Skript zur Vorlesung

Datenbanken und Informationssysteme (Datenbankprogrammierung)

Fachhochschule Stralsund
Fachbereich Elektrotechnik & Informatik

Prof. Dr. Uwe Hartmann

Gliederung DB&IS (Datenbankprogrammierung)

1 Eigebettetes SQL

1.1 Statisches eingebettetes SQL (sESQL)

1.1.1 Bestandteile eines statischen ESQL-Programms
1.1.2 Datenbankverbindung herstellen und beenden
1.1.3 Host-Variablen
1.1.4 Einzeilen-SELECT
1.1.5 Cursor-Konzept
1.1.6 Positioniertes UPDATE/DELETE
1.1.7 Transaktion
1.1.8 Fehlerbehandlung und Rückmeldungen
1.1.9 Erstellen einer Anwendung in DB2

1.2 Dynamisches eingebettetes SQL (dESQL)

1.2.1 Dynamischens Übersetzen und Ausführen einer SQL-Anweisung
1.2.2 Dynamisches Cursor-Konzept
1.2.3 Deskriptor SQLDA

2 Call Level Interface (CLI)

2.1 Initialisierungs-/Terminierungsrahmen

2.2 Datenbankzugriff

2.2.1 Überblick
2.2.2 Zugriff mit Parametermarkern
2.2.3 Tupelweise Ein-/Ausgabe
2.2.4 Ein-/Ausgabe von Tupelmengen (arrays)
2.2.5 Behandlung unbekannter Anfagen
2.2.6 Transaktion

3 Externe Datenbankroutinen

3.1 Externe Funktionen

3.1.1 Externe skalare Funktionen
3.1.1.1 Definition einer externen skalaren Funktion
3.1.1.2 Anmelden einer externen skalaren Funktion
3.1.2 Externe Tabellenfunktionen
3.1.2.1 Tabellenausdrücke
3.1.2.2 Definition einer externen Tabellenfunktion
3.1.2.3 Anmelden einer externen Tabellenfunktion

3.2 Externe Prozeduren

3.2.1 Definition einer externen Prozedur
3.2.2 Anmelden einer externen Prozedur
3.2.3 Aufruf einer externen Prozedur

4 SQL-Routinen

4.1 Prozedurale Erweiterungen von SQL

4.1.1 SET-Variable-Anweisung
4.1.2 IF-Anweisung
4.1.3 CASE-Anweisung
4.1.4 WHILE-Anweisung
4.1.5 REPEAT-Anweisung
4.1.6 LOOP-Anweisung mit LEAVE und ITERATE
4.1.7 FOR-Anweisung
4.1.8 Variablen-Deklaration
4.1.9 Condition-Deklaration
4.1.10 Handler-Deklaration
4.1.11 SIGNAL- und RESIGNAL-Anweisung
4.1.12 RETURN-Anweisung

4.2 SQL-Funktionen

4.2.1 Definition einer SQL-Funktion
4.2.2 Verbundanweisung für SQL-Funktionen

4.3 SQL-Prozeduren

4.3.1 Definition einer SQL-Prozedur
4.3.2 Verbundanweisung für SQL-Prozeduren
4.3.3 Übersetzen, Binden, Anmelden von SQL-Routine
4.3.4 Aufruf einer SQL-Prozedur

Vorbemerkungen

Literatur

Can Türker
"SQL: 1999 & SQL: 2003", dpunkt.verlag

Don Chamberlin
"DB2 Universal Database", Addison-Wesley

"SQL Reference Volume 1", IBM Corp.

"SQL Reference Volume 2", IBM Corp.

"Application Development Guide: Building and Running Applications", IBM Corp.

"Application Development Guide: Programming Client Applications", IBM Corp.

"Application Development Guide: Programming Server Applications", IBM Corp.

Syntaxnotation für die Definition von SQL-Statements

Symbol	Bedeutung
GROSSBUCHSTABEN	Schlüsselworte der Sprache, Terminale, z.B. SELECT
kleinbuchstaben	Syntaktische Elemente, Nicht-Terminale, z.B. query-expression
::=	Definitionsoperator, z.B. bei *x* ::= *y* wird *x* durch *y* definiert
|	Alternative, z.B. bei *x* | *y* | *z* muß *x* oder *y* oder *z* angegeben werden.
[ ]	Option, z.B. bei [*x*] muß *x* nicht angegeben werden
{ }	Gruppierung, z.B. bei { *x* | *y* } muß *x* oder *y* angegeben werden
{ }...	Wiederholung, z.B. bei { *x* }... wird *x* ein oder mehrmals wiederholt

C/C++-Namen nicht kursiv (char, struct, ...)

Verwendete Datenbank

- Maler (K)

KNR	Name	Geburt	Geb_Ort	Tod
K1	Lorrain	1734	Paris	1812
K2	Endler	1931	München	-
K3	Veccio	1480	Paris	1528
...	...	...	...	...

- Museum (M)

ENR	Bezeichnung	Sitz
E1	Louvre	Paris
E3	Nationalmuseum	Kopenhagen
...	...	...

- Galerie (G)

ENR	Bezeichnung	Sitz
E2	National Gallery	London
E5	Le Corbet	Paris
...	...	...

- Bild (B)

BNR	Titel	Jahr	Wert	KNR	ENR
B1	Landschaft mit Pfeilen	1965	150	K2	E2
B2	Wasserlandschaft	1946	200	K5	E6
B8	Maria Magdalena	1422	1000	-	E5
...	...	...	...	...	...

KNR: Maler des Bildes, ENR: Besitzer des Bildes

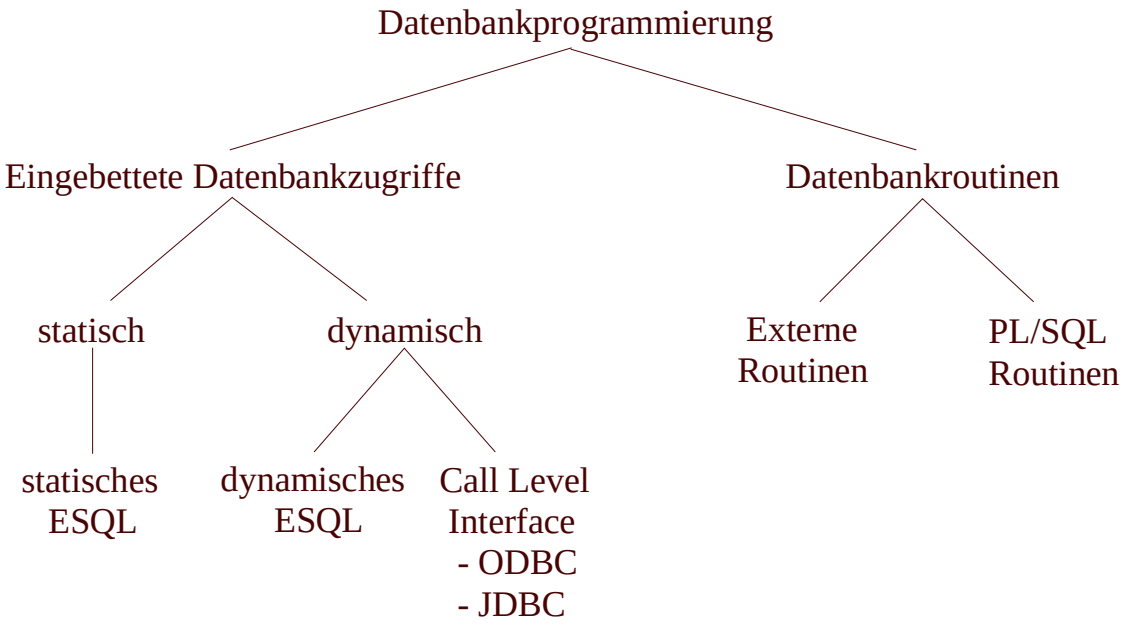
- Ausstellung (A)

ANR	A_Titel	Beginn	ENR	BNR
A1	Portraits des 18. Jahrhunderts	1997	E3	B4
A1	Portraits des 18. Jahrhunderts	1997	E3	B14
A2	Poststrukturalismus	1994	E7	B1
...	...	...	...	...

ENR: Einrichtung, die Ausstellung durchführt, BNR: ausgestelltes Bild

Datenbankprogrammierung

- Bislang haben wir uns ausschließlich mit der interaktiven Anwendung von SQL-Anweisungen beschäftigt. Wir haben einen Befehl in das Datenbanksystem eingegeben und haben eine Ergebnis erhalten. Nun wollen weitere Formen des Datenbankzugriffs kennen-lernen:



1. Zugriff auf Datenbank von einer Programmiersprache aus.
   - Damit eröffnet sich uns eine Möglichkeit, Datenbanksysteme in umfassende und ggf. bereits existierende Informationssysteme zu integrieren.
   - Wir unterscheiden dabei statischen und dynamischen Datenbankzugriff
   - Statisch: Zugriffsplan bereits zur Compilationszeit
     --> Folge: DB-Anweisung steht beieits bei der Compilierung fest
   - Dynamisch: Zugriffsplan erst zur Laufzeit
     --> Folge: DB-Anweisung kann zur Laufzeit erfolgen

2. Erweiterung der Datenbanksprache um nutzerdefinierte Zugiffsroutinen
   - 2 Formen:
   - In einer höheren Programmiersprache geschriebene Routinen
   - Routinen durch prozedurale Erweiterung von SQL

1. Eingebettetes QSL

Aufbau eines ESQL-Programms und Übersetzungsprozeß

- Konventionen der File-Extension für ESQL-Pogramme (in Winows für DB2):
  - .sqc - ESQL-C-Programm
  - .sqx - ESQL-C++-Programm

- Einbinden der Datei sqlenv.h
  - enthält Strukturdeklarationen, Konstanten und Funktionsdeklarationen der vom Pre-compiler erzeugten Funktionsaufrufe für Zugriff auf Datenbank

- Bestandteile eines ESQL-Programms:
  - Anweisungen der Wirtssprache (Präcompiler, Deklarationen, Statements)
  - embedded-SQL-Anweisungen

- SQL-Anweiungen werden zur Unterscheidung von den Anweisungen der Host-Sprache in C durch den Präfix ‘EXEC SQL’ gekennzeichnet und mit einem Semikolon abgeschlossen. Sie können an allen Stellen eines C-Programms auftreten, an denen auch C-Statements möglich sind.

Beispiel

Insert eines Tupels in die Maler-Relation

insert.sqc

```
#include <sqlenv.h>
main()
{
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL CONNECT TO "Kunst_DB";
    EXEC SQL INSERT INTO Maler
                    VALUES ('K33', 'Penck', 1939, 'Dresden', NULL);
    EXEC SQL CONNECT RESET;
}
```

Beispiel

Ergebnis des ESQL-Precompilers in DB2

insert.c

```
#include "sqladef.h"
static char sqla_program_id[40] = {111,65,65,66,65, ... , 50,32,32};
static struct sqla_runtime_info sqla_rtinfo =
          {{'S','Q','L','A','R','T','I','N'}, sizeof(wchar_t), 0, {' ',' ',' ',' '}};
#include <sqlenv.h>
main()
{
// EXEC SQL INCLUDE SQLCA;
    #include "sqlca.h"
    struct sqlca sqlca;

// EXEC SQL CONNECT TO "Kunst_DB";
    {
     sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
     sqlaaloc(2,1,1,0L);
```

```
        {
            struct sqla_setd_list sql_setdlist[1];
            sql_setdlist[0].sqltype = 460;
            sql_setdlist[0].sqllen = 9;
            sql_setdlist[0].sqldata = (void*)"Kunst_DB";
            sql_setdlist[0].sqlind = 0L;
            sqlasetd(2,0,1,sql_setdlist,0L);
         }
         sqlacall((unsigned short)29,4,2,0,0L);
         sqlastop(0L);
        }

    // EXEC SQL Insert into maler values ('K30', 'Penck', 1939,   'Dresden', NULL);
        {
         sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
         sqlacall((unsigned short)24,1,0,0,0L);
         sqlastop(0L);
        }

    // EXEC SQL CONNECT RESET;
        {
         sqlastrt(sqla_program_id, &sqla_rtinfo, &sqlca);
         sqlacall((unsigned short)29,3,0,0,0L);
         sqlastop(0L);
        }
    }
```

1.1 Statisches eingebettetes SQL (sESQL)

- statische SQL-Anweisungen innerhalb eines Programms werden zur Compilezeit übersetzt und ein Zugriffsplan angelegt
- Zur Laufzeit sind sie nicht mehr änderbar

1.1.1 Bestandteile eines statischen ESQL-Programms

- ESQL-Programm besteht aus:
  - Präprozessor-Anweisungen der Host-Sprache
  - Deklarationen der Host-Sprache
  - Statements der Host-Sprache
  - ESQL-Preprozessor-Anweisungen (embedded-sql-include)
  - ESQL-Deklarationen (embedded-sql-declare-section)
  - ESQL-Statements (embedded-sql-statement)

ESQL-Preprozessor-Anweisungen

- Zweck: Einbinden eigener header-files in ein ESQL-Programm

Syntax

embedded-sql-include ::=

EXEC SQL INCLUDE { header-file | SQLCA } ;

ESQL-Deklarationen

- Zweck: Definition von Variablen, über die Informationen zwischen Programm und Datenbank ausgetauscht werden – Host-Variable

ESQL-Statements

- eigentliche SQL-Anweisungen, durch die ein Programm mit einer Datenbank kommuniziert

Syntax

embedded-sql-statement ::=

EXEC SQL statement-or-declaration ;

Syntax

statement-or-declaration ::=

SQL-procedure-statement

| *embedded-exeption-declaration*

| *declare-cursor*

| *dynamic-declare-cursor*

Syntax

SQL-procedure-statement ::=

SQL-connection-statement

| *SQL-data-statement*

| *SQL-transaction-statement*

| *SQL-dynamic-statement*

| *SQL-dynamic-data-statement*

| *SQL-schema-statement*

Syntax

SQL-data-statement ::=

insert-statement

| *delete-statement*

| *update-statement*

| *select-statement-single-row*

| *open-statement*

| *fetch-statement*

| *close-statement*

| *delete-statement-positioned*

| *update-statement-positioned*

1.1.2 Datenbankverbindung herstellen und beenden

Syntax

SQL-connection-statement ::=

CONNECT TO { db-name | host-variable }

[IN SHARE MODE | IN EXCLUSIVE MODE]

[USER { username | host-variable }]

[USING { passwd | host-variable }]

| *CONNECT RESET*

- SHARE MODE: andere Nutzer (Programme) können gleichzeitig Datenbankverbindung aufbauen, voreingestellt
- EXCLUSIVE MODE: verhindert, daß andere Nutzer ebenfalls Verbindung zur Datenbank aufbauen, solange das Programm verbunden ist
- USER/USING: optionale Nutzer/Password-Angabe
- DISCONNECT: hebt Datenbankverbindung zur einer bestimmten DB auf

1.1.3 Host-Variablen

Syntax

host-variable ::=

: variable-name [[INDICATOR] : variable-name]

- Hostvariablen übergeben/übernehmen Werte an/von eingebetteten SQL-Anweisungen

SQL-Datentyp	***C-Datentyp***
SMALLINT	short
INTEGER	long
DECIMAL(p,s)	(kein C Äquivalent)
REAL	float
DOUBLE	double
CHAR(n)	char[n+1] (durch Null zu beenden)

VARCHAR(n)	char[n+1] (durch Null zu beenden)
	oder
	struct
	{
	short length;
	char data[n];
	}

DATE	char[11]
TIME	char[9]
TIMESTAMP	char[27]
Indikatorvariable	short

Indikator-wert	***Semantik***
-1	Die Hostvariable hat den Wert NULL
0	Die Hostvariable hat einen realen Wert (nicht NULL)
>0	Die Hostvariable hat einen realen Wert, der zum Zwecke der Längenanpassung abgeschnitten wurde.

- -1 : Wert von variable-name ist beliebig und braucht nicht ausgewertet zu werden
- 0 : variable-name hat den eigentlichen Wert
- >0 : Der Indikatorwert gibt die ursprüngliche Länge vor dem Runden an. Für Eingabevariablen ist ein positiver Indikatorwert dem Wert 0 gleichbedeutend.

Syntax

embedded-sql-declare-section ::=

EXEC SQL BEGIN DECLARE SECTION ;

host-variable-definition [{ , host-variable-definition }...]

EXEC SQL END DECLARE SECTION ;

Beispiel

Wertsteigerung eines Bildes um einen prozentualen Anteil, deren Maler vor einem bestimmten Datum geboren wurden

```
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  dbname[20]="Kunst_DB";
        short steigerung;
        short geburt;
    EXEC SQL END DECLARE SECTION;

    printf("Wertsteigerung: ");
    scanf("%d", &steigerung);
    printf("Geburt vor: ");
    scanf("%d", &geburt);

    EXEC SQL CONNECT TO :dbname;

    EXEC SQL UPDATE Bild
        SET Wert = Wert + Wert * :steigerung / 100
        WHERE KNR IN
          (SELECT KNR
           FROM Maler
           WHERE geburt < : geburt);

    EXEC SQL CONNECT RESET;
}
```

Beispiel

Änderung des Besitzers eines Bildes, dabei kann auch festgelegt werden, daß der Besitzer unbekannt ist

```
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  bnr[4];
        char  enr[4];
        short enr_ind=0;
```

```
    EXEC SQL END DECLARE SECTION;

    printf("BNR    :  ");
    scanf("%s", bnr);
    printf("Besitzer:  ");
    scanf("%s", enr);
    if(*enr == '*')
        enr_ind = -1;

    EXEC SQL CONNECT TO "Kunst_DB";

    EXEC SQL UPDATE Bild
         SET ENR = :enr :enr_ind
         WHERE BNR = :bnr;

    EXEC SQL CONNECT RESET;
}
```

1.1.4 Einzeilen-SELECT

Syntax

select-statement-single-row ::=

SELECT [ALL | DISTINCT] select-list

INTO host-variable [{ , host-variable }...]

FROM table-reference [{ , table-reference }...]

[WHERE search-condition]

Beispiel

Ausgabe des Maler-Tupels für eine eingegebene KNR

```
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  knr[4];
        char  name[21];
        short geburt;
        short tod;
        short tod_ind=0;
    EXEC SQL END DECLARE SECTION;

    printf("KNR :  ");
    scanf("%s", knr);

    EXEC SQL CONNECT TO "Kunst_DB";

    EXEC SQL SELECT Name, Geburt, Tod
         INTO :name, :geburt, :tod :tod_ind
         FROM Maler
         WHERE KNR = :knr;

    printf("Name:  %s\n", name);
    printf("geb :  %d\n", geburt);
    if(!tod_ind)
        printf("gest:  %d\n", tod);

    EXEC SQL CONNECT RESET;
}
```

1.1.5 Cursor-Konzept

- Cursor: Zeiger auf ein aktuelles Ergebnistupel einer SELECT-Anweisung --> Eintupel-schnittstelle

- Schritte beim Zugriff auf die Ergebnistupel einer SELECT-Anweisung innerhalb eines ESQL-Programms:
  1. Cursor für eine SELECT-Anweisung definieren (DECLARE)
  2. Cursor öffnen (OPEN)
  3. solange ein (weiteres) Ergebnistupel existiert

  Tupel über Hostvariable aus Ergebnispuffer holen, Cursor auf nächstes Tupel stellen (FETCH)
  4. Cursor schließen (CLOSE)

Cursor erzeugen

Syntax

declare-cursor ::=

DECLARE cursor-name CURSOR [WITH HOLD] FOR cursor-specification

cursor-specification ::=

query-expression [updatability-clause]

updatability-clause ::=

FOR READ ONLY

| FOR UPDATE [OF column-name [{ , colunm-name }...]]

- Definiert einen Cursor (name) für eine Anfrage (query-expression: SELECT-Anweisung)
- WITH HOLD: Cursor bleibt nach einer COMMIT-Anweisung geöffnet, d.h. er zeigt dann auf das nächste Tupel, sonst wird der Cursor geschlossen
- FOR READ ONLY: Cursor ausschließlich zum Lesen von Tupeln
- FOR UPDATE: Cursor für positioniertes UPDATE/DELETE

Cursor öffnen

Syntax

open-statement ::=

OPEN cursor-name

- bewirkt die Ausführung der SELECT-Anweisung, für die der Cursor definiert wurde.
- nach dem OPEN zeigt der Cursor *vor* das erste Tupel (Cursorposition)

Tupel holen / Cursor weiterstellen

Syntax

fetch-statement ::=

FETCH cursor-name INTO host-variable [{ , host-variable }...]

- liest Attributwerte eines Tupels in die Host-Variablen der Anfrage ein und positioniert den Cursor auf das folgenden Tupel
- **SQLCODE**
  - über Struktur SQLCA definiert
  - *Folgt* auf die aktuelle Cursorposition kein weiteres Tupel, ist SQLCODE=100, sonst <>100

Beispiel

SELECT-Anweisung, die zwei Tupel (mit den Attributwerten a1, a2) als Ergebnis hat, wird mit Hilfe des Cursors ausgewertet

```
EXEC SQL DECLARE curs CURSOR FOR
    SELECT A
    FROM R;

EXEC SQL OPEN curs;

while(1)
{
    EXEC SQL FETCH curs INTO :a
    if(SQLCODE == 100) break;
    // hier Auswertung des aktuellen Tupels
}
```

Cursor schließen

Syntax

close-statement ::=

CLOSE cursor-name

- Schließt Cursor und gibt Ressourcen (temporäre Ergebnismenge) frei

Beispiel

Select mit Cursor

```
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  knr[4];
        char  name[21];
        char  titel[31];
        short titel_ind=0;
        short jahr;
        short jahr_ind=0;
    EXEC SQL END DECLARE SECTION;

    printf("KNR :  ");
    scanf("%s", knr);

    EXEC SQL DECLARE cur CURSOR FOR
        SELECT x.Name, y.Titel, y.Jahr
        FROM Maler x LEFT JOIN Bild y ON x.KNR=y.KNR
        WHERE x.KNR = :knr;

    EXEC SQL CONNECT TO "Kunst_DB";

    EXEC SQL OPEN cur;

    while(1)
    {
        EXEC SQL FETCH cur INTO :name, :titel :titel_ind, :jahr :jahr_ind;
        if(SQLCODE==100) break;
        printf("Name: %s\n", name);
        if(!titel_ind)
        {
            printf("\tTitel: %s\n", titel);
            if(!jahr_ind)
                printf("\tJahr : %d\n", jahr);
            else
                printf("\tJahr : unbekannt\n");
        }
        else
            printf("\tkeine Bilder\n");
    }

    EXEC SQL CLOSE cur;
    EXEC SQL CONNECT RESET;
}
```

1.1.6 Positioniertes UPDATE/DELETE

Syntax

update-statement-positioned ::=

update-statement WHERE CURRENT OF cursor-name

Syntax

delete-statement-positioned ::=

delete-statement WHERE CURRENT OF cursor-name

- andert/löscht das Tupel, auf das ein Cursor aktuell zeigt
- Einschränkungen für die SELECT_Anweisung der Cursor-Deklaration:
  1. table-reference (FROM-Klausel) darf nur eine Relation oder Sicht enthalten
  2. select-list darf keine Aggregatfunktion enthalten
  3. kein DISTINCT, GROUP BY, HAVING

Beispiel

Ausgabe eines jeden Tupels aus Bild mit der Frage, ob das Tupel gelöscht werden soll. Wenn ja: positioniertes Löschen

```
EXEC SQL INCLUDE SQLCA;
main()
{
   EXEC SQL BEGIN DECLARE SECTION;
      char  bnr[4];
      char  titel[31];
      short jahr;
      short jahr_ind=0;
   EXEC SQL END DECLARE SECTION;

   char antwort;

   EXEC SQL DECLARE cur CURSOR FOR
      SELECT BNR, Titel, Jahr
      FROM Bild
      FOR UPDATE;

   EXEC SQL CONNECT TO "Kunst_DB";

   EXEC SQL OPEN cur;

   while(1)
   {
      EXEC SQL FETCH cur INTO :bnr, :titel, :jahr :jahr_ind;
      if(SQLCODE==100) break;
```

```
            printf("\n%s\t%-32s", bnr, titel);
            if(!jahr_ind)
                printf("\t%d", jahr);
            printf("\n==>Loeschen? j/n:");
            scanf("%c", &antwort);
            flushall();
            if(antwort=='j')
                EXEC SQL DELETE FROM Bild WHERE CURRENT OF cur;
        }

        EXEC SQL CLOSE cur;
        EXEC SQL CONNECT RESET;
    }
```

1.1.7 Transaktion

Syntax

SQL-transaction-statement ::=

COMMIT [WORK]

| *ROLLBACK [WORK]*

| *compound-statement*

- COMMIT: alle Datenbankaktionen vom letzten COMMIT/ROLLBACK an bis zum aktuellen Datenbankzustand werden physisch festgeschrieben
- ROLLBACK: alle Datenbankaktionen vom letzten COMMIT/ROLLBACK an bis zum aktuellen Datenbankzustand werden zurückgesetzt

Beispiel

```
Beginn des Programms
    statement_1
    statement_2
EXEC SQL COMMIT;
    statement_3
    statement_4
EXEC SQL ROLLBACK;
    statement_5
    statement_6
Ende des Programms
```

- COMMIT und ROLLBACK schließen alle offenen Cursoren, es sei denn, sie sind mit WITH HOLD definiert

Syntax

compound-statement ::=

BEGIN COMPOUND { ATOMIC | NOT ATOMIC } STATIC

SQL-statement ; [{ SQL-statement ; }...]

END COMPOUND

- zwischen BEGIN und END enthaltene Anweisungen werden als Block an den Server gesendet, sog. zusammengesetzte Anweisungen.
- ATOMIC: falls eine Anweisung fehlschlägt, werden alle anderen zurückgesetzt
- Cursor-Anweisungen innerhalb zusammengesetzter Anweisungen nicht möglich
- innerhalb der zusammengesetzten Anweisung ist ein COMMIT als letzte Anweisung möglich, ROLLBACK nicht

Beispiel

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
    DELETE FROM Maler
    WHERE KNR = :knr;
    UPDATE Bild
    SET KNR = NULL
    WHERE KNR = :knr;
END COMPOUND;
```

1.1.8 Fehlerbehandlung und Rückmeldungen

SQLCA

- SQLCA: SQL Communication Area
- in sqlca.h definiert
- mit EXEC SQL INCLUDE SQLCA in das ESQL-Programm eingebunden:

```
struct  sqlca
{
    unsigned char       sqlcaid[8];     //  Eyecatcher = 'SQLCA   '
    long                sqlcabc;        //  SQLCA size in bytes = 136
    long                sqlcode;        //  SQL return code
    short               sqlerrml;       //  Length for SQLERRMC
    unsigned char       sqlerrmc[70];   //  Error message tokens
    unsigned char       sqlerrp[8];     //  Diagnostic information
    long                sqlerrd[6];     //  Diagnostic information
    unsigned char       sqlwarn[11];    //  Warning flags
    unsigned char       sqlstate[5];        //  State corresponding to SQLCODE
};
```

- Statusanzeigen im SQLCODE:

SQLCODE Wert	*Semantik*
0	SQL-Anweisung erfolgreich ist sqlwarn[0] == 'W', dann mindestens eine Warnung
100	SQL-Anweisung erfolgreich, aber keine Daten gefunden
<0	Fehler, SQL-Anweisung nicht ausgeführt, Fehlercode in sqlerrm oder sqlstat

WHENEVER-Anweisung

Syntax

embedded-exeption-declaration ::=

WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action

action ::=

{ CONTINUE | GO TO label }

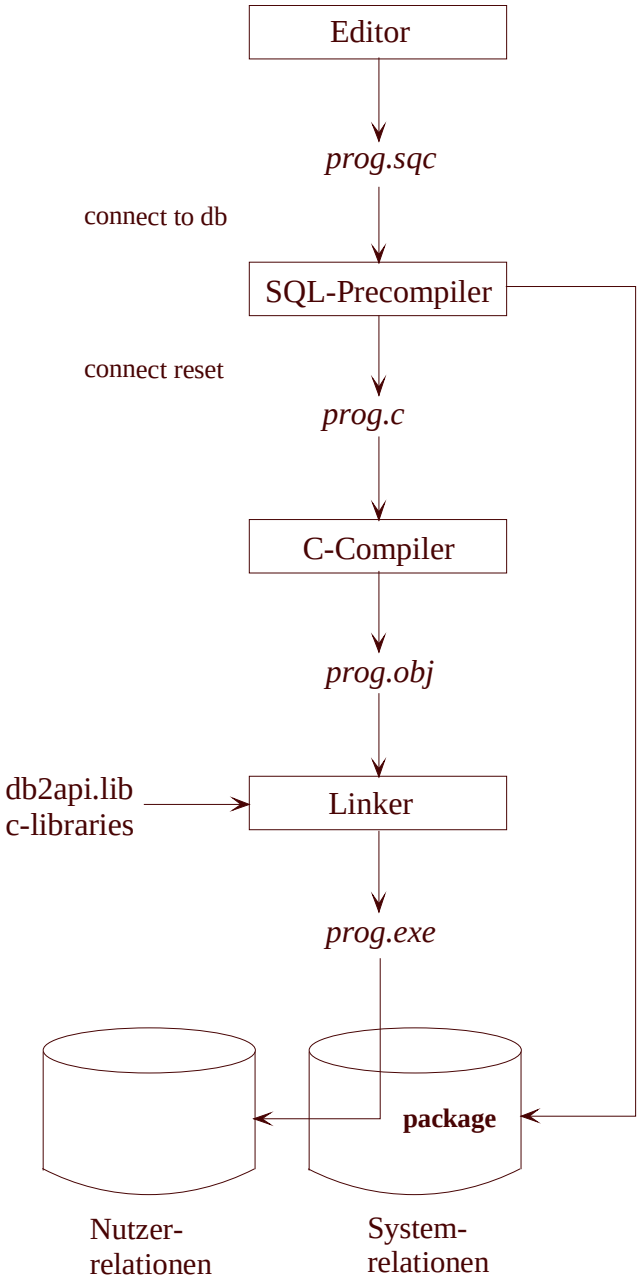
- NOT FOUND: wenn SQLCODE = 100 wird action ausgeführt

- SQLERROR: wenn SQLCODE < 0 wird action ausgeführt
- SQLWARNING: wenn SQLCODE = 0 und sqlwarn[0] = 'W' wird action ausgeführt
- CONTINUE: keine Reaktion (default)
- GO TO label: Programm springt zur Marke (schlechte Programmierung)

Beispiel

```
...
    EXEC SQL WHENEVER SQLWARNING CONINUE;
    EXEC SQL COMMIT;

// 1. Transaktion
    EXEC SQLWHENEVER SQLERROR GO TO fehler_1;
    statement_1
    statement_2
    statement_3
    EXEC SQL COMMIT;
fehler_1:
    EXEC SQL ROLLBACK;

// 2. Transaktion
    EXEC SQLWHENEVER SQLERROR GO TO fehler_2;
    statement_4
    statement_5
    EXEC SQL COMMIT;
fehler_2:
    EXEC SQL ROLLBACK;
...
```

1.1.9 Erstellen einer Anwendung in DB2



Schritte von der Erstellung bis zur Ausführung

1. Quelltext editieren

2. Datenbankverbindung herstellen
   - **Befehl:** DB2 CONNECT TO database-name

3. Quelltext SQL-precompilieren/binden
   - **Befehl:** PREP quelltext-name

4. Datenbankverbindung lösen
   - **Befehl:** DB2 CONNECT RESET

5. C-Quelltext compilieren

6. Binden

7. ausführen

Batch-Datei zur Anwendungserstellung

bldsesql.bat

```
@echo off
1. Schritt: Datenbankverbindung herstellen
db2 connect to kunst_db

2. Schritt: Präcompilieren des prog_name.sqc, Ergebnis ist prog_name.c
db2 prep %1.sqc

3. Schritt: Datenbankverbindung lösen
db2 connect reset

4. Schritt: Compilieren von prog_name.c
cl -Z7 -Od -c -W2 -D_X86_=1 -DWIN32 -Id:\sqllib\include %1.c

5. Schritt: Linken von prog_name.obj
link -debug:full -debugtype:cv -out:%1.exe %1.obj d:\sqllib\lib\db2api.lib
@echo on
```

Nachträgliches Binden

1. Möglichkeit: Implizites Binden

2. Möglichkeit: Explizites Binden

- **Befehl:** REBIND quelltext-name

oder

- **Befehl:** PREP quelltext-name BINDFILE
- **Befehl:** BIND quelltext-name

1.2 Dynamisches eingebettetes SQL (sESQL)

1.2.1 Dynamischens Übersetzen und Ausführen einer SQL-Anweisung

Syntax

SQL-dynamic-statement ::=

prepare-statement

| *execute-statement*

| *execute-immediate-statement*

| *describe-statement*

EXECUTE IMMEDIATE

Syntax

execute-immediate-statement ::=

EXECUTE IMMEDIATE statement-variable

Beispiel

Insert eines Tupels in die Maler-Relation

```
#include <sqlenv.h>
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  dbname[20]="Kunst_DB";
        char  sqlstring[100]=
                "INSERT INTO Maler VALUES ('E10', 'Penck', 1939, 'Dresden', NULL)";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO :dbname;
    EXEC SQL EXECUTE IMMEDIATE :sqlstring;
    EXEC SQL CONNECT RESET;
}
```

Beispiel

Insert eines Tupels in die Maler-Relation, Eingabe der SQL-Anweisung über Tastatur

```
#include <sqlenv.h>
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  dbname[20]="Kunst_DB";
        char  sqlstring[100];
        char  errormessage[500];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR GOTO error;
    EXEC SQL CONNECT TO :dbname;
    printf("ANWEISUNG: ");
    gets(sqlstring);
    EXEC SQL EXECUTE IMMEDIATE :sqlstring;
    EXEC SQL CONNECT RESET;
    exit(1);
error:
    sqlaintp(errormessage, 500, 70, &sqlca);
    printf("FEHLER: %s\n", errormessage);
    getchar();
}
```

Parametermarker

- Parametermarker sind Platzhalter in dynamischen SQL-Anweisungen, die zur Laufzeit bei der Ausführung des Programms durch Werte ersetzt werden

Beispiel

```
INSERT INTO Maler VALUES (?,?,?,?,?)
```

Beispiel

```
DELETE FROM Maler
WHERE KNR = ?
```

PREPARE

Syntax

prepare-statement ::=

PREPARE statement-name [INTO descriptor] FROM statement-variable

- PREPARE über setzt eine SQL-Anweisung in eine ausführbare Form und erstellt dafür den Zugriffsplan

EXECUTE

Syntax

execute-statement ::=

EXECUTE statement-name [using-clause]

using-clause ::=

USING host-variable [{ , host-variable }...]

| *USING DESCRIPTOR descriptor*

- EXECUTE führt eine mit PREPARE vorbereitete SQL-Anweisung aus

Beispiel

Für eingegeben ENR wird der Sitz einer Galerie geändert, bei Eingabe von * ist Sitz unbekannt

```
#include <sqlenv.h>
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;

main()
{
   EXEC SQL BEGIN DECLARE SECTION;
      char  dbname[20]="Kunst_DB";
      char  sqlstring[100]="UPDATE Galerie SET Sitz=? WHERE ENR=?";
      char  enr[4];
      char  sitz[16];
      short sitz_ind=0;
   EXEC SQL END DECLARE SECTION;

   printf("UPDATE für ENR: "); scanf("%s", enr);
   printf("Neuer Sitz    : "); scanf("%s", sitz);
   if(*sitz=='*')
      sitz_ind=-1;

   EXEC SQL CONNECT TO :dbname;

   EXEC SQL PREPARE s1 FROM :sqlstring;
   EXEC SQL EXECUTE s1 USING :sitz :sitz_ind, :enr;
   EXEC SQL CONNECT RESET;
}
```

1.2.2 Dynamisches Cursor-Konzept

- dient wie statisches Cursorkonzept dem tupelweisen Zugriff auf die Ergebnisrelation einer SELECT-Anweisung

Cursor erzeugen

- syntaktischer Ursprung: SQL-procedure-statement aus statischem ESQL

Syntax

dynamic-declare-cursor ::=

DECLARE cursor-name CURSOR [WITH HOLD] FOR statement-name

- erzeugt einen Cursor mit dem Namen cursor-name für eine mit PREPARE vorbereitete und dort mit statement-name bezeichnete SQL-Anfrage (keine andere Anweisung als SELECT)

Syntax

SQL-dynamic-data-statement ::=

dynamic-open-statement

| *dynamic-fetch-statement*

| *dynamic-close-statement*

Cursor öffnen

Syntax

dynamic-open-statement ::=

OPEN cursor-name [using-clause]

using-clause ::=

USING host-variable [{ , host-variable }...]

| *USING DESCRIPTOR descriptor*

- führt eine mit PREPARE vorbereitete SQL-Anfage (SELECT) aus
- positioniert den zuvor mit OPEN geöffneten Cursor vor das erste Ergebnistupel
- USING-Klausel: damit werden die Parametermarker der SQL-Anfrage referenziert
  - über Host-Variable
  - über Einträge in der SQLDA-Struktur (-->später)

Tupel holen / Cursor weiterstellen

Syntax

dynamic-fetch-statement ::=

FETCH cursor-name into-using-clause

into-using-clause ::=

INTO host-variable [{ , host-variable }...]

| *USING DESCRIPTOR descriptor*

- liest Ergebnistupel einer Anfrage in
  - Host-Variablen (wie statisches SQL) oder
  - SQLDA-Struktur

  ein.

Cursor schließen

Syntax

dynamic-close-statement ::=

CLOSE cursor-name

Beispiel

Für eine eingegebene ENR werden die Titel aller Bilder, die sich im Besitz der Einrichtung befinden sowie deren Malernamen ausgegeben. Ist der Maler unbekannt, wird auch das ausgegeben

```
EXEC SQL INCLUDE SQLCA;
main()
{
   EXEC SQL BEGIN DECLARE SECTION;
      char  dbname[20]="Kunst_DB";
      char  sqlstring[100]=  "SELECT x.Titel, y.Name
                              FROM BIld x LEFT JOIN Maler y ON x.KNR=y.KNR
                              WHERE x.ENR=?";
      char  enr[4];
      char  titel[31];
      char  name[21];
      short name_ind=0;
   EXEC SQL END DECLARE SECTION;
   printf("ENR: "); scanf("%s", enr);
   EXEC SQL CONNECT TO :dbname;
   EXEC SQL PREPARE stmt FROM :sqlstring;
   EXEC SQL DECLARE cur CURSOR FOR stmt;
```

```
    EXEC SQL OPEN cur USING :enr;

    while(1)
    {
        EXEC SQL FETCH cur INTO :titel, :name :name_ind;
        if(SQLCODE==100) break;
        printf("\t%s ",titel);
        if(!name_ind)
            printf("(%s)\n",name);
        else
            printf("(unbekannt)\n");
    }

    EXEC SQL CLOSE cur;
    EXEC SQL CONNECT RESET;
    getchar();
    getchar();
}
```

1.2.3 Deskriptor SQLDA

- Ein **Deskriptor** ist ein vom Programmierer zum Zwecke das Datenaustauschs zwischen ESQL-Programm und Datenbank definierter Speicherbereich

Aufgabenbereiche eines Deskriptor:

1. Übernahme von Daten aus der Datenbank als Ergebnistupel aufgrund einer SELECT-Anweisung --> OPEN/FETCH
2. Übergabe von Daten an die Datenbank über Parametermarker in DDL/DML-Anweisungen --> EXECUTE

Schritte der Vorbereitung eines Deskriptors:

1. Deskriptor definieren
2. Stukturinformationen für Attribute einlesen
   --> PREPARE/DESCRIBE INTO descriptor
3. Speicher zur Aufnahme von Daten allokieren
   --> FETCH USING descriptor

Definition eines Deskriptors

Dekriptor-Struktur

```
struct  sqlda
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlvar   sqlvar[1];
};
```

- sqldaid[8] : 'eye catcher' 'SQLDA ' oder 'SQLDA 2 '
- sqldabc : Größe eines konkteten Deskriptors in Byte
- sqln : maximale Anzahl von sqlvar-Elementen = maximal mögliche Attribute als Ergebnis einer Anfrage bzw. Parametermarker
- sqld : konkrete Anzahl von sqlva-Elementen = Anzahl der Attribute einer Anfrage

- sqlvar[] : Für jedes Attribut der select-liste bzw. jeden Parametermarker muß ein Feldelement von sqlvar, daß den Attributwert aufnimmt, existieren

```
struct  sqlvar
{
   short           sqltype;
   short           sqllen;
   char *          sqldata;
   short *         sqlind;
   struct sqlname  sqlname;
};
```

- sqltype : Datentyp des Attributs
- sqllen : Länge des Attributwertes
- sqldata : Zeiger auf Speicher, in dem der Attributwert steht
- sqlind : Zeiger auf Speicher, in dem der Indikatorwert steht
- sqlname : Name de Attributs

```
struct sqlname
{
   short           length;
   char            data[30];
};
```

- length : Länge des Attributnamens (1..30)
- data : Namensstring

Beispiel

```
int             MaxAnzahlAttribute     = 5;
int             GroesseDeskriptor      = SQLDASIZE(MaxAnzahlAttribute);
struct sqlda*   sqlda_ptr              = (struct sqlda*) malloc(GroesseDeskriptor);

sqlda_ptr->sqln      = MaxAnzahlAttribute;
sqlda_ptr->sqldabc   = GroesseDeskriptor;
```

Strukturinformationen im Deskriptor ablegen

Syntax

describe-statement ::=

DESCRIBE statement-name INTO descriptor

- lädt Strukturinformationen der Ergebnistupel einer zuvor mit PREPARE vorbereiteten SELECT-Anweisung (statement-name) in den Deskriptor:
  - sqltype (Datentyp eines Attributs)
  - sqllen (Länge eines Attributs)
  - sqlname (Name eines Atttributs)

SQL-Datentypen (Auszug)

```
        ...
// Increment for type with null indicator
    #define SQL_TYP_NULINC          1
        ...
// VARCHAR(n) - varying length string
    #define SQL_TYP_VARCHAR         448
    #define SQL_TYP_NVARCHAR        (SQL_TYP_VARCHAR+SQL_TYP_NULINC)
        ...
// varying length string for C (null terminated string)
    #define SQL_TYP_CSTR            460
    #define SQL_TYP_NCSTR           (SQL_TYP_CSTR+SQL_TYP_NULINC)
        ...
// SMALLINT - 2-byte signed integer
    #define SQL_TYP_SMALL           500
    #define SQL_TYP_NSMALL          (SQL_TYP_SMALL+SQL_TYP_NULINC)
        ...
```

***Beispiel** (Fortführung oben)*

```
    EXEC SQL BEGIN DECLARE SECTION;
        char  sqlstring[100]="SELECT KNR, Name, Geburt, Tod FROM Maler";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL PREPARE stmt INTO :*sqlda_ptr FROM :sqlstring
```

Speicher zur Aufnahme von Daten allokieren

- mit sqldata und sqlind stehen Bereiche zur Aufnahme der zeiger auf die allokierten Daten-Speicherplätze zur Verfügung
- die Größe und der Typ der zu allokierenden Speicherplätze wird durch Auswertung der im Deskriptor nun vorhandenen Strukturinformationen (sqltype, sqllen) bestimmt

***Beispiel** (Fortführung oben)*

```
for(i=0; i<sqlda_ptr->sqld; i++)
{
    switch(sqlda_ptr->sqlvar[i].sqltype)
    {
        case SQL_TYP_SMALL:
            sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(short));
            break;
        case SQL_TYP_NSMALL:
            sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(short));
            sqlda_ptr->sqlvar[i].sqlind = (short*)malloc(sizeof(short));
```

```
            break;
        case SQL_TYP_VARCHAR:
            sqlda_ptr->sqlvar[i].sqltype = SQL_TYP_CSTR;
            sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(char)*(sqlda_ptr->sqlvar[i].sqllen + 1));
            break;
        case SQL_TYP_NVARCHAR:
            sqlda_ptr->sqlvar[i].sqltype = SQL_TYP_NCSTR;
            sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(char)*(sqlda_ptr->sqlvar[i].sqllen + 1));
            sqlda_ptr->sqlvar[i].sqlind = (short*)malloc(sizeof(short));
            break;
    }
}
```

Datentupel einlesen

Beispiel (Fortführung oben)

```
EXEC SQL DECLARE cur CURSOR FOR stmt;
EXEC SQL OPEN cur;
EXEC SQL FETCH cur USING DESCRIPTOR :*sqlda_ptr;
```

Beispiel

Auswertung beliebiger Anfragen mittels Deskriptor
Die Ergebnistupelwerte der Anfage werden durch Tabulator getrennt auf eine BS-Zeile ausgegeben

```
EXEC SQL INCLUDE SQLCA;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char   dbname[20]="Kunst_DB";
        char   sqlstring[200];
    EXEC SQL END DECLARE SECTION;
    int i;

    int     MaxAttrib = 10;
    struct sqlda     *sqlda_ptr;
    int     GrDeskr = SQLDASIZE(MaxAttrib);
    sqlda_ptr = (struct sqlda*) malloc(GrDeskr);
    sqlda_ptr->sqln = MaxAttrib;
    sqlda_ptr->sqldabc = GrDeskr;

    printf("Anfrage: "); gets(sqlstring);
    EXEC SQL CONNECT TO :dbname;

    EXEC SQL PREPARE stmt INTO :*sqlda_ptr FROM :sqlstring;

    for(i=0; i<sqlda_ptr->sqld; i++)
    {
```

```
        if(sqlda_ptr->sqlvar[i].sqltype % 2)
            sqlda_ptr->sqlvar[i].sqlind = (short*)malloc(sizeof(short));
        switch(sqlda_ptr->sqlvar[i].sqltype)
        {
            case SQL_TYP_SMALL:
            case SQL_TYP_NSMALL:
                sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(short));
                break;
            case SQL_TYP_VARCHAR:
            case SQL_TYP_NVARCHAR:
                if(sqlda_ptr->sqlvar[i].sqltype==SQL_TYP_VARCHAR)
                    sqlda_ptr->sqlvar[i].sqltype=SQL_TYP_CSTR;
                else
                    sqlda_ptr->sqlvar[i].sqltype=SQL_TYP_NCSTR;
                sqlda_ptr->sqlvar[i].sqldata = (char*)malloc(sizeof(char)*(sqlda_ptr->sqlvar[i].sqllen + 1));
                break;
        }
    }

    EXEC SQL DECLARE cur CURSOR FOR stmt;
    EXEC SQL OPEN cur;
    while(1)
    {
        EXEC SQL FETCH cur USING DESCRIPTOR :*sqlda_ptr;
        if(SQLCODE==100) break;
        for(i=0; i<sqlda_ptr->sqld; i++)
        {
            if((sqlda_ptr->sqlvar[i].sqltype % 2) && (*sqlda_ptr->sqlvar[i].sqlind))
                printf("unbekannt\t");
            else
                switch(sqlda_ptr->sqlvar[i].sqltype)
                {
                    case SQL_TYP_SMALL:
                    case SQL_TYP_NSMALL:
                        printf("%d\t", *(short*)sqlda_ptr->sqlvar[i].sqldata);
                        break;
                    case SQL_TYP_CSTR:
                    case SQL_TYP_NCSTR:
                        printf("%s\t", sqlda_ptr->sqlvar[i].sqldata);
                        break;
                }
        }
        printf("\n");
    }
    EXEC SQL CLOSE cur;

    getchar();
    EXEC SQL CONNECT RESET;
}
```

Feststellung des erforderlichen Deskriptor-Speicherplatzes

Beispiel

Ergänzung zum obigen Beispiel mit SELECT KNR, Name, Geburt, Tod FROM Maler

```
...
    int     MaxAnzahlAttribute = 1;
    int     GroesseDeskriptor = SQLDASIZE(MaxAnzahlAttribute);
    struct sqlda*    sqlda_ptr = (struct sqlda*) malloc(GroesseDeskriptor);
    sqlda_ptr->sqln = MaxAnzahlAttribute;
    sqlda_ptr->sqldabc = GroesseDeskriptor;
    EXEC SQL PREPARE stmt INTO :*sqlda_ptr FROM :sqlstring;
//1-->
    if(SQLCODE == 236)
    {
        MaxAnzahlAttribute = sqlda_ptr->sqld;
        int     GroesseDeskriptor = SQLDASIZE(MaxAnzahlAttribute);
        free(sqlda_ptr);
        struct sqlda*    sqlda_ptr = (struct sqlda*) malloc(GroesseDeskriptor);
        sqlda_ptr->sqln = MaxAnzahlAttribute;
        sqlda_ptr->sqldabc = GroesseDeskriptor;
        EXEC SQL DESCRIBE stmt INTO :*sqlda_ptr;
    }
//2-->
...
```

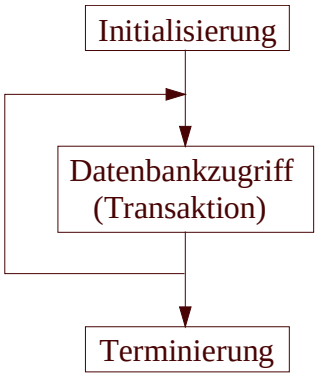
2 Call Level Interface (CLI)

- **Vorteile gegenüber dem eingebetteten dynamischen SQL:**

  1. kein Precompiler erforderlich
  2. Portierbarkeit
  3. Mehrfachverbindungen zur gleichen Datenbank
  4. Standardisierter Zugriff auf Systemrelationen
  5. Ein-/Ausgabe von Tupelmengen
  6. verschiebbarer Cursor
  7. automatische Typumwandlung
  8. zusammengesetzte SQL-Anweisungen
  9. gesicherte Prozeduren

2.1 Initialisierungs-/Terminierungsrahmen

- 3 grundlegende Phasen eines CLI-Programms:



Handle-Konzept

- ein Handle ist eine C-Variable (Typ: long int), die Informationen repräsentiert, die für ein Anwendungsprogramm im Hintergrund durch die CLI-Implementierung verwaltet wird

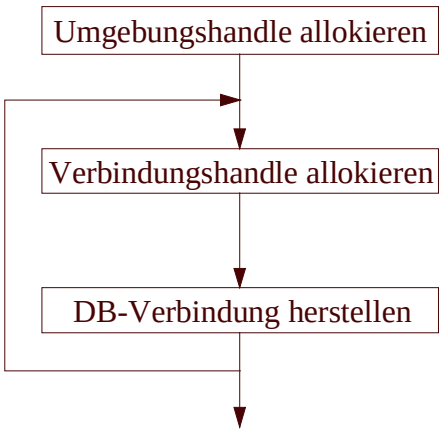
Umgebungshandle (Environment Handle)
- enthält Informationen über definierte und aktive Datenbankverbindungen (Verbindungshandles)

Verbindungshandle (Connection Handle)
- enthält Informationen über definierte Anweisungen und darüber, ob eine Transaktion aktuell offen ist

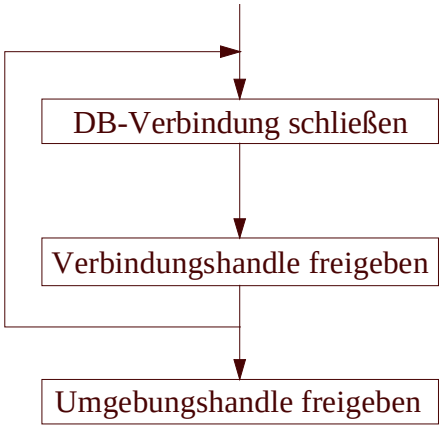
Anweisungshandle (Statement Handle)
- enthält Informationen über
  - Rückgabecodes und Nachrichten (SQLCA)
  - Datentypen und Bindungen (SQLDA)
  - aktuelle Position in einer Menge von Tupeln (Cursor)

Deskriptorhandle (Descriptor Handle)
- enthält Informationen, wie Daten zwischen einer SQL-Anweisung und einem CLI-Programm ausgetauscht werden

Initialisierung



Terminierung



Rückgabewerte von CLI-Funktionen

- Typ: SQLRETURN
- Werte:
  - SQL_SUCCESS
  - SQL_SUCCESS_WITH_INFO
  - SQL_NO_DATA _FOUND
  - SQL_ERROR
  - SQL_INVALID_HANDLE

Allokieren und Freigeben von Handles

Funktion

SQLAllocHandle (SQLSMALLINT HandleType,
SQLHANDLE InputHandle,
SQLHANDLE * HandlePtr)

Erläuterung

Erzeugt ein Umgebungs-, Verbindungs, Anweisungs- oder Deskriptorhandle

- HandleType
  legt den Typ des zu allokierenden Handles fest:
  SQL_HANDLE_ENV (Umgebungshandle)
  SQL_HANDLE_DBC (Verbindungshandle)
  SQL_HANDLE_STMT (Anweisungshandle)
  SQL_HANDLE_DESC (Deskriptorhandle)

- InputHandle
  legt den Kontext für das neue Handle fest. Für HandleType

  SQL_HANDLE_ENV : NULL-Zeiger (SQL_NULL_HANDLE)
  SQL_HANDLE_DBC : existierendes Umgebungshandle
  SQL_HANDLE_STMT : existierendes Verbindungshandle
  SQL_HANDLE_DESC : existierendes Verbindungshandle

- OutputHandlePtr
  Zeiger auf die neue Handle-Variable

Funktion

SQLFreeHandle **(** SQLSMALLINT **HandleType**,
SQLHANDLE **Handle** **)**

Erläuterung

```
gibt ein zuvor allokiertes Handle wieder frei.
```

DB-Verbindung herstellen und freigeben

Funktion

SQLConnect **(** SQLHDBC **ConnectionHandle**,
SQLCHAR * **ServerName**,
SQLSMALLINT **NameLength_1**,
SQLCHAR * **UserName**,
SQLSMALLINT **NameLength_2**,
SQLCHAR * **Password**,
SQLSMALLINT **NameLength_3** **)**

Erläuterung

stellt für ein zuvor allokiertes Verbindungshandle eine Verbindung zu einer Datenbank her

- ServerName
  Daten Quelle

- NameLength_1, NameLength_2, NameLength_3
  Länge des ServerName-Strings, UserName-Strings, Password-Strings

Funktion

SQLDisconnect (SQLHDBC ConnectionHandle)

Erläuterung

schließt eine zuvor hergestellte Verbindung

Beispiel

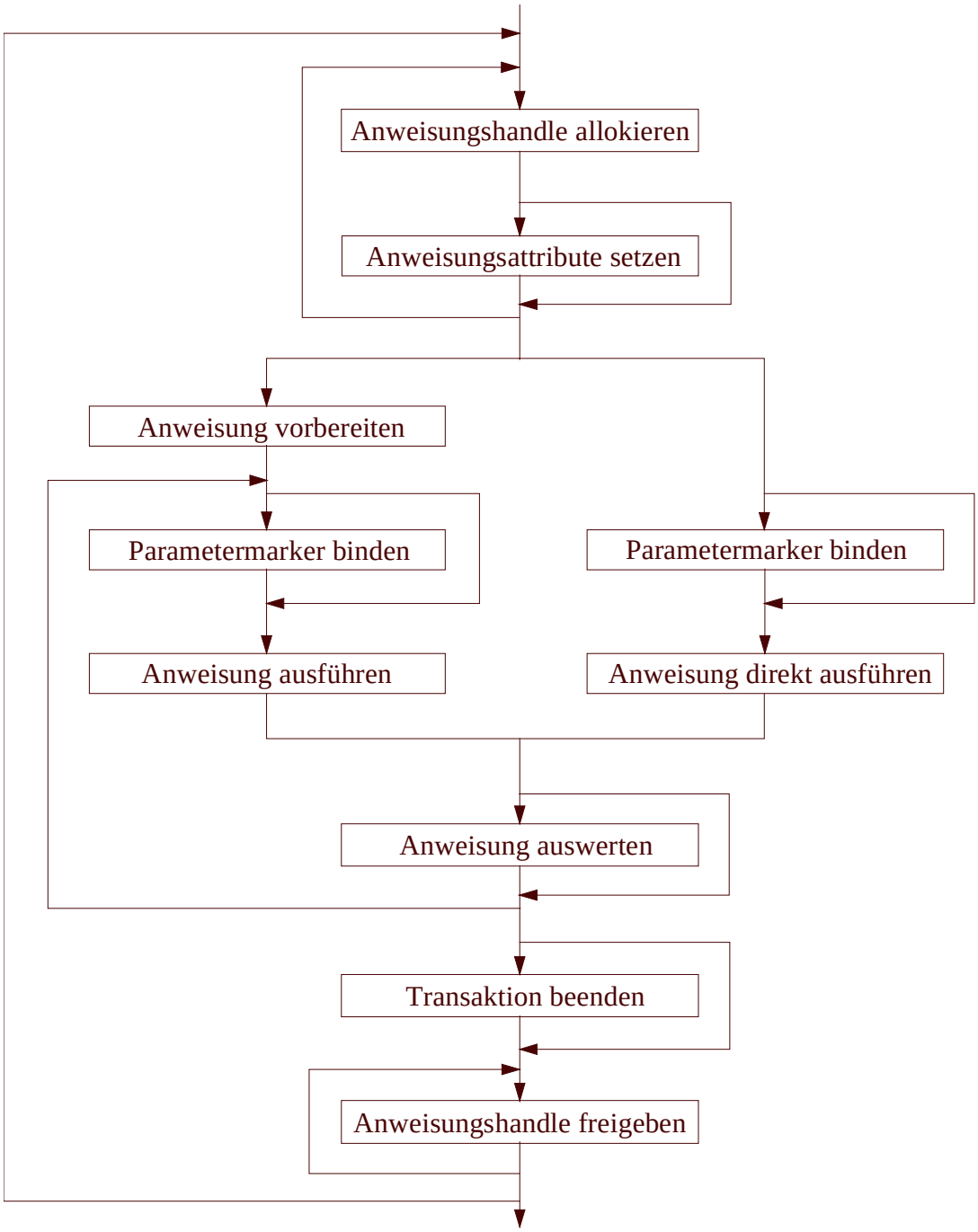
Inititialisierungs-/Terminierungsrahmen für den Datenbankzugriff

```
main()
{

    SQLHANDLE    henv;          /* environmenthandle */
    SQLHANDLE    hdbc;          /* connectionhandle */

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

    SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Nutzer1", SQL_NTS, "Passwort1", SQL_NTS);

/* DATENBANKZUGRIFFE */

    SQLDisconnect( hdbc ) ;

    SQLFreeHandle(SQL_HANDLE_DBC, hdbc) ;
    SQLFreeHandle(SQL_HANDLE_ENV, henv) ;
}
```

2.2 Datenbankzugriff

2.2.1 Überblick

Datenbankzugriff



Anweisung vorbereiten

Funktion

```
SQLPrepare ( SQLHSTMT      StatementHandle,
             SQLCHAR *     StatementText,
             SQLINTEGER    StatementTextLength  )
```

Erläuterung

es wird eine SQL-Anweisung zur Ausführung auf eine Datenbank vorbereitet

- StatementText
  SQL-Anweisung

- StatementTextLength
  Länge des SQL-Anweisungstrings

Anweisung ausführen

Funktion

```
SQLExecute ( SQLHSTMT      StatementHandle  )
```

Erläuterung

führt eine zuvor vorbereitete SQL-Anweisung aus.

Anweisung direkt ausführen

Funktion

```
SQLExecDirect ( SQLHSTMT      StatementHandle,
                SQLCHAR *     StatementText,
                SQLINTEGER    StatementTextLength )
```

Erläuterung

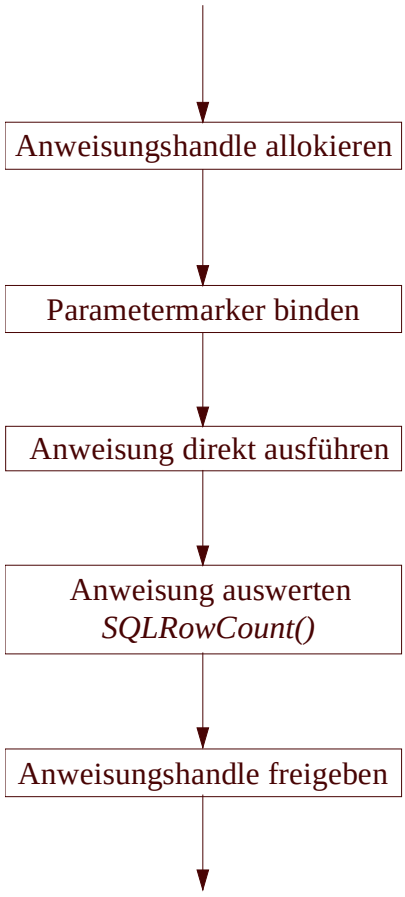
führt eine (nicht vorbereitete) SQL-Anweisung direkt aus

Beispiel

Erzeugen einer Relationenstruktur

```
#include <sqlcli1.h>
main()
{
   SQLHANDLE    henv;
   SQLHANDLE    hdbc;
   SQLHSTMT     hstmt;
   SQLRETURN    rc;

   SQLCHAR sql_stmt[] = "CREATE TABLE Atelier
                              (KNR VARCHAR(3), Ort VARCHAR(20))";

   rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   if (rc != SQL_SUCCESS) printf("ERROR: SQLAllocHandle(henv)");

   rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
   if (rc != SQL_SUCCESS) printf("ERROR: SQLAllocHandle(hdbc)");

   rc = SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz",
                        SQL_NTS);
   if (rc != SQL_SUCCESS) printf("ERROR: SQLConnect()");

   rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
   if (rc != SQL_SUCCESS) printf("ERROR: SQLAllocHandle(hstmt)");

   RC = SQLExecDirect(hstmt, sql_stmt, SQL_NTS);
   if (rc != SQL_SUCCESS) printf("ERROR: SQLExecDirect()");

   SQLFreeHandle(SQL_HANDLE_STMT, hstmt);

   SQLDisconnect( hdbc ) ;

   SQLFreeHandle(SQL_HANDLE_DBC, hdbc) ;
   SQLFreeHandle(SQL_HANDLE_ENV, henv) ;
}
```

2.2.2 Zugriff mit Parametermarkern



Binden von Parametermarkern

Funktion

SQLBindParameter **(** SQLHSTMT **StatementHandle**,
SQLUSMALLINT **ParameterNumber**,
SQLSMALLINT **InputOutputType**,
SQLSMALLINT **ValueType**,
SQLSMALLINT **ParameterType**,
SQLUINTEGER **ColumnSize**,
SQLSMALLINT **DecimalDigits**,
SQLPOINTER **ParameterValuePtr**,
SQLINTEGER **BufferLength**,
SQLINTEGER * **StrLen_or_IndPtr** **)**

Erläuterung

Bindet einen Parametermarker einer SQL-Anweisung an eine Variable des Wirtsprogramms

- ParameterNumber (input)
  Lfd. Nummer des Parametermarkers

- InputOutputType (input)
  Typ des Parametermarkers:
  SQL_PARAM_INPUT
  SQL_PARAM_INPUT_OUTPUT
  SQL_PARAM_OUTPUT

- ValueType
  symbolischer C-Datentyp des Parametermarkers

- ParameterType
  symbolischer SQL-Datentyp des Parametermarkers

- ColumnSize
  Länge des zum Parametermarker korrespondierenden Attributs.

- DecimalDigits
  Für Datentyp DECIMAL, NUMERIC: Anzahl Nachkommastellen

- ParameterValuePtr (input/output)
  Zeiger auf einen Puffer, aus dem Daten übernommen, bzw. in die Daten geschrieben werden sollen.

- BufferLength
  Für char-Datentyp (String): Größe des Puffers ParameterValuePtr

- StrLen_or_IndPtr
  StrLen:
  Zeiger auf Bereich (Variable), der die (wahre) Länge des Strings im Puffers ParameterValuePtr enthält.
  IndPtr:
  Soll hier ein Nullwert an die Datenbank übergeben werden, so erhält StrLen_or_IndPtr den Wert SQL_NULL_DATA

Datentypkonvertierung

Grunddatentypen

CLI-Makro-C-Typ	Standard- C-Typ
SQLCHAR	unsigned char
SQLSCHAR	signed char
SQLUSMALLINT	unsigned short int
SQLSMALLINT	short int
SQLUINTEGER	unsigned long int
SQLINTEGER	long int
SQLDOUBLE	double
SQLREAL	float

DATE_STRUCT	typedef DATE_STRUCT
	{
	short int year;
	unsigned short int month;
	unsigned short int day;
	};

TIME_STRUCT	typedef TIME_STRUCT
	{
	unsigned short int hour;
	unsigned short int minute;
	unsigned short int second;
	};

besondere CLI-Datentypen

CLI-Makro-Typ	Standard- C-Typ	Typische Verwendung
SQLPOINTER	void *	Zeiger auf Datenbereich und Parameter
SQLHANDLE	long int	Handle für alle 4 Typen von Referenzen
SQLHENV	long int	Handle für Umgebungsinformationen
SQLHDBC	long int	Handle für Datenbankverbindungsinformationen.
SQLHSTMT	long int	Handle für Anweisungsinformationen
SQLRETURN	short int	Rückgabewerte der DB2 CLI Funktionen.

SQL-Datentypen und Standard-Datentypen

SQL-Datentyp	symbolischer SQL-Datentyp	symbolische C-Datentyp	C-Datentyp
CHAR	SQL_CHAR	SQL_C_CHAR	unsigned char
VARCHAR	SQL_VARCHAR	SQL_C_CHAR	unsigned char
SMALLINT	SQL_SMALLINT	SQL_C_SHORT	short int
INTEGER	SQL_INTEGER	SQL_C_LONG	long int
DECIMAL	SQL_DECIMAL	SQL_C_CHAR	unsigned char
DOUBLE	SQL_DOUBLE	SQL_C_DOUBLE	double
FLOAT	SQL_FLOAT	SQL_C_DOUBLE	double
NUMERIC	SQL_NUMERIC	SQL_C_CHAR	unsigned char
REAL	SQL_REAL	SQL_C_FLOAT	float
DATE	SQL_TYPE_DATE	SQL_C_TYPE_DATE	siehe DATE_STRUCT
TIME	SQL_TYPE_TIME	SQL_C_TYPE_TIME	siehe TIME_STRUCT

Anweisung auswerten

Funktion

SQLRowCount (SQLHSTMT **StatementHandle**,
SQLINTEGER * **RowCountPtr** **)**

Erläuterung

Zahlt die Tupel, die von einer INSERT-, UPDATE- oder DELETE-Anweisungen betroffen wurden

- RowCountPtr
  Zeiger auf Speicher (Variable), der die Anzahl aufnimmt

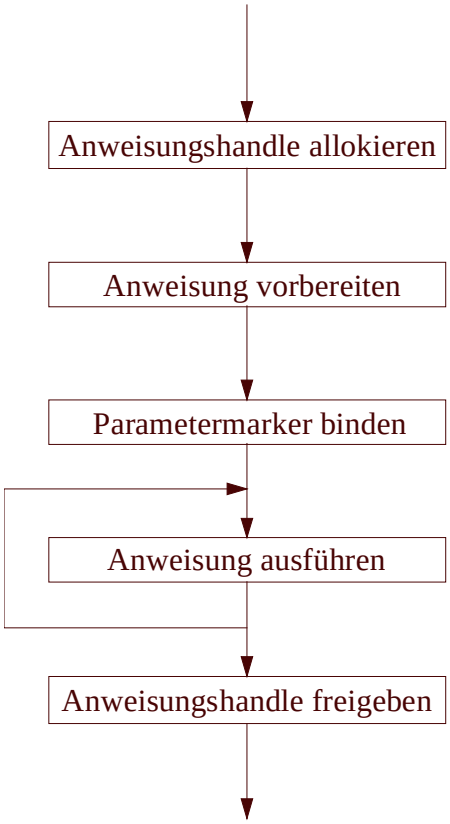
Beispiel

UPDATE mit Parametermarkern und Auswertung RowCount

```
int main()
{
   SQLHANDLE      henv;
   SQLHANDLE      hdbc;
   SQLHSTMT       hstmt;
   SQLSMALLINT    tod;
   SQLVARCHAR     knr[4];
   SQLINTEGER     rows;

   SQLCHAR sql_stmt[] = "UPDATE Maler SET Tod = ? WHERE KNR = ?";

   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
   SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
   SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

   SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SHORT,
                            SQL_SMALLINT, 0, 0, &tod, 0, NULL);
   SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                            SQL_VARCHAR, 4, 0, knr, 4, NULL);

   printf("KNR: "); scanf("%s", knr);
   printf("Tod: "); scanf("%d", &tod);

   SQLExecDirect(hstmt, sql_stmt, SQL_NTS);

   SQLRowCount(hstmt, &rows);
   printf("%d Tupel geaendert",rows);
   getchar();getchar();

   SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
   SQLDisconnect(hdbc);
   SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
   SQLFreeHandle(SQL_HANDLE_ENV, henv);

   return (SQL_SUCCESS);
}
```

2.2.3 Tupelweise Ein-/Ausgabe

Tupelweise Eingabe (INSERT)



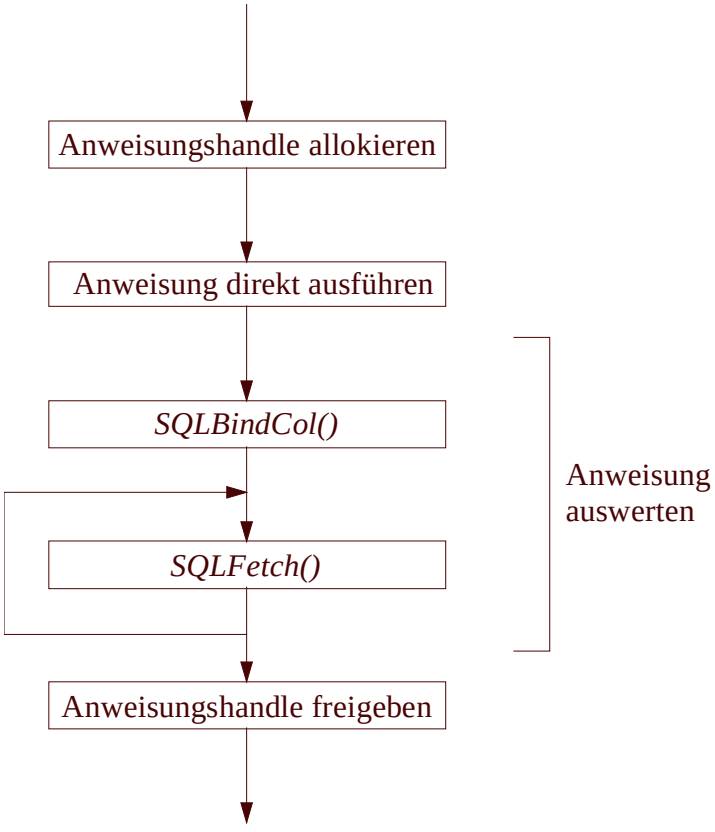
Beispiel

Tupelweises INSERT mit Parametermarkern, Prepare/Execute

```
int main()
{
    SQLHANDLE    henv;
    SQLHANDLE    hdbc;
    SQLHSTMT     hstmt;
    SQLCHAR      knr[4];
    SQLCHAR      ort[21];
    SQLCHAR      weiter;

    SQLCHAR sql_stmt[] = "INSERT INTO Atelier VALUES (?, ?)";

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
    SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    SQLPrepare(hstmt, sql_stmt, SQL_NTS);
```

```
    SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                             4, 0, knr, sizeof(knr), NULL);
    SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR,
                             21, 0, ort, sizeof(ort), NULL);

    do
    {
        printf("KNR: "); scanf("%s", knr);
        printf("Ort: "); scanf("%s", ort);
        printf("weiter? (j): "); scanf("%s", &weiter);
        SQLExecute(hstmt);
    } while(weiter == 'j');

    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    SQLDisconnect(hdbc);
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    SQLFreeHandle(SQL_HANDLE_ENV, henv);

    return (SQL_SUCCESS);
}
```

Tupelweise Ausgabe (SELECT)



Binden von Ausgabeattributen

Funktion

SQLBindCol (SQLHSTMT **StatementHandle**,
SQLUSMALLINT **ColumnNumber**,
SQLSMALLINT **TargetType**,
SQLPOINTER **TargetValuePtr**,
SQLINTEGER **BufferLength**,
SQLINTEGER * **StrLen_or_IndPtr** **)**

Erläuterung

bindet die Attribute einer Ergebnisrelation an eine Wirtsvariable.

- ColumnNumber
  Nummer des zu bindenden Attributs.

- TargetType
  symbolischer C-Datentyp des Attributs ColumnNumber

- TargetValuePtr
  Zeiger auf einen Puffer (Wirtsvariable), in die Daten (Attributwerte) geschrieben werden sollen

- BufferLength
  Für char-Datentyp (String): Größe des Puffers TargetValuePtr

- StrLen_or_IndPtr
  Zeiger auf Bereich (Variable), der die Länge eines zurückgegebenen Attributwertes (String: Anzahl der Zeichen) im Puffers TargetValuePtr enthält

Einlesen des aktuellen Tupels

Funktion

SQLFetch (SQLHSTMT **StatementHandle** **)**

Erläuterung

liest das aktuelle Tupel in Wirtsvariablen und erhöht den Cursor

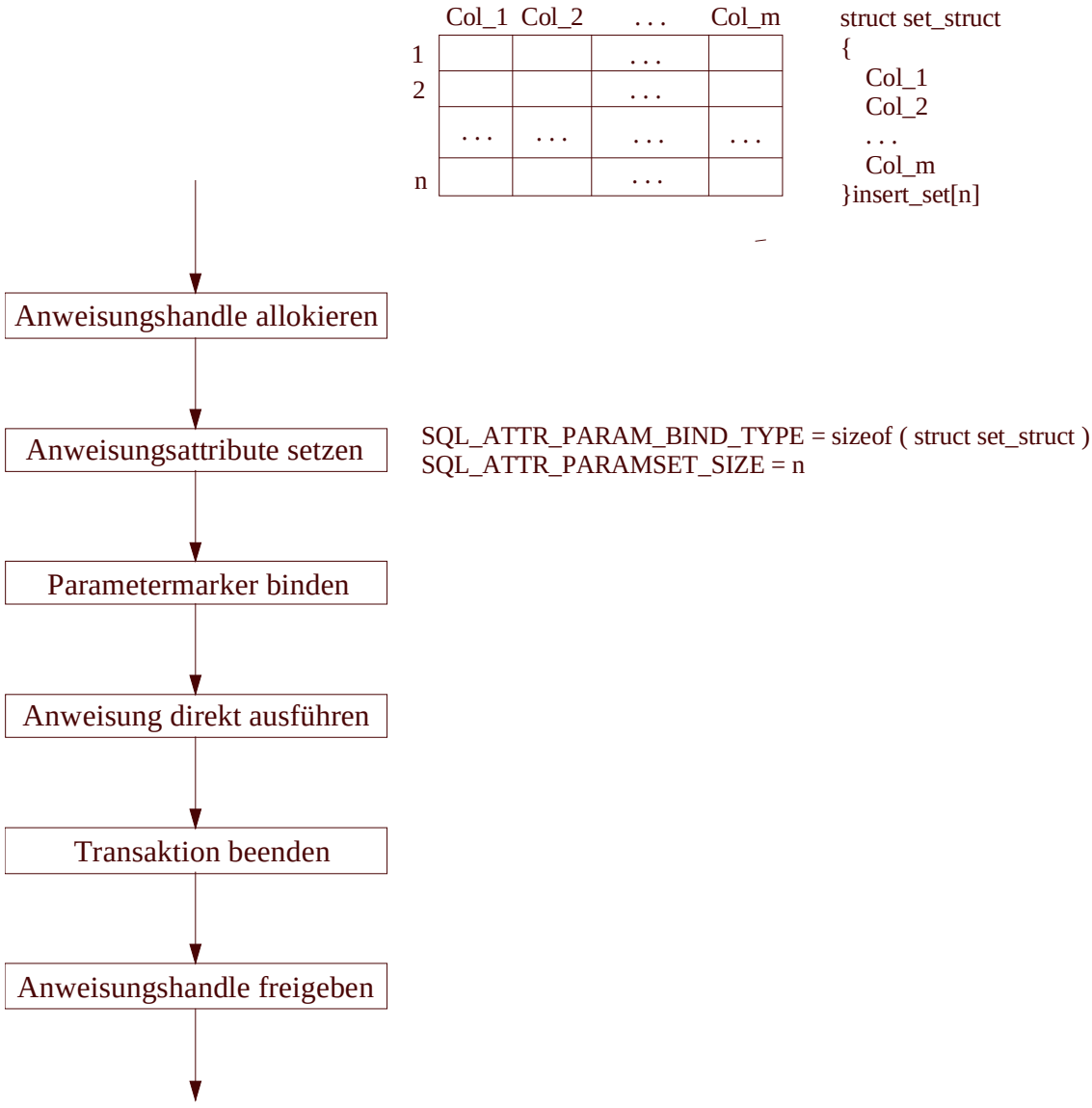
Beispiel

SELECT-Anweisung, tupelweises Fetch(), mit Auswertung Indikator-Variable

```
int main()
{
    SQLHANDLE       henv;
    SQLHANDLE       hdbc;
    SQLHSTMT        hstmt;
    SQLCHAR         mname[21];
    SQLSMALLINT     geburt = 0;
    SQLSMALLINT     tod = 0;
    SQLINTEGER      tod_ind;

    SQLCHAR sql_stmt[] = "SELECT Name, Geburt, Tod FROM maler";

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
    SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

    SQLExecDirect(hstmt, sql_stmt, SQL_NTS);

    SQLBindCol( hstmt, 1, SQL_C_CHAR, mname, 21, 0);
    SQLBindCol( hstmt, 2, SQL_C_SHORT, &geburt, 0, 0);
    SQLBindCol( hstmt, 3, SQL_C_SHORT, &tod, 0, &tod_ind);

    while (SQLFetch(hstmt) == SQL_SUCCESS)
    {
        if(tod_ind == SQL_NULL_DATA)
            printf("%s\t\t%d\n",mname, geburt);
        else
            printf("%s\t\t%d bis %d\n",mname, geburt, tod);
    }
    getchar();

    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    SQLDisconnect(hdbc);
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    SQLFreeHandle(SQL_HANDLE_ENV, henv);

    return (SQL_SUCCESS);
}
```

2.2.4 Ein-/Ausgabe von Tupelmengen (arrays)

Eingabe (INSERT) von Tupelmengen



Anweisungsattribute setzen

Funktion

SQLSetStmtAttr (SQLHSTMT **StatementHandle**,
SQLINTEGER **Attribute**,
SQLPOINTER **ValuePtr**,
SQLINTEGER **StringLength)**

Erläuterung

definiert Attribute zur Modifikation der Ausführung einer Anweisung

- Attribute
  Ausführungsattribut, das definiert werden soll

- ValuePtr
  Definition des Ausführungsattributs

Ausführungsattribute

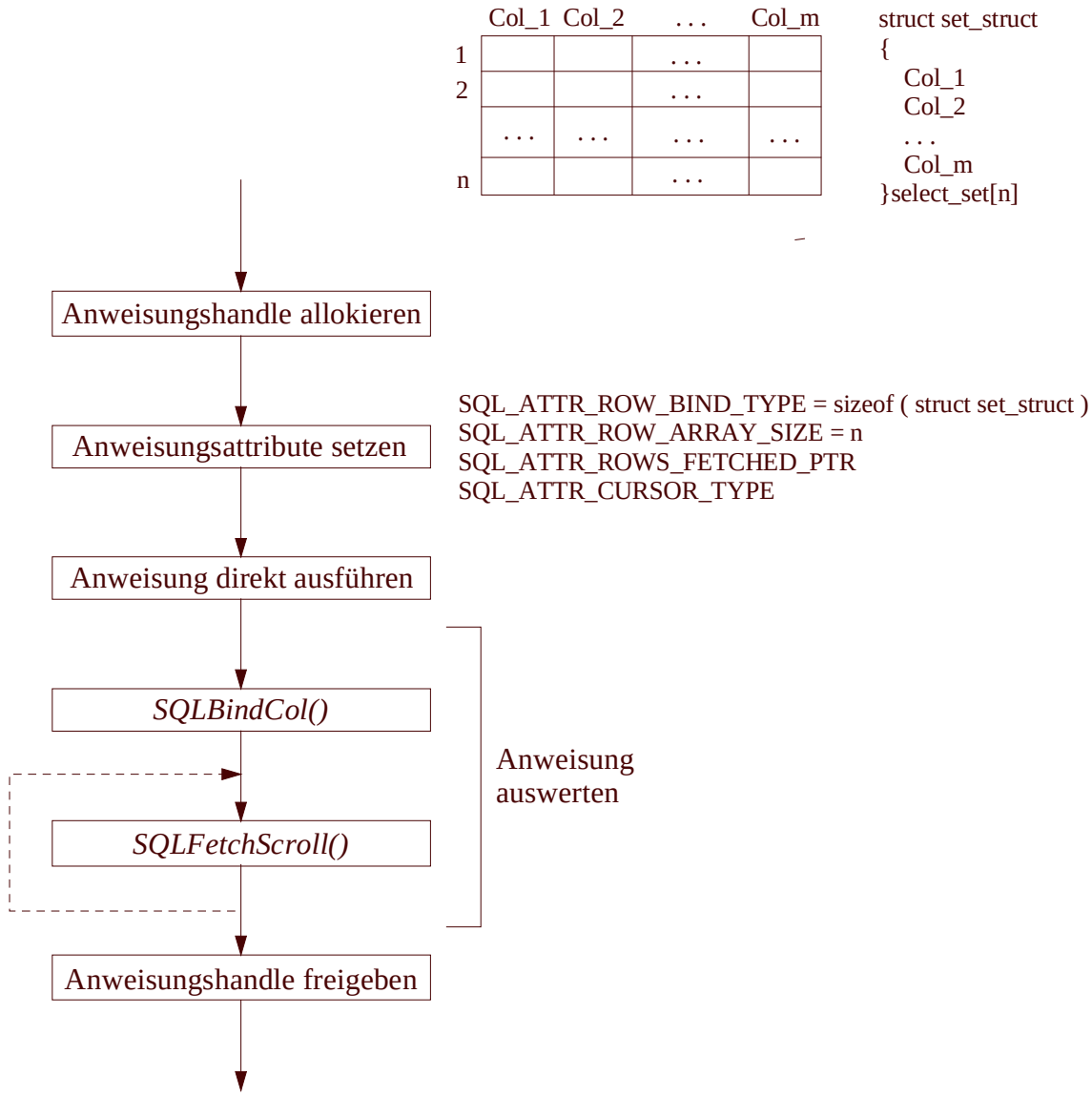
Attribute	Werte (ValuePtr)
SQL_ATTR_PARAM_BIND_TYPE	Größe der Struktur eines Eingabetupels
SQL_ATTR_PARAMSET_SIZE	Anzahl der Tupel der Eingabemenge
SQL_ATTR_ROW_BIND_TYPE	Größe der Struktur eines Ausgabetupels
SQL_ATTR_ROW_ARRAY_SIZE	Anzahl der Tupel der Ausgabemenge
SQL_ATTR_ROWS_FETCHED_PTR	Zeiger auf Speicher (Variable) zur Aufnahme der Anzahl der ausgegebenen Tupel
SQL_ATTR_CURSOR_TYPE	SQL_CURSOR_FORWARD_ONLY SQL_CURSOR_STATIC

Beispiel

Tupelmengenweises INSERT mit Parametermarkern

```
int main()
{
   SQLHANDLE    henv;
   SQLHANDLE    hdbc;
   SQLHSTMT     hstmt;

   SQLINTEGER   set_size=3;
   struct    set_struct
   {
      SQLCHAR    knr[4];
      SQLCHAR    ort[21];
   } insert_set[] = {{"K11","Paris"},{"K22","Berlin"},{"K33","Rom"}};

   SQLCHAR sql_stmt[] = "INSERT INTO Atelier VALUES (?, ?)";


   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
   SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
   SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE,
                  (SQLPOINTER)sizeof(struct set_struct), 0);
   SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, (SQLPOINTER)set_size, 0);

   SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,  SQL_VARCHAR, 4, 0,
                    insert_set[0].knr, 4, NULL);
   SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_VARCHAR, 21, 0,
                    insert_set[0].ort, 21, NULL);

   SQLExecDirect(hstmt, sql_stmt, SQL_NTS);

   SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
   SQLDisconnect(hdbc);
   SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
   SQLFreeHandle(SQL_HANDLE_ENV, henv);

   return (SQL_SUCCESS);
}
```

Ausgabe (SELECT) von Tupelmengen



Scrollbarer Cursor

Funktion

SQLFetchScroll (SQLHSTMT **StatementHandle**,
SQLSMALLINT **FetchOrientation**,
SQLINTEGER **FetchOffset** **)**

Erläuterung

liest eine Menge von Tupeln in ein Wirtsvariablen-Set und verschiebt den Cursor

- FetchOrientation
  Positionierung des Cursors und Fetch-Richtung

- FetchOffset
  Beginn der Position oder Anzahl der Tupel, die gelesen werden

- Anzahl der Tupel (Tupelmenge), die gelesen werden wird bestimmt durch SQL_ATTR_ROW_ARRAY_SIZE in SQLSetStmtAttr().

Fetch-Richtung

FetchOrientation	Bedeutung
SQL_FETCH_NEXT	liest die nächste Tupelmenge
SQL_FETCH_PRIOR	liest die vorhergehende Tupelmenge
SQL_FETCH_RELATIVE	liest n Tupel vom Beginn der aktuellen Tupelmenge an n ist der Wert von FetchOffset
SQL_FETCH_ABSOLUTE	liest die nächste Tupelmenge beginnend von Tupel n n ist der Wert von FetchOffset
SQL_FETCH_FIRST	liest die erste Tupelmenge der Ergebnisrelation
SQL_FETCH_LAST	liest die letzte Tupelmenge der Ergebnisrelation
SQL_FETCH_BOOKMARK	liest die nächste Tupelmenge beginnend von einem vorher definierten Lesezeichen an

- SQL_ATTR_CURSOR_TYPE in SQLSetStmtAttr()

  SQL_CURSOR_FORWARD_ONLY

  SQL_CURSOR_STATIC

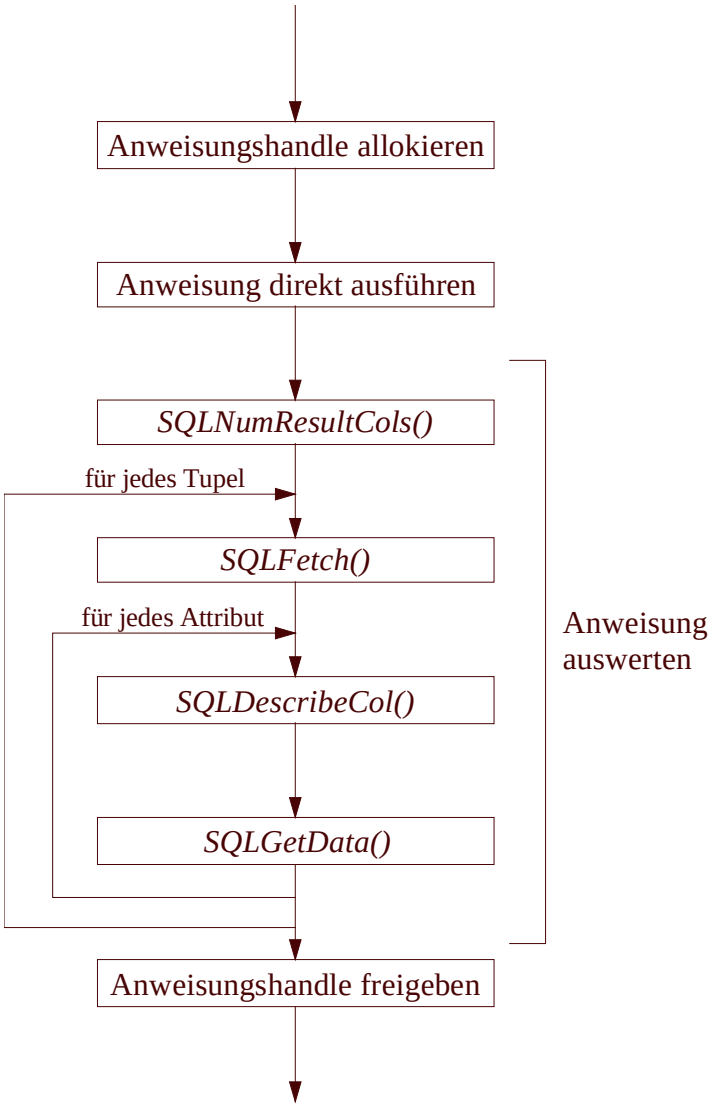
Beispiel

Mengenweises SELECT, FetchScroll()

```
int main()
{
   SQLHANDLE    henv;
   SQLHANDLE    hdbc;
   SQLHSTMT     hstmt;

   SQLINTEGER   i;
   SQLINTEGER   set_size=4;
   struct    set_struct
   {
      SQLCHAR           mname[21];
      SQLSMALLINT       geburt;
   } select_set[4];

   SQLUINTEGER numrowsfetched;

   SQLCHAR sql_stmt[] = "SELECT Name, Geburt FROM Maler";

   SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
   SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
   SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

   SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE,
                  (SQLPOINTER) sizeof(struct set_struct), 0);
   SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER) set_size, 0);
   SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &numrowsfetched, 0);
   SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE,
                  (SQLPOINTER) SQL_CURSOR_STATIC, 0);

   SQLExecDirect(hstmt, sql_stmt, SQL_NTS);

   SQLBindCol( hstmt, 1, SQL_C_CHAR, select_set[0].mname, 21, 0);
   SQLBindCol( hstmt, 2, SQL_C_SHORT, &select_set[0].geburt, 0, 0);

   SQLFetchScroll(hstmt, SQL_FETCH_FIRST, 0);
   printf("1. Fetch\n");
   for(i=0; i<numrowsfetched; i++)
      printf("\t%s\t\t%d\n", select_set[i].mname, select_set[i].geburt);

   SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
   printf("2. Fetch\n");
   for(i=0; i<numrowsfetched; i++)
      printf("\t%s\t\t%d\n", select_set[i].mname, select_set[i].geburt);

   SQLFetchScroll(hstmt, SQL_FETCH_PRIOR, 0);
   printf("3. Fetch\n");
```

```
    for(i=0; i<numrowsfetched; i++)
        printf("\t%s\t\t%d\n", select_set[i].mname, select_set[i].geburt);

    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    SQLDisconnect(hdbc);
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    SQLFreeHandle(SQL_HANDLE_ENV, henv);

    return (SQL_SUCCESS);
}
```

2.2.5 Behandlung unbekannter Anfagen



Funktion

SQLNumResultCols (SQLHSTMT StatementHandle,
SQLSMALLIN * ColumnnCountPtr)

Erläuterung

liefert die Anzahl der Attribute der Ergebnisrelation einer Anfrage

- ColumnnCountPtr
  Anzahl der Attribute

Funktion

SQLDescribeCol (SQLHSTMT StatementHandle,
SQLSMALLINT ColumnNumber,
SQLCHAR * ColumnName,
SQLSMALLINT BufferLength,
SQLSMALLINT * NameLengthPtr,
SQLSMALLINT * DateTypePtr,
SQLSMALLINT * ColumnSizePtr,
SQLSMALLINT * DecimalDigitsPtr,
SQLSMALLINT * NullablePtr)

Erläuterung

liefert eine Beschreibung eines Attributs einer Ergebnisrelation

- ColumnNumber
  lfd. Nummer des Attributs

- ColumnName
  Zeiger auf den Speicher (Variable) für den Attributnamen

- BufferLength
  Größe des Speichers

- NameLengthPtr
  Anzahl der Zeichen des ermittelten Attributnamens

- DateTypePtr
  symbolischer SQL-Datentyp des Attributs

- ColumnSizePtr
  Länge des Attributs in Anzahl Byte

- DecimalDigitsPtr
  Anzahl der Nachkommastellen

- NullablePtr
  Angabe, ob NULL-Werte erlaubt (SQL_NULLABLE) oder nicht erlaubt (SQL_NO_NULLS) sind

Funktion

```
SQLGetData (   SQLHSTMT          StatementHandle,
               SQLUSMALLINT      ColumnNumber,
               SQLSMALLINT       TargetType,
               SQLPOINTER        TargetValuePtr,
               SQLINTEGER        BufferLength,
               SQLINTEGER *      StrLen_or_IndPtr    )
```

Erläuterung

ermittelt den Wert eines Attributs des aktuellen Tupels einer Ergebnisrelation

- ColumnNumber
  lfd. Nummer des Attributs

- TargetType
  korrepondierender symbolischer C-Datentyp des Attributs

- TargetValuePtr
  Zeiger auf den Speicher (Wirtsvariable) für den Attributwert

- BufferLength
  Größe des Speichers (Wirtsvariable)

- StrLen_or_IndPtr
  Zeiger auf Bereich (Variable), der die Länge eines zurückgegebenen Attributwertes (String: Anzahl der Zeichen) im Puffers TargetValuePtr enthält

Beispiel

unbestimmte SELECT-Anweisung

```
int main()
{
    SQLHANDLE    henv;
    SQLHANDLE    hdbc;
    SQLHSTMT     hstmt;

    SQLCHAR      sql_stmt[100];

    SQLSMALLINT degree;

    SQLCHAR      col_name[SQL_MAX_ID_LENGTH + 1];

    SQLSMALLINT col_type;
    SQLSMALLINT col_num;

    SQLSMALLINT int_col_value;
    SQLCHAR      char_col_value[100];
    SQLINTEGER   nullindicator;

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    SQLConnect(hdbc, "KUNST_DB", SQL_NTS, "Admin", SQL_NTS, "xyz", SQL_NTS);
    SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt) ;


    printf("SELECT-Anweisung eingeben:\n");
    gets(sql_stmt);

    SQLExecDirect(hstmt, sql_stmt, SQL_NTS);


    SQLNumResultCols(hstmt, &degree);


    while(SQLFetch(hstmt) == SQL_SUCCESS)
    {
       for(col_num=1; col_num<=degree; col_num++)
       {
          SQLDescribeCol(hstmt, col_num, col_name, SQL_MAX_ID_LENGTH+1, NULL,
                         &col_type, NULL, NULL, NULL);
          switch(col_type)
          {
             case SQL_VARCHAR:
                SQLGetData(hstmt, col_num, SQL_C_CHAR, char_col_value, 100, &nullindicator);
                if(nullindicator==SQL_NULL_DATA)
                   printf("\n%-10s:\t-", col_name);
                else
```

```
                    printf("\n%-10s:\t%s", col_name, char_col_value);
                break;
            case SQL_SMALLINT:
                SQLGetData(hstmt, col_num, SQL_C_SHORT, &int_col_value, 0, &nullindicator);
                if(nullindicator==SQL_NULL_DATA)
                    printf("\n%-10s:\t-", col_name);
                else
                    printf("\n%-10s:\t%d", col_name, int_col_value);
                break;
            }
        }
        printf("\n");
        getchar();
    }

    SQLDisconnect( hdbc ) ;
    SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    SQLFreeHandle(SQL_HANDLE_DBC, hdbc) ;
    SQLFreeHandle(SQL_HANDLE_ENV, henv) ;

    return ( SQL_SUCCESS ) ;
}
```

2.2.6 Transaktion

Transaktion beenden

Funktion

SQLEndTran **(** SQLSMALLINT **HandleType**,
SQLHANDLE **Handle**,
SQLSMALLINT **CompletionType** **)**

Erläuterung

setzt ein Transaktionsende für aktive SQL-Anweisungen

- HandleType
  SQL_HANDLE_ENV (Umgebungshandle)
  SQL_HANDLE_DBC (Verbindungshandle)

- Handle
  SQL_HANDLE_ENV: existierendes Umgebungshandle
  SQL_HANDLE_DBC: existierendes Verbindungshandle

- CompletationType
  SQL_COMMIT
  SQL_ROLLBACK

3 Externe Datenbankroutinen

Kennzeichen externer Routinen

- in Wirtsprogrammiersprache geschrieben
- werden i.d. Regel auf dem DB-Server abgelegt, Zugriff vom Client
- Schritte
  1. Definition der Routine
  2. Compilieren und Linken (.dll-file)
  3. Anmeldung im Datenbanksystem (Systemrelation)
  4. Aufruf in SQL-Anweisung (Funktion) oder CALL (Routine)

3.1 Externe Funktionen

3.1.1 Externe skalare Funktionen

- werten 0..n Eingabeparameter aus und liefert einen einzigen Ausgabeparameter
- können wie vordefinierte (built-in) Funktionen in SQL-Anweisungen aufgerufen werden, d.h. immer dort, wo ein Ausdruck (<expression>) auftreten kann.

3.1.1.1 Definition einer externen skalaren Funktion

Syntax

extern-scalar-function ::=

void SQL_API_FN function-name (parameter)

{

function-body

}

parameter ::=

[{ input-parameter , }...]

output-parameter ,

[{ input-parameter-indicator , }...]

output-parameter-indicator ,

sql-state,

function-name,

specific-name,

error-message

[, scratchpad [, call-type [, dbinfo]]]

input-parameter

input-parameter ::=

*c-data-type * variable-pointer*

- Funktion kann bis zu 90 Parameter haben
- *variable-pointer*: Zeiger auf Datenobjekte (Speicherbereiche)., die Kopien der eigentlichen Eingabeparameter sind.
- Funktion kann Eingabeparameter nicht über die Zeiger manipulieren

output-parameter

output-parameter ::=

*c-data-type * variable-pointer*

- *variable-pointer*: Zeiger auf C-Variable im der Funktionsrumpf, die den Rückgabewert enthält

input-parameter-indicator

input-parameter-indicator ::=

*short * variable-pointer*

| *SQLUDF_NULLIND * variable-pointer*

- zu jedem definierten Eingabeparameter muß ein NULL-Indikator definiert werden
- Werte: = 0 - Argument ist nicht NULL
<>0 - Argument ist NULL

output-parameter-indicator

output-parameter-indicator ::=

*short * variable-pointer*

| *SQLUDF_NULLIND * variable-pointer*

- zum Rückgabeparameter muß genau ein NULL-Indikator definiert werden

sql-state

sql-state ::=

*char * sql-state-pointer*

- *sql-state-pointer*: Zeiger auf einen 5 Zeichen langen nullterminierten String (char[6]-Feld)
- enthält SQLSTATE-Information (aus Fehlerstruktur sqlca)
- Werte:
  - 00000 Funktion ohne Fehler und Warnungen beendet
  - 01Hxx für benutzerdefinierte Warnungen reserviert, xx = 00 .. 99
  - 38xxx für benutzerdefinierte Fehler reserviert, xxx = 600 .. 999
  - 02000 nur für FETCH-Aufrufe: kein weiteres Tupel gefunden
  - alle anderen Werte als Fehler interpretiert (z.B. 38503 function abnormal termination)

function-name

function-name ::=

*char * function-name-pointer*

- *function-name-pointer*: Zeiger auf ein 27 Zeichen langen nullterminierten String (char[28])
- erhält intern den vollen qualifizierten Namen der SQL-Funktion in Form

db-schema-name **.** *f unction-name*

specific-name

specific-name ::=

*char * specific-name-pointer*

- *specific-name-pointer*: Zeiger auf ein 18 Zeichen langen nullterminierten String (char[19])
- enthält den DB-System-generierten Namen der Funktion, dieser kann dann in Änderungsoperationen oder beim Setzen von Kommentaren für die Funktion verwendet werden

error-message

error-message ::=

*char * error-message-pointer*

- *error-message-pointer*: Zeiger auf ein 70 Zeichen langen nullterminierten String (char[71])
- Inhalt des Strings wird bei Rückkehr der Funktion mit SQLSTATE<>00000 in das Feld sqlerrmc des SQLCA-Kontrollblocks kopiert und als Fehlernachricht (z.B. in Befehlszentrale) ausgegeben

scratchpad

scratchpad ::=

*struct sqludf_scratchpad * scratchpad-pointer*

- über diesen Speicherbereich können Informationen (Werte) von einem Funktionsaufruf an einen nachfolgenden Aufruf der gleichen Funktion innerhalb einer SQL-Anweisung übergeben werden

call-type

call-type ::=

*enum sqludf_call_type * call-type-pointer*

- der Wert zeigt an, ob es sich um einen ersten oder einen letzten Aufruf der Funktion innerhalb einer SQL-Anweisung handelt

dbinfo

dbinfo ::=

*struct sqludf_dbinfo * dbinfo-pointer*

- über diesen Speicherbereich werden Informationen über die Datenbank bei Aufruf der Funktion durch das DB-System an die Funktion übergeben
  - Datenbankname
  - Authorisierung
  - Relationenname (nur beim Funktionsaufruf in UPDATE/INSERT)
  - Attributname (nur beim Funktionsaufruf in UPDATE/INSERT)
  - ...

Parametermakros

- Pflichtparameter:

```
#define   SQLUDF_TRAIL_ARGS
            char   sqludf_sqlstate[SQLUDF_SQLSTATE_LEN+1], \
            char   sqludf_fname[SQLUDF_FQNAME_LEN+1], \
            char   sqludf_fspecname[SQLUDF_SPECNAME_LEN+1], \
            char   sqludf_msgtext[SQLUDF_MSGTEXT_LEN+1]
```

- Pflichtparameter + optionale Parameter:

```
#define SQLUDF_TRAIL_ARGS_ALL
        char   sqludf_sqlstate[SQLUDF_SQLSTATE_LEN+1], \
        char   sqludf_fname[SQLUDF_FQNAME_LEN+1], \
        char   sqludf_fspecname[SQLUDF_SPECNAME_LEN+1], \
        char   sqludf_msgtext[SQLUDF_MSGTEXT_LEN+1], \
        SQLUDF_SCRATCHPAD *sqludf_scratchpad, \
        SQLUDF_CALL_TYPE  *sqludf_call_type
```

BEISPIEL

```
void SQL_API_FN funktion
    (
    short *out,
    short *nullout,
    SQLUDF_TRAIL_ARGS
    )
```

BEISPIEL

```
void SQL_API_FN funktion
    (
    short *in,
    short *out,
    short *nullin,
    short *nullout,
    SQLUDF_TRAIL_ARGS_ALL
    )
```

Hinweise für die Entwicklung von Nutzerdefinierten Routinen

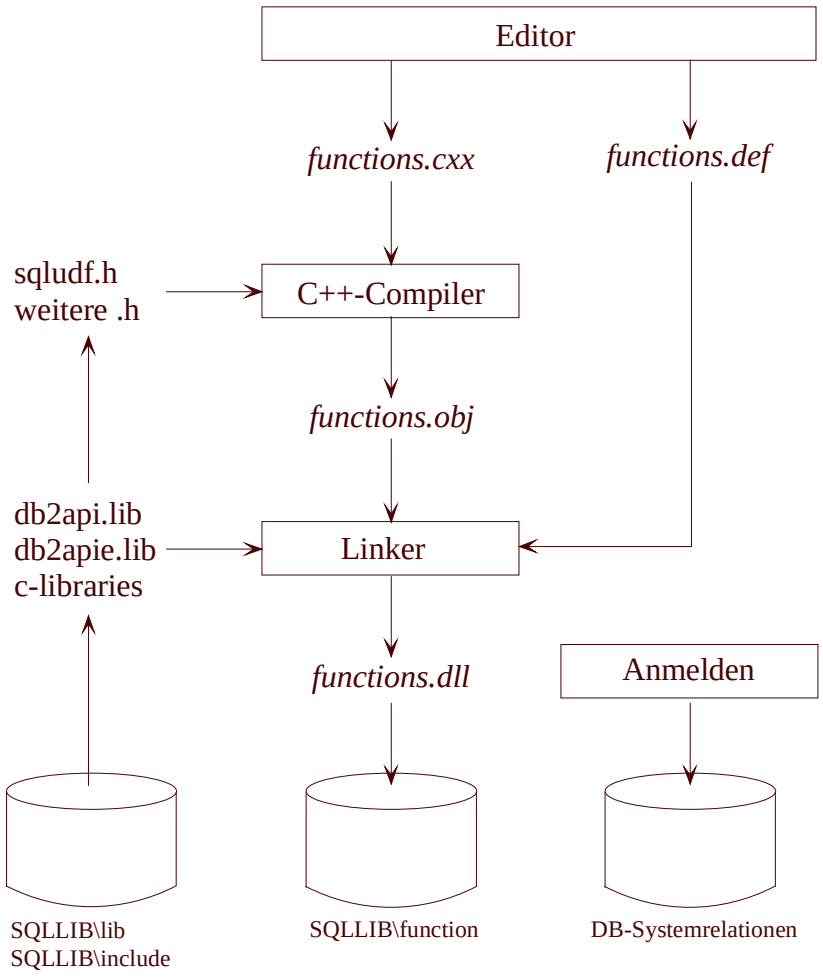
- Routinen sollte keine statischen oder globaleVariablen verwenden.
- dynamisch allokierter Speicher sollte vor return wieder freigegeben werden.
- Funktionen müssen, Prozeduren sollten immer mit return zurückkehren.
- keine Verwendung von scanf(), printf() etc. zum Schreiben/Lesen auf Standardausgabe/-eingabe.
- Möglichkeit des Debuggens nicht vorhanden, auch nicht BS-Ausgabe.
- Routine darf keine verbindungsbezogenen Anweisungen wie BACKUP,CONNECT, CREATE DATABASE, RESTORE ... enthalten
- Änderungen des Codes einer Routine sollten nicht bei laufendem DBS durchgeführt werden

- Für Parameter darf auf dem Server kein Speicher dynamisch allokiert werden. Der Datenbankmanager allokiert dafür automatisch Speicher beim Anmelden der Routine

- Parameterpointer dürfen in der Routine nicht geändert werden.

- Alle SQL-Argumente an die Routine sind gepuffert, d.h. es werden der Routine nur Kopien der Werte übergeben.

Erstellungprozeß



1. C++ Quelltext editieren

2. Moduldefinitionsdatei editieren

3. Compilieren und Linken

   bldmsudf.bat
   ```
   cl -Z7 -Od -c -W2 -D_X86_=1 -DWIN32 %1.cxx
   ```

link -debug:full -debugtype:cv -dll -out:%1.dll %1.obj db2api.lib db2apie.lib -def:%1.def

4. functions.dll im Funktionenverzeichnis auf Server ablegen
5. Funktion im Datenbanksysytem anmelden

3.1.1.2 Anmelden einer externen skalaren Funktion

Syntax

create-extern-scalar-function ::=

CREATE FUNCTION function-name
([parameter-declaration [{ , parameter-declaration }...]])
RETURNS return-type
EXTERNAL [NAME external-name]
LANGUAGE { C | JAVA | CLR | OLE }
PARAMETER STYLE { DB2GENERAL | JAVA | SQL }
[{ option } ...]

option ::=

specific-clause
| *fenced-clause*
| *null-call-clause*
| *dbinfo-clause*
| *deterministic-clause*
| *external-action-clause*
| *scratchpad-clause*
| *final-call-clause*
| *sql-clause*

- Anmeldung schafft eine Verbindung zwischen der Funktionenimplementation auf dem Server und der Datenbank
- für die Parameter, Return-Wert sowie Scratchpad wird Speicherplatz vom DB-System allokiert und zur Verfügung gestellt.

function-name

function-name ::=

[schema-name.] name

- Funktionsname: SQL-Identifikator, max. 18 Zeichen
- zu einer Implementation (Definition) können mehrere Funktionen angemeldet werden, die dann im unterschiedlichen Kontext in SQL-Anweisungen verwendet werden

parameter-declaration

parameter-declaration ::=

[parameter-name] sql-data-type [AS LOCATOR]

- Funktion kann 0 bis 90 Parameter haben
- bei Übergabe an die C-Implementation wird jeder Parameter vom SQL-Datentyp in einen der Tabelle entsprechenden C-Typ konvertiert (symbolischer Datentyp definiert in sqludf.h)

SQL Datentyp	**Symolischer Datentyp**	**C-Datentyp**
SMALLINT	SQLUDF_SMALLINT	short
INTEGER	SQLUDF_INTEGER	long
DECIMAL(p,s)	-	-[1]
REAL	SQLUDF_REAL	float
DOUBLE	SQLUDF_DOUBLE	double
CHAR(n)	SQLUDF_CHAR	char[n+1] [2]
VARCHAR(n)	SQLUDF_VARCHAR	char[n+1] [2]
DATE	SQLUDF_DATE	char[11] [2,3]
TIME	SQLUDF_TIME	char[9] [2,4]
...	...	...

[1] keine Entsprechung in C, Konvertierung z.B. durch Funktion double(Decimal) in Double möglich
[2] abschließendes \0
[3] in yyy-mm-dd
[4] in hh.mm.ss

return-type

return-type ::=

sql-data-type [AS LOCATOR]

| *sql-data-type1 CAST FROM sql-data-type2 [AS LOCATOR]*

- SQL-Rückgabetyp, in den der C-Typ der Implementation nach obiger Tabelle konnvertiert wird
- *CAST FROM*: erforderlich, wenn im SQL-Aufruf der Funktion ein anderer Rückgabetyp, als durch die C-Implementation zurückgegeben, verwendet werden soll.

external-name

external-name ::=

[path] library [! funct-id]

- *funct-id*: Name der Funktionsimplementation innerhalb der DLL (.def-Datei)
- ist kein Pfad angegeben, dann sucht das DBS im Pfad '...\SQLLIB\FUNCTION' auf dem Server

LANGUAGE

- *C*: C/C++-Funktion
- *JAVA*: Funktion ist Methode einer Java-Klasse
- *CLR*: Funktion ist Methode einer .NET-Klasse
- *OLE*: Funktion ist OLE-Automation-Objekt

PARAMETER STYLE

- *DB2GENERAL*: nur für Java-Methoden
- *JAVA*: nur für Java-Methoden
- *SQL*: nur für C-, .NET und OLE-Funktionen

specific-clause

specific-clause ::=

SPECIFIC specific-function-name

- Spezieller DB-interner Name für Funktion. Erleichtert die Identifikation von überladenen Funktionen.

fenced-clause

fenced-clause ::=

[NOT] FENCED

- *FENCED* : abgeschirmt, default, Funktion wird immer in einem von der DB verschiedenen Adreßraum ausgeführt
- *NOT FENCED*: Funktion läuft im selben Adreßraum wie Datenbank

null-call-clause

null-call-clause ::=

RETURNS NULL ON NULL INPUT

| *CALLED ON NULL INPUT*

- *RETURNS NULL ON NULL INPUT:* wird einem Parameter der Funktion ein NULL-Wert übergeben, so wird die Funktion nicht ausgeführt und es wird sofort NULL zurückgegeben

dbinfo-clause

dbinfo-clause ::=

[NO] DBINFO

- *DBINFO*: wird gesetzt, wenn über Parameter *dbinfo* ein Zeiger auf einen strukturierten Speicherbereich gesetzt wurde.

deterministic-clause

deterministic-clause ::=

[NOT] DETERMINISTIC

- *DETERMINISTIC*: Funktion kehrt bei mehrfachem Aufruf mit gleichen Parameterwerten in der gleichen SQL-Anweisung immer mit demselben Ergebnis zurück

BEISPIEL

Funktionsdefinition

```
#include <stdio.h>
#include <sqludf.h>

void SQL_API_FN Alter
   (
   short *inGeburt,
   short *inTod,
   short *outAlter,
   short *nullinGeburt,
   short *nullinTod,
```

```
    short *nullOutAlter,
    SQLUDF_TRAIL_ARGS
    )
{
    if(*nullinTod<0)
        *outAlter = 2008 - *inGeburt;
    else
        *outAlter= *inTod - *inGeburt;
    return;
}
```

Funktionsanmeldung

```
DROP FUNCTION Alter;
CREATE FUNCTION Alter(INTEGER, INTEGER)
RETURNS INTEGER
EXTERNAL NAME '... \Beispiel\beispiel!Alter'
LANGUAGE C
PARAMETER STYLE SQL
CALLED ON NULL INPUT
DETERMINISTIC;
```

Funktionsaufruf-Beispiele

„Gesucht sind KNR, Name und Alter aller Maler“

```
SELECT KNR, Name, Alter(Geburt, Tod) AS Alter
FROM Maler;
```

„Gesucht sind KNR und Namen der lebenden Maler, die älter als 50 sind“

```
SELECT KNR, Name
FROM Maler
WHERE Alter(Geburt, Tod) > 50
AND Tod IS NULL;
```

external-action-clause

external-action-clause ::=

[NO] EXTERNAL ACTION

- *EXTERNAL ACTION*: informiert den DB-Optimierer darüber, daß die Funktion Aktionen außerhalb der DB ausführt

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <stdio.h>
```

```
#include <string.h>

void SQL_API_FN Bienale1
    (
    char  *InKNR,
    short *OutAntwort,
    short *nullInKNR,
    short *nullOutAntwort,
    SQLUDF_TRAIL_ARGS
    )
{
    char  knr[4];
    FILE *datei;
    datei = fopen("...\\Beispiel\\bienale.dat","r");
    *OutAntwort = 0;
    *nullOutAntwort = 0;
    if(!(*nullInKNR))
        while(fscanf(datei,"%s",knr)!=EOF)
            if(!strcmp(InKNR,knr))
            {
                *OutAntwort=1;
                break;
            }
    fclose(datei);
    return;
}
```

Funktionsanmeldung

```
CREATE FUNCTION Bienale1(VARCHAR(3))
RETURNS SMALLINT
EXTERNAL NAME '...\Beispiel\beispiel!Bienale1'
LANGUAGE C
PARAMETER STYLE SQL
CALLED ON NULL INPUT
DETERMINISTIC
EXTERNAL ACTION
NO SQL;
```

Funktionsaufruf-Beispiele

bienale.dat: Datei, die KNR von Malern enthält, die an der Bienale teilnahmen

„Gesucht sind die Namen der Galerien, die Bilder von Malern besitzen, die auf der Bienale in Venedig ausgestellt haben“

```
SELECT x.Bezeichnung
FROM Galerie x, Bild y
WHERE x.ENR=y.ENR
AND Bienale1(y.KNR)=1
```

Scratchpad

- persistenter Speicherbereich, der seinen Inhalt von einem Funktionsaufruf zum nächsten beibehält

```
#define   SQLUDF_SCRATCHPAD_LEN   100
struct sqludf_scratchpad
{
  unsigned long     length;
  char              data[SQLUDF_SCRATCHPAD_LEN];
};
```

scratchpad-clause

scratchpad-clause ::=

NO SCRATCHPAD

| *SCRATCHPAD [length]*

- *SCRATCHPAD*: zeigt an, daß für die Funktion ein Scratchpad definiert worden ist
- sqludf_scratchpad *sqludf_scratchpad
  bzw.
  SQLUDF_TRAIL_ARGS_ALL
  müssen als Funktionsparameter definiert worden sein
- *length*: nutzerdefinierte Scratchpadlänge, 1 .. 32 767 Byte möglich

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <stdio.h>
#include <string.h>

void SQL_API_FN Bienale2
   (
   char  *InKNR,
   short *OutAntwort,
   short *nullInKNR,
   short *nullOutAntwort,
   SQLUDF_TRAIL_ARGS_ALL
   )
{
   *OutAntwort = 0;
   *nullOutAntwort = 0;
   char  knr[4];
   struct save_area_struct
```

```
    {
        short open;
        FILE *datei;
    } *save_area;
    save_area=(save_area_struct *)(sqludf_scratchpad->data);
    if(!(save_area->open))
    {
        save_area->open=1;
        save_area->datei= fopen("...\\Beispiel\\bienale.dat","r");
    }
    if(!(*nullInKNR))
    {
        rewind(save_area->datei);
        while(fscanf(save_area->datei,"%s",knr)!=EOF)
            if(!strcmp(InKNR,knr))
            {
                *OutAntwort=1;
                break;
            }

    }
    return;
}
```

Funktionsanmeldung

```
CREATE FUNCTION Bienale2(VARCHAR(3))
RETURNS SMALLINT
EXTERNAL NAME '...\Beispiel\beispiel!Bienale2'
LANGUAGE C
PARAMETER STYLE SQL
CALLED ON NULL INPUT
DETERMINISTIC
EXTERNAL ACTION
SCRATCHPAD;
```

Funktionsaufruf-Beispiele

bienale.dat: Datei, die KNR von Malern enthält, die an Bienale teilnahmen

„Gesucht sind die Namen der Galerien, die Bilder von Malern besitzen, die auf der Bienale in Venedig ausgestellt haben“

```
SELECT x.Bezeichnung
FROM Galerie x, Bild y
WHERE x.ENR=y.ENR
AND Bienale2(y.KNR)=1
```

final-call-clause

final-call-clause ::=

NO FINAL CALL

| *FINAL CALL*

- *FINAL CALL* ist nur wirksam, wenn für die Funktion der *call-type*-Parameter definiert worden ist
- der *call-type*-Paremeter der Funktion erhält dann vom DBS folgende Werte

SQLUDF_FIRST_CALL	(Wert: -1)	beim ersten Funktionsaufruf
SQLUDF_NORMAL_CALL	(Wert: 0)	bei den folgenden Funktionsaufrufen
SQLUDF_FINAL_CALL	(Wert: 1)	beim letzten Funktionsaufruf

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <stdio.h>
#include <string.h>

void SQL_API_FN Bienale3
   (
   char  *InKNR,
   short *OutAntwort,
   short *nullInKNR,
   short *nullOutAntwort,
   SQLUDF_TRAIL_ARGS_ALL
   )
{
   *OutAntwort = 0;
   *nullOutAntwort = 0;
   char  knr[4];
   struct save_area_struct
   {
         FILE *datei;
   } *save_area;
   save_area=(save_area_struct *)(sqludf_scratchpad->data);
   switch(*sqludf_call_type)
   {
      case SQLUDF_FIRST_CALL:
         save_area->datei=fopen("...\\Beispiel\\bienale.dat","r");
      case SQLUDF_NORMAL_CALL:
         if(!(*nullInKNR))    // nullInKNR==0: kein NULL-Wert übergeben
         {
            rewind(save_area->datei);
```

```
                while(fscanf(save_area->datei,"%s",knr)!=EOF)
                    if(!strcmp(InKNR,knr))
                    {
                        *OutAntwort=1;
                        break;
                    }
            }
            break;
        case SQLUDF_FINAL_CALL:
            fclose(save_area->datei);
            break;
    }
    return;
}
```

Funktionsanmeldung

```
CREATE FUNCTION Bienale3(VARCHAR(3))
RETURNS SMALLINT
EXTERNAL NAME '...\Beispiel\beispiel!Bienale3'
LANGUAGE C
PARAMETER STYLE SQL
CALLED ON NULL INPUT
DETERMINISTIC
EXTERNAL ACTION
SCRATCHPAD
FINAL CALL;
```

Funktionsaufruf-Beispiele

bienale.dat: Datei, die KNR von Malern enthält, die an Bienale teilnahmen
„Gesucht sind die Namen der Galerien, die Bilder von Malern besitzen, die auf der Bienale in Venedig ausgestellt haben“

```
SELECT x.Bezeichnung
FROM Galerie x, Bild y
WHERE x.ENR=y.ENR
AND Bienale2(y.KNR)=1
```

Laufzeitverhalten von FINAL-CALL- / NO-FINAL-CALL-Funktionen

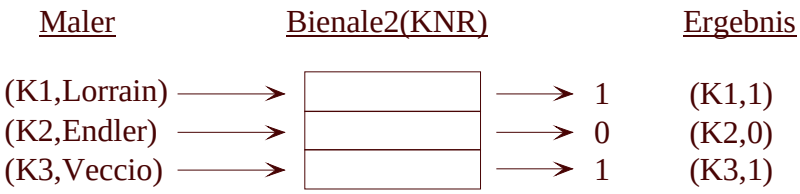
- Angenommen, die Relation Maler besitzt 3 Tupel (verkürzt):

KNR	Name
K1	Lorrain
K2	Endler
K3	Veccio

- die Datei bienale.dat enthält die Werte 'K1' und 'K3'

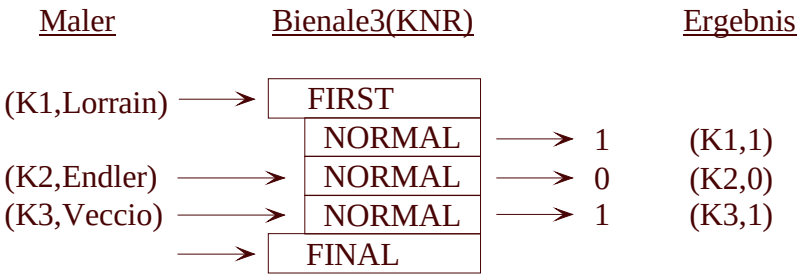
a) NO FINAL CALL

```
SELECT KNR, Bienale2(KNR)
FROM Maler
```



b) FINAL CALL

```
SELECT KNR, Bienale3(KNR)
FROM Maler
```



Fehlerbehandlung

NO FINAL CALL
- nur NORMAL CALL Aufrufe
- sobald ein (NORMAL CALL) Aufruf fehlschlägt: kein weiterer Funktionsaufruf

FINAL CALL
- tritt Fehler beim FIRST CALL Aufruf auf: kein weiterer Funktionsaufruf
- tritt Fehler beim NORMAL CALL Aufruf auf: Aufruf mit FINAL CALL und Ende

sql-clause

sql-clause ::=

NO SQL

| *CONTAINS SQL*

| *READS SQL DATA*

- *NO SQL*: Funktion kann keinerlei SQL-Anweisung ausführen

- *CONTAINS SQL*: Funktion darf keine lesenden und schreibenden SQL-Anweisungen enthalten
- *READS SQL DATA*: kann lesende SQL-Anweisungen enthalten

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <sqlenv.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void SQL_API_FN Einrichtung
    (
    char   *InENR,
    char   *OutTyp,
    short  *nullInENR,
    short  *nullOutTyp,
    SQLUDF_TRAIL_ARGS
    )
{
    EXEC SQL BEGIN DECLARE SECTION;
        char   in_enr[4];
        char   enr[4];
    EXEC SQL END DECLARE SECTION;

    struct sqlca sqlca;

    if(*nullInENR == -1)
    {
        *nullOutTyp=-1;
        return;
    }
    *nullOutTyp=0;
    strcpy(in_enr,InENR);

    EXEC SQL SELECT ENR
         INTO :enr
         FROM Museum
         WHERE ENR = :in_enr;

    if(SQLCODE == 100)
        strcpy(OutTyp,"Galerie");
    else
        strcpy(OutTyp,"Museum");
    return;
}
```

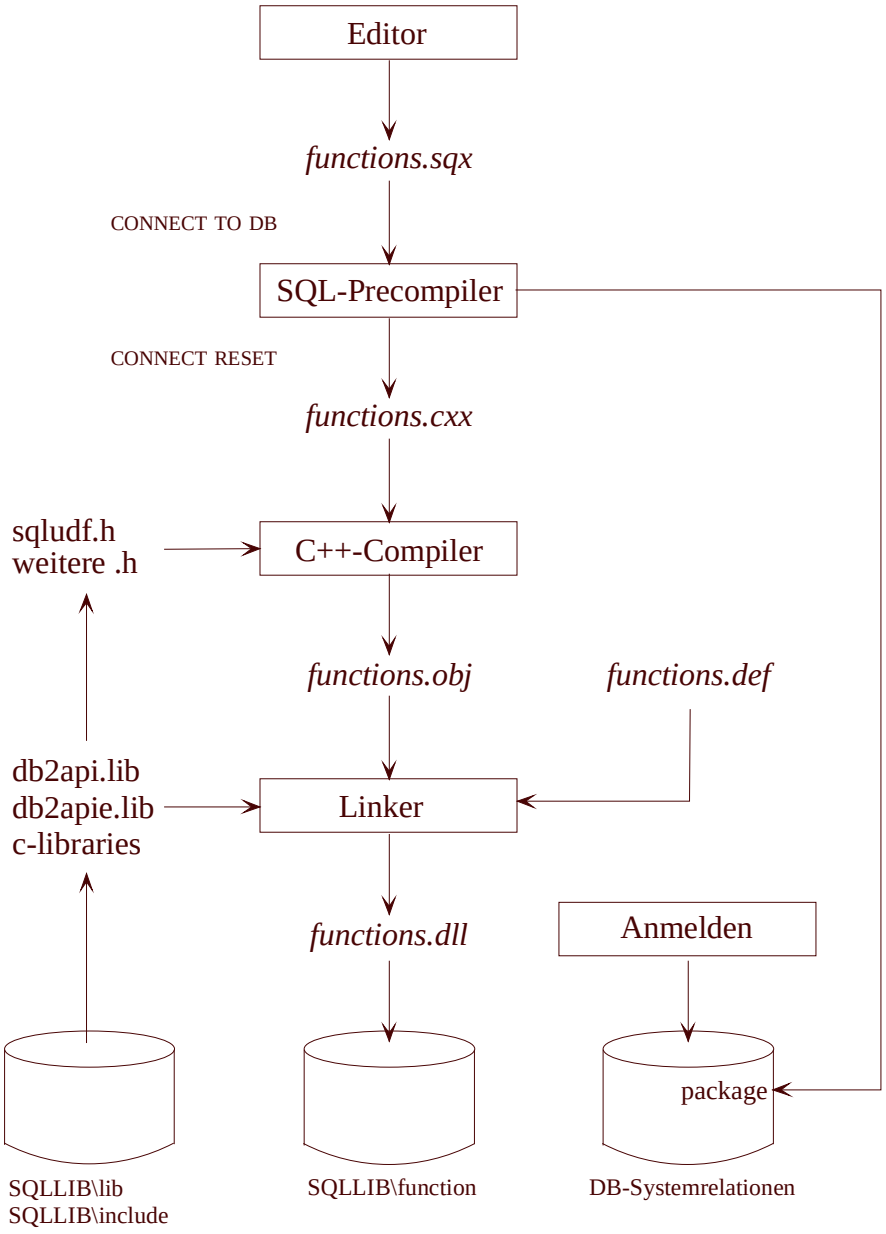
Funktionsanmeldung

```
CREATE FUNCTION Einrichtung(VARCHAR(3))
RETURNS VARCHAR(10)
EXTERNAL NAME '...\Beispiel\beispiel!Einrichtung'
LANGUAGE C
PARAMETER STYLE SQL
CALLED ON NULL INPUT
DETERMINISTIC
READS SQL DATA;
```

Funktionsaufruf-Beispiele

```
SELECT BNR, Titel, Einrichtung(ENR) AS Einrichtung
FROM Bild
```

Übersetzungsprozeß von Funktionen mit eingebettetem SQL



Schritte von der Erstellung bis zur Ausführung:

1. Quelltext editieren
2. Datenbankverbindung herstellen
3. Quelltext SQL-precompilieren/binden
4. Datenbankverbindung lösen
5. C-Quelltext compilieren
6. Binden

Batch-Datei zur Anwendungserstellung: bldmsSQLudf.bat

```
db2 connect to kunst_db
db2 prep %1.sqx
db2 connect reset
cl -Zi -Od -c -W2 -DWIN32 -MD %1.cxx
link -debug -out:%1.dll -dll %1.obj db2api.lib -def:%1.def
```

3.1.2 Externe Tabellenfunktionen

3.1.2.1 Tabellenausdrücke

Syntax

query-specification ::=

SELECT [ALL | DISTINCT] select-list

FROM table-reference [{ , table-reference }...]

[WHERE search-condition]

. . .

select-list ::=

***

| *select-sublist [{ , select-sublist }...]*

select-sublist ::=

value-expression [[AS] column-name]

| *table-name.**

| *range-variable.**

table-reference ::=

table-name [range-variable]

| *(query-expression) [AS] range-variable*

| *joined-table*

| *TABLE (function-name ([table-function-parameter]) [AS] range-variable*

table-function-parameter ::=

value-expression [{ , value-expression }]

- Funktion *function-name* liefert eine Menge von Tupeln zurück.
- Diese Ergebniswerte werden dann innerhalb der SQL-Anweisung wie eine temporäre Tabelle behandelt.
- *table-function-paramete:* Eingabeparameterwerte in die Funktion.

3.1.2.2 Definition einer externen Tabellenfunktion

- werten 0..n Eingabeparameter aus und liefert Tupelmenge (Relation)
- kann wie Datenbankrelation in SQL-Anweisungen aufgerufen werden, d.h. immer dort, wo eine *table-reference* auftreten kann.

Syntax

extern-table-function ::=

void SQL_API_FN function-name (parameter)

{

function-body

}

parameter ::=

[{ input-parameter, }...]

output-parameter, [{ output-parameter, }...]

[{ input-parameter-indicator, }...]

output-parameter-indicator, [{ output-parameter-indicator, }...]

sql-state,

function-name,

specific-name,

error-message,

scratchpad,

call-type,

[dbinfo]

output-parameter / output-parameter-indicator

- mehrere *output-parameter* und entspechende *output-parameter-indicatoren* möglich
- die Werte aller *output-parameter* bilden ein Tupel der Ergebnisrelation der Funktion
- Ein Funktionsaufruf gibt an aufrufende SQL-Anweisung ein Wertetupel zurück

sql-state

- sqlstate = "00000" wird von der Funktion gesetzt, wenn Ergebnistupel zurückgeliefert wird (kann programmtechnisch auch entfallen)
- sqlstate = "02000" wird von der Funktion gesetzt, wenn kein Ergebnistupel mehr zurückgeliefert werden soll --> Abschluß der Ergebnisrelation

call-type

- wird durch das System beim Aufruf der Funktion übergeben
- Wert zeigt an, ob es sich bei der Bildung einer Ergebnistabelle um den ersten (OPEN-CALL), einen weiteren (FETCH-CALL) oder den letzten Funktionsaufruf (CLOSE-CALL) handelt (jeder Funktionsaufruf erzeugt ein Tupel der Ergebnisrelation)
- in Abhängigkeit von der FINAL-CALL-Option kann *call-type* darüber hinaus die Werte FIRST-CALL und FINAL-CALL annehmen

3.1.2.3 Anmelden einer externen Tabellenfunktion

Syntax

create-extern-table-function ::=

CREATE FUNCTION function-name

([parameter-declaration [{ , parameter-declaration }...]])

RETURNS TABLE (column-declaration [{ , column-declaration }...])

EXTERNAL [NAME external-name]

LANGUAGE { C | JAVA | CLR | OLE }

PARAMETER STYLE { DB2GENERAL | SQL }

DISALLOW PARALLEL

[{ option }...]

option ::=

specific-clause
| *fenced-clause*
| *null-call-clause*
| *dbinfo-clause*
| *deterministic-clause*
| *external-action-clause*
| *scratchpad-clause*
| *final-call-clause*
| *sql-clause*

RETURNS TABLE-Klausel

- Definition der Rückgabewerte: hier der Struktur der Ergebnisrelation der Funktion: Attribute und Datentypen
- Im Unterschied zu der Parameterliste der Funktion, die auch leer sein kann, muß die Funktion eine Ergebnisrelation mit wenigstens einem Attribut liefern

column-declaration ::=

column-name sql-data-type [AS LOCATOR]

- Ergebnisrelation kann 1 bis 255 Attribute besitzen

final-call-clause

<final-call-clause>::=

NO FINAL CALL
| *FINAL CALL*

- im Unterschied zu skalaren Funktionen kann der *call-type*-Parameter folgende Werte enthalten:

SQLUDF_TF_FIRST	(Wert: -2)	beim ersten Funktionsaufruf
SQLUDF_TF_OPEN	(Wert: -1)	öffnet die Ergebnisrelation
SQLUDF_TF_FETCH	(Wert: 0)	holt das nächste Tupel der Ergebnisrelation
SQLUDF_TF_CLOSE	(Wert: 1)	schließt die Ergebnisrelation

SQLUDF_TF_FINAL (Wert: 2) letzter Funktionsaufruf

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <stdio.h>
#include <string.h>

void SQL_API_FN Bienale
    (

    char   *OutKNR,
    short  *nullOutKNR,
    SQLUDF_TRAIL_ARGS_ALL
    )
{
    *nullOutKNR = 0;
    char   knr[4];
    struct save_area_struct
    {
        FILE *datei;
    } *save_area;

    save_area=(save_area_struct *)(sqludf_scratchpad->data);
    switch(*sqludf_call_type)
    {
        case SQLUDF_TF_OPEN:
            save_area->datei=fopen("...\\Beispiel\\bienale.dat","r");
            break;
        case SQLUDF_TF_FETCH:
            if (fscanf(save_area->datei,"%s",knr)!=EOF)
                strcpy(OutKNR,knr);
            else
                strcpy(sqludf_sqlstate,"02000");
            break;
        case SQLUDF_TF_CLOSE:
            fclose(save_area->datei);
            break;
    }
    return;
}
```

Funktionsanmeldung

```
CREATE FUNCTION Bienale()
RETURNS TABLE (KNR VARCHAR(3))
EXTERNAL NAME '...\Beispiel\beispiel!Bienale'
```

```
LANGUAGE C
PARAMETER STYLE SQL
DISALLOW PARALLEL
CALLED ON NULL INPUT
DETERMINISTIC
EXTERNAL ACTION
SCRATCHPAD;
```

Funktionsaufruf-Beispiele

```
SELECT x.Bezeichnung
FROM Galerie x, Bild y
WHERE x.ENR=y.ENR
AND y.KNR IN
    (SELECT z.KNR
     FROM TABLE(Bienale()) z)
```

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <stdio.h>
#include <string.h>

void SQL_API_FN Bienale
    (
    char   *OutKNR,
    short  *nullOutKNR,
    SQLUDF_TRAIL_ARGS_ALL
    )
{
    *nullOutKNR = 0;
    char   knr[4];
    struct save_area_struct
    {
        FILE *datei;
    } *save_area;

    save_area=(save_area_struct *)(sqludf_scratchpad->data);
    switch(*sqludf_call_type)
    {
        case SQLUDF_TF_FIRST:
            save_area->datei=fopen("...\\Beispiel\\bienale.dat","r");
            break;
        case SQLUDF_TF_OPEN:
            rewind(save_area->datei);
            break;
        case SQLUDF_TF_FETCH:
```

```
            if (fscanf(save_area->datei,"%s",knr)!=EOF)
                strcpy(OutKNR,knr);
            else
                strcpy(sqludf_sqlstate,"02000");
            break;
        case SQLUDF_TF_CLOSE:
            break;
        case SQLUDF_TF_FINAL:
            fclose(save_area->datei);
            break;
    }
    return;
}
```

Funktionsanmeldung

```
CREATE FUNCTION Bienale()
RETURNS TABLE (KNR VARCHAR(3))
EXTERNAL NAME '...\Beispiel\beispiel!Bienale'
LANGUAGE C
PARAMETER STYLE SQL
DISALLOW PARALLEL
CALLED ON NULL INPUT
DETERMINISTIC
EXTERNAL ACTION
SCRATCHPAD
FINAL CALL;
```

Funktionsaufruf-Beispiele

```
SELECT x.Bezeichnung
FROM Galerie x, Bild y
WHERE x.ENR=y.ENR
AND y.KNR IN
    (SELECT z.KNR
     FROM TABLE(Bienale()) z)
```

Laufzeitverhalten von FINAL-CALL- und NO-FINAL-CALL-Funktionen

- NO- FINAL-CALL-Funktion: hier Bienale1()
- FINAL-CALL-Funktion: hier Bienale2()
- die Datei bienale.dat enthält die Werte 'K1' und 'K3'
- die Relation Maler enthält vereinfacht die Tupel:

KNR	Name
K1	Lorrain
K2	Endler
K3	Veccio

EXISTS-Anfragen

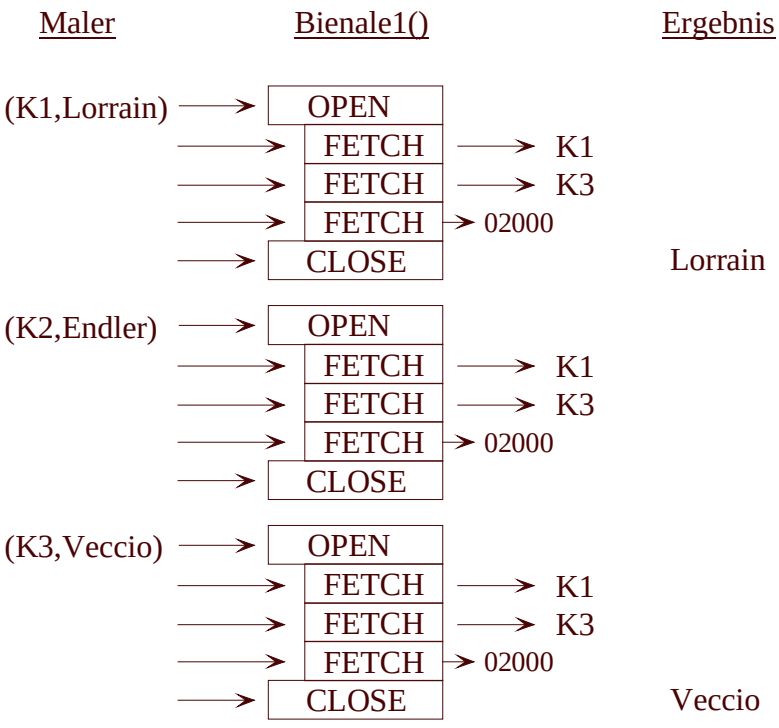
```
SELECT x.Name
FROM Maler x
WHERE EXISTS
  (SELECT *
   FROM TABLE(Bienale()) y
   WHERE x.KNR=y.KNR);
```

NOT-EXIST-Anfrage

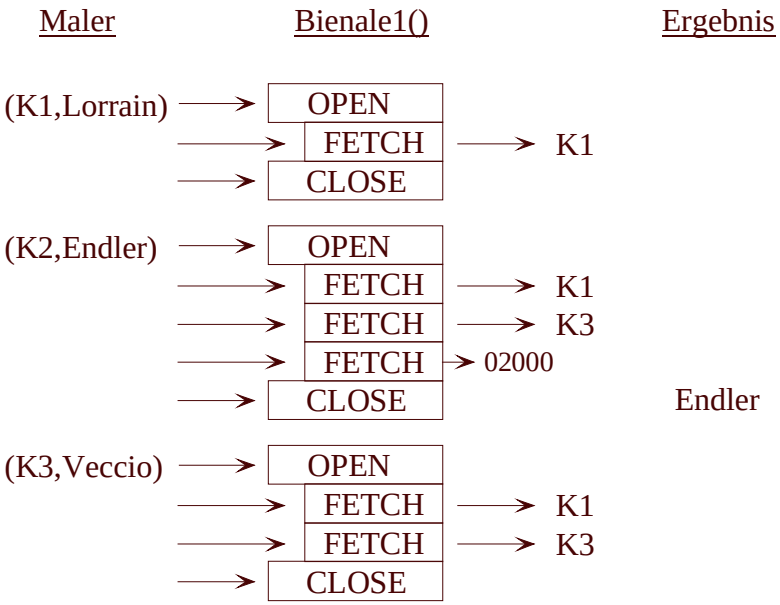
```
SELECT x.Name
FROM Maler x
WHERE NOT EXISTS
  (SELECT *
   FROM TABLE(Bienale()) y
   WHERE x.KNR=y.KNR);
```

Laufzeitverhalten der NO-FINAL-CALL-Funktion Bienale1()

a) bei den EXISTS-Anfragen

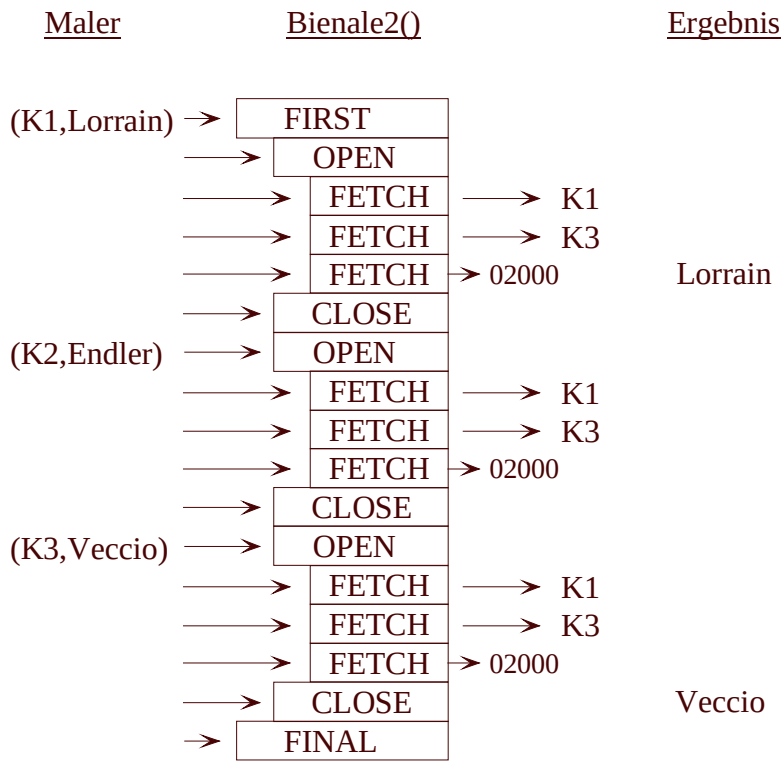


b) bei den NOT-EXISTS-Anfragen

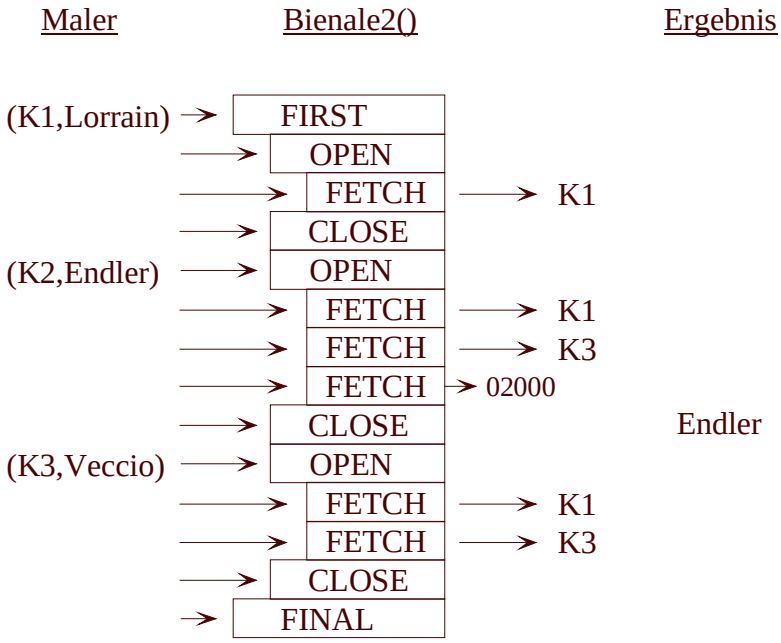


Laufzeitverhalten der FINAL-CALL-Funktion Bienale2()

a) bei den EXISTS-Anfragen



b) bei den NOT-EXISTS-Anfragen



Fehlerbehandlung

NO FINAL CALL

- schlägt OPEN Aufruf fehl: kein weiterer Funktionsaufruf
- schlägt FETCH Aufruf fehl: Aufruf mit CLOSE und Ende

FINAL CALL

- schlägt FIRST Aufruf fehl: kein weiterer Funktionsaufruf
- schlägt OPEN Aufruf fehl: Aufruf FINAL und Ende
- schlägt FETCH Aufruf fehl: Aufruf CLOSE, FINAL und Ende
- schlägt CLOSE Aufruf fehl: Aufruf FINAL und Ende

sql-clause

<sql-clause> ::=

NO SQL

| *CONTAINS SQL*

| *READS SQL DATA*

BEISPIEL

Funktionsdefinition

```
#include <sqludf.h>
#include <sqlenv.h>
#include <stdio.h>
#include <string.h>
#include <sqlca.h>

void SQL_API_FN Einrichtungen
    (
    char  *OutENR,
    char  *OutBezeichnung,
    char  *OutSitz,
    short *nullOutENR,
    short *nullOutBezeichnung,
    short *nullOutSitz,
    SQLUDF_TRAIL_ARGS_ALL
    )
{
    EXEC SQL BEGIN DECLARE SECTION;
        char  enr[4];
        char  bezeichnung[21];
        char  sitz[16];
        short sitz_ind=0;
    EXEC SQL END DECLARE SECTION;

    struct sqlca sqlca;

    EXEC SQL DECLARE cur CURSOR FOR
        SELECT ENR, Bezeichnung, Sitz
        FROM Museum
        UNION
        SELECT ENR, Bezeichnung, Sitz
        FROM Galerie;

    switch(*sqludf_call_type)
    {
        case SQLUDF_TF_OPEN:
            EXEC SQL OPEN cur;
            break;
        case SQLUDF_TF_FETCH:
            EXEC SQL FETCH cur INTO :enr, :bezeichnung, :sitz :sitz_ind;
            if(SQLCODE == 100)
            {
                strcpy(sqludf_sqlstate,"02000");
                break;
            }
            strcpy(OutENR,enr);
            strcpy(OutBezeichnung,bezeichnung);
```

```
            if(sitz_ind==0)
                strcpy(OutSitz,sitz);
            *nullOutSitz=sitz_ind;
            break;
        case SQLUDF_TF_CLOSE:
            EXEC SQL CLOSE cur;
            break;
    }
    return;
}
```

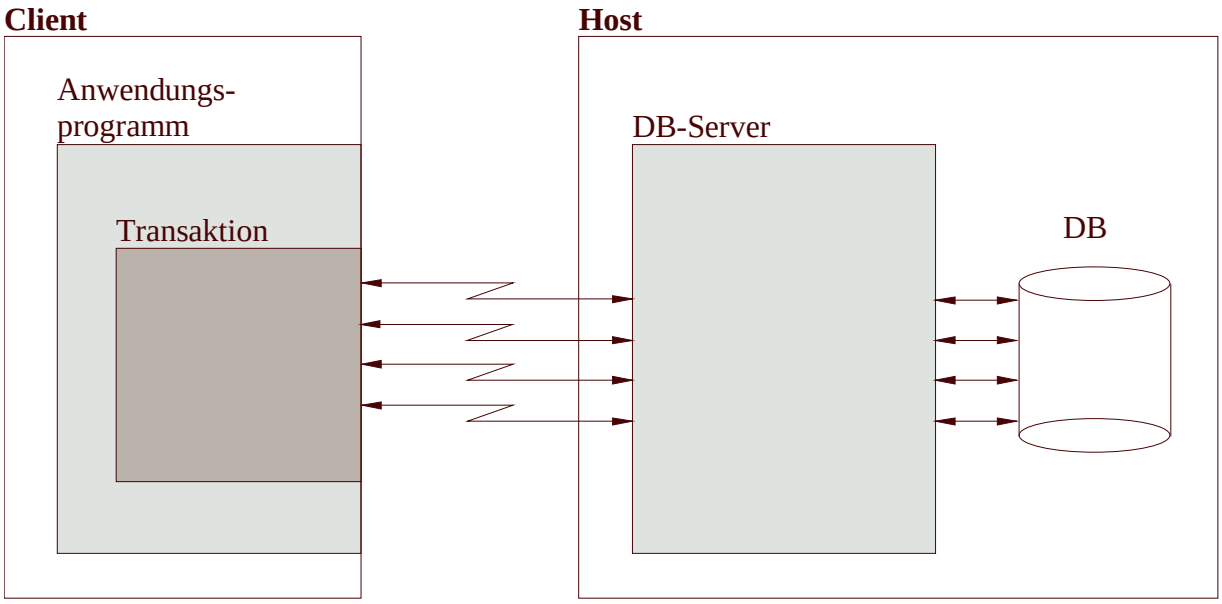
Funktionsanmeldung

```
CREATE FUNCTION Einrichtungen()
RETURNS TABLE (ENR VARCHAR(3), Bezeichnung VARCHAR(20), Sitz
VARCHAR(15))
EXTERNAL NAME '...\Beispiel\beispiel!Einrichtungen'
LANGUAGE C
PARAMETER STYLE SQL
DISALLOW PARALLEL
DETERMINISTIC
SCRATCHPAD
READS SQL DATA;
```

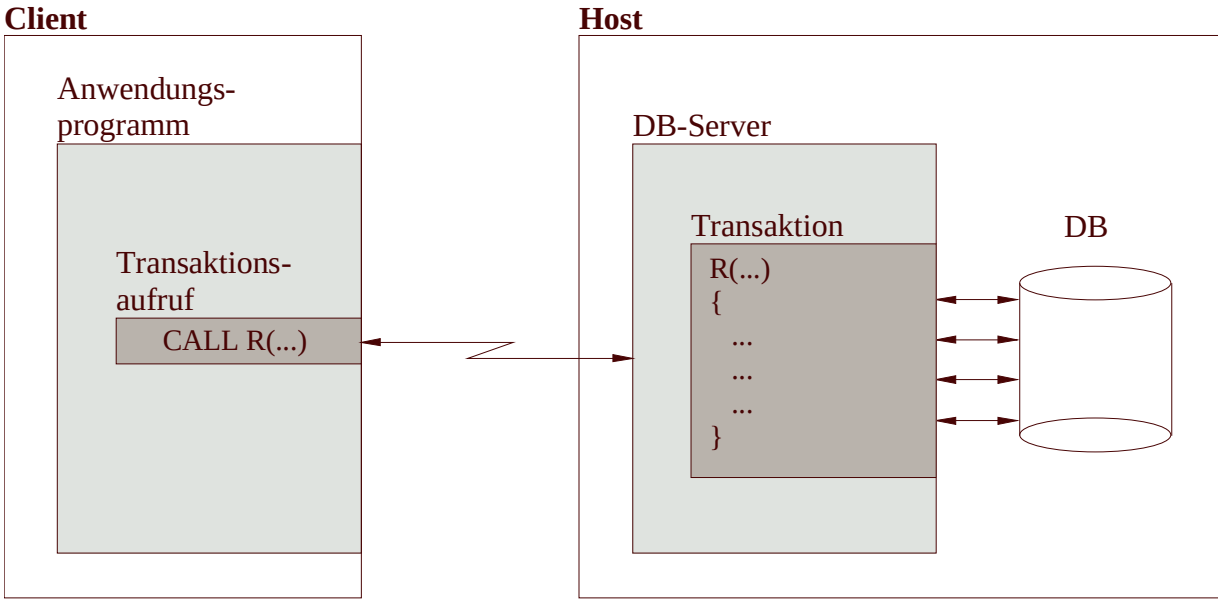
Funktionsaufruf-Beispiele

```
SELECT x.BNR, x.Titel, y.Bezeichnung, y.Sitz
FROM Bild x, TABLE(Einrichtungen()) y
where x.ENR=y.ENR;
```

3.2 Externe Prozeduren



- **stored procedure:**
  Ausführbare DB-Transaktion (Routine), die auf dem DB-Server gespeichert und ausgeführt wird. Der Aufruf der Routine erfolgt vom Client aus.



Vorteile
- Verringerung der Netzbelastung
- Verbesserung der Antwortzeitverhaltens
- Strukturierungsmittel für größere Anwendungen
- einfache Wiederverwendbarkeit häufig benutzter Anfrage

Einschränkungen
- wärend der Prozedur-Ausführung ist keine Kommunikation zwischen Client-Anewendung und Server-Prozedur möglich
- stored procedures können nur auf eine vom Client bereits hergestellte DB-Verbindung aufsetzen.

1.2.1 Definition einer externen Prozedur

Syntax

extern-procedure ::=

SQL_API_RC SQL_API_FN procedure-name (parameter-list)

{

procedure-body

}

parameter

- sind immer von der Form: *c-data-type * variable-pointer*
- Parameter können Ein- oder Aus- oder Ein-/Ausgabe-Parameter sein. Von welcher Art sie sind, wird in der Prozeduranmeldung (CREATE PROCEDURE) festgelegt

procedure-body

- Prozedur kann stat. und dyn. ESQL enthalten, auch CLI-Funktionen
- Prozedur muß mindestens eine return-Anweisung enthalten
- exit ist nicht erlaubt
- die Parameter der Prozedur selbst können nicht als Host-Variable in ESQL-Anweisungen verwendet werden

SQL_API_RC
- Return-Code: wegen Kompatibilität zu alten Versionen

3.2.2 Anmelden einer externen Prozedur

Syntax

create-extern-procedure ::=

CREATE PROCEDURE procedure-name

([parameter-declaration [{ , parameter-declaration }...]])

EXTERNAL [NAME external-name]

LANGUAGE { C | JAVA | COBOL | CLR | OLE }

PARAMETER STYLE { parameter-style }

[{ option }...]

option ::=

specific-clause
| *fenced-clause*
| *null-call-clause*
| *dbinfo-clause*
| *deterministic-clause*
| *external-action-clause*
| *sql-clause*
| *program-type-clause*
| *result-set-clause*

parameter-declaration

parameter-declaration ::=

[IN | OUT | INOUT] [parameter-name] sql-data-type

- müssen in Anzahl, Reihenfolge und Datentyp den Parametern der Prozedurdefinition entsprechen (wie in der Tabelle SQL-/C-Datentyp bei Funktionsparametern)
- *IN*: Eingabeparameter
  - werden von der aufrufenden Anwendung initialisiert
- *OUT*: Ausgabeparameter
  - Wertezuweisung durch die Prozedur, sind also die Rückgabewerte der Prozedur
- *INOUT*: Ein-/Ausgabeparameter
  - dienen sowohl der Hin- als auch der Rückkommunikation der Prozedur

parameter-style

parameter-style ::=

SQL
| *DB2SQL*
| *GENERAL*
| *GENERAL WITH NULLS*
| *DB2GENERAL*
| *JAVA*

- *parameter-style* bestimmt, in welcher Weise die Argumente der Prozedurdefinition beim Prozeduraufruf interpretiert werden
- Vielfalt der Argumente wegen Kompatibilität zu älteren Versionen und anderen Systemen sowie SQL3-Standard

Prozedurdefinition für PARAMETER STYLE SQL

parameter-list ::=

[{ parameter , }... { indicator , }...]
sql-state,
function-name,
specific-name,
error-message
[, scratchpad [, call-type [, dbinfo]]]

- jeder Parameter besitzt einen Indikator-Parameter

Prozedurdefinition für PARAMETER STYLE DB2SQL

parameter-list ::=

[{ parameter , }... { indicator , }...]
sql-state,
function-name,
specific-name,
error-message
[, dbinfo]

Prozedurdefinition für PARAMETER STYLE GENERAL

parameter-list ::=

[{ parameter , }...] [dbinfo]

Prozedurdefinition für PARAMETER STYLE GENERAL WITH NULLS

parameter-list ::=

[{ parameter , }... parameter-indicator-array] [dbinfo]

PARAMETER STYLE DB2GENERAL und JAVA

- nur für JAVA-Prozeduren

sql-clause

<sql-clause> ::=

NO SQL
| *CONTAINS SQL*
| *READS SQL DATA*
| *MODIFIES SQL DATA*

- neu hier: *MODIFIES SQL DATA*

program-type-clause

program-type-clause ::=

PROGRAM TYPE { SUB | MAIN }

result-set-clause

result-set-clause ::=

DYNAMIC RESULT SET { 0 | int-value }

3.2.3 Aufruf einer externen Prozedur

call-statement

call-statment ::=

CALL procedure-name ([expression [{ , expressiont] }])

- jeder Ausdruck (*expression)* ist ein Aufrufargument und korrespondiert mit den in der Prozeduranmeldung (CREATE PROCEDURE) aufgeführten IN/OUT/INOUT-Parametern
- werden Prozeduren aus ESQL-Programmen mittels EXEC SQL CALL ... aufgerufen, sind die Aufrufargumente im einfachsten Fall Host-Variablen der Syntax:

host-variable ::=

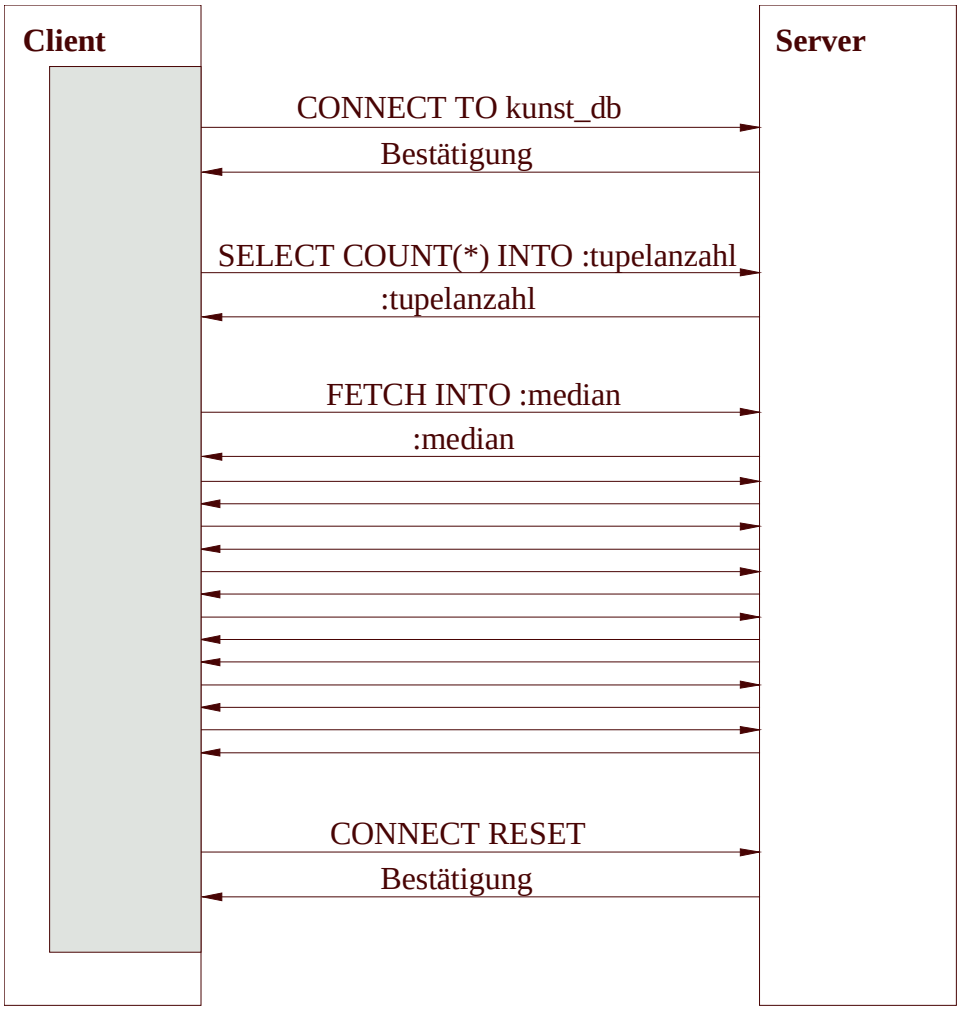
: variable-name [[INDICATOR] : variable-name]

- IN und INOUT Parameter müssen vor Aufruf initialisiert werden
- vor CALL-Aufruf muß die Verbindung zur DB, für die die Prozedur definiert worden ist, hergestellt worden sein (EXEC SQL CONNECT ...)

BEISPIEL (Teil 1)

```
#include <stdlib.h>
#include <stdio.h>
#include <sqlenv.h>
int main()
{
   EXEC SQL INCLUDE SQLCA;

   EXEC SQL BEGIN DECLARE SECTION;
      short  median = 0;
      char   enr[4];
      long   tupelanzahl;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL CONNECT TO kunst_db;

   printf("ENR: "); scanf("%s",enr);

   EXEC SQL SELECT COUNT(*)
      INTO :tupelanzahl
      FROM Bild
      WHERE ENR=:enr;

   if(!tupelanzahl)
```

```
    {
        printf("keine Bilder");
getchar();getchar();
        return 0;
    }

    EXEC SQL DECLARE cur CURSOR FOR
        SELECT Wert
        FROM Bild
        WHERE ENR=:enr
        ORDER BY Wert;

    EXEC SQL OPEN cur;

    long  leftmedian;
    long  i=0;
    while (i++ <= tupelanzahl/2)
    {
        leftmedian=median;
        EXEC SQL FETCH cur INTO :median;
    }
    if ((tupelanzahl % 2)==0)
        median=(median+leftmedian)/2;

    EXEC SQL CLOSE cur;

    printf("Median: %d\n", median);

    EXEC SQL CONNECT RESET;
    return 0;
}
```



BEISPIEL (Teil 2)

Prozedurdefinition auf den SERVER

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqlca.h>
#include <sqludf.h>

SQL_API_RC SQL_API_FN Zentralwert
    (
    char      Enr[4],
    short     *Median,
    short     nullinds[2]
    )
{
    struct sqlca sqlca;

    EXEC SQL BEGIN DECLARE SECTION;
       short median = 0;
```

```
        char   enr[4];
        long   tupelanzahl=0;
    EXEC SQL END DECLARE SECTION;

    strcpy(enr,Enr);

    EXEC SQL SELECT COUNT(*)
        INTO :tupelanzahl
        FROM Bild
        WHERE ENR=:enr;
    if(!tupelanzahl)
    {
        nullinds[1]=-1;
        return 0;
    }

    EXEC SQL DECLARE cur CURSOR FOR
        SELECT Wert
        FROM Bild
        WHERE ENR=:enr
        ORDER BY Wert;

    EXEC SQL OPEN cur;

    long  leftmedian;
    long  i=0;
    while (i++ <= tupelanzahl/2)
    {
        leftmedian=median;
        EXEC SQL FETCH cur INTO :median;
    }
    if ((tupelanzahl % 2)==0)
        median=(median+leftmedian)/2;

    EXEC SQL CLOSE cur;

    *Median=median;
    nullinds[1]=0;

    return 0;
}
```

Prozeduranmeldung auf dem SERVER

```
CREATE PROCEDURE Zentralwert(IN ENR VARCHAR(3), OUT Median SMALLINT)
EXTERNAL NAME '...Beispiel\server!Zentralwert'
LANGUAGE C
PARAMETER STYLE GENERAL WITH NULLS
READS SQL DATA
FENCED;
```

Funktionsaufruf-Anwendungsprogramm auf dem CLIENT

```
#include <stdlib.h>
#include <stdio.h>
#include <sqlenv.h>
int main()
{
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      char     enr[4];
      short    enr_ind=0;
      short    median=0;
      short    median_ind=0;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL CONNECT TO kunst_db;

   printf("ENR: "); scanf("%s",enr);

   EXEC SQL CALL Zentralwert(:enr :enr_ind, :median :median_ind);

   if(median_ind==-1)
      printf("keine Bilder");
   else
      printf("Median: %d\n", median);

   EXEC SQL CONNECT RESET;
getchar();getchar();
   return 0;
}
```

4 SQL-Routinen

SQL-Routinen
Routinen, die ausschließlich in prozedural erweitertem SQl geschrieben sind (PL/SQL)

- SQL-Routinen bestehen in der Regel aus einer Folge von *SQL-routine-statement*s. Es gilt:

  SQL-routine-statement ::= *SQL-statement**

  wobei *SQL-statement** die Menge aller SQL-Anweisungen (einschließlich der folgenden Erweiterungen) ist vermindert um eine Anzahl spezieller DDL-Anweisungen ist (CREATE ..., CONNECT, ...)

4.1 Prozedurale Erweiterungen von SQL

Syntax

SQL-routine-statement ::=

SQL-procedure-statement

| *SQL-function-statement*

- *SQL-procedure-statement*: Anweisungen, die in einer SQL-Prozedur erlaubt sind
- *SQL-function-statement*: Anweisungen, die in einer SQL-Funktion erlaubt sind
- es gilt:
  Menge der *SQL-function-statement* $\subset$ Menge der *SQL-procedure-statement*

2.1.1 SET-Variable-Anweisung

Syntax (gekürzt)

set-variable-statement ::=

SET sql-variable-name = { expression | NULL }

| *SET (sql-variable-name [{ , sql-variable-name }...])*
= ({ expression | NULL } [{ , { expression | NULL } }...])

| *SET (sql-variable-name [{ , sql-variable-name }...]) = (row-select)*

| *SET parameter-name = { expression | NULL }*

- weist einer Variablen einen Wert zu
- *sql-variable-name* müssen vorher mittels DECLARE-Anweisung deklariert worden sein

- in SQL-Routinen wird nicht - wie in externen Routinen - zwischen Variablen und Indikatorvariablen unterschieden.
- In SQL-Routinen kann der NULL-Wert direkt zugewiesen werden

BEISPIEL

```
SET leftmedian = median;
SET (mittelwert, median) = ( summe/anzahl, NULL);
SET (mittelwert) =
    (SELECT AVG(Wert)
     FROM Bild);
```

4.1.2 IF-Anweisung

Syntax

if-statement ::=

IF search-condition THEN { SQL-routine-statement ; }...

[{ ELSEIF search-condition THEN { SQL-routine-statement ; }... }...]

[ELSE { SQL-routine-statement ; }...]

END IF

BEISPIEL

```
IF in_prozent = 5 THEN
    UPDATE Bild
    SET Wert = Wert*1.05;
ELSEIF in_prozent = 10 THEN
    UPDATE Bild
    SET Wert = Wert*1.10;
ELSE
    UPDATE Bild
    SET Wert = Wert*1.01;
END IF;
```

4.1.3 CASE-Anweisung

Syntax

case-statement ::=

CASE { searched-when-clause | simple-when-clause } END CASE

simple-when-clause ::=

expression1 { WHEN expression2 THEN { SQL-procedure-statement ; }... }...

[ELSE { SQL-procedure-statement ; }...]

searched-when-clause ::=

{ WHEN search-condition THEN { SQL-procedure-statement ; }... }...

[ELSE { SQL-procedure-statement ; }...]

- *simple-when-clause*: Wert von *expression1* wird mit Wert von *expression2* verglichen. Bei Übereinstimmung wird THEN-Zweig abgearbeitet
- *searched-when-clause*: *search-condition* wird überprüft. Wenn wahr, dann THEN-Zweig abgearbeitet.
- *case-statement* darf nur in SQL-Prozeduren verwendet werden, nicht in SQL-Funktionen

BEISPIEL (simple-when-clause)

```
CASE in_prozent
    WHEN 5 THEN
        UPDATE Bild
        SET Wert = Wert*1.05;
    WHEN 10 THEN
        UPDATE Bild
        SET Wert = Wert*1.10;
    ELSE
        UPDATE Bild
        SET Wert = Wert*1.01;
END CASE;
```

BEISPIEL (searched-when-clause)

```
CASE
    WHEN in_prozent = 5 THEN
        UPDATE Bild
        SET Wert = Wert*1.05;
    WHEN in_prozent = 10 THEN
        UPDATE Bild
        SET Wert = Wert*1.10;
    ELSE
        UPDATE Bild
        SET Wert = Wert*1.01;
END CASE;
```

4.1.4 WHILE-Anweisung

Syntax

while-statement ::=

[label :]

WHILE search-condition DO { SQL-routine-statement ; }... END WHILE

[label]

- Wiederholung mit Vorbedingung
- *label*: Marke für das WHILE-Statement. Kann in den *SQL-routine-statements* des WHILE für LEAVE (Abbruch des WHILE) oder ITERATE verwendet werden.

BEISPIEL

```
WHILE i <= tupelanzahl/2 DO
    SET leftmedian = median;
    FETCH cur INTO median;
    SET i = i + 1;
END WHILE;
```

4.1.5 REPEAT-Anweisung

Syntax

repeat-statement ::=

[label :]

REPEAT { SQL-routine-statement ; }... UNTIL search-condition END REPEAT

[label]

- Wiederholung mit Nachbedingung
- *repeate-statement* darf nur in SQL-Prozeduren verwendet werden, nicht in SQL-Funktionen

BEISPIEL

```
REPEAT
    SET leftmedian = median;
    FETCH cur INTO median;
    SET i = i + 1;
UNTIL  i = tupelanzahl/2
END REPEAT;
```

4.1.6 LOOP-Anweisung mit LEAVE und ITERATE

Syntax

loop-statement ::=

[label :]

LOOP { SQL-routine-statement ; }... END LOOP

[label]

- Wiederholung ohne Bedingung
- *loop-statement* darf nur in SQL-Prozeduren verwendet werden, nicht in SQL-Funktionen

Syntax

leave-statement ::=

LEAVE label

- bewirkt das Verlassen einer Schleife mit der Marke *label*
- *LEAVE* kann nur in Schleifen auftreten, für die eine Marke *label* gesetzt worden ist, prizipiell in LOOP, WHILE, REPEAT, FOR

Syntax

iterate-statement ::=

ITERATE label

- bewirkt die Fortsetzung der Abarbeitung am beginn der mit *label* markierten Schleife
- *ITERATE* kann nur in Schleifen auftreten, für die eine Marke *label* gesetzt worden ist, prizipiell in LOOP, WHILE, REPEAT, FOR

BEISPIEL

- vorher: DECLARE cur CURSOR FOR
  SELECT wert
  FROM bild
  ORDER BY wert

```
mw_loop:
LOOP
    FETCH cur INTO wert;
    IF wert < median THEN
      anzahl = anzahl + 1;
      summe = summe + wert;
```

```
      ITERATE mw_loop;
    ELSE
      mittelwert = summe / anzahl;
      LEAVE mw_loop;
    END IF;
END LOOP;
```

4.1.7 FOR-Anweisung

***Syntax** (eingeschränkt)*

for-statement ::=

[label :]

FOR for-loop-name AS select-statement DO { SQL-routine-statement ; }... END FOR

[label]

- führt *SQL-routine-statements* für jedes Tupel der Ergebnisrelation des *select-statement* aus.

BEISPIEL

```
for_loop:
FOR  tupel  AS
        SELECT enr, bezeichnung, sitz
        FROM museum
        UNION
        SELECT enr, bezeichnung, sitz
        FROM galerie
    DO
        INSERT INTO einrichtung
        VALUES (tupel.enr, tupel.bezeichnung, tupel.sitz);
END FOR;
```

4.1.8 Variablen-Deklaration

Syntax

SQL-variable-declaration ::=

DECLARE SQL-variable-name

[{ , SQL-variable-name }] SQL-data-type [DEFAULT { NULL | constant }]

- definiert eine oder mehrere Variablen und weist ihnen einen Datentyp zu

- alle Variablennamen werden intern in Großbuchstaben umgewandelt (keine Unterscheidung zwischen Klein- und Großbuchstaben wie in C)
- Variablennamen müssen innerhalb einer Routine (eigentlich innerhalb compound-statement, später) eindeutig sein
- hat eine Verbundanweisung (statements zwichen BEGIN und END, später) eine Marke *label*, dann kann auf die Variablen auch durch *label.SQL-variable-name* zugegriffen werden

BEISPIEL

```
DECLARE v1, v2  SMALLINT DEFAULT NULL;
DECLARE v3 CHAR(10);
SET v3 = 'Museum';
```

4.1.9 Condition-Deklaration

Syntax (eingeschränkt)

condition-declaration ::=

DECLARE condition-name CONDITION FOR SQLSTATE string-constant

- jede SQL-Anweisung gibt eine Zustandsmeldung über ihre Abarbeitung zürück
- *string-constant* ist eine aus 5 Zeichen bestehende Konstante, die eine Zustandsmeldung darstellt
- *string-constant* darf nicht '00000' sein

4.1.10 Handler-Deklaration

Syntax)

handler-declaration ::=

DECLARE handler-type HANDLER FOR condition SQL-procedure-statement

handler-type ::=

CONTINUE

| *EXIT*

| *UNDO*

condition ::=

general-condition [{ , general-condition }]

| *specific-condition [{ , specific-condition }]*

general-condition

SQLEXCEPTION

| *SQLWARNING*

| *NOT FOUND*

specific-condition ::=

condition-name

| *SQLSTATE string-constant*

- *handler-type*: Aktion, die ein *handler* nach Auftreten eines Zustandes *condition* ausführt
- *condition*: Zustand, der nach Abarbeitung einer SQL-Anweisung einer Routine auftritt
- tritt nach Abarbeitung einer SQL-Anweisung in der Routine eine *condition* auf und ist ein *handler* dafür definiert, dann wird die Ablaufsteuerung an den *handler* übergeben und
  1. der *handler* führt die Aktion *handler-type* aus,
  2. der *handler* führt das *SQL-procedure-statement* aus

Zustände (general-condition)

SQLEXCEPTION
- SQLCODE < 0

NOT FOUND
- SQLCODE = 100
- SQLSTATE = '02xxx', x weitere Ziffern

SQLWARNING
- SQLCODE > 0 aber <> 100
- SQLSTATE = '01xxx', x weitere Ziffern

condition-name
- Name eines Zustandes, der vorher in der Condition-Deklaration definiert wurde

SQLSTATE string-constant
- Definition eines Zustandes in der Handler-Definition

Aktionen in Abhängigkeit von Handler-type

CONTINUE
- Handler führt *SQL-procedure-statement* aus und danach weitere Programmausführung hinter der Anweisung, die mit der *condition* den Aufruf des Handlers bewirkte

EXIT

- Handler führt *SQL-procedure-statement* aus und danach weitere Programmausführung mit dem Ende des Blockes (BEGIN .. END).

UNDO

- Handler setzt alle bisherigen ändernden SQL-Anweisungen des Routinen-Blockes, in dem er definiert wurde, zurück, führt *SQL-procedure-statement* aus und danach weitere Programmausführung mit dem Ende des Blockes (BEGIN .. END).

BEISPIEL

```
DECLARE text CHAR(80);
DECLARE no_table CONDITION FOR SQLSTATE '42704';
BEGIN
   DECLARE EXIT HANDLER FOR no_table SET text = 'Relation Einrichtung existiert nicht';
   DROP TABLE Einrichtung;
   SET text = 'Relation Einrichtung gelöscht';
END;
INSERT INTO Test VALUES (text);
```

BEISPIEL

```
BEGIN ATOMIC
   DECLARE UNDO HANDLER FOR SQLSTATE '42704' INSERT INTO Test
                                          VALUES ('Relation Einrichtung existiert nicht');
   INSERT INTO Test VALUES ('Relation Einrichtung gelöscht');
   DROP TABLE Einrichtung;
END;
```

4.1.11 SIGNAL- und RESIGNAL-Anweisung

Syntax

signal-statement ::=

SIGNAL SQLSTATE state [message]

message ::=

SET MESSAGE_TEXT = string-expression

| *(string-expression)*

- *SIGNAL*-Anweisung löst eine Fehlerbedingung mit dem *SQLSTATE state* aus
- *state*: 5 Byte lange Zeichenkette zur Rückgabe des Fehlerzustandes
- *message*: max. 70 Byte langer String, der bei Rückkehr zurückgegeben wird

- *string-expression*: enthält den Nachrichtentext, der zurückgegeben wird

Syntax)

resignal-statement ::=

RESIGNAL [SQLSTATE state] [message]

message ::=

SET MESSAGE_TEXT = { string-expression | variable-name }

- dient dazu, eine durch *SIGNAL* ausgelöste Fehlerbedingung zu überschreiben
- *resignal-statement* darf nur in SQL-Prozeduren verwendet werden, nicht in SQL-Funktionen

BEISPIEL

```
DECLARE Abbruchschalter SMALLINT;

DECLARE text VARCHAR(70);
SET text = 'Es wurde SIGNAL 7500'||CHAR(Abbruchschalter)||'gesendet';
BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE '75002'
       RESIGNAL SET MESSAGE_TEXT = text;
    DECLARE EXIT HANDLER FOR SQLSTATE '75001'
       RESIGNAL SET MESSAGE_TEXT = text;
    IF (Abbruchschalter = 1) THEN
       SIGNAL SQLSTATE '75001';
    ELSEIF (Abbruchschalter = 2) THEN
       SIGNAL SQLSTATE '75002';
    END IF;
END;
```

4.1.12 RETURN-Anweisung

Syntax

return-statement ::=

RETURN [return-value]

return-value ::=

expression

| NULL

| query-expression

- *expression*: Werteausdrücke, wie sie schon bei der Einführung von SQL behandelt wurden
- *query-expression*: in der Syntax der Handbücher *fullselect* genannt. Ist eine vollständige SELECT-Anweisung
- Regeln
  - Funktionen müssen einen *return-value* zurückgeben
  - Tabellenfunktionen können keine *expression* zurückgeben
  - Prozeduren können nur eine *expression* zurückgeben. Der Rückgabetyp muß INTEGER sein
  - ist der Rumpf einer Tabellenfunktion eine Verbundanweisung, dann muß RETURN am Ende der Verbundanweisung stehen

4.2 SQL-Funktionen

4.2.1 Definition einer SQL-Funktion

Syntax

create-SQL-function ::=

CREATE FUNCTION function-name

([parameter-declaration [{ , parameter-declaration }...]])

RETURNS return-types

[{ option }...]

SQL-function-body

option ::=

specific-clause

| *language-clause*

| *null-call-clause*

| *deterministic-clause*

| *external-action-clause*

| *sql-clause*

parameter-declaration

parameter-declaration ::=

parameter-name sql-data-type

- auf die Parameter kann innerhalb des Funktionsrumpfes direkt zugegriffen werden
- Parameter, wie alle SQL-Variablen, können auch NULL-Werte beseitzen

RETURNS return-types

return-types ::=

sql-data-type

| *TABLE (column-name sql-data-type [{ , column-name sql-data-type }...])*

| *ROW (column-name sql-data-type [{ , column-name sql-data-type }...])*

- spezifiziert, ob es sich um eine skalare Funktion (*return-types ::= sql-data-type)* oder Tabellenfunktion (*return-types ::= TABLE ...)* handelt

language-clause

language-clause ::=

LANGUAGE SQL

sql-clause

sql-clause ::=

CONTAINS SQL

| *READS SQL DATA*

| *MODIFIES SQL DATA*

SQL-function-body

SQL-function-body ::=

RETURN SQL-statement

| SQL-function-compound-statement

- *SQL-statement*: SQL-Anweisung, die entweder
  - bei skalaren Funktionen einen Wert liefert (value, expression, case, select)
  - bei Tabellenfunktionen Tupelmenge liefert (select)
- *SQL-function-compound-statement*, spezielle Verbundanweisung für Funktionen
- **Regeln**:
  - ist der Funktionsrumpf eine Verbundanweisung, muß diese wenigstens eine RETURN-Anweisung enthalten, die auch ausgeführt werden muß
  - eine Tabellenfunktion darf dabei nur eine einzige RETURN-Anweisung enthalten, die als letzte Anweisung in der Verbundanweisung stehen muß

BEISPIEL

Funktionsdefinition

```
CREATE FUNCTION Dollar(Euro SMALLINT)
RETURNS SMALLINT
LANGUAGE SQL
RETURN Euro*1.3;
```

Funktionsaufruf

```
SELECT BNR, Titel, Dollar(Wert) AS Wert
FROM Bild;
```

BEISPIEL

Funktionsdefinition

```
CREATE FUNCTION Einrichtung()
RETURNS TABLE (ENR VARCHAR(3), Name VARCHAR(20), Sitz VARCHAR(15))
LANGUAGE SQL
RETURN
   SELECT ENR, Bezeichnung, Sitz
   FROM Museum
   UNION
   SELECT ENR, Bezeichnung, Sitz
   FROM Galerie;
```

Funktionsaufruf

```
SELECT DISTINCT x.ANR, x.A_Titel, y.Name, y.Sitz
FROM Ausstellung x, TABLE (Einrichtung()) y
WHERE x.ENR=y.ENR;
```

4.2.2 Verbundanweisung für SQL-Funktionen

Syntax

SQL-function-compound-statement ::=

[label:]

BEGIN ATOMIC

[{ SQL-variable-declaration ; }...]

[{ condition-declaration ; }...]

{ SQL-function-statement ; }...

END

[label]

label

- Marke für einen Block,
- kann zur Spezifikation von Variablennamen dienen: *label . SQL-variable-name*

ATOMIC

- Verbundanweisung stellt eine Transaktion dar, d.h.

SQL-function-statement

- eingeschränkter Satz von SQL-Anweisungen, die innerhalb des Verbundes verwendet werden können:

SQL-function-statement ::=

query-expression

| insert-statement

| delete-statement

| update-statement

| call-statement

| set-variable-statement

| if-statement

| for-statement

| while-statement

| iterate-statement

| leave-statement

| signal-statement

BEISPIEL

Funktionsdefinition

```
CREATE FUNCTION Rang(Nummer INTEGER)
RETURNS SMALLINT
LANGUAGE SQL
BEGIN ATOMIC
   DECLARE i INTEGER DEFAULT 0;
   FOR  tupel  AS
      SELECT Wert
      FROM Bild
      ORDER BY Wert DESC
   DO
      SET i=i+1;
      IF (i=Nummer) THEN
         RETURN Wert;
      END IF;
   END FOR;
   SIGNAL SQLSTATE '75000'
      SET MESSAGE_TEXT='Rangnummer groesser als die Anzahl der Bilder';
END
@
```

Funktionsaufrufe

```
SELECT *
FROM Bild
WHERE Wert=Rang(3)

SELECT *
FROM Bild
WHERE Wert=Rang(4)

SELECT *
FROM Bild
WHERE Wert=Rang(20)
```

BEISPIEL

Funktionsdefinition

```
CREATE FUNCTION Drei_Bestseller()
RETURNS TABLE (KNR VARCHAR(3), Gesamtwert SMALLINT)
```

```
LANGUAGE SQL
BEGIN ATOMIC
   DECLARE KNR1, KNR2, KNR3 VARCHAR(3);
   SET KNR1 =
      (SELECT KNR
       FROM Bild
       WHERE KNR IS NOT NULL
       GROUP BY KNR
       HAVING SUM(Wert) >= ALL
         (SELECT SUM(Wert)
          FROM Bild
          WHERE KNR IS NOT NULL
          GROUP BY KNR));
   SET KNR2 =
      (SELECT KNR
       FROM Bild
       WHERE KNR IS NOT NULL
       AND KNR<>KNR1
       GROUP BY KNR
          HAVING SUM(Wert) >= ALL
         (SELECT SUM(Wert)
          FROM Bild
          WHERE KNR IS NOT NULL
          AND KNR<>KNR1
          GROUP BY KNR));
   SET KNR3 =
      (SELECT KNR
       FROM Bild
       WHERE KNR IS NOT NULL
       AND KNR<>KNR1
       AND KNR<>KNR2
       GROUP BY KNR
          HAVING SUM(Wert) >= ALL
         (SELECT SUM(Wert)
          FROM Bild
          WHERE KNR IS NOT NULL
          AND KNR<>KNR1
          AND KNR<>KNR2
          GROUP BY KNR));
   RETURN
      SELECT KNR, SUM(Wert)
      FROM Bild
      WHERE KNR=KNR1
      OR KNR=KNR2
      OR KNR=KNR3
      GROUP BY KNR;
END
@
```

Funktionsaufruf

```
SELECT x.*, y.Gesamtwert
FROM Maler x, TABLE(Drei_Bestseller()) y
WHERE x.KNR=y.KNR
```

Schritte zur Anmeldung

1. Quellfile als SQL-Skript in einem beliebigen Verzeichnis, Namenskonvention in DB2:
   <script-name>.db2

2. ins Befehlsfenster DB2 CLP gehen

3. Datenbank aufrufen
   db2 connect to <database-name>

4. SQL-Funktion anmelden
   db2 -td@ -vf <script-name>.db2

5. Datenbank abmelden
   db2 connect reset

4.3 SQL-Prozeduren

4.3.1 Definition einer SQL-Prozedur

Syntax

create-SQL-procedure ::=

CREATE PROCEDURE procedure-name

([parameter-declaration [{ , parameter-declaration }...]])

[{ option }...]

SQL-procedure-body

option ::=

specific-clause

| *language-clause*

| *null-call-clause*

| *deterministic-clause*

| *external-action-clause*

| *sql-clause*

| *result-set-clause*

parameter-declaration

parameter-declaration ::=

[IN | OUT | INOUT] parameter-name sql-data-type

- im Unterschied zu externen Prozeduren muß hier der *parameter-name* angegeben werden

language-clause

language-clause ::=

LANGUAGE SQL

sql-clause

sql-clause ::=

CONTAINS SQL

| *READS SQL DATA*

| *MODIFIES SQL DATA*

SQL-procedure-body

SQL-procedure-body ::=

SQL-procedure-statement

| *SQL-procedure-compound-statement*

- *SQL-procedure-statement*: SQL-Anweisung, die in einer SQL-Prozedur verwendet werden darf
- es können prinzipiell alle SQL-Anweisungen in einer SQL-Prozedur verwendet werden, außer z.B. *CONNECT, DISCONNECT, ALTER, REVOKE* und einige andere, siehe Handbuch

4.3.2 Verbundanweisung für SQL-Prozeduren

Syntax

SQL-procedure-compound-statement ::=

[label:]
BEGIN [NOT ATOMIC | ATOMIC]
[{ SQL-variable-declaration ; }...]
[{ condition-declaration ; }...]
[{ return-codes-declaration ; }...]
[{ statement-declaration ; }...]
[{ declare-cursor-statement ; }...]
[{ handler-declaration ; }...]
{ SQL-procedure-statement ; }...
END
[label]

return-codes-declaration

return-codes-declaration ::=

DECLARE return-code-variable

return-code-variable ::=

SQLSTATE CHAR(5) [DEFAULT { '00000' | string-const }]
| *SQLCODE INTEGER [DEFAULT { 0 | integer-const }]*

- Definition der beiden speziellen Variablen *SQLSTATE* oder *SQLCODE*, die automatisch nach jeder SQL-Anweisung vom System einen entsprechenden Wert erhalten
- Variablen können hier mit Default-Werten belegt werden
- *SQLSTATE* und *SQLCODE* können nur einmal für eine Prozedur definiert werden und dann auch nur im äußersten Verbund-Block

statement-declaration

statement-declaration ::=

DECLARE statement-name STATEMENT

- Definition einer Variablen für eine Anweisungszeichenkette

declare-cursor-statement

declare-cursor-statement ::=

DECLARE cursor-name CURSOR [WITH HOLD]

[WITH RETURN [TO CALLER | TO CLIENT]]

FOR { select-statement | statement-name }

- *WITH RETURN*: Cursor wird als result-set für die SQL-Prozedur verwendet

BEISPIEL

```
DECLARE stmt STATEMENT;
DECLARE cur CURSOR FOR stmt;
SET stmt = 'SELECT Name, Geburt FROM Maler';
OPEN cur;
...
```

BEISPIEL

```
--
-- SQL-Prozedur zur Berechnung des Zentralwertes der Werte der Bilder einer Einrichtung
--
CREATE PROCEDURE Zentralwert_SQL
        (IN IN_ENR VARCHAR(3), OUT Median SMALLINT)
LANGUAGE SQL
BEGIN
   DECLARE Leftmedian, Tupelanzahl SMALLINT;
   DECLARE i SMALLINT DEFAULT 0;

   DECLARE cur CURSOR FOR
      SELECT Wert
      FROM Bild
      WHERE ENR=IN_ENR
      ORDER BY Wert;

   DECLARE EXIT HANDLER FOR NOT FOUND
      SET Median = NULL;

   SELECT COUNT(x.ENR)
   INTO Tupelanzahl
   FROM (SELECT * FROM Galerie UNION SELECT * FROM Museum) x
   WHERE x.ENR = IN_ENR;
   IF (Tupelanzahl = 0) THEN
      SIGNAL SQLSTATE '75000'
         SET MESSAGE_TEXT = 'ENR nicht vorhanden';
```

```
    END IF;

    SELECT COUNT(*)
    INTO Tupelanzahl
    FROM Bild
    WHERE ENR=IN_ENR;

    SET Median = 0;
    OPEN cur;
    loop_label:
    LOOP
        SET Leftmedian = Median;
        FETCH cur INTO Median;

        IF (i < (Tupelanzahl / 2)) THEN
            SET i = i + 1;
        ELSE
            LEAVE loop_label;
        END IF;
    END LOOP;
    IF (MOD(Tupelanzahl,2) = 0) THEN
        SET Median = (Median + Leftmedian) / 2;
    END IF;

END
@
```

4.3.3 Übersetzen, Binden, Anmelden von SQL-Routine

Schritte

vor Übersetzung des SQL-Skripts folgende Einstellung für Visual-C-Compiler erforderlich:

```
db2set DB2_SQLROUTINE_COMPILER_PATH=
                    "D:\Programme\ Microsoft Visual Studio\vc98\bin\vcvars32.bat"
```

muß nur einmal gemacht werden

1. Quellfile als SQL-Skript in einem beliebigen Verzeichnis, Namenskonvention in DB2:
   <script-name>.db2

2. ins Befehlsfenster DB2 CLP gehen

3. Datenbank aufrufen
   db2 connect to <database-name>

4. SQL-Skript übersetzen, binden und anmelden
   db2 -td@ -vf <script-name>.db2

5. Datenbank abmelden
   db2 connect reset

4.3.4 Aufruf einer SQL-Prozedur

Syntax

call-statement ::=

CALL procedure-name ([argument [{ , argument] }])

argument ::=

expression

| *NULL*

| *?*

- SQL-Prozeduren können in ESQL-, CLI-Programmen und anderen Routinen oder direkt aus der Befehlszentrale/Befehlsfenster heraus aufgerufen werden
- jedes Aufrufargument korrespondiert mit den in der Prozeduranmeldung (CREATE PROCEDURE) aufgeführten IN/OUT/INOUT-Parametern
- *?*: Angabe des Fragezeichens beim Aufruf in der Befehlszentrale für OUT-Parameter (Rückgabewerte).

BEISPIEL
Aufruf aus Befehlszentrale

```
CALL Zentralwert_SQL ('E1', ?)

CALL Zentralwert_SQL ('E3', ?)

CALL Zentralwert_SQL ('E9', ?)
```

Aufruf aus ESQL-Programm

```
int main()
{
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      char  enr[4];
      short enr_ind=0;
      short median;
      short median_ind=0;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL CONNECT TO kunst_db;

   printf("ENR: "); scanf("%s",enr);
   EXEC SQL CALL Zentralwert_SQL(:enr :enr_ind, :median :median_ind);

   if(!strcmp(sqlca.sqlstate,"75000"))
      printf("%s",sqlca.sqlerrmc);
```

```
    else
        if(median_ind==-1)
            printf("keine Bilder");
        else
            printf("Median: %d\n", median);

    EXEC SQL CONNECT RESET;
getchar();getchar();
    return 0;
}
```

BEISPIEL

```
CREATE PROCEDURE Erzeuge(IN IN_ENR VARCHAR(3))
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
    DECLARE stmt VARCHAR(200);
    DECLARE table VARCHAR(20);
    DECLARE Tupelzahl INTEGER DEFAULT 0;
    DECLARE EXIT HANDLER FOR NOT FOUND
        SIGNAL SQLSTATE '38600'
        SET MESSAGE_TEXT = 'Die Einrichtung besitzt keine Bilder.';

    DECLARE EXIT HANDLER FOR SQLSTATE '42710'
        SIGNAL SQLSTATE '38600'
        SET MESSAGE_TEXT = 'Die Einrichtung-Relation existiert bereits.';

    SELECT COUNT(x.ENR)
    INTO Tupelzahl
    FROM (SELECT * FROM Museum UNION SELECT * FROM Galerie) x
    WHERE x.ENR=IN_ENR;
    IF (Tupelzahl=0) THEN
        SIGNAL SQLSTATE '38600'
        SET MESSAGE_TEXT = 'Die Einrichtung existiert nicht.';
    END IF;

    SET table = 'Einrichtung_'||IN_ENR;
    SET stmt = 'CREATE TABLE '||table||'(BNR VARCHAR(3), Titel VARCHAR(30),'||
                'Maler VARCHAR(20), Wert SMALLINT)';
    PREPARE s FROM stmt;
    EXECUTE s;

    SET stmt = 'INSERT INTO '||table||' SELECT x.BNR, x.Titel, y.Name, x.Wert '||
                'FROM Bild x, Maler y WHERE x.KNR=y.KNR AND x.ENR=?';
    PREPARE s FROM stmt;
    EXECUTE s USING IN_ENR;
END
@
```