1. DB-Anwendungsprogrammierung (Teil 1)

- Einleitung
- Eingebettetes SQL
  - statisches SQL / Cursor-Konzept
  - dynamisches SQL
- Unterstützung von Transaktionen / Isolation Level
- Gespeicherte Prozeduren (Stored Procedures)
  - prozedurale Spracherweiterungen von SQL (SQL PSM)
  - gespeicherte Prozeduren mit Java

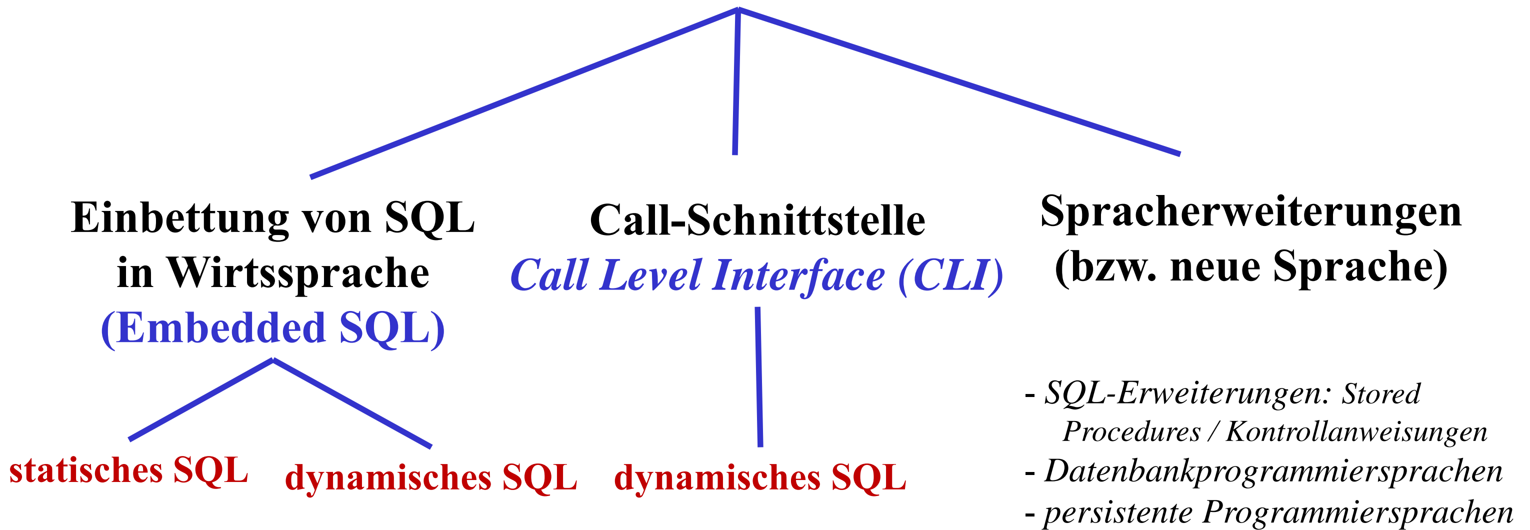
Kap. 2: DB-Anwendungsprogrammierung Teil 2

- JDBC
- SQLJ
- Web-Anbindung von Datenbanken

Kopplung Programmiersprache – DBS/SQL

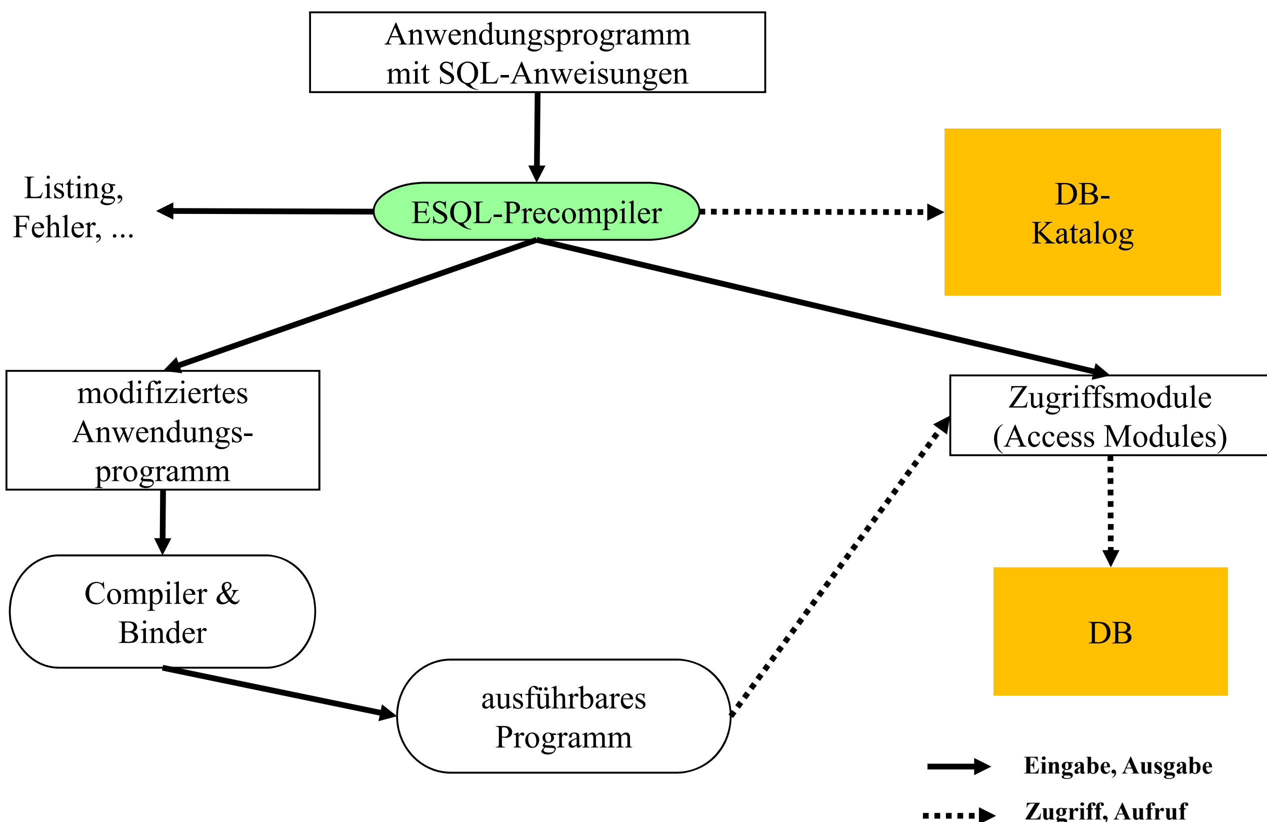
- Zwei wesentliche Kriterien
  - Embedded SQL vs. Call-Level-Interface (CLI)
  - Statisches vs. Dynamisches SQL
- Einbettung von SQL (Embedded SQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - Vorübersetzer (Prä-Compiler) wandelt DB-Aufrufe in Prozeduraufrufe um
- Call-Schnittstelle (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

Kopplung Programmiersprache – DBS



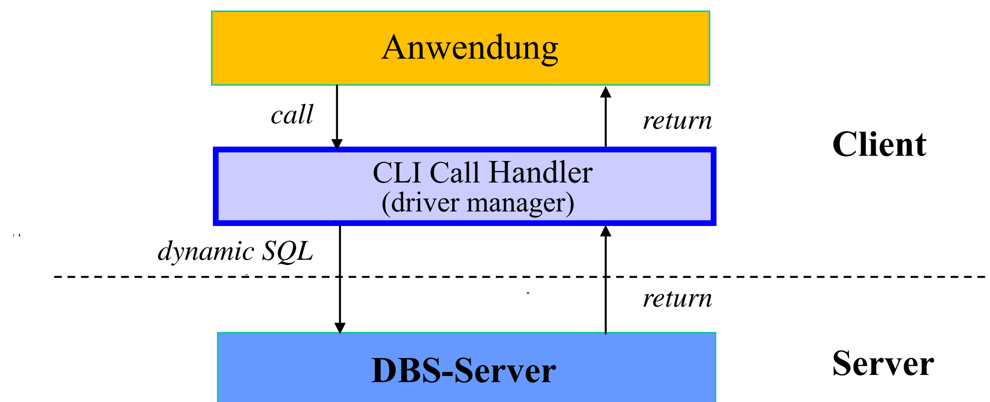
- SQL-Erweiterungen: Stored Procedures / Kontrollanweisungen
- Datenbankprogrammiersprachen
- persistente Programmiersprachen

Verarbeitung von ESQL-Programmen



Call-Level-Interface

- alternative Möglichkeit zum Aufruf von SQL-Befehlen innerhalb von Anwendungsprogrammen: direkte Aufrufe von Prozeduren/Funktionen einer standardisierten Bibliothek (API)
- Hauptvorteil: keine Präkompilierung von Anwendungen
  - Anwendungen mit SQL-Aufrufen brauchen nicht im Source-Code bereitgestellt zu werden
  - wichtig zur Realisierung von kommerzieller Anwendungs-Software bzw. Tools
- Einsatz v. a. in Client/Server-Umgebungen



Call-Level-Interface (2)

- 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
- SQL-Standardisierung des CLI stark an ODBC angelehnt
- CLI für Java-Anwendungen: JDBC

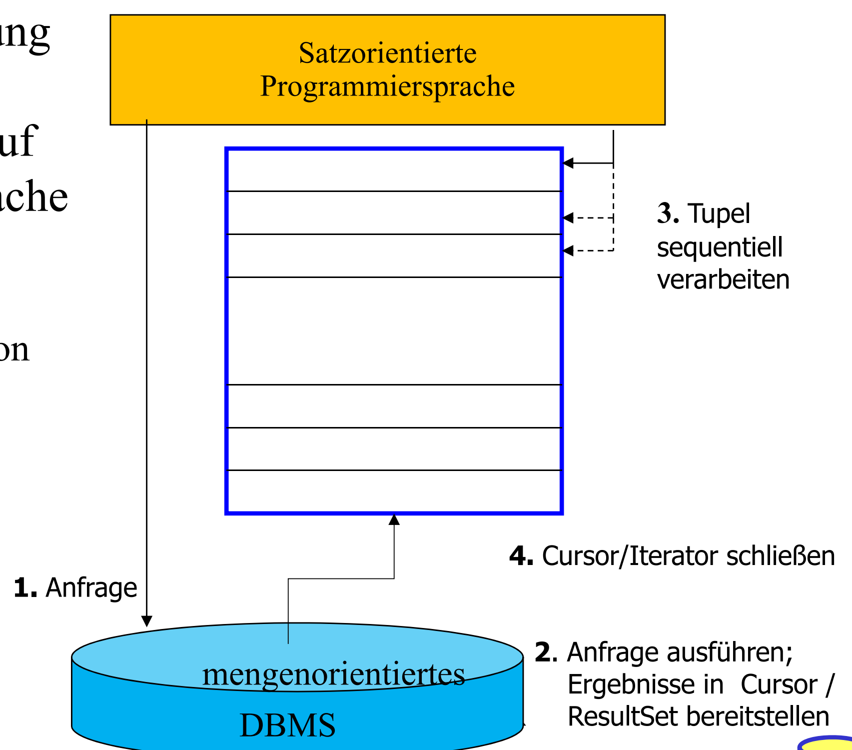
Statisches SQL: Beispiel für C

```
exec sql include sqlca; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
   char  X[8];
   int   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.)

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
  - Nutzung von Cursor/Iteratoren bzw. Result-Sets zur satzweisen Bereitstellung und Abarbeitung von DBS-Ergebnismengen



Cursor-Konzept in Embedded SQL

- Cursor ist ein Iterator, der einer Anfrage (Relation) zugeordnet wird und mit dessen Hilfe die Tupeln des Ergebnismenge einzeln (one tuple at a time) im Programm bereitgestellt werden
  - Trennung von Query-Spezifikation (Cursor-Deklaration) und Bereitstellung/Verarbeitung von Tupeln im Query-Ergebnis
- Operationen auf einen Cursor C1
  - DECLARE C1 CURSOR FOR table-exp
  - OPEN C1
  - FETCH C1 INTO VAR1, VAR2, . . ., VARn
  - 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. Wirtsprogramms)
  - Anpassung der Datentypen (Konversion)
- kein Cursor erforderlich für Select-Anweisungen, die nur einen Ergebnissatz liefern (SELECT INTO)

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 die Werte der Eingabevariablen
- Systemvariable SQLCODE zur Übergabe von Fehlermeldungen (Teil von SQLCA)

Verwaltung von Verbindungen

- Zugriff auf DB erfordert i.a. zunächst, eine Verbindung herzustellen, v.a. in Client/Server-Umgebungen
  - Aufbau der Verbindung mit CONNECT, Abbau mit DISCONNECT
  - jeder Verbindung ist eine Session zugeordnet
  - Anwendung kann Verbindungen (Sessions) zu mehreren Datenbanken offenhalten
  - Umschalten der "aktiven" Verbindung durch SET CONNECTION

```
CONNECT TO   target [AS connect-name] [USER user-name]
```

```
SET  CONNECTION    { connect-name | DEFAULT }
```

```
DISCONNECT    { CURRENT | connect-name | ALL }
```

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
  - Dekaration 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 potentiell geringe Performance
  - kann für einmalige Query-Ausführung ausreichen

Transaktionskonzept (ACID)

Transaktion: Folge von DB-Operationen (DML-Befehlen), für die DBS die ACID-Eigenschaften gewährleistet

- **A**tomicity: die Änderungen einer Transaktion werden vollständig oder gar nicht in die Datenbank eingebracht (Alles oder Nichts')
- **C**onsistency: eine erfolgreiche Transaktion erhält die DB-Konsistenz (Menge der definierten Integritätsbedingungen)
- **I**solation: DB-Zugriffe gleichzeitig ausgeführter Transaktionen werden synchronisiert (logischer Einbenutzerbetrieb)
- **D**urability: Änderungen erfolgreich beendeter Transaktionen sind persistent gegenüber Fehlern wie Systemabstürzen

- Programmierschnittstelle für Transaktionen
  - begin of transaction (BOT): implizit in SQL
  - commit transaction („commit work“ in SQL)
  - rollback transaction („rollback work“ in SQL)

Transaktionsbeispiel: Debit/Credit

```
void main ( ) {
        exec sql        BEGIN DECLARE SECTION
                         int b /*balance*/, a /*accountid*/, 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;
      }
```

Synchronisationsprobleme

- DBS müssen Mehrbenutzerbetrieb unterstützen
- ohne Synchronisation kommt es zu so genannten Mehrbenutzer-Anomalien
  - verloren gegangene Änderungen (lost updates)
  - Zugriffe auf nicht freigegebene Änderungen (dirty read, read uncommitted)
  - inkonsistente Analyse (non-repeatable read)
  - Phantom-Probleme
- Anomalien sind nur durch Änderungen verursacht

Beispiel paralleler Ausführung (Synchronisationsproblem)

P1	Werte (Variablen, DB)	P2
	/* b1=0, a.Balance=100, b2=0 */	
select Balance Into :b1 from Account where Account_ID = :a		
		select Balance Into :b2 from Account where Account_ID = :a
	/* b1=100, a.Balance=100, b2=100 */	
b1 = b1-50		
	/* b1=50, a.Balance=100, b2=100 */	
		b2 = b2 +100
	/* b1=50, a.Balance=100, b2=200 */	
update Account set Balance = :b1 where Account_ID = :a		
	/* b1=50, a.Balance=50, b2=200 */	
		update Account set Balance = :b2 where Account_ID = :a
	/* b1=50, a.Balance=200, b2=200 */	

Non-repeatable Read

P1	Werte (Variable, DB)	P2
	/* b1=0, b2=0, a.Balance=100 */	
select Balance Into :b1 from Account where Account_ID = :a		
	/* b1=100, a.Balance=100 */	
		update Account set Balance = Balance + 100 where Account_ID = :a
		commit work;
	/* a.Balance=200 */	
select Balance Into :b2 from Account where Account_ID = :a		
	/* b2=200 */	

IF b1<> b2 THEN
<Fehlerbehandlung>

Synchronisation

- Synchronisation automatisch durch DBS, z.B. durch
  - Setzen von (Lese/Schreib-) Sperren vor Datenzugriff
  - Freigabe der Sperren am Transaktionsende
- idealerweise werden alle Anomalien beseitigt (logischer Einbenutzerbetrieb)
  - Synchronisation gewährleistet *„Serialisierbarkeit“*
  - gleichzeitige Ausführung von Transaktionen hat den gleichen Effekt auf die DB wie eine serielle Ausführung dieser Transaktionen
- Inkaufnahme einiger Anomalien verbessert jedoch i.a. Leistungsfähigkeit
  - weniger Sperren, Blockierungen (Warteverzögerungen auf frei werdende Sperren) und Deadlocks

SQL-Isolationsstufen

- 4 Konsistenzebenen (Isolation Level) zur Synchronisation von Transaktionen
  - Default ist Serialisierbarkeit (serializable)
  - Lost-Update muß generell vermieden werden
  - READ UNCOMMITTED für Änderungstransaktionen unzulässig

Isolation Level	Anomalie: Dirty Read	Anomalie: Non-Repeatable Read	Anomalie: Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- SQL-Anweisung zum Setzen der Konsistenzebene (Embedded SQL)

```
SET TRANSACTION [READ WRITE | READ ONLY] ISOLATION LEVEL <level>
```

Beispiel: exec sql SET TRANSACTON Read Only ISOLATION LEVEL Read Committed

Recovery-Unterstützung

- automatische Behandlung aller erwarteten Fehler durch das DBS
- Transaktionsparadigma verlangt:
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen
- Voraussetzung: Sammeln redundanter Informationen während Normalbetrieb (Logging)

Fehlerarten

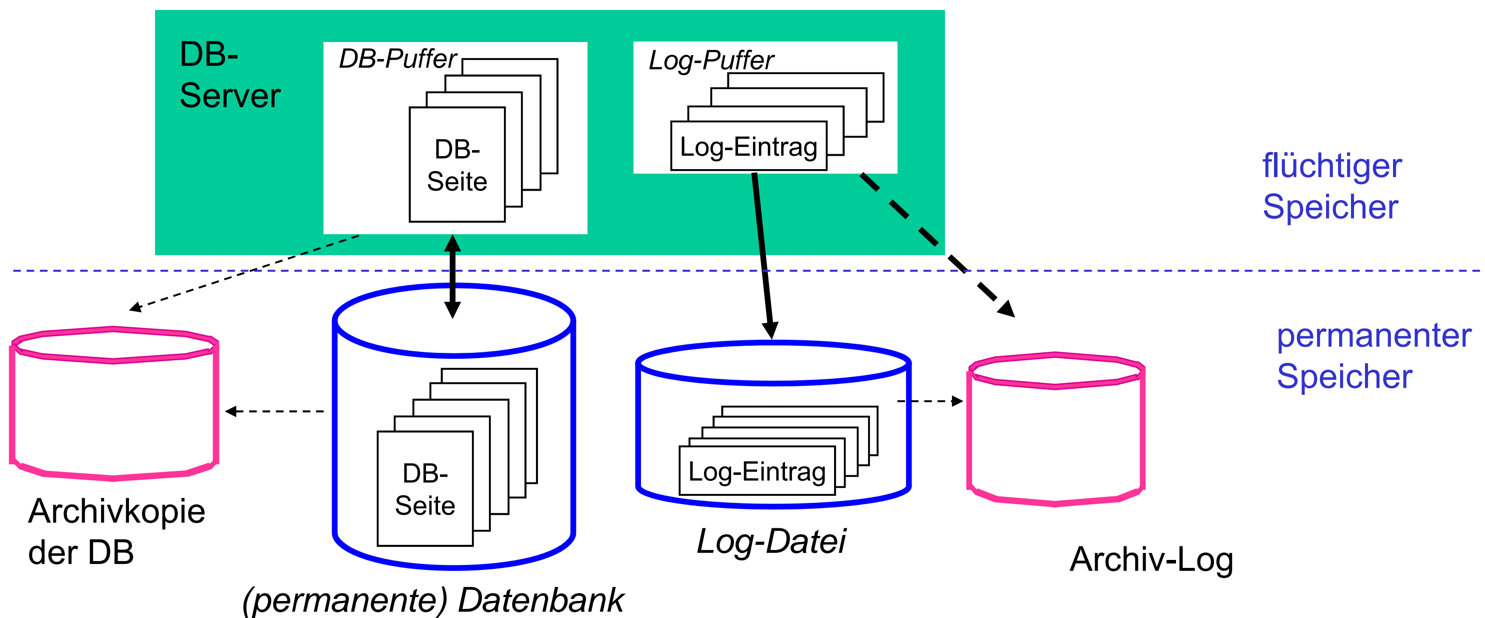
- *Transaktionsfehler*: vollständiges Zurücksetzen auf Transaktionsbeginn (Undo)
- *Systemfehler* (Rechnerausfall, DBS-Absturz)
  - REDO für erfolgreiche Transaktionen (Wiederholung von Änderungen, die aufgrund des Systemfehlers nicht in der Datenbank sind)
  - UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen derer Änderungen aus der Datenbank)
- *Gerätefehler* (Plattenausfall):
  - vollständiges Wiederholen (REDO) aller Änderungen auf einer Archivkopie
  - oder: Spiegelplatten bzw. RAID-Disk-Arrays
- *Katastrophen* (Komplettausfall Rechenzentrum, etc.)
  - Verteilte Datensicherung auf geographisch separierten Systemen

Atomaritätsproblem: Beispiel

- Unterbrechung während einer Überweisung

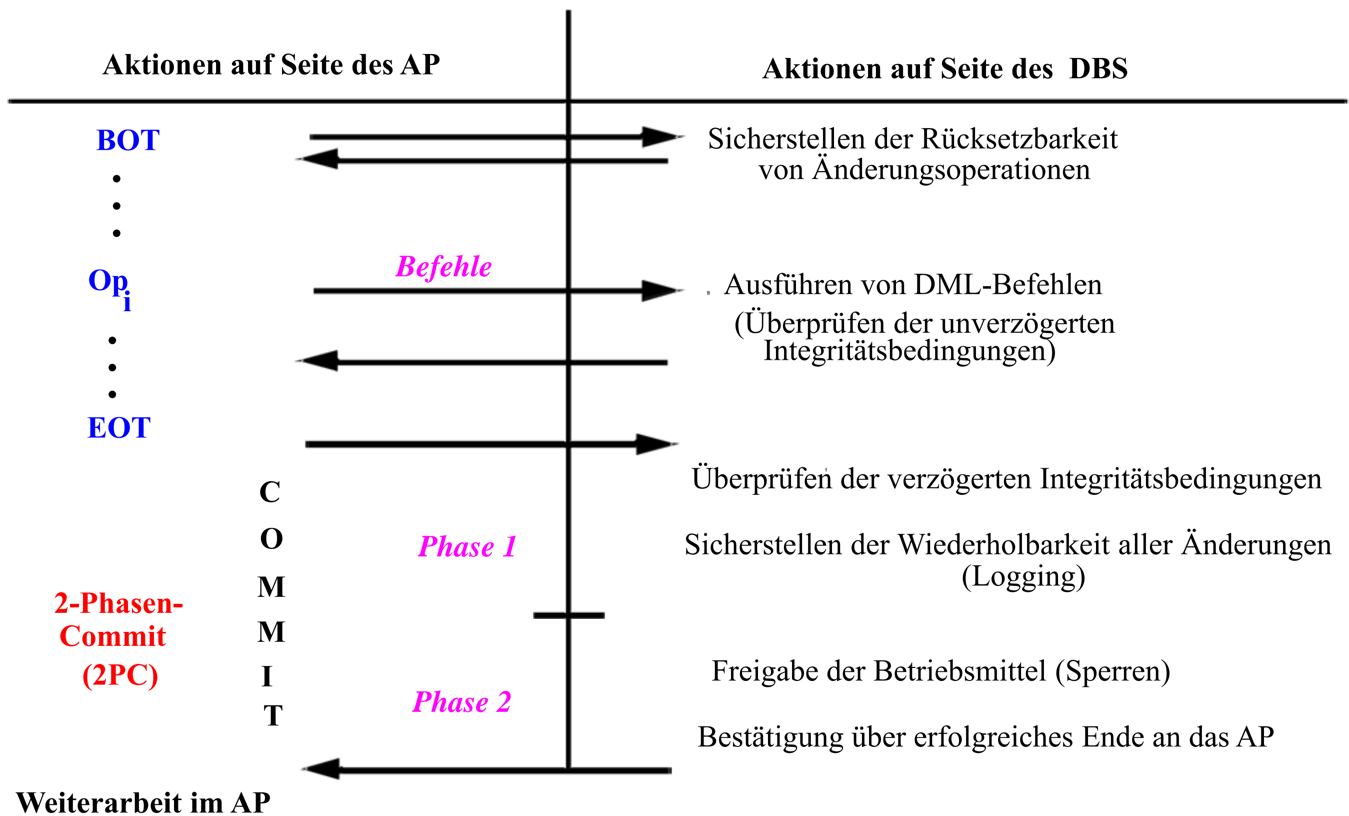
```
void main ( ) {
  /* read user input */
  scanf („%d %d %d“, &sourceid, &targetid, &amount);

  /* subtract amount from source account */
    exec sql update Account
            set Balance = Balance - :amount where Account_Id = :sourceid;

  /* add amount to target account */
    exec sql update Account
            set Balance = Balance + :amount where Account_Id = :targetid;
    exec sql commit work; }
```

Systemkomponenten zur Recovery



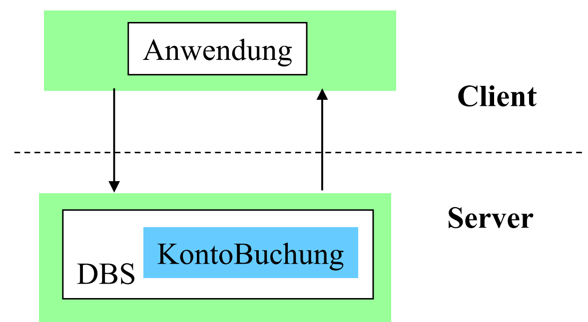
- Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)
  – Ausschreiben spätestens am Transaktionsende ("Commit")
- Log-Datei zur Behandlung von Transaktions- und Systemfehler
- Behandlung von Gerätefehlern: Archivkopie + Archiv-Log => aktuelle DB

Die Transaktion als Schnittstelle zwischen Anwendungsprogramm und DBS



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 des Anwendungsprogramms auf die DB werden reduziert
  - Performance-Vorteile v.a. in Client-Server-Umgebungen
  - höherer Grad der Isolation der Anwendung von DB wird erreicht
- Nachteile ?

Persistente SQL-Module (PSM)

- SQL-Prozeduren erfordern Spracherweiterungen gegenüber SQL1992
  - u.a. allgemeine Kontrollanweisungen IF, WHILE, etc.
  - herstellerspezifische Festlegungen bereits seit 1987 (Sybase Transact-SQL)
  - PSM: seit SQL99 standardisiert
- Routinen: Prozeduren und Funktionen
  - Routinen sind Schema-Objekte (wie Tabellen etc.) und werden im Katalog aufgenommen (beim DBS/Server)
  - geschrieben in SQL (SQL routine) oder in externer Programmiersprache (C, Java, FORTRAN, ...) -> 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, Blockbildung, IF, LOOP, etc.
  - Exception Handling (SIGNAL, RESIGNAL)
  - integrierte Programmierumgebung
  - keine Typkonversionen
- Beispiel

```
CREATE PROCEDURE KontoBuchung
    (IN konto INTEGER, IN betrag DECIMAL (15,2));
          BEGIN DECLARE C1 CURSOR FOR ...;

                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 INTEGER;
           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
```

- Prozedur- und Funktionsaufrufe können rekursiv sein

Prozedurale Spracherweiterungen: Kontrollanweisungen

Compound Statement	BEGIN ... END;
SQL-Variablendeklaration	DECLARE var type;
If-Anweisung	IF condition THEN ... ELSE ... :
Case-Anweisung	CASE expression WHEN x THEN ... WHEN ... :
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);
Zuweisung	SET x = “abc”;
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';
  DECLARE DB_inconsistent EXCEPTION FOR SQLSTATE VALUE 'yyyyy';

  SELECT account_#, balance INTO account, balance FROM accounts ...;
  IF (balance - 10) < 0 THEN SIGNAL no_money;
  BEGIN ATOMIC
    DECLARE cursor1 SCROLL CURSOR ...;
    DECLARE balance DECIMAL (15,2);
    SET balance = outer.balance - 10;
    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 ...
   ELSE SIGNAL DB_inconsistent;

   WHEN DB_inconsistent THEN
        BEGIN .... END;

   END;
```

Zusammenfassung

- statisches (eingebettetes) SQL
  - hohe Effizienz, relativ einfache Programmierung
  - begrenzte Flexibilität (Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen)
- 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
- Transaktionskonzept
  - Kontrolle der Synchronisation durch Isolation Level
- Stored Procedures
  - Performance-Gewinn durch reduzierte Häufigkeit von DBS-Aufrufen
  - SQL-Standardisierung: Persistent Storage Modules (PSM)
  - umfassende prozedurale Spracherweiterungen von SQL