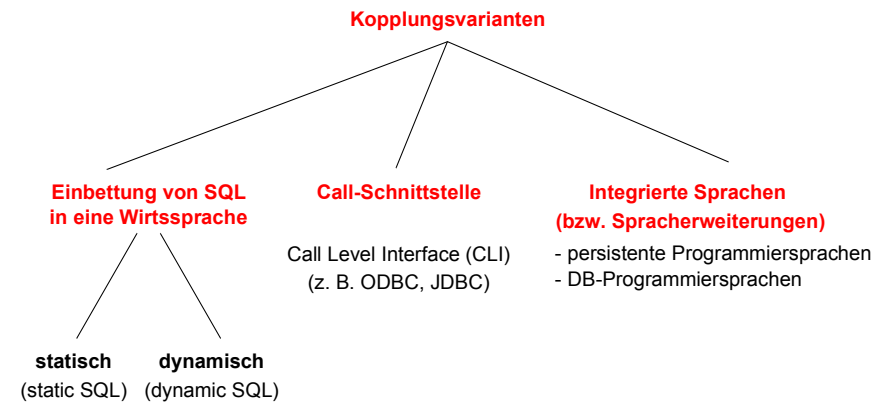


5. Anwendungsprogrammierschnittstellen

- **Kopplung mit einer Wirtssprache:**
 - Übersicht und Aufgaben
- **Eingebettetes statisches SQL**
 - Cursor-Konzept
 - SQL-Programmiermodell
 - Rekursion
 - Ausnahme- und Fehlerbehandlung
- **Aspekte der Anfrageauswertung**
 - Vorbereitung und Ausführung
- **Dynamisches SQL**
 - Eingebettetes dynamisches SQL
- **Call-Level-Interface**
 - Ansatz und Vorteile
 - DB-Zugriff via JDBC
- **SQL/PSM**

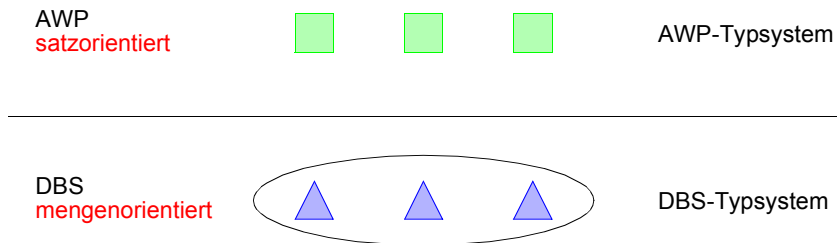
Kopplung mit einer Wirtssprache



- **Einbettung von SQL (Embedded SQL, ESQL)**
 - 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 PS-Übersetzer für umgebendes Programm
 - SQL-Anweisungen müssen zur Übersetzungszeit feststehen
 - im SQL-Standard unterstützte Sprachen: C, COBOL, FORTRAN, Ada, PL1, Pascal, MUMPS, Java, ...
- **dynamische Einbettung:**
 - Konstruktion von SQL-Anweisungen (als Zeichenkette) zur Laufzeit
- **Call-Schnittstelle (prozedurale bzw. objektorientierte Schnittstelle, CLI)**
 - DB-Funktionen werden durch Bibliothek von Prozeduren/Methoden realisiert
 - Anwendung enthält lediglich Prozedur-/Methodenaufrufe
 - Zeichenkette mit SQL-Anweisung als Parameter

Kopplung mit einer Wirtssprache (2)

- Integrationsansätze unterstützen typischerweise nur
 - ein Typsystem
 - Navigation (satz-/objektorientierter Zugriff)
 - ➔ Wünschenswert sind jedoch Mehrsprachenfähigkeit und deskriptive DB-Operationen (mengenorientierter Zugriff)
- Relationale AP-Schnittstellen (API) bieten diese Eigenschaften, erfordern jedoch Maßnahmen zur Überwindung der sog. Fehlanpassung (impedance mismatch)



- Kernprobleme der API bei konventionellen Programmiersprachen
 - Konversion und Übergabe von Werten
 - Übergabe aktueller Werte von Wirtssprachenvariablen (Parametrisierung von DB-Operationen)
 - DB-Operationen sind i. allg. mengenorientiert: Wie und in welcher Reihenfolge werden Zeilen/Sätze dem AP zur Verfügung gestellt?
 - ➔ Cursor-Konzept

Kopplung mit einer Wirtssprache (3)

- Embedded (static) SQL: Beispiel für C

```

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 ("Anr ? "); 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;
}
  
```

- 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

- **Cursor-Konzept zur satzweisen 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
 - Wie viele Cursor können im AWP sein?

- **Cursor-Deklaration**

```
DECLARE cursor CURSOR FOR table-exp
[ORDER BY order-item-commalist]
```

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

- **Operationen auf einen Cursor C1**

```
OPEN C1
FETCH C1 INTO Var1, Var2, . . . , Varn
CLOSE C1
```

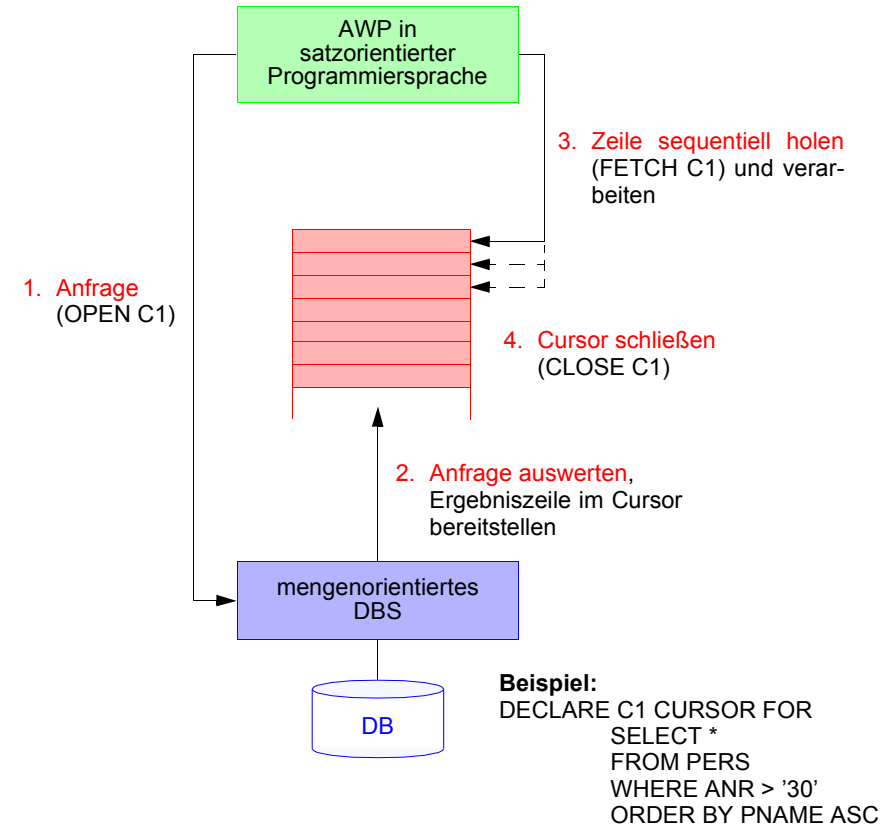


- **Reihenfolge der Ergebniszeilen**

- systembestimmt
- benutzerspezifisch (ORDER BY)

Cursor-Konzept (2)

- **Veranschaulichung der Cursor-Schnittstelle**



- **Wann wird die Ergebnismenge angelegt?**

- **lazy**: schrittweise Auswertung durch das DBS? Verzicht auf eine explizite Zwischenspeicherung ist nur bei einfachen Anfragen möglich
- **eager**: Kopie bei OPEN? Ist meist erforderlich (ORDER BY, Join, Aggregat-Funktionen, ...)

Cursor-Konzept (3)

- **Beispielprogramm in C (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 Anr 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;
```

- **Anmerkungen**

- 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)

Cursor-Konzept (4)

- **Aktualisierung mit Bezugnahme auf eine Position**

- Wenn die Zeilen, die ein Cursor verwaltet (*active set*), eindeutig Zeilen einer Tabelle entsprechen, können sie über Bezugnahme durch den Cursor geändert werden.
- Keine Bezugnahme bei INSERT möglich !

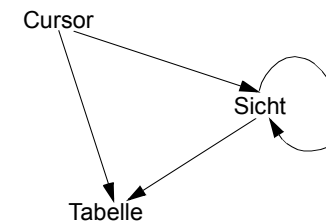
```
positioned-update ::=
    UPDATE table SET update-assignment-comma-list
    WHERE CURRENT OF cursor

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

- **Beispiel:**

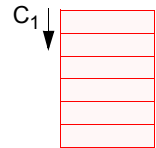
```
while (sqlcode == ok) {
  exec sql fetch C1 into :X;
  /* Berechne das neue Gehalt in Z */
  exec sql update Pers
    set Gehalt = :Z
    where current of C1;
}
```

- **Vergleich: Cursor – Sicht**

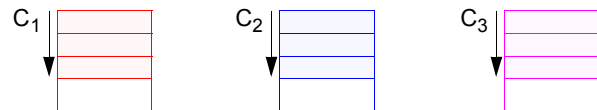


SQL-Programmiermodell für Mengenzugriff

- 1) **ein Cursor**: π , σ , \bowtie , \cup , $-$, \dots , Agg, Sort, \dots

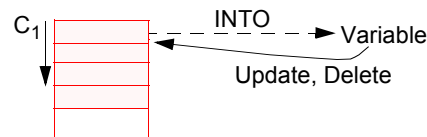


- 2) **mehrere Cursor**: π , σ , Sort, \dots

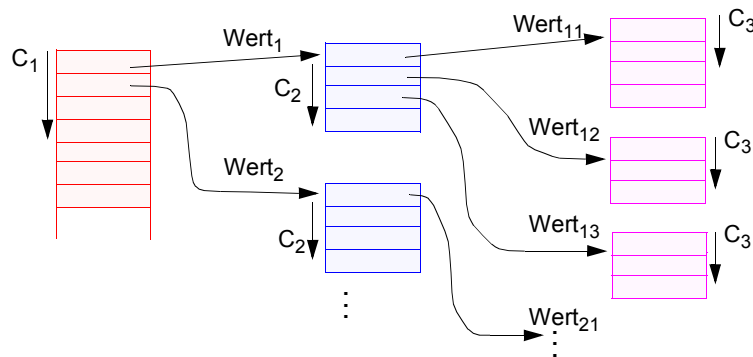


Verknüpfung der gesuchten Zeilen im AP

- 3) **positionsbezogene Aktualisierung**



- 4) **abhängige Cursor**



Beispiel: Stücklistenauflösung

- **Tabelle Struktur (Otnr, Utnr, Anzahl)**

- Aufgabe: Ausgabe aller Endprodukte sowie deren Komponenten
- max. Schachtelungstiefe sei bekannt (hier: 2)

```
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);
```

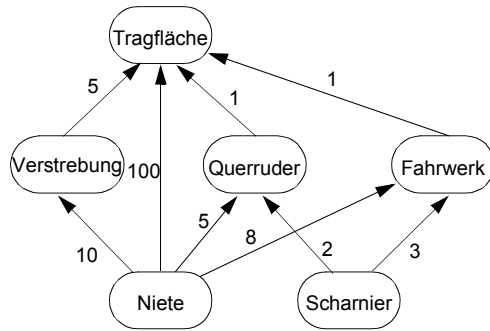
```
exec sql declare C1 cursor for
select Utnr, Anzahl from Struktur
where Otnr = :T0;
```

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

```
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); } /* end while (2) */
  exec sql close C1; } /* end while (1) */
exec sql close (C0);
```

Beispiel: Stücklistenauflösung (2)

- **Gozinto-Graph**



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

- **Strukturierte Ausgabe aller Teile von Endprodukten**

Erweiterung des Cursor-Konzeptes

```

cursor-def ::= DECLARE cursor [SENSITIVE | INSENSITIVE | ASENSITIVE]
              [SCROLL] CURSOR
              FOR table-exp
              [ORDER BY order-item-commalist]
              [FOR {READ ONLY | UPDATE [OF column-commalist]}]
  
```

- **Erweiterte Positionierungsmöglichkeiten durch SCROLL**

- **Cursor-Definition (Beispiel):**

```

EXEC SQL DECLARE C2 SCROLL CURSOR
FOR SELECT ...
  
```

- **Erweitertes FETCH-Statement:**

```

EXEC SQL FETCH[ [<fetch orientation>] FROM ] <cursor>
INTO <target list>
  
```

fetch orientation:

NEXT, PRIOR, FIRST, LAST

ABSOLUTE <expression>, RELATIVE <expression>

Bsp.:

```
EXEC SQL FETCH ABSOLUTE 100 FROM C2 INTO ...
```

```
EXEC SQL FETCH ABSOLUTE -10 FROM C2 INTO ...
(zehntletzte Zeile)
```

```
EXEC SQL FETCH RELATIVE 2 FROM C2 INTO ...
(übernächste Zeile)
```

```
EXEC SQL FETCH RELATIVE -10 FROM C2 INTO ...
```

Erweiterung des Cursor-Konzeptes (2)

- **Problemaspekt:**

Werden im geöffneten Cursor Änderungen sichtbar?

- **INSENSITIVE CURSOR**

- T sei die Zeilenmenge, die sich für den Cursor zum OPEN-Zeitpunkt (Materialisierung) qualifiziert
- Spezifikation von INSENSITIVE bewirkt, dass eine separate Kopie von T angelegt wird und der Cursor auf die Kopie zugreift
 - ➔ Aktualisierungen, die T betreffen, werden in der Kopie nicht sichtbar gemacht. Solche Änderungen könnten z. B. direkt oder über andere Cursor erfolgen
- Über einen insensitiven Cursor sind keine Aktualisierungsoperationen möglich (UPDATE nicht erlaubt)
- Die Kombination mit SCROLL bietet keine Probleme

- **ASENSITIVE (Standardwert)**

- Bei OPEN muss nicht zwingend eine Kopie von T erstellt werden: die Komplexität der Cursor-Definition verlangt jedoch oft seine Materialisierung als Kopie
- Ob Änderungen, die T betreffen und durch andere Cursor oder direkt erfolgen, in der momentanen Cursor-Instanzierung sichtbar werden, ist implementierungsabhängig
- Falls UPDATE deklariert wird, muss eine eindeutige Abbildung der Cursor-Zeilen auf die Tabelle möglich sein (siehe aktualisierbare Sicht). Es wird definitiv keine separate Kopie von T erstellt.

Aspekte der Anfrageverarbeitung

- **Deskriptive, mengenorientierte DB-Anweisungen**

- **Was**-Anweisungen sind in zeitoptimaler Folge interner DBMS-Operationen umzusetzen
- Anfrageauswertung/-optimierung des DBMS ist im wesentlichen für die effiziente Abarbeitung verantwortlich, d.h., das DBMS bestimmt, **wie** eine Ergebnismenge (abhängig von existierenden Zugriffspfaden) satzweise aufzusuchen und auszuwerten ist

- **Anfrageverarbeitung erfordert Vorbereitung der Anfrage**

- Parsing, Prüfung der syntaktischen Korrektheit
- Semantische Analyse (z.B. Auflösung von Sichten)
- Zugriffskontrolle und Berücksichtigung von Integritätsbedingungen
- **Anfrageoptimierung:**
Standardisierung und Vereinfachung (Normalform für den Anfragegraph), algebraische Optimierung (Restrukturierung aufgrund heuristischer Regeln), nicht-algebraische Optimierung (kostenbasierte Auswahl von Planoperatoren, die logische Operatoren implementieren)
liefert: Ausführungsplan
- Code-Generierung (bei Verwendung eines Kompilationsansatzes):
erzeugt ausführbares Zugriffsmodul, wird in einer DBMS-Bibliothek verwaltet

- **Ausführung der Anfrage**

- entweder Ausführung des Zugriffsmoduls (durch DBMS kontrolliert)
- oder Interpreter-basierte Abarbeitung des Ausführungsplans

- **Statisches, eingebettetes SQL - mögliche Vorgehensweisen**

- Anfrage wird als aktueller Parameter eines (internen) CALL-Aufrufs abgelegt, Vorbereitung (und Ausführung) erfolgen zur Laufzeit, oder
- Vorbereitung zum Übersetzungszeitpunkt, Ausführung zur Laufzeit

Dynamisches SQL

- **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 DBMS
 - ➔ **Dynamisches SQL erlaubt Behandlung solcher Fälle**
- **Mehrere Sprachansätze**
 - Eingebettetes dynamisches SQL
 - Call-Level-Interface (CLI):
 - SQL/CLI oder ODBC-Schnittstelle¹ für prozedurale Programmiersprachen
 - Java Database Connectivity² (JDBC) zur Verwendung mit Java
 - ➔ **Funktionalität ähnlich, jedoch nicht identisch**
- **Gleiche Anforderungen (LZ)**
 - Zugriff auf Metadaten
 - Übergabe und Abwicklung dynamisch berechneter SQL-Anweisungen
 - Optionale Trennung von Vorbereitung und Ausführung
 - einmalige Vorbereitung mit Platzhalter (?) für Parameter
 - n-malige Ausführung
 - Explizite Bindung von Platzhaltern (?) an Wirtsvariable
 - Variable sind zur ÜZ nicht bekannt!
 - Variablenwert wird zur Ausführungszeit vom Parameter übernommen

Eingebettetes dynamisches SQL (EDSQL)

- **Wann wird diese Schnittstelle gewählt?**
 - Sie unterstützt auch andere Wirtssprachen als C
 - Sie ist im Stil statischem SQL ähnlicher; sie 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
- **EDSQL**
 - besteht im wesentlichen aus 4 Anweisungen:
 - DESCRIBE
 - PREPARE
 - EXECUTE
 - EXECUTE IMMEDIATE
- **SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt**
 - Deklaration DECLARE STATEMENT
 - Anweisungen enthalten Platzhalter für Parameter (?) statt Programmvariablen

1. Die Schnittstelle Open Database Connectivity (ODBC) wird von Microsoft definiert.
2. 'de facto'-Standard für den Zugriff auf relationale Daten von Java-Programmen aus: Spezifikation der JDBC-Schnittstelle unter <http://java.sun.com/products/jdbc>

Eingebettetes dynamisches SQL (2)

• Trennung von Vorbereitung und Ausführung

```
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;
```

• Bei einmaliger Ausführung EXECUTE IMMEDIATE ausreichend

```
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

```
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 (3)

• Dynamische Parameterbindung

```
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 DBMS verwaltet wird (kein SQLDA vorhanden)

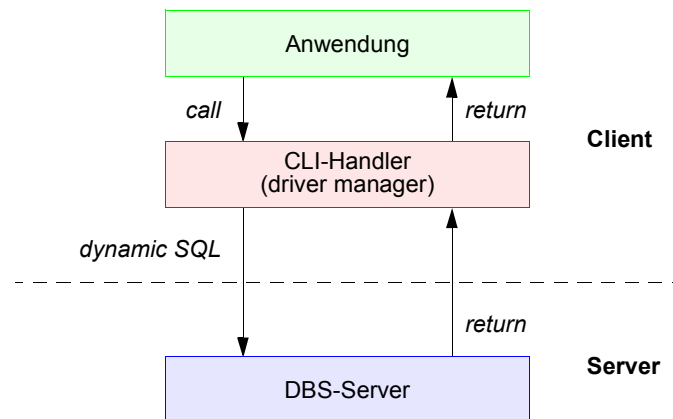
```
Anweisung = 'INSERT INTO Pers VALUES (?, ?, ...)';  
exec sql prepare SQLanw from :Anweisung;  
exec sql allocate descriptor 'Eingabeparameter';  
exec sql describe input SQLanw into sql descriptor 'Eingabeparameter';  
exec sql get descriptor 'Eingabeparameter' :n = count;
```

```
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';
```

Call-Level-Interface

- **Spezielle Form von dynamischem SQL**
 - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
 - Direkte Aufrufe der Routinen einer standardisierten Bibliothek
 - Keine Vorübersetzung (Behandlung der DB-Anweisungen) von Anwendungen
 - Vorbereitung der DB-Anweisung geschieht erst beim Aufruf zur LZ
 - Anwendungen brauchen nicht im Quell-Code bereitgestellt werden
 - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools
- ➔ **Schnittstelle wird sehr häufig in der Praxis eingesetzt!**
- **Einsatz typischerweise in Client/Server-Umgebung**



Call-Level-Interface (2)

- **Vorteile von CLI**
 - **Schreiben portabler Anwendungen**
 - keinerlei Referenzen auf systemspezifische Kontrollblöcke wie SQLCA/SQLDA
 - kann die ODBC-Schnittstelle implementieren
 - **Systemunabhängigkeit**
 - Funktionsaufrufe zum standardisierten Zugriff auf den DB-Katalog
 - **Mehrfache Verbindungen zur selben DB**
 - unabhängige Freigabe von Transaktionen in jeder Verbindung
 - nützlich für AW mit GUIs (graphical user interfaces), die mehrere Fenster benutzen
 - **Optimierung des Zugriffs vom/zum Server**
 - Holen von mehreren Zeilen pro Zugriff
 - Lokale Bereitstellung einzelner Zeilen (Fetch)

DB-Zugriff via JDBC

• Java Database Connectivity Data Access API (JDBC)³

- unabhängiges, standardisiertes CLI, basierend auf SQL:1999
- bietet Schnittstelle für den Zugriff auf (objekt-)relationale DBMS aus Java-Anwendungen
- besteht aus zwei Teilen
 - Core Package: Standardfunktionalität mit Erweiterungen (Unterstützung von SQL:1999-Datentypen, flexiblere ResultSets, ...)
 - Optional Package: Ergänzende Funktionalität (Connection Pooling, verteilte Transaktionen, ...)

• Allgemeines Problem

Verschiedene DB-bezogene APIs sind aufeinander abzubilden



• Überbrückung/Anpassung durch Treiber-Konzept

- setzen JDBC-Aufrufe in die DBMS-spezifischen Aufrufe um
- Treiber werden z.B. vom DBMS-Hersteller zur Verfügung gestellt
- Treiber-Unterstützung kann auf vier verschiedene Arten erfolgen

3. Standard: JDBC API 3.0 Specification Final Release
<http://java.sun.com/products/jdbc>

JDBC – wichtige Funktionalität

• Laden des Treiber

- kann auf verschiedene Weise erfolgen, z.B. durch explizites Laden mit dem Klassenlader:

```
Class.forName (DriverClassName)
```

• Aufbau einer Verbindung

- Connection-Objekt repräsentiert die Verbindung zum DB-Server
- Beim Aufbau werden URL der DB, Benutzername und Paßwort als Strings übergeben:

```
Connection con = DriverManager.getConnection (url, login, pwd);
```

• Anweisungen

- Mit dem Connection-Objekt können u.a. Metadaten der DB erfragt und Statement-Objekte zum Absetzen von SQL-Anweisungen erzeugt werden
- Statement-Objekt erlaubt das Erzeugen einer SQL-Anweisung zur direkten (einmaligen) Ausführung

```
Statement stmt = con.createStatement();
```

- PreparedStatement-Objekt erlaubt das Erzeugen und Vorbereiten von (parametrisierten) SQL-Anweisungen zur wiederholten Ausführung

```
PreparedStatement pstmt = con.prepareStatement (  
    "select * from personal where gehalt >= ?");
```

- Ausführung einer Anfrageanweisung speichert ihr Ergebnis in ein spezifisiertes ResultSet-Objekt

```
ResultSet res = stmt.executeQuery ("select name from personal");
```

• Schließen von Verbindungen, Statements usw.

```
stmt.close();  
con.close();
```

JDBC – Anweisungen

- **Anweisungen (Statements)**

- Sie werden in einem Schritt vorbereitet und ausgeführt
- Sie entsprechen dem Typ EXECUTE IMMEDIATE im dynamischen SQL
- JDBC-Methode erzeugt jedoch ein Objekt zur Rückgabe von Daten

- **executeUpdate-Methode**

wird zur direkten Ausführung von UPDATE-, INSERT-, DELETE- und DDL-Anweisungen benutzt

```
Statement stat = con.createStatement ();
int n = stat.executeUpdate ("update personal
                           set gehalt = gehalt * 1.1
                           where gehalt < 5000.00");
// n enthält die Anzahl der aktualisierten Zeilen
```

- **executeQuery-Methode**

führt Anfragen aus und liefert Ergebnismenge zurück

```
Statement stat1 = con.createStatement ();
ResultSet res1 = stat1.executeQuery (
    "select pnr, name, gehalt from personal where
    gehalt >=" + gehalt);

// weitere Verarbeitungsschritte: siehe JDBC-Ergebnismengen und Cursor
```

JDBC – Prepared-Anweisungen

- **PreparedStatement-Objekt**

```
PreparedStatement pstmt;
double gehalt = 5000.00;
pstmt = con.prepareStatement (
    "select * from personal where gehalt >= ?");
```

- Vor der Ausführung sind dann die aktuellen Parameter einzusetzen mit Methoden wie setDouble, setInt, setString usw. und Indexangabe

```
pstmt.setDouble (1, gehalt);
```
- Neben setXXX () gibt es Methoden getXXX () und updateXXX () für alle Basistypen von Java

- **Ausführen einer Prepared-Anweisung als Anfrage**

```
ResultSet res1 = pstmt.executeQuery ();
```

- **Vorbereiten und Ausführung einer Prepared-Anweisung zur DB-Aktualisierung**

```
pstmt = con.prepareStatement (
    "delete from personal
    where name = ?");
// set XXX-Methode erlaubt die Zuweisung von aktuellen Werten
pstmt.setString (1, "Maier")

int n = pstmt.executeUpdate ();

// Methoden für Prepared-Anweisungen haben keine Argumente
```

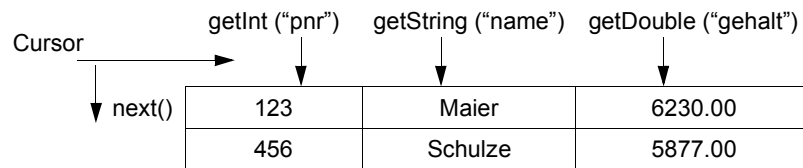
JDBC – Ergebnismengen und Cursor

• Select-Anfragen und Ergebnisübergabe

- Jede JDBC-Methode, mit der man Anfragen an das DBMS stellen kann, liefert ResultSet-Objekte als Rückgabewert

```
ResultSet res = stmt.executeQuery (
    "select pnr, name, gehalt from personal where
    gehalt >=" + gehalt);
```

- Cursor-Zugriff und ist durch die Methode next() der Klasse ResultSet implementiert
- Übertragung und ggf. Typkonvertierung von Resultatswerten in Java-Variablen erfolgt durch typspezifische getter-Methoden („getXXX()“)



- Zugriff aus Java-Programm


```
while (res.next() ) {
    System.out.print (res.getInt ("pnr") + "\t");
    System.out.print (res.getString ("name") + "\t");
    System.out.println (res.getDouble ("gehalt") );
}
```

• Aktualisierbare ResultSets

```
Statement s1 = con1.createStatement (
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE);
ResultSet res= s1.executeQuery ( . . . ); . . .
res.updateString ("name", "Müller"); . . .
res.updateRow ();
```

- Zeilen können in aktualisierbaren ResultSets geändert und gelöscht werden. Mit res.insertRow () wird eine Zeile in res und gleichzeitig auch in die DB eingefügt.

JDBC – Ergebnismengen und Cursor (2)

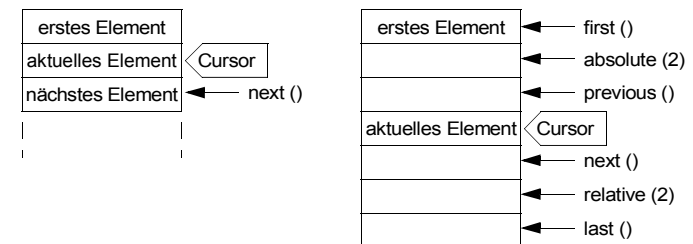
JDBC definiert drei Typen von ResultSets

• ResultSet: forward-only

Default-Cursor vom Typ ASENSITIVE: nur next()

• ResultSet: scroll-insensitive

Scroll-Operationen sind möglich, aber DB-Aktualisierungen verändern ResultSet nach seiner Erstellung nicht



• ResultSet: scroll-sensitive

- Scroll-Operationen sind möglich, wobei ein nicht-INSENSITIVE Cursor benutzt wird
- Semantik der Operation, im Standard nicht festgelegt, wird vom darunterliegenden DBMS übernommen, die vom Hersteller definiert wird!
- Oft wird die sogen. KEYSET_DRIVEN-Semantik⁴ (Teil des ODBC-Standards) implementiert.

4. Bei Ausführung der Select-Anweisung wird der ResultSet durch eine Menge von Zeigern auf die sich qualifizierenden Zeilen repräsentiert. Änderungen und Löschungen nach Erstellen des ResultSet werden dadurch sichtbar gemacht, Einfügungen aber nicht!

JDBC – Zugriff auf Metadaten

• Allgemeine Metadaten

- Welche Information benötigt ein Browser, um seine Arbeit beginnen zu können?
- JDBC besitzt eine Klasse DatabaseMetaData, die zum Abfragen von Schema- und anderer DB-Information herangezogen wird

• Informationen über ResultSets

- JDBC bietet die Klasse ResultSetMetaData
 - ResultSet rs1 = stmt1.executeQuery ("select * from personal");
 - ResultSetMetaData rsm1 = rs1.getMetaData ();
- Es müssen die Spaltenanzahl sowie die einzelnen Spaltennamen und ihre Typen erfragt werden können (z. B. für die erste Spalte)
 - int AnzahlSpalten = rsm1.getColumnCount ();
 - String SpaltenName = rsm1.getColumnName (1);
 - String TypName = rsm1.getColumnName (1);
- Ein Wertzugriff kann dann erfolgen durch
 - rs1.getInt (2), wenn
 - rsm1.getColumnName (2)
 - den String "Integer" zurückliefert.

JDBC – Fehler und Transaktionen

• Fehlerbehandlung

- Spezifikation der Ausnahmen, die eine Methode werfen kann, bei ihrer Deklaration (throw exception)
- Ausführung der Methode in einem try-Block, Ausnahmen werden im catch-Block abgefangen

```
try {  
    . . . Programmcode, der Ausnahmen verursachen kann  
}  
catch (SQLException e) {  
    System.out.println ("Es ist ein Fehler aufgetreten :\n");  
    System.out.println ("Msg: " + e.getMessage ());  
    System.out.println ("SQLState: " + e.getSQLState ());  
    System.out.println ("ErrorCode: " + e.getErrorCode ());  
};
```

• Transaktionen

- Bei Erzeugen eines Connection-Objekts (z.B. con1) ist als Default der Modus **autocommit** eingestellt
- Um Transaktionen als Folgen von Anweisungen abwickeln zu können, ist dieser Modus auszuschalten
 - con1.setAutoCommit(false);

• Beendigung oder Zurücksetzen

```
con1.commit();  
con1.rollback();
```

• Programm kann mit mehreren DBMS verbunden sein

- selektives Beenden/Zurücksetzen von Transaktionen pro DBMS
- kein globales atomares Commit möglich

DB-Zugriff via JDBC – Beispiel 1

```
import java.sql.*;
public class Select {
    public static void main (String [] args) {
        Connection con = null;
        PreparedStatement pstmt;
        ResultSet res;
        double gehalt = 5000.00;
        try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
            con = java.sql.DriverManager.getConnection (
                "jdbc:odbc:personal", "user", "passwd");
            pstmt = con.prepareStatement (
                "select pnr, name, gehalt from personal where gehalt >= ?");
            pstmt.setDouble (1, gehalt);
            ...
            res = pstmt.executeQuery ();
            while (res.next () ) {
                System.out.print (res.getInt ("pnr") + "\t");
                System.out.print (res.getString ("name") + "\t");
                System.out.println (res.getDouble ("gehalt") );
            }
            res.close ();
            pstmt.close ();
        } // try
        catch (SQLException e) {
            System.out.println (e) ;
            System.out.println (e.getSQLState () );
            System.out.println (e.getErrorCode () );
        }
        catch (ClassNotFoundException e) {
            System.out.println (e) ;
        }
    } // main
} // class Select
```

DB-Zugriff via JDBC – Beispiel 2

```
import java.sql.*;
public class Insert {
    public static void main (String [] args) {
        Connection con = null;
        PreparedStatement pstmt;
        try {
            Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
            con = java.sql.DriverManager.getConnection (
                "jdbc:odbc:personal", " ", " ");
            pstmt = con.prepareStatement (
                "insert into personal values (?, ?, ?)");
            pstmt.setInt (1, 222);
            pstmt.setString (2, "Schmitt");
            pstmt.setDouble (3, 6000.00);
            pstmt.executeUpdate ();
            pstmt.close ();
            con.close ();
        } // try
        catch (SQLException e) {
            System.out.println (e);
            System.out.println (e.getSQLState () );
            System.out.println (e.getErrorCode () );
        }
        catch (ClassNotFoundException e) {System.out.println (e);
        }
    }
    ...
    pstmt = con.prepareStatement (
        "update personal set gehalt = gehalt * 1.1 where gehalt < ?");
    pstmt.setDouble (1, 10000.00);
    pstmt.executeUpdate ();
    pstmt.close ();
    ...
    pstmt = con.prepareStatement ("delete from personal where pnr = ?");
    pstmt = setInt (1, 222);
    pstmt.executeUpdate ();
    pstmt.close ();
```

SQL/PSM

- **PSM**

(Persistent Stored Modules)

- 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**

- ins AWP eingebettet

```
...
EXEC SQL INSERT INTO Pers VALUES (...);
EXEC SQL INSERT INTO Abt VALUES (...);
...
```
- Erzeugen einer SQL-Prozedur

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

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

- **Vorteile**

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

SQL/PSM – 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 division_by_zero |

5 - 33

Zusammenfassung

- **Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen**
 - Anpassung von mengenorientierter Bereitstellung und satzweiser Verarbeitung von DBMS-Ergebnissen
 - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
 - Erweiterungen: Scroll-Cursor, Sichtbarkeit von Änderungen
- **Statisches (eingebettetes) SQL**
 - relativ einfache Programmierung
 - Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen
 - zur Laufzeit kann nur eine Datenbanken angesprochen werden
 - hohe Effizienz, gesamte Typprüfung und Konvertierung, Vorbereitung der Anfrage kann durch Precompiler erfolgen
- **Dynamisches SQL**
 - Festlegung/Übergabe von SQL-Anweisungen zur Laufzeit
 - hohe Flexibilität, schwierige Programmierung
 - explizite Anweisungen zur Datenabbildung zw. DBMS und Anwendung
 - klare Trennung von Anwendungsprogramm und SQL
(➔ einfacheres Debugging)
- **CLI**
 - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
 - JDBC bietet Schnittstelle für Java-Anwendungen
 - Keine Vorübersetzung oder Vorbereitung
 - Anwendungen brauchen nicht im Quell-Code bereitgestellt werden
 - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools
- **PSM, Stored Procedures**
 - zielt ab auf Leistungsverbesserung vor allem in Client/Server-Umgebung
 - erhöht die Verarbeitungsmächtigkeit des DBMS

5 - 34