Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme I

Wintersemester 2015/2016

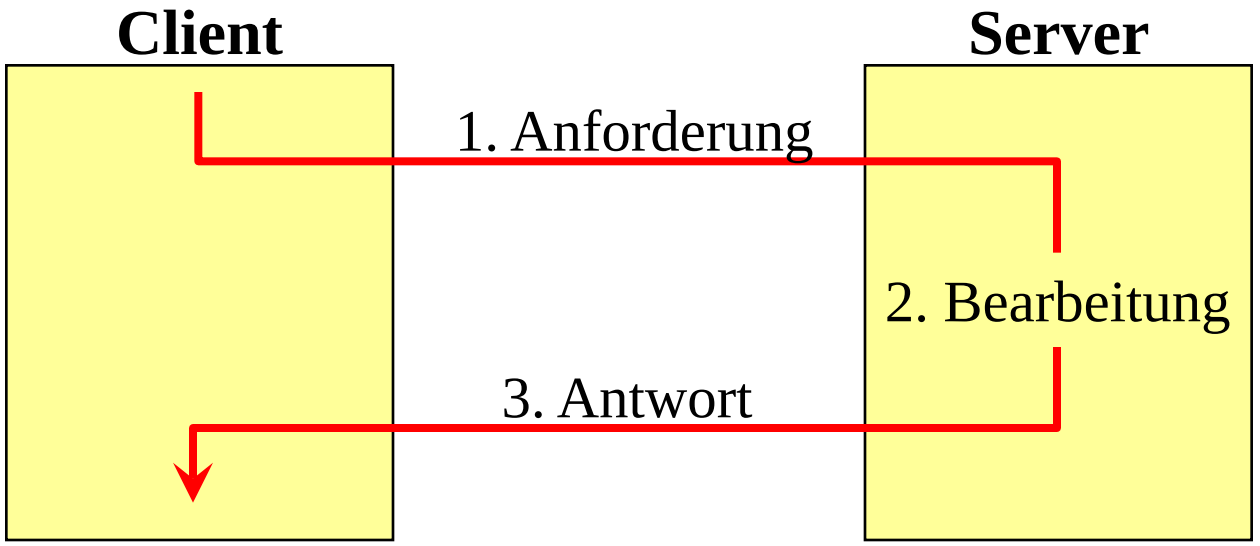
Kapitel 11: Anwendungsentwicklung

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Sebastian Goebl

http://www.dbs.ifi.lmu.de/Lehre/DBS

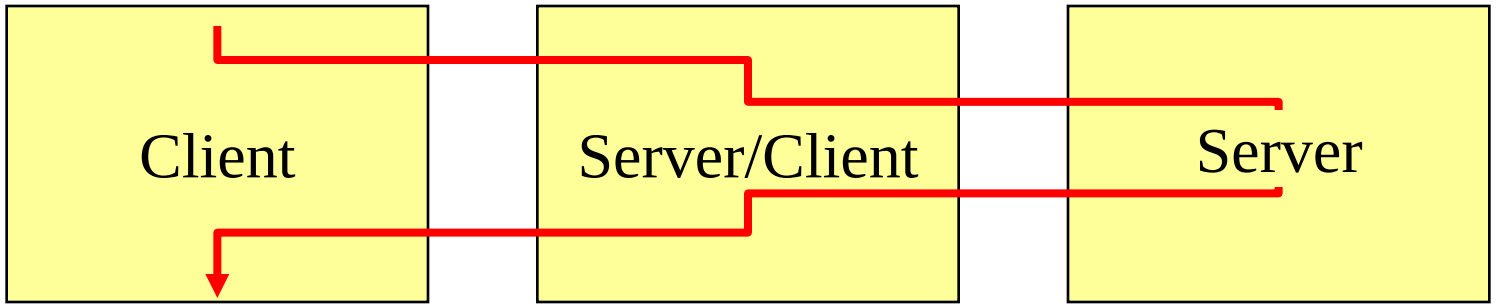
Client-Server-Architektur

- Datenbankanwendungen unterstützen gleichzeitige Arbeit von vielen Benutzern
- Dienstnehmer (Client) nimmt Dienste eines Dienstleisters (Server) zur Erfüllung seiner Aufgaben in Anspruch:



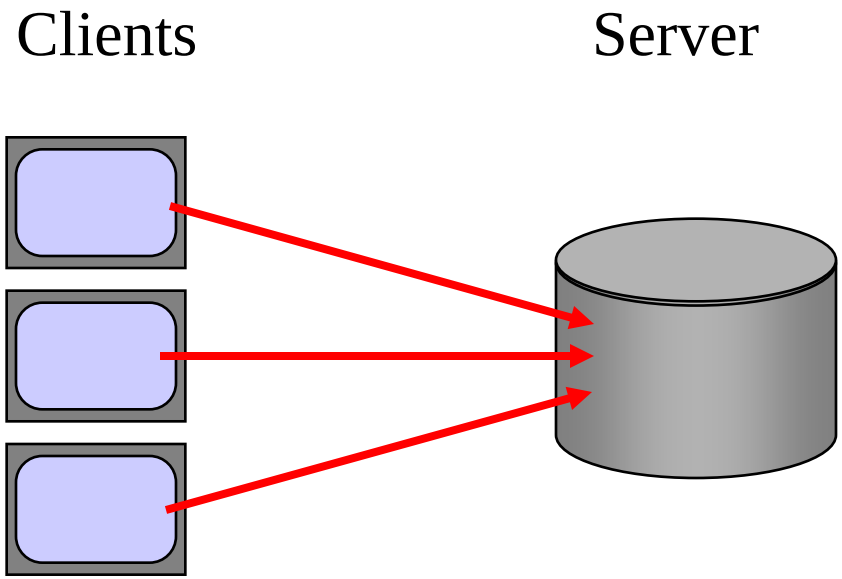
Client-Server-Architektur

- Für jeden Server bekannt, welche Dienste er erbringen kann
- Protokoll:
  Regelt Interaktionen (d.h. Aufbau der Anforderung und der Antwort)
- Asymmetrische Beziehung zw. Client und Server
  - Möglich, daß ein Client seinerseits wieder Server für einen anderen Dienst ist:



DB-Server

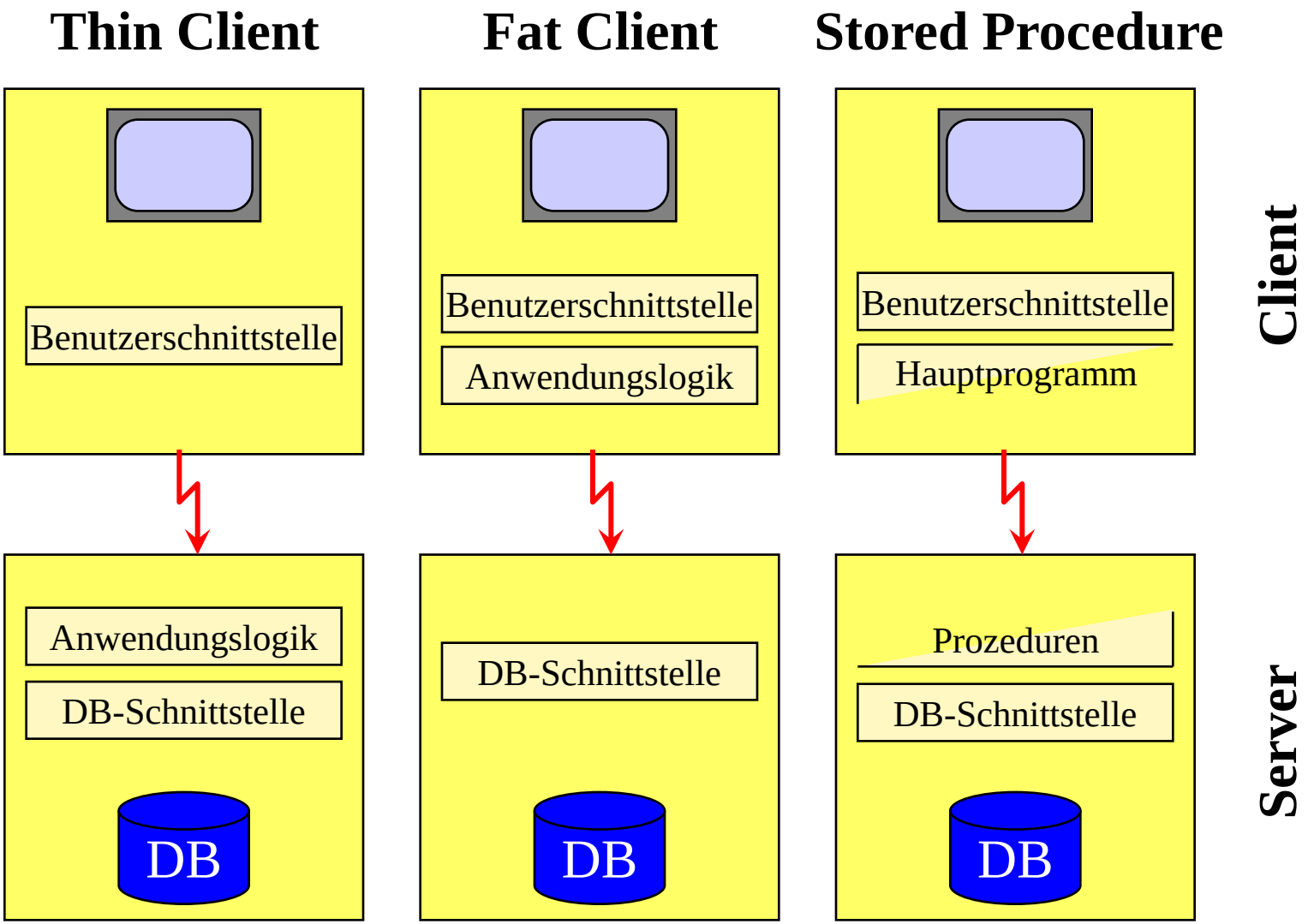
- Datenbankanwendungen:
  - Mehrere Benutzer arbeiten auf gemeinsamen Daten
  - Zentrale Kontrollinstanz ist erforderlich
  - PCs sind Frontends zur Darstellung und benutzerfreundlichen Manipulation der Daten



Aufteilung der Funktionalität

- Funktionsgruppen von Datenbankanwendungen:
  - Präsentation und Benutzerinteraktion
  - Anwendungslogik (im eigentlichen Sinn)
  - Datenmanagement (Anfragebearbeitung)
- Trennung an jeder beliebigen Stelle:
  - Präsentation und Interaktion meist vollst. im Client
  - Datenmanagement meist vollständig im Server
  - Anwendungslogik:
    - meist im Client
    - im Server: typ. Großrechner-Anwendungen
    - Aufteilung der Logik: „gespeicherte Prozeduren“

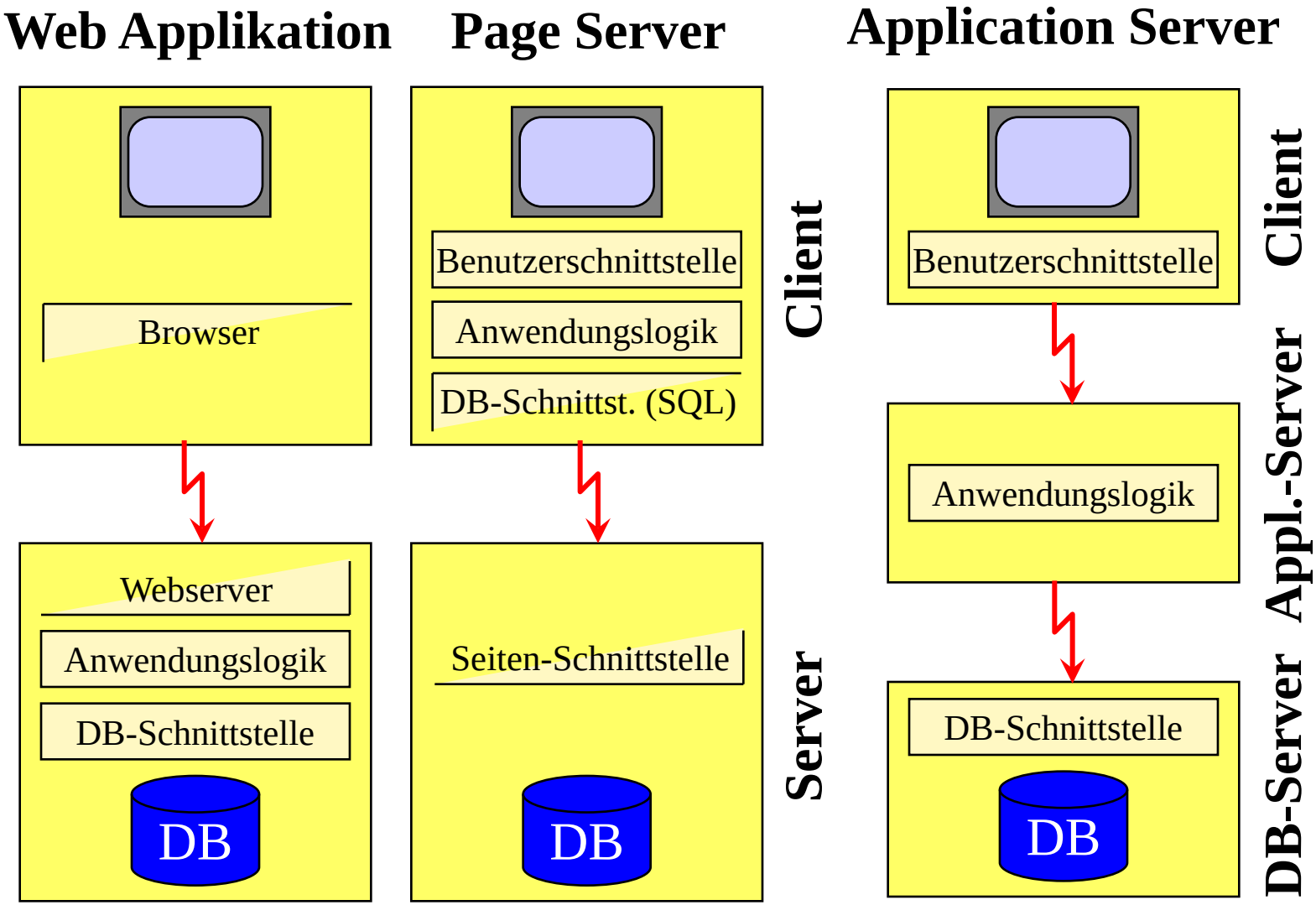
Aufteilung der Funktionalität



Aufteilung der Funktionalität

- Trennung zwischen Client und Server auch innerhalb einer Funktionsgruppe möglich:
  - Web-Anwendung:
    Server erbringt teilweise Präsentationsfunktionalität
    Client hat nur Standard-Darstellungsfunktionalität
  - Seiten-Server:
    Teil der Anfragefunktionalität (SQL) liegt im Client
    Server hat nur die Funktionalität, Datenblöcke gem. Adresse zu speichern und zu laden
- 3-stufige Hierarchie:
  - Applikationslogik bildet eigenes Client-Server-Modell
  - Applikations-Server ist Datenbank-Client

Aufteilung der Funktionalität



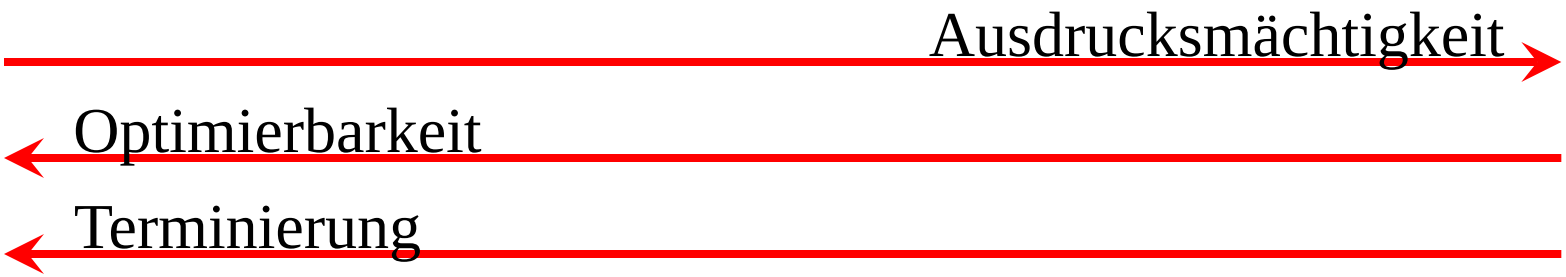
Protokoll

- Client und Server können
  - auf dem selben Rechner laufen
  - auf verschiedenen Rechnern laufen, die über ein Netzwerk miteinander kommunizieren
- Die Kommunikation zwischen Client und Server wird in jedem Fall durch ein **Protokoll** geregelt:
  - Bei Web-Anwendungen: **http**
  - A.-Logik/DB-Schnittstelle: Spez. Protokoll des DBMS
    - Anforderung: im wesentlichen SQL-Anfrage
    - Antwort: Tabelle bzw. Liste von Tupeln
  - Bei Applikationsservern: Standard-Protokolle wie RMI, CORBA

Vor-/Nachteile der Varianten

- Kriterien:
  - Netzbelastung
  - Belastung des Server-Rechners (gemeinsame Ressource für alle)
  - Belastung der Clients-Rechner (oft schlecht ausgestattete Bürorechner)
  - Flexibilität
  - Entwicklungsaufwand

SQL ↔ Java



- SQL: keine Schleifen, etc.
- Anwendungslogik:
  Konventionelle Programmiersprache nötig

- SQL wird mit herkömmlicher Programmiersprache (Hostsprache) gekoppelt

Kopplung SQL/Host-Sprache

- Idee:
  - Java-Programm (z.B.) enthält SQL-Anfragen
  - Diese werden an DB-Server gesandt
  - Server schickt als Antwort Ergebnistabelle
  - Java-Programm verarbeitet diese Tabelle:
    - Darstellung der Tupel am Bildschirm
    - ggf. weitere Anforderungen an die DB
- Probleme:
  - Wie können die SQL-Anfragen in das Programm integriert werden (verschiedene Sprachen)
  - Unterschiedliche Datenstruktur-Konzepte
    - Java kann nicht Tabellen verarbeiten

Integration der Anfragen

Grundsätzlich zwei verschiedene Möglichkeiten:

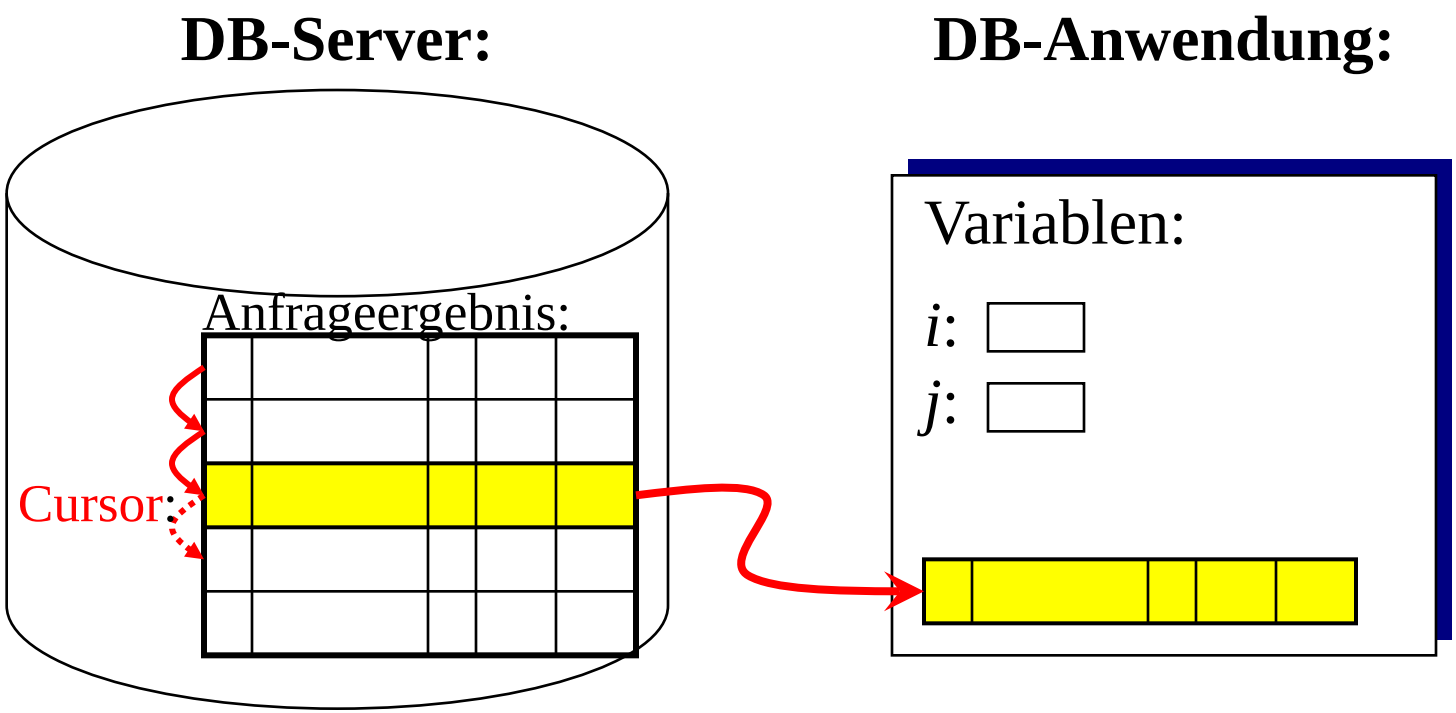
- Call-Level-Schnittstellen:
  Es gibt eine Bibliothek mit speziellen Prozeduren, die die Anfrage als Parameter enthalten:
  - CLI, ODBC, OCI, JDBC:
    Statement stmt = con.createStatement () ;
    ResultSet rs = stmt.executeQuery (“select * from ...“) ;
- Einbettung (Embedded SQL):
  Ein spezielles Schlüsselwort oder Zeichen kennzeichnet SQL-Anfrage (Vorübersetzung):
  - Embedded SQL-C (und versch. andere), JSQL, SQLJ:
    #sql {insert into Mitarbeiter values (25, “Meier“, 1) } ;

Das Cursor-Konzept

- In beiden Fällen nicht möglich/sinnvoll, sofort die gesamte Ergebnis-Tabelle an das Hostprogramm zu übergeben:
  - Nicht möglich wegen sog. Impedance Mismatch:
    - SQL beruht auf der Datenstruktur „Tabelle“
    - Java beruht auf der Datenstruktur „Datensatz“ (bzw. Tupel, Klasse)
  - Nicht sinnvoll, weil Ergebnis sehr groß werden kann:
    - Client-Rechner oft schwach ausgestattet
    - Übertragung des gesamten Ergebnisses kostet Zeit, in der der Benutzer warten muß (erste Ergebnisse)
    - Anwendungsprogramm braucht oft nur einen Teil der Ergebnistupel

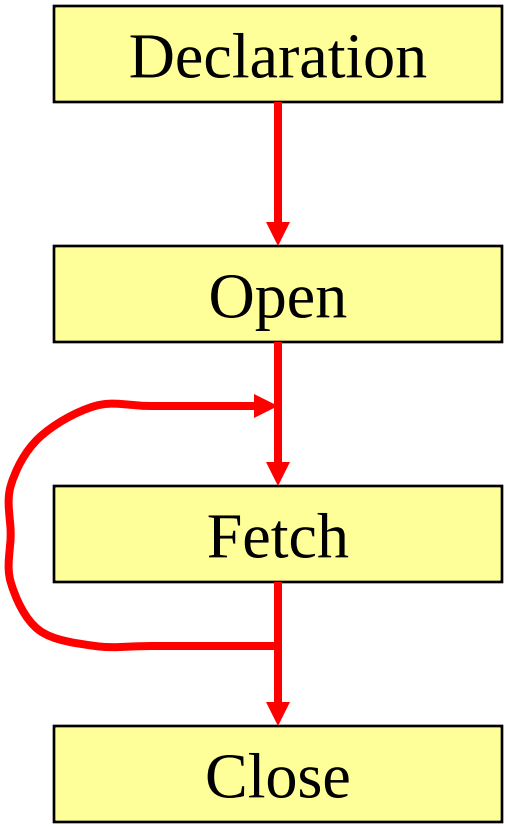
Das Cursor-Konzept

- Deshalb wird die Ergebnistabelle grundsätzlich tupelweise an das Hostprogramm übergeben.
- Cursor: Position des aktuellen Tupels



Programmierschritte beim Cursor

- Grundsätzlich sind 4 Schritte erforderlich:



Verknüpfung des Cursors mit einer bestimmten SQL-Anfrage

Auftrag an den DB-Server, die Anfrage zu bearbeiten

Verarbeitung der Ergebnistupel durch das Anwendungsprogramm

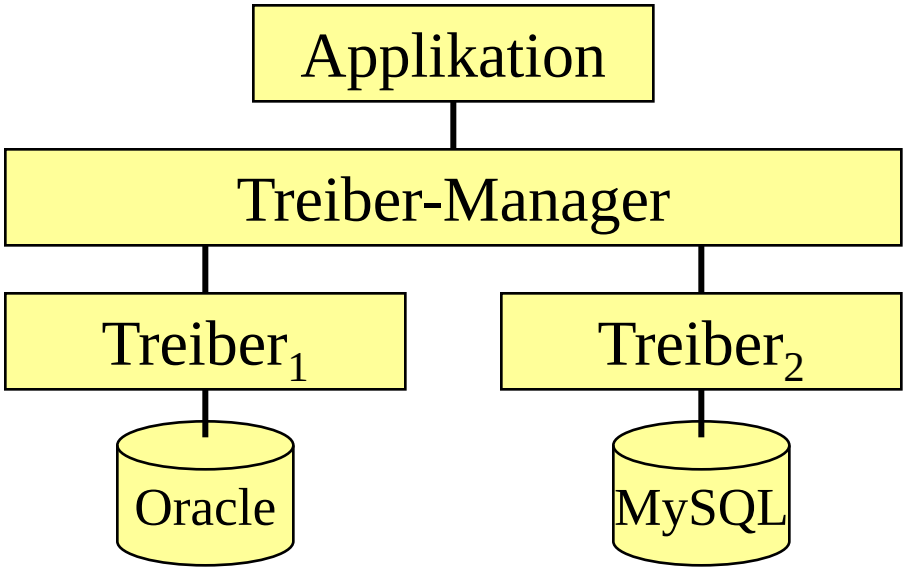
Freigabe von Ressourcen (z.B. Ergebnistabelle in der DB)

Weitere Aufgaben

- Über das reine Absetzen von Anfragen hinaus sind weitere Interaktionen mit dem DBS nötig:
  - Verwalten der Verbindung:
    - Verbindungsaufbau zu einer oder mehreren DBs
    - Verbindungsabbau
    - Authentifizierung mit Benutzername/Passwort
  - Management von Transaktionen
    - commit
    - rollback
  - Fehlerbehandlung
  - Aufruf von gespeicherten Prozeduren sowie deren Verwaltung

Java Database Connectivity JDBC

- Call Level Interface (Bibliotheksfunktionen)
- Eng verwandt mit Microsofts ODBC
  - ähnliches Konzept für verschiedene Wirtssprachen
- Idee:
  Hersteller-Unabhängigkeit durch Treiber



Verbindungsaufbau

- Der prinzipielle Ablauf einer DB-Anwendung umfasst die folgenden Einzelschritte:
  - Laden des geeigneten Treibers
  - Aufbau einer Verbindung zur Datenbank
  - Senden einer SQL-Anweisung
  - Verarbeiten der Anfrageergebnisse
- Diesen Schritten sind jeweils Java-Klassen zugeordnet:
  - `java.sql.DriverManager`
  - `java.sql.Connection`
  - `java.sql.Statement`
  - `java.sql.ResultSet`

Treiber und Verbindungsaufbau

- Es muß also java.sql.* importiert werden.
- Laden des Datenbank-Treibers für Oracle:
  Class.forName (“oracle.jdbc.driver.OracleDriver“) ;
- Einrichten einer Verbindung durch Treiber-Mgr:

  Connection con = DriverManager.getConnection (
  “jdbc:oracle:oci8:@dbis01“, “scott“, “tiger“) ;

  richtet folgende Verbindung ein:
  - Datenbankname: dbis01
  - Benutzername: scott
  - Passwort: tiger
- Connection-Objekt (hier: *con*) ist Ausgangspunkt für alle weiteren Aktionen

Änderungs-Anweisungen

- Ausgangspunkt: das Connection-Objekt
- Zunächst muß ein Statement-Objekt erzeugt werden (dient für Verwaltungszwecke)
- insert, delete und update mittels der Methode executeUpdate:

```
Statement stmt = con.createStatement () ;
int rows = stmt.executeUpdate
        (“update Produkt set Bestand=5 where ProdID=3“) ;
```

Transaktionssteuerung:

- Nach Verbindungsherstellung Auto-Commit-Modus (jede Änderung wird ohne explizites Commit sofort permanent)
- später Näheres

Retrieval-Anweisungen

- Ausgangspunkt: Wieder Connection-Objekt
- Für Verwaltungszwecke: Statement-Objekt
- Ein Cursor wird deklariert *und* geöffnet
  durch die Methode executeQuery (query):
  Statement stmt = con.createStatement () ;
  ResultSet rs = stmt.executeQuery (“select * from Mitarbeiter“) ;
- Das Rückgabeobjekt (hier *rs*) vom Typ ResultSet ist aber noch nicht die Ergebnistabelle selbst sondern eine ID für den entsprechenden Cursor
- Um 1. (2., 3,...) Datensatz zu laden, Methode next:

```
while (rs.next ( ) ) {
        /* Datensatz verarbeiten */
}
```

false, wenn kein Datensatz (mehr) vorhanden ist

Schließen in JDBC implizit durch Garbage Collection

Retrieval-Anweisungen

- Zugriff auf die einzelnen Attribute durch Spalten-Indizes (beginnend bei 1) über die Funktionen:
  - getString (*i*)
  - getDouble (*i*)
  - getInt (*i*) usw.

  die einen Wert des entsprechenden Typs liefern.
- Beispiel:

```
Statement stmt = con.createStatement ( ) ;
ResultSet rs = stmt.executeQuery (“select * from Mitarbeiter“) ;
while (rs.next ( ) ) {
        int pnr = rs.getInt (1) ;
        String name = rs.getString (2) ;
        double gehalt = rs.getDouble (5) ;
        System.out.println (pnr + “ “ + name + “ “ + gehalt) ;
}
```

Fehlerbehandlung

- Datenbankanweisungen können fehlschlagen, z.B.
  - Connect mit einer nicht existenten Datenbank
  - Select auf eine nicht existente Tabelle usw.
- In Java generell mit folgendem Konstrukt:

```
try {
        // Aufruf von JDBC-Anweisungen,
        // die möglicherweise Fehler generieren
} catch (SQLException exc) {
        // Fehlerbehandlung, z.B.
        System.out.println (exc.getMessage ( ) ) ;
}
```

Vorübersetzte SQL-Anfragen

- Bisher:
  Jede Anfrage wird als String an DBMS übergeben
  - Vorteil: Flexibel
  - Nachteil: Jedes mal Übersetzungsaufwand
- Werden oft ähnliche Anfragen gestellt, empfiehlt sich, auf wiederholte Übersetzung zu verzichten
  - Parametrisierbarkeit ist wichtig

```
PreparedStatement stmt = con.prepareStatement
        ( " insert into Mitarbeiter values (?, ?, ?, NULL, 0.0) " );
```

- Die einzelnen Platzhalter müssen vor dem Ausführen mit SetInt, SetString, ... besetzt werden
  - Spaltenindizes (beginnend bei 1)

Vorübersetzte SQL-Anfragen

- Beispiel:

```
PreparedStatement stmt = con.prepareStatement
        ( “ insert into Mitarbeiter values (?, ?, ?, NULL, 0.0) “ );
// erstes Insert:
stmt.setInt (1, 25) ;
stmt.SetString (2, “Wagner“) ;
stmt.SetString (3, “Richard“) ;
stmt.executeUpdate ( ) ;
// zweites Insert:
stmt.SetInt (1, 26) ;
...
stmt.executeUpdate ( ) ;
```