(order by sum(lmenge) desc) as [Rang],
sum(lmenge) as [Gesamtliefermenge]
from lieferant a join lieferung b on a.Innr = b.Innr
group by a.Innr, Iname;
```

| Innr | Iname | Rang | Gesamtliefermenge |
|------|---------|-------|-------------------|
| --- | ----- | ----- | ----- |
| L01 | Schmidt | 1 | 1300 |
| L03 | Blank | 2 | 1100 |
| L04 | Clark | 3 | 900 |
| L02 | Jonas | 3 | 900 |
| L06 | Schulze | 4 | 200 |

Wenn zwei oder mehr Zeilen den gleichen Rang in einer Partition haben, erhalten alle diese Zeilen denselben Rang. Wenn zum Beispiel zwei Lieferanten die gleiche Anzahl an Lieferungen besitzen und die besten damit sind, erhalten sie den Rang 1. Der Lieferant mit der nächsthöheren Anzahl von Lieferungen erhält den Rang 2. Dadurch gibt es keine Lücken in der Rangfolge.

6.5.3.3 Rangfolgefunktion NTILE

Verteilt die Zeilen in einer sortierten Partition in eine angegebene Anzahl von Gruppen. Die Gruppen sind nummeriert, dabei wird mit eins begonnen. Für jede Zeile gibt NTILE die Nummer der Gruppe zurück zu der die Zeile gehört.

NTILE wird häufig beim Data Warehousing verwendet.

Syntax:

NTILE(int_wert) OVER ([< partition_by_klausel >] < order_by_klausel >)

< int_wert >

Positive ganze Zahl, die die Anzahl der Gruppen angibt, in die jede Partition unterteilt werden kann.

< partition_by_klausel >

Teilt das von der FROM- Klausel erzeugte Ergebnis in Partitionen, auf die die RANK- Funktion angewendet wird.

< order_by_klausel >

Bestimmt die Reihenfolge, in der die RANK- Werte auf die Zeilen in einer Partition sortiert werden.

Beispiel:

Bildung von Liefermengenbereichen und Zuordnung der jeweiligen gelieferten Artikel entsprechend der jeweiligen Liefermenge.

```
select ntile(4) over(partition by aname order by lmenge)
as mengenkategorie,aname,lmenge
from artikel a join lieferung b on a.anr = b.anr
order by aname;
```

| Mengenkategorie | aname | lmenge |
|-----------------|-------------|--------|
| 1 | Bolzen | 200 |
| 2 | Bolzen | 200 |
| 3 | Bolzen | 200 |
| 4 | Bolzen | 400 |
| 1 | Mutter | 300 |
| 2 | Mutter | 300 |
| 1 | Nockenwelle | 100 |
| 2 | Nockenwelle | 400 |
| 1 | Schraube | 200 |
| 2 | Schraube | 300 |
| 3 | Schraube | 400 |
| 1 | Zahnrad | 100 |

Falls die Anzahl der Zeilen einer Partition nicht durch den Integerwert geteilt werden kann, führt dies zu Gruppen in zwei unterschiedlichen Größen, die sich um ein Mitglied unterscheiden.

6.5.3.4 Rangfolgefunktion ROW_NUMBER

Gibt die fortlaufende Nummer einer Zeile innerhalb einer Partition zurück. Sie kann mit oder ohne PARTITION BY- Klausel verwendet werden.

Syntax:

ROW_NUMBER() OVER ([< partition_by_klausel >] < order_by_klausel >)

< partition_by_klausel >

Teilt das von der FROM- Klausel erzeugte Ergebnis in Partitionen, auf die die RANK- Funktion angewendet wird.

< order_by_klausel >

Bestimmt die Reihenfolge, in der die RANK- Werte auf die Zeilen in einer Partition sortiert werden.

Beispiel:

Zeilenummern für jeden Lieferanten

```
select row_number() over(order by lnr desc) as 'row number', lname, lstadt
from lieferant;
```

| Row Number | Iname | Istadt |
|------------|---------|--------------|
| 1 | Adam | Aachen |
| 2 | Clark | Hamburg |
| 3 | Blank | Ludwigshafen |
| 4 | Jonas | Ludwigshafen |
| 5 | Schmidt | Hamburg |

Die ORDER BY- Klausel bestimmt die Reihenfolge, in der den Zeilen eine eindeutige ROW_NUMBER innerhalb der angegebenen Partition zugewiesen wird.

Zusammenfassung:

| | RANK | DENSE_RANK | NTILE | ROW_NUMBER |
|---|------|------------|-------|------------|
| Datensätze werden auf gleichgroße Gruppen verteilt | | | | |
| Datensätzen mit gleichem Wert wird der gleiche Rang zugewiesen | | | | |
| Fortlaufende Nummerierung von Datensätzen in einer Partition | | | | |

6.6 SELECT INTO- Klausel

Die SELECT INTO- Anweisung erstellt eine neue Tabelle und füllt sie mit dem Resultset der SELECT- Anweisung auf. Die Struktur der neuen Tabelle wird durch die Attribute der Ausdrücke in der SELECT- Liste definiert.

Mithilfe von SELECT INTO können Daten aus mehreren Tabellen oder Sichten in einer Tabelle kombiniert werden.

Das Ergebnis kann in temporäre und auch in permanente Tabellen gespeichert werden.

6.6.1 Temporäre Tabellen

Es können sowohl lokale als auch globale temporäre Tabellen erstellt werden. Lokale temporäre Tabellen sind nur während der aktuellen Sitzung sichtbar, globale temporäre Tabellen sind von allen Sitzungen aus sichtbar. Temporäre Tabellen können von jedem Datenbankbenutzer erstellt werden.

| | |
|-----------------------------|--------------|
| Lokale temporäre Tabellen: | #table_name |
| Globale temporäre Tabellen: | ##table_name |

Beispiel:

```
select lnr, lname
into #gute_lieferanten
from lieferant
where status >= 30;
```

6.6.2 Temporäre Tabellen durch Tabellenvariable

Seit SQL Server 2005 kann der Datentyp „table“ nun auch für die Variablen Deklaration verwendet werden. Diese nennt man Tabellenvariable die wie normale temporäre Tabellen auch, Zwischenergebnisse abspeichern können.

Die Datensatzmenge kann beliebig groß sein, da sie aber im Arbeitsspeicher gehalten werden sollte hier Vorsicht geboten sein.

Beispiel:

Die Nummern, Namen und das Lieferdatum der Hamburger Lieferanten sollen in eine Tabellenvariable gespeichert werden.

```
declare @lief_lief table (lnr char(3), lname varchar(40), ldatum datetime)

insert into @lief_lief
select a.lnr, lname, ldatum
from lieferant a, lieferung b
where a.lnr = b.lnr
and lstadt = 'Hamburg';

select * from @lief_lief;
```

6.6.3 Permanente Tabellen

Permanente Tabellen die mit der SELECT INTO- Anweisung erstellt werden, werden in der aktuellen Datenbank abgelegt.

Ersteller müssen über die CREATE TABLE- Berechtigung zum Erstellen der Tabellen und die ALTER- Berechtigung für das jeweilige Schema besitzen oder Mitglied einer berechtigten Serverrolle bzw. Datenbankrolle sein.

Beispiel:

```
select Inr, Iname
into gute_lieferanten
from lieferant
where status >= 30;
```

6.7 Mengenoperator UNION

Kombiniert die Ergebnisse zweier oder mehrerer Abfragen zu einem Resultset, das aus allen Zeilen besteht, die zu allen Abfragen der UNION gehören.

Im Gegensatz zu Verknüpfungen, mit denen Zeilen der Basistabellen zu einzelnen Zeilen kombiniert werden, werden mit UNION die Zeilen aus den Resultsets nacheinander aufgeführt.

Häufig wird UNION verwendet, um zuvor partitionierte Tabellen wieder zusammenzuführen.

Syntax:

```
select  Select- Liste [into- Klausel]
        [FROM- Klausel]
        [WHERE- Klausel]
        [GROUP BY- Klausel]
        [HAVING- Klausel]
UNION [ALL]
select  Select- Liste
        [FROM- Klausel]
        [WHERE- Klausel]
        [GROUP BY- Klausel]
        [HAVING- Klausel]
UNION [ALL]
[...]
[Oder by- Klausel]
```

Für das Verwenden von UNION- Operatoren gelten folgende Richtlinien:

- Alle Auswahllisten in den mit UNION kombinierten Anweisungen müssen über dieselbe Anzahl von Ausdrücken (Spaltennamen, arithmetische Ausdrücke, Aggregatfunktionen etc.) verfügen.
- Die einander entsprechenden Spalten in den mit UNION kombinierten Resultsets bzw die in einer einzelnen Abfrage verwendeten Spalten einer Teilmenge müssen denselben Datentyp aufweisen. Ist dies nicht der Fall, muss eine Datenkonvertierung angegeben werden.
- Einander entsprechende Spalten in der SELECT- Liste der einzelnen Anweisungen, die mit UNION kombiniert werden sollen, müssen in derselben Reihenfolge vorliegen.
- Die Spaltennamen der Tabelle, die sich aus der UNION- Operation ergibt, werden aus der ersten Einzelabfrage der UNION- Anweisung übernommen.
- Union entfernt doppelte Reihen aus dem Ergebnis.

Beispiel:

Die Nummern, Namen und Wohn- bzw. Lagerorte aller Lieferanten und Artikel aus Hamburg und Ludwigshafen.

```
select lnr as 'Nummern', lname as 'Namen', istadt as 'Orte'
from lieferant
where istadt in ('Hamburg', 'Ludwigshafen')
union
select anr, aname, astadt
from artikel
where astadt in ('Hamburg', 'Ludwigshafen')
order by 'Nummern' DESC;
```

6.8 Except und Intersect

Die Operatoren ermöglichen es, die Ergebnisse von zwei SELECT- Anweisungen miteinander zu vergleichen, und unterschiedliche Werte zurückzugeben zu lassen.

Der **EXCEPT**- Operator gibt sämtliche unterschiedlichen Werte der Abfrage links vom Operator zurück, die nicht von der Abfrage rechts zurückgegeben werden.

Der **INTERSECT**- Operator gibt sämtliche unterschiedlichen Werte zurück, die von der Abfrage links und rechts vom Operator zurückgegeben werden.

Die mithilfe von EXCEPT oder INTERSECT verglichenen Abfragen müssen alle dieselbe Struktur aufweisen. Sie müssen über dieselbe Anzahl an Spalten verfügen, und die einander entsprechenden Ergebnisspalten müssen kompatible Datentypen aufweisen.

Syntax:

```
{ <sql_abfrage> | ( <abfrage_ausdruck> ) }
{ EXCEPT | INTERSECT }
{ <sql_abfrage> | ( <abfrage_ausdruck> ) }
```

6.8.1 Except Operator

Gibt alle unterschiedlichen Werte aus der linken Abfrage zurück die nicht in der rechten Abfrage gefunden werden.

Beispiel:

Die Abfrage liefert die Wohnorte von Lieferanten die kein Lagerort von Artikeln sind.

```
select istadt from lieferant
except
select astadt from artikel;
```

6.8.2 Intersect Operator

Gibt alle Werte der linken und der rechten Abfrage zurück, die in beiden Abfragen ermittelt werden. Es ist der Querschnitt von zwei Ergebnismengen.

Beispiel:

Die Abfrage liefert die Wohnorte der Lieferanten die auch Lagerorte von Artikel sind

```
select istadt from lieferant
intersect
select astadt from artikel;
```

6.9 Die TABLESAMPLE- Klausel

Mit der TABLESAMPLE- Klausel wird die Anzahl der Zeilen eingeschränkt, die aus einer Tabelle wiedergegeben werden. Die Anzahl der Datensätze in der Ergebnismenge entspricht annähernd dem angegebenen Wert.

Teilsyntax:

```
TABLESAMPLE [SYSTEM] (sample_number [ PERCENT | ROWS ] )
```

Verwenden Sie die Klausel, wenn nur eine Stichprobe des Ergebnisses aus einer Tabelle zurückgegeben werden muss, aus der ansonsten mehr Zeilen zurückgegeben würden, als es nützlich wäre. Sie können die Klausel für jede Tabelle anwenden, die in der FROM- Klausel einer Abfrage angegeben wird. Die Klausel kann nicht in der Definition einer Sicht oder einer Inlinefunktion mit Tabellenrückgabe angegeben werden.

Beispiel:

```
select *
from lieferung tablesample ( 5 ROWS);
```

6.10 Die Operatoren PIVOT und UNPIVOT

Die relationalen Operatoren PIVOT und UNPIVOT, pivotieren einen Tabellenwertausdruck in eine andere Tabelle.

6.10.1 PIVOT

Der PIVOT- Operator setzt einen Tabellenwertausdruck um, indem er die eindeutigen Werte einer Spalte des Ausdrucks in mehrere Spalten der Ausgabe versetzt und dabei gegebenenfalls Aggregationen für verbliebene Spaltenwerte vornimmt, die in der endgültigen Ausgabe erwünscht sind. Das bedeutet die distinkten Werte einer Spalte werden zu Spaltenköpfen einer Ergebnis- Kreuztabelle generiert.

Beispiel:

Anzahl von Lieferungen die jeder Lieferant ausgeführt hat Die Lieferantennummern sollen als Spaltenüberschriften im Ergebnis dargestellt werden.

```
select *
from (select 'Anzahl' as 'Type',lnr, anr from lieferung) as a
pivot(count(anr) for lnr in([L01],[L02],[L03],[L04])) as pvt;
```

| Type | L01 | L02 | L03 | L04 |
|--------|-----|-----|-----|-----|
| Anzahl | 6 | 2 | 1 | 3 |

oder:

Wie viel hat jeder Lieferant von jedem jeden Artikel geliefert (mit Anzeige von anr)?

```
select *
from (select lnr, anr, lmenge from lieferung) as a
pivot(sum(lmenge) for lnr in([L01],[L02],[L03],[L04])) as pvt;
```

| anr | L01 | L02 | L03 | L04 |
|-----|-----|------|------|------|
| A01 | 300 | 300 | NULL | NULL |
| A02 | 200 | 400 | 200 | 200 |
| A03 | 400 | NULL | NULL | NULL |
| A04 | 200 | NULL | NULL | 300 |
| A05 | 100 | NULL | NULL | 400 |
| A06 | 100 | NULL | NULL | NULL |

oder:

Wie ist die Gesamtliefermenge die jeder Lieferant geliefert hat (ohne Anzeige von anr)?

```
select *
from (select lnr, lmenge from lieferung) as a
pivot(sum(lmenge) for lnr in([L01],[L02],[L03],[L04])) as pvt;
```

| L01 | L02 | L03 | L04 |
|------|-----|-----|-----|
| 1300 | 700 | 200 | 900 |

6.10.2 UNPIVOT

Der UNPIVOT-Operator führt den umgekehrten Vorgang aus, d.h. er setzt Spalten eines Tabellenwertausdrucks in Spaltenwerte zurück

Beispiel:

Das Ergebnis der letzten oben stehenden Abfrage soll wieder als normalisierte relationale Tabelle dargestellt werden

```
select lnr, lmenge
from (select *
       from (select lnr, lmenge from lieferung) as a
             pivot(sum(lmenge) for lnr in([L01],[L02],[L03],[L04])) as pvt) as p
unpivot(lmenge for lnr in([L01],[L02],[L03],[L04])) as unpvt;
```

| Inr | lmenge |
|-----|--------|
| L01 | 1300 |
| L02 | 700 |
| L03 | 200 |
| L04 | 900 |

6.11 Unterabfragen

Unter einer Unterabfrage versteht man eine Abfrage innerhalb einer Abfrage. Den ersten SELECT- Befehl nennt man die äußere Abfrage und die weiteren SELECT- Befehle nennt man innere Abfrage.

6.11.1 Einfache Unterabfragen

Das Prinzip einer einfachen Unterabfrage besteht darin, dass das Ergebnis der inneren Abfrage den Vergleichswert für die WHERE- Klausel der äußeren Abfrage liefert.

Die innere Abfrage kann genau einen Wert oder auch mehrere Werte für genau eine Spalte liefern.

Bei diesem Typ von Abfragen werden immer zuerst die innere und danach die äußere Abfrage ausgeführt. Einfache Unterabfragen können verschachtelt werden.

6.11.1.1 Einfache Unterabfragen mit Vergleichsoperatoren

Bei diesen Abfragen darf die innere Abfrage immer nur einen Wert an die äußere Abfrage zurückliefern.

Beispiel:

Gesucht sind die Lieferanten, die in derselben Stadt wohnen, in der auch Artikel 'A02' lagert.

```
select *
from lieferant
where lstadt = (select astadt
                  from artikel
                  where anr = 'A02');
```

Gesucht sind die Lieferanten, deren Status über dem durchschnittlichen Status aller Lieferanten liegt.

```
select *
from lieferant
where status > (select avg(status)
                  from lieferant);
```

6.11.1.2 Einfache Unterabfragen mit IN- Operatoren

Es kommt vor, dass eine innere Abfrage nicht genau einen Wert als Ergebnis liefert. Der Vergleich mit der äußeren Abfrage muss dann mit den Mengenoperator IN stattfinden.

Beispiel:

Gesucht sind alle Angaben über Lieferanten, die bereits geliefert haben.

```
select distinct *
from lieferant
where lnr in (select lnr
                  from lieferung);
```

6.11.1.3 Die Operatoren ANY, SOME und ALL

Vergleichsoperatoren, die eine Unterabfrage einleiten, können mit den Schlüsselwörtern ALL oder ANY geändert werden (SOME ist eine SQL-92-Entsprechung für ANY). Diese Operatoren werden mit den Vergleichsoperatoren (=, <, >, ...) eingesetzt.

Mit einem geänderten Vergleichsoperator eingeleitete Unterabfragen geben eine Liste aus null oder mehr Werten zurück und können eine GROUP BY- oder HAVING- Klausel einschließen. Diese Unterabfragen könnten auch mit EXISTS ausgedrückt werden. Im Allgemeinen spielen die mit ANY/ALL gebildeten Abfragen keine allzu große Rolle in der Praxis.

Beschreibung:

- „>ALL“ bedeutet „größer als jeder Wert“ - mit anderen Worten „größer als das Maximum“.
- „>ANY“ bedeutet „größer als mindestens ein Wert“, das heißt, „größer als das Minimum“.

Beispiel:

Gesucht sind die Artikel, deren Gewicht größer ist als jedes Gewicht der Artikel aus Ludwigshafen.

```
select *
from artikel
where gewicht >ALL (select gewicht
                      from artikel
                      where astadt = ,Ludwigshafen');
```

oder:

```
select *
from artikel
where gewicht > (select max(gewicht)
                   from artikel
                   where astadt = ,Ludwigshafen');
```

Gesucht sind die Artikel, deren Gewicht größer ist als mindestens ein Gewicht der Artikel aus Ludwigshafen.

```
select *
from artikel
where gewicht >ANY (select gewicht
                      from artikel
                      where astadt = ,Ludwigshafen');
```

Auch diese Abfrage könnte durch ein min() ersetzt werden und würde das gleiche Ergebnis liefern

6.11.1.4 Unterabfragen in der FROM- Klausel

Durch eine Unterabfrage in der FROM- Klausel wird eine abgeleitete Tabelle für die Abfrage erstellt. Eine abgeleitete Tabelle ist eine besondere Verwendung einer Unterabfrage in einer FROM- Klausel, auf die ein Alias oder ein vom Benutzer angegebener Name verweist. Das Resultset der Unterabfrage in der FROM- Klausel bildet eine Tabelle, die von der äußeren Abfrage verwendet wird

Beispiel:

Gesucht sind die Artikelnummern und Artikelnamen der Artikel, deren Lagermenge zwischen 500 und 900 Stück liegt.

```
select a.anr, aname
from (select anr, aname
       from artikel
       where amenge between 500 and 900) as a;
```

6.11.1.4.1 Spalten- Alias und Unterabfragen in der FROM-Klausel

Jedes Objekt innerhalb einer Abfrage muss einen eindeutigen Namen besitzen, so auch bei Unterabfragen in der FROM- Klausel.

Das Schreiben von Alias- Namen kann Inline oder Extern erfolgen.

Beispiel:

Spalten- Alias Inline definieren.

```
select lieferjahr, count(lnr) as [Lieferantennummer]
from  (select year(Idatum) as [lieferjahr], lnr
       from lieferung) as lief
group by lieferjahr;
```

Beispiel:

Spalten- Alias Extern definieren.

```
select lieferjahr, count(lnr) as [Lieferantennummer]
from  (select year(Idatum), lnr
       from lieferung) as lieff(lieferjahr,lnr)
group by lieferjahr;
```

6.11.1.5 Unterabfragen in der SELECT- Liste

Unterabfragen können immer eingesetzt werden, wenn ein Ausdruck in SELECT-, UPDATE-, INSERT- und DELETE- Anweisungen verwendet werden soll. Die Unterabfrage in der SELECT- Liste muss einen skalaren Wert ergeben.

Beispiel:

Gesucht sind die Artikelnummer, der Artikelname, das Gewicht, das Durchschnittsgewicht aller Artikel sowie die Differenz des Gewichts jedes einzelnen Artikels zum Durchschnittsgewicht aller Artikel.

```
select anr, aname, gewicht,
       (select avg(gewicht) from artikel) as 'Durchschnittsgewicht aller Artikel',
       gewicht - (select avg(gewicht) from artikel) as 'Differenz zum Durchschnittsgewicht'
  from artikel;
```

6.11.2 Korrelierte Unterabfragen

Korrelierte Unterabfragen sind Abfragen bei denen die Unterabfrage für jede Zeile der äußeren Abfrage wiederholt ausgeführt wird.

Korrelierte Unterabfragen sind leicht daran zu erkennen, dass eine Spalte einer Tabelle in der Unterabfrage mit einer Spalte einer Tabelle außerhalb der Unterabfrage verglichen wird.

Eine korrelierte Unterabfrage wird wie folgt ausgewertet:

- 1 Die äußere Abfrage übergibt die Spaltenwerte des ersten Datensatzes (Prüfdatensatz) an die innere Abfrage.
- 2 Die innere Abfrage verwendet diesen Wert, um die innere Abfrage auszuwerten.
- 3 Die innere Abfrage gibt einen Wert (das Ergebnis der inneren Abfrage) an die äußere Abfrage zurück Damit wird die Bedingung der WHERE- Klausel für den Prüfdatensatz überprüft Ist das Ergebnis „Wahr“, dann geht der Datensatz der äußeren Abfrage in die Ausgabe.
- 3 Dann beginnt das Ganze mit dem nächsten Datensatz der äußeren Tabelle.

Beispiel:

Gesucht sind die Lieferanten, die mindestens dreimal geliefert haben.

```
select a*
from lieferant a
where 3 <=      (select count(*)
                  from lieferung b
                  where alnr = blnr);
```

6.11.3 EXISTS und NOT EXISTS in Unterabfragen

Die Operatoren werden in korrelierten Unterabfragen verwendet

Die Bedingung für den Prüfdatensatz der äußeren Abfrage wird dann als WAHR ausgewertet, wenn die innere SELECT- Anweisung zumindest eine Ergebnisreihe liefert.

Die innere Anfrage ist immer von einer Variablen (Spalte) abhängig, die in der äußeren Abfrage berechnet (bereitgestellt) wird.

Wenn eine Unterabfrage mit dem EXISTS- Operator eingeleitet wird, wird getestet, ob Daten vorhanden sind, die die Unterabfrage erfüllen. Tatsächlich werden keine Daten an die äußere Abfrage zurückgegeben, sondern der Wert TRUE oder FALSE. Das Abrufen von Zeilen wird beendet, sobald bekannt ist, dass mindestens eine Zeile die WHERE- Bedingung in der Unterabfrage erfüllt, bei EXISTS beim ersten zurückgeben des Wertes TRUE.

Laut Vereinbarung wird in der Unterabfrage (*) verwendet, da die Angabe der zurückgegebenen Spalten unerheblich ist. Es wird nur auf Vorhandensein einer Bedingung geprüft.

Beispiel:

Gesucht sind die Lieferanten, die bereits geliefert haben.

```
select a*
from lieferant a
where exists      (select *
                  from lieferung b
                  where a.lnr = b.lnr);
```

Mit dem Operator NOT EXISTS würden die Lieferanten ausgegeben, die noch nicht geliefert haben.

6.12 Verknüpfung von Tabellen – JOIN

Die Möglichkeit, zwei oder mehrere Tabellen einer Datenbank miteinander zu verknüpfen, ist eine der grundsätzlichen Eigenschaften des relationalen Datenmodells. Diese Eigenschaft stellt gleichzeitig einen der wichtigsten Unterschiede zwischen relationalen und nichtrelationalen Datenbanken dar.

Bei einer Verknüpfung handelt es sich um eine Operation, mit der eine oder mehrere Tabellen abgefragt werden können, um ein Ergebnis zu erstellen, das Zeilen und Spalten der an der Verknüpfung beteiligten Tabellen enthält.

Tabellen werden normalerweise über Spalten (PS- und FS- Spalten) verknüpft, die in beiden Tabellen enthalten sind. Verknüpfungen ermöglichen, die im Zuge der Normalisierung aufgeteilten Daten wieder zusammenzuführen (Herstellen der logischen Gesamtaussage).

Man unterscheidet drei Arten von Verknüpfungen:

- Innere Verknüpfungen (INNER JOIN),
- äußere Verknüpfungen (OUTER JOIN)
- und das kartesische Produkt (CROSS JOIN oder Kreuzverknüpfung).

Allgemeine Syntax (SQL Server Syntax):

```
select spaltenliste
from {tablename1 | viewname1}, {tablename2 | viewname2}[, ...]
[where verknüpfungsspalte1 operator verknüpfungsspalte2 [{and | or} ...]
 [{and | or} Suchbedingungen]
```

ANSI Syntax:

```
select spaltenliste
from {tablename1 | viewname1} Jointyp join {tablename2 | viewname2}
      [on Verknüpfungsspalte1 operator Verknüpfungsspalte2] [...]
[where Suchbedingungen]
```

Join- Typen:

| | |
|--------------------|---|
| CROSS JOIN | - kennzeichnet das kartesische Produkt |
| [INNER] JOIN | - Kennzeichnet die natürliche Verknüpfung zweier Tabellen |
| LEFT [OUTER] JOIN | - linke Außenverknüpfung |
| RIGHT [OUTER] JOIN | - rechte Außenverknüpfung |
| FULL [OUTER] JOIN | - eine Kombination beider Außenverknüpfungen |

6.12.1 Das kartesische Produkt – CROSS JOIN

Beim Kartesischen Produkt wird ein Ergebnis erzeugt, welches alle Spalten der an der Operation beteiligten Tabellen enthält. Weiterhin wird jeder Datensatz der einen Tabelle mit allen Datensätzen der anderen Tabelle verkettet. Die Ergebnistabelle enthält also so viele Datensätze wie das Produkt der Datensätze der ersten und der zweiten Tabelle ergibt.

Beispiel:

Das Kartesische Produkt zwischen der Tabelle „Lieferant“ und der Tabelle „Lieferung“

```
select *
from lieferant, lieferung;
```

oder:

```
select *
from lieferant cross join lieferung;
```

Die Ergebnistabelle besitzt 8 Spalten und enthält 60 Datensätze Aber nur 12 Datensätze im Ergebnis haben wirklich eine logische Beziehung zueinander

Wann ist ein Kartesisches Produkt sinnvoll?

In der Praxis wird das Kartesische Produkt äußerst selten bewusst benutzt Sie können jedoch verwendet werden, um Testdaten für eine Datenbank zu generieren oder alle möglichen Kombinationen für Checklisten oder Vorlagen für Unternehmensprozesse aufzulisten

6.12.2 Der INNER JOIN

Bei einem INNER JOIN werden zwei oder mehrere Tabellen über zwei Spalten (Domänengleich) miteinander verknüpft. Die innere Verknüpfung ist der Standardverknüpfungstyp.

Die dabei verwendeten Operatoren und Tabellen bestimmen die Art der Tabellenverknüpfung.

Das Resultset enthält nur Datensätze, die den Verknüpfungsbedingungen entsprechen.

Richtlinien:

- Die INNER JOIN- Klausel kann mit JOIN abgekürzt werden.
- In der SELECT- Liste können nur Spaltennamen verwendet werden, die in den am Join beteiligten Tabellen enthalten sind.
- Es sollten keine NULL- Werte als Verknüpfungsbedingung verwendet werden, da NULL- Werte nicht zueinander ausgewertet werden können.
- Um mehr als zwei Tabellen in derselben Abfrage zu verknüpfen, können mehrere JOIN- Klauseln verwendet werden.

6.12.2.1 Der THETA JOIN

Die allgemeinste Form eines Joins wird THETA JOIN genannt. Gekennzeichnet ist dieser Join durch den Vergleichsoperator, mit dem die Verknüpfungsspalten verbunden werden. Folgende Vergleichsoperatoren werden verwendet: =, >, <, <>, <=, >=

Beispiel:

Gesucht sind die Angaben von Lieferanten und Artikeln für die gilt, der Wohnort ist alphabetisch kleiner als der Lagerort irgendeines Artikels.

```
select *
from lieferant, artikel
where Istadt < astadt;
```

oder:

```
select *
from lieferant join artikel
on Istadt < astadt;
```

6.12.2.2 Der EQUI JOIN

Ein THETA JOIN mit dem Vergleichsoperator „=“ wird auch EQUI JOIN genannt.

Diese Art von Join ist die logischste Art Tabellen miteinander zu verknüpfen. In der Regel werden die Tabellen dabei über ihre Schlüsselpalten (PS- und FS- Spalten) miteinander verknüpft. In anderen Fällen über domänengleiche Spalten.

Beispiel:

Gesucht sind die Lieferanten mit ihren Lieferungen.

```
select *
from lieferant a, lieferung b
where a.lnr = b.lnr;
```

oder:

```
select *
from lieferant a join lieferung b
on a.lnr = b.lnr;
```

6.12.2.3 Der NATURAL JOIN

Das ist ein Join in dessen Ergebnistabelle die beiden gleichen Verknüpfungsspalten nur einmal enthalten sind. Das wird in der Regel nur bei EQUI JOINS und SELF JOINS erforderlich sein.

Beispiel:

Gesucht sind die Lieferanten und ihre Lieferungen.

```
select a.*, anr, lmenge, ldatum
from lieferant a, lieferung b
where a.lnr = b.lnr;
```

oder:

```
select a.*, anr, lmenge, ldatum
from lieferant a join lieferung b
on a.lnr = b.lnr;
```

6.12.2.4 Der SELF JOIN

Der SELF JOIN stellt eine Verknüpfung einer Tabelle mit sich selbst dar.

Der SELF JOIN wird verwendet, um Hierarchien oder Baumstrukturen darzustellen.

Beispiel:

Gesucht sind die Lieferanten, die in einer Stadt wohnen, in welcher auch noch andere Lieferanten wohnen.

```
select a. *
from lieferant a, lieferant b
where a.Istadt = b.Istadt
and a.lnr <> b.lnr;
```

oder:

```
select a. *
from lieferant a join lieferant b
on a.Istadt = b.Istadt
and a.lnr <> b.lnr;
```

6.12.3 OUTER JOIN

Wie innere Verknüpfungen geben auch äußere Verknüpfungen kombinierte Werte zurück, die der Verknüpfungsbedingung entsprechen. Zusätzlich dazu werden jedoch auch die Datensätze der linken, oder der rechten oder der beiden Tabellen angezeigt, die der Verknüpfungsbedingung nicht entsprechen.

Der SQL Server unterstützt die SQL-92- Syntax für äußere Verknüpfungen. Die Legacysyntax für die Angabe von äußeren Verknüpfungen basierend auf der Verwendung der Operatoren *= und =* in der WHERE- Klausel wird nicht mehr unterstützt. Um Abwärtskompatibel zu bleiben muss der Kompatibilitätsgrad für die Datenbank geändert werden.

Die SQL-92- Syntax wird empfohlen, da bei ihr die Mehrdeutigkeiten nicht auftreten, die sich manchmal aus äußeren Verknüpfungen gemäß der Legacysyntax von Transact- SQL ergeben.

Jointypen:

| Type | Beschreibung |
|----------------------------------|---|
| LEFT OUTER JOIN oder LEFT JOIN | Beinhaltet sämtliche Reihen der zuerst (linken) genannten Tabelle, sowie die Reihen der zweiten (rechten) Tabelle, die die Verknüpfungsbedingung erfüllen. |
| RIGHT OUTER JOIN oder RIGHT JOIN | Beinhaltet sämtliche Reihen der zuletzt (rechten) genannten Tabelle, sowie die Reihen der ersten (linken) Tabelle, die die Verknüpfungsbedingung erfüllen. |
| FULL OUTER JOIN oder FULL JOIN | Beinhaltet sämtliche Reihen der linken und rechten Tabelle, die die Verknüpfungsbedingung erfüllen und auch die Reihen der linken und rechten Tabelle die diese Bedingung nicht erfüllen. |

6.12.3.1 Linke Außenverknüpfung

Beispiel:

Gesucht sind alle Lieferanten und die von ihnen durchgeführten Lieferungen. Angezeigt werden sollen aber auch die Lieferanten, die noch nicht geliefert haben.

```
select a.*, anr, lmenge, ldatum
from lieferant a left join lieferung b
on a.lnr = b.lnr;
```

6.12.3.2 Rechte Außenverknüpfung

Beispiel:

Gesucht sind alle Lieferungen und die Lieferanten, die diese Lieferung durchgeführt haben. Weiterhin sollen alle Lieferungen angezeigt werden, für die kein Lieferant mehr existiert.

```
select lname, status, lstadt, b.*
from lieferant a right join lieferung b
on a.lnr = b.lnr;
```

6.12.3.3 Volle Außenverknüpfung

Beispiel:

Gesucht sind alle Lieferanten mit ihren Lieferungen, außerdem sollen die Lieferanten angezeigt werden, die noch nicht geliefert haben und auch die Lieferungen, für die kein Lieferant mehr existiert.

```
select *  
from lieferant a full join lieferung b  
on a.lnr = b.lnr;
```

Hinweis: Die Operatoren für äußere Verknüpfungen ($*=$ und $=*$) werden nicht unterstützt, wenn der Kompatibilitätsgrad der Datenbank auf 90 festgelegt ist

6.12.4 APPLY- Operator

Dieser Operator ermöglicht das Abrufen von Ergebnissen die hierarchisch geordnet sein müssen oder als Liste (z.B. kommasepariert) dargestellt werden.

Syntax:

```
linker_tabellenausdruck {outer | cross} apply rechter_tabellenausdruck
```

Der rechte Ausdruck des APPLY- Operators wird für jede Zeile des linken Ausdrucks ausgewertet.

Der Hauptunterschied zwischen den beiden Ausdrücken besteht darin, dass der rechte eine Tabellenwertfunktion verwenden kann, die eine Spalte aus dem linken Ausdruck als eines der Argumente der Funktion verwendet.

Es muss CROSS oder OUTER angegeben werden.

- Bei CROSS werden keine Zeilen erstellt, wenn der rechte Ausdruck für eine bestimmte Zeile im linken Ausdruck ausgewertet wird Das zurückgegebene Ergebnis ist leer
- Bei OUTER wird selbst dann für jede Zeile des linken Ausdrucks eine Zeile erstellt, wenn der rechte Ausdruck für diese Zeile ausgewertet wird Das zurückgegebene Ergebnis ist leer

Beispiel:

Auflisten der Lieferanten und die Nummern der von Ihnen gelieferten Artikel als Kommaseparierte Werteliste.

```
SELECT Iname, LEFT(Artikelliste, LEN(Artikelliste)-1) AS 'gelieferte Artikel'
FROM lieferant a
CROSS APPLY
    (SELECT CAST(anr AS NVARCHAR(10)) + ', '
     FROM lieferung b
     WHERE b.lnr = a.lnr
     ORDER BY b.lnr
     FOR XML PATH("")) AS x(Artikelliste);
```

| <u>Iname</u> | <u>gelieferte Artikel</u> |
|--------------|------------------------------|
| Schmidt | A01, A02, A03, A04, A05, A06 |
| Jonas | A01, A02 |
| Blank | A02 |
| Clark | A02, A04, A05 |
| Adam | NULL |
| Müller | NULL |

Weitere Beispiele wie hierarchische Darstellungen durch Verwendung einer Tabellenwertfunktion im rechten Tabellenausdruck, finden Sie in der Onlinedokumentation.

6.12.5 Die OVER Klausel

Die OVER- Klausel bestimmt die Partitionierung und Reihenfolge eines Ergebnisses vor der Anwendung der zugehörigen Fensterfunktion. Das heißt die OVER- Klausel definiert ein Fenster oder eine benutzerdefinierte Reihe von Zeilen innerhalb eines Abfrageergebnis. Eine Fensterfunktion berechnet dann einen Wert für jede Zeile im Fenster.

Verwenden Sie die OVER- Klausel mit folgenden Funktionen.

- Rangfolgefunktionen (siehe Punkt 6.5.3)
- Aggregatfunktionen
- Analytische Funktionen
- NEXT VALUE FOR- Funktion für die Arbeit mit Sequenzen

Syntax:

```
over  (
    [<partition by klausel>]
    [<order by klausel>]
    [<row or range klausel>]
)
```

<partition by klausel> ::=
partition by spalte [, ...n]

- Gibt die Spalte an, nach der das Ergebnis partitioniert wird.
- Kann nur auf Spalten verweisen die von der FROM- Klausel bereitgestellt werden.
- Kann nicht auf Ausdrücke oder Aliase verweisen.
- Kann ein Spaltenname, eine skalare Funktion, eine benutzerdefinierte Variable oder eine skalare Unterabfrage sein.

<order by klausel> ::=
order by spalte [{asc | desc}]
[collate collation_name] [,...n]

- Definiert die logische Reihenfolge der Zeilen innerhalb jeder Partition, das heißt die logische Reihenfolge in der die Fensterfunktion ausgeführt wird.
- Kann eine Spalte oder einen Ausdruck angeben.
- Kann nur auf Spalten verweisen die von der FROM- Klausel bereitgestellt werden.

<row or range klausel> ::=
{ rows | range } <window frame extent>

- Grenzt die Zeilen innerhalb der Partition weiter ein, indem Start- und Endpunkte innerhalb der Partition angegeben werden. Das erfolgt entweder anhand der logischen oder physischen Zuordnung.
 Die physische Zuordnung wird durch die ROWS- Klausel erreicht.
- Die ROWS- Klausel gibt eine feste Anzahl an Zeilen an, die der aktuellen Zeile vorausgehen oder nachfolgen.
- Die RANGE- Klausel schränkt die Zeilen innerhalb einer Partition logisch ein, indem eine Reihe von Werten unter Berücksichtigung des Werts der aktuellen Zeile angegeben wird.

```

<window frame extent> ::= 
  {<window frame preceding> | <window frame between>}

<window frame between> ::= 
  between <window frame bound> and <window frame bound>

<window frame bound> ::= 
  {<window frame preceding> | <window frame following>}

<window frame preceding> ::= 
  {unbounded preceding | <unsigned_value> preceding | current row}

<window frame following> ::= 
  {unbounded following | <unsigned_value> following | current row}

<unsigned value> ::= 
  { <unsigned integer literal> }

```

| | |
|----------------------------------|--|
| UNBOUNDED PRECEDING | Gibt an, dass das Fenster bei der ersten Zeile der Partition startet und markiert damit den Fensterstartpunkt. |
| <unsigned_value> PRECEDING | Wert der die Anzahl von Zeilen angibt, die der aktuellen Zeile vorausgehen sollen. Kann nicht mit RANGE verwendet werden. |
| CURRENT ROW | Gibt an, dass das Fenster bei Verwendung mit ROWS oder auf Basis des aktuellen Werts, und bei Verwendung von RANGE bei der aktuellen Zeile Startet oder endet. Kann als Start- oder als Endpunkt angegeben werden. |
| BETWEEN ... AND ... | Wird mit ROWS oder RANGE verwendet, um den unteren und den oberen Grenzpunkt des Fensters anzugeben. Zum Beispiel erstellt RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING ein Fenster, das mit der aktuellen Zeile startet und mit der letzten Zeile der Partition endet. |
| <unsigned_value> FOLLOWING | Wert der die Anzahl von Zeilen angibt, die der aktuellen Zeile folgen sollen. Zum Beispiel erstellt ROWS BETWEEN 2 FOLLOWING AND 10 FOLLOWING ein Fenster, das mit der 2. Zeile startet, die auf der aktuellen Zeile folgt, und mit der 10. Zeile endet, die der aktuellen Zeile folgt. Kann nicht mit RANGE verwendet werden. |
| <unsigned_value> integer literal | Ist ein positives ganzzahliges Literal, das die Anzahl an Zeilen oder Werten angibt, die der aktuellen Zeile oder dem aktuellen Wert vorausgehen oder folgen. Kann nur mit ROWS verwendet werden. |

In einer Abfrage mit einer einzelnen FROM- Klausel können mehrere Fensterfunktionen verwendet werden, wobei die OVER- Klause für jede Funktion sich in der Partitionierung und Reihenfolge unterscheiden kann.

Beispiel:

Liefervorgänge und Gesamtliefermenge der Artikel A01 und A02.

```
select a.anr, a.name,
       row_number() over (partition by a.anr
                           order by a.name desc) as [Liefervorgang],
       year(Idatum) as [Lieferjahr], lmenge as [Liefermenge],
       sum(lmenge) over (partition by a.anr
                           order by year(Idatum)
                           rows between unbounded preceding
                           and current row) as [Gesamtliefermenge]
  from artikel a join lieferung b on a.anr = b.anr
 where a.anr in('A01','A02');
```

| anr | a.name | Liefervorgang | Lieferjahr | Liefermenge | Gesamtliefermenge |
|-----|--------|---------------|------------|-------------|-------------------|
| A01 | Mutter | 1 | 1990 | 300 | 300 |
| A01 | Mutter | 2 | 1990 | 300 | 600 |
| A01 | Mutter | 3 | 2013 | 100 | 700 |
| A02 | Bolzen | 1 | 1990 | 200 | 200 |
| A02 | Bolzen | 2 | 1990 | 400 | 600 |
| A02 | Bolzen | 3 | 1990 | 200 | 800 |
| A02 | Bolzen | 4 | 1990 | 200 | 1000 |
| A02 | Bolzen | 5 | 2013 | 200 | 1200 |
| A02 | Bolzen | 6 | 2013 | 200 | 1400 |
| A02 | Bolzen | 7 | 2013 | 400 | 1800 |
| A02 | Bolzen | 8 | 2013 | 400 | 2200 |

6.13 Abfragen mit CTE

Die Rückgabe hierarchischer Daten ist eine häufige Verwendung rekursiver Abfragen, z.B. Anzeigen von Mitarbeitern in einem Organisationsdiagramm.

CTE (Common Table Expression) gibt ein temporäres Ergebnis an, das als allgemeiner Tabellenausdruck bekannt ist. Dieser wird von einer einfachen Abfrage abgeleitet und innerhalb des Ausführungsreichs einer SELECT-, INSERT-, UPDATE- oder DELETE-Anweisung definiert. Diese Klausel kann auch in einer CREATE VIEW-Anweisung als Teil der definierenden SELECT-Anweisung verwendet werden. Ein allgemeiner Tabellenausdruck kann auch Verweise auf sich selbst enthalten. In diesem Fall handelt es sich um einen rekursiven allgemeinen Tabellenausdruck.

Beispiel:

```
with ArtikelLief(lnr,anz) as
(
  select lnr, count(distinct anr) as 'anz'
  from lieferung
  group by lnr
)
select alnr, lname,anz
  from lieferant a join ArtikelLief b on alnr = blnr;
```

6.14 Volltextdaten abfragen

Mit der Volltextsuche können Anwendungen und Benutzer Abfragen auf zeichenbasierte Daten ausführen, die in Spalten mit den Datentypen CHAR, VARCHAR, NCHAR, NVARCHAR, TEXT, IMAGE, XML oder VARBINARY(MAX) und FILESTREAM abgespeichert sind.

Jede Spalte auf die eine Volltextsuche durchgeführt werden soll muss mindestens durch einen Volltextindex indiziert sein. Für jede Spalte kann hierbei eine eigene Sprache verwendet werden.

Volltextabfragen führen linguistische Suchvorgänge für Textdaten in Volltextindizes durch, wobei Wörter und Ausdrücke anhand von Regeln einer bestimmten Sprache verarbeitet werden. Eine Volltextabfrage gibt alle Dokumente zurück, die mindestens eine Übereinstimmung mit dem Suchbegriff enthalten.

Bei diesen Abfragen kann nach folgendem gesucht werden:

- einfacher Begriff: - Mindestens ein Wort oder Ausdruck.
- Präfixbegriff: - Ein Wort oder Ausdruck, bei dem die Wörter mit dem angegebenen Text beginnen.
- Generierungsbegriff: - Flexionsformen (grammatische Merkmale) eines bestimmten Worts.
- NEAR- Begriff: - Ein Wort oder Ausdruck in der Nähe eines anderen Worts oder Ausdrucks.
- Thesaurus: - Synonyme Formen eines bestimmten Worts.
- gewichteter Begriff: - Wörter oder Ausdrücke mit gewichteten Werten.

Der SQL Server stellt zwei Volltextprädikate (CONTAINS und FREETEXT) zum Abfragen von Volltextdaten bereit. Zusätzlich gibt es noch zwei Volltextfunktionen (CONTAINSTABLE und FREETEXTTABLE) die ein Ergebnis mit zusätzlichen Informationsspalten generieren.

Wenn ein Volltextbefehl in einer Abfrage angegeben wird, übergibt SQL Server den Suchbegriff an das Volltextindizierungsmodul, wo eine Worttrennung durchgeführt wird, um das Argument in Token zu zerlegen. Auf der Basis dieser Daten werden Verteilungsstatistiken an den Optimierer zurückgegeben, der daraufhin den Volltextteil der Abfrage mit dem relationalen Teil zusammenführt, um einen Abfrageplan zu erstellen.

6.14.1 Prädikate CONTAINS und FREETEXT

CONTAINS und FREETEXT werden in der WHERE- Klausel oder der HAVING- Klausel einer SELECT- Anweisung angegeben. Sie können mit beliebigen anderen SQL- Prädikaten (like, between, ...) kombiniert werden.

Die Prädikate CONTAINS und FREETEXT geben TRUE oder FALSE zurück. Sie können nur verwendet werden, um Auswahlkriterien anzugeben, anhand derer ermittelt wird, ob eine angegebene Zeile mit der Volltextabfrage übereinstimmt. Diese werden im Ergebnis zurückgegeben.

- Bei der Verwendung von CONTAINS oder FREETEXT können Sie entweder eine einzelne Spalte, eine Liste mit Spalten oder alle Spalten der Tabelle für die Suche auswählen. Optional können Sie die Sprache angeben, welche die Volltextabfrage für die Wörtertrennung, die Wortstammerkennung und Thesaurus- Suche sowie die Entfernung von Füllwörtern verwenden soll.

CONTAINS und FREETEXT eignen sich für unterschiedliche Arten von Übereinstimmungen:

- CONTAINS (oder CONTAINSTABLE) wird verwendet, für genaue oder ungenaue (Fuzzy-) Übereinstimmungen mit einzelnen Wörtern und Satzteilen, für innerhalb einer bestimmten Entfernung angrenzende Wörter sowie für gewichtete Übereinstimmungen. Wenn CONTAINS verwendet wird, muss mindestens eine Suchbedingung festgelegt werden, die den zu suchenden Text und die Bedingungen für Übereinstimmungen angibt.
Es können logische Operation zwischen Suchbedingungen verwendet werden.
- FREETEXT (oder FREETEXTTABLE) wird verwendet, um nach der Bedeutung ohne exakter Übereinstimmung des Wortlauts, nach angegebenen Wörtern, Ausdrücken oder Sätzen (sogenannten Freitextzeichenfolgen) zu suchen. Übereinstimmungen werden dann generiert, wenn ein Begriff oder eine Form eines Begriffs im Volltextindex einer angegebenen Spalte gefunden wird.

Um Abfragen auf Verbindungsservern auszuführen, können vierteilige Namen in CONTAINS oder FREETEXT verwendet werden.

6.14.1.1 CONTAINS

Folgendes kann mit CONTAINS gesucht werden:

- Ein Wort oder ein Ausdruck.
- Das Präfix eines Worts oder eines Ausdrucks.
- Ein Wort, das einem anderen Wort ähnlich ist.
- Ein Wort, das mithilfe von Beugung aus einem anderen generiert.
- Ein Wort, das ein Synonym für ein anderes Wort ist. Dazu wird ein Thesaurus verwendet.

Syntax:

```

CONTAINS
(
{spaltenname | (spaltenliste) | *}, '<contains_suchbedingung>'
[LANGUAGE sprache]
)

<contains_suchbedingung>::=
{
    <einfacher_ausdruck>
    | <präfixausdruck>
    | <generierungsausdruck>
    | <abstandsausdruck>
    | <gewichteter_ausdruck>
    | {(<contains_suchbedingung>) [{and | and not | or}
        <contains_suchbedingung> [...]
    ]
}

<einfacher_ausdruck>::=
wort | "Phrase"

<präfixausdruck>::=
wort | "Phrase"

<generierungsausdruck>::=
FORMSOF({INFLECTIONAL | THESAURUS}, <einfacher_ausdruck< [...])

<abstandsausdruck>::=
{{<einfacher_ausdruck> | <präfixausdruck>}
{{NEAR | ~}
{<einfacher_ausdruck> | <präfixausdruck>} } [...]

```

```
<gewichteter_ausdruck>::=
  ISABOUT ({  
    {<einfacher_ausdruck>  
     | <präfixausdruck>  
     | <generierungsausdruck>  
     | <abstandsausdruck>}  
    [WEIGHT(gewichtungswert)]  
  } [...])
```

| | |
|----------------------|--|
| INFLECTIONAL: | Es wird die sprachenabhängige Wortstammkennung für den angegebenen einfachen Ausdruck verwendet. |
| THESAURUS: | Es wird der Wortschatz verwendet, der der Volltext-SpaltenSprache oder der angegebenen Sprache entspricht |
| NEAR ~: | Das Wort oder der Ausdruck auf beiden Seiten des NEAR- Operators bzw. des ~- Operators muss in einem Dokument enthalten sein um eine Übereinstimmung zurückzugeben. Es können mehrere NEAR- Begriffe verkettet werden. |

Beispiel:

Alle Artikel in deren Artikelbeschreibung das Wort „Nocken“ oder „Splinte“ vorkommt.

```
select *  
from artikel  
where contains(beschreibung, N'Nocken OR Splinte');  
go
```

[Weitere Beispiele in der Projektmappe Volltext\(Index, Katalog, Suche\).](#)

6.14.1.2 FREETEXT

Bei Verwendung von FREETEXT führt das Volltextabfragemodul intern die folgenden Aktionen für den "freetext_string" aus, weist jedem Begriff eine Gewichtung zu und sucht dann nach Übereinstimmungen.

- Trennt die Zeichenfolge in einzelne Wörter auf der Basis von Wortgrenzen (Wörtertrennung).
- Generiert Flexionen der Wörter (Wortstammerkennung).
- Legt eine Liste mit Erweiterungen oder Ersetzungen für die Begriffe auf der Basis von Übereinstimmungen im Thesaurus fest.

Syntax:

```
FREETEXT ({ spalten_name | (spaltenliste) | * }, 'freetext_string'  
[ , LANGUAGE language_term ] )
```

Volltextabfragen mit FREETEXT sind nicht so genau wie Volltextabfragen mit CONTAINS. Das SQL Server- Volltextsuchmodul identifiziert wichtige Wörter und Ausdrücke. Reservierten Schlüsselwörtern und Platzhalterzeichen, die dann eine Bedeutung besitzen, wenn sie im <contains_suchbedingung>- Parameter des CONTAINS- Prädikats angegeben werden, wird keine spezielle Bedeutung zugewiesen.

Beispiel:

Suchen nach allen Artikelbeschreibungen in denen Wörter im Zusammenhang mit Formelementen zum formschlüssigen Kontakt enthalten sind.

```
select *
from artikel
where freetext(beschreibung, 'Formelementen zum formschlüssigen Kontakt');
go
```

6.14.2 Funktionen CONTAINSTABLE und FREETEXTTABLE

Die Funktionen werden in der FROM- Klausel einer SELECT- Anweisung verwendet und es wird analog zu einem regulären Tabellennamen darauf verwiesen.

Es wird eine SQL Server-Volltextsuche für volltextindizierten Spalten mit zeichenbasierten Datentypen durchgeführt.

6.14.2.1 CONTAINSTABLE- Funktion

Gibt eine Tabelle mit keiner, einer oder mehreren Zeilen für jene Spalten zurück, die präzise oder weniger präzise (Fuzzy- Suche) Übereinstimmungen mit einzelnen Wörtern bzw. Ausdrücken aufweisen, die den Abstand von Wörtern oder gewichtete Worttreffer enthalten.

Sie ist für dieselben Arten von Übereinstimmungen geeignet wie das CONTAINS- Prädikat und verwendet die gleichen Suchbedingungen.

Im Gegensatz zu CONTAINS werden bei Abfragen mit CONTAINSTABLE ein Relevanz- Rangfolgenwert (Relevance Ranking Value, RANK) und ein Volltextschlüssel für jede Ergebniszeile zurückgeben.

Syntax:

```
CONTAINSTABLE
(
    table , {spaltenname | (spaltenliste) | * } , '<contains_search_condition>'
    [, LANGUAGE language_term]
    [, top_n_by_rank]
)
```

Parameterbeschreibung siehe Punkt 6.14.1.1 oben.

Die zurückgegebene Tabelle besitzt eine Spalte mit dem Namen "KEY" und "RANK", die Volltextschlüsselwerte für die Zeilen enthält, die mit dem Auswahlkriterium übereinstimmen.

Beispiel:

Die Artikel in deren Artikelbeschreibung das Wort "Verbindungen" in der Nähe von "Schrauben" gefunden wird.

```
select anr, aname, beschreibung
from artikel as a
join containstable(dbo.artikel, beschreibung, N' NEAR(verbindungen,schrauben)') as tab
on a.anr = tab.[KEY]
order by RANK desc;
```

6.14.2.2 FREETEXTTABLE- Funktion

Diese Funktion gibt eine Tabelle mit keiner, einer oder mehreren Zeilen für alle Spalten mit Werten zurück, die mit der Bedeutung, nicht aber mit dem genauen Wortlaut des Texts in freetext_string übereinstimmen.

Die Funktion ist für dieselben Arten von Übereinstimmungen geeignet wie FREETEXT.

Syntax:

```
FREETEXTTABLE
(
    table , {spaltenname | (spaltenliste) | * }, 'freetext_string'
    [, LANGUAGE language_term ]
    [, top_n_by_rank]
)
```

FREETEXTTABLE verwendet die gleichen Suchbedingungen wie das FREETEXT- Prädikat.

Wie bei CONTAINSTABLE enthält die zurückgegebene Tabelle die Spalten "KEY" und "RANK", auf die innerhalb der Abfrage verwiesen wird, um die entsprechenden Zeilen abzurufen und die Zeilenrangfolgenwerte zu verwenden.

Beispiel:

Suchen nach allen Artikelbeschreibungen in denen Wörter mit Formelementen zum formschlüssigen Kontakt enthalten sind.

```
select anr, aname, beschreibung
from artikel as a
join freetexttable(dbo.artikel, beschreibung, 'formschlüssigen Kontakt') as tab
on a.anr = tab.[KEY]
order by RANK desc;
```

6.15 Zusammenfassung SELECT- Anweisung

Mit der SELECT- Anweisung ist es möglich, Daten aus den Tabellen einer Datenbank abzufragen. Fast für jede Fragestellung ergeben sich ein oder mehrere Antworten in Form von SELECT- Anweisungen. Das wäre ja nicht weiter tragisch, wenn dabei gewährleistet wäre, dass die verschiedenen Abfragen nicht nur das gleiche Ergebnis liefern, sondern sich auch in ihrer Ausführungszeit nicht nennenswert voneinander unterschieden. Leider kann davon nicht die Rede sein, die Antwortzeiten inhaltlich gleicher Abfragen können in Größenordnungen schwanken. Darum ist es notwendig, schon bei der Formulierung von Abfragen, an deren Effizienz zu denken.

- Abfragen mit Unterabfragen sind in der Regel langsamer als Abfragen ohne Unterabfragen.
- Abfragen mit korrelierter Unterabfrage sind in der Regel langsamer wie Abfragen mit einfachen Unterabfragen.
- Abfragen wo zwei oder mehrere Tabellen mit dem JOIN- Operator verknüpft werden sind schneller als alle anderen Abfragen über mehr als eine Tabelle.
- Abfragen mit einfacher, nicht negierter EXISTS- Unterabfrage können (und sollten) immer durch Abfragen ohne Unterabfrage ersetzt werden.
- Einfache, negierte EXIST- Unterabfragen können immer durch andere Unterabfragen („NOT IN“, ...) ersetzt werden.

7 Datenänderung mit SQL

7.1 Einfügen von Daten

Mit der INSERT- Anweisung mit der VALUES- Klausel, können Datensätze in eine Tabelle eingefügt werden.

Syntax:

```
[WITH <common_table_expression> [ ,...n ] ]
INSERT
[ TOP ( n ) [ PERCENT ] ]
[INTO] {table_name | view_name} [with(table_hint [..])] [(spaltenliste)]
[output Klausel]
{VALUES ({DEFAULT | NULL | wert} [,...n]) | SOL_Anweisung |
DEFAULT VALUES}
```

Richtlinien:

- Die einzufügenden Werte müssen die Einschränkungen der Tabellenspalten einhalten, sonst schlägt die INSERT- Anweisung fehl.
- Die Insert- Anweisung kann bis zu 1024 Spaltenwerte enthalten.
- Die einzufügenden Daten werden mit Hilfe der VALUES- Klausel angegeben.
- Wenn eine INSERT- Anweisung eine Einschränkung oder Regel verletzt bzw. die Anweisung einen Wert enthält, der mit dem Datentyp der Spalte nicht kompatibel ist, schlägt die Anweisung fehl, und SQL Server zeigt eine Fehlermeldung an.

Die INSERT- Berechtigungen erhalten standardmäßig Mitglieder der festen Serverrolle **sysadmin**, der festen Datenbankrollen **db_owner** und **db_datawriter** und der Tabellenbesitzer. Zum Ausführen von INSERT mit der Option BULK der OPENROWSET- Funktion müssen Sie Mitglied der Serverrolle **sysadmin** oder **bulkadmin** sein.

7.1.1 In der Reihenfolge der Tabellenspalten

Die Spaltenreihenfolge und der Datentyp der neuen Daten müssen mit der Spaltenreihenfolge und dem Datentyp in der Tabelle übereinstimmen.

Beispiel:

```
insert into lieferant
values ('L06', 'Müller', 30, 'Erfurt');
go
```

7.1.2 Veränderte Reihenfolge

Die Reihenfolge der anzugebenden Werte wird durch die Reihenfolge in der Spaltenliste bestimmt.

Beispiel:

```
insert into lieferant (Istadt,Inr,Iname,status)
values ('Weimar', 'L07', 'Conrad', '10');
go
```

7.1.3 Unbekannte Werte aufnehmen

Manchmal ist ein Wert zum Zeitpunkt der Datenerfassung noch nicht bekannt. Dafür muss dann eine Nullmarke aufgenommen werden.

Es gibt zwei Methoden um unbekannte Werte aufzunehmen. Einmal durch die Angabe des Schlüsselworts "NULL" und zum anderen durch weglassen von Spalten in der Spaltenliste der INSERT- Anweisung.

Beispiele:

```
insert into lieferant
values ('L08', 'Schulze', NULL, 'Erfurt');
go
```

oder:

```
insert into lieferant (Inr, Iname)
values ('L09', 'Maria');
go
```

Hinweis: Für die betreffenden Spalten müssen auch Nullmarken erlaubt sein. Ist für einer der weggelassenen Spalten, im zweiten Beispiel, eine Default-Einschränkung oder ein Standard definiert, so wird keine NULL- Marke eingefügt sondern der entsprechende Standardwert.

7.1.4 Default Werte aufnehmen

Mit einer INSERT- Anweisung können auch die für Tabellenspalten definierten Standardwerte in die Tabelle übertragen werden.

Beachten Sie:

- Standardwerte können nicht auf Spalten mit der Eigenschaft IDENTITY angewendet werden.
- Auch macht es wenig Sinn für Primärschlüsselspalten oder Unique- Spalten einen Standardwert zu vereinbaren.

Beispiele:

```
insert into lieferant
values ('L09', 'Klausen', default, default);
go
```

oder:

```
insert into lieferant (Inr, Iname)
values ('L09', 'Klausen');
go
```

7.1.5 Insert mit DEFAULT VALUES

"Default Values" fügt eine Reihe mit Standardwerten für sämtliche Spalten in die Tabelle ein.

Hinweis: Da eine Tabelle keine zwei gleichen Datensätze besitzen darf, sollte eine Spalte (die Primärschlüsselspalte) mit der Eigenschaft IDENTITY (oder Uniqueidentifier, ...) versehen sein. Für jede andere Spalte muss ein Standardwert definiert sein.

Beispiel:

```
create table zitrone
(nr integer identity(1,1) not null,
name varchar(40) not null default 'Mustermann',
vname varchar(40) null default 'Sabiene',
ort varchar(50) null default 'Erfurt');
go

insert into zitrone
default values;
go
```

7.1.6 Einfügen von Daten mit Select und Execute

Es ist mit der INSERT- Anweisung auch möglich das Ergebnis einer Abfrage in eine Tabelle einzufügen.

Beachten:

- Die Spaltenzahl und Spaltenreihenfolge der Abfrage muss mit der Anzahl der Spalten und deren Reihenfolge in der Zieltabelle übereinstimmen.
- Werden Spalten weggelassen und es sind für diese Spalten Standardwerte definiert, dann werde diese in die Spalten eingefügt.

Beispiel zu Insert mit Select- Anweisung:

```
insert into Lief_name
select Iname, Istadt
from lieferant
where Iname like '%eu%';
go
```

Beispiel zu Insert mit Select- Anweisung in einer gespeicherten Prozedur:

```
create procedure lief_ausw @name nvarchar(30) = '%eu%'
as
select Iname, Istadt
from lieferant
where Iname like @name;
go

insert into lief_name
execute lief_ausw '%a%';
go
```

Beispiel zu Insert mit Select- Anweisung als Literalzeichenfolge:

```
insert into lief_name
execute
(
  select Iname, Istadt
  from lieferant
  where Iname like @name
);
go
```

7.1.7 Insert mit ROW CONSTRUCTOR

Mit Hilfe des Zeilenkonstruktors ist es möglich, mit einer INSERT- Anweisung mehr als einen Datensatz in eine Tabelle oder View einzufügen. Das kann in der Reihenfolge der Spalten erfolgen oder in einer anderen Reihenfolge. Dazu muss aber dann auch eine Spaltenliste angegeben werden.

Beispiel:

```
insert into lieferant
values ('L06', 'Schulze', 10, 'Erfurt'),
       ('L07', 'Krause', 15, 'Gotha'),
       ('L08', 'Friedrichs', 10, 'Erfurt');
go
```

7.1.8 Insert mit der Option BULK der OPENROWSET- Funktion

Die OPENROWSET- Funktion unterstützt unter anderem auch Massenvorgänge über einen integrierten BULK-Anbieter, mit dem Daten aus einer Datei gelesen und als Rowset zurückgegeben oder gespeichert werden können.

Beispiel:

```
alter table lieferant add texte varbinary(max) null;
go
insert into lieferant with(tablock, ignore_triggers)
values ('L06', 'Meier', 10, 'Erfurt',
        (select * from openrowset(BULK N'C:\meintext.txt', SINGLE_BLOB) as a));
go
select lname, cast(texte as varchar(200)) from lieferant where lnr = 'L06';
go
```

7.1.9 Verwenden von Output bei Insert

Die Output- Klausel wird verwendet um die Ergebnisse der Anweisung an das aufrufende Programm zurückzugeben. Zum Beispiel für Bestätigungen, Archivierungen und andere Anwendungsanforderungen verwendet. Alternativ können Ergebnisse in eine Tabelle oder Tabellenvariable eingefügt werden.

Beispiel:

```
declare @testtab table (nr char(3), name varchar(50), ort varchar(50))

insert into lieferant
output inserted.lnr, inserted.lname, inserted.istadt into @testtab
values('L06', 'Krause', null, 'Erfurt');

select * from @testtab
go
```

7.1.10 Insert mit TOP n

Gibt die Anzahl oder den Prozentsatz zufälliger Zeilen an, die in eine Tabelle eingefügt werden. Die Zeilen, auf die im TOP-Ausdruck verwiesen wird, die mit INSERT verwendet werden, sind nicht in einer bestimmten Reihenfolge angeordnet.

Beispiel:

```
insert top (5) into lief_art
select a.lnr, lname, ldatum, lmenge, c.anr, aname, farbe
from lieferant a
inner join lieferung b on a.lnr = b.lnr
inner join artikel c on b.anr = c.anr;
go
```

7.1.11 Insert mit CTE

Ein CTE (Common Table Expression) ist ein allgemeiner Tabellenausdruck der ein temporäres Ergebnis zurückgibt. Dieser wird von einer einfachen Abfrage abgeleitet und innerhalb des Ausführungsbereichs einer einzigen SELECT-, INSERT-, UPDATE- oder DELETE-Anweisung definiert.

Im nachfolgenden Beispiel wird ein allgemeiner Tabellenausdruck (lief_lief) erstellt der die in die Tabelle (lft_lief) einzufügenden Spalten definiert. Die INSERT- Anweisung verweist auf die Spalten im allgemeinen Tabellenausdruck.

Beispiel:

```
create table lft_lief
(
lnr char(3), lname varchar(30), anr char(3), ldatum smalldatetime
);
go

with lief_lief (lnr, lname, lstadt, anr, lmenge, ldatum)
as
(
select a.lnr, lname, lstadt, anr, lmenge, convert(char(10), ldatum, 104)
from lieferant a join lieferung b
on a.lnr = b.lnr
)

insert into lft_lief
select lnr, lname, anr, ldatum
from lief_lief
select * from lft_lief;
go
```

7.2 Ändern von Daten

Ändert vorhandene Daten in einer Tabelle oder Sicht. Eine UPDATE- Anweisung kann immer nur auf eine Tabelle oder Sicht angewendet werden.

Wenn die Aktualisierung einer Zeile eine Einschränkung oder Regel verletzt, wenn sie die NULL- Einstellung für die Spalte verletzt oder wenn der neue Wert einen inkompatiblen Datentyp hat, wird die Anweisung abgebrochen, ein Fehler wird zurückgegeben und es werden keine Datensätze aktualisiert.

UPDATE- Anweisungen sind in benutzerdefinierten Funktionen nur zulässig, wenn es sich bei der Tabelle, die geändert wird um eine Variable vom Typ "table" handelt.

Die UPDATE- Berechtigungen erhalten standardmäßig Mitglieder der festen Serverrolle **sysadmin**, der festen Datenbankrollen **db_owner** und **db_datawriter** und der Tabellenbesitzer.

Syntax:

```
[WITH <common_table_expression> [ ,...n ] ]
UPDATE [ TOP ( n ) [ PERCENT ] ]{tablename|viewname} [with(table_hint [..])]
SET spalte1 = ausdruck1 [, spalte2 = ausdruck2 [,...]]
[output Klausel]
[FROM {tablename|viewname}[, {tablename|viewname}[,...]]]
[WHERE bedingungen]
```

7.2.1 Einfache Update- Anweisung

Eine UPDATE- Anweisung ohne WHERE- Klausel wird eher selten verwendet, da aller Werte der angegebenen Spalten geändert werden.

Alle Statuswerte der Lieferanten um 25 Prozent erhöhen.

```
update lieferant
set status = status * 1.25;
go
```

Alle Statuswerte der Lieferanten auf 20 setzen und die Wohnorte aller Lieferanten ändern.

```
update lieferant
set status = 20, lstadt = 'Urbich';
go
```

Lieferant L03 wohnt jetzt in Gotha.

```
update lieferant
set lstadt = 'Gotha'
where lnr = 'L03';
go
```

Ändern des Status der Lieferanten die nach dem 13.08.90 geliefert haben.

```
update lieferant
set status = status * 0.2
from lieferant, lieferung
where lieferant.lnr = lieferung.lnr
and ldatum >= '13.08.1990';
go
```

oder:

```
update lieferant
set status = status * 1.2
where lnr in      (select lnr
                     from lieferung
                   where idatum >= '13.08.1990');
go
```

Änderung des Status in Abhängigkeit von mehreren bestimmten Bedingungen.

```
update lieferant
set status =      case when status > 0 and < 10 then status * 1.2
                      when status >= 10 and < 20 then status * 1.3
                      when status >= 20 and < 30 then status * 1.1
                      else status * 1.05
                  end;
go
```

7.2.2 Update mit der TOP- Klausel

Mit der TOP n Klausel kann eine zufällige Anzahl von Spaltenwerten (absolut oder in Prozent) für eine Änderung festgelegt werden. Die Auswahl ist absolut unabhängig von einer Reihenfolge der Datensätze.

Beispiel:

Für drei zufällige Lieferanten soll der Status um 25 Prozent erhöht werden.

```
update top (3) lieferant
set status = status * 1.25;
go
```

7.2.3 Update mit der Output- Klausel

Genau wie bei der INSERT- Anweisung kann man sich mit der OUTPUT- Klausel die von der Anweisung gerade bearbeiteten Werte anzeigen lassen.

Beispiel:

Die nachfolgende UPDATE- Anweisung erhöht den Status der Hamburger Lieferanten um 25 Prozent. Die alten Statuswerte und die neuen Statuswerte werden durch die OUTPUT- Klausel in eine Variable vom Datentyp "table" geschrieben.

```
declare @mytablevar table
(
lnr char(3) not null,
altstatus int,
neustatus int
);

update lieferant
set status = status * 1.20
output inserted.lnr, deleted.status, inserted.status
into @mytablevar
where lstadt = 'Hamburg';
go

select * from @mytablevar;
go
```

7.2.4 Update mit CTE

Ein CTE (Common Table Expression) ist ein allgemeiner Tabellenausdruck der ein temporäres Ergebnis zurückgibt. Dieser wird von einer einfachen Abfrage abgeleitet und innerhalb des Ausführungsreichs einer einzigen SELECT-, INSERT-, UPDATE- oder DELETE-Anweisung definiert.

Beispiel:

Der Status soll für den jeweiligen Lieferanten um die Anzahl der von ihm im Jahre 1990 durchgeführten Lieferungen erhöht werden.

```
with all_ort(nr, anz)
as
(
select lnr, count(*) as anz
from lieferung
where ldatum between '01.01.1990' and '31.12.1990'
group by lnr
)

update lieferant
set status = status + anz
from lieferant a, all_ort b
where a.lnr = b.nr;
go
```

7.2.5 Update mit .WRITE- Klausel zum Ändern von Daten

Die .WRITE- Klausel wird benutzt um einen bestimmten Teilwert einer Zeichenfolge in einer Datenspalte vom Datentyp "nvarchar(max)" zu ändern.

Die Funktion benötigt drei Parameter. Der erste Parameter ist das Ersetzungswort, der zweite Parameter ist die Startposition des zu ersetzenen Teilwertes und der dritte Parameter bezeichnet die Länge des zu ersetzenen Teilwertes.

Diese Klausel ist aktuell in SQL Server 2005 eingeführt worden und ersetzt die WRITETEXT- und UPDATETEXT- Anweisungen.

Beispiel:

Das Wort "gut", Bestandteil der Zeichenfolge in der Spalte "Bemerkung" der Tabelle Lieferung, soll in "sehr gut" geändert werden.

```
alter table lieferung
add bemerkung nvarchar(max) null;
go

insert into lieferung values('L01','A01',100, convert(char(10),getdate(),104),
'Die Qualität des Artikels war gut, alle Teile sind komplett.');
go

update lieferung
set bemerkung .WRITE(N'sehr gut',30,3)
where convert(char(10),ldatum,104) = convert(char(10),getdate(),104);
go
```

7.2.6 Update mit .WRITE- Klausel zum Hinzufügen und Entfernen von Daten

Die .WRITE- Klausel wird benutzt um einen bestimmten Teilwert einer Zeichenfolge in einer Datenspalte vom Datentyp "nvarchar(max)" zu ändern.

Die Funktion benötigt drei Parameter. Der erste Parameter ist das Ersetzungswort, der zweite Parameter ist die Startposition des zu ersetzenen Teilwertes und der dritte Parameter bezeichnet die Länge des zu ersetzenen Teilwertes.

Diese Klausel ist aktuell in SQL Server 2005 eingeführt worden und ersetzt die WRITETEXT- und UPDATETEXT- Anweisungen.

Beispiel:

Nachfolgend sind vier Möglichkeiten für die Arbeit mit der .WRITE- Klausel in einer UPDATE- Anweisung dargestellt.

```
insert into lieferung values('L01','A01',100, convert(char(10),getdate(),104),null);
go
```

Das erste Beispiel ersetzt temporäre Daten durch reale Daten. Da .WRITE keine Werte mit NULL- Marken ändern kann muss die NULL- Marke vorher mit einem temporären Wert versehen werden.

```
update lieferung
set bemerkung = (N'Ein NULL Wert')
where lnr = 'L01' and anr = 'A01' and idatum = convert(char(10),getdate(),104);
go

update lieferung
set bemerkung .WRITE(N'Die Muttern waren nicht ordnungsgemäß verpackt ',0,null)
where lnr = 'L01' and anr = 'A01' and idatum = convert(char(10),getdate(),104);
go
```

Das zweite Beispiel fügt an eine bereits existierende Zeichenkette weitere Daten hinten an.

```
update lieferung
set bemerkung .WRITE(N'und verölt.',null,0)
where lnr = 'L01' and anr = 'A01' and idatum = convert(char(10),getdate(),104);
go
```

Im dritten Beispiel werden Daten ab einer bestimmten Position bis zum Ende der Zeichenkette entfernt.

```
update lieferung
set bemerkung .WRITE(null,47,0)
where lnr = 'L01' and anr = 'A01' and idatum = convert(char(10),getdate(),104);
go
```

Und das vierte Beispiel entfernt Daten einer Zeichenfolge ab einer bestimmten Position und einer bestimmten Länge.

```
update lieferung
set bemerkung .WRITE('',24,14)
where lnr = 'L01' and anr = 'A01' and idatum = convert(char(10),getdate(),104);
go
```

7.3 Löschen von Daten

Die DELETE- Anweisung entfernt eine Zeile oder mehrere Zeilen einer Tabelle oder Sicht. Jede Tabelle, aus der alle Zeilen entfernt wurden, verbleibt in der Datenbank. Die DELETE- Anweisung löscht lediglich Zeilen aus der Tabelle; die Tabelle muss mit der DROP TABLE- Anweisung aus der Datenbank entfernt werden.

Syntax:

```
[WITH <common_table_expression> [ ,...n ] ]
DELETE [ TOP ( n ) [ PERCENT ] ]{tablename|viewname} [with(table_hint [...])]
[output Klausel]
FROM {tablename|viewname}[,{tablename|viewname}[,...]]
[WHERE bedingung]

DELETE FROM {tablename|viewname}
[with(table_hint [...])]
[WHERE bedingung]
```

} Das ist ANSI Standard

Für die Zieltabelle sind DELETE-Berechtigungen erforderlich. SELECT- Berechtigungen werden ebenfalls benötigt, wenn die Anweisung eine WHERE-Klausel enthält.

Mitglieder der Serverrolle **sysadmin**, der Datenbankrollen **db_owner** und **db_datawriter** und der **Tabellenbesitzer** erhalten standardmäßig DELETE- Berechtigungen.

7.3.1 Einfache DELETE- Anweisungen

Eine DELETE- Anweisung ohne WHERE- Klausel löscht alle Datensätze der angegebenen Tabelle. Damit das nicht aus Versehen passieren kann muss der DBA geeignete Maßnahmen ergreifen. Sollte es beabsichtigt sein große Datentabellen vollständig zu leeren bietet sich außerdem eine weitere Anweisung an.

Beispiele:

Löschen aller Lieferungen.

```
delete from lieferung
```

Löschen der Hamburger Lieferanten.

```
delete from lieferant
where lstadt = 'Hamburg';
```

Löschen des Lieferanten mit der Lieferantennummer L03.

```
delete from lieferant
where lnr = 'L03';
```

Löschen der Lieferungen des Lieferanten Schmidt.

```
delete from lieferung
where lnr in      (select lnr
                   from lieferant
                   where lname = 'Schmidt');
```

oder:

```
delete lieferung
from lieferung, lieferant
where lieferung.lnr = lieferant.lnr
and lname = 'Schmidt';
```

Die DELETE- Anweisung löscht die Datensätze einer Tabelle, Satz für Satz. Das kann bei großen Tabellen einige Zeit in Anspruch nehmen. Hat aber den Vorteil, dass jeder gelöschte Datensatz im Transaktionsprotokoll protokolliert wird und dadurch auch wieder hergestellt werden könnte. Allerdings kann das Transaktionsprotokoll auch sehr stark anwachsen.

Eine schnellere Möglichkeit wäre die Verwendung der TRUNCATE Table- Anweisung. Sie entspricht einer DELETE- Anweisung ohne WHERE- Klausel.

Syntax:

```
truncate table tabname
```

Bei der TRUNCATE TABLE- Anweisung werden ganze physische Seiten gelöscht. Dieser Vorgang wird aber nicht protokolliert wodurch das Protokoll während des Vorgangs nicht so stark wächst.

Die Berechtigungen für die TRUNCATE TABLE- Anweisung hat standardmäßig der Tabellenbesitzer, Mitgliedern der festen Serverrolle **sysadmin** und der festen Datenbankrollen **db_owner** und **db_ddladmin**. Die Berechtigungen sind nicht übertragbar.

7.3.2 DELETE mit der TOP- Klausel

Seit der Version 2005 von SQL Server ist es möglich den TOP n- Operator in einer DELETE- Anweisung einzusetzen.

Beispiel:

Löschen von 2,5 Prozent aller Lieferungen. Die Wahl der Datensätze ist beliebig.

```
delete top (2.5) percent  
from lieferung;  
go
```

7.3.3 DELETE mit der OUTPUT-Klausel

Mit der OUTPUT- Klausel ist es möglich sich die gerade gelöschten Datensätze anzeigen zu lassen.

Hinweis: Für die Tabelle auf die die DELETE- Anweisung mit der OUTPUT- Klausel angewendet wird, dürfen keine DELETE- Trigger aktiviert sein.

Beispiel:

Nachfolgende Anweisung löscht alle Lieferungen die vor dem 12. Mai 1990 stattgefunden haben. Gleichzeitig werden die gelöschten Datensätze angezeigt.

```
use standard;  
go  
  
delete lieferung  
output deleted.*  
where Idatum < '12.05.1990';  
go
```

7.4 MERGE- Statement

Das MERGE- Statement ist vor allem in Ladeprozessen großer Datawarehouse- Szenarien interessant. Mithilfe der MERGE- Anweisung, können in einer Anweisung INSERT-, UPDATE- und DELETE- Anweisungen ausgeführt werden.

In der MERGE- Anweisung kann eine Datenquelle mit einer Zieltabelle oder einer Zielsicht verknüpft werden und für das Ziel basierend auf den Ereignissen dieser Verknüpfung (Wahr oder Falsch, in der Quelle oder im Ziel) mehrere DML- Aktionen ausgeführt werden.

Folgende Aufgaben lassen sich lösen:

- Einfügen oder Aktualisieren von Datensätzen in eine Zielt- oder Quelltabelle in Abhängigkeit von Verknüpfungsbedingungen.
- Synchronisieren von zwei Tabellen.

Die Anweisung besteht aus fünf primären Klauseln:

- Die MERGE- Klausel gibt die Tabelle oder Sicht an, die das Ziel für die INSERT-, UPDATE- oder DELETE- Anweisungen darstellt.
- Die USING- Klausel gibt die Datenquelle an, die mit dem Ziel verknüpft wird.
- Die ON- Klausel gibt die Verknüpfungsbedingungen an, die bestimmen, ob Ziel und Quelle übereinstimmen oder nicht übereinstimmen.
- Die WHEN- Klausel geben die Aktionen an, die nach Maßgabe der Ereignisse der ON- Klausel und etwaiger weiterer in der WHEN- Klausel angegebener Suchkriterien ausgeführt werden sollen.
- Die OUTPUT- Klausel gibt für jede Zeile im Ziel, die eingefügt, aktualisiert oder gelöscht wird, eine Zeile zurück.

Zeilen in der Quelle werden anhand des in der ON- Klausel angegebenen Verknüpfungsprädikats mit Zeilen im Ziel verglichen.

Je nach Definition der WHEN-Klauseln in der Anweisung kann die Eingabezeile mit einer der folgenden übereinstimmen:

- Ein zueinander passendes Paar aus einer Zeile aus dem Ziel und einer Zeile aus der Quelle. Dies ist das Ergebnis der **WHEN MATCHED**-Klausel.
- Eine Zeile aus der Quelle, für die im Ziel keine entsprechende Zeile vorhanden ist. Dies ist das Ergebnis der **WHEN NOT MATCHED BY TARGET**-Klausel.
- Eine Zeile aus dem Ziel, für die in der Quelle keine entsprechende Zeile vorhanden ist. Dies ist das Ergebnis der **WHEN NOT MATCHED BY SOURCE**-Klausel.

Syntax:

```
[WITH <common_table_expression> [...n]]
MERGE
    [TOP( {n | @var }) [PERCENT]]
    [INTO] tabellen_ziel [ [AS] tabellen_alias]
    USING tabellen_quelle
        ON verknüpfungsbedingung [{and | or} verknüpfungsbedingung
    [...]]
    [WHEN MATCHED [ AND bedingung ]
        THEN dml_aktion ]
    [WHEN NOT MATCHED [ BY TARGET ] [ AND bedingung ]
        THEN dml_aktion]
    [WHEN NOT MATCHED BY SOURCE [ AND bedingung ]
        THEN dml_aktion]
    [output klausel]
    [option(table_hint [,...])];
```

Beispiel:

```
use standard;
go

if not exists(select * from sys.objects where object_id = OBJECT_ID('lieferung_history'))
    CREATE TABLE lieferung_history
    (
        lnr nchar(3) NOT NULL,
        anr nchar(3) Not Null,
        lmenge int NOT NULL,
        ldatum datetime NOT NULL
    );
    go

    merge into dbo.lieferung_history a
    using dbo.lieferung b
    on a.lnr = b.lnr and a.anr = b.anr
        and convert(char(10),a.ldatum,104) = convert(char(10),b.ldatum,104)
    when not matched
    then insert (lnr, anr, lmenge, ldatum) values(b.lnr, b.anr, b.lmenge, b.ldatum);
    go
```

Beispiel mit OUTPUT- Klausel:

Es wird vorausgesetzt dass die Tabelle "lieferung_history" bereits existiert. Die Variable \$Action liefert eine Spalte in der die Aktionen stehen, die von der Anweisung durchgeführt wurden.

```
merge into lieferung_history a
using dbo.lieferung b
on a.lnr = b.lnr and a.anr = b.anr
    and convert(char(10),a.ldatum,104) = convert(char(10),b.ldatum,104)
when not matched
then insert (lnr, anr, lmenge, ldatum) values(b.lnr, b.anr, b.lmenge, b.ldatum)
output $action, inserted.lnr, inserted.anr, inserted.ldatum ;
go
```

Hinweis: Beachten Sie, dass die MERGE- Anweisung immer mit einem Semikolon abgeschlossen werden muss.

8 Arbeiten mit XML

Seit über zehn Jahren gibt es eine, von der herstellerunabhängigen Organisation W3C definierte, Beschreibungssprache **XML** (Extensible Markup Language). Sie dient als Schnittstelle für Daten zwischen verschiedenen Systemen. Seit der Version 2000 unterstützt SQL Server diese Sprache.

Bei XML werden nicht nur die Daten sondern auch die Beschreibung der Daten hinterlegt.

In nichthomogenen Systemen ist es wichtig einen allgemeinen und anerkannten Standard für den Datenaustausch zwischen den Systemen zu etablieren. Deshalb hat sich XML heutzutage als Standard für den Import und Export von Daten durchgesetzt.

Es ist in SQL Server 2005 nicht nur möglich relationale Daten als XML- Daten darzustellen sondern auch diese in dem Format zu speichern. Dazu hat SQL Server den neuen Datentyp "**xml**" implementiert. Weiterhin ist es möglich Daten aus XML- Formaten in relationale Form zu bringen und in Tabellen abzulegen.

8.1 XML- Daten

Nachfolgend kurz eine Einführung in die Thematik XML. Für die Lieferanten der Firma sollen weitere Informationen gespeichert werden. Zurzeit liegen sie in Papierform vor. Hier ein Beispiel für den Lieferanten Schmidt.

| | |
|-------------------------|---|
| Bisherige Wohnadressen: | Müllergasse 14, 20359 Hamburg Hopfenberg 66, 21038 Weisebach |
| Adresse Privat: | Unter dem Feldweg 4, 21039 Escheburg |
| Adresse Firma: | Hafenstraße 22, 20359 Hamburg |
| Geburtsdatum: | 12.09.1977 |

Diese Informationen sollen in ein XML- Format überführt werden um sie in die Datenbank aufzunehmen.

Hinweise zur Wohlgeformtheit von XML- Dokumenten:

- XML- Elemente werden immer in "<>" eingeschlossen und besitzen eine öffnende und eine schließende Markierung (Tag). Das schließende Tag bekommt zur Unterscheidung ein "/" im Namen vorangestellt:
<Geburtsdatum>12.09.1977</Geburtsdatum>
Man spricht von einem Geburtsdatum- Element (auch Knoten ist üblich) mit dem Wert 12.09.1977.
- Das entsprechende XML- Dokument beginnt immer mit einem Instruktions-Element.
<?xml version="1.0" encoding="iso-8859-1" ?>
Es werden nicht alle Sonderzeichen automatisch unterstützt. Um zum Beispiel deutsche Umlaute zu verwenden, sollte der Zeichensatz (Encoding) "**iso-8859-2**" verwendet werden.
- Jedes XML- Dokument besitzt genau ein Grundelement (Wurzelknoten), das alle anderen Elemente umschließt. XML- Dokumente ohne Grundelement werden als XML- Fragmente bezeichnet.
- Elemente müssen korrekt geschachtelt sein, d.h. Elemente, die innerhalb anderer liegen, müssen korrekt innerhalb der Elemente geöffnet und wieder geschlossen werden.
- Die Groß- und Kleinschreibung muss beachtet werden.
- Verschiedene Attribute innerhalb eines Elements müssen eindeutige Bezeichner haben.

Wenn Sie die oben stehenden Punkte beachten dann könnte aus den Informationen des Lieferanten dieses XML- Dokument entstehen:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<Informationen ID="L01">
  <BisherigeWohnadressen>
    <Adresse>
      <strasse>Müllergasse 14</strasse>
      <plz>20359</plz>
      <ort>Hamburg</ort>
    </Adresse>
    <Adresse>
      <strasse>Hopfenberg 66</strasse>
      <plz>21038</plz>
      <ort>Weisebach</ort>
    </Adresse>
  </BisherigeWohnadressen>
  <AktuelleAdresse>
    <strasse>Unter dem Feldweg 4</strasse>
    <plz>21039</plz>
    <ort>Escheburg</ort>
  </AktuelleAdresse>
  <FirmenAdresse>
    <strasse>Hafenstraße 22</strasse>
    <plz>20359</plz>
    <ort>Hamburg</ort>
  </FirmenAdresse>
  <Geburtsdatum>12.09.1977</Geburtsdatum>
</Informationen>
```

In der neuen Version SQL Server 2005 ist endlich der Punkt erreicht, wo relationale und strukturierte Daten nebeneinander in der Datenbank ihre Berechtigung finden. XML ist aber nicht die "Eierlegende Wollmilchsau", sondern man sollte sich schon genau überlegen welche Daten für XML geeignet sind und welche besser relational gespeichert werden.

8.2 XML- Schema

Der Datentyp "**XML**" lässt zu, dass XML- Dokumente in Spalten relationaler Tabellen abgelegt werden können. Nun ist es aber sinnvoll dass in einer Spalte die zum Beispiel angelegt wurde um Lebensläufe zu speichern, auch wirklich nur Lebensläufe stehen. Weiterhin sollen diese auch noch eine bestimmte Struktur besitzen um die Daten leicht zu finden.

Dazu sollte das Dokument vor dem Einfügen geprüft werden, ähnlich wie Werte durch Einschränkungen geprüft werden bevor sie für eine Spalte zugelassen werden.

Um den Aufbau von XML- Dokumenten zu prüfen werden sogenannte XML- Schemas verwendet.

Zunächst gab es dafür DTD (Document Type Definition) welches später durch XRD (XML Data Reduced) erweitert wurde. Dieses Format war allerdings sehr unflexibel.

Die W3C- Organisation hat für XML- Dokumente ein neues Format definiert, das in sogenannte XSD- Dateien abgelegt wird.

Tabellenspalten vom Datentyp XML und hinterlegtem XML- Schema nennt man typisierte XML- Spalten.

Wird nun versucht ein XML- Dokument in eine solche Spalte einzufügen welches dem hinterlegten Schema nicht entspricht so wird die Aktion abgebrochen.

Ein XML- Schema ist also eine Beschreibung der XML- Elemente.

8.2.1 Aufbau eines XML- Schemas

Im nachfolgenden Kapitel soll nur kurz der allgemeine Aufbau eines XML- Schemas beschrieben werden. Weitergehende Informationen finden Sie in einschlägiger Fachliteratur.

Kleine XML- Schemas können Sie von Hand schreiben aber größere sollten Sie sich einfach in einer Entwicklungsumgebung, wie Visual Studio, aus einem verfügbaren XML- Dokument erstellen lassen.

Grundsätzlich unterscheidet man bei der Beschreibung von XML- Elementen einfache Typen (simple Types) und komplexe Typen (complex Types).

8.2.1.1 Simple Types

Die simplen Typen besitzen selbst keine Unterelemente oder Attribute, sondern ausschließlich einen Textknoten für den Wert. Sie beschreiben einfache Elemente, die auf der untersten Ebene des XML- Dokuments liegen.

Sie besitzen ein Attribut **NAME**, welches den Namen des XML- Elements beschreibt und ein Attribut **TYPE** welches den Datentyp des XML- Elements beschreibt.

Beispiel:

Beschreibung des XML- Elements "Strasse" durch einen "simple Type".

```
<xs:element name="Strasse" type="xs:string" />
```

Für einfache XML- Elemente ist diese Beschreibung ausreichend, wollen Sie aber eventuell den Wertebereich für die zulässigen Werte einschränken müssen Sie den simple Type erweitern.

Beispiel:

Für ein fiktives XML- Element "**Geburtsjahr**" soll eine Einschränkung der zulässigen Werte mit dem XSD- Element "**xs:restriction**" definiert werden. Das Geburtsjahr soll eine Positive Zahl zwischen 1930 und 2100 sein.

```
<xs:element name="Geburtsjahr">
  <xs:simpleType>
    <xs:restriction base="xs:positiveInteger">
      <xs:minExclusive value="1930"/>
      <xs:maxExclusive value="2100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

8.2.1.2 Complex Types

Sind Elemente die andere Untergeordnete Elemente oder Attribute umschließen. Einige Beispiele sollen das verdeutlichen.

Nachfolgend wird ein XSD- Element **xs:sequence** verwendet um die Reihenfolge der Unterelemente in einer XML- Datei festzulegen.

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="Strasse" type="xs:string" />
    <xs:element name="plz" type="xs:string" />
    <xs:element name="ort" type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

Im Schema oben, sehen Sie dass das Element "**BisherigeWohnadressen**" ein oder mehrere Unterelemente "**Adresse**" besitzt. Das mehrfache Auftreten der Unterelemente kann mit dem XSD- Attribut "**maxOccurs**" wie folgt im Schema beschrieben werden.

```
<xs:element name="BisherigeWohnadressen">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="Adresse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="strasse" type="xs:string" />
            <xs:element name="plz" type="xs:string" />
            <xs:element name="ort" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

8.2.1.3 Attribute

Einen speziellen Fall stellt die Beschreibung von Attributen eines XML- Dokuments dar. Mit dem XSD- Element "**xs:attribute**" wird es beschrieben und dann genauso behandelt wie ein untergeordnetes simple Type- Element.

Im obenstehenden XML- Dokument befindet sich ein Informationen- Element mit einem Attribut "ID". Im Schema sieht das wie folgt aus.

```
<xs:element name="Informationen">
  <xs:complexType>
    <xs:sequence>
      ...
    </xs:sequence>
    <xs:attribute name="ID" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>
```

Nachdem die Grundbegriffe eines XML- Schemas bekannt sind, könnte das Schema für unsere obenstehende XML- Datei wie folgt aussehen.

```
<?xml version="1.0" encoding="iso-8859-2"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Informationen">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="BisherigeWohnadressen">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="Adresse">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="strasse" type="xs:string" />
                    <xs:element name="plz" type="xs:string" />
                    <xs:element name="ort" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```

<xs:element name="AktuelleAdresse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="strasse" type="xs:string" />
      <xs:element name="plz" type="xs:string" />
      <xs:element name="ort" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="FirmenAdresse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="strasse" type="xs:string" />
      <xs:element name="plz" type="xs:string" />
      <xs:element name="ort" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Geburtsdatum" type="xs:string" />
</xs:sequence>
<xs:attribute name="ID" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
</xs:schema>

```

8.2.1.4 Globale Typen

Mit einem Schema kann man ein XML- Dokument, von der Wurzel an, komplett beschreiben. Allerdings wird das Schema dann auch sehr umfangreich und komplex. Es gibt die Möglichkeit einfache Typen, als globale Typen zu definieren und sie in anderen Typen wieder verwenden.

Beispiel:

Das ist ein fiktives Beispiel welches die Verwendung eines globalen Typs verdeutlichen soll. Zum Beispiel sollen in einem Lebenslauf die Namen von Vater und Mutter angegeben werden. Dazu könnte man einen globalen Typ "**ElternTyp**" definieren der aus den beiden Unterelementen Vater und Mutter besteht.

```

<xs:complexType name="ElternTyp">
  <xs:sequence>
    <xs:element name="Vater" type="xs:string" />
    <xs:element name="Mutter" type="xs:string" />
  </xs:sequence>
</xs:complexType>

```

Anwendung:

```

<xs:element name="PersönlicheDaten">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Eltern" ref="ElternTyp">
      ...
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

8.2.1.5 Gruppierungen:

Wenn sich innerhalb eines XML- Schemas eine Gruppe von Elementen und Attributen häufig wiederholen, kann man diese als Gruppe global definieren und an mehreren anderen Stellen wieder verwenden.

Beispiel:

In den Adressen wiederholen sich immer wieder die gleichen Angaben. Egal ob es sich um Wohn- oder Firmenadressen handelt.

```
<xs:group name="AdressElemente">
  <xs:sequence>
    <xs:element name="strasse" type="xs:string" />
    <xs:element name="plz" type="xs:string" />
    <xs:element name="ort" type="xs:string" />
  </xs:sequence>
</xs:group>
```

Nun wird die Gruppe im oben stehenden Schema verwendet:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:group name="AdressElemente">
    <xs:sequence>
      <xs:element name="strasse" type="xs:string" />
      <xs:element name="plz" type="xs:string" />
      <xs:element name="ort" type="xs:string" />
    </xs:sequence>
  </xs:group>
  <xs:element name="Informationen">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="BisherigeWohnadressen">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" name="Adresse">
                <xs:complexType>
                  <xs:sequence>
                    <xs:group ref="AdressElemente"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="AktuelleAdresse">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="AdressElemente"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="FirmenAdresse">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="AdressElemente"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Geburtsdatum" type="xs:string" />
</xs:sequence>
<xs:attribute name="ID" type="xs:string" use="required" />
</xs:complexType>
</xs:element>
</xs:schema>
```

Zur Beschreibung der Elemente eines XML- Dokuments in einem Schema gibt es nach Definition des W3C verschiedenen festgelegte Elemente im Namensraum **xs=http://www.w3.org/2001/XMLSchema**.

In den beiliegenden Handzetteln sind XSD- Elemente und Attribute beschrieben.

8.3 Typisieren von XML- Daten

XML- Schemas stellen Typisierungsinformationen bereit, die die Beschaffenheit der Daten in Elementen und Attributen beschreiben. Dadurch werden die Struktur und der Inhalt von XML- Dokumenten beschränkt.

SQL Server 2005 unterstützt typisierte und nicht typisierte XML- Daten. Sobald eine Variable, eine Spalte oder ein Parameter an ein XML- Schema gebunden wird, spricht man von typisierten XML- Daten.

Das XML- Schema entspricht, seiner Wirkungsweise entsprechend, einer Einschränkung für das XML- Objekt. Sind die Daten in einem XML- Dokument dem zugrundeliegenden XML- Schema entsprechend, dann werden sie als gültig bezeichnet.

XML- Schemas werden auf Datenbankebene deklariert. Sie stellen Metadaten zur Verfügung, die XML- Datentypen definieren und beschränken.

8.3.1 XML- Schema erstellen

Es gibt mindestens zwei Wege ein XML- Schema in einer Datenbank zu erstellen. Sie können das Schema in der **CREATE XML SCHEMA**- Anweisung direkt schreiben oder es mit der **OPENROWSET**- Anweisung aus einer Datei in eine Variable vom Typ XML laden.

Syntax:

```
create xml schema collection [<relational_schema>.]name as expression
```

Argumente:

| | |
|---------------------------|--|
| relational_schema: | Identifiziert den relationalen Schemanamen. Wird kein Wert angegeben, wird vom relationalen Standardschema ausgegangen. |
| name: | Der Name für die XML- Schemaauflistung. |
| Expression: | Eine Zeichenfolgenkonstante oder eine skalare Variable. Ist ein Wert vom Datentyp varchar, varbinary, nvarchar, nvarbinary oder xml. |

Beispiel:

Die Struktur des Schemas steht direkt in der Anweisung.

```
create xml schema collection LftInfoSchema
as
...      hier steht jetzt die Schema- Definition (siehe oben)
```

Die Struktur des Schemas wird aus einer Datei geladen.

```
declare @varxml xml;
select @varxml = c
from openrowset(bulk 'C:\XML\infos zu lieferanten.xsd', single_blob) as temp(c);

create xml schema collection LftInfoSchema
as @varxml;
go
```

Zum Erstellen von XML Schemas ist mindestens eine der folgenden Berechtigungen erforderlich:

- **CONTROL**- Berechtigung für den Server.
- **ALTER ANY DATABASE**- Berechtigung für den Server.
- **ALTER**- Berechtigung für die Datenbank.
- **CONTROL**- Berechtigung in der Datenbank.
- **ALTER ANY SCHEMA**- Berechtigung und **CREATE XML SCHEMA COLLECTION**- Berechtigung in der Datenbank.
ALTER- oder **CONTROL**- Berechtigung für das relationale Schema und **CREATE XML SCHEMA COLLECTION**- Berechtigung in der Datenbank.

Informationen zu den XML- Schemas in einer Datenbank finden Sie in den Katalogsichten **sys.xml_schema_collection**, **sys.xml_schema_namespaces**, **sys.xml_schema_elements**, **sys.xml_schema_attributes** und anderen.

Beispiel:

Anzeigen einer Liste von XML Schemas in der Datenbank.

```
select * from sys.xml_schema_collections;
```

Anzeigen einer Liste von Namespaces in der Datenbank.

```
select name from sys.xml_schema_namespaces;
```

8.3.2 XML- Schema ändern

Sie können einem bestehendem XML- Schema weitere Schemakomponenten hinzufügen.

Syntax:

```
alter xml schema collection [relational_schema.]name add 'component'
```

Sie können damit neue XML- Schemas, deren Namespaces noch nicht in der XML- Schemaauflistung vorhanden sind hinzufügen, oder neue Komponenten zu vorhandenen Namespaces in der Auflistung hinzufügen.

8.3.3 XML- Schema löschen

Um die gesamte XML-Schemaauflistung und alle zugehörigen Komponenten zu löschen verwenden Sie die DROP XML SCHEMA- Anweisung.

Syntax:

```
drop xml schema collection [relational_schema.]name
```

Das Löschen einer XML- Schemaauflistung ist ein Transaktionsvorgang. Das heißt, wenn Sie eine XML-Schemaauflistung innerhalb einer Transaktion löschen und später ein Rollback für die Transaktion ausführen, wird die XML- Schemaauflistung nicht gelöscht.

Eine XML- Schemaauflistung, die verwendet wird, kann nicht gelöscht werden.

Folgendes darf für die zu löschende Auflistung nicht zutreffen:

- Sie darf keinem Parameter bzw. keiner Spalte vom Typ XML zugeordnet sein.
- Sie darf nicht in Tabelleneinschränkungen angegeben sein.
- In einer schemagebundenen Funktion oder gespeicherten Prozedur darf nicht darauf verwiesen werden.

8.3.4 XML- Schema verwenden

Das eben erstellte XML- Schema kann nun an Tabellenspalten vom Typ XML oder an XML- Variable gebunden werden.

Beispiele:

Typisierung einer Tabellenspalte.

```
alter table lieferant
add info xml(LftInfoSchema) null;
go

declare @varxml xml;
select @varxml = c
from openrowset(bulk 'C:\XML\infos zu lieferanten.xml', single_blob) as temp(c);
go

update lieferant
set info = @varxml
where lnr = 'L01';
go
```

Typisierung einer XML- Variablen.

```
declare @varxml xml(LftInfoSchema);
select @varxml = c
from openrowset(bulk 'C:\XML\infos zu lieferanten.xml', single_blob) as temp(c);
go
```

8.4 Relationale Daten in XML konvertieren

Mit Hilfe der SELECT- Anweisung und der darin enthaltenen FOR XML- Klausel können relationale Daten in XML- Daten umgewandelt werden. Dabei wird das Ergebnis einer Abfrage in eine XML- Struktur konvertiert.

Syntax:

```
[ FOR {BROWSE | <XML>}]

<XML> ::= 
  XML
  { {RAW [('ElementName')] | AUTO}
    [
      <CommonDirectives>
      [, {XMLDATA | XMLSCHEMA [ ('TargetNameSpaceURI')]}]
      [, ELEMENTS [{XSIMIL | ABSENT}]]
    ]
  | EXPLICIT
    [
      <CommonDirectives>
      [, XMLDATA ]
    ]
  | PATH [ ('ElementName') ]
    [
      <CommonDirectives>
      [, ELEMENTS [{XSIMIL | ABSENT}]]
    ]
  }
}

<CommonDirectives> ::=
  [, BINARY BASE64 ]
  [, TYPE ]
  [, ROOT [('RootName')]]
```

Die FOR XML- Klausel ist gegenüber SQL Server 2000 erweitert und verbessert worden. Dazu zählen die Darstellung des **timestamp**- Datentyps als numerischer Wert anstatt eines base64- Binärwertes und die Rückgabe des Ergebnisses als eine **nvarchar(max)**- Zelle.

Folgende Formatierungsmodi stehen zur Verfügung:

- FOR XML RAW
- FOR XML AUTO
- FOR XML PATH
- FOR XML EXPLICIT

8.4.1 Verwenden von FOR XML RAW

Dieser Modus erzeugt standardmäßig für jede Ergebnissezeile ein Element **<row>** in welchem die Spalten der Abfrage als Attribute dargestellt sind.

Beispiel:

Die Abfrage

```
select lname, anr, idatum
from lieferant a join lieferung b on a.lnr = b.lnr
where idatum between '20.07.1990' and '10.08.1990'
for xml raw;
```

liefert folgendes Ergebnis.

```
<row lname="Schmidt" anr="A04" idatum="1990-07-25T00:00:00" />
<row lname="Schmidt" anr="A05" idatum="1990-08-01T00:00:00" />
<row lname="Schmidt" anr="A06" idatum="1990-07-23T00:00:00" />
<row lname="Jonas" anr="A01" idatum="1990-08-02T00:00:00" />
<row lname="Jonas" anr="A02" idatum="1990-08-05T00:00:00" />
<row lname="Blank" anr="A02" idatum="1990-08-06T00:00:00" />
<row lname="Clark" anr="A02" idatum="1990-08-09T00:00:00" />
```

Um das **<row>**- Element umzubenennen, kann man direkt hinter dem Schlüsselwort RAW einen neuen Tag- Namen angeben. Die Attribute benennt man um indem man in der Abfrage Spaltenalias verwendet.

Beispiel:

Die Abfrage

```
select lname as 'Name', anr as Artikelnummer', idatum as 'Datum'
from lieferant a join lieferung b on a.lnr = b.lnr
where idatum between '20.07.1990' and '10.08.1990'
for xml raw('Lief');
```

liefert folgendes Ergebnis.

```
<Lief Name="Schmidt" Artikelnummer="A04" Datum="1990-07-25T00:00:00" />
<Lief Name="Schmidt" Artikelnummer="A05" Datum="1990-08-01T00:00:00" />
<Lief Name="Schmidt" Artikelnummer="A06" Datum="1990-07-23T00:00:00" />
<Lief Name="Jonas" Artikelnummer="A01" Datum="1990-08-02T00:00:00" />
<Lief Name="Jonas" Artikelnummer="A02" Datum="1990-08-05T00:00:00" />
<Lief Name="Blank" Artikelnummer="A02" Datum="1990-08-06T00:00:00" />
<Lief Name="Clark" Artikelnummer="A02" Datum="1990-08-09T00:00:00" />
```

Soll von der Attributs- auf Elementorientierte Formatierung umgestellt werden, verwenden Sie das Schlüsselwort ELEMENTS.

Beispiel:

Die Abfrage

```
select lname as 'Name', anr as Artikelnummer', ldatum as 'Datum'
from lieferant a join lieferung b on a.lnr = b.lnr
where ldatum between '20.07.1990' and '10.08.1990'
for xml raw('Lief'), ELEMENTS;
```

liefert folgendes Ergebnis. Das ist ein Auszug.

```
...
<Lief>
  <Name>Schmidt</Name>
  <Artikelnummer>A05</Artikelnummer>
  <Datum>1990-08-01T00:00:00</Datum>
</Lief>
<Lief>
  <Name>Schmidt</Name>
  <Artikelnummer>A06</Artikelnummer>
  <Datum>1990-07-23T00:00:00</Datum>
</Lief>
<Lief>
  <Name>Jonas</Name>
  <Artikelnummer>A01</Artikelnummer>
  <Datum>1990-08-02T00:00:00</Datum>
</Lief>
...
```

FOR XML RAW stellt einen einfachen Weg dar, um grundlegende XML- Strukturen aus relationalen Daten zu bilden.

- Die XML- Struktur ist ein XML- Fragment weil es keinen Stammknoten bereitstellt. Damit ist es kein wohlgeformtes XML- Dokument.
- Alle Spalten sind auf ein und dieselbe Weise formatiert. Es ist nicht möglich in einer Abfrage sowohl XML- Elemente als auch XML- Attribute festzulegen.

Optional können Sie mit dem Parameter "ROOT" ein einzelnes Element angeben, dass auf oberster Ebene dem Ergebnis hinzugefügt wird. Der Standardwert lautet "root".

Beispiel:

Die Abfrage

```
select lname as 'Name', anr as Artikelnummer', ldatum as 'Datum'
from lieferant a join lieferung b on a.lnr = b.lnr
where ldatum between '20.07.1990' and '10.08.1990'
for xml raw('Lief'), root('Info'), ELEMENTS;
```

liefert folgendes Ergebnis. Das ist ein Auszug.

```
<Info>
  ...
  <Lief>
    <Name>Schmidt</Name>
    <Artikelnummer>A05</Artikelnummer>
    <Datum>1990-08-01T00:00:00</Datum>
  </Lief>
```

```

<Lief>
    <Name>Schmidt</Name>
    <Artikelnummer>A06</Artikelnummer>
    <Datum>1990-07-23T00:00:00</Datum>
</Lief>

<Lief>
    <Name>Jonas</Name>
    <Artikelnummer>A01</Artikelnummer>
    <Datum>1990-08-02T00:00:00</Datum>
</Lief>
...
</Info>

```

8.4.2 Verwenden von FOR XML AUTO

Bei der Formatierung mit FOR XML AUTO wird eine verschachtelte XML- Struktur erstellt. Für jede Zeile wird ein Element mit dem Namen der Tabelle erzeugt, die in der SELECT- Abfrage verwendet wird. Außerdem wird für jede Tabelle in der Abfrage eine neue Ebene in die XML- Struktur eingefügt. Die Reihenfolge , in der die XML- Struktur verschachtelt wird, wird durch die Reihenfolge der Tabellen in der FROM- Klausel der Abfrage deklariert.

Beispiel:

Die Abfrage

```

select lname, anr, idatum
from lieferant join lieferung on lieferant.lnr = lieferung.lnr
where idatum between '20.07.1990' and '10.08.1990'
for xml auto;

```

liefert folgendes Ergebnis.

```

<lieferant Name="Schmidt">
    <lieferung Artikelnummer="A04" Datum="1990-07-25T00:00:00" />
    <lieferung Artikelnummer="A05" Datum="1990-08-01T00:00:00" />
    <lieferung Artikelnummer="A06" Datum="1990-07-23T00:00:00" />
</lieferant>
<lieferant Name="Jonas">
    <lieferung Artikelnummer="A01" Datum="1990-08-02T00:00:00" />
    <lieferung Artikelnummer="A02" Datum="1990-08-05T00:00:00" />
</lieferant>
<lieferant Name="Blank">
    <lieferung Artikelnummer="A02" Datum="1990-08-06T00:00:00" />
</lieferant>
<lieferant Name="Clark">
    <lieferung Artikelnummer="A02" Datum="1990-08-09T00:00:00" />
</lieferant>

```

Die Standardformatierung ist, wie bei XML RAW, attributorientiert, kann aber auch hier durch das Schlüsselwort ELEMENTS verändert werden.

Beispiel:

Die Abfrage

```

select lname, anr, idatum
from lieferant join lieferung on lieferant.lnr = lieferung.lnr
where idatum between '20.07.1990' and '10.08.1990'
for xml auto, ELEMENTS;

```

liefert folgendes Ergebnis. Das ist ein Auszug.

```
...
<lieferant>
    <Name>Schmidt</Name>
    <lieferung>
        <Artikelnummer>A04</Artikelnummer>
        <Datum>1990-07-25T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A05</Artikelnummer>
        <Datum>1990-08-01T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A06</Artikelnummer>
        <Datum>1990-07-23T00:00:00</Datum>
    </lieferung>
</lieferant>
<lieferant>
    <Name>Jonas</Name>
    <lieferung>
        <Artikelnummer>A01</Artikelnummer>
        <Datum>1990-08-02T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A02</Artikelnummer>
        <Datum>1990-08-05T00:00:00</Datum>
    </lieferung>
</lieferant>
...

```

Beachten Sie bei FOR XML AUTO folgende Aspekte zur Formatierung:

- Die XML- Struktur ist ein XML- Fragment weil es keinen Stammknoten bereitstellt. Damit ist es kein wohlgeformtes XML- Dokument. Arbeiten Sie mit dem Schlüsselwort ROOT.
- Alle Spalten sind auf ein und dieselbe Weise formatiert. Es ist nicht möglich in einer Abfrage sowohl XML- Elemente als auch XML- Attribute festzulegen.
- Es wird für jede Tabelle in der Abfrage eine neue Hierarchieebene in folgender Reihenfolge erstellt:
 - ★ Die erste Ebene wird der Tabelle zugeordnet, die die erste in der Abfrage deklarierte Spalte enthält, die zweite Ebene der Tabelle nächsten deklarierten Spalte und so weiter.
 - ★ Wenn Spalten in der Abfrage gemischt werden, sortiert XML AUTO die XML- Knoten neu, sodass alle Knoten, die zur selben Ebene gehören, unter demselben übergeordneten Knoten gruppiert werden.
- Im Gegensatz zu XML RAW stellt XML AUTO keine Möglichkeit zur Umbenennung zur Verfügung, sondern verwendet die Tabellen- und Spaltennamen oder, falls vorhanden, die Aliase.
- Die Formatierung wird zeilenweise angewandt. SQL Server unterstützt verschachtelte FOR XML- Abfragen um komplexe verschachtelte XML- Strukturen zu erstellen.

Um die Wohlgeformtheit zu erreichen sollten Sie ein Wurzelement mit dem Namen root erzeugen oder einen eigenen Element- Namen verwenden.

Beispiel:

Die Abfrage

```
select lname, anr, ldatum
from lieferant join lieferung on lieferant.lnr = lieferung.lnr
where ldatum between '20.07.1990' and '10.08.1990'
for xml auto, ELEMENTS, ROOT('LiefLief');
```

liefert folgendes Ergebnis. Das ist ein Auszug.

```
<LiefLief>
...
<lieferant>
    <Name>Schmidt</Name>
    <lieferung>
        <Artikelnummer>A04</Artikelnummer>
        <Datum>1990-07-25T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A05</Artikelnummer>
        <Datum>1990-08-01T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A06</Artikelnummer>
        <Datum>1990-07-23T00:00:00</Datum>
    </lieferung>
</lieferant>
<lieferant>
    <Name>Jonas</Name>
    <lieferung>
        <Artikelnummer>A01</Artikelnummer>
        <Datum>1990-08-02T00:00:00</Datum>
    </lieferung>
    <lieferung>
        <Artikelnummer>A02</Artikelnummer>
        <Datum>1990-08-05T00:00:00</Datum>
    </lieferung>
</lieferant>
...
</LiefLief>
```

8.4.3 Verwenden von FOR XML EXPLICIT

Wie in den vorangegangenen Kapiteln schon beschrieben, bieten die Modi RAW und AUTO kaum Möglichkeiten für die Steuerung des generierten XML- Codes. Größere Flexibilität bietet da der EXPLICIT Modus. Er ist aber viel aufwendiger beim Schreiben von Abfragen.

Die Select- Abfrage muss in einem ganz besonderen Muster geschrieben werden wenn Sie den EXPLICIT- Modus verwenden wollen. Dieses wird als universelle Tabelle bezeichnet.

Regeln für die Abfrageerstellung:

- Die erste Spalte muss die Tagnummer des aktuellen Elements bereitstellen und der Name der Spalte muss TAG lauten. Jedes aus dem Ergebnis konstruierte Element muss eine eindeutige Tagnummer besitzen.
- Die zweite Spalte muss die Tagnummer des übergeordneten Elements bereitstellen. Der Name dieser Spalte muss PARENT sein. Damit wird die Hierarchie der XML Abfrage bereitgestellt.
- Die Spaltennamen werden müssen einen Alias nach folgendem Muster bereitstellen.

ElementName!TagNummer!AttributName[!Direktive]

Beispiel einer universellen Tabelle:

| tag | parent | lieferant!1!lnr | lieferant!1!lname | artikel!2!anr | artikel!2!aname | artikel!2!datum |
|-----|--------|-----------------|-------------------|---------------|-----------------|-----------------|
| 1 | NULL | L02 | Jonas | NULL | NULL | NULL |
| 2 | 1 | L02 | NULL | A01 | Mutter | 02.08.1990 |
| 2 | 1 | L02 | NULL | A02 | Bolzen | 05.08.1990 |

- ElementName:** Ein allgemeiner Bezeichner des Elements im Ergebnis.
- TagNummer:** Ein eindeutiger, einem Element zugewiesener Tagwert. Dieser bestimmt mit hilfe der TAG - und PARENT- Spalte die Schachtelung der XML- Ausgabe.
- AttributName:** Der Name des zu konstruierenden Attributs im angegebenen Element. Attribute werden standardmäßig erstellt, wenn kein Wert für Direktive angegeben wird.
- Direktive:** Ist optional, und wird zum Bereitstellen zusätzlicher Informationen für das Konstruieren der XML- Ausgabe verwendet. Sie besitzt zwei Aufgaben.
Die erste besteht darin mit den Schlüsselwörtern ID, IDREF und IDREFS die Attributtypen zu überschreiben und dokumentierte Verknüpfungen zu erstellen.
Die zweite besteht darin zu bestimmen wie die Zeichenfolgedaten der XML- Ausgabe zugeordnet werden sollen.
- hide:** Zeigt an das die Spalte nicht in die XML- Struktur aufgenommen werden soll. Zum Beispiel für Spalten die nur zu Sortierungszwecken benötigt werden.
- element:** Macht aus dem XML- Attribut ein XML- Element. NULL- Marken werden ignoriert.
- elementxsinnil:** Macht aus dem XML- Attribut ein XML- Element. NULL- Marken werden nicht ignoriert.
- cdata:** Erstellt den Spaltenwert als XML- Kommentar in einem CDATA- Abschnitt.

Beispiel:

Für jedem Lieferanten soll der Name und die Lieferantennummer, sowie die Artikelnummer, der Artikelname und das Lieferdatum seiner gelieferten Artikel in eine XML- Struktur ausgegeben werden.

```

select distinct
    1 as tag, null as parent,
    lieferant.lnr as [lieferant!1!lnr],
    lname as [lieferant!1!lname],
    null as [artikel!2!anr],
    null as [artikel!2!aname],
    null as [artikel!2!datum]
from lieferant, lieferung
where lieferant.lnr = lieferung.lnr
union all
select 2 as tag, 1 as Parent,
    lieferung.lnr,
    null,
    lieferung.anr,
    aname,
    convert(char(10),ldatum,104)
from lieferung, artikel
where lieferung.anr = artikel.anr
order by [Lieferant!1!lnr], [artikel!2!aname]
for xml explicit, root('Lieferungen');
```

8.4.4 TYPE- Direktive

Wenn Daten, abgespeichert in SQL Server 2000 und früher, welche mit FOR XML ermittelt wurden, als XML weiterverarbeitet werden sollen, verwendet man die TYPE- Direktive. Mithilfe dieser Direktive wird das Ergebnis als XML- Datentyp anstatt des nvarchar(max) Datentyps zurückgegeben.

Diese Direktive ist optional wenn die Abfrage mit FOR XML ab SQL Server 2005 ausgeführt wird oder in einer Variablen vom Datentyp XML abgelegt wird.

Beispiel:

Nummern und Namen der Lieferanten sowie die Nummern der von ihnen gelieferten Artikel als XML- Dokument.

```
select lnr, lname,
       isnull((select * from lieferung
               where lieferung.lnr = lieferant.lnr
               for xml auto, type).query('<doc>{for $c in /lieferung
                                                 return <geliefert anr="{data($c/@anr)}"/>}</doc>'),
               '<doc>nicht geliefert</doc>') as 'Lieferungen'
  from lieferant
  for xml auto, type, root('doku');
```

Ergebnis:

```
<doku>
  <lieferant lnr="L01" lname="Schmidt">
    <Lieferungen>
      <doc>
        <geliefert anr="A01" />
        <geliefert anr="A02" />
        <geliefert anr="A03" />
        <geliefert anr="A04" />
        <geliefert anr="A05" />
        <geliefert anr="A06" />
      </doc>
    </Lieferungen>
  </lieferant>
  <lieferant lnr="L02" lname="Jonas">
    <Lieferungen>
      <doc>
        <geliefert anr="A01" />
        <geliefert anr="A02" />
      </doc>
    </Lieferungen>
  </lieferant>
  ...
  <lieferant lnr="L05" lname="Adam">
    <Lieferungen>
      <doc>nicht geliefert</doc>
    </Lieferungen>
  </lieferant>
</doku>
```

8.4.5 PATH- Direktive

Diese Methode ist neu in SQL Server 2005 und bietet Entwicklern die vollständige Kontrolle darüber, wie die XML- Struktur erzeugt wird. Dabei können einige Spalten als Attribute und andere als Elemente dienen. Jede Spalte wird unabhängig von der anderen konfiguriert.

- Jede Spalte bekommt einen Spaltenalias, der mitteilt wo dieser Knoten in der XML- Hierarchie zu finden ist.
- Hat eine Spalte keinen Alias, wird der Stammknoten <row> verwendet, ähnlich wie bei XML RAW.
- Spaltenaliase werden mit XPhat- Pseudoausdrücken deklariert.

Optionen für FOR XML PATH:

| Option | Beschreibung |
|-----------------------------|--|
| 'elementname' | Ein XML- Element, <elementname>, wird mit dem Inhalt der Tabellenspalte im Kontextknoten erstellt. |
| '@attributname' | Ein XML- Attribut, attributname, wird mit dem Inhalt der Tabellenspalte im Kontextknoten erstellt. |
| 'elementname/unterelement' | Ein XML- Element, <elementname> wird erstellt. Darunter ein weiteres XML- Element <unterelement> mit dem Inhalt der Tabellenspalte. |
| 'elementname/@attributname' | Ein XML- Element, <elementname> wird erstellt mit einem Attribut, attributname' mit dem Inhalt der Tabellenspalte. |
| text() | Der Inhalt dieser Spalte wird als Textknoten in die XML- Struktur eingefügt. |
| comment() | Der Inhalt dieser Spalte wird als Kommentar in die XML- Struktur eingefügt. |
| node() | Der Inhalt dieser Spalte wird so eingefügt, als ob kein Spaltenname angegeben wäre. |
| data() | Der Inhalt dieser Spalte wird als unteilbarer Wert behandelt. Ist das nächste Element in der Serialisierung auch ein unteilbarer Wert, wird ein Leerzeichen eingefügt. |

Beispiel:

```
select lnr as "@Nr", lname as "Lief_Detail/Name", lstadt as "Lief_Detail/Ort"
from lieferant
where status = 20
for xml path ('Lieferant')
```

Ergebnis:

```
<Lieferant Nr="L01">
  <Lief_Detail>
    <Name>Schmidt</Name>
    <Ort>Hamburg</Ort>
  </Lief_Detail>
</Lieferant>
<Lieferant Nr="L04">
  <Lief_Detail>
    <Name>Clark</Name>
    <Ort>Hamburg</Ort>
  </Lief_Detail>
</Lieferant>
```

8.5 Arbeit mit XML- Daten vom Datentyp XML

Neu seit SQL Server 2005 ist der Datentyp XML. Vorher war es nur möglich aus relationalen Daten XML- Strukturen zu erstellen. XML- Strukturen die der Entwickler in der Datenbank speichern wollte konnte er bis dahin nur in Spalten mit alphanumerischen Datentypen abspeichern.

8.5.1 Welche Speicherform für XML- Daten?

Die Art und Weise wie XML- Daten gespeichert werden hat Einfluß darauf, welche Aktionen zur Verfügung stehen.

8.5.1.1 Überlegungen für XML- Daten im XML- Format

Das speichern von Daten im XML- Format hat mehrere Vorteile:

- XML ist selbstbeschreibend, sodass Anwendungen XML- Daten verwenden können, ohne deren Struktur oder Schema zu kennen. Sie werden immer in einer hierarchischen Baumstruktur angeordnet.
- XML behält die Dokumentreihenfolge bei. Durch den hierarchischen Aufbau, ist das Beibehalten der Knotensortierung wichtig, da diese den Abstand zwischen den Knoten innerhalb der Baumstruktur vorschreibt.
- Durch die Verbindung mit einem XML- Schema wird eine Typ- und Strukturüberprüfung der XML- Daten sichergestellt.
- XML ist einfach durchsuchbar. Vielfältige Algorithmen können angewendet werden um die Baumstruktur nach bestimmten Werten zu durchsuchen. XQUERY und XPATH sind Abfragesprachen die zum Durchsuchen von XML- Strukturen entworfen wurden.
- XML- Daten sind erweiterbar. Es können Knoten eingefügt, geändert oder gelöscht werden.
- Ermöglicht das Speichern von bis zu 2 GB großen XML- Dokumenten.

8.5.1.2 Überlegungen für XML- Daten im Textformat

Mithilfe der Datentypen **char()**, **nchar()**, **varchar()**, **nvarchar()**, **varchar(max)**, **nvarchar(max)** oder **varbinary(max)** können XML- Daten in einer Textspalte gespeichert werden.

Vorteile zum speichern XML- Daten in Textspalten:

- Textgenauigkeit, alle Kommentare, Leerzeichen und ähnliches bleiben erhalten.
- Unabhängigkeit vom Funktionsumfang der Datenbank.
- Reduzierung der Arbeitsauslastung des Datenbanksystems während der Verarbeitung. Die gesamte Verarbeitung erfolgt in der mittleren Schicht (Anwendung).
- Beste Leistung für das Einfügen und Abrufen auf Dokumentebene. Das bedeutet, dass beim Ausführen von Vorgängen auf Knotenebene es erforderlich ist, mit dem vollständigen XML- Document zu arbeiten, da SQL Server nicht weiß, was in dieser Spalte gespeichert ist.

Es ergeben sich aber auch folgende Einschränkungen:

- Gestiegene Codierungskomplexität in der mittleren Schicht.
- Beim Suchen von daten muss immer das gesamte Dokument gelesen werden.
- Überprüfung Wohlgeformtheit und Typüberprüfung müssen in der mittleren Schicht ausgeführt werden.

8.5.2 XPath und XQuery

Seit der Einführung des XML- Datentyps stellt sich auch die Frage, wie man möglichst einfach auf bestimmte Informationen eines XML- Dokuments zugreifen kann.

Da es nicht die schnellste Methode ist, das gesamte Dokument auszulesen und auf dem Client weiterzuverarbeiten, ist in SQL Server eine Abfragesprache integriert die solche Abfragen möglich macht. Sie orientiert sich an anerkannten Standards für XML- Dokumente.

XQuery ist eine Abfragesprache, die strukturierte oder halbstrukturierte XML- Daten abfragen kann. Durch die Unterstützung für den xml- Datentyp können Dokumente in einer Datenbank gespeichert und dann mithilfe von XQuery abgefragt werden.

XQuery basiert auf der vorhandenen XPath 2.0- Abfragesprache. Damit ist es möglich, Inhalte aus einem XML- Dokument zu extrahieren und überarbeitet zurückzugeben.

8.5.2.1 XPath

Ist eine vom W3C definierte Sprache, um bestimmte Knoten eines XML- Dokumentes zu identifizieren. Zusätzlich besitzt XPath verschiedene Funktionen zum Umgang mit Zeichenketten, Boolschen Werten und numerischen Inhalten.

8.5.2.1.1 Pfadangaben

Um einen bestimmten Knoten in einem XML- Dokument zu ermitteln, können Elemente mit einem '/' getrennt dargestellt werden. Startpunkt der Beschreibung ist normalerweise immer die Wurzel des XML- Dokuments, die auch mit '.' beschrieben werden kann. Dieser kann in einem XPath- Ausdruck aber auch weggelassen werden, es wird dann davon ausgegangen, dass der Pfad relativ zum Wurzelknoten definiert ist.

Die Pfadangabe besteht grundsätzlich aus drei Teilen: der Achsenangabe, dem Knotentest und eventuell vorhandenen Bedingungen. Achsangabe und Knotentest werden immer mit einem doppelten Doppelpunkt getrennt und die Bedingung in eckige Klammer gesetzt.

Beispiel:

achse::knotentest[bedingung1][bedingung2]...

8.5.2.1.2 Achsangaben

Mithilfe der Achsangabe wird innerhalb der Struktur, die ein XML- Dokument darstellt. navigiert. Mögliche Achsangaben für XMLSQL 4.0 sind:

| Achse | Adressierte Knoten | Abkürzung |
|--------------------|---------------------------------------|--------------|
| child | direkt untergeordnet | keine oder * |
| parent | direkt übergeordnete | .. |
| self | der Referenzknoten selbst | . |
| descendant | untergeordnete | |
| descendant-or-self | untergeordnete oder der Knoten selbst | // |
| attribute | alle Attribute | |

Durch diese Angaben wird eine bestimmte Teilmenge der Knoten definiert, auf die der Knotentest und die weiteren Bedingungen angewendet werden.

Beispiel:

Informationen//descendant::BisherigeWohnadressen

Informationen/BisherigeWohnadressen//descendant-or-self:: BisherigeWohnadressen

Beide Angaben liefern den gesamten "BisherigeWohnadressen"- Knoten.

8.5.2.1.3 Knotentest

Dadurch wird ein Filter auf die Menge der Knoten angewandt, der auf den Namen und/oder auf dem Knotentyp basiert. Dafür stehen folgende Funktionen zur Verfügung:

| Knotentest | Beschreibung |
|--------------------------|--|
| <Elementname> | Filtert alle Elemente die den angegebenen Namen haben. |
| * | Alle Knoten (Elemente, Attribute oder Namensräume) |
| node() | Alle Knoten (Elemente, Attribute oder Namensräume) |
| processing-instruction() | Filtert alle Parserinformations- Knoten |
| text() | Filtert alle Textknoten |
| comment() | Filtert alle Kommentar- Knoten |

Beispiel:

Bezugnehmend auf die oben geänderte Tabelle Lieferant, sollen mit nachfolgender Abfrage alle bisherigen Wohnadressen des Lieferanten 'Schmidt' ausgelesen werden.

```
select
info.query('<Inhalt>{/Informationen/BisherigeWohnadressen/descendant::text()})
           </Inhalt>') as 'alte Wohnadressen'
from lieferant
where Inr = 'L01';
```

Es ist auch möglich, mehrere Achsangaben und Knotentests zu kombinieren.

```
select
info.query('<Inhalt>{descendant::BisherigeWohnadressen/child::node()})
           </Inhalt>') as 'alte Wohnadressen'
from lieferant
where Inr = 'L01';
```

8.5.2.1.4 Prädikate

Prädikate geben weitere Filterbedingungen für die Knoten an. Diese werden in eckige Klammern geschrieben und werden immer zu WAHR oder FALSCH ausgewertet.

Knotenfunktionen

| Funktion | Beschreibung |
|---------------------------------|--|
| number last() | Letzte Positionsnummer des Elements. |
| number position() | Aktuelle Position des Elements. |
| number count(node-set) | Anzahl der Knoten in der Ergebnismenge. |
| node-set id(object) | Selektiert ein Objekt nach seiner ID. |
| string local-name(node-set?) | Name des Knotens ohne Namensraumpräfix. |
| string namespace-uri(node-set?) | Name des Namensraumes für Element- und Attributknoten. |
| string name(node-set?) | Kompletter Name des Knotens samt evtl. Namensraumpräfix. |

Zeichenkettenfunktionen werden zurzeit von SQLXML 4.0 nicht unterstützt.

Boolsche Funktionen

| Funktion | Beschreibung |
|-------------------------|--|
| boolean boolean(object) | Wandelt das Objekt in ein Boolean: <ul style="list-style-type: none"> - Das Ergebnis ist falsch, wenn ein übergebener numerischer Wert NULL oder leer ist, sonst wahr. - Eine übergebene Knotenmenge ist falsch, wenn sie leer ist, sonst wahr. - Eine übergebene Zeichenkette ist falsch, wenn sie einen Länge 0 hat, sonst wahr. - Ob ein beliebiges anderes Objekt falsch oder wahr ist, wird durch den jeweiligen Typ definiert. |
| boolean not(boolean) | kehrt den übergebenen Boolean um. |
| boolean true() | Liefert immer wahr zurück. |
| boolean fals() | Liefert immer falsch zurück. |
| number number(object?) | Wandelt das Objekt in eine Zahl um: <ul style="list-style-type: none"> - Eine Zeichenkette wird in einen Zahl gewandelt, wenn sie mit Ziffern beginnt, auch wenn nicht- numerische Zeichen folgen, ansonsten wird sie zu NaN (not a number). - Ein Boolean wird wahr zu 1 und falsch zu 0. - Eine Knotenmenge wird erst mit der string(node-set)- Methode zu einer Zeichenkette gewandelt und anschließend wie oben beschrieben als Zeichenkette behandelt. - Ein beliebiges anderes Objekt wird je nach Typ behandelt. |

Der Parser untersucht nacheinander alle Knoten, die der Achse und dem Knotentest entsprechen. Wird das Prädikat für den aktuell untersuchten Knoten insgesamt WAHR, verbleibt der Knoten in der Ergebnismenge.

Beispiel:

Überprüfen ob Lieferant Schmidt mehr als eine bisherige Wohnadresse hat.

```
select
info.query('<Inhalt>{//Adresse}</Inhalt>') as 'alte Wohnadressen'
from lieferant
where lnr = 'L01' and
info.exist('Informationen/BisherigeWohnadressen[count(descendant::Adresse)>1]')=1;
```

Um eine bestimmte bisherige Wohnadressen zurückzugeben kann die Position des Knotens abgefragt werden. In folgender Abfrage werden zwei verschiedene Prädikate zum Ermitteln der Position des Elements verwendet.

```
select
info.query('//Adresse[1]') as 'erste alte Adresse',
info.query('//Adresse[position()=2]') as 'zweite alte Adresse'
from lieferant
where lnr = 'L01';
```

8.5.2.2 XQuery

XQuery ist eine Erweiterung von XPath, die den XPath- Ausdrücken folgende Funktionalität hinzufügt:

- ORDER BY zum Sortieren der Ergebnisse.
- Definition von Namensräumen.
- Erzeugen neuer Knoten.

Dafür definiert XQuery sogenannte FLWOR- Statements. Damit kann komplizierte Abfragelogik geschrieben werden, die eine Iteration über einen Satz mit einem Filter übereinstimmender Knoten durchführt. FLWOR steht dabei für FOR, LET, WHERE, ORDER BY und RETURN. Nicht alle Features von XQuery sind in SQL Server implementiert, insbesondere das LET- Statement ist nicht vorhanden.

8.5.2.2.1 FOR- Klausel

Diese Klausel wird wie die FROM- Klausel in Transact SQL verwendet.

Ein oder mehrere FOR- Ausdrücke können auf einen XQUERY- Ausdruck angewandt werden. Die Klausel deklariert Variablen, die eine Abfolge von Knoten als Ergebnis eines XPATH- Ausdrucks enthalten. Dies bildet die von XQUERY zu verarbeitende Eingabesequenz. Die FOR- Klausel führt eine Iteration über alle Elemente in der Knotenmenge durch. Für jedes Element gibt der FLWOR- Ausdruck ein Ergebnis zurück.

Beispiel:

Ausgeben aller bisherigen Wohnadressen des Lieferanten Schmidt.

```
select
info.query('for $i in Informationen/BisherigeWohnadressen return $i')
from lieferant
where Inr = 'L01';
```

8.5.2.2.2 WHERE- Klausel

Mit der Where Klausel wird in XQuery genauso verfahren wie in TSQL.

Die Klausel optional und verleiht dem XQUERY- Ausdruck die Fähigkeit, die Eingabesequenz zu filtern, die verschiedene Operatortypen (Bedingungs- und logische Vergleichsoperatoren) und Funktionen (Aggregat-, numerische und boolesche Funktionen) verwenden.

Beispiel:

Es sollen nur die Informationen der Lieferanten angezeigt werden deren Geburtsdatum vor dem 01.01.1979 war.

```
select
info.query('for $i in Informationen where $i/Geburtsdatum > "01.01.1979" return $i')
from lieferant
where Inr = 'L01';
```

Die Bedingung kann natürlich auch schon im XPath- Ausdruck für die FOR- Klausel einbetten, aber durch die Schreibweise mit der WHERE- Klausel wird das Ganze viel übersichtlicher. Leider ist die Übersichtlichkeit durch höhere Kosten bei der Abfrage erkauft.

8.5.2.2.3 ORDER BY- Klausel

Auch diese Klausel in XQuery entspricht der ORDER BY- Klausel in TSQL.

Sie ist optional und verleiht dem XQUERY- Ausdruck die Fähigkeit, die Ausgabesequenz in einer anderen Reihenfolge zu sortieren als die Eingabesequenz.

Beispiel:

Alle bisherigen Wohnorte des Lieferanten in absteigender alphabetischer Reihenfolge nach dem Ort zurückgeben.

```
select
info.query('for $i in //BisherigeWohnadressen/Adresse
            order by $i/ort[1] descending return $i')
from lieferant
where lnr = 'L01';
```

8.5.3 Methoden für Abfragen aus XML- Datentypen

Der Datentyp XML stellt umfangreiche Such- und Abfragemöglichkeiten für eine XML-Struktur bereit. Damit erhalten Entwickler die Möglichkeit eine XML- Instanz in eine andere XML- Instanz zu transformieren, einen Wert im SQL- Typsystem zu extrahieren, zu prüfen, ob Knoten und Werte in XML- Strukturen vorhanden sind, und eine Vorhandenen XML-Struktur durch das Hinzufügen, Aktualisieren oder Löschen bestehender Knoten zu ändern.

Dafür stellt dieser datentyp fünf Methoden zur Verfügung:

| | |
|-----------------|---|
| query() | Bietet die Möglichkeit, einen XPATH- oder XQUERY- Ausdruck auszuführen um das resultierende XML- Fragment zurück zu geben. |
| value() | Bietet die Möglichkeit, einen XPATH- oder XQUERY- Ausdruck auszuführen um das einzelnen skalaren Wert zurück zu geben, der in einen SQL- Typ konverteiert wird. |
| exist() | Bietet die Möglichkeit, einen XPATH- oder XQUERY- Ausdruck auszuführen um zu prüfen ob Knoten existieren. |
| modify() | Bietet die Möglichkeit XML- Daten zu bearbeiten. |
| nodes() | Bietet die Möglichkeit, einen XPATH- oder XQUERY- Ausdruck auszuführen um das resultierende XML- Fragment in ein Tabellenergebnis zu konverteieren. |

8.5.3.1 Die Methode query()

Mit dieser Methode können einfache XQuery- Abfragen an den XML- Datentyp übergeben. Dazu wird einer XML- Spalte oder -Variablen in Klammern die Abfrage in Hochkomma übergeben. Das Ergebnis ist eine nichttypisierte Instanz des Datentyps „xml“.

Beispiel:

Für den Lieferant L01 soll zusätzlich die Information zu seinen bisherigen Wohnadressen aus der Spalte Information ausgegeben werden.

```
select lname, lstadt,
information.query('Informationen/BisherigeWohnadressen/Adresse') as 'gewohnt'
from lieferant
where lnr = 'L01';
```

8.5.3.2 Die Methode value()

Diese Methode ruft den Inhalt eines XML- Elements oder eines XML- Attributes ab. Der zurückgegebene Wert muss ein skalarer Wert sein. Darum wird an den XQuery- Ausdruck einfach ein Index mit dem Wert 1 angehängt. Damit ist das erste Auftreten dieses Wertes gemeint. Wenn es weggelassen wird meldet SQL Server einen Fehler.

Anschließend wird der resultierende Wert in ein Transact SQL- Typ konvertiert.

Mit dieser Methode ist es leicht möglich XML- Werte in relationale Werte zu konvertieren.

Beispiel:

Zusätzlich sollen die PLZ und die Straßen der Firma des Lieferanten ausgegeben werden.

```
select lname, lstadt,
information.value('(/FirmenAdresse/plz)[1]', 'char(5)') as 'plz',
information.value('(/FirmenAdresse/strasse)[1]', 'varchar(50)') as 'strasse'
from lieferant
where lnr = 'L01';
```

8.5.3.3 Die Methode exist()

Mit der exist- Methode kann man nach bestimmten Elementen (Knoten) oder Attributen suchen.

Dieser Methode wird ein XQuery- Ausdruck als Parameter übergeben, der das gesuchte Element zurückliefert.

EXIST wertet sich zu 1 aus wenn das Element existiert und zu 0 wenn es im XML- Fragment nicht gefunden wurde.

Die Methode wird gewöhnlich in der Where- Klausel einer Abfrage verwendet.

Beispiel:

Zusätzliche Ausgabe der bisherigen Wohnadressen des Lieferanten, dessen Firma sich in einer bestimmten Stadt mit einer bestimmten Postleitzahl befindet.

```
select lname, lstadt,
information.query('<Adr>{//Adresse}</Adr>') as 'bisher gewohnt'
from lieferant
where information.exist('//FirmenAdresse[plz="20359"][ort="Hamburg"]') = 1;
```

8.5.3.4 Die Methode modify()

Mit Hilfe dieser Methode können Sie den XML- Datentyp mit XML- DML Funktionen manipulieren. Wie die klassische DML stehen auch hier die insert-, update- und delete- Anweisung zur Verfügung.

Diese Methode unterstützt die Änderung der Werte in den XML- Knoten und der Struktur des XML- Fragments.

8.5.3.4.1 INSERT

Das Einfügen von Elementen in ein bestehendes XML- Dokument geschieht durch die insert- Anweisung.

Syntax:

```
insert
  Ausdruck1
  (
    {[as first | as last | } into | after | before
    Ausdruck2
  )
```

Beispiele:

Einfügen eines neuen Elements mit einem Unterelement an die erste Stelle im XML- Dokument.

```
update lieferant
set information.modify('insert <Persönlich>
                           <Geburtsdatum>12.09.1977</Geburtsdatum>
                           </Persönlich>
as first into (/Informationen)[1]')
where lnr = 'L01';
```

Geburtsort vor das Geburtsdatum einfügen einfügen.

```
update lieferant
set information.modify('insert <Geburtsort>Weimar</Geburtsort>
as first into(//Persönlich)[1]')
where Inr = 'L01';
```

Vorname- Element nach dem Geburtsdatum einfügen.

```
update lieferant
set information.modify('insert <Vorname>Klaus</Vorname>
as last into(//Persönlich)[1]')
where Inr = 'L01';
```

8.5.3.4.2 DELETE

Mit delete können Elemente aus dem XML Dokument entfernt werden. Diese erwartet wieder einen XQuery- Ausdruck zur Selektion der zu löschenen Knoten, wobei es sich auch um Werte von Elementen oder Attribute handeln kann.

Besitzt der angegebene Knoten weitere Unterelemente, werden diese ebenfalls gelöscht.

Beispiele:

Löschen des alten Elements Geburtsdatum

```
update lieferant
set information.modify('delete /Informationen/Geburtsdatum')
where Inr = 'L01';
```

Das Attribut ID löschen

```
update lieferant
set information.modify('delete /Informationen/@ID')
where Inr = 'L01';
```

8.5.3.4.3 REPLACE VALUE OF

Damit ist es möglich, den Wert von Elementen zu ändern.

Die Konstruktion umfasst zwei Ausdrücke. der erste muss ein einzelner eindeutiger Knoten sein, der durch einen XQuery- Ausdruck definiert wird. Der zweite kann mithilfe konstanter Werte oder durch die Bereitstellung eines XQuery- Ausdrucks konstruiert werden, der eine Gruppe von Knoten zurückgibt.

Syntax:

```
replace value of
  Ausdruck1
with
  Ausdruck2
```

Beispiel:

Den Vornamen von Lieferant L01 ändern.

```
update lieferant
set information.modify('replace value of (//Persönlich/Vorname/text())[1]
with "Karsten")
where Inr = 'L01';
```

Den Wert des Attribut ID ändern

```
update lieferant
set information.modify('replace value of (/Informationen/@ID)[1]
with "1")'
where Inr = 'L01';
```

8.5.3.5 Die Methode nodes()

Damit können bestimmte Elemente innerhalb eines XML- Dokuments identifiziert werden und mit Schleifen durchlaufen werden.

Die Methode erzeugt zunächst ein Rowset mit den durch die XQuery identifizierten XML-Elementen.

Beispiel:

Auslesen aller Straßeneinträge in der XML- Spalte für L01.

```
declare @xmltest xml
select @xmltest = information from lieferant where Inr = 'L01'
select T.s.query('.') as 'Strassen' from @xmltest.nodes('//strasse') t(s);
```

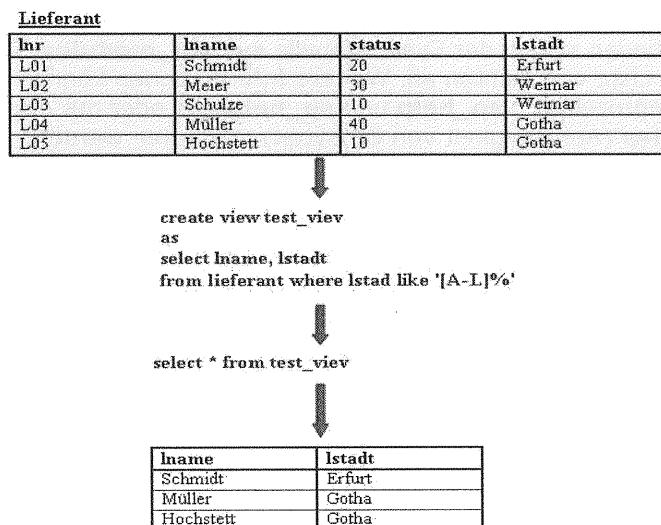

9 Sichten

Eine Sicht (View) ist eine virtuelle Tabelle, oder auch eine aus einer Datenbank abgeleitete Sichtweise auf den Datenbestand, deren Inhalt durch eine Abfrage definiert wird. Wie eine echte Tabelle besteht auch eine Sicht aus einem Satz benannter Spalten und Zeilen mit Daten. Man spricht auch von einer Benutzersicht.

Die Sicht wird jedoch nicht wie Tabellen auf interner Ebene durch Daten repräsentiert sondern nur durch ihre DML- Anweisung. Die Zeilen und Spalten mit Daten stammen aus Tabellen, auf die in der die Sicht definierenden Abfrage verwiesen wird. Diese Datenzeilen und -spalten werden dynamisch erstellt, wenn auf die Sicht verwiesen wird.

Eine Sicht dient als horizontaler und/oder vertikaler Filter für die zugrunde liegenden Tabellen, auf die in der Sicht verwiesen wird. Durch Sichten werden unterschiedlichen Benutzergruppen die Daten einer Datenbank unter verschiedenen Blickwinkeln präsentiert.

Die Abfrage, die die Sicht definiert, kann Daten aus einer oder mehreren Tabellen oder aus anderen Sichten in der aktuellen Datenbank oder anderen Datenbanken verwenden. Sie im Grunde nichts anderes, als eine gespeicherte Select- Abfrage.



Beispiele für häufig verwendete Sichten sind:

- Eine Teilmenge von Zeilen oder Spalten einer Basistabelle.
- Eine Vereinigung von zwei oder mehreren Basistabellen mit dem UNION- Operator.
- Verknüpfung von zwei oder mehreren Basistabellen mit dem JOIN- Operator.
- Eine statistische Zusammenfassung einer Basistabelle.
- Eine Teilmenge einer anderen Sicht oder eine Kombination von Sichten und Basistabellen.

Vorteile von Sichten sind:

- Konzentrieren der Daten für den Benutzer. Sichten schaffen eine kontrollierte Umgebung, die den Zugriff auf bestimmte Daten ermöglicht, während sie andere Daten verbirgt. Der Benutzer verwendet die Sicht wie eine Tabelle.
- Sichten verbergen die Komplexität des Datenbankentwurfs vor dem Benutzer. Dies bietet Entwicklern die Möglichkeit, den Entwurf zu ändern, ohne dass sich dies auf den Benutzerdialog mit der Datenbank auswirkt.
- Sichten vereinfachen die Verwaltung von Benutzerberechtigungen. Statt Benutzern die Berechtigungen auf alle von einer Abfrage benutzten Tabellen zu geben, erteile ich ihnen nur noch Berechtigungen auf die abgeleitete Sicht.

- Serverseitige Programmierung und Speicherung von umfangreichen Abfragen auf verschiedene Tabellen die miteinander verknüpft werden. Bereitstellung dieser Abfragen für Tools wie Reporting Services oder Crystal Reports.
- Mit Sichten kann die Leistung des DBS verbessert werden indem das Ergebnis komplexer Abfragen abgespeichert werden kann. Andere Abfragen können diese Ergebnisse verwenden.
- Änderungen von Objekten verbergen. Zum Beispiel bei einer neuen Version einer Datenbank- Applikation sind häufig Änderungen an der Tabellenstruktur notwendig. Alten Client- Programmen kann das aber Schwierigkeiten bereiten. Damit sie weiterarbeiten können, können die alten Objektnamen durch Views nach außen weiter bestehen bleiben.
- Aufteilen von großen Tabellen auf verschiedene Server. Durch Sichten (z.B. partitionierte Sichten) können die Daten wieder zu einer Gesamtabelle zusammengeführt werden.

9.1 Erstellen von Sichten

Die CREATE VIEW- Anweisung darf nicht mit anderen SQL- Anweisungen in einem Stapel stehen.

Die Namen der Sichten werden mit der Katalogsicht **sys.objects** angezeigt. Die DML- Anweisung der Sichten finden sich in der Systemtabelle **sys.sql_modules**. Die Katalogsicht **sys.views** zeigt eine Liste von Sichten an. Weitere Informationen können mit den gespeicherten Systemprozeduren **sp_help** und **sp_helptext** oder mit der Katalogsicht **sys.sql_dependencies** (Abhängigkeit von Objekten) abgerufen werden.

Beim erstmaligen Ausführen einer Sicht wird nur ihre Abfragestruktur im Prozedurcache gespeichert und bei jedem Zugreifen auf die Sicht wird ihr Ausführungsplan erneut kompiliert. , Sichten bedeuten einen zusätzlichen Leistungsaufwand da sie dynamisch aufgelöst werden.

Sichten können von Mitgliedern der Rolle **sysadmin**, **db_owner** oder **db_ddladmin** sowie von Benutzern mit CREATE VIEW- Berechtigung und ALTER SCHEMA- Berechtigung, für das Schema in der die Sicht erstellt wird, erstellt werden. Darüber hinaus benötigen sie SELECT- Berechtigungen für alle Tabellen und Sichten, auf die in ihren Sichten verwiesen wird.

Syntax:

```
create view viewname [(spaltenname, ...)]
[with {{encryption} [, schemabinding] [, view_metadata]}]
as
select_anweisung
[with check option]
```

Einschränkungen:

- Eine Sicht darf auf maximal 1024 Spalten verweisen.
- Sichten können nur in der aktuellen Datenbank erstellt werden. Die Tabellen und Sichten, auf die die neue Sicht verweist, können sich jedoch in anderen Datenbanken oder sogar auf anderen Servern befinden, wenn die Sicht mithilfe verteilter Abfragen definiert wurde.
- Die Namen von Sichten müssen den Regeln für Bezeichner entsprechen und innerhalb der Datenbank eindeutig sein
- Sichten können auf der Grundlage von anderen Sichten oder von Prozeduren, die auf Sichten verweisen, erstellt werden. Sichten können bis zu 32 Ebenen geschachtelt werden.
- An Sichten können keine Regeln oder DEFAULT gebunden werden.
- Sichten können keine AFTER- Trigger, sondern nur INSTEAD OF- Trigger zugeordnet werden.

- Die Abfrage, die die Sicht definiert, darf nicht die ORDER BY- Klausel, die Option-Klausel oder das INTO- Schlüsselwort enthalten. Sie kann die ORDER BY- Klausel nur verwenden wenn sie auch den Schlüsselwort TOP- Operator verwendet. Sie darf keinen Verweis auf eine temporäre Tabelle oder Tabellenvariable enthalten.
- Für indizierte Sichten können Volltextindexdefinitionen definiert werden.
- Das Erstellen temporärer Sichten oder das Erstellen von Sichten auf temporäre Tabellen oder Tabellenvariable ist nicht möglich.
- Sichten oder Tabellen, die Bestandteil einer mit der SCHEMABINDING- Klausel erstellten Sicht sind, können erst dann gelöscht werden, wenn die entsprechende Sicht gelöscht oder geändert wird, so dass die Schemabindung nicht mehr vorhanden ist. Auch schlagen ALTER TABLE- Anweisungen in Tabellen fehl, die Bestandteil dieser Sichten sind, falls diese Anweisungen die Sichtdefinition betreffen.
- Sie können keine Volltextabfragen auf einer Sicht ausführen, obwohl eine Sichtdefinition eine Volltextabfrage einschließen kann, wenn die Abfrage auf eine Tabelle verweist, die für die Volltextindizierung konfiguriert wurde.

Unter den folgenden Bedingungen muss für jede Spalte in der Sicht ein Name angegeben werden:

- Spalten in der Sicht sind aus einem arithmetischen Ausdruck, einer integrierten Funktion oder einer Konstanten abgeleitet.
- Zwei oder mehr Spalten der Sicht würden denselben Namen haben, weil die Definition der Sicht eine Verknüpfung enthält und die Spalten aus zwei oder mehr unterschiedlichen Tabellen denselben Namen besitzen.

Beispiel:

Eine Sicht der die Artikelnummer, den Artikelnamen und den Lagerort von Artikeln anzeigt.

```
create view art_einf
as
select anr, aname, astadt
from artikel;

select * from art_einf;
```

Dieselbe Sicht mit der Option „SCHEMABINDING“, dadurch kann die Tabelle „ARTIKEL“ nicht gelöscht werden bevor nicht die Sicht gelöscht wurde.

```
create view art_einf
with schemabinding
as
select anr, aname, astadt
from dbo.artikel;
```

9.2 Ändern von Sichten

Mit der ALTER VIEW- Anweisung kann eine vorhandene Sicht geändert werden. ALTER VIEW wirkt sich nicht auf abhängige gespeicherte Prozeduren oder Trigger aus und ändert keine Berechtigungen.

Die Syntax der Anweisung ist identisch mit der CREATE VIEW- Anweisung.

Beispiel:

Ändern der Sicht im vorangegangenen Beispiel. Die Schemabinding- Klausel soll entfernt werden.

```
alter view art_einf
as
select anr, aname, astadt
from dbo.artikel;
```

9.3 Löschen von Sichten

Es können eine Sicht oder mehrere Sichten gleichzeitig gelöscht werden. Jede Sicht deren zugrunde liegende Tabelle oder Sicht mit der DROP TABLE- Anweisung bzw. DROP VIEW- Anweisung gelöscht wurde, muss einzeln gelöscht werden.

Sichten können nur von ihrem Besitzer gelöscht werden. Der Systemadministrator oder der Datenbankbesitzer kann jedoch durch Angabe des Besitzernamens in der DROP VIEW- Anweisung jedes beliebige Objekt löschen.

Syntax:

```
drop view viewname [, ...]
```

Beispiel:

```
drop view art_einf;
```

9.4 Ändern von Daten über eine Sicht

Die Anweisungen INSERT, UPDATE und DELETE können nur eingeschränkt auf Sichten angewendet werden.

Einschränkungen:

- wenn eine Sicht aus mehreren Tabellen abgeleitet ist, dann dürfen nur Reihen eingefügt werden die den Reihen einer einzigen der der Sicht zugrunde liegenden Tabellen entspricht.
- Keine Spalte der Sicht darf eine Aggregatfunktion oder andere Funktion enthalten, oder von einer Konstanten oder einem arithmetischen Ausdruck abgeleitet sein.
- In der SELECT- Anweisung der Sicht darf kein DISTINCT stehen.
- Die Sicht darf kein GROUP BY oder HAVING enthalten.
- WITH CHECK OPTION kann nicht angegeben werden wenn die Sicht den TOP- Operator verwendet.

9.4.1 Insert

Das Einfügen von Daten über eine Sicht bedeutet tatsächlich das Einfügen in die der Sicht zugrunde liegende Basistabelle.

Die Sicht über den Sie Daten einfügen wollen muss alle PS- Spalten der Basistabelle anzeigen und alle Spalten für die NOT NULL festgelegt und kein DEFAULT- Wert definiert wurde.

Durch INSERT- Anweisungen über eine Sicht sollten auch nur solche Datensätze aufgenommen werden, die von dieser Sicht auch wieder gelesen werden können.

Beispiel:

```
create view art_ort
as
select anr, aname, amenge, astadt
from artikel
where astadt = 'Hamburg';
go
```

Nachfolgende INSERT- Anweisung nimmt den Datensatz zwar auf aber kann von dieser Sicht nicht wiedergegeben werden.

```
insert into art_ort values ('A07', 'Kugellager', 200, 'Weimar');
```

Damit über eine Sicht nur solche Datensätze aufgenommen werden können die von dieser Sicht auch wieder angezeigt werden können, muss die Option WITH CHECK OPTION in der Deklaration der Sicht verwendet werden.

Beispiel:

```
create view art_ort
as
select anr, aname, amenge, astadt
from artikel
where astadt = 'Hamburg'
with check option;
go
```

Nachfolgende INSERT- Anweisung wird nun vom DBMS abgewiesen.

```
insert into art_ort values ('A08', 'Unterlegscheibe', 400, 'Weimar');
```

Hinweis: Die WITH CHECK OPTION- Klausel überprüft den Datensatz der über die Sicht aufgenommen werden soll, ob die entsprechenden Spaltenwerte die Bedingungen der WHERE- Klausel der Sicht einhalten.

9.4.2 Update

Das Ändern von Datenwerten über eine Sicht bedeutet tatsächlich das Ändern in der der Sicht zugrunde liegenden Tabelle.

Beispiel:

```
create view art_ort
as
select anr, aname, amenge, astadt
from artikel
where astadt = 'Hamburg'
with check option;
go

update art_ort
set amenge = 250
where anr = 'A07';
```

Tatsächlich wird in der Tabelle ARTIKEL folgende Änderung vorgenommen.

```
update artikel
set amenge = 250
where astadt = 'Hamburg'
and anr = 'A07';
```

Die WIHT CHECK OPTION- Klausel hat bei UPDATE- Anweisungen über eine Sicht die gleiche Aufgabe wie bei einer INSERT- Anweisung.

Beispiel:

```
update art_ort
set astadt = 'Erfurt'
where anr = 'A07';
```

Diese Anweisung wird mit einer Fehlermeldung abgewiesen, da eine Änderung vorgenommen werden soll, welche die Bedingung der WHERE- Klausel der Sicht nicht erfüllt.

9.4.3 Delete

Bei einer DELETE- Anweisung über eine Sicht werden tatsächlich die Datensätze der der Sicht zugrunde liegenden Tabelle gelöscht.

Beispiel:

```
create view art_ort
as
select anr, aname, amenge, astadt
from artikel
where astadt = 'Hamburg';
go
```

Nachfolgende Anweisung löscht alle Artikel die in Hamburg lagern.

```
delete from art_ort;
```

Nachfolgende Anweisung löscht alle Artikel die in Hamburg lagern und deren Artikelname mit einem "N" beginnt.

```
delete from art_ort
where aname like 'N%';
```

Obwohl Sichten verwendet werden können um Daten zu manipulieren, werden Sichten fast nie für diesen Zweck verwendet. Hier sind gespeicherte Prozeduren die bessere Wahl. Sie sind flexibler und schneller.

9.5 Indizierte Sichten

Wenn für eine Sicht ein gruppierter Index erstellt wurde, spricht man von einer indizierten Sicht. Dabei wird das Ergebnis (Resultset) einer Sicht auf der Blattebene des Indexes gespeichert. SQL Server kann schnell einen Verweis zum Index herstellen um die Daten abzurufen. Änderungen an den Daten in den Basistabellen werden in den Daten wiedergegeben die in der indizierten Sicht gespeichert sind.

Es werden Sichten indiziert die Aggregatfunktionen verwenden. Dadurch werden die Aggregatwerte physisch in der Datenbank materialisiert. Diese indizierten Sichten können die Abfrageleistung verbessern.

Die Möglichkeit Views zu indizieren, steht nur in der Enterprise Edition von SQL Server 2005 zur Verfügung

Eine Sicht muss die folgenden Anforderungen erfüllen, damit Sie einen gruppierten Index dafür erstellen können:

- Der **erste** für die Sicht erstellte Index muss ein **gruppierter eindeutiger Index** sein.
- Die Optionen ANSI_NULLS und QUOTED_IDENTIFIER müssen beim Ausführen der CREATE VIEW-Anweisung auf ON festgelegt sein.
- Die Option ANSI_NULLS muss für die Ausführung aller CREATE TABLE-Anweisungen, die Tabellen erstellen, auf die die Sicht verweist, auf ON festgelegt sein.
- Die Sicht darf nicht auf andere Sichten verweisen, sondern nur auf Basistabellen.
- Alle Basistabellen, auf die die Sicht verweist, müssen in derselben Datenbank wie die Sicht vorhanden sein.
- Die Sicht muss mit der Option **SCHEMABINDING** erstellt werden.

- Auf jeden Fall muss die SQL- Aggregatfunktion **COUNT_BIG** mit einbezogen werden, auch wenn sie nicht benötigt wird.
- Benutzerdefinierte Funktionen, auf die in der Sicht verwiesen wird, müssen mit der Option SCHEMABINDING erstellt werden.
- Auf Tabellen und benutzerdefinierte Funktionen muss mit zweiteiligen Namen in der Sicht verwiesen werden.
- Alle Funktionen, auf die Ausdrücke in der Sicht verweisen, müssen deterministisch sein.

Beispiel:

```
create view lief_menge
with schemabinding
as
select lnr, sum(lmengen) as 'Liefermenge', count_big(*) as 'Zähler'
from dbo.lieferung
group by lnr;

go

create unique clustered index lief_menge_idx on dbo.lief_menge(lnr);
```

Indizierte Sichten haben aber durch den zusätzlichen Aufwand für das Verwalten des Indexes auch Nachteile. Verwenden Sie darum indizierte Sichten wenn:

- Höhere Abfrageleistung überwiegt den Verwaltungsaufwand.
- Daten in den Basistabellen werden selten aktualisiert.
- Die Sicht verarbeite eine große Menge von Daten mit komplexen Operatoren und viele User greifen häufig darauf zu.

9.6 Partitionierte Sichten

Eine partitionierte Sicht verknüpft horizontale partitionierte Daten in mehreren Tabellen die sich auf einem oder mehreren Servern befinden. Es werden die Ergebnisse der Select-Anweisungen mit einer UNION ALL- Klausel zu einem Ergebnis zusammengefasst.

SQL Server 2008 R2 unterscheidet zwischen lokalen und verteilten partitionierten Sichten. Bei einer lokalen partitionierten Sicht befinden sich alle beteiligten Tabellen und die Sicht auf derselben Instanz von SQL Server. Bei einer verteilten partitionierten Sicht befindet sich mindestens eine der beteiligten Tabellen auf einem anderen Server. SQL Server 2008 R2 unterscheidet darüber hinaus zwischen partitionierten Sichten, die aktualisiert werden können, und Sichten, bei denen es sich um schreibgeschützte Kopien der zugrunde liegenden Tabellen handelt.

Partitionierte Sichten werden dann benötigt wenn die Daten, einer Gruppe von Informationen, aus Gründen der Arbeitslast auf mehrere Server verteilt werden. Oder bei einer Firma die mehrere Regionalbüros besitzt welche alle gleiche Informationen in einer Datenbank ablegen.

Beispiel:

```
create view lieferant_ges
as
select *
from SQLServerErfurt.dbo.lieferant
union all
select *
from SQLServerGotha.dbo.lieferant;
```

Bei dieser partitionierten Sicht befindet sich mindestens eine der zugrundeliegenden Tabellen auf einem anderen (remote) Server.

10 SQL Server Datenbanken

Eine Datenbank in SQL Server besteht aus einer Sammlung von Tabellen mit Daten sowie anderen Objekten, wie z. B. Sichten, Indizes, gespeicherten Prozeduren und Triggern, die definiert wurden, um die mit den Daten durchgeführten Aktivitäten zu unterstützen.

Datenbank

Eine Datenbank ist insofern mit einer Datendatei vergleichbar, als sie einen Speicherort für Daten darstellt. Die Informationen werden dem Benutzer jedoch nicht direkt zur Verfügung gestellt, sondern der Benutzer führt eine Anwendung (DBMS) aus, die auf die Daten in der Datenbank zugreift und die Daten für den Benutzer in einer verständlichen Form darstellt.

Datenbanksysteme sind leistungsfähiger als Datendateien, da die Daten besser organisiert sind. Eine sorgfältig entworfene Datenbank enthält keine Redundanzen, die vom Benutzer oder der Anwendung gleichzeitig aktualisiert werden müssen. Verwandte Datenelemente sind in einer einzelnen Struktur oder einem Datensatz gruppiert, und zwischen diesen Strukturen und Datensätzen können Beziehungen (Foreign Key) definiert werden.

Ein Datenbanksystem besteht in der Regel aus zwei Teilen: zum einen aus den Dateien, die die physische Datenbank enthalten, und zum anderen aus der Software des DBMS, die von Anwendungen für den Zugriff auf die Daten verwendet wird. Das DBMS ist für das Erzwingen der Datenbankstruktur verantwortlich, wie z. B.:

- Verwalten von Beziehungen zwischen den Daten in der Datenbank.
- Sicherstellen, dass die Daten korrekt gespeichert werden und die Regeln, die die Beziehungen zwischen den Daten definieren, nicht verletzt werden.
- Wiederherstellen aller Daten in einen bekannten Konsistenzzustand im Falle von Systemfehlern.

Relationale Datenbank

Obwohl es verschiedene Möglichkeiten gibt, Daten in einer Datenbank zu organisieren, stellt eine relationale Datenbank eines der effizientesten Verfahren dar. In relationalen Datenbanksystemen wird die mathematische Mengenlehre auf das Problem des effizienten Organisierens von Daten angewendet. In einer relationalen Datenbank werden Daten zu Tabellen zusammengestellt (den so genannten Relationen im Relationen Modell).

Eine Tabelle stellt eine Klasse von Objekten dar, die für eine Organisation relevant sind. Jede Tabelle besteht aus Spalten und Zeilen (im Relationen Modell Attribute und Tupel genannt). Jede Spalte stellt ein Attribut des Objekts dar, das von der Tabelle dargestellt wird. Jede Zeile stellt eine Instanz des Objekts dar, das von der Tabelle dargestellt wird.

Wenn Daten in Form von Tabellen organisiert werden, gibt es in der Regel viele verschiedene Möglichkeiten zum Definieren von Tabellen. Das relationale Datenbankmodell definiert einen Prozess, der Normalisierung genannt wird, der sicherstellt, dass die Daten effizient durch die Gruppe der definierten Tabellen organisiert werden.

Bevor Sie Objekte in einer Datenbank erstellen können, müssen Sie die Datenbank erstellen.

Sie können die Einstellungen und die Konfiguration der Datenbank ändern, dazu gehört das Erweitern oder Verkleinern der Datenbank oder das Festlegen der zum Erstellen der Datenbank verwendeten Dateien.

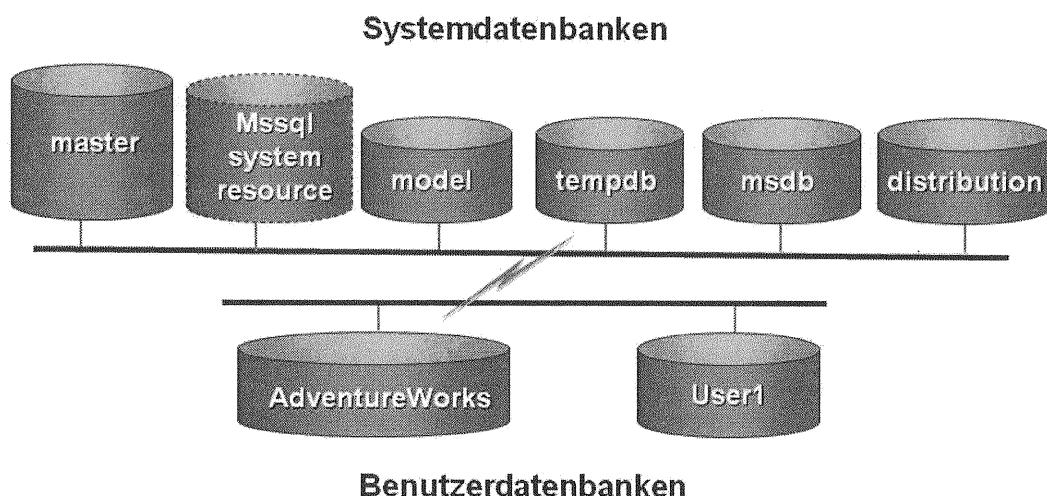
Definition: Als Datenbank werden die inhaltlich zusammengehörenden, von einem Datenbankmanagementsystem verwalteten Daten bezeichnet.

SQL Server kann mehrere Datenbanken unterstützen. Die Daten, die in den einzelnen Datenbanken gespeichert werden, können mit den Daten anderer Datenbanken in Beziehung stehen. Ein Server kann z. B. eine Datenbank enthalten, in der Personaldaten gespeichert werden, und über eine andere Datenbank verfügen, in der Produktbezogene Daten gespeichert werden.

10.1 Datenbanktypen

Jedes Datenbanksystem von SQL Server weist zwei Datenbanktypen auf:

- Systemdatenbanken
- Benutzerdatenbanken



Mit den Systemdatenbanken betreibt und verwaltet SQL Server das System. SQL Server kann eine oder mehrere Benutzerdatenbanken enthalten.

Bei der Installation des Servers werden die Systemdatenbanken erstellt. Die Datenbank „Distribution“ wird erst installiert wenn der SQL Server für die Replikation eingerichtet wird.

10.2 Systemdatenbanken:

mssqlsystemresources:

Diese schreibgeschützte Datenbank ist im Management Studio nicht sichtbar. Im Standardverzeichnis der Instanz im Ordner "BINN" liegen die Dateien dieser Datenbank.

In dieser Datenbank befinden sich die Systemobjekte von SQL Server. Systemobjekte werden physikalisch in dieser Datenbank gespeichert, logisch jedoch im **sys- Schema** jeder Datenbank angezeigt. Sie enthält Code der gespeicherten Prozeduren, Sichten und

Funktionen, sowie die Definition der Systemtabellen. Sie entspricht im Prinzip einem Container für ausführbaren Code, ähnlich einer DLL.

Dadurch, dass die Systemobjekte an einem zentralen Ort liegen, wird zum Beispiel das Einspielen von Servicepacks erheblich vereinfacht und schneller.

SQL Server kann die Ressourcendatenbank nicht sichern. Sie können eine eigene Datei- oder Datenträgergestützte Sicherung der Datei erstellen, indem Sie die Datei **mssqlsystemresource.mdf** als Binärdatei (EXE) anstatt als Datenbankdatei behandeln. SQL Server kann jedoch nicht zum Wiederherstellen der Sicherungen verwendet werden. Die Wiederherstellung einer Sicherungskopie von **mssqlsystemresource.mdf** kann nur manuell erfolgen.

master:

In der **master**- Datenbank werden alle Informationen auf Systemebene für ein SQL Server-Instanz aufgezeichnet. In der Datenbank werden Anmeldekonten, Endpunkte und Konfigurationseinstellungen für die SQL Server Instanz aufgezeichnet. In **master** werden das Vorhandensein aller anderen Datenbanken und der Speicherort der primären Dateien aufgezeichnet, die die Initialisierungsinformationen für Benutzerdatenbanken enthalten.

Systemobjekte werden nicht mehr in der **master**- Datenbank gespeichert. Stattdessen werden sie in der **mssqlsystemresource**- Datenbank gespeichert

tempdb:

Die **tempdb** nimmt alle temporären Benutzerobjekte, lokale und globale temporäre Tabellen, temporär gespeicherten Prozeduren, Tabellenvariablen und Cursor, auf. Sie wird darüber hinaus in allen anderen Situationen verwendet, in denen temporärer Speicherplatz benötigt wird, z.B. für Arbeitstabellen, die von SQL Server bei Mengen- und Bereichsoperatoren erzeugt werden.

Die **tempdb** ist eine globale Ressource; die temporären Tabellen und temporär gespeicherten Prozeduren für alle Benutzer, die eine Verbindung zum System hergestellt haben, in dieser Datenbank speichert.

Die **tempdb** wird bei jedem Start von SQL Server neu erstellt. Die **tempdb** wird nach Bedarf automatisch vergrößert.

Da diese Datenbank sehr häufig von vielen Abfragen und Operationen verwendet wird, sollten Sie die Verschiebung dieser Datenbank auf ein dezidiertes Laufwerk in Betracht ziehen. Dadurch kann die Leistung des DBMS erheblich verbessert werden.

model:

Die **model**- Datenbank wird als Vorlage für alle Datenbanken verwendet, die in einem System erstellt werden. Wenn eine CREATE DATABASE- Anweisung ausgegeben wird, wird der erste Teil der Datenbank erstellt, indem der Inhalt der **model**- Datenbank kopiert wird. Anschließend wird der verbleibende Teil der neuen Datenbank mit leeren Seiten gefüllt.

msdb:

Die **msdb**- Datenbank umfasst wichtige Informationen für die Administration von Datenbanken und wird vom SQL Server-Agenten verwendet, um Termine für Warnungen und Aufträge zu planen und Operatoren aufzuziehen. Sie enthält außerdem Informationen darüber wann welche Sicherung auf welchem Medium gelaufen ist und macht Vorschläge zur Rücksicherung der Datenbank.

distribution:

Speichert für die Replikation verwendete Chronik- und Transaktionsdaten. Diese Datenbank existiert nur wenn die Replikation eingerichtet wurde.

10.3 Server- und Datenbankinformationen abfragen

10.3.1 Systemtabellen - Systemsichten

Systeminformationen werden in den Systemtabellen der SQL Server Instanz abgelegt. Im Gegensatz zum SQL Server 2000 kann in der aktuellen Version nicht mehr direkt auf die Systemtabellen zugegriffen werden. Für die Abfrage von Systeminformationen stehen Systemsichten und Systemfunktionen zur Verfügung.

Die Basistabellen tragen solche Namen wie **sys.sysdbreg** (Informationen über alle Benutzerdatenbanken einer Instanz) aus der **master**- Datenbank.

Eine Abfrage auf diese Tabellen sind nur durch Verwendung einer "DEDICATED ADMINISTRATOR CONNECTION (DAC)" möglich und nur als äußerstes Mittel für den Notfall gedacht.

Alle Datenbanken enthalten einen gemeinsamen Satz an Systemtabellen. Nur in der **master**- Datenbank und anderen Systemdatenbanken sind ein paar zusätzliche für den Systemkatalog enthalten.

SQL Server 2012 bietet die folgenden Sammlungen an Systemsichten (früher Systemtabellen), die Metadaten offen legen:

- **Katalogsichten**
- **Kompatibilitätssichten**
- **Dynamische Verwaltungssichten und -funktionen**
- **Informationsschemasichten**
- **Replikationssichten**
- **Notification Services-Sichten**

10.3.1.1 Katalogsichten

Katalogsichten geben Informationen zurück, die von dem SQL Server 2012 Datenbankmodul verwendet werden. Es ist die allgemeinste Schnittstelle um Katalogmetadaten darzustellen und die effizienteste Methode zum Abrufen, Transformieren und Präsentieren dieser Informationen in eine benutzerdefinierte Form. Alle für Benutzer verfügbaren Katalogmetadaten werden über Katalogsichten verfügbar gemacht.

Einige Katalogsichten erben Zeilen von anderen Katalogsichten. So erbt beispielsweise die **sys.tables**- Katalogsicht von der **sys.objects**- Katalogsicht. Die **sys.objects**- Katalogsicht wird als Basissicht bezeichnet, und die **sys.tables**- Sicht wird abgeleitete Sicht genannt.

Katalogsichten sind dem Schema "**SYS**" zugeordnet und stehen in jeder Datenbank bereit.

Die Katalogsichten in SQL Server 2012 wurden in den folgenden Kategorien organisiert:

- **CLR-Assemblykatalogsichten**
- **Datenbanken und Dateikatalogsichten**
- **Katalogsichten des Datenbank-Spiegelungszeugen**
- **Datenspeicher und Volltextkatalog-Sichten**
- **Endpunkte-Katalogsichten**
- **Katalogsichten für erweiterte Eigenschaften**
- **Verbindungsserver-Katalogsichten**
- **Meldungskatalogsichten (Fehlermeldungen)**
- **Katalogsichten für Objekte**
- **Katalogsichten für Partitionsfunktionen**
- **Katalogsichten für Skalartypen**
- **Schema-Katalogsichten**
- **Sicherheitskatalogsichten**
- **Service Broker-Katalogsichten**
- **Katalogsichten für die serverweite Konfiguration**
- **Katalogsichten für XML-Schemas (XML-Typ)**

10.3.1.2 Kompatibilitätssichten

Da es seit dem SQL Server 2005 nicht mehr möglich ist direkt auf Systemtabellen zuzugreifen wurde aus Gründen der Abwärtskompatibilität eine Gruppe von Sichten, mit den Bezeichnungen der ehemaligen Systemtabellen, implementiert. Diese Sichten werden als Kompatibilitätssichten bezeichnet und sollen ausschließlich für die Abwärtskompatibilität verwendet werden.

Sie machen die gleichen Metadaten verfügbar wie in SQL Server 2000. Sie machen jedoch keine Metadaten bezüglich der seit SQL Server 2005 neu eingeführten Features verfügbar. Wenn Sie also neue Features, wie z. B. Service Broker oder die Partitionierung, verwenden, müssen Sie Katalogsichten verwenden.

In der folgenden Tabelle werden die Systemtabellen der **master**- Datenbank von SQL Server 2000 ihren entsprechenden Systemsichten bzw. -funktionen in SQL Server 2012 zugeordnet.

| Systemtabelle SQL Server 2000 | Systemsichten bzw. – funktionen SQL Server 2012 | Art der Sicht bzw. Funktion |
|--|--|--|
| sysaltfiles | sys.master_files | Katalogsicht |
| syscacheobjects | sys.dm_exec_cached_plans sys.dm_exec_plan_attributes sys.dm_exec_sql_text sys.dm_exec_cached_plan_dependent_objects | Dynamische Verwaltungssichten |
| syscharsets | sys.syscharsets | Kompatibilitätssicht |
| sysconfigures | sys.configurations | Katalogsicht |
| syscurconfigs | sys.configurations | Katalogsicht |
| sysdatabases | sys.databases | Katalogsicht |
| sysdevices | sys.backup_devices | Katalogsicht |
| syslanguages | sys.syslanguages | Kompatibilitätssicht |
| syslockinfo | sys.dm_tran_locks | Dynamische Verwaltungssicht |
| syslocks | sys.dm_tran_locks | Dynamische Verwaltungssicht |
| syslogins | sys.server_principals sys.sql_logins (Transact-SQL) | Katalogsicht |
| sysmessages | sys.messages | Katalogsicht |
| systoledbusers | sys.linked_logins | Katalogsicht |
| sysopentapes | sys.dm_io_backup_tapes | Dynamische Verwaltungssicht |
| sysperfinfo | sys.dm_os_performance_counters | Dynamische Verwaltungssicht |
| sysprocesses | sys.dm_exec_connections sys.dm_exec_sessions sys.dm_exec_requests | Dynamische Verwaltungssichten |
| sysremotelogins | sys.remote_logins | Katalogsicht |
| sysservers | sys.servers | Katalogsicht |

In der folgenden Tabelle werden die in jeder Datenbank von SQL Server 2000 enthaltenen Systemtabellen bzw. -funktionen ihren entsprechenden Systemsichten bzw. -funktionen in SQL Server 2012 zugeordnet.

| Systemtabelle bzw. – funktion SQL Server 2000 | Systemsicht bzw. –funktion SQL Server 2012 | Art der Sicht bzw. Funktion |
|--|---|---|
| fn_virtualfilestats | sys.dm_io_virtual_file_stats | Dynamische Verwaltungssicht |
| syscolumns | sys.columns | Katalogsicht |
| syscomments | sys.sql_modules | Katalogsicht |
| sysconstraints | sys.check_constraints sys.default_constraints sys.key_constraints sys.foreign_keys | Katalogsichten |
| sysdepends | sys.sql_dependencies | Katalogsicht |
| sysfilegroups | sys.filegroups | Katalogsicht |
| sysfiles | sys.database_files | Katalogsicht |
| sysforeignkeys | sys.foreign_keys | Katalogsicht |
| sysindexes | sys.indexes sys.partitions sys.allocation_units sys.dm_db_partition_stats | Katalogsicht Katalogsicht Katalogsicht Dynamische Verwaltungssicht |

| Systemtabelle bzw. – funktion SQL Server 2000 | Systemsicht bzw. –funktion SQL Server 2012 | Art der Sicht bzw. Funktion |
|--|---|--|
| sysindexkeys | sys.index_columns | Katalogsicht |
| sysmembers | sys.database_role_members | Katalogsicht |
| sysobjects | sys.objects | Katalogsicht |
| syspermissions | sys.database_permissions sys.server_permissions | Katalogsichten |
| sysprotects | sys.database_permissions sys.server_permissions | Katalogsichten |
| sysreferences | sys.foreign_keys | Katalogsicht |
| systypes | sys.types | Katalogsicht |
| sysusers | sys.database_principals | Katalogsicht |
| sysfulltextcatalogs | sys.fulltext_catalogs | Katalogsicht |

10.3.1.3 Dynamische Verwaltungssichten und Verwaltungsfunktionen

Dynamische Verwaltungssichten und -funktionen geben Serverstatusinformationen zurück, mit denen der Zustand einer Serverinstanz überwacht, Probleme diagnostiziert und die Leistung optimiert werden kann.

Es gibt zwei Arten von dynamischen Verwaltungssichten und -funktionen:

- Dynamische Verwaltungssichten und -funktionen mit Serverbereich. Sie erfordern die VIEW SERVER STATE-Berechtigung auf dem Server.
- Dynamische Verwaltungssichten und -funktionen mit Datenbankbereich. Sie erfordern die VIEW DATABASE STATE-Berechtigung für die Datenbank.

Alle dynamischen Verwaltungssichten und -funktionen sind im **sys**- Schema vorhanden und verwenden die Benennungskonvention **dm_***.

Kategorien:

| Präfix | Grundlegende Bedeutung |
|-----------------|---|
| sys.dm_db_* | Allgemeine Datenbankstatistiken, wie Speicherplatzverbrauch und Indexnutzung. |
| sys.dm_exec_* | Stellt Abfragestatistiken zur Verfügung. |
| sys.dm_io_* | Stellt Eingabe/Ausgabe Statistiken zur Verfügung. |
| sys.dm_os_* | Informationen über die Hardware- Ebene. |
| sys.dm_tran_* | Informationen in Verbindung mit Transaktionen. |
| sys.dm_fts_* | Informationen in Verbindung mit der Volltextsuche. |
| sys.dm_clr_* | Mit Common Language Runtime verbundene Verwaltungssicht. |
| sys.dm_qn_* | Informationen zur Abfragebenachrichtigung. |
| sys.dm_repl_* | Informationen zur Replikation. |
| sys.dm_broker_* | Informationen zum Service Broker |

Beispiel:

Nachfolgende Abfrage liefert die zehn SQL- Anweisungen seit dem 01.06.2007 die die meiste CPU- Zeit gebraucht haben.

```
select top 10 total_worker_time/execution_count as [avg cpu time],
convert(char(20),creation_time,113) as [startzeit],
substring(b.text, (a.statement_start_offset/2)+1,
((case a.statement_end_offset
when -1 then datalength(b.text)
else a.statement_end_offset
end - a.statement_start_offset)/2) + 1) as statement_text
from sys.dm_exec_query_stats as a
cross apply sys.dm_exec_sql_text(a.sql_handle) as b
where creation_time between '01.06.2007' and getdate()
order by total_worker_time/execution_count desc;
```

Nachfolgende Abfrage liefert Informationen zu den Wartevorgängen in Threads, die zurzeit ausgeführt werden. Dadurch können Leistungsprobleme bei bestimmten Abfragen und Batches diagnostiziert werden.

```
select wait_type, wait_time_ms
from sys.dm_os_wait_stats
```

10.3.1.4 Informationsschemasichten

Eine Informationsschemasicht ist eine der Methoden, die SQL Server 2012 zum Abrufen von Metadaten bereitstellt

Informationsschemasichten stellen eine interne, von den Systemtabellen unabhängige Darstellung der SQL Server- Metadaten bereit. Informationsschemasichten ermöglichen die einwandfreie Ausführung von Anwendungen, auch wenn an den zugrunde liegenden Systemtabellen erhebliche Änderungen vorgenommen wurden. Die in SQL Server 2012 enthaltenen Informationsschemasichten entsprechen der Definition des SQL-92-Standards für INFORMATION_SCHEMA.

SQL Server unterstützt eine dreiteilige Benennungskonvention beim Verweis auf den aktuellen Server. Der SQL-92-Standard unterstützt ebenfalls eine dreiteilige Benennungskonvention. Die Namen, die in den beiden Konventionen verwendet werden, sind jedoch unterschiedlich. Die Informationsschemasichten sind in einem speziellen Schema namens **INFORMATION_SCHEMA** definiert. Dieses Schema ist in jeder Datenbank enthalten. Jede Informationsschemasicht enthält die Metadaten für alle in der jeweiligen Datenbank gespeicherten Datenobjekte.

In der folgenden Tabelle werden die Beziehungen zwischen den SQL Server-Namen und den SQL-Standardnamen aufgeführt.

| SQL Server-Name | Entsprechender SQL-Standardname |
|------------------------------|---------------------------------|
| Datenbank | Katalog |
| Schema | Schema |
| Objekt | Objekt |
| Benutzerdefinierter Datentyp | Domäne |

Diese Namenuordnungskonvention betrifft die folgenden SQL-92-kompatiblen SQL Server-Sichten.

- CHECK_CONSTRAINTS
- COLUMN_DOMAIN_USAGE
- COLUMN_PRIVILEGES
- COLUMNS
- CONSTRAINT_COLUMN_USAGE
- CONSTRAINT_TABLE_USAGE
- DOMAIN_CONSTRAINTS
- DOMAINS
- KEY_COLUMN_USAGE
- PARAMETERS
- REFERENTIAL_CONSTRAINTS
- ROUTINES
- ROUTINE_COLUMNS
- SCHEMATA
- TABLE_CONSTRAINTS
- TABLE_PRIVILEGES
- TABLES
- VIEW_COLUMN_USAGE
- VIEW_TABLE_USAGE
- VIEWS

Beispiel:

Anzeigen der Spalten aller Tabellen in der Datenbank.

```
select *
from standard.information_schema.columns;
```

Anzeigen aller Spalten einer bestimmten Tabelle.

```
select *
from standard.information_schema.columns
where table_name = 'lieferant';
```

10.3.1.5 Gespeicherte Systemprozeduren:

In SQL Server 2012 können viele Verwaltungs- und Informationsabläufe mithilfe gespeicherter Systemprozeduren ausgeführt werden.

- sind vorgefertigte Abfragen
- beginnen mit Präfix sp_...

Die gespeicherten Systemprozeduren sind in die in der folgenden Tabelle gezeigten Kategorien unterteilt.

| Kategorie | Beschreibung |
|--|---|
| Gespeicherte Active Directory-Prozeduren | Werden zum Registrieren der Instanzen von SQL Server und Datenbanken in Microsoft Windows 2000 Active Directory verwendet. |
| Gespeicherte Katalogprozeduren | Implementieren Funktionen ODBC-Datenwörterbüchern und isolieren ODBC-Anwendungen von Änderungen an den zugrunde liegenden Systemtabellen. |
| Gespeicherte Cursorprozeduren | Werden zum Implementieren von Cursorvariabelfunktionen verwendet. |
| Gespeicherte Datenbankmodulprozeduren | Werden für die allgemeine Wartung von SQL Server Database Engine (Datenbankmodul) verwendet. |

| Kategorie | Beschreibung |
|--|---|
| Gespeicherte Datenbank-E-Mail- und SQL Mail-Prozeduren | Werden zum Ausführen von E-Mail-Operationen innerhalb einer Instanz von SQL Server verwendet. |
| Gespeicherte Prozeduren für Datenbank-Wartungspläne | Werden zum Einrichten zentraler Wartungsaufgaben verwendet, die zur Optimierung der Datenbankleistung ausgeführt werden müssen. |
| Gespeicherte Prozeduren für verteilte Abfragen | Werden zum Implementieren und Verwalten verteilter Abfragen verwendet. |
| Gespeicherte Prozeduren für die Volltextsuche | Werden zum Implementieren und Abfragen von Volltextindizes verwendet. |
| Gespeicherte Prozeduren für den Protokollversand | Werden zum Konfigurieren, Ändern und Überwachen von Protokollversandkonfigurationen verwendet. |
| Gespeicherte Automatisierungsprozeduren | Aktivieren standardmäßige Automatisierungsobjekte für die Verwendung innerhalb eines Transact-SQL-Standardbatches. |
| Gespeicherte Prozeduren von Notification Services | Werden zum Verwalten von SQL Server 2012 Notification Services verwendet. |
| Gespeicherte Prozeduren für die Replikation | Werden für die Replikation verwendet. |
| Gespeicherte Sicherheitsprozeduren | Werden für die Verwaltung der Sicherheit verwendet. |
| Gespeicherte Prozeduren für SQL Server Profiler | Werden von SQL Server Profiler verwendet, um die Leistung und die Aktivitäten zu überwachen. |
| Gespeicherte SQL Server-Agent-Prozeduren | Werden vom SQL Server-Agent zum Verwalten geplanter und ereignisgesteuerter Aktivitäten verwendet. |
| Gespeicherte Prozeduren für Webtasks | Werden zum Erstellen von Webseiten verwendet. |
| Gespeicherte XML-Prozeduren | Werden für die XML-Textverwaltung verwendet. |
| Gespeicherte allgemeine erweiterte Prozeduren | Stellen eine Schnittstelle zwischen einer Instanz von SQL Server und externen Programmen für verschiedene Wartungsaktivitäten bereit. |

10.3.1.6 Replikations- Sichten

Enthält Informationen, die von der Replikation in SQL Server 2012 verwendet werden. Diese Sichten ermöglichen einen leichteren Zugriff auf Daten in Replikationstabellen. Sichten werden in einer Benutzerdatenbank erstellt, wenn diese Datenbank als Publikations- oder Abonnementdatenbank aktiviert wird. Wird die Datenbank aus der Replikationstopologie entfernt, dann werden auch alle Replikationsobjekte aus Benutzerdatenbanken entfernt.

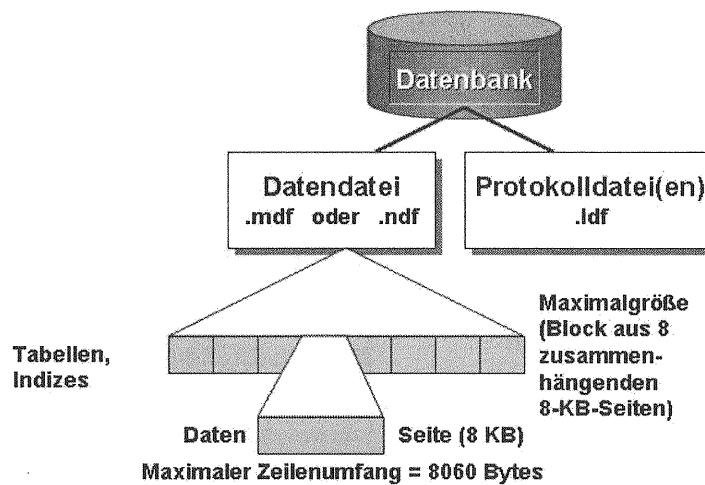
10.3.1.7 Notification Services- Sichten

Sichten bieten eine zusammenfassende Ansicht von Instanz- und Anwendungsdaten. SQL Server 2012 Notification Services stellt die in diesem Abschnitt beschriebenen Sichten für das Debuggen, das Nachverfolgen und die Problembehandlung von Notification Services-Anwendungen bereit.

10.4 Benutzerdatenbanken

10.4.1 Art der Datenspeicherung

Datenbanken setzen sich aus einer primären Datendatei (.mdf) und mindestens einer Transaktionsprotokolldatei (.ldf) zusammen. Weiterhin kann eine Datenbank über sekundäre Datendateien (.ndf) verfügen.



Alle Datendateien und Transaktionsprotokolldateien sind physische Dateien. Sie besitzen sowohl einen Betriebssystem-Dateinamen als auch logische Dateinamen, die in SQL-Anweisungen verwendet werden können.

Bei der Erstellung einer neuen Datenbank wird immer eine Kopie der Model- Datenbank (Mindestgröße 4 MB) erzeugt und diese in den entsprechenden Datenbanknamen umbenannt. Das heißt die zu erzeugende Datenbank ist mindestens so groß wie die Model- Datenbank.

Sie enthält die Systemtabellen der Datenbank.

Daten werden in zusammenhängenden Datenblöcken von acht 8 KB- Seiten auf der Festplatte gespeichert.

Datensätze können nicht Seitenübergreifend gespeichert werden.

Abzüglich Seitenoverhead stehen pro Seite und Datensatz 8060 Byte für Daten zur Verfügung. SQL Server 2012 unterstützt die Zeilenüberlaufspeicherung, sodass Spalten Variabler Länge verschoben werden können. Dadurch ist die tatsächliche Größe höher.

Tabellen und Indizes werden in Blöcken (8 zusammenhängende Seiten, also 64 KB) gespeichert. Kleine Tabellen können zusammen mit anderen Datenbankobjekten Blöcke gemeinsam nutzen.

In Transaktionsprotokollen werden alle Informationen gespeichert die zur Veränderung des Datenbestandes (Insert, Update und Delete) geführt haben, da sie für die Wiederherstellung einer Datenbank im Falle eines Systemfehlers erforderlich sind. Die Größe des Transaktionsprotokolls sollte ca. 25% der Größe der Datendateien betragen.

Beispiel:

Ermitteln des Speicherplatzbedarfs für die Datenbank "Standard".

Im Verlauf eines Geschäftsjahres der Firma, wird mit ca. 450000 Lieferungen von geschätzten 400 Lieferanten gerechnet. Der Artikelbestand wird voraussichtlich 100000 verschiedene Artikel nicht überschreiten.

1. Ermitteln der Datensatzlänge für jede Tabelle.

Lieferant:

| | | | |
|---------|----------|----|---------|
| Inr | char(5) | 5 | Byte |
| Iname | char(20) | 20 | Byte |
| status | integer | 4 | Byte |
| Istadt | char(25) | 25 | Byte |
| Gesamt: | | | 54 Byte |

Lieferung:

| | | | |
|---------|----------|----|---------|
| Inr | char(5) | 5 | Byte |
| anr | char(5) | 5 | Byte |
| lmenge | integer | 4 | Byte |
| Idatum | datetime | 30 | Byte |
| Gesamt: | | | 44 Byte |

Artikel:

| | | | |
|---------|----------|----|---------|
| anr | char(5) | 5 | Byte |
| aname | char(20) | 20 | Byte |
| farbe | char(7) | 7 | Byte |
| gewicht | float | 8 | Byte |
| astadt | char(25) | 25 | Byte |
| amenge | integer | 4 | Byte |
| Gesamt: | | | 69 Byte |

2. Für jede Tabelle ermitteln wie viele Datensätze in eine Speicherseite abgelegt werden können. Wir gehen davon aus, dass von den 8192 Byte die eine Page groß ist, 8060 Byte für die Daten und der Rest für den Seitenoverhead (132 Byte) verwendet werden.

Für die Tabelle "Lieferung" gilt:

8060 Byte / 44 Byte = 183,18 Das ergibt 183 Datensätze pro Speicherseite.

Für die Tabelle "Lieferant" gilt:

8060 Byte / 54 Byte = 149,25 Das ergibt 149 Datensätze pro Speicherseite.

Für die Tabelle "Artikel" gilt:

8060 Byte / 69 Byte = 116,81 Das ergibt 116 Datensätze pro Speicherseite.

3. Jetzt ermitteln wir die Anzahl der Speicherseiten für die zu erwartende Anzahl von Datensätzen in jeder Tabelle. Dabei wird jetzt das Ergebnis bei Notwendigkeit nach oben gerundet.

Für die Tabelle "Lieferung" gilt:

450000 Datensätze / 183 Datensätze pro Seite = 2460 Speicherseiten a. 8192 Byte, das bedeutet für die Tabelle Lieferung müssen ca. 19,22 MB bereitgestellt werden.

Für die Tabelle "Lieferant" gilt:

400 Datensätze / 149 Datensätze pro Seite = 3 Speicherseiten a. 8192 Byte, das bedeutet für die Tabelle Lieferant müssen ca. 24 KB bereitgestellt werden.

Für die Tabelle "Artikel" gilt:

100000 Datensätze / 116 Datensätze pro Seite = 863 Speicherseiten a. 8192 Byte, das bedeutet für die Tabelle Artikel müssen ca. 6,75 MB bereitgestellt werden.

Für diese Datenbank muss ein Speicherbedarf allein für den Datenbestand von rund 26 MB erwartet werden.

Für das Transaktionsprotokoll reservieren wir auf einen anderen Medium Speicherplatz. Die Größe sollte dabei je nach Aktivität der Datenmanipulation 5% bis maximal 25% der Datenbankgröße betragen.

10.4.2 Arbeitsweise des Transaktionsprotokolls

10.4.2.1 Was ist eine Transaktion?

Eine Transaktion besteht aus einer oder mehreren SQL-Anweisungen, die zusammen einen einzigen Arbeits- und Wiederherstellungsvorgang darstellen. Sie können SQL-Anweisungen oder Funktionen der Datenbank-API verwenden, um den Beginn und das Ende von Transaktionen anzugeben.

Das Akronym **ACID** wird häufig für die Beschreibung der Eigenschaften von Transaktionen genutzt und steht für folgende Begriffe:

- Unteilbarkeit (**Atonicity**). Eine Transaktion ist eine unteilbare Einheit, die entweder ganz oder ganz gar nicht ausgeführt wird.
- Konsistenz (**Consistency**). Eine Transaktion belässt Daten grundsätzlich in einem konsistenten Zustand.
- Isolation. Eine Transaktion wird immer isoliert von anderen Datenbankaktivitäten ausgeführt. Das bedeutet die Änderungen der Transaktion beeinflusst keine weitere gleichzeitig ausgeführte Transaktion. Das wird im DBMS durch Sperren realisiert.
- Beständigkeit (**Durability**). Wenn für die Transaktion ein COMMIT ausgeführt wird, werden die Ergebnisse dauerhaft im Datenbestand abgelegt.

Zwei Arten:

- Implizite Transaktionen, bei denen jede Insert-, Update- und Delete-Anweisung als eigenständige Transaktion ausgeführt wird.
- Explizite bzw. benutzerdefinierte Transaktionen, bei denen SQL Anweisungen zu einer Transaktion (begin transaction..., commit transaction..., rollback transaction...) zusammengefasst werden.

Standardmäßig arbeitet SQL Server im Autocommit-Modus. Das bedeutet, dass eine implizite Transaktion, ohne eine Commit Transaction-Anweisung, automatisch ein Commit ausführt um die Transaktion zu beenden.

Ändern: `SET IMPLICIT_TRANSACTIONS {ON | OFF}`

SQL Server führt implizite Transaktionen bei SQL Anweisungen, wie alter table, create, delete, insert, update, drop, fetch, grant, open, revoke, select, truncate table, aus:

10.4.2.2 Was ist ein Transaktionsprotokoll?

Um die Datenbankkonsistenz zu gewährleisten und eine problemlose Wiederherstellung (Recovery) zu ermöglichen, werden sämtliche ändernden Transaktionen in einem Transaktionsprotokoll aufgezeichnet.

Daten und Transaktionsprotokollinformationen werden niemals in derselben Datei vermischt, und die einzelnen Dateien werden immer nur von einer einzigen Datenbank verwendet.

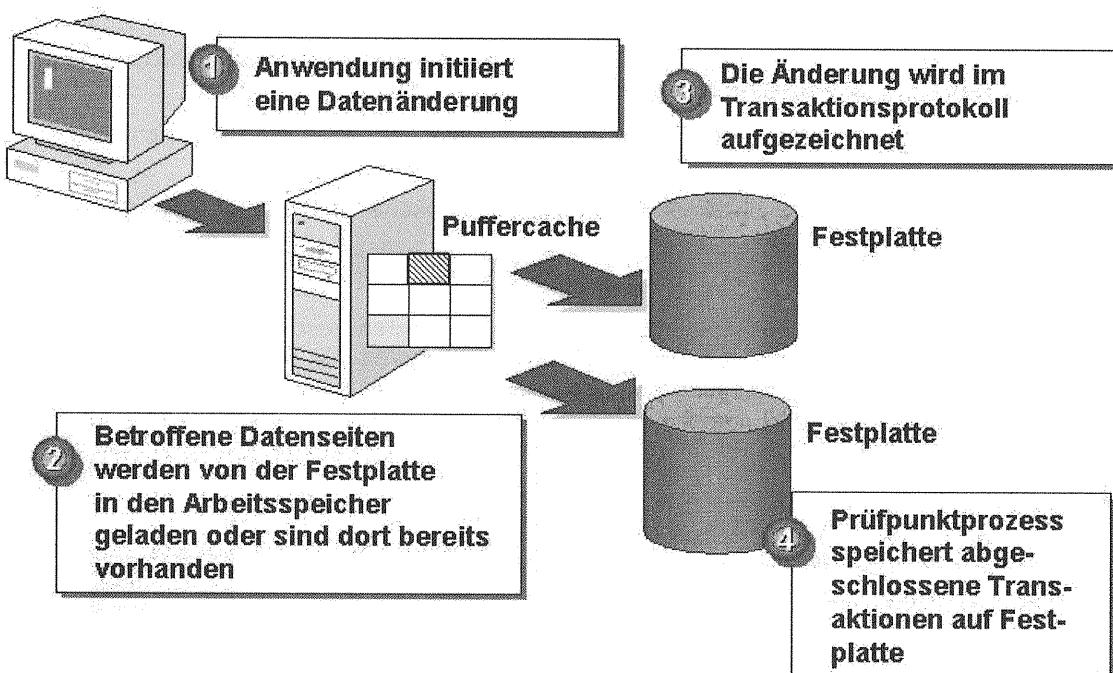
Alle Änderungen am Datenbestand werden sofort (noch bevor sie tatsächlich in die Datenbank geschrieben werden) im Protokoll auf der Festplatte gespeichert.

Mithilfe des Transaktionsprotokolls der jeweiligen Datenbank kann SQL Server Änderungsoperationen einer Transaktion wiederherstellen wenn für die Transaktion ein Rollback ausgeführt wurde. Das Transaktionsprotokoll ist eine fortlaufende Aufzeichnung aller Änderungen, die in der Datenbank vorgenommen wurden. Zudem wird protokolliert, mit welcher Transaktion die jeweilige Änderung durchgeführt wurde. Im Transaktionsprotokoll

wird der Beginn jeder Transaktion aufgezeichnet. Weiterhin werden die Änderungen, die an den Daten vorgenommen werden, sowie ausreichende Informationen zum Rückgängigmachen der Änderungen (falls dies später erforderlich sein sollte) aufgezeichnet, die während jeder Transaktion vorgenommen wurden. Bei einigen umfangreichen Operationen, wie z. B. für CREATE INDEX, wird im Transaktionsprotokoll lediglich aufgezeichnet, dass die Operation stattgefunden hat. Das Protokoll wird kontinuierlich vergrößert, während zu protokollierende Operationen in der Datenbank ausgeführt werden.

Im Transaktionsprotokoll werden die Reservierung oder Freigabe von Seiten und der Commit oder Rollback jeder Transaktion aufgezeichnet. Auf diese Weise kann SQL Server jede Transaktion entweder anwenden (Rollforward) oder zurücksetzen (Rollback).

10.4.2.3 Arbeitsweise des Transaktionsprotokoll



Im Transaktionsprotokoll werden alle Datenänderungen während ihrer Durchführung in folgenden Schritten aufgezeichnet:

1. Die Anwendung initiiert eine Datenänderung.
2. Die betroffenen Seiten werden von der Festplatte in den Arbeitsspeicher (falls sie sich noch nicht dort befinden) geladen.
3. Die Datenänderung wird im Protokoll auf der Festplatte aufgezeichnet. Diese Art des Protokolls wird als Write-Ahead Protokoll (Vorausschreiben-Protokoll) bezeichnet. Danach wird die Änderung an den Daten im Arbeitsspeicher vorgenommen.
4. Der Inhalt des Datenbankpuffers (Puffercache) wird von den immer wiederkehrenden Prüfpunktprozessen auf die Festplatte (Datenbank und Transaktionsprotokoll) ausgeschrieben.

Durch Prüfpunkte wird der Teil des Protokolls minimiert, der im Rahmen einer vollständigen Wiederherstellung einer Datenbank verarbeitet werden muss.

Ein Prüfpunkt erzwingt das Ausschreiben des Puffercache auf die Festplatte.

In SQL Server 2012 werden durch einen Prüfpunkt die folgenden Operationen in der aktuellen Datenbank eingeleitet:

- Schreiben eines Datensatzes in die Protokolldatei, mit dem der Beginn des Prüfpunkts markiert wird.
- Speichern der aufgezeichneten Informationen für den Prüfpunkt in einer Kette von Prüfpunkt-Protokolleinträgen. Die LSN des Anfangs dieser Kette wird auf die Datenbank-Startseite geschrieben.
- Ein Teil der in den Prüfpunktdatensätzen aufgezeichneten Informationen besteht aus der LSN des ersten Protokollabilds, das für eine erfolgreiche Durchführung eines datenbankweiten Rollbacks vorhanden sein muss. Diese LSN wird als Mindestwiederherstellungs- LSN (MinLSN) bezeichnet und gibt den Mindestwert für folgendes an:
 - Die LSN des Beginns des Prüfpunktes.
 - Die LSN des Beginns der ältesten aktiven Transaktion.
 - Die LSN des Beginns der ältesten Replikationstransaktion, die noch nicht auf alle Abonnenten repliziert worden ist.
- Ein weiterer Informationsteil, der in den Prüfpunktdatensätzen aufgezeichnet wird, ist eine Liste aller ausstehenden aktiven Transaktionen.
- Löschen aller Protokolleinträge vor der neuen MinLSN, wenn die Datenbank das einfache Wiederherstellungsmodell verwendet.
- Schreiben aller modifizierten Protokoll- und Datenseiten auf den Datenträger.
- Schreiben eines Datensatzes in die Protokolldatei, mit dem das Ende des Prüfpunkts markiert wird.

Der Teil der Protokolldatei von der MinLSN bis zu dem als letzten geschriebenen Protokolleintrag wird aktiver Teil des Protokolls genannt. Dies ist der Teil des Protokolls, der für eine vollständige Wiederherstellung der Datenbank erforderlich ist. Vom aktiven Teil des Protokolls kann niemals ein Teil abgeschnitten werden. Alle Protokollkürzungen müssen Teile des Protokolls betreffen, die vor der MinLSN liegen.

Beispiel:

| LSN 141 | LSN 142 | LSN 143 | LSN 144 | LSN 145 | LSN 146 | LSN 147 | LSN 148 |
|-------------|-------------|--------------|-------------|--------------|--------------|-------------|--------------|
| Begin Tran1 | Begin Tran2 | Update Tran2 | Check point | Update Tran1 | Commit Tran1 | Check point | Update Tran2 |

LSN 148 ist der letzte Eintrag im Transaktionsprotokoll. Zum Zeitpunkt der Verarbeitung des Prüfpunktes, der bei LSN 147 aufgezeichnet wurde, wurde für Tran 1 ein Commit ausgeführt, und Tran 2 war die einzige aktive Transaktion. Hierdurch wird der erste Protokolleintrag für Tran 2 zum ältesten Protokolleintrag für eine Transaktion, die zum Zeitpunkt des letzten Prüfpunktes aktiv war. LSN 142, der Eintrag für den Transaktionsbeginn von Tran 2, wird somit zur MinLSN.

Prüfpunkte treten unter folgenden Bedingungen auf:

- Wenn eine CHECKPOINT- Anweisung ausgeführt wird. Der Prüfpunkt wird für die aktuelle Datenbank der Verbindung gesetzt.
- Wenn ALTER DATABASE zum Ändern einer Datenbankoption verwendet wird. ALTER DATABASE setzt Prüfpunkte für die Datenbank, wenn Datenbankoptionen geändert werden.
- Wenn eine Instanz von SQL Server wie folgt unterbrochen wurde:
Durch beide Methoden wird ein Prüfpunkt für jede Datenbank in der Instanz von SQL Server gesetzt.
 - Ausführen einer SHUTDOWN- Anweisung.
 - Verwenden des SQL Server-Dienstkontroll-Managers zum Beenden des Dienstes, der eine Instanz des Datenbankmoduls ausführt.
- Wenn eine Instanz von SQL Server regelmäßig automatische Prüfpunkte in jeder Datenbank erzeugt, um die Zeit zu verkürzen, die die Instanz zum Wiederherstellen der Datenbank benötigen würde.

10.4.2.4 Automatische Prüfpunkte

SQL Server 2012 erzeugt immer automatische Prüfpunkte. Das Intervall zwischen automatischen Prüfpunkten wird anhand der Anzahl der Einträge im Protokoll und nicht mithilfe einer Zeitangabe festgelegt. Der Zeitraum zwischen automatischen Prüfpunkten ist sehr variabel. Werden nur wenige Änderungen in der Datenbank vorgenommen, vergeht mehr Zeit zwischen den automatischen Prüfpunkten. Wenn eine Vielzahl von Daten geändert wird, treten automatische Prüfpunkte häufig auf.

Das Intervall zwischen automatischen Prüfpunkten wird mithilfe der Serverkonfigurationsoption "**recovery interval**" berechnet. Die Standardeinstellung ist **recovery interval = 0**.

Durch diese Option wird festgelegt, wie viel Zeit SQL Server höchstens benötigen darf, um jede Datenbank während des Systemstarts wiederherzustellen. SQL Server schätzt, wie viele Protokolleinträge während einer Datenbankwiederherstellung in dem durch "recovery interval" festgelegten Intervall verarbeitet werden können. Das Intervall zwischen automatischen Prüfpunkten hängt zudem davon ab, ob die Datenbank das einfache Wiederherstellungsmodell verwendet.

- Wenn die Datenbank entweder das Modell der vollständigen oder der massenprotokollierten Wiederherstellung verwendet, wird ein automatischer Prüfpunkt erzeugt, sobald die Anzahl der Protokolleinträge die von SQL Server geschätzte Anzahl an Einträgen erreicht, die im durch die Option "**recovery interval**" festgelegten Zeitraum verarbeitet werden können.

Wenn die Datenbank das einfache Wiederherstellungsmodell verwendet, wird ein automatischer Prüfpunkt erzeugt, sobald die Anzahl der Protokolleinträge dem jeweils kleineren der beiden folgenden Werte entspricht:

- Das Protokoll ist zu 70 Prozent gefüllt.
- Die Anzahl der tatsächlichen Protokolleinträge erreicht die von SQL Server geschätzte Anzahl an Einträgen, die im durch die Option "recovery interval" festgelegten Zeitraum verarbeitet werden können.

Automatische Prüfpunkte schneiden den ungenutzten Teil des Transaktionsprotokolls ab, wenn die Datenbank das einfache Wiederherstellungsmodell verwendet. Das Protokoll wird nicht durch automatische Prüfpunkte abgeschnitten, wenn die Datenbank das Modell der vollständigen oder massenprotokollierten Wiederherstellung verwendet.

Die CHECKPOINT-Berechtigungen sind standardmäßig Mitgliedern der festen Serverrolle **sysadmin** und der festen Datenbankrolle **db_owner** und **db_backupoperator** zugewiesen und nicht übertragbar.

Automatische Wiederherstellung (Recovery)

Diese Art der Wiederherstellung bezieht sich auf die Wiederherstellung nach einem Systemabsturz (Bsp. Stromausfall).

In der Standardeinstellung von "**recovery interval**", hat SQL Server 1 Minute für jede Datenbank Zeit.

Während einer automatischen Wiederherstellung müssen zwei Arten von Aktionen durchgeführt werden:

1. Das Protokoll enthält möglicherweise Einträge zu Änderungen, die vor dem Systemausfall nicht auf den Datenträger geleert wurden. Für diese Änderungen muss ein Rollforward ausgeführt werden.
2. Für alle Änderungen, die mit unvollständigen Transaktionen verbunden sind (Transaktionen, für die kein COMMIT- oder ROLLBACK- Protokolleintrag vorliegt), muss ein Rollback ausgeführt werden.

Bei einem Systemfehler sorgt das Wiederherstellungsverfahren mit Hilfe des Transaktionsprotokolls dafür, dass beim Wiederaufstarten des DB Servers, für alle ausführten Transaktionen ein Rollforward und für alle unvollständigen Transaktionen ein Rollback durchgeführt wird.

Mit Hilfe von Transaktionsmarkierungen im Protokoll werden während der automatischen Wiederherstellung Anfang und Ende einer Transaktion ermittelt.

Eine Transaktion gilt als abgeschlossen (als endgültig ausgeführt), wenn für die BEGIN TRANSACTION- Markierung eine entsprechende COMMIT TRANSACTION- Markierung vorhanden ist.

Der Prozess der automatischen Wiederherstellung kann wie folgt beschrieben werden.

1. Die LSN des letzten Prüfpunktes wird zusammen mit der Mindestwiederherstellungs- LSN (MinLSN) von der Datenbankstartseite gelesen.
2. Das Transaktionsprotokoll wird beginnend bei der MinLSN bis zum Ende des Protokolls gescannt. Für alle modifizierten Seiten, für die ein Commit ausgeführt wurde, wird ein Rollforward ausgeführt, durch den die logische Operation wiederholt wird, die in dem Protokolleintrag aufgezeichnet wurde.
3. Danach wird die Protokolldatei rückwärts gelesen und für alle unvollständigen Transaktionen wird ein Rollback (zurückrollen) ausgeführt, indem die logische Umkehrung der im Protokolleintrag aufgezeichneten Operationen angewendet wird.

Seit SQL Server 2005 steht eine Datenbank in der Rollback- Phase schon wieder für die Arbeit bereit.

Hinweis: Wenn der Festplattencontroller, für die Festplatte mit der Transaktionsprotokolldatei, über einen Schreibcache verfügt ist das sofortige Ausschreiben des Transaktionsprotokolls nicht möglich. Dadurch kann es vorkommen das Protokolleinträge sich bei einem Systemausfall (Stromausfall) noch im Cache des Controllers befinden. Beim Wiederaufstarten des DBS ist die betroffene Datenbank in einem nichtkonsistenten Zustand und damit defekt. Darum sollte der Controller einen batteriepufferten Cache besitzen oder sie müssen den Cache für diesen Controller ausschalten.

10.4.3 Erstellen von Datenbanken

Zum Erstellen von Datenbanken sollten Sie sich in der master- Datenbank befinden (in älteren Versionen mussten Sie sich in der master Datenbank befinden).

Drei Arten von Dateien werden zum Speichern einer Datenbank verwendet:

- Die primäre Datei enthält die Startinformationen und Systemtabellen für die Datenbank. Die primäre Datei wird auch zum Speichern von Daten verwendet. Jede Datenbank hat eine primäre Datei.
- Sekundäre Dateien enthalten alle Daten, die nicht in die primäre Datendatei passen. Datenbanken, deren primäre Datei groß genug ist, um alle Daten der Datenbank zu speichern, brauchen keine sekundären Datendateien. Datenbanken können aber auch so groß sein, dass sie mehrere sekundäre Datendateien benötigen, oder sekundäre Dateien auf separaten Laufwerken verwenden, um die Daten über mehrere Datenträger zu verteilen.
- Transaktionsprotokolldateien enthalten die Protokollinformationen, die zum Wiederherstellen der Datenbank benötigt werden. Für jede Datenbank muss mindestens eine Transaktionsprotokolldatei vorhanden sein, es kann jedoch auch mehrere geben. Die Minimalgröße für eine Transaktionsprotokolldatei beträgt 512 KB.

Jede Datenbank besteht aus mindestens zwei Dateien, einer primären Datei und einer Transaktionsprotokolldatei. Maximal kann eine Datenbank 32767 Dateien besitzen. Dabei ist die maximale Größe der Daten auf 16 TB und der Protokolldatei auf 2 TB festgelegt. Eine Datenbank kann auf maximal 32767 Dateigruppen aufgeteilt werden.

| Dateityp | Empfohlene Dateinamenerweiterung |
|----------------------------|----------------------------------|
| Primäre Datendatei | .mdf |
| Sekundärdatendatei | .ndf |
| Transaktionsprotokolldatei | .ldf |

10.4.3.1 SQL Anweisung CREATE DATABASE

Sie können eine CREATE DATABASE- Anweisung verwenden, um eine Datenbank und die Dateien zu erstellen, welche die Datenbank enthalten. SQL Server führt die CREATE DATABASE- Anweisung in zwei Schritten aus:

1. SQL Server verwendet eine Kopie der model- Datenbank, um die Datenbank und ihre Metadaten zu initialisieren.
2. Der Datenbank wird eine Service Broker- GUID zugewiesen.
3. Dann füllt SQL Server den Rest der Datenbank mit leeren Seiten, mit Ausnahme der Seiten mit internen Daten, die protokolliert, wie der Speicherplatz in der Datenbank verwendet wird.

Benutzerdefinierte Objekte in der model- Datenbank werden daher in alle neu erstellten Datenbanken kopiert. Sie können zur model- Datenbank alle Objekte (z. B. Tabellen, Sichten, gespeicherte Prozeduren, Datentypen usw.) hinzufügen, die in allen Ihren Datenbanken enthalten sein sollen.

Wer die Datenbank erstellt hat ist der Besitzer der alle Aktivitäten in der Datenbank ausführen kann. Den Besitzer ändern Sie mit der gespeicherten Systemprozedur **sp_changedbowner**.

Datenbanken können von Mitgliedern der Serverrolle **sysadmin** und von autorisierten Benutzern erstellt werden. Die erfordert die Berechtigung **create database**, **create any database** oder **alter any database**.

Syntax:

```

CREATE DATABASE database_name
[CONTAINMENT = {NONE | PARTIAL}]
[
ON [PRIMARY]
  (NAME = logischer_Dateiname,
  FILENAME = {'BS_Dateiname' | 'filestream_pfad'}
  [, SIZE = Größe {KB | MB | GB | TB}]
  [, MAXSIZE = {max_größe {KB | MB | GB | TB}} | UNLIMITED}]
  [, FILEGROWTH = vergrößerung {KB | MB | GB | TB}]] )
  [...n]
]
[
[,][FILEGROUP gruppen_name [CONTAINS FILESTREAM] [default]]
  (NAME = logischer_Dateiname,
  FILENAME = {'BS_Dateiname' | 'filestream_pfad'}
  [, SIZE = Größe {KB | MB | GB | TB}]
  [, MAXSIZE = {max_größe {KB | MB | GB | TB}} | UNLIMITED}]
  [, FILEGROWTH = vergrößerung {KB | MB | GB | TB}]] )
  [...n]
]
[, ...n]
[
LOG ON
  (NAME = logischer_Dateiname,
  FILENAME = 'BS_Dateiname'
  [, SIZE = Größe {KB | MB | GB | TB}]
  [, MAXSIZE = {max_größe {KB | MB | GB | TB}} | UNLIMITED}]
  [, FILEGROWTH = vergrößerung {KB | MB | GB | TB}]] )
  [...n]
]
[COLLATE collation_name]
[FOR ATTACH | ATTACH_REBUILD_LOG]
[WITH [DB_CHAINING {ON | OFF}] [,TRUSTWORTHY {ON | OFF}]];

```

| | |
|---------------|--|
| database_name | Der Name der neuen Datenbank kann bis zu 128 Zeichen umfassen und muss innerhalb der Datenbank eindeutig sein. Wenn kein logischer Dateiname für die Daten oder Protokoll angegeben ist, generiert SQL Server einen logischen Namen durch Anhängen eines Suffixes an database_name. Dadurch wird database_name auf 123 Zeichen beschränkt, so dass der generierte logische Protokolldateiname weniger als 128 Zeichen umfasst. |
| ON | Gibt an, dass die zum Speichern der Datenteile der Datenbank (Datendateien) verwendeten Datenträgerdateien explizit definiert werden. Dem Schlüsselwort folgt eine Liste durch Kommas voneinander getrennter Elemente, die die Datendateien für die primäre Dateigruppe definieren. |
| PRIMARY | Gibt an, dass die zugehörige Liste die Primärdatei definiert. Die primäre Dateigruppe enthält alle Datenbanksystemtabellen. Außerdem enthält sie alle Objekte, die nicht einer Benutzerdateigruppen zugewiesen sind. Der erste Eintrag in der primären Dateigruppe wird zur primären Datei, die den logischen Anfang der Datenbank und ihre Systemtabellen enthält. Eine Datenbank kann nur eine primäre Datei haben. Ist PRIMARY nicht angegeben, wird die erste in der CREATE DATABASE- Anweisung aufgeführte Datei die primäre Datei. |

| | |
|---------------------|--|
| LOG ON | Gibt an, dass die zum Speichern des Datenbankprotokolls verwendeten Datenträgerdateien (Protokolldateien) explizit definiert werden. Dem Schlüsselwort folgt eine Liste durch Kommas voneinander getrennter Elemente, die die Protokolldateien definieren. Wenn LOG ON nicht angegeben ist, wird automatisch eine einzelne Protokolldatei mit einem vom System erzeugten Namen angelegt, deren Größe 25 % der Gesamtgröße aller Datendateien für die Datenbank beträgt. |
| FILESTREAM_PATH | Für eine FILESTREAM- Dateigruppe verweist FILENAME auf einen Pfad, wo FILESTREAM- Daten gespeichert werden. Der Pfad muss bis zum letzten Ordner vorhanden sein, und der letzte Ordner darf nicht vorhanden sein. Wenn Sie den Pfad C:\Files\FilestreamData für die FILESTREAM- Daten vorgesehen haben, muss C:\Files vor der Ausführung von CREATE DATABASE vorhanden sein, der Ordner "FilestreamData" muss jedoch noch nicht existieren. Die Dateigruppe und die Dateidefinition müssen in derselben Anweisung erstellt werden. Für eine FILESTREAM- Dateigruppe kann es nur eine Dateidefinition geben. Die Eigenschaften SIZE, MAXSIZE und FILEGROWTH gelten nicht für eine FILESTREAM- Dateigruppe. |
| CONTAINS FILESTREAM | Gibt an, dass die Dateigruppe FILESTREAM- BLOBs (Binary Large Objects) im Dateisystem speichert. |
| FOR ATTACH | Gibt an, dass eine Datenbank aus einem vorhandenen Satz von Betriebssystemdateien angefügt wird. Es muss ein Eintrag vorhanden sein, der die erste primäre Datei angibt. Darüber hinaus werden nur Einträge für Dateien benötigt, deren Pfad sich seit dem erstmaligen Erstellen oder letztem Anfügen der Datenbank geändert hat. Für diese Dateien muss ein Eintrag angegeben werden |
| NAME | Gibt den logischen Namen für die definierte Datenbankdatei an. Der NAME- Parameter ist nicht erforderlich, wenn FOR ATTACH angegeben ist. |
| FILENAME | Gibt den Betriebssystemdateinamen für die definierte Datenbankdatei an. Wenn die Datei auf einer RawPartition erstellt wird, darf BS- Dateiname nur den Laufwerkbuchstaben einer vorhandenen RawPartition angeben. Auf jeder RawPartition kann nur eine Datei erstellt werden. Dateien auf RawPartitionen vergrößern sich automatisch; daher werden die Parameter MAXSIZE und FILEGROWTH nicht benötigt. |
| SIZE | Gibt die Anfangsgröße der definierten Datei an. Wenn für eine primäre Datei kein SIZE- Parameter angegeben ist, verwendet SQL Server die Größe der primären Datei in der model- Datenbank. Wenn für eine sekundäre oder Protokolldatei kein SIZE- Parameter angegeben ist, verwendet SQL Server ebenfalls die Standardgröße der model- Datenbank. Die Suffixe KB , MB , GB und TB können verwendet werden. Die Standardeinheit ist MB. Geben Sie eine ganze Zahl an. Die für die primäre Datei angegebene Größe muss mindestens der Größe der primären Datei der model- Datenbank entsprechen. |
| MAXSIZE | Gibt die maximale Größe an, auf die die definierte Datenbankdatei vergrößert werden darf. Die Suffixe KB , MB , GB und TB können verwendet werden. Die Standardeinheit ist MB. Geben Sie eine ganze Zahl an. Wenn MAXSIZE nicht angegeben ist, kann die Datei so lange vergrößert werden, bis der Speicherplatz auf dem Datenträger erschöpft ist. |
| UNLIMITED | Gibt an, dass die definierte Datenbankdatei so lange vergrößert werden kann, bis der Speicherplatz auf dem Datenträger erschöpft ist aber maximal 16 TB für die Datendateien und 2 TB für die Protokolldateien . |

| | |
|--------------------|--|
| FILEGROWTH | Gibt die Schrittweite für die automatische Vergrößerung der definierten Datenbankdatei an. Geben Sie eine ganze Zahl an. Der Wert kann in KB, MB, GB, TB oder % angegeben werden. Bei Zahlen ohne Angabe von MB, KB oder % wird standardmäßig MB verwendet. Ist FILEGROWTH nicht angegeben, ist der Standardwert 1 MB. |
| ATTACH_REBUILD_LOG | Datenbank wird erstellt, indem ein vorhandener Satz von Betriebssystemdateien angefügt wird. Diese Option ist auf Datenbanken mit Lese-/Schreibzugriff beschränkt. Wenn eines oder mehrere Transaktionsprotokolle fehlen, wird das Protokoll neu erstellt. Die primäre Datei muss angegeben werden.. |

Beispiel:

Erstellt eine Datenbank „Zitrone“ mit einer primären Datendatei zitrone.mdf und einer Transaktionsprotokolldatei zitrone.ldf im Standardverzeichnis des SQL Servers.

```
create database zitrone;
```

Hinweise:

Die master-Datenbank sollte immer dann gesichert werden, wenn eine Benutzerdatenbank erstellt, geändert oder gelöscht wird.

Die CREATE DATABASE- Anweisung muss im Auto-Commit-Modus (Standardeinstellung) ausgeführt werden und ist in einer expliziten oder impliziten Transaktion nicht zulässig.

Sie können eine CREATE DATABASE- Anweisung verwenden, um eine Datenbank und die Dateien zu erstellen, in denen die Datenbank gespeichert ist. SQL Server implementiert die CREATE DATABASE-Anweisung in folgenden Schritten:

- SQL Server verwendet eine Kopie der model-Datenbank, um die Datenbank und ihre Metadaten zu initialisieren.
- Der Datenbank wird eine Service Broker-GUID zugewiesen.
- Dann füllt das Datenbankmodul den Rest der Datenbank mit leeren Seiten auf, mit Ausnahme der Seiten mit internen Daten, in denen aufgezeichnet ist, wie der Speicherplatz in der Datenbank verwendet wird. Weitere Informationen finden Sie unter Datenbankdatei-Initialisierung.

Maximal 32.767 Datenbanken können auf einer Instanz von SQL Server angegeben werden.

Jede Datenbank hat einen Besitzer, der besondere Aktivitäten in der Datenbank ausführen kann. Der Besitzer ist der Benutzer, der die Datenbank erstellt. Der Datenbankbesitzer kann mit der Anweisung **ALTER AUTHORIZATION...** oder aus Gründen der Abwärtskompatibilität mit der Prozedur **sp_changedbowner** geändert werden.

10.4.3.2 Datenbankdateien und Dateigruppen

Jede Datenbank verfügt über mindestens zwei Dateien, und zwar einer primären Datei und einer Transaktionsprotokolldatei, sowie über mindestens eine Dateigruppe. Für jede Datenbank können maximal 32.767 Dateien und 32.767 Dateigruppen angegeben werden.

Wenn Sie eine Datenbank erstellen, sollten die Datendateien möglichst groß sein. Orientieren Sie sich dabei an den maximal zu erwartenden Datenmengen, die in der Datenbank gespeichert werden sollen.

10.4.3.3 FILESTREAM- Daten

Die Menge an unstrukturierten Daten die in einer Datenbank abgelegt werden, ist in den letzten Jahren geradezu explodiert.

Bisher konnten BLOB's in Spalten vom Datentyp NVARCHAR(MAX) (seit Version 2005) abgelegt werden. Der Nachteil, waren diese größer als 2 GB funktionierte das auch nicht. Darum haben sich viele Datenbankadministratoren damit geholfen in den Tabellen nur die Pfade zu den betreffenden Dateien abzuspeichern und die Dateien im Dateisystem zu belassen. Werden diese Dateien jedoch verschoben oder aus Versehen gelöscht, dann sind sie für die Datenbank nicht mehr vorhanden.

Hinweis:

- FILESTREAM kann nicht in einer 32- Bit Version von SQL Server aktiviert werden, die unter einem 64- Bit Betriebssystem ausgeführt wird.
- Für FILESTREAM- Dateigruppen können keine Datenbank- Snapshots erstellt werden.
- Die Datenbankspiegelung wird für FILESTREAM- Datenbanken nicht unterstützt, aber für den Protokollversand (muss auf primären und sekundären Server aktiviert sein).
- Wird für Failoverclustering bereitgestellt, allerdings muss die FILESTREAM- Dateigruppe auf einem freigegebenen Datenträger abgelegt werden und FILESTREAM muss auf jedem Knoten aktiviert werden.

Mithilfe des FILESTREAM- Features können die Dateien nun mit einer Datenbank verknüpft werden. Die Dateien werden zwar immer noch in einen Ordner des Dateisystems gespeichert, sind aber direkt mit einer Datenbank verknüpft, wo sie gesichert, wiederhergestellt, als Volltext indiziert und mit anderen strukturierten Daten kombiniert werden können.

Beispiel:

Im nachfolgenden Beispiel wird eine Datenbank erstellt, welche die FILESTREAM- Eigenschaft verwendet. Bevor die Datenbank erstellt werden kann muss folgendes sichergestellt werden:

- Aktivieren der FILESTREAM- Aktivität für die Instanz von SQL Server. Das geschieht gleich bei der Installation oder nachträglich über den Konfigurationsmanager von SQL Server.
- Festlegen des FILESTREAM_ACCESS_LEVEL (0, 1 oder 2) für die Instanz mit der gespeicherten Systemprozedur **sp_configure**.

| Wert | Beschreibung |
|------|---|
| 0 | Deaktiviert die FILESTREAM-Unterstützung für diese Instanz. |
| 1 | Aktiviert FILESTREAM für den Transact-SQL-Zugriff. |
| 2 | Aktiviert FILESTREAM für Transact-SQL und für den Win32-Streamingzugriff. |

Beispiel:

```
create database standard_filestream
on primary
    (name = standard_fs_daten,
     filename = 'c:\daten_dbstandard\standard_fs_daten.mdf'),
filegroup dokumentefilestream contains filestream
    (name = standard_dokumente,
     filename = 'c:\dokumente_dbs\standard_dokumente'),
log on
    (name = standard_fs_log,
     filename = 'c:\daten_dbstandard\standard_fs_log.ldf');
```

10.4.3.4 Die Datenbank **tempdb**

Die Datenbank **tempdb** wird durch das DBMS deutlich stärker als in den Vorgängerversionen benutzt. Darum sollten Sie besonderen Wert auf die Speicherung dieser Datenbank legen.

SQL Server nutzt **tempdb** nicht nur für temporäre Objekte, sondern auch für Arbeitstabellen bei Gruppierungs- und Sortierungsfunktionen, Arbeitstabellen bei der Verwendung von Mengen- und Bereichsoperatoren, Arbeitstabellen für die Unterstützung von Cursors, den Versionsspeicher für die Snapshotisolierungsstufe und als Puffer für Tabellenvariablen. Außerdem, wenn festgelegt, auch für das Erstellen von Indizes.

Dadurch werden unter Umständen sehr große Datenmengen in diese Datenbank geschrieben.

Sie sollten der **tempdb** einen anderen Festplattensatz als ihren Datenbanken, den System- als auch Benutzerdatenbanken, und Sicherungsdateien zuweisen.

Das kann schon während der Installation von SQL Server geschehen oder nachträglich mit der ALTER DATABASE- Anweisung.

10.4.3.5 Datenbankoptionen

Für jede Datenbank können eine Reihe von Datenbankoptionen festgelegt werden, durch welche die Eigenschaften einer Datenbank bestimmt werden.

Nur der **Systemadministrator**, der **Datenbankbesitzer** sowie Mitglieder der Serverrollen **sysadmin** und **dbcreator** und der festen Datenbankrolle **db_owner** können diese Optionen ändern.

Diese Optionen sind für jede Datenbank eindeutig und haben keinen Einfluss auf andere Datenbanken. Die Datenbankoptionen können mithilfe der SET- Klausel der ALTER DATABASE- Anweisung oder in einigen Fällen auch mit SQL Server Management Studio festgelegt werden.

Nachdem Sie eine Datenbankoption festgelegt haben, wird automatisch ein Prüfpunkt ausgegeben, so dass die Änderung sofort wirksam wird.

Es gibt fünf Kategorien von Datenbankoptionen:

- Automatische Optionen
- Cursoroptionen
- Wiederherstellungsoptionen
- SQL-Optionen
- Statusoptionen

Nachfolgend eine kleine Übersicht über wichtige Datenbankoptionen.

| Kategorie | Option | Beschreibung |
|----------------------|--|---|
| Automatisch | auto_create_statistics | Erstellt automatisch fehlende Statistiken für die Optimierung einer Abfrage. Standard: ON |
| | auto_update_statistics | Automatisiert automatisch veraltete Statistiken. Standard: ON |
| | auto_close | Wenn der letzte Benutzer der Datenbank seine Anwendung beendet, wird die Datenbank geschlossen. Standard: OFF |
| | auto_shrink | Verkleinert die Datenbank automatisch. Standard: OFF |
| Verfügbarkeit | offline online emergency | Steuert, ob die Datenbank sich im Online- oder Offlinemodus befindet. Emergency hindert Benutzer die nicht zur Gruppe der Systemadministratoren gehören, eine Verbindung zur Datenbank herzustellen und ermöglicht nur den schreibgeschützten Zugriff. Standard: ONLINE |
| | read_only read_write | Steuert, ob Benutzer Daten ändern können. Standard: READ_WRITE |
| | singel_user restricted_user multi_user | Steuert, welche Benutzer eine Verbindung mit der Datenbank herstellen können. Bei SINGLE_USER kann nur ein Benutzer eine Verbindung herstellen, bei RESTRICTED_USER können das nur Benutzer der DB- Rolle db_owner und der Serverrollen dbcreator und sysadmin. Standard: MULTI_USER |

| Kategorie | Option | Beschreibung |
|-------------------|------------------------|--|
| Cursor | cursor_close_on_commit | Führt eine Transaktion einen Cursor aus wird bei COMMIT der Cursor geschlossen. Standard: OFF |
| | cursor_default_local | Schränkt den Bereich des Cursors ein. Standard: CURSOR_DEFAULT_GLOBAL |
| Wiederherstellung | recovery | Bestimmt in welchem Wiederherstellungsmodell die Datenbank läuft. Standard: FULL |
| | page_verify | Ermöglicht SQL Server unvollständige E/A- Vorgänge zu erkennen die beispielsweise durch Stromausfälle verursacht werden. Standard: CHECKSUM |
| SQL | ansi_null_default | Steuert die NULL- Zulässigkeit beim Erstellen von Tabellen ohne Angabe der NULL- Regel. Standard: OFF (not null) |
| | ansi_nulls | Bei ON ergeben alle Vergleiche mit einem NULL- wert den Wert NULL. Standard: OFF |
| | recursive_triggers | Steuert, ob das rekursive Auslösen von AFTER- Triggern zulässig ist. Standard: OFF |

Weiter Optionen und detaillierte Beschreibungen sind in der Onlinehilfe unter "ALTER DATABASE" zu finden.

10.4.3.6 Anzeigen von Datenbankinformationen

Die Datenbankoptionen können durch SQL Abfragen auf die entsprechenden Systemdatenbanken abgerufen werden oder durch die Anwendung gespeicherter Systemprozeduren. Weiterhin können über Berichte im SQL Server Management Studio Informationen über die Datenbank angezeigt werden.

Berichte in SQL Server Management Studio

- Informationen über Server
- Informationen über die Datenbanken
- Service Broker innerhalb des Knotens Datenbank
- Anmeldungen innerhalb des Knotens Sicherheit
- Verwaltung

10.4.3.6.1 Katalogsichten

Anzeigen von Metadaten zu Datenbankobjekten. Katalogsichten und auch Informationsschemasichten werden für jede Datenbank im Ordner Sichten\Systematischeien aufgeführt. Obwohl diese Sichten wie benutzerdefinierte Sichten mit Transact- SQL Syntax abgefragt werden, werden diese Sichten nicht als herkömmliche Sichten mit zugrundeliegenden Tabellen implementiert, sondern fragen stattdessen die Systemmetadaten direkt ab. Es gibt mehr als 200 Katalogsichten die im Schema SYS definiert sind.

Nachfolgende Tabelle führt einige am häufigsten verwendeten Katalogsichten auf.

| Kategorie | Katalogsicht | Beschreibung |
|--------------------------------|---------------------------|---|
| Datenbanken und Dateien | sys.databases | Gibt für jede Datenbank auf dem Server eine Zeile zurück. |
| | sys.database_files | Gibt für jede Datei einer Datenbank eine Zeile zurück. |
| Datenbankobjekte | sys.columns | Gibt für jede Spalte einer Tabelle oder Sicht eine Zeile zurück. |
| | sys.events | Gibt für jedes Trigger-Ereignis oder Benachrichtigung eine Zeile zurück. |
| | sys.indexes | Gibt für jeden Index oder HEAP eine Zeile zurück. |
| | sys.tables | Gibt für jede Tabelle in der Datenbank eine Zeile zurück. |
| | sys.views | Gibt für jede Sicht in der Datenbank eine Zeile zurück. |
| Schemas | sys.schema | Gibt für jedes Schema in der Datenbank eine Zeile zurück. |
| Sicherheit | sys.database_permissions | Gibt für jede in der Datenbank definierte Berechtigung eine Zeile zurück. |
| | sys.database_principals | Gibt für jeden Sicherheitsprinzipal eine Zeile zurück. |
| | sys.database_role_members | Gibt für jedes Mitglied einer Datenbankrolle eine Zeile zurück. |

10.4.3.6.2 Metadatenfunktionen

Verschiedene Funktionskategorien geben Informationen zur Datenbank und zu den Datenbankobjekten zurück. Diese Funktionen geben nur einen einzelnen Wert zurück.

Häufig verwendete Metadatenfunktionen.

| | |
|-------------------------|--|
| db_id() | Gibt die Datenbank-ID für die angegebene Datenbank zurück. Wird kein Name angegeben, dann wird die ID der aktuellen Datenbank zurückgegeben. |
| db_name() | Gibt den Datenbankname für eine angegebene Datenbank-ID zurück. Wird keine ID angegeben, dann wird der Name der aktuellen Datenbank zurückgegeben. |
| file_id() | Gibt die ID des angegebenen logischen Dateinamen in der aktuellen Datenbank zurück. |
| file_name() | Gibt den logischen Dateiname für die angegebene Datei-ID zurück |
| filegroup_id() | Gibt die Dateigruppen-ID für einen angegebenen Dateigruppennamen zurück. |
| filegroup_name() | Gibt den Dateigruppenname für eine angegebene Dateigruppen-ID zurück. |
| object_id() | Gibt die Objekt-ID für ein angegebenes Datenbankobjekt zurück. |
| object_name() | Gibt den Objektname für eine angegebene Objekt-ID zurück. |

10.4.3.6.3 Gespeicherte Systemprozeduren

Mit gespeicherten Systemprozeduren können ebenfalls Datenbankmetadaten abgerufen werden. Sie stellen eine alternative Art der Abfrage von in Katalogsichten gespeicherten Informationen. Es gibt mehrere hundert gespeicherte Systemprozeduren.

Einige gespeicherten Systemprozeduren die häufig verwendet werden.

| Prozedur | Beschreibung |
|----------------------|---|
| sp_helpdb | Gibt Informationen über alle Datenbanken auf dem Server zurück. Wird ein Datenbankname angegeben, dann werden ausführliche Informationen zur angegebenen Datenbank angezeigt. |
| sp_helpfile | Gibt Informationen über die Daten- und Protokolldateien zurück. Wird ein logischer Dateiname angegeben, dann werden ausführliche Informationen zur angegebenen Datei angezeigt. |
| sp_helpfilegroup | Gibt Informationen über die Dateigruppen der aktuellen Datenbank zurück. Wird ein logischer Dateigruppenname angegeben, dann werden ausführliche Informationen zur angegebenen Dateigruppe angezeigt. |
| sp_help | Gibt Informationen über alle Datenbankobjekte der aktuellen Datenbank zurück. Wird ein Objektname (Tabelle oder Sicht) angegeben, dann werden ausführliche Informationen über das angegebene Objekt angezeigt. |
| sp_database | Führt Datenbanken auf, die in einer Instanz verfügbar sind oder auf die über ein Verbindungsserver zugegriffen werden kann. |
| sp_stored_procedures | Gibt eine Liste der gespeicherten Prozeduren in der aktuellen Datenbank zurück. |
| sp_spaceused | Gibt die Anzahl der Zeilen sowie den zugeordneten und verwendeten Speicherplatz für eine bestimmte Tabelle, eine indizierte Sicht oder eine SQL Server Service Broker-Warteschlange in der aktuellen Datenbank bzw. den zugeordneten und verwendeten Speicherplatz für die gesamte Datenbank an |

10.4.3.7 Ändern von Datenbankoptionen

Optionen an Datenbanken können auf verschiedenen Einstellungsebenen festgelegt werden.

Mit der Prozedur **sp_configure** können globale Konfigurationseinstellungen für den aktuellen Server angezeigt oder geändert werden. Mit der Anweisung **alter database...** können alle Datenbankoptionen für eine spezielle Datenbank verändert werden. Mit der **SET-** Anweisung kann man die Datenbankoptionen für die aktuelle Datenbank auf Sitzungsebene (die aktuelle Verbindung betreffend) ändern.

Einstellungen die mit **sp_configure** vorgenommen wurden, können durch **alter database...** außer Kraft gesetzt werden. Diese wiederum können mit der **SET-** Anweisung für den Zeitraum einer Sitzung verändert werden.

Den Status einzelner Datenbankoptionen bestimmt man mit **databasepropertyex()**.

Beispiel:

```
select databasepropertyex('standard', 'IsANSINullDefault');
```

Beispiel:

Serverweite Festlegung für das kaskadieren von Triggern.

```
sp_configure 'nested triggers', '1';
```

Festlegung der NULL- Zulässigkeit für die Datenbank "standard". In diesem Fall wird die Serverseitige Einstellung auf Datenbankebene verändert.

```
alter database standard  
set ansi_null_default off;
```

oder

Festlegen der NULL- Zulässigkeit für die aktuelle Sitzung. Wenn diese Anweisung in der Datenbank "standard" ausgeführt wird, ist die Einstellung für die Dauer dieser Verbindung außer Kraft gesetzt.

```
set ansi_null_dflt_on on;
```

10.4.4 Ändern von Datenbanken

Nachdem eine Datenbank erstellt wurde, können Änderungen an der ursprünglichen Definition vorgenommen werden. Folgende Änderungen sind möglich:

- Erweitern des Speicherplatzes, der in der Datenbank für Daten oder Transaktionsprotokolle reserviert ist.
- Verkleinern des Speicherplatzes, der in der Datenbank für Daten und Transaktionsprotokolle reserviert ist.
- Hinzufügen oder Entfernen von Daten- und Transaktionsprotokolldateien.
- Erstellen von Dateigruppen.
- Ändern der Standarddateigruppe.
- Ändern der Konfigurationseinstellungen für die Datenbank.
- Verwenden des Offlinestatus für Datenbanken.
- Anfügen neuer und Trennen von nicht verwendeten Datenbanken.
- Ändern des Namens der Datenbank.
- Ändern des Kompatibilitäts- Levels der Datenbank.
- Ändern des Besitzers der Datenbank.

Bevor die Datenbank geändert wird, muss eventuell der normale Betriebsmodus der Datenbank beendet werden. Bestimmen Sie in diesen Situationen die entsprechende Methode, um Transaktionen zu beenden.

10.4.4.1 Vergrößern von Datenbanken- und Protokolldateien

Zum Vergrößern kann der SSMS oder die Anweisung "ALTER DATABASE..." (sollte nur in **master**- DB ausgeführt werden) verwendet werden.

Standardmäßig erweitert SQL Server 2012 eine Datenbank automatisch gemäß den Vergrößerungsparametern, die beim Erstellen der Datenbank definiert wurden.

Sie können eine Datenbank auch manuell erweitern, indem Sie einer vorhandenen Datenbankdatei zusätzlichen Speicherplatz zuordnen oder eine neue Datei erstellen.

Das Erweitern des Speicherplatzes für Daten und Transaktionsprotokolle kann notwendig werden, wenn vorhandene Dateien vollständig gefüllt sind. Wenn eine Datenbank bereits den gesamten zugeordneten Speicherplatz verbraucht hat und eine automatische Vergrößerung der Datenbank nicht mehr möglich ist.

Wenn keine automatische Vergrößerung der Datenbank eingerichtet wurde oder wenn nicht ausreichend Speicherplatz auf der Festplatte vorhanden ist wird der **Fehler 1105** ausgelöst.

Für alle Änderungen an einer Datenbank wird nachfolgende SQL Anweisung verwendet.

Syntax:

```

ALTER DATABASE database_name
  {ADD FILE <filespec> [,...n] [TO FILEGROUP filegroup_name]
  | ADD LOG FILE <filespec> [,...n]
  | REMOVE FILE logical_file_name
  | MODIFY FILE <filespec>
  | ADD FILEGROUP filegroup_name [CONTAINS FILESTREAM]
  | REMOVE FILEGROUP filegroup_name
  | MODIFY FILEGROUP filegroup_name {<filegroup_property>
    | DEFAULT
    | NAME = new_filegroup_name}
  | SET <optionspec> [,...n] [with <termination>]
  | MODIFY NAME = new_database_name
  | COLLATE <collationsname>}

<filespec> ::=
(
  NAME = logical_file_name
  [, NEWNAME = new_logical_name]
  [, FILENAME = {'os_file_name' | 'filestream_pfad'}]
  [, SIZE = size {KB | MB | GB | TB}]
  [, MAXSIZE = { max_size {KB | MB | GB | TB} | UNLIMITED }]
  [, FILEGROWTH = vergrößerung {KB | MB | GB | TB}]
  [, OFFLINE]
)

<filegroup_property>::=
{ READONLY | READWRITE } | {READ_ONLY | READ_WRITE}

<termination>::=
{ ROLLBACK AFTER integer [SECONDS]
  | ROLLBACK IMMEDIATE
  | NO WAIT}

< optionspec>::=
{<db_state_option>
  |<db_user_access_option>
  |<db_update_option> | <external_access_option>
  |<cursor_option>
  |<auto_option>
  |<sql_option>
  |<recovery_option>
  |<database_mirroring_option>
  |<service_broker_option>
  |<date_correlation_optimization_option>
  |<compatibility_level>
  |<parameterization_option>}
```

Für weitere Informationen, vor allem was die **set- Optionen** betrifft, benutzen Sie bitte die Onlinedokumentation unter dem Begriff **alter database...**.

10.4.4.1.1 Automatische Vergrößerung

Für jede Datendatei kann die automatische Vergrößerung mit Hilfe des SQL Server Management Studio oder der Anweisung „ALTER DATABASE...“ eingestellt werden. Legen sie einfach eine Anfangsgröße, eine Schrittweite der automatischen Vergrößerung und eine Maximalgröße fest.

10.4.4.1.2 Vergrößerung der Datenbankdateien

Wenn die automatische Verkleinerung der Datenbankdatei ausgeschalten ist (Schrittweite auf 0), kann die Datenbank dennoch vergrößert werden.

Beispiele:

1. Vergrößern einer Datenbank auf 15MB, danach Hochsetzen der Maximalgröße auf 60MB und Einstellen der Schrittweite für die automatische Vergrößerung auf 2MB.

```
use master;
go

alter database Standard
modify file
(
    NAME = 'Standard_Daten',
    SIZE = 15MB
);

alter database Standard
modify file
(
    NAME = 'Standard_Daten',
    MAXSIZE = 60MB
);

alter database Standard
modify file
(
    NAME = 'Standard_Daten',
    FILEGROWTH = 2MB
);
```

2. Vergrößern der Datenbank „Standard“ durch Hinzufügen einer sekundären Datendatei welche in eine Dateigruppe mit dem Namen "aktive" gestellt wird. danach wird die neue Dateigruppe als Standarddateigruppe definiert.

```
use master;
go

alter database standard
add filegroup active;
go

alter database standard
add file
(
    name = 'standard_daten3',
    filename = 'e:\mssql7\data\standard_daten3.ndf',
    size = 10mb,
    maxsize = 20mb
) to filegroup active;
go

alter database standard
modify filegroup aktive default;
```

10.4.4.1.3 Vergrößern von Transaktionsprotokollen

Wenn für das Transaktionsprotokoll nicht genügend Speicherplatz zur Verfügung steht, können keine Transaktionen mehr aufgezeichnet werden und der SQL Server lässt keine Änderungsoperationen an der Datenbank mehr zu.

Die Transaktionsprotokolle (und auch Datenbanken) können mit Hilfe des Systemmonitors von Windows (Objekt: SQL Server: Datenbanken) überwacht werden.

Dabei können folgende Leistungsindikatoren (Auswahl) beobachtet werden:

| | |
|---------------------------------------|--|
| Protokolldatei(en) Größe (KB) | Die kumulierte Größe aller Protokolldateien in der Datenbank. |
| Wartezeit für Protokolleerung | Gesamte Wartezeit (in Millisekunden) bis zum Entleeren des Protokolls. |
| Ausstehende Protokolleerungen/Sekunde | Anzahl der Commits pro Sekunde, die auf eine Protokolleerung warten. |
| Protokolleerungen/Sekunde | Anzahl der Protokolleerungen pro Sekunde. |
| Protokollvergrößerungen | Gesamtzahl der Protokollvergrößerungen für diese Datenbank. |
| Protokollverkleinerungen | Gesamtzahl der Protokollverkleinerungen für diese Datenbank. |
| Protokollkürzungen | Gesamtzahl der Protokollkürzungen für diese Datenbank. |
| Protokoll verwendet (Prozent) | Der prozentuale Anteil des Speicherplatzes im Protokoll, der verwendet wird. |

Transaktionsprotokolle können mit SQL Server Management Studio oder mit der Anweisung ALTER DATABASE... vergrößert werden.

Beispiele:

1. Vergrößern eines vorhandenen Transaktionsprotokolls auf 15 MB.

```
use master;
go
alter database Standard
    modify file
    (
        NAME = 'Standard_Protokoll',
        SIZE = 15MB
    );
```

2. Hinzufügen einer neuen Protokolldatei zur Datenbank 'Standard'

```
use master;
go
alter database Standard
    add log file
    (
        NAME = 'Standard_Protokoll1',
        FILENAME = 'e:\mssql7\data\standard_protokoll1.ldf',
        SIZE = 10MB,
        MAXSIZE = 20MB
    );
```

10.4.4.2 Verkleinern von Datenbanken und Dateien

Die gesamte Datenbank oder einige ihrer Datendateien kann entweder mit dem SQL Server Management Studio, mit der Anweisung "DBCC SHRINKDATABASE..." oder mit der Anweisung "DBCC SHRINKFILE..." verkleinert werden.

10.4.4.2.1 Automatisches Verkleinern einer Datenbank

Durch Festlegen der Datenbankoption AUTOSHRINK auf den Wert TRUE wird nicht belegter Speicherplatz automatisch wiederhergestellt. Diese Option kann mit der Anweisung ALTER DATABASE... oder mit dem SQL Server Management Studio geändert werden.

Auf die automatische Verkleinerung sollte bei sehr produktiven Datenbanken verzichtet werden, da der Vorgang sehr zeitaufwendig ist.

10.4.4.2.2 Verkleinern der gesamten Datenbank

Das Verkleinern einer gesamten Datenbank kann nur in der master- DB vorgenommen werden. Die Zielgröße für Daten- und Protokolldateien, die von DBCC SHRINKDATABASE berechnet wird, kann niemals kleiner sein als die Mindestgröße (Option: SIZE) einer Datei. Die Mindestgröße einer Datei ist die Größe, die bei der Erstellung der Datei angegeben wird.

Beispiel:

Wurde beispielsweise die Größe der Daten- und Protokolldateien der Datenbank "Standard" zum Ausführungszeitpunkt von CREATE DATABASE... mit jeweils 10 MB angegeben, liegt die Mindestgröße der jeweiligen Dateien bei 10 MB.

Die Datenbank kann nicht kleiner als die Größe der model- Datenbank werden.

Die Datenbank, die verkleinert werden soll, muss sich nicht im Einzelbenutzermodus befinden. Andere Benutzer können während der Verkleinerung darin arbeiten. Das gilt auch für Systemdatenbanken.

Syntax:

```
DBCC SHRINKDATABASE
  (database_name [, ziel_prozent]
   [, {NOTRUNCATE | TRUNCATEONLY}])
```

database_name: Der Name der Datenbank, die verkleinert werden soll.

ziel_prozent: Der gewünschte Prozentsatz an freiem Speicherplatz, der in der Datenbankdatei übrig bleiben soll, nachdem die Datenbank verkleinert wurde.

Bei der Anwendung auf Datendateien stehen für DBCC SHRINKDATABASE die Optionen NOTRUNCATE und TRUNCATEONLY zur Verfügung. Bei Protokolldateien werden beide Optionen ignoriert. DBCC SHRINKDATABASE ohne Option ist gleichwertig mit der Auseinanderfolgenden Ausführung von DBCC SHRINKDATABASE mit der Option NOTRUNCATE gefolgt von DBCC SHRINKDATABASE mit der Option TRUNCATEONLY.

NOTRUNCATE: Diese Option, mit oder ohne Angabe von target_percent, führt die tatsächlichen Datenbewegungen von DBCC SHRINKDATABASE aus, einschließlich der Bewegung von reservierten Seiten am Dateiende zu nicht reservierten Seiten am Dateianfang. Jedoch wird der freie Speicherplatz am Dateiende nicht dem Betriebssystem zur Verfügung gestellt, und die physikalische Größe der Datei bleibt unverändert. Deshalb wirkt es so, als würden Datendateien bei Angabe dieser Option nicht verkleinert.

TRUNCATEONLY: Diese Option stellt den freien Bereich am Ende der Datei dem Betriebssystem zur Verfügung. Jedoch werden bei dieser Option keine Seiten innerhalb der Datei(en) verschoben. Die angegebene Datei wird nur bis zum letzten reservierten Block verkleinert, target_percent wird ignoriert, wenn es mit der Option TRUNCATEONLY angegeben wird.

DBCC SHRINKDATABASE verkleinert Datendateien pro Datei. Jedoch werden von DBCC SHRINKDATABASE die Protokolldateien so verkleinert, als lägen alle Protokolldateien in einem zusammenhängenden Protokollpool vor.

Beispiel:

Die Datenbank Standard besteht aus zwei Daten- und aus zwei Protokolldateien. Jede Datei ist 10 MB groß. Die Datenbank wollen wir so weit reduzieren, dass 25 % freier Speicherplatz übrig bleiben. Wenn die erste Datendatei der Datenbank zum Beispiel 6MB Daten enthält, beträgt die neue Größe der dieser Datendatei 8MB (6MB für die Daten und 2 MB freier Speicherplatz).

```
use master;
go
DBCC SHRINKDATABASE ('Standard', 25);
```

10.4.4.2.3 Verkleinern einer Datendatei in der Datenbank

Das Verkleinern einer Datendatei kann nur in der aktuellen Datenbank durchgeführt werden.

Syntax:

```
DBCC SHRINKFILE
  ({dateiname | datei_ID} [, ziel_größe]
  [, {EMPTYFILE | NOTRUNCATE | TRUNCATEONLY}]
  )
```

ziel_größe: Gibt die endgültige Größe in MB an.

EMPTYFILE: Verlagert alle Daten der angegebenen Datei in andere Dateien der gleichen Dateigruppe. Mit SQL Server ist es nicht mehr möglich, Daten in der Datei abzulegen, die mit dieser Option verwendet wurde. Diese Option ermöglicht das Löschen der Datei mit der ALTER DATABASE- Anweisung.

NOTRUNCATE: Bewirkt, dass der freigegebene Dateispeicherplatz in den Dateien zurück behalten wird. Die einzige Wirkung von DBCC SHRINKFILE besteht darin, dass verwendete Seiten von oberhalb der target_size Linie an den Anfang der Datei verschoben werden. Falls NOTRUNCATE nicht angegeben ist, wird der gesamte freigegebene Speicherplatz an das Betriebssystem zurückgegeben.

TRUNCATEONLY: Bewirkt, dass unbenutzter Speicherplatz in den Dateien an das Betriebssystem freigegeben und die Datei auf die zuletzt reservierte Größe verkleinert wird, wodurch die Dateigröße ohne die Verschiebung von Daten reduziert wird. Es wird nicht versucht, Zeilen auf nicht reservierte Seiten zu verschieben. Bei Verwendung von TRUNCATEONLY wird target_size ignoriert.

Mit dieser Anweisung ist es möglich eine Datenbank auf eine Größe unter die Größe bei der Erstellung der Datenbank zu bringen. Die Datei leeren und mit EMPTYFILE markieren. Danach kann die Datei gelöscht werden.

Es ist aber nicht möglich alle Dateien aus einer Dateigruppe zu löschen solange darin noch Objekte gespeichert sind.

Beispiel:

Entfernen der zweiten Datendatei von der Datenbank "Standard"

```
use standard;
go
DBCC SHRINKFILE (Standard_daten2, EMPTYFILE);
go
use master;
ALTER DATABASE Standard
    remove file standard_daten2;
go
```

10.4.5 Löschen von Datenbanken

Beim Löschen einer Datenbank gehen alle Daten und zugehörigen Festplattendateien verloren. Das Löschen einer Datenbank erfolgt in der Master Datenbank.

Syntax:

```
drop database {datenbank_name | database_snapshot_name}[ , ...n]
```

Eine Datenbank kann nur von Mitgliedern der Serverrolle sysadmin, von Mitgliedern der Serverrolle dbcreator oder vom DBO oder gelöscht werden.

Mit dem SQL Server Management Studio kann immer nur jeweils eine Datenbank gelöscht werden. Interaktiv können mehrere Datenbanken gleichzeitig gelöscht werden.

Unter folgenden Umständen kann eine Datenbank nicht gelöscht werden:

- Die Datenbank ist gerade in Benutzung.
- Die Datenbank wird gerade wiederhergestellt.
- Datenbanken, deren Tabellen für die Replikation publiziert werden.
- Die Datenbank befindet sich im schreibgeschützten Modus.

Systemdatenbanken (msdb, master, model, tempdb) können nicht gelöscht werden.

10.4.6 Datenbanksnapshot

Ein Datenbanksnapshot, ist eine in SQL Server 2008 neu eingeführte Funktion

Mit der CREATE DATABASE- Anweisung kann ein Schnapschuss einer Benutzerdatenbank erstellt werden. Es handelt sich hierbei um eine schreibgeschützte statische Sicht einer vorhandenen Datenbank. Ein Datenbanksnapshot ist im Hinblick auf Transaktionen konsistent mit der Quelldatenbank zu dem Zeitpunkt, an den der Snapshot erstellt wurde.

Für eine Quelldatenbank kann es mehrere Snapshots geben.

Jeder Snapshot bleibt solange bestehen, bis er mit DROP DATABASE- Anweisung gelöscht wird.

Snapshots können im Rahmen der Berichterstellung verwendet werden. Darüber hinaus bieten sie die Möglichkeit, die Quelldatenbank wieder in den Zustand zum Zeitpunkt der Snapshoterstellung zu versetzen, wenn bei einer Quelldatenbank ein Benutzerfehler auftritt. Damit wird der Datenverlust auf Updates beschränkt, die nach der Snapshot- Erstellung erfolgten.

Syntax:

```
CREATE DATABASE database_snapshot_name
ON
(
    NAME = logical_file_name,
    FILENAME = 'os_file_name'
) [...n ]
AS SNAPSHOT OF source_database_name
```

Damit der Datenbanksnapshot funktionsfähig ist, müssen alle Datendateien einzeln angegeben werden. Protokolldateien sind für Datenbanksnapshots nicht zulässig.

Beispiel:

Erstellen eines Snapshots für die Datenbank STANDARD. Die Quelldatenbank besitzt drei Datendateien.

```
create database standard_snapshot1600
on (
    name = 'standard_kat',
    filename = 'g:\microsoft sql server\mssql.1\mssql\data\standard_kat.ss'
),
(
    name = 'standard_daten1',
    filename = 'g:\microsoft sql server\mssql.1\mssql\data\standard_daten1.ss'
),
(
    name = 'standard_daten2',
    filename = 'g:\microsoft sql server\mssql.1\mssql\data\standard_daten2.ss'
)
as snapshot of standard;
```

10.4.7 Datenintegrität pflegen

Da nicht immer, alles was auf Festplatte gespeichert wird, richtig geschrieben wird und gelesen wird ist es notwendig in regelmäßigen Abständen die Datenbankintegrität zu überprüfen.

Eine Datenbankoption heißt PAGE_VERIFY, also Seitenüberprüfung. Sie können die Werte TORN_PAGE_DETECTION (nur noch aus Gründen der Abwärtskompatibilität) oder CHECKSUM einstellen.

Bei CHECKSUM berechnet SQL Server für die Seite eine Prüfsumme, bevor sie auf Festplatte geschrieben wird. Jedes Mal, wenn eine Seite wieder von Festplatte gelesen wird, wird eine Prüfsumme über die gelesenen Daten berechnet und mit der ursprünglich geschriebenen Prüfsumme verglichen. Bei unterschiedlichen Prüfsummen, ist die Seite beschädigt.

Erkennt SQL Server einen beschädigte Seite, meldet er den Fehler und der Befehl welcher auf die beschädigte Seite zugreifen wollte wird beendet. In der Tabelle **suspect_pages** der Datenbank **msdb** wird der Fehler eingetragen. Dort können maximal 1000 Fehler eingetragen werden. Ist diese Zahl erreicht schaltet SQL Server die Datenbank offline.

Da es besser ist nicht erst auf einen Fehlermeldung an den Benutzer zu warten, sollte vorbeugend nach Beschädigungen gesucht werden und eventuelle Probleme mit Hilfe der Datensicherung repariert werden.

Mit dem Befehl DBCC CHECKDB können Sie SQL Server zwingen jede Seite von Festplatte zu lesen und die Integrität zu überprüfen.

Syntax:

```
DBCC CHECKDB [( database_name | database_id | 0
[, NOINDEX
[, { REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST | REPAIR_REBUILD } ]])
[ WITH
{[ ALL_ERRORMSG ] [, EXTENDED_LOGICAL_CHECKS ] [, NO_INFOMSGS ]
[, TABLOCK ] [, ESTIMATEONLY ] [, { PHYSICAL_ONLY | DATA_PURITY } ]}]]
```

Wird dieser Befehl ausgeführt, werden vom Server folgende Aktionen durchgeführt:

- Prüft die Seitenzuordnung innerhalb der Datenbank.
- Prüft die strukturelle Integrität aller Tabellen und indizierten Sichten.
- Berechnet für alle Daten- und Indexseiten eine Prüfsumme und vergleicht sie mit der gespeicherten Prüfsumme.
- Prüft den Inhalt jeder indizierten Sicht.
- Überprüft den Datenbankkatalog.
- Prüft die Service Broker- Daten innerhalb der Datenbank.

Um diese Überprüfungen durchzuführen, führt DBCC CHECKDB folgende Befehle aus:

- Die Ausführung von DBCC CHECKALLOC für die Datenbank.
- Die Ausführung von DBCC CHECKTABLE für jede Tabelle und Sicht in der Datenbank.
- Die Ausführung von DBCC CHECKCATALOG für die Datenbank.

Alle gefundenen Fehler werden ausgegeben, damit kann der Fehler beseitigt werden.
Bei einem Integritätsfehler in einem Index, wird der Index gelöscht und neu erstellt.
Werden Integritätsfehler in einer Tabelle gefunden, muss die beschädigte Seite mithilfe der letzten Datensicherung repariert werden.

11 SQL Server Tabellen

Tabellen sind Datenbankobjekte, die sämtliche in einer Datenbank enthaltenen Daten umfassen. Eine Tabellendefinition ist eine Auflistung von Spalten. Jede Zeile stellt einen eindeutigen Datensatz und jede Spalte ein Feld innerhalb des Datensatzes dar.

Tabellen sind die einzigen Datenbankobjekte die auf interner Ebene durch Daten repräsentiert werden.

Beim Entwurf einer Datenbank wird festgelegt, welche Tabellen benötigt werden, welche Typen von Daten die einzelnen Tabellen enthalten sollen und vieles mehr.

Die effektivste Methode, eine Tabelle zu erstellen, besteht darin, im Voraus alle erforderlichen Merkmale zu definieren, einschließlich der Datenbeschränkungen und zusätzlicher Komponenten. Diese Merkmale der Basistabelle können auch nachträglich erstellt werden, wenn die Tabelle schon mit Daten gefüllt ist und damit gearbeitet wird. Dieser Ansatz bietet die Möglichkeit, herauszufinden, welche Typen von Transaktionen am häufigsten vorkommen und welche Typen von Daten regelmäßig eingegeben werden, bevor man sich schließlich für einen festen Aufbau entscheidet und Einschränkungen, Indizes, Standards, Regeln und andere Objekte hinzufügt.

Es empfiehlt sich, die Entwürfe vor dem Erstellen der Tabelle und ihrer Objekte zunächst auf Papier zu skizzieren. Folgende Entscheidungen müssen getroffen werden:

- Die Typen der Daten, die die Tabelle enthalten wird.
- Die Spalten in der Tabelle und die Datentypen (und gegebenenfalls Größen) für jede Spalte.
- Die Spalten, die NULL- Werte zulassen.
- Ob und, wenn ja, wo Einschränkungen oder Standards und Regeln verwendet werden.
- Die Typen der benötigten Indizes, wo diese Indizes erforderlich sind und welche Spalten als Primär- und welche als Fremdschlüssel dienen sollen.

Regeln für reguläre Bezeichner für Kompatibilitätsgrad:

Die Regeln für das Format von regulären Bezeichnern hängen vom Kompatibilitätsgrad der Datenbank ab, der mit der nachfolgenden Anweisung festgelegt werden kann.

```
ALTER DATABASE datenbank_name  
SET COMPATIBILITY_LEVEL = {90 | 100 | 110}
```

Zum Anzeigen des aktuellen Kompatibilitätsgrades der Datenbank fragen Sie die compatibility_level- Spalte in der sys.databases- Katalogsicht ab.

Dabei bedeutet der Wert 90, SQL Server 2005, der Wert 100, SQL Server 2008 bzw. SQL Server 2008 R2 und der Wert 110, SQL Server 2012.

1. Das erste Zeichen muss eines der folgenden Zeichen sein:

- Ein vom Unicode-Standard 3.2 definierter Buchstabe. Die Definition von Buchstaben enthält die lateinischen Buchstaben von a bis z und von A bis Z, zusätzlich zu den Buchstaben anderer Sprachen.
- Das Sonderzeichen Unterstrich (_), At- Zeichen (@) oder Nummernzeichen (#). Bestimmte Sonderzeichen am Anfang eines Bezeichners haben in SQL Server eine besondere Bedeutung. Mit dem Zeichen @ beginnen lokale Variable. Mit dem Zeichen # beginnen temporäre Tabellen oder Prozeduren. Einige Transact- SQL- Funktionen haben Namen, die mit doppelten Zeichen @@ beginnen (@@).

2. Im Anschluss daran können die folgenden Zeichen verwendet werden:
 - Im Unicode-Standard 3.2 definierte Buchstaben.
 - Dezimalzahlen aus dem lateinischen Grundalphabet oder anderen nationalen Schriftarten.
 - Das At-Zeichen, Dollar-Zeichen (\$), Nummernzeichen oder Unterstrich-Zeichen.
3. Der Bezeichner darf kein reserviertes Transact- SQL- Wort sein. SQL Server reserviert sowohl die groß- als auch die kleingeschriebene Version von reservierten Wörtern.
4. Eingebettete Leerzeichen oder Sonderzeichen sind nicht zugelassen.

In Transact- SQL- Anweisungen müssen alle Bezeichner, die diese Regeln nicht einhalten, durch doppelte Anführungszeichen oder eckige Klammern begrenzt werden.

11.1 Erstellen von Datentypen und Tabellen

11.1.1 Datentypen

Jede Spalte, jede lokale Variable, jeder Ausdruck und jeder Parameter besitzt einen entsprechenden Datentyp. Es handelt sich dabei um ein Attribut, das für das jeweilige Objekt angibt, welchen Typ von Daten es aufnehmen kann.

SQL Server stellt eine Reihe von Basisdatentypen zur Verfügung, die alle Typen von Daten definieren, die mit SQL Server verwendet werden können.

Es können auch benutzerdefinierte Datentypen definiert werden. Dabei handelt es sich um Aliasnamen für Datentypen, die das System bereitstellt.

11.1.1.1 Basisdatentypen

Vom SQL Server werden mehrere Basisdatentypen zur Verfügung gestellt.

| | | | | |
|----------------|----------------|-------------|------------------|---------------|
| bigint | binary | bit | char | cursor |
| datetime | decimal | float | image | int |
| money | nchar | ntext | nvarchar | real |
| smalldatetime | smallint | smallmoney | text | timestamp |
| tinyint | varbinary | varchar | uniqueidentifier | xml |
| table | numeric | sql_variant | datetime2 | date |
| time | datetimeoffset | hierarchyid | varchar(max) | nvarchar(max) |
| varbinary(max) | geography | geometry | | |

11.1.1.2 Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen (USER- DEFINED DATATYPE – UDDT) werden immer unter Verwendung eines Basisdatentyps definiert.

Die maximale Größe eines UDT wurde auf 2.147.483.647 Byte erhöht.

Sie stellen einen Mechanismus zum Übernehmen eines Namens für einen Datentyp zur Verfügung, um innerhalb einer Datenbank Konsistenz in Tabellendefinitionen zu gewährleisten. So kann ein Programmierer oder Datenbankadministrator leichter den Verwendungszweck aller mit dem Datentyp definierten Objekte verstehen. Weiterhin wird die Arbeit mit Datentypen dadurch einheitlich und übersichtlich.

Die Namen der Datentypen müssen innerhalb der Datenbank eindeutig sein und den Regeln für Bezeichner entsprechen.

Syntax (Erstellen):

```

create type [schema.] typename
{
  (spaltenname datentyp nullregel [...])
  |
  from basisdatentyp [(genauigkeit [,dezimalstellen])]
  [null | not null]
  |
  external name assemblyname [.klassenname]
}

```

Beispiel:

```
create type orte from nvarchar(50) null;
```

Syntax (Löschen):

```
drop type [schema.] typename
```

Hinweis: Ein benutzerdefinierter Datentyp kann nicht gelöscht werden, wenn der benutzerdefinierte Datentyp in einer Tabellendefinition verwendet wird oder wenn eine Regel oder ein Standard an ihn gebunden ist.

11.1.1.3 Benutzerdefinierte Tabellentypen

Es handelt um einen benutzerdefinierten Typ, der die Definition einer Tabellenstruktur darstellt. Mit einem benutzerdefinierten Tabellentyp können Sie Tabellenwertparameter für gespeicherte Prozeduren oder Funktionen deklarieren oder Tabellenvariablen deklarieren, die in einem Stapel oder im Textteil einer gespeicherten Prozedur oder Funktion verwendet werden sollen

Verwenden Sie zum Erstellen eines benutzerdefinierten Tabellentyps die CREATE TYPE-Anweisung.

Es können eindeutige Einschränkungen und Primärschlüssel für den benutzerdefinierten Tabellentyp erstellt werden.

Einschränkungen:

- Ein benutzerdefinierter Tabellentyp kann nicht als Spalte in einer Tabelle oder als Feld in einem strukturierten benutzerdefinierten Typ verwendet werden.
- Für CHECK- Einschränkungen muss eine berechnete Spalte beibehalten werden.
- Der Primärschlüssel für berechnete Spalten muss PERSISTED und NOT NULL sein.
- Ein nicht gruppierter Index kann nur dann für einen benutzerdefinierten Typ erstellt werden, wenn der Index das Ergebnis der Erstellung einer PRIMARY KEY- oder UNIQUE- Einschränkung des benutzerdefinierten Tabellentyps ist.
- Ein DEFAULT-Wert kann in der Definition eines Tabellentyps nicht angegeben werden.
- Die Definition des benutzerdefinierten Tabellentyps kann nach dem Erstellen nicht geändert werden.
- Benutzerdefinierte Funktionen können in der Definition von berechneten Spalten eines Tabellentyps nicht aufgerufen werden.
- Berechtigungen für Tabellentypen orientieren sich am Objektsicherheitsmodell für SQL Server, indem die folgenden Transact-SQL-Schlüsselwörter verwendet werden: CREATE, GRANT, DENY, ALTER, CONTROL, TAKE OWNERSHIP, REFERENCES, EXECUTE, VIEW DEFINITION und REVOKE.

- Zum Deklarieren einer Tabellenvariablen, die einen Tabellentyp verwendet, ist die EXECUTE- Berechtigung für diesen benutzerdefinierten Tabellentyp erforderlich.

Beispiel:

```
create type ortstabellatype as table
(id int not null
orname nvarchar(200) not null,
region varchar(200))
plz char(5) not null);
go
```

11.1.1.4 Benutzerdefinierte Typen der CLR

Mit CLR- Integration können neben den bereits existierenden Datentypen weitere in der Regel komplexere Datentypen erstellt werden.

Die maximale Größe eines UDT wurde auf 2.147.483.647 Byte erhöht.

Sie sollten die Verwendung benutzerdefinierter Datentypen der CLR nur für kleine, diskrete Datentypen in Betracht ziehen, die deutlich definierte Wertebereiche haben und eine minimale Codemenge in der Datentypdefinition erfordern.

Bevor CLR- Typen erstellt werden können, muss die CLR- Unterstützung aktiviert werden.

```
sp_configure 'clr enabled', 1;
reconfigure with overide;
```

Ein CLR- Typ wird in einer .NET- Programmiersprache, z.B. C# oder VB, erstellt indem eine Klasse erstellt wird die der Spezifikation des CLR- Typs entspricht. Die resultierende DLL kann dann von einem Mitglied der sysadmin- Serverrolle als Assembly in der SQL Server Instanz registriert werden. Danach wird der benutzerdefinierte Datentyp der CLR in der Datenbank implementiert.

Beispiel:

Im nachfolgenden Beispiel wird ein Datentyp erstellt, der für Spalten verwendet werden soll die X/Y- Kordinaten speichern sollen.

```
<Serializable()> -
<Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)> _
Public Structure Point
Implements IComparable
Private m_Null As Boolean
Private _x As Integer
Private _y As Integer

Public Overrides Function ToString() As String
    Return String.Format("Point x: {0}, y: {1}", _x, _y)
End Function

Public ReadOnly Property IsNull() As Boolean Implements IComparable.IsNull
    Get
        Return m_Null
    End Get
End Property

Public Shared ReadOnly Property Null() As Point
    Get
        Dim h As Point = New Point
        h.m_Null = True
        Return h
    End Get
End Property
```

```
Public Shared Function Parse(ByVal s As SqlString) As Point
    If s.IsNull Then
        Return Null
    End If

    Dim u As Point = New Point
    Dim innerString As String = CStr(s)
    Dim parts() As String = innerString.Split(";"c)
    u.X = Integer.Parse(parts(0))
    u.Y = Integer.Parse(parts(1))
    Return u
End Function

Public Function Add(ByVal addPoint As Point) As Point
    X += addPoint.X
    Y += addPoint.Y
    Return Me
End Function

Public Property X() As Integer
    Get
        Return _x
    End Get
    Set(ByVal value As Integer)
        _x = value
    End Set
End Property

Public Property Y() As Integer
    Get
        Return _y
    End Get
    Set(ByVal value As Integer)
        _y = value
    End Set
End Property
End Structure
```

Jetzt muss die DLL als Assembly in der Datenbank registriert werden.

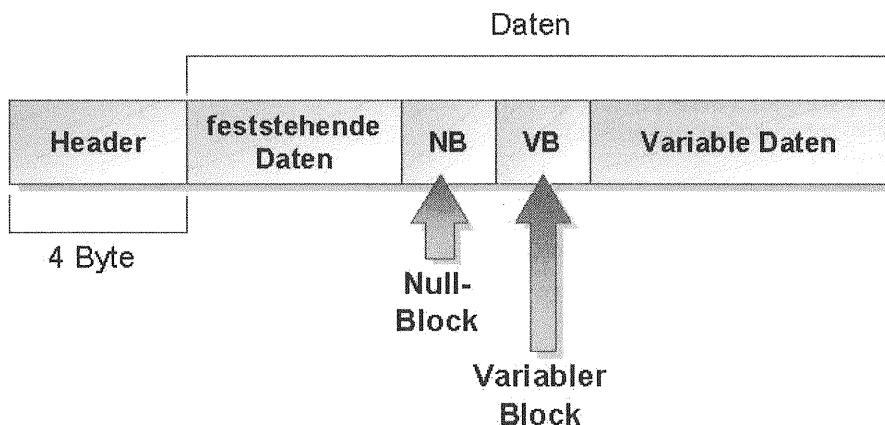
```
create assembly DT_Point
from 'c:\datentypen\piont.dll'
```

Und danach wird in der Datenbank der Datentyp erstellt.

```
create type dbo.piont
external name DT_point.[udt.point]
```

11.1.2 Anordnung der Daten in einer Zeile

Eine Datenzeile besteht aus einem Zeilenkopf und aus einem Datenteil. Für die Planung der Tabellengröße ist es wichtig die Elemente des Datenteils jeder Zeile zu kennen.

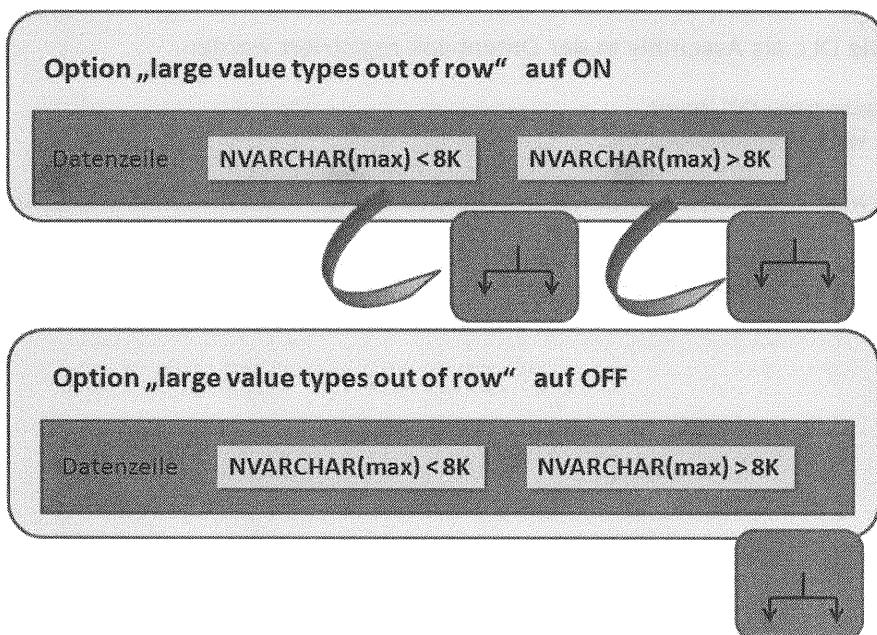


Zeilenkopf: Ist 4 Byte groß und enthält Informationen über die Spalten in der Datenzeile. Zum Beispiel einen Verweis auf die Endposition der unveränderlichen Daten im Datensatz und ob sich Spalten mit variabler Länge im Datensatz befinden.

Datenteil: Hier befinden Daten fester Länge, ein Nullblock in dem die Anzahl der Spalten steht die Nullmarken enthalten, ein variabler Block der angibt wie viel es Spalten mit variabler Länge gibt und am Ende die Daten mit variabler Länge.

11.1.3 Organisation von großen Datentypen

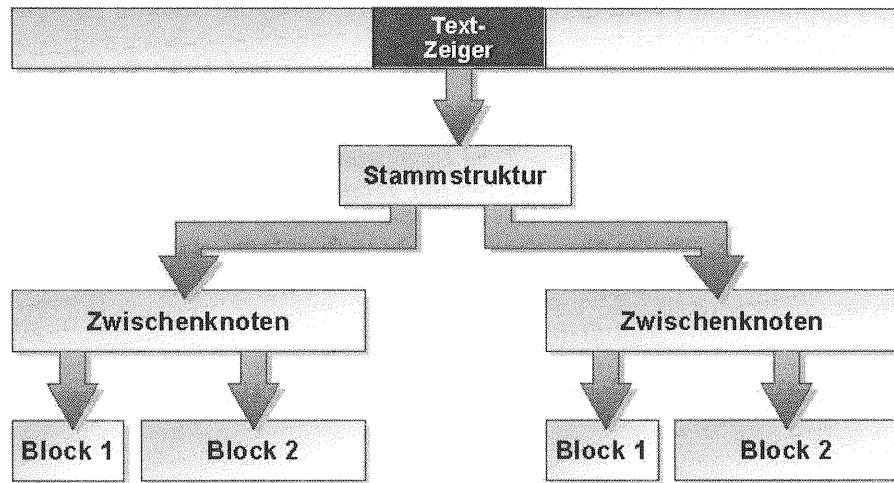
Für die Datentypen **varchar**, **nvarchar** und **varbinary** kann der **max-** Bezeichner verwendet werden. Diese Datentypen bezeichnet man als LOB- Datentypen und dienen zum Speichern von Datenwerten die über 8000 Bytes groß sind.



Die Tabellen Option kann mit der Prozedur **sp_tableoption** ein- oder ausgeschalten werden. Das kann nur auf Tabellen erfolgen die einen der oben genannten Datentypen verwenden

11.1.4 Organisation der Daten vom Typ TEXT, NTEXT und IMAGE

Da Datentypen TEXT, NTEXT und IMAGE in der Regel sehr groß sind, speichert der SQL Server diese Datentypen außerhalb der Datensätze.



Ein 16-Byte-Zeiger in der Datensatzzeile verweist auf eine Stammstruktur, die die Daten enthält.

Bei geringerer bis mittlerer Größe des Inhalts (text, ntext und image) bietet SQL Server die Option, die Werte in der Datenzeile zu speichern anstatt in einer separaten Baumstruktur. Dazu wird die Tabellenoption "TEXT IN ROW" der Prozedur **sp_tableoption** verwendet.

Diese Option wird nur noch aus Gründen der Abwärtskompatibilität bereitgestellt. Verwenden Sie daher die Datentypen varchar(max), nvarchar(max) und varbinary(max).

11.1.5 Spalten mit geringer Dichte

Spalten mit geringer Dichte sind gewöhnliche Spalten, die einen optimierten Speicher für NULL-Werte haben. Spalten mit geringer Dichte reduzieren die Speicherplatzanforderungen von NULL-Werten auf Kosten eines erhöhten Aufwands bei Abfragen auf NichtNULL-Werte. Spalten mit geringer Dichte sollten nur verwendet werden, wenn dadurch mindestens 20 Prozent bis 40 Prozent Speicherplatz eingespart werden. Spalten mit geringer Dichte werden mit dem Schlüsselwort SPARSE beim Erstellen oder Ändern von Tabellen definiert.

Spalten mit geringer Dichte können mit Spaltensätzen und gefilterten Indizes verwendet werden:

Spaltensätze: Die Anweisungen INSERT, UPDATE und DELETE können anhand des Namens auf die Spalten mit geringer Dichte verweisen. Sie können jedoch auch alle Spalten mit geringer Dichte in einer Tabelle anzeigen und mit ihnen arbeiten, wenn sie zu einer einzelnen XML-Spalte zusammengeschlossen werden. Diese Spalte wird dann als Spaltensatz bezeichnet.

Gefilterte Indizes: Da Spalten mit geringer Dichte viele Zeilen mit NULL-Werten haben, sind sie besonders für gefilterte Indizes geeignet. Ein gefilterter Index für eine Spalte mit geringer Dichte kann nur die Zeilen indizieren, die Werte enthalten. Dadurch wird ein kleinerer und effizienterer Index erstellt.

11.1.6 Basistabellen erstellen

In einer Datenbank können maximal 2 Milliarden Tabellen erstellt werden. Eine Tabelle mit einem definierten Spaltensatz (das sind Spalten die mit der Option SPARSE erstellt wurden und mit der COLUMN_SET FOR ALL_SPARSE_COLUMNS- Option in einer Spalte der Tabelle als Spaltensatz zusammengefasst wurden) kann bis zu 30000 Spalten mit maximal 1024 normalen (nicht mit geringer Dichte) + berechneten Spalten aufweisen.

Tabellen, die keinen Spaltensatz haben, sind wie bisher auf 1024 Spalten begrenzt.

Die Anzahl der Zeilen je Tabelle und die Gesamtgröße der Tabelle sind nur durch den verfügbaren Speicherplatz begrenzt. Eine Zeile (Datensatz) kann maximal 8.060 Byte enthalten. Diese Einschränkung ist für Tabellen mit varchar-, nvarchar-, varbinary- oder sql_variant- Spalten gelockert, die bewirken, dass die definierte Gesamtbreite der Tabelle 8.060 Bytes überschreitet. Für die Länge jeder dieser Spalten gilt weiterhin die Grenze von 8.000 Bytes. Ihre gemeinsame Breite darf jedoch die innerhalb einer Tabelle geltende Grenze von 8.060 Bytes überschreiten.

Jede Tabelle kann maximal **999** nicht gruppierte und **1** gruppierten Index enthalten. Dazu zählen auch die Indizes die durch PRIMARY KEY- und UNIQUE- Einschränkungen erzeugt werden.

Mit der CREATE TABLE- Anweisung können auch lokale und globale temporäre Tabellen erstellt werden.

Syntax:

```
create table {[db_name.] [schema.] | [schema.] tab_name}
(
  {spaltenname {basisdatentyp | ben_def_datentyp} [sparse] [nullregel]
  |berechnete_spalte
  | spaltensatz_name xml column_set for all_sparse_columns}
  [einschränkung [,...]]
  [, ...]
)
[on {filegroup | "default" | partition_scheme_name (partition_spalte)}]
[textureimage_on { filegroup | "default"}]
[filestream_on { filegroup | "default" | partition_scheme_name }]
[with (data_compression = { none | row | page }
      [ on partitions ({partition_number | <range>}[,...n]))]
```

Neu sind die Schalter PARTITION_SCHEME_NAME, TEXTIMAGE_ON und FILESTREAM_ON.

Mit der ersten Option ist es möglich eine Tabelle entsprechend eines Partitionsschemas auf verschiedene Dateigruppen zu verteilen.

Der zweite Schalter macht das Speichern von Spalten vom Datentyp **text, ntext, image, xml, varchar(max), nvarchar(max), varbinary(max)** und eines **CLR-benutzerdefinierten Datentyp** in einer angegebenen Dateigruppe möglich.

Der dritte Schalter gibt die Dateigruppe an, in der FILESTREAM- Daten gespeichert werden sollen. Besitzt die Datenbank nur eine Dateigruppe für FILESTREAM- Daten dann kann der Schalter weggelassen werden.

Wird der Eintrag für die Nullregel (NULL oder NOT NULL) weggelassen, übernimmt der SQL Server die ANSI NULL- Einstellungen der jeweiligen Datenbank oder die Einstellungen für die momentane Sitzung.

Das Standardverhalten des SQL Server bei nicht expliziter Angabe von Nullregeln ist "NOT NULL". Der ANSI- Standard besagt aber, dass NULL- Marken erlaubt werden wenn keine Angaben zur Nullregel gemacht werden. Darum wird bei der Arbeit mit dem SQL Server Management Studio, das wichtigste Werkzeug zur Übermittlung von SQL- Code an SQL Server, die Option "ANSI_NULL_DFLT_ON" automatisch aktiviert.

Sie können die Standardeinstellung Ihres Servers mit der Funktion **getansiNULL()** überprüfen. Gibt die Funktion eine 1 zurück, dann werden für neue Spalte NULL- Marken zugelassen wenn keine explizite Angabe zur Nullregel vorgenommen wurde.

Beispiel:

Die Tabelle "Lieferant" mit Basisdatentypen.

```
create table lieferant
(
    lnr char(3) not null,
    lname varchar(25) not null,
    status int null
    lstadt varchar(30) null
);
```

11.1.6.1 Sequenzobjekte

Eine Sequenz ist ein benutzerdefiniertes schemagebundenes Objekt, welches numerische Werte anhand von definierten Regeln erzeugt.

Diese Regeln können aufsteigende oder absteigende Sortierung, Intervalle und Zyklen beinhalten.

Eine Sequenz ist unabhängig von Tabellenobjekten, die Beziehung zwischen Tabellen und Sequenzen wird von der Anwendung gesteuert.

Syntax:

```
create sequence [schema_name .] sequence_name
[as [integer_datentyp | user_defined_integer_datentyp]]
[start with <wert> ]
[increment by <wert> ]
[{minvalue [ <wert> ] | no minvalue }]
[{maxvalue [ <wert> ] | no maxvalue }]
[{cycle | no cycle }]
[{ cache [ <wert> ] | no cache }];
```

Der ganzzahlige Datentyp kann tinyint, smallint, int, bigint oder decimal bzw. numeric mit einer Skalierung von 0 sein. Default ist bigint.

Sequenznummern werden außerhalb der aktuellen Transaktion generiert, und unabhängig davon genutzt, ob ein Commit oder ein Rollback für die Transaktion ausgeführt wird.

Mit der Angabe des CACHE- Arguments kann die Leistung des SQL Servers verbessert werden, weil die nächsten n Sequenznummern im Cache gehalten werden.

Beispiel:

Eine Sequenz die den Wertebereich von 0 bis 255 verwendet. Die Sequenz beginnt mit 10 und wird um 25 inkrementiert. Wenn sie den maximalen Wert von 250 erreicht hat wird sie bei 20 neu gestartet.

```
create sequence dbo.testseq as tinyint
start with 10
increment by 25
minvalue 20
maxvalue 250
cycle
cache 10;
```

Sequenz überprüfen. Es wird die Zahl 10 angezeigt.

```
select next value for dbo.testseq;
```

Sequenznummer in einer Insert Anweisung verwenden.

```
insert into dbo.testtab(lfdnr, namen, vname)
values(next value for dbo.testseq, 'John', 'Gunther');
```

In eine Tabelle eingefügte Sequenznummern können Lücken aufweisen, wen nein Rollback für eine Transaktion ausgeführt wird, wenn ein Sequenzobjekt von mehreren Tabellen gemeinsam verwendet wird oder wenn Sequenznummern abgerufen werden ohne das sie in Tabellen verwendet werden.

11.1.6.2 Die IDENTITY Eigenschaft

Mit Hilfe dieser Eigenschaft können vom System generierte sequentielle Werte für eine Spalte erstellt werden. Eine Identitätsspalte wird häufig für primäre Schlüsselwerte verwendet. Die Eigenschaft IDENTITY macht das erzeugen eindeutiger numerischer Werte einfach

Syntax:

```
IDENTITY[(seed, increment)]
```

Folgendes muss berücksichtigt werden:

- Es ist nur eine IDENTITY Spalte pro Tabelle zulässig.
- Diese Spalte muss vom Datentyp INTEGER (INT, BIGINT, SMALLINT oder TINYINT), NUMERIC/DECIMAL sein. Die Datentypen NUMERIC und DECIMAL müssen mit 0 Dezimalstellen angegeben werden.
- Diese Spalte kann nicht aktualisiert werden.
- NULL Werte sind für diese Spalte nicht zulässig.

Hinweis: **IDENTITY erzwingt keine Eindeutigkeit. Eindeutigkeit kann erzwungen werden, indem ein eindeutiger Index auf die Spalte erstellt wird.**

Jede Tabelle darf nur eine Spalte mit dieser Eigenschaft besitzen. Der Ersteller der Tabelle kann die Startnummer (SEED) und den Betrag festlegen, um den dieser Wert steigt oder fällt (INCREMENT). Standardwert für SEED und INCREMENT ist 1.

11.1.6.2.1 Abfragen über die IDENTITY- Spalte

Das IDENTITYCOL- Schlüsselwort kann anstelle des Spaltennamens in einer Abfrage verwendet werden. Das heißt, ich muss den Namen der IDENTITY- Spalte nicht kennen um in einer Abfrage auf sie zu verweisen.

Beispiel:

Angenommen, die Spalte "vnum" der Tabelle "Verkäufer" besitzt die Eigenschaft IDENTITY.

```
select *
from verkäufer
where identitycol = 1053;
```

11.1.6.2.2 Anfangswert und Schrittfolge bestimmen

Mit zwei Systemfunktionen werden der Anfangswert und die Schrittfolge einer IDENTITY-Spalte ermittelt.

Syntax:

IDENT_SEED('tab_name' | 'sicht_name')

Ermittelt den Anfangswert der IDENTITY-Spalte.

IDENT_INCR('tab_name' | 'sicht_name')

Ermittelt die Schrittweite der IDENTITY-Spalte.

Beispiel:

```
select ident_seed('verkaeufer');
```

```
select ident_incr('verkaeufer');
```

11.1.6.2.3 Letzten erzeugten IDENTITY- Wert ermitteln

Es ist nach der Ausführung von INSERT möglich den letzten erzeugten IDENTITY- Wert abzufragen. Verwenden Sie die Funktionen **ident_current()** oder **@@identity**.

Beispiel:

```
select @@identity;
```

11.1.6.2.4 Deaktivieren und Aktivieren der Eigenschaft IDENTITY

Durch DML Anweisungen auf eine Tabelle entstehen in der IDENTITY-Spalte Lücken in den fortlaufenden Werten, die geschlossen werden sollen. Außerdem kann es vorkommen, dass beim Laden großer Datenmengen Datensätze mit bereits vorhandenen IDENTITY-Werten vorkommen. Um die Aufgaben zu erfüllen ist es notwendig die Eigenschaft zu deaktivieren und danach wieder zu aktivieren.

Deaktivieren:

```
set identity_insert tablename on
```

Aktivieren:

```
set identity_insert tablename off
```

Werden in einer Tabelle mit einer Identitätsspalte häufig Daten gelöscht, können Lücken zwischen einzelnen Identitätswerten auftreten. Falls dies ein Problem darstellt, sollten Sie die IDENTITY-Eigenschaft nicht verwenden. Um sicherzustellen, dass keine Lücken erstellt wurden, oder um eine bereits vorhandene Lücke zu füllen, sollten Sie die vorhandenen Identitätswerte prüfen, bevor Sie mit SET IDENTITY_INSERT ON explizit einen Wert eingeben.

Mit DBCC CHECKIDENT können Sie den aktuellen Identitätswert überprüfen und mit dem Maximalwert in der Identitätsspalte vergleichen.

11.1.6.3 Der Datentyp "UNIQUEIDENTIFIER"

Der Datentyp UNIQUEIDENTIFIER wird auch "Global Eindeutiger Bezeichner" (GLOBAL UNIQUE IDENTIFIER – GUID) genannt. Dabei handelt es sich um einen Wert von 128 Bit, der auf eine Art erzeugt wird, die für alle praktischen Zwecke weltweite Eindeutigkeit garantiert, selbst auf nicht vernetzten Rechnern. Es wird zur wichtigsten Methode, um Daten, Objekte, Softwareanwendungen und Applets in verteilten Systemen zu kennzeichnen.

Der Datentyp wird mit der Funktion "newid()" zusammen eingesetzt. Die Funktion erzeugt eine GUID. Diese Funktion kann als DEFAULT- Constraint für die Spalte verwendet werden oder bei der INSERT- Anweisung explizit angegeben werden.

Beispiel:

```
create table kunden
(
kdn_nr uniqueidentifier not null default newid(),
...
);
```

11.1.6.4 Tabellen mit berechneten Spalten

Eine berechnete Spalte ist eine virtuelle Spalte in einer Tabellendefinition, die nicht physisch in der Tabelle gespeichert ist. Die benannte Spalte enthält eine Formeldefinition die sich auf andere Spalten derselben Tabelle bezieht.

Beispiel:

```
create table verkauf
(
vnr int identity(1,1) not null,
artnr int not null,
netto_preis money not null,
brutto_preis as netto_preis * 1.16,
vermenge integer not null,
ges_preis as netto_preis * 1.16 * vermenge
);

go

insert into verkauf values(1001, 32.95, 200);
```

11.1.6.5 Tabelle mit einer FILESTREAM- Spalte

Die FILESTREAM- Spalte muss eine vom Datentyp VARBINARY(MAX) sein. Die Tabelle muss auch eine Spalte mit dem UNIQUEIDENTIFIER- Datentyp aufweisen, der das ROWGUIDCOL- Attribut enthält. Diese Spalte darf keine NULL- Werte zulassen und muss eine UNIQUE- oder eine PRIMARY KEY- Einschränkung für einzelne Spalten enthalten. Der GUID- Wert für die Spalte muss entweder durch eine Anwendung bereitgestellt werden, wenn Daten eingefügt werden, oder durch eine DEFAULT- Einschränkung, die die NEWID ()- Funktion verwendet.

Die Spalte mit dem ROWGUIDCOL-Attribut kann nicht gelöscht werden, und die zugehörigen Einschränkungen können nicht geändert werden, wenn für die Tabelle eine FILESTREAM- Spalte definiert ist. Die ROWGUIDCOL-Spalte kann nur gelöscht werden, nachdem die letzte FILESTREAM- Spalte gelöscht wurde.

Wenn das FILESTREAM- Speicherattribut für eine Spalte angegeben wird, werden alle Werte dieser Spalte in einem FILESTREAM- Datencontainer des Dateisystems gespeichert.

Dieser Container ist der Datenbank bekannt, weil sie eine FILESTREAM- Dateigruppe besitzt.

Beispiel:

```

use standard_filestream;
go

create table dbo.lieferant
(
    Inr char(3) not null primary key,
    Iname nvarchar(200) not null,
    status int null,
    Istadt nvarchar(200),
    photo varbinary(max) filestream null,
    lebenslauf varbinary(max) filestream null,
    meinerowguid uniqueidentifier not null rowguidcol unique default newid()
)on daten_db filestream_on DokumenteFilestream;

go

insert into dbo.lieferant values('L01','Heinrichs',10,'Erfurt',
    (select cast(x as varbinary(max))
    from openrowset(bulk 'c:\bild\coloneljoneil.jpg',single_blob) as temp(x)),
    (select cast(x as varbinary(max))
    from openrowset(bulk 'c:\lebenslauf\heinrichs_lbl.docx',single_blob) as temp(x)),
    default);

```

11.1.6.6 Tabellen mit dem Datentyp HIERARCHYID

Der Datentyp wird vom System bereitgestellt. Mit dem Datentyp können Sie Tabellen mit einer hierarchischen Struktur erstellen oder auf hierarchisch geordnete Daten an einem anderen Speicherort verweisen

Hierarchische Daten sind definiert als Satz von Datenelementen, die durch hierarchische Beziehungen miteinander verbunden sind. Hierarchische Beziehungen liegen vor, wenn ein Datenelement einem anderen Element übergeordnet ist. Hierarchische Daten kommen in Datenbanken häufig vor:

- Eine Organisationsstruktur
- Ein Dateisystem
- eine Gruppe von Aufgaben in einem Projekt.

Der HIERARCHYID vereinfacht die Speicherung und Abfrage hierarchischer Daten. Er dient für die Darstellung von Baumstrukturen, dem verbreitetesten Typ hierarchischer Daten.

Eigenschaften:

Ein Wert des Datentyps stellt eine Position in einer Strukturhierarchie dar.

- Äußerst komprimiert.
Die durchschnittliche Zahl der Bits, die erforderlich sind, um einen Knoten in einer Baumstruktur mit n Knoten darzustellen, hängt von der durchschnittlichen Anzahl der Verzweigungen (der durchschnittlichen Anzahl untergeordneter Knoten) eines Knotens ab.
- Vergleiche erfolgen in Tiefensuchreihenfolge.
- Unterstützung willkürlicher Einfüge- und Löschoperationen.
Es ist immer möglich, rechts oder links von einem gegebenen Knoten einen gleichgeordneten Knoten zu erstellen oder ihn auch zwischen zwei Knoten einzufügen. Die Vergleichseigenschaft bleibt gewahrt, auch wenn eine beliebige Anzahl von Knoten in die Hierarchie eingefügt oder aus ihr gelöscht wird.

Einschränkungen:

- Eine Spalte des Typs HIERARCHYID stellt nicht automatisch eine Baumstruktur dar. Es ist Aufgabe der Anwendung, HIERARCHYID- Werte so zu erstellen und zuzuweisen, dass die gewünschte Beziehung anhand der Werte zu erkennen ist. Einige Anwendungen möchten nicht einmal, dass eine Spalte des Typs HIERARCHYID eine Baumstruktur repräsentiert. Vielleicht sind die Werte Verweise auf Positionen in einer Hierarchie, die in einer anderen Tabelle definiert ist.
- Es ist Aufgabe der Anwendung, die Parallelität zu verwalten, indem sie geeignete HIERARCHYID- Werte generiert und zuweist. Es gibt keine Garantie dafür, dass die HIERARCHYID- Werte einer Spalte eindeutig sind, sofern die Anwendung keine Einschränkung für einen eindeutigen Schlüssel verwendet oder selbst dafür sorgt, dass eindeutige Werte erzeugt werden.
- Hierarchische, durch HIERARCHYID- Werte dargestellte Beziehungen werden nicht wie Fremdschlüsselbeziehungen durchgesetzt. Es ist möglich und manchmal auch angemessen, eine hierarchische Beziehung herzustellen, in der das Element B dem Element A untergeordnet ist, um dann A zu löschen, wodurch B mit einer Beziehung zu einem nicht mehr vorhandenen Datensatz verbleibt. Wenn dieses Verhalten unannehmbar ist, muss die Anwendung vor dem Löschen von Elementen prüfen, ob untergeordnete Elemente vorhanden sind.

Beispiel:

```
use standard;
go

create table dbo.mitarbeiter
(
    mnr int not null primary key,
    position hierarchyid not null,
    mname nvarchar(100)
);
go

insert into dbo.mitarbeiter values
    (1000, '/', 'Wiesenhupfer'),
    (1000, '/1/', 'Mithupfer'),
    (1000, '/2/', 'Löffler'),
    (1000, '/1/1/', 'Jemand'),
    (1000, '/1/2/', 'Wasserträger'),
    (1000, '/2/1/', 'Schönschreiber');

go

select mnr, position.getlevel() + 1, mname from dbo.mitarbeiter;
```

Es werden durch den Datentyp folgende Methoden unterstützt:

| | |
|-------------------------|--|
| GetAncestor(): | Ermittelt verschiedene Vorfahren, also die darüber liegende Ebene. |
| GetDescendant(): | ermittelt je nach Syntax ein Kind, einen Nachbarn der aktuellen Ebene, also der darunter liegenden Ebene. |
| GetLevel(): | gibt die Nummern der Ebene beginnend mit 0 aus. |
| GetRoot(): | Gibt immer die am höchsten stehende Ebene aus. Da es einen statischen Methode ist, wird sie am Type aufgerufen (HIERARCHYID::GetRoot()). |
| IsDescendant(): | Ermittelt, ob jemand ein Nachfolger der übergebenen ID ist. |
| Parse(): | Konvertiert den übergebenen String in eine HIERARCHYID (set @manager = HIERARCHYID::Parse('/')). |

| | |
|--------------------------|--|
| Reparent(): | Hier kann ein potentieller bisheriger und möglicher neuer übergeordneter Knoten angegeben werden, diese Funktion ermöglicht also das Umhängen von Strukturen in einem hierarchischen Baum. |
| ToString(): | Konvertiert die HIERARCHYID in die ursprüngliche Zeichenfolge. |
| Read() Write(): | Konvertiert eine HIERARCHYID in oder aus einem varbinary- Wert. |

11.1.6.7 Tabellen mit Geodatentypen

Es gibt seit SQL Server 2008 zwei neue Datentypen, die Geodatentypen GEOMETRY und GEOGRAPHY. Beide Datentypen sind als .NET Datentypen implementiert.

| | |
|-------------------|--|
| GEOGRAPHY: | Speichert ellipsoide Erddaten, wie zum Beispiel GPS Latitude und Longitude Koordinaten (Breiten- und Längengrade). |
| GEOMETRY: | Unterstützt sogenannte simple Features der SQL Spezifikation Version 1.10, des OPEN Geospatial Consortium (OGC). Der planare Typ GEOMETRY ist für die Darstellung räumliche Daten bestimmt. Dieser Typ stellt Daten in einem euklidischen (flachen) Koordinatensystem dar. |

Beispiel:

Die folgenden zwei Beispiele zeigen, wie Geometriedaten hinzugefügt und abgefragt werden. Im ersten Beispiel wird eine Tabelle mit einer Identitätsspalte und der GEOMETRY- Spalte "Geosp1" erstellt. Eine dritte Spalte rendert die GEOMETRY- Spalte als Darstellung im Open Geospatial Consortium (OGC) WKT-Format und verwendet die STAsText()- Methode. Dann werden zwei Zeilen eingefügt: eine enthält eine LINESTRING- Instanz des Typs GEOMETRY und die andere eine POLYGON- Instanz.

```
use standard;
go
create table raumdaten
(
    id int identity (1,1),
    geosp1 geometry,
    geosp12 as geosp1.STAsText());
go
insert into raumdaten (geosp1)
values (geometry::STGeomFromText('LINESTRING (100 100, 20 180, 180 180)', 0));
insert into raumdaten (geosp1)
values (geometry::STGeomFromText('POLYGON ((0 0, 150 0, 150 150, 0 150, 0 0))',
0));
go
```

Im nachfolgenden Beispiel werden mithilfe der **STIntersection()**- Methode die Punkte zurückgegeben, an denen die beiden zuvor eingegebenen GEOMETRY- Instanzen sich schneiden.

```
declare @geo1 geometry;
declare @geo2 geometry;
declare @erg geometry;

select @geo1 = geosp1 from raumdaten where id = 1;
select @geo2 = geosp1 from raumdaten where id = 2;
select @erg = @geo1.STIntersection(@geo2);
select @erg.STAsText();
```

11.1.6.8 FileTable

Die Funktionalität von FileTable basiert auf der SQL Server- FILESTREAM- Technologie. Sie bietet für nichtstrukturierte Datendateien (Texte, Bilder, Vidiostreams,...) Unterstützung für den Windows- Dateamespace und die Kompatibilität mit Windows- Anwendungen.

Volltextsuche und semantische Suche wird für die nichtstrukturierten Daten und deren Metadaten bereitgestellt.

Ziel:

- Windows- API Kompatibilität für Datendateien die in einer Datenbank gespeichert sind. Das umfasst:
 - Direktes Update an den FILESTREAM- Daten und nicht transaktionaler Streamingzugriff.
 - Hierarchischer Namensbereich von Verzeichnissen und Dateien.
 - Speichern von Metadaten.
 - Unterstützung von APIs für die Windows- Datei und Verzeichnisverwaltung.
- Kompatibel mit anderen SQL Server Funktionen, Tools und Diensten.

FileTables sind spezielle Benutzertabellen mit einem vordefinierten Schema und Aufbau. In ihr werden eine Hierarchie von Dateien und Verzeichnissen beginnend von einem Stammverzeichnis gespeichert. Jeder Datensatz dieser Tabelle stellt eine Datei oder ein Verzeichnis dar, mit den entsprechenden Pfadinformationen, Dateiattributen, Dokumenttyp für die Volltextsuche und einem eindeutigen Bezeichner.

An die Tabelle können Trigger gebunden werden um Referentielle- und Geschäftsregeln sicherzustellen.

Wird die Datenbank für den nicht transaktionalen Zugriff konfiguriert, wird die in der Tabelle dargestellte Datei- und Verzeichnisstruktur in der für die Instanz konfigurierten FILESTREAM- Freigabe verfügbar gemacht, und der Dateisystemzugriff für Windows- Anwendungen bereitgestellt. Das Erstellen, Ändern oder Löschen einer Datei im Dateisystem wird durch die FileTable abgefangen und im Inhalt wiedergespiegelt.

Daten in FileTable können aber auch durch Transact SQL abgefragt und aktualisiert werden.

In einer Datenbank mit FileTables kann auch weiterhin die FILESTREAM- Funktionalität genutzt werden.

Vorbereitung der Instanz und der Datenbank für FileTable:

1. Aktivieren von FILESTREAM auf Instanzebene.

```
exec sp_configure 'filestream_access_level', 2;
```
2. Firewall für FILESTREAM konfigurieren.
3. Bereitstellen einer FILESTREAM- Dateigruppe in der Datenbank.
4. Aktivieren des transaktionalen Zugriffs auf Datenbankebene.

```
alter database standard
set filestream(non_transacted_access = full, directory_name = 'Werbung');
```
5. FileTable erstellen.

Syntax:

```
create table <name> as filetable
with (
    filetable_directory = 'directory_name',
    filetable_collate_filename = {<collations> | database_default}
);
```

Beispiel:

```
create table webung_video as filetable
with(filetable_directory = 'Werbung', filetable_collate_filename = database_default);
```

11.1.6.9 Partitionierte Tabellen

Partitionierte Tabellen erweisen sich als günstig bei Datenbankgrößen von einigen hundert Megabyte oder ein paar Terabyte. Werden Datensätze über mehrere Jahre hinweg aufgehoben, dann können sehr große Tabellen entstehen.

Um den schnellen und permanenten Zugriff auf die Daten zu gewährleisten oder um die große Menge an Daten in eine sinnvolle Backup-Strategie einzubinden ist das partitionieren dieser Tabellen- und Indexdaten eine gute Methode.

Unter Partitionieren versteht man das Aufteilen der Datensätze in verschiedene Datenbereiche anhand eines Partitionsschlüssels. Effektiv ist es, die Partitionen auf verschiedene Dateigruppen auf unterschiedlichen physischen Datenträgern zu legen.

Tabellen können ausschließlich in der Enterprise- oder Developer-Edition des SQL Servers partitioniert werden.

Folgende Datenobjekte werden benötigt um eine Tabelle zu partitionieren:**- Partitionierungsfunktion:**

Über diese Funktion werden die Wertebereiche für die Partitionen der Daten vorgegeben. Sie wird zunächst vollkommen unabhängig von einer Tabelle definiert. Sie liefert eine Ordnungsnummer zurück über die später eine bestimmte Partition angesprochen werden kann. Für diese Funktion kommen nur Spalten mit den Datentypen in Frage, die auch normal indiziert werden können.

- Dateigruppen:

Die Partitionen einer Tabelle können nur einer Dateigruppe (auch PRIMARY) zugeordnet werden. Das ist in der Regel wenig sinnvoll. Wenn die Daten physisch verteilt werden sollen benötigt man eine Datenbank mit mehreren Dateigruppen.

- Partitionsschema:

Kombiniert man eine Partitionsfunktion mit der notwendigen Anzahl von Dateigruppen, erhält man ein Partitionsschema. Dadurch verteilt SQL Server bei INSERT die Daten (Index) automatisch in die darunterliegenden Dateigruppen.

- Partitionierungsschlüssel:

- Das Partitionieren der Datensätze wird immer anhand der Werte einer bestimmten Spalte durchgeführt. Die bezeichnet man als Partitionierungsschlüssel. Dabei müssen die Werte nicht eindeutig sein.

11.1.6.9.1 Partitionierungsfunktion erzeugen

Erstellen einer Funktion in der aktuellen Datenbank, die auf der Basis der Werte einer angegebenen Spalte die in einer Tabelle oder einem Index enthaltenen Zeilen Partitionen zuordnet. Das Verwenden von CREATE PARTITION FUNCTION ist der erste Schritt beim Erstellen einer partitionierten Tabelle oder eines partitionierten Indexes.

Syntax:

```
create partition function partition_function_name (input_parameter_type)
as range [left | right]
for values ([ grenzwert [,...n ]])
```

11.1.6.9.2 Partitionsschema erstellen

Erstellt ein Schema in der aktuellen Datenbank, das die Partitionen einer partitionierten Tabelle oder eines partitionierten Indexes Dateigruppen zuordnet. Die Anzahl und die Domäne der Partitionen einer partitionierten Tabelle oder eines partitionierten Indexes werden in einer Partitionsfunktion bestimmt. Eine Partitionsfunktion muss zunächst in einer CREATE PARTITION FUNCTION- Anweisung erstellt werden, bevor ein Partitionsschema erstellt wird.

Syntax:

```
create partition scheme partition_scheme_name
as partition partition_function_name
[all] to ({file_group_name | [ primary]} [,...n ])
```

Beispiel:

```
create database standard_part
on primary
    (name = N'standard_sk',
     filename = N'e:\standard_sk.mdf'),
filegroup passiv
    (name = N'standard_dat1',
     filename = N'f:\standard_dat1.ndf'),
filegroup lief_vor_2000
    name = N'standard_lief_vor_2000_dat',
    filename = N'g:\standard_lief_vor_2000dat.ndf'),
filegroup lief_2000
    (name = N'standard_lief_2000_dat',
     filename = N'h:\standard_lief_2000dat.ndf'),
filegroup lief_2001
    (name = N'standard_lief_2001_dat',
     filename = N'i:\standard_lief_2001dat.ndf')
log on
    (name = N'standard_log',
     filename = N'j:\standardlog.ldf');
go

create partition function standardliefefunktion(datetime)
as range left for values ('01.01.2000', '01.01.2001');
go

create partition scheme standardliefpartition
as partition standardliefefunktion
to (lief_vor_2000, lief_2000, lief_2001);
go

create table dbo.lieferung
(
    lnr nchar(3) not null,
    anr nchar(3) not null,
    lmenge int not null,
    lddatum datetime not null,
    constraint lief_ps primary key nonclustered (lnr, anr, lddatum)
) on standardliefpartition(ddatum);
go
```

11.1.6.10 Tabellen mit Spalten geringer Dichte

SQL Server verwendet das Schlüsselwort SPARSE in einer Spaltendefinition, um die Speicherung von Werten in dieser Spalte zu optimieren. Wenn der Spaltenwert in einer Zeile der Tabelle NULL ist, belegen die Werte keinen Speicherplatz.

Katalogsichten für eine Tabelle, die Spalten mit geringer Dichte aufweist, entsprechen denen einer typischen Tabelle. Diese **sys.columns**- Katalogsicht enthält eine Zeile für jede Spalte in der Tabelle und einen Spaltensatz, wenn einer definiert wurde.

Spalten mit geringer Dichte benötigen mehr Speicherplatz für Werte, die ungleich NULL sind, als für identische Daten benötigt wird, die nicht als SPARSE gekennzeichnet wurden.

Beispiel:

```
use standard;
go

create table dbo.lieferant_gd
(
    Inr char(3) not null constraint Inr_ps primary key,
    Iname nvarchar(200) not null,
    status int null,
    Istadt nvarchar(200) null,
    tel_privat nvarchar(20) sparse null,
    email nvarchar(200) sparse null
);
go
```

Spalten mit den Datentypen text, timestamp, geography, geometry, image, ntext und UDDT können nicht als SPARSE festgelegt werden.

11.1.6.11 Tabellen mit Spaltensätzen

Für Tabellen, die Spalten mit geringer Dichte aufweisen, können Sie einen Spaltensatz festlegen, der alle Spalten in der Tabelle mit geringer Dichte zurückgibt.

Bei einem Spaltensatz handelt es sich um eine nicht typisierte XML- Darstellung, die alle Tabellenspalten mit geringer Dichte in einer strukturierten Ausgabe kombiniert. Wie auch berechnete Spalten werden Spaltensätze nicht physisch in der Tabelle gespeichert.

Der Unterschied zwischen einem Spaltensatz und einer berechneten Spalte besteht darin, dass der Spaltensatz direkt aktualisiert werden kann.

Verwenden Sie Spaltensätze, wenn die Tabelle eine große Anzahl an Spalten enthält und es sehr aufwändig ist, jede Spalte einzeln zu verarbeiten. Außerdem verbessert sich die Anwendungsleistung, wenn zum Auswählen und Einfügen von Daten in diese Tabellen Spaltensätze verwendet werden. Die Leistung von Spaltensätzen kann jedoch beeinträchtigt werden, wenn für die Spalten in der Tabelle sehr viele Indizes definiert werden.

Richtlinien:

- Spalten mit geringer Dichte und ein Spaltensatz können als Teil der gleichen Anweisung hinzugefügt werden.
- Der Spaltensatz kann nicht geändert werden. Soll ein Spaltensatz geändert werden, muss er erst gelöscht werden um danach einen neuen Spaltensatz zu erstellen. Sie können einer Tabelle, die bereits Spalten mit geringer Dichte enthält, keinen Spaltensatz hinzufügen.

- Sie können einen Spaltensatz Tabellen hinzufügen, die keine Spalten mit geringer Dichte enthalten. Wenn Sie der Tabelle zu einem späteren Zeitpunkt Spalten mit geringer Dichte hinzufügen, werden diese im Spaltensatz angezeigt.
- Es ist nur ein Spaltensatz pro Tabelle zulässig.
- Für einen Spaltensatz können keine Einschränkungen oder Standardwerte definiert werden.
- Berechnete Spalten können keine Spaltensatz-Spalten enthalten.
- Verteilte Abfragen werden von Tabellen mit Spaltensätzen nicht unterstützt.
- Die Replikation wird nicht unterstützt.
- Ein Spaltensatz darf nicht Teil irgendeiner Art von Index sein. Dies gilt auch für XML-Indizes, Volltextindizes und indizierte Sichten. Ein Spaltensatz kann einem Index nicht als eingeschlossene Spalte hinzugefügt werden.
- Ein in einer Sicht enthaltener Spaltensatz wird als XML-Spalte angezeigt.
- Ein Spaltensatz darf nicht in einer indizierten Sichtdefinition enthalten sein.
- Partitionierte Sichten, die Tabellen mit Spaltensätzen enthalten, können aktualisiert werden, wenn die Spalten mit geringer Dichte in der partitionierten Sicht mit Namen angegeben sind. Eine partitionierte Sicht kann nicht aktualisiert werden, wenn sie lediglich einen Verweis auf den Spaltensatz enthält.
- Für XML-Daten gilt eine Größenbeschränkung von 2 GB. Wenn die gesamte Datengröße aller Spalten mit geringer Dichte ungleich NULL in einer Zeile diesen Wert überschreitet, wird für die Abfrage bzw. den DML-Vorgang ein Fehler ausgegeben.

Beispiel:

```

use standard;
go

create table dbo.lieferant_sps
(
    lnr char(3) not null constraint sps_lnr_ps primary key,
    lname nvarchar(200) not null,
    status int null,
    lstadt nvarchar(200) null,
    tel_privat nvarchar(20) sparse null,
    email nvarchar(200) sparse null,
    spezialinfo xml column_set for all_sparse_columns
);
go

insert into dbo.lieferant_sps(lnr,lname,status,lstadt,tel_privat,email)
values('L01','Schulze', 10, 'Erfurt',null,null);

insert into dbo.lieferant_sps (lnr,lname,status,lstadt,tel_privat,email)
values('L02','Krause', 10, 'Erfurt','+4936133445566',null);

insert into dbo.lieferant_sps (lnr,lname,status,lstadt,spezialinfo)
values('L03','Maria', 10, 'Erfurt','<tel_privat>+49361324354</tel_privat>');

insert into dbo.lieferant_sps (lnr,lname,spezialinfo)
values('L04','Karla',
      '<tel_privat>+4936199080877</tel_privat><email>s_rost@web.de</email>');

go

```

Überlegungen zur Sicherheit

Das Sicherheitsmodell für Spaltensätze funktioniert ähnlich wie das Sicherheitsmodell von Tabellen und Spalten. Bei der Beziehung zwischen Spaltensatz und Spalten mit geringer Dichte handelt es sich jedoch nicht um einen Container, sondern um eine Gruppenbeziehung. Das Sicherheitsmodell prüft die Sicherheit der Spaltensatz-Spalte, wobei die DENY-Anweisung für die zugrunde liegenden Spalten mit geringer Dichte berücksichtigt werden. Für das Sicherheitsmodell gelten die folgenden Eigenschaften:

- Wie bei anderen Spalten in der Tabelle können Sicherheitsberechtigungen für den Spaltensatz erteilt und aufgehoben werden.
- Der GRANT- bzw. REVOKE-Befehl für SELECT-, INSERT-, UPDATE-, DELETE- und REFERENCES- Berechtigungen für einen Spaltensatz wirkt sich nicht auf die zugrunde liegenden Spalten aus. Dieser Vorgang gilt nur für die Verwendung der Spaltensatz-Spalte. - Die DENY- Berechtigung für einen Spaltensatz gilt jedoch auch für die zugrunde liegenden Spalten mit geringer Dichte.
- Zur Ausführung von SELECT-, INSERT-, UPDATE- und DELETE- Anweisungen für den Spaltensatz muss der Benutzer über die entsprechenden Berechtigungen für den Spaltensatz und über die entsprechenden Berechtigungen für alle Spalten mit geringer Dichte in der Tabelle verfügen.
- Durch die Ausführung einer REVOKE- Anweisung für eine Spalte mit geringer Dichte bzw. einen Spaltensatz wird die Sicherheitseinstellung auf die Einstellung des übergeordneten Objekts zurückgesetzt.

11.1.7 Basistabellen löschen

Durch das Löschen einer Tabelle, werden die Tabellendefinition und alle Daten sowie die Berechtigungen für die Tabelle entfernt.

Bevor die Tabelle gelöscht wurde, müssen Sie alle Abhängigkeiten der Tabelle und anderen Objekten entfernen. Die Abhängigkeiten kann man sich mit der Systemprozedur "sp_depends" anzeigen lassen.

Syntax:

```
drop table tablename [, ...]
```

11.1.8 Basistabellen ändern

SQL Server ermöglicht es, bestehende Tabellen zu ändern. Dazu nutzt man die Anweisung "ALTER TABLE".

Mit Hilfe dieses Befehls können folgende Änderungen an einer bestehenden Tabelle vorgenommen werden:

- Den Datentyp oder die Nullregel einer einzelnen Spalte ändern.
- Eine oder mehrere Spalten hinzufügen (mit oder ohne Definition von Einschränkungen).
- Eine oder mehrere Einschränkungen hinzufügen.
- Eine oder mehrere Einschränkungen löschen.
- Eine oder mehrere Spalten löschen.
- Eine oder mehrere Einschränkungen aktivieren oder deaktivieren.
- Einen oder mehrere Trigger aktivieren oder deaktivieren.

11.1.8.1 Datentyp ändern

Mit der Klausel "ALTER COLUMNS" des Befehls "ALTER TABLE" lassen sich Datentyp und Nullregel einer bestehenden Spalte ändern.

Einschränkungen:

- Spalten mit dem Datentyp TEXT, IMAGE, NTEXT oder TIMESTAMP können nicht geändert werden.
- Es kann keine berechnete Spalte oder replizierte Spalte geändert werden.
- Für die zu ändernde Spalte darf keine PRIMARY KEY- oder FOREIGN KEY-Einschränkung definiert sein.
- Die zu ändernde Spalte darf nicht in einer berechneten Spalte referenziert werden.
- Ist die zu ändernde Spalte indiziert, dann sind nur das Verlängern bei einer Spalte variabler Länge und das Ändern der Nullregel oder beides erlaubt.
- Ist für die Spalte eine UNIQUE- oder CHECK- Einschränkung definiert, ist nur das Ändern der Länge bei einer Spalte variabler Länge erlaubt. Bei einer UNIQUE- Einschränkung muss die neue Länge größer sein als die alte.

Beispiel:

Ändern der Spalte Iname (char(20)) der Tabelle Lieferant in der Länge.

```
alter table lieferant
alter column Iname varchar(100) not null;
```

11.1.8.2 Neue Spalte hinzufügen

Man kann eine neue Spalte zu einer Tabelle mit oder ohne gleichzeitige Definition von Einschränkungen hinzufügen. Es kann pro Anweisung mehr als eine Spalte hinzugefügt werden.

Hinweis: Wenn die Tabelle bereits Datensätze enthält, muss für die neue Spalte entweder NULL erlaubt werden oder eine DEFAULT- Einschränkung definiert werden.

Beispiel:

Hinzufügen einer Spalte "plz" zur Tabelle Lieferant.

```
alter table lieferant
add plz char(5) null;
```

11.1.8.3 Eine Spalte löschen

Mit "ALTER TABLE" kann eine Spalte oder mehrere Spalten aus einer Tabelle entfernt werden.

Folgende Spalten können nicht gelöscht werden:

- Eine replizierte Spalte.
- Spalten, die in einen Index verwendet werden.
- Spalten, die in einer CHECK-, FOREIGN-, UNIQUE- oder PRIMARY KEY- Einschränkung benutzt werden.
- Spalten, die mit einem definierten Standardwert (DEFAULT- Einschränkung) verknüpft sind oder die an ein Standardobjekt (DEFAULT) gebunden sind.
- eine Spalte, die an eine RULE gebunden ist.

Beispiel:

Entfernen der Spalte "plz" von der Tabelle Lieferant.

```
alter table lieferant
drop column plz;
```

11.2 Implementieren der Datenintegrität

Die Sicherstellung der Datenintegrität ist ein wichtiger Schritt bei der Planung einer Datenbank. Einschränkungen bieten eine leistungsfähige und dennoch einfache Möglichkeit Daten-Integrität zu erzwingen.

Folgende drei Datenintegritätstypen sind bekannt:

- | | |
|---------------------------|--|
| Domänenintegrität: | Auch Spaltenintegrität genannt, gibt eine Menge von Datenwerten an, die für eine Spalte gültig sind, und legt fest ob NULL- Marken zulässig sind. |
| Entitätsintegrität: | Auch Tabellenintegrität genannt. Hierbei benötigen alle Datensätze einen eindeutigen Bezeichner (PRIMARY KEY). Ob der Primärschlüssel geändert oder der ganze Datensatz gelöscht werden kann, hängt von den Operationsregeln des Fremdschlüssels ab. |
| Referenzielle Integrität: | Diese Integrität erzwingt die Beziehung zwischen zwei Tabellen, in der Regel über den Primärschlüssel und den Fremdschlüssel. Es wird sowohl die referenzielle Integrität als auch die Operationsregeln für Fremdschlüssel durchgesetzt. |

Optionen zum Durchsetzen der Datenintegrität

SQL Server stellt verschiedene Mechanismen zum Durchsetzen der Datenintegrität bereit.

| Mechanismus | Integritätstyp | Beschreibung |
|---------------------------------|-------------------------------|--|
| Datentypen | Domäne | Grundlegende Einschränkung für die Typen von Daten für eine Spalte |
| Regeln und Standardwerte | Domäne, Entität | Regeln definieren zulässige Werte die für eine Spalte zulässig sind und Standardwerte legen fest welcher Wert in eine Spalte eingetragen wird wenn er nicht explizit bei einer INSERT- Anweisung angegeben wird. Ein Nachteil besteht darin dass sie nicht ANSI-kompatibel sind. Daher sollten besser Einschränkungen verwendet werden. |
| Einschränkungen | Domäne, Entität, Referentiell | Einschränkungen definieren auf welche Weise das Datenbankmodul automatisch Integrität in einer Datenbank erzwingt. Sie grenzen die in Spalten zulässigen Werte ein. Sie stellen den Standardmechanismus zum Erzwingen der Datenintegrität dar. Sie sollten allen anderen Mechanismen vorgezogen werden. |
| Trigger | Domäne, Entität, Referentiell | Trigger werden ausgeführt wenn ein DML- Ereignis im Datenbankserver auftritt. Sie werden zur Durchsetzung von Geschäftsregeln verwendet. |
| XML- Schemas | Domäne (XML) | Sie definieren den Namespace, die Struktur sowie den zulässigen Inhalt von XML- Dokumenten. Sie werden auf Spalten angewendet die mit dem Datentyp XML definiert sind. |

Es gibt zwei Methoden um Datenintegrität zu erzwingen:

- Deklarative Integrität: Bei der Deklaration von Tabellen wird die Datenintegrität definiert. Sie ist damit Teil der Datenbankdefinition, indem Einschränkungen direkt in Tabellen oder Spalten definiert werden bzw. mit Hilfe von Standards und Regeln.
- Prozedurale Integrität: Wird sichergestellt durch Scripts. Sie wird verwendet für komplizierte Geschäftslogik und Ausnahmen. Dazu werden in der Regel Trigger und gespeicherte Prozeduren verwendet oder andere Programmiersprachen.

11.2.1 Einschränkungen

Einschränkungen (Constraints) sind eine auf den ANSI- Standard beruhende Methode zum Erzwingen der Datenintegrität in Tabellen einer Datenbank. Einschränkungen stellen sicher, dass gültige Datenwerte in Spalten eingegeben und die Beziehungen zwischen Tabellen beibehalten werden.

Sie sollten stets versuchen die Datenintegrität erst mit Constraints sicherzustellen bevor Sie ander Datenbankobjekte in Betracht ziehen.

Jeder Datenintegritätstyp wird mit ganz bestimmten Einschränkungstypen erzwungen. Nachfolgende Tabelle zeigt eine Übersicht.

| Integritätstyp | Einschränkungstyp | Beschreibung |
|----------------|-------------------|---|
| Domäne | DEFAULT | Gibt einen Wert an, der für eine Spalte verwendet wird, wenn keine explizite Angabe dieses Werts erfolgt. |
| | CHECK | Schränkt den Wertebereich für eine Spalte ein. |
| | FOREIGN KEY | Gibt Datenwerte an, die basierend auf Werten einer Spalte in einer anderen Tabelle aktualisiert werden können. |
| | NULL | Gibt an, ob der Wert einer Spalte NULL sein kann. |
| Entität | PRIMARY KEY | Eindeutige Identifikation jedes Datensatzes in einer Tabelle. Stellt Eindeutigkeit her und lässt keine NULL-Marken zu. |
| | UNIQUE | Verhindert doppelte Einträge in Spalten. NULL-Marken sind zulässig. |
| Referenziell | FOREIGN KEY | Definiert eine oder mehrere Spalten, deren Werte mit dem Primärschlüssel derselben oder einer anderen Tabelle übereinstimmen. |
| | CHECK | Gibt Datenwerte an, die basierend auf Werten in anderen Spalten derselben Tabelle für die Spalte zulässig sind. |

11.2.2 Erstellen von Einschränkungen

Einschränkungen zur Sicherstellung der Datenintegrität einer Tabelle können mit der CREATE TABLE- aber auch mit der ALTER TABLE- Anweisung erstellt werden. Einschränkungen können zu einer leeren Tabelle hinzugefügt werden oder zu einer Tabelle die bereits Daten enthält. Einschränkungen können für eine oder mehrere Spalten erstellt werden.

Einschränkungen können erstellt, geändert oder gelöscht werden, ohne dass die Tabelle gelöscht und neu erstellt werden muss.

Für die Einschränkungen sollten Namen vergeben werden, da SQL Server komplizierte vom System generierte Namen erzeugt. Diese Namen müssen innerhalb der Datenbank eindeutig sein und den Regeln für Bezeichner entsprechen.

Um Informationen über Einschränkungen zu bekommen führt man die gespeicherten Systemprozeduren **sp_helpconstraint** bzw. **sp_help** oder die Informationsschemasichten **check_constraints**, **referential_constraints** und **table_constraints** ab.

11.2.2.1 PRIMARY KEY

Ein zentraler Lehrsatz des relationalen Models lautet, dass jede Tabelle einen Primärschlüssel besitzt über den sich ein Datensatz eindeutig identifizieren lässt.

Man könnte die Kombination aller Tabellenspalten (eine Tabelle enthält keine doppelten Datensätze) als diesen eindeutigen Bezeichner verwenden, aber normalerweise minimiert man die Anzahl der Spalten für einen Primärschlüssel. Es kann vorkommen, dass möglicherweise mehrere eindeutige Spalten für einen Primärschlüssel in Frage kommen. Dann sollte man sich für eine (in der Regel die Spalte mit der Verknüpfungen mit anderen Tabellen vorgenommen werden) entscheiden. Die Spalten die Primärschlüsseleigenschaften besitzen aber nicht für den Primärschlüssel ausgewählt werden, bezeichnet man häufig als Alternativ- oder Kandidatenschlüssel. Diese werden mit der UNIQUE- Einschränkung deklariert.

Syntax:

```
[constraint constraint_name]
primary key [{clustered | nonclustered}] [(spalte [{asc | desc}][, ...])]
[ {with fillfactor = fillfactor | with (<index_option> [...])}]
[on {filegroup | "default" | partition_scheme_name (partition_spalte)}]

<index_option>::=
{
    pad_index = {on | off}
    fillfactor = fillfactor
    ignore_dup_key = {on | off}
    statistics_norecompute = {on | off}
    allow_row_locks = {on | off}
    allow_page_locks = {on | off}
    data_compression = {none | row | page }
    [on partitions ({partition_number | range } [ , ...n ])]
}
```

Richtlinien:

- Pro Tabelle kann es nur einen Primärschlüssel geben.
- Die Einschränkung erstellt standardmäßig einen gruppierten Index.
- Der Index kann nicht gelöscht werden
- NULL- Marken sind nicht zulässig.
- Ein zusammengesetzter Primärschlüssel kann maximal 16 Spalten enthalten und die Größe darf 900 Bytes nicht überschreiten.

- Die Einschränkung überprüft die in der Tabelle enthaltenen Daten.
- Die Einschränkung kann nicht verzögert oder deaktiviert werden.
- Die Spaltenwerte der Primärschlüsselspalten müssen eindeutig sein.

Beispiel:

Primärschlüssel für die Tabelle Lieferant erstellen.

```
alter table lieferant
add constraint lnr_ps primary key(lnr)
```

11.2.2.2 UNIQUE

Eine UNIQUE- Einschränkung legt fest, dass zwei Zeilen einer Spalte nicht denselben Wert enthalten können.

Diese Einschränkung ist hilfreich, wenn bereits ein Primärschlüssel für die Tabelle definiert wurde aber sichergestellt werden soll, dass die so genannten Alternativschlüssel ebenfalls nur eindeutige Werte enthalten.

Syntax:

```
[constraint constraint_name]
unique ({clustered | nonclustered}) [[(spalte [{asc | desc}][,...])]]
[ {with fillfactor = fillfactor | with (<index_option> [,...])}]
[on {filegroup | "default" | partition_scheme_name (partition_spalte)}]

<index_option>::= 
{
    pad_index = {on | off}
|    fillfactor
|    ignore_dup_key = {on | off}
|    statistics_norecompute = {on | off}
|    allow_row_locks = {on | off}
|    allow_page_locks = {on | off}
|    data_compression = {none | row | page }
    [on partitions ({partition_number | range } [ , ...n ])]
}
```

Richtlinien:

- Pro Tabelle kann es mehr als eine UNIQUE- Einschränkung geben.
- Die Einschränkung erstellt standardmäßig einen nicht gruppierten Index.
- Der Index kann nicht gelöscht werden.
- NULL- Marken sind erlaubt.
- Die Einschränkung überprüft die in der Tabelle enthaltenen Daten.
- Die Einschränkung kann nicht verzögert oder deaktiviert werden.
- Die Spaltenwerte müssen eindeutig sein.

Beispiel:

Festlegen, dass die Spalte "vers_nr" in Tabelle Lieferant nur eindeutige Werte enthalten darf.

```
alter table lieferant
add constraint ver_nr_unq unique(vers_nr);
```

11.2.2.3 DEFAULT

Eine DEFAULT- Einschränkung gibt einen Wert in eine Spalte ein, wenn für diese Spalte bei einer INSERT- Anweisung kein entsprechender Wert angegeben wird.

Syntax:

```
[constraint constraint_name]
default default_wert [for spaltenname] [ with values]
```

Richtlinien:

- Die Einschränkung überprüft nicht die in der Tabelle enthaltenen Daten.
- Die Einschränkung gilt nur für die INSERT- Anweisung.
- Die WITH VALUES- Klausel wird nur beim Hinzufügen einer neuen Spalte mit gleichzeitiger DEFAULT- Einschränkung wirksam.
- Diese Einschränkung erstellt keinen Index.
- Die Einschränkung kann nicht verzögert oder deaktiviert werden.
- NULL- Marken sind erlaubt.
- Es kann mehr als eine DEFAULT- Einschränkung pro Tabelle geben. Es ist wenig sinnvoll, auf PRIMARY KEY- oder UNIQUE- Spalten eine DEFAULT- Einschränkung zu erstellen.
- Die Einschränkung sollte nicht gegen eine CHECK- Einschränkung oder eine RULE verstößen.
- Eine Spalte kann nur eine Default- Einschränkung besitzen.

Beispiel:

Den Standardwert für die Spalte "plz" festlegen

```
alter table lieferant
add constraint plz_def default '99085' for plz;
```

11.2.2.4 CHECK

Eine CHECK- Einschränkung beschränkt die Daten, die Benutzer in eine Spalte eingeben können, auf bestimmte Werte.

Syntax:

```
[constraint constraint_name]
check [not for replication] (logische_bedingung)
```

Richtlinien:

- Diese Einschränkung erzeugt keinen Index.
- Es können mehr als eine CHECK- Einschränkung pro Tabelle erstellt werden.
- Pro Spalte kann es mehr als eine CHECK- Einschränkung geben.
- Die Daten werden bei jeder INSERT- oder UPDATE- Anweisung überprüft.
- Die Einschränkung überprüft die in der Tabelle enthaltenen Daten.
- Diese Einschränkung kann verzögert und deaktiviert werden.
- NULL- Marken sind erlaubt.
- Die Einschränkung kann auf andere Spalten der Tabelle verweisen.

- Die Bedingung ist vergleichbar mit einer WHERE- Klausel, da mehrere Bedingungen mit AND oder OR verknüpft werden können.
- Die Bedingung darf keine Unterabfrage enthalten.

Beispiel:

Den Wertebereich für die Spalte "plz" festlegen.

```
alter table lieferant
add constraint plz_chk check (plz like '[0-9] [0-9] [0-9] [0-9] [0-9]'
and plz <> '00000');
```

11.2.2.5 FOREIGN KEY

Die FOREIGN KEY- Einschränkung erzwingt referenzielle Integrität. Das heißt, alle Nicht-NULL- Werte der Fremdschlüsselspalte müssen in der referenzierten Primärschlüsselspalte (oder auch in einer referenzierten UNIQUE- Spalte) vorhanden sein. Eine FOREIGN KEY- Einschränkung definiert eine Referenz auf eine PRIMARY KEY- oder UNIQUE- Einschränkung in derselben oder einer anderen Tabelle.

Eine Foreign- Key- Einschränkung darf maximal 16 Spalten enthalten und die Größe darf 900 Bytes nicht überschreiten. Es wird empfohlen, in einer Tabelle nicht mehr als 253 FOREIGN KEY- Einschränkungen zu erstellen, und nicht mehr als 253 FOREIGN KEY- Einschränkungsverweise auf eine Tabelle zuzulassen.

Weiterhin ermöglicht die FOREIGN KEY- Einschränkung eine Festlegung zu Operationsregeln für Fremdschlüssel zu treffen. Die Auswahl besteht zwischen den **Updateregeln** "bedingtes Ändern", "kaskadierendes Ändern", "Ändern mit NULL- Setzung" und "Ändern mit DEFAULT- Setzung", sowie den **Löschregeln** "bedingtes Löschen", "kaskadierendes Löschen", "Löschen mit NULL- Setzung" und "Löschen mit DEFAULT- Setzung".

Syntax:

```
[constraint constraint_name]
[foreign key [(spalte [...])]] references tab_name [(spalte [...])]
[on delete {cascade | no action | set null | set default}]
[on update {cascade | no action | set null | set default }]
[not for replication]
```

Beispiel:

Die Fremdschlüssel für die Tabelle "Lieferung" definieren. Dabei wird die referenzielle Integrität eingerichtet und die Operationsregeln "bedingtes Löschen" und "kaskadierendes Ändern".

```
alter table lieferung
add constraint Inr_fs foreign key(Inr) references lieferant(Inr)
on update cascade,
constraint anr_fs foreign key(anr) references artikel(anr)
on update cascade;
```

11.2.3 Verzögern und Deaktivieren von Einschränkungen

Aus Leistungsgründen kann es empfehlenswert sein, Einschränkungen für eine gewisse Zeit zu deaktivieren. Oder es muss eine Einschränkung erstellt werden, aber es ist nicht bekannt ob alle bereits in der Tabelle vorhandenen Werte die Bedingungen der Einschränkung erfüllen.

Diese Option steht nur für das CHECK- und FOREIGN KEY- Constraint zur Verfügung. Alle anderen Einschränkungen dürfen nicht verzögert oder deaktiviert werden.

11.2.3.1 Verzögern der Einschränkungsüberprüfung

Diese Möglichkeit wird verwendet, wenn eine Einschränkung für eine Tabelle erzeugt werden soll in welcher bereits Daten enthalten sind. SQL Server prüft bei der Erstellung einer Einschränkung prinzipiell vorhandene Daten. Wenn nun einige Datensätze die Anforderung der Einschränkung nicht erfüllen würden, könnte die Einschränkung nicht erstellt werden. Damit das nicht passiert können die Einschränkungen CHECK und FOREIGN KEY bei ihrer Erstellung verzögert werden.

Syntax:

```
alter table tablename with {check | nocheck}
add constraint ...
```

11.2.3.2 Deaktivieren der Einschränkungsüberprüfung

Für vorhandene CHECK- und FOREIGN KEY- Einschränkungen kann die Einschränkungsüberprüfung deaktiviert werden, damit Daten die in der Tabelle geändert werden oder der Tabelle hinzugefügt werden, nicht im Hinblick auf Einhaltung der Einschränkung überprüft werden.

Syntax:

```
alter table tablename
{check | nocheck} constraint {all | constraint_name [,...]}
```

Beispiel:

Deaktivieren des Fremdschlüssels "Inr_fs" in der Tabelle "Lieferung".

```
alter table lieferung
nocheck constraint Inr_fs;
```

11.2.4 Standardwerte und Regeln

Standards und Regeln sind globale Objekte, die an eine oder mehrere Spalten oder benutzerdefinierte Datentypen gebunden werden können.

Sie werden nur einmal definiert und können wiederholt verwendet werden.

Hinweis: Da die Funktionalität von Standards und Regel nicht dem ANSI-Standard entsprechen, werden sie in zukünftigen Versionen von SQL Server nicht mehr unterstützt. Sie können die gleichen Einschränkungen mit CHECK- oder DEFAULT- Constraints sicherstellen.

11.2.4.1 Erstellen eines Standards

Wenn bei einer INSERT- Anweisung kein Wert für eine Spalte angegeben wurde, wird durch einen Standard ein Wert für diese Spalte angegeben, an die das Objekt gebunden ist.

| | | |
|--------------------------|---------------------------------------|--|
| Erstellen: | create default name as defaultwert | create default def_ort as 'Erfurt' |
| Binden: | sp_bindefault name, 'objektname' | Tabellenspalte: sp_bindefault def_ort, 'lieferant.Istadt' Benutzerdef. Datentyp: sp_bindefault def_ort, 'orte' |
| Bindung aufheben: | sp_unbindefault 'objektname' | Tabellenspalte: sp_unbindefault 'lieferant.Istadt' Benutzerdef. Datentyp: sp_bindefault 'orte' |
| Löschen: | drop default name | drop default def_ort |

Richtlinien:

- Der Standardwert wird von sämtlichen Regeln überprüft, die an die Spalte oder Datentypen gebunden sind.
- Der Standardwert muss jede CHECK- Einschränkung einhalten, die für die Spalte definiert wurde.
- Wenn eine Spalte einen benutzerdefinierten Datentyp verwendet, an den ein Standard gebunden ist, kann für diese Spalte keine DEFAULT- Einschränkung erstellt werden.

11.2.4.2 Erstellen einer Regel

Regeln geben die Werte an, die in eine Spalte eingegeben werden können. Dadurch wird sichergestellt, dass Daten innerhalb eines angegebenen Wertebereiches liegen, einem bestimmten Muster entsprechen oder mit Einträgen einer Liste übereinstimmen.

Richtlinien:

- Die Regeldefinition kann Ausdrücke enthalten, die in einer WHERE- Klausel zulässig sind.
- Es kann nur eine Regel an eine Spalte oder einen benutzerdefinierten Datentyp gebunden werden.
- Sie kann nicht an Spalten vom Datentyp varchar(max), nvarchar(max), varbinary(max), text, image, varbinary und timestamp gebunden werden. Weiterhin kann sie auch an keinen XML Datentyp oder CLR- benutzerdefinierten Datentyp gebunden werden.

| | | |
|--------------------------|----------------------------------|---|
| Erstellen: | create rule name as bedingung | create rule rul_ort as @var like '[A-Z]%' |
| Binden: | sp_bindrule name, 'objektname' | Tabellenspalte: sp_bindrule rul_ort, 'lieferant.Istadt' Benutzerdef. Datentyp: sp_bindrule rul_ort, 'orte' |
| Bindung aufheben: | sp_unbindrule 'objektname' | Tabellenspalte: sp_unbindrule 'lieferant.Istadt' Benutzerdef. Datentyp: sp_bindrule 'orte' |
| Löschen | drop rule name | drop rule rul_ort |

Die CREATE DEFAULT- und CREATE RULE- Anweisung stehen in kommenden Versionen von SQL Server evtl. nicht mehr zur Verfügung. Sie sollten anstelle dieser Objekte mit CHECK- und DEFAULT- Constraints arbeiten.

12 Indizes

Durch verwenden von Indizes kann die Datenbankleistung bei Abfragen erheblich verbessert werden. Es ist eines der leistungsstärksten Werkzeuge für Entwickler von Datenbanken die Zugriffszeit auf große Datenmengen erheblich zu verkürzen.

SQL Server kann den Zugriff auf die Daten auf zwei Arten durchführen.

- Durchsuchen aller Datenseiten einer Tabelle. Man spricht von einem Tabellenscan.
Diese wird wie folgt ausgeführt:
 1. Startpunkt ist der Anfang der Tabelle.
 2. Alle Zeilen der Tabelle werden Seite für Seite durchsucht.
 3. Die Datensätze die den Abfragebedingungen entsprechen werden selektiert.
- Durchsuchen der Tabelle über einen Index
 1. Die Baumstruktur wird nach Daten durchsucht die den Abfragebedingungen entsprechen.
 2. Nur auf die Datensätze die den Abfragebedingungen entsprechen wird zugegriffen.

SQL Server ermittelt zuerst ob ein Index vorhanden ist. Danach wird durch den Abfrageoptimierer ermittelt ob die Nutzung des Indexes oder ein Tabellenscan effektiver ist, die Datensätze zu ermitteln.

Bei einem Index handelt es sich um eine Hilfsdatenstruktur, die von SQL Server für den Datenzugriff verwendet wird. Ein Index kann für Tabellen oder Sichten erstellt werden.

Der Tabellen- oder Sichtbesitzer kann jederzeit Indizes erstellen, unabhängig davon ob sich in den Objekten schon Daten befinden. Der Benutzer muss ein Mitglied der festen Serverrolle **sysadmin** bzw. der festen Datenbankrollen **db_ddladmin** und **db_owner** sein.

Ein Index wird je nach Typ zusammen mit den Daten oder getrennt davon gespeichert.

Die Struktur von Indizes in SQL Server 2008 sind balancierte Bäume (B- Tree). Ein solcher Baum besteht aus einem Stammknoten (Wurzel), der eine einzige Datenseite enthält, null oder mehrere Zwischenebenen (Knoten) und einer Blattebene.

12.1 Planen von Indizes

Um sicherzustellen, dass der Index effizienter als ein Tabellenscan ist, sollten zwei Faktoren berücksichtigt werden: **die Art der Daten in der Tabelle und die Art der Abfragen für die Tabelle.**

Gründe für die Erstellung:

- Beschleunigen die Ausführung von Abfragen, bei denen Tabellen verknüpft oder Sortier- bzw. Gruppierungsfunktionen angewendet werden.
- Eindeutige Indizes erzwingen die Eindeutigkeit von Spaltenwerten.
- Indizes werden in aufsteigend sortierter Reihenfolge erstellt und verwaltet.
- Indizes sollten für Spalten mit hoher Selektivität erstellt werden.

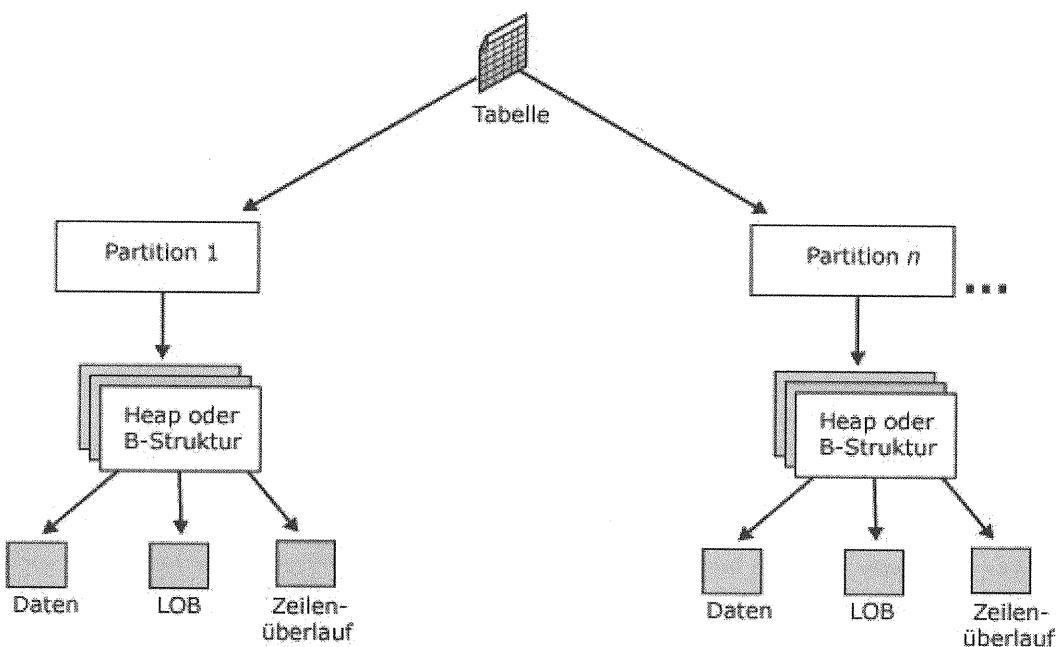
Gründe gegen die Erstellung:

- Beim Einfügen, Löschen oder Ändern von Daten, werden die zugehörigen Indizes aktualisiert.
- Die Verwaltung der Indizes erfordert Zeit und Ressourcen.
- Bei Spalten mit geringer Selektivität, bieten Indizes nur wenig Vorteile.

12.2 Organisationsstruktur von Tabellen und Indizes

12.2.1 Organisation von Tabellen

Eine Tabelle befindet sich in einer oder mehreren Partitionen, und jede Partition enthält Datenzeilen entweder in einer Heap- oder in einer gruppierten Indexstruktur. Die Seiten des Heaps oder des gruppierten Indexes werden je nach den Spaltentypen in den Datenzeilen in einer oder mehreren Zuordnungseinheiten verwaltet.



12.2.2 Partitionen

Tabellen- und Indexseiten sind in einer oder mehreren Partitionen enthalten. Eine Partition ist eine benutzerdefinierte Datenorganisationsstruktur. Standardmäßig befindet sich jede Tabelle und jeder Index nur in einer Partition. Darin sind alle Tabellen- bzw. Indexseiten enthalten. Die Partition befindet sich in einer einzigen Dateigruppe.

Wenn eine Tabelle oder ein Index mehrere Partitionen verwendet, sind die Daten horizontal partitioniert, Gruppen von Datenzeilen sind basierend auf einer angegebenen Spalte einzelnen Partitionen zugeordnet. Die Partitionen können sich in einer oder mehreren Dateigruppen in der Datenbank befinden. Die Tabelle oder der Index wird als einzelne logische Einheit, bei Abfragen oder Aktualisierungen von Daten, behandelt

Mit der Katalogsicht **sys.partitions** können die Partitionen, die von einer Tabelle oder einem Index verwendet werden, angezeigt werden.

12.2.3 Gruppierte Tabellen, Heaps und Indizes

Tabellen verwenden eine von zwei möglichen Methoden, um ihre Datenseiten innerhalb einer Partition zu organisieren:

- Gruppierte Tabellen. Das sind Tabellen die über einen gruppierten Index verfügen.
- Heaps sind Tabellen, die nicht über einen gruppierten Index verfügen. Die Tabelle befindet sich auch noch in der Methode Heap wenn sie einen oder mehrere nichtgruppierte Indizes besitzt.

Indizierte Sichten weisen dieselbe Speicherstruktur auf wie gruppierte Tabellen.

12.2.4 XML-Indizes

Für jede XML- Spalte in der Tabelle können ein primärer und mehrere sekundäre XML- Indizes erstellt werden. Ein XML-Index ist eine aufgeteilte und dauerhafte Darstellung der XML-BLOBs in der XML- Datentypspalte. XML-Indizes werden als interne Tabellen gespeichert.

12.2.5 Zuordnungseinheiten

Eine Zuordnungseinheit ist eine Auflistung von Speicherseiten innerhalb einer Heap- oder B- Baumstruktur, die zum Verwalten von Daten basierend auf ihrem Seitentyp verwendet wird. Die folgende Tabelle zeigt die Typen von Zuordnungseinheiten.

| Typ | Verwalten von |
|-------------------|--|
| IN_ROW_DATA | Daten- oder Indexzeilen, die alle Daten enthalten, mit Ausnahme von LOB-Daten. Die Seiten besitzen den Typ Daten oder Index. |
| LOB_DATA | Daten für große Objekte (LOB), die in einem oder mehreren der folgenden Datentypen gespeichert sind: text, ntext, image, xml, varchar(max), nvarchar(max), varbinary(max) oder CLR- benutzerdefinierte Typen. Die Seiten besitzen den Typ Text/Bild. |
| ROW_OVERFLOW_DATA | Daten variabler Länge, die in varchar-, nvarchar-, varbinary- oder sql_variant-Spalten gespeichert sind, die das Zeilengrößenlimit von 8.060 Byte überschreiten. Die Seiten besitzen den Typ Daten. |

Eine Heap- oder B- Baumstruktur kann in einer bestimmten Partition jeweils nur eine Zuordnungseinheit jedes Typs enthalten. Zum Anzeigen der Informationen zu Zuordnungseinheiten kann die Katalogsicht **sys.allocation_units** abgefragt werden.

12.3 Indexarchitektur

Heap, gruppierte und nicht gruppierte Indizes weisen eine unterschiedliche Architektur auf.

Die Katalogsicht **sys.indexes** liefert wichtige Informationen Indizes betreffend. Sie Enthält eine Zeile pro Index oder Heap eines Tabellenobjekts, wie z. B. eine Tabelle, Sicht oder Tabellenwertfunktion. Die Spalte "**index_id**" gibt Auskunft über die Art der Datenspeicherung in einer Tabelle.

- Besitzt die Tabelle keinen Index enthält die Spalte den Wert 0, das bedeutet die Daten der Tabelle befinden sich in einem HEAP. SQL Server verwendet die IAM- Seiten für die Suche der Datenseiten.
- Für einen gruppierten Index auf die Basistabelle enthält die Spalte "index_id" den Wert 1. Die "root_page"- Spalte in **sys.system_internals_allocation_units** zeigt auf den Anfang des B*- Baumes des gespeicherten Index.
- Für jeden nicht gruppierten Index einer Tabelle (bis 999 Indizes) gibt es in "index_id" einen Wert (2 – 1000). Die "root_page"- Spalte in **sys.system_internals_allocation_units** zeigt auf den Anfang des B*- Baumes des nicht gruppierten Index.

12.3.1 Verwenden von Heap

Wenn für eine Tabelle kein gruppierter Index definiert wurde, verwaltet SQL Server Datenseiten in einem Heap (Haufen).

- Ein Heap besteht aus mehreren Datenseiten, die Zeilen einer Tabelle enthalten.
- Jede Datenseite enthält 8 KB Daten. Eine Gruppe von acht aufeinander folgenden Seiten ist ein Block.
- Die Datensätze werden nicht in einer bestimmten Reihenfolge gespeichert und es gibt auch keine signifikante Reihenfolge für die Datenseiten.
- Die Datenseiten sind nicht in einer verknüpften Liste verknüpft.
- Wenn eine Datenseite voll ist, wird die Datenseite geteilt.

SQL Server verwaltet Heaps mit Hilfe von IAM- Seiten (Index Allocation Map). Sie enthalten Informationen über den Speicherort der Blöcke innerhalb eines Heap und ermöglichen die Navigation in einem Heap. Sie ordnen der Tabelle Datenseiten zu.

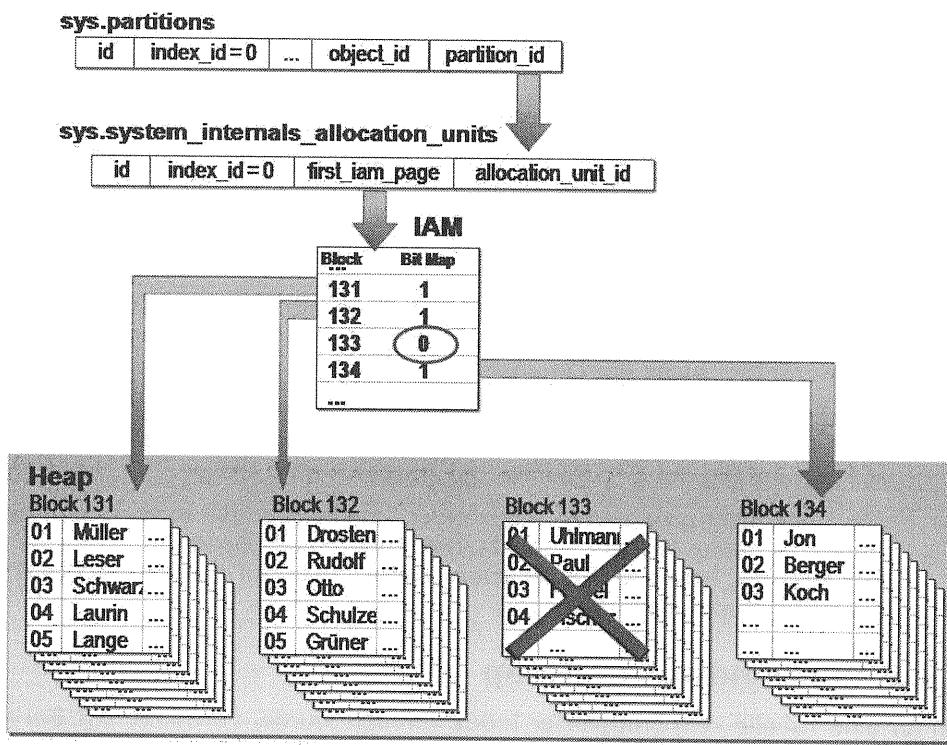
Je nach den im Heap enthaltenen Datentypen weist jede Heapstruktur eine oder mehrere Zuordnungseinheiten auf, um die Daten für eine bestimmte Partition zu speichern und zu verwalten.

Zumindest verfügt jeder Heap über eine IN_ROW_DATA- Zuordnungseinheit pro Partition. Der Heap hat außerdem eine LOB_DATA- Zuordnungseinheit pro Partition, wenn diese LOB-Spalten enthält.

Darüber hinaus verfügt er über eine ROW_OVERFLOW_DATA- Zuordnungseinheit pro Partition, wenn diese Spalten mit variabler Länge enthält, die das Zeilengrößenlimit von 8.060 Byte überschreiten

Gründe die für die Verwendung eines Heap sprechen sind folgende.

- Die Tabelle enthält strukturierte Daten, für die aufgrund ihrer Verwendung ein gruppierter Index unzweckmäßig ist.
- Die Tabellenwerte werden sehr häufig gelöscht und neue eingefügt. dadurch entfällt die Zeit für die Indexpflege durch das DBMS.
- Tabellen mit geringen Datenmengen.
- Spalten mit überwiegend doppelt vorhandenen Datenwerten.
- Spalten mit Daten die hauptsächlich geschrieben und selten gelesen werden.



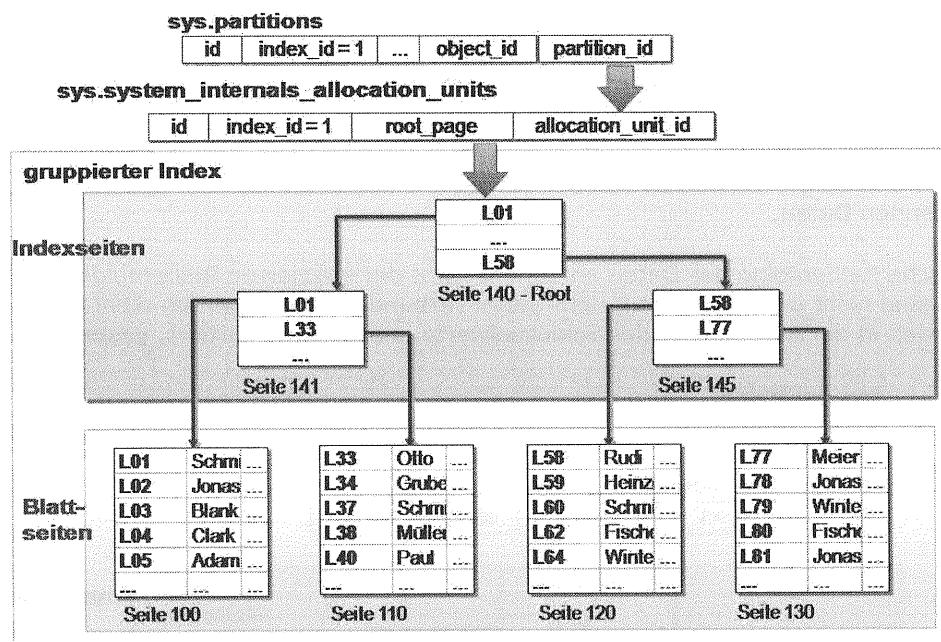
Die Systemsicht **sys.system_internals_allocation_units** enthält einen Zeiger auf die erste IAM- Tabelle des Heap. Die IAM- Tabelle ersetzt die serielle Verknüpfung eines konventionellen Heaps und dient dazu den Heap zu durchsuchen und Speicherplatz für neu einzufügende Datensätze zu finden. Der Speicherplatz in einem Heap wird für neue Datensätze freigegeben, wenn ein Datensatz gelöscht wird.

Ist für eine Tabelle überhaupt kein Index definiert, verwendet SQL Server einen Tabellenscan zum Abrufen der Daten. Dabei wird über die Systemtabelle **sys.indexes** nach der ersten IAM- Seite gesucht und alle in dieser Liste aufgeführten Datenblöcke sequentiell durchsucht.

Die Zeilen werden nicht sortiert zurückgegeben. Nach Löschtätigkeiten werden die Lücken durch neue Eingaben aufgefüllt.

12.3.2 Verwenden eines gruppierten Index

Ein gruppierte Index organisiert die Datensätze in einer Baum- Struktur. Bei einem gruppierten Index sind die Datenseiten unmittelbarer Bestandteil des Indexes. Die Datensätze sind die Blätter des B- Baums. Das heißt, die physische Reihenfolge der Datensätze entspricht der indizierten Reihenfolge. Darum kann eine Tabelle auch nur einen gruppierten Index besitzen.



Ein gruppierte Index ist mit einem Inhaltsverzeichnis eines Buches zu vergleichen. So wie die Organisation eines Inhaltsverzeichnisses jemandem die Suche erleichtert, genauso kann SQL Server eine Tabelle mit einem gruppierten Index schneller durchsuchen.

Die Datensätze werden in der indizierten Reihenfolge zurückgegeben, das heißt er legt die Reihenfolge fest wie die Daten in der Tabelle gespeichert werden.

Hinweis: Je kürzer der Schlüsselwert eines gruppierten Index, desto mehr Indexeinträge passen in eine Indexseite und desto weniger Ebenen müssen durchsucht werden. Der Indexschlüssel sollte möglichst wenige Spalten umfassen.

Ein gruppierte Index sollte so vorteilhaft wie möglich eingesetzt werden, da es immer nur einen pro Tabelle geben kann.

Gruppierte Indizes sind bei Abfragen mit folgenden Merkmalen am effektivsten.

- Abfragen mit den Operatoren BETWEEN, <, > , <= und >=.
- Abfragen die die ORDER BY- oder Die GROUP BY- Klausel verwenden.
- Bei Abfragen die sehr große Resultset zurückgeben.

Gruppierte Indizes sollten für Spalten mit folgenden Eigenschaften erstellt werden.

- Die Spaltenwerte sind eindeutig, Eigenschaft IDENTITY, oder sie besitzen eine hohe Selektivität.
- Spalten die häufig zum Sortieren von Datensätzen verwendet werden.
- Auf Spalten auf die häufig sequentiell zugegriffen wird.

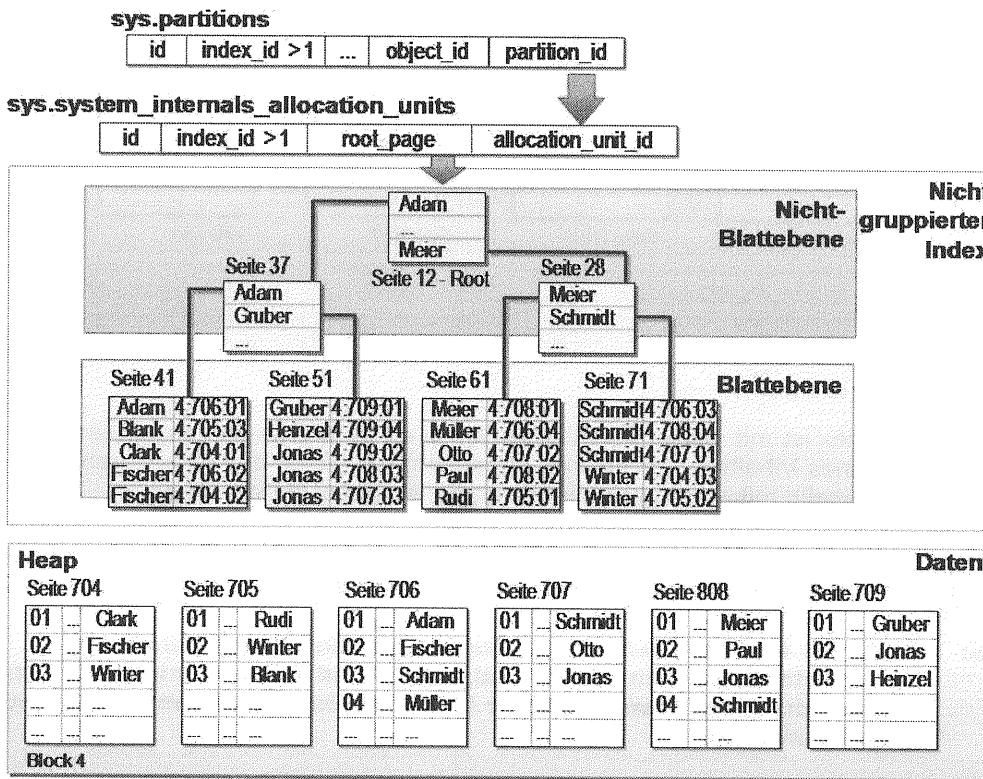
Für folgende Spalten sollten keine gruppierten Indizes verwendet werden.

- Spalten deren Daten häufig geändert werden.
- Auf Spalten mit sehr großen Datenwerten und auf mehrere gemeinsame Spalten sollte kein gruppierter Index erstellt werden.

12.3.3 Verwenden eines nicht gruppierten Index

Ein nicht gruppierter Index ist mit einem Sachwortregister eines Buches vergleichbar der einer Baum- Struktur entspricht. Die Daten und der Index sind an verschiedenen Orten gespeichert und Zeiger verweisen von der Blattebene des Index auf den Speicherort der entsprechenden Daten.

Die physische Reihenfolge der Daten entspricht nicht der indizierten Reihenfolge. Das heißt, die Daten sind nicht unmittelbar mit dem Index verbunden. Die Zeilen im nicht gruppierten Index werden in der Reihenfolge der Schlüsselwerte, aufsteigend sortiert, gespeichert



Auch der nicht gruppierter Index ist als B- Baum (B*- Tree) organisiert. **Eine Tabelle kann bis zu 999 nicht gruppierte Indizes besitzen.**

Wenn die indizierte Tabelle einen gruppierten Index aufweist, werden die im gruppierten Index definierten Spalten automatisch an das Ende sämtlicher nicht gruppierter Indizes für diese Tabelle angefügt. Dadurch kann eine abgedeckte Abfrage erstellt werden.

Bspiel:

Eine Tabelle besitze einen gruppierten Index auf die Spalte B und einen nicht gruppierten zusammengesetzten Index auf die Spalten D und E, dann verwendet der nicht gruppierte Index die Spalten D, E, und B.

Nicht gruppierte Indizes sind bei Abfragen mit folgenden Merkmalen am effektivsten.

- Wenn Benutzer über mehrere Möglichkeiten zum Abfragen von Daten verfügen müssen. Ein Mitarbeiter sucht in einer Kundentabelle häufig nach Nachname und Vorname aber auch nur nach Nachnamen.
- Steigern die Leistung bei häufig verwendeten Abfragen, die nicht durch einen gruppierten Index abgedeckt sind.
- Wenn durch den Index alle Spalten der Abfrage abgedeckt sind. Wobei Indizes über mehrere Spalten (zusammengesetzter Index) sich auch nachteilig auf die allgemeine Datenbankleistung auswirken können.
- Bei Abfragen mit JOIN- oder GROUP BY- Klausel. Indizieren sie gegebenenfalls die Fremdschlüsselspalten.
- Bei Abfragen auf Tabellen die sehr selten aktualisiert werden die aber eine große Menge an Daten enthalten.
- Bei Abfragen welche ein sehr kleines Resultset zurückgeben.
- Auf Spalten die sehr häufig in der WHERE- Klausel einer Abfrage vorkommen.

12.4 Richtlinien für die Indizierung

Um einen Index effektiv erstellen zu können sind einige wichtige Informationen über die Daten notwendig. Dazu gehören:

- Der logische und Physische Datenbankentwurf
- Die Eigenschaften der Daten.
- Die Art und Verwendung der Daten. Welche Art von Abfragen wird durch die Benutzer ausgeführt und wie häufig sind diese Abfragen.

Welche Spalten sollten nicht indiziert werden?

- Spalten die sehr selten abgefragt werden.
- Spalten die wenig eindeutige Werte enthalten.
- Spalten mit den Datentypen VARCHAR(max), NVARCHAR(max), VARBINARY(max), XML, TEXT, NTEXT und IMAGE, diese Spalten **können nicht** indiziert werden.

Um die Vorteile einer indizierten Spalte voll auszunutzen, ist es auch notwendig effektive Abfragen zu entwickeln.

12.5 Erstellen von Indizes

Um einen Index zu erstellen, verwendet man die CREATE INDEX- Anweisung oder das SQL Server Management Studio.

Syntax:

```
create [unique] [{clustered | nonclustered}] index index_name
on {tabname | viewname} (spaltenname [{asc | desc}] [...])
[include (spaltenname [...])]
[where spalte operator wert [{and | or}...]]
[with
(
[pad_index = {on | off}]
[, fillfactor = füllgrad]
[, ignore_dup_key = {on | off}]
[, drop_existing = {on | off}]
[, statistics_norecompute = {on | off}]
[, sort_in_tempdb = {on | off}]
[, online = {on | off}]
[, allow_row_locks = {on | off}]
[, allow_page_locks = {on | off}]
[, maxdop = max_grad_an_parallelität]
[, data_compression = {none | row | page}
    on partitions({partitions_nummer | bereich} [...])
)
[on {patiton_schema_name(spalte)
    | filegroup_name
    | default}]
[filestream_on {filestream_filegroup_name
    | partition_schema_name
    | "null"}]
```

Beschreibung einiger Argumente der Syntax:

- INCLUDE
Gibt die Nichtschlüsselpalten an die zur Blattebene eines nicht gruppierten Indexes (auch unique) hinzugefügt werden sollen. Maximale Anzahl der Spalten beträgt 1023 und minimal 1 Spalte.
- ONLINE
Gibt, an ob zugrunde liegende Tabellen und dazugehörige Indizes bei der Indizierung für Abfragen und Datenänderungen verfügbar sind.
- ALLOW_ROW_LOCKS
gibt an, ob Zeilensperren beim Zugreifen auf den Index zulässig sind. Standardwert ist ON.
- ALLOW_PAGE_LOCKS
Gibt an, ob Seitensperren beim Zugreifen auf den Index zulässig sind. Standardwert ist ON.
- MAXDOP
Damit kann die Anzahl der Prozessoren (max. 64), für den Indexvorgang, begrenzt werden die bei der Ausführung paralleler Pläne verwendet werden. Mögliche Werte 1 (Unterdrückt das Generieren paralleler Pläne), >1 (Beschränkt die maximale Anzahl der Prozessoren die bei einem parallelen Indexvorgang verwendet werden auf den angegebenen Wert) und die 0 (verwendet je nach Systemauslastung die tatsächliche Anzahl der Prozessoren) die Standardwert ist.
- FILESTREAM_ON
Gibt die Platzierung der FILESTREAM- Daten für die Tabelle an, wenn ein gruppierter Index erstellt wird. Die Klausel lässt zu, dass FILESTREAM- Daten in einen anderen FILESTREAM- Dateigruppe oder ein anderes Partitionsschema verschoben werden.

- DATA_COMPRESSION
Legt die Datenkomprimierungsoption für den angegebenen Index, die Partitionsnummer oder den Bereich von Partitionen fest.

Alle anderen Argumente sind schon aus früheren Versionen bekannt. Die Anwendung der alten Syntax ist aus Gründen der Abwärtskompatibilität möglich.

Folgende Aspekte und Richtlinien sollten berücksichtigt werden.

- Wenn für eine Tabelle eine PRIMARY KEY- oder ein UNIQUE- Einschränkung erstellt wird, werden automatisch Indizes erstellt.
- Eine Tabelle darf 1 gruppierten und 999 nichtgruppierte Indizes besitzen.
- Die Länge des Index darf 900 Byte nicht überschreiten.
- In einem Zusammengesetzten Index dürfen maximal 16 Spalten angegeben werden.
- Partitionierte Indizes werden ähnlich wie partitionierte Tabellen erstellt und verwaltet. Es kann für eine partitionierte Tabelle ein nicht partitionierter Index erstellt werden und für eine nicht partitionierte Tabelle kann ein partitionierter Index erstellt werden.
- Beim Erstellen eines gruppierten Indexes muss in der Datenbank ungefähr das 1,2 fache der Größe der Daten an Speicherplatz frei sein. Bei Verwendung der Option „drop_existing“ wird für den gruppierten Index ebenso viel Speicherplatz benötigt wie für den vorhandenen Index.
- Indizes können auch für temporäre Tabellen erstellt werden. Wird die Tabelle gelöscht oder die Sitzung beendet, werden alle Indizes gelöscht.
- Informationen über den Index werden in die Systemtabelle **sys.indexes** geschrieben.
- Pro Spalte (Spaltenkombination) kann nur ein Index erstellt werden.
- Alle Indizes haben innerhalb der Datenbank einen eindeutigen Namen.
- Ein gruppierter Index sollte immer vor den nicht gruppierten Indizes erstellt werden weil sonst alle nicht gruppierten Indizes wieder neu durch das DBMS erstellt werden müssen.

12.5.1 Füllgrad von Indizes

Um die Leistung von Insert- und Update-Anweisungen auf indizierte Tabellen zu erhöhen, ist es möglich einen "globalen Erstfüllgrad" für die Indexerstellung festzulegen.

12.5.1.1 Option FILLFACTOR

Die Größe des Füllfaktorwertes ist davon abhängig, wie oft Daten geändert (Insert/Update) geändert werden.

Der angegebene Prozentsatz wird von SQL Server nicht dynamisch verwaltet. Er gilt nur für den Zeitpunkt zu dem der Index erstellt wird.

| Füllgrad in Prozent | Blattseiten | Seiten der Nicht-Blatteinheit | Arbeitsumgebung |
|--------------------------|---|-------------------------------|------------------------------|
| 0 (Standard-einstellung) | 100% gefüllt | Platz für einen Indexeintrag | Analysis Service |
| 1-99 | gefüllt bis zum angegebenen Prozentsatz | Platz für einen Indexeintrag | OLTP oder gemischte Umgebung |
| 100 | 100% gefüllt | Platz für einen Indexeintrag | Analysis Service |

Der Standardfüllfaktor kann mit der gespeicherten Systemprozedur "**sp_configure**" geändert werden.

12.5.1.2 Option PAD_INDEX

Mit der Option PAD_INDEX kann der Prozentsatz angegeben werden, bis zu dem zusätzlich die Seiten der Nicht- Blatteinheit gefüllt werden können.

Die Option PAD_INDEX wird in Verbindung mit der Option FILLFACTOR verwendet.

| Füllgrad in Prozent | Blattseiten | Seiten der Nicht- Blatteinheit | Arbeitsumgebung |
|---------------------------|---|---|------------------|
| 0 (Standard- einstellung) | 100% gefüllt | Platz für einen Indexeintrag | Analysis Service |
| 1-99 | gefüllt bis zum angegebenen Prozentsatz | gefüllt bis zum angegebenen Prozentsatz | OLTP |
| 100 | 100% gefüllt | Platz für einen Indexeintrag | Analysis Service |

Beispiel:

Erzeugen eines nichtgruppierten Index mit einem Erstfüllgrad von 75 % der Blatt- und Indexseiten.

```
create index lstadt_ind on lieferant(lstadt) with (pad_index = on, fillfactor = 75);
```

12.5.1.3 DROP_EXISTING

Gibt an, dass der benannte, bereits vorhandene gruppierte oder nichtgruppierte Index gelöscht und neu erstellt wird.

Bei stark fragmentierten Daten ist eine Defragmentierung oftmals zeitaufwendiger als die Neuerstellung des Index.

Mit DROP_EXISTING kann die Eigenschaft eines Index verändert werden, Spalten hinzufügen oder löschen, Optionen ändern, die Sortierreihenfolge für Spalten ändern sowie das Partitionsschema oder die Dateigruppe ändern. Der Indextyp kann nicht geändert werden!

Indizes die mit der Einschränkung PRIMARY KEY und UNIQUE erstellt wurden, können mit der Anweisung auch geändert werden.

Die Klausel erhöht die Leistung beim Neuerstellen eines gruppierten Indexes (mit der gleichen oder einer anderen Schlüsselmenge) für eine Tabelle die auch nichtgruppierte Indizes besitzt. Die nichtgruppierten Indizes werden dabei nur neu erstellt wenn sich die Indexdefinition des gruppierten Schlüssels geändert hat. Die Blattseiten und die Nicht- Blattseiten des Index werden neu organisiert, die Fragmentierung wird entfernt und die Indexstatistiken werden neu erstellt.

Folgende Indexeigenschaften können verändert werden:

- Einem zusammengesetzten Index können zusätzliche Spalten hinzugefügt werden oder Spalten wieder entfernt werden.
- Ein nicht gruppierter Index kann in einen gruppierten Index geändert werden. Umgekehrt ist das nicht möglich.
- Für einen Index können neue Spalten angegeben werden.
- Ein eindeutiger Index kann in einen nicht eindeutigen Index umgewandelt werden und umgekehrt.
- Der Füllgrad für die Blattseiten und Nicht- Blattseiten kann verändert werden.

Beispiel:

Erzeugen eines nichtgruppierten Index mit der Option DROP_EXISTING.

```
create index lstadt_ind on lieferant(lstadt) with (drop_existing = on);
```

12.5.2 Eindeutige Indizes

Durch eindeutige Indizes wird sichergestellt, dass in einer Spalte (Spaltenkombination) zu keiner Zeit doppelte Werte vorkommen.

Eindeutige Indizes werden für Spalten erstellt, die die Eigenschaften eines Primärschlüssels besitzen aber nicht zum Primärschlüssel gemacht wurden (z.B. eindeutige Versicherungsnummern).

Wenn beim Einfügen neuer Datensätze in eine Tabelle mit einem eindeutigen Index für die indizierte Spalte doppelte Datenwerte festgestellt werden, dann wird die ganze Transaktion abgebrochen. Um das zu vermeiden kann der Index mit der Option IGNORE_DUP_KEY erzeugt werden. Dadurch werden nicht mehr alle einzufügenden Datensätze abgewiesen sondern nur noch die Datensätze mit den doppelten Schlüsselwerten.

Die Eigenschaften PRIMARY KEY und UNIQUE erzeugen immer eindeutige Indizes.

Beispiel:

Erzeugen eines nichtgruppierten eindeutigen Index.

```
create index vers_nr_ind on lieferant(vers_nr);
```

Die Eindeutigkeit bei vorhandenen Spaltenwerten muss vor der Indexerstellung gewährleistet werden.

12.5.3 Index für Spalten mit variabler Breite

Für einen Indexschlüssel beträgt die maximal zulässige Größe 900 Byte. SQL Server lässt aber auch Indizes für Spalten zu, die eine große variable Breite haben und damit die maximale Größe von 900 Byte überschreiten. SQL Server gibt aber eine Warnung aus.

Solange die eingefügten oder geänderten Datenwerte für diese Spalten 900 Byte nicht überschreiten wird die Datenänderung in die Tabelle übernommen. Andernfalls wird die Aktualisierungsanweisung abgebrochen.

12.5.4 Zusammengesetzte Indizes

Bei einem zusammengesetzten Index werden immer mehrere Spalten als Schlüsselwert angegeben.

Einen zusammengesetzten Index wird auf Spalten erstellt die zusammen gesucht werden (Suchbedingung in der WHERE- Klausel) oder wenn sich Abfragen nur auf diese Indexspalten beziehen (abgedeckter Index).

Beim Erstellen zusammengesetzter Indizes ist folgendes zu berücksichtigen:

- Bei einem zusammengesetzten Index darf die Gesamtgröße aller Spalten höchstens 900 Byte umfassen und er darf aus maximal 16 Spalten bestehen. Bei Spalten mit fester Länge wird die Indexerstellung abgelehnt, bei Spalten mit variabler Länge wird der Index mit einer Warnung erstellt.
- Alle Spalten des Index entstammen derselben Tabelle.

- Die Spalte die am eindeutigsten ist, sollte zuerst angegeben werden, danach die weniger selektiven Spalten.
- In der WHERE- Klausel der Abfrage muss auf die erste Spalte des Index verwiesen werden damit der Index für die Abfrage genutzt wird.
- Ein zusammengesetzter Index (Spalte_A, Spalte_B) unterscheidet sich grundsätzlich von einem zusammengesetzten Index (Spalte_B, Spalte_A).

12.5.5 Deckende Indizes

Wird ein Index erstellt, werden alle Werte des Indexschlüssels in den Index geladen. Somit ist jeder Index praktisch eine Minitabelle, die alle Werte enthält, aus denen sich der Index zusammensetzt. Darum lässt sich eine Abfrage unter Umständen allein aus den Daten im Index beantworten.

Ein Index der so aufgebaut ist wird als "deckender Index" bezeichnet.

Wenn eine Abfrage einen solchen Index verwenden kann, erhöht sich unter Umständen die Geschwindigkeit der Abfrage und die Parallelität, weil Abfragen auf den Index zugreifen können, während Änderungen, die sich nicht auf den Index auswirken in die zugrundeliegende Tabelle geschrieben werden.

SQL Server ist auch in der Lage, mehrere Indizes für eine bestimmte Abfrage zu nutzen. Wenn zwei Indizes mindestens eine Spalte gemeinsam haben, kann SQL Server die beiden Indizes verknüpfen, um eine Abfrage zu beantworten.

12.5.6 Index mit eingeschlossenen Spalten

Bei der Erstellung von zusammengesetzten Indizes ist die Anzahl der Indexspalten auf 16 Spalten und die Indexlänge auf 900 Byte beschränkt. Das verhindert die Indizierung von Spalten mit umfangreichen Datentypen. Aber gerade für solche Spalten wäre ein deckender Index oftmals sehr nützlich.

Mit der INCLUDE- Klausel können "eingeschlossene" Spalten für den Index angegeben werden. Die Werte dieser Spalten werden im Index gespeichert, aber nur auf Blattebene. das bedeutet, dass sie nicht in der Wurzel- oder Astebenen stehen. damit gehen sie nicht in das 900 Byte- Limit eines Index ein.

Beispiel:

Erzeugen eines nichtgruppierten deckenden Index mit einer INCLUDE- Klausel.

```
create index ort_farb_ind on artikel(astadt, farbe) include(aname);
```

12.5.7 Gefilterte Indizes

Wenn ein größerer Teil der Tabelle doppelte Werte enthält, die innerhalb eines engen Bereichs aus dem gesamten Wertebereichs liegen, können die Datenwerte des Indexschlüssels sehr ungleichmäßig verteilt sein.

Wurde durch eine Abfrage mit einer hohen Selektivität der abgefragten Daten, ein bestimmter Index verwendet, wird der Abfrageoptimierer bei einer nachfolgenden Abfrage mit geringer Datenselektivität sehr wahrscheinlich den denselben Index verwenden. Das führt zu einer schlechten Abfrageleistung.

Um das zu verhindern können gefilterte Indizes erstellt werden. Das ist ein Index mit einer WHERE- Klausel. Nur die Indexschlüssel, die mit der WHERE- Klausel übereinstimmen, werden zum Index hinzugefügt.

Somit können Indizes erzeugt werden, die sich auf ganz bestimmte Abschnitte der Tabelle konzentrieren.

Beschränkungen:

- Nur nicht gruppierte Indizes können gefiltert werden.
- Berechnete Spalten, UDT- Spalten, Spalten mit räumlichen Datentypen und Spalten mit dem HIERARCHYID- Datentyp können nicht mit einem gefilterten Index versehen werden.
- Die mit einem gefilterten Index versehenen Spalte können in einer Abfrage keiner impliziten oder expliziten Datenkonvertierung unterzogen werden.
- Das Filterprädikat kann Spalten einschließen, die keine Schlüsselspalten im gefilterten Index sind.

Beispiel:

Erzeugen eines nichtgruppierten gefilterten Index für bestimmte Artikel.

```
create index ldatum_ind on lieferung(ldatum)
where ann in ('A02,A03',A04');
```

Erzeugen eines nichtgruppierten gefilterten Index für einen bestimmten Lieferzeitraum.

```
create index ldatum_ind on lieferung(ldatum)
where ldatum >= '02.01.2000' and ldatum <= '01.07.2000';
```

12.5.8 Partitionierte Indizes

Partitionierte Indizes werden ähnlich wie partitionierte Tabellen erstellt und verwaltet. Jedoch werden sie wie gewöhnliche Indizes als separate Datenobjekte behandelt.

Es kann ein partitionierter Index für eine nicht partitionierte Tabelle erstellt werden, und es kann ein nicht partitionierten Index für eine partitionierte Tabelle erstellt werden.

Wenn Sie einen Index für eine partitionierte Tabelle erstellen und keine Dateigruppe angeben, in die der Index platziert werden soll, wird der Index auf die gleiche Weise partitioniert wie die zugrunde liegende Tabelle. Der Grund hierfür ist, dass Indizes standardmäßig in dieselben Dateigruppen wie die zugrunde liegenden Tabellen platziert werden. Bei partitionierten Tabellen werden Indizes in dasselbe Partitionsschema platziert, das dieselben Partitionierungsspalten verwendet.

Beim Partitionieren eines nicht eindeutigen gruppierten Index fügt das Datenbankmodul standardmäßig alle Partitionierungsspalten zu der Liste der gruppierten Indexschlüssel hinzu, sofern sie dort noch nicht angegeben wurden.

Indizierte Sichten können für partitionierte Tabellen auf die gleiche Weise wie Indizes für Tabellen erstellt werden.

Beispiel:

Erzeugen eines nichtgruppierten partitionierten Index. Dabei wird das Partitionsschema aus dem Beispiel aus dem Kapitel "Partitionierte Tabellen" verwendet.

```
create index ldatum_ind on lieferung(ldatum)
on standardliefpartition(ldatum);
```

12.5.9 Komprimierter Index

SQL Server 2008 unterstützt sowohl die Zeilen- als auch die Seitenkomprimierung für Tabellen und Indizes. Die Datenkomprimierung kann für die folgenden Datenbankobjekte konfiguriert werden:

- Vollständige, als Heaps gespeicherte Tabellen
- Vollständige, als gruppierte Indizes gespeicherte Tabellen

- Vollständige nicht gruppierte Indizes
- Vollständige indizierte Sichten
- Bei partitionierten Tabellen und Indizes kann die Komprimierungsoption für jede Partition konfiguriert werden, wobei die verschiedenen Partitionen eines Objekts nicht die gleiche Komprimierungseinstellung aufweisen müssen.

Die Komprimierungseinstellung einer Tabelle wird für die zugehörigen nicht gruppierten Indizes nicht automatisch übernommen. Jeder Index muss einzeln festgelegt werden.

Für Systemtabellen ist die Komprimierung nicht verfügbar.

Tabellen und Indizes können komprimiert werden, wenn sie mit der CREATE TABLE- Anweisung bzw. mit der CREATE INDEX- Anweisung erstellt wurden. Verwenden Sie die ALTER TABLE- Anweisung oder die ALTER INDEX- Anweisung, um den Komprimierungsstatus einer Tabelle, eines Indexes oder einer Partition zu ändern.

Hinweis: Wenn die vorhandenen Daten fragmentiert sind, können Sie die Indexgröße möglicherweise ohne Komprimierung reduzieren, indem Sie den Index neu erstellen. Der Füllfaktor eines Indexes wird bei der Indexneuerstellung angewendet, wodurch sich die Indexgröße erhöhen kann.

12.5.9.1 Verwendung von Zeilen- und Seitenkomprimierung

Beachten Sie die folgenden Punkte, wenn Sie die Zeilen- und Seitenkomprimierung verwenden:

- Die Komprimierung ist nur in der Enterprise und Developer Edition von SQL Server 2008 verfügbar.
- Die Komprimierung ermöglicht, dass mehr Zeilen auf einer Seite gespeichert werden, die maximale Datensatzgröße einer Tabelle bzw. eines Indexes kann dadurch allerdings nicht geändert werden.
- Eine Tabelle kann nicht komprimiert werden, wenn die Zeilengröße die maximale Größe von 8060 Bytes überschreitet. Bei der Zeilen- und Seitenkomprimierung wird die Überprüfung der Zeilengröße immer durchgeführt.
Bei der Komprimierung gelten die folgenden zwei Regeln:
 - Ein Update für einen Datentyp mit fester Länge muss immer erfolgreich sein.
 - Die Deaktivierung der Datenkomprimierung muss immer erfolgreich sein. Selbst wenn die komprimierte Zeile auf die Seite passt, das heißt wenn ihre Größe weniger als 8060 Bytes beträgt, verhindert SQL Server Updates, die nicht in die unkomprimierte Zeile passen würden.
- Bei Angabe einer Liste mit Partitionen kann der Komprimierungstyp für einzelne Partitionen auf ROW, PAGE oder NONE gesetzt werden.
Ohne Angabe einer Liste mit Partitionen wird für alle Partitionen die in der Anweisung angegebene Datenkomprimierungseigenschaft festgelegt. Bei Erstellung einer Tabelle oder eines Indexes wird die Datenkomprimierung auf NONE festgelegt, falls nicht anders angegeben. Bei Änderung einer Tabelle wird die vorhandene Komprimierung beibehalten, falls nicht anders angegeben.
- Wenn eine Partitionsliste bzw. eine Partition außerhalb des zulässigen Bereichs angeben wird, wird ein Fehler generiert.
- Nicht gruppierte Indizes erben die Komprimierungseigenschaft der Tabelle nicht! In diesem Fall müssen Sie die Komprimierungseigenschaft explizit festlegen, um die Indizes zu komprimieren. Standardmäßig wird die Komprimierungseinstellung bei Erstellung eines Indexes auf NONE festgelegt.
- Wenn ein gruppierter Index auf einem Heap erstellt wird, erbt der gruppierter Index den Komprimierungsstatus des Heaps, sofern kein anderer Komprimierungsstatus angegeben wird.

- Wenn ein Heap zur Komprimierung auf Seitenebene konfiguriert wird, erfolgt die Komprimierung auf Seitenebene für die Seiten ausschließlich mit folgenden Methoden:
 - Die Daten werden mit der BULK INSERT- Syntax eingefügt.
 - Die Daten werden mit INSERT INTO ... eingefügt. WITH (TABLOCK)- Syntax.
 - Eine Tabelle wird durch Ausführung von ALTER TABLE ... erneut erstellt. REBUILD- Anweisung mit der PAGE- Komprimierungsoption.
- Neue Seiten, die in einem Heap als Teil von DML- Vorgängen zugeordnet sind, verwenden die PAGE- Komprimierung erst nach der Neuerstellung des Heaps. Erstellen Sie den Heap neu, indem Sie die Komprimierung entfernen und neu anwenden oder indem Sie einen gruppierten Index erstellen und entfernen.
- Zur Änderung der Komprimierungseinstellung für einen Heap müssen alle nicht gruppierten Indizes der Tabelle neu erstellt werden, sodass sie auf die neuen Zeilenpositionen im Heap zeigen.
- Die Speicherplatzanforderungen zur Aktivierung bzw. Deaktivierung der Zeilen- oder Seitenkomprimierung, entsprechen den Anforderungen zur Indexerstellung bzw. Indexneuerstellung. Für partitionierte Daten kann der erforderliche Speicherplatz reduziert werden, indem die Komprimierung für die Partitionen einzeln aktiviert bzw. deaktiviert wird.
- Um den Komprimierungsstatus der Partitionen in einer partitionierten Tabelle zu ermitteln, fragen Sie die DATA_COMPRESSION- Spalte der **sys.partitions**- Katalogsicht ab.
- Bei der Indexkomprimierung können Seiten auf Blattebene sowohl mit der Zeilen- als auch mit der Seitenkomprimierung komprimiert werden. Für Seiten auf Nichtblattebene erfolgt keine Seitenkomprimierung.

12.5.9.2 Partitionierte Tabellen und Indizes

Beachten Sie die folgenden Überlegungen, wenn Sie die Datenkomprimierung mit partitionierten Tabellen und Indizes verwenden:

- **Teilen eines Bereichs**
Wenn Sie Partitionen mit der ALTER PARTITION- Anweisung teilen, erben die geteilten Partitionen das Datenkomprimierungsattribut der ursprünglichen Partition.
- **Zusammenführen eines Bereichs**
Wenn Sie zwei Partitionen zusammenführen, erbt die zurückgegebene Partition das Datenkomprimierungsattribut der Zielpartition.
- **Wechseln von Partitionen**
Zum Wechseln einer Partition muss die Datenkomprimierungseigenschaft der Partition mit der Komprimierungseigenschaft der Tabelle übereinstimmen.
- **Löschen eines partitionierten gruppierten Indexes**
Beim Löschen eines gruppierten Indexes behalten die entsprechenden Heappartitionen ihre Einstellung für die Datenkomprimierung bei, sofern nicht das Partitionierungsschema geändert wird. Wenn das Partitionierungsschema geändert wird, werden alle Partitionen neu erstellt und erhalten einen unkomprimierten Status. Um einen gruppierten Index zu löschen und das Partitionierungsschema zu ändern, sind folgende Schritte erforderlich:
 1. Löschen Sie den gruppierten Index.
 2. Ändern Sie die Tabelle mit der Option ALTER TABLE ... REBUILD, zur Angabe der Komprimierungsoption.

OFFLINE kann ein gruppierter Index sehr schnell gelöscht werden, da lediglich die oberen Ebenen des gruppierten Indexes entfernt wird. Wenn ein gruppierter Index ONLINE gelöscht wird, muss SQL Server den Heap zweimal neu erstellen, einmal für Schritt 1 und einmal für Schritt 2.

Ein Benutzer der Tabellen oder Indizes komprimieren will benötigt die ALTER- Berechtigung für die Tabelle oder Sicht. Er muss ein Mitglied der festen Serverrolle **sysadmin** bzw. der festen Datenbankrollen **db_ddladmin** und **db_owner** sein.

Beispiele:

Durch Zeilenkomprimierung einen Index für einen nichtpartitionierte Tabelle erstellen.

```
create index aname_ind on artikel(aname)
with (data_compression = ROW);
```

Einen gruppierten Index für eine partitionierte Tabelle mithilfe von Zeilenkomprimierung für alle Partitionen des Indexes erstellen. Die Anweisung bezieht sich auf die im Kapitel "**Partitionierte Tabellen**" erstellte partitionierte Tabelle "lieferung".

```
create index lmenge_ind on lieferung(lmenge)
with (data_compression = ROW);
```

Einen gruppierten Index für eine partitionierte Tabelle erstellen, wobei die Seitenkomprimierung für Partition 1 des Index und die Zeilenkomprimierung für die Partitionen 2 bis 3 des Index verwendet werden soll. Die Anweisung bezieht sich auf die im Kapitel "**Partitionierte Tabellen**" erstellte partitionierte Tabelle "lieferung".

```
create index lmenge_ind on lieferung(lmenge)
with (data_compression = PAGE on partitions(1),
      data_compression = ROW on partitions(2 to 3));
```

12.5.10 Indizes für berechnete Spalten

Berechnete Spalten einer Tabelle können unter bestimmten Umständen indiziert werden.

Folgende Voraussetzungen müssen erfüllt sein:

- Die berechnete Spalte ist deterministisch und darf nicht vom Datentyp text, ntext oder image sein.
- Die Option ANSI NULL ist bei der Tabellenerstellung auf ON gesetzt.
- Kein Spaltenverweis ruft Daten aus mehreren Zeilen ab. Beispielsweise hängen Aggregatfunktionen wie SUM oder AVG von Daten aus mehreren Zeilen ab und der Ausdruck der berechneten Spalte wäre dadurch nicht deterministisch. Die ISDETERMINISTIC- Eigenschaft der COLUMNPROPERTY- Funktion stellt fest ob ein Ausdruck deterministisch ist oder nicht.
- Die Verbindung die den Index erstellt und alle Verbindungen die auf die indizierte Tabelle eine INSERT-, UPDATE- oder DELETE- Anweisung ausführen, müssen die SET- Optionen ANSI_NULLS, ANSI_PADDING, ANSI_WARNINGS, ARITHABORT, CONCAT_NULL_YIELDS_NULL und QUOTED_IDENTIFIER auf ON stehen haben und die SET- Option NUMERIC_ROUNDABORT auf OFF eingestellt haben.

12.5.11 Indizierte Sichten

Wenn in Abfragen häufig auf Sichten verwiesen wird, können Sie die Leistung verbessern, indem Sie einen eindeutigen gruppierten Index für die Sicht erstellen. Wird ein eindeutiger, gruppiert Index für eine Sicht erstellt, wird die Sicht ausgeführt, und das Resultset wird genauso wie eine Tabelle mit einem gruppierten Index in der Datenbank gespeichert.

Die Wartung indizierter Sichten kann komplexer sein als die Wartung von Indizes für Basistabellen. Sie sollten nur dann Indizes für Sichten erstellen, wenn die erhöhte Geschwindigkeit beim Abrufen von Ergebnissen den zusätzlichen Aufwand aufwiegt, der zum Vornehmen von Änderungen erforderlich ist. Dies gilt normalerweise für Sichten, die über relativ statische Daten verknüpft sind, viele Zeilen verarbeiten und auf die in vielen Abfragen verwiesen wird.

Anforderungen:

- Die Verbindung die den Index erstellt und alle Verbindungen die auf die indizierte Tabelle eine INSERT-, UPDATE- oder DELETE- Anweisung ausführen, müssen die SET- Optionen ANSI_NULLS, ANSI_PADDING, ANSI_WARNINGS, ARITHABORT, CONCAT_NULL_YIELDS_NULL und QUOTED_IDENTIFIER auf ON stehen haben und die SET- Option NUMERIC_ROUNDABORT auf OFF eingestellt haben.
- Die Option ANSI_NULLS muss für die Ausführung aller CREATE TABLE- Anweisungen, die Tabellen erstellen, auf die die Sicht verweist, auf ON festgelegt sein.
- Die Sicht darf nicht auf andere Sichten verweisen, sondern nur auf Basistabellen.
- Alle Basistabellen, auf die die Sicht verweist, müssen in derselben Datenbank wie die Sicht vorhanden sein und denselben Besitzer wie die Sicht haben.
- Auf jeden Fall muss die SQL- Aggregatfunktion COUNT_BIG mit einbezogen werden, auch wenn sie nicht benötigt wird.
- Es werden Sichten indiziert die Aggregatfunktionen verwenden.
- Die Sicht muss mit der Option SCHEMABINDING erstellt werden. SCHEMABINDING bindet die Sicht an das Schema der zugrunde liegenden Basistabellen, benutzerdefinierte Funktionen, auf die in der Sicht verwiesen wird, müssen mit der Option SCHEMABINDING erstellt werden.
- Auf Tabellen und benutzerdefinierte Funktionen muss mit zweiteiligen Namen verwiesen werden. Ein-, drei- und vierteilige Namen sind nicht zulässig.

Die SELECT- Anweisung in der Sicht darf die folgenden Transact- SQL- Syntaxelemente nicht enthalten:

- Die Auswahlliste darf die Syntax * oder tab_name.* nicht zur Angabe von Spalten verwenden. Spaltennamen müssen explizit angegeben werden.
- Eine abgeleitete Tabelle, Rowsetfunktionen, UNION- Operator, Unterabfragen, OUTER JOIN oder SELF JOIN, TOP- Klausel, ORDER BY- Klausel, DISTINCT- Schlüsselwort, COUNT(*), Die Aggregatfunktionen AVG, MAX, MIN, STDEV, STDEVP, VAR oder VARP.

12.5.11.1 Erstellen von XML- Indizes

XML- Daten werden in Spalten mit dem Datentyp xml als LOBs (Large Objects) gespeichert. Diese Daten können bis zu 2 GB groß sein. Durch eine Indizierung dieser Spalten kann die Leistung von Abfragen auf XML- Daten erheblich gesteigert werden.

Beim Erstellen erstellt SQL Server eine gruppierte B- Struktur- Darstellung der Knoten in den XML- Daten und speichert diese in einer internen Tabelle. Der XML- Index wird mit dem Primärschlüssel gruppiert.

Nachdem ein Primärer XML- Index erstellt ist, können zusätzlich ein oder mehrere sekundäre XML- Indizes angelegt werden. Diese können über PATH, VALUE oder PROPERTY erstellt werden.

Bei der Verwendung ist folgendes zu beachten:

- Die Tabelle muss einen gruppierten Index besitzen, bevor der XML- Index erstellt werden kann.
- Dieser Index kann nicht mehr geändert werden sobald ein XML- Index vorhanden ist.
- Pro XML- Spalte kann ein primärer Index festgelegt werden.
- Kein XML- Index und normaler Index dürfen den gleichen Namen haben.
- Die Optionen ONLINE und IGNORE_DUP_KEY können nicht verwendet werden wenn ein XML- Index erstellt wird.

- Sichten mit XML- Spalten, Tabellenwertvariable mit XML- Spalte oder XML- Variable können nicht mit einem XML- Index indiziert werden.
- Wenn ein XML- Index erstellt wird und Datenänderungen auf diese Tabelle vorgenommen werden sollen, muss die Option ARITHABORT auf ON gestellt werden.

Die XML- Indizes werden in zwei Kategorien unterteilt:

- **Primärer Index**, es handelt sich um eine gruppierte B- Struktur- Darstellung der Knoten in den XML- Daten. Der erste Index für eine XML- Spalte muss ein primärer XML- Index sein. Durch einen primären Index wird die Abfrageleistung grundsätzlich verbessert.
- **Sekundärer Index**, es handelt sich um einen nicht gruppierten Index des primären XML- Indexes. Der primäre Index muss vorhanden sein um einen sekundären zu erstellen. Sie bieten je nach Art zusätzliche Leistungsverbesserungen. Es gibt drei Typen:
 - Pfadindizes (FOR PATH), sie verbessern die Leistung von Abfragen, die Pfade und Werte innerhalb eines XML- Dokuments suchen. Sie werden über die PATH- und NODE- Werte des primären XML- Index aufgebaut.
 - Eigenschaftenindizes (FOR VALUE), sie verbessern die Leistung von Abfragen, die nur Werte innerhalb eines XML- Dokuments suchen. Sie werden über die PATH- und VALUE- Werte des primären XML- Index aufgebaut.
 - Wertindizes (FOR PROPERTY), sie verbessern die Leistung von Abfragen, die Daten einer XML- Spalte zusammen mit weiteren Spalten der Tabelle abfragen. Sie werden über die NODE- und PATH- Werte des primären XML- Index aufgebaut.

Syntax:

```
create [primary] xml index index_name
on {tablename | viewname} (xml_spaltenname)
[using xml index xml_index_name [for {value | path | property}]]
[with
[pad_index = {on | off}]
[[], fillfactor = füllgrad]
[[], drop_existing = {on | off}]
[[], statistics_norecompute = {on | off}]
[[], sort_in_tempdb = {on | off}]
[[], allow_row_locks = {on | off}]
[[], allow_page_locks = {on | off}]
[[], maxdop = max_grad_an_parallelität]]
```

- [PRIMARY] XML
Erstellt einen XML- Index für eine angegebene XML- Spalte. Jede Tabelle kann bis zu 249 XML- Indizes haben. Wird PRIMARY angegeben, wird ein gruppierter Index mit dem gruppierten Schlüssel (besteht aus Gruppierungsschlüssel, Benutzertabelle und dem Bezeichner für einen XML- Knoten) erstellt.
- USING XML INDEX
Gibt den primären XML- Index an, der beim Erstellen deines sekundären XML- Index verwendet werden soll.
- FOR {VALUE | PATH | PROPERTY}
Gibt den Typ des sekundären Index an.

Beispiele:

1. Primärer XML- Index auf die Spalte "beschreibung" der Tabelle "artikel".

```
create primary xml index beschr_xml_ind on artikel(beschreibung);
```

2. Sekundärer XML- Index als XML- Pfadindex.

```
create xml index beschr_xmlpath_ind on artikel(beschreibung)
using xml index beschr_xml_ind
for path;
```

3. Sekundärer XML- Index als XML- Eigenschaftenindex.

```
create xml index beschr_xmlproperty_ind on artikel(beschreibung)
using xml index beschr_xml_ind
for property;
```

4. Sekundärer XML- Index als XML- Werteindex.

```
create xml index beschr_xmlvalue_ind on artikel(beschreibung)
using xml index beschr_xml_ind
for value;
```

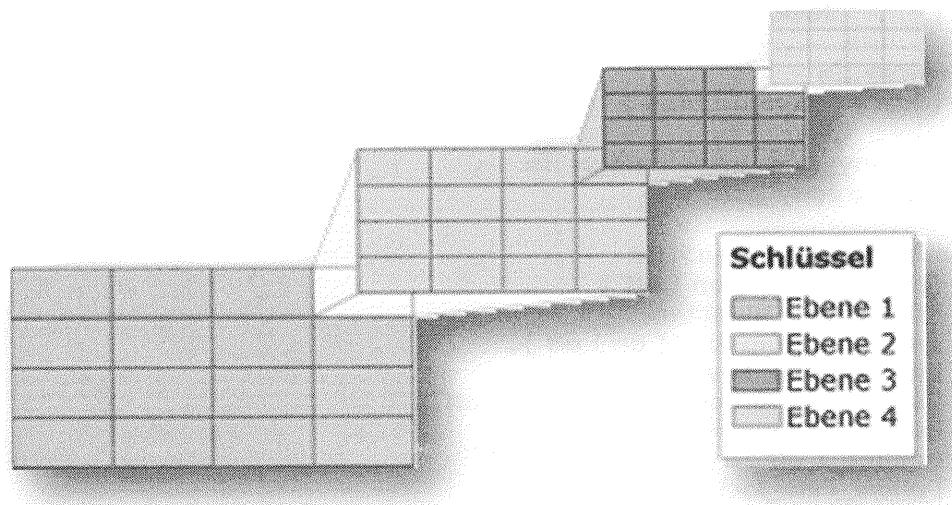
12.5.11.2 Räumliche Indizes

In SQL Server 2008 räumlichen Daten unterstützt. Hierunter fällt die Unterstützung des planaren räumliche-Daten- Typs GEOMETRY, der geometrische Daten (Punkte, Linien und Polygone) in einem euklidischen Koordinatensystem unterstützt. Der GEOGRAPHY- Datentyp stellt geografische Objekte auf einem Bereich der Erdoberfläche dar.

Räumliche Indizes werden über eine räumliche Spalte aufgebaut die entweder vom Datentyp GEOMETRY oder GEOGRAPHY sind. Jeder räumliche Index verweist auf einen endlichen Raum.

Räumliche Indizes werden mithilfe von B- Strukturen erstellt, das heißt, dass die Indizes die zweidimensionalen räumlichen Daten in der linearen Reihenfolge der B- Strukturen darstellen müssen. Bevor Daten in einen räumlichen Index eingelesen werden, implementiert SQL Server daher eine einheitliche hierarchische Zerlegung des Raums. Während der Indexerstellung wird der Raum in eine vier Ebenen umfassende uniforme Rasterhierarchie zerlegt. Diese Ebenen (Hemisphären) werden als Ebene 1 (die oberste Ebene), Ebene 2, Ebene 3 und Ebene 4 bezeichnet.

Die Abbildung zeigt eine uniforme Rasterhierarchie mit vier Ebenen.



Die Zerlegung wird in einem Prozess durchgeführt, der Mosaikbildung heißt.

Auf jeder nachfolgenden Ebene wird die ihr übergeordnete Ebene weiter zerlegt, sodass jede Zelle der übergeordneten Ebene ein vollständiges Raster der nächsten Ebene enthält.

Syntax:

```

create spatial index index_name
on <object> ( spatial_column_name )
{
[ using <geometry_grid_tessellation> ]
  with   ( <bounding_box> [ [,] <tesselation_parameters> [ ,...n ] ]
            [ [,] <spatial_index_option> [ ,...n ] ] )
| [ using <geography_grid_tessellation> ]
  [ with ( [ <tesselation_parameters> [ ,...n ] ]
            [ [,] <spatial_index_option> [ ,...n ] ] ) ]
}
[ on { filegroup_name | "default" } ];

<object> ::=
  [ database_name. [ schema_name ] . | schema_name. ] table_name

<geometry_grid_tessellation> ::=
  { geometry_grid }

<bounding_box> ::=
  bounding_box = ( {xmin, ymin, xmax, ymax}
  |      <named_bb_coordinate>, <named_bb_coordinate>,
  <named_bb_coordinate>, <named_bb_coordinate> ) )

<named_bb_coordinate> ::=
  { xmin = xmin | ymin = ymin | xmax = xmax | ymax=ymax }

<tesselation_parameters> ::=
  {grids=( {<grid_density> [,...n ]
  |      <density>,<density>,<density>,<density> }
  |      cells_per_object = n }

<grid_density> ::=
  { level_1 = <density>
  | level_2 = <density>
  | level_3 = <density>
  | level_4 = <density> }

<density> ::=
  { low | medium | high }

<geography_grid_tessellation> ::=
  { geography_grid }

<spatial_index_option> ::=
  {pad_index = { on | off }
  | fillfactor
  | sort_in_tempdb = { on | off }
  | ignore_dup_key = off
  | statistics_norecompute = { on | off }
  | drop_existing = { on | off }
  | online = off
  | allow_row_locks = { on | off }
  | allow_page_locks = { on | off }
  | maxdop = max_degree_of_parallelism}

```

12.5.11.2.1 Mosaikbildung

Wenn die Rasterhierarchie mit vier Ebenen fertig ist, wird jede Zeile mit räumlichen Daten gelesen und auf dem Raster abgebildet. Der Prozess beginnt auf Ebene 1 und bildet das räumliche Objekt auf die Rasterzellen ab, die das Objekt berührt. Der Satz der berührten Zellen wird dann im Index aufgezeichnet.

Sehr kleine Objekte berühren nur wenige Zellen innerhalb der Rasterhierarchie, große Objekte können dagegen sehr viele Zellen berühren. Um die Größe überschaubar zu halten ohne das Informationen zur Genauigkeit verloren gehen wendet der Mosaikbildungsprozess Regeln an, um die fertigen Aufgaben zu ermitteln, die in den Index geschrieben werden.

| | |
|--------------------------|--|
| Überlagerung | Falls ein Objekt eine Zelle vollständig bedeckt, wird die Zelle nicht beim Mosaikbildungsprozess berücksichtigt. |
| Zellen pro Objekt | Diese Regel erzwingt den Parameter CELLS_PER_OBJECT für den räumlichen Index in den Ebenen 2,3 und 4 der Rasterhierarchie. |
| Tiefste Zelle | Es werden nur die am weitesten unten liegenden Zellen aufgezeichnet, die beim Mosaikbildungsprozess markiert wurden. |

Wird eine Zelle durch diesen Prozess nicht berücksichtigt, werden keine Informationen werden keine Informationen zu den nachfolgenden Ebenen der Hierarchie über diese Zelle aufgezeichnet. Solange die ZEILEN_PRO_OBJECT- Regel nicht überschritten wurde und das Objekt eine Zelle nicht vollständig bedeckt, wird die Zelle dem Mosaikbildungsprozess unterworfen. Dabei wird der Teil des Objekts, das innerhalb der Zelle liegt, auf das Raster der nächsten Ebene der Hierarchie abgebildet, wo wiederum der Mosaikbildungsprozess angewendet wird. Das kann bis zur vierten Ebene passieren.

Die Zellen die auf der untersten Ebene der Mosaikbildung unterworfen wurden, definieren den Schlüssel des räumlichen Index für die Zeile.

12.5.11.2.2 Umgebende Felder

Soll eine Spalte mit GEOMETRY- Daten indiziert werden, muss eine weitere Option für räumliche Indizes definiert werden, das umgebende Feld (bounding box).

Der Parameter BOUNDING_BOX definiert die maximalen und minimalen x/y- Koordinaten, die beim Erstellen der Rasterhierarchie und bei der Mosaikbildung der Zeilen mit GEOMETRY- Daten berücksichtigt werden.

Alle Objekte oder Objektteile, die außerhalb des umgebenden Felds liegen, werden für die Indexbildung nicht berücksichtigt. Wenn die Grenzen des umgebenden Feldes festgelegt werden, müssen Werte gewählt werden, die den größten Teil der Objekte einschließen, die innerhalb der Tabelle indiziert werden sollen.

12.6 Die ALTER INDEX- Anweisung

Mit der ALTER INDEX- Anweisung kann ein Index oder alle Indizes einer Tabelle oder Sicht geändert werden. Dabei kann der Index neu organisiert (Defragmentierung) werden, die Einstellungsoptionen geändert werden sowie der Index deaktiviert oder wieder aktiviert wird.

Mit dieser Anweisung ist es auch möglich den Index einer Primärschlüssel- Einschränkung oder einer Unique- Einschränkung zu deaktivieren.

Alter Index kann nicht verwendet werden, um einen Index neu zu partitionieren oder ihn in eine andere Dateigruppe zu verschieben. Außerdem kann er nicht verwendet werden um die Indexdefinition, wie z. B. das Hinzufügen oder Löschen von Spalten oder das Ändern der Spaltenreihenfolge, zu ändern. Um den Index dahingehend neu zu definieren muss die CREATE INDEX- Anweisung mit der Option DROP_EXISTING verwendet werden.

Syntax:

```
alter index {index_name | all}
on {tab_name | view_name}
{ rebuild
  [[with (
    [pad_index = {on | off}]
    [,] fillfactor = fillfactor]
    [,] sort_in_tempdb = {on | off}]
    [,] ignore_dup_key = {on | off}]
    [,] statistics_norecompute = {on | off}]
    [,] online = {on | off}]
    [,] allow_row_locks = {on | off}]
    [,] allow_page_locks = {on | off}]
    [,] maxdop = max_degree_of_parallelism]
    [,] data_compression = {none | row | page}
      on partitions({partitions_nummer | bereich} [...])
  )
  ]
  | [ partition = partition_number
    [with
      (
        [sort_in_tempdb = {on | off}]
        [,] maxdop = max_degree_of_parallelism]
      )
    ]
    ]
  ]
  | disable
  | reorganize
    [partition = partition_number]
    [with (lob_compaction = {on | off})]
  | set (
    [allow_row_locks= {on | off}]
    [,]allow_page_locks = {on | off}]
    [,]ignore_dup_key = {on | off}]
    [,]statistics_norecompute = {on | off}]
  )
}
```

Beispiele:

1. Neuerstellen aller Indizes einer Tabelle und Angeben von Optionen.

```
alter index all on lieferung
rebuild
with (fillfactor = 65, sort_in_tempdb = on, statistics_norecompute = on);
```

2. Deaktivieren und aktivieren eines Indexes.

```
alter index lstadt_ind on lieferant disable;
alter index lstadt_ind on lieferant rebuild;
```

3. Deaktivieren und aktivieren eines Primärschlüssel- Index.

```
alter index lnr_ps on lieferant disable;
```

Das Ergebnis gibt eine Warnmeldung zurück, dass durch das Deaktivieren des Primärschlüssels gleichzeitig der Fremdschlüssel, der auf den Primärschlüssel verweist, deaktiviert wurde. Darum muss nachdem man den Primärschlüssel- Index wieder aktiviert hat auch der Fremdschlüssel wieder aktiviert werden.

```
alter index lnr_ps on lieferant rebuild;
go
alter table lieferung check constraint lnr_fs;
```

12.7 Indexinformationen abrufen

Es existieren vier Möglichkeiten um Informationen über Indizes abzurufen. Es steht dafür das SQL Server Management Studio (Objektexplorer, Eigenschaftenfenster, Berichte), gespeicherte Systemprozeduren, Katalogsichten und Systemfunktionen zur Verfügung.

12.7.1 Gespeicherte Systemprozeduren

| | |
|--------------|---|
| sp_helpindex | gibt Details der für eine bestimmte Tabelle erstellten Indizes zurück. |
| sp_help | gibt ebenfalls Informationen zu den Indizes einer Tabelle zurück und zusätzlich ausführliche Informationen zur angegebenen Tabelle. |

12.7.2 Katalogsichten

| | |
|-------------------|--|
| sys.indexes | Enthält Informationen zum Indextyp, Dateigruppe oder Partitionsschema- ID sowie die aktuelle Einstellung der Indexoptionen. |
| sys.index_columns | Enthält die Spalten- ID, Position innerhalb des Index, Typ (Schlüssel oder Nichtschlüssel) und Sortierreihenfolge. |
| sys.stats | Enthält die dem Index zugeordneten Statistiken und die Angabe ob es sich um eine automatische oder benutzerdefinierte Statistik handelt. |
| sys.stats_columns | Zeigt die zugeordnete Spalten- ID für die Statistik. |
| sys.xml_indexes | XML- Indextyp (primär oder sekundär) sowie Beschreibung |

12.7.3 Systemfunktionen

| | |
|-----------------------------------|---|
| sys.dm_db_index_physical_stats | Indexgröße und Fragmentierungsstatistik |
| sys.dm_db_index_operational_stats | Aktueller Index und E/A- Statistik. |
| sys.dm_db_index_usage_stats | Statistik zur Indexverwendung nach Abfragetyp. |
| indexkey_property | Indexspaltenposition innerhalb des Indexes und Spaltenreihenfolge (ASC oder DESC) |

| | |
|---------------|--|
| indexproperty | Indextyp, Anzahl der Ebenen und aktuelle Einstellung der Indexoptionen, die in den Metadaten gespeichert sind. |
| index_col | Name der Schlüsselspalte des angegebenen Indexes |

12.7.4 Indexfragmentierung

Eine Fragmentierung der Daten tritt dann auf, wenn Daten geändert werden. SQL Server verwaltet den Index zwar selbständig durch Seitenteilung und Seitenzusammenlegung, aber trotzdem kann es Unterbelegungen von Speicherseiten kommen. Es können zwei Arten von Fragmentierung auftreten.

Die interne Fragmentierung ist eine ineffiziente Nutzung von Seiten innerhalb eines Indexes, da jede Seite eine größere als die real gespeicherte Datenmenge enthalten kann. dadurch kommt es zu einem höheren Aufkommen an logischen und physischen E/A- Vorgängen und es wird mehr Arbeitsspeicher benötigt. Die Abfrageleistung bei Lesevorgängen kann beeinträchtigt werden. Dagegen kann die Fragmentierung bei Einfügungen auch nützlich sein da die Wahrscheinlichkeit der Seitenteilung geringer wird.

Die externe Fragmentierung ist eine ineffiziente Nutzung von Seiten innerhalb eines Indexes, da die logische Reihenfolge falsch ist. Dadurch wird der Festplattenzugriff auf die Daten, durch zusätzliche Lesekopfbewegungen, verlangsamt.

Um die Fragmentierung eines Indexes zu ermitteln steht die dynamische Verwaltungsfunktion **sys.dm_db_index_physical_stats** zur Verfügung. Man kann sich die Fragmentierung eines bestimmten Indexes, aller Indizes einer Tabelle oder indizierten Sicht, aller Indizes in der Datenbank oder aller Indizes aller Datenbanken anzeigen lassen. Die Information über die Fragmentierung steht in der Spalte **avg_fragmentation_in_percent**.

Im SQL Server Management Studio können diese Informationen über das Eigenschaftenfenster des betreffenden Index angezeigt werden.

Beispiel:

```
select a.index_id, name, avg_fragmentation_in_percent
from sys.dm_db_index_physical_stats(db_id('standard'), object_id('lieferung'),
null, null, null) as a join sys.indexes as b
on a.object_id = b.object_id and a.index_id = b.index_id;
```

Die Entscheidung für das Neuorganisieren bzw. das Neuerstellen des Indexes um Fragmentierungen zu entfernen ist abhängig vom Fragmentierungsgrad.

Als Richtlinie gilt bei einem Grad <= 30% eine Neuorganisation und bei mehr als 30 % das Neuerstellen des Indexes.

Dafür wird die ALTER INDEX- Anweisung eingesetzt.

Beispiel:

Neuorganisation eines Index.

```
alter index lnr_ps on lieferant reorganize;
```

Neuerstellen eines Index.

```
alter index lnr_ps on lieferant rebuild;
```

12.8 Index löschen

Nachdem sie einen Index gelöscht haben steht der Speicherplatz der von ihm belegt wurde wieder zur Verfügung.

Die DROP INDEX- Anweisung ist nicht für Indizes anwendbar, die durch das Definieren von PRIMARY KEY- oder UNIQUE- Einschränkungen erstellt wurden. Indizes von Systemtabellen können nicht gelöscht werden. Diese müssen durch die ALTER TABLE- Anweisung entfernt werden.

Syntax:

```
drop index tabellenname.indexname;
```

12.9 Volltextindizierung

Damit Anwendungen effizient mit unstrukturierten Daten umgehen können, die in FILESTREAM-, XML- und großen Zeichenspalten gespeichert sind, sollten Sie Volltextindizes erstellen.

Damit ist es möglich große Mengen unstrukturierter Daten sehr schnell abzufragen. Beispielsweise müssen Sie alle Autoren und ihre Werke suchen in denen etwas zu einem bestimmten Thema steht.

Die Volltextsuche in SQL Server 2012 unterscheidet sich erheblich von der in früheren Versionen. Da befanden sich die Volltextindizes in Volltextkatalogen, die Dateigruppen angehörten, physische Pfade aufwiesen und als Datenbankdateien behandelt wurden. In SQL Server 2012 wird ein Volltextkatalog logisch als virtuelles Objekt dargestellt, das nicht auf eine Datenbankdatei mit einem physischen Pfad, sondern auf eine Gruppe von Volltextindizes verweist.

Es wird nicht mehr der Dienst "Microsoft Search" verwendet, sondern das Microsoft Volltextsuchmodul für SQL Server, das alle Indizierungs- und Abfragefunktionen übernimmt und vollständig in den Abfrageprozessor integriert ist. Ein Filterdaemonhost (fdhost.exe) ersetzt den Filterdaemonprozess (msftefd.exe). Er lädt die für die Indizierung und Abfrage verwendeten Komponenten, darunter die Wörtertrennung zum Auffinden von Wortgrenzen, die Wortstammerkennung zum Konjugieren von Verben und die Dokumentfilter zum Extrahieren von Textinformationen.

Um solche Daten zu indizieren müssen folgende Schritte beachtet werden:

- Erstellen eines Volltextkatalogs für die Datenbank
- Erstellen eines eindeutigen Index für die Tabelle
- Erstellen eines Volltextindex für einzelne Tabellen oder Sichten
- Konfigurieren des Auffüllmodus für den Volltextindex
- Verwalten der Volltextindexauffüllung

In SQL Server, seit der Version 2005, sind standardmäßig alle von Benutzern erstellten Datenbanken volltextfähig. Weiterhin wird eine einzelne Tabelle automatisch für die Volltextindizierung aktiviert, sobald ein Volltextindex für die Tabelle erstellt wird und dem Index eine Spalte dieser Tabelle hinzugefügt wird. Eine Tabelle wird für die Volltextindizierung automatisch deaktiviert, sobald die letzte Spalte der Tabelle aus dem Volltextindex entfernt wird. Die Volltextindizierung für eine Tabelle kann aber auch manuell aktiviert oder deaktiviert werden. Verwenden Sie dazu das SSMS oder die "ALTER FULLTEXT INDEX"- Anweisung.

Jede Instanz nutzt ihren eigenen Satz von Wörtertrennung, Wortstämmen und Filtern und ist damit nicht mehr von der Betriebssystemversion dieser Komponenten abhängig.

Die Voiltextsuche in SQL Server 2012 unterstützt 40 Sprachen. Eine Liste können Sie sich mit der Katalogsicht "sys.fulltext_languages" anzeigen lassen.

12.9.1 Volltextkataloge

Bevor Sie einen Volltextindex aufbauen können benötigen Sie eine Speicherungsstruktur, einen sogenannten Volltextkatalog. Da Volltextindizes eine, im Gegensatz zu relationalen Indizes, besondere interne Struktur haben, wird dafür ein separates Speicherungsformat, der Volltextkatalog, benötigt. Jeder Volltextkatalog enthält einen oder mehrere Volltextindizes und wird als Teil der Datenbank behandelt. Jede Datenbank muss einen eigenen Volltextkatalog besitzen.

Man einen Zeitplan für das Auffüllen des Katalogs festlegen oder ihn manuell Auffüllen. Bei der Auffüllung werden seine Volltextindizes aktualisiert und so die Genauigkeit der Suchergebnisse sichergestellt.

Methoden zum Auffüllen:

1. Vollständige Auffüllung:
Das Datenbankmodul erstellt Indexeinträge für alle Zeilen in allen Tabellen oder Sichten, für die der Volltextkatalog gilt. Das wird in der Regel beim Erstellendes Katalogs oder beim Aktualisieren des Katalogs durchgeführt.
2. Inkrementelle Auffüllung:
Das Datenbankmodul ändert die Indexeinträge nur für die Zeilen, die seit der letzten Auffüllung hinzugefügt, geändert oder gelöscht wurden. Diese Methode kann nur für Tabellen oder Sichten verwendet werden, die über eine TIMESTAMP- Spalte verfügen.
3. Auffüllen nach Update:
Wird vom Datenbankmodul verwendet in Verbindung mit der Änderungsnachverfolgung. Die Änderungsverfolgung erlaubt der Volltextsuche, Änderungen im Katalog oder in Sichten zu verfolgen, sodass der Katalog automatisch oder manuell aktualisiert werden kann.

Syntax:

```
create fulltext catalog katalog_name
[on filegroup filegroup ]
[with accent_sensitivity = {on|off}]
[as default]
[authorization owner_name ]
```

-FILEGROUP- Klausel: legt fest in welche Dateigruppe alle Volltextindizes gespeichert werden sollen.

-ACCENT_SENSITIVITY: legt fest ob das Volltextmodul diakritische Zeichen (dazu gehören Zeichen wie Punkte, Striche, Häkchen oder kleine Kreise) berücksichtigt, wenn es einen Volltextindex aufbaut oder abfragt.

Beispiel:

```
create fulltext catalog standard_db_texte
on filegroup dokumente
as default;
```

Ein Benutzer der einen Volltextkatalog erstellen soll muss über die CREATE FULLTEXT CATALOG- Berechtigung in der Datenbank verfügen oder ein Mitglied der festen Datenbankrolle db_owner oder db_ddadmin sein.

12.9.2 Volltextindizes

Nachdem ein Volltextkatalog erstellt worden ist, können nun Volltextindizes aufgebaut werden.

Volltextindizes können für Spalten mit den Datentypen char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary und varbinary(max) erstellt werden. Beim Erstellen eines Volltextindex für image-, varbinary- oder varbinary(max)- Spalten muss eine Typspalte mit der jeweiligen Dateierweiterung (.doc, .pdf, .xls, ...) für das entsprechende Dokument angegeben werden. SQL Server 2008 enthält im Lieferumfang 50 Filter die den Umgang mit den gebräuchlichen Dokumenttypen beherrschen, weitere können von der Downloadseite bei Microsoft oder Drittherstellern heruntergeladen werden. Damit diese dann durch SQL Server verwendet werden können müssen sie erst einmal für das DBMS geladen werden.

```
exec sp_fulltext_service 'load_os_resources', 1;
```

Das Volltextindizierungsmodul greift auf Hilfsdienste zurück, etwa für Worttrennungen und Wortstammerkennung. Dazu müssen diese Komponenten die jeweilige Dokumentsprache kennen.

Die Liste der Wörter wird anhand einer Liste häufiger Wörter gefiltert, der sogenannten "Stopwörter". Das verhindert, dass der Index durch eine riesige Zahl von Wörtern aufgeblättert wird.

Volltextindizes können über mehrere Spalten einer Tabelle erstellt werden, jedoch kann nur ein Volltextindex pro Tabelle oder Sicht existieren. Ein Volltextindex kann 1024 Spalten besitzen.

Syntax:

```
create fulltext index on tab_name
[({spalten_name [ type column spalten_name][language
language_term]}[,...n])]
key index index_name
[on <catalog_filegroup_option>]
[with [(] <with_option> [,...n][)]];

<catalog_filegroup_option>::=
{ fulltext_catalog_name
| ( fulltext_catalog_name, filegroup filegroup_name )
| ( filegroup filegroup_name, fulltext_catalog_name )
| ( filegroup filegroup_name ) }
```



```
<with_option>::=
{ change_tracking [ = ] { manual | auto | off [, no population ] }
| stoplist [ = ] { off | system | stoplist_name } }
```

Argumente:

| | |
|---------------------------|--|
| spalten_name: | Name der Spalte die den Volltextindex enthält. Wenn keine Spalte angegeben wird, werden alle Spalten der Tabelle in dem Volltextindex aufgenommen. Es kann sich hierbei nur um Spalten mit den Datentypen char, varchar, nchar, nvarchar, text, ntext, xml, varbinary oder varbinary(max) handeln. |
| type column spalten_name: | Name der Spalte die den Dokumenttyp (.doc, .pdf, .xls, usw.) für ein Dokument, welches in einer Spalte vom Datentyp varbinary, varbinary(max) oder image enthalten ist, angibt. |

| | |
|------------------------|---|
| fulltext_catalog_name: | Name des Volltextkatalogs den der Volltextindex verwenden soll. Diese Angabe ist optional, wird kein Katalogname angegeben, wird der Index im Standardkatalog erstellt. |
| key index index_name: | Name einer eindeutigen Schlüsselspalte der Tabelle die keine NULL- Marken (PS- Spalte) zulässt. Empfohlen ist eine Spalte vom Datentyp integer. |
| change_tracking: | Gibt an, ob abgedeckte Änderungen (Aktualisierungen, Löschungen oder Einfügungen) an Tabellenspalten an den Volltextindex weitergegeben werden sollen. |

Benutzer müssen über die REFERENCES- Berechtigung für den Volltextkatalog und die ALTER- Berechtigung für die Tabelle oder indizierte Sicht verfügen, oder Mitglied der festen Serverrolle sysadmin oder der festen Datenbankrolle db_owner oder db_ddladmin sein. Wird eine Stopliste angegeben, benötigen die Benutzer REFERENCES- Berechtigungen für die Stopliste.

Beispiel:

Angenommen, die Tabelle Lieferant enthält eine Spalte "lebenslauf" vom Datentyp varbinary(max) und eine weitere Spalte "doktyp" vom Datentyp nvarchar. Es soll ein Volltextindex auf die Spalte "lebenslauf" erstellt werden, welcher aber erst zu einem späteren Zeitpunkt aufgefüllte werden soll und der keine Änderungsnachverfolgung durchführen soll.

```
create fulltext index on dbo.lieferant
(lebenslauf type column doktyp language 1031)
key index lnr_ps
on standard_db_texte
with change_tracking off, no population;
```

Auffüllen des Index.

```
alter fulltext index on dbo.lieferant set change_tracking auto;
```

13 Programmieren mit Transact- SQL

Über die Sprach Konstrukte DDL, DML und DCL hinaus bietet T- SQL einfache Elemente an, wie sie auch in richtigen Entwicklersprachen vorkommen. Allerdings ist der Sprachumfang nicht gerade umfangreich und erinnert mehr an eine einfache Batch- Programmierung. Aber, auf Grund des einfachen Aufbaus lässt sich die Sprache schnell analysieren, optimieren und übersetzen. T- SQL wird in erster Linie dazu benutzt um Skripte, einfache Prozeduren, Funktionen und Trigger zu schreiben.

13.1 Sprachelemente

Das sind Befehle zur Speicherung von Werten in Variablen und zur Steuerung des Programmablaufs.

13.1.1 Skripte und Batches

Skriptdateien enthalten ein Stück Programm und sind als Textdatei (*.sql) auf der Festplatte gespeichert. Sie finden oft Anwendung wenn es um die Administration eines Servers oder einer Datenbank geht.

Skripte werden im SQL Management Studio ausgeführt oder mit dem Kommandozeilen-Werkzeug **sqlcmd**.

Ein Skript besteht aus einem oder mehreren Stapel (Batches). Die Befehle eines Stapels werden gemeinsam analysiert, optimiert und ausgeführt.

Stapelprogramme werden mit einem **GO-** Befehl (Stapeltrennzeichen) beendet. Das **sqlcmd**- Dienstprogramm sowie das **osql**- Dienstprogramm verwenden den GO- Befehl, um das Ende eines Batches anzuzeigen.

GO ist keine Transact- SQL Anweisung, sondern zeigt den Dienstprogrammen lediglich an, wie viele SQL- Anweisungen in einem Batch enthalten sein sollen.

In SQL Server Management Studio werden alle Transact-SQL- Anweisungen von einem GO- Befehl bis zum nächsten in die an SQLExecDirect gesendete Zeichenfolge eingefügt.

Beispiel:

Anweisung wie sie im SQL Server Management Studio eingegeben wurde.

```
select @@version;
set nocount on;
go
```

SQL Server Management Studio führt einen dem Folgenden entsprechenden Schritt aus.

```
sqlexecdirect(hstmt,"select @@version set nocount on",sql_nts);
```

Regeln:

- CREATE- Befehle gehören an den Anfang eines Batches.
- CREATE DEFAULT-, CREATE FUNCTION-, CREATE PROCEDURE-, CREATE RULE-, CREATE SCHEMA-, CREATE TRIGGER- und CREATE VIEW- Anweisungen können nicht mit anderen SQL- Anweisungen in einem Batch verwendet werden.
- Objekte können innerhalb eines Batches nicht gelöscht werden und gleich wieder neu erstellt werden. Dafür sind mindestens zwei getrennte Stapel notwendig.

- Der für einen Batch erstellte Ausführungsplan kann nicht auf Variablen verweisen, die in einem anderen Batch deklariert werden.
- Kommentare müssen in einem Batch beginnen und enden.
- Es ist nicht möglich, im selben Batch eine Tabelle zu ändern und dann auf die neuen Spalten zu verweisen.
- Falls an erster Stelle des Stapels eine gespeicherte Prozedur ausgeführt wird, ist das Schlüsselwort EXECUTE nicht zwingend erforderlich.
- Eine Transact-SQL-Anweisung kann nicht dieselbe Zeile wie ein GO-Befehl belegen. Allerdings kann die Zeile Kommentare enthalten.

Hinweis: Die meisten Laufzeitfehler beenden die aktuelle Anweisung und die im Batch darauf folgenden Anweisungen.
Einige Laufzeitfehler, wie z. B. Einschränkungsverletzungen, beenden nur die aktuelle Anweisung. Alle verbleibenden Anweisungen im Batch werden ausgeführt.

Es ist möglich einem Skript zur Laufzeit Werte von Skriptvariablen zu übergeben. Dadurch kann ein Skript direkt aus der Eingabeaufforderung (cmd) heraus gestartet werden. Es gibt **zwei Varianten**, die eine mit dem **setvar**- Befehl und die andere wie im nachfolgenden Beispiel dargestellt.

Beispiel:

Es wurde folgendes Skript erstellt und in einem Verzeichnis als **testskript.sql** abgelegt.

```
use standard
select $(sp_name)
from $(tabelle);
go
```

Der Aufruf des Skripts und die Übergabe von Werten erfolgt mit dem nachfolgenden Befehl.

```
sqlcmd -S ersql -E -v sp_name = "Inr" tabelle = "lieferant" -i c:\beispiel\testskript.sql
```

13.1.2 Anweisungsblöcke

Anweisungsblöcke sind eine Anzahl von SQL- Anweisungen die zu einem logischen Codeblock zusammengefasst werden. Anweisungsblöcke werden in Verbindung mit vorhandenen Kontrollstrukturen IF und WHILE eingesetzt.

Syntax:

```
begin
    {SQL Anweisung | SQL Anweisungen}
end
```

13.1.3 Kommentare

Zu einem Programmtext gehören selbstverständlich auch beschreibende Kommentare. In T-SQL gibt es Zeilenkommentare und Blöckkommentare.

Mit Blockkommentaren lassen sich sehr einfach Programmteile auskommentieren. dabei sollte der GO- Befehl nicht mit eingeschlossen werden, da es zu unerwarteten Fehlermeldungen führen kann.

Syntax:**Zeilenkommentar**

```
--      Kommentar
```

Blockkommentar

```
/*
      mehrzeiliger Kommentar
*/
```

13.1.4 Meldungen

Wenn bei der Ausführung eines Skripts, einer gespeicherten Prozedur oder einer benutzerdefinierten Funktion unerwartete Ereignisse oder Fehler auftreten, dann ist es günstig dies an den ausführenden Client zurückzugeben.

13.1.4.1 PRINT

Der Befehl PRINT generiert Meldungen die vom Client ausgewertet werden können. Im SQL-Editor werden die Meldungen im Register "Meldungen" angezeigt.

Das Abfangen von PRINT generierten Meldungen auf Clientanwendungen ist von der verwendeten Datenschnittstelle abhängig.

Syntax:

```
print {'msg_str' | @local_variable | string_expr}
```

Für die Ausgabe mit Print sind nur Zeichenfolgen zulässig. Das bedeutet dass hier sehr häufig mit den Funktionen CAST und CONVERT gearbeitet wird.

13.1.4.2 RAISERROR

RAISERROR kann auch zum Zurückgeben von benutzerdefinierten Fehlermeldungen verwendet werden. RAISERROR hat im Vergleich zu PRINT die folgenden Vorteile:

- RAISERROR unterstützt das Ersetzen von Argumenten in eine Fehlermeldung-Zeichenfolge. Dabei wird ein Mechanismus verwendet, der auf der **printf**-Funktion der Standardbibliothek der C-Programmiersprache modelliert wurde.
- RAISERROR kann neben der Textmeldung eine eindeutige Fehlernummer, einen Schweregrad und einen Statuscode angeben.
- Mit RAISERROR lassen sich mithilfe der gespeicherten Systemprozedur **sp_addmessage** benutzerdefinierte Meldungen zurückgeben.

Der von der RAISERROR-Anweisung ausgelöste Fehler ist nicht Teil eines Resultsets. Fehler werden an Anwendungen über einen Fehlerbehandlungsmechanismus zurückgegeben, der von der Verarbeitung von Resultsets getrennt ist.

Es gibt zwei Möglichkeiten, wie Datenbankmodul Informationen an den Aufrufer zurückgeben kann:

1. Fehler

- Die Fehler aus **sys.messages** mit einem Schweregrad von 11 oder höher.
- Jede RAISERROR- Anweisung mit einem Schweregrad von 11 oder höher.

2. Meldungen

- Die Ausgabe der PRINT- Anweisung.
- Die Ausgabe mehrerer DBCC-Anweisungen.
- Die Fehler aus **sys.messages** mit einem Schweregrad von 10 oder niedriger.
- Jede RAISERROR-Anweisung mit einem Schweregrad von 10 oder niedriger.

Syntax:

```
raiserror({msg_id | msg_str | @local_variable},severity ,state  
[,argumentt [,....n ]])  
[with {log | nowait | seterror}]
```

Alle mit RAISERROR zurückgegebenen Fehlermeldungen besitzen eine Meldungsnummer größer 50000. Dabei ist es egal ob sie temporär durch die RAISERROR- Anweisung erzeugt werden oder ob eine Benutzerdefinierte Fehlermeldung aus sys.messages gelesen wird.

Ein RAISERROR- Schweregrad von 11 bis 19 in einem TRY-Block eines TRY...CATCH-Konstrukts bewirkt, dass die Steuerung an den zugehörigen CATCH-Block übertragen wird. Geben Sie einen Schweregrad von 10 oder geringer an, um Meldungen mithilfe von RAISERROR zurückzugeben, ohne einen CATCH-Block aufzurufen. PRINT übergibt die Steuerung nicht an einen CATCH- Block.

Ein RAISERROR- Schweregrad von 20 bis 25 werden als schwerwiegende Fehler angesehen und die Verbindung des Clients zum Server wird getrennt. Schweregrade von 19 bis 25 können nur von Mitgliedern der festen Serverrolle **sysadmin** oder Benutzern mit **ALTER TRACE**- Berechtigungen angegeben werden. Für Schweregrade von 19 bis 25 ist die Option WITH LOG erforderlich. Der Fehler wird in den Fehler- und Anwendungsprotokollen dokumentiert.

Verwenden Sie RAISERROR für folgende Aufgaben:

- Unterstützung bei der Problembehandlung von Transact-SQL-Code.
- Überprüfen der Werte von Daten.
- Zurückgeben von Meldungen, die variablen Text enthalten.
- Veranlassen der Ausführung zu einem Sprung von einem TRY-Block zum zugehörigen CATCH-Block.
- Zurückgeben von Fehlerinformationen vom CATCH-Block an den aufrufenden Batch oder die Anwendung.

Beispiel:

Zurückgeben einer einfachen Meldung.

```
raiserror('SQL Server find ich gut.',10,1);
```

Fehlermeldung mit Parametervariablen

```
declare @dbid int, @dbname nvarchar(128);  
  
set @dbid = db_id();  
set @dbname = db_name();  
  
raiserror('Datenbank-ID: %d, Datenbankname: %s.', 10,1,@dbid,dbname);
```

Bei benutzerdefinierten Fehlermeldungen ab einem Schweregrad 11 gibt die globale Systemfunktion @@ERROR die Fehlernummer der Meldung zurück.

Beispiel:

Schweregrad 10, @@error ist 0.

```
raiserror('Dienstag',10,1);
select @@error;
```

Schweregrad über 11, @@error ist in diesem Fall 50000.

```
raiserror('Dienstag',11,1);
select @@error;
```

Sonst, bei allen SQL- Anweisungen, @@ERROR ist 0 wenn die vorangegangene SQL- Anweisung ordnungsgemäß gearbeitet hat. Tritt hier ein Fehler auf dann wird die entsprechende Systemfehlernummer durch @@ERROR aufgefangen. Darum muss die Funktion immer nach der zu prüfenden Anweisung abgefragt werden.

Eigene dauerhafte Fehlermeldungen können mit der gespeicherten Prozedur sp_addmessage oder im SQL Server Management Studio erzeugt werden. Diese müssen eine Fehlernummer über 50000 besitzen und werden in der Systemsicht sys.messages abgespeichert.

Syntax:

```
sp_addmessage [@msgnum =] msg_id ,[@severity =] severity ,
[@msgtext =] 'msg'
[, [@lang =] 'language'][, [@with_log =] {'true' | 'false'}]
[, [@replace =] 'replace']
```

Beachten Sie, wenn Sie nicht mit einer englischen SQL Serverversion arbeiten, dass zuerst die englischsprachige und danach die Meldung in der Versionssprache erstellt werden muss.

Beispiel:

```
sp_addmessage 50100,16,'Can not deleted', 'us_english', 'true'
go
sp_addmessage 50100,16,'Kann nicht gelöscht werden', 'German', 'true'
go
```

Aufrufen dieser Fehlermeldung

```
raiserror(50100,16,1)
```

13.1.4.3 THROW

Die THROW- Anweisung wurde in SQL Server 2012 neu eingeführt und ergänzt die RAISERROR- Anweisung.

Genau wie RAISERROR löst THROW eine Ausnahme aus und übergibt die Ausführung einem CATCH- Block in SQL Server 2012. Ist kein TRY...CATCH- Block verfügbar, wird die Sitzung durch THROW beendet und die Zeilennummer und die Prozedur, in der die Ausnahme ausgelöst wurde, und der Schweregrad 16 werden festgelegt.

Syntax:

```
throw [error_number, message, state];
```

Die THROW- Anweisung muss immer durch ein Semikolon abgeschlossen werden. Die Fehlernummer ist eine Konstante oder Variable vom Datentyp int und muss größer oder gleich 50000 und kleiner oder gleich 2147483647 sein.

Die Fehlernummer muss nicht in sys.messages definiert sein, der Schweregrad ist immer 16 und es werden keine Parameter akzeptiert.

Wird die Anweisung ohne Parameter angegeben, muss sie in einem CATCH- Block stehen.

Beispiel:

```
throw 71000, 'Das ist hier nicht erlaubt!', 1;
```

13.1.5 Verwenden von Variablen

Variablen sind wichtige Elemente einer Programmiersprache. In T-SQL ist es nur möglich lokale Variable zu erstellen.

Der Name der Variablen kann jedes Zeichen des Zeichensatzes enthalten, muss aber mit einem @ beginnen und darf nicht länger sein als 128 Zeichen.

Variablen können nur in Ausdrücken verwendet werden, nicht anstelle von Objektnamen oder Schlüsselwörtern. Um dynamische SQL-Anweisungen zu erstellen, verwenden Sie EXECUTE.

Es gibt nur einfache Variable, keine Arrays (Felder) und Records (zusammengesetzte Typen). Sie können aber jederzeit eine lokale Variable in einem CLR-basiertem Benutzerdatentypen deklarieren und so strukturierte Informationen verwenden.

Eine Ausnahme bildet der Systemdatentyp **table**. Mit diesem Typ können Aufgaben erledigt werden, die ein Fall für strukturierte Arrays sind.

13.1.5.1 Deklaration

Variablen werden im Hauptteil eines Batches oder einer Prozedur mit einer DECLARE-Anweisung deklariert. Nach der Deklaration werden alle Variablen mit NULL initialisiert.

Es ist seit SQL Server 2008 möglich, eine Variable gleich bei der Deklaration mit einem Wert zu belegen.

Syntax:

```
declare
    {{ @local_variable [as] data_type }
 |   { @cursor_variable_name cursor }
 |   { @table_variable_name [as] <table_type_definition> } } [ ,...n]

<table_type_definition> ::= 
    table ({ <column_definition> | <table_constraint> } [ ,... ])
```

Eine Variable kann nicht vom Datentyp **text**, **ntext** oder **image** sein. Eine Variable ist nur in dem Batch, in der Prozedur oder Funktion gültig in der sie deklariert wurde.

Beispiel:

Einfache Variable deklarieren.

```
declare @name nvarchar(50), @ort nvarchar(50);
declare @doku xml;
```

Variable vom Typ table deklarieren.

```
declare @table_var table (namen nvarchar(50) null,
                        ort nvarchar(50) null);
```

13.1.5.2 Wertzuweisung zu Variablen

Variablen können Werte mit der SET- Anweisung oder durch Select- Anweisungen zugewiesen werden. Pro SET- Anweisung kann immer nur einer Variablen ein Wert zugewiesen werden.

Beispiel:

Wertzuweisung mit SET

```
declare @name nvarchar(50), @anzahl int;  
  
set @name = 'Schulze';  
print @name;  
  
set @anzahl = (select count(*) from lieferung);  
print cast(@anzahl as varchar(20));
```

oder:

```
declare @zahl int;  
  
set @zahl = 0;  
set @zahl += 2;  
print cast(@anzahl as varchar(20));
```

Wertzuweisung durch Select- Anweisung

```
declare @name nvarchar(50), @ort nvarchar(50), @nr nchar(3);  
  
set @nr = 'L01';  
  
select @name = lname, @ort = lstadt  
from lieferant  
where lnr = @nr;  
  
print @name + ' ' + @ort;
```

Wertzuweisung an eine Tabellenvariable

```
declare @table_var table (namen nvarchar(50) null,  
                           ort nvarchar(50) null);  
  
insert into @table_var  
select lname, lstadt  
from lieferant;  
  
select * from @table_var;
```

13.1.5.3 Vordefinierte Systemvariable

SQL Server 2012 stellt eine große Anzahl vordefinierter Variablen bereit. In der Onlinedokumentation werden diese als Funktionen geführt. Doch ihre Verwendung zeigt, dass diese Bezeichnung nicht korrekt ist.

Diese Variablen sind global und stehen in jeder Benutzerverbindung und jedem Gültigkeitsbereich zur Verfügung.

Gekennzeichnet sind sie durch ein doppeltes @ als Präfix.

| Systemvariable / Systemfunktion | Beschreibung |
|---------------------------------|--|
| @@CONNECTIONS | Gibt die Anzahl der erfolgreichen oder nicht erfolgreichen versuchten Verbindungen zurück, die seit dem letzten Start von SQL Server aufgetreten sind. |
| @@CPU_BUSY | Gibt die Zeit zurück, die SQL Server seit dem letzten Start beansprucht hat. |
| @@CURSOR_ROWS | Gibt die Anzahl der kennzeichnenden Zeilen zurück, die sich aktuell im letzten für die Verbindung geöffneten Cursor befinden |
| @@DATEFIRST | Gibt den aktuellen Wert des SET DATEFIRST-Parameters für die Sitzung zurück. |
| @@DBTS | Gibt den nächsten timestamp- Wert, für die aktuelle Datenbank zurück. Dieser timestamp- Wert ist in der Datenbank definitiv nur einmal vorhanden. |
| @@ERROR | Gibt die Fehlernummer für die zuletzt ausgeführte Transact-SQL-Anweisung zurück. |
| @@FETCH_STATUS | Gibt den Status der letzten Cursor- FETCH- Anweisung zurück, die für einen beliebigen aktuell von der Verbindung geöffneten Cursor ausgegeben wurde. |
| @@IDENTITY | Eine Systemfunktion, die den zuletzt eingefügten Identitätswert zurückgibt. |
| @@IDLE | Gibt die Zeit zurück, während der sich SQL Server seit dem letzten Start im Leerlauf befunden hat. |
| @@IO_BUSY | Gibt die Zeit zurück, die SQL Server seit dem letzten Start von SQL Server für Eingabe- und Ausgabevorgänge aufgewendet hat. |
| @@LANGID | Gibt den lokalen Sprachenbezeichner der zurzeit verwendeten Sprache zurück. |
| @@LANGUAGE | Gibt den Namen der zurzeit verwendeten Sprache zurück. |
| @@LOCK_TIMEOUT | Gibt die aktuelle Einstellung für das Spervertimeout für die aktuelle Sitzung in Millisekunden zurück. |
| @@MAX_CONNECTIONS | Gibt die maximale Anzahl gleichzeitiger Benutzerverbindungen an, die für eine SQL Server-Instanz zulässig sind. |
| @@MAX_PRECISION | Gibt den Genauigkeitsgrad zurück, der von den Datentypen decimal und numeric verwendet wird, so wie er aktuell auf dem Server festgelegt ist. |
| @@NESTLEVEL | Gibt die Schachtelungsebene der aktuellen Ausführung einer gespeicherten Prozedur auf dem lokalen Server zurück |
| @@OPTIONS | Gibt Informationen zu den aktuellen SET- Optionen zurück. |

| Systemvariable / Systemfunktion | Beschreibung |
|---------------------------------|--|
| @@PACK_RECEIVED | Gibt die Anzahl der von SQL Server seit dem letzten Start aus dem Netzwerk gelesenen Eingabepakete zurück. |
| @@PACK_SENT | Gibt die Anzahl der von SQL Server seit dem letzten Start in das Netzwerk geschriebenen Ausgabepakete zurück |
| @@PACKET_ERROR | Gibt die Anzahl der Netzwerkpacket- Fehler zurück, die bei SQL Server- Verbindungen seit dem letzten Start von SQL Server aufgetreten sind. |
| @@PROCID | Gibt den Objektbezeichner (ID) des aktuellen Transact- SQL- Moduls zurück. Bei einem Transact -SQL- Modul kann es sich um eine gespeicherte Prozedur, eine benutzerdefinierte Funktion oder einen Trigger handeln. |
| @@ROWCOUNT | Gibt die Anzahl der Zeilen zurück, auf die sich die letzte Anweisung ausgewirkt hat. |
| @@SERVERNAME | Gibt den Namen des lokalen Servers zurück, auf dem SQL Server ausgeführt wird. |
| @@SERVICENAME | Gibt den Namen des Registrierungsschlüssels zurück, unter dem SQL Server ausgeführt wird. @@SERVICENAME gibt 'MSSQLSERVER' zurück, wenn die aktuelle Instanz die Standardinstanz ist. Diese Funktion gibt den Namen der Instanz zurück, wenn die aktuelle Instanz eine benannte Instanz ist. |
| @@SPID | Gibt die Sitzungs-ID des aktuellen Benutzerprozesses zurück. |
| @@TEXTSIZE | Gibt den aktuellen Wert der TEXTSIZE-Option der SET- Anweisung zurück. Dieser gibt die maximale Länge (in Byte) von varchar(max)-, nvarchar(max)-, varbinary(max)-, text- oder image-Daten an, die von einer SELECT- Anweisung zurückgegeben werden. |
| @@TIMETICKS | Gibt die Anzahl von Mikrosekunden pro Zeitsignal zurück. |
| @@TOTAL_ERROR | Gibt die Anzahl der in SQL Server seit dem letzten Start von SQL Server aufgetretenen Schreibfehler auf dem Datenträger zurück. |
| @@TOTAL_READ | Gibt die Anzahl der von SQL Server seit dem letzten Start ausgeführten Lesezugriffe auf den Datenträger zurück. |
| @@TOTAL_WRITE | Gibt die Anzahl der von SQL Server seit dem letzten Start ausgeführten Schreibzugriffe auf den Datenträger zurück. |
| @@TRANCOUNT | Gibt die Anzahl von aktiven Transaktionen für die aktuelle Verbindung zurück. |
| @@VERSION | Gibt die Version, die Prozessorarchitektur, das Erstellungsdatum und das Betriebssystem für die aktuelle Installation von SQL Server zurück. |

Beispiel:

```
select @@version
```

13.1.6 Bedingungen mit IF..Else

Legt Bedingungen für die Ausführung einer T- SQL- Anweisung fest. Die Anweisung wird ausgeführt, falls die Bedingung zu TRUE ausgewertet wird. Die alternative Anweisung nach dem optionalen ELSE-Schlüsselwort wird ausgeführt, falls die Bedingung zu FALSE oder NULL ausgewertet wird.

Syntax:

```
if bedingung
{sql_anweisung | sql_anweisungs_block }
[else
{sql_anweisung | sql_anweisungs_block}]
```

Beispiel:

```

if datepart(yy,getdate()) > datepart(yy,'31.12.2006')
begin
    select *
    into lief_2006
    from lieferung;
    raiserror('Dateien kopiert',10,1) with log;
end;
else
    print 'Kopieren nicht erforderlich';

oder

if not exists (select * from artikel where anr = 'A09');
    print 'Artikel existiert nicht';

```

13.1.7 Bedingungen mit IIF

Überprüft einen booleschen Ausdruck und gibt einen von zwei Werten zurück, je nachdem ob der boolesche Ausdruck Wahr oder Falsch ergibt.

Syntax:

```
iif(bedingung, wahr_wert, falsch_wert);
```

IIF ist eine schnelle Möglichkeit zum Schreiben einer CASE- Anweisung. Die Regeln für CASE, NULL- Behandlung und Rückgabetypen, gelten auch für IIF.

Genau wie CASE- Anweisungen können auch IIF- Anweisungen nur bis zur Ebene 10 geschachtelt werden.

Beispiel:

```

declare @a int =33, @b int = 40;
select iif( @a > @b, 'Wahr', 'Falsch');

```

13.1.8 Schleifen

Die WHILE- Anweisung legt eine Bedingung für die wiederholte Ausführung einer SQL- Anweisung oder eines Anweisungsblockes fest. Die Anweisungen wird solange ausgeführt, wie die angegebene Bedingung WAHR ist. Die Ausführung der Anweisungen in der WHILE- Schleife kann mithilfe der Schlüsselwörter BREAK und CONTINUE auch innerhalb der Schleife gesteuert werden.

WHILE- Schleifen werden in T- SQL sehr selten benötigt, weil das mengenorientierte SQL meistens das gleiche Ergebnis bringt und außerdem um Größenordnungen schneller arbeitet.

Bei der Programmierung so genannter Cursor kommt man ohne sie nicht aus.

Syntax:

```

while boolean_expression
{sql_anweisung | sql_anweisungs_block}
[continue] [break]

```

Beispiel:

Skriptteil mit WHILE sowie BREAK und CONTINUE

```
while (select avg(status) from dbo.lieferant) < 40
begin
    update dbo.lieferant
    set status= status + 2;
    select max(status) from production.product;
    if (select max(listprice) from production.product) > $500
        break;
    else
        continue;
end;
```

13.1.9 TRY...Catch- Konstrukte

Fehler in SQL- Code (in Batches, gespeicherten Prozeduren und Triggern) können mithilfe eines TRY...CATCH- Konstrukt verarbeitet werden.

So ein Konstrukt besteht aus zwei Teilen: einem TRY- Block und einem CATCH- Block.

Wenn in einer Transact-SQL-Anweisung innerhalb eines TRY- Blocks eine Fehlerbedingung erkannt wird, wird die Steuerung an einen CATCH- Block übergeben, wo der Fehler verarbeitet werden kann.

Nachdem der CATCH-Block die Ausnahme verarbeitet, wird die Steuerung an die erste Transact-SQL-Anweisung übertragen, die auf die END CATCH- Anweisung folgt.
Falls die END CATCH- Anweisung die letzte Anweisung in einer gespeicherten Prozedur oder einem Trigger ist, wird die Steuerung an den Code zurückgegeben, der die gespeicherte Prozedur oder den Trigger aufgerufen hat. Transact-SQL-Anweisungen in dem TRY- Block, der auf die Anweisung folgt, durch die ein Fehler generiert wurde, werden nicht ausgeführt.

Wenn der TRY-Block keine Fehler aufweist, wird die Steuerung sofort nach der zugeordneten END CATCH-Anweisung an die Anweisung übergeben.

Falls es sich bei der END CATCH-Anweisung um die letzte Anweisung in einer gespeicherten Prozedur oder einem Trigger handelt, wird die Steuerung an die Anweisung übergeben, die die gespeicherte Prozedur oder den Trigger ausgelöst hat.

Syntax:

```
BEGIN TRY
    { sql_anweisung | anweisungs_block }
END TRY
BEGIN CATCH
    { sql_anweisung | anweisungs_block }
END CATCH
[ ; ]
```

Regeln:

- Ein TRY- Block beginnt mit der BEGIN TRY- Anweisung und endet mit der END TRY- Anweisung. Eine oder mehrere Transact-SQL-Anweisungen können zwischen den BEGIN TRY- und END TRY- Anweisungen angegeben werden.
- Ein TRY-Block muss unmittelbar von einem CATCH- Block gefolgt werden. Ein CATCH-Block beginnt mit der BEGIN CATCH- Anweisung und endet mit der END CATCH- Anweisung.
- In Transact- SQL ist jeder TRY- Block nur einem CATCH- Block zugeordnet
- Jedes TRY...CATCH- Konstrukt muss sich vollständig in einem einzelnen Stapelprogramm, einer gespeicherten Prozedur oder einem Trigger befinden. Es ist nicht möglich, einen TRY- Block in einem Stapel und den zugehörigen CATCH- Block in einem anderen Stapel zu platzieren.

- Fehler mit einem Schweregrad von 20 oder höher, aufgrund derer die Verbindung zum DB-Server beendet wird, werden nicht vom TRY...CATCH-Block behandelt. Wird die Verbindung jedoch beibehalten werden die Fehler auch behandelt.
- Fehler mit einem Schweregrad von 10 oder niedriger werden als Warnungen oder Informationsmeldungen angesehen und nicht von TRY...CATCH-Blocks behandelt.
- TRY...CATCH-Konstrukte können geschachtelt werden.

Fehlerfunktionen von TRY...Catch, um Fehlerinformationen zu sammeln:

- ★ ERROR_NUMBER() gibt die Fehlernummer zurück.
- ★ ERROR_MESSAGE() gibt den vollständigen Text der Fehlermeldung zurück, einschließlich der angegebenen Parameter.
- ★ ERROR_SEVERITY() gibt den Fehlerschweregrad zurück.
- ★ ERROR_STATE() gibt die Fehlerstatusnummer zurück.
- ★ ERROR_LINE() gibt die Zeilennummer innerhalb der Routine zurück, die den Fehler verursacht hat.
- ★ ERROR_PROCEDURE() gibt den Namen der gespeicherten Prozedur oder des Triggers mit dem Fehler zurück.

Beispiel:

Es soll ein Fehler provoziert werden durch eine Division durch Null. Dadurch wird der CATCH-Block ausgelöst.

```
use standard;
go

begin try
    select 1/0;
end try
begin catch
    select error_number() as Fehlernummer, error_severity() as Schweregrad,
           error_state() as Fehlerstatus, error_procedure() as Fehlerprozedur,
           error_line() as Fehlerzeile, error_message() as Fehlermeldung;
end catch;
go
```

Es gibt zwei Fehlertypen, die nicht von TRY...CATCH behandelt werden:

- Kompilierfehler, beispielsweise Syntaxfehler, die verhindern, dass ein Batch ausgeführt wird.
- Fehler, die während der Neukompilierung auf Anweisungsebene auftreten, beispielsweise Fehler bei der Objektnamensauflösung, die aufgrund einer verzögerten Namensauflösung nach der Kompilierung auftreten.

Beispiel:

```
use standard;
go

begin try
    print n'startet Programm';
    select * from nicht_lieferant;
end try
begin catch
    select
        error_number() as Fehlernummer,
        error_message() as Fehlermeldung;
end catch;
go
```

Wenn die oben stehende Select- Anweisung in einer gespeicherten Prozedur auf eine nicht vorhandene Tabelle zugreift und diese im TRY- Block aufgerufen wird, dann würde der CATCH- Block auf den Fehler reagieren.

13.1.10 RETURN- Anweisung

Das Kommando RETURN beendet einen Befehlsstapel oder eine benutzerdefinierte Prozedur. Keine der Anweisungen in einer gespeicherten Prozedur oder einem Batch, die auf die RETURN- Anweisung folgen, wird ausgeführt.

Wenn die RETURN- Anweisung in einer gespeicherten Prozedur verwendet wird, kann sie einen ganzzahligen Wert an die aufrufende Anwendung oder Prozedur oder den aufrufenden Batch zurückgegeben. Falls kein Wert für RETURN angegeben wird, gibt eine gespeicherte Prozedur den Wert 0 zurück.

Dieser Wert lässt sich per **ADO.Net** auf dem Client, als Parameter vom Typ **return**, auswerten.

Syntax:

```
return [ganzzahl]
```

13.1.11 CHOOSE- Anweisung

Diese Anweisung gibt einen Wert entsprechend eines angegebenen Index aus einer Werteliste zurück.

Syntax:

```
choose(index, wert_1, wert_2[, wert_n]);
```

CHOOSE hat die gleiche Funktion wie ein Index in einem Array, wobei das Array aus den Argumenten besteht, die dem Indexargument folgen. Das Indexargument bestimmt, welcher der Werte zurückgegeben wird.

Beispiel:

Es soll der dritte Wert aus der Liste ausgegeben werden.

```
select choose( 3, 'Weimar', Jena', 'Urbich', 'Erfurt', 'Gera');
```

13.1.12 Dynamische SQL Anweisungen

Immer wieder gibt es Situationen in denen der Programmierer dynamische Anweisungen benötigt, die zur Laufzeit eine Wertzuweisung erfahren.

Mit dem Kommando EXECUTE kann man nicht nur gespeicherte Prozeduren aufrufen, sondern auch dynamische Anweisungen starten.

Syntax:

```
exec[ute] ({@string_var | {[N] 'tsql_string'} + [...]})  
[as {login | user} = 'name']  
[with <execute_optionen>]
```

```

<execute_optionen>::=
{
  result sets undefined
  | result sets none
  | result sets (<ergebnis_definition> [, ...])
}

<ergebnis_definition>::=
{
  (column_name data_type [{null | not null}] [...])
  | as object {tab_name | view_name | funkt_name}
  | as type table_type_name
  | as for xml
}

```

Beispiel:

```

declare @modus int, @sort int, @ordby varchar(10);
declare @tab sysname, @sql varchar(1000);

set @modus = 1;
set @sort = 2;

if @modus = 1
  set @tab = 'lieferant';
else
  set @tab = 'artikel';
if @sort = 1
  set @ordby = 'asc';
else
  set @ordby = 'desc';

set @sql = 'select * from ' + @tab + ' order by 2 ' + @ordby;
exec(@sql);
go

```

In SQL Server 2012 wurde die EXECUTE- Anweisung durch das WITH RESULT SETS- Argument erweitert. Dadurch können unter anderem neue Spaltennamen und Datentypen des erwarteten Resultsets verwendet werden.

Beispiel:

```

declare @sql nvarchar(max);
set @sql = 'select a.lnr, lname, anz
            from lieferant a join (
              select lnr, count(distinct anr) as [anz]
              from lieferung
              group by lnr) as b on a.lnr = b.lnr;'

exec(@sql) with result sets
(
  ([Lieferantennummer] nvarchar(10) not null,
  [Lieferantenname] nvarchar(200) not null,
  [unterschiedliche Artikel geliefert] int null)
);

```

Das sind relativ simple SQL Anweisungen EXECUTE. Wenn Sie komplexere dynamische Anweisungen, zum Beispiel mit variablen Spaltennamen in der WHERE- Klausel, schreiben müssen, dann reicht EXECUTE nicht mehr aus. Verwenden Sie dann die gespeicherte Systemprozedur **sp_executesql**.

13.2 GOTO- Anweisung

Verändert den Kontrollfluss. Sie sollten auf den Einsatz von GOTO verzichten. GOTO-Anweisungen können geschachtelt werden.

Die Transact- SQL- Anweisung oder SQL Anweisungen, die GOTO folgen, werden ausgelassen, und die Verarbeitung wird an der Marke fortgesetzt die von GOTO angegeben wird. GOTO- Anweisungen können geschachtelt werden.

Syntax:

marke:

...

goto marke

Beispiel:

```
use standard;
go
select * from lieferant;
if @@rowcount = 0
    goto k_datensatz;
...
k_datensatz:
    raiserror('Kein Datensatz gefunden.',10,1);
go
```

13.3 WAITFOR- Anweisung

WAITFOR blockiert die Ausführung eines Batches, einer gespeicherten Prozedur oder einer Transaktion bis zum Erreichen einer bestimmten Zeit oder eines bestimmten Zeitintervalls, oder bis eine angegebene Anweisung mindestens eine Zeile ändert oder zurückgibt.

Syntax:

```
WAITFOR
{
    DELAY 'time_to_pass'
    | TIME 'time_to_execute'
    | [ ( receive_statement ) | ( get_conversation_group_statement ) ]
    | [, TIMEOUT timeout ]
}
```

| | |
|--------------------|--|
| DELAY: | Die angegebene Zeit die verstreichen muss, bevor die Ausführung fortgesetzt wird. |
| 'time_to_pass': | Der Zeitraum, der gewartet werden soll. |
| TIME: | Die angegebene Zeit, zu der der Batch, die gespeicherte Prozedur oder die Transaktion ausgeführt wird. |
| 'time_to_execute': | Die Zeit, zu der die WAITFOR-Anweisung beendet wird. |

Beispiel:

```
use standard;
go
begin
    waitfor delay '02:00';
    execute sp_helpdb;
end;
go
```

13.4 Synonyme

Ein Synonym (seit SQL Server 2012) stellt einen alternativeren Namen für ein anderes Datenbankobjekt auf dem lokalen Server oder einen Remoteserver bereit.

Informationen über vorhandene Synonyme werden in der Katalogsicht **sys.synonyms** abgespeichert.

Es dient dem leichteren Datenzugriff durch Clientanwendungen. Bisher mussten sie über einen vierteiligen Namen (server.datenbank.schema.object) auf eine Ressource auf einem anderen Server zugreifen.

Außerdem muss nun nur noch das Synonym verändert werden wenn zum Beispiel die Tabelle in ein anderes Schema der Datenbank verschoben wird. Dazu muss allerdings das Synonym gelöscht und neu erstellt werden.

Syntax:

```
create synonym <schema.><name> for <objekt>;
<objekt>::=
    [servername.] [database_name.] [schema_name.] object_name
```

Synonyme können für folgende Datenbankobjekte erstellt werden:

- Gespeicherte Assemblyprozedur (CLR)
- Gespeicherte Tabellenwertfunktion (CLR)
- Assemblyskalarfunktion (CLR)
- Assemblyaggregatfunktion CLR)
- Replikationsprozeduren
- Erweiterte gespeicherte Prozeduren
- SQL- Skalarfunktionen, - Tabellenwertfunktionen und - Inlinefunktionen
- Gespeicherte SQL- Prozeduren, Tabellen und Views

Ein Synonym kann nicht die Basis für ein neues Synonym sein. Und auf ein Synonym welches auf einem Verbindungsserver gespeichert ist, kann nicht verwiesen werden.

Auch in DDL- Anweisungen kann nicht auf ein Synonym verwiesen werden.

Von schemagebundenen Ausdrücken (check, schemagebundene Funktionen und Sichten usw.) kann ebenfalls nicht auf Synonyme verwiesen werden.

Synonyme können in SELECT-, UPDATE-, INSERT- und DELETE- Anweisungen sowie in Unterabfragen und EXECUTE verwendet werden.

Beispiel:

Erstellen eines Synonyms für eine Tabelle in einer anderen Datenbank der gleichen SQL Server Instanz.

```
use standard;
go

create synonym dbo.produkt for ersql2012.adventureworks2012.production.product;
go

select * from produkt;
go
```

Synonyme können mit der DROP SYNONYME- Anweisung wieder gelöscht werden.

13.5 Cursor

Abfragen in einer Datenbank beziehen sich immer auf eine vollständige Gruppe von Zeilen. Die von einer SELECT- Anweisung zurückgegebenen Zeilen bestehen aus allen Zeilen, die die Bedingungen der WHERE- Klausel der Anweisung erfüllen. Diese Gruppe von zurückgegebenen Zeilen wird als Resultset bezeichnet.

Anwendungen, vor allem interaktive Onlineanwendungen, sind nicht immer effektiv, wenn das gesamte Resultset als eine Einheit bearbeitet wird. Diese Anwendungen benötigen einen Mechanismus, um jeweils eine Zeile oder einen kleinen Zeilenblock zu bearbeiten.

Cursor ist eine Erweiterung zu Resultsets und stellen diesen Mechanismus bereit.

Cursors erweitern die Verarbeitung von Ergebnissen folgendermaßen:

- Ermöglichen der Positionierung an bestimmten Zeilen des Resultsets.
- Abrufen einer Zeile oder eines Zeilenblockes von der aktuellen Position im Resultset.
- Unterstützen von Datenänderungen in den Zeilen an der aktuellen Position im Resultset.
- Unterstützen von unterschiedlichen Sichtbarkeitsebenen bei Änderungen, die von anderen Benutzern an den Datenbankdaten, die im Resultset dargestellt werden, ausgeführt wurden.
- Bereitstellen des Zugriffs auf Daten in einem Resultset für Transact-SQL-Anweisungen in Skripts, gespeicherte Prozeduren und Trigger.

SQL Server unterstützt zwei Methoden zum Anfordern eines Cursors:

- Transact-SQL
Die Transact-SQL-Sprache unterstützt eine Syntax für das Verwenden von Cursorn, die sich an der Syntax von SQL-92-Cursorn orientiert.
- Cursorfunktionen über Datenbank-APIs
SQL Server unterstützt die Cursorfunktionen der folgenden Datenbank-APIs:
 - ADO (Microsoft ActiveX-Datenobjekt)
 - OLE DB
 - ODBC (Open Database Connectivity)

Transact-SQL- Cursor und API- Cursor verfügen über eine unterschiedliche Syntax, es wird jedoch der folgende allgemeine Prozess bei allen SQL Server- Cursors verwendet:

1. Verbinden Sie einen Cursor mit dem Resultset einer Transact-SQL-Anweisung, und definieren Sie die Eigenschaften des Cursors (z. B., ob die Zeilen im Cursor aktualisiert werden können).
2. Führen Sie die Transact-SQL-Anweisung aus, um den Cursor aufzufüllen.
3. Rufen Sie die Zeilen in den Cursor ab, die Sie anzeigen möchten. Der Vorgang, durch die eine Zeile oder ein Zeilenblock von einem Cursor abgerufen wird, wird als Abrufvorgang bezeichnet. Wird eine Folge von Abrufvorgängen vorwärts oder rückwärts ausgeführt, wird dies als Scrollen bezeichnet.
4. Führen Sie optional Änderungsvorgänge (Aktualisieren oder Löschen) in der Zeile an der aktuellen Position im Cursor aus.
5. Schließen Sie den Cursor.

13.5.1 DECLARE- Anweisung

Die DECLARE- Anweisung definiert die Attribute eines Servercursors, wie z. B. dessen Scroll Verhalten, sowie die Abfrage, die zum Erstellen des Resultsets verwendet wird, auf das der Cursor ausgeführt wird. DECLARE CURSOR unterstützt sowohl die Syntax basierend auf dem SQL-92-Standard als auch eine Syntax, für die eine Teilmenge der Transact-SQL- Erweiterungen verwendet wird.

SQL 92 Syntax:

```
declare cursor_name [insensitive] [scroll] cursor
for select_anweisung
[for {read only | update [of spalten_name [,...n]]}]
```

Erweiterte T- SQL Syntax:

```
declare cursor_name cursor [local | global]
[forward_only | scroll]
[static | keyset | dynamic | fast_forward]
[read_only | scroll_locks | optimistic]
[type_warning]
for select_anweisung
[for update [of column_name [,...n]]]
```

Die erste DECLARE CURSOR-Anweisung verwendet zum Deklarieren des Cursorverhaltens die SQL-92-Syntax. Die zweite DECLARE CURSOR-Anweisung verwendet Transact-SQL- Erweiterungen, die die Definition von Cursor mithilfe der gleichen Cursortypen zulassen, die in den Datenbank-API-Cursor-Funktionen von ODBC oder ADO verwendet werden.

Die beiden Formen können nicht gleichzeitig verwendet werden.

Wenn die Schlüsselwörter SCROLL oder INSENSITIVE vor dem Schlüsselwort CURSOR angeben werden, können keine Schlüsselwörter zwischen CURSOR und FOR verwendet werden. Das heißt wenn eines der Schlüsselwörter zwischen CURSOR und FOR angeben wurde, kann weder SCROLL noch INSENSITIVE vor dem Schlüsselwort CURSOR angegeben werden.

Regeln:

- Ist STATIC- und FAST_FORWARD angegeben dann ist der Cursor standardmäßig READ_ONLY.
- DYNAMIC- und KEYSET- Cursor sind standardmäßig OPTIMISTIC.
- Bei fehlenden Berechtigungen, bei Zugriff auf Remotetabellen die keine Aktualisierung unterstützen, ist der Cursor READ_ONLY.
- Wenn die in der Select- Anweisung angegebenen Tabellen keinen eindeutigen Index besitzen oder ein OUTER JOIN verwendet wird, ist der Cursor automatisch INSENSITIVE und damit READ_ONLY.
- Wenn in der Select- Anweisung die Schlüsselwörter DISTINCT, UNION, GROUP BY und HAVING vorkommen oder ein Konstanten Ausdruck verwendet wird, ist der Cursor automatisch INSENSITIVE und damit READ_ONLY.
- Das Schlüsselwort INTO ist in der Cursordefinition nicht zulässig.

Nach der Deklaration eines Cursors können die Eigenschaften des Cursors mithilfe der folgenden gespeicherten Systemprozeduren bestimmt werden:

| Gespeicherte Systemprozeduren | Beschreibung |
|--------------------------------------|---|
| sp_cursor_list | Gibt eine Liste der in der Verbindung aktuell sichtbaren Cursor und ihrer Attribute zurück. |
| sp_describe_cursor | Beschreibt die Attribute eines Cursors, z. B. ob es sich um einen Vorwärtscursor oder einen Scrollcursor handelt. |
| sp_describe_cursor_columns | Beschreibt die Spaltenattribute im Resultset des Cursors. |
| sp_describe_cursor_tables | Beschreibt die Basistabellen, auf die der Cursor zugreift. |

13.5.2 OPEN- Anweisung

Öffnet einen SQL- Servercursor und füllt den Cursor auf, indem die in der Anweisung DECLARE CURSOR- Anweisung oder der SET cursor_variable angegebene SQL- Anweisung ausgeführt wird.

Syntax:

```
open {[ [ global ] cursor_name } | cursor_variable_name}
```

Falls der Cursor mit der Option INSENSITIVE oder STATIC deklariert wurde, wird eine temporäre Tabelle für das Resultset erstellt.

Das Öffnen schlägt fehl, wenn die Größe eines Datensatzes im Ergebnis die Maximalgröße für SQL Server- Tabellen überschreitet. Die temporären Tabellen werden in tempdb gespeichert. Verwenden Sie die Funktion @@cursor_rows um Statusmeldungen über den letzten geöffneten Cursor zu erhalten.

Nach dem Öffnen des Cursors steht der Datensatzzeiger des Cursors vor dem ersten Datensatz der Treffermenge.

13.5.3 FETCH- Anweisung

Diese Anweisung ruft einen Datensatz aus dem Resultset des Cursors ab und lädt ihn in Variable

Syntax:

```
fetch {[ {next | prior | first | last | absolute n | relative n} ]
from {[ [global] cursor_name | @cursor_variable_name }
[into @variable_name [, ...n]]}
```

Wird SCROLL in der SQL-92- Syntax DECLARE CURSOR nicht angegeben, wird lediglich die FETCH- Option NEXT unterstützt.

Wenn die DECLARE CURSOR- Erweiterungen von Transact- SQL verwendet werden, gelten folgende Regeln:

- Falls entweder FORWARD_ONLY oder FAST_FORWARD angegeben wird, wird lediglich die FETCH- Option NEXT unterstützt.
- Wenn keine der Optionen DYNAMIC, FORWARD_ONLY oder FAST_FORWARD, aber eine der Optionen KEYSET, STATIC oder SCROLL angegeben ist, werden alle FETCH-Optionen unterstützt.
- DYNAMIC SCROLL- Cursor unterstützen alle FETCH-Optionen außer ABSOLUTE.

Bei jeder Ausführung der FETCH- Anweisung wird **@@fetch_status** neu initialisiert.

| Rückgabewert | Beschreibung |
|--------------|--|
| 0 | Die FETCH-Anweisung war erfolgreich. |
| -1 | Die FETCH-Anweisung ist fehlgeschlagen, oder die Zeile war außerhalb des Resultsets. |
| -2 | Die abgerufene Zeile fehlt. |

Da **@@fetch_status** global ist und für alle Cursor der Sitzung gilt, sollte sie sofort nach der FETCH- Anweisung geprüft werden.

13.5.4 CLOSE- Anweisung

Entfernt das Resultset aus dem Speicher und gibt alle Cursorsperren auf die Datensätze des Ergebnisses wieder frei. Der Cursor kann jederzeit wieder mit OPEN geöffnet werden.

Syntax:

```
close {[ global ] cursor_name} | cursor_variable_name}
```

13.5.5 DEALLOCATE- Anweisung

Entfernt einen Cursor. Es werden alle Datenstrukturen, die den Cursor bilden freigegeben.

Syntax:

```
deallocate {[ global ] cursor_name } | @cursor_variable_name}
```

Durch DEALLOCATE werden alle Scroll- Sperren freigegeben, die zum Schützen und der Isolierung der Abrufvorgänge verwendet werden. Transaktionssperren, auf vom Cursor vorgenommenen Aktualisierungen bleiben bis zum Ende der Transaktion wirksam.

Beispiel:

```
declare      @aname varchar(200), @datum varchar(15), @menge smallint,
            @lnr char(3), @anzahl int;

set @lnr = 'L01';
set @anzahl = 0;

declare lief_art cursor
for select aname, lmenge, convert(varchar(15), ldatum, 104) as ldatum
from lieferung a join artikel b
on a.anr = b.anr
where lnr = @lnr;

open lief_art;

fetch from lief_art into @aname, @menge, @datum;

if @@fetch_status <> 0
begin
    raiserror('Keinen Datensatz gefunden',10,1);
    deallocate lief_art;
    return;
end;
```

```

while @@fetch_status = 0
begin
    print 'Artikelname: ' + @aname + ', Liefermenge: ' +
    cast(@menge as varchar(5)) + ', Lieferdatum: ' + @datum;
    fetch from lief_art into @aname, @menge, @datum;
    set @anzahl += 1;
    if @@fetch_status = -1;
        begin
            print 'Gefundene Datensätze: ' + cast(@anzahl as varchar(10));
        end;
    end;
    deallocate lief_art;

```


14 Gespeicherte Prozeduren

Bei gespeicherten Prozeduren handelt es sich um eine Gruppe von Transact- SQL- Anweisungen, die zu einem einzigen Anweisungsplan kompiliert werden. Gespeicherte Prozeduren sind leistungsfähige und flexible Werkzeuge, die der Durchführung verschiedener Verwaltungs- und Datenbearbeitungsfunktionen dienen, z.B. dem Erstellen von Tabellen, dem Zuweisen von Berechtigungen oder der Durchführung mehrstufiger Datenbankaktualisierungen.

Obwohl es sich bei SQL selbst um eine nicht prozedurale Sprache handelt, umfasst der Dialekt Transact- SQL mehrere SQL- Erweiterungen, zu denen auch die Verwendung von Schlüsselwörtern zur Ablaufsteuerung gehört. Dadurch können gespeicherte Prozeduren eine komplexe Logik enthalten und unterschiedlichste Aufgaben erfüllen.

SQL Server unterstützt mehrere Typen von gespeicherten Prozeduren:

Gespeicherte Systemprozeduren:

Beginnen mit den Präfix sp_ und sind in der resources- Datenbank gespeichert. Sie stellen eine effiziente Methode dar um Daten aus den Systemtabellen abzurufen. Sie können in jeder beliebigen Datenbank ausgeführt werden.

Benutzerdefinierte gespeicherte Prozeduren:

Werden in jeder einzelnen Benutzerdatenbank erstellt. Stellen in SQL Server den Pedanten zu Subroutinen in prozeduralen Programmiersprachen dar.

Temporär gespeicherte Prozeduren:

Hierbei kann es sich um lokale (#) oder globale (##) handeln. Lokale temporäre Prozeduren sind nur innerhalb der Benutzersitzung verfügbar in der sie erstellt wurden. Globale temporäre Prozeduren stehen für alle Benutzersitzungen zur Verfügung.

Erweiterte gespeicherte Prozeduren:

Werden in Form von DLL's (Dynamic Link Libraries) implementiert. Sie werden in Programmiersprachen wie Visual C++ geschrieben. Diese Prozeduren werden wie Gespeicherte Systemprozeduren ausgeführt und mit Hilfe der gespeicherten Prozedur **sp_addextendedproc** in der **master**- Datenbank registriert.

Diese Prozeduren sind nur noch aus Gründen der Abwärtskompatibilität vorhanden. Sie sollten in vorhandenen Anwendungen ersetzt und in neuen Anwendungen nicht mehr verwendet werden. Dafür sollte die **CLR- Integration** verwendet werden.

Merkmale und Fähigkeiten:

- Annehmen von Eingabeparametern und Zurückgeben mehrerer Werte in Form von Ausgabeparametern an die aufrufende Prozedur oder den aufrufenden Batch.
- Aufnehmen von Programmieranweisungen, die Operationen in der Datenbank ausführen, einschließlich des Aufrufs anderer Prozeduren.
- Zurückgeben eines Statuswertes an eine aufrufende Prozedur oder einen aufrufenden Batch, der Erfolg oder Fehlschlagen (sowie die Ursache) anzeigt.

Vorteile:

- Modulare Programmierung.
Sie können die Prozeduren einmal erstellen, sie dann in der Datenbank speichern und beliebig oft in einem Programm aufrufen. Gespeicherte Prozeduren können unabhängig vom Quellcode des Programms geändert werden. Die Anwendungslogik der Prozedur kann gemeinsam von den Anwendungen genutzt werden. Dadurch wird die Datenkonsistenz bei Zugriffen und Änderungen gewährleistet.

- Schnellere Ausführung.
Wenn die Operation umfangreichen Transact- SQL- Code erfordert oder wiederholt ausgeführt wird, können gespeicherte Prozeduren oftmals schneller ausgeführt werden als Batches mit Transact- SQL- Code. Gespeicherte Prozeduren werden analysiert und optimiert, wenn sie erstellt werden. SQL Server speichert dann den Namen der Prozedur in der **sys.objects**- Katalogsicht und den Text der Prozedur in der **sys.sql_modules**- Katalogsicht. Nachdem die Prozedur das erste Mal ausgeführt wurde, kann beim wiederholtem Aufruf der Prozedur ein im Arbeitsspeicher (Prozedurcache) gehaltener optimierter Ausführungsplan dieser Prozedur verwendet werden, dagegen werden Transact- SQL- Anweisungen, die bei jeder Ausführung erneut von dem Client gesendet werden, jedes Mal kompiliert und optimiert, wenn sie von SQL Server ausgeführt werden.
- Reduzierung des Netzwerkverkehrs.
Eine Operation, die Hunderte von Zeilen mit Transact- SQL- Code erfordert, kann mittels einer einzigen Anweisung durchgeführt werden, die den Code in einer Prozedur ausführt, anstatt Hunderte von Codezeilen über das Netzwerk senden zu müssen.
- Verwendung als Sicherheitsmechanismus.
Den Benutzern können explizit Berechtigungen zur Ausführung einer gespeicherten Prozedur erteilt werden, und müssen nicht über die Berechtigungen verfügen, die Anweisungen in der Prozedur direkt auszuführen. Benutzer erhalten keinen Einblick in die Details der Datentabellen.
- Verzögerte Namensauflösung.
Gespeicherte Prozeduren können bei ihrer Erstellung auf nicht vorhandene Objekte verweisen. Das erhöht die Flexibilität bei der Erstellung der Datenbank. Die Objekte müssen erst vorhanden sein wenn die Prozedur ausgeführt wird.

14.1 Programmieren gespeicherter Prozeduren

Eine gespeicherte Prozedur wird in der aktuellen Datenbank erstellt. Eine Ausnahme bilden temporäre gespeicherte Prozeduren, die werden immer in der tempdb- Datenbank erstellt.

Syntax:

```
CREATE PROC [EDURE] [schema.] prozedurname [;number]
[{@parameter data_type } [VARYING] [= default] [OUTPUT]] [,...n]
[WITH [RECOMPILE] [ENCRYPTION] [EXECUTE_AS_Klausel]]
[FOR REPLICATION]
as
{proc_body | EXTERNAL NAME assembly_name.class_name.method_name}
[;]

<EXECUTE_AS_Klausel> ::=
{ EXEC | EXECUTE } AS { CALLER | SELF | OWNER | 'user_name' }
```

| | |
|--------------|---|
| schema_name: | Der Name des Schemas, zu dem die Prozedur gehört. |
| ;number: | Eine optionale ganze Zahl zum Gruppieren von Prozeduren mit den gleichen Namen. Wird nur noch aus Gründen der Abwärtskompatibilität zur Verfügung gestellt. Zum Beispiel eine Anwendung verwendet die Prozeduren lief_wohn;1 und lief_wohn;2. Durch die Anweisung DROP PROCEDURE lief_wohn werden beide Prozeduren gelöscht. |
| VARYING: | Gibt das als Ausgabeparameter unterstützte Resultset an. Sein Inhalt kann variieren. Gilt nur für Cursor- Parameter. |
| OUTPUT: | Gibt an, dass es sich bei der Parametervariable um einen Ausgabeparameter handelt. Parameter mit den Datentypen TEXT, NTEXT und IMAGE können als OUTPUT- Parameter verwendet werden, es sei denn, es handelt sich um eine CLR- Prozedur. |

| | |
|-------------|---|
| RECOMPILE: | Wenn angegeben, wird der Plan für diese Prozedur nicht zwischengespeichert. Die Prozedur wird zur Laufzeit neu kompiliert. |
| ENCRYPTION: | Der Quelltext der Prozedur wird verschlüsselt im Systemkatalog abgelegt. Diese Option ist für CLR- Prozeduren nicht gültig. |
| EXECUTE AS: | Gibt den Sicherheitskontext an, unter dem die gespeicherte Prozedur ausgeführt wird. Standardwert ist CALLER. |

Beispiel:

```
use standard;
go
create procedure lief_datum
as
select lname, lstadt, lmenge
from lieferant a, lieferung b
where a.lnr = b.lnr
and ldatum < getdate() and status between 5 and 25;
```

Regeln für das Programmieren

- Die **CREATE PROCEDURE**- Definition kann SQL- Anweisungen von beliebiger Anzahl und eines beliebigen Typs enthalten. Folgende Create Anweisungen können nie innerhalb einer gespeicherten Prozedur verwendet werden:
CREATE VIEW, CREATE DEFAULT, CREATE RULE, CREATE PROCEDURE und
CREATE TRIGGER.
- Die **CREATE PROCEDURE**- Anweisung darf nicht mit anderen SQL- Anweisungen in einem Stapel stehen.
- In einer gespeicherten Prozedur kann ein **TRY... CATCH**- Konstrukt verwendet werden.
- Erstellen Sie innerhalb der Prozedur eine lokale temporäre Tabelle, steht diese Tabelle nur für die gespeicherte Prozedur zur Verfügung und wird nach Beendigung der Prozedur gelöscht.
- Sie können auf temporäre Tabellen innerhalb einer Prozedur verweisen.
- Prozeduren können bis zu 32 Ebenen verschachtelt werden. Die aktuelle Schachtelungsebene ist in der Systemfunktion **@@nestlevel** gespeichert.
- Die maximale Größe einer gespeicherten Prozedur ist nicht festgelegt.
- Erfordert die CREATE PROCEDURE- Berechtigung in der Datenbank und die ALTER- Berechtigung auf dem Schema, in dem die Prozedur erstellt wird.
Bei CLR- gespeicherten Prozeduren müssen Sie der Besitzer der Assembly sein, auf die in <methodSpecifier> verwiesen wird, oder über die REFERENCES- Berechtigung für diese Assembly verfügen.
- Um gespeicherte Prozeduren von gespeicherte Systemprozeduren unterscheiden zu können, vermeiden Sie das Präfix **sp_**, wenn Sie den Namen für die Prozedur festlegen.

14.2 Ausführen von gespeicherten Prozeduren

Eine gespeicherte Prozedur kann allein oder als Teil einer Insert- Anweisung ausgeführt werden. Zum Ausführen einer Prozedur muss die Execute- Berechtigung erteilt worden sein.

Syntax:

```

[[exec [ute]]
{[@return_status =] {prozedurname [;number] | @prozedur_name_var }
[[@parameter =] {value | @variable [output] | [default]] [,...n]
[with <execute_optionen>]
[]]

<execute_optionen>::=
{
    recompile
    | result sets undefined
    | result sets none
    | result sets (<ergebnis_definition> [, ...])
}

<ergebnis_definition>::=
{
    (column_name data_type [{null | not null}] [,...])
    | as object {tab_name | view_name | funkt_name}
    | as type table_type_name
    | as for xml
}

```

Execute führt eine benutzerdefinierte Skalarfunktion, eine Systemprozedur, eine benutzerdefinierte gespeicherte Prozedur oder eine erweiterte gespeicherte Prozedur aus. Unterstützt auch die Ausführung einer Zeichenfolge innerhalb eines Transact- SQL- Batches.

Hinweis:

Sie müssen das EXECUTE- Schlüsselwort beim Ausführen von gespeicherten Prozeduren nicht angeben, wenn es sich dabei um die erste Anweisung in einem Batch handelt.

Beispiel:

Ausführen der gespeicherten Prozedur "lief_datum"

```
execute lief_datum;
```

Mit der Insert- Anweisung kann eine lokale Tabelle mit einem Ergebnis aufgefüllt werden, das von einer lokalen oder einer remote gespeicherten Prozedur zurückgegeben wird.

Beispiel:

Die im Jahr 1990 durchgeföhrten Lieferungen werden in eine Tabelle aufgenommen.

```

create procedure lief_history
as
select a.lnr, lname, convert(char(10),ldatum,104), lmenge, b.anr, aname astadt
from lieferant a, artikel b, lieferung c
where a.lnr = c.lnr
and b.anr = c.anr
and ldatum between '01.01.1990' and '31.12.1990';
go

insert into lieferungen_1990 execute lief_history;

```

Einen Bericht über die Objekte die mit der Prozedur verbunden sind erhalten Sie durch eine Abfrage der Katalogsicht **sys.sql_dependencies** oder verwenden Sie die gespeicherte Prozedur **sp_depends**. Zum Anzeigen von Informationen über CLR- gespeicherte Prozeduren wird die Katalogsicht **sys.assembly_modules** verwendet.

Benötigen Sie einen Bericht zu Informationen der in einer Prozedur definierten Parameter dann sollten Sie die **sys.parameters**- Katalogsicht verwenden.

14.3 Ändern und Löschen von gespeicherten Prozeduren

Um eine gespeicherte Prozedur zu ändern, verwenden Sie die ALTER PROCEDURE- Anweisung. Sie können immer nur eine gespeicherte Prozedur ändern.

Syntax:

```
alter proc [edure] [schema.] prozedur_name [;number]
[{@parameter data_type} [varying] [= default] [output]] [...n]
[with [RECOMPILE] [ENCRYPTION] [EXECUTE_AS_Klausel]]
[for replication]
as
proc_body | EXTERNAL NAME assembly_name.class_name.method_name
[;]

<EXECUTE_AS_Klausel> ::= 
    { EXEC | EXECUTE } AS { CALLER | SELF | OWNER | 'user_name' }
```

Die Berechtigung zum Ausführen dieser Anweisung ist standardmäßig den Erstellern der Prozedur sowie den Mitgliedern der Serverrolle sysadmin und den Mitgliedern der festen Datenbankrollen db_owner und db_ddladmin vorbehalten.

Beispiel:

```
use standard;
go
alter procedure lief_datum
as
select lname, lstadt, lmenge
from lieferant a, lieferung b
where a.lnr = b.lnr
and ldatum > '12.05.1990' and status < 10;
go
```

Um eine benutzerdefinierte gespeicherte Prozedur zu Löschen führen Sie die DROP PROCEDURE- Anweisung aus. Sie sollten vorher mit der gespeicherten Prozedur **sp_depends** oder durch eine Abfrage mit der Katalogsicht **sys.sql_dependencies** prüfen welche Objekte von der Prozedur abhängig sind.

Syntax:

```
drop procedure [schema.] procedurname [...n];
```

Beispiel:

```
drop procedure lief_datum, lief_history;
```

14.4 Eingabeparameter in gespeicherten Prozeduren

Eingabeparameter erweitern die Funktionalität von gespeicherten Prozeduren. Es können sowohl Parameter an eine Prozedur übergeben als auch Parameter von einer Prozedur zurückgegeben werden. Man spricht von Eingabe- und Ausgabeparametern.

- Alle eingehenden Parameterwerte sollten am Anfang einer Prozedur überprüft werden, um fehlende und ungültige Werte frühzeitig aufzuspüren.
- Es sollten zweckmäßige Standardwerte für die Parameter festgelegt werden.
- Die maximale Anzahl der Parameter die einer gespeicherten Prozedur übergeben werden können, beträgt 1024.
- Die maximale Anzahl lokaler Variablen in einer Prozedur ist nur durch die Größe des Arbeitsspeichers begrenzt.

14.4.1 Erstellen gespeicherter Prozeduren mit Eingabeparametern

Parameter werden lokal in einer gespeicherten Prozedur verwendet. Daraus folgt, die gleichen Parameternamen können auch in anderen Prozeduren oder in den aufrufenden Batches verwendet werden.

Wenn die Prozedur abgearbeitet ist, dann sind auch alle in ihr erstellten Variablen unbekannt. Soll durch die Prozedur ein Zwischenergebnis abgespeichert werden dann eignet sich dazu eine permanente Tabelle oder globale temporäre Tabelle.

Teilsyntax:

```
create procedure name [{@parameter data_type } [varying] [= default] [,...n]
as ...
```

Beispiel:

Die Prozedur soll den Wohnort und Namen eines bestimmten Lieferanten zusammen mit der Anzahl der von ihm durchgeföhrten Lieferungen zurückgeben.

```
create procedure lief_anz @nr char(3) = 'L00'
as
if not exists(select * from lieferant where lnr = @nr)
begin
raiserror('Den Lieferanten gibt es nicht!',10,1);
return;
end;
select lname, lstadt, count(*) as 'Lieferungen'
from lieferant a join lieferung b on a.lnr = b.lnr
where a.lnr = @nr
group by lname, lstadt;
if @@rowcount = 0
begin
raiserror('Der Lieferant hat nicht geliefert!', 10,1);
return;
end;
go
```

14.4.2 Ausführen gespeicherter Prozeduren mit Eingabeparametern

Parameter können einer Prozedur entweder durch Parameternamen oder durch die Position übergeben werden. Die unterschiedlichen Formate sollten nicht kombiniert werden.

Syntax:

```
exec[ute] [@return_status = ] {prozedur_name | @prozedurname_var}
[[@parameter = ] {wert | @variable [output] | [default]}] [,...n]
[with recompile];
```

14.4.2.1 Übergeben von Werten durch Position

Diese Option wird verwendet wenn der Prozedur nur ein Wert übergeben wird bzw. wenn die Reihenfolge der übergebenen Parameter der Reihenfolge der Parameterdeklaration in der CREATE PROCEDURE- Anweisung entspricht.

Es können Parameter ausgelassen werden, wenn für diese Standardwerte vorhanden sind, die Reihenfolge der Parameter darf aber nicht unterbrochen werden. Das bedeutet, am Ende der Parameterkette können Parameter weggelassen werden aber nicht mittendrin.

Beispiel:

```
create procedure art_lief_anz @beg_dat datetime = null, @end_dat datetime = null,
    @anr char(3) = 'A01'
as
if @beg_dat is null and @end_dat is null
begin
    set @beg_dat = '01.01.' + cast(datepart(yyyy ,getdate())as char(4));
    set @end_dat = '31.12.' + cast(datepart(yyyy ,getdate())as char(4));
end;
select sum(lmenge) as 'Stückzahl'
from lieferung
where ldatum between @beg_dat and @end_dat
and anr = @anr
group by anr;
go
```

Aufrufender Stapel:

```
exec art_lief_anz '01.04.1990', '31.08.1990', 'A03';
```

Die Werte können im aufrufenden Programmstapel auch vorher Variablen übergeben werden, dann müssen in der EXECUTE- Anweisung die entsprechenden Variablen stehen.

14.4.2.2 Übergeben von Werten durch Parameternamen

Bei dieser Variante der Übergabe von Werten können die Parameterwerte in einer beliebigen Reihenfolge angegeben werden. Die Parameternamen müssen den Namen der Parametervariablen in der aufgerufenen Prozedur entsprechen. Sie müssen im aufrufenden Stapel nicht deklariert werden.

Beispiel:

```
exec art_lief_anz @end_dat = '31.12.2004', @anr = 'A02', @beg_dat = '01.01.2004';
```

14.4.3 Zurückgeben von Werten

Mit Hilfe von Ausgabeparametern die mit dem Schlüsselwort OUTPUT gekennzeichnet werden, können gespeicherte Prozeduren Informationen an einen aufrufenden Stapel oder eine aufrufende gespeicherte Prozedur zurückgeben.

Weiterhin kann eine gespeicherte Prozedur mit Hilfe einer RETURN- Anweisung einen Wert zurückgeben.

14.4.3.1 Return- Anweisung

Mit der RETURN- Anweisung kann eine Ganzzahl an ein aufrufendes Programm zurückgegeben werden, die dort dann entsprechend ausgewertet werden kann.

Syntax:

```
return(ganzzahl);
```

Beispiel:

```
create procedure anz_lief @nr char(3)
as
if not exists(select * from lieferung where lnr = @nr)
begin
    return(2);
end
select count(*)
from lieferung
where lnr = @nr;
go
```

Aufrufender Stapel:

```
declare @rw integer;
exec @rw = anz_lief 'L02';
if @rw = 2
begin
    raiserror ('Der Lieferant hat nicht geliefert!', 10, 1);
end;
go
```

14.4.3.2 Ausgabeparameter

Um mit Ausgabeparametern arbeiten zu können, muss das OUTPUT- Schlüsselwort sowohl in der CREATE PROCEDURE- Anweisung als auch in der EXECUTE- Anweisung angegeben werden.

Wird Das Schlüsselwort beim Aufrufen der Prozedur nicht verwendet, wird die Prozedur zwar ausgeführt gibt aber keinen Wert zurück.

Hinweise:

- Die Rückgabeparameter dürfen nicht vom Datentyp TEXT oder IMAGE sein.
- Es darf eine Variable vom Datentyp CURSOR sein.
- Die Variablen können in weiteren SQL- Anweisungen des Stapels verwendet werden.
- Die Rückgabevariable kann keine Tabellenvariable sein.

Beispiel:

```

create procedure liefanz @nr char(3) = 'L01', @name varchar(20) output,
    @ort varchar(30) output, @status int output
as
if not exists(select * from lieferant where Inr = @nr)
begin
    raiserror('Den Lieferanten gibt es nicht!',10,1);
    return;
end
select @name = lname, @ort = lstadt, @anz = status
from lieferant
where Inr = @nr;
go

```

Aufrufender Stapel:

```

declare @name varchar(25), @stadt varchar(25), @status int;
exec liefanz 'L01', @name output, @stadt output, @status output;
print 'Name: ' + @name;
print 'Wohnort: ' + @stadt;
print 'Status: ' + @status;
go

```

14.5 Erneutes Kompilieren gespeicherter Prozeduren

Das explizite Kompilieren von gespeicherten Prozeduren ist möglich sollte aber nur durchgeführt werden wenn:

- die übergebenen Parameter stark voneinander abweichende Ergebnisse zurückliefern.
- an den zugrunde liegenden Tabellen signifikante Änderungen durchgeführt wurden.
- der, der Prozedur, übergebene Parameterwert untypisch ist.

Das Rekompilieren einer Prozedur kann auf zwei Wegen festgelegt werden. Erstens bei der Erstellung der Prozedur oder zweitens beim Aufruf der Prozedur. Dazu wird das Schlüsselwort WITH RECOMPILE verwendet.

Mit der gespeicherten Systemprozedur **sp_recompile** kann eine gespeicherte Prozedur oder ein Trigger angegeben werden, der bei der nächsten Ausführung erneut kompiliert werden soll. Sie sollte verwendet werden wenn eine der Prozedur zugrunde liegende Tabelle geändert wurde.

14.6 CLR- Prozeduren

Seit SQL Server 2005 können Sie ein Datenbankobjekt wie eine benutzerdefinierte Prozedur innerhalb einer Instanz von SQL Server erstellen, das in einer Assembly programmiert ist, das in Microsoft .NET Framework CLR (Common Language Runtime) erstellt ist

Dazu müssen folgende Schritte ausgeführt werden:

- Definieren der Prozedur als statische Methode einer Klasse in einer Sprache, die von .NET Framework unterstützt wird. Kompilieren Sie die Klasse mithilfe des entsprechenden Sprachcompilers, um eine Assembly in .NET Framework zu erstellen.
- Registrieren der Assembly in SQL Server mithilfe der CREATE ASSEMBLY-Anweisung.
- Erstellen der Prozedur, die auf die registrierte Assembly verweist, mithilfe der CREATE PROCEDURE- Anweisung.

15 Benutzerdefinierte Funktionen

Eine Funktion besteht aus einer oder mehreren SQL- Anweisungen. Sie kapselt den Code zur Wiederverwendung. Neben den im SQL Server integrierten Funktionen hat der Benutzer die Möglichkeit eigene benutzerdefinierte Funktionen zu erstellen.

Benutzerdefinierte Funktionen können in Transact- SQL oder einer anderen .NET- Programmiersprache geschrieben werden.

Benutzerdefinierte Funktionen können aus einer Abfrage heraus aufgerufen werden. Darüber hinaus können benutzerdefinierte Skalarfunktionen, genau wie Prozeduren, mit einer EXECUTE- Anweisung ausgeführt werden.

Benutzerdefinierte Funktionen werden mit der ALTER FUNCTION- Anweisung geändert und mit der DROP FUNCTION- Anweisung gelöscht.

Vorteile:

- Modulare Programmierung.
- Schnelle Ausführung.
- Reduzierung des Netzwerkverkehrs.

Einschränkungen:

- Zum Erstellen einer Funktion wird die Anweisungsberechtigung CREATE FUNCTION in der Datenbank und die ALTER- Berechtigung für das Schema in dem die Funktion erstellt wird, benötigt. Gibt die Funktion einen benutzerdefinierten Typ an, wird die EXECUTE- Berechtigung für den Typ benötigt.
- Zum Erstellen oder Ändern von Tabellen mit Verweisen auf benutzerdefinierte Funktionen (in der CHECK- Einschränkung, der DEFAULT- Einschränkung oder der Definition berechneter Spalten) wird die Objektberechtigung REFERENCES für die Funktion benötigt.
- Mit benutzerdefinierten Funktionen können keine Aktionen ausgeführt werden, die den Status der Datenbank ändern.
- Die EXECUTE AS- Klausel kann nicht bei benutzerdefinierten Inlinefunktionen angegeben werden.
- Die TRY...CATCH- Anweisung ist in benutzerdefinierten Funktionen nicht zulässig.
- Eine Funktion wird mit der Anweisung CREATE FUNCTION erstellt, ALTER FUNCTION geändert und mit DROP FUNCTION gelöscht.
- In Funktionen bewirken SQL- Fehler, dass die Ausführung der Funktion beendet wird. Dies hat zur Folge, dass die Anwendung abgebrochen wird, die die Funktion aufgerufen hat.
- Einer Funktion können mehrere Eingabeparameter übergeben werden, kann aber auch ohne Parameter ausgeführt werden. Die maximale Anzahl an Eingabeparameter beträgt 1024.
- Die Schachtelungsebene darf 32 nicht überschreiten.
- Wenn ein Parameter der Funktion über einen Standardwert verfügt, muss beim Aufrufen der Funktion das DEFAULT- Schlüsselwort angegeben werden. Dieses Verhalten unterscheidet sich von Parametern mit Standardwerten bei gespeicherten Prozeduren.
- Skalarfunktionen geben einen einzigen Datenwert des in der RETURN- Klausel definierten Datentyps zurück, einschließlich bigint und sql_variant. Benutzerdefinierte Datentypen, nicht skalare Datentypen (cursor, table) und der Datentyp timestamp werden nicht unterstützt.

Folgende Anweisungstypen sind zulässig:

- DECLARE- Anweisungen zum Definieren von lokalen Variablen und Cursorn für die Funktion.
- Zuweisungen von Werten zu lokalen Objekten für die Funktion, wie z. B. das Zuweisen von Werten zu lokalen Skalar- und Tabellenwerten mithilfe von SET.
- Cursoroperationen, die auf lokale Cursor verweisen, die in der Funktion deklariert, geöffnet, geschlossen und freigegeben werden. FETCH- Anweisungen, die Daten an den Client zurückgeben, sind nicht zulässig. Nur FETCH- Anweisungen, die lokalen Variablen Werte mithilfe der INTO- Klausel zuweisen, sind erlaubt.
- Anweisungen zur Ablaufsteuerung (IF, WHILE).
- SELECT- Anweisungen, deren Auswahllisten Ausdrücke enthalten, die lokalen Variablen der Funktion Werte zuweisen.
- UPDATE-, INSERT- und DELETE- Anweisungen, die lokale TABLE- Variablen der Funktion ändern.
- EXECUTE- Anweisungen, die eine erweiterte gespeicherte Prozedur aufrufen.

Hinweis:

Eine Funktion die beim wiederholten Aufruf mit bestimmten Eingabewerten immer das gleiche Ergebnis zurückliefert, ist deterministisch. Eine Funktion die bei jedem Aufruf mit den gleichen Eingabewerten unterschiedliche Ergebnisse liefert, ist nicht deterministisch. Nicht deterministische Funktionen können Nebenwirkungen verursachen. Integrierte nicht deterministische Funktionen sind im Funktionskörper einer benutzerdefinierten Funktion nicht erlaubt.

Funktionen die erweiterte gespeicherte Prozeduren aufrufen, werden als nicht deterministische Funktionen angesehen, da sie Nebenwirkungen für die Datenbank verursachen können.

15.1 Arten von benutzerdefinierten Funktionen

SQL Server unterstützt zwei Typen von benutzerdefinierten Funktionen, Skalarfunktionen und Tabellenwertfunktionen (Inlinefunktionen) und Funktionen mit Tabellenrückgabe und mehreren Anweisungen.

15.1.1 Skalarfunktionen

Skalarfunktionen verarbeiten einen einzelnen Wert oder mehrere Werte und geben einen einzelnen Wert zurück. Sie ähneln integrierten Funktionen. Eine oder mehrere SQL-Anweisungen stehen im Funktionskörper in einem BEGIN...END- Block. Skalare Funktionen können überall dort verwendet werden, wo in der SELECT- Anweisung ein Ausdruck zulässig ist.

Syntax:

```
CREATE FUNCTION [schema.] function_name
([@parameter_name [AS] scalar_parameter_data_type [= default] [...n]])
RETURNS scalar_return_data_type
[WITH <funktions_optionen>]
[AS]
BEGIN
function_body
RETURN scalar_expression
END
[;]
```

```
<funktions_optionen>::=
{
    [ENCRYPTION]
    [SCHEMABINDING]
    [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]
    [EXECUTE_AS Klausel]
}
```

Es können alle Datentypen einschließlich CLR- benutzerdefinierte Typen außer TEXT, NTEXT, IMAGE, CURSOR oder TIMESTAMP zurückgegeben werden.

Beispiel:

Eine Funktion die ermitteln soll wie oft ein bestimmter Artikel in einem bestimmten Zeitraum geliefert wurde.

```
create function art_anz (@nr char(3), @beg_dat datetime, @end_dat datetime)
returns integer
as
begin
    declare @anz integer
    select @anz = count(anr)
    from lieferung
    where anr = @nr
    and ldatum between @beg_dat and @end_dat
    group by anr;
    return @anz;
end;
select dbo.art_anz('A02', '01.01.90', '28.10.90');
```

15.1.2 Inlinefunktionen mit Tabellenrückgabe

Diese Funktionen geben eine Tabelle zurück. Sie ermöglichen die Funktionalität parametrisierter Sichten. Im Funktionskörper gibt es keinen BEGIN...END- Block und die RETURN- Klausel enthält in Klammern eine einzige SELECT- Anweisung. RETURNS gibt TABLE als Datentyp an. Sie wird wie eine Sicht in der FROM- Klausel einer Abfrage verwendet.

Syntax:

```
CREATE FUNCTION [schema.] function_name
([@parameter_name [AS] scalar_parameter_data_type [= default] [ ,...n]])
RETURNS TABLE
[WITH <funktions_optionen>]
[AS]
RETURN ( select-anweisung )
[]

<funktions_optionen>::=
{
    [ENCRYPTION]
    [SCHEMABINDING]
    [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]
    [EXECUTE_AS Klausel]
}
```

Beispiel:

Eine Funktion die die Namen und Wohnorte der Lieferanten zurückgibt, die in einem bestimmten Zeitraum geliefert haben.

```
create function lief_inf (@beg_dat datetime, @end_dat datetime)
returns table
as
return  (
    select distinct lname, lstadt
    from lieferant a, lieferung b
    where a.lnr = b.lnr
    and ldatum between @beg_dat and @end_dat
);
select * from lief_inf('01.01.90', '28.10.90');
```

In einer Sicht darf kein benutzerdefinierter Parameter eingeschlossen werden. Bedingungen können in Sichten nur in einer WHERE- Klausel festgelegt werden und sind damit nicht mehr dynamisch änderbar. Mit einer Inlinefunktion kann dieses Problem behoben werden.

Inlinefunktionen können die Leistung von Abfragen erheblich steigern, wenn sie mit indizierten Sichten verwendet werden.

15.1.3 Funktion mit mehreren Anweisungen und Tabellenrückgabe

Diese Funktion stellt eine Kombination einer Sicht mit einer gespeicherten Prozedur dar. Der Funktionskörper kann eine komplexe Logik und mehrere SQL- Anweisungen zum Erstellen einer Tabelle enthalten. Sie wird wie eine Sicht in der FROM- Klausel einer Abfrage verwendet. Ein BEGIN...END- Block begrenzt den Funktionskörper der Funktion. Die RETURNS- Klausel gibt TABLES für den Rückgabewert an die einen Namen und eine benutzerdefinierte Spaltendefinition enthält.

Syntax:

```
CREATE FUNCTION [schema.] function_name
([@parameter_name [AS] scalar_parameter_data_type [= default] [, ...n]])
RETURNS @returnvar TABLE ( table_type_definition )
[WITH <funktions_optionen>]
[AS]
BEGIN
function_body
RETURN
END
[:]

<funktions_optionen>::=
{
    [ENCRYPTION]
    [SCHEMABINDING]
    [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]
    [EXECUTE_AS Klausel]
}
```

Beispiel:

Diese Funktion gibt abhängig vom bereitgestellten Parameter den Namen und Wohnort der Lieferanten zurück oder den Artikelnamen und den Lagerort.

```
create function uebersicht (@wert varchar(50) = 'Lieferanten')
returns @nam_ort table
(
    nummern char(3) primary key not null,
    namen varchar(25) not null,
    ort varchar(25) null
)
as
begin
    if @wert = 'Lieferanten'
        insert into @nam_ort select Inr, Iname, Istadt from lieferant;
    if @wert = 'Artikel'
        insert into @nam_ort select anr, aname, astadt from Artikel;
    return;
end
select * from dbo.uebersicht('Artikel');

select * from dbo.uebersicht('Lieferanten');

select * from dbo.uebersicht(default);
```

15.2 Schemagebundene Funktionen

CREATE FUNCTION unterstützt eine SCHEMABINDING- Klausel, die die Funktion an das Schema von Objekten bindet, auf die verwiesen wird. Der Versuch, ein Objekt zu ändern oder zu löschen, auf die eine Funktion verweist, schlägt fehl.

Dabei müssen folgende Bedingungen erfüllt sein:

- Alle Views und benutzerdefinierten Funktionen, auf die die Funktion verweist, müssen schemagebunden sein.
- Alle Objekte, auf die die Funktion verweist müssen sich in derselben Datenbank befinden wie die Funktion.
- Für alle Objekte auf die sich die Funktion bezieht, wird die REFERENCES-Berechtigung benötigt.

15.3 Leistungsaspekte

Beim Implementieren von benutzerdefinierten Funktionen sollten folgende Methoden berücksichtigt werden.

- Komplexe Skalarfunktionen werden für kleine Resultsets verwendet.
- Anstelle von gespeicherten Prozeduren die Tabellen zurückgeben sollten Funktionen mit Tabellenrückgabe verwendet werden. Das kann zu Leistungsverbesserungen führen.
- Inlinefunktionen werden verwendet um Sichten mit Hilfe von Parametern zu erstellen. Parameter können Verweise auf Tabellen und Sichten vereinfachen. In Views sind keine Parameter erlaubt.
- Mit Inlinefunktionen können Sichten gefiltert werden. Die Verwendung von Inlinefunktionen mit indizierten Sichten kann die Leistung erheblich verbessern.

15.4 Funktionen ändern

Die Syntax zum Ändern von Funktionen ist analog der Syntax zum Erstellen von Funktionen nur dass das Schlüsselwort CREATE durch ALTER ausgetauscht wird.

ALTER FUNCTION kann nicht verwendet werden, um eine Skalarfunktion in eine Tabellenwertfunktion oder umgekehrt zu ändern. Ebenso kann ALTER FUNCTION nicht verwendet werden, um eine Inlinefunktion in eine Funktion mit mehreren Anweisungen oder umgekehrt zu ändern. ALTER FUNCTION kann nicht zum Ändern einer Transact-SQL-Funktion in eine CLR-Funktion und umgekehrt verwendet werden.

15.5 Funktionen löschen

Syntax:

```
drop [schema.]funktions_name [, ...]
```

Um eine Funktion löschen zu können, benötigt ein Benutzer mindestens ALTER-Berechtigungen für das Schema, zu dem die Funktion gehört, oder CONTROL-Berechtigungen für die Funktion.

Es wird ein Fehler erzeugt, wenn Transact- SQL- Funktionen oder - Sichten in der Datenbank vorhanden sind, die auf diese Funktion verweisen und mithilfe von SCHEMABINDING erstellt wurden, oder wenn berechnete Spalten, CHECK- Einschränkungen oder DEFAULT- Einschränkungen vorhanden sind, die auf die Funktion verweisen.

DROP FUNCTION erzeugt außerdem einen Fehler, wenn berechnete Spalten vorhanden sind, die auf diese Funktion verweisen und indiziert wurden.

15.6 CLR- Funktionen

Seit SQL Server 2005 können Sie ein Datenbankobjekt wie eine benutzerdefinierte Funktion innerhalb einer Instanz von SQL Server erstellen, das in einer Assembly programmiert ist, das in Microsoft .NET Framework CLR (Common Language Runtime) erstellt ist

Syntax:

```
CREATE FUNCTION [schema.] function_name
([{{@parameter_name [AS] scalar_parameter_data_type [= default]}[,...n]}])
RETURNS { return_data_typ | TABLE ( clr_table_type_definition )
[WITH <clr_funktions_optionen>]
[AS] EXTERNAL NAME <methoden_spezifikation>
[:]

<clr_funktions_optionen>::=
{
|   [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]
|   [EXECUTE_AS Klausel]
}

<methoden_spezifikation>::=
assembly_name.class_name.method_name
```

Bei Rückgabe von TABLE werden nur der Spaltenname und der Datentyp angegeben.

Zum Erstellen einer CLR-Funktion in SQL Server müssen folgende Schritte ausgeführt werden:

- Definieren der Funktion als statische Methode einer Klasse in einer Sprache, die von .NET Framework unterstützt wird. Kompilieren Sie die Klasse mithilfe des entsprechenden Sprachcompilers, um eine Assembly in .NET Framework zu erstellen.
- Registrieren der Assembly in SQL Server mithilfe der CREATE ASSEMBLY-Anweisung.
- Erstellen der Funktion, die auf die registrierte Assembly verweist, mithilfe der CREATE FUNCTION-Anweisung.

16 Trigger

Ein Trigger ist eine spezielle Form einer gespeicherten Prozedur, die immer dann ausgeführt wird, wenn Daten in einer angegebenen Tabelle (Triggertabelle) geändert werden.

Trigger werden häufig erstellt, um die referentielle Integrität, die Konsistenz von logisch verknüpften Daten in unterschiedlichen Tabellen oder Geschäftsregeln, mit denen Datenintegrität sichergestellt wird, durchzusetzen.

Eigenschaften:

- Trigger werden immer für eine bestimmte Tabelle definiert.
- Trigger werden automatisch aufgerufen wenn auf die Trigger Tabelle eine spezielle Änderungsaktion, für die ein Trigger definiert wurde, ausgeführt wird.
- Trigger können nicht direkt aufgerufen werden.
- Trigger sind immer Teil der auslösenden Transaktion.
Trigger können im Gegensatz zu CHECK- Constraints auf Spalten anderer Tabellen verweisen. Dadurch können komplexere Einschränkungen definiert werden.
- Der Benutzer der den Trigger auslöst, muss in der Lage sein alle Anweisungen des Triggers auf andere Tabellen auszuführen.

Trigger können direkt mit Transact- SQL- Anweisungen oder mit Methoden von Assemblies erstellt werden, die in der Microsoft .NET Framework CLR erstellt und auf eine Instanz von SQL Server geladen werden können.

16.1 DML- Trigger erstellen

Es gibt zwei Arten von DML- Trigger, AFTER Trigger und INSTEAD OF- Trigger.

AFTER Trigger werden nach der DML- Anweisung ausgeführt.

INSTEAD OF- Trigger werden anstelle der auslösenden DML- Anweisung ausgeführt.

Syntax:

```
CREATE TRIGGER trigger_name
ON { tabelle | view }
[ {WITH ENCRYPTION | EXECUTE_AS_Klausel} ]
{ FOR | AFTER | INSTEAD OF } { [ INSERT ] [,][ UPDATE ] [,][ DELETE ] }
[NOT FOR REPLICATION]
AS
[ IF UPDATE ( column ) [{ AND | OR } UPDATE ( column )][...n]]
{sql_anweisung(en) | EXTERNAL NAME <method_spez.>}

<method_spez.>::=
assembly_name.class_name.method_name
```

Einschränkungen:

- Die CREATE TRIGGER- Anweisung muss die erste Anweisung in einem Batch sein und kann sich nur auf eine Tabelle oder View beziehen.
- Sind Einschränkungen für eine Trigger Tabelle definiert werden diese bei AFTER Triggern zuerst geprüft und bei INSTEAD OF- Triggern nach der Triggerausführung.
- Tabellen können mehrere Trigger für beliebige Aktionen besitzen. Mit der gespeicherten Prozedur **sp_settriggerorder** kann der erste und letzte auszulösende Trigger festgelegt werden.

- Ein Trigger kann nur in der aktuellen Datenbank erstellt werden; er darf jedoch auf Objekte außerhalb der aktuellen Datenbank verweisen.
- Es ist möglich, eine CREATE TRIGGER- Anweisung für mehrere Benutzeraktionen festzulegen (z.B. INSERT und UPDATE).
- Es kann immer nur ein INSTEAD OF- Trigger pro INSERT | DELETE | UPDATE- Anweisung für eine Tabelle oder Sicht erstellt werden.
- Ein INSTEAD OF- Trigger kann nicht auf aktualisierbare Sichten erstellt werden die mit WITH CHECK OPTION erstellt wurden.
- INSTEAD OF- Trigger für die Operationen UPDATE / DELETE können nicht für eine Tabelle definiert werden, die einen Fremdschlüssel mit der Lösch- oder Änderungsregel ON CASCADE besitzt.
- Es können keine AFTER- Trigger für Sichten oder temporäre Tabellen erstellt werden. Sie können aber auf Sichten oder temporäre Tabellen verweisen.
- Ein INSTEAD OF- Trigger kann für Sichten und Tabellen erstellt werden.
- In einem Trigger kann jede beliebige SET- Anweisung angegeben werden. Die gewählte SET- Option bleibt während der Ausführung des Triggers in Kraft und kehrt dann zur vorherigen Einstellung zurück.
- Es sollten in einem Trigger keine SELECT- Anweisungen verwendet werden, die Ergebnisse zurückgeben, oder Anweisungen, die Variablenzuweisungen durchführen. Wenn Variablenzuweisungen in einem Trigger erfolgen müssen, wird die SET NOCOUNT- Anweisung am Anfang des Triggers verwendet, um die Rückgabe von Resultsets zu verhindern.
- Eine TRUNCATE TABLE- Anweisung wird von einem DELETE- Trigger nicht berücksichtigt. Eine TRUNCATE TABLE- Anweisung wirkt sich zwar genauso aus wie eine DELETE- Anweisung ohne WHERE- Klausel, sie wird jedoch nicht protokolliert und kann deshalb keinen Trigger ausführen.
- Die WRITETEXT- Anweisung, ob protokolliert oder nicht protokolliert, aktiviert keinen Trigger.
- Die folgenden SQL- Anweisungen sind in einem Trigger nicht zulässig: ALTER DATABASE, CREATE DATABASE, DISK INIT, DISK RESIZE, DROP DATABASE, LOAD DATABASE, LOAD LOG, RECONFIGURE, RESTORE DATABASE, RESTORE LOG
- Der SQL Server unterstützt keine Trigger für Systemtabellen.

Informationen über Trigger werden in den Katalogsichten **sys.objects**, **sys.sql_modules**, **sys.triggers** und **sys.trigger_events** gespeichert.

Um einen Trigger zu erstellen ist die ALTER- Berechtigung für die Tabelle oder Sicht für die der Trigger erstellt wird erforderlich. Die Berechtigung ist nicht übertragbar.

Mit der gespeicherten Systemprozedur **sp_depends** kann ermittelt werden welche Trigger für eine Tabelle definiert wurden. Weitere Systemprozeduren sind **sp_helptext** und **sp_helptrigger**.

Trigger können maximal 32 Ebenen tief geschachtelt werden. Mit **sp_configure** kann die Serveroption "**nested triggers**" auf **0** gesetzt werden um geschachtelte Trigger zu deaktivieren.

Trigger können auch indirekt rekursiv oder direkt rekursiv aufgerufen werden. Die direkte Rekursion wird erreicht indem mit der ALTER DATABASE- Anweisung die RECURSIVE_TRIGGER- Einstellung aktiviert.

Die indirekte Rekursion wird mit **sp_configure** verhindert indem die Serveroption "**nested triggers**" auf **0** gesetzt wird.

Trigger werden mit der ALTER TRIGGER- Anweisung geändert und mit der DROP TRIGGER- Anweisung gelöscht. Trigger werden automatisch gelöscht wenn die Trigger Tabelle gelöscht wird.

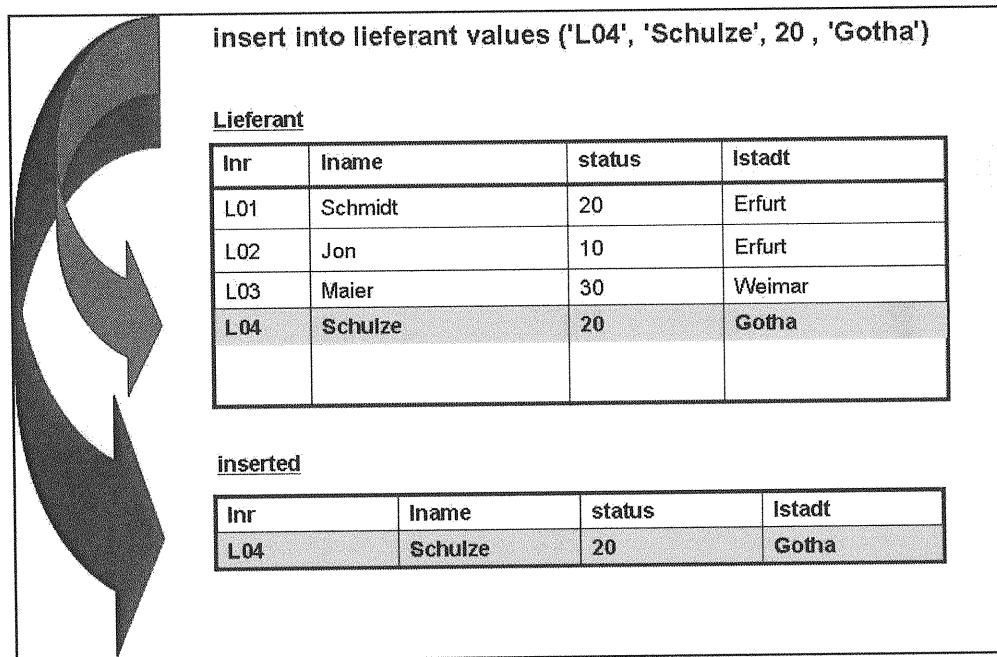
Mit der Anweisung ENABLED/DISABLED TRIGGER oder ALTER TABLE (aus Gründen der Abwärtskompatibilität) können Trigger einer Tabelle deaktiviert oder auch wieder aktiviert werden. Dabei geht die Definition des Triggers nicht verloren.

16.1.1 Funktionsweise von DML- Triggern

Trigger werden automatisch angestoßen nachdem eine INSERT-, UPDATE- oder DELETE-Anweisung auf die Trigger Tabelle ausgeführt wurde.

16.1.1.1 INSERT- Trigger

Ein INSERT- Trigger wird ausgeführt wenn eine INSERT- Anweisung auf die Trigger Tabelle ausgeführt wird. Dabei wird eine logische Tabelle INSERTED gebildet, die ein genaues Abbild der Trigger Tabelle ist. Inhalt der Tabelle ist der Datensatz der durch die INSERT- Anweisung gerade in die Trigger Tabelle aufgenommen wurde.



Die Zeilen in INSERTED sind also immer Duplikate einer oder mehrerer Zeilen in der Trigger Tabelle. Die INSERTED- Tabelle ermöglicht es, auf die durch INSERT verursachten protokollierten Änderungen zu verweisen. Dadurch kann man auf eingefügte Daten verweisen, obwohl die Transaktion noch nicht beendet ist und ohne das die Information in Variable gespeichert sein muss.

Beispiel:

```
create trigger lieferant_neu
on lieferant
after insert
as
if (select status from inserted) < 0
begin
    raiserror('Status muss größer 0 sein!',10,1);
    rollback transaction;
end;
```

Dieses Beispiel überprüft ob der eingegebene Statuswert größer null ist. Diese Überprüfung macht man normalerweise mit einem CHECK- Constraint, weil die Leistung besser ist.

16.1.1.2 Delete Trigger

Ein DELETE- Trigger wird ausgeführt wenn eine DELETE- Anweisung auf die Trigger Tabelle ausgeführt wird. Dabei wird eine logische Tabelle DELETED gebildet, die ein genaues Abbild der Trigger Tabelle ist. Inhalt dieser Tabelle ist der Datensatz der durch die DELETE- Anweisung gerade aus der Trigger Tabelle gelöscht wurde.



delete from lieferant where Inr = 'L04'

| <u>Lieferant</u> | | | |
|------------------|---------|--------|--------|
| Inr | Iname | status | Istadt |
| L01 | Schmidt | 20 | Erfurt |
| L02 | Jon | 10 | Erfurt |
| L03 | Maier | 30 | Weimar |
| | | | |
| | | | |

| <u>deleted</u> | | | |
|----------------|---------|--------|--------|
| Inr | Iname | status | Istadt |
| L04 | Schulze | 20 | Gotha |

Die Zeilen in der Trigger Tabelle und der Tabelle DELETED sind also nicht identisch. Man kann sich auf die Daten in der DELETED- Tabelle beziehen obwohl die Transaktion noch nicht abgeschlossen wurde.

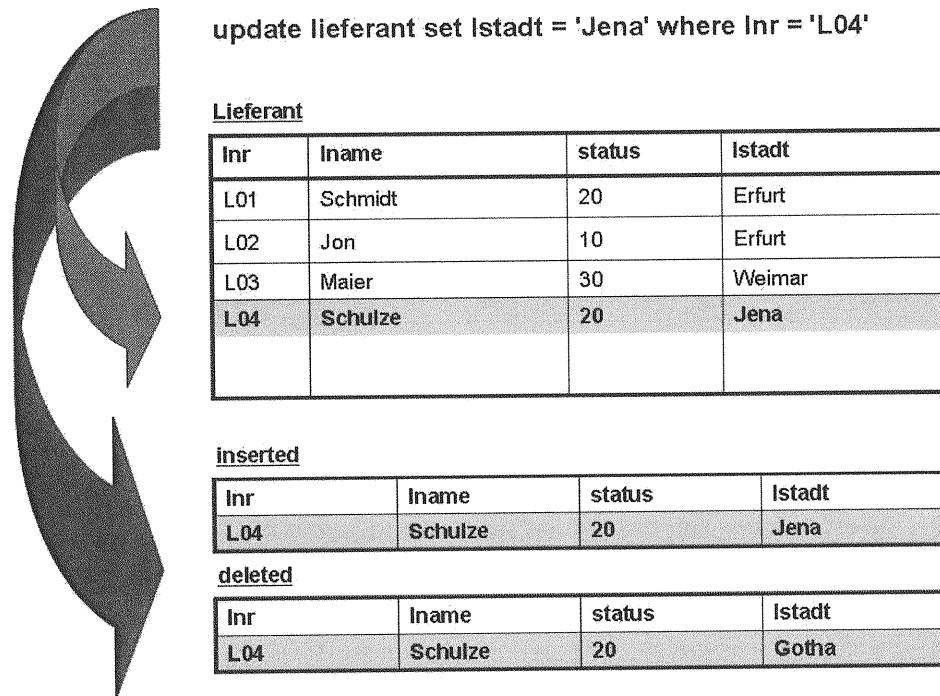
Beispiel:

```
create trigger lieferant_weg
on lieferant
after delete
as
if exists (select * from lieferung, deleted where lieferung.Inr = deleted.Inr)
begin
    raiserror('Löschen abgebrochen, der Lieferant hat noch Lieferungen!',10,1);
    rollback transaction;
end;
```

Dieses Beispiel lässt das Löschen eines Lieferanten nur dann zu wenn es für ihn keine eingetragenen Lieferungen mehr gibt. Auch diese Überprüfung macht man aus Gründen der Leistungssteigerung besser mit einer FOREIGN KEY- Einschränkung.

16.1.1.3 Update Trigger

Ein Update- Trigger wird ausgeführt wenn eine Update- Anweisung auf die Trigger Tabelle ausgeführt wird. Dabei wird eine logische Tabelle DELETED und eine logische Tabelle INSERTED gebildet, die beide ein genaues Abbild der Trigger Tabelle sind. Inhalt der INSERTED- Tabelle ist der Datensatz der durch die UPDATE- Anweisung gerade geändert wurde und der Inhalt der DELETED- Tabelle ist der Datensatz wie er vor der Änderung existiert hat.



Das heißt, der Inhalt der Trigger Tabelle und der Inhalt von INSERTED sind gleich. Der Inhalt der Trigger Tabelle und von DELETED sind verschieden.

Beispiel:

```
create trigger lieferant_update
on lieferant
for update
as
if update (Inr)
if exists(select * from lieferung, deleted where lieferung.Inr = deleted.Inr)
begin
    update lieferung
    set Inr = inserted.Inr
    from lieferung, inserted, deleted
    where lieferung.Inr = deleted.Inr;
end;
```

In diesem Beispiel soll erreicht werden, wenn die Lieferantennummer eines Lieferanten geändert wird und er hat noch eingetragene Lieferungen dann sollen diese Lieferungen mit seiner neuen Lieferantennummer versehen werden. Auch diesen Vorgang sollte man aus Gründen der Geschwindigkeit besser mit einer FOREIGN KEY- Einschränkung (kaskadierendes Ändern) realisieren.

16.1.1.4 INSTEAD OF- Trigger

Diese Trigger werden an Stelle der den Trigger auslösenden DML- Anweisung ausgeführt. Dadurch wird die auslösende Aktion außer Kraft gesetzt.

Sie bieten eine größere Vielfalt an Aktualisierungsoptionen, die für eine Sicht ausgeführt werden können.

Jede Tabelle oder Sicht ist auf einen INSTEAD OF- Trigger pro Trigger Aktion beschränkt.

Beispiel:

Ein Trigger der verhindert dass Lieferanten gelöscht werden und schließt im Anschluss alle eventuell noch offenen Transaktionen.

```
create trigger del_lieferant
on lieferant
instead of delete
as
begin
    set nocount on;
    declare @del_anz int;
    select @del_anz = count(*) from deleted;
    if @del_anz > 0
        begin
            raiserror('Lieferanten können nicht gelöscht werden.',10,1);
            insert into del_lief_hist values(suser_name(),getdate());
            if @@trancount > 0
                rollback transaction;
        end;
end;
```

16.2 DDL- Trigger erstellen

Seit SQL Server 2005 gibt es DDL- Trigger im Server- und Datenbankbereich. Diese Trigger werden angestoßen wenn auf einer Datenbank oder auf der Serverinstanz eine CREATE-, ALTER-, DROP-, GRANT-, DENY-, REVOKE- oder UPDATE STATISTICS- Anweisung ausgeführt wird. Bestimmte gespeicherte Systemprozeduren, die DDL- ähnliche Vorgänge ausführen, können ebenfalls DDL- Trigger auslösen. Dies sollten sie aber vorher prüfen da nicht alle gespeicherten Systemprozeduren einen Trigger auslösen.

Auf Ereignisse die sich auf lokale oder globale temporäre Tabellen auswirken reagieren DDL- Trigger nicht.

DDL- Trigger sind an keine Schemas gebunden.

Syntax:

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[WITH [ENCRYPTION] [,] [EXECUTE AS Klausel]]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS {sql_anweisung [;] [,...n] | EXTERNAL NAME <method spez. [;]> }

<method_spez.> ::= 
    assembly_name.class_name.method_name
```

DDL-Trigger können für folgende Ereignisse bzw. Ereignisgruppen erstellt werden.

| | | Server- bereich | Datenbank- bereich |
|--|---|--------------------|-----------------------|
| DDL_EVENTS | | | |
| DDL_SERVER_LEVEL_EVENTS | (CREATE DATABASE, ALTER DATABASE, DROP DATABASE, ALTER_INSTANCE) | x | |
| DDL_LINKED_SERVER_EVENTS | (CREATE_LINKED_SERVER, ALTER_LINKED_SERVER, DROP_LINKED_SERVER) | x | |
| DDL_LINKED_SERVER_LOGIN_EVENTS | (CREATE_LINKED_SERVER_LOGIN, DROP_LINKED_SERVER_LOGIN) | x | |
| DDL_REMOTE_SERVER_EVENTS | (CREATE_REMOTE_SERVER, ALTER_REMOTE_SERVER, DROP_REMOTE_SERVER) | x | |
| DDL_EXTENDED_PROCEDURE_EVENTS | (CREATE_EXTENDED_PROCEDURE, DROP_EXTENDED_PROCEDURE) | x | |
| DDL_MESSAGE_EVENTS | (CREATE_MESSAGE, ALTER_MESSAGE, DROP_MESSAGE) | x | |
| DDL_ENDPOINT_EVENTS | (CREATE ENDPOINT, ALTER ENDPOINT, DROP ENDPOINT) | x | |
| DDL_SERVER_SECURITY_EVENTS | (ADD ROLE MEMBER, DROP ROLE MEMBER, ADD SERVER ROLE MEMBER, DROP SERVER ROLE MEMBER) | x | |
| DDL_LOGIN_EVENTS | (CREATE LOGIN, ALTER LOGIN, DROP LOGIN) | x | |
| DDL_GDR_SERVER_EVENTS | (GRANT SERVER, DENY SERVER, REVOKE SERVER) | x | |
| DDL_AUTHORIZATION_SERVER_EVENTS | (ALTER AUTHORIZATION SERVER) | x | |
| DDL_DATABASE_LEVEL_EVENTS | | x | x |
| DDL_FULLTEXT_CATALOG_EVENTS | (CREATE_FULLTEXT_CATALOG, ALTER_FULLTEXT_CATALOG, DROP_FULLTEXT_CATALOG) | x | x |
| DDL_DEFAULT_EVENTS | (CREATE_DEFAULT, DROP_DEFAULT, BIND_DEFAULT) | x | x |
| DDL_EXTENDED_PROPERTY_EVENTS | (CREATE_EXTENDED_PROPERTY, DROP_EXTENDED_PROPERTY) | x | x |
| DDL_PLAN_GUIDE_EVENTS | (CREATE_PLAN_GUIDE, ALTER_PLAN_GUIDE, DROP_PLAN_GUIDE) | x | x |
| DDL_RULE_EVENTS | (CREATE_RULE, DROP_RULE, BIND_RULE) | x | x |
| DDL_TABLE_VIEW_EVENTS | | x | x |
| DDL_TABLE_EVENTS | (CREATE TABLE, ALTER TABLE, DROP TABLE) | x | x |
| DDL_VIEW_EVENTS | (CREATE VIEW, ALTER VIEW, DROP VIEW) | x | x |
| DDL_INDEX_EVENTS | (CREATE INDEX, ALTER INDEX, DROP INDEX, CREATE XML INDEX, CREATE_FULLTEXT_INDEX, ALTER_FULLTEXT_INDEX, DROP_FULLTEXT_INDEX) | x | x |
| DDL_STATISTICS_EVENTS | (CREATE STATISTICS, UPDATE STATISTICS, DROP STATISTICS) | x | x |
| DDL_SYNONYM_EVENTS | (CREATE SYNONYM, DROP SYNONYM) | x | x |
| DDL_FUNCTION_EVENTS | (CREATE FUNCTION, ALTER FUNCTION, DROP FUNCTION) | x | x |
| DDL_PROCEDURE_EVENTS | (CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE) | x | x |
| DDL_TRIGGER_EVENTS | (CREATE TRIGGER, ALTER TRIGGER, DROP TRIGGER) | x | x |
| DDL_EVENT_NOTIFICATION_EVENTS | (CREATE EVENT NOTIFICATION, DROP EVENT NOTIFICATION) | x | x |
| DDL_ASSEMBLY_EVENTS | (CREATE ASSEMBLY, ALTER ASSEMBLY, DROP ASSEMBLY) | x | x |
| DDL_TYPE_EVENTS | (CREATE TYPE, DROP TYPE) | x | x |
| DDL_DATABASE_SECURITY_EVENTS | | x | x |
| DDL_CERTIFICATE_EVENTS | (CREATE CERTIFICATE, ALTER CERTIFICATE, DROP CERTIFICATE) | x | x |
| DDL_USER_EVENTS | (CREATE USER, ALTER USER, DROP USER) | x | x |
| DDL_ROLE_EVENTS | (CREATE ROLE, ALTER ROLE, DROP ROLE) | x | x |
| DDL_APPLICATION_ROLE_EVENTS | (CREATE_APPROLE, ALTER_APPROLE, DROP_APPROLE) | x | x |
| DDL_SCHEMA_EVENTS | (CREATE SCHEMA, ALTER SCHEMA, DROP SCHEMA) | x | x |
| DDL_GDR_DATABASE_EVENTS | (GRANT DATABASE, DENY DATABASE, REVOKE DATABASE) | x | x |
| DDL_AUTHORIZATION_DATABASE_EVENTS | (ALTER AUTHORIZATION DATABASE) | x | x |
| DDL_SSB_EVENTS | | x | x |
| DDL_MESSAGE_TYPE_EVENTS | (CREATE MSGTYPE, ALTER MSGTYPE, DROP MSGTYPE) | x | x |
| DDL_CONTRACT_EVENTS | (CREATE CONTRACT, DROP CONTRACT) | x | x |
| DDL_QUEUE_EVENTS | (CREATE QUEUE, ALTER QUEUE, DROP QUEUE) | x | x |
| DDL_SERVICE_EVENTS | (CREATE SERVICE, ALTER SERVICE, DROP SERVICE) | x | x |
| DDL_ROUTE_EVENTS | (CREATE ROUTE, ALTER ROUTE, DROP ROUTE) | x | x |
| DDL_REMOTE_SERVICE_BINDING_EVENTS | (CREATE REMOTE SERVICE BINDING, ALTER REMOTE SERVICE BINDING, DROP REMOTE SERVICE BINDING) | x | x |
| DDL_XML_SCHEMA_COLLECTION_EVENTS | (CREATE XML SCHEMA COLLECTION, ALTER XML SCHEMA COLLECTION, DROP XML SCHEMA COLLECTION) | x | x |
| DDL_PARTITION_EVENTS | | x | x |
| DDL_PARTITION_FUNCTION_EVENTS | (CREATE PARTITION FUNCTION, ALTER PARTITION FUNCTION, DROP PARTITION FUNCTION) | x | x |
| DDL_PARTITION_SCHEME_EVENTS | (CREATE PARTITION SCHEME, ALTER PARTITION SCHEME, DROP PARTITION SCHEME) | x | x |

Beispiel:

Dieser Trigger soll verhindern, dass Tabellen in der aktuellen Datenbank gelöscht werden können. Diese sollen auch nicht von berechtigten Usern so einfach gelöscht werden können.

```
use standard;
go

create trigger tab_del_stand
on database
for drop_table
as
raiserror('Erst Trigger "tab_del_stand" löschen, bevor Sie die Tabelle löschen!',10,1);
rollback;
```

16.3 LOGON- Trigger erstellen

Diese Trigger werden durch ein LOGON- Ereignis ausgelöst, welches wiederum ausgelöst wird, wenn eine Benutzersitzung eingerichtet wird.

Syntax:

```
CREATE TRIGGER trigger_name
ON ALL SERVER
[WITH [ENCRYPTION] [,] [EXECUTE AS Klausel]]
{FOR | AFTER} LOGON
AS {sql_anweisung [;] [,...n] | EXTERNAL NAME <method spez.> [;]}

<method_spez.>::=
assembly_name.class_name.method_name
```

Logon-Trigger werden ausgelöst, nachdem die Authentifizierungsphase der Anmeldung abgeschlossen ist und bevor die Benutzersitzung erstellt wird. Aus diesem Grund werden alle Meldungen, die aus dem Trigger stammen und normalerweise den Benutzer erreichen (z.B. Fehlermeldungen und Meldungen aus der PRINT-Anweisung) zum SQL Server-Fehlerprotokoll umgeleitet. Sie werden nicht ausgelöst, wenn die Authentifizierung nicht ausgeführt werden kann.

Sie können Logon-Trigger zum Überwachen und Steuern von Serversitzungen verwenden, beispielsweise durch Nachverfolgung der Anmeldeaktivität, Einschränkung von Anmeldungen auf SQL Server oder durch Einschränkung der Anzahl der Sitzungen für einen bestimmten Anmeldenamen.

Beispiel:

Im folgenden Beispiel wird durch den Logon- Trigger sichergestellt, dass ein User mit dem Anmeldenamen "Paul" nur drei Verbindungen zum Server öffnen darf.

```
use master
go
grant view server state to paul;

create trigger verbindung_limit_trigger
on all server with execute as 'paul'
for logon
as
begin
if original_login() = 'paul' and
(select count(*) from sys.dm_exec_sessions
where is_user_process = 1 and
original_login_name = 'paul') > 3;
rollback;
end;
```

16.4 Trigger ändern

Das Ändern von Triggern erfolgt bei allen Triggertypen mit der ALTER TRIGGER- Anweisung.

Zum Ändern eines DML- Triggers ist eine ALTER- Berechtigung für die Tabelle oder Sicht erforderlich, für die der Trigger definiert ist.

Zum Ändern eines DDL- Triggers, der mit einem Serverbereich (ON ALL SERVER) definiert ist, oder eines LOGON- Triggers ist die CONTROL SERVER- Berechtigung auf dem Server erforderlich. Zum Ändern eines DDL-Triggers, der mit einem Datenbankbereich (ON DATABASE ANY) definiert ist, ist die ALTER ANY DATABASE DDL TRIGGER- Berechtigung in der aktuellen Datenbank erforderlich.

Wenn eine ALTER TRIGGER- Anweisung den ersten oder letzten Trigger ändert, wird das erste oder letzte für den geänderten Trigger festgelegte Attribut gelöscht, und der Reihenfolgewert muss mit **sp_settriggerorder** neu festgelegt werden.

Syntax DML- Trigger:

```
ALTER TRIGGER [schema.]trigger_name
ON { table | view }
[ WITH [ENCRYPTION] [,] [EXECUTE AS Klausel]]
{FOR | AFTER | INSTEAD OF} {[ DELETE ] [,] [ INSERT ] [,] [ UPDATE ]}
[NOT FOR REPLICATION]
AS {sql_anweisung [;] [...n] | EXTERNAL NAME <method spez.> [;]}

<method_spez.>::=
assembly_name.class_name.method_name
```

Syntax DDL- Trigger:

```
ALTER TRIGGER trigger_name
ON { DATABASE | ALL SERVER }
[WITH[ENCRYPTION] [,] [EXECUTE AS Klausel]]
{FOR | AFTER} { event_type [,...n] | event_group}
AS { sql_anweisung [;] | EXTERNAL NAME <method spez.> [;]}

<method_spez.>::=
assembly_name.class_name.method_name
```

Syntax LOGON- Trigger

```
ALTER TRIGGER trigger_name
ON ALL SERVER
[WITH [ENCRYPTION] [,] [EXECUTE AS Klausel]]
{FOR | AFTER} LOGON
AS {sql_anweisung [;] [...n] | EXTERNAL NAME <method spez.> [;]}

<method_spez.>::=
assembly_name.class_name.method_name
```

16.5 Trigger löschen

Sie können einen DML- Trigger entfernen, indem Sie ihn löschen oder die Triggertabelle löschen. Beim Löschen der Tabelle werden auch alle zugeordneten Trigger gelöscht.

Wird ein Trigger gelöscht, werden die Informationen zum Trigger aus den Katalogsichten **sys.objects**, **sys.triggers** und **sys.sql_modules** entfernt.

Mehrere DDL- Trigger können nur über die DROP TRIGGER- Anweisung gelöscht werden, wenn alle Trigger mithilfe identischer ON- Klauseln erstellt wurden.

Sie können mit DROP TRIGGER- und CREATE TRIGGER- Anweisungen einen Trigger umbenennen.

Syntax DML- Trigger:

```
DROP TRIGGER [schema_name.]trigger_name [,....n ]  
[;]
```

Syntax DDL- Trigger:

```
DROP TRIGGER trigger_name [,....n ]  
ON {DATABASE | ALL SERVER}  
[;]
```

Syntax LOGON- Trigger:

```
DROP TRIGGER trigger_name [,....n ]  
ON ALL SERVER  
[;]
```