Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

wirtschafts informatik

Anlegen von Datenbanken im Microsoft SQL Server 2014

Modellbasiertes Forward Engineering und Migration von Access-Datenbanken

Manfred Sommer

Stand: Februar 2016

Inhaltsverzeichnis

1 Einleitung

In diesem Beitrag werden verschiedene Möglichkeiten beschrieben, wie eine SQL Server[1]-Datenbank angelegt und mit Daten gefüllt werden kann. Dabei kann man unterschiedliche Wege gehen:

- Im Mittelpunkt dieses Papers steht das modellbasierte Anlegen leerer Datenbanktabellen mit ihren sämtlichen Eigenschaften. Im Abschnitt 2.1 wird ausführlicher das Forward Engineering mit einem Datenmodellierungswerkzeug beschrieben. Anschließend wird im Abschnitt 2.2 kurz darauf eingegangen, wie eine Datenbank rein SQL-Skript-basiert angelegt werden kann. In beiden Fällen sollte die neu angelegte, noch datenlose Datenbank mit einem oder mehreren Datenbankdiagrammen dokumentiert werden (→ Abschnitt 2.3). Abschließend fügt man in die Tabellen Daten ein, wofür ebenfalls unterschiedliche Vorgehensweisen und Datenquellen in Frage kommen (→ Abschnitt 2.4).
- Liegt eine Datenbank bereits als Access-Datenbank vor, kann man sie mit einem Migrationstool in eine Microsoft SQL Server-Datenbank mit derselben Struktur und identischem Datenbestand transformieren (→ Abschnitt 3).
- Schließlich wird der spezielle Fall behandelt, dass eine Quelldatenbank auf demselben oder einem anderen SQL Server in identische Zieldatenbanken dupliziert werden soll (→ Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Die Darstellung bezieht sich auf den Microsoft SQL Server 2014, ist aber in großen Teilen auch auf frühere Versionen des Microsoft SQL Servers anwendbar und lässt sich konzeptionell auch auf SQL Server anderer Hersteller übertragen.

2 Anlegen einer neuen SQL Server Datenbank

In diesem Kapitel wird beschrieben, wie eine SQL Server-Datenbank mit Tabellen angelegt werden kann. Dabei wird zuerst und hauptsächlich der modellbasierte Weg mit einem Datenmodellierungswerkzeuge beschrieben (Abschnitt 2.1). Anschließend wird dargestellt, wie eine SQL Server-Datenbank ohne den „Umweg“ über ein Datenmodell direkt mit SQL-Befehlen erzeugt werden kann (Abschnitt 2.2). Abschließend wird erläutert, wie man im SQL Server ein Datenbankdiagramm für die zuvor modell- oder skriptbasiert angelegten Tabellen erstellt (Abschnitt 2.3) und wie diese Tabellen mit Daten befüllt werden können (Abschnitt 2.4).

Vom modellbasierten Generieren einer Datenbank profitieren die Kommunikation zwischen IT- und Domänenexperten sowie die Entwurfsqualität im Entwurfsprozess und die Dokumentation und Wartbarkeit der Datenbank in ihrem Lebenszyklus. In professionellen Datenbank-Projekten sollte es deshalb Standard sein. Bei diesem als **Forward Engineering** bezeichneten Vorgehen erzeugt man aus einem grafischen Datenmodell SQL-Code; genauer: die Datendefinitionsanweisungen für Tabellen und andere Datenbankobjekte wie z.B. Indizes. Daten werden beim Forward

[1] Mit der verkürzenden Schreibweise „SQL Server“ ist hier der „Microsoft SQL Server“ gemeint, sofern aus dem Kontext nichts anderes hervorgeht.

Engineering noch nicht in die Tabellen eingefügt, das Ergebnis ist also eine Datenbank mit leeren Tabellen. Der Fokus liegt somit auf der Struktur der Datenbank, nicht auf ihren Inhalten.

Häufig beginnt ein Datenbankprojekt mit der Erstellung eines Datenmodells. Das ist immer dann der Fall,

- wenn es sich um ein IT-Projekt „auf der grünen Wiese“ handelt, also um ein völlig neues datenbankgestütztes Informationssystem ohne Vorgängerdatenbank,
- wenn die zu erstellende Datenbank eine vorrelationale Datenhaltung ablösen soll, wie z.B. bei der Konsolidierung vieler Excel-Tabellen verschiedener Anwender („Wildwuchs der individuellen Datenverarbeitung“) oder
- wenn das abzulösende Informationssystem zwar bereits auf einer relationalen Datenbank basiert, deren Struktur aber dermaßen schlecht oder fachlich so überholt ist, dass sie von Grund auf neu entworfen werden muss.

Es kann aber auch vorkommen, dass die zu erstellende Datenbank ihre Struktur (und meistens auch ihre Daten) aus einer vorhandenen Desktop-Datenbank wie Microsoft Access oder aus einer SQL Server-Datenbank eines anderen Herstellers übernehmen soll. Auch eine solche Datenbankmigration wird von einem leistungsfähigen Datenmodellierungswerkzeug wirksam unterstützt. Hierbei wird zunächst ein logisches Datenmodell quasi „rückwärts“ aus der Quell-Datenbank erstellt – ein als **Reverse Engineering** bezeichnetes Vorgehen. Anschließend wird aus dem logischen Datenmodell nach einigen vor allem das physikalische Datenmodell tangierenden Anpassungen „vorwärts“ eine Microsoft SQL Server-Datenbank erzeugt. Reverse und Forward Engineering werden hier also kombiniert eingesetzt.

Forward und Reverse Engineering setzen voraus, dass das **Datenmodellierungswerkzeug**

- kein reines „Malwerkzeug“ für Entity-Relationship-Diagramme ist, sondern SQL-Code generieren kann (was z.B. auch für Microsoft Visio zutrifft), und
- über Treiber für die involvierten Datenbanksysteme verfügt (nativ oder über ODBC).

Relationale Datenbanken können nicht nur mit Datenmodellierungswerkzeugen, sondern auch mit **Objektmodellierungswerkzeugen** erstellt werden. Da moderne Softwaresysteme inzwischen zwar überwiegend objektorientiert entworfen und programmiert, ihre Daten aber nach wie vor meistens in (objekt-)relationalen Datenbanken gespeichert werden, können viele UML-Tools die „Welt der Klassen“ in eine „Welt der Tabellen“ transformieren. Man spricht dann von **objektrelationalem Mapping**. Auf das Forward Engineering mit Objektmodellierungswerkzeugen wird hier jedoch nicht näher eingegangen[2].

2.1 Anlegen mit einem Datenmodellierungswerkzeug

Das toolbasierte Anlegen einer relationalen Datenbank mit einem Datenmodellierungswerkzeugen wird hier am Beispiel von ERwin 9.6 dargestellt[3]. Ein großer Vorteil dieses Tools liegt in der

[2] In früheren Versionen dieses Papers wurden die Objektmodellierungswerkzeuge Rational Rose und Enterprise Architect behandelt.

[3] Detaillierte Informationen zu ERwin erhält man z.B. aus dessen über HELP | HELP TOPICS erreichbarer, sehr ausführliche Online-Hilfe.

breiten Palette der unterstützten SQL Server großer Hersteller[4], was vielfältige Migrationspfade eröffnet. Ferner erlaubt ERwin einen ausgefeilten Abgleich des Datenmodells mit der implementierten Datenbank (**Complete Compare**) sowie deren Synchronisation mit dem Datenmodell (Abschnitt 2.1.6). Auf diese Weise lässt sich verhindern, dass das Datenmodell einerseits und seine Implementierung auf dem SQL Server andererseits ein Eigenleben entwickeln und im Lebenszyklus einer Datenbank auseinanderdriften.

2.1.1 Logisches und physikalisches ERwin-Modell erzeugen

Beim Anlegen eines neuen ERwin-Modells über FILE | NEW... sollte man immer eine logische und eine physikalische Sicht anlegen (Abb. 1), weil das Forward Engineering (FE) nur aus der physikalischen Sicht heraus möglich ist. In dem gesonderten Bericht „Einführung in das Datenmodellierungswerkzeug ERwin 9.6“ wird beschrieben, wie man in der logischen Sicht Entitäten, Attribute, Primärschlüssel, Alternativschlüssel, Gültigkeitsregeln, Standardwerte etc. sowie die Beziehungen zwischen den Entitäten anlegt. ERwin erzeugt daraus korrespondierende physikalische Datenbankobjekte wie Tabellen, Spalten, Indizes usw.

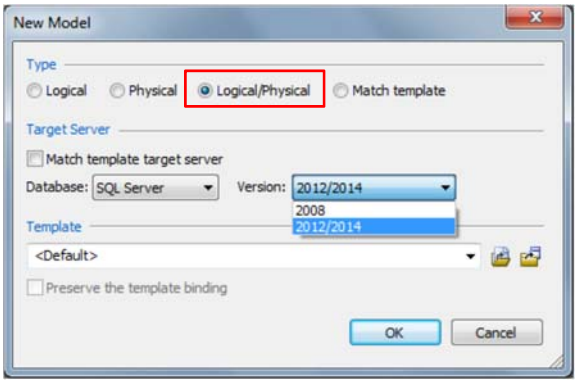


Abb. 1: Neues logisches und physikalisches Datenmodell erzeugen

In der hier als Beispielmodell verwendeten Datei *Nordwind_FE_SQL Server 2014.erwin* liegen das logische sowie das physikalische Modell bereits vor. Abb. 2 zeigt das physikalische Modell.

[4] Vgl. Abb. 3, aus der auch hervorgeht, dass die Desktop-Datenbank Microsoft Access von ERwin 9 im Gegensatz zu ERwin 7 nicht mehr unterstützt wird. Dafür werden jetzt die Versionen 2008 und 2012/2014 des Microsoft SQL Servers unterstützt, was in ERwin 7 noch nicht der Fall war.

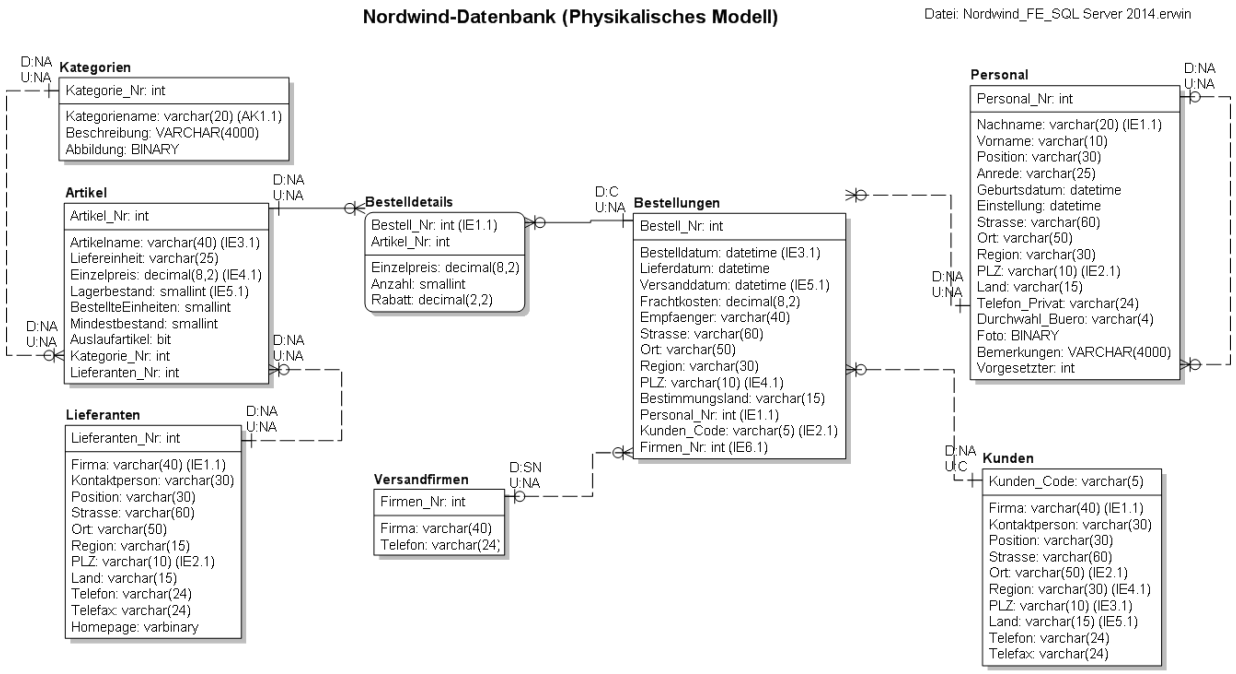


Abb. 2: Physikalisches Datenmodell der Nordwind-Datenbank für den SQL Server 2014

2.1.2 Ziel-Server wählen

Im physikalischen Modell (nicht im logischen!) kann man über ACTIONS | TARGET DATABASE ... den Zielserver-Typ überprüfen und gegebenenfalls auf den SQL Server 2014 ändern (Abb. 3). Für die im Target Server-Dialog aufgeführten Datenbanksysteme stehen sog. native Treiber zur Verfügung. Will man sich mit einem hier nicht aufgeführten relationalen Datenbanksystem verbinden, kann und muss man sich mit ODBC/Generic behelfen. Damit dürften alle relevanten Datenbanksysteme zugänglich sein. Seit ERwin 8 ist dies auch die einzige Möglichkeit, sich mit einer Access-Datenbank zu verbinden, da es keinen nativen Access-Treiber mehr gibt. Schließlich kann man in diesem Dialog auch den Standarddatentyp einstellen und festlegen, ob Nichtschlüsselattribute standardmäßig eingabepflichtig (NOT NULL) sein sollen oder nicht (NULL).

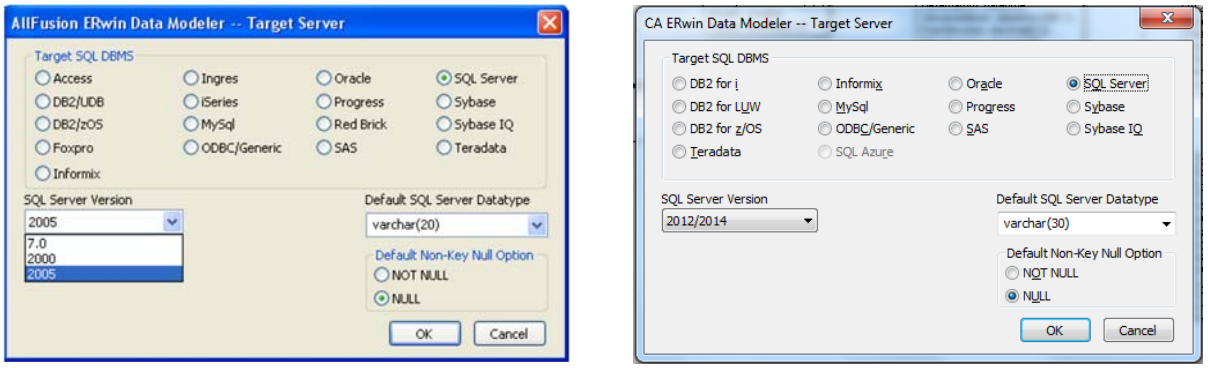


Abb. 3: Von ERwin 7.2 (links) und ERwin 9.6 (rechts) unterstützte Zieldatenbanksysteme

2.1.3 Ziel-Datenbank mit dem SQL Server Management Studio anlegen

Bevor man sich im nächsten Schritt mit der SQL Server-Datenbank verbinden kann, in der die Tabellen etc. angelegt werden sollen, muss diese auf dem SQL Server existieren. Das erledigt man am besten mit dem SQL Server Management Studio.

Auf einem Microsoft SQL Server können nur Systemadministratoren und solche Benutzer Datenbanken anlegen, die Mitglied der Serverrolle „dbcreator“ sind. Ein derart berechtigter Benutzer kann seine Zieldatenbank unterhalb des SQL Server-Knotens angelegen, auf dem sie liegen soll. Dies kann im grafischen Dialog des Objekt-Explorers über DATENBANKEN | NEUE DATENBANK... oder mit einem SQL-Befehl geschehen, z.B. für eine Datenbank mit dem Namen „Nordwind“:

```
CREATE DATABASE Nordwind
```

Mitglieder der dbcreator-Rolle können beliebig viele Datenbanken anlegen. Sie können Datenbanken aus dem Objekt-Explorer heraus nur dann löschen, wenn sie im Fenster „Objekt löschen“ das Kontrollkästchen „Sicherungs- und Wiederherstellungsverlaufsinformationen für Datenbanken löschen“ demarkieren. Mit dem SQL-Befehl

```
DROP DATABASE Nordwind
```

kann eine Datenbank ebenfalls gelöscht werden[5].

Für Benutzer, die nicht zu den Serverrollen „sysadmin“ oder „dbcreator“ gehören, muss man die Zieldatenbank für das Forward Engineering bereitstellen, z.B. für den Benutzer „BAA1234“ mit

```
-- Datenbank für Benutzer BAA1234 anlegen
CREATE DATABASE BAA1234_Nordwind
```

Damit dieser Benutzer in „seiner“ Datenbank Objekte erzeugen kann, muss ihm der Besitz dieser nicht von ihm selbst erstellten Datenbank übertragen werden[6].

```
ALTER AUTHORIZATION ON DATABASE::BAA1234_Nordwind TO [UNI-HAMBURG\BAA1234]
```

Jetzt kann der Benutzer “BAA1234” in seiner Datenbank „BAA1234_Nordwind“ Tabellen, Indizes, Constraints etc. anlegen.

2.1.4 Mit der SQL Server-Datenbank verbinden

Mit der Auswahl eines Zielservertyps alleine ist noch keine Verbindung zu einer bestimmten Datenbank auf einem bestimmten SQL Server herstellbar. Dies erfolgt im physikalischen Modell über ACTIONS | DATABASE CONNECTION... Bei Verwendung der zu empfehlenden Windows-Authentifizierung werden Benutzernamen und Passwort von der Windows-Domäne, an der man sich angemeldet hat, an den SQL Server durchgereicht. Die umständlichere Alternative der Database Authentication setzt voraus, dass der Benutzer auf diesem SQL Server mit SQL Server-Authentifizierung registriert ist.

---

[5] Nicht nur Mitglieder der sysadmin-Rolle, sondern auch Mitglieder der dbcreator-Rolle können beliebige Datenbanken löschen - nicht etwa nur von ihnen selbst erstellte Datenbanken, sondern auch die anderer Benutzer, und das, obwohl sie auf deren Datenbankobjekte keinen Zugriff haben!

[6] Gegebenenfalls muss dem Benutzernamen der Domänenname (hier „UNI-HAMBURG“) vorangestellt werden.

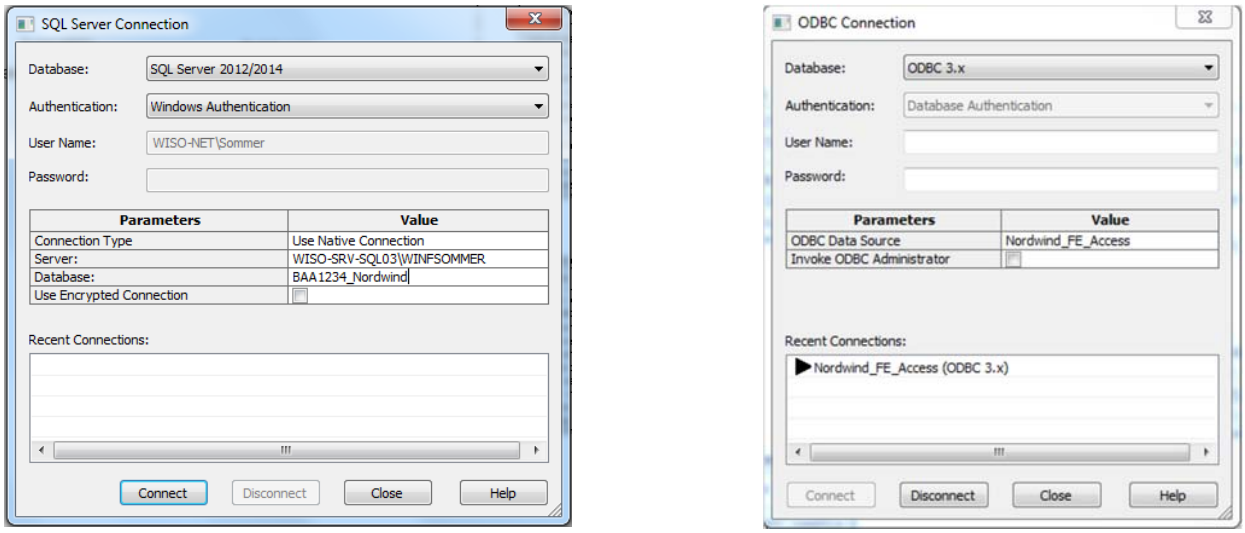


Abb. 4: Native Verbindung zum SQL Server (links) und ODBC-Verbindung zu einer Access-Datenbank (rechts)

Es stehen zwei Verbindungstypen zur Verfügung: die native Verbindung und die Verbindung über eine ODBC-Datenquelle. Eine native Verbindung ist vorzuziehen, weil sie im Zweifelsfall die Möglichkeiten des Zieldatenbanksystems vollständiger unterstützt als der „kleinste gemeinsame Nenner“ ODBC und weil man sich die Einrichtung einer ODBC-Datenquelle erspart. Bei der nativen Verbindung müssen der Servername und der Datenbankname von Hand eingetragen werden (Abb. 4 links). Das Listenfeld „Recent Connections“ ist beim ersten Aufruf leer. Dort werden später die zuletzt verwendeten Verbindungen angezeigt. Eine aktive Verbindung steht ganz oben und ist durch ein schwarzes Dreieck gekennzeichnet.

Mit einer Access-Datenbank kann man sich seit ERwin 8.2 nur noch über ODBC verbinden, da native Access-Connections nicht mehr unterstützt werden. Eine ODBC-Verbindung zu einer Access - Datenbank (Abb. 4 rechts) muss man zuvor mit dem 32-Bit-ODBC-Administrator eingerichtet haben, der mit der über den Windows-Startbutton auffindbaren Datei „odbcad32.exe“ geöffnet werden kann. Die so eingerichtete ODBC-Datenquelle kann man in ERwin aus einer Dropdownliste auswählen, in der die Namen der ODBC-Datenquellen angezeigt werden. Die ODBC-Datenquelle-Namen (hier „Nordwind_FE_Access“) müssen nicht mit den Datenbanknamen (hier die leere Datenbank „Nordwind.accdb“) übereinstimmen. Bei der Auswahl der ODBC-Datenquelle ist das Häkchen bei „Invoke ODBC Administrator“ zu setzen.

Das physikalische ODBC-Datenmodell muss sehr sorgfältig auf die Access-Tabelleneigenschaften abgestimmt werden, um beim Forward Engineering die Nacharbeiten am SQL-Skript in Grenzen zu halten. Nach ersten Tests scheint der Aufwand erheblich höher zu sein als bei den früheren nativen Access-Treibern. So bietet z.B. die Dropdownliste für die physikalischen Datentypen kein LONGTEXT für den Felddatentyp „Memo“ an, auch IMAGE für den Felddatentyp „OLE-Objekt“ sucht man vergeblich. Umgekehrt liefert das Reverse Engineering einer Access 2010-Datenbank via ODBC 3.0 ein recht korrekturbedürftiges physikalisches Datenmodell.

Auch für eine SQL Server-Datenbank kann man eine ODBC-Verbindung erstellen und sich anschließend in ERwin über diese statt über den nativen Treiber mit der Datenbank verbinden. Im ODBC-Administrator wählt man zuerst den Server aus, mit dem man sich verbinden möchte

(Abb. 5 links). Wenn die Dropdownliste leer ist, muss man den Namen des Servers oder der Server-Instanz (hier „WISO-SRV-SQL03\WINFSOMMER“) in das Kombinationsfeld händisch eintragen. In einem weiteren Schritt wählt man die Datenbank auf diesem SQL Server aus (Abb. 5 rechts). Die weiteren Schritte sind selbsterklärend.

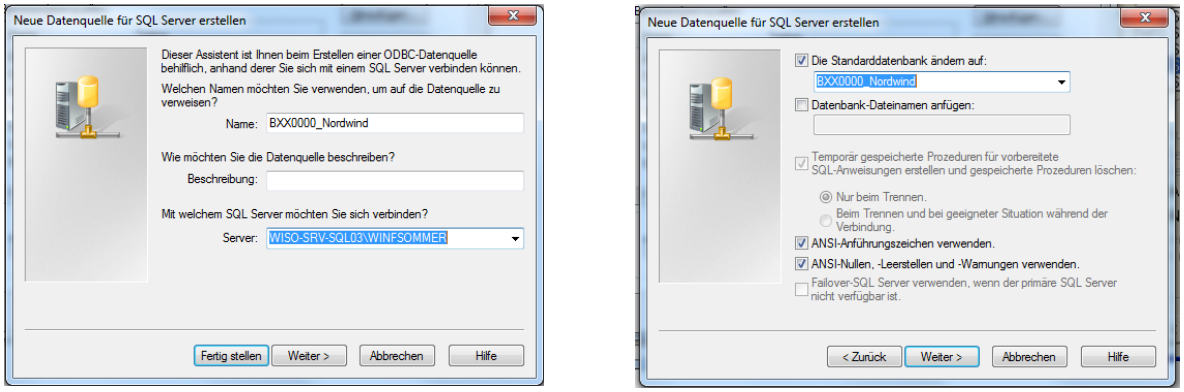
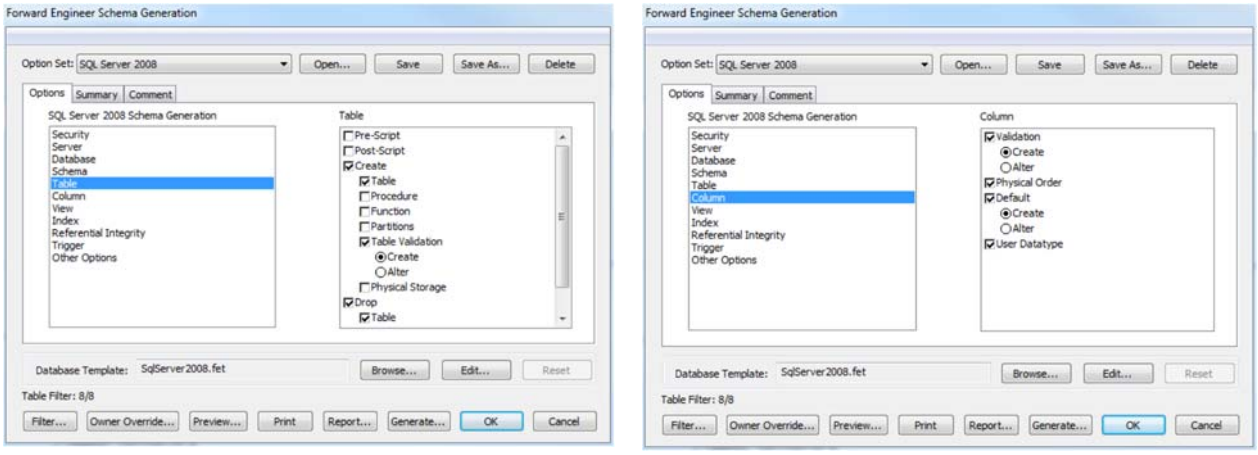
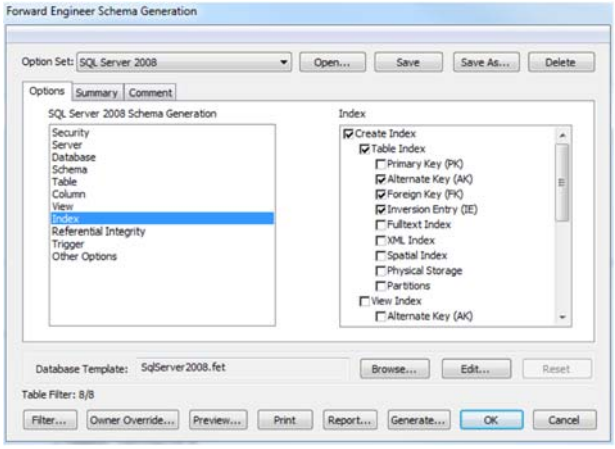


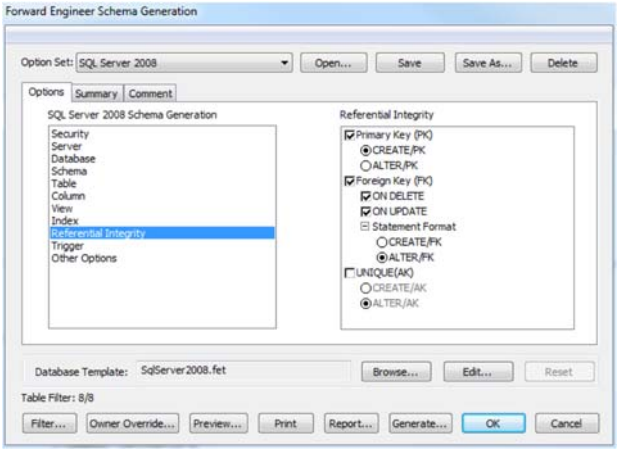
Abb. 5: ODBC-Datenquelle einrichten

2.1.5 Tabellen mit Forward Engineering anlegen

Nach der erfolgreichen Anmeldung an die Datenbank kann über ACTIONS | FORWARD ENGINEER | SCHEMA... das Forward Engineering gestartet werden. Die im Folgenden gewählten Einstellungen sollte man unter einem speziellen Option Set wie hier „SQL Server 2014“ speichern. In der Datei *Nordwind_FE_SQL Server 2014.erwin* wurden folgende Einstellungen gewählt: Bei Security, Server, Database, Schema, View und Trigger sind sämtliche Optionen demarkiert. Die übrigen Optionen gehen aus Abb. 6 hervor.







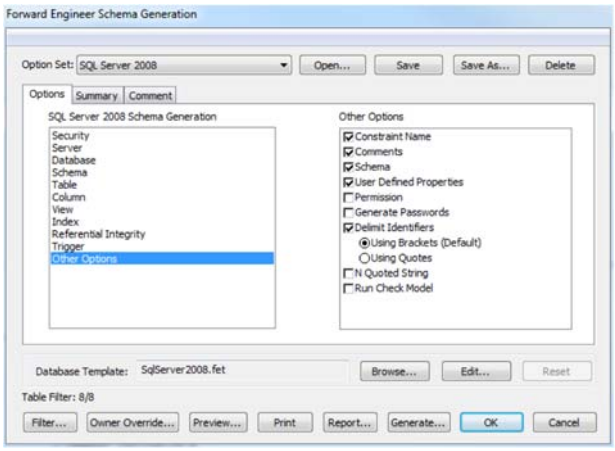


Abb. 6: Geeignete Forward Engineering-Optionen für den Microsoft SQL Server 2012

Bevor man die Datenbankobjekte mit GENERATE... direkt in der Zieldatenbank anlegt, sollte man sich mit PREVIEW... das dahinter stehende SQL-Skript ansehen. Wenn dieses aus ERwin mit GENERATE... angestoßene Skript nicht völlig fehlerfrei ist, bricht die Generierung bei dem Fehler verursachenden Datenbankobjekt ab. Nach Auftreten eines Fehlers müssen alle bereits angelegten Datenbankobjekte wieder entfernt werden, bevor man die ERwin-Generierung erneut starten kann, weil bereits vorhandene Datenbankobjekte nicht unter demselben Namen erneut angelegt werden können. Das Löschen dieser Datenbankobjekte kann man entweder im SQL Server Management Studio mit dem Objekt-Explorer „zu Fuß“ oder mit einem SQL-Skript aus DROP-Befehlen erledigen oder dadurch, dass man im Forward Engineer Schema Generation-Fenster von ERwin die Schema-Optionen „Drop“ markiert.

Statt die Tabellen, Indizes etc. mit GENERATE... direkt anzulegen, kann man die DDL-Befehle[7] auch mit REPORT... in eine SQL-Skriptdatei schreiben und diese in einem Abfragefenster des SQL Server Management Studios testen und ausführen, bis sich die Datenbank komplett fehlerfrei anlegen

[7] DDL ist die Abkürzung für Data Definition Language und bezeichnet die Untermenge der SQL-Befehle, mit denen Datenbankobjekte im Systemkatalog (nicht Datensätze) angelegt werden. Hierzu gehören vor allem die CREATE-Befehle (`CREATE TABLE` und `CREATE INDEX`, aber auch `CREATE RULE`, `CREATE TRIGGER` etc.) sowie die analogen ALTER-Befehle zur Änderung von Datenbankobjekten.

lässt[8]. Man sollte die Korrekturen, die man im SQL Server Management Studios an den SQL-Befehlen vorgenommen hat, aber unbedingt im ERwin-Datenmodell nachpflegen.

2.1.6 Synchronisation mit Complete Compare

Selbst wenn eine Datenbank sorgfältig entworfen wurde, unterliegt sie im Laufe ihres Lebenszyklus Veränderungen. Datenbank und Datenmodell passen dann nicht mehr exakt zu einander, was unbedingt vermieden werden sollte, und sei es nur, um eine korrekte Dokumentation beizubehalten. Wenn z.B. beim Importieren von Daten in die Lieferantentabelle auffällt, dass Lieferanten mit einem Ortsnamen von mehr als 15 Zeichen wegen des Datentyps `VARCHAR(15)` nicht in die Datenbank aufgenommen werden können, könnte man die Zeichenkettelänge zunächst im Modell von 15 auf 50 erhöhen. Modell und Datenbank weichen diesbezüglich jetzt voneinander ab, was man durch einen vollständigen Vergleich (Complete Compare) zwischen beiden aufdecken kann. Über ACTIONS | COMPLETE COMPARE... wählt man links ein ERwin-Datenmodell (Abb. 7) und rechts eine Datenbank (Abb. 8) aus[9].

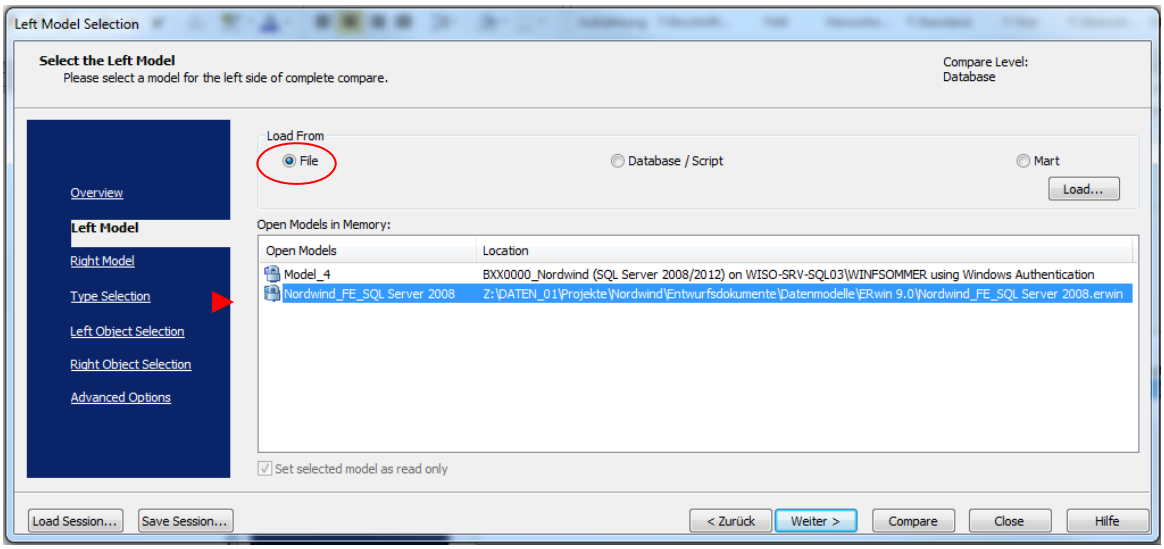


Abb. 7: Auswahl eines Datenmodells für den Vollständigen Vergleich

[8] Näheres hierzu ist Abschnitt 2.2 zu entnehmen.

[9] Genau genommen wird nicht die Datenbank selbst, sondern das aus ihr reverse-engineerte Datenmodell (hier als „Model_4“ bezeichnet) in den Vergleich einbezogen

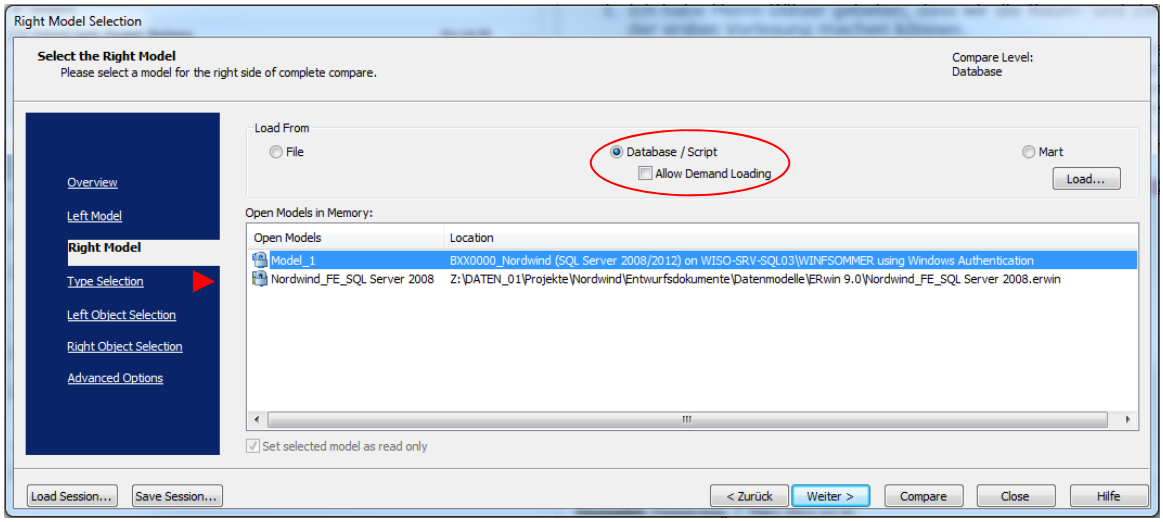


Abb. 8: Auswahl einer Datenbank für den Vollständigen Vergleich

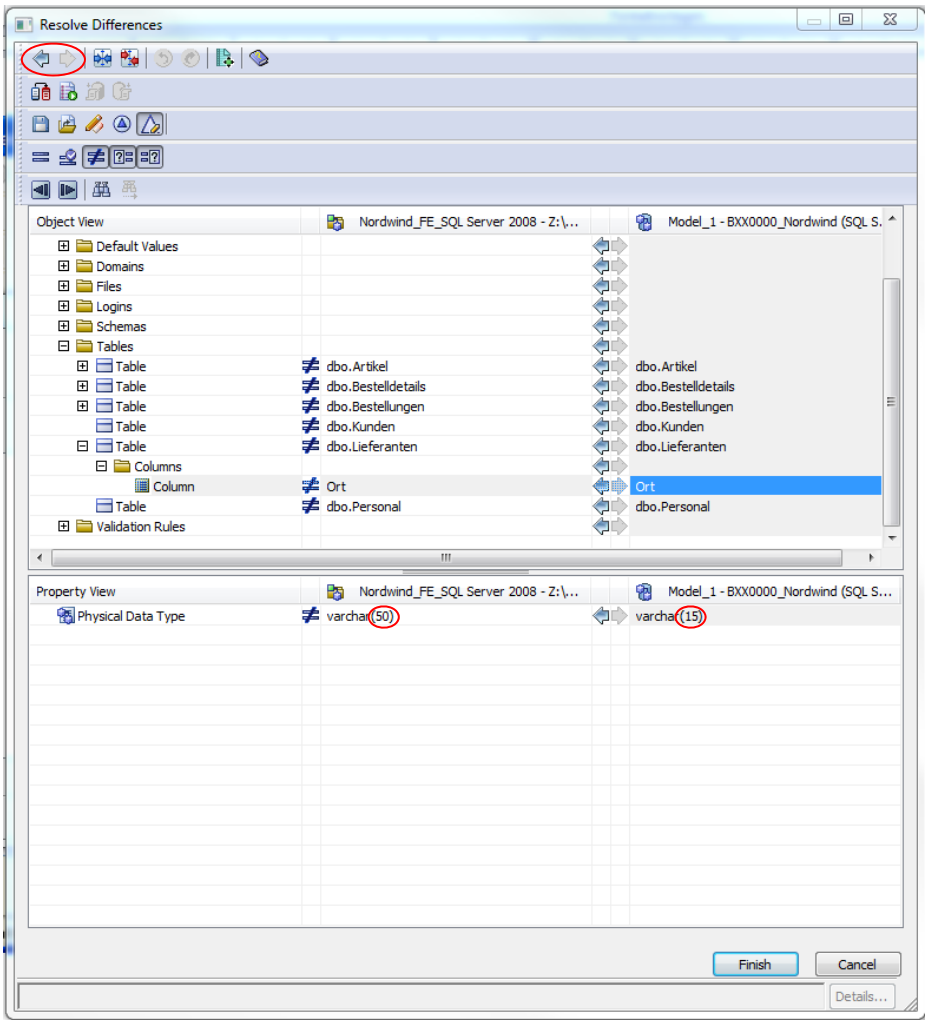


Abb. 9: Synchronisation von Datenmodell und SQL Server-Datenbank

Im Resolve Differences-Fenster (Abb. 9) kann man sich sämtliche am ≠-Symbol erkennbaren Divergenzen zwischen dem Datenmodell (im Screenshot in der Mitte) und der SQL Server-Datenbank (im Screenshot rechts) ansehen. Durch Markieren der differenten Eigenschaft und Klick auf die Taste ⇦ oder ⇨ passt man die Datenbank dem Modell oder umgekehrt an. Mit der Schaltfläche „Right Alter Script“ bzw. „Left Alter Script“ wird ein Skript mit ALTER-Befehlen erzeugt, das sofort ausgeführt werden kann. Dadurch werden das Datenmodell und die Datenbank wieder in Übereinstimmung gebracht.

2.2 Anlegen mit einem SQL-DDL-Skript

Die bereits mehrfach erwähnten SQL-DDL-Skripte bestehen im Wesentlichen aus

- CREATE TABLE-Befehlen zum Anlegen der Tabellen und ihrer Constraints (Primärschlüssel, Alternativschlüssel, Werteeinschränkungen und Defaultwerte) und
- CREATE INDEX-Befehlen zum Anlegen nicht-eindeutiger Suchschlüssel.

Den CREATE-Befehlen können bedingte, in IF EXISTS eingebettete DROP-Befehle vorausgehen, wenn man sicher gehen will, dass gleichnamige Tabellen zuvor gelöscht werden. Am Ende eines solchen Skripts können entsprechende ALTER-Befehle stehen, wenn man z.B. die referentielle Integrität erst nach dem Anlegen von Tabellen ergänzen will.

Das SQL-Skript für das Nordwind-Datenmodell sollte etwa wie folgt in aussehen (vgl. Datei „Nordwind_Tabellen anlegen.sql“). Besonders wichtige SQL-Schlüsselworte sind beim ersten Auftreten in diesem Skript gelb hervorgehoben.

```
USE BXX0000_Nordwind

CREATE TABLE Kategorien (
   Kategorie_Nr int IDENTITY,
   Kategoriename varchar(20) NOT NULL,
   Beschreibung varchar(MAX) NULL,
   Abbildung varbinary(MAX) NULL,
      CONSTRAINT PK_Kategorien PRIMARY KEY CLUSTERED (Kategorie_Nr ASC),
      CONSTRAINT AK1_Kategoriename UNIQUE (Kategoriename ASC))

CREATE TABLE Lieferanten (
   Lieferanten_Nr int IDENTITY,
   Firma varchar(40) NOT NULL,
   Kontaktperson varchar(30) NULL,
   Position varchar(30) NULL,
   Strasse varchar(60) NULL,
   Ort varchar(15) NULL,
   Region varchar(15) NULL,
   PLZ varchar(10) NULL,
   Land varchar(15) NULL,
   Telefon varchar(24) NULL,
   Telefax varchar(24) NULL,
   Homepage nvarchar(MAX) NULL,
      CONSTRAINT PK_Lieferanten PRIMARY KEY CLUSTERED (Lieferanten_Nr ASC))

CREATE INDEX IE1_Firma ON Lieferanten (Firma ASC)
CREATE INDEX IE2_PLZ ON Lieferanten (PLZ ASC)

CREATE TABLE Artikel (
   Artikel_Nr int IDENTITY,
   Artikelname varchar(40) NOT NULL,
   Liefereinheit varchar(25) NULL,
   Einzelpreis decimal(8,2) NULL
      CONSTRAINT DF1_Einzelpreis
         DEFAULT 0
      CONSTRAINT CK1_Einzelpreis
         CHECK ([Einzelpreis] >= 0),
```

```
    Lagerbestand smallint NULL
       CONSTRAINT DF_Lagerbestand
          DEFAULT 0
       CONSTRAINT CK_Lagerbestand
          CHECK ([Lagerbestand] >= 0),
    BestellteEinheiten  smallint NULL
       CONSTRAINT DF_BestellteEinheiten
          DEFAULT 0
       CONSTRAINT CK_BestellteEinheiten
          CHECK ([BestellteEinheiten] >= 0),
    Mindestbestand  smallint NULL
       CONSTRAINT DF_Mindestbestand
          DEFAULT 0
       CONSTRAINT CK_Mindestbestand
          CHECK ([Mindestbestand] >= 0),
    Auslaufartikel bit NOT NULL
       CONSTRAINT DF_Auslaufartike
          DEFAULT 0,
    Kategorie_Nr int NOT NULL,
    Lieferanten_Nr int NOT NULL,
       CONSTRAINT PK_Artikel
          PRIMARY KEY CLUSTERED (Artikel_Nr ASC),
       CONSTRAINT FK_Artikel_Kategorien
          FOREIGN KEY (Kategorie_Nr) REFERENCES Kategorien  (Kategorie_Nr),
       CONSTRAINT FK_Artikel_Lieferanten
          FOREIGN KEY (Lieferanten_Nr) REFERENCES Lieferanten  (Lieferanten_Nr))

CREATE TABLE Kunden (
    Kunden_Code varchar(5) NOT NULL,
    Firma varchar(40) NOT NULL,
    Kontaktperson varchar(30) NULL,
    Position varchar(30) NULL,
    Strasse varchar(60) NULL,
    Ort varchar(50) NULL,
    Region varchar(30) NULL,
    PLZ varchar(10) NULL,
    Land varchar(15) NULL,
    Telefon varchar(24) NULL,
    Telefax varchar(24) NULL,
       CONSTRAINT PK_Kunden PRIMARY KEY CLUSTERED (Kunden_Code ASC))

CREATE INDEX IE_KundeName ON Kunden (Firma ASC)
CREATE INDEX IE_KundeOrt ON Kunden (Ort ASC)
CREATE INDEX IE_KundePLZ ON Kunden (PLZ ASC)
CREATE INDEX IE_KundeRegion ON Kunden (Region ASC)
CREATE INDEX IE_KundeLand ON Kunden (Land ASC)

CREATE TABLE Personal (
    Personal_Nr int NOT NULL,
    Nachname varchar(20) NOT NULL,
    Vorname varchar(10) NOT NULL,
    Position varchar(30) NULL,
    Anrede varchar(25) NULL,
    Geburtsdatum smalldatetime NULL
       CONSTRAINT CK_Geburtsdatum
          CHECK ([Geburtsdatum] < getdate()),
    Einstellung smalldatetime NULL,
    Strasse varchar(60) NULL,
    Ort varchar(50) NULL,
    Region varchar(30) NULL,
    PLZ varchar(10) NULL,
    Land varchar(15) NULL,
    Telefon_Privat varchar(24) NULL,
    Durchwahl_Buero varchar(4) NULL,
    Foto varbinary(MAX) NULL,
    Bemerkungen varchar(MAX) NULL,
    Vorgesetzter int NULL,
       CONSTRAINT PK_Personal
          PRIMARY KEY CLUSTERED (Personal_Nr ASC),
       CONSTRAINT FK_Personal_Vorgesetzter
          FOREIGN KEY (Vorgesetzter) REFERENCES Personal (Personal_Nr))

CREATE INDEX IE_PersonalName ON Personal (Nachname ASC)
CREATE INDEX IE_PersonalPLZ ON Personal (PLZ ASC)
```

```
CREATE TABLE Versandfirmen (
   Firmen_Nr int IDENTITY,
   Firma varchar(40) NOT NULL,
   Telefon varchar(24) NULL,
      CONSTRAINT PK_FirmenID
         PRIMARY KEY CLUSTERED (Firmen_Nr ASC))

CREATE INDEX IE_Kategorie_Nr ON Artikel (Kategorie_Nr ASC)
CREATE INDEX IE_Lieferanten_Nr ON Artikel (Lieferanten_Nr ASC)
CREATE INDEX IE_ArtikelName ON Artikel (Artikelname ASC)
CREATE INDEX IE_Einzelpreis ON Artikel (Einzelpreis ASC)
CREATE INDEX IE_Lagerbestand ON Artikel (Lagerbestand ASC)

CREATE TABLE Bestellungen (
   Bestell_Nr int IDENTITY(10248,1),
   Kunden_Code varchar(5) NOT NULL,
   Personal_Nr int NOT NULL,
   Bestelldatum smalldatetime NULL,
   Lieferdatum smalldatetime NULL,
   Versanddatum smalldatetime NULL,
   Frachtkosten decimal(8,2) NULL,
   Empfaenger varchar(40) NULL,
   Strasse varchar(60) NULL,
   Ort varchar(50) NULL,
   Region varchar(30) NULL,
   PLZ varchar(10) NULL,
   Bestimmungsland varchar(15) NULL,
   Firmen_Nr int NULL,
   CONSTRAINT PK_Bestellungen
      PRIMARY KEY CLUSTERED (Bestell_Nr ASC),
   CONSTRAINT FK_Bestellungen_Kunden
      FOREIGN KEY (Kunden_Code) REFERENCES Kunden (Kunden_Code)
         ON UPDATE CASCADE,
   CONSTRAINT FK_Bestellungen_Personal
      FOREIGN KEY (Personal_Nr) REFERENCES Personal (Personal_Nr),
   CONSTRAINT FK_Bestellungen_Versandfirmen
      FOREIGN KEY (Firmen_Nr) REFERENCES Versandfirmen (Firmen_Nr)
         ON DELETE SET NULL,
   CONSTRAINT CK_Lieferdatum
      CHECK ([Lieferdatum]>=[Bestelldatum]),
   CONSTRAINT CK_Versanddatum
      CHECK ([Versanddatum]>=[Bestelldatum]))

CREATE INDEX IE_Personal_Nr ON Bestellungen (Personal_Nr ASC)
CREATE INDEX IE_Kunden_Code ON Bestellungen (Kunden_Code ASC)
CREATE INDEX IE_Bestelldatum ON Bestellungen (Bestelldatum ASC)
CREATE INDEX IE_PLZ ON Bestellungen (PLZ ASC)
CREATE INDEX IE_Versanddatum ON Bestellungen (Versanddatum ASC)
CREATE INDEX IE_Firmen_Nr ON Bestellungen (Firmen_Nr ASC)

CREATE TABLE Bestelldetails (
   Bestell_Nr int NOT NULL,
   Artikel_Nr int NOT NULL,
   Einzelpreis decimal(8,2) NOT NULL
      CONSTRAINT DF2_Einzelpreis
         DEFAULT 0
      CONSTRAINT CK2_Einzelpreis
         CHECK ([Einzelpreis] >= 0),
   Anzahl smallint NOT NULL
      CONSTRAINT DF_Anzahl
         DEFAULT 1
      CONSTRAINT CK_Anzahl
         CHECK ([Anzahl] > 0),
   Rabatt decimal(2,2) NOT NULL
      CONSTRAINT DF_Rabatt
         DEFAULT 0
      CONSTRAINT CK_Rabatt
         CHECK ([Rabatt] >= 0 and [Rabatt] <= 1),
       CONSTRAINT PK_Bestelldetails
   PRIMARY KEY CLUSTERED (Bestell_Nr ASC, Artikel_Nr ASC),
      CONSTRAINT FK_Bestelldetails_Artikel
         FOREIGN KEY (Artikel_Nr) REFERENCES Artikel  (Artikel_Nr),
      CONSTRAINT FK_Bestelldetails_Bestellungen
```

```
          FOREIGN KEY (Bestell_Nr) REFERENCES Bestellungen  (Bestell_Nr)
             ON DELETE CASCADE)

CREATE INDEX IE_Bestell_Nr ON Bestelldetails (Bestell_Nr ASC)
CREATE INDEX IE_Artikel_Nr ON Bestelldetails (Artikel_Nr ASC)
```

Wenn man über kein forward engineering-fähiges Datenmodell verfügt und auch der in Abschnitt 3 beschriebene Weg des Upsizens einer Access-Datenbank nicht beschritten werden kann, bleibt einem das Schreiben eines solchen SQL-Skripts nicht erspart, sofern man die Tabellen nicht im grafischen Dialog des SQL Server Management Studios „zu Fuß“ anlegen will.

2.3 Datenbankdiagramm erzeugen

Bevor im nächsten Schritt (Abschnitt 2.4) die Tabellen mit Daten gefüllt werden, sollte man sich darüber Gedanken machen, in welcher Reihenfolge die Tabellen zu füllen sind. Wegen der referentiellen Integritätsbeziehungen zwischen den Tabellen ist diese nämlich nicht beliebig. So kann man z.B. die Artikel-Tabelle nicht vor der Kategorien-Tabelle mit Daten füllen, weil die Artikel die Kategorien referenzieren[10]. Man kann die referentielle Integrität allerdings, wie andere Einschränkungen auch, vor dem Einfügen von Datensätzen mit dem Befehl

```
ALTER TABLE Artikel NOCHECK CONSTRAINT ALL
```

außer Kraft setzen. Nach dem Einfügen der Datensätze sollten die Einschränkungen wieder aktiviert werden:

```
ALTER TABLE Artikel CHECK CONSTRAINT ALL
```

Im SQL Server Management Studio kann man unter *Datenbankname* | DATENBANKDIAGRAMME | NEUES DATENBANKDIAGRAMM ein solches erzeugen.

Es gibt fünf Tabellen, die von keiner anderen Tabelle abhängig sind und die deshalb zuerst mit Daten gefüllt werden:
Gruppe 1:
Kunden, Personal, Versandfirmen, Kategorien, Lieferanten

Danach werden die beiden Tabellen gefüllt, die von den Tabellen der Gruppe 1 abhängen:
Gruppe 2:
Artikel (von Kategorie), Bestellungen (von Kunden, Personal und Versandfirmen)

Schließlich wird die Tabelle gefüllt, die von Tabellen der Gruppe 2 abhängt.
Gruppe 3:
Bestelldetails (von Artikel und Bestellungen)

---

[10] Man kann die referentielle Integrität allerdings, wie andere Einschränkungen auch, vor dem Einfügen von Datensätzen mit dem Befehl

```
ALTER TABLE Artikel NOCHECK CONSTRAINT ALL
```

außer Kraft setzen. Nach dem Einfügen der Datensätze sollten die Einschränkungen wieder aktiviert werden:

```
ALTER TABLE Artikel CHECK CONSTRAINT ALL
```

Neu

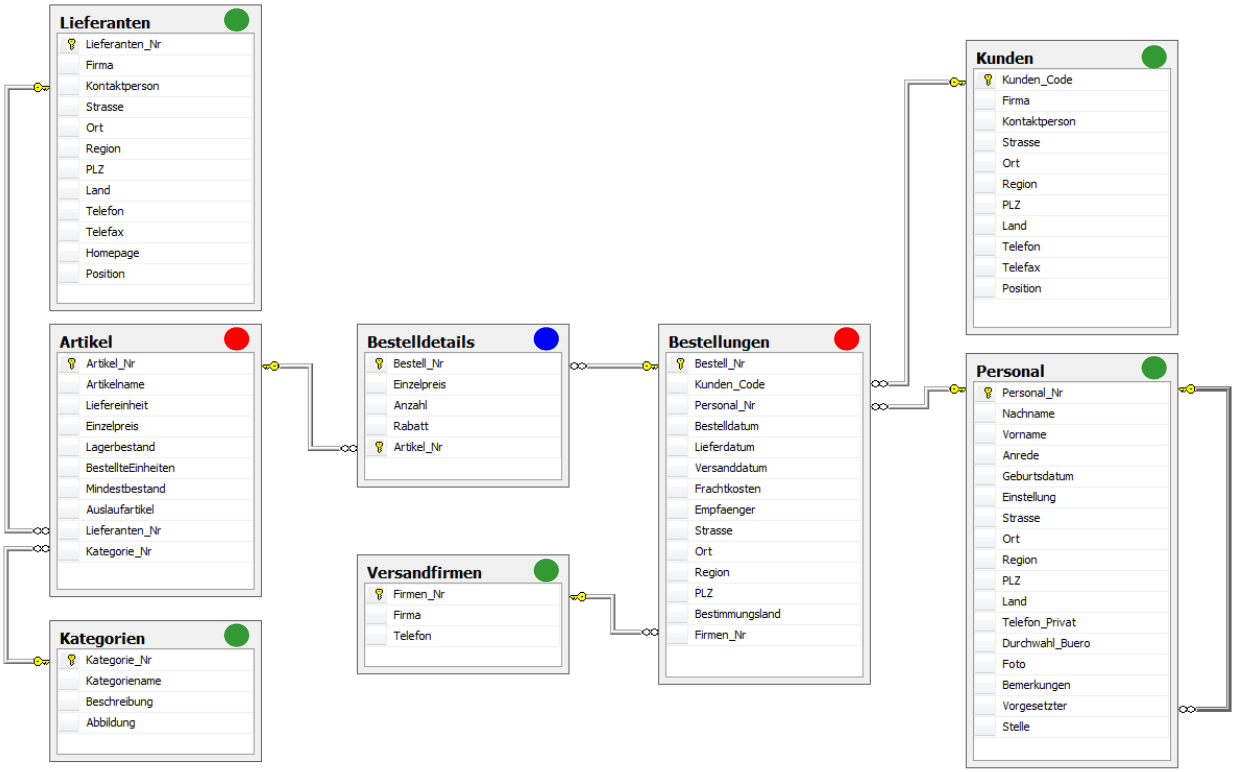


Abb. 10: SQL Server-Datenbankdiagramm für die Nordwind-Datenbank

2.4 Daten in SQL Server-Tabellen einfügen

Jetzt können die Daten in der genannten Reihenfolge in die Tabellen eingefügt werden, was auf verschiedene Art und Weise erfolgen kann:

- Händische Eingabe Datensatz für Datensatz
- Datenimport-Assistent z.B. aus Excel-Tabellen und ASCII-Dateien
- Datensatzorientiertes Einfügen mit einem INSERT-VALUES-Skript
- Mengenorientiertes Einfügen aus einer anderen SQL Server-Datenbank mit einem INSERT-SELECT-Skript
- Datenexport aus einer Access-Datenbank (.accdb) mit Anfügeabfrage an verknüpfte SQL-Server Tabellen (hier nicht näher erläutert).

2.4.1 Händische Dateneingabe

Man kann jede Tabelle einer Datenbank im SQL Server Management Studio über *Tabellenname* | OBERSTE 200 ZEILEN BEARBEITEN in der Benutzeransicht öffnen und die Datensätze einzeln eingeben. Mit *Tabellenname* | OBERSTE 1000 ZEILEN AUSWÄHLEN wird eine Tabelle ebenfalls in der Benutzeransicht geöffnet, allerdings nur mit lesendem Zugriff. Man hier also keine Datensätze eingeben und natürlich auch nicht ändern oder löschen.

Tabelle - dbo.Kategorien | YBPD058665.No...e einfügen.sql | Zusammenfassung

Kategorie_Nr	Kategoriename	Beschreibung	Abbildung
1	Beverages	Soft drinks, coffees, teas, beers, and ales	<Binärdaten>
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	<Binärdaten>
3	Confections	Desserts, candies, and sweet breads	<Binärdaten>
4	Dairy Products	Cheeses	<Binärdaten>
5	Grains/Cereals	Breads, crackers, pasta, and cereal	<Binärdaten>
6	Meat/Poultry	Prepared meats	<Binärdaten>
7	Produce	Dried fruit and bean curd	<Binärdaten>
8	Seafood	Seaweed and fish	<Binärdaten>
NULL	NULL	NULL	NULL

Abb. 11: Händische Dateneingabe im SQL Server Management Studio

Dieser Weg ist bei größeren Datenmengen sehr aufwändig. Zudem können Binärdaten in diesem Fenster nicht eingegeben werden. Wenn die Daten in maschinenlesbarer Form vorliegen, wird man deshalb einen der folgenden Wege beschreiten.

2.4.2 Mengenorientierte Einfügen mit dem SQL Server-Import/Export-Assistenten

Hinsichtlich der Datenquelle sehr flexibel ist man mit dem SQL Server-Import/Export-Assistenten. Über *Datenbankname* | TASKS | DATEN IMPORTIEREN... wird der SQL Server-Import/Export-Assistent gestartet. Als Datenquelle wird hier die Excel-Tabelle „Bestelldetails.xls“ gewählt. Das setzt voraus, dass alle übrigen Tabellen gefüllt sind, weil es sonst zu Verletzungen der referentiellen Integrität kommt. Danach wird die Zieldatenbank gewählt. Diese erscheint automatisch, wenn man den Import-Assistenten aus der Zieldatenbank heraus startet. Auch die folgenden Schritte sind weitgehend selbsterklärend. Im Fenster „Spaltenzuordnung“ muss geprüft werden, ob die Zuordnung der Quell- und Zielspalten korrekt ist. Ferner ist zu entscheiden, ob die in der Zieltabelle vorhandenen Datensätze zuvor gelöscht werden sollen.

Nach jedem Datenimportversuch sollte man sich durch Öffnen der Zieltabelle oder mit dem Befehl

```
SELECT * FROM tabellenname
```

vom Erfolg oder Fehlschlag des Imports überzeugen. Beim Datenimport aus der Excel-Tabelle Personal stellt man z.B. fest, dass diese Anfügeabfrage dreimal nacheinander ausgeführt werden muss, bis sämtliche Datensätze eingefügt sind. Grund dafür ist die rekursive Beziehung auf die Personaltabelle selbst, die dazu führt, dass nur Mitarbeiter eingegeben werden können, deren Vorgesetzter bereits in der Tabelle vorhanden ist.

Die Bestellpositionen könnten auch aus der ASCII-Tabelle „Bestelldetails.txt“ importiert werden. Diese Datei enthält Tabstopps als Spaltentrennzeichen. Der Import gestaltet sich aber deutlich schwieriger und fehleranfälliger als aus der entsprechenden Excel-Tabelle.

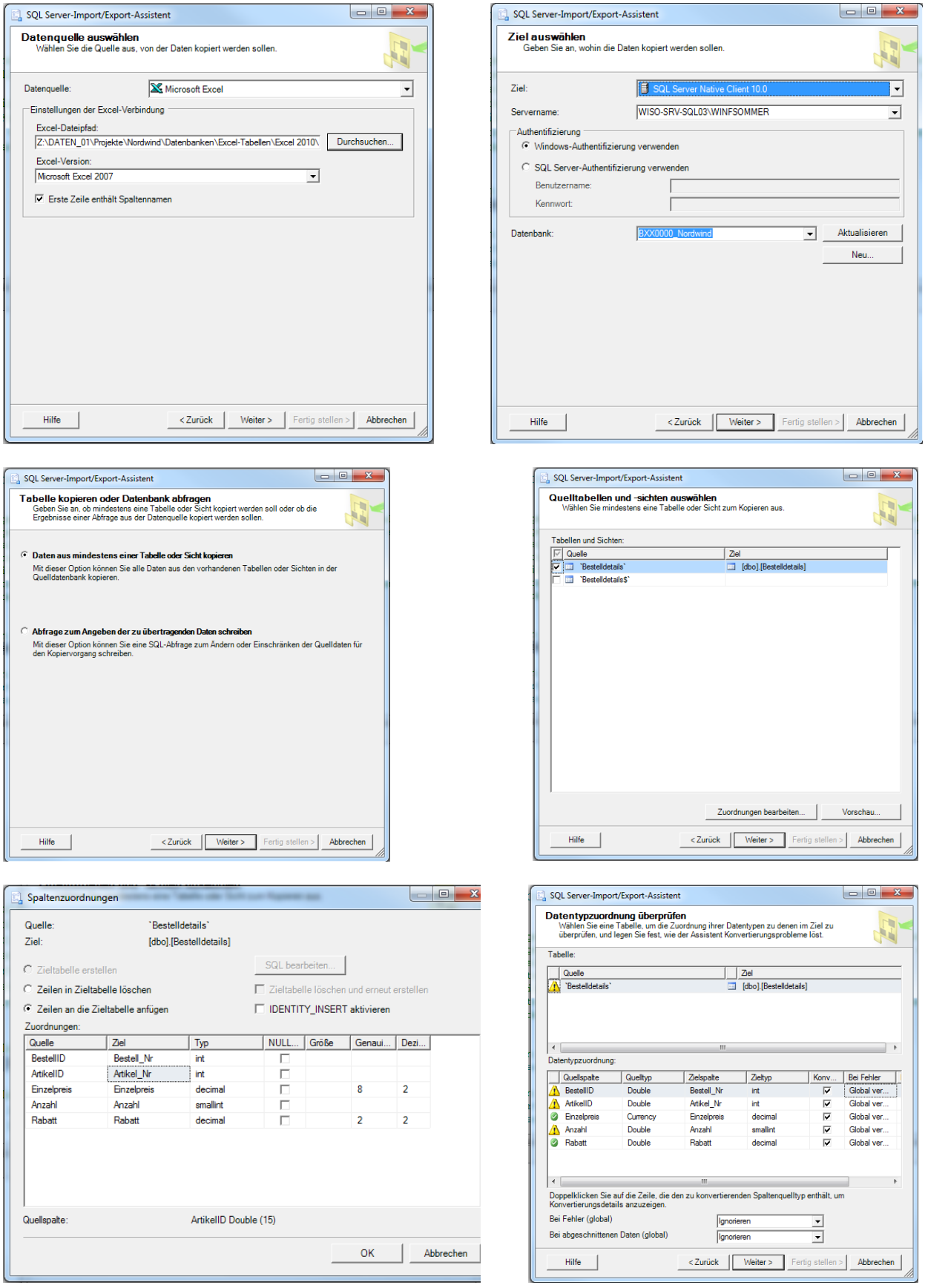


Abb. 12: Datenimport aus einer Excel-Tabelle mit dem SQL Server-Import/Export-Assistenten

Mit den DELETE-Befehlen aus der Datei „Nordwind_Tabelleninhalte löschen.sql“ können bei Bedarf sämtliche Datensätze einer oder mehrerer Tabellen wieder gelöscht werden.

2.4.3 Datensatzorientiertes Einfügen mit einem INSERT-VALUES-Skript

Am einfachsten, weil von der Verfügbarkeit einer konfigurierten Datenquelle unabhängig, ist die massenhafte Dateneingabe mit INSERT-Befehlen nach folgendem Muster:

```
INSERT INTO Bestelldetails VALUES(10248,11,14,12,0)
INSERT INTO Bestelldetails VALUES(10248,42,9.8,10,0)
INSERT INTO Bestelldetails VALUES(10248,72,34.8,5,0)
INSERT INTO Bestelldetails VALUES(10249,14,18.6,9,0)
INSERT INTO Bestelldetails VALUES(10249,51,42.4,40,0)
...
```

Dieses Verfahren eignet sich gut für die Dateneingabe in eine Demodatenbank, in der man mit einem solchen Skript nach dem Löschen aller Datensätze stets dieselben Referenzdaten wiederherstellten will. Auch Binärdaten wie die Bilder in der Kategorientabelle lassen sich auf diese Weise hexadezimal einfügen.

```
INSERT Kategorien(Kategorie_Nr,Kategoriename,Beschreibung,Abbildung) VALU-
ES(1,'Beverages','Soft drinks, coffees, teas, beers, and a-
les',0x151C2F00020000000D000E00...

INSERT Kategorien(Kategorie_Nr,Kategoriename,Beschreibung,Abbildung) VALU-
ES(2,'Condiments','Sweet and savory sauces, relishes, spreads, and seasonin-
gs',0x151C2F00020000000D000E0...

...
```

Nachteilig ist der Aufwand, der mit der Erstellung eines solchen SQL-Skripts verbunden ist.

Die Datei „Nordwind_Datensätze einfügen.sql“ enthält sämtliche INSERT-Befehle, mit denen die Referenzdaten der Nordwind-Datenbank (teilweise mit englischen Bezeichnungen wie Sales Representative statt Vertriebsmitarbeiter) erstellt werden können.

2.4.4 Mengenorientiertes Einfügen mit einem INSERT-SELECT-Skript

Liegen die Daten in einer anderen Datenbank auf demselben SQL Server, kann man sie von dort in einem Schritt abfragen und an eine strukturkompatible Tabelle der aktuellen Datenbank anfügen:

```
INSERT INTO Kunden SELECT * FROM Nordwind.dbo.Kunden
INSERT INTO Personal SELECT * FROM Nordwind.dbo.Personal
...
```

Sämtliche notwendigen INSERT-Befehle enthält die Datei „Nordwind_Daten importieren.sql“. Wenn sich die Tabellen in der Quell- und der Ziel-Datenbank hinsichtlich der Spaltennamen oder der Spaltenreihenfolge unterscheiden oder wenn nur Daten ausgewählter Spalten übernommen werden sollen, dann müssen die Spaltennamen der Zieldatenbank vor dem Schlüsselwort SELECT in Klammern und die Spaltennamen der Quelldatenbank hinter dem Schlüsselwort SELECT statt des Platzhalterzeichen (*) verwendet werden.

3 Migration einer Access-Datendatenbank zum SQL Server

Das Upsizing einer Access-Datenbank zu einer SQL Server-Datenbank kann in drei Varianten erfolgen und bietet einige Vorteile[11]:

- Beim **Upsizing zu einem Access-Projekt mit echter Client-Server-Architektur** (vgl. Abschnitt 3.1) werden außer den Tabellen und den darin gespeicherten Daten auch die meisten Abfragen[12] auf den SQL Server portiert. Zudem bleiben die Benutzerschnitte (Formulare und Berichte) und die clientseitige Applikationslogik (VBA-Programme) erhalten und müssen nicht vollständig neu entwickelt werden.
- Beim **Upsizing einer Access-Datenbank mit eingebunden SQL Server-Tabellen** (vgl. Abschnitt 3.2) werden nur die Tabellen samt Daten, jedoch keine Abfragen portiert. Auch hier werden die Formulare und Berichte sowie ein Großteil des VBA-Codes weiterverwendet. Der Zugriff auf die Daten erfolgt über das Einbinden der SQL Server-Tabellen.
- Beim **Upsizing ohne Anwendungsänderungen** (vgl. Abschnitt 3.3) werden die Tabellen samt Daten in eine SQL Server-Datenbank kopiert. Da die Access-Datenbank aber völlig unverändert bleibt und deshalb auch nicht mit dem SQL Server kommuniziert, ist sie weder als echter Client noch als einfaches Frontend einsetzbar. Dieses Upsizing macht nur dann Sinn, wenn der Client mit Windows Forms oder Web Forms neu entwickelt werden und damit Access komplett abgelöst werden soll.

Alle drei Upsizing-Varianten bieten den Vorteil, dass sie im Gegensatz zum Reverse/Forward Engineering nicht nur die Tabellenstruktur aus einer Access-Datenbank in eine SQL Server-Datenbank übertragen, sondern auch die Datenübernahme gleich mit erledigen können.

Illustriert wird die Funktionsweise des Upsizing-Assistenten hier mit der Access-Datenbank „Nordwind.accdb“.

3.1 Upsizing zu einem Access-Projekt mit echter Client-Server-Architektur

Das Upsizing einer Access-Datenbank, erkennbar an der Dateiendung MDB (Microsoft Database), zu einem Access-Projekt, erkennbar an der Dateiendung ADP (Access Data Project) ist die anspruchsvollste Variante.

Folgende Fehlerquellen sollte man sich bewusst machen und möglichst vor dem Upsizing in der Access-Quelldatenbank ausräumen.

3.1.1 Grenzen und Fehlerquellen

Zu den bekannten, auch in Access 2007 noch vorhandenen Schwächen des automatisierten Upsizing zu einer Client-Server-Anwendung zählen SQL-Dialekt-bedingte Unterschiede zwischen Ac-

[11] Das Upsizen hat sich in Access 2007 gegenüber Access 2003 nicht verändert. Access-Projekte werden auch unter Access 2007 im Access 2002/2003-Dateiformat gespeichert. Ein Konvertieren in ein Access 2007-Dateiformat gibt es hier nicht.

[12] Zu nicht zu Views und überhaupt nicht transformierten Abfragen vgl. Abschnitt 3.1.1.1.

cess und dem SQL Server sowie zwischen dem inzwischen recht veralteten Datenobjektmodell DAO (Data Access Objects) und dem neueren ADO (ActiveX Data Objects). Weitere Informationen sind den „Informationen zum Upsizing einer Microsoft Access-Datenbank“ (Access-Hilfe: Access-Startseite > Access > Hilfe und Anleitungen zu Access 2003 > Access-Projekte > Upsizing einer Access-Datenbank) zu entnehmen.

3.1.1.1. Diskrepanzen zwischen Access SQL und Transact-SQL

Einige Unterschiede zwischen Access SQL und Transact-SQL werden vom Upsizing-Assistenten zwar erkannt, aber nicht behoben.

Access-Abfragen mit der Eigenschaft „Eindeutige Datensätze“ = Ja werden nicht in Views der SQL Server-Datenbank transformiert. Der Upsizing-Assistent dokumentiert dies mit dem Hinweis, dass diese Abfrage die Klausel **DISTINCTROW** enthält. Diesen Fehler kann man dadurch vermeiden, dass man statt der Eigenschaft „Eindeutige Datensätze“ die Abfrageeigenschaft „Keine Duplikate“ auf Ja setzt. Im SQL-Fenster sieht man, dass DISTINCTROW dadurch zu DISTINCT wird.

Da der SQL Server in Views keine **ORDER BY**-Klausel zulässt, werden Access-Abfragen mit Sortierkriterium statt zu Views in Funktionen transformiert. Das ist soweit unproblematisch. Abfragen mit einem Sortierkriterium, die ihrerseits als Unterabfragen weiterverwendet werden, werfen jedoch Probleme auf, da eine FUNCTION in einem SELECT-Statement nicht angesprochen werden kann. Deshalb sollte man vor dem Upsizing Sortierkriterien aus gespeicherten Unterabfragen entfernen. In der Nordwind.accdb trifft dies auf die Abfrage „Bestelldetails erweitert“ zu, die als Unterabfrage in der Abfrage „Umsätze nach Kategorie“ auftritt.

Auch **Parameterabfragen** wie „Personalumsätze nach Land“ und „Rechnungsfilter“ in Nordwind.accdb werden zu Funktionen auf dem SQL Server. Die Parameterabfrage „Umsätze nach Jahr“ wird allerdings nicht migriert.

Eine Abfrage mit einer **UNION**-Klausel wie „Kunden und Lieferanten nach Standort“ wird weder zu einer View noch zu einer Funktion, sondern zu einer gespeicherten Prozedur.

Andere Abfragen wie z.B. **Kreuztabellenabfragen** („Quartalsbestellungen nach Artikeln“) beherrscht nur die Jet Engine, der SQL Server jedoch nicht. Der Upsizing-Bericht weist darauf hin.

Dieser Problemkreis betrifft nur Access-Projekte. Bei der File-Server-Architektur (Abschnitt 3.2) werden keine einzige Abfrage in die SQL Server-Datenbank migriert. Alle Abfragen werden dort weiterhin von der hinter dem Access-Client stehenden Jet Engine lokal ausgeführt.

3.1.1.2 Keine Transformation von DAO zu ADO

Der Upsizing-Assistent lässt den VBA-Code in Modulen und Ereignisprozeduren unverändert. Deshalb muss Code, der DAO-Objekte verwendet, entweder vor oder nach dem Upsizing manuell auf ADO umgeschrieben werden. Das eventuelle spätere Upsizing ist also ein weiterer Grund dafür, beim Entwickeln einer Access-Datenbank von vornherein ADO-Objekte für den Datenzugriff einzusetzen und auf das ältere DAO-Objektmodell zu verzichten.

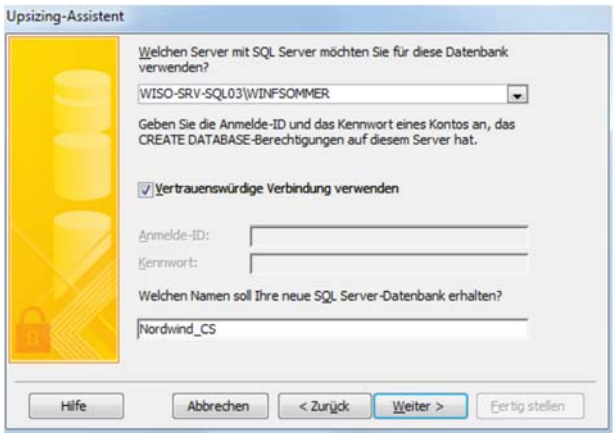
Insbesondere dürfen keine DAO-Methoden in Prozeduren enthalten sein, die von einem AutoExec-Makro aufgerufen werden, oder die als Ereignisprozeduren eines Startformulars automatisch beim Öffnen des neuen Access-Projekts ausgeführt würden, da sich das Programm sonst an dieser Stelle aufhängt. Vor dem Upsizing der Nordwind-Datenbank muss deshalb über ACCESS-OPTIONEN | AKTUELLE DATENBANK ANWENDUNGSOPTIONEN | FORMULAR ANZEIGEN von „Start“ auf „(keines)“ umgestellt werden. Andernfalls hängt sich die Anwendung bei der CurrentDB-Methode auf.

3.1.2 Der Upsizing-Vorgang

Im Folgenden werden die einzelnen Upsizing-Schritte und die dabei zu treffenden Entscheidungen beschrieben.

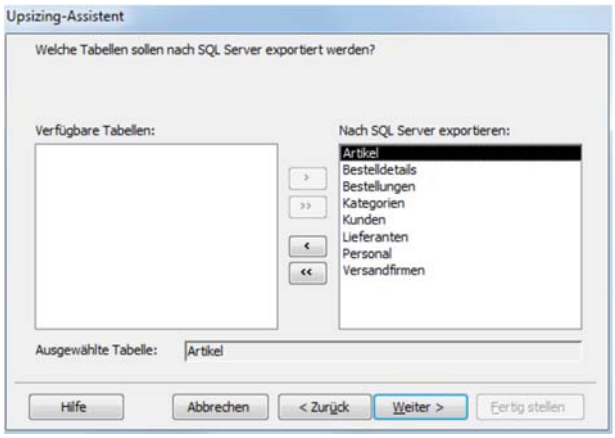
❶ Über DATENBANKTOOLS | DATEN VERSCHIEBEN | SQL SERVER wird der Upsizing-Assistent gestartet.

Man erstellt eine neue SQL Server-Datenbank oder verwendet eine bereits vorhandene. Diese sollte dann aber völlig leer sein.



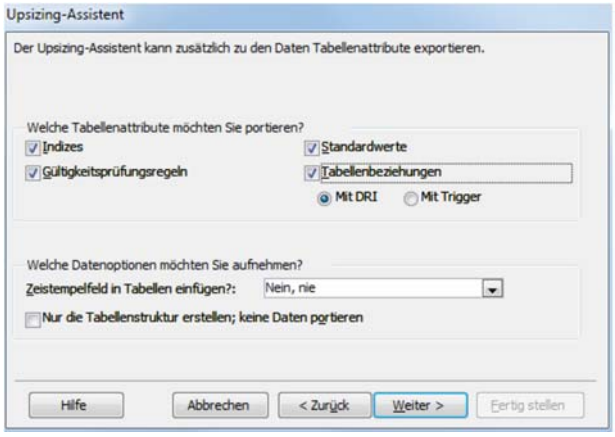
❷ Man wählt den SQL Server aus, auf dem die Datenbank mit den zu übertragenden Tabellen etc. anlegt werden soll, hier die SQL Server-Instanz „WISO-SRV-SQL03\WINFSOMMER“.

Dafür braucht man auf dem SQL Server dbcreator-Rechte. (Wenn im ersten Schritt entschieden wurde, eine vorhandene SQL Server-Datenbank zu verwenden, dann ist die SQL Server-Rolle „dbcreator“ nicht erforderlich. Es reichen dann db_ddladmin-Rechte in dieser Datenbank.)



❸ In der Regel wird man sämtliche Tabellen exportieren.

Da auch eingebundene Tabellen in den SQL-Server transformiert werden, funktioniert das Upsizing auch für Access-Datenbanken problemlos, die in eine Backend-MDB und eine Frontend-MDB aufgeteilt wurden.



❹ Man wird normalerweise so viele Tabelleneigenschaften wie möglich in die SQL Server-Datenbank übernehmen und bei den Tabellenbeziehungen die referentielle Integrität deklarativ (DRI) statt mit Triggern durchsetzen.

Zeitstempelfelder müssen nicht hinzugefügt werden.

Außer der Tabellenstruktur sollten auch die Daten mit übernommen werden.

❺ An dieser Stelle entscheidet sich, ob a) eine echte Client-Server-Lösung erstellt wird (in diesem Abschnitt beschrieben) b) der SQL Server ähnlich wie bei einer File-Server-Lösung lediglich als Datenspeicher genutzt wird, dessen Tabellen in die bestehende Access-MDB-Anwendung eingebunden werden (vgl. Abschnitt 3.2) oder c) ob die Access-Tabellen lediglich auf den SQL Server kopiert werden und die Access-MDB völlig unverändert bleibt (vgl. Abschnitt 3.3).

❻ Nach der Entscheidung für das Öffnen der neuen ADP-Datei wird der eigentliche Upsizingvorgang auf dem SQL Server ausgeführt.

Abschließend erzeugt der Assistent einen Bericht, der das Upsizing der Tabellen und Abfragen dokumentiert. Diesen Bericht sollte man als PDF-Datei speichern und auf Besonderheiten und Probleme (vgl. Abschnitt 3.1.1.1) durchsehen und diese, wenn möglich, beheben. Der Upsizing-Bericht gibt keine Hinweise auf Probleme, die bei Formularen, Berichten, Makros und Modulen auftreten können. Hier bleibt einem das Austesten all dieser Datenbankobjekte nicht erspart.

Im neuen Access-Projekt werden nur noch die Access-Client-Objekte (Formulare, Berichte, Makros, Module) gespeichert[13], während die Tabellen und „Abfragen“ jetzt in der SQL Server-Datenbank liegen (Abb. 11 links). Die ebenfalls auf dem SQL Server liegenden Datenbankdiagramme werden seit Access 2007 im Access-Projekt nicht mehr angezeigt.

Im SQL Server Management Studio sieht man, dass die Mehrzahl der Access-Abfragen zu Sichten (Views) geworden sind, andere zu gespeicherten Prozeduren oder Funktionen (Abb. 11 rechts).

Natürlich werden auch keine VBA-Prozeduren in gespeicherte Prozeduren mit DML-Anweisungen oder mit prozeduralem Transact-SQL-Code portiert. Schließlich werden auch keine Trigger aus Datenereignisprozeduren erzeugt.

Abschließend sollte man sämtliche in der SQL Serverdatenbank erzeugten Sichten testweise öffnen und eventuell auftretende Fehler korrigieren. In der Nordwind-Datenbank müssen in Views wie z.B. „Quartalsbestellungen“ Datumskonstanten vom US-Format ('1/1/1997' bzw. '12/31/1997') in das deutsche Format ('1.1.1997' bzw. '31.12.1997') geändert werden. Es können auch Konvertierungsfehler wie in den Views „Umsätze nach Artikeln für 1997“, „Umsätze nach Kategorie für 1997“, „Umsätze nach Kategorie“ und „Umsatzsumme nach Anzahl“ auftreten. Eine ganze Reihe von Abfragen kann aus unterschiedlichen Gründen, die dem Upsizing-Bericht zu entnehmen sind, überhaupt nicht konvertiert werden. Sie müssen neu erstellt werden.

[13] Es kann vorkommen, dass Formulare und Berichte nicht automatisch in das neue Access-Projekt übernommen werden. Sie lassen sich aber problemlos aus der Access-Datenbank (ACCDB-Datei) importieren, die als Basis für das Upsizing fungiert hat.

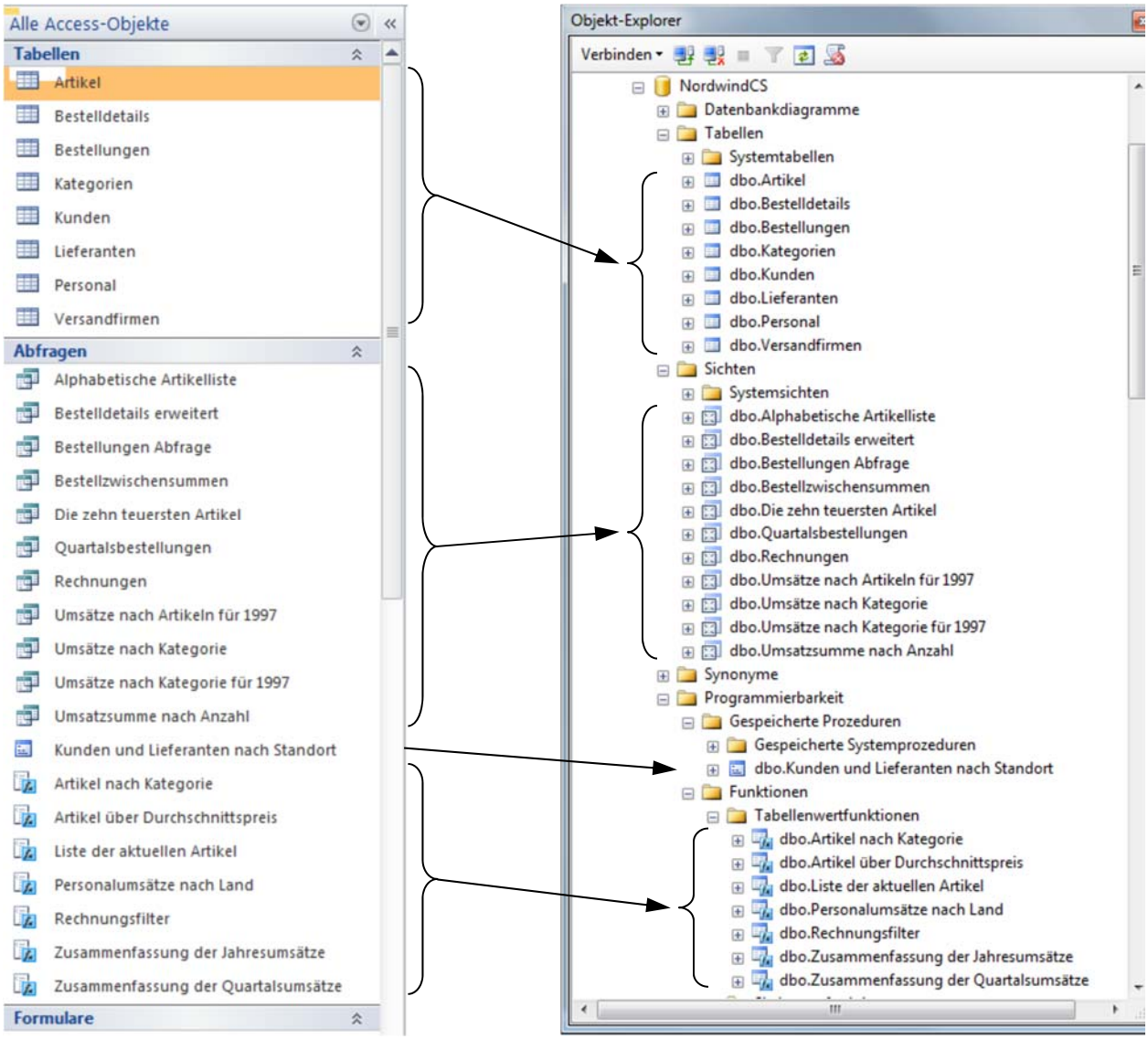
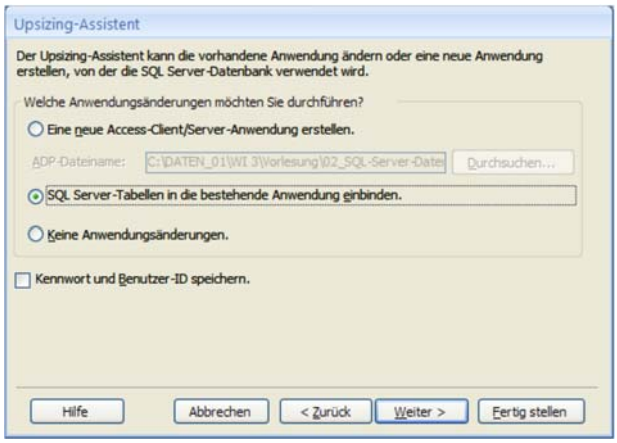


Abb. 13: Upsizing zu einer Client-Server-Architektur

3.2 Upsizing zu einer Access-Datenbank mit eingebundenen SQL Server-Tabellen

Im Folgenden werden nur die Upsizing-Schritte beschrieben, die sich vom Upsizing zu einer Client-Server-Architektur unterscheiden. Die Original-Datenbank „Nordwind.accdb“ sollte als Kopie unter Namen „Nordwind_FS.accdb“ gespeichert werden, da sie im Upsizing-Prozess verändert wird. Die SQL Server-Datenbank erhält hier den Namen „Nordwind_FS“.



❺ An dieser Stelle wird jetzt die Option gewählt, lediglich die Access-Tabellen in eine SQL Server-Datenbank zu migrieren und diese Tabellen in die bestehende ACCDB-Anwendung einzubinden. Der Assistent erzeugt dann wieder einen Bericht, der den Upsizing-Prozess der Tabellen dokumentiert. Da Probleme beim Upsizing von Access-Abfragen hier nicht auftreten können, ist der Bericht kürzer und nicht so bedeutsam.

Auch hier lohnt sich die Gegenüberstellung der Access-Objekte und der SQL Server-Datenbankobjekte. Die Access-Tabellen bleiben bestehen, werden jedoch umbenannt in „*tabellenname*_lokal“ und von den Access-Frontend-Objekten nicht mehr genutzt (Abb. 13 links). Diese Tabellen sollte man deshalb bei der weiteren Datenpflege ignorieren. Da sie zum Upsizing-Zeitpunkt quasi „eingefroren“ werden, ist ihr Aussagewert nur historischer Natur. Man kann sie auch löschen.

Die auf den SQL Server migrierten Access-Tabellen erkennt man in der Access-Datenbank durch das die Einbindung symbolisierende Icon (Abb. 13 links), im SQL Server Management Studio stehen sie wieder im Tabellen-Baum (Abb. 13 rechts). Sichten, gespeicherte Prozeduren, Funktionen wurden in der SQL Server-Datenbank nicht angelegt. Sämtliche Access-Frontend-Objekte (Abfragen, Formulare, Berichte, Seiten, Makros und Module) greifen auf die eingebundenen SQL-Server-Tabellen zu.

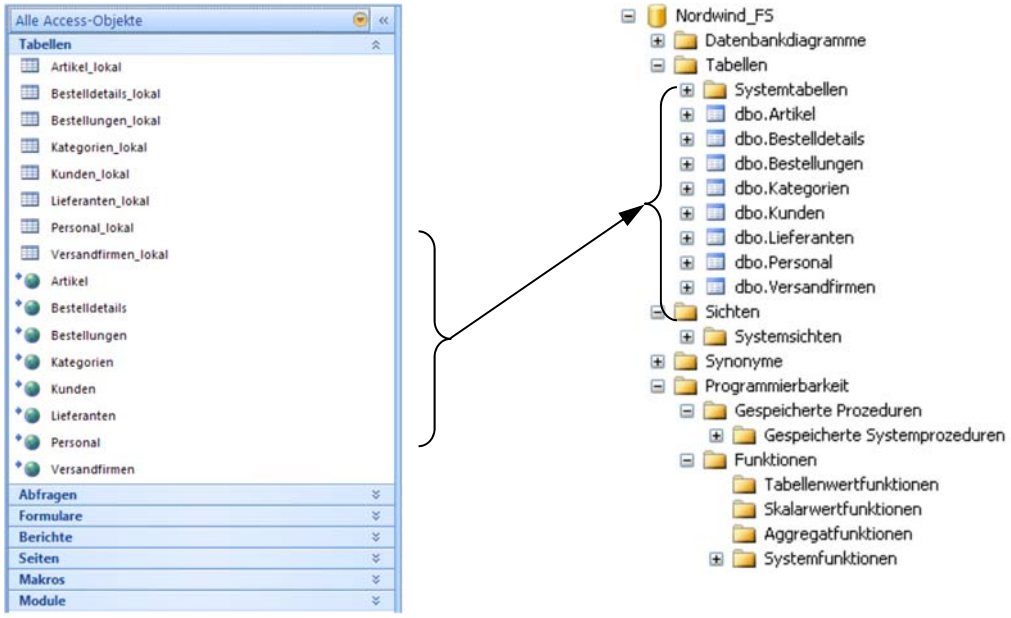
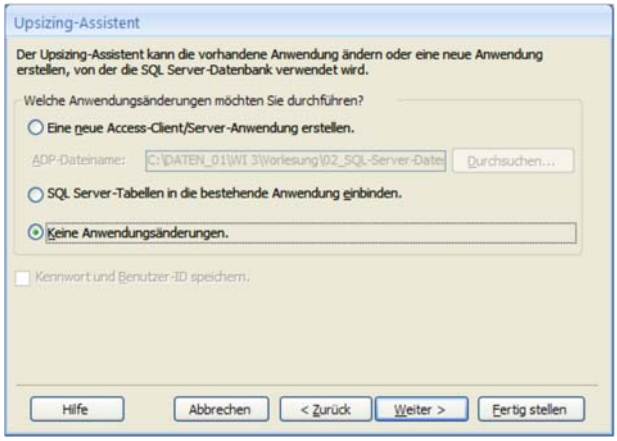


Abb. 14: Upsizing zu einer File-Server-Architektur

3.3 Upsizing zu einer SQL Server-Datenbank ohne Access-Anbindung

Der entscheidende Unterschied zu den beiden vorherigen Upsizing-Strategien ergibt sich bei der Frage nach den intendierten Anwendungsänderungen. Im zweiten Upzising-Schritt nennen wir die SQL Server-Datenbank „Nordwind_NEU“.



❺ An dieser Stelle wird jetzt die Option gewählt, die bestehende ACCDB-Anwendung völlig unverändert zu lassen. Die Access-Tabellen werden wie im Abschnitt 3.2 in eine SQL Server-Datenbank transformiert, aber anschließend nicht in die Access-Quell-Datenbank eingebunden. Der Assistent erzeugt wieder einen Bericht, der den Upsizing-Prozess der Tabellen dokumentiert.

Keine Anwendungsänderungen bedeutet, dass nur die Tabellen mit Daten auf den Server kopiert werden und die Access-Datenbank völlig losgelöst von der SQL Server-Datenbank weiterexistiert. Es entsteht also weder eine Client-Server-Lösung noch eine File-Server-Lösung. Die SQL Server-Datenbank „Nordwind_NEU“ ist identisch mit der SQL Server-Datenbank „Nordwind_FS“.

4 Duplizieren einer SQL Server-Datendatenbank

Gelegentlich möchte man eine komplette SQL Server-Datenbank kopieren bzw. duplizieren, entweder unter demselben Namen auf einem anderen SQL Server oder unter einem oder mehreren anderen Namen auf demselben SQL Server. Beide Anwendungsszenarien seien hier kurz beschrieben.

4.1 Kopieren auf einen anderen Server

Die Gründe hierfür können vielfältig sein. Z.B. hat man eine Datenbank auf einem lokalen SQL Server entwickelt, die man später auf einem Netzwerk-SQL Server produktiv einsetzen möchte. Oder man möchte einem Kunden eine Datenbank oder einem Leser eine Demodatenbank zur Verfügung stellen.

Am einfachsten erledigt man das durch Kopieren der Datenbankdateien, genauer der Datendateien (*.mdf) und der Transaktionslogdateien (*.ldf), vom Speicherort des Quellservers zum Speicherort der Zielservers. Dabei ist darauf zu achten, dass die Datenbank des lokalen SQL Servers üblicherweise im Standarddatenbankordner auf dem lokalen Laufwerk C liegen, während das Laufwerk mit den Datenbankdateien eines Netzwerk-SQL Servers einen höheren Laufwerksbuchstaben (z.B. E) haben dürfte. U.U. muss man diesen bei der Netzwerkadministration erfragen, mit der auch die Zugriffsrechte auf dieses Verzeichnis zu klären sind.

Vor dem Kopieren muss man die Datenbank mit

datenbankname | TASKS | TRENNEN...

vom Quellserver getrennt haben. Das Trennen der Datenbank setzt wiederum voraus, dass man diese aktuell nicht verwendet, was man durch

USE *AndererDatenbankname*

erreicht. Nach dem Kopieren der Datenbankbankdateien kann man die Datenbank auf dem Zielserver über

Datenbanken | Anfügen...

zur Verfügung stellen. Natürlich geht das auch auf dem Quellserver, damit sie auch dort wieder verfügbar ist.

4.2 Kopieren auf demselben Server

Wir beschreiben zuerst, wie man auf demselben SQL Server eine einzige Kopie einer Datenbank unter einem anderen Namen anlegt. Anschließend wird dargelegt, wie aus einer einzige Quelldatenbank eine Vielzahl von Kopien unter synthetisch gebildeten Datenbanknamen erzeugt werden kann.

4.2.1 Erzeugen einer einzigen Datenbankkopie

Man kann eine Datenbank auf demselben Server unter anderem Namen kopieren, indem man sie zuerst sichert und anschließend unter einem anderen Namen wiederherstellt. Dialogorientiert erfolgt das Sichern mit dem SQL Server Management Studio im Objekt-Explorer über
datenbankname | TASKS | SICHERN...

Kommandoorientiert sichert man die Datenbank „Nordwind“ als Datei „Nordwind.bak“ im Backup-Ordner der Datenbank-Stammverzeichnisses mit

```
BACKUP DATABASE Nordwind
   TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\
   MSSQL\Backup\Nordwind.bak'
```

Das Wiederherstellen unter dem Namen „BXX0000_Nordwind“ (für den User „BXX0000“) erfolgt am besten kommandoorientiert, zumal dann, wenn man diesen Vorgang für eine größere Gruppe von Benutzern vornehmen will, wie anschließend gezeigt wird. Dabei wird unterstellt, dass in der Sicherungsdatei „Nordwind.bak“ folgende logischen Name verwendet werden:

- für die Datendatei: „Nordwind“
- für die Logdatei: „Nordwind_log“

```
RESTORE DATABASE BXX0000_Nordwind
   FROM DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL10_50.MSSQLSERVER\
   MSSQL\Backup\Nordwind.bak'
   WITH
   MOVE 'Nordwind' TO 'C:\Program Files\Microsoft SQL Server\
   MSSQL10_50.MSSQLSERVER\MSSQL\DATA\BXX0000_Nordwind.mdf',
   MOVE 'Nordwind_log' TO 'C:\Program Files\Microsoft SQL Server\
   MSSQL10_50.MSSQLSERVER\MSSQL\DATA\BXX0000_Nordwind_log.ldf'
```

Wenn man die logischen Namen der Daten- und Logdatei nicht kennt, muss man sie in einer gespeicherten Prozedur mit RESTORE ermitteln.

```
CREATE PROCEDURE DatenbankKopieren(@QuellDB VARCHAR(50)='Nordwind', @ZielDB VARCHAR(50))
AS
DECLARE @I INTEGER
DECLARE @DataDir VARCHAR(1000)
DECLARE @BackupDir VARCHAR(1000)
DECLARE @fileListTable AS TABLE(
        LogicalName          NVARCHAR(128),
        PhysicalName         NVARCHAR(260),
        [Type]               CHAR(1),
        FileGroupName        NVARCHAR(128),
        Size                 NUMERIC(20,0),
        MaxSize              NUMERIC(20,0),
        FileID               BIGINT,
        CreateLSN            NUMERIC(25,0),
        DropLSN              NUMERIC(25,0),
        UniqueID             UNIQUEIDENTIFIER,
        ReadOnlyLSN          NUMERIC(25,0),
        ReadWriteLSN         NUMERIC(25,0),
        BackupSizeInBytes    BIGINT,
        SourceBlockSize      INT,
        FileGroupID          INT,
        LogGroupGUID         UNIQUEIDENTIFIER,
        DifferentialBaseLSN  NUMERIC(25,0),
        DifferentialBaseGUID UNIQUEIDENTIFIER,
        IsReadOnl            BIT,
        IsPresent            BIT,
        TDEThumbprint        VARBINARY(32))
DECLARE @logicalNameMDF AS VARCHAR(100)         -- Logischer Name der Datendatei
DECLARE @logicalNameLDF AS VARCHAR(100)         -- Logischer Name der Logdatei
DECLARE @cmd VARCHAR(1000)
BEGIN TRY
-- Datenverzeichnis ermitteln
SET @DataDir = (SELECT physical_name FROM sys.database_files WHERE physical_name LIKE '%.mdf')
SET @I = CHARINDEX('DATA\',@DataDir) + 4
SET @DataDir = LEFT(@DataDir, @I)
-- Backupverzeichnis ermitteln
SET @BackupDir = LEFT(@DataDir, LEN(@DataDir)-5) + 'Backup\'
-- BACKUP der Datenbank
SET @cmd = 'BACKUP DATABASE ' + @QuellDB + ' TO DISK = ''' + @BackupDir + @QuellDB + '.bak'''
EXEC (@cmd)
-- Auslesen der logischen Dateinamen aus der BAK-Datei
SET @cmd = 'RESTORE FILELISTONLY FROM DISK = ''' + @BackupDir + @QuellDB + '.bak'''
INSERT INTO @fileListTable
        EXEC(@cmd)
SET @logicalNameMDF = (SELECT LogicalName FROM @fileListTable WHERE Type ='D')
SET @logicalNameLDF = (SELECT LogicalName FROM @fileListTable WHERE Type ='L')
-- RESTORE  unter neuem Datenbanknamen
SET @cmd = 'RESTORE DATABASE ' + @ZielDB +
        ' FROM DISK = ''' + @BackupDir + @QuellDB + '.bak''' +
        ' WITH ' +
        'MOVE ''' + @logicalNameMDF + ''' TO ''' + @DataDir + @ZielDB + '.mdf''' +
        ',MOVE ''' + @logicalNameLDF + ''' TO ''' + @DataDir + @ZielDB + '_log.ldf'''
EXEC (@cmd)
```

Mit `RESTORE FILELISTONLY` werden die Eigenschaften der Daten- und Logdateien aus der Datensicherungsdatei (*.bak) augelesen, die anschließend in die Tabellenvariable `@fileListTable` geschrieben werden, aus der wiederum die logischen Dateinamen der Daten- und Logdatei den beiden Variablen `@logicalNameMDF` und `@logicalNameLDF` zugewiesen werden. Damit liegen dann alle Informationen für den abschließenden `RESTORE DATABASE`-Befehl vor, dessen Ergebnis eine Kopie der Datenbank @QuellDB unter dem Namen @ZielDB ist.

4.2.2 Erzeugen multipler Datenbankkopien

Abschließend sei erläutert, wie man eine Datenbank für eine größere Anzahl von Benutzern vervielfältigt, wobei den Datenbanknamen ein Identifikationsmerkmal der Benutzer/innen vorangestellt wird. Es gebe drei Benutzer mit den Kennungen BAA0927, BAA1234 und BAF1614, für die mit Backup/Restore aus der Quelldatenbank „Nordwind“ drei identische Datenbanken

„BAA0927_Nordwind“, „BAA1234_Nordwind“ und „BAF1614_Nordwind“ erzeugt werden sollen. Die Identifikationsmerkmal stehen in einer Spalte „Kennung“ einer Tabelle „Student“.

```
CREATE PROCEDURE DatenbankMehrfachKopieren(
                @QuellDB VARCHAR(50)='Nordwind')
AS
SET NOCOUNT ON
-- A. Backup der Quelldatenbank erstellen
DECLARE @I INTEGER
DECLARE @DataDir VARCHAR(1000)
DECLARE @BackupDir VARCHAR(1000)
DECLARE @fileListTable AS TABLE(
        LogicalName          NVARCHAR(128),
        PhysicalName         NVARCHAR(260),
        [Type]               CHAR(1),
        FileGroupName        NVARCHAR(128),
        Size                 NUMERIC(20,0),
        MaxSize              NUMERIC(20,0),
        FileID               BIGINT,
        CreateLSN            NUMERIC(25,0),
        DropLSN              NUMERIC(25,0),
        UniqueID             UNIQUEIDENTIFIER,
        ReadOnlyLSN          NUMERIC(25,0),
        ReadWriteLSN         NUMERIC(25,0),
        BackupSizeInBytes    BIGINT,
        SourceBlockSize      INT,
        FileGroupID          INT,
        LogGroupGUID         UNIQUEIDENTIFIER,
        DifferentialBaseLSN  NUMERIC(25,0),
        DifferentialBaseGUID UNIQUEIDENTIFIER,
        IsReadOnl            BIT,
        IsPresent            BIT,
        TDEThumbprint        VARBINARY(32))
DECLARE @logicalNameMDF AS VARCHAR(100)
DECLARE @logicalNameLDF AS VARCHAR(100)
DECLARE @cmd VARCHAR(1000)
BEGIN TRY
-- Datenverzeichnis ermitteln
SET @DataDir = (SELECT physical_name FROM sys.database_files WHERE physical_name LIKE '%.mdf')
SET @I = CHARINDEX('DATA\',@DataDir) + 4
SET @DataDir = LEFT(@DataDir, @I)
-- Backupverzeichnis ermitteln
SET @BackupDir = LEFT(@DataDir, LEN(@DataDir)-5) + 'Backup\'
-- BACKUP der Datenbank
SET @cmd = 'BACKUP DATABASE ' + @QuellDB + ' TO DISK = ''' + @BackupDir + @QuellDB + '.bak'''
EXEC (@cmd)
-- Auslesen der logischen Dateinamen aus der BAK-Datei
SET @cmd = 'RESTORE FILELISTONLY FROM DISK = ''' + @BackupDir + @QuellDB + '.bak'''
INSERT INTO @fileListTable
        EXEC(@cmd)
SET @logicalNameMDF = (SELECT LogicalName FROM @fileListTable WHERE Type ='D')
SET @logicalNameLDF = (SELECT LogicalName FROM @fileListTable WHERE Type ='L')
--
-- B. Erzeugen der Zieldatenbanken mit vorangestellter Benutzerkennung
DECLARE student_cursor INSENSITIVE CURSOR
                FOR SELECT Kennung
                FROM Student
DECLARE @kennung VARCHAR(20)
DECLARE @ZielDB VARCHAR(1000)
OPEN student_cursor
FETCH NEXT FROM student_cursor INTO @kennung
SET @ZielDB = @kennung+'_'+@QuellDB
IF @ZielDB NOT IN(SELECT name FROM sys.databases)
     BEGIN
     -- RESTORE der Quell-Datenbank als Zieldatenbank
     SET @cmd = 'RESTORE DATABASE ' + @ZielDB +
               ' FROM DISK = ''' + @BackupDir + @QuellDB + '.bak''' +
               ' WITH ' +
               'MOVE ''' + @logicalNameMDF + ''' TO ''' + @DataDir + @ZielDB + '.mdf''' +
               ',MOVE ''' + @logicalNameLDF + ''' TO ''' + @DataDir + @ZielDB + '_log.ldf'''
     EXEC (@cmd)
     -- Besitzer ändern
     SET @cmd = 'ALTER AUTHORIZATION ON DATABASE:: ' +  @ZielDB + ' TO [UNI-HAMBURG\' + @kennung +']'
```

```
     EXEC (@cmd)
     END
WHILE @@fetch_status = 0
     BEGIN
     FETCH NEXT FROM student_cursor INTO @kennung
     SET @ZielDB = @kennung+'_'+@QuellDB
     IF @ZielDB NOT IN(SELECT name FROM sys.databases)
          BEGIN
          -- RESTORE der Quell-Datenbank als Zieldatenbank
          SET @cmd = 'RESTORE DATABASE ' + @ZielDB +
                 ' FROM DISK = ''' + @BackupDir + @QuellDB + '.bak''' +
                 ' WITH ' +
                 'MOVE ''' + @logicalNameMDF + ''' TO ''' + @DataDir + @ZielDB + '.mdf''' +
                 ',MOVE ''' + @logicalNameLDF + ''' TO ''' + @DataDir + @ZielDB + '_log.ldf'''
          EXEC (@cmd)
          -- Besitzer ändern
          SET @cmd = 'ALTER AUTHORIZATION ON DATABASE:: ' +  @ZielDB + ' TO [UNI-HAMBURG\' + @kennung +']'

          EXEC (@cmd)
          END
     END
CLOSE student_cursor
DEALLOCATE student_cursor
END TRY
BEGIN CATCH
     SELECT ERROR_NUMBER() AS Fehlernummer, ERROR_SEVERITY() AS Fehlerschwere,
                 ERROR_STATE() AS Fehlerzustand, ERROR_PROCEDURE() AS 'Auslösende Prozedur',
                 ERROR_LINE() AS Fehlerzeile, ERROR_MESSAGE() AS Fehlernachricht;
END CATCH
RETURN
```

Das BACKUP der Quelldatenbank und die Ermittlung der logischen Dateinamen der Daten- und Logdateien im Backup erfolgt wie im Abschnitt 4.2.1 beschrieben. Erweitert wurde der RESTORE-Vorgang in die Zieldatenbanken. Mit einem Cursor werden die Kennungen aus der Tabelle „Student“ in einer Schleife in die Variable @kennung geschrieben. Die Namen der Zieldatenbanken ergeben sich dann aus der Verkettung der Kennungen, eines Unterstrichs und des Namens der Quelldatenbank. Das RESTORE erfolgt genauso wie oben beschrieben. Ergänzt werden muss noch die Besitzübergabe an die Benutzer mit ALTER AUTHORIZATION, wobei den Kennungen die Domäne (hier: „UNI-HAMBURG“) vorangestellt wird.