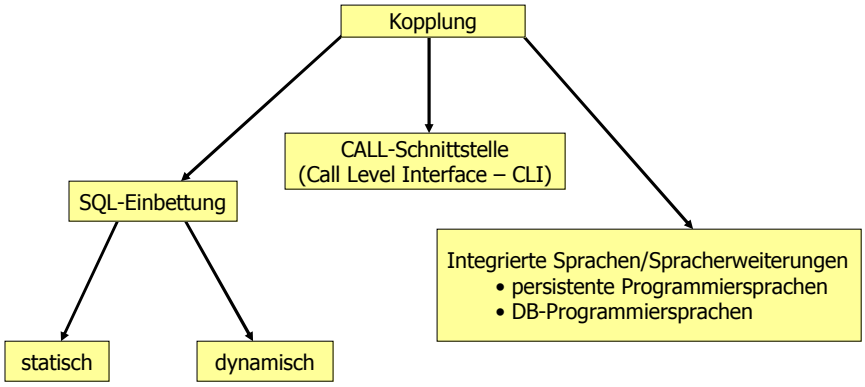
Kapitel 2
Anwendungsprogrammierschnittstelle und Anfrageverarbeitung

Inhalt
Kopplung von DB- und Programmiersprache
Eingebettetes statisches SQL,
SQL/PSM, Dynamisches SQL
Call-Level-Interface (CLI), ODBC, JDBC, SQLJ
Aspekte der Auswertung und Optimierung von DB-Anfragen

Kopplung (1)

- **Übersicht**



Kopplung (2)

- **Übersicht (Forts.)**
  - **Call-Schnittstelle** (prozedurale Schnittstelle, CLI)
    - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
    - Anwendung enthält lediglich Prozeduraufrufe
  - **Einbettung von SQL** (Embedded SQL)
    - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
    - komfortablere Programmierung als mit CLI
  - **statische Einbettung**
    - Vorübersetzer (Precompiler) wandelt DB-Aufrufe in Prozeduraufrufe um
    - Nutzung der normalen Compiler für umgebendes Programm
    - SQL-Anweisungen müssen zur Übersetzungszeit feststehen
    - im SQL-Standard unterstützte Sprachen: C, COBOL, FORTRAN, Ada, Pascal, Java, ...

Kopplung (3)

- **Übersicht (Forts.)**
  - **dynamische Einbettung:**
    - Konstruktion von SQL-Anweisungen zur Laufzeit
  - **(ältere) Integrationsansätze unterstützten typischerweise „nur"**
    - ein Typsystem
    - Navigation (satz-/objektorientierter Zugriff)
    - wünschenswert sind jedoch Mehrsprachenfähigkeit und deskriptive DB-Operationen (mengenorientierter Zugriff)
- **Relationale Anwendungsprogrammierschnittstellen (APIs)**
  - bieten Mehrsprachenfähigkeit und deskriptive DB-Operationen,
  - erfordern jedoch Maßnahmen zur Überwindung der sog. Fehlanpassung (impedance mismatch): Satzorientierung vs. Mengenorientierung

Kopplung (4)

- **Kernprobleme der API bei konventionellen Programmiersprachen**
  - Konversion und Übergabe von Werten
    (in beide Richtungen: Verwendung von Wirtssprachenvariablen sowohl zur Parametrisierung von DB-Operationen als auch zur Aufnahme von Ergebnisdaten)
  - Mengenorientierung von DB-Operationen
    - Wie und in welcher Reihenfolge werden Zeilen/Sätze dem AP zur Verfügung gestellt?
      - Cursor-Konzept

Eingebettetes statisches SQL (1)

- **C-Beispiel**

```
exec sql include sqlca; /* SQL Communication Area */
main () {
     exec sql begin declare section;
             char    X[3];
             int     GSum;
     exec sql end declare section;
     exec sql connect to dbname;
     exec sql insert into Pers (Pnr, Name) values (4711, 'Ernie');
     exec sql insert into Pers (Pnr, Name) values (4712, 'Bert');
     printf ("Abteilungsnummer: ") ; scanf ( " %s" , X);
     exec sql select sum(Gehalt) into :GSum
             from Pers where Anr = :X;
     /* Es wird nur ein Ergebnissatz zurückgeliefert! */
     printf ("Gehaltssumme: %d\n" , GSum);
     exec sql commit work;
     exec sql disconnect;
}
```

Eingebettetes statisches SQL (2)

- **Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung**
  - eingebettete SQL-Anweisungen werden durch **exec sql** eingeleitet und durch spezielles Symbol (hier ";") beendet, um dem Compiler eine Unterscheidung von anderen Anweisungen zu ermöglichen
  - Verwendung von AP-Variablen in SQL-Anweisungen verlangt Deklaration innerhalb eines ‚**declare section**'-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
  - Kommunikationsbereich **SQLCA** (Rückgabe von Statusanzeigern u.ä.)
  - Übergabe der Werte einer Zeile mit Hilfe der **INTO**-Klausel
    - INTO target-commalist (Variablenliste des Wirtsprogramms)
    - Anpassung der Datentypen (Konversion)
  - Aufbau/Abbau einer Verbindung zu einem DBS: **connect/disconnect**

Cursor-Konzept (1)

- **Zweck**
  - satzweise Abarbeitung von Ergebnismengen
  - Trennung von Qualifikation und Bereitstellung/Verarbeitung von Zeilen
- **Cursor ist ein Iterator,**
  - der einer Anfrage zugeordnet wird und
  - mit dessen Hilfe die Zeilen der Ergebnismenge einzeln *(one tuple at a time)* im Programm bereitgestellt werden
- **Cursor-Deklaration**
  - Syntax

    DECLARE cursor **CURSOR FOR** query
    [ORDER BY order-item-commalist**]**
  - Beispiel

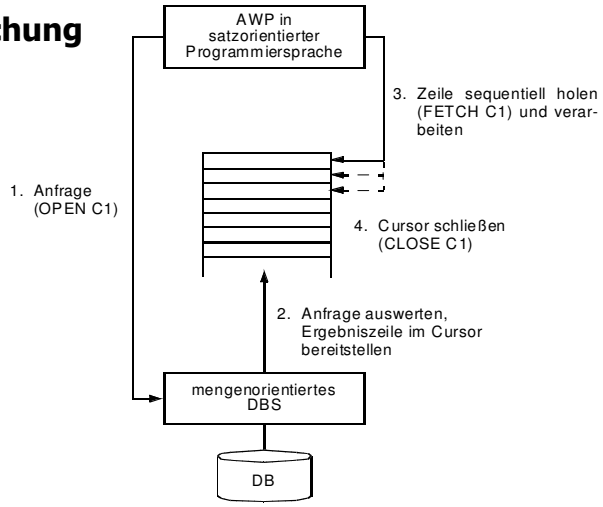
```
DECLARE C1 CURSOR FOR
        SELECT Name, Gehalt, Anr FROM Pers
        WHERE Anr = 'K55' ORDER BY Name;
```

Cursor-Konzept (2)

- **Cursor-Operation (am Beispiel oben)**
  - **Open** C1
  - **Fetch** C1 **INTO** Var1, Var2, Var3
  - **Close** C1
- **Wann wird die Ergebnismenge ermittelt?**
  - ***lazy:*** schritthaltende Auswertung durch das DBS? Verzicht auf eine explizite Zwischenspeicherung ist nur bei einfachen Anfragen möglich.
  - ***eager:*** daher Kopie bei OPEN meist erforderlich (ORDER BY, Join, Aggregat-Funktionen, ...)

Cursor-Konzept (3)

- **Veranschaulichung der Cursor-Schnittstelle**



Cursor-Konzept (4)

- **C-Beispiel (vereinfacht)**

```
exec sql begin declare section;
char X[50], Y[3];
exec sql end declare section;

exec sql declare C1 cursor for
     select Name from Pers where Anr = :Y;

printf("Bitte Abteilungsnummer eingeben: \n");
scanf("%d", Y);

exec sql open C1;
while (sqlcode == OK)
{
     exec sql fetch C1 into :X;
     printf("Angestellter %d\n", X);
}
exec sql close C1;
```

Cursor-Konzept (5)

- **Aktualisierung mit Bezugnahme auf eine Position**
  - Wenn die Zeilen, die ein Cursor verwaltet *(active set),* eindeutig den Zeilen einer Tabelle entsprechen, können sie über Bezugnahme durch den Cursor geändert werden.
  - Syntax

    positioned-update ::= **UPDATE** table **SET** update-assignment-commalist
    WHERE CURRENT OF cursor

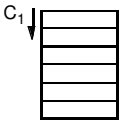
    positioned-delete ::= **DELETE FROM** table
    WHERE CURRENT OF cursor

  - Beispiel

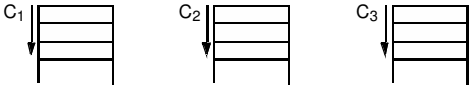
```
while (sqlcode == ok) {
    exec sql fetch C1 into :X;
    ... /* Berechne das neue Gehalt in Z /*
    exec sql update Pers
    set Gehalt = :Z
    where current of C1;
}
```

Cursor-Konzept (6)

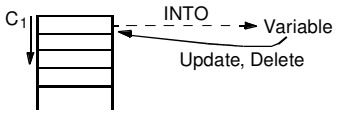
- **SQL-Programmiermodell**
  1. Ein Cursor



  2. Mehrere Cursor
     - Verknüpfung der Zeilen im Programm

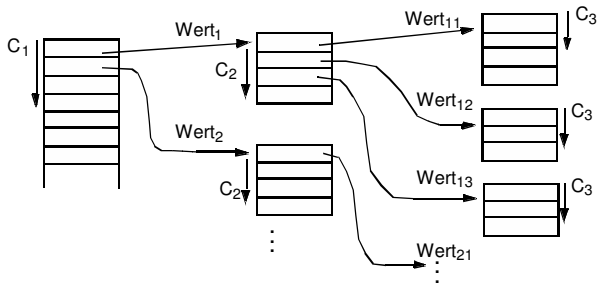


  3. Positionsbezogene Aktualisierung



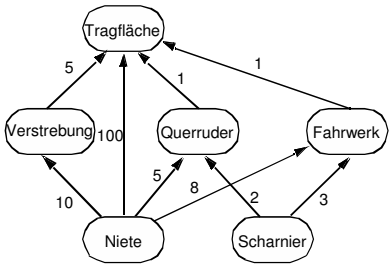
Cursor-Konzept (7)

- **SQL-Programmiermodell (Forts.)**
  4. Abhängige Cursor



Cursor-Konzept (8)

- **SQL-Programmiermodell (Forts.)**
  - Beispiel: Stücklistenauflösung
    - Tabelle Struktur (Otnr, Utnr, Anzahl)
    - Aufgabe: Ausgabe aller Endprodukte sowie deren Komponenten
    - max. Schachtelungstiefe sei bekannt (hier: 2)



Struktur (Otnr,	Utnr,	Anzahl)
T	V	5
T	N	100
T	Q	1
T	F	1
V	N	10
Q	N	5
Q	S	2
F	N	8
F	S	3

Cursor-Konzept (9)

- **SQL-Programmiermodell (Forts.)**
  - Beispiel: Stücklistenauflösung (Forts.)

```
exec sql begin declare section;
   char T0[10], T1[10], T2[10]; int Anz;
exec sql end declare section;

exec sql declare C0 cursor for
   select distinct Otnr from Struktur S1
   where not exists (
      select * from Struktur S2
      where S2.Utnr =  S1.Otnr
    );

exec sql declare C1 cursor for
   select Utnr, Anzahl from Struktur where Otnr = :T0;

exec sql declare C2 cursor for
   select Utnr, Anzahl from Struktur where Otnr = :T1;
...
```

Cursor-Konzept (10)

- **SQL-Programmiermodell (Forts.)**
  - Beispiel: Stücklistenauflösung (Forts.)

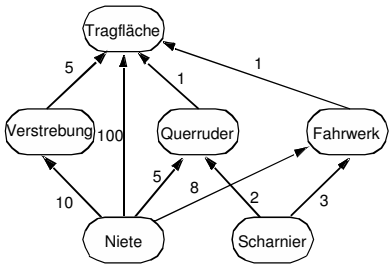
```
...
exec sql open C0;
while (1) {
    exec sql fetch C0 into :T0;
    if (sqlcode == notfound) break;
    printf ("%s\n " , T0);
    exec sql open C1;
    while (2) {
    exec sql fetch C1 into :T1, :Anz;
              if (sqlcode == notfound) break;
              printf ("%s:   %d\n " , T1, Anz);
              exec sql open (C2);
              while (3) {
                        exec sql fetch C2 INTO :T2, :Anz;
                        if (sqlcode == notfound) break;
                        printf ("%s:   %d\n " , T2, Anz);
              }
              exec sql close (C2);
    }
    exec sql close C1;
}
exec sql close (C0);
```


Cursor-Konzept (11)

- **SQL-Programmiermodell (Forts.)**
  - Beispiel: Stücklistenauflösung (Forts.)

Ausgabe:

```
T
    V:  5
          N:  10
    N:  100
    Q:  1
          N:  5
          S:  2
    F:  1
          N:  8
          S:  3
```



Aspekte der Anfrageverarbeitung

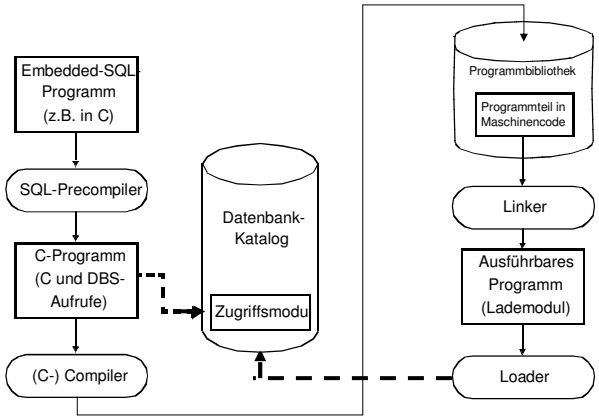
- **Deskriptive mengenorientierte DB-Anweisungen**
  - ‚**Was**-Anweisungen' sind in zeitoptimale Folgen interner DBMS-Operationen umzusetzen
  - bei navigierenden DB-Sprachen bestimmt der Programmierer, **wie** eine Ergebnismenge (abhängig von existierenden Zugriffspfaden) satzweise aufzusuchen und auszuwerten ist
  - **Aber**: Anfrageauswertung/-optimierung des DBMS ist im wesentlichen für die effiziente Abarbeitung verantwortlich!

Wirtsspracheneinbettung und Übersetzung (1)

- **Direkte Einbettung**
  - keine syntaktische Unterscheidung zwischen Programm- und DB-Anweisungen
  - DB-Anweisung wird als Zeichenkette A ins AP integriert, z. B. **exec sql open** C1
  - Verlangt spezielle Maßnahmen bei der AP-Übersetzung: typischerweise Einsatz eines Vorübersetzers (Precompiler)

Einbettung/Übersetzung (2)

- **Von der Übersetzung bis zur Ausführung (bei Einsatz eines Vorübersetzers)**



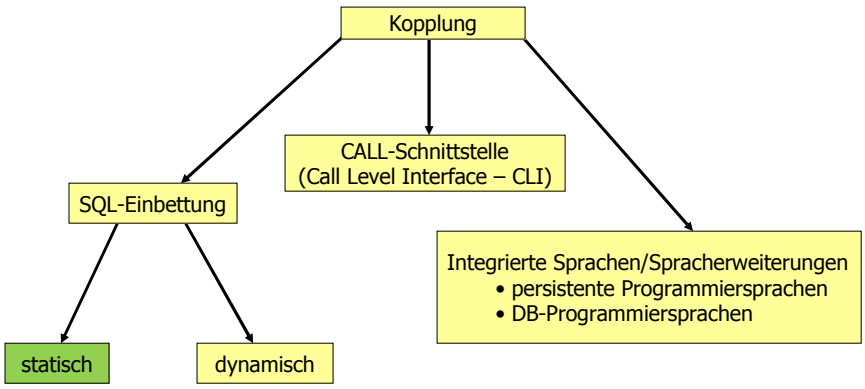
Einbettung/Übersetzung (3)

- **Von der Übersetzung bis zur Ausführung (Forts.)**
  - Vorübersetzung des AP
    - Entfernung aller Embedded-SQL-Anweisungen aus dem Programm (Kommentare)
    - Ersetzung durch Programmiersprachen-spezifische DBS-Aufrufe
    - Erzeugung eines „SQL-freien" Programms in der Programmiersprache
    - DBS-seitige Vorbereitung: Analyse und Optimierung der SQL-Anweisungen und Erstellung eines Zugriffsmoduls im DB-Katalog
  - Übersetzung des AP
    - Umwandlung der Anweisungen der höheren Programmiersprache in Maschinencode (Objektmodul) und Abspeicherung in Objektbibliothek
    - SQL-Anweisungen für Compiler nicht mehr sichtbar

Einbettung/Übersetzung (4)

- **Von der Übersetzung bis zur Ausführung (Forts.)**
  - Binden
    - Zusammenfügen der Objektmodule zu lauffähigem Programm
    - Hinzufügen des SQL-Laufzeitsystems
  - Laden und Ausführen
    - Laden des ausführbaren Programms in den Speicher
    - Anbinden des Zugriffsmoduls aus DB-Katalog und automatische Überprüfung seiner Gültigkeit
    - Programmstart
- **Von zentraler Bedeutung**
  - Anfrageauswertung/-optimierung des DBVS ist wesentlich für die effiziente Abarbeitung

Kopplung



Dynamisches SQL (1)

- **Festlegen/Übergabe von SQL-Anweisungen zur Laufzeit**
  - Benutzer stellt Ad-hoc-Anfrage
  - AP berechnet dynamisch SQL-Anweisung
  - SQL-Anweisung ist aktueller Parameter von Funktionsaufrufen an das DBVS
- **Eigenschaften**
  - Vorbereitung einer SQL-Anweisung kann erst zur Laufzeit beginnen
  - Bindung an das DB-Schema erfolgt zum spätest möglichen Zeitpunkt
    - DB-Operationen beziehen sich stets auf den aktuellen DB-Zustand
    - größte Flexibilität und Unabhängigkeit vom DB-Schema
  - Vorbereitung und Ausführung einer SQL-Anweisung
    - erfolgt typischerweise durch Interpretation (Übersetzung und Code-Generierung sind prinzipiell möglich)
    - Leistungsproblem: wiederholte Ausführung derselben Anweisung

Dynamisches SQL (2)

- **Mehrere Sprachansätze (ähnlicher Funktionalität)**
  - Eingebettetes dynamisches SQL (EDSQL)
  - Call-Level-Interface (CLI)
    - kann ODBC (Open Database Connectivity) implementieren
  - Java Database Connectivity (JDBC)
    - dynamische SQL-Schnittstelle zur Verwendung mit Java
    - 'de facto'-Standard für den Zugriff auf relationale Daten von Java-Programmen aus; Spezifikation der JDBC-Schnittstelle unter http://java.sun.com/products/jdbc
    - JDBC ist gut in Java integriert und ermöglicht einen Zugriff auf relationale Datenbanken in einem objektorientierten Programmierstil
    - JDBC ermöglicht das Schreiben von Java-Applets, die von einem Web-Browser auf eine DB zugreifen können

Dynamisches SQL (3)

- **Dynamischen Ansätzen ist gemeinsam**
  - Zugriff auf Metadaten zur Laufzeit
  - Übergabe und Abwicklung dynamisch (d.h. zur Laufzeit) berechneter SQL-Anweisungen
  - Optionale Trennung von Vorbereitung und Ausführung (Achtung: hier beides zur Laufzeit!)
    - einmalige Vorbereitung ggf. parametrisierter SQL-Anweisungen
    - n-malige Ausführung
  - Möglichkeit der Dynamischen Parameterbindung (Bindung von Programm-Variablen an SQL-Parameter)
  - Verwendung einer flexiblen Deskriptorstruktur, falls Anzahl und Typen der dynamischen Parameter nicht vorab bekannt

Eingebettetes Dynamisches SQL (1)

- **Eigenschaften**
  - Abkürzung: EDSQL
  - unterstützt mehrere Wirtssprachen
  - ist im Stil statischem SQL ähnlicher; wird oft von Anwendungen gewählt, die dynamische und statische SQL-Anweisungen mischen
  - Programme mit EDSQL sind kompakter und besser lesbar als solche mit CLI oder JDBC
  - 4 wesentliche Anweisungen
    - **DECLARE**
    - **PREPARE**
    - **EXECUTE**
    - **EXECUTE IMMEDIATE**
  - SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt
    - Deklaration **DECLARE STATEMENT**
    - Anweisungen enthalten Platzhalter für Parameter (?) statt Programmvariablen

Eingebettetes Dynamisches SQL (2)

- **Trennung von Vorbereitung und Ausführung**
  - **Beispiel:**

```
exec sql begin declare section;
   char   Anweisung [256], X[3];
exec sql end declare section;
exec sql declare SQLanw statement;
/* Zeichenkette kann zugewiesen bzw. eingelesen werden */
Anweisung = 'DELETE FROM Pers WHERE Anr = ?';
/* Prepare-and-Execute optimiert die mehrfache Verwendung
einer dynamisch erzeugten SQL-Anweisung */
exec sql prepare SQLanw from :Anweisung;
exec sql execute SQLanw using 'K51';
scanf (" %s " , X);
exec sql execute SQLanw using :X;
```


Eingebettetes Dynamisches SQL (3)

- **Einmalige Ausführung einer SQL-Anweisung**
  - Beispiel

```
scanf ("%s" , Anweisung);
exec sql execute immediate :Anweisung;
```

- **Cursor-Verwendung**
  - SELECT-Anweisung nicht Teil von DECLARE CURSOR, sondern von PREPARE-Anweisung
  - OPEN-Anweisung (und FETCH) anstatt EXECUTE
  - Beispiel

```
exec sql declare SQLanw statement;
exec sql prepare SQLanw from
                    "SELECT Name FROM Pers WHERE Anr=?" ;
exec sql declare C1 cursor for SQLanw;
exec sql open C1 using 'K51';
...
```

Eingebettetes Dynamisches SQL (4)

- **Dynamische Parameterbindung**
  - Beispiel

```
Anweisung = 'INSERT INTO Pers VALUES (?, ?, …)';
exec sql prepare SQLanw from :Anweisung;
vname = 'Ted';
nname = 'Codd';
exec sql execute SQLanw using :vname, :nname, ...;
```

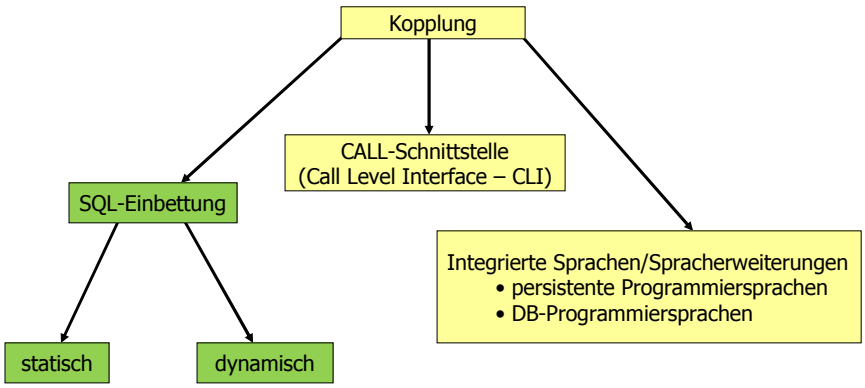
- **Zugriff auf Beschreibungsinformation wichtig**
  - wenn Anzahl und Typ der dynamischen Parameter nicht bekannt ist
  - Deskriptorbereich ist eine gekapselte Datenstruktur, die durch das DBVS verwaltet wird

Eingebettetes Dynamisches SQL (5)

- **Zugriff auf Beschreibungsinformation wichtig (Forts.)**
  - Beispiel

```
Anweisung = 'INSERT INTO Pers VALUES (?, ?, ...)';
exec sql prepare SQLanw from :Anweisung;
exec sql allocate descriptor 'Eingabeparameter';
exec sql describe input SQLanw into
  sql descriptor 'Eingabeparameter';
exec sql get descriptor 'Eingabeparameter' :n = count;
for (i = 1; i < n; i ++) {
   exec sql get descriptor 'Eingabeparameter' value :i
     :attrtyp = type, :attrlänge = length, :attrname = name;
   ...
   exec sql set descriptor 'Eingabeparameter' value :i
     data = :d, indicator = :ind;
}
exec sql execute SQLanw
  using sql descriptor 'Eingabeparameter';
```

Kopplung



SQL/PSM (1)

- **PSM: Persistent Stored Modules**
  - kann als moderne integrierte Sprache verstanden werden
  - zielt auf Leistungsverbesserung vor allem in Client/Server-Umgebung ab
    - Ausführung mehrerer SQL-Anweisungen durch ein EXEC SQL
    - Entwerfen von Routinen mit mehreren SQL-Anweisungen
  - erhöht die Verarbeitungsmächtigkeit des DBS
    - Prozedurale Erweiterungsmöglichkeiten (der DBS-Funktionalität aus Sicht der Anwendung)
    - Einführung neuer Kontrollstrukturen
  - erlaubt reine SQL-Implementierungen von komplexen Funktionen
    - Sicherheitsaspekte
    - Leistungsaspekte
  - ermöglicht SQL-implementierte Klassenbibliotheken (SQL-only)

SQL/PSM (2)

- **Beispiel**
  - Erzeugen einer SQL-Prozedur

```
CREATE PROCEDURE proc1 ( )
{
   BEGIN
   INSERT INTO Pers VALUES (...);
   INSERT INTO Abt VALUES (...);
   END;
}
```

  - Aufruf

```
...
EXEC SQL CALL proc1 ( );
...
```


SQL/PSM (3)

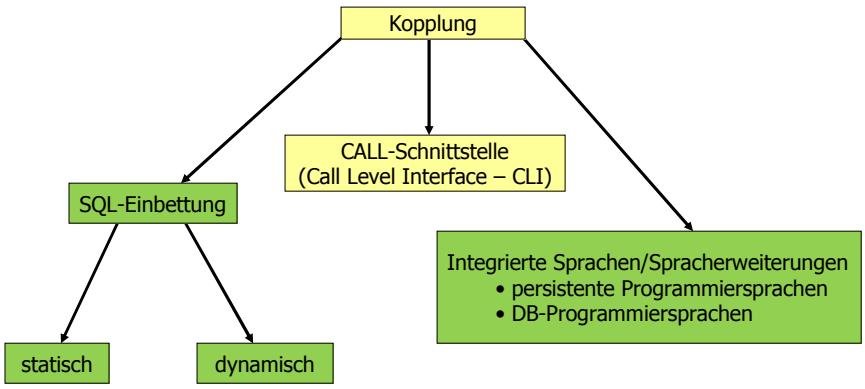
- **Vorteile**
  - vorübersetzte Ausführungspläne werden gespeichert, sind wiederverwendbar
  - Anzahl der Zugriffe des Anwendungsprogramms auf die DB wird reduziert
  - als gemeinsamer Code für verschiedene Anwendungsprogramme nutzbar
  - es wird ein höherer Isolationsgrad der Anwendung von der DB erreicht

SQL/PSM (4)

- **Prozedurale Spracherweiterungen**

• Compound statement	• BEGIN ... END;
• SQL variable declaration	• DECLARE var CHAR (6);
• If statement	• IF subject (var <> 'urgent') THEN ... ELSE ...;
• Case statement	• CASE subject (var) WHEN 'SQL' THEN ... WHEN ...;
• Loop statement	• LOOP <SQL statement list> END LOOP;
• While statement	• WHILE i<100 DO ... END WHILE;
• Repeat statement	• REPEAT ... UNTIL i<100 END REPEAT;
• For statement	• FOR result AS ... DO ... END FOR;
• Leave statement	• LEAVE ...;
• Return statement	• RETURN 'urgent';
• Call statement	• CALL procedure_x (1,3,5);
• Assignment statement	• SET x = 'abc';
• Signal/resignal statement	• SIGNAL divison_by_zero

Kopplung



Call-Level-Interface (1)

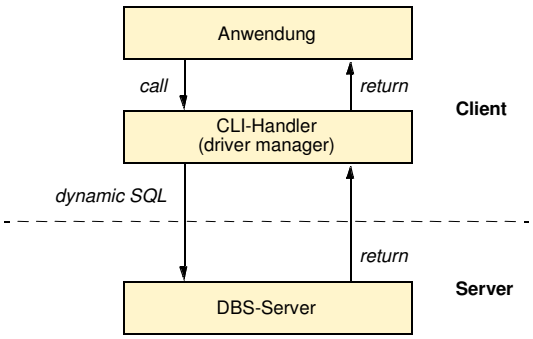
- **Spezielle Form von dynamischem SQL**
  - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
  - Direkte Aufrufe der Routinen einer standardisierten Bibliothek
  - Aufruftechnik
    - DB-Anweisung wird durch expliziten Funktionsaufruf an das Laufzeitsystem des DBS übergeben, z. B. **CALL DBS** ('**open** C1')
    - Verschiedene Formen der Standardisierung: Call-Level-Interface (CLI), JDBC
  - Keine Vorübersetzung (Behandlung der DB-Anweisungen) von Anwendungen
    - Vorbereitung der DB-Anweisung geschieht erst beim Aufruf zur Laufzeit
    - Anwendungen brauchen nicht im Source-Code bereitgestellt werden
    - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools

Call-Level-Interface (2)

- **wird sehr häufig in der Praxis eingesetzt!**
- **CLI-Standardisierung in SQL3**
  - ISO-Standard wurde 1996 verabschiedet
  - starke Anlehnung an ODBC bzw. X/Open CLI
  - Standard-CLI umfasst über 40 Routinen: Verbindungskontrolle, Ressourcen-Allokation, Ausführung von SQL-Befehlen, Zugriff auf Diagnoseinformation, Transaktionsklammerung, Informationsanforderung zur Implementierung

Call-Level-Interface (3)

- **Einsatz typischerweise in Client/Server-Umgebungen**



Call-Level-Interface (4)

- **Vorteile von CLI**
  - Schreiben portabler Anwendungen
  - Systemunabhängigkeit
  - Mehrfache Verbindungen zur selben DB
  - Verbindungen zu mehreren DBS
  - Optimierung des Zugriffs vom/zum Server
- **Kooperation von AP und DBS**
  - maximale gegenseitige Kapselung
  - Zusammenspiel AP/CLI und DBVS ist nicht durch Übersetzungsphase vorbereitet
  - Wahl des DBS zur Laufzeit
  - vielfältige LZ-Abstimmungen erforderlich

Call-Level-Interface (5)

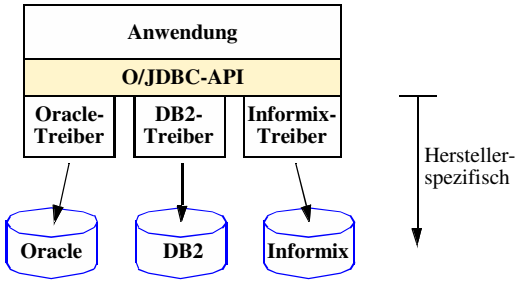
- **Wesentlich: Konzept der Handle-Variablen**
  - Ein Handle ist letztlich eine Programmvariable, die Informationen repräsentiert, die für ein AP durch die CLI-Implementierung verwaltet wird
  - gestattet Austausch von Verarbeitungsinformationen
  - Arten:
    - **Umgebungskennung** repräsentiert den globalen Zustand der Applikation
    - **Verbindungskennung**
      - separate Kennung: n Verbindungen zu einem oder mehreren DBS
      - Freigabe/Rücksetzen von Transaktionen, Isolationsgrad
    - **Anweisungskennung**
      - mehrfache Definition, auch mehrfache Nutzung
      - Ausführungszustand einer SQL-Anweisung; sie fasst Informationen zusammen, die bei statischem SQL in SQLCA, SQLDA und Cursorn stehen
    - **Deskriptorkennung** enthält Informationen, wie Daten einer SQL-Anweisung zwischen DBS und CLI-Programm ausgetauscht werden

CLI-Beispiel:

```
#include "sqlcli.h"
#include <string.h>
. . .
{
SQLCHAR * server;
SQLCHAR * uid;
SQLCHAR * pwd;
HENV henv; // environment handle
HDBC hdbc; // connection handle
HSTMT hstmt; // statement handle
SQLINTEGER id;
SQLCHAR name [51];

/* connect to database */
SQLAllocEnv (&henv);
SQLAllocConnect (henv, &hdbc) ;
if  (SQLConnect (hdbc, server, uid,
    pwd, ...) != SQL_SUCCESS)
    return (print_err (hdbc, ...) ) ;

/* create a table */
SQLAllocStmt (hdbc, &hstmt) ;
{  SQLCHAR create [ ] ="CREATE TABLE
    NameID (ID integer,
    Name varchar (50) )" ;
if  (SQLExecDirect (hstmt, create, ...)
    != SQL_SUCCESS)
    return (print_err (hdbc, hstmt) ) ;
}
/* commit transaction */
SQLTransact (henv, hdbc, SQL_COMMIT);

/* insert row */
{ SQLCHAR insert [ ] ="INSERT INTO
    NameID VALUES (?, ?)" ;
if (SQLPrepare (hstmt, insert, ...) !=
    SQL_SUCCESS)
    return (print_err (hdbc, hstmt) ) ;

SQLBindParam (hstmt, 1, ..., id, ...) ;
SQLBindParam (hstmt, 2, ..., name,
    ...) ;
id = 500; strcpy (name,"Schmidt") ;

if  (SQLExecute (hstmt) != SQL_SUCCESS)
    return (print_err (hdbc, hstmt) ) ; }

/* commit transaction */
SQLTransact (henv, hdbc, SQL_COMMIT) ;
}
```

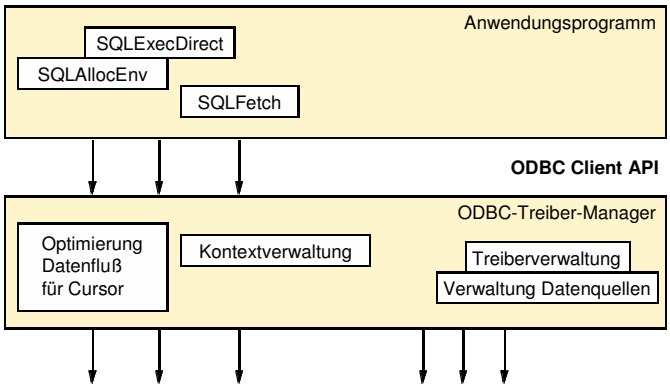
ODBC/JDBC-Überblick

- **Standard-APIs vom Typ CLI (Call Level Interface)**
  - Einheitlicher Zugriff auf Daten
  - Dynamisches, spätes Binden an DBS
  - Gleichzeitiger Zugriff auf mehrere DBS



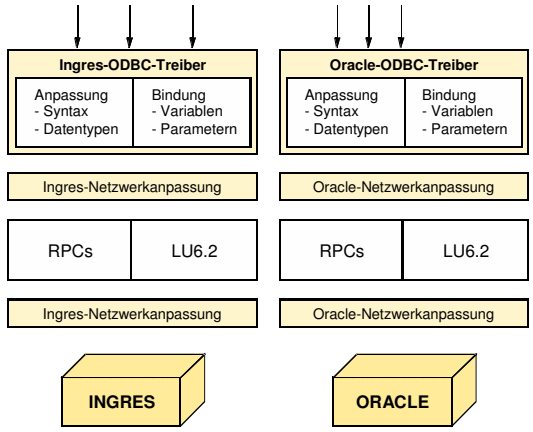
ODBC (1)

- **Architektur**



ODBC (2)

- **Architektur (Forts.)**



JDBC (1)

- JDBC: Java Database Connectivity
  - auf Grundlage von ODBC von SUN spezifiziert (http://java.sun.com/products/jdbc/overview.html)
  - Abstimmung auf Java, Vorteile von Java auch für API
  - Abstraktionsschicht zwischen Java-Programmen und SQL

Java-Anwendung
JDBC 3.0
SQL-92 / SQL:1999
(Objekt-)Relationales DBS

JDBC (2)

- **JDBC-Beispiel (Forts.)**

```
import java.sql.*;
import java.io.*;
public class Beispiel {
public static void main (String args[]) {
try {
  // Schritt 1: Aufbau einer Datenbankverbindung
  try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  }
  catch (ClassNotFoundException cex) {
    System.err.println(cex.getMessage());
  }
  Connection conn = DriverManager.getConnection(
    "jdbc:odbc:pizzaservice","","");
  // Schritt 2: Erzeugen einer Tabelle
  Statement stmt = conn.createStatement();
  stmt.executeUpdate("CREATE TABLE PizzaTabelle (" +
    "id INTEGER, " +
    "name CHAR(2), " +
    "preis FLOAT)" );
```


JDBC (3)

- **JDBC-Beispiel (Forts.)**

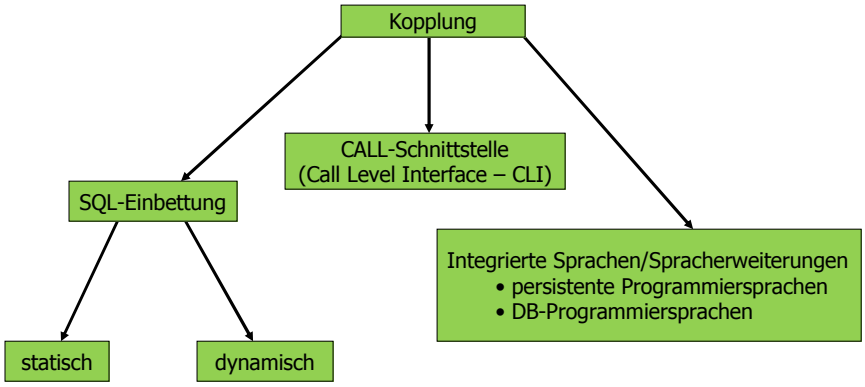
```
// Schritt 3: Füllen der Tabelle
stmt.executeUpdate("INSERT INTO PizzaTabelle " +
  "(id, name, preis) VALUES(12, 'Margherita', 7.20)");
...
// Schritt 4: Absetzen einer Anfrage
ResultSet rs = stmt.executeQuery("SELECT * FROM " +
  "PizzaTabelle WHERE preis = 7.20");
// Schritt 5: Ausgabe des Ergebnisses
while(rs.next()) {
  int id = rs.getInt("id");
  String name = rs.getString("name");
  float preis = rs.getFloat("preis");
  System.out.println("Treffer: " + name + " , " + preis);
}
// Schritt 6: Beenden der Verbindung
rs.close();
stmt.close();
conn.close();
}
catch(SQLException ex) {
System.err.println(ex.getMessage());
} } }
```

SQLJ

- **(Statisches) SQL eingebettet in Java**
  - kombiniert Vorteile von eingebettetem SQL mit Anwendungsportabilität ("binary portability")
  - nutzt JDBC als "Infrastruktur", kann mit JDBC-Aufrufen in der gleichen Anwendung kombiniert werden
- **Vorteile von SQLJ gegenüber JDBC (Entwicklung)**
  - bessere sprachliche Einbettung
  - vereinfachte Programmierung, kürzere Programme
  - Übersetzung und Optimierung von SQL-Anweisungen sowie Zugriffskontrolle zum Kompilierungszeitpunkt möglich

Kopplung

- **Übersicht**



Zusammenfassung APIs (1)

- **Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen**
  - Anpassung von mengenorientierter Bereitstellung und satzweiser Verarbeitung von DBS-Ergebnissen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
  - Erweiterungen: Scroll-Cursor, Sichtbarkeit von Änderungen
- **Statisches (eingebettetes) SQL**
  - hohe Effizienz, gesamte Typprüfung und Konvertierung erfolgen durch Precompiler
  - relativ einfache Programmierung
  - Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen
  - es können zur Laufzeit nicht verschiedene Datenbanken dynamisch angesprochen werden

Zusammenfassung APIs (2)

- **Dynamisches SQL**
  - Festlegung/Übergabe von SQL-Anweisungen zur Laufzeit
  - hohe Flexibilität, schwierige(re) Programmierung
- **PSM**
  - zielt ab auf Leistungsverbesserung, vor allem in Client/Server-Umgebung
  - erhöht die Verarbeitungsmächtigkeit des DBS

Zusammenfassung APIs (3)

- **CLI**
  - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
  - Keine Vorübersetzung oder Vorbereitung
    - Anwendungen brauchen nicht im Source-Code bereitgestellt werden
    - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools
- **JDBC**
  - bietet Schnittstelle für den Zugriff auf (objekt-) relationale DBS aus Java-Anwendungen
  - vermeidet einige syntaktischen Mängel (Lesbarkeit, Fehleranfälligkeit) von CLI
- **SQLJ**
  - eingebettete Sprache für „statische" Java-Programme
  - zielt auf verbesserte Laufzeiteffizienz im Vergleich zu JDBC ab, Syntax- und Semantikprüfung zur Übersetzungszeit
  - größere Unabhängigkeit von verschiedenen SQL-Dialekten