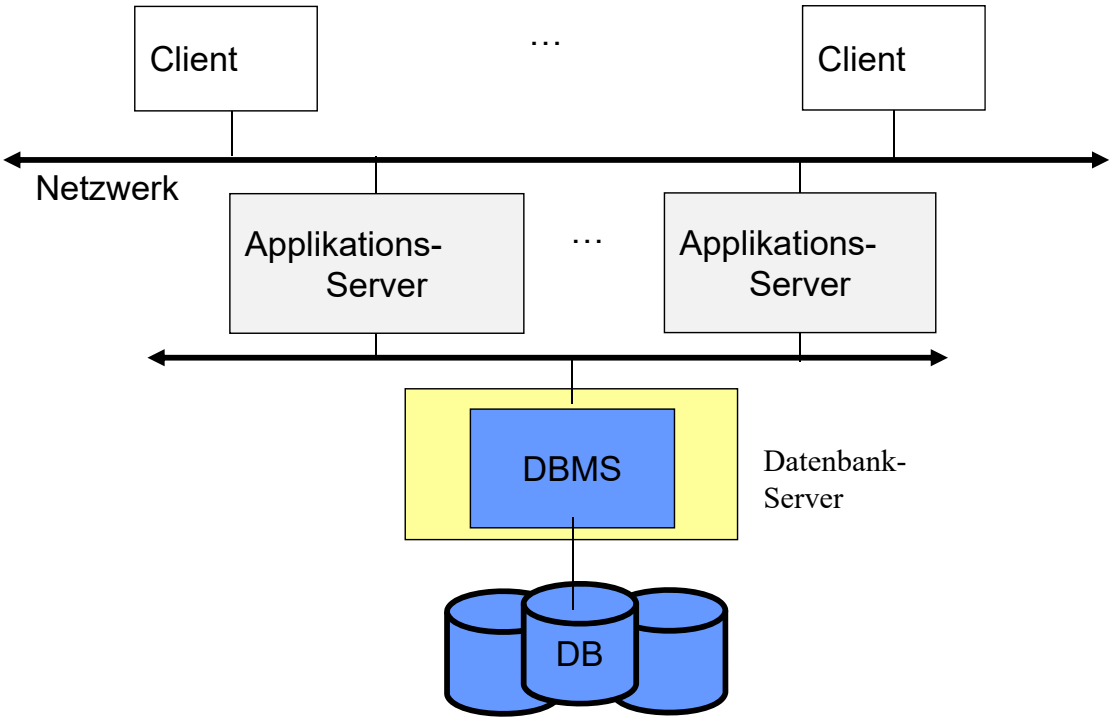
1. DB-Anwendungsprogrammierung (Teil 1)

- Einleitung
- Eingebettetes SQL
  - statisches SQL / Cursor-Konzept
  - dynamisches SQL
- Gespeicherte Prozeduren (Stored Procedures)
  - prozedurale Spracherweiterungen von SQL (SQL PSM)
  - SQL-Routinen

Kap. 2: DB-Anwendungsprogrammierung Teil 2

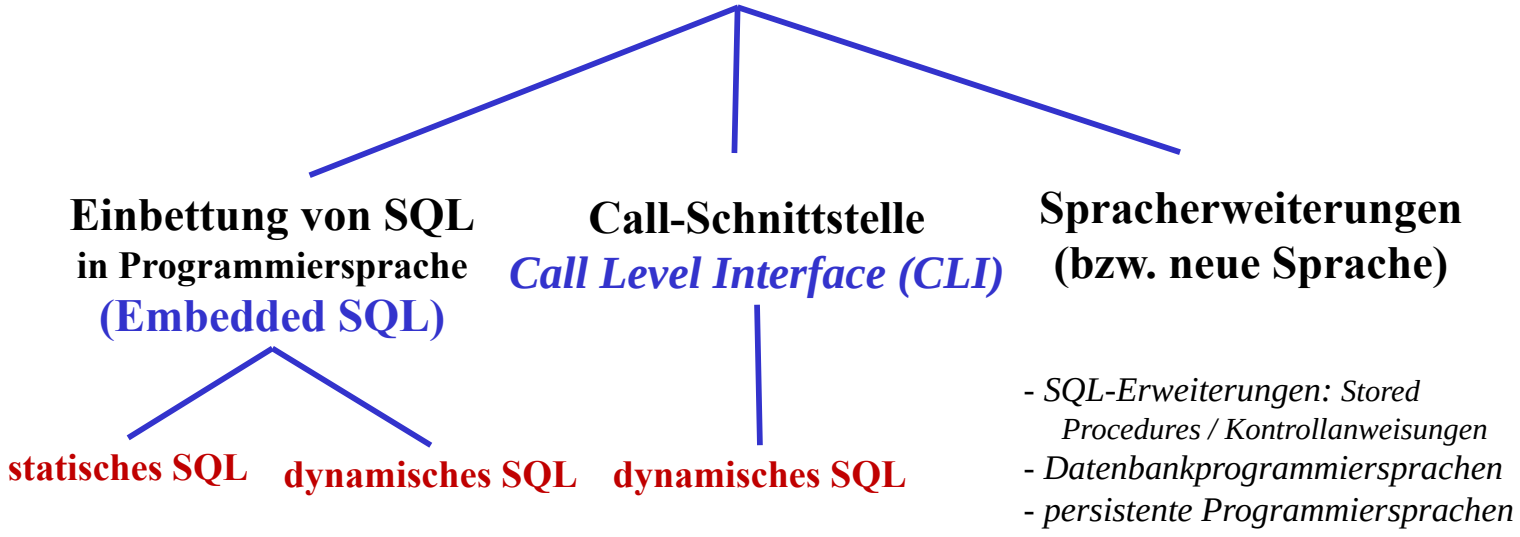
- JDBC
- SQLJ
- DB-Zugriff mit Skriptsprachen (PHP)
- SQL Injection Problem

Typische Datenbankumgebung



Kopplung Programmiersprache – SQL

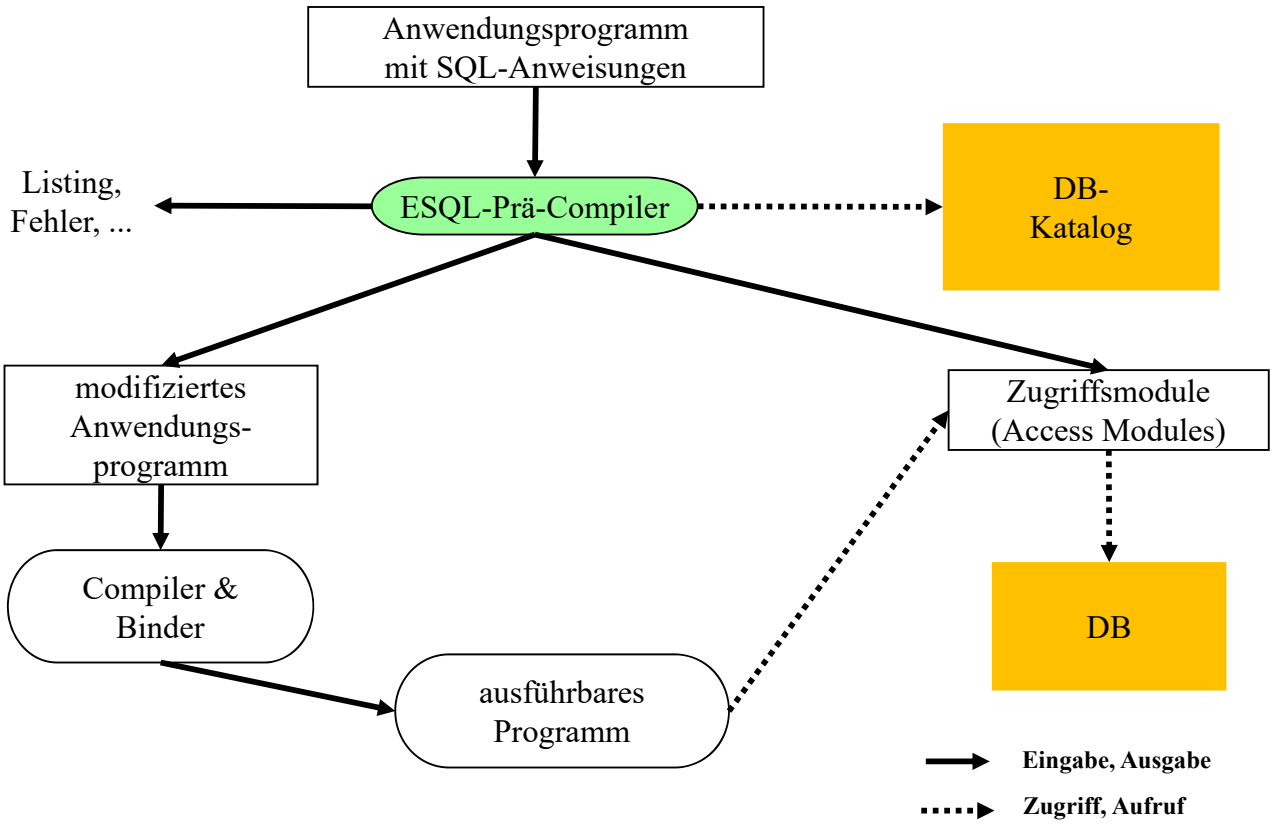
- Anwendungen und DBS basieren auf unterschiedlichen Sprachen: allgem. Programmiersprache (C, Java, …) vs. SQL
  - SQL-Befehle können eingebettet oder über Bibliothek aufgerufen werden (Call-Schnittstelle)



Kopplung Programmiersprache –SQL (2)

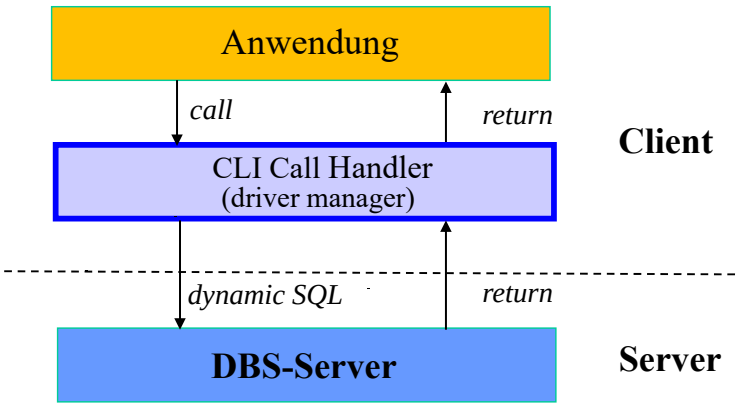
- Einbettung von SQL (Embedded SQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - Vorübersetzer (Prä-Compiler) wandelt DB-Aufrufe in Prozeduraufrufe um
- Call-Schnittstelle (CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
  - weniger komfortable Programmierung als mit Embedded SQL
- Statisches SQL: Anweisungen müssen zur Übersetzungszeit feststehen
  - Optimierung zur Übersetzungszeit ermöglicht hohe Effizienz (Performance)
- Dynamisches SQL: Konstruktion von SQL-Anweisungen zur Laufzeit
  - hohe Flexibilität

Verarbeitung von ESQL-Programmen



Call-Level-Interface

- SQL-Befehle werden über Aufrufe von Prozeduren/Funktionen einer standardisierten Bibliothek (API) ausgeführt
- Beispiele: ODBC, JDBC
- Hauptvorteil: keine Präkompilierung von Anwendungen
  - SQL-Anwendungen brauchen nicht im Source-Code bereitgestellt zu werden
  - wichtig zur Realisierung von kommerzieller Anwendungs-Software bzw. Tools



- Unterschiede in der SQL-Programmierung zu eingebettetem SQL
  - CLI impliziert i.a. dynamisches SQL (Optimierung zur Laufzeit)
  - komplexere Programmierung
  - explizite Anweisungen zur Datenabbildung zwischen DBS und Programmvariablen

Statisches SQL: Beispiel für C

```
exec sql include sqlca; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
   char      X[8];
   dec(15,2) GSum;
exec sql end declare section;
exec sql connect to dbname;
exec sql insert into PERS(PNR,PNAME,GEHALT) values (4711,'Ernie', 32000);
exec sql insert into PERS(PNR,PNAME,GEHALT) values (4712,'Bert',  38000);
printf("ANR ? "); scanf(" %s", X);
exec sql select sum (GEHALT) into :GSum from PERS where ANR = :X;
printf("Gehaltssumme: %d\n", GSum)
exec sql commit work;
exec sql disconnect;
}
```

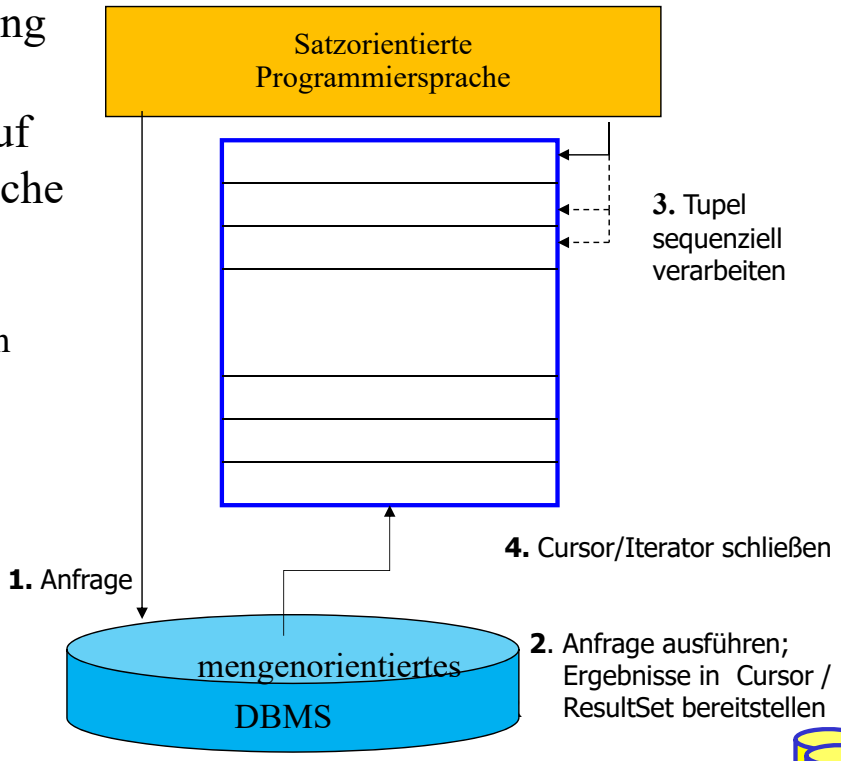
- Anmerkungen
  - eingebettete SQL-Anweisungen werden durch "EXEC SQL" eingeleitet und spezielles Symbol (hier ";") beendet, um Compiler Unterscheidung von anderen Anweisungen zu ermöglichen
  - Verwendung von AP-Variablen in SQL-Anweisungen verlangt Deklaration innerhalb eines "declare section"-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
  - Werteabbildung mit Typanpassung durch INTO-Klausel bei SELECT
  - Kommunikationsbereich SQLCA (Rückgabe von Statusanzeigern u. ä.)

Transaktionsbeispiel: Debit/Credit

```
void main ( ) {
        exec sql        begin declare section
                         int a /*accountid*/, dec(15,2) b /*balance*/, amount;
        exec sql        end declare section;
         /* read user input */
         scanf („%d %d“, &a, &amount);
         /* read account balance */
        exec sql        select balance into :b from account
                        where account_id = :a;
       /* add amount (positive for debit, negative for credit) */
          b = b + amount;
       /* write account balance back into database */
        exec sql        update account
                        set balance = :b where account_id = :a;
        exec sql        commit work;
       }
```

Mengenorientierte Anfragen

- Queries mit nur einem Ergebnissatz
  - einfache Übernahme der Ergebnisse in Programmvariable (SELECT attr INTO :var)
- Kernproblem bei SQL-Einbettung in Programmiersprachen: Abbildung von Tupelmengen auf Variablen der Programmiersprache
  - satzweise Bereitstellung und Abarbeitung von DBS-Ergebnismengen erforderlich
  - Nutzung von Cursor/Iteratoren bzw. Result-Sets (CLI)



Cursor-Konzept in Embedded SQL

- Cursor ist ein Iterator, der einer Anfrage (Ergebnisrelation) zugeordnet wird und mit dessen Hilfe die Sätze der Ergebnismenge einzeln (one tuple at a time) im Programm bereitgestellt werden
  - kein Cursor erforderlich für Select-Anweisungen, die nur einen Ergebnissatz liefern (SELECT INTO)
- 4 Operationen auf einen Cursor C1
  - DECLARE C1 CURSOR FOR table-exp (Cursor-Spezifikation)
  - OPEN C1 (Query-Ausführung)
  - FETCH C1 INTO VAR1, VAR2, . . ., VARn (Abruf eines Ergebnissatzes)
  - CLOSE C1
- Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung
  - Übergabe der Werte eines Tupels mit Hilfe der INTO-Klausel bei FETCH
    => INTO target-commalist (Variablenliste d. Anwendungsprogramms)
  - Anpassung der Datentypen (Konversion)

Cursor-Konzept (2)

- Beispielprogramm in C (vereinfacht)

```
   ...
      exec sql begin declare section;
           char  X[50];
           char  Y[8];
           double G;
      exec sql end declare section;
      exec sql declare C1 cursor  for
              select NAME, GEHALT from PERS where ANR = :Y;
      printf("ANR ? "); scanf(" %s", Y);
      exec  sql  open  C1;
      while  (sqlcode == ok) {
              exec  sql  fetch  C1  into  :X, :G;
              printf("%s\n", X)}
      exec  sql  close  C1;
...
```

- DECLARE C1 . . . ordnet der Anfrage einen Cursor C1 zu
- OPEN C1 bindet Werte der Eingabevariablen für Anfrageausführung
- Systemvariable SQLCODE zur Übergabe von Fehlermeldungen (Teil von SQLCA)

Dynamisches SQL

- dynamisches SQL: Festlegung von SQL-Anweisungen zur Laufzeit
  -> Query-Optimierung i.a. erst zur Laufzeit möglich
- SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt
  - Deklaration DECLARE STATEMENT
  - Anweisungen enthalten **SQL-Parameter (?)** statt Programmvariablen
- 2 Varianten: Prepare-and-Execute bzw. Execute Immediate

```
   exec sql begin declare section;
      char  Anweisung[256], X[6];
   exec sql end declare section;
   exec sql declare SQLanw statement;

   Anweisung = "DELETE FROM PERS WHERE  ANR = ? AND ORT = ?"; /*bzw. Einlesen
   exec sql prepare SQLanw from :Anweisung;
   exec sql execute SQLanw using
   scanf(" %s", X);
   exec sql execute SQLanw using
```

Dynamisches SQL (2)

- Variante ohne Vorbereitung: EXECUTE IMMEDIATE
- Beispiel

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

- maximale Flexibilität, jedoch potenziell geringe Performance
  - kann für einmalige Query-Ausführung ausreichen

Beispiel (ChatGPT)

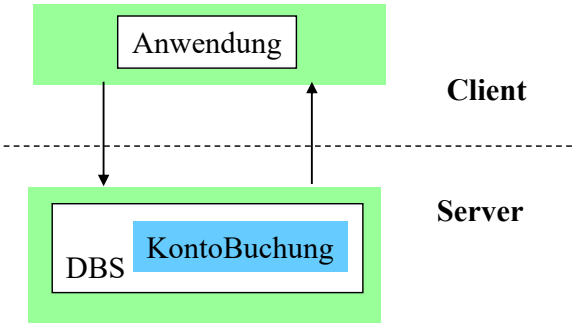
```c
#include <stdio.h>
#include <stdlib.h>
#include <sqlca.h>

EXEC SQL BEGIN DECLARE SECTION;
int product_id;
char query[100];
struct Product {
    int id;
    char name[50];
    float price;
} product;
EXEC SQL END DECLARE SECTION;

int main() {
    /* Verbindung zur Datenbank herstellen */
    EXEC SQL CONNECT TO database_name;

    /* Benutzereingabe für Produkt-ID */
    printf("Enter product ID: ");
    scanf("%d", &product_id);
```

```c
/* Dynamische SQL-Anweisung erstellen und ausführen */
sprintf(query, "SELECT ID, Name, Price FROM Products WHERE ID = %d", product_id);
EXEC SQL EXECUTE IMMEDIATE :query INTO :product.id, :product.name, :product.price

/* Verbindung zur Datenbank trennen */
EXEC SQL DISCONNECT database_name;

    /* Ergebnis ausgeben */
    printf("Product ID: %d, Name: %s, Price: $%.2f\n",
                        product.id, product.name, product.price);
    return 0;
}
```


- Einleitung
- Eingebettetes SQL
  - statisches SQL / Cursor-Konzept
  - dynamisches SQL
- Gespeicherte Prozeduren (Stored Procedures)
  - prozedurale Spracherweiterungen von SQL (SQL PSM)
  - SQL-Routinen

Gespeicherte Prozeduren (Stored Procedures)

- Prozeduren werden durch DBS gespeichert und verwaltet
  - benutzerdefinierte Prozeduren oder Systemprozeduren
  - Programmierung der Prozeduren in SQL oder allgemeiner Programmiersprache



- Vorteile:
  - als gemeinsamer Code für verschiedene Anwendungsprogramme wiederverwendbar
  - Anzahl der Zugriffe von Anwendungen auf DB wird reduziert
  - höherer Grad der Isolation der Anwendung von DB (Datenunabhängigkeit)
- Nachteile
  - mehr Arbeit für DB-Server
  - erhöhtes Sicherheitsrisiko für DBS

Persistente SQL-Module (PSM)

- SQL-Prozeduren erfordern Spracherweiterungen im SQL-Standard
  - u.a. allgemeine Kontrollanweisungen IF, WHILE, etc.
  - herstellerspezifische Festlegungen bereits seit 1987 (Sybase Transact-SQL)
  - PSM: seit SQL:1999 standardisiert

- Routinen: Prozeduren und Funktionen
  - Routinen sind Schema-Objekte (wie Tabellen, Views etc.)
  - geschrieben in SQL (SQL-Routine) oder in externer Programmiersprache (C, Java, ...) -> 2 Sprachen / Typsysteme

- zusätzliche DDL-Anweisungen
  - CREATE PROCEDURE
  - DROP PROCEDURE
  - CREATE FUNCTION
  - DROP FUNCTION

PSM: SQL-Routinen

- **SQL-Routinen**: in SQL geschriebene Prozeduren/Funktionen
  - Deklarationen lokaler Variablen etc. innerhalb der Routinen
  - Nutzung zusätzlicher Kontrollanweisungen: Zuweisung, IF, FOR etc.
  - Exception Handling (SIGNAL, RESIGNAL)
  - integrierte Programmierumgebung
  - keine Typkonversionen
  - Prozedur- und Funktionsaufrufe können rekursiv sein

- Beispiel

```
CREATE PROCEDURE KontoBuchung
    (IN konto INTEGER, IN betrag DECIMAL (15,2));
BEGIN      …
           UPDATE account
           SET balance = balance + betrag
           WHERE account_# = konto;
           …
END;
```

- Prozeduren werden über CALL-Anweisung aufgerufen:

```
exec sql CALL KontoBuchung  (:account_#, :amount);
```


PSM: SQL-Routinen (2)

- Beispiel einer SQL-Funktion:

```
CREATE FUNCTION vermoegen (kunr INTEGER) RETURNS DECIMAL (15,2);

 BEGIN     DECLARE vm DECIMAL(15,2);

           SELECT sum (balance) INTO vm
           FROM   account
           WHERE account_owner = kunr;

           RETURN vm;
 END;
```

- Aufruf persistenter Funktionen (SQL und externe) in SQL-Anweisungen wie Built-in-Funktionen

```
SELECT *
FROM kunde
WHERE vermoegen (KNR) > 100000.00
```

Prozedurale Spracherweiterungen: Kontrollanweisungen

Compound Statement	BEGIN ... END;
SQL-Variablendeklaration	DECLARE var type;
Zuweisung	SET x = “abc”;
If-Anweisung	IF condition THEN ... ELSE ... :
Case-Anweisung	CASE expression WHEN x THEN ... WHEN ... :
While/Loop-Anweisung	WHILE i < 100 LOOP ... END LOOP;
For-Anweisung	FOR result AS ... DO ... END FOR;
Leave-Anweisung	LEAVE ...;
Prozeduraufruf	CALL procedure_x (1, 2, 3);
Return-Anweisung	RETURN x;
Signal/Resignal	SIGNAL division_by_zero;

PSM Beispiel

```
outer: BEGIN
  DECLARE account INTEGER DEFAULT 0;
  DECLARE balance DECIMAL (15,2);
  DECLARE no_money EXCEPTION FOR SQLSTATE VALUE 'xxxxx';

  SELECT account_#, balance INTO account, balance FROM accounts WHERE ...;

  IF balance < 10.0 THEN SIGNAL no_money;


  BEGIN ATOMIC
    DECLARE cursor1 CURSOR ...;
    DECLARE balance DECIMAL (15,2);
    SET balance = outer.balance – 10.0;
    UPDATE accounts SET balance = balance WHERE account_# = account;
    INSERT INTO account_history VALUES (account, CURRENT_DATE, 'W', balance); .....
  END;


 EXCEPTION
    WHEN no_money THEN
    CASE (SELECT account_type FROM accounts WHERE account_# = account)
          WHEN 'VIP' THEN INSERT INTO send_letter ....
          WHEN 'NON-VIP' THEN INSERT INTO blocked_accounts ...)

                …
 END;
```


Zusammenfassung

- statisches (eingebettetes) SQL
  - hohe Effizienz
  - relativ einfache Programmierung
  - begrenzte Flexibilität (fester Aufbau aller SQL-Befehle)
- Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
- Call-Level-Interface (z.B. JDBC)
  - erfordert keinen Präcompiler
  - Verwendung von dynamischem SQL
- Stored Procedures
  - Performance-Gewinn durch reduzierte Häufigkeit von DBS-Aufrufen
  - SQL-Standardisierung: Persistent Storage Modules (PSM)
  - umfassende prozedurale Spracherweiterungen von SQL