Karlsruher Institut für Technologie

Datenbanksysteme

Kapitel 12: Anwendungsprogrammierung – Zusatzmaterial zur Vorlesung

Lehrstuhl für Systeme der Informationsverwaltung, Fakultät für Informatik

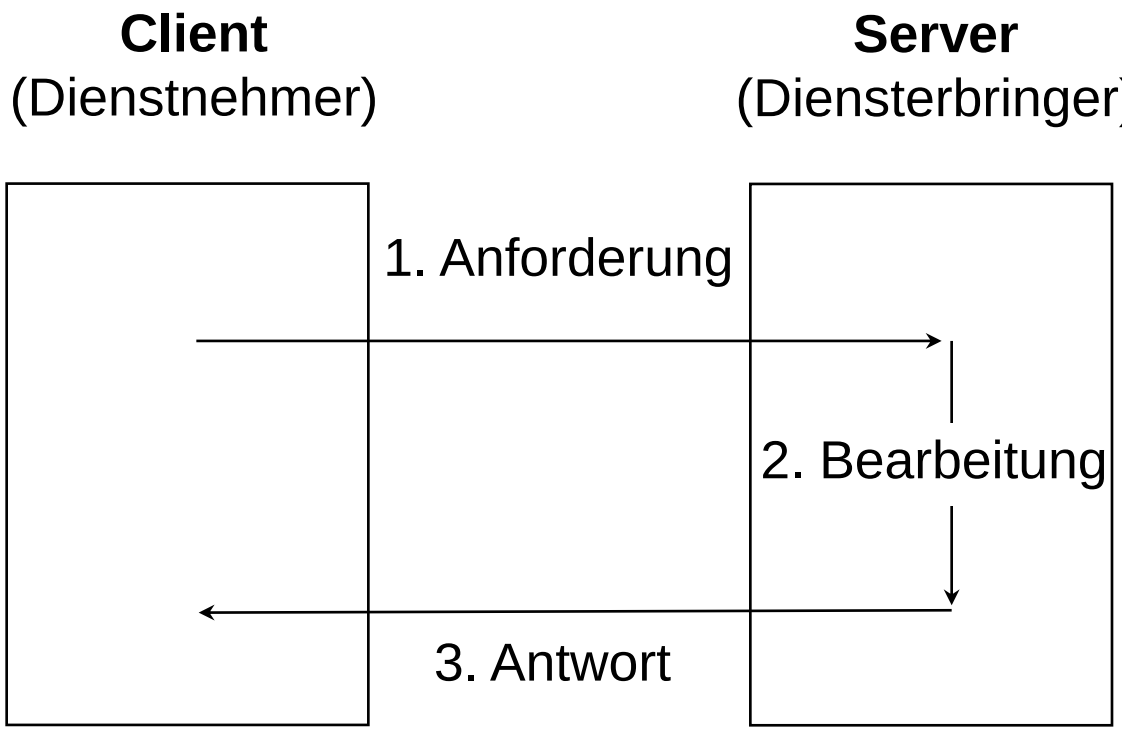
Photo by AshtonPal

Anwendungsprogrammierung

- Client-Server-Architektur,
- Anbindung von Programmiersprachen,
- Call-Level-Schnittstellen: SQL/CLI, JDBC,
- Einbettung: Embedded SQL, SQLJ,
- Persistenz von (Java-)Objekten,
- gespeicherte Prozeduren,
  prozedurale Erweiterungen: PL/SQL,

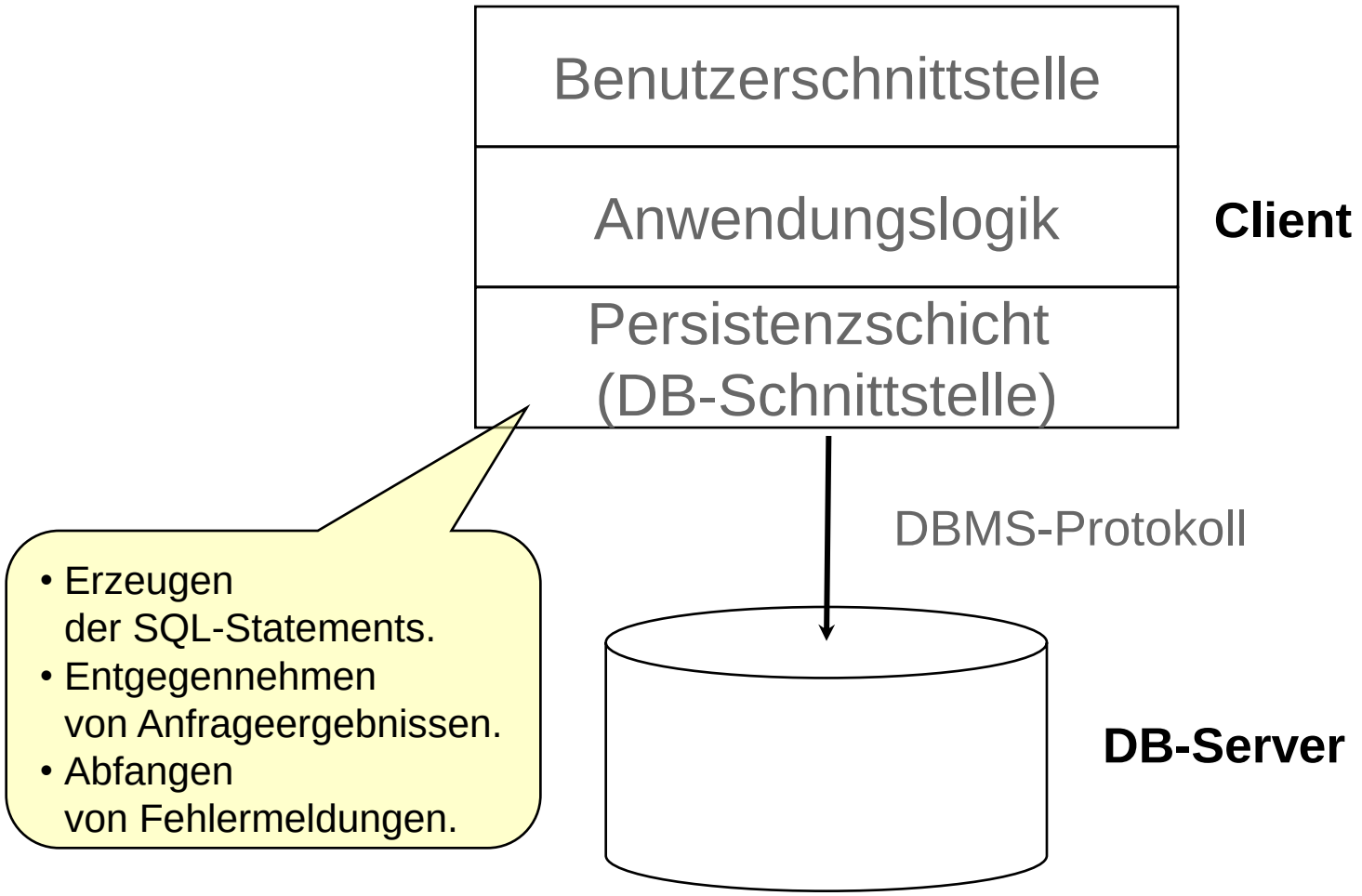
- Software Performance Antipatterns.

Client-Server Architektur

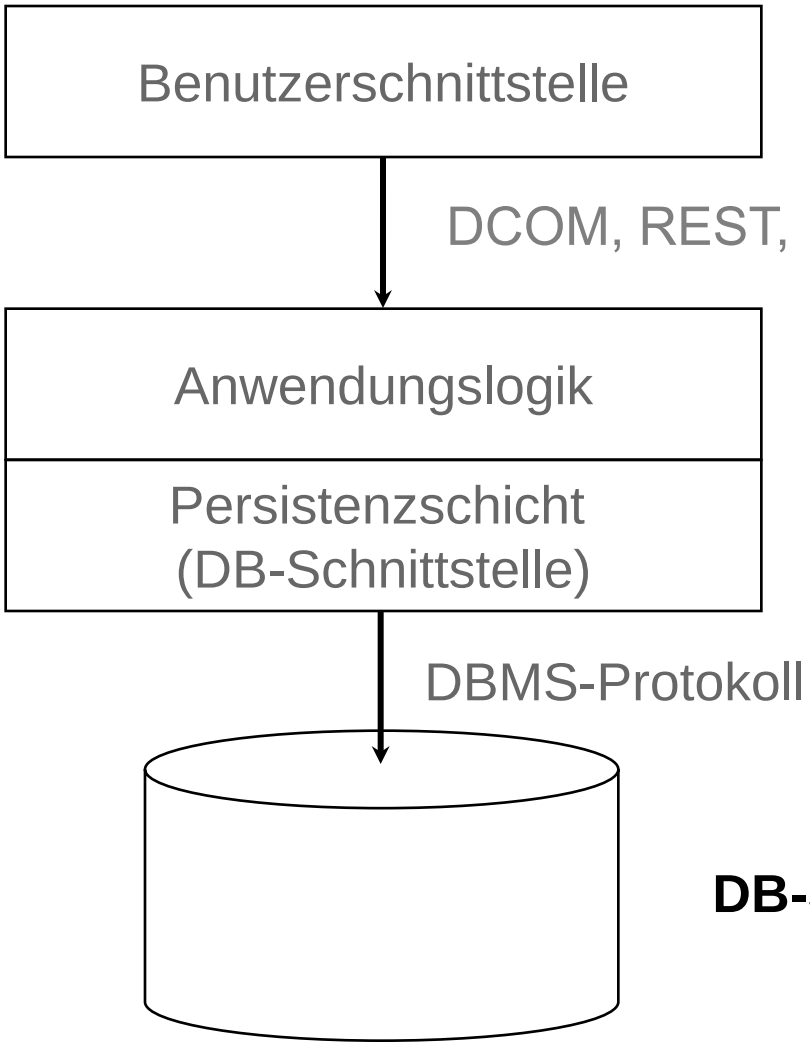


- Erfordert
  - Kenntnis über angebotene Dienste,
  - Protokoll zur Regelung der Interaktion.

2-Schichten-Architektur



3-Schichten-Architektur



Client

Applikations-server

DB-Server

Einleitung

Cursor

CLI

Embedded SQL

JPA

Stored Proc.

Antipatterns

Anwendungslogik – Illustration (1)

- *Anwendungslogik* = Algorithmen,
  die anwendungsspezifisches Wissen beinhalten.

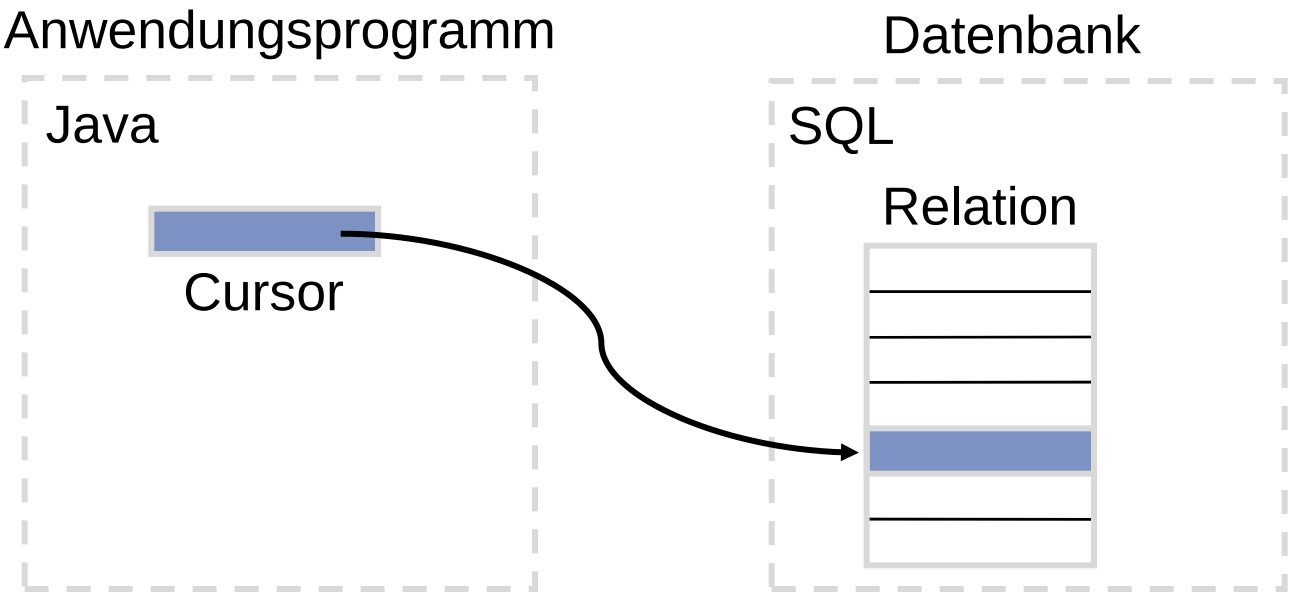
- Datenbank der Personalabteilung enthält Mitarbeiter-Daten – Stammdaten, Kompetenzen, vergangene Projekte, etc.
- Anwendung: Programm, das Teamleiter
  für konkrete Projekte Mitarbeiter vorschlägt.
- Wissen,
  - wie man Projekte klassifiziert,
  - wie bedeutsam bestimmte Fähigkeiten sind,
  - welche Mitarbeiter-Eigenschaften
    für welches Projekt wichtig sind,

  ist Teil der Anwendung.

Anwendungslogik – Illustration (2)

- Anwendungslogik i. Allg.:
  - Wie reagiert Anwendung auf bestimmte Benutzer-Eingaben?
  - Wie arbeitet Anwendung Input des Benutzers ab?
  - In welcher logischen Reihenfolge erfolgen die Zugriffe auf die Datenbank?

Cursor-Konzept



- Programmiersprachen – einzelne Datenobjekte als zugrundeliegende Struktur,
- SQL – Menge von Tupeln,
- *Impedance Mismatch* – Übergang muss geschaffen werden.
- Cursor ≡ Iterator.

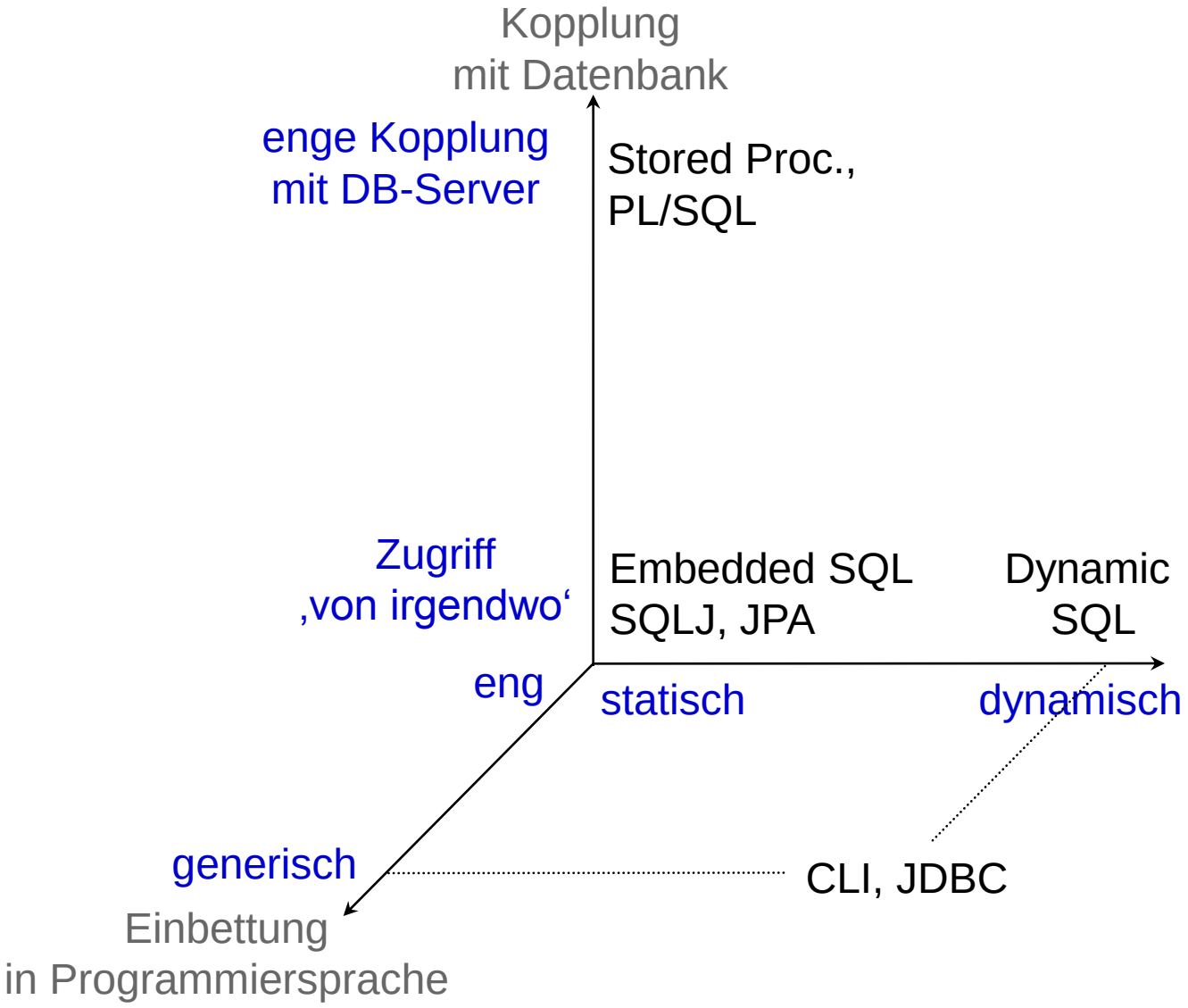
Cursor in SQL92

- Syntax:

  ```
  DECLARE CursorName
      [insensitive] [scroll] CRUSOR FOR...
  ```

- **next**: Gehe weiter zum nächsten Tupel.
- **prior**: Gehe zum vorherigen Tupel.
- **first** bzw. **last**:
  Gehe zum ersten bzw. letzten Tupel.
- **absolute** *n* **from**:
  Gehe zum n-ten Tupel des Cursors.
  Negative Werte werden relativ zum letzten Tupel rückwärts gewertet –
  absolute -1 ist also äquivalent zu **last**.
- **relative** *n* **from**: Gehe zum n-ten Tupel relativ zur aktuellen Cursor-Position.

Programmiersprachenanbindung



Einleitung

Cursor

CLI

- SQL/CLI

- JDBC

Embedded SQL

JPA

Stored Proc.

Antipatterns

Was bedeutet ‚Einbettung in Programmiersprache‘?

- Programmiersprache
  kennt Konzepte relationaler Datenbanken.
- Sei es, dass SQL Teil der Sprache ist
  (z. B. mit Embedded-Erweiterung),
- Sei es, dass Methoden existieren, die speziell sind
  fürs relationale Modell (mit JPA;
  in PL/SQL sind es nicht Methoden,
  sondern Konstrukte wie ROWTYPE).

SQL/CLI (1)

- Call-Level-Interface (CLI):
  Bibliothek von Prozeduren/Funktionen zur
  - Kommunikation mit dem DBMS,
  - Definition und Ausführung von Anfragen,
  - Verarbeitung von Ergebnissen.
- Prominentes Beispiel: JDBC
  - Datenbankzugriffsschnittstelle für Java,
  - abstrakt, datenbankneutral,
  - dynamisches Laden von Treibern.

SQL/CLI (2)

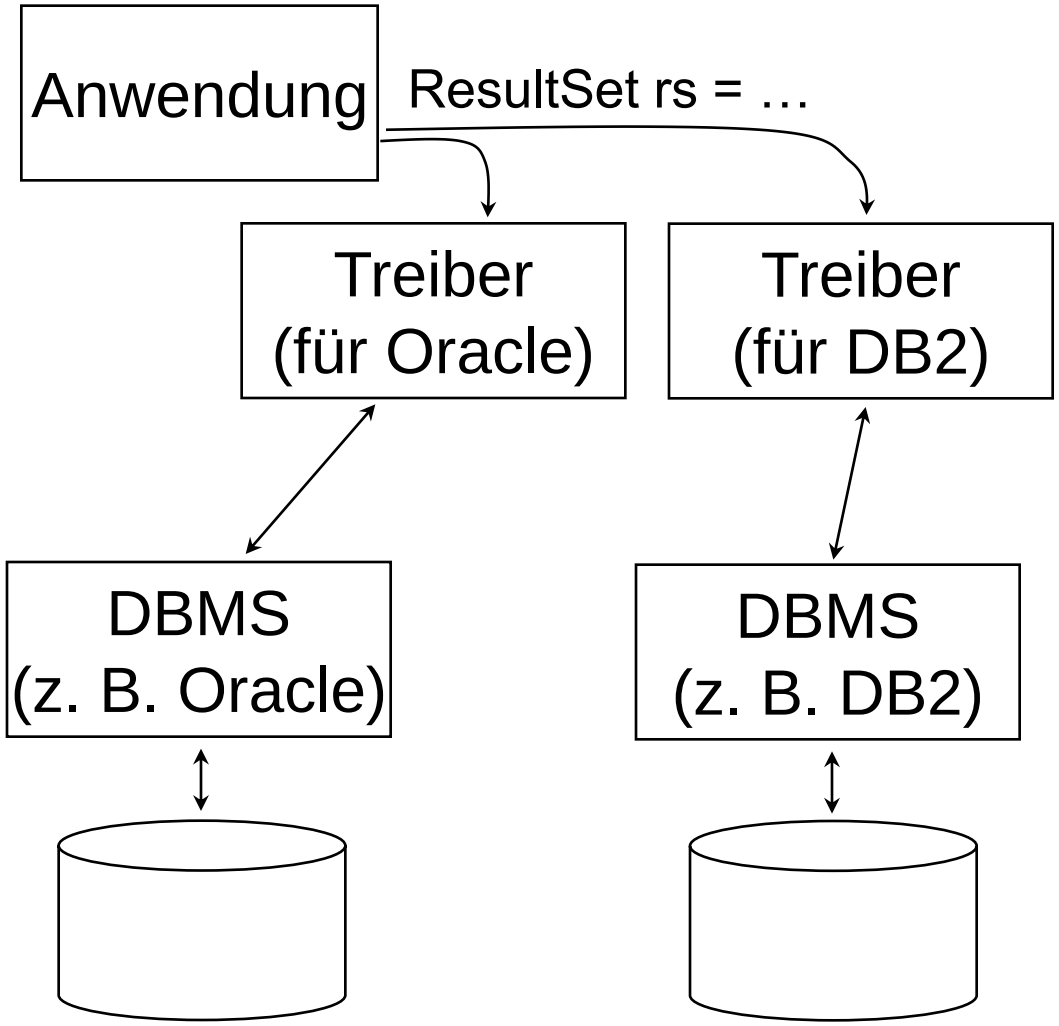
- Beispiel (JDBC):

```
String attrib = eingabe.readLine();
String query =
    "SELECT " + attrib + " FROM buch";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) { ... }
```

- SQL/CLI: ISO-Standard für API.
- Beispiel 2:

```
String attrib = eingabe.readLine();
String query =
    "SELEKTIERE " + attrib + " FROM buch";
ResultSet rs = stmt.executeQuery(query);
```

SQL/CLI (3)



JDBC: Überblick (1)

- Java-Package `java.sql` –
  wichtige Klassen dieses Packages:
  - `DriverManager`:
    Einstiegspunkt, Laden von Treibern
    und Grundlage für Aufbau der Datenbankverbindung.
  - `Connection`: Datenbankverbindung,
  - `Statement`: Ausführung von Anweisungen
    über eine Verbindung.

JDBC: Überblick (2)

- Java-Package `java.sql` –
  wichtige Klassen dieses Packages (Forts.):
  - `ResultSet`:
    verwaltet Ergebnisse einer Anfrage
    in Form einer Relation, Zugriff auf einzelne Spalten.
  - `ResultSetMetaData`: bietet Zugriff
    auf Schema-Informationen zu einem ResultSet.

JDBC: Ablauf

1. Aufbau einer Verbindung zur Datenbank:
   - Angabe der Verbindungsinformationen,
   - Auswahl und Laden des Treibers,
2. Senden einer SQL-Anweisung:
   - Definition der Anweisung,
   - Belegung von Parametern,
3. Verarbeiten der Anfrageergebnisse:
   - Navigation über Ergebnisrelation,
   - Zugriff auf Spalten.

JDBC: Verbindungsaufbau (1)

1. Treiber laden, z. B.:

```
DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
```

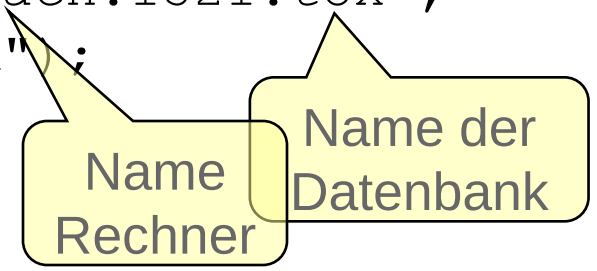
Auch möglich: System-Property
mit Treiber-Namen, die automatisch geladen werden, z. B.:

```
DriverManager.registerDriver(Class.forName
(System.getProperty("com.company.DBDriver")))
```

JDBC: Verbindungsaufbau (2)

2. Verbindung herstellen:

```
Connection con;  // Connection-Objekt
con = DriverManager.getConnection
("jdbc:oracle:thin:@benriach:1521:tox",
      "kboehm", "kbpasswd");
```



- Erster Parameter spezifiziert
  - Datenquelle/Datenbank,
  - Verbindungsmechanismus – Protokoll, Server-Host und Port
    ⇒ zu verwendender Treiber.
- Man kann mehrere unterschiedliche Verbindungen (zu unterschiedlichen Datenbanken) gleichzeitig offen haben.

JDBC: Anfrageausführung

1. Anweisungsobjekt (Statement) erzeugen:

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

2. Anweisung ausführen:

```
String query =
   "SELECT titel, preis FROM buch";
ResultSet rs = stmt.executeQuery(query);
```

JDBC: Ergebnisverarbeitung

1. Navigation über Ergebnismenge (Cursor-Prinzip):

```
while (rs.next ()) {
   // Verarbeitung der einzelnen Tupel
         ...
   }
```

2. Zugriff auf Spaltenwerte des aktuellen Tupels über `get<type>`-Methoden
   - über Spaltenindex:
     `String titel = rs.getString(1);`
   - über Spaltenname:
     `String titel = rs.getString("titel");`
   - Typ-Informationen bekommt man von
     `rs.getMetaData().getColumnType(colIndex).`

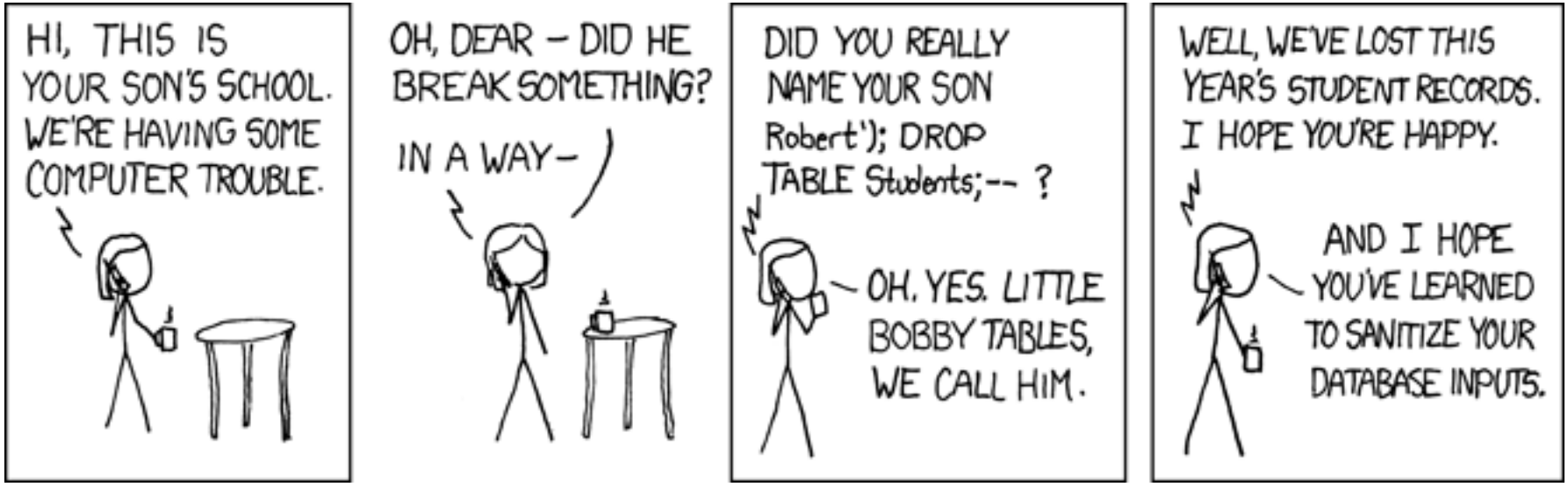
Prepared Statements (1)

- Oft Queries oder Updates mit gleicher Struktur, aber unterschiedliche Parameter hintereinander.
- Beispiel:
  - ```
    SELECT Umsatz FROM COFFEES
    WHERE COF_NAME LIKE "Colombian";
    ```
  - ```
    SELECT Umsatz FROM COFFEES
    WHERE COF_NAME LIKE "Java";
    ```
  - ```
    SELECT Umsatz FROM COFFEES
    WHERE COF_NAME LIKE "Celebes";
    ```
  - ```
    SELECT Umsatz FROM COFFEES
    WHERE COF_NAME LIKE "Denpasar";
    ```
  - …

Prepared Statements (2)

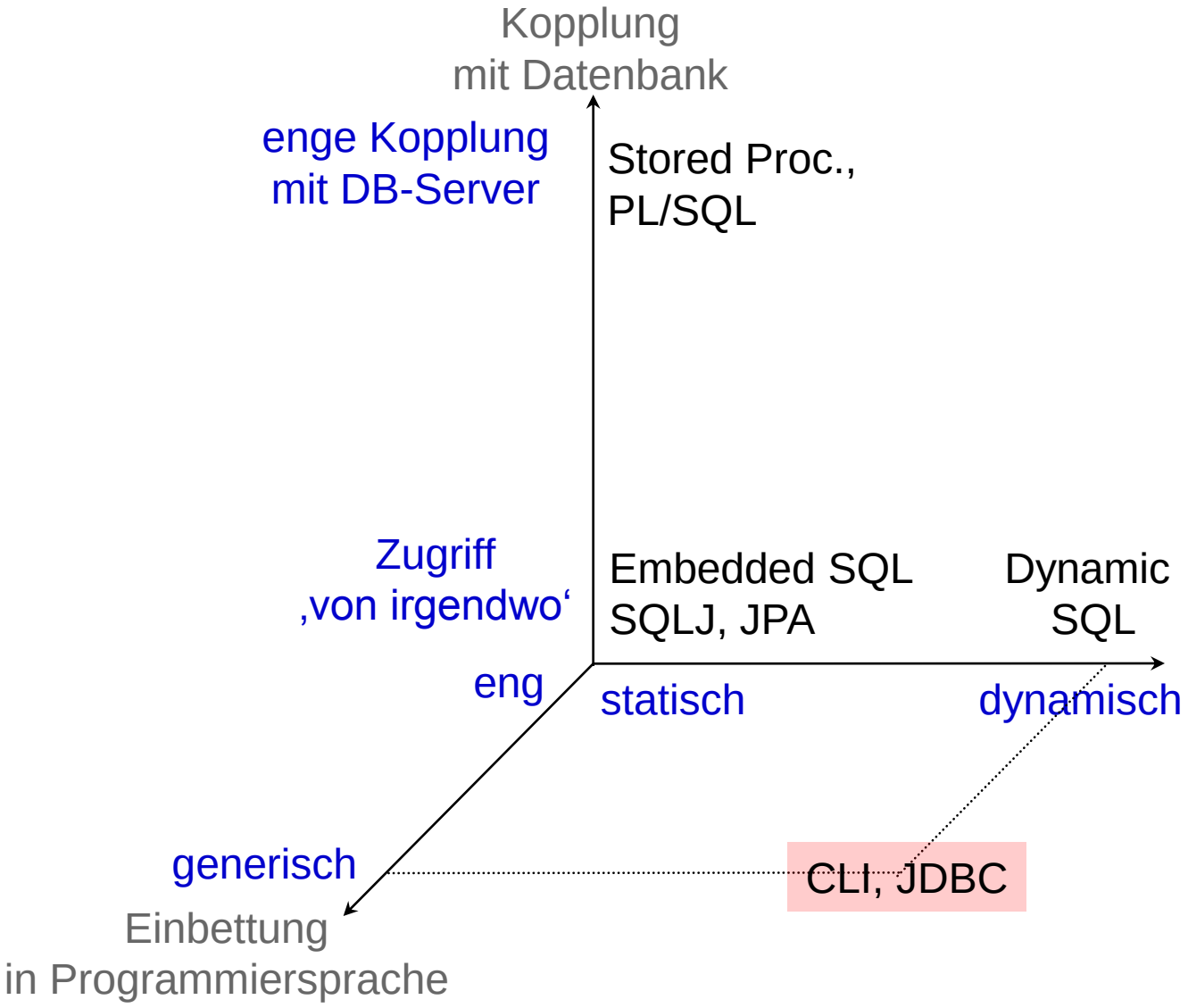
- Verwendung von `PreparedStatement` anstelle von `Statement` reduziert Ausführungszeit.
- Statement dieser Art wird bereits vorab kompiliert.
- Beispiel:
  - ```
    PreparedStatement updateSales =
    con.prepareStatement( "UPDATE COFFEES
    SET SALES = ? WHERE COF_NAME LIKE ?");
    ```
  - `updateSales.setInt(1, 75);`
    erster Parameter identifiziert Spalte.
  - `updateSales.setString(2, "Colombian");`
  - `updateSales.executeUpdate();`
    ```

Prepared Statements (3)



XKCD

Programmiersprachenanbindung



- Einleitung
- Cursor
- CLI
  - SQL/CLI
  - JDBC
- Embedded SQL
- JPA
- Stored Proc.
- Antipatterns

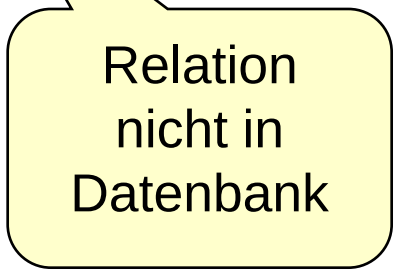
Schema-unabhängige vs. Schema-spezifische Fehler

- Beispiel für Schema-unabhängigen Fehler:

```
SELEKTIERE Ename, Sal
FROM Emp
WHERE Job = 'Professor';
```

- Beispiel für Schema-spezifischen Fehler:

```
SELECT Ename, Sal
FROM Pipapo
WHERE Job = 'Professor';
```



Embedded SQL (1)

- In Anwendungsprogramm eingestreute SQL-Anweisungen.
- Vorübersetzer.
- Umsetzung in Prozeduraufrufe einer Bibliothek.
- Schlüsselwort **`exec sql`** oder anderes – Erkennung durch Vorübersetzer.
- SQLJ – eine mögliche Ausprägung.
  Inzwischen Teil des SQL-Standards,
  unter der Bezeichnung ‚SQL/OBL‘.
  (‚OBL‘ steht für Object Language Bindings).

Embedded SQL (2)

- Beispiel – SQLJ Codefragment:

```
String vName; int vSalary; String vJob;
Java.sql.Timestamp vDate; ...
#sql {    SELECT Ename, Sal
          INTO :vName, :vSalary
          FROM Emp
          WHERE Job = :vJob
                and HireDate = :vDate };
```

- D. h. wir greifen auf das erste Tupel zu.

- Variablen einer Host-Sprache (hier Java), die in SQL-Anweisungen auftreten können.
- Verwendung: Austausch von Daten zwischen Host-Sprache und SQL.
- Kennzeichnung durch "`:variable`".

Deklarationen

- Deklaration benutzter Datenbankrelationen:

```
exec sql declare Buch table
        ( ISBN char(10) not null,
        Titel char(120) not null,
        Verlagsname char(30) not null);
```

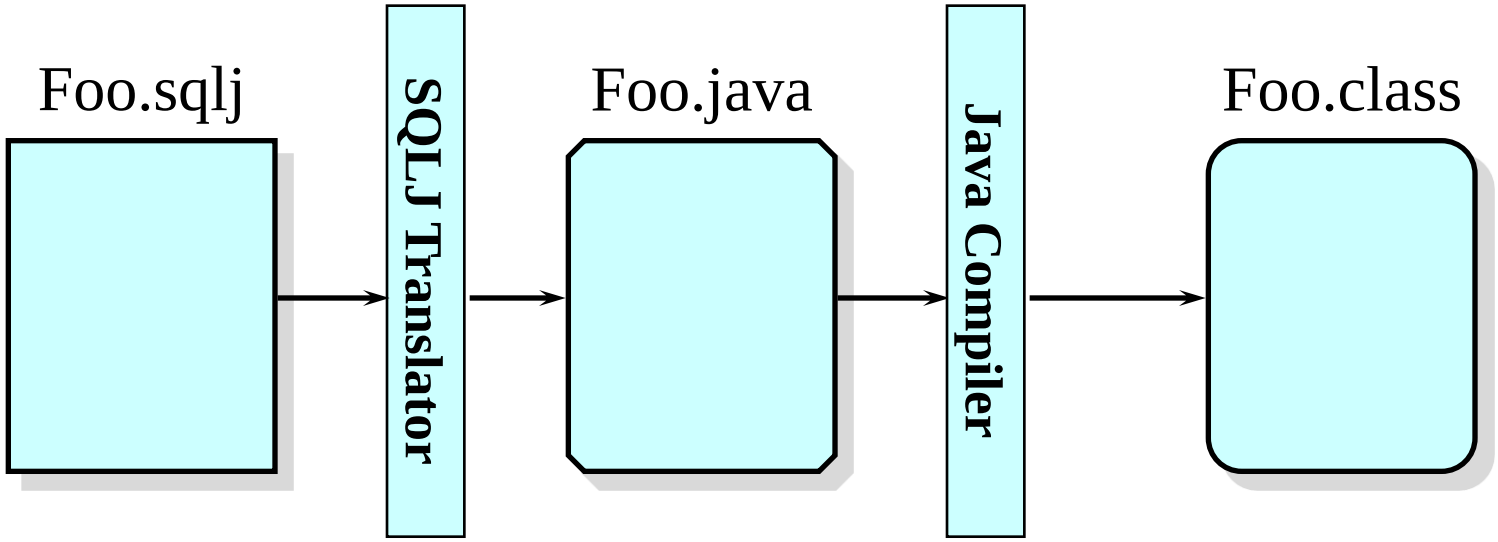
Ziel: Überprüfung der Korrektheit von SQL-Anweisungen bezüglich Relationendefinitionen.

- Naiver Ansatz:
  Jeder Übersetzerlauf macht Datenbank-Zugriff.
- Ansatz 2:
  Kein Datenbank-Zugriff, keine Analyse.
- Ansatz hier: Bekanntgabe existierender Daten.

Nachteile des ersten Ansatzes

- Relativ unpraktisch.
  (Bei jeder Übersetzung DB-Zugriff erforderlich.)
  Alternative: Schema ist im Cache, dafür braucht Übersetzer aber Verbindungsinformationen etc.
- Man kann nur übersetzen,
  wenn man mit der DB verbunden ist
  (oder Schema ist im Cache).
- Keine Garantie, dass sich Schema
  zur Laufzeit nicht geändert hat.

SQLJ – Ablauf



- SQLJ Übersetzer ersetzt eingebettete SQL Statements mit Aufrufen der SQLJ Runtime Library.
- Dann Übersetzung mit herkömmlichem Java Compiler.
- Laufzeitumgebung – ruft JDBC Treiber auf.

SQLJ: Embedded SQL für Java

- Vorübersetzung des erweiterten Quelltextes in echten Java-Code durch Übersetzer **sqlj**.
- Überprüfung der SQL-Anweisungen:
  - korrekte Syntax,
  - Übereinstimmung der Anweisungen mit DB-Schema,
  - Typkompatibilität der für Datenaustausch genutzten Variablen.
- Nutzung von JDBC-Treibern.

SQLJ – Vorteile

Insbesondere im Vergleich zu JDBC:

- Kompakter Code, verglichen mit JDBC,
- Syntax- und Semantik-Check der SQL-Statements zur Übersetzungszeit, gut für Performance.

Connection Contexts

- Default Connection, keine explizite Referenzierung nötig:
  ```
  Oracle.connect("jdbc:oracle:thin@MYSERVER:
  1521:ORCL", "klemens", "kbpasswd");
  ```
- Mehrere Connections mit unterschiedlichen Bezeichnungen:
  - ```
    DefaultContext myContext1 = Oracle.getConnection
    ("jdbc:oracle:oci8@MYSERVER_ORCL",
    "klemens", "kbpasswd");
    ```
  - ```
    #sql [myContext1] { SQL statement };
    ```
  - ```
    { #sql [myContext1] { commit };
    myContext1.close();} ...
    ```
    ```

Iteratoren

- Typisierung – sowohl Attributnamen als auch Typ.
- Implementierung des Cursor-Konzeptes.

1. Deklaration des Iterators:

```
#sql public iterator BookIter
      (String titel, double preis);
```

2. Definition des Iteratorobjektes, Instanz dieser Klasse:

```
BookIter iter;
```

3. Iterator populieren:

```
#sql iter = { SELECT titel, preis
      FROM buch };
```

4. Navigation, wie gehabt:

```
while (iter.next ()) {
      System.out.println (iter.titel () +
            " " + iter.preis ());
}
```

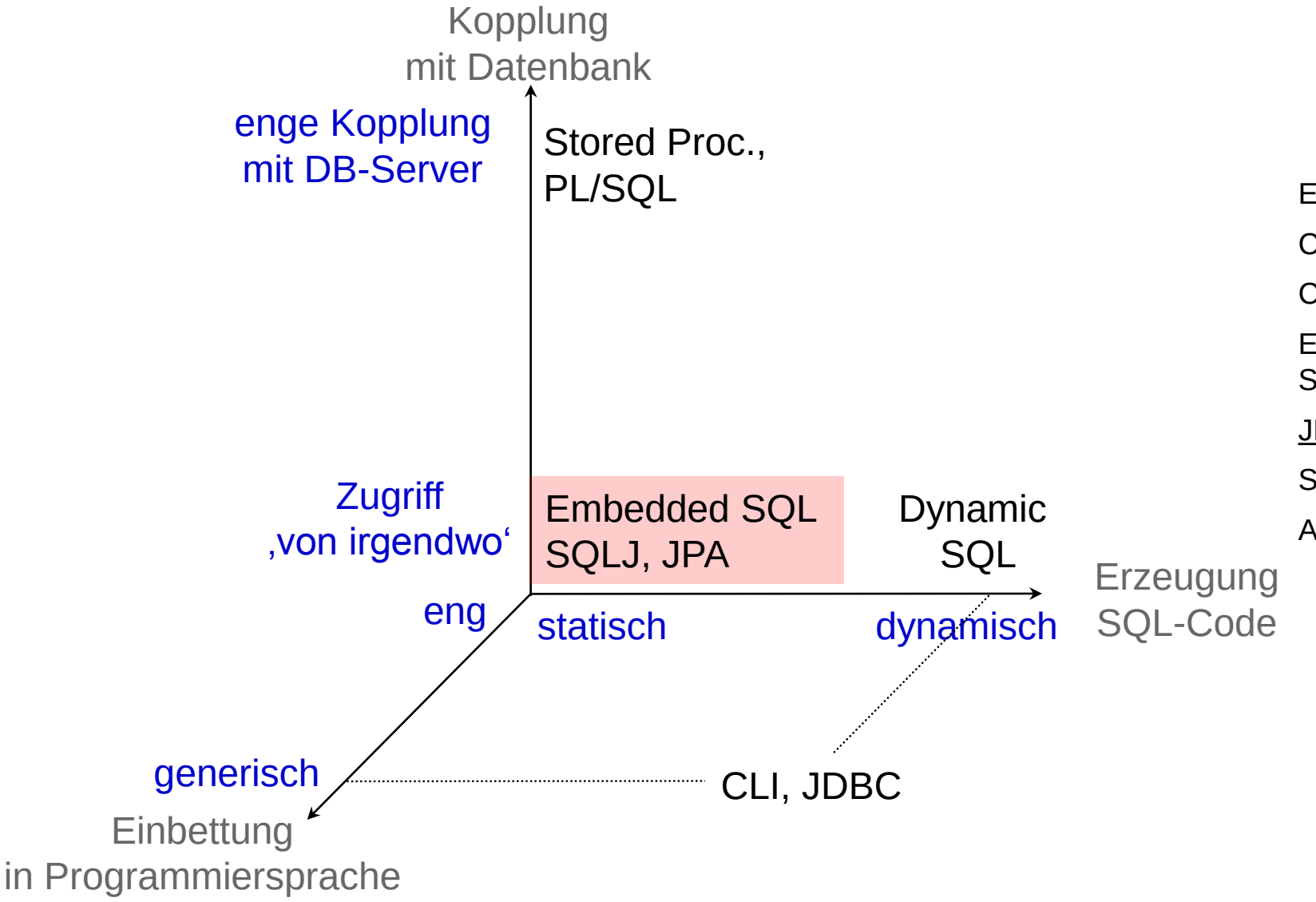
Performance Enhancements SQLJ

- *Statement Caching* –
  SQLJ cacht die letzten k Statements
  pro Connection. Nicht: ‚die letzten k Resultate‘.
- *Batching* – DML Statements mit gleicher Struktur werden gesammelt und gemeinsam ausgeführt.
  Über API kontrollierbar.
- *Row Prefetching*
  - normalerweise ein Nachrichtenpaar
    pro Ergebniszeile,
  - Prefetching fasst mehrere Zeilen zusammen;
    Anzahl Zeilen über API einstellbar.

Embedded SQL – Fazit

- Hat sich nicht wirklich durchgesetzt
  und ist nicht mehr unbedingt Stand der Kunst.
- Ein Problem: Präcompiling.
  Präcompiler oft nicht miteinander kombinierbar.

Programmiersprachenanbindung



Einleitung

Cursor

CLI

Embedded SQL

JPA

Stored Proc.

Antipatterns

JPA

- *Java Persistence API*
- Ziel: Objekte (in Programmiersprache, Java) persistent machen.
- Einfachste Variante: Elemente einer Klasse werden Tupel einer Relation.
  Umfangreiche Konfigurationsmöglichkeiten, insbesondere der physischen Repräsentation.
- Festlegung z. B. der zu persistierenden Attribute und des Schlüssels u. a.
  - über Annotationen des Java-Codes
  - oder in separaten XML-Dateien.
- Derzeit wohl Stand der Kunst.
  Zahlreiche Ausprägungen, z. B. Hibernate, EclipseLink; diverse ‚Konkurrenten', z. B. JDO (Java Data Objects).

JDO – Beispiel

```
Transaction tx=pm.currentTransaction();
try
{
    tx.begin();
    Inventory inv = new Inventory("My Inventory");
    Product product = new Product("Sony Discman",
            "A standard discman from Sony", 49.99);
    inv.getProducts().add(product);
    pm.makePersistent(inv);
    tx.commit();
}
finally
{
    if (tx.isActive())
    {
        tx.rollback();
    }
    pm.close();
}
```


Quellenangabe hinten.

JDO Beispiel – Erläuterung

- Was zeigt das Beispiel?
  - Persistierung erfolgt nicht automatisch.
  - Einbettung in Transaktion, auch Teil des Codes.
  - Persistierung mit Mitteln der Programmiersprache.
- Abhängige Objekte
  können ebenfalls persistent gemacht werden.

JDO – weiteres Beispiel

```
Transaction tx = pm.currentTransaction();
try
{
    tx.begin();
    Query q = pm.newQuery("SELECT FROM " +
            Product.class.getName() +
                        " WHERE price < 150.00
                       ORDER BY price ASC");
    List<Product> products = (List<Product>)q.execute();
    Iterator<Product> iter = products.iterator();
    while (iter.hasNext())
    {
        Product p = iter.next();
        ... (use the retrieved objects)
    }
    tx.commit();
}
finally
{ if (tx.isActive() { tx.rollback(); }
    pm.close();}
```

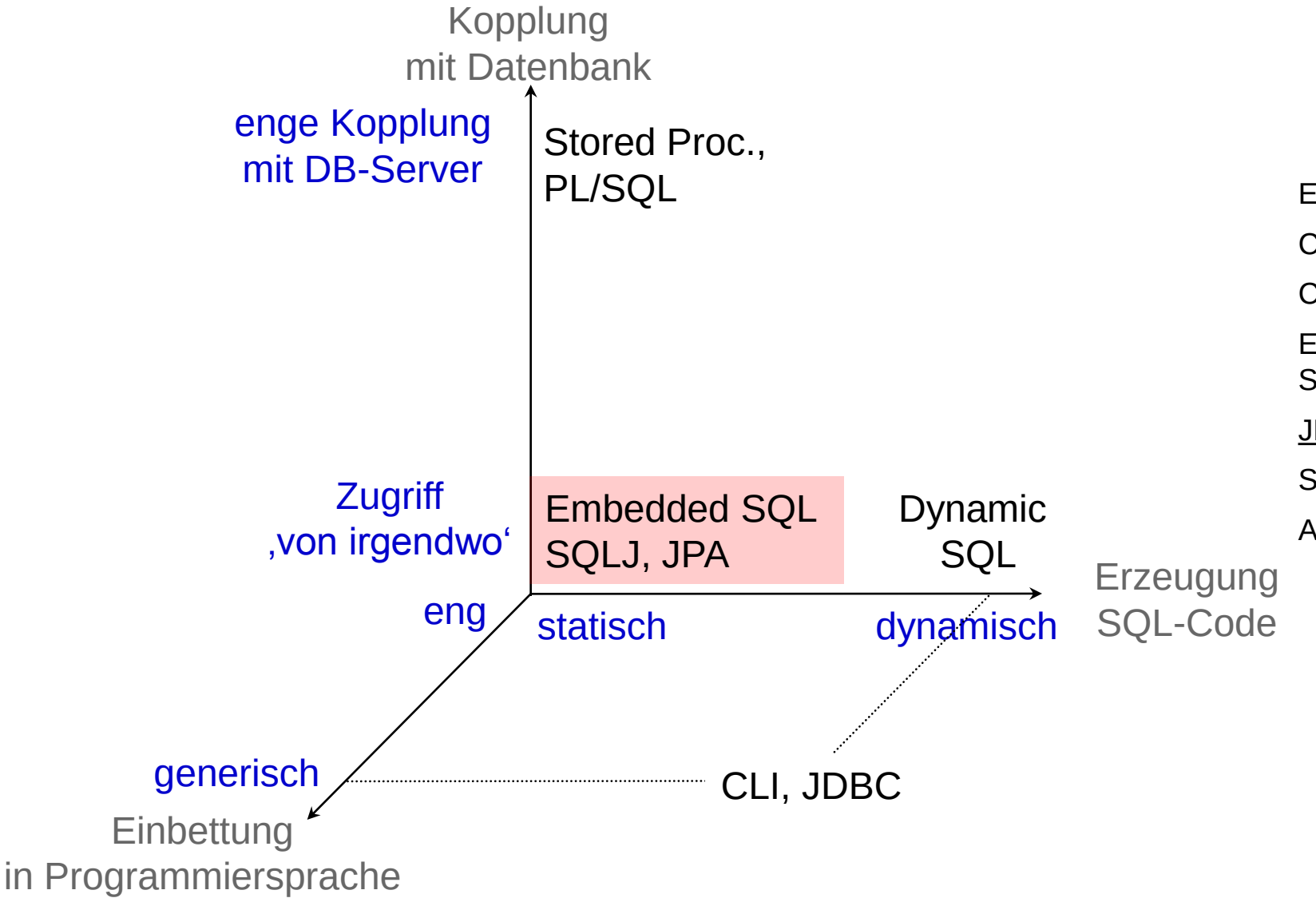
Weiteres Beispiel – Erläuterung

- Was zeigt dieses Beispiel?
  - Lesender Zugriff u. a. möglich mit deklarativer Anfrage.
  - SQL als Anfragesprache.
    Löschen oder Updates in gleicher Weise möglich.
  - Einbettung höherwertig ist als CLI.
    (‚`Product.class.getName()`‘ besser,
    als dass man Relationennamen kennen muss.)
  - Datenbankkonzepte
    sind in der Programmierumgebung bekannt.
    Genauer gesagt, Prinzip der Abbildung ist dort bekannt.
- Alternativen (Hibernate) mit Anfragesprache für Objekte.

Schlussbemerkung zu JPA/JDO/Hibernate

- Hier sehr oberflächliche Darstellung,
  die die vielfältigen Konfigurationsmöglichkeiten unterschlägt.
  Z. B.:
  - Unterstützung speziell für OO-Features, z. B. Vererbung.
  - Persistente vs. entkoppelte Objekte.
  - Caching.

Programmiersprachenanbindung



Einleitung

Cursor

CLI

Embedded SQL

JPA

Stored Proc.

Antipatterns

JPA – Diskussion

- statisch – diese Zuordnung ist nicht so eindeutig.
- „Zugriff ‚von irgendwo‘“,
  da Teil der Entwicklung der Anwendung,
  die irgendwo laufen kann.
  (I. d. R. auf Application Server.)
- „enge Einbettung in Programmiersprache“
  – nutzt offensichtlich Eigenheiten von Java
  (insbesondere Objektorientierung,
  z. B. aber auch generischen Listentyp).

Gespeicherte Prozeduren (1)

- Ein Problem von CLI und Embedded SQL:
  - Ständiger Wechsel der Ausführungskontrolle zwischen Anwendung und DBS; *Kontextwechsel*.
  - Anweisungsübergreifende Optimierung kann nur schwer automatisch stattfinden.

Gespeicherte Prozeduren (3)

- Lösung – *gespeicherte Prozeduren* („SQL um Ablaufkonstrukte erweitern“):
  - Im Datenbank-Server verwaltete und auch dort ausgeführte Software-Module in Form von Prozeduren bzw. Funktionen.
  - Aufruf aus Anwendungen und Anfragen heraus.

Vorteile Gespeicherter Prozeduren (1)

- Zentrale Kontrolle der Prozeduren:
  Redundanzfreie Darstellung
  relevanter Aspekte der Anwendungsfunktionalität.
- Prozeduren nur vom DBMS abhängig
  und nicht von externen Programmiersprachen
  oder Betriebssystemumgebungen,
  wenn entsprechende Standardisierung gegeben.
  (Vielleicht kein _Vorteil_, ist aber wesentliche Eigenschaft.)
- Strukturierungsmittel für größere Anwendungen.
  (Vielleicht nicht kriegsentscheidend, geht auch Client-seitig.)

Vorteile Gespeicherter Prozeduren (2)

- *Optimierung* der Prozeduren.
  Beispiel:
  - Abfolge von Anfragen:
    - `SELECT * FROM Person WHERE Salary > 500.000`
    - `SELECT * FROM Person WHERE Salary > 550.000`
  - Wohl effizienter:
    - `SELECT * FROM Person WHERE Salary > 550.000`
    - `SELECT * FROM Person WHERE Salary > 500.000 and Salary <= 550.000`
  - Gespeicherte Prozedur ermöglicht mehr Transparenz.
- Ausführung der Prozeduren unter Kontrolle des DBMS.
  (Keine ständigen Kontextwechsel.)
- Rechtevergabe für Prozeduren.
- In der Integritätssicherung: Aktionsteil von Triggern.

Variablen und Typen

- Informationsaustausch zwischen PL/SQL-Programm und Datenbank durch Variablen, getypt.
- Erstes Beispiel für Variablendeklaration:

```
DECLARE
    preis NUMBER;
    meinBier VARCHAR(20);
```

- Elegante Möglichkeit, um sicherzustellen, dass Attributtyp in Datenbank und Variablen-Typ in PL/SQL-Programm identisch – Illustration:

```
DECLARE
    meinBier Biere.name%Type;
```

- Variable mit Record-Typ, bestehend aus mehreren getypten Feldern – Beispiel:

```
DECLARE
    bierTupel Biere%ROWTYPE;
```

PL/SQL Programm – Beispiel

```
CREATE TABLE T1(
      e INTEGER,
      f INTEGER
);
DELETE FROM T1;
INSERT INTO T1 VALUES(1, 3);
INSERT INTO T1 VALUES(2, 4);

/* Bis hierher SQL; jetzt PL/SQL Programm. */

DECLARE
      a NUMBER;
      b NUMBER;
BEGIN
      SELECT e,f INTO a,b FROM T1 WHERE e>1;
      INSERT INTO T1 VALUES(b,a);
END;
.
run;
```

Kontrollfluss (1)

- Kanonisch.
- Beispiele:
  -
```
DECLARE
      a NUMBER;
      b NUMBER;
BEGIN
      SELECT e,f INTO a,b
      FROM T1 WHERE e>1;
      IF b=1 THEN
         INSERT INTO T1 VALUES(b,a);
      ELSE
         INSERT INTO T1 VALUES(b+10,a+10);
      END IF;
END;
.
run;
```

Kontrollfluss (2)

- Beispiele (Fortsetzung):
  - ```
    DECLARE
        i NUMBER := 1;
    BEGIN
        LOOP
            INSERT INTO T1 VALUES(i,i);
            i := i+1;
            EXIT WHEN i>100;
        END LOOP;
    END;
    .
    run;
    ```
    ```

Cursors

- Tupel an der aktuellen Cursor-Position kann i. Allg. auch modifiziert und gelöscht werden.
- Beispiel:

```
DECLARE
  a T1.e%TYPE;  b T1.f%TYPE;
  CURSOR T1Cursor IS
  SELECT e, f FROM T1 WHERE e < f FOR UPDATE;
BEGIN
  OPEN T1Cursor;
  LOOP
    FETCH T1Cursor INTO a, b;
    EXIT WHEN T1Cursor%NOTFOUND;
    DELETE FROM T1 WHERE CURRENT OF T1Cursor;
            /* Insert the reverse tuple: */
    INSERT INTO T1 VALUES(b, a);
    END LOOP;  /* Free cursor used by the query. */
  CLOSE T1Cursor;
END;
.
run;
```

```
[SQL> CREATE TABLE t1(e integer, f integer);

 Tabelle wurde erstellt.

[SQL> DELETE FROM t1;

 0 Zeilen wurden geloscht.

[SQL> INSERT INTO t1 VALUES(1,3);

 1 Zeile wurde erstellt.

[SQL> INSERT INTO t1 VALUES(2,4);

 1 Zeile wurde erstellt.

[SQL> DECLARE
[  2  a NUMBER;
[  3  b NUMBER;
[  4  BEGIN
[  5  SELECT e,f INTO a,b FROM t1 WHERE e>1;
[  6  INSERT INTO t1 VALUES(b,a);
[  7  END;
[  8  .
[SQL> RUN;
   1  DECLARE
   2  a NUMBER;
   3  b NUMBER;
   4  BEGIN
   5  SELECT e,f INTO a,b FROM t1 WHERE e>1;
   6  INSERT INTO t1 VALUES(b,a);
   7* END;

 PL/SQL-Prozedur erfolgreich abgeschlossen.

[SQL> SELECT * FROM t1;

          E          F
 ---------- ----------
          1          3
          2          4
          4          2

 SQL>
```

Einleitung

Cursor

CLI

Embedded SQL

JPA

Stored Proc.

- Einleitung

- PL/SQL

Antipatterns

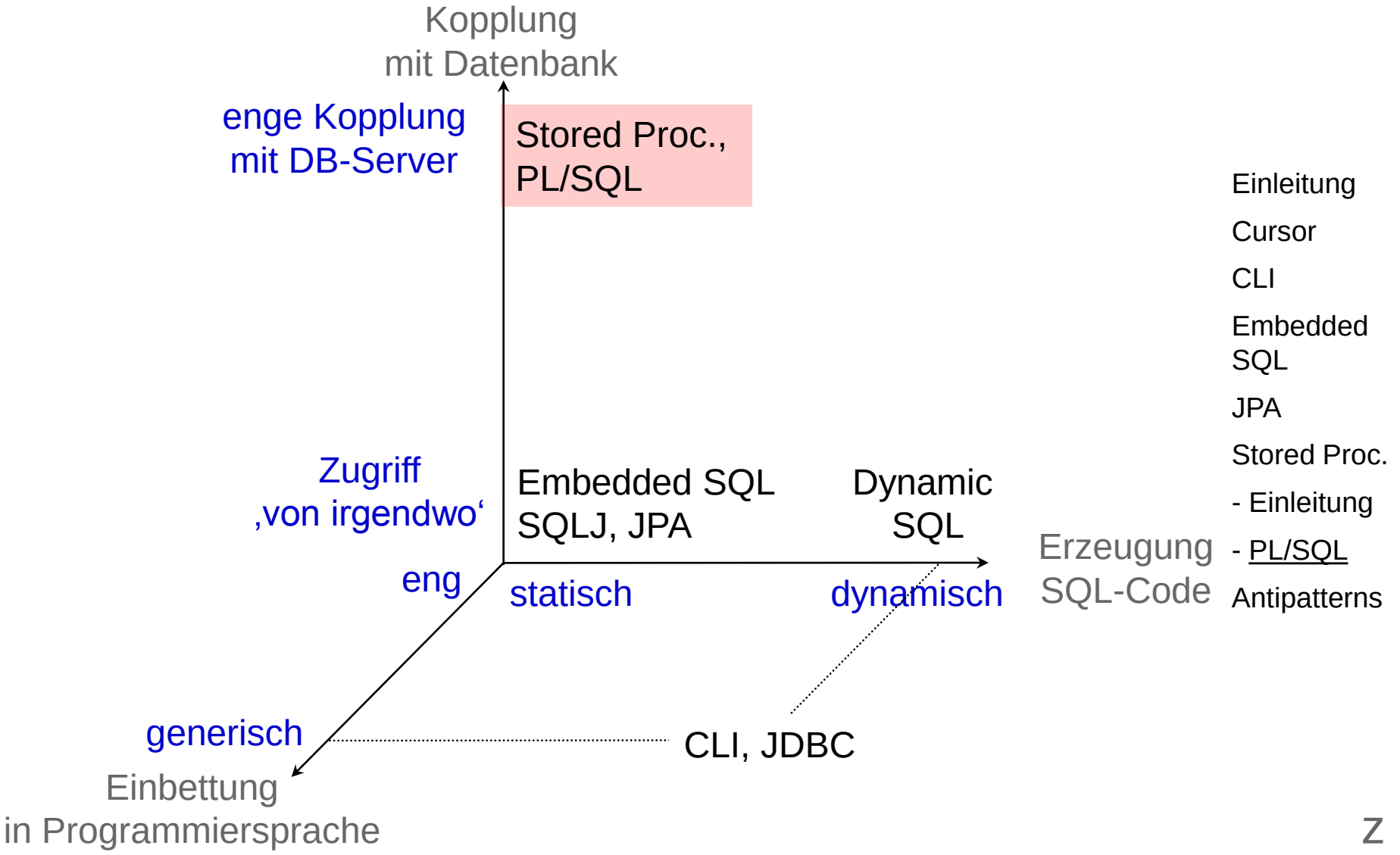
Stored Procedures + JDBC

- Stored Procedures –
  auch mit Hilfe von JDBC erzeugbar und aufrufbar.
- Erzeugung – Beispiel:
  - `String createProcedure = "create procedure SHOW_SUPPLIERS " + "as " + "SELECT SUPPLIERS.SUP_NAME, COFFEES.COF_NAME " + "FROM SUPPLIERS, COFFEES " + "WHERE SUPPLIERS.SUP_ID = COFFEES.SUP_ID " + "ORDER BY SUP_NAME";`
  - `Statement stmt = con.createStatement();`
  - `stmt.executeUpdate(createProcedure);`
  - `CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");`
  - `ResultSet rs = cs.executeQuery();`
    Hier `executeQuery`, bei Updates `executeUpdate`,
    bei Kombination `execute`
    (letzteres umständlich in der Handhabung).

Stored Procedures – weitere Aspekte

- Außerdem: Objektorientierte Strukturierungsmöglichkeiten.
- Stored Procedures insgesamt also deutlich leistungsfähiger als imperative SQL-Scriptsprache.
- In PostgreSQL serverseitig z. B. auch C-Code verwendbar, in Oracle Java. Nachteil i. Allg.: produktspezifisch.

Programmiersprachenanbindung



Performance von Anfragen und Anwendungen

Thema im Folgenden, kurz:

- Was kann Anwendungsentwickler tun
  für gute Performance seiner Anfragen?
- Nicht: Performance i. Allg./Tuning von Datenbanken.
  (Das wäre zu umfangreich.
  Z. T. Thema der LV „Datenhaltung in der Cloud“.)
- Vermeiden bestimmter Muster in Anfragen
  („Performance Anti-Patterns“).
- Thema dieses Kapitels,
  weil Anbindung der Anwendung manchmal relevant.
- Muster sind i. Allg. weniger offensichtlich
  als in anschaulicher Darstellung, die folgt.

Performance Anti-Patterns – Übersicht

- Excessive Dynamic Allocation
- The Stifle
- Circuitous Treasure Hunt
- Sisyphus Database Retrieval
- Spaghetti Query
- Insufficient Caching
- Wrong Caching Strategy

Excessive Dynamic Allocation (1)

- Problem: „frequent, unnecessary creation and destruction of objects of the same class“.
- Z. B. Datenbankverbindung, Threads.
- Offensichtliche Lösung:
  Wiederholte Nutzung der gleichen Ressource.

Excessive Dynamic Allocation (2)

```
-- Connection im Schleifenrumpf öffnen/schließen:

for (int i = 1; i <= 100; i++) {

        Class.forName("oracle.jdbc.driver.OracleDriver");
                //Treiber laden
        Connection con = DriverManager.getConnection
                        ("jdbc:oracle:thin:@host:port:sid",
                         "username", "passwort");
        Statement stmtSQL = con.createStatement();

        String sql = "SELECT page_title FROM wikipc
                                    WHERE count_views ="+ i;
        ResultSet result = stmtSQL.executeQuery(sql);
        //..

        stmtSQL.close();
        con.close();
}
```

Excessive Dynamic Allocation (3)

```
-- Connection außerhalb von Schleife öffnen/schließen:

Class.forName("oracle.jdbc.driver.OracleDriver");
        //Treiber laden
Connection con = DriverManager.getConnection
        ("jdbc:oracle:thin:@host:port:sid",
         "username", "passwort");
Statement stmtSQL = con.createStatement();

for (int i = 1; i <= 100; i++) {

        String sql = "SELECT page_title FROM wikipc
                        WHERE count_views ="+ i;
        ResultSet result = stmtSQL.executeQuery(sql);
        //..
}
stmtSQL.close();
con.close();
```

Excessive Dynamic Allocation (4)

- Zeitmessungen (jeweils in ms):

```
-- Connection
-- im Schleifenrumpf
-- öffnen/schließen:
13241
11871
10004
13226
9616
10164
11510
11086
9671
11003
AVG: 10139
```

```
-- Connection
-- außerhalb von Schleife
-- öffnen/schließen:
4690
4775
7145
4797
4702
4737
4717
4761
7307
4824
AVG: 5246
```

The Stifle

- Problem:
  Unpassende Nutzung der Datenbankschnittstelle.
- Lässt sich illustrieren
  anhand des vorangegangenen Beispiels
  (bessere Variante).

Excessive Dynamic Allocation (3)

– Folie von eben –

```
-- Connection außerhalb von Schleife öffnen/schließen:

Class.forName("oracle.jdbc.driver.OracleDriver");
        //Treiber laden
Connection con = DriverManager.getConnection
        ("jdbc:oracle:thin:@host:port:sid",
         "username", "passwort");
Statement stmtSQL = con.createStatement();

for (int i = 1; i <= 100; i++) {

        String sql = "SELECT page_title FROM wikipc
                        WHERE count_views ="+ i;
        ResultSet result = stmtSQL.executeQuery(sql);
        //..
}
stmtSQL.close();
con.close();
```

- Was geschieht hier?
- Was wäre bessere Vorgehensweise?

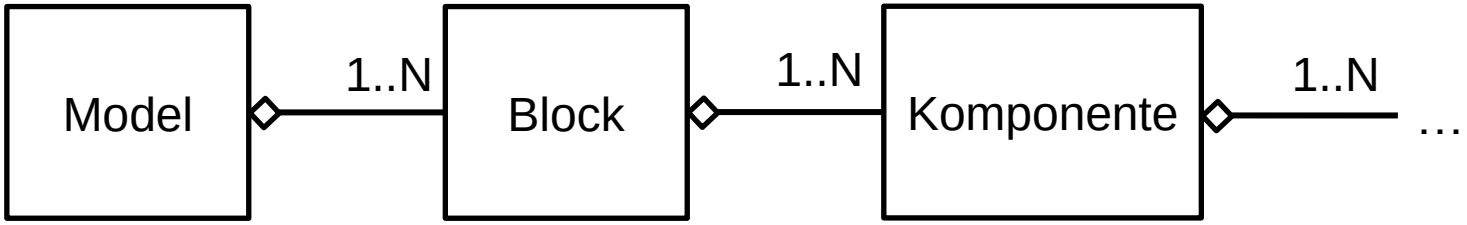
The Stifle

- Problem tritt durchaus auf
  mit ORM (object-relational mapper, z. B. Hibernate),
  d. h. bei automatischer Generierung der Anfrage.
- Eine SQL-Anfrage.

```
SELECT page_title, count_views FROM wikipc
                WHERE count_views > 0 AND count_views < 101;
```

Circuitous Treasure Hunt (1)

- Muster: Programm fragt Daten von Relation A ab, nutzt diese, um Relation B nach Daten abzufragen, nutzt diese, um Tabelle C abzufragen usw., bis endgültige Ergebnisse abgefragt werden.
- Beispiel: Gegeben eine Instanz von Model, finde dazu alle Komponenten, usw.




```
SELECT *
FROM Model
WHERE name = "c"
```

```
SELECT *
FROM Block
WHERE m_id = 5
```

```
foreach Block i {
    SELECT Koordinate
    FROM Komponente
    WHERE b_id = i.id
}
…
```

Circuitous Treasure Hunt (2)

- Ziel: Vermeidung von remote calls.
  Ideal wäre ein Datenbankaufruf.
- Lösungen:
  - Auf Modellierungsebene:
    - Zusätzliches (redundantes) Attribut für Koordinaten bei 'model' anbringen.
    - Dann reicht select auf entsprechender Relation.
  - Auf Datenbankebene:
    - Sicht anstelle dieser (abhängigen) Anfragen.
      (Oder auch Skript.)
    - Vorschlag aus Literatur:
      Nutzung des ‚Adapter'-Musters.
      (Führt mehrere Queries aus
      und gibt nur Endergebnis zurück.)
      In technischer Hinsicht (Anzahl der remote calls)
      mit Sicht vergleichbar.

Sisyphus Database Retrieval (1)

- Beispiel:

Adressbuch

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Aachen, Anton
Aaron, Tanya
Berger, Johannes
Berthold, Paul
Christian, Dieter
Christian, Mareike
Dietrich, Richard
Ernst, Thomas

Seite 1 | Seite 2 | Seite 3 | … | Seite 100

- Muster: Abfragen einer riesigen Datenmenge, obwohl Anwendung nur kleine Teilmenge nutzt.
- Fortsetzung Beispiel: `SELECT * FROM kontakte`

Sisyphus Database Retrieval (2)

- Mögliche Lösungen:
  - Untere/obere Grenze hartkodieren

```
SELECT *
FROM kontakte
WHERE name < "Ernst" and name > "Berthold"
```

Sisyphus Database Retrieval (3)

- Mögliche Lösungen (Forts.):
  - Index und Rownum
    - Beispiel:

```
SELECT *
    FROM
(SELECT *
     FROM kontakte
  ORDER BY nachname)
WHERE rownum <= 10;
```

    - Anmerkungen:
      - Rownum ist Oracle-Primitiv.
        Andere Datenbanken haben Vergleichbares.
      - Top-N Anfrage
        – Pagination mit rownum ist komplizierter.
        Geht aber.
      - nachname-Werte sollten einmalig sein.

Spaghetti Query (1)

- Problem:
  Mehrere Informationsbedürfnisse unnötigerweise in einer Anfrage.
  - Fehleranfälligkeit.
  - Unnötige Berechnungen.
- Quasi Gegenstück zu The Stifle.

Spaghetti Query (2)

- Beispiel:
  - „I need to know how many products we work on,
    how many developers fixed bugs,
    the average bugs fixed per developer,
    and how many of our fixed bugs were reported by customers.“
  - Code, nicht richtig!

```
SELECT count(bp.product_id) as how_many_products,
count(dev.account_id) as how_many_developers,
count(b.bug_id)/count(dev.account_id)
                          as avg_bugs_per_developer,
count(cust.account_id) as how_many_customers
FROM  Bugs b join BugsProducts bp on (b.bug_id = bp.bug_id)
join Accounts dev on (b.assigned_to = dev.account_id)
join Accounts cust on (b.reported_by = cust.account_id)
WHERE  cust.email not like '%@example.com'
GROUP BY bp.product_id;
```

Quellenangabe hinten.

Spaghetti Query (3)

- Beispiel (Forts.):
  - „I need to know how many products we work on,
    how many developers fixed bugs,
    the average bugs fixed per developer,
    and how many of our fixed bugs were reported by customers.“
  - Relationen entsprechen also:
    product, developer, bug, customer,
    mit erwarteten Attributen.
  - Erstes Informationsbedürfnis: Kein Join erforderlich.
  - Andere Informationsbedürfnisse brauchen ‚product‘ nicht.

Spaghetti Query (4)

- Beispiel (Forts.):
  - Unnötiger Join, der außerdem Ergebnis falsch macht.
  - Stattdessen mehrere separate Anfragen.
    - Anzahl Produkte:
      ```
      SELECT count(*) as how_many_products FROM Products;
      ```
    - Anzahl Entwickler, die …:
      ```
      SELECT count(distinct assigned_to)
                   as how_many_developers
      FROM Bugs
      WHERE status = 'FIXED';
      ```
    - Durchschnittliche Anzahl Fehler pro Entwickler:
      ```
      SELECT avg(bugs_per_developer)
                   as average_bugs_per_developer
      FROM (SELECT dev.account_id, count(*)
                   as bugs_per_developer
            FROM Bugs b join Accounts dev
              on (b.assigned_to = dev.account_id)
            WHERE b.status = 'FIXED'
            GROUP BY dev.account_id) t;
      ```

Spaghetti Query (5)

- Beispiel (Forts.):
  - Stattdessen mehrere separate Anfragen (Forts.).
    - Anzahl Fehler, die Kunden auch gemeldet haben:

```
SELECT count(*) as how_many_customer_bugs
FROM Bugs b join Accounts cust
      on (b.reported_by = cust.account_id)
WHERE b.status = 'FIXED'
      and cust.email not like '%@example.com';
```

  - Außerdem Query/Sicht,
    wenn alle Zahlen in einem Tupel sein müssen.

Insufficient Caching (1)

- Caching – Ziel: Reduzierung der Anzahl zeitaufwendiger Datenbankzugriffe.
- Unterschiedliche Caches in komplexen Web-Applikationen.

Insufficient Caching (2)

- Unterschiedliche Caches – Hibernate als Beispiel:
  - First Level Cache. Für jede Session separat.
    Ziel in etwa:
    „Weniger Datenbankzugriffe in einer Transaktion.“
  - Second Level Cache:
    - Transaktionsübergreifendes Caching.
    - Inhalt dieses Caches: Objekte der Anwendung.
      Zugriff über Schlüssel.
  - Query Cache:
    - Hibernate verfügt über Querysprache, HQL.
      Unterscheidet sich von SQL durch Ebene der Anwendung.
      Z. B. Klassen der Anwendung (im Gegensatz zu Relationen)
      Gegenstand des Zugriffs.
    - Query Cache enthält Ergebnisse von HQL-Anfragen.
    - Genauer: Nur Schlüssel der Objekte.
      Objekte selbst in Second Level Cache.

Insufficient Caching (3)

- Größe der Caches üblicherweise konfigurierbar.
  Je nach Anwendungsfall erhebliche Verlangsamung durch schlechte Konfiguration.

Wrong Cache Strategy

- Welche Objekte werden in Cache abgelegt
  bzw., wenn Cache voll, aus ihm verdrängt?
- Unterschiedliche Strategien wie LFU, LRU.
- I. Allg. beeinflussbar,
  mit eventuell erheblichem Einfluss auf Performance.
- Z. B. bei Hibernate.
  Für DB-Cache von Oracle jedoch nicht.

Schlussbemerkungen

- Zusammenspiel von Programmen mit Datenbanken fundamental wichtig.
- Unterschiedliche Schnittstellen hierfür.
- Unsachgemäße Nutzung verschlechtert Performance.
- Antipatterns in diesem Kontext sind in der Wirklichkeit durchaus häufig.

Prüfungsfragen, beispielhaft

- Erläutern Sie die Dimensionen des Raums der Möglichkeiten des Zugriffs auf Datenbanken aus Anwendungen heraus.
- Erläutern Sie die folgenden Begriffe:
  - Anwendungslogik,
  - Cursor,
  - Call-Level Interface,
  - Host Variablen
- Kann man mit Embedded SQL sicherstellen, dass keine Schema-spezifischen Fehler auftreten? Wenn ja, wie geht es?
- Was sind die Vorteile von Stored Procedures? Erläutern Sie das Konzept.

Literaturhinweise

- http://www.datanucleus.org/products/accessplatform/jdo/samples/tutorial_rdbms.html
- B. Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. 1st Pragmatic Bookshelf. 2010.