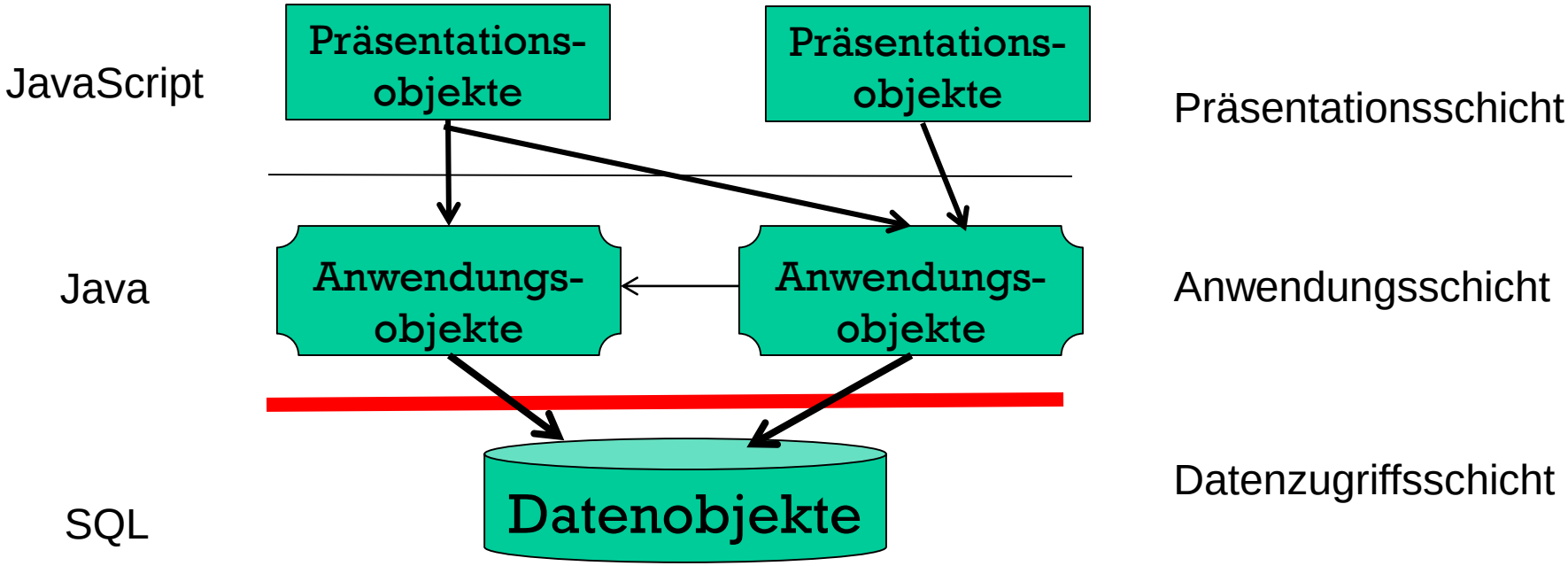
6. Anwendungsprogrammierung

- **Reale Problemstellungen**
  - Verwaltung persistenter Daten in Datenbanken
  - Entwicklung problemspezifischer Operationen
  - GUI-Entwicklung
  - ➔ Klassische 3-Schichten Architekturen



SQL

- Deskriptive Anfragesprache
  - sehr komfortabel für die Datenbankprogrammierung
    - Effiziente Verarbeitung von Mengen
  - Datenunabhängigkeit durch Sichten
- Wenig Funktionalitätsumfang für die “alltägliche” Programmierung
  - SQL ist nicht Turing-vollständig.
  - Einfache Aufgaben in nativem SQL sind nicht umsetzbar.
    - ➔ Benutzerdefinierte Funktionen und Operatoren
  - SQL hat ein restriktives Datenmodell.
    - Im Wesentlichen wird nur die Verarbeitung von Mengen (Relationen) unterstützt.

Programmiersprachen

- Anwendungen werden typischerweise in imperativen Sprachen entwickelt.
  - C, Java, C++, C#, JavaScript, Python, PHP, ….
- Objektorientiertes Typsystem
  - Anwendungsnahe Datenmodelle
- Reichhaltige Funktionalität vorhanden
  - Bibliotheken
- Schlechte Unterstützung von Persistenz
  - ➔ Wunsch: Zugriff auf vorhandene Datenbankfunktionalität
- Verarbeitung von einzelnen Datenobjekten

Impedance Mismatch

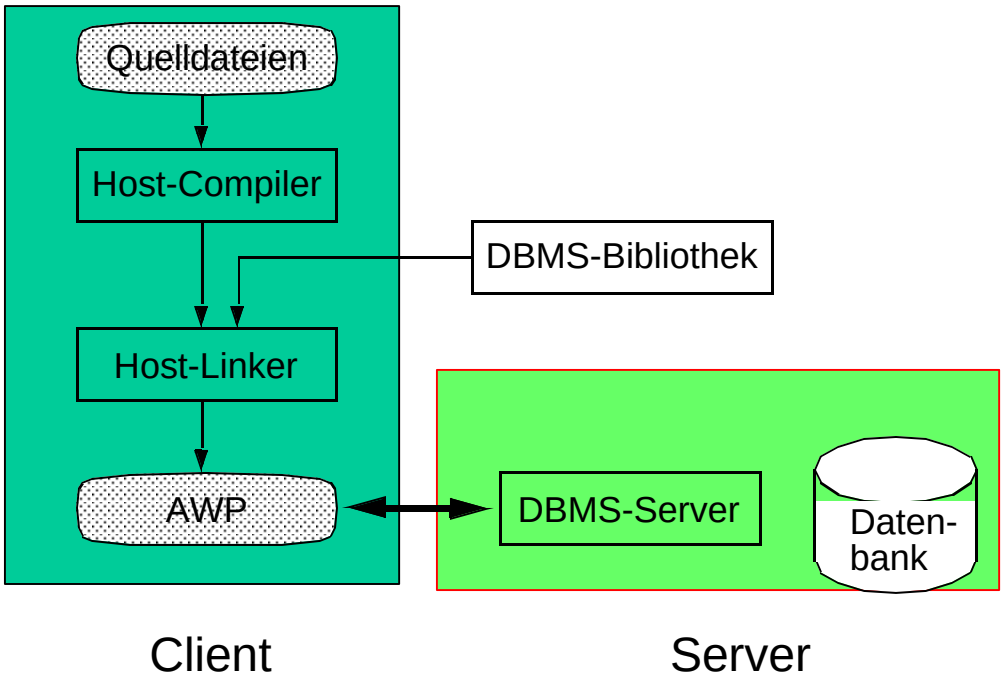
- **Klassischer Impedance Mismatch**
  - Mengenorientierte Verarbeitung in SQL
  - Verarbeitung einzelner Objekte in Java
- **Objektorientierter Impedance Mismatch**
  - Modellierung der Daten in Form von „flachen“ Relationen
  - Objektorientierte Sicht bzw. hierarchische Sicht auf die Daten in den Anwendungen
- **Überwindung dieser „Mismatchs“?**
  - Wie kann die Java und SQL miteinander verknüpft werden, um das Beste aus beiden Welten zur Verfügung zu stellen?

Kopplungsmöglichkeiten

- CALL-Schnittstelle
  - Bereitstellung von Bibliotheken (API) wie z. B. **JDBC**
- Einbettung von SQL in Java mit Vorübersetzer
  - Embedded SQL, wie z. B. **SQLJ**
- Spracherweiterungen
  - Erweiterung von SQL
    - imperative Konzepte, wie z. B. PL/SQL.
    - Unterstützung von JSON in Relationen.
  - Programmiersprachen
    - SQL-Erweiterungen: LINQ in C#
- Entwicklung von Persistenz-Frameworks
  - Hibernate
  - JPA

6.1 CALL-Schnittstelle

- **Ziel**
  - Überwindung des klassischen Mismatch: Mengenverarbeitung ←→ Objektverarbeitung
- **Bereitstellung in Form einer Bibliothek**
  - ODBC (C++), JDBC (Java), OCI (C)

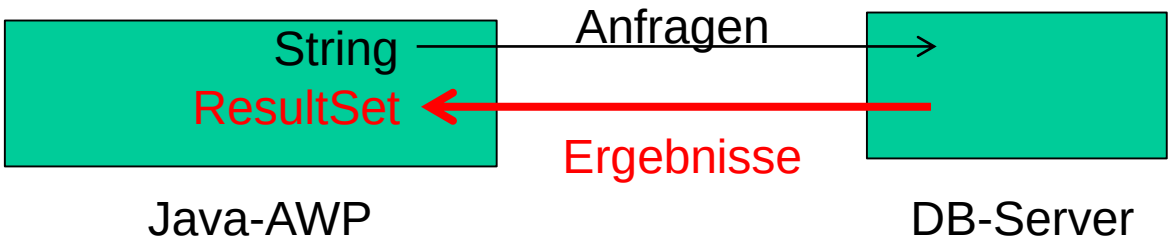


Komponenten der CALL-Schnittstelle

- **Gemeinsam vom AWP und dem DBMS-Server genutzte Datenstrukturen.**
  - Aufbau der Kommunikation
  - Bearbeitung einer Anfrage
    - AWP:
      Übertragung von SQL-Anfragen als Zeichenketten (String) an den DBMS-Server
    - DBMS-Server:
      Resultatsmenge der Anfrage werden in Form eines **Cursors (Iterator)** dem AWP bereitgestellt.
  - Direkte Kommunikation durch **gemeinsame Variablen** im AWP und dem DBMS-Server

JDBC

- **JDBC steht für Java Database Connectivity**
  - Java API für die Programmierung relationaler DBMS
    - Integraler Bestandteil der Standard-API (Paket java.sql)



    - Unterstützung des Client-Server Konzepts
      - DBMS läuft in einem anderem Prozess als das AWP
  - Wichtige Konzepte in JDBC
    - Verschiedene Typen von Anfragen
      - statische Anfragen, parametrisierbare Anfragen, …
      - Änderungsoperationen auf Relationen und dem Schema
      - Zugriff auf Metadaten

Beispiel

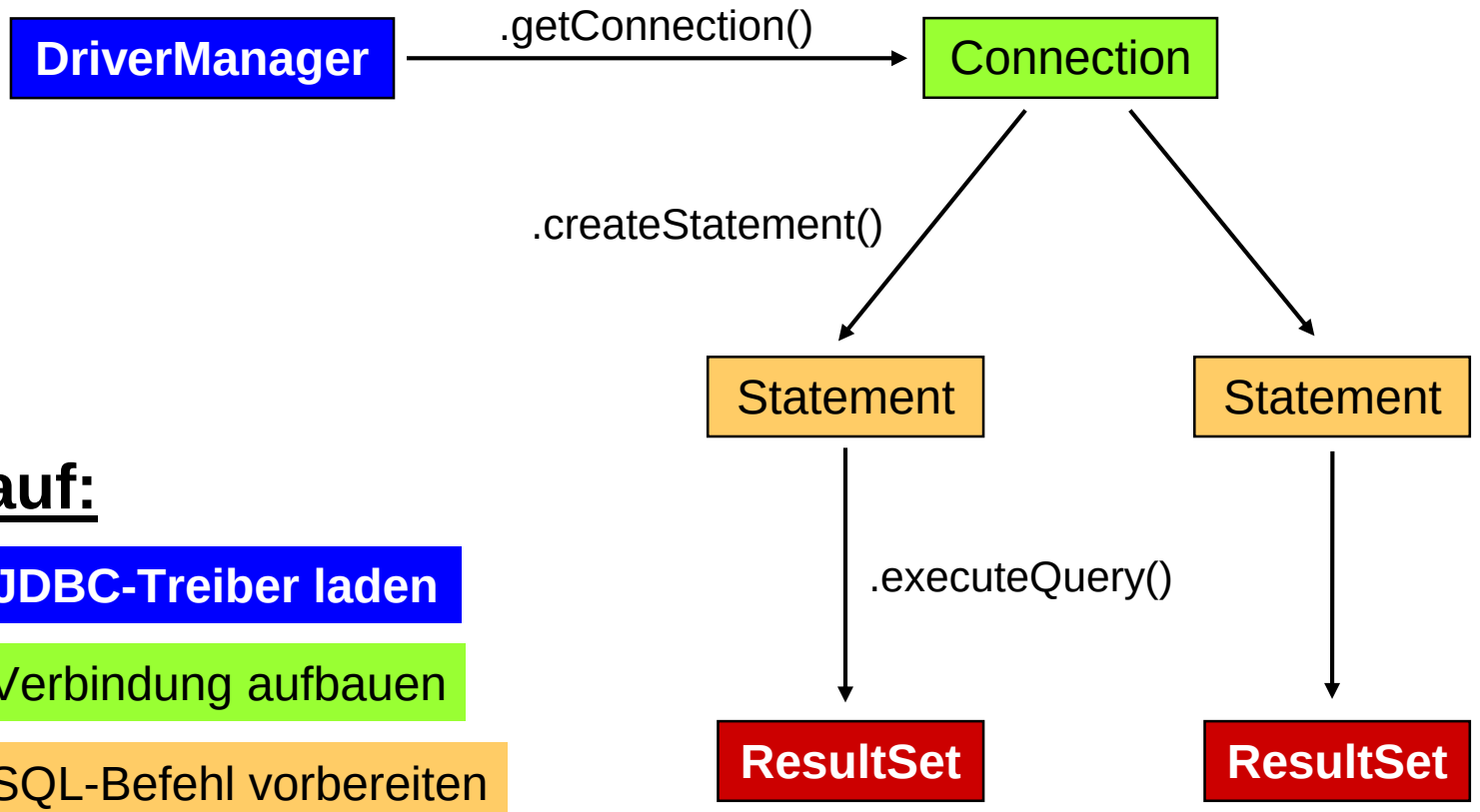
```
public static void main(String[] args) {
    try {
      // 1. Schritt: JDBC-Treiber registrieren
      Class.forName("org.postgresql.Driver");
      // 2. Schritt: Verbindung zum DBMS aufbauen
      Connection con = DriverManager.getConnection(
                                    "jdbc:postgresql://localhost/erp", "scott", "tiger");
      // 3. Schritt: Anfragen vorbereiten und ausführen
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("select * from PMZuteilung;");
      // 4. Schritt Ergebnisse abholen und verarbeiten
      while (rs.next()) {
        System.out.println(rs.getInt(1) + "  " + rs.getInt("mnr"));
      }
    } catch (Exception se) {
      System.out.println("SQL-Problem");
    }
  }
```

Ausnahmebehandlung in JDBC

- Wie üblich in Java werden ungewöhnliche Situationen durch Verwendung des Exception-Konzepts behandelt.
  - Siehe spezielle Klasse **SQLException**

- Zusätzlich können “unkritische” Fehler auch als Warnungen nach außen gegeben werden.
  - Objekte der Klasse **SQLWarning** und entsprechender Unterklassen

Struktur von JDBC



Arbeitsablauf:

(1) JDBC-Treiber laden

(2) Verbindung aufbauen

(3) SQL-Befehl vorbereiten

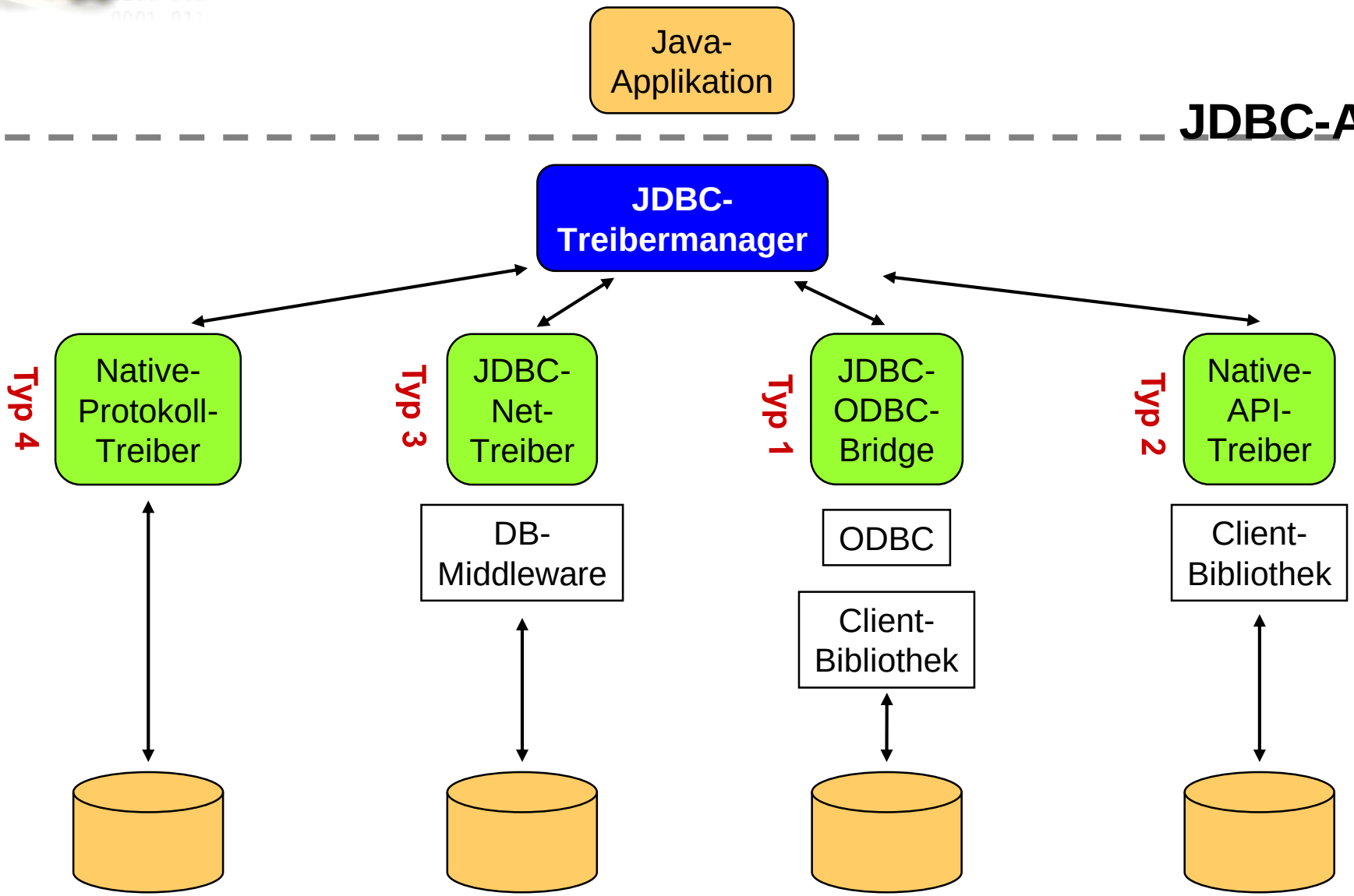
(4) SQL-Befehl ausführen

(5) ggfs. Ergebnis verarbeiten

6.1.1 Verbindungsaufbau in JDBC

- **Anmeldung der Treiberklasse**
  - Z. B. durch Class.forName("org.postgresql.Driver") ;
  - 4 verschiedene Arten von Treiberklassen bei JDBC
    - Native API-Treiber
    - JDBC-ODBC Bridge
    - JDBC-Middleware
    - Natives Protokoll
- **Erzeugung eines Connection-Objekts**

JDBC-Treiberarchitektur



Details zum JDBC-Treibermanager

- **Treibermanager**
  - Vermittlungsschicht zwischen Anwendung und dem eigentlichen Datenbanktreiber
  - Es gibt nur **einen** Treibermanager pro Programm
    ➔ nur statische Methoden in der Klasse DriverManager
  - Unterstützung folgender Operationen
    - Registrieren und Laden von Treibern
    - Herstellen einer Verbindung zur Datenbank
    - Konfiguration der Verbindung
  - Laden eines Treibers (mehrere Optionen)
    - Explizites Laden im AWP über Methode Class.forName
    - Über die Property jdbc.drivers der JVM

      java -Djdbc.drivers=foo.bas.Driver:wom.sql.Driver DBFoo
      - Programm DBFoo wird gestartet und in der Kommandozeile der verwendete Treiber mitgeteilt.

Funktionalität bei der Datenbankverbindung

- **Funktionalitätsumfang des Treibers**
  - Methode jdbcCompliant prüft, ob SQL92 Entry Level unterstützt wird.
- **Aufbau einer Datenbankverbindung**
  - Methode *getConnection aus der Klasse DriverManager.*
    - Input: eine JDBC-URL als Parameter
      - jdbc:<subprotocol>:<subname>
- **Weitere Funktionalität**
  - Setzen einer maximalen Wartezeit zur Anmeldung am Server
  - Protokollierung von Fehlermeldungen
  - …

Connection-Objekt

- Repräsentation einer Datenbankverbindungen
  - Objekt der Klasse Connection wird durch Aufruf der Methode getConnection erzeugt.
  - Beispiel

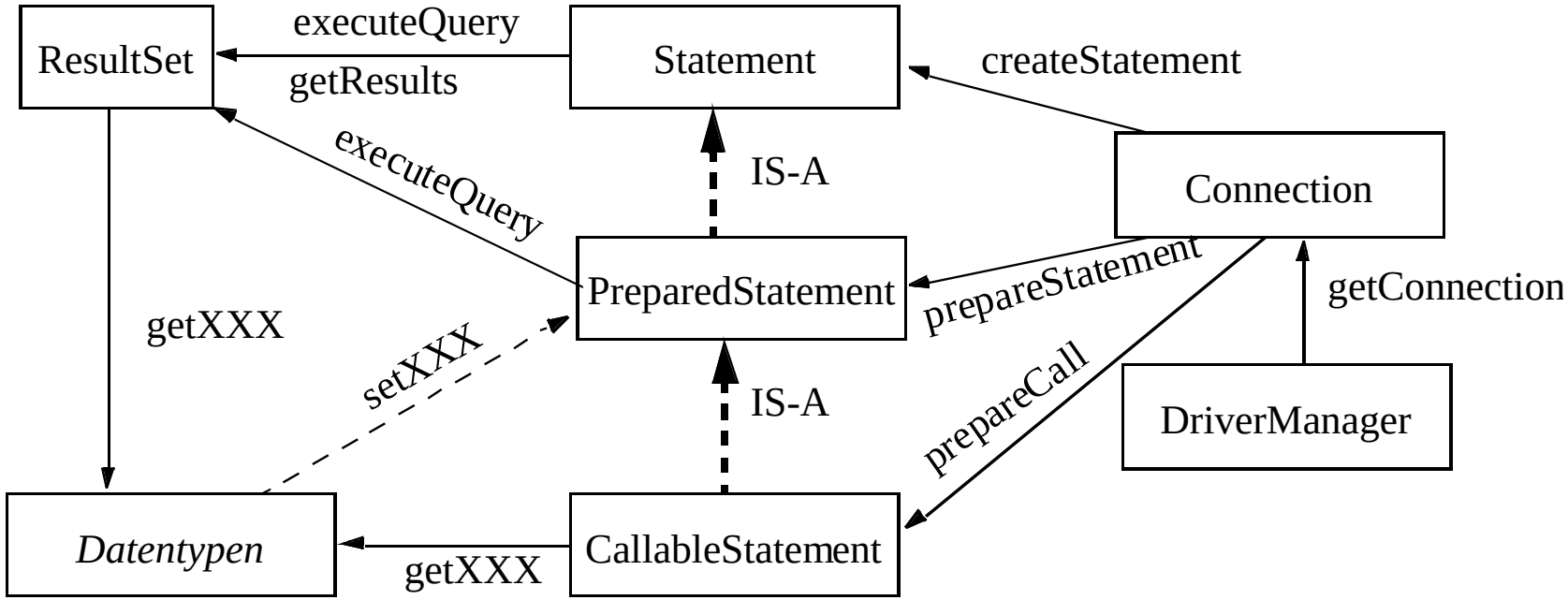
```
Connection con = DriverManager.getConnection
        ("jdbc:postgresql://localhost/erp", "scott", "tiger");
```

- Funktionalität eines Connection-Objekts
  - Senden von SQL-Anweisungen an den Datenbankserver
  - Steuerung von Transaktionen
  - Empfang von Resultaten vom Server
  - Abgreifen von Metainformation über die Datenbank

6.1.2 Anfragen

- Drei Anfragetypen
  - **Statement**
    - Einfache Anfragen ohne Parametrisierung
  - **PreparedStatement**
    - Vorübersetzte parametrisierte Anfrage, die Eingabeparameter unterstützt.
  - **CallableStatement**
    - Aufruf von gespeicherten Prozeduren, die auf dem DBMS hinterlegt sind. → Details im Kapitel 6.2

Wichtige Methoden der Klassen



Ausführen von Anfragen

- Anfragen werden durch execute-Methoden ausgeführt
  - ResultSet *executeQuery(String sql)*
    - *Lesende Anfrage liefert die Ergebnisse als ein Objekt der Klasse ResultSet*
  - int *executeUpdate(String sql)*
    - *Änderungsoperationen (Einfügen neuer Tupel, Löschen, …)*
      - *Ergebnis ist die Anzahl der involvierten Tupel.*
    - *Schemaänderungen (z. B. create table, …). Ergebnis = 0.*
  - boolean execute(String sql)
    - *Methode wird dann genutzt, wenn es nicht bekannt ist, ob die SQL-Anweisung eine Anfrage (Rückgabewert true) oder eine Änderungsoperation (false) ist.*

Diskussion

- **Bei einer Anfrage kann bisher der String zur Laufzeit dynamisch erstellt werden.**
  - Vorteil
    - Hohe Flexibilität
  - Nachteil
    - Sicherheitsproblem bei Parameterübergabe
      - SQLInjection
    - Kosten für die Übersetzung einer Anfrage zur Laufzeit
      - Übersetzungszeit kann höher als die Ausführungszeit sein.

PreparedStatement

- **Ziel**
  - Vermeidung der Nachteile bei Statement ohne dabei die Flexibilität zu verlieren.

- **Grundlegende Idee**
  - Bei der Erzeugung des Anfrageobjekts muss die komplette SQL-Anweisung (bis auf Parameter) vorliegen.
    - Diese parametrisierte Anfrage wird einmal übersetzt.
  - Anfrage kann ohne Neuübersetzung beliebig oft ausgeführt werden.
    - Typisierte Eingabeparameter bekommen zur Laufzeit Werte zugewiesen.

Eingabeparameter

- **Eingabeparameter**
  - Parameter als „?“ im SQL-Befehl
    - PreparedStatement stmt = con.prepareStatement(“select x, y from Points where x < ? and x > ?”);
  - Setzen der Parameter durch Aufruf von set-Methoden. Für jeden Typ gibt es eine spezielle Methode.
    - stmt.setInt(1, 20); stmt.setInt(2, 10);
  - Spezielle Methoden zur Übergabe von NULL-Werten
    - void setNull(int stelle, int jdbcType);

- **Ausführen der Anfrage mit rs = stmt.execute();**

Anfrageergebnisse

- Anfragen mit select-Klausel liefern als Ergebnis ein Objekt der Klasse ResultSet (Cursor).
  - Methode ***next()***
    - Setzt den Cursor auf das nächste Element. Wenn erfolgreich, wird true zurückgeliefert.
    - Anfangs steht der Cursor vor dem ersten Element.
  - Zugriff auf die Spalten erfolgt über **getXXX-Methoden**, wobei es für jeden Typ int, double, … eine eigene Methode gibt.
    - Parameter ist die Position der Spalte in der Relation oder der Name des Attributs.
    - Methode ***getObject*** liefert zu dem SQL-Typ einen passenden Java-Objekttyp.

Abbildung von SQL-Typen in Javatypen

- Problem
  - Typsystem der verschiedenen DBMS ist unterschiedlich
- Rudimentäre Lösung
  - JDBC bietet zur Überwindung der Inkompatibilitäten in der Klasse **java.sql.Types** einige generische Typbezeichner (als ganzzahlige Konstanten) an.
    - z. B. **VARCHAR, LONGVARBINARY, INTEGER, DOUBLE, DATE, TIME**
  - Typen sind dann dort einzusetzen, wo SQL-Typen anzugeben sind.
    - Umsetzung in die spezifischen Typen des DBMS erfolgt automatisch über den Treiber.
  - Entsprechend wird bei den Aufrufen der Methoden ***getXXX- und setXXX verfahren.***
    - Konvertierung von einem Java-Typ in einen JDBC-Typ und implizit durch den Treiber in einen Typ des zugrunde liegenden DBMS.

Null-Werte in Ergebnissen

- Null-Werte erfordern wieder eine Sonderbehandlung!
  - Null-Werte können in folgenden Fällen auftreten:

    a) Wenn die getXXX-Methode den Wert null liefert.
    b) Wenn ***getBoolean*** false liefert.
    c) Wenn ***getInt*** und ***getDouble*** 0 liefert.
  - Durch Aufruf der Methode boolean ***wasNull()*** kann dann überprüft, ob das zuletzt abgefragte Attribut tatsächlich ein Nullwert war.

Spezielle Cursor

- Bewegung des Cursors
  - TYPE_FORWARD_ONLY
    - Bewegung des Cursors nur in eine Richtung
    - Dies ist die Default-Einstellung!
  - TYPE_SCROLL_INSENSITIVE
    - Änderungen in der Datenbank haben keinen Einfluss auf den Snapshot des Cursors
  - TYPE_SCROLL_SENSITIVE
- Lesender oder ändernder Cursor
  - CONCUR_READ_ONLY
    - Nur lesender Zugriff
  - CONCUR_UPDATABLE
    - Änderungsoperationen sind über den Cursor möglich

Navigationsmöglichkeiten

- Methoden
  - next()
  - previous()
  - first()
  - last()
  - beforeFirst()
  - afterLast()
  - relative(int rows)
  - absolute(int r)

BatchUpdates

- Bündelung von mehreren Änderungsoperationen
  - Gemeinsame Verarbeitung als eine Einheit in der Datenbank

- Hierzu gibt es folgende Methoden
  - void ***addBatch(String sql);***
    // Fügt eine neue Änderungsoperationen zum Batch
  - int[] ***executeBatch();***
    // Führt einen Batch aus, wobei als Ergebnis die
    // Anzahl der betroffenen Tupel geliefert wird.

Beispiel

- **Einfügen von 4 Tupel in die Relation Emp(Name,Gehalt)**

```
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO Emp " + "VALUES('Müller', 3000)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Schmidt', 4000)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Becker', 3500)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Bauer', 900)");
int [] updateCounts = stmt.executeBatch();
```

6.1.3 Metadaten

- Flexible Anwendungen erfordern die Auswertung von Metadaten zur Laufzeit.
- JDBC bietet folgende Schnittstellen für Metadaten an
  - ***ResultSetMetaData***
    - Beschreibung des Ergebnisses einer Anfrage, die einem Objekt der Klasse ResultSet entspricht.
  - ***DatabaseMetaData***
    - Beschreibung der eigentlichen Datenbank

ResultSetMetaData

- Metadaten einer Ergebnisrelation
  - Anzahl der Spalten
  - Datentyp und Namen der Spalten
- Erforderlich wenn nicht bekannt ist, welche der getXXX-Methoden bei einem ResultSet-Objekt rs angewendet werden können.

  ➔ Aufruf von rs.getMetaData() liefert dann die Metadaten
- Beispiel:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Personal");
ResultSetMetaData rsm = rs.getMetaData();
for (int i = 1; i <= rsm.getColumnCount(); i++) {
      int ct = rsm.getColumnType(i);
      String cn = rsm.getColumnName(i);
      String ctn = rsm.getColumnTypeName(i);
      System.out.println("Column #" + i + ": " + cn + " of type " + ctn +
                                          " (JDBC type: " + ct + ")");
}
```

DatabaseMetaData

- Bereitstellung von Information über
  - das Schema der Datenbank
    - Namen der Relationen
  - die Eigenschaften des DBMS
- Konkrete Funktionalität
    - Methoden zur Beschreibung von Datenbankobjekten, wie z. B. ***getTables***, ***getColumns***
    - Methoden zum Abfragen wichtiger Systemkonstanten, wie z. B. ***getMaxRowSize***, ***getMaxConnection***
    - Methoden zum Abfragen von Features des DBMS, wie z. B. ***supportsUnion***, ***supportsANSI92FullSQL***
    - Methoden zu Informationen über die Datenbank, wie z. B. ***getURL***, ***getSQLKeyWords***

6.1.4 Transaktionssteuerung

- JDBC kennt keinen Befehl zum Starten einer Transaktion
  - vergl. SQL:92
- Transaktionen beenden:
- Im Auto-Commit-Modus (default) wird nach jedem SQL-Befehl automatisch ein Commit durchgeführt:
- JDBC unterstützt fünf Isolationsstufen:

```
// java.sql.Connection
void commit();
void rollback();

void setAutoCommit(boolean enable);
boolean getAutoCommit();

static int TRANSACTION_NONE;
static int TRANSACTION_READ_UNCOMMITTED;
static int TRANSACTION_READ_COMMITTED;
static int TRANSACTION_REPEATABLE_READ;
static int TRANSACTION_SERIALIZABLE;
void setTransactionIsolation(int level);
int getTransactionIsolation();
```

JDBC mit Transaktionen

```
try {
    Class.forName("oracle.jdbc.OracleDriver");
} catch (ClassNotFoundException e) {...}

Connection con = DriverManager.getConnection(
                    " jdbc:oracle:thin:@venus.mathematik.uni-
                      marburg.de:1521:Init_DB ", <name>, <passwd>);
try {
    con.setAutoCommit(false);
    con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

    //Anfrage erzeugen und ausführen

    con.commit();
    con.close();
} catch (SQLException e) {
    con.rollback();
    ...
}
```

Erweiterungen JDBC 2-4

- **Entwicklung von JDBC**
  - Bisherige Beschreibung entspricht dem ursprünglichen JDBC-Standard.
  - 1998 wurde bereits JDBC 2 verabschiedet.
  - Ab Java 6 wird inzwischen JDBC 4 unterstützt.
    - JDBC4.2-kompatible Treiber existieren.
- **Überblick der Erweiterungen**
  - Flexibles Scrollen auf Objekten der Klasse **ResultSet**
    - **Unterstützung von Änderungsoperationen**
  - Unterstützung von Typen aus SQL99 (JDBC3) und SQL 2003 (JDBC4)
  - Batch-Updates
  - Sicherungspunkte
  - Weitere optionale Änderungen findet man im Package javax.sql

Zusammenfassung

- **JDBC ist ein bibliotheksorientierter Ansatz**
  - Überwindung des Mismatch der Mengenverarbeitung in SQL und der Objektverarbeitung in Java
- **Problem**
  - Abbildung der Objekte auf Relationen bleibt bestehen.
  - In JDBC muss dies explizit durchgeführt werden.
    - Nutzung von getXXX-Methoden ist sehr aufwändig
    - Laufzeitfehler durch falsche Typkonvertierungen

Beurteilung des Ansatzes

- **Nachteil einer CALL-Schnittstelle:**
  - Objektorientierter Impedance Mismatch bleibt
    - ➔ komplizierte Programmierung
      - Abbildung zwischen den Objekten des AWP und den Objekten der Datenbank durch den Programmierer
    - ➔ fehleranfällig
      - Typsystem der Programmiersprache und des Datenbanksystems sind unterschiedlich
- **Vorteile**
  - hohe Flexibilität
    - Generierung dynamischer Anfragen zur Laufzeit durch String-Operationen
      - ➔ Security-Probleme

6.2 Prozedurale Erweiterung von SQL

- Idee
  - Prozedurale Konzepte in SQL.
    - Als **SQL/PSM** (Persistent Stored Modules) Teil des SQL-Standards.
      - **PL/SQL** ist eine sehr erfolgreiche Erweiterung von SQL der Firma Oracle
      - PostgreSQL bietet mit **PLPGSQL** ebenfalls eine prozedurale Erweiterung von SQL an.
  - Primäres Ziel: Ausführung von benutzerdefinierten **Funktionen auf dem Server**
    - Oracle, IBM, … bieten inzwischen an, in Java codierte Prozeduren im DBS ablaufen zu lassen.

- Vorteile im Vergleich zu bisherigen Ansätzen
  - Unabhängigkeit vom Client bei der Ausführung von Prozeduren
  - Leistungsverbesserung, da keine Kommunikation
  - Zentrale Verwaltung von gemeinsam genutzter Funktionalität
  - Implementierung von Funktionalität, die sonst mit SQL nicht so einfach abbildbar ist.

Vorgehensweise

- Implementierung einer Prozedur oder Funktion
  - Syntax der Sprache orientiert sich an der objektorientierten Programmiersprache Ada
- Installation der Prozedur auf dem Server
- Registrierung der Prozedur auf dem Server
  - Hierzu benutzt man die SQL-Operationen

    create procedure bzw. create function.
- Aufruf der Funktion/Prozedur vom Client
  - Aufruf in der SQL-Shell des Datenbanksystems
  - Aufruf über ein Objekt der Klasse CallableStatement in JDBC

Beispiel einer Funktion in PLPGSQL

- Deklaration und Definition einer Funktion

```
create function simple(x in int) returns float as
 $$
begin
        return x * 1.1;
end;
 $$
LANGUAGE PLPGSQL;
```

  - Funktion wird übersetzt und auf dem Server registriert (wenn nicht bereits vorhanden).
- Direkter Aufruf der Funktion

```
select simple(42);
```

- Aufruf der Funktion in einer SQL-Anfrage

```
select simple(note) from pmzuteilung;
```

Beispiel einer Prozedur in PLPGSQL

- Deklaration und Definition einer Prozedur

```
create function psimple(x in int, y float out) as
 $$
begin
        select into y x * 1.1;
end;
 $$
LANGUAGE PLPGSQL;
```

  - Prozedur wird übersetzt und auf dem Server registriert (wenn nicht bereits vorhanden).
- Direkter Aufruf der Prozedur

```
select psimple(42);
```

- Aufruf der Funktion in einer SQL-Anfrage

```
select psimple(note) from pmzuteilung;
```

Parameter in SQL-Funktionen

- **SQL-Funktionen bestehen aus Kopf (Name und ggf. Parameterliste) und Rumpf**
- **Angabe des I/O-Modus bei den Parametern von Funktionen:**
  - IN nur Eingabe
  - OUT nur Ausgabe
  - INOUT nutzbar für Ein- und Ausgabe
- **Richtige Funktionen haben genau einen Rückgabewert, dessen Typ im Kopf mit RETURNS angegeben wird.**
  - Parameter haben dann nur I/O-Modus IN
- **SQL-Funktionen und SQL-Prodzeduren:**
  - sowohl Kopf als auch Rumpf sind in SQL geschrieben
- **Im Gegensatz dazu gibt es auch externe Funktionen:**
  - Kopf ist in SQL
  - Rumpf in Hostsprache, wie z. B. Java, geschrieben.

Lokale Variablen und Zuweisungen

- **Wie in prozeduralen Sprachen gibt es in in Funktionen die folgenden Möglichkeiten**
  1. Deklaration lokaler Variablen

     DECLARE

     a INTEGER;

     - Diese Variablen müssen in dem Deklarationsblock definiert werden.
  2. Zuweisungsoperatoren
     - a := a + 1;

Table Functions (1)

- **Funktionen können nicht nur skalare Werte als Ergebnis liefern, sondern sogar komplette Tabellen.**
  - Die Tabellen können dabei durch eine SQL-Anfrage erzeugt werden oder durch eine iterative Erzeugung von Tupeln.

- **Beispiele einer Table-Funktion mit SQL-Anfrage**

```
CREATE or replace FUNCTION get_pers (abteilung INT)
RETURNS TABLE (lname VARCHAR, fname VARCHAR) AS
        $$ BEGIN
                RETURN QUERY
                        SELECT pname, vorname
                        FROM personal
                        where abtnr = abteilung;
        END; $$
    LANGUAGE PLPGSQL;

select * from get_pers(10);
```

Table Functions (2)

SQL:2003

- Beispiele einer Table-Funktion mit iterativer Erzeugung der Tupel

```
CREATE or replace FUNCTION gauss (upto INT)
 RETURNS TABLE (i int, sum int) AS $$
                    BEGIN
                              i := 0; sum := 0;
                              LOOP
                                        i := i+1;
                                        sum := sum +i;
                                        return next;
                                        exit when (i >= upto);
                              END LOOP;
                    END; $$
          LANGUAGE PLPGSQL;

   select * from gauss(10);
```

Externe Gespeicherte Prozeduren

- Prinzipiell ist es möglich in anderen Sprachen die Prozeduren zu implementieren.
  - Derzeit unterstützt PostgreSQL als Sprache C
- Andere DBMS unterstützen auch Java (z. B. Oracle)
  - Erstellen einer Quelldatei Hello.java

```
public class Hello {
        public static String world() { return “Hello World”;}
}
```

  - Übersetzen der Quelldatei auf dem Client

```
javac Hello.java
```

  - Übergeben der class-Datei an das DBMS

```
loadjava -user scott/tiger Hello.class
```

  - Registrieren der Prozedur als SQL-Funktion

```
create function HelloWorld returns varchar as
language java name ’Hello.world() return java.lang.String’;
```

Callable Statement

- **Und übrigens können Prozeduren auch als Callable-Statement direkt in JDBC aufgerufen werden.**
  - Ausführen der gespeicherten Prozedur

```
CallableStatement cstmt = con.prepareCall( "{call ? HelloWorld()}");
cstmt.registerOutParameter(1, java.sql.Types.String);
ResultSet rs = cstmt.executeQuery();
String str = cstmt.getString(1);
```

- **Aber schön und elegant ist dies nicht.**

Zusammenfassung

- **SQL-Erweiterungen**
  - Implementierung der Funktionalität im DBMS
  - Vermeidung von teuren Datentransfers auf den Client
    - Function-Shipping statt Data-Shipping
- **Nachteile**
  - Trotz SQL-Standard unterscheiden sich die Systeme, so dass man schnell einen sogenannten Vendor-Lock-in produziert.
    - Ein Wechsel zu einem anderen DBMS ist dann sehr teuer.
  - Schlechte Werkzeugunterstützung der Programmierung

6.3 JSON und SQL

- **SQL passt sich immer neu an die Bedürfnisse von Anwendungen an.**
  - Derzeit von besonderem Interesse ist die Unterstützung von Daten, die als JSON-Dokument gegeben sind.
  - Dies ist bereits in den aktuellen Standard von SQL eingeflossen.

- **JSON steht für JavaScript Object Notation.**
  - Derzeit wird JSON gerne als Datenaustauschformat und zur Beschreibung von Daten genutzt.
  - JSON bietet die Möglichkeit zur Beschreibung von strukturierten als auch semi-strukturierten Daten.
    - Insbesondere können Daten hierarchisch modelliert werden.

JSON in 5 Minuten

- Aufbau eines JSON-Objekts
- Verwendung eines JSON-Datentyps in SQL
- Anfragebeispiele für JSON in SQL

Objekte in JSON

- **JSON-Objekte setzen sich zusammen aus einer Liste von sogenannten Key-Value Paaren.**
  - **Key** wird durch eine Zeichenkette repräsentiert.
    - Key muss in der Liste eindeutig sein.
  - **Value** ein Wert aus einem Wertebereich.
    - Wertebereich können atomar sein (vom Typ int, string) sein.
    - Werte können aber auch strukturiert sein.
      - Array
      - JSON-Objekt
    - Spezialwerte: true, false, null
  - Listenelemente werden mit Komma getrennt.
  - Zwischen Key und Value steht ein ':‘.

Beispiel

```
{  "a"   :“dbs1",
   "b"   :42,
   "c"   :{"c1":10, "c2":11},
   "d"   :[{"x" : 1, "y" : 2},{"x" : 3, "y" : 4},],
   "e"   :true,
   "f"   :null}
```

Umgang mit Tabellen

- **In Postgres gibt es den Datentyp json.**

```
CREATE TABLE bestellung (
   oi int PRIMARY KEY,
   info json NOT NULL
);
```

- **Einfügen in die Tabelle**

```
INSERT INTO bestellung
VALUES (
  1,  '{ "customer": “Fritz", "items": {"product": “Bier","qty": 6}}'
   );
```

- **Anfragen der Tabelle**

```
SELECT *
FROM bestellung;
```

Anfragen (1)

- **Zugriff auf Schlüssel und Werte**
  - PostgreSQL bietet die Operatoren -> und ->> an.
    - -> liefert den Schlüssel als JSON-Objekt.
      - Dann kann der Operator rekursiv angewendet werden.
    - ->> liefert den Schlüssel als Text.

- **Beispiel**

```
SELECT info ->>'customer' AS kunde
FROM bestellung
WHERE info -> 'items' ->> 'product' = 'Windel'
```

Anfragen (2)

- **Umwandlung von JSON in einen Datentyp**

```
SELECT info ->> 'customer' AS customer,
            info -> 'items' ->> 'product' AS product
FROM orders
WHERE CAST (info -> 'items' ->> 'qty' AS INTEGER) = 2
```

- **Damit können auch Aggregate berechnet werden.**

```
SELECT   AVG (CAST (info -> 'items' ->> 'qty' AS INT) )
FROM bestellung
```

Anfragen (3)

- **Die Funktion each_json liefert alle Key-Value Paare des JSON-Objekts**

  SELECT json_each (info)

  FROM bestellung;

- **Die Funktion json_object_keys alle Schlüssel.**

  SELECT json_each (info)

  FROM bestellung;

6.3 Erweiterungen von Programmiersprachen: LINQ in C#

- **Idee**
  - Statt über eine Bibliothek die Funktionalität zur Verfügung zu stellen, erweitert man die Programmiersprache um die SQL-Syntax.
- **Vorteile**
  - Nahtlose Nutzung der Funktionalität in der Programmiersprache
    - Keine Typkonvertierungsprobleme zwischen SQL und der Programmiersprache.
  - SQL kann dann auch für andere Datenstrukturen, wie z. B. Sets verwendet werden.
- **Derzeit eine der komfortabelsten Schnittstelle für die Datenbankprogrammierung!**

Beispiel mit C#

```
// Verbindungsobjekt zur Datenbank
SQLConnection nw = new SQLConnection("Data Source = (local); Initial
                                Catalog = MyDatabase; Integrated Security=SSPI");

// Anfrage
var companyNameQuery =
            from cust in nw.Customers
            where cust.City == "London"
            select cust.CompanyName;

// Zugriff auf die Ergebnisse
foreach (var customer in companyNameQuery) {
            Console.WriteLine(customer);
}
```