

Datenmanagement - Zusammenfassung

13. April 2015 16:13

master:	Masterdatenbank hält als relationale Datenbank sämtliche Informationen zum Server, zu den Datenbanken, zu den Anwendenden und deren Verbindungen. Wird hauptsächlich durch das DBMS bewirtschaftet und kann nur indirekt über den Systemkatalog eingesehen werden.
model:	Diese Datenbank dient als Vorlage für alle neu entstehenden Datenbanken. Sie kann durch den Systemadministrator angepasst werden: z.B. mit eigenen Datentypen und gespeicherten Prozeduren oder einer bestimmten Kollation. Hier liesse sich also z.B. eine Vorlage (Template) für firmeninterne Datenbanken realisieren.
msdb:	Diese Datenbank dient dem Systemadministrator und hält Informationen zu allen seinen Aufgaben, die sich automatisieren lassen: Datensicherung, Replikation, Warnmails usw. Sie arbeitet also eng mit SQL Server Agent zusammen.
tempdb:	Diese Datenbank hält temporär Daten, so z.B. während einer Sortierung. Sie kann deshalb bei guter Konfiguration die Systemleistung mit beeinflussen. Bei entsprechender Berechtigung lassen sich auch Tabellen und andere Objekte in tempdb anlegen. Die tempdb persistiert indessen nichts: Es gibt in tempdb nichts, das z.B. einen Neustart des Servers überlebt. Deshalb ist auch die Transaktionsverwaltung eingeschränkt.

Relationale Datenbanken und SQL

7. April 2015 08:43

Grundlegende Eigenschaften von relationalen Datenbanken

Regel 1:	Informationsregel Jede Information einer relationalen Datenbank wird in genau einer Weise durch Werte in Relationen dargestellt und damit redundanzfrei und einheitlich verwaltet. Dazu gehören die Anwendungsdaten und die Metadaten, also Daten über den Aufbau der Datenbank.
Regel 2:	Garantierter Zugriff Jedes einzelne Feld einer Datenbank ist durch eine Kombination von Relationsname, Primärschlüssel und Spaltenname erreichbar.
Regel 3:	Systematische Behandlung fehlender Information In einer relationalen Datenbank müssen Spalten mit fehlender Information (Nullwerte) einheitlich darstellbar sein. Diese Nullwerte werden systematisch als fehlende Informationen von den Standardwerten (z. B. Strings mit Leerzeichen) unterschieden. Spalten können auch so eingerichtet werden, dass Nullwerte nicht erlaubt sind.
Regel 4:	Dynamischer Online-Katalog (Data Dictionary) Ein Datenbankschema wird in derselben Weise wie die gespeicherten Daten selbst beschrieben – nämlich in Tabellen, dem sogenannten Data Dictionary. Autorisierte Benutzer können das Data Dictionary in gleicher Weise abfragen wie die eigentliche Datenbasis.
Regel 5:	Allumfassende Sprache Ein (relationales) Datenbanksystem muss eine Sprache unterstützen, die allumfassend im folgenden Sinne ist. Diese Sprache erfüllt die folgenden Aufgaben: <ul style="list-style-type: none">- Definition der Benutzerdaten- Definition von Sichten als virtuelle Tabellen- Manipulation von Benutzerdaten- Überprüfung von Integritätsregeln- Vergabe von Benutzerrechten und Autorisierung- Transaktionskontrolle und Transaktionshandling (eine Transaktion ist eine Folge von Datenänderungen, die immer ganz oder gar nicht durchgeführt werden muss)
Regel 6:	Benutzersichten und Datenänderungen Für unterschiedliche Benutzergruppen und Anwendungen sind unterschiedliche Sichten (Views) auf die Datenbank notwendig. In einfachen Views, z. B. Teilansichten einer Tabelle, soll es auch Datenänderungen möglich sein.
Regel 7:	HIGH-LEVEL INSERT, UPDATE, und DELETE In einer Datenbank muss das Einfügen, Ändern und Löschen von Daten möglich sein. Dabei soll das System sich den optimalen Zugriffspfad zur schnellen Durchführung der Transaktion selbst suchen.
Regel 8:	Physische Datenunabhängigkeit Anwendungsprogramme und Anwenderoberflächen bleiben unverändert, wenn Veränderungen an der Speicherstruktur oder der Zugriffsmethode in der Datenbank vorgenommen werden.
Regel 9:	Logische Datenunabhängigkeit Anwendungsprogramme und Anwenderoberflächen bleiben unverändert, wenn sich Basisrelationen verändern, die nicht direkt die Anwendungsprogramme betreffen. Es können z.B. Spalten in Relationen ergänzt oder neue Relationen hinzugefügt werden.
Regel 10:	Integritätsunabhängigkeit Integritätsbedingungen, die von der Datenbank erfüllt werden müssen, werden mithilfe der relationalen Datenbanksprache definiert, im Data Dictionary abgelegt und vom DBMS ausgeführt. Diese Integritätsbedingungen gehören nicht ins Anwenderprogramm.
Regel 11:	Verteilungsunabhängigkeit Eine relationale Datenbank besitzt die Verteilungsunabhängigkeit. Das heisst, die Anwendungsprogramme bleiben unverändert, wenn die verteilte Datenhaltung auf mehreren Rechnern eingeführt oder wieder zurückgenommen wird und mehrere Datenbanken zu einer Datenbank zusammengelegt werden.

Relationale Datenbanken und SQL

7. April 2015 09:36

Regel 12: Unterwanderungsverbot

Falls das DBMS eine andere 3GL-Sprache wie C oder Java zulässt, darf diese Sprache nicht die aufgestellten Regeln 1 bis 11 verletzen oder ausser Kraft setzen.

Datenbewirtschaftung in relationalen Datenbanken

Den Zugriff auf relationale Datenbanken bewerkstelligen wir mit SQL.

SQL steht für Structured Query Language und beinhaltet folgende Komponenten:

DCL (Data Control Language):	Definition von Zugriffsberechtigungen und Speicherstrukturen (z. B. CREATE LOGIN...)
DDL (Data Definition Language):	Definition von Datenbankobjekten (z. B. ALTER TABLE...)
DML (Data Manipulation Language):	Anlegen, Ändern oder Löschen von Daten (z. B. DELETE FROM...)
DQL (Data Query Language):	Abfragen von Daten (z. B. SELECT * FROM...)

Ferner unterstützt SQL durch das Transaktionskonzept die konsistente Haltung von Daten .

Transaktionen verfügen über die ACID-Eigenschaften:

A – Atomicity:	Transaktionen sind atomar, also als unteilbare Einheiten zu betrachten, Transaktionen werden ganz oder gar nicht ausgeführt.
C – Consistency:	Eine Transaktion überführt eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand. Ein Datenzustand heisst konsistent, wenn alle Daten semantisch richtig, also im Anwendungskontext korrekt sind.
I – Isolation:	Transaktionen laufen isoliert ab. Obwohl im Mehrbenutzerbetrieb gleichzeitig mehrere Transaktionen abgearbeitet werden, läuft jede einzelne Transaktion wie in einem simulierten Einbenutzerbetrieb ab.
D – Durability:	Die Ergebnisse einer Transaktion werden dauerhaft (persistent) in der Datenbank gespeichert.

Relationale Datenbanken und SQL

13. April 2015 16:21

TRANSACT-SQL Syntaxkonventionen

Wir arbeiten mit dem SQLDialekten TRANSACT-SQL. Die Konvention orientiert sich an der Metasprache EBNF (Extended Backus - Naur Form) und enthält folgende Elemente:

GROSSBUCHSTABEN	Transact-SQL-Schlüsselwörter.
<i>Kursiv</i>	Vom Benutzer anzugebende Parameter der Transact-SQL-Syntax.
(senkrechter Strich)	Trennt in eckigen oder geschweiften Klammern eingeschlossene Syntaxelemente. Sie können nur eines der Elemente verwenden.
[] (eckige Klammern)	Optionale Syntaxelemente. Geben Sie die eckigen Klammern nicht ein.
{ } (geschweifte Klammern)	Erforderliche Syntaxelemente. Geben Sie die geschweiften Klammern nicht ein.
[...n]	Zeigt an, dass das vorherige Element n-mal wiederholt werden kann. Die einzelnen Vorkommen werden durch Trennzeichen getrennt.
[...n]	Zeigt an, dass das vorherige Element n-mal wiederholt werden kann. Die einzelnen Vorkommen werden durch Leerzeichen voneinander getrennt.
;	Transact-SQL Anweisungsabschlusszeichen. Dieses Abschlusszeichen ist für die meisten Anweisungen in dieser Version von SQL Server nicht zwingend erforderlich, in zukünftigen Versionen kann sich dies jedoch ändern.
<Bezeichnung> ::=	Der Name eines Syntaxblocks. Diese Konvention dient zur Gruppierung und Bezeichnung von Abschnitten einer langen Syntax oder einer Syntaxeinheit, die an mehreren Stellen innerhalb einer Anweisung verwendet werden kann. Jede Stelle, an der der Syntaxblock verwendet werden kann, wird durch die in spitze Klammern eingeschlossene Bezeichnung angezeigt: <label>.

Um uns mit dieser Syntax vertraut zu machen, zeigen wir hier die Anwendung an einer Beispiel-Aufgabe auf:

Definieren Sie eine Tabelle mit «quick_n_dirty» als Tabellename und «id», «bzch» als Spaltennamen. «id» soll als Ganzzahl, «bzch» als Textfeld definiert werden. «id» soll als Primärschlüssel der Tabelle definiert werden. Die Schlüsselvergabe soll vom System als aufsteigende Zahlensequenz (Autowert) vorgenommen werden.

Lösung:

```
CREATE TABLE quick_n_dirty
(
id INTEGER IDENTITY PRIMARY KEY
bzch VARCHAR(50) NOT NULL
)
```

SQL-Anweisungen

Schlüsselwörter werden gross geschrieben.

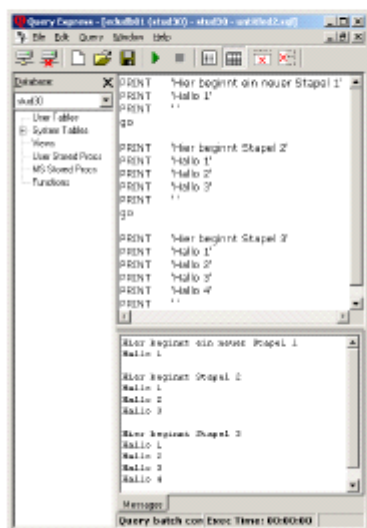
IDENTITY(100)	Erste ID beginnt mit 100
PRIMARY KEY	Niemals Null
SELECT id AS 'Spalte 1' FROM quick_n_dirty;	Spaltenzuweisung

Anweisungssequenz und Anweisungsstapel

- Eine Aneinanderreihung von SQL Anweisungen bildet eine Anweisungssequenz.
- Sie wird dem Server als Anweisungsstapel (Batch) übermittelt.
- Ein Stapel wird immer als Ganzes bearbeitet.
- Er wird nach der Alles-oder-Nichts Regel ausgeführt. Ein Stapel ist per Default durch zwei GO (batch separator) begrenzt. Dazwischen können beliebig viele SQL Anweisungen stehen.

So sieht die Anwendung des Stapeltrennzeichens go bei Query Express aus:

Fall 1



Fall 2

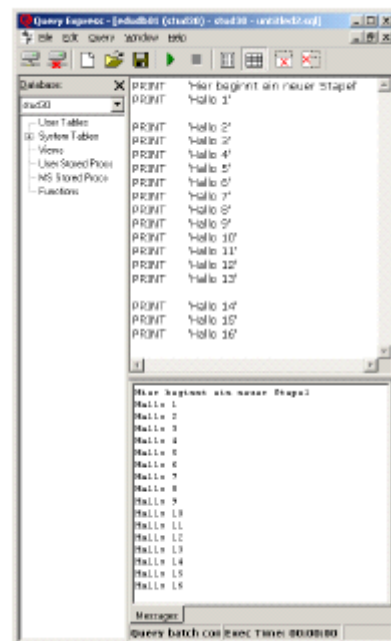


Abbildung 6: Eingabefenster mit mehreren Befehlsstapeln

- Query Express nimmt eine ganze Sequenz von mindestens einem oder mehreren Stapeln auf.
- Die erste Anweisung eröffnet einen neuen Stapel. Davor steht also ein implizites go .
- Fehlt ein abschliessendes go , dann behandelt Query Express alle Anweisungen ab dem letzten impliziten oder expliziten go als einen Stapel.
- Query > Execute oder [F5] oder der grüne Pfeil lösen ein go aus und senden den oder die Stapel an den Server.
- Leerzeilen und Leerzeichen spielen keine Rolle.
- Abhängig von den Anweisungen (z.B. mehrere SELECT) werden mehrere Ausgabefenster (unten) mit je eigenem Register eröffnet. Hier ist das nicht der Fall.

Im Fall 1 werden also drei Anweisungsstapel bearbeitet.

Im Fall 2 wird ein Anweisungsstapel bearbeitet.

Kommentar

13. April 2015 16:44

Die Rolle von GO

In Query Express ist kein GO zur Beendigung einer Anweisung oder eines Anweisungsstapels nötig ist. GO kann denn auch getrost unterlassen werden. Zweier Fakten sollte man sich indessen bewusst sein:

1. Query Express fügt jedem zur Ausführung abgefeuerten Anweisungsfenster ein GO an.
2. Bei gewissen, kommandozeilenbasierten Klienten muss ein GO befohlen werden.
3. Automatische Skripts: gewisse Anweisungen müssen an erster Stelle eines Stapels stehen (z.B. CREATE TRIGGER).

Kommentare

Codierung

- Für einzeilige Kommentare gibt es das "Doppelminus":
`-- Die folgende Zeile selektiert alle Angestellten von northwind`
`SELECT * FROM dbo.employees`
- Für Kommentare am Zeilenende gilt dasselbe:
`SELECT * FROM dbo.employees -- alle Angestellten anzeigen`
- Für mehrzeilige Kommentare klammern Sie mit
`/*`
`Mehrzeilenkommentar ohne go`
`Mehrzeilenkommentar ohne go`
`Mehrzeilenkommentar ohne go`
`*/`

Datenbank erstellen, ändern, löschen

29. Mai 2015 15:46

-- neue Datenbank anlegen

```
USE master go
CREATE DATABASE database_name
[
    ON [ PRIMARY ] <filespec> [ LOG ON <filespec> ]
    [ COLLATE collation_name ]
] [;] go
USE database_name go
```

-- Datenbankdefinition ändern

```
USE master go
ALTER DATABASE database_name
{ <datei_optionen> | MODIFY NAME = new_database_name | COLLATE collation_name }
[;] go
USE database_name go
```

-- Datenbank löschen

```
USE master go
DROP DATABASE database_name [ ,...n ] [;] go
Ausführliche Syntax in SQL-Nutshell, S. 75 / 93ff.
```

Metadaten zum Server:

```
SELECT * FROM sys.servers -- Systemsicht in master
EXECUTE sys.sp_helpserver [server] -- gespeicherte Systemprozedur in master
```

Metadaten zu Datenbanken:

```
SELECT * FROM sys.databases -- Systemsicht in master
SELECT DB_ID(['db_name']) -- Funktion
SELECT DB_NAME([db_id]) -- Funktion
EXECUTE sys.sp_helpdb [db_name] -- gespeicherte Systemprozedur in master
```

CREATE TABLE , DISTINCT

29. Mai 2015 12:31

Metadaten zu Tabellenobjekten (veraltete Sichten und Prozeduren werden nicht angegeben) :

```
SELECT * FROM INFORMATION_SCHEMA.TABLES -- ISO konform
SELECT * FROM INFORMATION_SCHEMA.COLUMNS -- ISO konform
SELECT * FROM sys.objects -- Systemsicht in master: alle Objekte, nicht
aufschlussreich
SELECT * FROM sys.tables -- Systemsicht in master: alle Tabellen
SELECT * FROM sys.columns -- Systemsicht in master: alle Kolonnen, nicht
aufschlussreich
EXECUTE sys.sp_help table_name -- gespeicherte Systemprozedur in master SUPER!!!
EXECUTE sys.sp_columns table_name -- gespeicherte Systemprozedur in master
```

TABELLE ERSTELLEN

```
CREATE TABLE table_name
(column_name datatype { [DEFAULT default_value] | [IDENTITY (seed, increment)]
[NULL | NOT NULL] | {[PRIMARY KEY | UNIQUE]}
[, ...]
)
```

Beispiel:

```
CREATE TABLE abc
(spalte1 INT NOT NULL IDENTITY(4711,1) PRIMARY KEY,
Spalte2 VARCHAR(25) NOT NULL,
Spalte3 VARCHAR(39) NOT NULL DEFAULT 'Luzern',
)
```

```
SELECT DISTINCT Title FROM dbo.Employees
```

--> zeigt alle verschiedenen Titel nur einmal an - ohne Distinct gibt es Mehrfachnennungen

ALTER TABLE

13. April 2015 17:43

```
ALTER TABLE tabellen_name
ADD spalten_name datentyp [ spaltenbezogene_einschraenkung ] [ ...n ]]
| DROP COLUMN spalten_name
| ALTER COLUMN alt_spalten_name neu_datentyp [ NULL | NOT NULL ] [ COLLATE
    kollation ]
| ADD [CONSTRAINT tabellenbezogene_einschraenkung] Einschraenkung [,...n]
| DROP [CONSTRAINT] einschraenkung_name [, ...n]
[:]
```

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype
```

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name
```

To change the data type of a column in a table, use the following syntax:

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

Examples:

```
-- Fremdschlüssel weg
ALTER TABLE tbl_kunde DROP CONSTRAINT FK_fi_moral_nr;
GO

-- Primärschlüssel weg
ALTER TABLE tkey_moral DROP CONSTRAINT PK_id_moral_nr;
GO

-- neuer Datentyp bei PK
ALTER TABLE tkey_moral ALTER COLUMN id_moral_nr TINYINT NOT NULL;
GO

-- neuer Datentyp bei FK
ALTER TABLE tbl_kunde ALTER COLUMN fi_moral_nr TINYINT NOT NULL
GO

-- neuer Primärschlüssel
ALTER TABLE tkey_moral ADD CONSTRAINT PK_id_moral_nr PRIMARY KEY (id_moral_nr)
GO

-- neuer Fremdschlüssel
ALTER TABLE tbl_kunde ADD CONSTRAINT FK_fi_moral_nr FOREIGN KEY (fi_moral_nr)
REFERENCES tkey_moral -- referenziert auf Tabelle des Primärschlüssels
ON UPDATE CASCADE
ON DELETE NO ACTION;
GO
```

Anzahl Zeilen die geändert wurden: PRINT @@rowcount

Sp_help

Sp_helpconstraint

Zuerst Primär und Fremdschlüssel löschen und dann die Spalten ändern!

Daten mutieren

29. Mai 2015 16:31

```
UPDATE tabellen_name
    SET spalten_name = { ausdruck | DEFAULT | NULL } [ ,...n ]
    [ FROM tabellen_verbund [ ,...n ] ]
    [ WHERE bedingung ]
[ ; ]
```

Beispiel

```
SELECT * FROM tbl_kunde WHERE name LIKE 'Walt%'; -- ein Eintrag
UPDATE tbl_kunde SET name = 'Menzer' WHERE name LIKE 'Walt%';
```

```
DELETE [ FROM ] tabellen_name
    [ FROM tabellen_verbund [ ,...n ] ]
    [ WHERE bedingung ] [ ; ]
```

Beispiel

```
SELECT * FROM tkey_moral WHERE id_moral_nr = 4;
DELETE FROM tkey_moral WHERE id_moral_nr = 4;
```

DELETE löscht tupelweise bedingt. Jede Löschung ...

- ... respektiert die referenzielle Integrität
- ... sperrt das Tupel (Locking: siehe später)
- ... wird im Transaktions-Log protokolliert
- ... feuert entsprechende Trigger.

Die Löschung aller Tupel mit DELETE lässt die Definition der Tabelle, ihrer Indizes und Trigger bestehen und ist aus Performanzgründen nicht zu empfehlen.

TRUNCATE (TRUNCATE TABLE tabellen_name [;]) löscht alle Tupel bedingungslos. Die Löschung

- ... respektiert die referenzielle Integrität
- ... sperrt die Tabelle (Locking: siehe später)
- ... wird im Transaktions-Log als eine atomare Aktivität protokolliert
- ... feuert keine Trigger. (Haben Sie das gut gelesen?)

TRUNCATE TABLE ist eine so genannte bulk Operation - eine Massenoperation.

Sehr wichtig ist: Hat die zu löschende Tabelle eine Beziehung in eine Kindtabelle, wird sie nicht gelöscht, dies auch wenn kindseitig keine referenzielle Integrität verletzt würde! Also:

TRUNCATE TABLE tkey_moral geht nicht,

DROP TABLE löscht alle Tupel, die Tabellendefinition sowie alle zugehörigen Indizes und Trigger. Die referenzielle Integrität wird respektiert. Die Löschung wird als eine atomare Operation protokolliert. Es feuern keine tabellenbasierenden Trigger.

Indizes

29. Mai 2015 17:28

Suchmöglichkeiten eines DBMS:

DBMS	Analogie Buchsuche
(Full) Table Scan	Buch durchblättern
Nonclustered Index	Im Index Stichwort nachschlagen: Verweis auf Seitenzahlen
Clustered Index	Lexikon / Wörterbuch: Einträge liegen in sortierter Reihenfolge vor

Der Speicherort eines Datensatzes wird beim SQL-Server mit einer dreiteiligen Adresse, nämlich [\[File:page:Slot\]](#) vorgenommen:

- File: SQL-Server-interne ID der Datendatei, in der sich der Datensatz befindet.
- Page: Seite der Datendatei, auf der sich der Datensatz befindet.
- Slot: Position innerhalb der Seite, die der Datensatz einnimmt.

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
ON <object> ( column [ ASC | DESC ] [ ,...n ] )
[ INCLUDE ( column_name [ ,...n ] ) ]
[ WITH ( <Optionen> [ ,...n ] ) ]
[ ; ]
```

Bsp.:	CREATE INDEX ix_kund_name ON tbl_kunde (name ASC); GO
-------	--

```
ALTER INDEX { index_name | ALL }
ON <object>
{ REBUILD | DISABLE | REORGANIZE | SET ( <Option> [ ,...n ] ) }
[ ; ]
```

```
DROP INDEX index_name ON table_or_view_name [ ,...n ]
```

- WITH Optionen beziehen sich auf die Statistiken, Füllfaktoren usw. und werden nicht näher besprochen, siehe online Hilfe.
- REBUILD defragmentiert den Index intern und extern und balanciert ihn aus. Diese Aktivität ist weitgehend gleich bedeutend mit DROP > CREATE INDEX. Sie ist kostenintensiv und belastet andere Prozesse. REBUILD empfiehlt sich bei grosser Baumfragmentierung.
- REORGANIZE optimiert die Beziehung zwischen den Blattknoten und der zugehörigen phys. Speicherablage, die u.U. umgelagert wird. Andere Prozesse werden kaum behelligt. Die Operation kann lange dauern. REORGANIZIE empfiehlt sich bei kleiner Baumfragmentierung.
- DISABLE deaktiviert einen Index (und pflegt ihn auch nicht weiter). Er wird reaktiviert und neu aufgebaut mit REBUILD.

Metadaten zu Indexobjekten:

```
EXECUTE sp_helpindex tabelle
SELECT * FROM sys.indexes -- Übersicht
SELECT * FROM sys.index_columns -- gibt Auskunft, welche Spalten einbezogen sind
```

CREATE INDEX

17. April 2015 11:34

Mit dem richtigen Index kann das Datenbanksystem zunächst den Index nach dem Speicherort der Daten durchsuchen und die benötigten Daten dann direkt an diesem Ort aufrufen. Dies spart viel Zeit.

```
CREATE INDEX "INDEX_NAME" ON "TABELLEN_NAME" (SPALTEN_NAME);
```

Gehen wir von folgender Tabelle aus:

Tabelle ***Customer***

Spalten Name	Datentyp
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime

Um einen Index für die Spalte Nachname zu erzeugen, würden wir eingeben:

```
CREATE INDEX IDX_CUSTOMER_LAST_NAME  
ON Customer (Last_Name);
```

Um einen Index für die beiden Spalten Wohnort und Land zu erstellen, würden wir eingeben:

```
CREATE INDEX IDX_CUSTOMER_LOCATION  
ON Customer (City, Country);
```

Wertbeschränkungen und Domänen

31. Mai 2015 11:53

Skript S. 42ff.

```
-- Neue Domäne einstellige Ganzzahl
CREATE TYPE tp_moralisches FROM NUMERIC(1,0);
GO

-- Löschen des Fremdschlüssels
ALTER TABLE tbl_kunde DROP CONSTRAINT FK-fi_moral_nr ;
GO

-- Löschen des Primärschlüssels
ALTER TABLE tkey_moral DROP CONSTRAINT PK-id_moral_nr;
GO

-- Neuer Datentyp zuweisen und Check hinzufügen
ALTER TABLE tkey_moral ALTER COLUMN id_moral_nr tp_moralisches NOT NULL;
ALTER TABLE tkey_moral ADD CHECK (id_moral_nr >= 0);
GO

-- Neuer Datentyp zuweisen und Check hinzufügen
ALTER TABLE tbl_kunde ALTER COLUMN fi_moral_nr tp_moralisches;
ALTER TABLE tbl_kunde ADD CHECK (fi_moral_nr >= 0);
GO

-- Primärschlüssel hinzufügen
ALTER TABLE tkey_moral ADD CONSTRAINT PK-id_moral_nr PRIMARY KEY (id_moral_nr);
GO

-- Fremdschlüssel hinzufügen
ALTER TABLE tbl_kunde ADD CONSTRAINT FK-fi_moral_nr FOREIGN KEY (fi_moral_nr)
REFERENCES tkey_moral
ON UPDATE CASCADE
ON DELETE SET NULL;
GO

-- Vorgabewert setzenc
ALTER TABLE tbl_kunde ADD CONSTRAINT DF_kafiluz DEFAULT 'Luzern' FOR wohnort;
GO

-- Nur Werte grösser oder gleich 100 sind gültig
ALTER TABLE tass_police ADD CONSTRAINT CK_centopercento CHECK (praem_stufe >=
100);
GO

-- Nur Werte grösser als 0 sind gültig
ALTER TABLE tass_police ADD CHECK (bezahlt >= 0);
GO
```

Benutzer und Gruppen

31. Mai 2015 14:52

Login

Anlegen einer Serverbenutzerin (LOGIN)

- jede Person, welche in irgendeiner Datenbank operieren will, muss Serverbenutzer sein
- nur die Systemadministratorin (und andere privilegierte Serverrollen) kann einen Serverbenutzer anlegen.

```
CREATE LOGIN login_name
WITH PASSWORD = 'password' [ HASHED ] [ MUST_CHANGE ]
[ ,
DEFAULT_DATABASE = database
| DEFAULT_LANGUAGE = language
| CHECK_EXPIRATION = { ON | OFF }
| CHECK_POLICY = { ON | OFF }
[ ,... ]
]
```

```
Bsp.:      -- Passwort ändern
           ALTER LOGIN studxx WITH
           PASSWORD = 'pass_weiss_nicht'
           OLD_PASSWORD = 'pass_wdxx';
           GO
```

User

Anlegen eines Datenbankbenutzers (USER)

- jede Person, welche in einer speziellen Datenbank operieren will, muss in dieser als Datenbankbenutzerin angelegt sein
- es können nur Serverbenutzerinnen Datenbankbenutzer werden (es gibt Ausnahmen)
- die anonyme Anmeldung an einer Datenbank ist durch jeden Serverbenutzer als guest möglich; guest muss indessen explizit zugelassen sein
- nur die Datenbankeigentümerin (und andere privilegierte Datenbankrollen) kann einen Datenbankbenutzer anlegen.

```
CREATE USER user_name
[ FOR LOGIN login_name | WITHOUT LOGIN ]
[ WITH DEFAULT_SCHEMA = schema_name ]
```

```
-- user einer gruppe zuordnen: EXEC sp_addrolemember gl, pia
```

Rolle

(Optionales) Anlegen einer Rolle (ROLE)

- ein Rollenname bedarf keiner Abbildung auf einen Serverbenutzer
- eine Rolle gilt auf Datenbankebene
- nur die Datenbankeigentümerin (und andere privilegierte Datenbankrollen) kann eine Rolle anlegen
- eine Rolle gehört dem Benutzer, der sie angelegt hat oder kann mit AUTHORIZATION übertragen werden.

```
CREATE ROLE role_name [ AUTHORIZATION owner_name ];
```

Gruppe

(Optionale) Aufnahme eines Datenbankbenutzers in eine Gruppe (MEMBERSHIP)

- gängige Anweisungen dazu sind GRANT MEMBERSHIP IN rolle TO benutzer
- SQL Server unterstützt dies nicht und benutzt stattdessen eine Speicherprozedur

```
EXEC sp_addrolemember 'role', 'benutzer_oder_rolle'; (SQL Server 2008)
-- ALTER ROLE Sales ADD MEMBER Barry; (Bsp. Ab SQL Server 2012)
```

```
-- Rechte auf allen Tabellenobjekten
```

```
GRANT ALTER ANY SCHEMA, CREATE TABLE, INSERT, UPDATE, REFERENCES, DELETE, SELECT TO gruppe
```

Objektberechtigungen

31. Mai 2015 15:06

GRANT (gewähren)

```
GRANT {  
    ALL [ PRIVILEGES ]  
    | permission [ ( column [ ,...n ] ) ] [ ,...n ]  
    [ ON objekt ] TO principal [ ,...n ]  
    [ WITH GRANT OPTION ] -- Recht zum Vergeben von Rechten  
}
```

PERMISSIONS: select, update, insert, delete, references

ALTER ANY SCHEMA

CREATE TABLE

Bsp.:	GRANT DELETE, INSERT, SELECT, UPDATE ON tkey_moral TO romulus WITH GRANT OPTION;
-------	--

REVOKE (verweigern)

```
REVOKE [ GRANT OPTION FOR ]  
{  
    [ ALL [ PRIVILEGES ] ]  
    | permission [ ( column [ ,...n ] ) ] [ ,...n ]  
}  
[ ON objekt ]  
FROM principal [ ,...n ] [ CASCADE]
```

Deny

DENY CONNECT TO remus; -- verweigert Zugriff auf DB studxx bei use studxx;

Views

31. Mai 2015 16:32

Zur Definition einer Sicht müssen Sie die Rechte für ALTER ANY SCHEMA und CREATE VIEW haben.

```
CREATE VIEW view_name [ (column [ ,...n ] ) ]
AS select_statement [ ; ]
go
DROP VIEW sicht_name
Go
```

Beispiele:

```
CREATE VIEW v_umsatz AS
SELECT sum(bezahlt) AS 'Umsatz aus Policen' FROM tass_police
go
SELECT * FROM v_umsatz
Go
```

```
-----

CREATE VIEW v_kunde AS
SELECT id_kunde AS 'Kundennummer', name AS 'Name', vorname AS 'Vorname', wohnort AS
'Ort',
vers_bez AS 'Police', vers_gebiet AS 'Gebiet', bezahlt AS 'bezahlt' FROM tkey_moral
JOIN tbl_kunde ON id_moral_nr = fi_moral_nr
JOIN tass_police ON id_kunde = id_fi_kunde
JOIN tkey_versicherung ON id_vers_art = id_fi_vers_art

go
```

```
SELECT * FROM v_kunde
```

```
SELECT Kundennummer, Name, Police, bezahlt FROM v_kunde
WHERE bezahlt BETWEEN 1000 AND 10000
ORDER BY bezahlt DESC
```

```
SELECT avg(bezahlt) AS 'im Mittel bezahlte Risiko-Praämien'
FROM v_kunde
WHERE Gebiet = 'Risiko'
go
```

```
-----

CREATE VIEW v_debiparms AS(SELECT id_kunde AS 'Kundennummer', name AS 'Name', moral_bez AS
'Zahlungsmoral', bezahlt AS 'Betrag'
FROM tkey_moral
JOIN tbl_kunde ON id_moral_nr = fi_moral_nr

JOIN tass_police ON id_kunde = id_fi_kunde
WHERE bezahlt = (SELECT max(bezahlt) FROM tass_police)
UNION SELECT id_kunde AS 'Kundennummer', name AS 'Name', moral_bez AS
'Zahlungsmoral', bezahlt AS 'Betrag'FROM tkey_moral

JOIN tbl_kunde ON id_moral_nr = fi_moral_nr

JOIN tass_police ON id_kunde = id_fi_kunde
WHERE bezahlt = (SELECT min(bezahlt) FROM tass_police)
UNION SELECT 0 AS 'Kundennummer', '---' AS 'Name', '---' AS
'Zahlungsmoral', avg(bezahlt) AS 'Betrag'FROM tass_police )

go
```

```
SELECT * FROM v_debiparms go
```


SELECT

13. April 2015 17:02

SQL-Anweisungen

SELECT * FROM Student;	listet alle Spalten und alle Zeilen der Tabelle <i>Student</i> auf
SELECT VorlNr, Titel FROM Vorlesung;	listet die Spalten <i>VorlNr</i> und <i>Titel</i> aller Zeilen der Tabelle <i>Vorlesung</i> auf.
SELECT DISTINCT MatrNr FROM hört;	listet nur unterschiedliche Einträge der Spalte <i>MatrNr</i> aus der Tabelle <i>hört</i> auf. Dies zeigt die Matrikelnummern aller Studenten, die mindestens eine Vorlesung hören, wobei mehrfach auftretende Matrikelnummern nur einmal ausgegeben werden.
SELECT MatrNr AS Matrikelnummer, Name FROM Student;	listet die Spalten <i>MatrNr</i> und <i>Name</i> aller Zeilen der Tabelle <i>Student</i> auf. <i>MatrNr</i> wird beim Anzeigeergebnis als Matrikelnummer aufgeführt.
SELECT VorlNr, Titel FROM Vorlesung WHERE Titel = 'ET';	listet <i>VorlNr</i> und <i>Titel</i> aller derjenigen Zeilen der Tabelle <i>Vorlesung</i> auf, deren Titel 'ET' ist.
SELECT Name FROM Student WHERE Name LIKE 'F%';	listet die Namen aller Studenten auf, deren Name <u>mit F beginnt</u> . (im Beispiel: Fichte und Fauler). %H% OR %H OR H%
SELECT Vorname, Name, StrasseNr, Plz, Ort FROM Student WHERE Plz = '20095' ORDER BY Name;	listet Vorname, Name, StrasseNr, Plz und Ort aller Studenten aus dem angegebenen Postleitzahlbereich, sortiert nach Nachnamen, auf. DESC = absteigend , ASC = aufsteigend
SELECT fi_interpret, COUNT(dt_stueck_titel) AS 'Anzahl' FROM tbl_stueck GROUP BY fi_interpret HAVING count(dt_stueck_titel) > 9 ORDER BY Anzahl DESC	Anzahl Stücke pro Interpret aber nur solche mit mehr als 10 Stücken

Reihenfolge:

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

COUNT, SUM

1. Juni 2015 21:11

COUNT

SELECT COUNT(*) FROM tbl_stueck --> oder COUNT(DISTINCT dt_stueck_titel)
WHERE dt_stueck_titel LIKE 'let%'

Tabelle Buecher

Titel	Rubrik	Seitenanzahl	Autor
Friedhof der Kuscheltiere	Horror	459	Stephen King
Ich bin dann mal weg	Reise	320	Hape Kerkeling
Der Schwarm	Thriller	956	Frank Schätzing
Wahn	Horror	800	Stephen King
The Ring	Horror	301	Koji Suzuki

COUNT(name)	Zählt wie viele Einträge es in einer Spalte gibt die nicht NULL sind							
SELECT COUNT(Rubrik) AS AnzahlHorrorBuecher FROM Buecher WHERE Rubrik='Horror'	<table><tr><th>AnzahlHorrorBuecher</th></tr><tr><td>3</td></tr></table>		AnzahlHorrorBuecher	3				
AnzahlHorrorBuecher								
3								
SELECT Autor, COUNT(Rubrik) AS AnzahlHorrorBuecher FROM buecher WHERE Rubrik='Horror' GROUP BY Autor	<table><tr><th>Autor</th><th>AnzahlHorrorBuecher</th></tr><tr><td>Koji Suzuki</td><td>1</td></tr><tr><td>Stephen King</td><td>2</td></tr></table>		Autor	AnzahlHorrorBuecher	Koji Suzuki	1	Stephen King	2
Autor	AnzahlHorrorBuecher							
Koji Suzuki	1							
Stephen King	2							

SELECT SUM(column_name) FROM table_name;

JOINS

1. Juni 2015 21:49

Theta Join = Inner Join

```
SELECT dt_stueck_titel AS 'Songtitel',  
dt_name AS 'Interpret'  
FROM tbl_stueck, tkey_interpret  
WHERE dt_name LIKE '%clapton'  
AND id_interpret = fi_interpret;
```

INNER JOIN

```
SELECT abc, def  
FROM tbl_abc INNER JOIN tbl_def  
ON id_abc = fi_def
```

- Weil wir (meist, so wie hier) die Tupel über ihre Primär-Fremdschlüssel-Beziehung verbinden ("joinen"), haben wir es mit einem **Key Join** zu tun.
- Weil wir (meist, so wie hier) durch Gleichheit von Schlüsselwerten verbinden, heisst der Verbundtyp ferner **Equi Join**.
- Weil wir (meist, so wie hier) nur Tupel wollen, in welchen die Schlüsselattribute auch tatsächlich reale Werte tragen (also keine NULL-Marken bei den verbindenden Attributen), haben wir es mit einem **Inner Join** zu tun.

Bei gleichnamigen Attributen muss man diese qualifizieren durch: tabellenname.attribbutname

Oder als Aliasen z.B:

```
SELECT k.name AS 'Name', k.zusatz AS 'Zusatzname', b.datum AS 'Datum',  
b.zusatz AS Bezeichnung  
FROM tbl_kunde AS k, tass_bestellung AS b  
WHERE k.ku_Nr = b.ku_Nr;
```

Syntax:

```
... FROM tabellen_name1 [AS alias]  
[CROSS JOIN tabellen_name | [join_typ] JOIN tabellen_name ON Verknüpfungsbedingung  
[{AND | OR} Verknüpfungsbedingung] [...]]  
[...]
```

Je nach Join-Typ wird die Abfrage in verschiedenen logischen Phasen durchgeführt:

1. Kartesisches Produkt erzeugen (CROSS JOIN)
2. Filtern (INNER JOIN)
3. «Äussere» Datenzeilen hinzufügen (OUTER JOIN)

OUTER JOIN

Es werden aber auch Verbund-Tupel in die Ergebnismenge aufgenommen, in welchen der Verknüpfungswert

- in der als rechter Operand genannten Tabelle NULL sein kann (**LEFT OUTER** - "links alles, rechts auch NULLS").
- in der als linker Operand genannten Tabelle NULL sein kann (**RIGHT OUTER** - "rechts alles, links auch NULLS")

```
SELECT dt_name, dt_stueck_titel  
FROM tkey_interpret LEFT OUTER JOIN tbl_stueck  
ON id_interpret = fi_interpret
```

CROSS JOIN

Jedes mit jedem --> Anzahl linke Tabelle x Anzahl rechte Tabelle

```
SELECT dt_name, dt_stueck_titel  
FROM tkey_interpret, tbl_stueck
```

Unterabfragen

2. Juni 2015 10:29

```
SELECT dt_stueck_titel AS 'Titel', dt_zeit AS 'Zeit'
FROM tbl_stueck
WHERE dt_zeit > (SELECT AVG(dt_zeit) FROM tbl_stueck)
ORDER BY dt_zeit;
```

1. Die geklammerte Unterabfrage gibt einen Wert zurück.
2. Die Unterabfrage kommt dorthin, wo ich den Wert eingäbe, wenn er mir bekannt wäre.

Unterabfrage IN

Das Prädikat (NOT) IN untersucht (im mengenorientierten SQL!) , ob ein Attributswert (nicht) Element einer Menge sei und gibt dann true bzw. false zurück.

Gib mir alle Musikstücke, die Originale, also nicht Coverversionen sind:

```
SELECT * FROM tbl_stueck
WHERE id_stueck_nr NOT IN
(SELECT id_stueck_nr FROM tass_stueck_original);
```

Unterabfrage EXISTS

Das Prädikat (NOT) EXISTS untersucht ob ein mindestens ein Tupel mit der entsprechenden Bedingung vorhanden ist oder nicht und gibt dann true bzw. false zurück.

Melde mir, wenn noch mindestens ein Musikstück ohne Zeitangabe existiert:

```
SELECT 'Es hat noch Stücke ohne Zeitangabe'
WHERE EXISTS
(SELECT * FROM tbl_stueck
WHERE dt_zeit IS NULL);
```

Korrelierte Unterabfragen

Operand nimmt direkt Bezug auf eine Tabelle, welche in der äusseren Abfrage deklariert wurde.

Welche Mundart-Stücke sind erfasst?

```
SELECT DISTINCT dt_stueck_titel
FROM tbl_stueck
WHERE 'mu' IN
(SELECT id-fi_stil
FROM tass_stueck_stil
WHERE id_stueck_nr = id-fi_stueck_nr);
```

Das innere WHERE der korrelierten Unterabfrage arbeitet mit einem Attribut, dessen Tabelle im inneren Tabellenverbund gar nicht genannt ist!

```
SELECT dt_stueck_titel AS 'Titel', dt_zeit AS 'Dauer'
FROM tbl_stueck
WHERE dt_zeit BETWEEN
    (0.9 * (SELECT AVG(dt_zeit) FROM tbl_stueck WHERE dt_zeit IS NOT NULL))
    AND (1.1 * (SELECT AVG(dt_zeit) FROM tbl_stueck WHERE dt_zeit IS NOT NULL))
ORDER BY dt_zeit ASC
```

LEN(string) --> Funktion für Ganzzahl eines Wortes

Mengenabfragen

2. Juni 2015 12:43

INTERSECT alles was in abfrage 1 gleich ist wie in abfrage2

Schnittmenge: Die Resultat-Tupel zweier einzelner Abfragen haben in allen gemeinsamen Attributen dieselben Werte und werden als Ergebnis ausgegeben. Lösung:

- entweder Syntax: **abfrage1 INTERSECT abfrage2**
- oder entsprechend aufwändig formulierte abfrage3 mit einer WHERE-Klausel unter Verwendung von EXISTS oder IN.

EXCEPT alles in abfrage1 was gleich ist wie in abfrage2

Differenzmenge: Die Resultat-Tupel kommen in Abfrage 1, nicht aber ebenfalls in Abfrage 2 vor. Lösung:

- entweder Syntax: **abfrage1 EXCEPT abfrage2** (in gewissen Dialekten MINUS oder DIFFERENCE statt EXCEPT)
- oder entsprechend aufwändig formulierte abfrage3 mit einer WHERE-Klausel unter Verwendung von NOT EXISTS oder NOT IN .

UNION alles der beiden

Vereinigungsmenge: Die Resultat-Tupel kommen in mindestens einer der beteiligten Abfragen vor. Lösung:

- entweder Syntax: **abfrage1 UNION abfrage2**
- oder entsprechend aufwändig formulierte abfrage3 mit WHERE ... OR.

Eine Anwendung könnte z.B. darin bestehen, mehrere Adressbestände für einen Massenversand zu vereinigen und dabei Dubletten zu eliminieren (was allerdings nur bei völliger Resultatgleichheit passiert).

```
SELECT dt_name AS 'Ergebnis' FROM tkey_interpret WHERE dt_name LIKE '%boy%'
UNION
SELECT dt_stueck_titel FROM tbl_stueck WHERE dt_stueck_titel LIKE '%boy%'
UNION
SELECT dt_stao FROM tkey_standort WHERE dt_stao LIKE '%boy%'
go
```

LIKE

13. April 2015 17:36

LIKE

%	Eine Zeichenfolge aus null oder mehr Zeichen	WHERE title LIKE '%Computer%' findet alle Buchtitel, die das Wort 'Computer' enthalten.
_ (Unterstrich)	Ein einzelnes Zeichen	WHERE au_fname LIKE '_ean' findet alle Vornamen mit vier Buchstaben, die auf ean enden (Dean, Sean usw.).
[]	Beliebiges einzelnes Zeichen im angegebenen Bereich ([a-f]) oder in der angegebenen Menge ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' findet alle Autorennachnamen, die auf 'arsen' enden und mit einem einzelnen Zeichen zwischen C und P beginnen, z. B. Carsen, Larsen, Karsen usw. In Bereichssuchvorgängen unterscheiden sich die im Bereich enthaltenen Zeichen möglicherweise je nach den Sortierungsregeln für die Sortierung.
[^]	Beliebiges einzelnes Zeichen, das sich nicht im angegebenen Bereich ([^a-f]) oder in der angegebenen Menge ([^abcdef]) befindet.	WHERE au_lname LIKE 'de[^]%' findet alle Autoren Nachnamen, die mit 'de' beginnen und deren dritter Buchstabe nicht l ist.

INSERT, UPDATE, DELETE, TRUNCATE, DROP

17. April 2015 11:05

INSERT

The first form does not specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
VALUES (value1,value2,value3,...);
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

UPDATE

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

Example:

```
UPDATE Customers
SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName='Alfreds Futterkiste';

SELECT id_kunde, name, moral_bez, bezahlt;
UPDATE tass_police SET bezahlt = bezahlt - 10
FROM tkey_moral JOIN tbl_kunde ON id_moral_nr = fi_moral_nr
JOIN tass_police ON id_kunde = id-fi_kunde
WHERE moral_bez = 'sehr gut';
```

DELETE

```
DELETE FROM table_name
WHERE some_column=some_value;
```

Example:

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';

-- Mit einer Unterabfrage
DELETE FROM tbl_kunde WHERE id_kunde IN(
SELECT id_kunde FROM tbl_kunde
EXCEPT
SELECT id-fi_kunde FROM tass_police);

-- Mit Joining
DELETE FROM tbl_kunde
FROM tbl_kunde LEFT OUTER JOIN tass_police ON id_kunde = id-fi_kunde
WHERE id-fi_kunde IS NULL;
```

TRUNCATE

Es ist auch möglich, nur die Daten zu löschen, nicht jedoch die Tabelle selbst. Diese Funktion erfüllt der Befehl **TRUNCATE TABLE**. Die Syntax für **TRUNCATE TABLE** lautet:

```
TRUNCATE TABLE "Tabellen_Name";
```

DROP

Manchmal ist es wünschenswert, eine Tabelle aus welchem Grund auch immer aus der Datenbank zu löschen. Die Syntax für **DROP TABLE** lautet:

```
DROP TABLE "Tabellen_Name";
```

Transaktionen

15. Juni 2015 15:24

Autocommit-Modus:	«Automatische Transaktionen werden beim SQL Server standardmässig verwendet, falls keine Transaktion explizit gestartet wird. Sie werden bei jedem einzelnen Befehl gestartet und nach dessen Verarbeitung sofort abgeschlossen. Das heisst, Änderungen werden bei einer erfolgreichen Verarbeitung einer DML-Anweisung sofort festgeschrieben.
--------------------------	---

--> normal

Expliziter Modus:	«Explizite Transaktionen werden vom Benutzer gestartet [und beendet, VZ] und immer dann benötigt, wenn eine Transaktion aus mehr als einer Anweisung besteht.
--------------------------	---

```
BEGIN TRANSACTION
```

```
BEGIN TRY
```

```
    INSERT INTO transi_test VALUES (6,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (7,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (8,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (6,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (10,'Bezeichnung', 'Bemerkung')
    PRINT 'Transaktion erfolgreich';
    COMMIT TRANSACTION;
```

```
END TRY
```

```
BEGIN CATCH
```

```
    PRINT 'Transaktion NICHT erfolgreich';
    ROLLBACK TRANSACTION;
```

```
END CATCH
```

--> Funktioniert gemäss dem «alles-oder-nichts»-Prinzip.

Impliziter Modus:	«Wenn Sie nicht jede Transaktion manuell [explizit, VZ] starten möchten, aber sich dennoch die Möglichkeit offenlassen wollen, Änderungen gegebenenfalls wieder rückgängig zu machen, dann verwenden Sie implizite Transaktionen.»
--------------------------	--

```
SET IMPLICIT_TRANSACTIONS ON;
```

```
-- SET IMPLICIT_TRANSACTIONS OFF;
```

```
    INSERT INTO transi_test VALUES (1,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (2,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (3,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (4,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (5,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (6,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (7,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (8,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (6,'Bezeichnung', 'Bemerkung');
    INSERT INTO transi_test VALUES (10,'Bezeichnung', 'Bemerkung');
```

```
GO
```

```
-- COMMIT;
```

```
-- ROLLBACK;
```


Mehrbenutzersynchronisation

15. Juni 2015 15:54

Die aktuelle Isolationsstufe einer Verbindung ermitteln Sie wie folgt:

```
/* aktuelle Isolationsstufe */
SELECT
  CASE transaction_isolation_level
    WHEN 0 THEN 'Unspecified'
    WHEN 1 THEN 'ReadUncommitted'
    WHEN 2 THEN 'ReadCommitted'
    WHEN 3 THEN 'Repeatable'
    WHEN 4 THEN 'Serializable'
    WHEN 5 THEN 'Snapshot'
  END AS Isolationsstufe
FROM sys.dm_exec_sessions
WHERE session_id = @@SPID;

-- Ausführung mit Annullation
BEGIN TRANSACTION
go
UPDATE tass_police SET bezahlt=0
WHERE id_fi_kunde = 2367 AND id_fi_vers_art = 1500 UPDATE tass_police SET
bezahlt=880
WHERE id_fi_kunde = 2367 AND id_fi_vers_art = 1600 go
ROLLBACK TRANSACTION
go
-- nichts ist passiert:

-- Ausführung mit Erfolg
BEGIN TRANSACTION
go
UPDATE tass_police SET bezahlt=0
WHERE id_fi_kunde = 2367 AND id_fi_vers_art = 1500 UPDATE tass_police SET
bezahlt=880
WHERE id_fi_kunde = 2367 AND id_fi_vers_art = 1600 go
COMMIT TRANSACTION
go
-- es ist etwas passiert
```

Elemente der Programmierung

15. Juni 2015 16:13

Variablen

- sind nur gültig innerhalb eines Stapels oder einer Prozedur
- tragen als Auszeichnung an führender Position einen Klammeraffen
- werden deklariert mit DECLARE und sind danach mit NULL belegt
- werden belegt mit SET oder SELECT (siehe Online Hilfe und folgende Beispiele)
- können überall verwendet werden, wo auch eine Konstante verwendet werden könnte, z.B. als Teil einer WHERE-Bedingung.

```
DECLARE @lokale_variable Datentyp [ ,...n]
SET @lokale_variable = {Ausdruck | (SELECT-Anweisung)}
```

Bsp.:

```
DECLARE @lokale_variable1 AS char(5) = 'gugus';
DECLARE @lokale_variable2 AS char(5);
SET @lokale_variable2=@lokale_variable1; -- Zuweisung einer Variablen
PRINT @lokale_variable2; -- in Konsole, danach Zeilenschaltung
GO
```

Kontrollstrukturen

BEGIN...END	Definiert einen Anweisungsblock.
BREAK	Verlässt die innerste WHILE-Schleife.
CONTINUE	Setzt die Bearbeitung am Beginn der WHILE-Schleife fort.
GOTO	Führt mit der Bearbeitung bei der Anweisung fort, die sich hinter der von label angegebenen Sprungmarke befindet.
IF...ELSE	Definiert eine bedingte Ausführung und eine (optionale) Alternative, wenn die Bedingung gleich FALSE ist.
RETURN	Bewirkt einen unbedingten Abbruch und gibt u.U. einen Wert zurück.
WAITFOR	Setzt eine Verzögerungsdauer oder einen Zeitpunkt für die Ausführung einer Anweisung.
WHILE	Wiederholt Anweisungen, solange eine bestimmte Bedingung gleich TRUE ist.
CASE ... END	Bedingte Ausführung je nach Ergebnis eines Ausdrucks

CASE

15. Juni 2015 16:25

Simple CASE (simple comparison operation) :

```
SELECT name AS 'Familiennamen', wohnort AS 'Wohnort',
CASE fi_moral_nr
  WHEN 1 THEN 'naja, liegt im Keller'
  WHEN 2 THEN 'kommt, gebt ihm eine Chance'
  WHEN 3 THEN 'das helle Licht in dunkler Zeit'
  ELSE 'gewissermassen moralfrei'
END AS 'Zahlungsmoral',
id_kunde AS 'Kundennummer'
FROM tbl_kunde;
GO
```

Familiennamen	Wohnort	Zahlungsmoral	Kundennummer
Meier	Luzern	kommt, gebt ihm eine Chance	1795
Müller	Bern	das helle Licht in dunkler Zeit	1809
Klausen	Zürich	kommt, gebt ihm eine Chance	2367
Hauser	Kriens	naja, liegt im Keller	3533
Menzer	Horw	das helle Licht in dunkler Zeit	3788

Tabelle 12: Die Wirkung von Simple CASE

Searched CASE (boolean searched operation):

```
PRINT 'Transaktionszähler' + CHAR(10) + CHAR(13) + 'Derzeit aktiv: ' +
' +
CASE
  WHEN @@TRANCOUNT = 0 THEN 'keine Transaktion. Keine Probleme.'
  WHEN @@TRANCOUNT BETWEEN 1 AND 5 THEN 'einige Transaktionen. Geringe Gefahr
    von Deadlock.'
  WHEN @@TRANCOUNT BETWEEN 6 AND 15 THEN 'viele Transaktionen. Gefahr von
    Deadlocks.'
  WHEN @@TRANCOUNT > 15 THEN 'sehr viele Transaktionen. Performanzprobleme
    möglich.'
END;
END;
```

PRINT 'Transaktionszähler' + CHAR(10) + CHAR(13) + 'Derzeit aktiv: ' + CASE WHEN @@TRANCOUNT = 0 THEN 'keine Transaktion. Keine Probleme.' WHEN @@TRANCOUNT BETWEEN 1 AND 5 THEN 'einige Transaktionen. Geringe Gefahr von Deadlock.' WHEN @@TRANCOUNT BETWEEN 6 AND 15 THEN 'viele Transaktionen. Gefahr von Deadlocks.' WHEN @@TRANCOUNT > 15 THEN 'sehr viele Transaktionen. Performanzprobleme möglich.' END
Transaktionszähler
Derzeit aktiv: viele Transaktionen. Gefahr von Deadlocks.

Tabelle 13: Codierung von Searched CASE

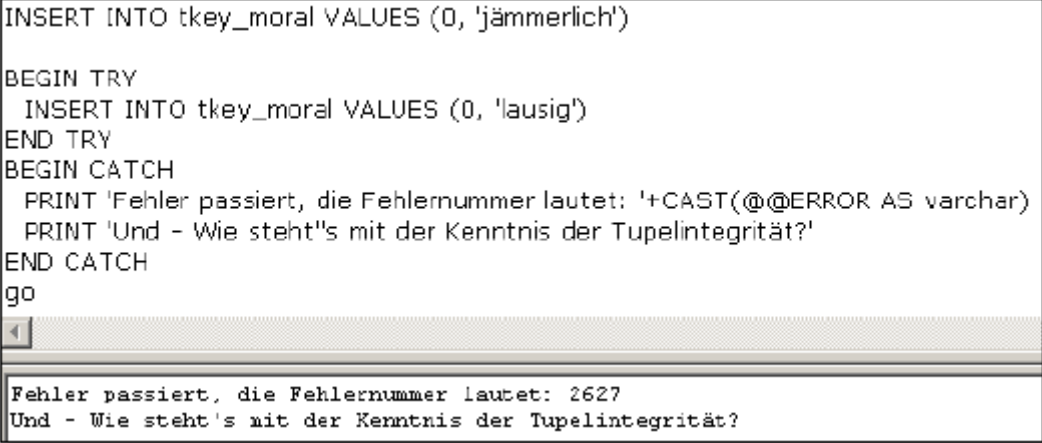
TRY ... CATCH

15. Juni 2015 16:37

```
BEGIN TRY
    { anweisung | anweisungsblock }
END TRY
BEGIN CATCH
    { anweisung | anweisungsblock }
END CATCH
```

Beispiel

```
INSERT INTO tkey_moral VALUES (0, 'jämmerlich');
BEGIN TRY
    INSERT INTO tkey_moral VALUES (0, 'lausig');
END TRY
BEGIN CATCH
    PRINT 'Fehler passiert, die Fehlernummer lautet: '
    +CAST(@@ERROR AS varchar);
    PRINT 'Und - Wie steht's mit der Kenntnis der Tupelintegrität?';
END CATCH
GO
```



```
INSERT INTO tkey_moral VALUES (0, 'jämmerlich')
BEGIN TRY
    INSERT INTO tkey_moral VALUES (0, 'lausig')
END TRY
BEGIN CATCH
    PRINT 'Fehler passiert, die Fehlernummer lautet: '+CAST(@@ERROR AS varchar)
    PRINT 'Und - Wie steht's mit der Kenntnis der Tupelintegrität?'
END CATCH
go
```

Fehler passiert, die Fehlernummer lautet: 2627
Und - Wie steht's mit der Kenntnis der Tupelintegrität?

Funktionen

15. Juni 2015 16:43

Zeichenketten-Funktionen		
ASCII	NCHAR	SOUNDEX
CHAR	PATINDEX	SPACE
CHARINDEX	QUOTENAME	STR
DIFFERENCE	REPLACE	STUFF
LEFT	REPLICATE	SUBSTRING
LEN	REVERSE	UNICODE
LOWER	RIGHT	UPPER
LTRIM	RTRIM	
Systemstatistische Funktionen		
@@CONNECTIONS	@@PACK_RECEIVED	@@TOTAL_ERRORS
@@CPU_BUSY	@@PACK_SENT	@@TOTAL_READ
@@IDLE	@@PACKET_ERRORS	@@TOTAL_WRITE
@@IO_BUSY	@@TIMETICKS	fn_virtualfilestats

- gibt einmal einen Überblick und dokumentiert die Objekt -ID der Funktion

```
SELECT * FROM sys.objects;
```

- listet alle Funktionen und dokumentiert deren Quellencode; vom Quellencode ist nur die erste Zeile sichtbar, aber copy and paste erschliesst den ganzen Code

```
SELECT * FROM sys.sql_modules;
```

- isScalarFunction ist eine der vielen Eigenschaften, die sich überprüfen und mit true oder false bestätigen lassen; siehe Original-Dokumentation

```
SELECT OBJECTPROPERTY (object_id, 'isScalarFunction');
```

- gibt den Quellencode der Funktion funktions_name zurück; vom Quellencode ist nur die erste Zeile sichtbar, aber copy and paste erschliesst den ganzen Code
- selbstverständlich könnte auch direkt die Objekt -ID eingegeben werden

```
SELECT OBJECT_DEFINITION ((SELECT object_id ('funktions_name')));
```

- zeigt ebenfalls den ganzen Quellencode der Funktion als mehrzeilige Tabelle, gibt ihn mit copy and paste aber nicht so einfach her; gefährlich ist zudem, dass sich die Zeilen im Klienten sortieren lassen:

```
EXEC sp_helptext 'funktions_name';
```

Funktionen

15. Juni 2015 16:58

```
CREATE FUNCTION funktions_name
    ([@parameter_name [AS] Datentyp [= Initialisierung] [,...n ]])
    RETURNS Datentyp
    [ AS ]
    BEGIN
        function_body
    RETURN [Ausdruck]
    END
```

Ändern (Rechtsklick auf Funktion in QueryExPlus)

```
ALTER FUNCTION funktions_name ...
```

Löschen

```
DROP FUNCTION funktions-name [, ...]
```

Rechte eines (eröffneten) Benutzers

```
GRANT EXECUTE ON funktions_name TO db_prinzipal [, ...]
```

Beispiele

A1:	<p>Schreiben Sie eine Funktion, welche durch den Erzeuger wie folgt aufgerufen wird:</p> <pre>SELECT dbo.f_polic_bez() AS 'Summe aller bezahlten Leistungen' go CREATE FUNCTION f_polic_bez() RETURNS DECIMAL(8,2) AS BEGIN DECLARE @bez_summe DECIMAL(8,2) SET @bez_summe = (SELECT SUM(bezahlt) FROM tass_police) RETURN (@bez_summe) END</pre>
A2:	<p>Diese Funktion ermittelt die pro Versicherungsprodukt total bezahlten Leistungen:</p> <pre>SELECT dbo.f_bezahlt_versich('Auto') go SELECT dbo.f_bezahlt_versich('hausrat') go CREATE FUNCTION f_bezahlt_versich(@versicherung VARCHAR(30)) RETURNS DECIMAL(8,2) AS BEGIN DECLARE @v_summe DECIMAL(8,2) SET @v_summe = (SELECT SUM(bezahlt) FROM tkey_versicherung JOIN tass_police ON id_vers_art = id-fi_vers_art WHERE vers_bez = @versicherung) RETURN (@v_summe) END</pre>
A3:	<p>Die Buchhaltung will Auskunft über bezahlte Leistungen in Dollar. Sie konstruieren eine Funktion, bei deren Aufruf Sie eine Police und den aktuellen Dollarkurs mitgeben. Die Funktion meldet die bezahlten Leistungen in Dollar.</p> <pre>SELECT dbo.f_bezahlt_in_dollar('Menzer', 'Haftpflcht', 1.16) AS 'Bezahlt' go SELECT dbo.f_bezahlt_in_dollar('Müller', 'Hausrat', 1.14) AS 'Bezahlt' go CREATE FUNCTION f_bezahlt_in_dollar (@kunde VARCHAR(30), @versicherung VARCHAR(30), @kurs FLOAT) RETURNS DECIMAL(6,2) AS BEGIN DECLARE @leistung DECIMAL(6,2) SET @leistung = (SELECT bezahlt FROM tbl_kunde JOIN tass_police ON id_kunde = id-fi_kunde JOIN tkey_versicherung ON id-fi_vers_art = id_vers_art WHERE name = @kunde AND vers_bez = @versicherung) SET @leistung = @leistung / @kurs RETURN (@leistung) END go</pre>

Prozeduren

15. Juni 2015 17:13

```
CREATE PROC[EDURE] prozedur_name
[@parameter_name [AS] Datentyp [= Initialisierung] [OUTPUT]]
[,...n ]
[ AS ]
BEGIN
procedure_body
[RETURN Ausdruck]
END
```

Ändern

```
ALTER PROCEDURE prozedur_name ...
```

Löschen

```
DROP PROCEDURE prozedur_name [, ...]
```

Rechte eines (eröffneten) Benutzers

```
GRANT EXECUTE ON prozedur_name TO userid [, ...];
```

Die Prozedur enthält eine einfache Abfrage. Die Resultate sind allen für die Prozedur autorisierten Anwenderinnen zugänglich, unabhängig von ihren Objektberechtigungen auf die abgefragte Tabelle:

```
CREATE PROCEDURE p_ku_liste @name VARCHAR(40)
AS
BEGIN
    SELECT name AS 'Name', vorname AS 'Vorname'
    FROM tbl_kunde
    WHERE name LIKE @name
    ORDER BY name, vorname
    PRINT CAST (@@ROWCOUNT AS VARCHAR(10)) + ' Personen'
    RETURN @@ROWCOUNT
END;
GO

EXECUTE p_ku_liste 'Me%';
```

Name	Vorname
Meier	Laura
Meier	Max
Menzer	Claudia

Die folgende Prozedur wird mit einer Variablen als Parameter aufgerufen. Sie verändert deren Wert, der im gleichen Stapel verfügbar bleibt, wenn die Prozedur beendet ist (es könnten auch mehrere Variablen so verändert werden):

```
CREATE PROC p_ums_tot @umsatz int OUTPUT -- Variante 1
AS
BEGIN
    SET @umsatz = (SELECT sum(bezahlt) FROM tass_police);
END;
GO
```

Folgender Aufruf muss als *ein* go-Stapel laufen. Er zeigt ferner, dass Kontrollstrukturen (z.B. IF) auch in normalen SQL-Stapeln verwendet werden können:

```
DECLARE @umsatz_total INT;
EXECUTE p_ums_tot @umsatz_total OUTPUT;

IF (@umsatz_total > 20000)
    PRINT 'Wir haben wieder mal vorwärts gemacht.';
ELSE PRINT 'Wir sollten uns sputen.';
GO
```

Prozeduren

15. Juni 2015 17:21

Weil obige Prozedur *nur einen Wert* zurück gibt, könnte auch wie folgt codiert werden. Die beiden Formen (OUTPUT-Variablen und RETURN) können vermischt werden. Das Beispiel zeigt ferner, dass in der Prozedur und ausserhalb die selben Variablennamen verwendet werden können - aber nicht müssen. Das Gleiche gilt auch für die obige Variante:

```
ALTER PROCEDURE p_ums_tot -- Variante 2
AS
BEGIN
    DECLARE @u INT
    SET @u = (SELECT sum(bezahlt) FROM tass_police);
    RETURN @u;
END
GO

DECLARE @umsatz INT;
EXEC @umsatz = p_ums_tot;

IF (@umsatz > 20000)
    PRINT 'Wir haben wieder mal vorwärts gemacht.';
ELSE PRINT 'Wir sollten uns sputen.';
GO
```

Die folgende Prozedur ruft eine Versicherungspolice auf und reduziert deren Rabatt um 10% - falls dies überhaupt möglich ist:

```
CREATE PROCEDURE p_rabatt_mut @fname VARCHAR(30), @versich VARCHAR(30)
AS
BEGIN
    DECLARE @anzahl INT;
    UPDATE tass_police SET praem_stufe = praem_stufe-1 WHERE

        (praem_stufe > 100 AND
         id-fi-vers_art = @versich AND
         (id-fi_kunde IN (SELECT id_kunde FROM tbl_kunde WHERE name =
         @fname)))
    );
    SET @anzahl = @@ROWCOUNT;
    PRINT CAST (@anzahl AS VARCHAR(10)) + ' Policen mutiert.';
    RETURN @anzahl; -- Die Anzahl wird zurückgegeben und unten weiter
verwendet
END
GO

DECLARE @a AS INT;
EXEC @a = p_rabatt_mut 'Müller', 1200;
IF @a = 0 PRINT 'Es hat nicht geklappt!';
GO
```

Diskussion

- Beachten Sie die Systemfunktion @@ROWCOUNT.
- Die Aktivität wird gemeldet (PRINT) und die Prozedur gibt als Rückgabewert die Anzahl betroffener Tupel mit. Damit können nach dem Prozeduraufruf abhängige Massnahmen getroffen werden

Prozeduren

15. Juni 2015 17:33

A1: Schreiben Sie eine Prozedur, welche nach der Eingabe des Kundennamens und der Versicherungsbezeichnung die entsprechende Police löscht.

- Ein Aufruf EXECUTE p_polic_del 'Meier', 'Taggeld' go --> sollte mit Police existiert nicht. quittiert werden
- Ein Aufruf EXECUTE p_polic_del 'Meier', 'Leben' go --> sollte mit Löschung vollzogen. quittiert werden.

```
CREATE PROCEDURE p_polic_del @fname VARCHAR(30), @versich VARCHAR(30) AS
BEGIN
    DELETE FROM tass_police
    FROM tbl_kunde, tkey_versicherung
    -- Formulierung des Join mit WHERE, weil eine beteiligte Tabelle oben genannt ist --
    -- die Tabelle tass_police könnte hier allerdings nochmals aufgeführt werden
    WHERE name = @fname
        AND id_kunde = id-fi_kunde
        AND id-fi_vers_art = id-vers_art AND vers_bez = @versich
    IF @@ROWCOUNT = 0
        PRINT 'Police existiert nicht.'
    ELSE PRINT 'Löschung vollzogen.'
END
```

A2: Schreiben Sie eine Prozedur p_polic_bez(), über welche Sie eine von Ihrer Gesellschaft bezahlte Leistung für eine Police (Kunde, Produkt) zur Summe der bisherigen Leistungen addieren können.

- Die Prozedur hat drei Eingabe- und einen Rückgabe-Parameter (OUTPUT oder RETURN).
- Der Betrag für Ihre Leistung sollte via eine SQL-Variable aufrufbar sein - siehe unten.
- Die Prozedur sollte melden, wenn es ein Tupel nicht gibt.
- Die Prozedur sollte die Erhöhung des Betrags und dessen neuen Wert melden.
- Ein- oder Rückzahlungen des Kunden kann man durch die Eingabe von negativen Werten vornehmen.
- Keine Plausibilitätsprüfungen - wir gehen von einer korrekten Bedienung aus.

```
CREATE PROCEDURE p_polic_bez @fname VARCHAR(30), @versich VARCHAR(30), @leistung SMALLINT,
@bezahlt INT OUTPUT
AS
BEGIN
    UPDATE tass_police
    SET bezahlt = bezahlt + @leistung
    FROM tbl_kunde
        JOIN tass_police ON id_kunde = id-fi_kunde
        JOIN tkey_versicherung ON id-fi_vers_art = id-vers_art
        WHERE name = @fname AND vers_bez = @versich
    IF @@ROWCOUNT = 0
        PRINT 'Police existiert nicht.'
    ELSE
        BEGIN
            SELECT @bezahlt = bezahlt
            FROM tbl_kunde
                JOIN tass_police ON id_kunde = id-fi_kunde
                JOIN tkey_versicherung ON id-fi_vers_art = id-vers_art WHERE name = @fname AND
                vers_bez = @versich
            PRINT 'Änderung vollzogen, die Leistungen belaufen sich neu auf ' +
            CAST(@bezahlt AS VARCHAR) + ' Franken.'
        END
END
```

Prozeduren

15. Juni 2015 17:44

```
-- 3.
ALTER TABLE tass_police ADD praemie SMALLINT
go

CREATE PROCEDURE p_polic_praem @fname VARCHAR(30), @versich VARCHAR(30), @betrag
SMALLINT
AS
BEGIN
    DECLARE @temp_praem SMALLINT
    SELECT @temp_praem = praemie
    FROM tbl_kunde
    JOIN tass_police ON id_kunde = id_fi_kunde
    JOIN tkey_versicherung ON id_fi_vers_art = id_vers_art
    WHERE name = @fname AND vers_bez = @versich;
    IF @@ROWCOUNT = 0
        PRINT ('Police existiert nicht.')
    ELSE
        IF (@temp_praem IS NULL OR @temp_praem = 0)
            BEGIN
                UPDATE tass_police
                SET praemie = @betrag
                FROM tbl_kunde
                JOIN tass_police ON id_kunde = id_fi_kunde
                JOIN tkey_versicherung ON id_fi_vers_art = id_vers_art
                WHERE name = @fname AND vers_bez = @versich
                SELECT @temp_praem = praemie
                FROM tbl_kunde
                JOIN tass_police ON id_kunde = id_fi_kunde
                JOIN tkey_versicherung ON id_fi_vers_art = id_vers_art
                WHERE name = @fname AND vers_bez = @versich
                PRINT ('Erledigt. Prämienbetrag neu: ' + CAST(@temp_praem AS VARCHAR(10)))
            END
        ELSE
            IF ((@temp_praem > 0) AND (@betrag > 0))
                PRINT ('Kunde hat noch Prämienschulden! Nachtrag nicht erledigt!
                Schickt Harley Davidson Konvoi vorbei!')
            ELSE
                BEGIN
                    UPDATE tass_police
                    SET praemie = praemie + @betrag
                    FROM tbl_kunde
                    JOIN tass_police ON id_kunde = id_fi_kunde
                    JOIN tkey_versicherung ON id_fi_vers_art = id_vers_art
                    WHERE name = @fname AND vers_bez = @versich
                    SELECT @temp_praem = praemie
                    FROM tbl_kunde
                    JOIN tass_police ON id_kunde = id_fi_kunde
                    JOIN tkey_versicherung ON id_fi_vers_art = id_vers_art
                    WHERE name = @fname AND vers_bez = @versich
                    PRINT ('Erledigt. Prämienbetrag neu: ' + CAST(@temp_praem AS VARCHAR(10)))
                END
            END
    go
```

Cursor

15. Juni 2015 17:45

- definiert eine Tupelmenge zur Bearbeitung
- erlaubt das Vorwärts- und (ggf.) Rückwärts bewegen innerhalb der Menge
- muss nach Gebrauch geschlossen und der Speicherplatz freigegeben werden

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[;]
```

```
DECLARE @nachname VARCHAR(40);
DECLARE @vorname VARCHAR(40);
DECLARE cur_kunde CURSOR
FOR SELECT name, vorname FROM tbl_kunde
ORDER BY name, vorname;
```

```
OPEN cur_kunde;
FETCH NEXT FROM cur_kunde
INTO @nachname, @vorname;
```

```
WHILE @@FETCH_STATUS = 0
BEGIN
/* hier müsste jetzt der effektive Versand greifen... Wir begnügen uns mit einer
einfachen Ausgabe an die Konsole */
```

```
PRINT 'E-Mail an ' + @nachname + ', ' + @vorname;
FETCH NEXT FROM cur_kunde
    INTO @nachname, @vorname;
END;
CLOSE cur_kunde;
DEALLOCATE cur_kunde;
```

Trigger

```
CREATE TRIGGER trigger_name
ON { table | view }
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
BEGIN
Anweisungsblock
ENDE
```

Ändern

```
ALTER TRIGGER trigger-name ...
```

Löschen

```
DROP TRIGGER trigger-name [, ...n]
```

DELETED / INSERTED:	gelöschte Tupel vor der Änderung bzw. eingefügte Tupel nach der Änderung
BEFORE Trigger	sehen die Daten noch nicht, welche sie ändern (werden / würden)! Sie feuern schon bevor der Code zur Wirkung kommt, der Manipulationen an der Triggertabelle vornehmen möchte. BEFORE Trigger können je nachdem diese Manipulationen sogar unterbinden!
AFTER Trigger	feuere, nachdem die Manipulationen an der Triggertabelle vollständig vollzogen sind und dabei auch keinerlei Einschränkungen verletzt wurden (z.B. doppelter Primärschlüsselwert), also nach der Prüfung der CONSTRAINTs. Ein AFTER Trigger kann die Persistierung dieser Manipulationen mit ROLLBACK TRANSACTION "gerade noch" verhindern.
FOR EACH ROW	SQL ist mengenorientiert. UPDATES und DELETES betreffen immer Tupelmengen. Mit FOR EACH ROW kann deshalb explizit befohlen werden, dass ein Trigger auf jedes einzelne von UPDATE oder DELETE betroffene Tupel separat feuern soll.

Denken Sie beim Definieren eines Triggers immer daran, dass mehrere Tupel vom Ereignis betroffen sind! Ein häufiger Fehler ist, nur ein Tupel zu behandeln.

- Pro INSERT / UPDATE / DELETE ist nur ein INSTEAD OF Trigger zugelassen.
- SET NOCOUNT ON ist zu empfehlen, u.a. damit Echos den Ablauf nicht stören und damit der Trigger unsichtbar bleibt.
- Die logischen Tabellen inserted und deleted erlauben nur lesenden Zugriff:
 - o inserted kennt alle Tupel die von INSERT oder UPDATE betroffen waren und jetzt persistent "Gültigkeit erlangen wollen"
 - o deleted kennt alle Tupel, die gelöscht oder geändert wurden, nicht mehr aktuell sind und persistent "entfernt werden wollen"

Metadaten zu den Triggern:

```
SELECT * FROM sys.triggers
```

✓ gibt einen Überblick über alle benutzerspezifischen DML-Trigger, dokumentiert dessen Objekt-ID, das Erstellungs- und Änderungsdatum und anderes

```
EXEC sp_helptrigger 'tabellen_name'
```

✓ listet alle Trigger zur Tabelle `tabellen_name`, den Eigentümer und das Schema sowie die Ereignisse und den Ereigniszeitpunkt

```
SELECT * FROM sys.sql_modules
```

✓ listet u.a. alle Trigger und dokumentiert deren Quellencode; vom Quellencode ist nur die erste Zeile sichtbar, aber copy and paste erschliesst den ganzen Code

Trigger

1. Juli 2015 09:40

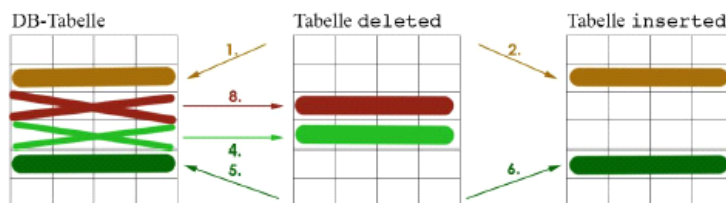
```
SELECT * FROM sys.trigger_events
```

✓ listet für jeden Trigger (identifiziert durch seine Objekt-ID) die zugehörigen Tabellenereignisse und allfällige Definition als Erst - oder Letzt-Trigger

```
EXEC sp_helptext 'trigger_name'
```

✓ zeigt ebenfalls den ganzen Quellencode des Triggers als mehrzeilige Tabelle, gibt ihn mit copy and paste aber nicht so einfach her; gefährlich ist zudem, dass sich die Zeilen im Klienten sortieren lassen

Trigger - Tabellendynamik



Vorgänge:

INSERT INTO DB-Tabelle; was passiert?

1. einfügen in DB-Tabelle
2. kopieren in automatisch generierte Tabelle inserted
=> Tupel in DB-Tabelle = Tupel in inserted
3. impl. / expl. COMMIT vor impl. / expl. END

UPDATE DB-Tabelle; was passiert?

4. urspr. Tupel aus DB-Tabelle in Tabelle deleted übertragen
5. einfügen des 'neuen' Tupels in DB-Tabelle gemäss UPDATE-Anweisung
6. kopieren des 'neuen' Tupels in Tabelle inserted
=> Tupel in DB-Tabelle = Tupel in inserted
7. impl. / expl. COMMIT vor impl. / expl. END

DELETE FROM DB-Tabelle; was passiert?

8. zu löschenden Tupel aus DB-Tabelle in Tabelle deleted übertragen
9. impl. / expl. COMMIT vor impl. / expl. END

1. Trigger wurden und werden oft benutzt **zur Sicherung der relationalen und sonstigen Einschränkungen (Constraints)**. Unser Beispiel demonstriert dies, indem wir in einer Tabelle `t_demo` die Primärschlüssel-bedingung selber als Trigger implementieren (statt sie einfach bei der Definition der Spalte zu definieren).

Trigger

```
CREATE TRIGGER constraint_triggi ON t_demo
```

```
AFTER INSERT, UPDATE
```

```
AS
```

```
BEGIN
```

```
SET nocount on;
```

```
IF (SELECT COUNT(*) FROM t_demo JOIN inserted ON t_demo.id=inserted.id) >  
    (SELECT COUNT(*) FROM inserted)
```

```
BEGIN
```

```
    RAISERROR ('Attribut id lässt keine Doppelnennungen zu.', 15, 1);
```

```
    ROLLBACK TRANSACTION;
```

```
END
```

```
ELSE
```

```
IF ((SELECT COUNT(*) FROM inserted WHERE id IS NULL) > 0)
```

```
BEGIN
```

```
    RAISERROR ('Attribut id darf nicht NULL sein.', 15, 1);
```

```
    ROLLBACK TRANSACTION;
```

```
END
```

```
END
```

```
GO
```

Trigger

1. Juli 2015 09:54

- nocount on verhindert Rauschen.
- RAISERROR generiert eine ad hoc Fehlermeldung. RAISERROR sorgt für eine Fehlerbehandlung "ganz weit unten", also durch die Datenbankmaschine.
- AFTER-Trigger feuert erst nach vollzogenen Änderungen. Im Fehlerfall muss also die Änderung mit einem ROLLBACK rückgängig gemacht werden.
- ROLLBACK TRANSACTION braucht kein korrespondierendes, explizites BEGIN TRANSACTION. Es stellt einfach den Urzustand her, als wäre nichts passiert. Die Manipulation wurde nie persistiert.
- Vorsicht mit ELSE IF. Hier von der Logik her notwendig, weil nach dem ersten ROLLBACK TRANSACTION nichts mehr zu tun ist!

2. Zwei (noch zu ergänzende) Attribute in der Tabelle tbl_kunde sollen bei jedem Ändern eines Kunden die ID des erfassenden Benutzers sowie einen Zeitstempel an dieses Tupel anfügen:

```
CREATE TRIGGER t_ku_ins ON tbl_kunde
AFTER UPDATE
AS
BEGIN
SET nocount on
UPDATE tbl_kunde SET mut_id = SYSTEM_USER, mut_zeit = CURRENT_TIMESTAMP
FROM tbl_kunde JOIN inserted
ON tbl_kunde.id_kunde = inserted.id_kunde END
GO
```

3. Das Folgende ist ebenfalls ein Journal führender Trigger: t_praem_mut_log **vermerkt** in einer separaten Tabelle tbl_praem_log, **wann von wem welche Police gelöscht wurde**. Der Trigger liesse sich problemlos ausweiten auf die Protokollierung von UPDATES sowie INSERTs in die Tabelle tbl_praem_log.

```
CREATE TRIGGER t_praem_mut_log ON tass_police
AFTER DELETE
AS
BEGIN
SET NOCOUNT ON
DECLARE @polic_id VARCHAR(11);
SET @polic_id = CAST((SELECT id_fi_kunde FROM deleted) AS CHAR(4)) + ' / '
+ CAST((SELECT id_fi_vers_art FROM deleted) AS CHAR(4));
INSERT tbl_praem_log (mut_police_id, mut_art) VALUES (@polic_id, 'del');
END -- Trigger
GO
```

4. Der folgende **Löschtrigger verhindert die Löschung von Kunden mit guter Moral**:

```
CREATE TRIGGER t_ku_del ON tbl_kunde
AFTER DELETE
AS
BEGIN
DECLARE @m AS TINYINT, @e AS VARCHAR(80);
SET nocount on;
SET @m = (SELECT MAX(fi_moral_nr) FROM deleted);
IF @m >= 3
BEGIN
SET @e = 'Kunden mit Moral ' + CAST(@m AS CHAR(1))
+ ' werden nicht gelöscht!';
RAISERROR(@e, 15, 1);
ROLLBACK TRANSACTION;
END
END
```

Trigger

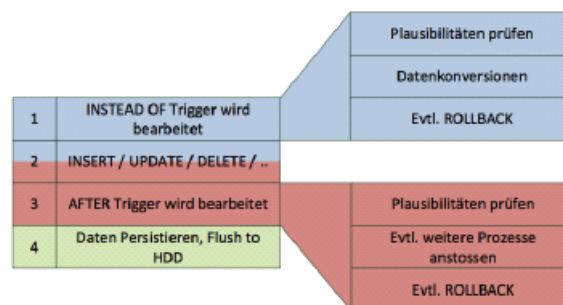
1. Juli 2015 10:05

5. Folgender INSTEAD OF Trigger sorgt dafür, dass **bei einem INSERT** in unserer Moraltabelle die **Tupelintegrität nicht verletzt wird**:

```
CREATE TRIGGER t_moral_pk_ins
ON tkey_moral
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON
SELECT i.id_moral_nr FROM inserted i
WHERE i.id_moral_nr IN (SELECT m.id_moral_nr FROM tkey_moral m)
IF @@ROWCOUNT > 0
RAISERROR ('Moral-INS-03: Tupelintegrität!!!',15,255);
ELSE
INSERT INTO tkey_moral
SELECT * FROM inserted;
END;
```

15.7 Vergleich INSTEAD OF / AFTER

Hier sehen Sie eine graphische Darstellung des Unterschiedes zwischen INSTEAD und AFTER Triggern:



Der Unterschied zwischen Instead of und After

INSTEAD OF: Der Triggercode wird anstatt (instead of) der eigentlichen Anweisung ausgeführt. Der Besitzer des Triggers muss den jeweiligen Mutationsbefehl selber im Trigger definieren. Bsp: INSTEAD OF INSERT unten

AFTER: Der Triggercode wird erst ausgeführt nachdem das eigentliche SQL Kommando bereits abgeschlossen ist. Die Daten befinden sich im Arbeitsspeicher, sind aber noch nicht persistiert. Sprich ist ein Rollback immer noch möglich!

- **INSTEAD OF:** Falls keine Fehler auftreten, muss die Datenänderung im Trigger explizit ausgeführt werden!
- **AFTER:** Falls ein Fehler auftritt, muss ein expliziter ROLLBACK-Befehl lanciert werden.

Die Prozedur **sp_helptrigger** liefert nur die Namen der Trigger auf einer Tabelle. Eine Übersicht liefert folgende View:

```
CREATE VIEW v_trigger AS
SELECT name, definition FROM sys.triggers AS t
JOIN sys.sql_modules AS m
ON t.object_id=m.object_id;
GO
```

```
SELECT * FROM v_trigger;
```

--> Mit SELECT * FROM v_trigger können Sie alle Trigger und ihren Programmcode anzeigen.

Katalogprozeduren in SQL Server

Catalog Stored Procedures	
sp_column_privileges	sp_special_columns
sp_columns	sp_sproc_columns
sp_databases	sp_statistics
sp_fkeys	sp_stored_procedures
sp_pkeys	sp_table_privileges
sp_server_info	sp_tables

Tabelle 16: Katalog-Prozeduren

Katalogfunktionen in SQL Server

Metadaten-Funktionen	
@@PROCID	fn_listextendedproperty
COL_LENGTH	FULLTEXTCATALOGPROPERTY
COL_NAME	FULLTEXTSERVICEPROPERTY
COLUMNPROPERTY	INDEX_COL
DATABASEPROPERTY	INDEXKEY_PROPERTY
DATABASEPROPERTYEX	INDEXPROPERTY
DB_ID	OBJECT_ID
DB_NAME	OBJECT_NAME
FILE_ID	OBJECTPROPERTY
FILE_IDEX (Transact-SQL)	OBJECTPROPERTYEX
FILE_NAME	SQL_VARIANT_PROPERTY
FILEGROUP_ID	TYPE_ID
FILEGROUP_NAME	TYPE_NAME
FILEGROUPPROPERTY	TYPEPROPERTY
FILEPROPERTY	

Tabelle 17: Metadaten-Funktionen

16.5 Anwendung der wichtigsten Katalogsichten

Metadaten zum Server:

```
SELECT * FROM sys.servers -- Systemsicht in master

EXECUTE sys.sp_helpserver [server] -- gespeicherte Systemprozedur in master
```

Metadaten zu Datenbanken:

```
SELECT * FROM sys.databases -- Systemsicht in master
SELECT DB_ID(['db_name']) -- Funktion
SELECT DB_NAME([db_id]) -- Funktion

EXECUTE sys.sp_helpdb [db_name] -- gespeicherte Systemprozedur in master
```

Metadaten zu Tabellenobjekten:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES -- ISO konform
SELECT * FROM INFORMATION_SCHEMA.COLUMNS -- ISO konform
SELECT * FROM sys.objects -- Systemsicht in master: alle Objekte, nicht aufschlussreich
SELECT * FROM sys.tables -- Systemsicht in master: alle Tabellen
SELECT * FROM sys.columns -- Systemsicht in master: alle Kolonnen, nicht aufschlussreich

EXECUTE sys.sp_help tabellen_name -- gespeicherte Systemprozedur in master SUPER!!!
EXECUTE sys.sp_columns tabellen_name -- gespeicherte Systemprozedur in master
```

Metadaten zu Indexobjekten:

```
SELECT * FROM sys.indexes -- Übersicht
SELECT * FROM sys.index_columns -- gibt Auskunft, welche Spalten einbezogen sind

EXECUTE sys.sp_help tabellen_name
EXECUTE sys.sp_helpindex tabellen_name
```


Systemkatalog

1. Juli 2015 10:15

Metadaten zu Einschränkungen:

```
-- Objekte und Indizes
EXECUTE sys.sp_help tabellen_name -- als ersten Überblick
SELECT * FROM sys.objects -- ohne Domänen

SELECT OBJECT_ID( 'objekt' [, 'objekttyp' ] )
SELECT OBJECT_NAME ( object_id )

EXECUTE sys.sp_helpindex table_name
SELECT * FROM sys.indexes -- inkl. Primärschlüssel und UNIQUE

-- alle Constraints
EXECUTE sp_helpconstraint -- tabelle
SELECT * FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS

-- Typen
SELECT * FROM INFORMATION_SCHEMA.DOMAINS -- Domänen
SELECT * FROM sys.types
SELECT * FROM INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE

-- CHECK
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS
SELECT * FROM sys.check_constraints

-- DEFAULT
SELECT * FROM sys.default_constraints

-- Schlüssel
SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
SELECT * FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
SELECT * FROM sys.key_constraints
SELECT * FROM sys.foreign_keys

-- tabellenweise, ohne Defaults
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE
-- spaltenweise
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
```

Metadaten zu den Serverbenutzern:

```
SELECT * FROM sys.server_principals
SELECT * FROM sys.sql_logins

EXECUTE sys.sp_helplogins
```

Metadaten zu Datenbankbenutzern:

```
SELECT * FROM sys.database_principals
SELECT * FROM sys.database_role_members
```

Metadaten zu Rollen:

```
EXECUTE sys.sp_helprole --[rolle]
EXECUTE sys.sp_helprolename --[rolle]
```

Metadaten zu den Berechtigungen:

```
SELECT * FROM sys.database_permissions
```

Metadaten zu Views:

```
SELECT * FROM sys.views
EXECUTE sp_helptext view_name
```

Metadaten zu den Funktionen:

```
SELECT * FROM sys.objects
SELECT * FROM sys.sql_modules
SELECT * FROM sys.parameters
WHERE object_id = (SELECT object_id ('funktions_name'))
SELECT OBJECTPROPERTY (object_id, 'isScalarFunction')
SELECT OBJECT_DEFINITION [(SELECT object_id ('funktions_name'))]

EXEC sp_helptext 'funktions_name'
```

Metadaten zu den Prozeduren:

```
SELECT * FROM sys.procedures
SELECT * FROM sys.sql_modules
EXEC sp_depends 'prozedur_name'
SELECT * FROM sys.all_parameters
WHERE object_id = (SELECT object_id ('prozedur_name'))
SELECT OBJECT_DEFINITION [(SELECT object_id ('prozedur_name'))]

EXEC sp_helptext 'prozedur_name'
```

Metadaten zu den Triggern:

```
SELECT * FROM sys.triggers
EXEC sp_helptrigger 'tabellen_name'
SELECT * FROM sys.sql_modules
SELECT * FROM sys.trigger_events

EXEC sp_helptext 'trigger_name'
```