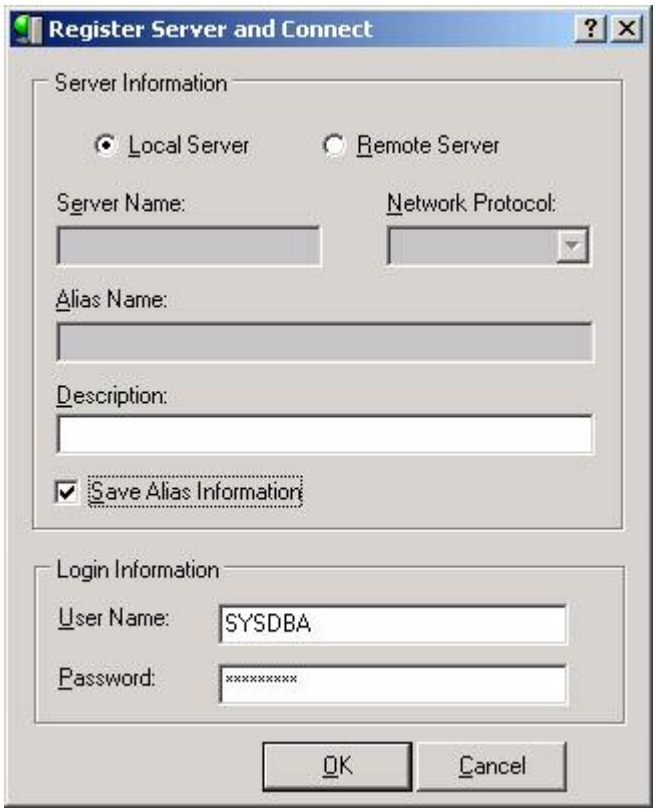
SQL Tutorial

In diesem Tutorial geht es um die Einführung in das Thema SQL (structured query language). Mit SQL ist es möglich in jedem DBMS (Datenbank Management System) Zugriff auf die Daten zu erhalten. Es gibt verschiedene SQL-Standards, der neueste ist SQL-92 (von 1992). Allerdings hält sich kein einziges DBMS komplett an diesen Standard, jede hat irgendwelche Erweiterungen oder Einschränkungen, um gewisse Features zu erhalten, die die anderen DBMS nicht haben.

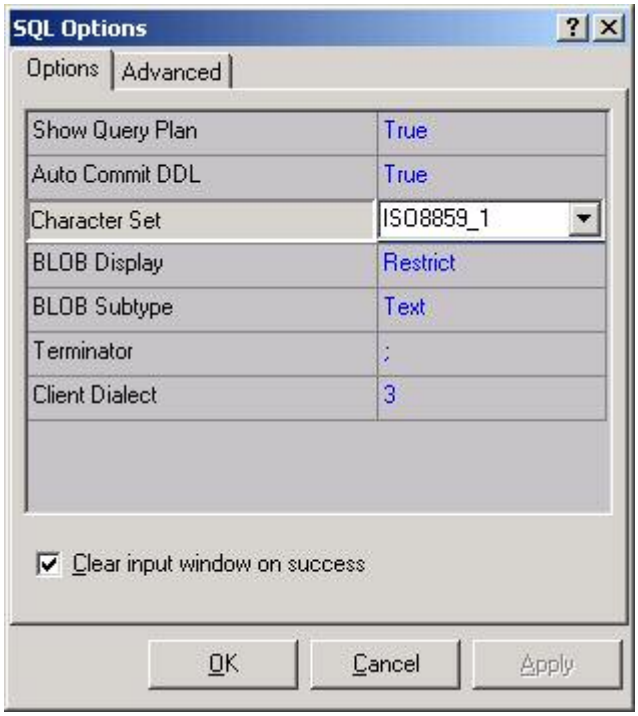
Das Tutorial könnt Ihr mit Interbase oder Firebird nachvollziehen. Interbase gibt es (auch) als OpenSource für Linux und Windows unter http://mers.com.
Für den Zugriff auf Interbase verwende ich die IBConsole. Diese ist bei Interbase schon mit dabei. Für Firebird müsst Ihr die aktuelle Version von CodeCentral runterladen: http://codecentral.borland.com. Damit auch Ihr damit umgehen könnt, kommt hier erst mal eine kleine Einführung in IBConsole.

Nach dem ersten Start muss zuerst der lokale Server registriert werden. Dazu einfach einen Doppelklick auf "InterBase Servers", dann erscheint folgendes Fenster:



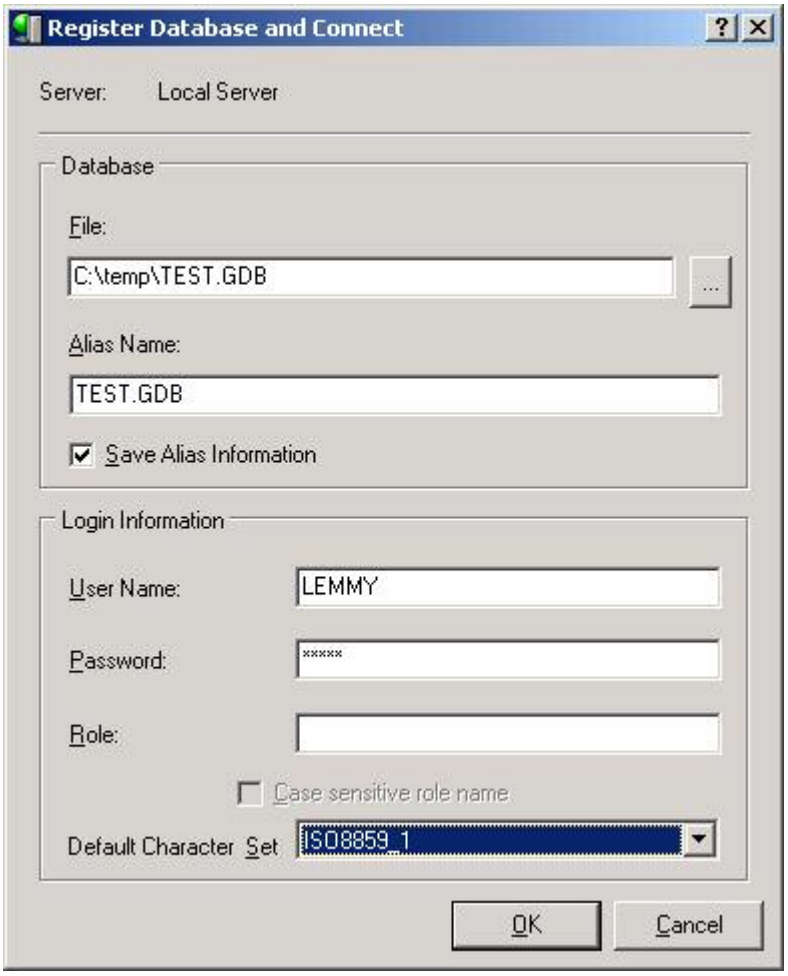
Das einzig vordefinierte User-Konto ist das SYSDBA:
User Name: SYSDBA
Password: masterkey

Nun ist der lokale Server registriert. Macht nun einen Doppelklick auf den Eintrag "Local Server". Um SQL-Statements ausführen zu können benötigen wir noch das ISQL-Fenster (*Tools - Interactiv SQL*). Im ISQL stellt Ihr unter *Edit - Options* folgende Dinge ein:



Achtet darauf, dass als Client Dialect "3" eingestellt ist. Beim Character Set stellt Ihr "ISO8859_1" ein.

Nach der Erzeugung einer Datenbank solltet Ihr diese in der IBConsole registrieren. Dann wirds später wesentlich einfacher:



Einfach einen Rechtsklick auf "Databases" machen und "Register" auswählen. Gebt im folgenden Fenster diese Daten ein (oder eben entsprechend!):

Achtet hier auf die richtige Einstellung des Default Character Set!

Die SQL-Kommandoklassen

Die SQL ist so umfangreich, dass die Anweisungen in verschiedene Kommandoklassen unterteilt werden:

Database Administration Language (DAL)

SQL-Befehl	**Verwendung**
Create Database	Erzeugt eine neue Datenbank
Drop Database	Löscht eine bestehende Datenbank

Data Definition Language (DDL)

SQL-Befehl	**Verwendung**
CREATE DOMAIN	Erzeugt einen Datentyp
Create Index	Erzeugt einen Index für eine bestehende Tabelle
Create Table	Eine Tabelle wird angelegt
Create View	Dient zur Erzeugung von speziellen Sichten auf Tabellen
Create Generator	Erzeugt einen Generator für Auto-Inc Werte

Transaction Control Language (TCL)

SQL-Befehl	**Verwendung**
Commit	Bestätigt die Aktionen innerhalb einer Transaktion und beendet diese (hartes Commit)
Commit Retaining	Bestätigt alle Aktionen innerhalb einer Transaktion, beendet diese aber nicht, so dass alle offenen Ressourcen weiterhin zur Verfügung stehen (weiches Commit)
Rollback	Verwirft alle Aktionen innerhalb der Transaktion und beendet diese

Data Manipulation Language (DML)

SQL-Befehl	**Verwendung**
Delete	Löscht einen oder mehrere Datensätze aus einer Tabelle
Insert	Legt einen oder mehrere neue Datensätze an
Update	Aktualisiert Daten eines oder mehrerer Datensätze

Data Query Language (DQL)

SQL-Befehl	**Verwendung**
Select	Liefert Spalten (Attribute) aus einer oder mehreren Tabellen, Views oder Stored Procedures zurück.

Die hier aufgeführten SQL-Befehle sind nur ein kleiner Teil der von IB/Fb unterstützen Kommandos. Eine komplette Übersicht (in Englisch) findet Ihr im Handbuch unter „SQL-Reference”.

Die Database Administration Language

Mit der DAL können neue Datenbanken angelegt, verändert sowie gelöscht werden.
CREATE DATABASE

Damit man die "Inereien" einer Datenbank erstellen kann, benötigt man zuerst mal eine Datenbank. Diese wird mit CREATE DATABASE erzeugt.

```
CREATE {DATABASE | SCHEMA} 'filespec'
[USER 'username' [PASSWORD 'password']]
[PAGE_SIZE [=] int]
[LENGTH [=] int [PAGE[S]]]
[DEFAULT CHARACTER SET charset]
[<secondary_file>];
<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]
<fileinfo> = [LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int }
```

Die Attribute im Einzelnen:

filesspec: Der Pfad und der Name der neuen Datenbank.
username, password: Der Benutzer mit seinem Passwort, der die Datenbank erzeugen soll.

PAGE_SIZE int: Die Page-Größe der Datenbank. Diese kann 1024, 2048, 4096 oder 8192 betragen. Hier sollte normalerweise 4096 gewählt werden. Weitere Hinweise dazu im 2. IBX-Tutorial.

DEFAULT CHARACTER SET charset: Setzt den Zeichensatz für die Datenbank. Für Deutschland (Umlaute, ß) hat sich der Zeichensatz ISO8859_1 bewährt. Alle in dieser Datenbank angelegten Spalten die Texte beinhalten werden mit diesem Zeichensatz definiert.

secondary_file: Es ist möglich eine IB/FB Datenbank auf verschiedene Dateien zu verteilen. Das hat folgende Vorteile:
1. Ist das Datenbankfile größer als 4GB kann diese auf verschiedene Dateien verteilt werden. Ein größeres Datenbankfile als 4 GB wird von IB/FB nicht unterstützt.
2. Man kann die Datenbankfiles auf unterschiedliche Festplatten verteilen (Geschwindigkeitsvorteil)

Das Verteilen auf mehrere Files macht allerdings nur bei großen Systemen Sinn!

Um eine Testdatenbank zu erstellen lautet der Befehl wie folgt:

```
CREATE DATABASE 'c:\temp\test.gdb' user 'lemmy' password
'lemmy' PAGE_SIZE 4096 DEFAULT CHARACTER SET ISO8859_1 COLLATE
DE_DE;
```

DROP DATABASE

```
DROP DATABASE;
```

Mit Drop Database wird die aktuell verbundene Datenbank gelöscht. Es erfolgt keine Sicherheitsabfrage! Eine Datenbank kann vom Erzeuger, vom SysDBA und von allen anderen (IB/FB)Usern gelöscht werden, die System-Administratoren sind.

Data Definition Language (DDL)

Mit der DDL wird die Struktur der Datenbank festgelegt.

DOMAINs

Eine Domain ist in Interbase/Firebird vergleichbar mit einem Typ in Delphi. Es können individuelle Variabelntypen erzeugt werden. Allerdings spielen Domains in IB/FB eine besonderer Rolle. Beim Erzeugen einer Tabelle werden für alle Attribute Domains angelegt. Bei einer umfangreichen Datenbank führt das dazu, dass eine Vielzahl nicht wiederverwendbaren Domains erzeugt werden. Werden allerdings vom Benutzer Domains bei der Erstellung von Tabellen verwendet werden keine neuen Domains angelegt. Das führt zu einer schlankeren Datenbank, die wesentlich schneller ein Backup bzw. Restore ausführen kann.

Hier jetzt erst mal eine Übersicht der in IB/FB verfügbaren Typen:

- BLOB: Für die Speicherung von beliebigen Daten (z.B. Memo-Feldern, Bilder, ZIP-Files,....)
- CHAR(n): Der erste String-Typ. Mit dem Parameter n kann die Anzahl der Zeichen eingestellt werden, wobei 1<=n<=32.767 sein muss.
- DATE: Eine 64 Bit Variable, in der Datumswerte vom 01. Januar 100 a.d. bis 29 Februar 32.768 gespeichert werden können. Für Standardanwendungen auch in den nächsten Jahren wohl ausreichend ;-)
- DECIMAL(n,m): Ein Datentyp zum Speichern von Gleitkommazahlen. Dazu später mehr.
- DOUBLE PRECISION: Dieser Typ ist ein 64 Bit Gleitkommazahltyp. Der Typ hat 15 genaue Stellen.
- FLOAT: Ein 32 Bit Typ mit 7 genauen Stellen.
- INTEGER: Ein 32 Bit für Ganzzahlen mit einem Gültigkeitsbereich von -2.147.483.648 bis 2.147.483.647
- NUMERIC(n,m): wie Decimal(n,m)
- SMALLINT: Ein 16 Bit Integer mit einem Gültigkeitsbereich von -32.768 bis 32.767
- TIME: Mit diesem Typ könne Zeitangaben von 0:00.0000 bis 32:59.9999 gespeichert werden.
- TIMESTAMP: Ein Typ in dem Datum und Zeit gespeichert werden (vgl. DateTime in Delphi). Der Gültigkeitsbereich ist wie bei Date und Time.
- VARCHAR(n): Der zweite String-Typ. Auch hier gilt: 1<=n<=32.767. Viele glauben (auch ich habe so gedacht), dass VARCHAR nur dann Speicher belegt, wenn auch was in der Spalte steht, Char aber immer den vollen Platz belegt. Fakt ist aber, dass IB beide, Char und VarChar, komprimiert in der Datenbank abspeichert. Dennoch sollte man nicht verschwenderisch mit der Längenangabe umgehen. IB/FB erstellen Temporäre Dateien (z.B. zum sortieren). Dort werden

die Strings unkomprimiert in voller Länge dargestellt. Bei IB 6.5 werden VarChars erstmals in verkürtzer Länge übertragen. Bei FB bin ich im Moment überfragt.

Gleitkommazahlen

Jeder kennt vermutlich das Problem bei den Gleitkommazahlen: sie sind nicht genau! Schuld daran ist die Art wie diese Zahlen gespeichert werden. Eigentlich sollten die 15 genauen Stellen von Double Precision für viele Anwendungen ausreichen, doch ein paar Rechenschritte genügen, um Rundungsfehler hervorzurufen. Um das zu vermeiden gibt es die Typen NUMERIC und DECIMAL. Da es so gut wie keine Unterschiede bei den beiden gibt, werde ich nur auf NUMERIC eingehen.
Je nach Größe der eingegebenen Parameter werden die NUMERIC Typen unterschiedlich gespeichert:

- NUMERIC(4): SMALLINT
- NUMERIC(9): INTEGER
- NUMERIC(18): INT64
- NUMERIC(4,2): SMALLINT
- NUMERIC(10,2): INTEGER

Bei NUMERIC(10,2) wird eine GLEITKOMMAZAHL in einem INTEGER gespeichert!?! Das geht eigentlich ganz einfach: Intern wird die Zahl als Integer gespeichert, für die Ausgabe wird Sie dann einfach mit 100 dividiert (10^2).
Hinweis: Ein NUMERIC(6) wird natürlich ebenfalls in einem INTEGER gespeichert. Die Angaben beziehen sich immer auf die Obergrenze.
Weitere Angaben finden sich im Handbuch von Interbase: Data Definition Guide.

Nun zurück zu Create Domain:

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [CHECK (<dom_search_condition>)]
[COLLATE collation];
```

domain: Eindeutiger Name für die Domain (z.B. TStr10)

datatype: Hier kommt der Datentyp rein. Der kann aus alle oben aufgeführten Datentypen bestehen:

```
CREATE DOMAIN TStr10 AS VARCHAR(10) COLLATE DE_DE;
```

Die Angabe COLLATE DE_DE stellt die deutsche Sortierung ein (Sonderzeichen wie Umlaute).

Um einen Standardwert zu setzen:

```
CREATE DOMAIN TStr10 AS VARCHAR(10) DEFAULT 'Hallo';
```

Um eine Kontrolle über die eingegebene Werte zu haben, kann man folgendes machen:

```
CREATE DOMAIN TBool1 CHAR(1) CHECK (VALUE IN ('J','N')) NOT
NULL;
```

Hier wird ein Bool'scher Typ erzeugt. Der Wert muss entweder 'J' oder 'N' sein, kein Wert ist nicht erlaubt.

Bei CHECK (VALUE...) kann man einiges machen:

```
CHECK (VALUE BETWEEN ())
CHECK (VALUE LIKE ())
CHECK (VALUE CONTAINING ())
CHECK (VALUE STARTING ())
```

Mit ALTER DOMAIN können die Domains verändert werden, allerdings nur in einem bestimmten Maß, aus einem String kann kein Integer gemacht werden.

```
ALTER DOMAIN TStr10 TYPE VarChar(20);
```

Um den Namen zu ändern reicht dann ein:

```
ALTER DOMAIN Tstr20 TO Tstr30;
```

Mit DROP DOMAIN wird eine bestehende DOMAIN gelöscht, allerdings darf diese DOMAIN nicht mehr verwendet werden.

Tabellen

In den Tabellen wird der eigentliche Inhalt einer Datenbank gespeichert. Diese müssen definiert werden. Dazu dient CREATE TABLE

```
CREATE TABLE table [EXTERNAL [FILE] 'filespec']
(<col_def> [, <col_def> | <tconstraint> …]);
<col_def> = col {<datatype> | COMPUTED [BY] (<expr>) | domain}
[DEFAULT {literal | NULL | USER}]
[NOT NULL]
[<col_constraint>]
[COLLATE collation]
```

Sieht erst mal nicht so spannend aus, aber wer das Interbase-Handbuch zur Hand hat, sollte sich mal die komplette Befehlssyntax anzeigen lassen.

table: Der Tabellenname. Dieser muss in einer Datenbank eindeutig sein.

EXTERNAL FILE: Mit Create Tabel kann auch eine externe Datei (ASCII) angesprochen werden. Die kann z.B. zum Datenim- und -export verwendet werden. Dazu später mehr...

col-def: Das ist die Typdefinition der Spalte. Hier kann eine Domain stehen oder auch die Standardtypen. Hier sind alle Dinge erlaubt, die schon bei den DOMAINS erlaubt sind und noch einige mehr. Das wichtigste ist die Definition des Primary Key:

```
CREATE TABLE Adresse (
ID TInt NOT NULL,
Name TStr30,
Vorname TStr30,
Adresse TStr50,
PRIMARY KEY (ID));
```

Der Primary Key oder auch Primärschlüssel ist für die eindeutige Identifikation eines Datensatzes zuständig. Für den Primärschlüssel wird zudem ein Index aufgebaut.

Neben dem Primärschlüssel lassen sich auch manuell Indizes aufbauen:

```
CREATE INDEX ON Adresse (Name);
```

Es wird dann ein Index in aufsteigender Reihenfolge erstellt. Um einen absteigenden Index zu erstellen genügt das Schlüsselwort DESC:

```
CREATE DESC INDEX ON Adresse (Name);
```

Ist eine Tabelle erst mal engelegt, kann man mit ALTER TABLE die Tabellenstruktur verändern. Nebenbei kann man damit Fremdschlüssel damit erstellen. Fremdschlüssel sind dazu da Tabellen miteinander zu verbinden. Sie werden auch oft dazu eingesetzt gewisse Automatismen anzustoßen.

```
CREATE TABLE Bestellung
( ID TInt NOT NULL,
Artikel TStr30,
Preis TMoney,
Adresse_ID TInt NOT NULL,
PRIMARY KEY (ID));

ALTER TABLE Bestellung ADD Foreign Key  (Adresse_ID)
REFERENCES Adresse (ID);
```

Dieser Fremdschlüssel dient dazu, dass keine Bestellungen aufgenommen werden können, wenn die entsprechende Adresse nicht existiert. Wird eine Adresse gelöscht zu der noch Bestellungen existieren, wird eine Fehlermeldung erzeugt, da die Bestellungen sonst ohne Adresse wären.

Mit Alter Table können Spalten zu Tabellen hinzugefügt werden:

```
ALTER TABLE Bestellung ADD Anzahl TInt;
```
oder gelöscht werden:

```
ALTER TABLE Bestellung drop anzahl;
```

Man kann auch in beschränktem Umfang den Typ einer Spalte ändern:

```
ALTER TABLE Adresse alter Name Type Tstr20;
```

Dabei muss allerdings der neue Typ mindestens so groß sein wie der alte Typ. Um den Namen zu ändern genügt ein

```
ALTER TABLE Adresse alter Name to Firma
```

Um eine Tabelle zu löschen genügt ein

```
DROP TABLE Bestellung;
```

Nach Änderungen an der Tabellenstruktur (ALTER TABLE, CREATE TABLE, DROP TABLE) ist es sinnvoll ein Backup-Restore durchzuführen. Das hat den Vorteil, dass zum einen sichergestellt ist, dass im Notfall eine Wiederherstellung der Daten möglich ist, zum anderen ermöglicht es dem DBMS die Datenbank neu aufzubauen.

Zulestzt möchte ich noch kurz auf externe Tabellen eingehen. Wenn ein Datenimport oder -export gemacht werden soll, kann dieser mit den externen Tabellen schnell und einfach erledigt werden. Die Daten stehen dann in ASCII-Dateien bereit. Eine externe Tabelle wird folgendermaßen erzeugt:

```
CREATE TABLE AdressenImport External File
'C:\Temp\Adresse.txt' (

ID TINT,
NAME TStr30,
Vorname TStr30,
Adresse TStr30,
CRLF SmallInt);
```

Das CRLF ist das „Zeilenumbruchzeichen" das bei externen Dateien immer angegeben werden muss (wenn es denn existiert). Für den Export muss dieses Zeichen natürlich extra geschrieben werden, sonst sind alle Daten in einer riesigen Bandwurmzeile gepackt. Für Windows beträgt der Wert des Smallintegers 2573.

Bei den externen Dateien gibts noch eine Besonderheit: Alle Veränderungen (Insert) laufen außerhalb der Transaktionssteuerung, d.h. diese können nicht mit Rollback zurückgenommen werden. Ändern oder löschen der Daten ist nicht möglich. Es können auch keine BLOb-Felder in Externe Dateien gespeichert werden.

Um Daten zu exportieren:

```
INSERT INTO ExterneDatei SELECT Name, Vorname FROM PERSONAL;
```

Um Daten zu importieren:

```
INSERT INTO Personal SELECT Name, Vorname FROM ExterneDatei;
```

Dabei muss die externe Datei die Attribute entsprechend abbilden, d.h. für Name müssen 40 Zeichen Platz da sein, bevor der Vorname kommt.
Es gibt aber ein paar Dinge, die man bei externen Tabellen beachten muss:

- Es können nur Daten mit festgelegter Länge in einer externen Tabelle gespeichert werden (also keine BLOb, VARCHAR nur bedingt).
- Bei diesen Dateien muss darauf geachtet werden, dass das richtige EOL-Zeichen (end of line) verwendet wird, Unix/Linux und Windows haben da unterschiedliche Definitionen! Für dieses EOL muss ein extra Attribut angelegt werden, ansonsten werden die exportierten Daten in eine Zeile geschrieben.
- Es können nur INSERTS und SELECTS in den externen Dateien ausgeführt werden, DELETES und UPDATES werden mit einer Fehlermeldung quittiert! Wenn Daten in die Datei gespeichert werden, geschieht das nicht im Rahmen einer Transaktion sondern sofort, d.h. die Änderungen können nicht zurückgenommen werden!

Zum CREATE VIEW gibt es nicht viel zu sagen. Ein View ist eine bestimmte Sicht auf eine oder mehrere Tabellen. Ein View wird folgendermaßen erstellt:

```
CREATE VIEW name [(view_col [, view_col ])]
AS <select> [WITH CHECK OPTION];
```

view_col sind wie bei den Tabellen die Spalten, die genauso auch bezeichnet werden. Bei <select> kommt ein Select-Statement, das ich später beschreiben werde. Bei jedem Zugriff auf den View wird der Select ausgeführt. Die Daten eines Views können nicht bearbeitet werden!

In IB/FB gibt es keine Auto-Inc Felder wie bei Paradox oder Access. Dafür gibt es als gewissen Ersatz die Generatoren. Diese erzeugen eine fortlaufende Integerzahl, die für die Primärschlüssel verwendet werden können:

```
CREATE GENERATOR gen_adresse;
```

Der Name des Generators muss wieder eindeutig sein.

Transaction Control Language (TCL)

Mit der TCL benötigt man im Umfeld von IB/FB. In diesen DMBS werden alle Aktionen im Rahmen einer Transaktion ausgeführt. Durch die Rücknahme einer Transaktion werden alle Änderungen rückgängig gemacht und erst bei einer Bestätigung einer Transaktion werden diese für alle anderen Benutzer sichtbar.

Mit COMMIT oder COMMIT RETAINING werden Transaktionen beendet und die Änderungen in die Datenbank übernommen. Der Unterschied: Bei Commit Retaining wird sofort nach dem COMMIT wieder eine neue Transaktion gestartet. In dieser stehen dann wieder alle Ressourcen wie in der vorhergehenden Transaktion zur

Verfügung. Nach einem Insert stehet z.B. der Zeiger in der Datenbank auf dem Datensatz, der in der letzten Transaktion erzeugt wurde. Bei einem Commit ist das nicht der Fall, da dort alles geschlossen wird.
Allerdings hat der Umgang mit Commit Retaining (weiches Commit) einen Nachteil: Jede Transaktion erhält eine einndeutige Nummer. Zudem wir für jede Transaktion ein Snapshot angelegt, d.h. der aktuelle Zustand der Datenbank wird sozusagen „kopiert". Bei einem Commit Retaining wird diese Nummer beibehalten. Der Unterschied zwischen aktueller Transaktionsnummer und ältester Transaktionsnummer (OAT oldest active transaction) wächst dabei ständig und damit sinkt die Leistungsfähigkeit der Datenbank. Erst durch das Beenden einer Transaktion bzw. durch ein Backup/Restore wird dieses Problem behoben.

Sollen Änderungen mal rückgängig gemacht werden genügt ein Rollback. Dabei werden alle Ressourcen geschlossen.

Data Manuipulation Language (DML)

In der DML sind die 3 Befehle zusammengefasst mit denen der Inhalt der Tabllen verändert werden kann:

INSERT - UPDATE - DELETE

INSERT

Mit Insert werden Daten in eine Tabelle geschrieben. Das kann auf 2 unterschiedliche Arten erfolgen:

```
INSERT INTO Adresse VALUES (1, 'Müller', NULL, 'Irrweg 12,
7000 Stuttgart');
```

oder

```
INSERT INTO Adresse (ID, Name, Anschrift) VALUES (2, 'Müller',
'Irrweg 12, 7000 Stuttgart');
```

Im ersten Fall werden die Spalten der Tabellen nicht angegeben. Dann müssen allerdings alle Spalten mit Inhalt gefüllt werden, die in der Tabelle existieren, und das auch noch in der richtigen Reihenfolge. Für Werte die man in diesem Fall nicht füllen möchte, muss NULL eingefügt werden.

Im zweiten Fall werden alle Spalten die gefüllt werden sollen aufgeführt. In den „Values" werden dann in entsprechender Reihenfolge die Werte aufgeführt.

Es gibt auch die Möglichkeit sehr einfach eine Kopie eines Datensatzes zu erstellen:

```
INSERT INTO Adresse Select 2, Firma, Vorname, Adresse from
Adresse where ID=1;
```

Das Insert holt sich die Daten direkt aus einem Select. Die ID wird hier fest vorgegeben, ein:

```
INSERT INTO Adresse Select Gen_ID(Gen_adresse,1), Firma,
Vorname, Adresse from Adresse where ID=1;
```

kann sich diese natürlich auch direkt aus dem Generator holen.

UPDATE

Bei einem Update werden bestehende Werte einer Tabelle verändert. Spätestens hier sollte man bedenken, dass SQL-Datenbanken Mengenorientiert arbeiten. Wenn man also Änderungen nur an bestimmten Datensätzen vornehmen will, muss man dafür Sorge tragen, dass die Auswahl entsprechend ausfällt. Die Auswahl geschieht in der sogenannten WHERE-Clause.

```
UPDATE Adresse SET Vorname='Manfred' WHERE Vorname is NULL;
```

Dieses Update würde allen Adressen deren Vornahme nicht gesetzt ist auf „Manfred" taufen.

DELETE

Mit Delete werden Datensätze aus der Datenbank entfernt. Wie bei Update wirkt sich Delete ohne WHERE-clause auf den gesamten Bestand einer Tabelle aus.

```
DELETE FROM Adresse;
```

löscht somit alle Datensätze, die sich in der Tabelle Adresse befinden.

Data Query Language (DQL)

Die DQL besteht aus einer einzigen Anweisung: SELECT. Dieser vermutlich am meisten verwendete SQL-Befehl ist gleichzeitig auch der mächtigste:

```
SELECT [TRANSACTION transaction]
[DISTINCT | ALL]
{* | <val> [, <val> …]}
[INTO :var [, :var …]]
FROM <tableref> [, <tableref> …]
[WHERE <search_condition>]
[GROUP BY col [COLLATE collation] [, col [COLLATE collation] …]
[HAVING <search_condition>]
[UNION <select_expr> [ALL]]
[PLAN <plan_expr>]
[ORDER BY <order_list>]
[FOR UPDATE [OF col [, col …]]];
<val> = {
col [<array_dim>] | :variable
| <constant> | <expr> | <function>
| udf ([<val> [, <val> …]])
| NULL | USER | RDB$DB_KEY | ?
} [COLLATE collation] [AS alias]
<array_dim> = [[x:]y [, [x:]y …]]
<constant> = num | 'string' | charsetname 'string'
<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
| SUM ([ALL] <val> | DISTINCT <val>)
| AVG ([ALL] <val> | DISTINCT <val>)
| MAX ([ALL] <val> | DISTINCT <val>)
| MIN ([ALL] <val> | DISTINCT <val>)
| CAST (<val> AS <datatype>)
| UPPER (<val>)
| GEN_ID (generator, <val>)
```

```
<tableref> = <joined_table> | table | view | procedure
[(<val> [, <val> …])] [alias]
<joined_table> = <tableref> <join_type> JOIN <tableref>
ON <search_condition> | (<joined_table>)
<join_type> = [INNER] JOIN
| {LEFT | RIGHT | FULL } [OUTER]} JOIN
<search_condition> = <val> <operator> {<val> | (<select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> …] | <select_list>)
| <val> IS [NOT] NULL
| <val> {>= | <=}
| <val> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| (<search_condition>)
| NOT <search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
<plan_expr> =
[JOIN | [SORT] [MERGE]] ({<plan_item> | <plan_expr>}
[, {<plan_item> | <plan_expr>} …])
<plan_item> = {table | alias}
{NATURAL | INDEX (<index> [, <index> …]) | ORDER <index>}
<order_list> =
{col | int} [COLLATE collation]
[ASC[ENDING] | DESC[ENDING]]
[, <order_list> …]
```

Das hier ist die komplette Befehlssyntax von Select. Ich werde aber nicht auf alles eingehen (können), sondern die wichtigsten Dinge wieder rauspicken.
Mit

```
SELECT * FROM Adresse
```

werden alle Datensätze der Tabelle von der Datenbank angefordert. Eine Einschränkungen der Datenmenge kann zum einen durch die Verringerung der Spalten:

```
SELECT Name, Vorname From Adresse
```

oder durch die Verringerung der Anzahl der Datensätze:

```
SELECT * FROM ADresse where Name<'B';
```

erreicht werden. Im ersten Fall werden wieder alle Datensätze zurückgeliefert, allerdings nur der Name und Vorname, im zweiten Fall werden nur die Datensätze zurückgeliefert, deren Name kleiner als „B" ist.

Wenn Daten aus verschiedenen Tabellen selektiert werde sollen funktioniert das so:

```
SELECT * FROM Adresse, Bestellung where Adresse.ID=Adresse_ID;
```

Das ist die Syntax des SQL89-Standards. Der aktuelle Standard ist allerdings SQL92 und da sieht das ganze so aus:

```
SELECT * FROM Adresse JOIN Bestellung ON
Adresse.ID=Adresse_ID;
```

Wo liegt der Unterschied? Im SQL92 wurde die Möglichkeit der JOINS eingeführt. Schauen wir mal an, was diese beiden als Ergebnismenge zurückliefern: Sie liefern alle Adressen, zu denen eine Bestellung erfasst worden ist incl. aller Daten der Bestellungen. Was aber wenn man auch die Adressen haben will, zu denen keine Bestellung erfasst worden ist? Mit der alten Syntax hat man kaum eine Möglichkeit, wohl aber mit den Joins:

```
SELECT * FROM Adresse LEFT OUTER JOIN Bestellung ON
Adresse.ID=Adresse_ID;
```

So, aus dem JOIN wurde ein LEFT OUTER JOIN und der liefert genau das was wir wollen: Alle Adressen und, falls welche existieren, die Bestellung gleich dazu. Aber wenn es einen „left outer join" gibt muss es auch einen „right outer join" geben. Den gibt es, aber der hat faktisch keinen Funktionsunterschied:

```
SELECT * FROM Bestellung RIGHT OUTER JOIN Adresse ON
Adresse.ID=Adresse_ID;
```

Das SQL-Statement ist identisch mit dem oberen. Neben dem Outer-Join gibt es auch noch einen Inner-Join, doch diesen kennt Ihr bereits:

```
SELECT * FROM Adresse JOIN Bestellung ON
Adresse.ID=Adresse_ID;
```

Hier erkennt Ihr übrigends wie man Spalten eindeutig anspricht. Die Spalte ID existiert in beiden Tabellen. Die Datenbank weiß nicht welche sie nehmen muss. Wenn bei einer Select mal Ergebnisse rauskommen, die überhaupt nichts mit dem Select zu tun haben, kontrolliert erst mal, ob Ihr irgendwo eine solche Doppelbezeichnung nicht ordentlich aufgelöst habt. Bei langen Tabellennamen oder für faule Menschen gibt es auch die Möglichkeit mit einen Tabellen-Alias zu arbeiten:

```
SELECT * FROM Adresse a JOIN Bestellung b ON a.ID=Adresse_ID;
```

Dabei wird hinter der Tabelle ein Alias angegeben. Dieser kann aus einem oder mehreren Zeichen bestehen, sollte aber sinnvollerweise kürzer als der Originaltabellenname sein.

Mit Joins werden die Selects in horizontaler Richtung erweitert (mehr spalten). Natürlich kann man die Selects auch in vertikaler Richtung erweitern, mit UNIONS:

```
SELECT * FROM Adresse WHERE Name < 'B'
UNION
SELECT * FROM Adresse WHERE Name >'Y';
```

Mit dieser Anweisung werden alle Adressen ausgegeben deren Name mit „A" oder mit „Z" beginnt. Natürlich ist diese Anweisung identisch mit

```
SELECT * FROM Adresse WHERE Name < 'B' OR Name >'Y';
```

aber mir ist im Moment nichts besseres eingefallen. Ein Union funktioniert dann, wenn alle Spalten die selektiert werden sollen vom gleichen Typ sind. Das bedeutet, wenn es sich um unterschiedliche Typen handelt, muss ein Typcasting gemacht werden:

```
SELECT CAST (Name AS TStr100) From Adresse;
```

Wenn es um eine geordnete Ausgabe geht gibt es zwei Möglichkeiten: GROUP BY oder ORDER BY. Mit ORDER BY werden Sortierungen vorgenommen:

```
SELECT * FROM Adresse ORDER BY Name;
```

gibt alle Adresse sortiert nach den Namen aus. Mit DESC kann wieder eine absteigende Liste erzeugt werden:

```
SELECT * FROM Adresse ORDER BY DESC Name;
```

Order by kann natürlich auch mehrere Spalten beinhalten. Dann wird immer nach der ersten Spalte sortiert, die weiteren Spalten stellen die Reihenfolge fest, wenn es identische Werte in der Hauptspalte der Sortierung gibt.

Group By wird hauptsächlich bei Aggregatfunktionen wie SUM(), AVG(),... eingesetzt:

```
SELECT SUM(Preis), Name From Bestellung JOIN Adresse ON
Adresse_ID=Adresse.ID GROUP BY Adresse.Name;
```

Dieses Statement gibt eine Liste aller Name aus zu denen Bestellungen existieren und zeigt die Summe des Preises zu jedem Namen an. Bei Group By muss beachtet werden, dass alle Spalten die bei Select ausgewählt werden auch in Group By erscheinen:

```
SELECT SUM(Preis), Name, Vorname From Bestellung JOIN Adresse
ON Adresse_ID=Adresse.ID GROUP BY Name, Vorname;
```

Mit den Funktionen MAX() und MIN() wird das Maximum bzw. Minimum einer Spalte bestimmt und mit COUNT() wird die Anzahl der Datensätze ermittelt:

```
SELECT COUNT(*) FROM Adresse;
```

ist nicht identisch mit

```
SELECT COUNT(Name) from Adresse;
```

Im ersten Fall werde alle Datensätze gezählt, im zweiten Fall nur diese Datensätze bei denen gilt (Name <> NULL).

Das letzte:
Wieder schließe ich ein Tutorial mit dem Satz: Wer mehr wissen will muss üben, Bücher lesen und nochmal üben. Mit dem Wissen aus dem Tutorial lässt sich schon einiges anfangen. Wer es noch nicht getan hat: Ich empfehle jedem das Handbuch von Interbase/Firebird runterzuladen (knapp 10 MB). Selbst mit wenigen Englischkentnissen (wie ich) kann man da sehr viele Tipps und Hilfe finden.

Anregungen, Fragen sowie Verbesserungsvorschläge an lemmy@delphi-tutorials.de

Quellen:

Interbase Handbuch
InterBase Datenbankentwicklung mit Delphi, Andreas Kosch im Software und Support Verlag, ISBN 3-935042-09-4