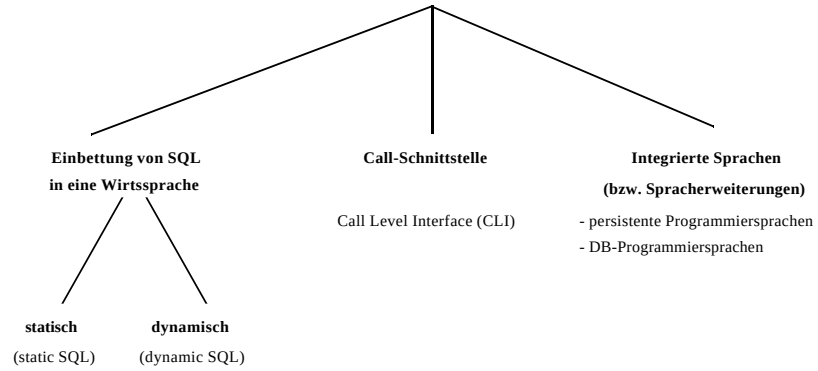
4. Anwendungsprogrammier-Schnittstelle

- **Kopplung mit einer Wirtssprache[1]:**
  Übersicht und Aufgaben
- **Eingebettetes statisches SQL**
  - Cursor-Konzept
  - SQL-Programmiermodell
  - Rekursion
  - Erweiterung des Cursor-Konzeptes
  - Ausnahme- und Fehlerbehandlung
- **Aspekte der Anfrageauswertung**
  - Aufgaben bei der Anfrageauswertung
  - Vorbereitung und Übersetzung
  - Bindung und Datenunabhängigkeit
- **SQL/PSM**
- **Dynamisches SQL**
  - Eingebettetes dynamisches SQL
  - Call-Level-Interface
- **Anhang:**
  Open Data Base Connectivity (ODBC)
- **DB-Zugriff aus Java-Programmen**
  - DB-Zugriff via JDBC
  - SQLJ

1. Synonyme: Relation - Tabelle, Tupel - Zeile, Attribut - Spalte, Attributwert - Zelle

Kopplung mit einer Wirtssprache



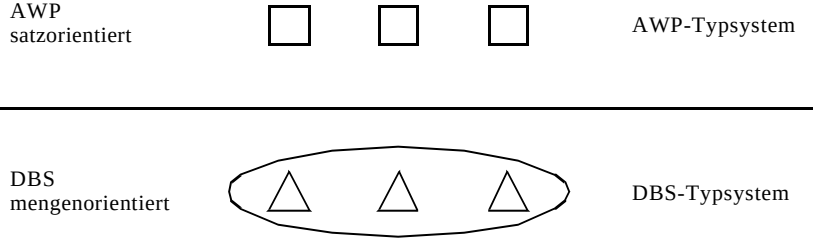
- **Call-Schnittstelle**
  (prozedurale Schnittstelle, CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
- **Einbettung von SQL** (Embedded SQL, ESQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - komfortablere 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 zur Laufzeit

Kopplung mit einer Wirtssprache (2)

- **Integrationsansätze unterstützen typischerweise nur**
  - ein Typsystem
  - Navigation (satz-/objektorientierter Zugriff)

  å 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?

  å 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 im AWP?

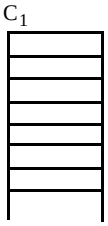
- **Cursor-Deklaration**

```
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**

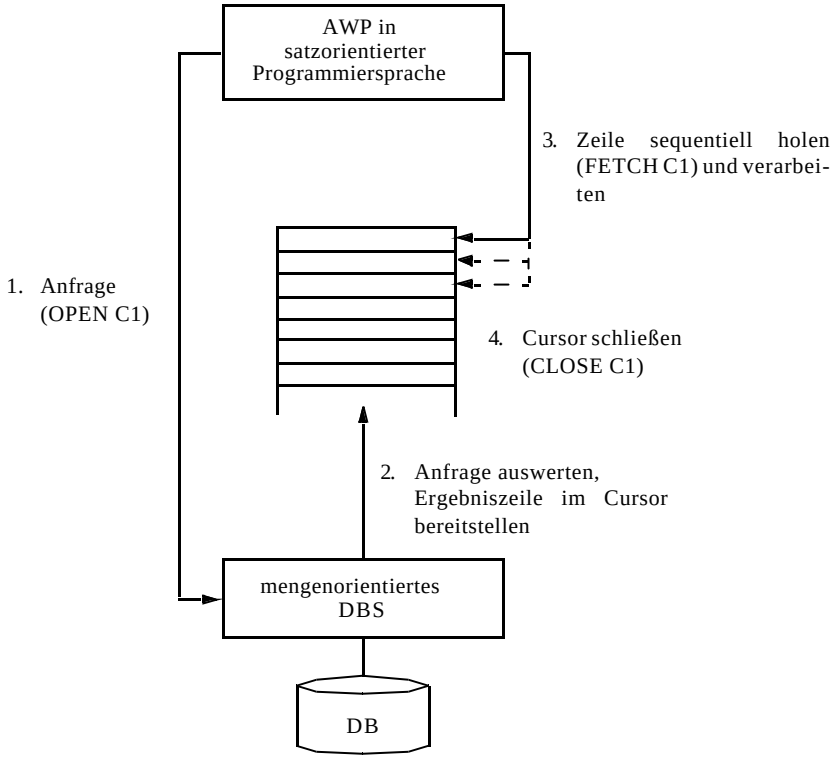
```
OPEN   C1
FETCH C1  INTO  Var1, Var2, . . ., Varn
CLOSE C1
```



- **Reihenfolge der Ergebniszeilen**
  - systembestimmt
  - benutzerspezifiziert (ORDER BY)

Cursor-Konzept (2)

- **Veranschaulichung der Cursor-Schnittstelle**



- **Wann wird die Ergebnismenge angelegt?**
  - schritthaltende Auswertung durch das DBS?
    Verzicht auf eine explizite Zwischenspeicherung ist nur bei einfachen Anfragen möglich
  - 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 !

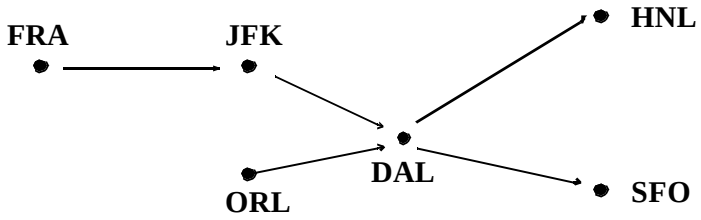
```
positioned-update ::=
        UPDATE table SET update-assignment-commalist
        WHERE CURRENT OF cursor

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;
}
```

- **Vergleich: Cursor - Sicht**

Rekursion in SQL ?



- Ausschnitt aus Tabelle Flüge

Flüge	(Nr,	Ab,	An,	Ab-Zeit,	An-Zeit, . . .)
	AA07	FRA	JFK		• • •
	AA43	JFK	ORL		
	AA07	JFK	DAL		
	AA85	ORL	DAL		
	AA70	DAL	HNL		
	• • •				

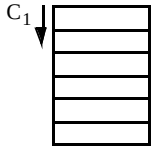
- Flug von FRA nach HNL ?

```
SELECT   Ab, An, . . .
FROM     Flüge
WHERE    Ab = 'FRA'  AND  An = 'HNL'
```

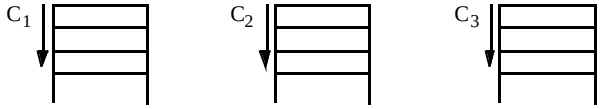
- Flug von FRA nach HNL (Anzahl der Teilstrecken bekannt) ?

SQL-Programmiermodell

1) **ein Cursor**: π, σ, ⋈, ∪, - ,. . . , Agg, Sort, . . .

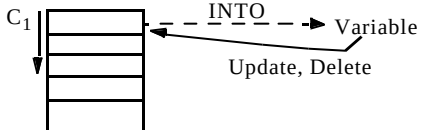


2) **mehrere Cursor**: π, σ, Sort, ...

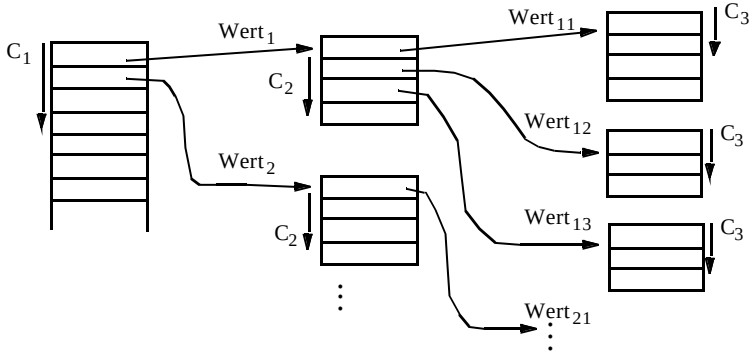


Verknüpfung der gesuchten Zeilen im AP

3) **Positionsbezogene Aktualisierung**



4) **abhängige Cursor**



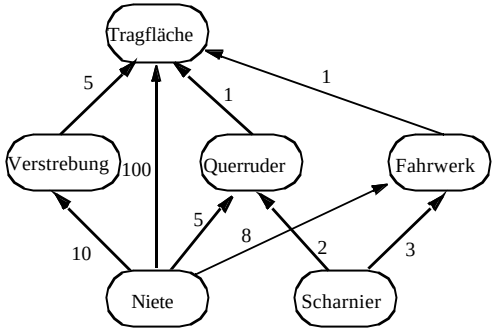
Beispiel: Stücklistenauflösung

- 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 [WITH HOLD] [WITH RETURN]
                        FOR table-exp
                     [ORDER BY order-item-commalist]
                  [FOR {READ ONLY | UPDATE [OF column-commalist]}]
```

Erweiterte Positionierungsmöglichkeiten durch SCROLL

Cursor-Definition (Bsp.):

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
    - å 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-Instanziierung 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.

Erweiterung des Cursor-Konzeptes (3)

- **Sichtbarkeit von Änderungen:**

```
exec sql declare C1 cursor for
   select Pnr, Gehalt from Pers where Anr = 'K55';
exec sql declare C2 cursor for
   select Pnr, Beruf, Gehalt from Pers where Anr > 'K53';


exec sql fetch C1 into :Y, :Z;        /* Berechne das neue Gehalt in Z /*
   . . .
exec sql update Pers
   set Gehalt = :Z
   where current of C1;
   . . .
exec sql fetch C2 into :U, :V, :W;  /* Welches Gehalt wird in W übergeben? /*
```

- **Fallunterscheidung**

Ausnahme- und Fehlerbehandlung

- **Indikatorkonzept:**

  Indikatorvariablen zum Erkennen von Nullwerten

```
        EXEC SQL FETCH C INTO :X INDICATOR :X_Indic
bzw.    EXEC SQL FETCH C INTO :X :X_indic, :Y :Y_Indic;
```

- **mögliche Werte einer Indikatorvariable**

  = 0: zugehörige Wirtsprogrammvariable hat regulären Wert

  = -1: es liegt ein Nullwert vor

  > 0: zugehörige Wirtsprogrammvariable enthält abgeschnittene Zeichenkette

- **Beispiel:**

```
exec sql begin declare section;
     int pnummer, mnummer, mind;
exec sql end declare section;

/* Überprüfen von Anfrageergebnissen */
exec sql select Mnr into :mnummer :mind
          from Pers
          where Pnr = :pnummer;
if (mind == 0) { /* kein Nullwert */
else { /* Nullwert */ }

/* ermöglicht die Kennzeichnung von Nullwerten */
exec sql insert into Pers (Pnr, Mnr)
          values ( :pnummer, :mnummer indicator :mind);
```

Ausnahme- und Fehlerbehandlung (2)

- **SQL-Implementierungen verwenden meist vordefinierten Kommunikationsbereich zwischen DBS und AP: SQL Communication Area**

  EXEC SQL INCLUDE SQLCA;

  enthält u.a. Integer-Variable SQLCODE

- **SQL92 nutzt neben SQLCODE** (aus Kompatibilität zu SQL89) **neue Variable SQL-STATE**
  - standardisierte Fehler-Codes
  - nähere Angaben zu Fehlersituationen in einem Diagnostik-Bereich des DBS

  å Anweisung GET DIAGNOSTICS

- **WHENEVER-Anweisung**

  EXEC SQL WHENEVER <Bedingung> <Aktion>;

  - Vordefinierte Bedingungen: NOT FOUND, SQLWARNING, SQLERROR
  - Aktionen: STOP, CONTINUE, GOTO <label>
  - WHENEVER ist Anweisung an Vorübersetzer, nach jeder SQL-Anweisung entsprechende SQLCODE- bzw. SQLSTATE-Prüfung einzubauen

Wirtssprachen-Einbettung und Übersetzung

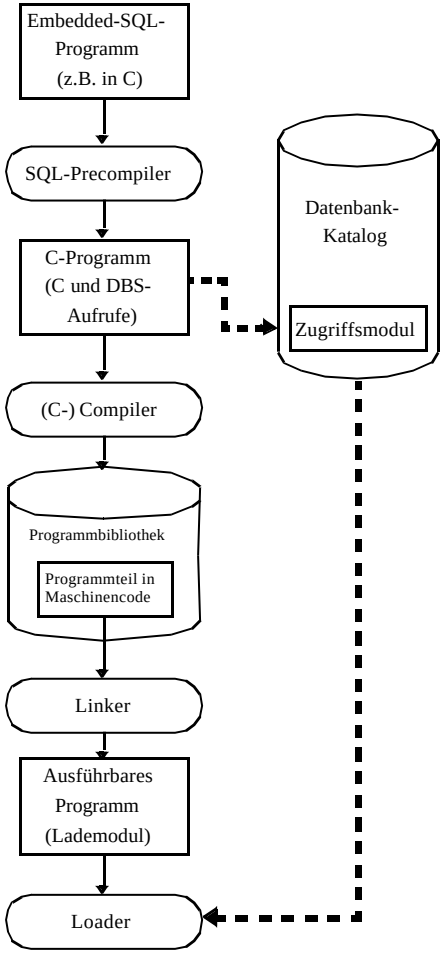
- **Prinzipielle Möglichkeiten**
  - **Direkte Einbettung**
    - keine syntaktische Unterscheidung zwischen Programm- und DB-Anweisungen
    - DB-Anweisung wird als Zeichenkette A ins AP integriert, z. B. **exec sql** open C1
  - **Aufruftechnik**
    DB-Anweisung wird durch expliziten Funktionsaufruf an das Laufzeitsystem des DBS übergeben, z. B.
    CALL DBS (‘open C1’))
    - Es sind prinzipiell keine DBS-spezifischen Vorkehrungen bei der AP-Übersetzung erforderlich!
    - Verschiedene Formen der Standardisierung:
      Call-Level-Interface (CLI), JDBC

- Eingebettetes SQL verlangt **Maßnahmen bei der AP-Übersetzung**
  - typischerweise Einsatz eines Vorübersetzers PC (Precompiler)[1]
  - PC erzeugt für DB-Anweisungen spezielle Call-Aufrufe im AP, so daß das modifizierte AP mit dem Wirtssprachencompiler C übersetzt werden kann

- **Vorbereitung der DB-Anweisung:**
  Was passiert wann?

1. Sonst ist ein erweiterter Compiler C’ der Wirtssprache bereitzustellen, der sowohl Anweisungen der Wirtssprache als auch der DB-Sprache behandeln kann.

Von der Übersetzung bis zur Ausführung - bei Einsatz eines Vorübersetzers -



- **Vorübersetzung des AP**

- Entfernung aller Embedded-SQLAnweisungen aus dem Programm (Kommentare)
- Ersetzung durch Programmiersprachen-spezifische DBS-Aufrufe
- Erzeugung eines „SQL-freien" Programmes 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

- **Binden**

- Zusammenfügen aller 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

Aspekte der Anfrageauswertung - zentrale Probleme

- **Deskriptive, mengenorientierte DB-Anweisungen**

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

- **Welche Auswertungstechnik soll gewählt werden?**

  Spektrum von Verfahren mit folgenden Eckpunkten:

  - **Maximale Vorbereitung**
    - Für die DB-Anweisungen von AP wird ein zugeschnittenes Programm (Zugriffsmodul) zur Übersetzungszeit (ÜZ) erzeugt
    - Zur Ausführung einer DB-Anweisung (Laufzeit (LZ)) wird das Zugriffsmodul geladen und abgewickelt. Dabei wird durch Aufrufe des DBVS (genauer: des Zugriffssystems) das Ergebnis abgeleitet.
  - **Keine Vorbereitung**
    - Technik ist typisch für Call-Schnittstellen (dynamisches SQL)
    - Allgemeines Programm (Interpreter) akzeptiert DB-Anweisungen als Eingabe und erzeugt durch Aufrufe des Zugriffssystems das Ergebnis

- **Wahl des Bindezeitpunktes**

  - Wann werden die für die Abwicklung einer DB-Anweisung erforderlichen Operationen von DB-Schema abhängig?
  - Übersetzungszeit vs. Laufzeit

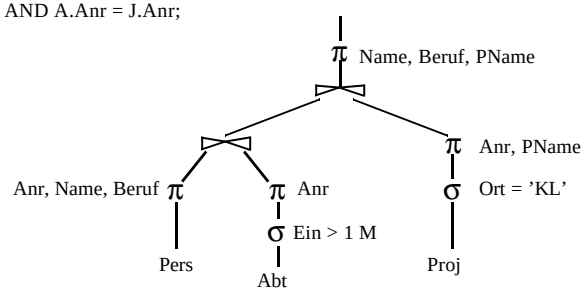
Aspekte der Anfrageauswertung

- **Problemdarstellung - Beispiel**

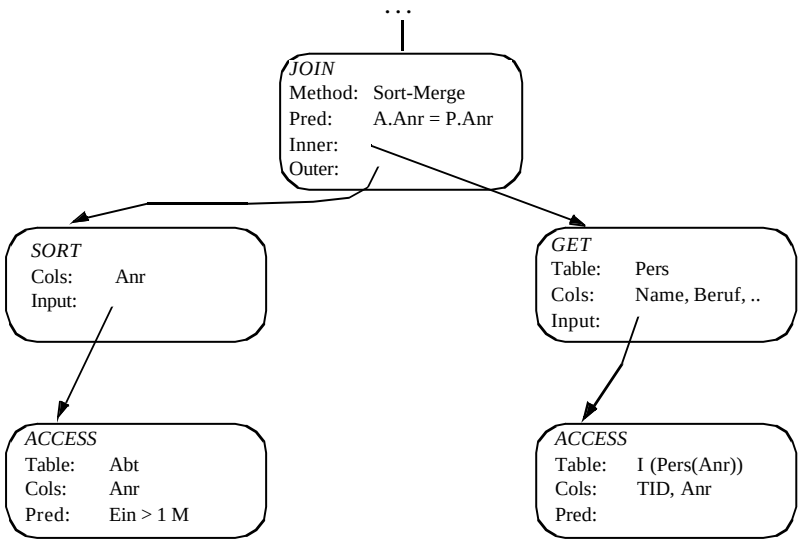
SQL:

```
SELECT   P.Name, P.Beruf, J.PName
FROM     Pers P, Abt A, Proj J
WHERE    A.Ein > 1000000 AND J.Ort = 'KL'
  AND    A.Anr = P.Anr AND A.Anr = J.Anr;
```

Zugehöriger Anfragegraph (mit algebraischer Optimierung)



- **Ausschnitt aus einen möglichen Zugriffsplan**



Auswertung von DB-Anweisungen

- **Verarbeitungsschritte** zur Auswertung von DB-Anweisungen:

1. Lexikalische und syntaktische Analyse

- Erstellung eines Anfragegraphs (AG) als Bezugsstruktur für die nachfolgenden Übersetzungsschritte
- Überprüfung auf korrekte Syntax (Parsing)

2. Semantische Analyse

- Feststellung der Existenz und Gültigkeit der referenzierten Tabellen, Sichten und Attribute
- Einsetzen der Sichtdefinitionen in den AG
- Ersetzen der externen durch interne Namen (Namensauflösung)
- Konversion vom externen Format in interne Darstellung

3. Zugriffs- und Integritätskontrolle

sollen aus Leistungsgründen, soweit möglich, schon zur Übersetzungszeit erfolgen

- Zugriffskontrolle erfordert bei Wertabhängigkeit Generierung von Laufzeitaktionen
- Durchführung einfacher Integritätskontrollen (Kontrolle von Formaten und Konversion von Datentypen)
- Generierung von Laufzeitaktionen für komplexere Kontrollen

4. Standardisierung und Vereinfachung

dienen der effektiveren Übersetzung und frühzeitigen Fehlererkennung

- Überführung des AG in eine Normalform
- Elimination von Redundanzen

Auswertung von DB-Anweisungen (2)

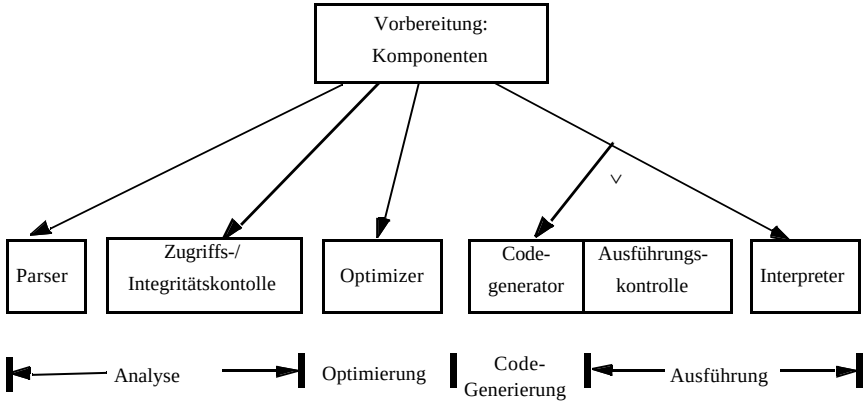
5. Restrukturierung und Transformation

- Restrukturierung zielt auf globale Verbesserung des AG ab; bei der Transformation werden ausführbare Operationen eingesetzt
- Anwendung von heuristischen Regeln (algebraische Optimierung) zur Restrukturierung des AG
- Transformation führt Ersetzung und ggf. Zusammenfassen der logischen Operatoren durch Planoperatoren durch (nicht-algebraische Optimierung): Meist sind mehrere Planoperatoren als Implementierung eines logischen Operators verfügbar
- Bestimmung alternativer Zugriffspläne (nicht-algebraische Optimierung): Meist sind viele Ausführungsreihenfolgen oder Zugriffspfade auswählbar
- Bewertung der Kosten und Auswahl des günstigsten Ausführungsplanes

  å Schritte 4 + 5 werden als Anfrageoptimierung zusammengefaßt

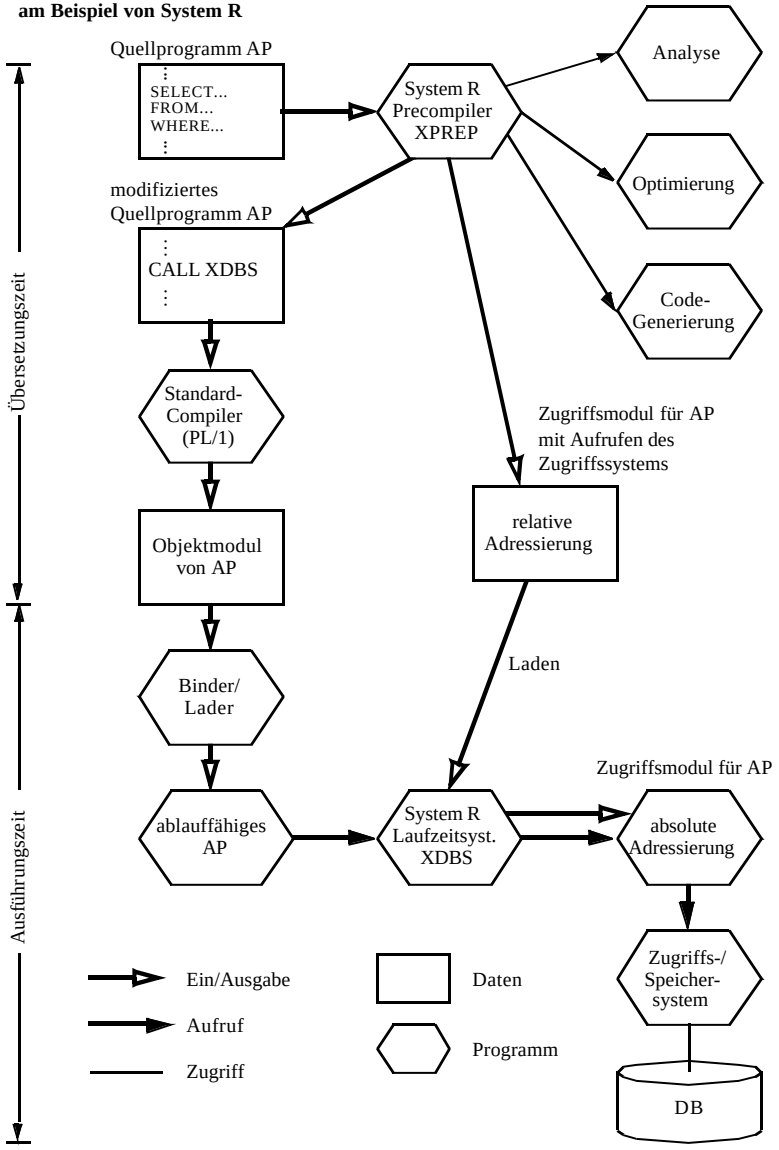
6. Code-Generierung

- Generierung eines zugeschnittenen Programms für die vorgegebene (SQL-) Anweisung
- Erzeugung eines ausführbaren Zugriffsmoduls
- Verwaltung der Zugriffsmodule in einer DBVS-Bibliothek



Vorbereitung und Ausführung von DB-Anweisungen

am Beispiel von System R



Auswertung von DB-Anweisungen (3)

- **Verschiedene Ansätze der Vorbereitung einer DB-Anweisung (zur Übersetzungszeit des AP)**
  - keine Vorbereitung
    DB-Anweisung wird aktueller Parameter einer Call-Anweisung im AP
  - Erstellung des Anfragegraphs (1-3)
  - Erstellung eines Zugriffsplans (1-5)
  - Erstellung eines Zugriffsmoduls (1-6)

- **Kosten der Auswertung**
  - Vorbereitung (ÜZ) + Ausführung (LZ)
  - Vorbereitung erfolgt durch „Übersetzung“ (Ü)
  - Ausführung
    - Laden und Abwicklung (A) des Zugriffsmoduls
    - sonst: Interpretation (I) der vorliegenden Struktur

- **Aufteilung der Kosten**

	Übersetzungszeit			**Laufzeit**
Vorbereitung	Analyse	Optimierung	Code-Gen.	**Ausführung**
Zugriffsmodul				
Zugriffsplan				
Anfragegraph				
keine				

Auswertung von DB-Anweisungen (4)

- **Vorbereitung erzeugt Bindung an DB-Schema (Abhängigkeiten!)**

- **Was heißt „Binden“?**

AP:

```
SELECT  Pnr, Name, Gehalt
FROM    Pers
WHERE   Beruf = ‘Programmierer‘
```

DB-Katalog:

SYSREL:
Tabellenbeschreibungen: Pers, . . .

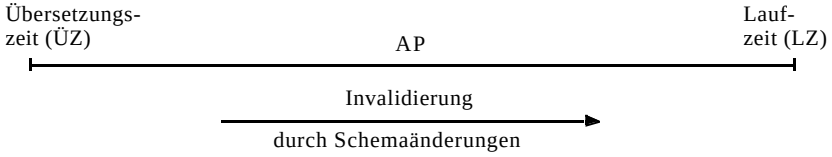
SYSATTR:
Attributbeschreibungen: Pnr, Name, Gehalt, . . .

SYSINDEX:
I_{Pers}(Beruf), . . .

SYSAUTH:
Nutzungsrechte

SYSINT/RULES:
Integritätsbedingungen, Zusicherungen, . . .

- **Zeitpunkt des Bindens**



Übersetzungskosten:
- unerheblich für Antwortzeit (AZ)

Interpretation:
- erheblich für AZ

Zugriffe (zur LZ):
- effizient
- datenabhängig!

Ausgleich gesucht!

Zugriffe (zur LZ):
- teuer
- datenunabhängig!

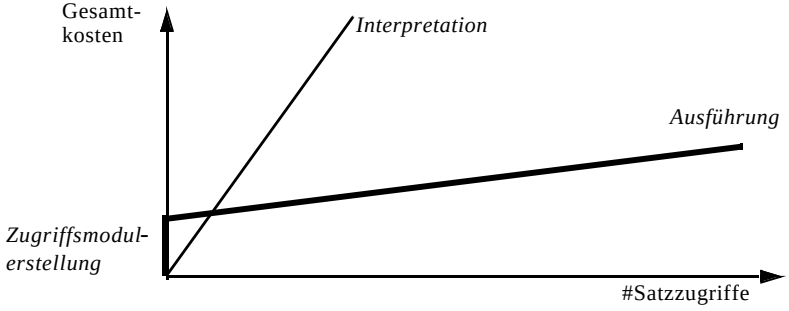
Auswertung von DB-Anweisungen (5)

- **Maximale Vorbereitung einer DB-Anweisung**
  - aufwendige Optimierung und Erstellung eines Zugriffsmoduls
  - maximale Auswirkungen von Schemaänderungen, welche die DB-Anweisung betreffen
  - Änderungen des DB-Zustandes nach der Übersetzung werden nicht berücksichtigt (neue Zugriffspfade, geänderte Statistiken etc.)

  å Invalidierung des Zugriffsmoduls und erneute Erstellung

- **Mischformen**
  - bestimmte Abhängigkeiten und Bindungen werden vermieden
  - jedoch: Invalidierungsmechanismus prinzipiell erforderlich

- **Interpretation einer DB-Anweisung**
  - Interpreter wertet Anweisung (als Zeichenfolge) zur Laufzeit aus
  - Aktueller DB-Zustand wird automatisch berücksichtigt
  - sehr hohe Ausführungskosten bei Programmschleifen sowie durch häufige Katalogzugriffe
  - interessant vor allem für Ad-hoc-Anfragen bzw. dynamisches SQL



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
  - 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 divison_by_zero

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 DBVS

    å **Dynamisches SQL** erlaubt Behandlung solcher Fälle

- **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

    å Bindung zur Übersetzungszeit muß dagegen Möglichkeit der Invalidierung/Neuübersetzung vorsehen

  - Vorbereitung und Ausführung einer SQL-Anweisung
    - erfolgt typischerweise durch Interpretation
    - Leistungsproblem: wiederholte Ausführung derselben Anweisung (DB2 UDB bewahrt Zugriffspläne zur Wiederverwendung im Cache auf)
    - Übersetzung und Code-Generierung ist jedoch prinzipiell möglich!

Dynamisches SQL (2)

- **Mehrere Sprachansätze**
  - Eingebettetes dynamisches SQL
  - Call-Level-Interface (CLI): kann ODBC-Schnittstelle[1] implementieren
  - Java Database Connectivity[2] (JDBC) ist eine dynamische SQL-Schnittstelle zur Verwendung mit Java
    - 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

    å 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

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 (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

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 DBVS verwaltet wird (kein SQLDA vorhanden)

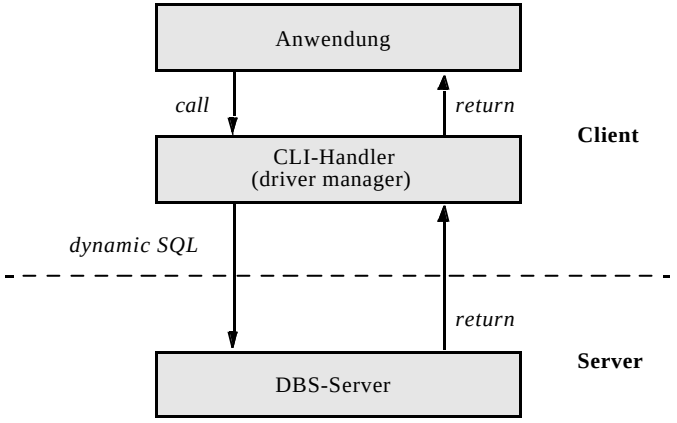
```
Anweisung = 'INSERT INTO Pers VALUES (?, ?, ...)';
exec sql prepare SQLanw from :Anweisung;
exec sql allocate descriptor 'Eingabeparameter';
exec sql describe input SQLanw into sql descriptor 'Eingabeparameter';
exec sql get descriptor 'Eingabeparameter' :n = count;

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

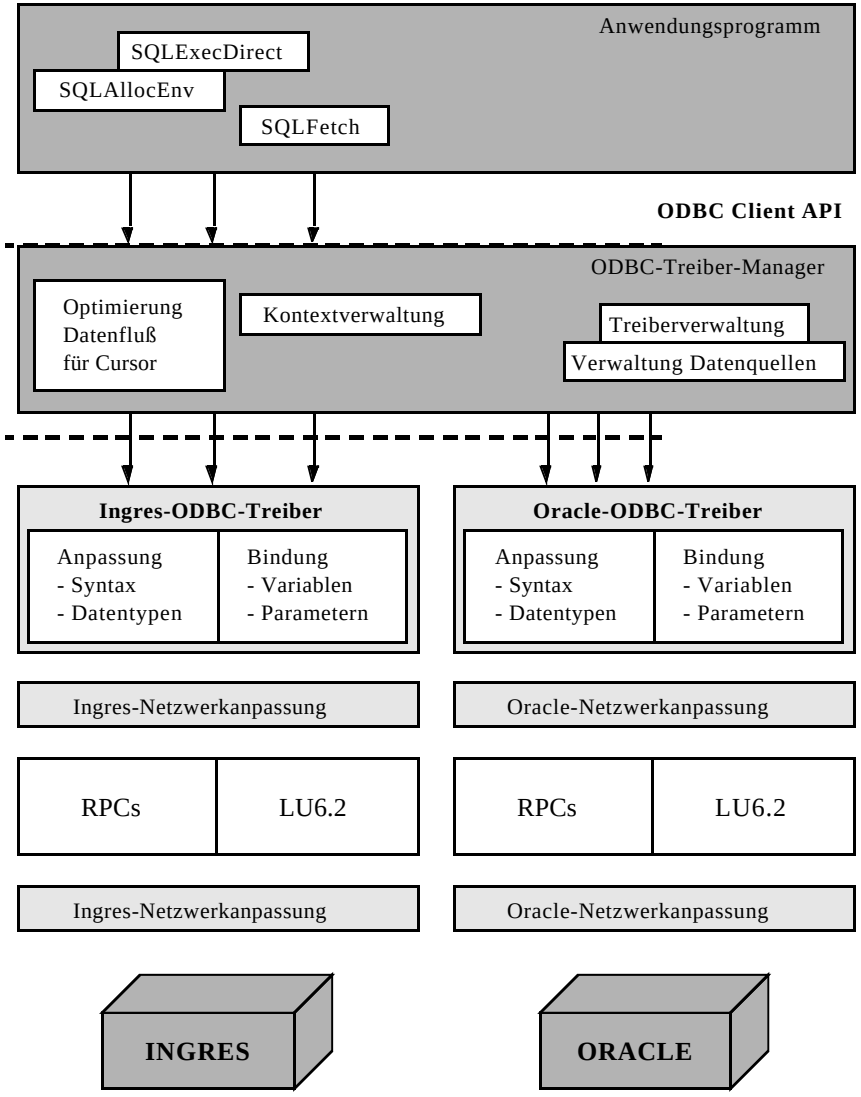
- **Spezielle Form von dynamischem SQL**
  - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
  - Direkte Aufrufe der Routinen einer standardisierten Bibliothek
  - Keine Vorübersetzung (Behandlung der DB-Anweisungen) von Anwendungen
    - Vorbereitung der DB-Anweisung geschieht erst beim Aufruf zur LZ
    - Anwendungen brauchen nicht im Source-Code bereitgestellt werden
    - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools

  å Schnittstelle wird sehr häufig im der Praxis eingesetzt!

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



Beispiel Microsoft
- ODBC-Architektur -

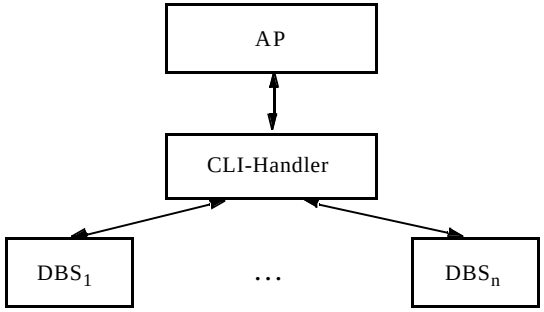


Call-Level-Interface (2)

- **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, die mehrere Fenster benutzen
  - Optimierung des Zugriffs vom/zum Server
    - Holen von mehreren Zeilen pro Zugriff
    - Lokale Bereitstellung einzelner Zeilen (Fetch)
    - Verschicken von zusammengesetzten SQL-Anweisungen
    - Client-Programme können Stored Procedures (PSM) aufrufen

Call-Level-Interface (3)

- **Wie kooperieren AP und DBS?**
  - maximale gegenseitige Kapselung
  - Zusammenspiel AP/CLI und DBVS ist nicht durch Übersetzungsphase vorbereitet
    - keine DECLARE SECTION
    - keine Übergabebereiche
  - Wahl des DBS zur Laufzeit
  - vielfältige LZ-Abstimmungen erforderlich



- **Konzept der Handle-Variablen wesentlich**
  - Handle (internes Kennzeichen) ist letztlich eine Programmvariable, die Informationen repräsentiert, die für ein AP durch die CLI-Implementierung verwaltet wird
  - gestattet Austausch von Verarbeitungsinformationen

Call-Level-Interface (4)

- **4 Arten von Handles**
  - **Umgebungskennung** repräsentiert den globalen Zustand der Applikation
  - **Verbindungskennung**
    - separate Kennung: n Verbindungen zu einem oder mehreren DBS
    - Freigabe/Rücksetzen von Transaktionen
    - Steuerung von Transaktionseigenschaften (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-Standardisierung in SQL3 wurde vorgezogen:**
  - 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

Standard-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) ;
}
```


Zusammenfassung

- **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
- **Interpretation einer DB-Anweisung**
  - allgemeines Programm (Interpreter) akzeptiert Anweisungen der DB-Sprache als Eingabe und erzeugt mit Hilfe von Aufrufen des Zugriffssystems Ergebnis
  - hoher Aufwand zur Laufzeit (v.a. bei wiederholter Ausführung einer Anweisung)
- **Übersetzung, Code-Erzeugung und Ausführung einer DB-Anweisung**
  - für jede DB-Anweisung wird ein zugeschnittenes Programm erzeugt (Übersetzungszeit), das zur Laufzeit abgewickelt wird und dabei mit Hilfe von Aufrufen des Zugriffssystems das Ergebnis ableitet
  - Übersetzungsaufwand wird zur Laufzeit soweit wie möglich vermieden
- **PSM**
  - zielt ab auf Leistungsverbesserung vor allem in Client/Server-Umgebung
  - erhöht die Verarbeitungsmächtigkeit des DBS

Zusammenfassung (2)

- **Dynamisches SQL**
  - Festlegung/Übergabe von SQL-Anweisungen zur Laufzeit
  - hohe Flexibilität, schwierige Programmierung
- **Unterschiede in der SQL-Programmierung zu eingebettetem SQL**
  - explizite Anweisungen zur Datenabbildung zwischen DBS und Programmvariablen
  - klare Trennung zwischen Anwendungsprogramm und SQL (→ einfacheres Debugging)
- **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 die 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

Überwindung der Heterogenität mit ODBC (Open Data Base Connectivity)

- **ODBC ist**
  - eine durch die Firma Microsoft definierte und von ihr favorisierte Architektur, die aus funktionaler Sicht Heterogenität (einigermaßen) überwindet,
  - jedoch z.T. erhebliche Leistungseinbußen gegenüber einer DBS-Hersteller-spezifischen Anbindung verzeichnet.
- **ODBC umfaßt u.a.**
  - eine Programmierschnittstelle vom CLI-Typ und
  - eine Definition des unterstützten SQL-Sprachumfangs (im Vergleich zu ISO SQL2).
- **DBS-Anbieter**
  - implementieren sogenannte ODBC-Treiber (Umsetzung von Funktionen und Daten auf herstellerspezifische Schnittstellen),
  - die gemäß den ebenfalls in der ODBC-Architektur definierten Windowsinternen Schnittstellen in die Windows-Betriebssysteme integriert werden können.
- **ODBC**
  - wird von praktisch allen relevanten DBS-Herstellern unterstützt und
  - stellt einen **herstellerspezifischen De-facto-Standard** dar,
  - der für die **unterschiedlichen Windows-Betriebssysteme** auf der Anwendungsseite Gültigkeit hat.

Beispiel Microsoft
- Open Data Base Connectivity (ODBC) -

```
RETCODE retcode;                                    /* Return Code */
HENV henv; HDBC hdbc;                  /* Environment und Connection Handle */
HSTMT hstmt;                                        /* Statement Handle */

UCHAR szName[33], szAbtName[33]; long IBonus;
SDWORD cbName, cbAbtName, cbBonus;

retcode = SQLALLocEnv (&henv);                  /* Anlegen Anwendungskontext */
retcode = SQLAllocConnect (henv, & hdbc);       /* Anlegen Verbindungskontext */
retcode = SQLAllocStmt (hdbc, & hstmt);         /* Anlegen Anweisungskontext */

retcode = SQLConnect    (hdbc, "DEMO-DB", SQL_NTS, "PePe", SQL-NTS,
                         "GEHEIM", SQL_NTS);/* Verbindung aufbauen */
retcode = SQLSetConnect Option (hdbc, SQL_ACCESS_MODE,
                                SQL_MODE_READ_ONLY;     /* Eigenschaften */

retcode = SQLExecDirect (hstmt, "UPDATE Mitarbeiter SET Bonus =
                          0.2 * Gehalt", SQL_NTS);          /* Ausführen */
retcode = SQLExecDirect (hstmt, "   SELECT M.Name, M.Bonus, A.Abtname
                                    FROM Mitarbeiter M, Abteilung A
                                    WHERE A.AbtNr = M.AbtNr", SQL-NTS);

retcode = SQLBindCol    (hstmt, 1, SQL_C_DEFAULT, szName, 33, &cbName);
retcode = SQLBindCol    (hstmt, 2, SQL_C_DEFAULT, szAbtName, 33,
                         &cbAbtName);                   /* Variablen binden */
retcode = SQLBindCol    (hstmt, 3, SQL_C_DEFAULT, szBonus, sizeof(long),
                         &cbBonus);
retcode = SQLFetch      (hstmt);                        /* Zeile anfordern */

retcode = SQLTransact (henv, hdbc , SQL_COMMIT);

/*  Freigabe der dynamisch angeforderten Kontexte */
retcode = SQLFreeStmt (hstmt); retcode = SQLDisconnect (hdbc);
retcode = SQLFreeConnect (hdbc); retcode = SQLFreeEnv (henv );
```


DB-Zugriff via JDBC

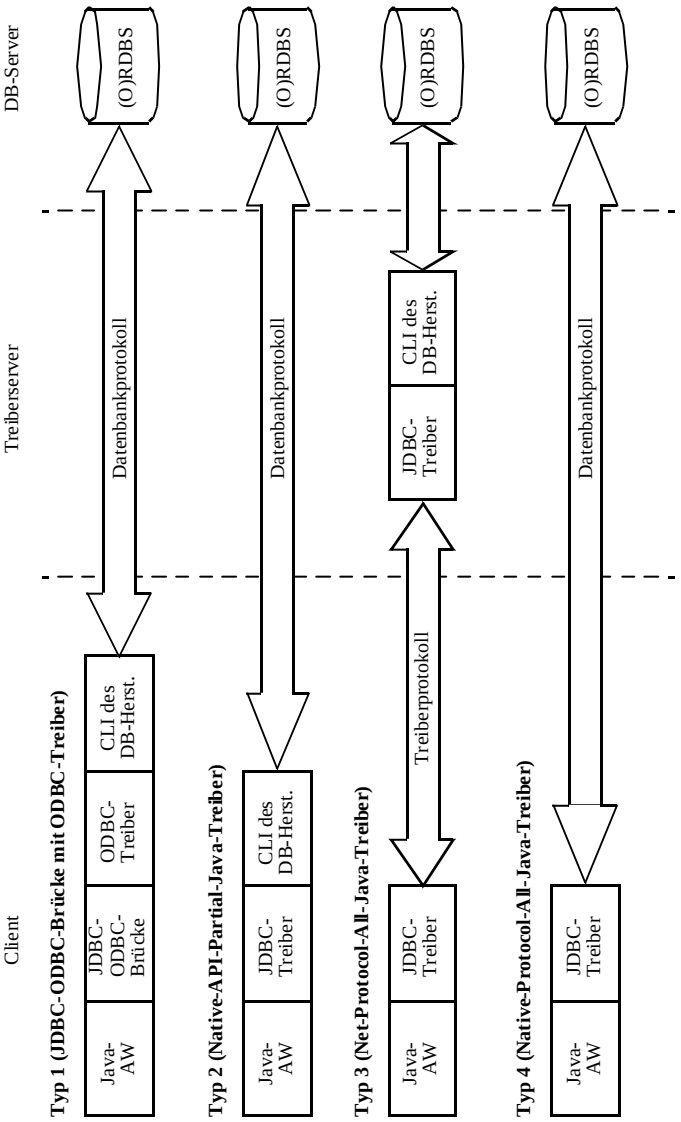
- **Java Database Connectivity Data Access API (JDBC)**[1]
  - unabhängiges, standardisiertes CLI, basierend auf SQL:1999
  - bietet Schnittstelle für den Zugriff auf (objekt-) relationale DBS aus Java-Anwendungen
  - besteht aus zwei Teilen
    - Core Package: Standardfunktionalität mit Erweiterungen (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 DBS-spezifischen Aufrufe um
  - Treiber werden z.B. vom DBS-Hersteller zur Verfügung gestellt
  - Treiber-Unterstützung kann auf vier verschiedene Arten erfolgen

1. Aktueller Standard: JDBC API 3.0 Specification Proposed Final Draft 4
   http://java.sun.com/j2ee/

DB-Zugriff via JDBC (2)
- JDBC-Treiber-Varianten



DB-Zugriff via JDBC (3)
- Arten der Treiberunterstützung[1]

- **Typ 1**
  - **Notlösung**, wenn kein JDBC-Treiber eines anderen Typs vorhanden ist
  - ODBC ist weit verbreitet und wird von fast allen DBS unterstützt
  - JDBC-ODBC-Brücke setzt JDBC-Anweisungen in ODBC-Anweisungen um
  - Herstellerspezifischer Code (CLI) und ODBC-Treiber müssen auf jedem Client-Rechner installiert und gewartet werden
  - Indirektion über ODBC bedeutet Zusatzaufwand
- **Typ 2**
  - direktere Anbindung als bei Typ 1, jedoch
  - herstellerspezifischer Code auf jedem Client-Rechner
- **Typ 3**
  - JDBC-Anweisungen werden zunächst in ein **DBS-unabhängiges** Netzwerkprotokoll übersetzt (net-protocol fully Java technology-enabled driver)
  - Middleware-Server übersetzt in die CLI-Aufrufe des jeweiligen DBS
  - flexible Lösung mit erheblich vereinfachter Wartung
- **Typ 4**
  - **DBS-spezifischer** Treiber übersetzt JDBC-Anweisungen direkt in das Netzwerkprotokoll des jeweiligen DBS (native-protocol fully Java technology-enabled driver)
  - Verwendung des aktuellen DBS muß bekannt sein (Intranet)
  - ist wie auch Typ 3 Applet-fähig

1. Alle großen DBS-Hersteller, aber auch Drittfirmen, bieten JDBC-Treiber an. SUN-Datenbank enthält etwa 150 Treiber, wobei Typ 4 momentan am weitesten verbreitet ist.

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 spezifiziertes 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

- **ececuteUpdate-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);

// Cursor-Zugriff und Konvertierung der DBS-Datentypen in
passende Java-Datentpyen erforderlich (siehe Cursor-Behandlung)
```

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
```

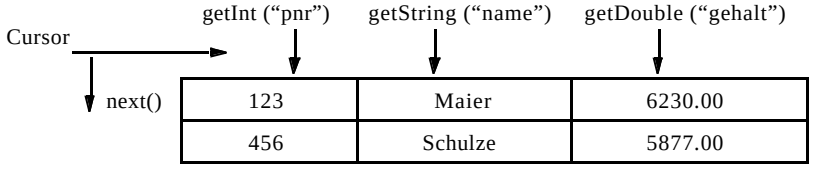

JDBC - Ergebnismengen und Cursor

- **Select-Anfragen und Ergebnisübergabe**

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

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

  - Cursor-Zugriff und Konvertierung der DBS-Datentypen in passende Java-Datentypen erforderlich
  - JDBC-Cursor ist durch die Methode next() der Klasse ResultSet implementiert



  - 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") );
```

- JDBC definiert drei Typen von ResultSets

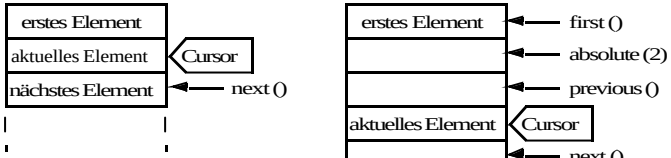
- **ResultSet: forward-only**

  Default-Cursor vom Typ INSENSITIVE: nur next()

JDBC - Ergebnismengen und Cursor (2)

- **ResultSet: scroll-insensitive**

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



- **ResultSet: scroll-sensitve**
  - 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-Semantik [1] (Teil des ODBC-Standards) implementiert.

- **Aktualisierbare ResultSets**

```
Statement s1 = con1.createStatement (ResultSet.TYPE_SCROLL_
                    SENSITIVE, 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.

1. 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.getColumnTypeName (1);
```

  - Ein Wertzugriff kann dann erfolgen durch

    rs1.getInt (2), wenn

    rsm1.getColumnTypeName (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);
```

  - Für eine Transaktion können sogen. Konsistenzebenen (isolation levels) wie TRANSACTION_SERIALIZABLE, TRANSACTION_REPEATABLE_READ usw. eingestellt werden

```
con1.setTransactionIsolation (
                        Connection.TRANSACTION_SERIALIZABLE);
```

- **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

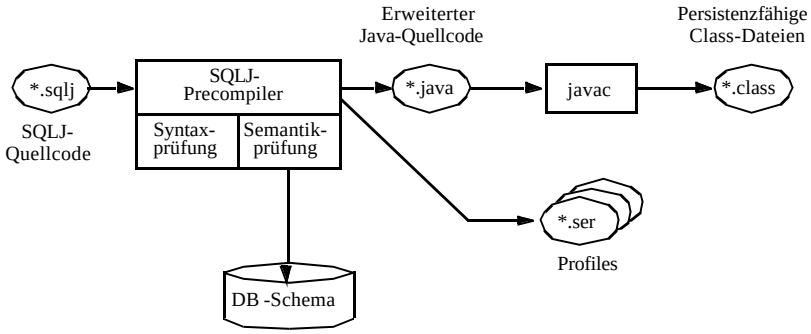
```
import java.sql.*;
public class Select   {
   public static void main (String [ ] args)   {
      Connection con = null;
      PreparedStatement pstmt;
      ResultSet res;
      double gehalt = 5000.00;
      try   {
        Class.forName (“sun.jdbc.odbc.JdbcOdbcDriver”);
        con = java.sql.DriverManager.getConnection (
                “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 ();
```


SQLJ

- **SQLJ**
  - Teil 0 der SQLJ-Spezifikation beschreibt die Einbettung von SQL in Java-Anwendungen (bereits ANSI-Standard)
  - besitzt bessere Lesbarkeit, Verständlichkeit und Wartbarkeit durch kurze und prägnante Schreibweise
  - zielt auf die Laufzeiteffizienz von eingebettetem SQL ab, ohne die Vorteile des DB-Zugriffs via JDBC aufzugeben
- **Abbildung auf JDBC durch Precompiler**
  - Überprüfung der Syntax sowie (gewisser Aspekte) der Semantik von SQL-Anweisungen (Anzahl und Typen von Argumenten usw.) zur Übersetzungszeit, was Kommunikation mit dem DBMS zur Laufzeit erspart
  - Ersetzung der SQLJ-Anweisungen durch Aufrufe an das SQLJ-Laufzeitmodul (Pakkage sqlj.runtime.*)
  - Erzeugung sog. Profiles, serialisierbare Java-Klassen, die die eigentlichen JDBC-Anweisungen enthalten



  - Abwicklung von DB-Anweisungen vom SQLJ-Laufzeitmodul dynamisch über die Profiles, die wiederum über einen JDBC-Treiber auf die DB zugreifen
  - Anpassung an ein anderes DBMS geschieht durch Austausch der Profiles (sog. Customizing)

SQLJ (2)

- Er werden nur einige **Unterschiede zu eingebettetem SQL und JDBC** aufgezeigt

- **Verbindung zum DBMS**

  - erfolgt über sog. Verbindungskontexte (ConnectionContext)
  - Sie basieren auf JDBC-Verbindungen und werden auch so genutzt (URL, Nutzername, Paßwort)
  - SQLJ-Programm kann **mehrere Verbindungskontexte** über verschiedene JDBC-Treiber aufbauen; sie erlauben den parallelen Zugriff auf mehrere DBMS oder aus mehreren Threads/Prozessen auf das gleiche DBMS

- **SQL-Anweisungen** sind im Java-Programm Teil einer SQLJ-Klausel

```
#SQL {  select p.pnr into :persnr
        from personal p
        where p.beruf     = :beruf
          and p.gehalt    > :gehalt} ;
```

  - Austausch von Daten zwischen SQLJ und Java-Programm erfolgt über Wirtssprachenvariablen
  - Parameterübergabe kann vorbereitet werden
  - ist viel effizienter als bei JDBC (mit ?-Platzhaltern)

- **Iteratoren**

  - analog zu JDBC-ResultSets
  - Definition von Iteratoren (Cursor), aus denen entsprechende Java-Klassen generiert werden, über die auf die Ergebnismenge zugegriffen wird

- **SQLJ und JDBC**

  Ebenso wie statische und dynamische SQL-Anweisungen in einem Programm benutzt werden können, können SQLJ-Anweisungen und JDBC-Aufrufe im selben Java-Programm auftreten.

SQLJ (3)

- **Nutzung eines Iterators in SQLJ**

```
import java.sql.*
  . . .
  #SQL iterator GetPersIter (int personalnr, String nachname);
  Get PersIter iter1;
  #SQL iter1 = {  select   p.pnr as “personalnr”,
                           p.name as “nachname”
                  from     personal p
                  where    p.beruf    = :Beruf
                    and    p.gehalt   = :Gehalt} ;
int Id ;
String Name ;
  while     (iter1.next ())  {
            Id = iter1. personalnr () ;
            Name = iter1.nachname () ;
            . . . Verarbeitung . . .
  }
  iter1.close () ;
```

  Die as-Klausel wird benutzt, um die SQL-Attributnamen im Ergebnis mit den Spaltennamen im Iterator in Beziehung zu setzen

- SQLJ liefert für eine Anfrage ein **SQLJ-Iterator-Objekt** zurück

  - SQLJ-Precompiler generiert Java-Anweisungen, die eine Klasse GetPersIter[1] definieren
  - Klasse GetPersIter kann als Ergebnisse Zeilen mit zwei Spalten (Integer und String) aufnehmen
  - Deklaration gibt den Spalten Java-Namen (personalnr und nachname) und definiert implizit Zugriffsmethoden personalnr () und nachname (), die zum Iterator-Zugriff benutzt werden

1. Sie implementiert das Interface sqlj.runtime.NamedIterator (spezialisiert vom Standard-Java-Interface java.util.Iterator)

SQLJ (4)

- **DB-seitige Nutzung von Java mit SQLJ[1]**
  - Teil 1 des SQLJ-Standards beschreibt, wie man Stored Procedures mit Java realisieren kann
  - Sprache für Stored Procedures bisher nicht standardisiert, Wildwuchs von Implementierungen
  - erste Sprache für portable Stored Procedures
  - automatisiertes Installieren/Entfernen von Stored Procedures in/aus DBMS (Einsatz sog. Deployment Descriptors)

- **DB-seitige Verwendung von Java-Datentypen**
  - Teil 2 des SQLJ-Standards beschreibt Verfahren, wie Java-Datentypen als SQL-Datentypen verwendet werden können
  - Umgekehrt können für herkömmliche SQL-Typen Wrapper-Klassen automatisch generiert werden

- **SQLJ-Standard**
  - Teil1 und 2 sind noch nicht verabschiedet
  - Es existieren aber bereits nicht-standardkonforme Implementierungen

1. Man spricht auch von Server-sided Java, von der Marketing-Abteilung von Sun auch als “300% Java” bezeichnet, also jeweils 100% für die 3 Schichten einer Standard-C/S-Architektur