3. Anwendungsprogrammier-Schnittstelle

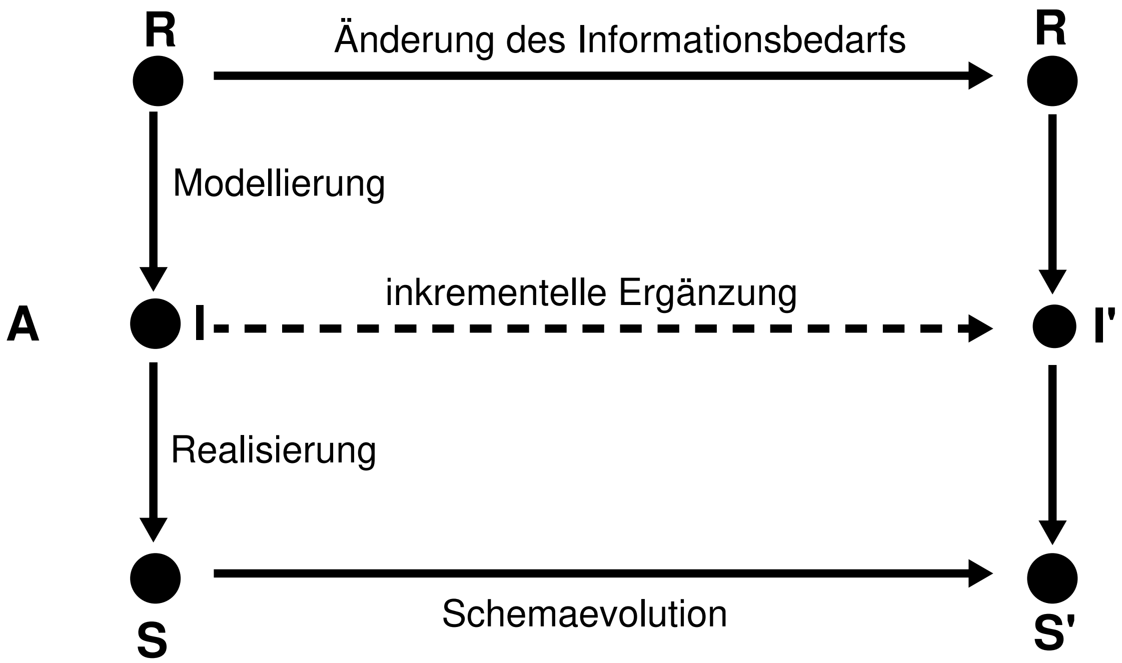
- **Weitere Möglichkeiten der Datendefinition (DDL)[1]**
  - Schemaevolution
  - Indexierung
- **Sichtkonzept**
  - Semantik von Sichten
  - Aktualisierung von Sichten
- **Kopplung mit einer Wirtssprache:**
  Übersicht und Aufgaben
- **Eingebettetes statisches SQL**
  - Cursor-Konzept
  - SQL-Programmiermodell
  - Rekursion
  - Erweiterung des Cursor-Konzeptes
  - Ausnahme- und Fehlerbehandlung
- **Aspekte der Anfrageübersetzung**
- **Dynamisches SQL**
  - Eingebettetes dynamisches SQL
  - Call-Level-Interface
  - DB-Zugriff via JDBC
- **PSM**
- **Anhang:**
  Open Data Base Connectivity (ODBC)

---

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

Evolution einer Miniwelt

- **Grobe Zusammenhänge**



R: Realitätsausschnitt (Miniwelt)

I: Informationsmodell
(zur Analyse und Dokumentation der Miniwelt)

S: DB-Schema der Miniwelt
(Beschreibung aller Objekt- und Beziehungstypen sowie aller Integritäts- und Zugriffskontrollbedingungen)

A: Abbildung aller wichtigen Objekte und Beziehungen sowie ihrer Integritäts- und Datenschutzaspekte
➥ Abstraktionsvorgang

- **Schemaevolution:**
  - Änderung, Ergänzung oder Neudefinition von Typen und Regeln
  - nicht alle Übergänge von S nach S' können automatisiert durch das DBS erfolgen

  ➥ gespeicherte Objekt- und Beziehungsmengen dürfen den geänderten oder neu spezifizierten Typen und Regeln nicht widersprechen

Schemaevolution

- **Wachsender oder sich ändernder Informationsbedarf**
  - Erzeugen/Löschen von Tabellen (und Sichten)
  - Hinzufügen, Ändern und Löschen von Spalten
  - Anlegen/Ändern von referentiellen Beziehungen
  - Hinzufügen, Modifikation, Wegfall von Integritätsbedingungen

➥ Hoher Grad an logischer Datenunabhängigkeit ist sehr wichtig!

- **Zusätzliche Änderungen im DB-Schema**
  durch veränderte Anforderungen bei der DB-Nutzung
  - Dynamisches Anlegen von Zugriffspfaden
  - Aktualisierung der Zugriffskontrollbedingungen

- **Dynamische Änderung einer Tabelle**
  Bei Tabellen können dynamisch (während ihrer Lebenszeit) Schemaänderungen durchgeführt werden

```
ALTER  TABLE  base-table
   {  ADD [COLUMN] column-def
    | ALTER [COLUMN] column
               {SET default-def  | DROP DEFAULT}
    | DROP [COLUMN] column {RESTRICT | CASCADE}
    | ADD  base-table-constraint-def
    | DROP  CONSTRAINT constraint {RESTRICT | CASCADE}}
```

➥ Welche Probleme ergeben sich?

Schemaevolution (2)

E1: Erweiterung der Tabellen Abt und Pers durch neue Spalten

```
ALTER  TABLE   Pers
       ADD   Svnr INT UNIQUE

ALTER  TABLE   Abt
       ADD   Geh-Summe INT
```

Abt	Anr	Aname	Ort
	K51	PLANUNG	KAISERSLAUTERN
	K53	EINKAUF	FRANKFURT
	K55	VERTRIEB	FRANKFURT

Pers	Pnr	Name	Alter	Gehalt	Anr	Mnr
	406	COY	47	50 700	K55	123
	123	MÜLLER	32	43 500	K51	-
	829	SCHMID	36	45 200	K53	777
	574	ABEL	28	36 000	K55	123

E2: Verkürzung der Tabelle Pers um eine Spalte

```
ALTER TABLE Pers

       DROP COLUMN Alter RESTRICT
```

- Wenn die Spalte die einzige der Tabelle ist, wird die Operation zurückgewiesen.
- Da RESTRICT spezifiziert ist, wird die Operation zurückgewiesen, wenn die Spalte in einer Sicht oder einer Integritätsbedingung (Check) referenziert wird.
- CASCADE dagegen erzwingt die Folgelöschung aller Sichten und Check-Klauseln, die von der Spalte abhängen.

Schemaevolution (3)

- **Löschen von Objekten**

```
DROP          {TABLE base-table |  VIEW view |
               DOMAIN domain | SCHEMA schema }
              {RESTRICT | CASCADE}
```

- Falls Objekte (Tabellen, Sichten, ...) nicht mehr benötigt werden, können sie durch die DROP-Anweisung aus dem System entfernt werden.
- Mit der CASCADE-Option können ’abhängige’ Objekte (z. B. Sichten auf Tabellen oder anderen Sichten) mitentfernt werden
- RESTRICT verhindert Löschen, wenn die zu löschende Tabelle noch durch Sichten oder Integritätsbedingungen referenziert wird

E3: Löschen von Tabelle Pers

DROP TABLE Pers RESTRICT

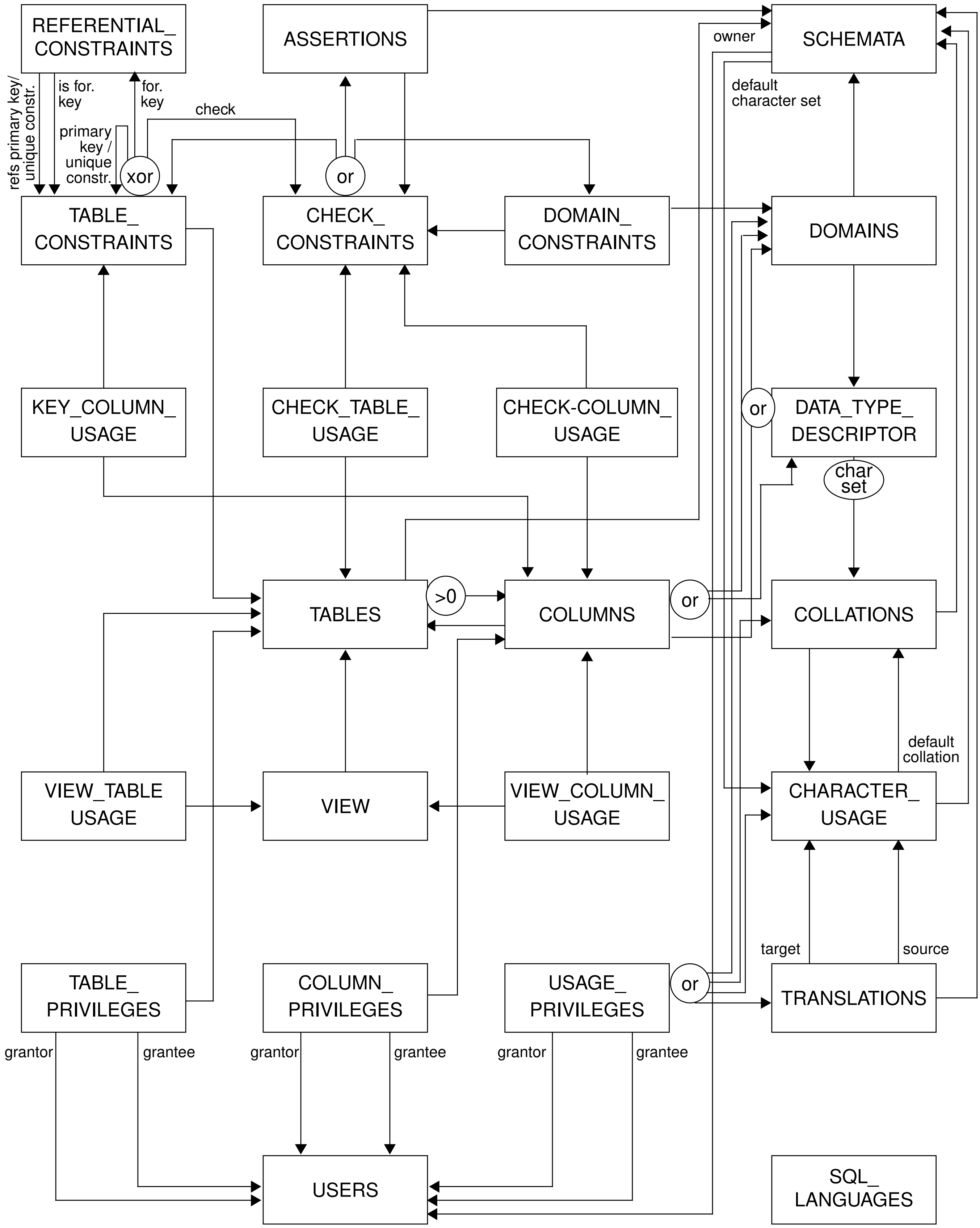
PersConstraint sei definiert auf Pers:

1. ALTER TABLE Pers
   DROP CONSTRAINT PersConstraint CASCADE

2. DROP TABLE Pers RESTRICT

- **Durchführung der Schemaevolution**
  - Aktualisierung von Tabellenzeilen des SQL-Definitionsschemas
  - “tabellengetriebene” Verarbeitung der Metadaten durch das DBS

SQL-Definitionsschema



Indexierung

- **Einsatz von Indexstrukturen**
  - Beschleunigung der Suche: Zugriff über Spalten (Schlüsselattribute)
  - Kontrolle von Integritätsbedingungen (relationale Invarianten)
  - Zeilenzugriff in der logischen Ordnung der Schlüsselwerte
  - Gewährleistung der Clustereigenschaft für Tabellen

    ➥ **aber:** erhöhter Aktualisierungsaufwand und Speicherplatzbedarf

- **Einrichtung von Indexstrukturen**
  - Datenunabhängigkeit des Relationenmodells erlaubt ein Hinzufügen und Löschen
  - jederzeit möglich, um z. B. bei veränderten Benutzerprofilen das Leistungsverhalten zu optimieren
  - "beliebig" viele Indexstrukturen pro Tabelle und mit unterschiedlichen Spaltenkombinationen als Schlüssel möglich
  - Steuerung der Eindeutigkeit der Schlüsselwerte, der Clusterbildung
  - Freiplatzanteil (PCTFREE) in jeder Indexseite beim Anlegen erleichtert das Wachstum

    ➥ **Spezifikation:** DBA oder Benutzer

- **Nutzung eines vorhandenen Index**

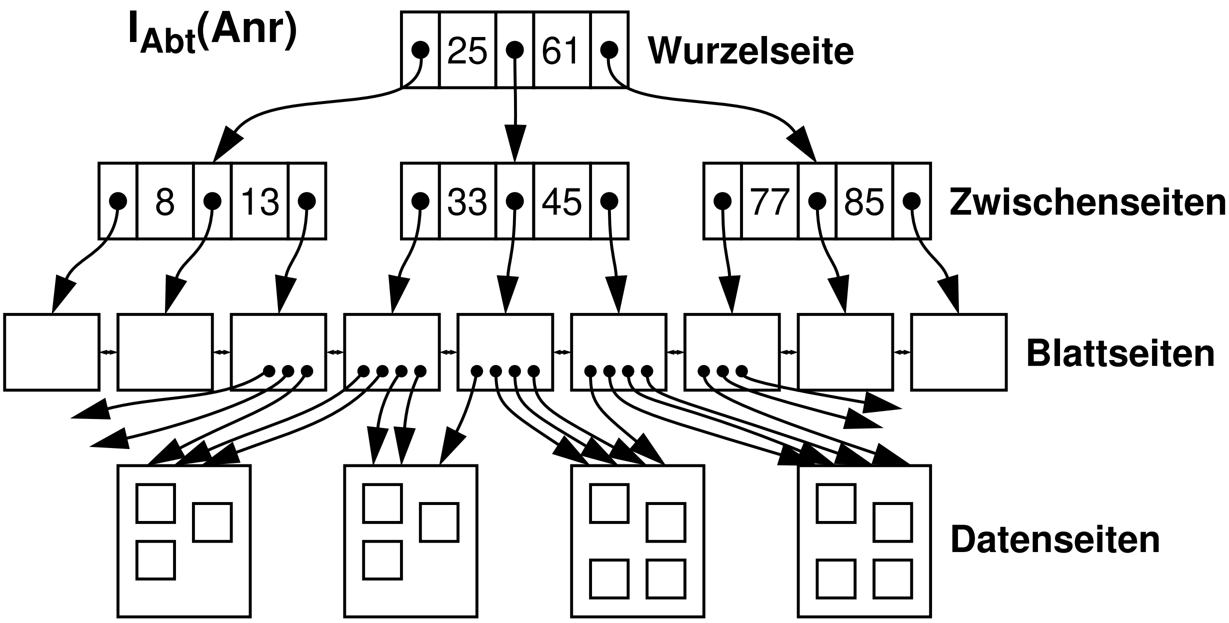
    ➥ Entscheidung durch DBS-Optimizer

- Im SQL-Standard keine Anweisung vorgesehen, jedoch in realen Systemen (z. B. IBM DB2):

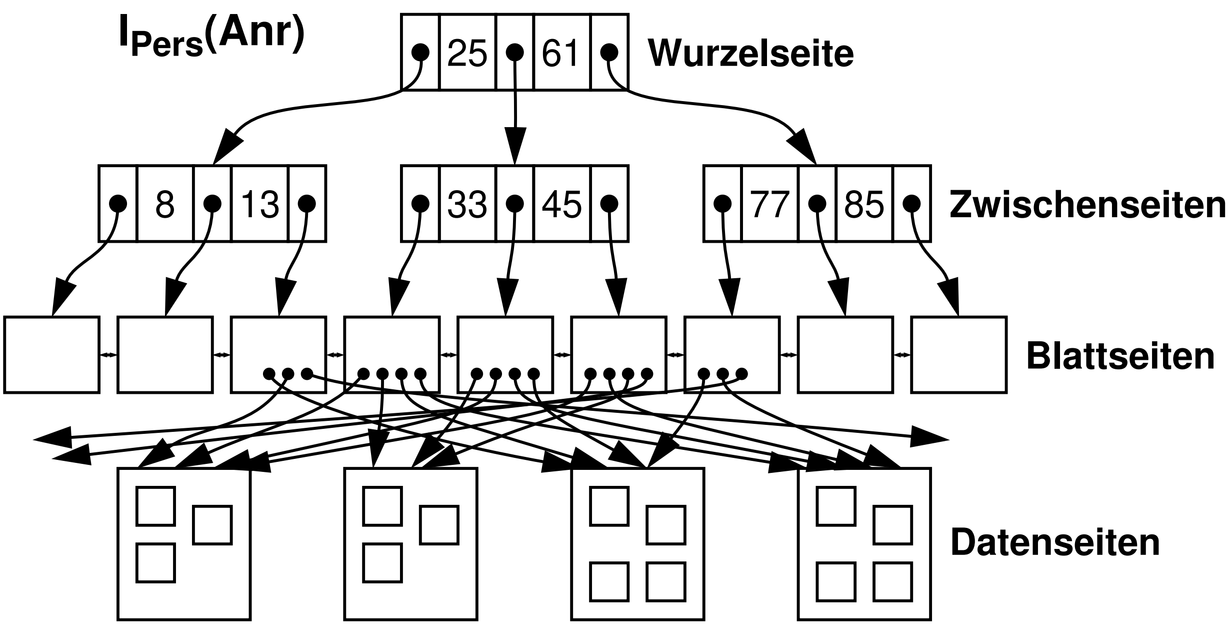
```
CREATE    [UNIQUE]  INDEX  index
              ON base-table (column [ORDER] [,column[ORDER]] ...)
              [CLUSTER] [PCTFREE]
```

Indexierung (2)

- **Index mit Clusterbildung**



- **Index ohne Clusterbildung**



Indexierung (3)

E4: Erzeugung einer Indexstruktur mit Clusterbildung auf der Spalte Anr von Abt

```
CREATE   UNIQUE   INDEX   Persind1
                  ON  Abt (Anr) CLUSTER
```

- Realisierung z. B. durch B*-Baum
  (oder Hashing mit verminderter Funktionalität)
- **UNIQUE:** keine Schlüsselduplikate im Index
- **CLUSTER:** zeitoptimale sortiert-sequentielle Verarbeitung
  (Scan-Operation)

E5: Erzeugung einer Indexstruktur auf den Spalten Anr (absteigend) und Gehalt (aufsteigend) von Pers.

```
CREATE   INDEX   Persind2
                  ON   Pers (Anr DESC,  Gehalt ASC)
```

- **Typische Implementierung eines Index: B*-Baum**
  (wird von allen DBS angeboten!)

  ➥ dynamische Reorganisation durch Aufteilen (Split) und Mischen von Seiten

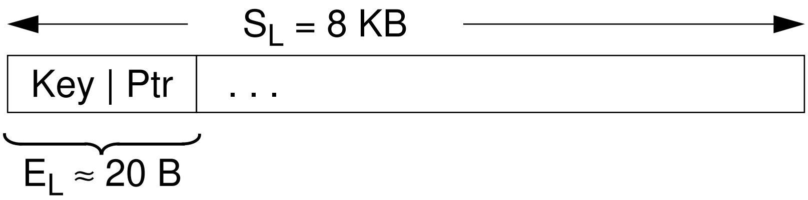
- **Wesentliche Funktionen**
  - direkter Schlüsselzugriff auf einen indexierten Satz
  - sortiert sequentieller Zugriff auf alle Sätze
    (unterstützt Bereichsanfragen, Verbundoperation usw.)

- **Balancierte Struktur**
  - unabhängig von Schlüsselmenge
  - unabhängig von Einfügereihenfolge

Indexierung (4)

- **Vereinfachtes Zahlenbeispiel zum B*-Baum**

Seitenformat
im B*-Baum



$$ES = \frac{S_L}{E_L} \quad = \text{max. \# Einträge/Seite}$$

h_B = Baumhöhe

N_T = #Zeilenverweise im B*-Baum

N_B = #Blattseiten im B*-Baum

$$N_{Tmin} = 2 \cdot \left(\frac{ES}{2}\right)^{h_B - 1} \leq N_T \leq ES^{h_B} = N_{Tmax}$$

➥ Welche Werte ergeben sich für $h_B = 3$ und $E_L = 20$ B?

Sichtkonzept

- **Ziel: Festlegung**
  - welche Daten Benutzer sehen wollen
    (Vereinfachung, leichtere Benutzung)
  - welche Daten sie nicht sehen dürfen (Datenschutz)
  - einer zusätzlichen Abbildung
    (erhöhte Datenunabhängigkeit)
- **Sicht (*View*):** mit Namen bezeichnete, aus Tabellen abgeleitete, virtuelle Tabelle (Anfrage)
- **Korrespondenz zum externen Schema** bei ANSI/SPARC
  (Benutzer sieht jedoch i. allg. mehrere Sichten (Views) und Tabellen)

```
CREATE VIEW view [ (column-commalist ) ]
      AS table-exp
           [WITH [ CASCADED | LOCAL] CHECK OPTION]
```

D5: Sicht, die alle Programmierer mit einem Gehalt < 30 TDM umfaßt.

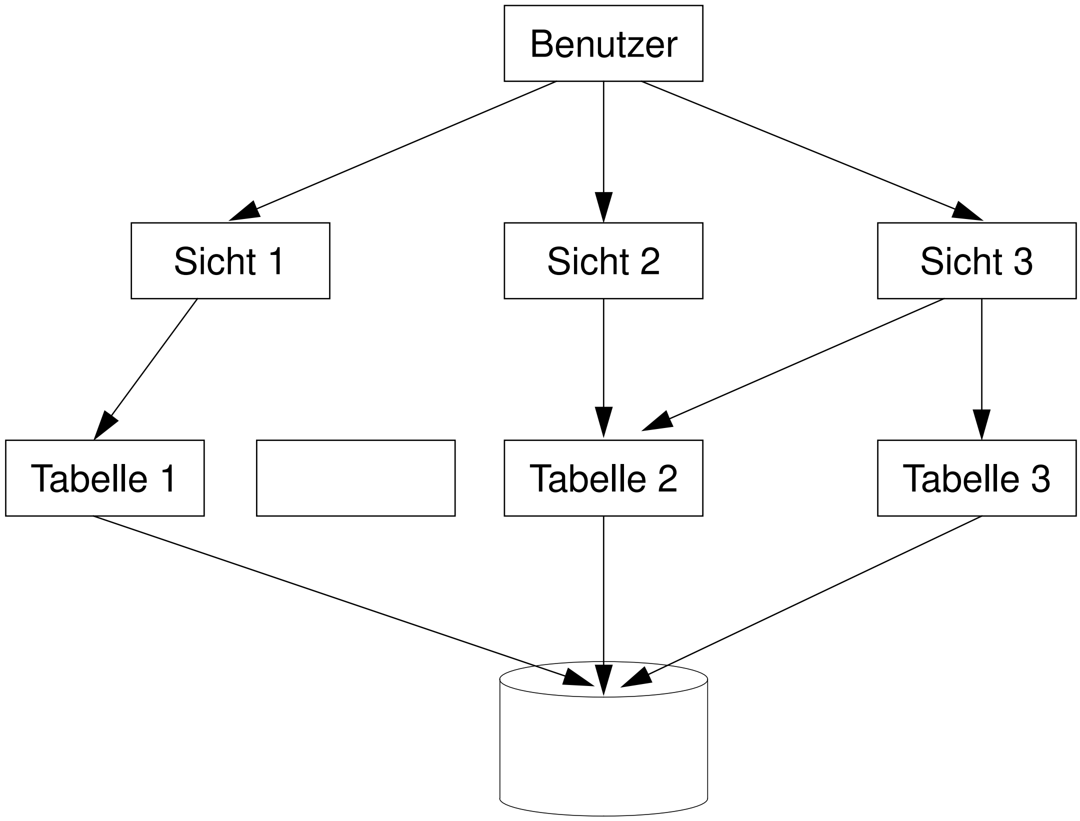
```
CREATE VIEW
       Arme_Programmierer (Pnr, Name, Beruf, Gehalt, Anr)
       AS     SELECT Pnr, Name, Beruf, Gehalt, Anr
              FROM Pers
              WHERE Beruf = 'Programmierer' AND Gehalt <  30 000
```

D6: Sicht für den Datenschutz

```
CREATE VIEW  Statistik (Beruf, Gehalt)
       AS     SELECT Beruf, Gehalt
              FROM Pers
```

Sichtkonzept (2)

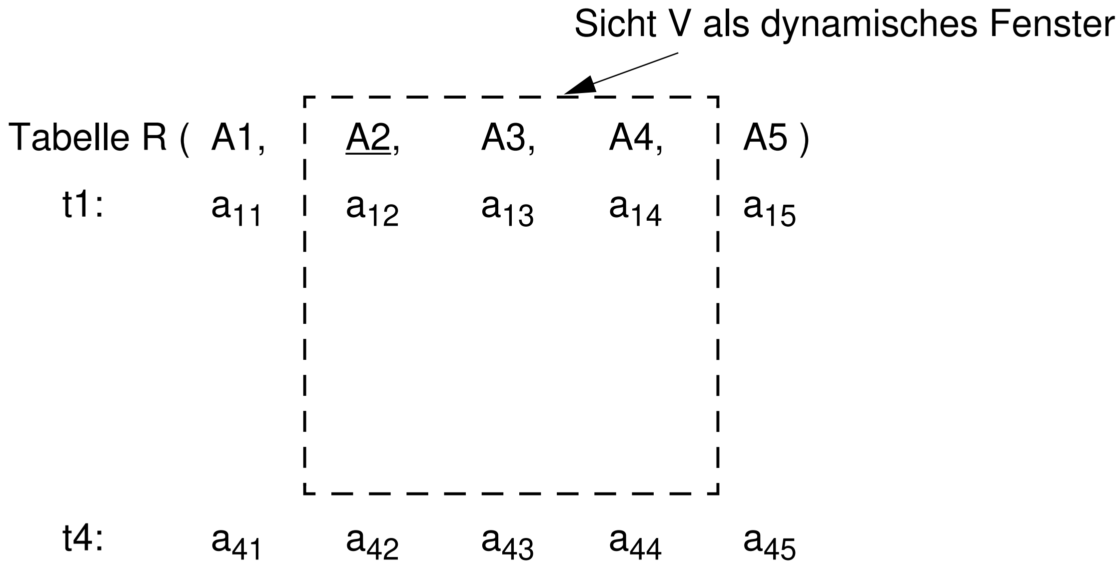
- **Sichten zur Gewährleistung von Datenunabhängigkeit**



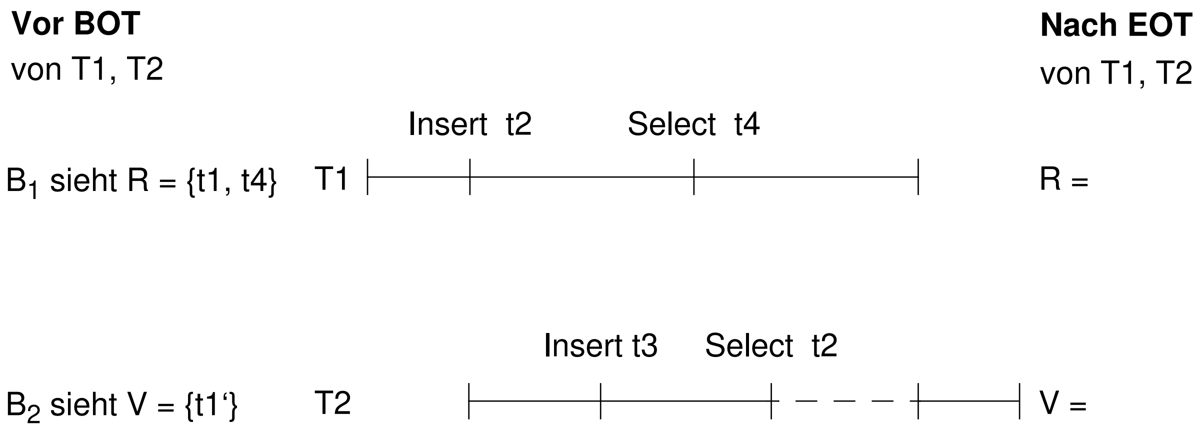
- **Eigenschaften von Sichten**
  - Sicht kann wie eine Tabelle behandelt werden
  - Sichtsemantik:
    „dynamisches Fenster“ auf zugrundeliegende Tabellen
  - Sichten auf Sichten sind möglich
  - eingeschränkte Änderungen:
    aktualisierbare und nicht-aktualisierbare Sichten

Sichtkonzept (3)

- **Zum Aspekt: Semantik von Sichten**



- **Sichtbarkeit von Änderungen - Wann und Was?**
  Wann werden welche geänderten Daten in der **Tabelle**/**Sicht** für die **anderen** Benutzer sichtbar?



Sichtkonzept (4)

- **Abbildung von Sicht-Operationen auf Tabellen**
  - Sichten werden i. allg. nicht explizit und permanent gespeichert, sondern Sicht-Operationen werden in äquivalente Operationen auf Tabellen umgesetzt
  - Umsetzung ist für Leseoperationen meist unproblematisch

Anfrage (Sichtreferenz):

```
SELECT Name, Gehalt
FROM Arme_Programmierer
WHERE Anr = 'K55'
```

Ersetzung durch:

```
SELECT Name, Gehalt
FROM
WHERE Anr = 'K55'
```

- **Abbildungsprozeß auch über mehrere Stufen durchführbar**

Sichtendefinitionen:

```
CREATE VIEW V  AS SELECT  ... FROM R WHERE P
CREATE VIEW W  AS SELECT  ... FROM V WHERE Q
```

Anfrage:

```
SELECT ... FROM W WHERE  C
```

Ersetzung durch

```
SELECT ... FROM V WHERE Q AND C
```

und

```
SELECT ... FROM R WHERE Q AND P AND C
```

Sichtkonzept (5)

- **Einschränkungen der Abbildungsmächtigkeit**
  - keine Schachtelung von Aggregat-Funktionen und Gruppenbildung (GROUP-BY)
  - keine Aggregat-Funktionen in WHERE-Klausel möglich

Sichtendefinition:

```
CREATE VIEW Abtinfo (Anr, Gsumme) AS
    SELECT Anr, SUM (Gehalt)
    FROM Pers
    GROUP BY Anr
```

Anfrage:

```
SELECT AVG (Gsumme) FROM Abtinfo
```

Ersetzung durch

```
SELECT
FROM Pers
GROUP BY Anr
```

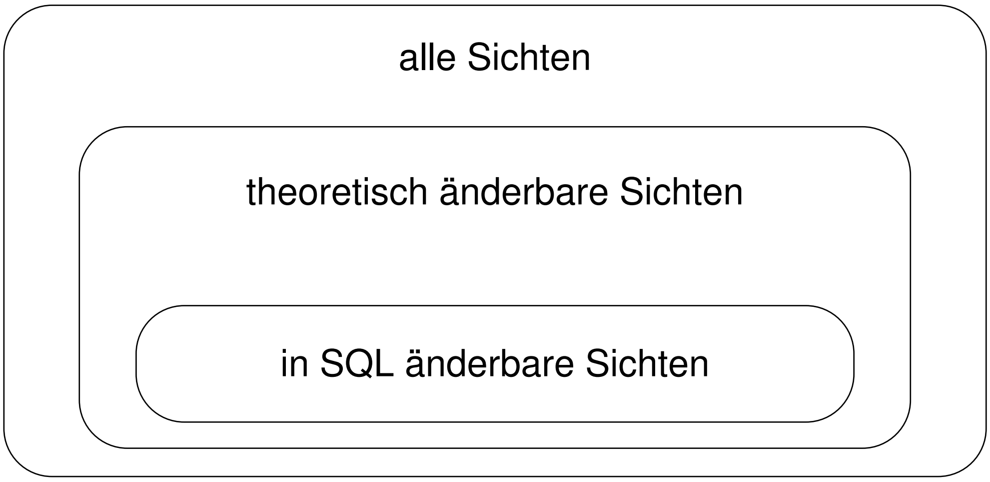
- **Löschen von Sichten:**

DROP VIEW Arme_Programmierer
CASCADE

  - Alle referenzierenden Sichtdefinitionen und Integritätsbedingungen werden mitgelöscht
  - RESTRICT würde eine Löschung zurückweisen, wenn die Sicht in weiteren Sichtdefinitionen oder CHECK-Constraints referenziert werden würde.

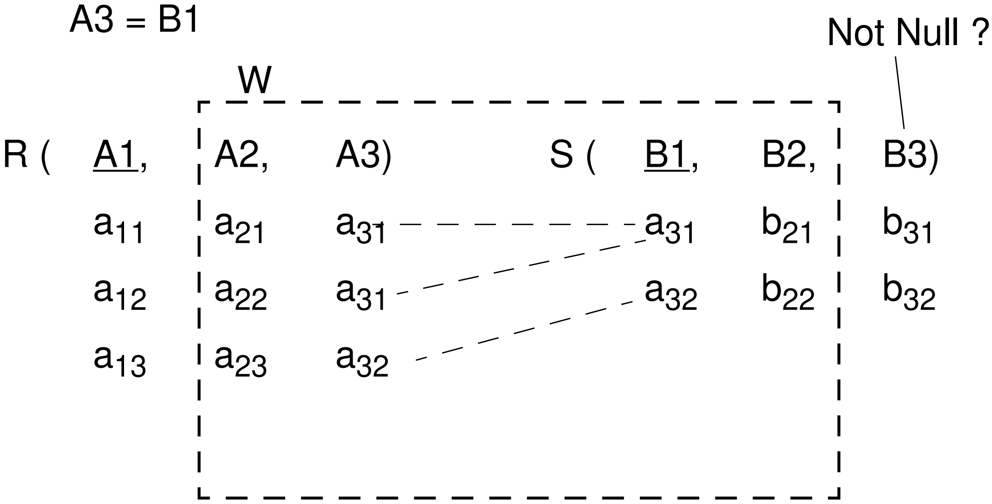
Sichtkonzept (6)

- **Änderbarkeit von Sichten**



- Sichten über mehr als eine Tabelle sind i. allg. nicht aktualisierbar!

$$W = \Pi_{A2,A3,B1,B2}\,(R \underset{A3 = B1}{\bowtie} S)$$



Einfügen ?

Ändern?

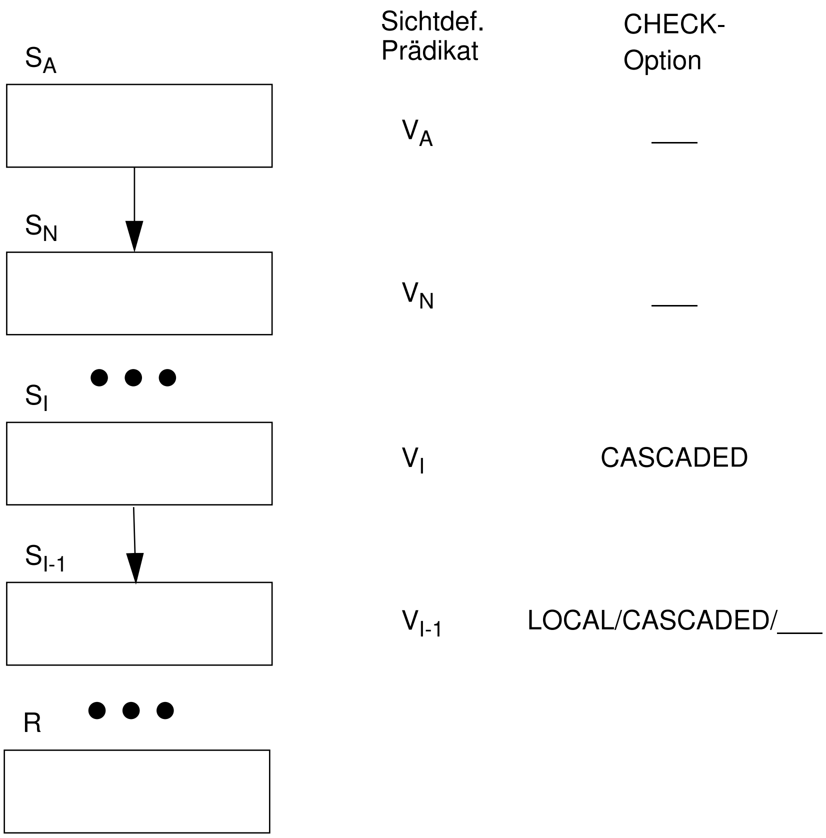
- **Änderbarkeit in SQL**
  - nur eine Tabelle (Basisrelation oder Sicht)
  - Schlüssel muß vorhanden sein
  - keine Aggregatfunktionen, Gruppierung und Duplikateliminierung

Sichtkonzept (7)

- **Überprüfung der Sichtdefinition:** WITH CHECK OPTION
  - Einfügungen und Änderungen müssen das die Sicht definierende Prädikat erfüllen. Sonst: Zurückweisung
  - nur auf aktualisierbaren Sichten definierbar

- **Zur Kontrolle der Aktualisierung von Sichten,** die wiederum auf Sichten aufbauen, wurde die CHECK-Option verfeinert.
  Für jede Sicht sind drei Spezifikationen möglich:
  - Weglassen der CHECK-Option
  - WITH CASCADED CHECK OPTION oder äquivalent WITH CHECK OPTION
  - WITH LOCAL CHECK OPTION

- **Vererbung der Prüfbedingung durch CASCADED**



Sichtkonzept (8)

- **Annahmen**
- Sicht S_A mit dem die Sicht definierenden Prädikat V_A wird aktualisiert
- S_I ist die höchste Sicht im Abstammungspfad von S_A, die die Option CASCADED besitzt
- Oberhalb von S_I tritt keine LOCAL-Bedingung auf

- **Aktualisierung von S_A**
  - als Prüfbedingung wird von S_I aus an S_A "vererbt":
    $V = V_I \wedge V_{I-1} \wedge \ldots \wedge V_1$

    ➥ Erscheint irgendeine aktualisierte Zeile von S_A nicht in S_I, so wird die Operation zurückgesetzt
  - Es ist möglich, daß Zeilen aufgrund von gültigen Einfüge- oder Änderungsoperationen aus S_A verschwinden

- **Aktualisierte Sicht besitzt WITH CHECK OPTION**
  - Default ist CASCADED
  - Als Prüfbedingung bei Aktualisierungen ergibt sich
    $V = V_A \wedge V_N \wedge \ldots \wedge V_I \wedge \ldots \wedge V_1$
  - Zeilen können jetzt aufgrund von gültigen Einfüge- oder Änderungsoperationen nicht aus S_A verschwinden

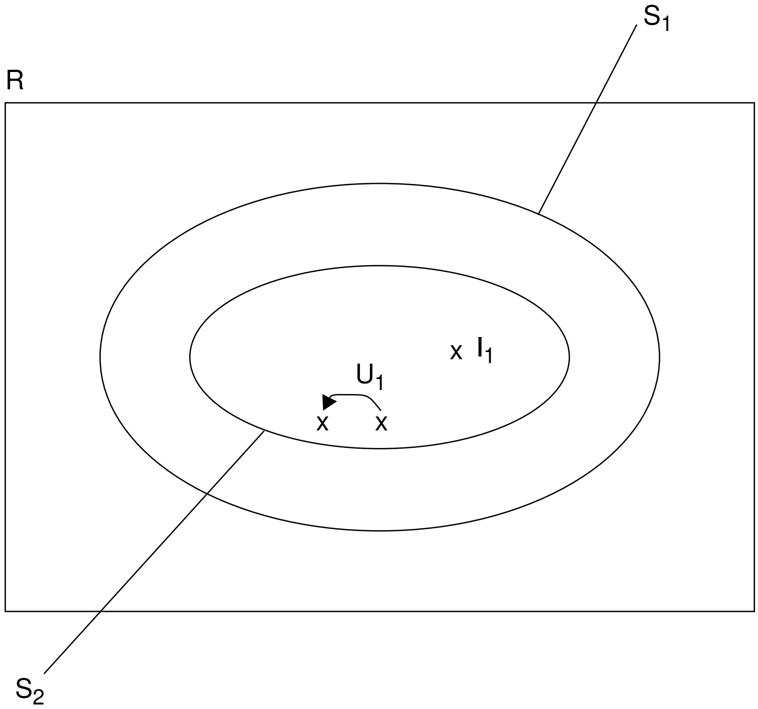
- **LOCAL hat eine undurchsichtige Semantik**
  - Sie wird hier nicht diskutiert
  - Empfehlung: nur Verwendung von CASCADED

Sichtbarkeit von Änderungen

Sichtenhierarchie:

S_2 mit $V_1 \wedge V_2$

S_1 mit V_1 und CASCADED



Aktualisierungsoperationen in S_2

I_1 und U_1 erfüllen das S_2-definierende Prädikat $V_1 \wedge V_2$

I_2 und U_2 erfüllen das S_1-definierende Prädikat V_1

I_3 und U_3 erfüllen das S_1-definierende Prädikat V_1 nicht

Welche Operationen sind erlaubt?

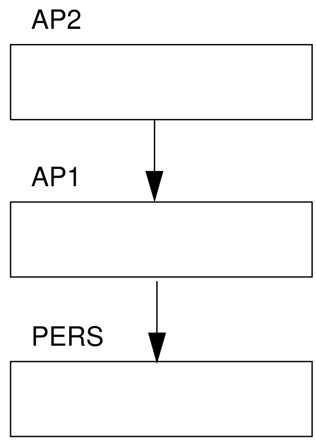
Insert in S2:	I_1	✔
	I_2	
	I_3	
Update in S2:	U_1	✔
	U_2	
	U_3	

Ohne Check-Option werden alle Operationen akzeptiert!

Sichtkonzept (9)

- **Beispiel**

Tabelle	Pers		
Sicht 1 auf Pers	AP1	mit	Beruf = ‘Prog.’ AND Gehalt < ‘30K’
Sicht 2 auf AP1	AP2	mit	Gehalt > ‘20K’



Sichtdef. Prädikat	CHECK-Optionen 1	2	3	4
> ‘20K’	—	—	CASC	CASC
< ‘30K’	—	CASC	—	CASC

- **Operationen**

  a) INSERT INTO AP2
  VALUES (. . . , ‘15K’)

  b) UPDATE AP2
  SET Gehalt = Gehalt + ‘5K’
  WHERE Anr = ‘K55’

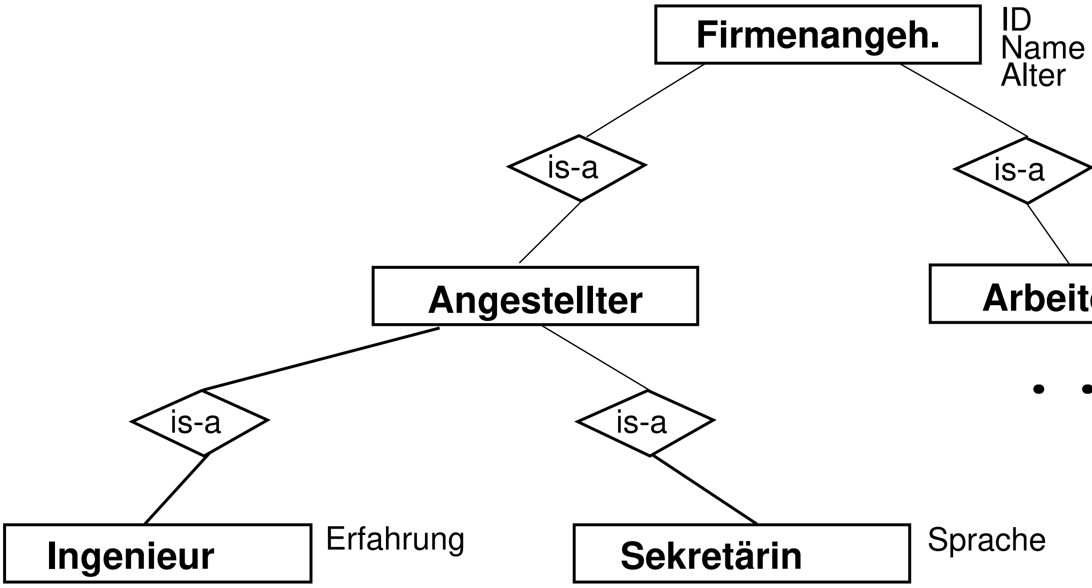
  c) UPDATE AP2
  SET Gehalt = Gehalt - ‘3K’

- **Welche Operationen sind bei den verschiedenen CHECK-Optionen gültig?**

	1	2	3	4
a				
b				
c				

Generalisierung mit Sichtkonzept

- **Ziel: Simulation einiger Aspekte der Generalisierung**



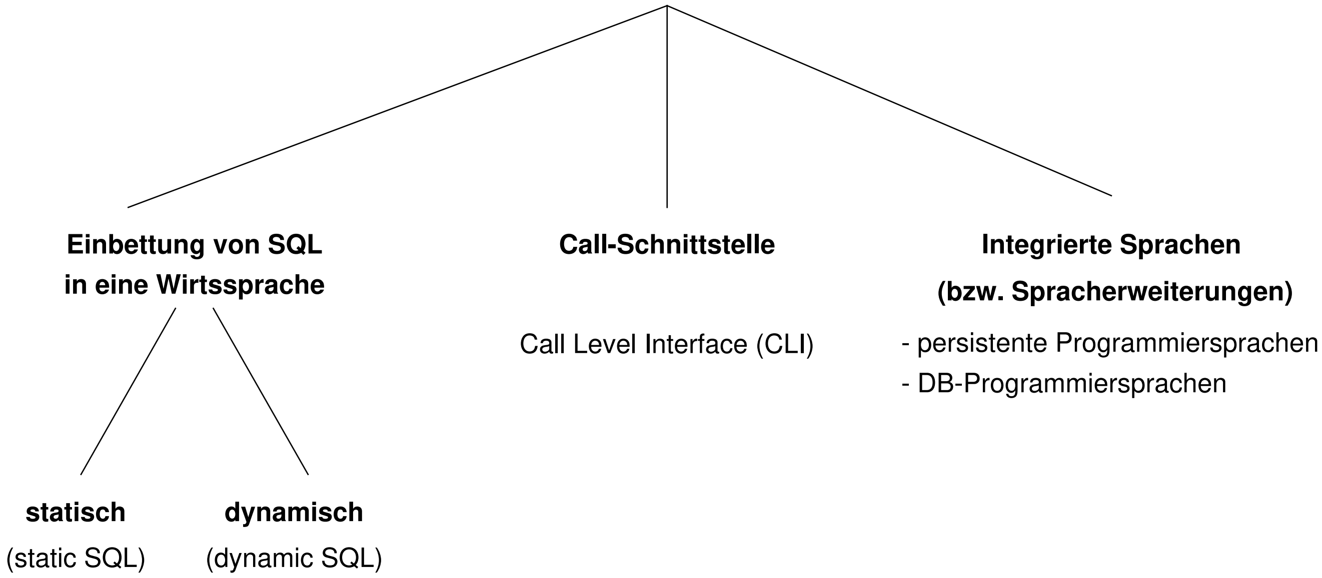
- **Einsatz des Sichtkonzeptes**

```
CREATE  TABLE  Sekretärin
        (ID          INT,
         Name        CHAR(20),
         Alter       INT,
         Sprache     CHAR(15)
         • • •);

INSERT INTO  Sekretärin
   VALUES (436,  'Daisy',  21,  'Englisch');
```

```
CREATE  TABLE  Ingenieur
        (ID          INT,
         Name        CHAR(20),
         Alter       INT,
         Erfahrung   CHAR(15)
         • • • );

INSERT INTO  Ingenieur
   VALUES (123,  'Donald',  37,  'SUN');
```

```
CREATE  VIEW  Angestellter
      AS   SELECT    ID,  Name,  Alter
           FROM      Sekretärin
           UNION
           SELECT    ID,  Name,  Alter
           FROM      Ingenieur;

CREATE  VIEW  Firmenangehöriger
      AS   SELECT    ID,  Name,  Alter
           FROM      Angestellter
           UNION
           SELECT    ID,  Name,  Alter
           FROM      Arbeiter;
```

Kopplung mit einer Wirtssprache



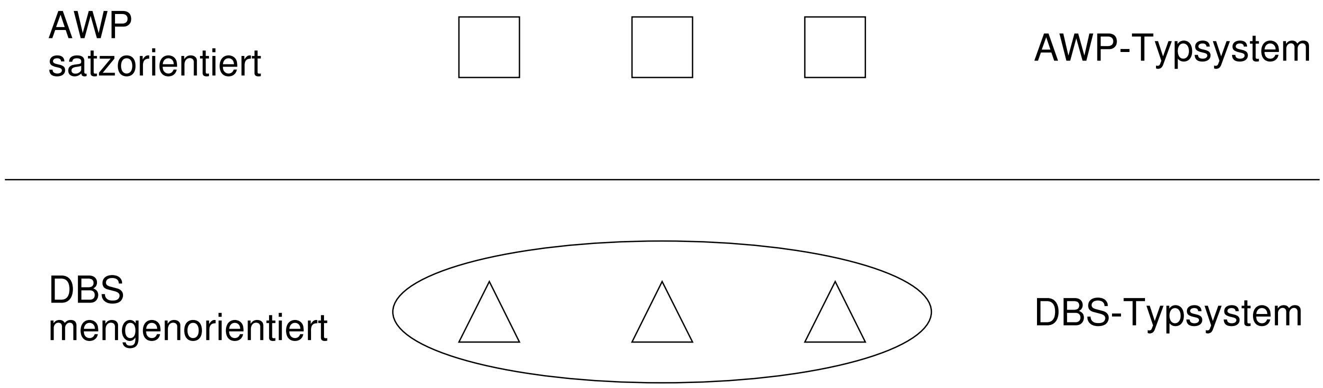
- **Call-Schnittstelle**
  (prozedurale Schnittstelle, CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
- **Einbettung von SQL** (Embedded SQL, ESQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - komfortablere Programmierung als mit CLI
- **statische Einbettung**
  - Vorübersetzer (Pre-Compiler) 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)

  ➥ 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 im AWP?

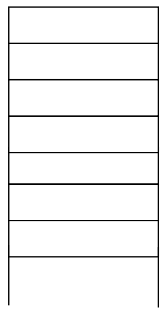
- **Cursor-Deklaration**

```
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**

C_1

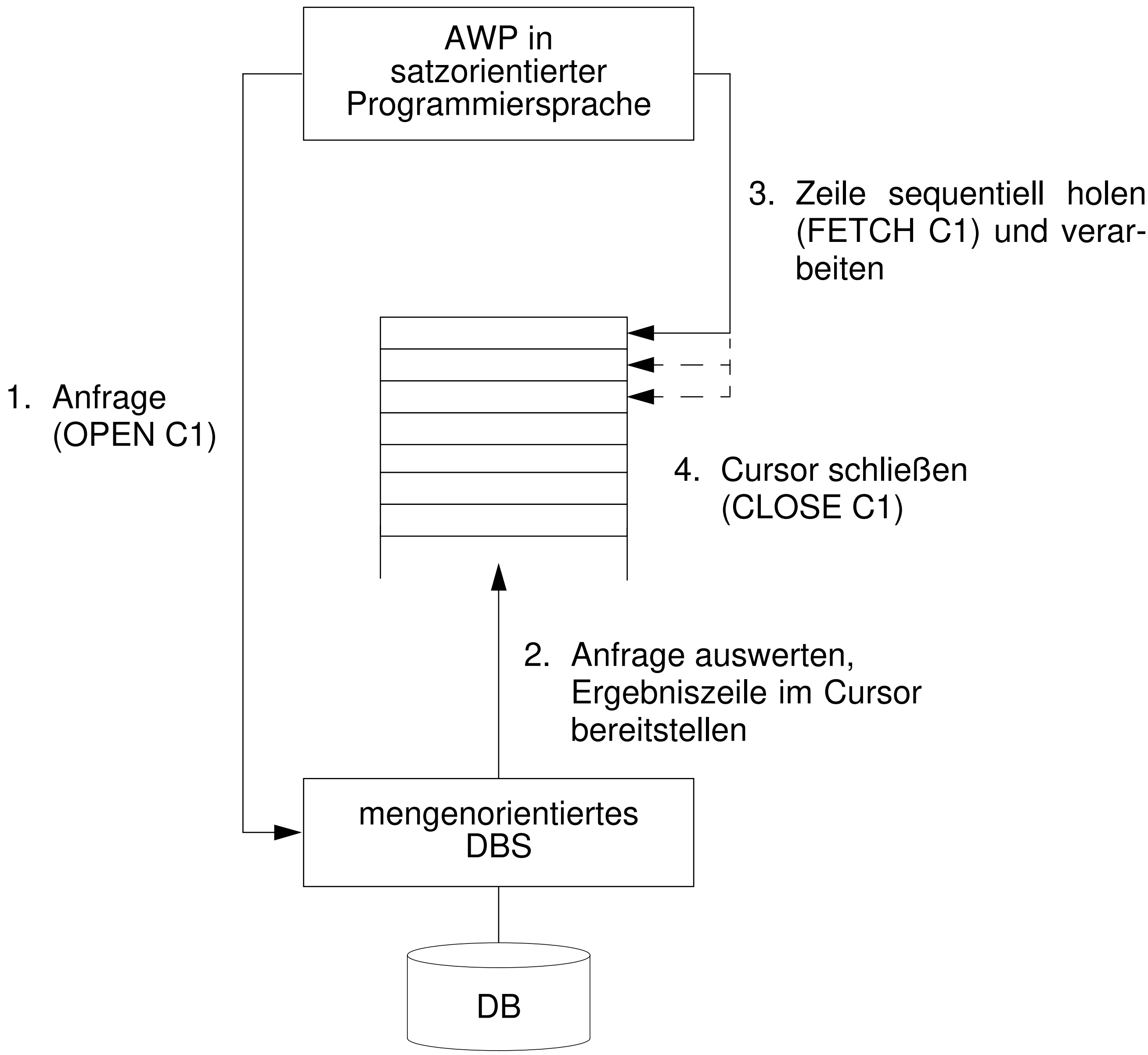


```
OPEN   C1
FETCH C1  INTO  Var1, Var2, . . ., Varn
CLOSE C1
```

- **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)

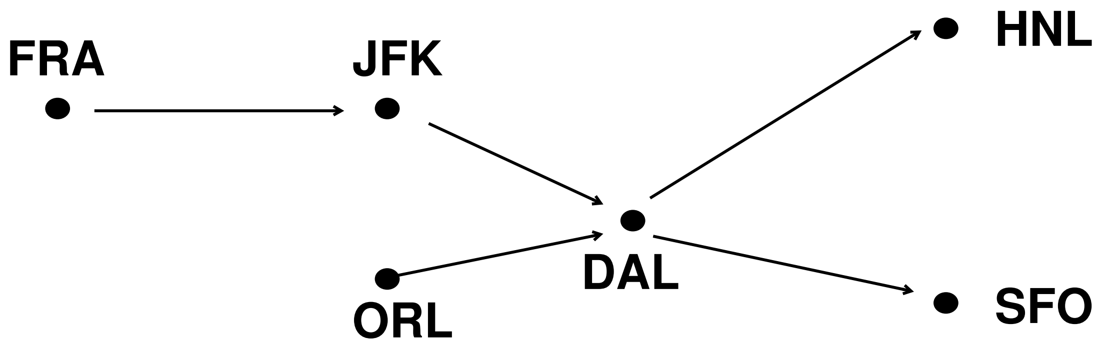
- **Aktualisierung mit Bezugnahme auf eine Position**
  - Wenn die Zeilen, die ein Cursor verwaltet (*active set*), eindeutig Zeilen einer Tabelle entsprechen, können sie über Bezugnahme durch den Cursor geändert werden.
  - Keine Bezugnahme bei INSERT möglich !

```
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;
    }
```

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		

• • •

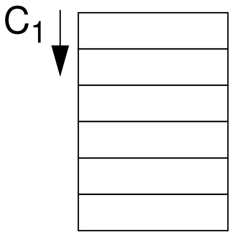
- Flug von FRA nach HNL ?

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

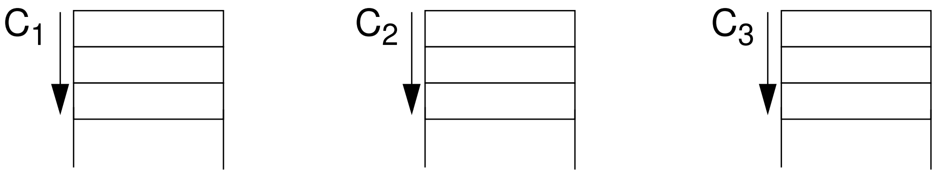
- Flug von FRA nach HNL (Anzahl der Teilstrecken bekannt) ?

SQL-Programmiermodell

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

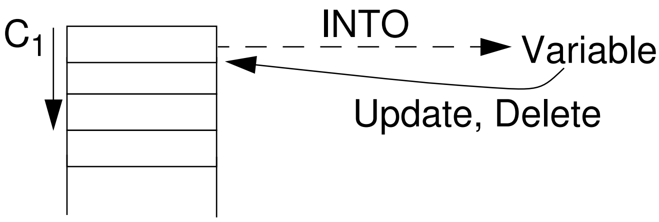


2) **mehrere Cursor**: π, σ, Sort, ...

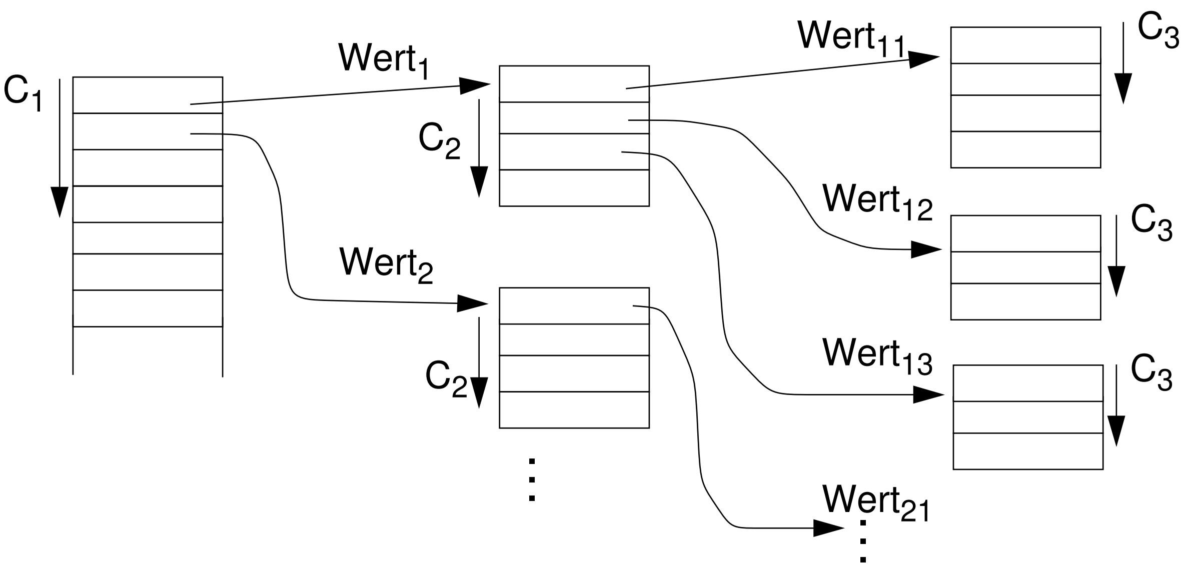


Verknüpfung der gesuchten Zeilen im AP

3) **Positionsbezogene Aktualisierung**



4) **abhängige Cursor**



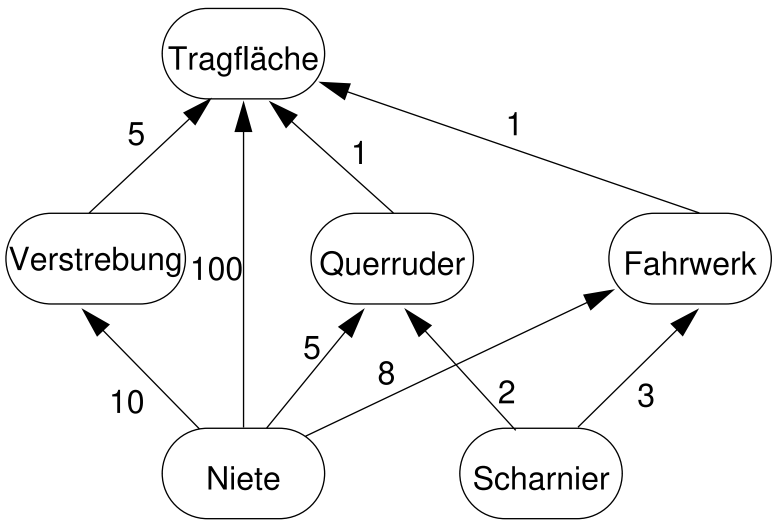
Beispiel: Stücklistenauflösung

- 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
    - ➥ 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 SQLSTATE**
  - standardisierte Fehler-Codes
  - nähere Angaben zu Fehlersituationen in einem Diagnostik-Bereich des DBS

  ➥ 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

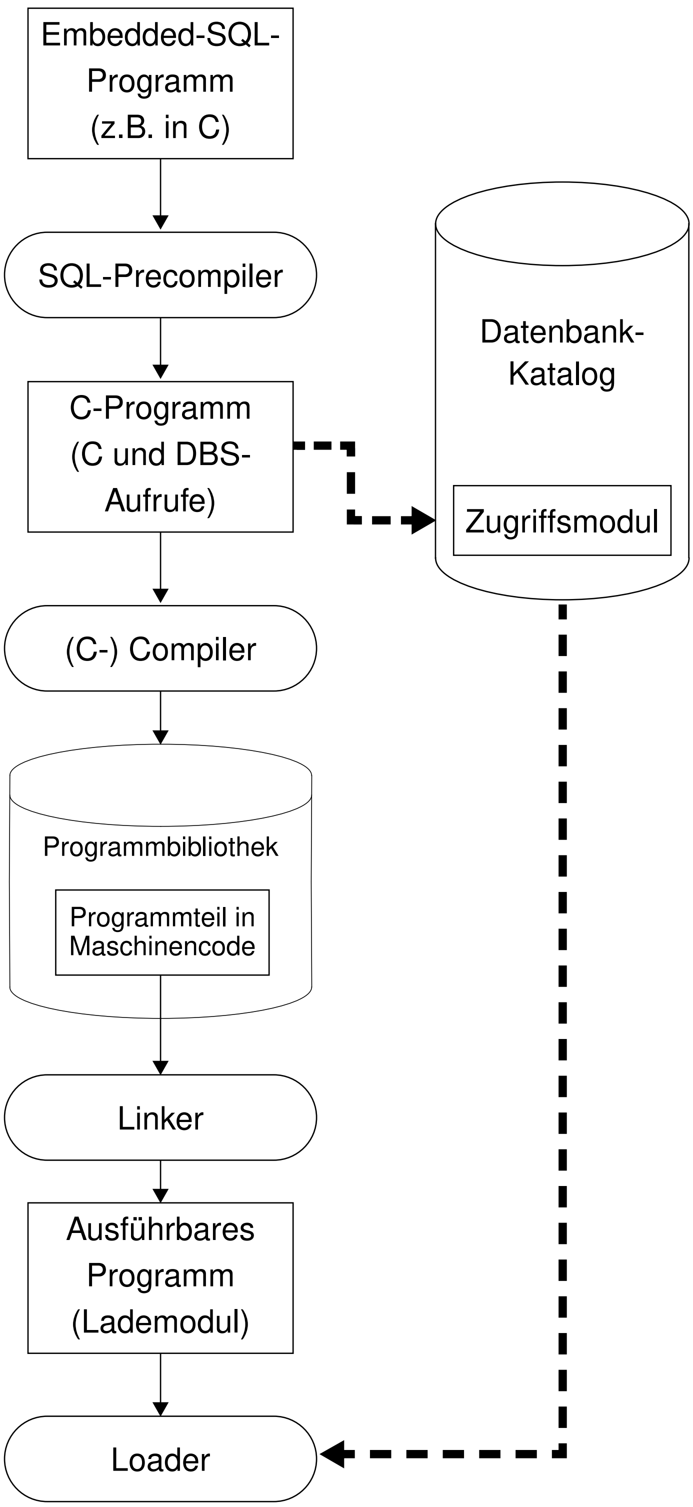
- **Prinzipielle Möglichkeiten**
  - **Direkte Einbettung**
    - keine syntaktische Unterscheidung zwischen Programm- und DB-Anweisungen
    - DB-Anweisung wird als Zeichenkette A ins AP integriert, z. B.
      exec sql open C1
  - **Aufruftechnik**
    DB-Anweisung wird durch expliziten Funktionsaufruf an das Laufzeit-system 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-SQL-Anweisungen 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

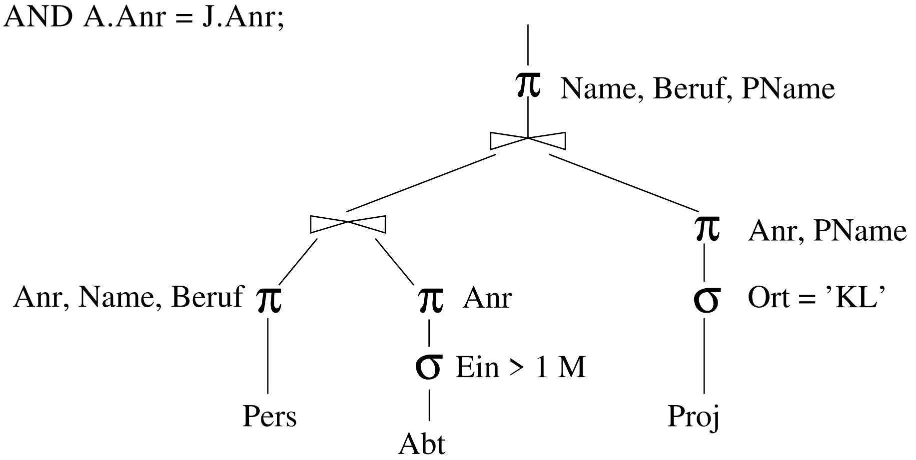
- **Problemdarstellung - Beispiel**

SQL:

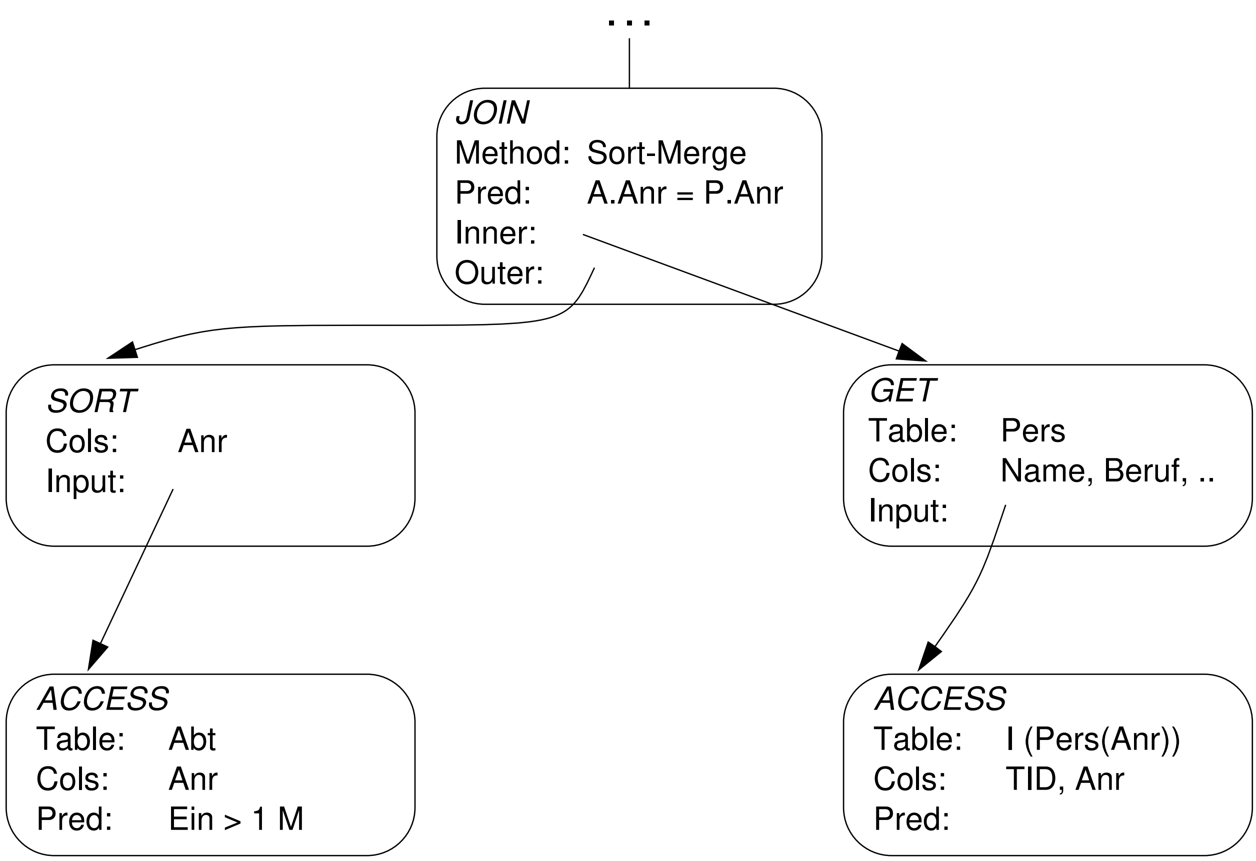
```
SELECT   P.Name, P.Beruf, J.PName
FROM     Pers P, Abt A, Proj J
WHERE    A.Ein > 1000000 AND J.Ort = ‘KL‘
  AND    A.Anr = P.Anr AND A.Anr = J.Anr;
```

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)

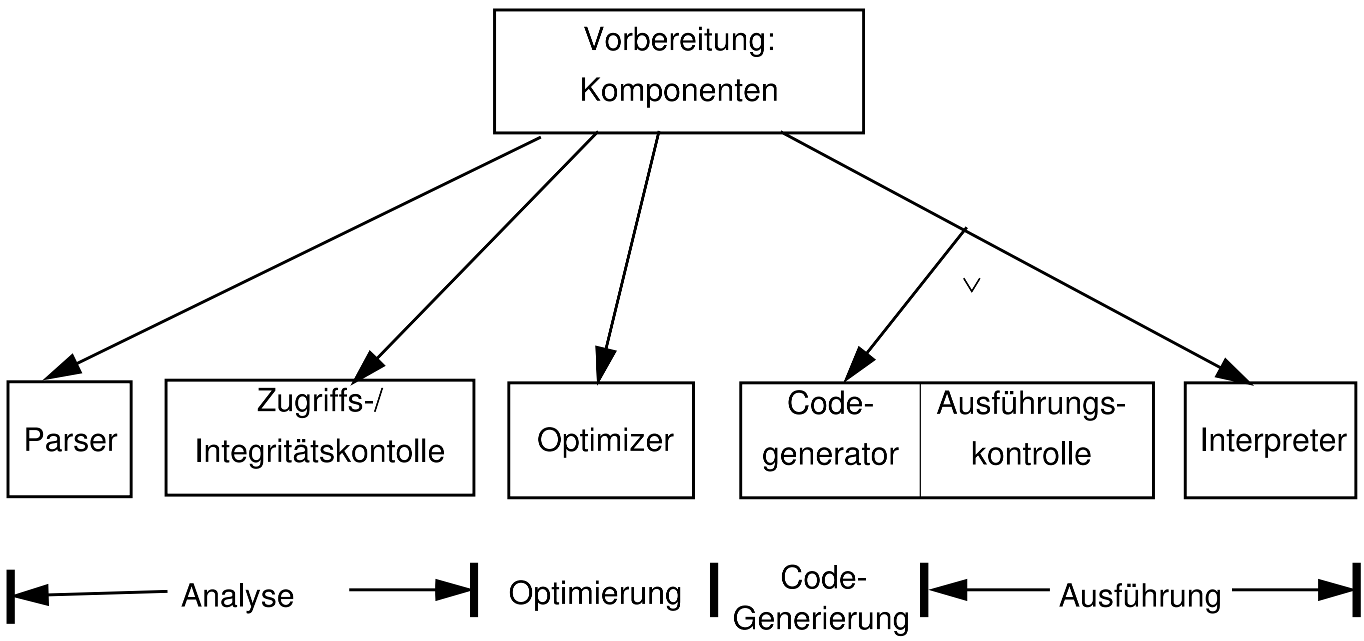
5. Restrukturierung und Transformation

- 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

  ➥ Schritte 4 + 5 werden als Anfrageoptimierung zusammengefaßt

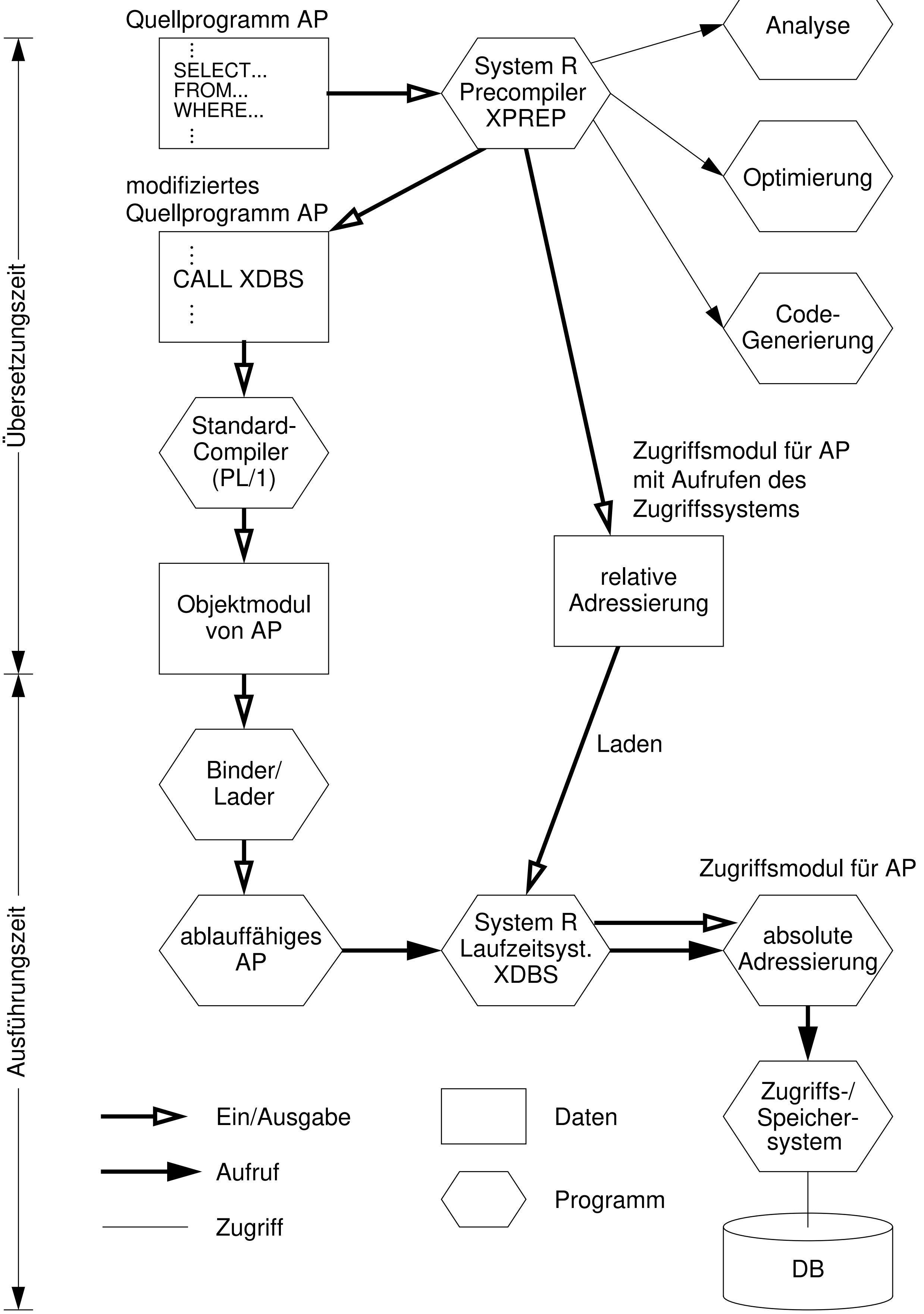
6. Code-Generierung

- 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**

Übersetzungs-
zeit (ÜZ)

AP

Lauf-
zeit (LZ)

Invalidierung

durch Schemaänderungen

Übersetzungskosten:

- unerheblich für Antwortzeit (AZ)

Zugriffe (zur LZ):

- effizient
- datenabhängig!

Interpretation:

- erheblich für AZ

Zugriffe (zur LZ):

- teuer
- datenunabhängig!

Ausgleich gesucht!

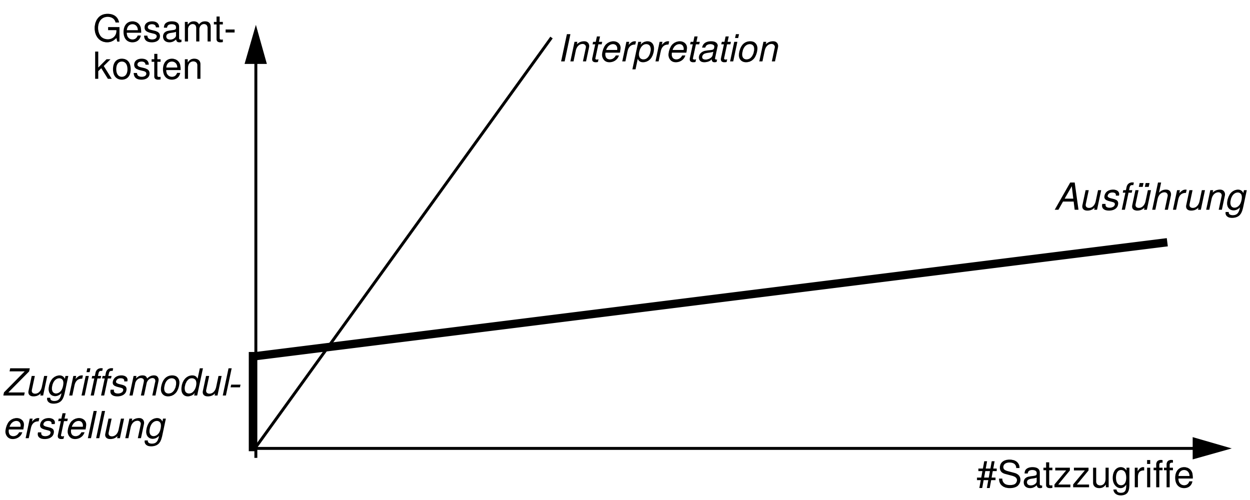
Auswertung von DB-Anweisungen (5)

- **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.)

    ➥ 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



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

  ➥ **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

    ➥ Bindung zur Übersetzungszeit (statisches SQL) 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

  ➥ Funktionalität ähnlich, jedoch nicht identisch

- **Gleiche Anforderungen**
  - Übergabe dynamisch berechneter SQL-Anweisungen (LZ)
  - Trennung von Vorbereitung und Ausführung
    - einmalige Vorbereitung mit Parametermarkern (Platzhalter für Parameter)
    - n-malige Ausführung
  - Explizite Bindung von Parametermarkern 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:
  - PREPARE
  - DESCRIBE
  - EXECUTE
  - EXECUTE IMMEDIATE

- **SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt**
  - Deklaration DECLARE STATEMENT
  - Anweisungen enthalten Parametermarker (?) 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)

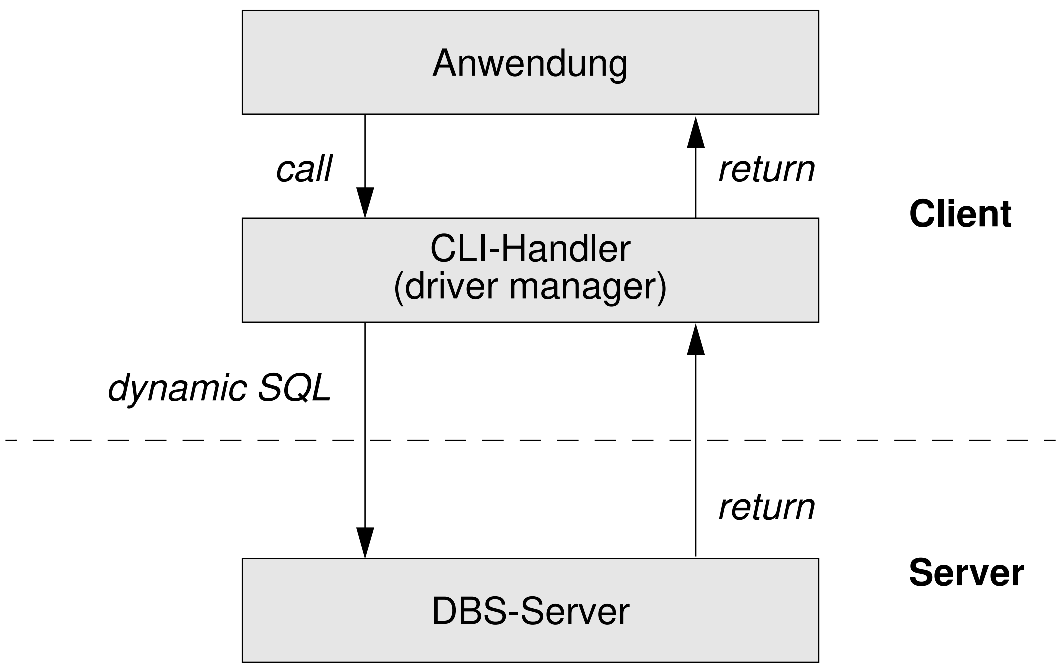
```
Anweisung = ‘INSERT INTO Pers VALUES (?, ?, ...)’;
exec sql prepare SQLanw from :Anweisung;
exec sql allocate descriptor ‘Eingabeparameter’;
exec sql describe input SQLanw into sql descriptor ‘Eingabeparameter’;
exec sql get descriptor ‘Eingabeparameter’ :n = COUNT;

for (i = 1; i < n; i ++)
  {
  exec sql get descriptor ‘Eingabeparameter’
          value :i, :t = type, ...;
  exec sql set descriptor ‘Eingabeparameter’
          value :i, data = :d, indicator = :ind;
  }

  exec sql execute SQLanw
          using sql descriptor ‘Eingabeparameter’;
```

Call-Level-Interface

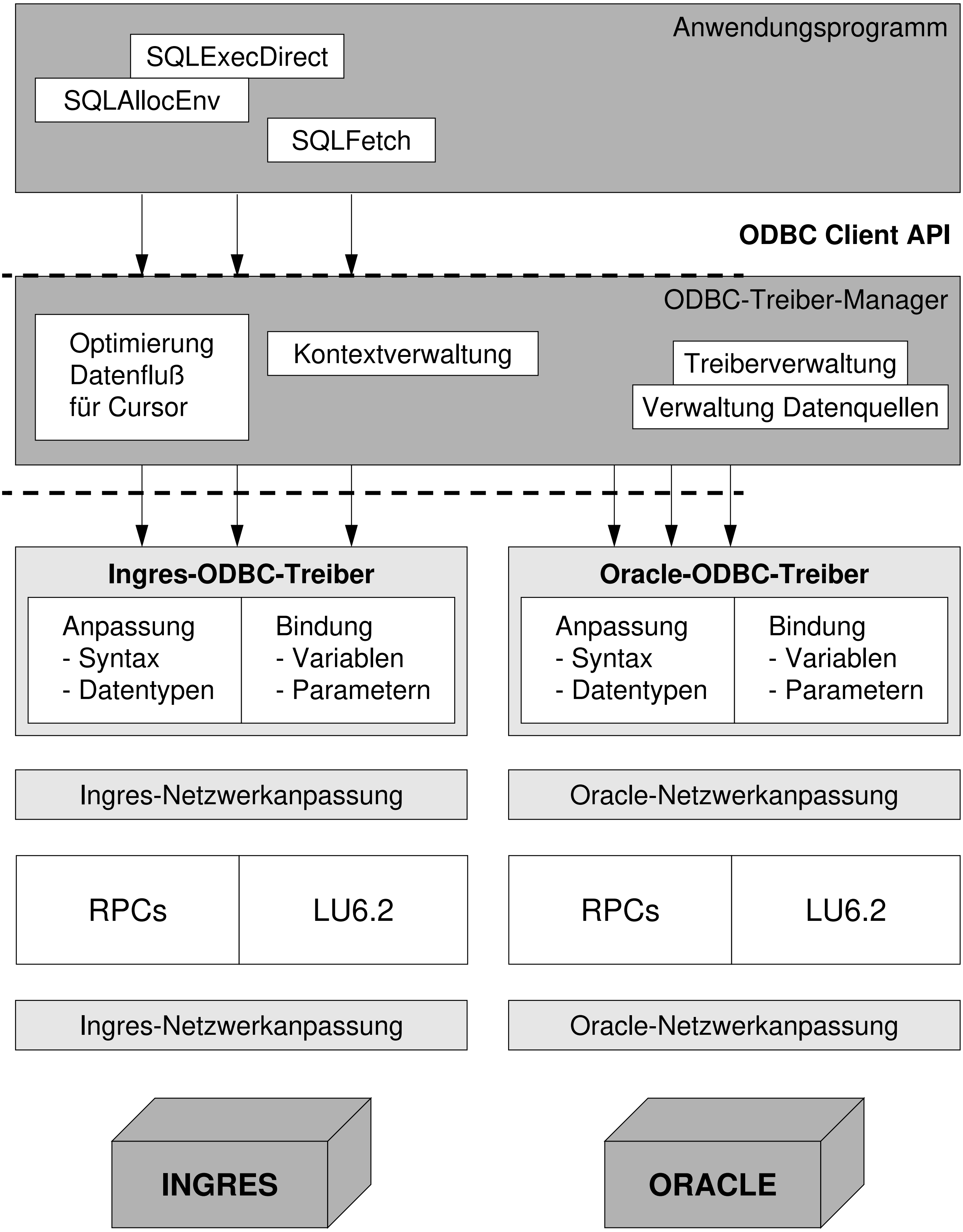
- **Spezielle Form von dynamischem SQL**
  - Schnittstelle ist als Sammlung von Prozeduren/Funktionen realisiert
  - Direkte Aufrufe der Routinen einer standardisierten Bibliothek
  - Keine Vorübersetzung (Behandlung der DB-Anweisungen) von Anwendungen
    - Vorbereitung der DB-Anweisung geschieht erst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

  ➥ Schnittstelle wird sehr häufig im der Praxis eingesetzt!

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



Beispiel Microsoft
- ODBC-Architektur -

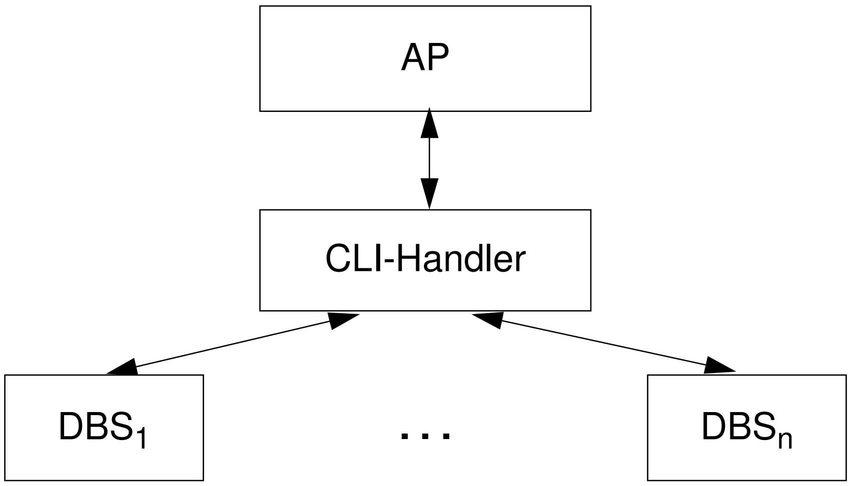


Call-Level-Interface (2)

- **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 gesicherte Prozeduren 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) ;
}
```

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

Zusammenfassung

- **Schemaevolution**
  Änderung/Erweiterung von Spalten, Tabellen, Integritätsbedingungen, ...

- **Sichtenkonzept**
  - Erhöhung der Benutzerfreundlichkeit
  - Flexibler Datenschutz
  - Erhöhte Datenunabhängigkeit
  - Rekursive Anwendbarkeit
  - Eingeschränkte Aktualisierungsmöglichkeiten

- **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 Pre-Compiler
  - relativ einfache Programmierung
  - Aufbau aller SQL-Befehle muß zur Übersetzungszeit festliegen
  - es können zur Laufzeit nicht verschiedene Datenbanken dynamisch angesprochen werden

Zusammenfassung (2)

- **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

- **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 ($\rightarrow$ einfacheres Debugging)

- **PSM**
  - zielt ab auf Leistungsverbesserung vor allem in Client/Server-Umgebung
  - erhöht die Verarbeitungsmächtigkeit des DBS

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 Windows-internen 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 lBonus;
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 */

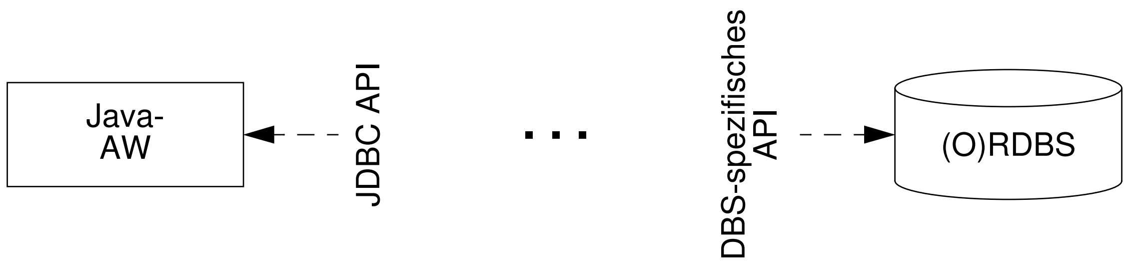
retcode = **SQLTransact** (*henv*, *hdbc*, SQL_COMMIT);

/* 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 Result Sets, ...)
    - Optional Package: Ergänzende Funktionalität (Connection Pooling, verteilte Transaktionen, ...)

- **Allgemeines Problem**
  Verschiedene DB-bezogene APIs sind aufeinander abzubilden

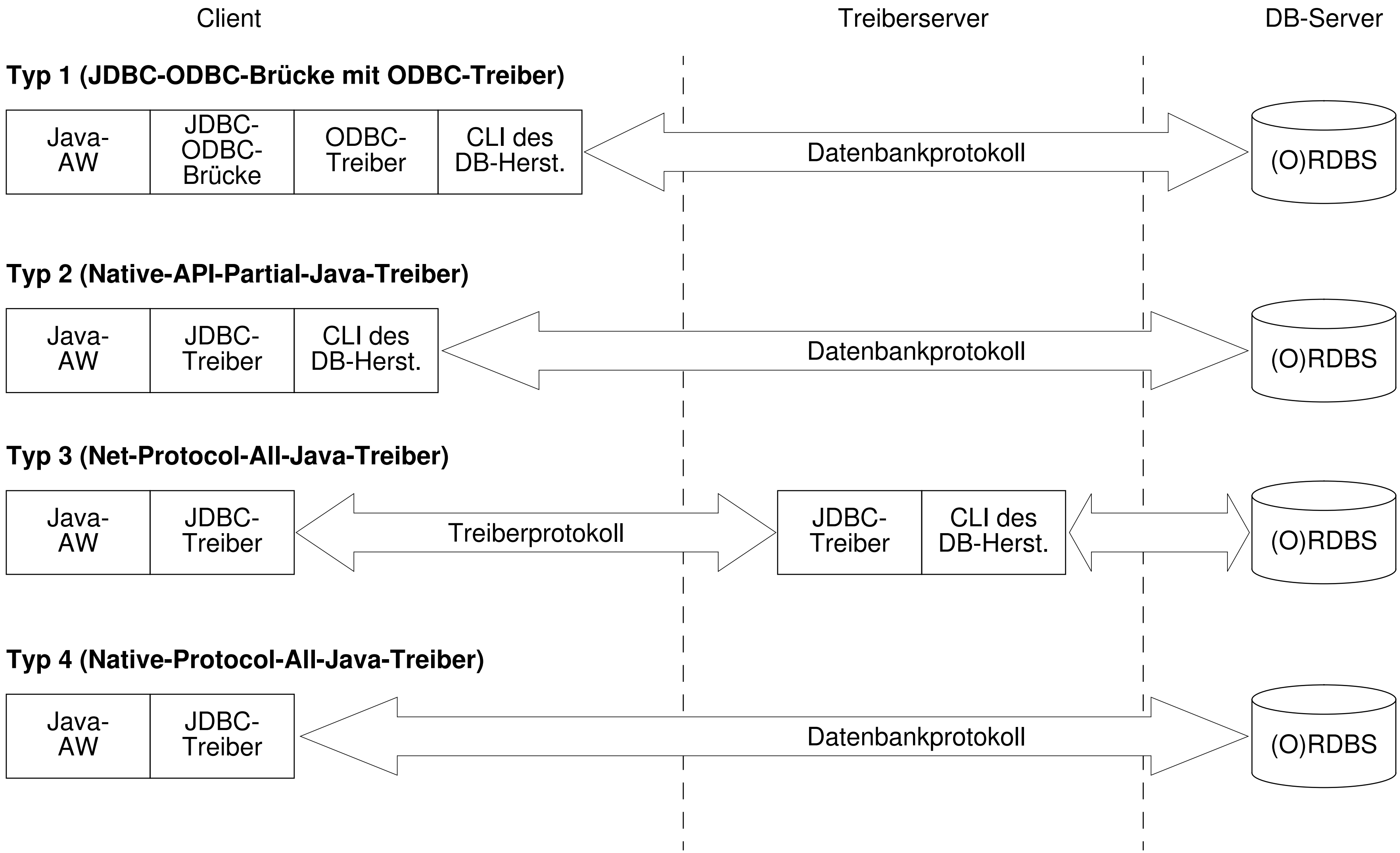


- **Überbrückung/Anpassung durch Treiber-Konzept**
  - setzen JDBC-Aufrufe in die DBS-spezifischen Aufrufe um
  - Treiber werden z.B. vom DBS-Hersteller zur Verfügung gestellt
  - Treiber-Unterstützung kann auf vier verschiedene Arten erfolgen

---

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
  - Herstellerspezificher 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.

DB-Zugriff via JDBC (4) - wichtige Funktionalität

- **Laden des Treiber**
  - kann auf verschiedene Weise erfoligen, 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();
    ```

- **Prepared-Anweisungen**
  - PreparedStatement-Objekt erlaubt das Erzeugen und Vorbereiten von (parametrisierten) SQL-Anweisungen zur wiederholten Ausführung
  - Bei der Ausführung sind dann die aktuellen Parameter einzusetzen:

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

  - Einfügen von aktuellen Parametern mit Methoden wie setDouble, setInt, setString usw. und Indexangabe

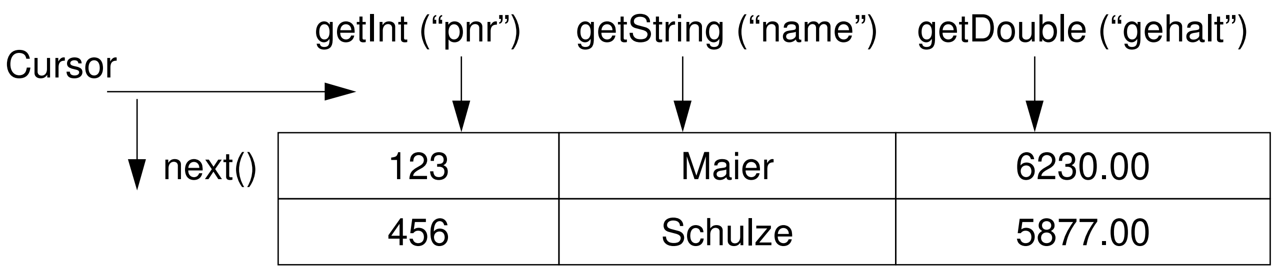
    ```
    pstmt.setDouble (1, gehalt);
    ```

DB-Zugriff via JDBC (5)
- wichtige Funktionalität

- **Select-Anfragen und Ergebnisübergabe**
  - Jede Methode von JDBC, 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



  - 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”) );
```

- **Datenmanipulation**
  - Mit Statement-Objekt werden Anfragen und Änderungen durchgeführt
  - Es sind auch Anweisungen zur Definition und Evolution von Schemaobjekten möglich

```
stmt.executeUpdate (updateStatement);
```

- **Weitere Anweisungen**
  - Schließen von Verbindungen, Statements und ResultSets durch close()
  - Sämtliche Methoden können Ausnahmen werfen (SQLException); sie sind daher in Try-Catch-Blöcken zu klammern

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 ();
```