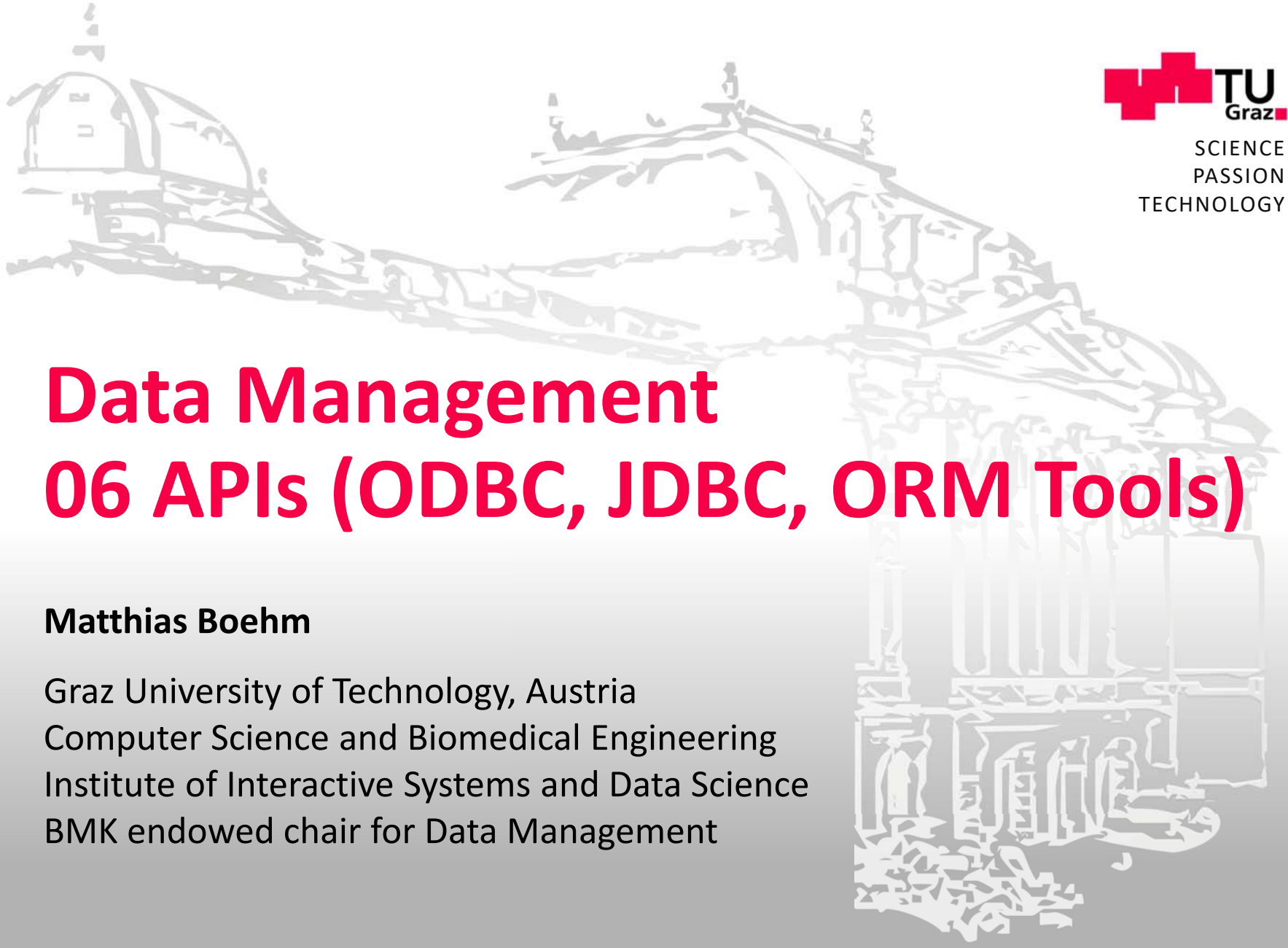


SCIENCE
PASSION
TECHNOLOGY

Data Management
06 APIs (ODBC, JDBC, ORM Tools)

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

Last update: Nov 16, 2020

ISDS

Announcements/Org

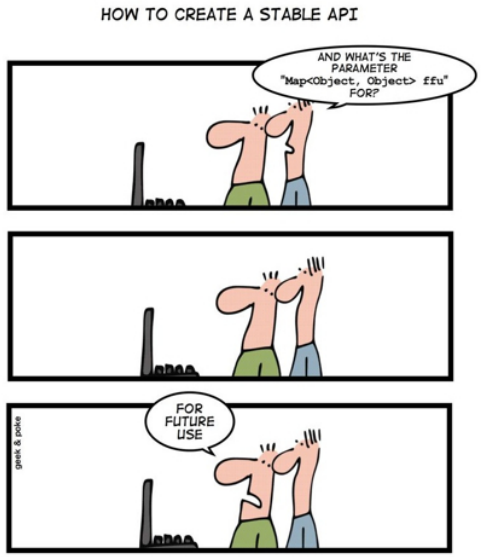
- **#1 Video Recording**
  - Link in **TeachCenter** & **TUbe** (lectures will be public)
  - Due to second lockdown: webex recording

- **#2 Reminder Communication**
  - **Newsgroup:** news://news.tugraz.at/tu-graz.lv.dbase (https://news.tugraz.at/cgi-bin/usenet/nntp.csh?tu-graz.lv.dbase)
  - **Office hours:** Mo 12.30-1.30pm (https://tugraz.webex.com/meet/m.boehm)

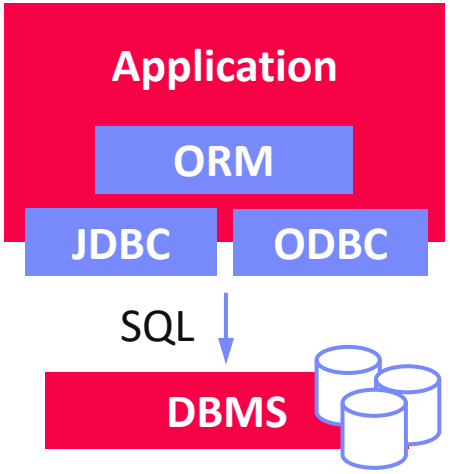
- **#3 Exercise Submissions**
  - **Exercise 2:** published Nov 09, deadline Dec 01
    - Updated Nov 13/14/16 (CreateSchema.sql, data, expected results Q8)
  - Exercise 1: grading by Dec 01

What's an API and again, why should I care?

- **Application Programming Interface (API)**
  - Defined **set of functions or protocols** for system or component communication
  - Interface independent of concrete implementation ➔ **decoupling of applications** from underlying libraries / systems
  - API stability of utmost importance



- **Examples**
  - **Linux:** kernel-user space API → system calls, POSIX (Portable Operating System Interface)
  - **Cloud Services:** often dedicated REST (Representational State Transfer) APIs
  - **DB Access: ODBC/JDBC and ORM frameworks**



Agenda

- **Exercise 2: Query Languages and APIs (Recap)**
- **Call-level Interfaces (ODBC/JDBC) and Embedded SQL**
- **Object-Relational Mapping Frameworks**

Exercise 2:
Query Languages and APIs

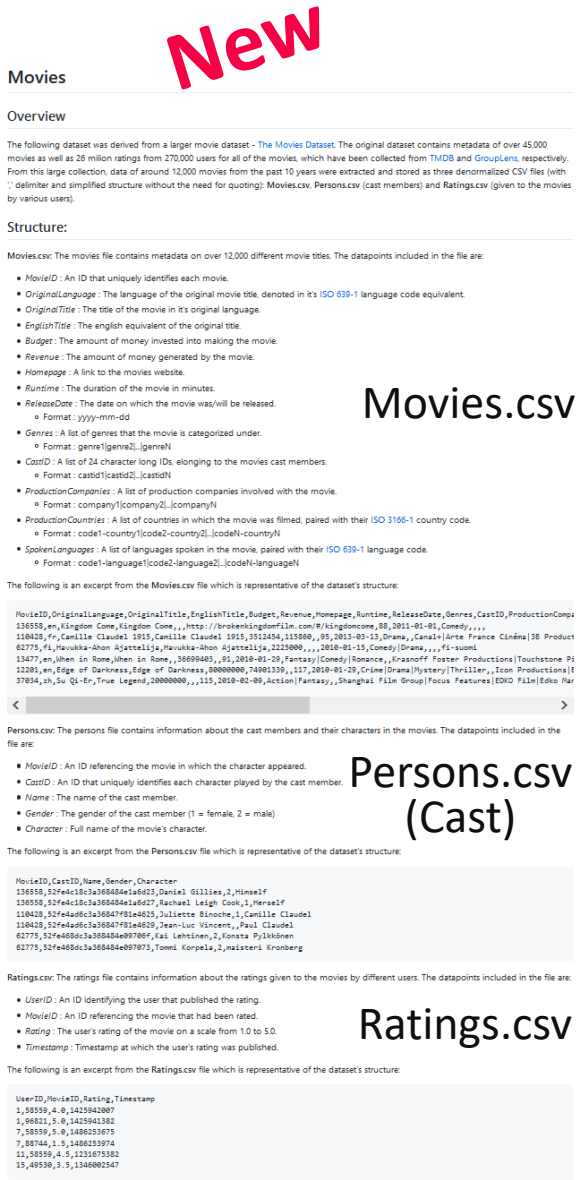
Published: **Nov 09, 2020**
(online since Nov 02)

Deadline: **Dec 01, 2020**

Exercises: The Movies Dataset

- **Dataset**
  - Derived (extracted, cleaned) from **The Movies Dataset** for movies year ≥ 2011
  - Clone or download your copy from https://github.com/tugraz-isds/datasets.git
  - Find CSV files in <datasets>/movies

- **Exercises**
  - **01** Data modeling (relational schema)
  - **02** **Data ingestion and SQL query processing**
  - **03** Physical design tuning, query processing, and transaction processing
  - **04** Large-scale data analysis (distributed query processing and ML model training)



Task 2.1: Schema Creation via SQL (3/25 points)

- **Schema creation via SQL**
  - Relies on lectures **04 Relational Algebra** and **05 Query Languages (SQL)**
  - Setup DBMS PostgreSQL, and start `pgAdmin` (UI), or `psql` (terminal)
  - Create database `db<studentID>` and setup relational schema
    - Ignore (1) **derived budget classification of movies**, and (2) **location of actors** (countries they live in)
    - Primary keys, foreign keys, `NOT NULL`, `UNIQUE`

- **Recommended Schema**
  - Feel free to use and submit the provided schema

- **Partial Results**
  - `CreateSchema.sql`, updated **Nov 13**

Task 2.2 Data Ingestion via CLI (10/25 points)

- **Data Ingestion Program via ODBC/JDBC**
  - Relies on lectures **05 Query Languages (SQL)** and **06 APIs (ODBC, JDBC)**
  - Write a program that performs **deduplication and data ingestion**
  - Programming language of your choosing (Python, Java, C#, C++ recommended)

- **Data Ingestion Process**
  - Data: https://github.com/tugraz-isds/datasets/tree/master/movies
  - Invoke your ingestion program as follows → script to compile and run

```
./runIngestData.sh ./Movies.csv ./Persons.csv ./Ratings.csv \
  <host> <port> <database> <user> <password>
```

(e.g., `127.0.0.1 5432 db1234567 postgres postgres`)

- **Partial Results**
  - Source code `IngestData.*`, and
  - Script `runIngestData.sh`

Task 2.3: SQL Query Processing (10/25 points)

- **SQL Query Processing**
  - Relies on lecture **05 Query Languages (SQL)**

New:
Expected result sets provided on website

- **List of Queries**
  - **Q01:** Obtain the details of the movie(s) with English title Interstellar.
    (return English title, release date, runtime, budget, and revenue)
  - **Q02:** Which movies were produced in Austria?
    (return original title, release date; sorted ascending by original title)
  - **Q03:** How many movies were filmed per year?
    (return year, count; sorted ascending by year)
  - **Q04:** Which movie had the most unique actors?
    (return English title, actor count)
  - **Q05:** Compute the top-20 highest-rated movies by average rating.
    (return English title, release date, average rating; sorted descending by the average rating, and for equal ratings, sorted ascending by English title)

Task 2.3: SQL Query Processing (10/25 points)

- **List of Queries, cont.**
  - **Q06:** How many movies of genre Science Fiction have been released each year between 2012 and 2016 (both inclusive)?
    (return year, count; sorted ascending by year)
  - **Q07:** Who are the top-10 most successful actors by number of movies they played in, and profits these movies generated?
    (return actor name, count, profit as union distinct of top-10 actors by maximum count and top-10 actors by maximum profit).
  - **Q08:** Which pairs of actors co-appeared most-frequently in movies each year between 2010 and 2013 (both inclusive)? (return name of actor1, name of actor2, year, count, sorted ascending by year)

- **Partial Results**
  - SQL Script for each query: `Q01.sql`, `Q02.sql`, …, `Q08.sql`

Task 2.4: Query Plans (2/25 points)

- **Explain Query Plans**
  - Relies on lecture **04 Relational Algebra** and **05 Query Languages (SQL)**
  - Obtain and **analyze execution plans** of Q06

- **Example**

```
EXPLAIN VERBOSE
SELECT L.location, count(*)
  FROM Participant P,
       Locale L
  WHERE P.lid = L.lid
  GROUP BY L.location
  HAVING count(*)>1
```

```
"HashAggregate  (...)"     // grouping
" Output: l.location, count(*)"
" Group Key: l.location"
" Filter: (count(*) > 1)"  // selection
" -> Hash Join  (...)"     // join
"     Output: l.location"  // projection
"     Hash Cond: (l.lid = p.lid)"
"     -> Seq Scan on Locale l  (...)"
"         Output: l.lid, l.location"
"     -> Hash  (...)"
"         Output: p.lid"   // projection
"         -> Seq Scan on Participant p  (...)"
"             Output: p.lid"
```

- **Partial Results**
  - `ExplainQ06.sql`

Call-level Interfaces (ODBC/JDBC) and Embedded SQL

Call-level Interfaces vs Embedded SQL

- **#1 Call-level Interfaces**
  - Standardized in ISO/IEC SQL – Part 3: CLI
  - **API of defined functions for dynamic SQL**
  - **Examples:** ODBC (C/C++), JDBC (Java), DB-API (Python)

- **#2 Embedded SQL**
  - Standardized in ISO/IEC SQL – Part 2: Foundation / Part 10 OLB
  - **Embedded SQL in host language** (typically static)
  - **Preprocessor** to compile CLI protocol handling
    ➔ **SQL syntax and type checking**, **but static** (SQL queries, DBMS)
  - **Examples:** ESQL (C/C++), SQLJ (Java)

Embedded SQL

- **Overview**
  - **Mix host language constructs and SQL** in data access program ➔ **simplicity?**
  - **Precompiler translates program** into valid host language program
  - Primitives for creating cursors, queries and updates, etc ➔ **In practice, limited relevance**

- **Example SQLJ**
  - Cursors with and without explicit variable binding

```
#sql iterator StudIter
   (int sid, String name);
StudIter iter;
#sql iter = {SELECT * FROM Students};

while( iter.next() )
   print(iter.sid, iter.name);

iter.close();
```

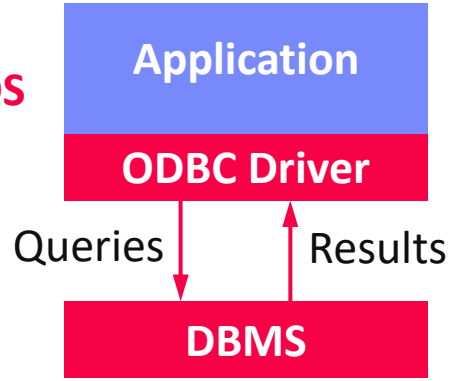
```
int id = 7;
String name;

#sql {SELECT LName INTO :name
   FROM Students WHERE SID=:id};

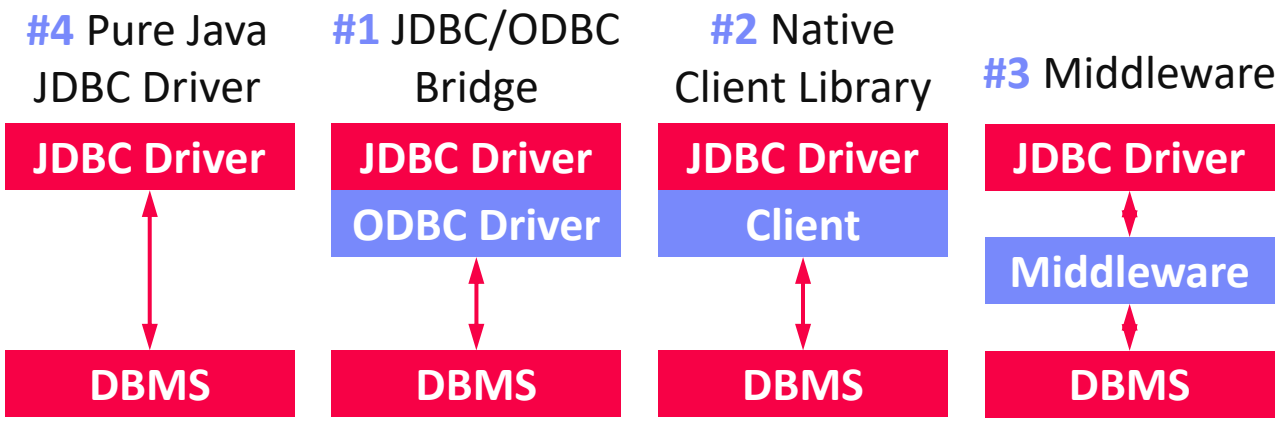
print(id, name);
```

CLI: ODBC and JDBC Overview

- **Open Database Connectivity (ODBC)**
  - **API for accessing databases independent of DBMS and OS**
  - Developed in the **early 1990s → 1992** by Microsoft (superset of ISO/IEC SQL/CLI and Open Group CLI)
  - **All relational DBMS have ODBC implementations**, good programming language support

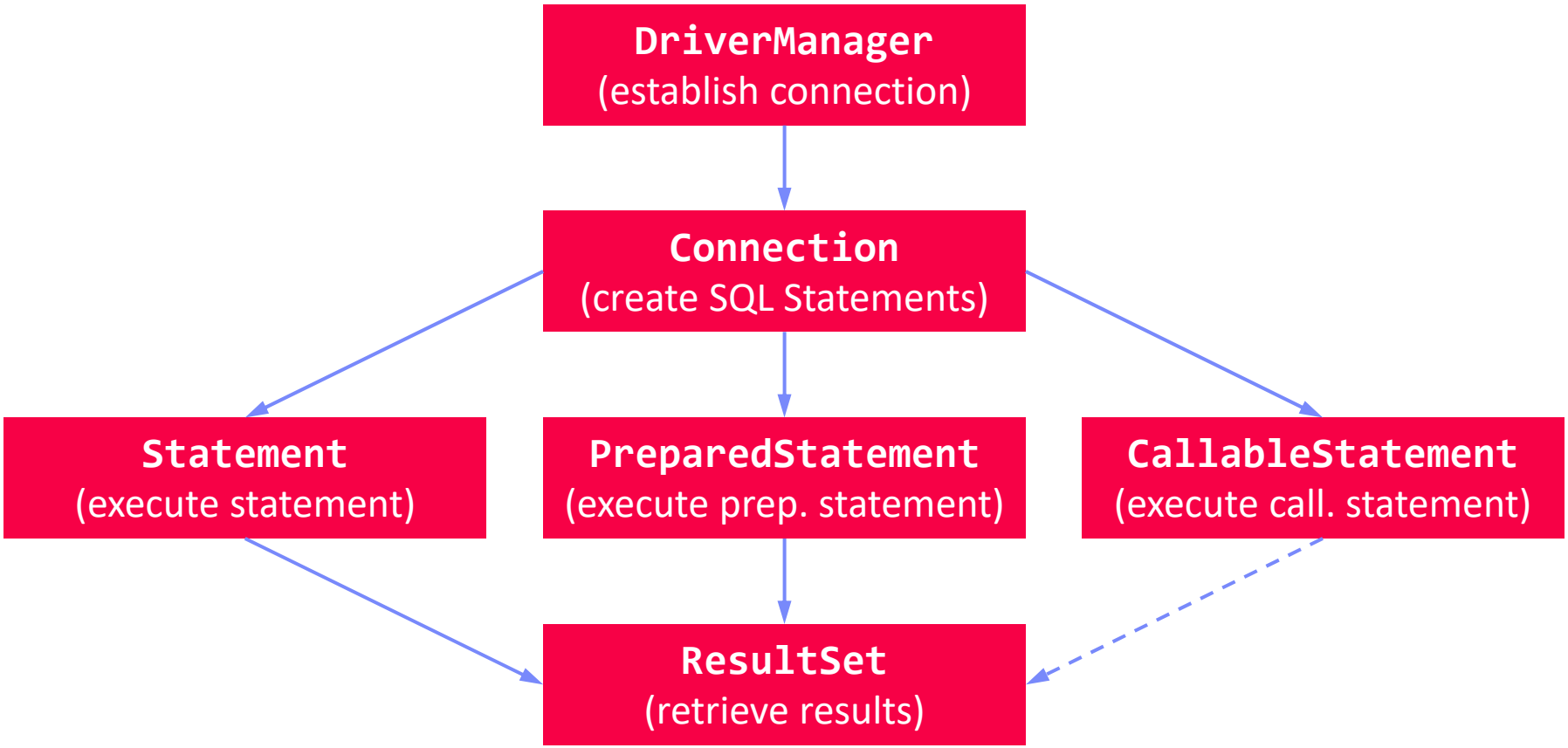


- **Java Database Connectivity (JDBC)**
  - **API for accessing databases independent of DBMS from Java**
  - Developed and released by Sun in **1997**, JDBC 4.0 (2006), JDBC 4.3 in Java 9
  - Most relational DBMS have JDBC implementations
  - Types of Drivers



Note: Reuse of drivers from open source DBMS

JDBC Components and Flow



JDBC Connection Handling

- **Establishing a Connection**
  - **DBMS-specific URL strings** including host, port, and database name
  - Stateful handles representing user-specific DB sessions
  - JDBC driver is usually a jar on the class path
  - **Connection and statement pooling** for performance

```
Connection conn = DriverManager
   .getConnection("jdbc:postgresql:"+
   "//localhost:5432/db1234567",
   username, password);
```

```
META-INF/services/
java.sql.Driver
```

- **JDBC 4.0**
  - **Explicit driver class loading and registration no longer required**
  - Improved connection management (e.g., status of DB connections)
  - Other: XML, Java classes, row ID, better exception handling

```
Class.forName(
   "org.postgresql.Driver");
```

JDBC Statements

- **Execute Statement**
  - Use for simple SQL statements w/o parameters
  - **Beware of SQL injection**
  - API allows fine-grained control over fetch size, fetch direction, batching, and multiple result sets

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql1);
...
int rows = stmt.executeUpdate(sql2);
stmt.close();
```

Note: PostgreSQL does not support fetch size but sends entire result

- **Process ResultSet**
  - Iterator-like cursor (app-level) w/ on-demand fetching
  - Scrollable / updatable result sets possible
  - Attribute access via column names or positions

```
ResultSet rs = stmt.executeQuery(
   "SELECT SID, LName FROM Students");

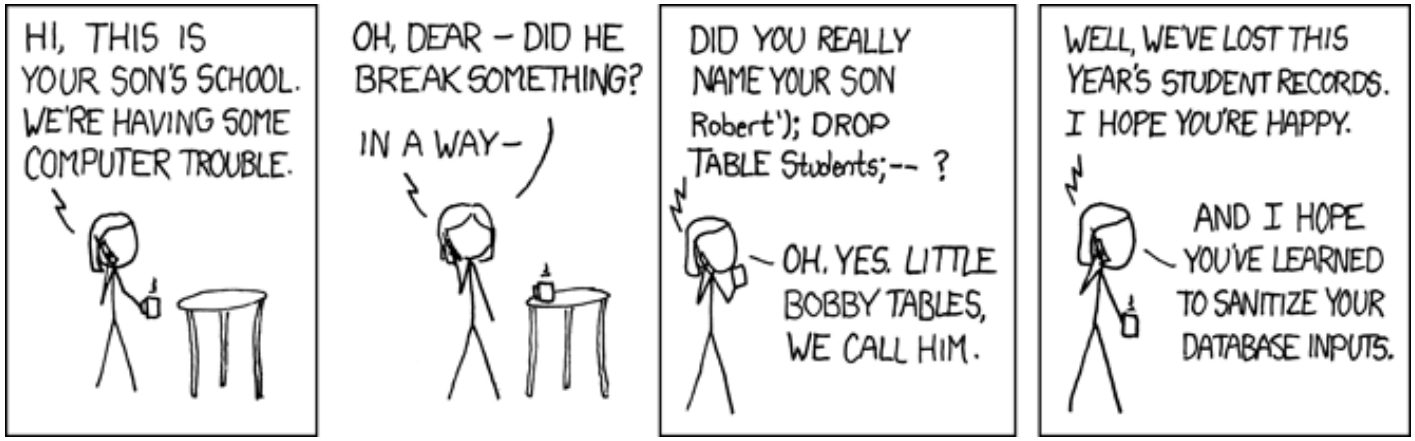
List<Student> ret = new ArrayList<>();
while( rs.next() ) {
   int id = rs.getInt("SID");
   String name = rs.getString("LName");
   ret.add(new Student(id, name));
}
```

Recap: Beware of SQL Injection

- **Problematic SQL String Concatenation**

```
INSERT INTO Students (Lname, Fname)
   VALUES (‘“+ @lname +”‘,’“+ @fname +”’);”;
```

- **Possible SQL-Injection Attack**



[https://xkcd.com/327/]

```
INSERT INTO Students (Lname, Fname) VALUES (‘Smith‘,’Robert’);
DROP TABLE Students; --’);
```

JDBC Prepared Statements

- **Execute PreparedStatement**
  - Use for precompiling SQL statements w/ input params
  - Inherited from Statement
  - **Precompile SQL once**, and execute many times

  ➔ **Performance**

  ➔ **No danger of SQL injection**

```
PreparedStatement pstmt =
   conn.prepareStatement(
     "INSERT INTO Students VALUES(?,?)");

for( Student s : students ) {
   pstmt.setInt(1, s.getID());
   pstmt.setString(2, s.getName());
   pstmt.executeUpdate();
}

pstmt.close();
```

- **Null Handling**
  - Pass null object (explicitly for primitive types)

```
pstmt.setString(2, p[1]);
pstmt.setObject(3, p[2].isEmpty() ?
    null : Integer.valueOf(p[2]),
    Types.INTEGER);
```

- **Queries and Updates**
  - Queries → `executeQuery()`
  - Insert, delete, update → `executeUpdate()`

JDBC Callable Statements

- **Recap: (Stored Procedures, see 05 Query Languages (SQL))**
  - Can be **called standalone via CALL** <proc_name>(<args>);
  - Procedures return no outputs, but might have **output parameters**

- **Execute `CallableStatement`**
  - Create prepared statement for call of a procedure
  - Explicit registration of output parameters
  - Example

```
CallableStatement cstmt = conn.prepareCall(
   "{CALL prepStudents(?, ?)}");

cstmt.setInt(1, 2019);
cstmt.registerOutParameter(2, Types.INTEGER);
cstmt.executeQuery();

int rows = cstmt.getInt(2);
```

Psycopg (Python PostgreSQL Adapter)

- **Overview Psycopg**
  - Implements **Python Database API Specification v2.0** (DB API 2.0)
  - Call-level interface for dynamic SQL, very similar to JDBC

- **Establish Connection**

```
conn = psycopg2.connect(
    host="localhost", port="5432",
    database="db1234567", user=username,
    password=password)
```

- **Execute Statements**
  - Use local cursors

```
cur = conn.cursor()
cur.execute("INSERT INTO Students VALUES(...)")
```

- **Process Result Sets**

```
cur.execute("SELECT SID, LName FROM Students")
students = cur.fetchall()
for row in students:
    print("SID = ", row[0], end = '')
    print("Lname = ", row[1])
```

Psycopg (Python PostgreSQL Adapter), cont.

- **Execute Prepared Statements**

```
cur = conn.cursor()
sql = "INSERT INTO Students VALUES(%s, %s)"
for s in students:
    cur.execute(sql, (s.getID(),s.getName()))
conn.commit()
```

- **Execute Callable Statement**
  - Result set
  - No output parameters

```
cur = conn.cursor()
cur.callproc("prepStudents", (2019, 2))
cur.fetchone()
```

- **Close Connection**

```
cur.close()
conn.close()
```

Preview Transactions

- **Database Transaction**
  - A transaction (TX) is a **series of steps** that brings a database from a **consistent state** into another (not necessarily different) **consistent state**
  - **ACID properties** (atomicity, consistency, isolation, durability)
  - See lecture **08 Transaction Processing and Concurrency**
- **Example**
  - Transfer 100 Euros from Account 107 to 999

```
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   UPDATE Account SET Balance=Balance-100
      WHERE AID = 107;
   UPDATE Account SET Balance=Balance+100
      WHERE AID = 999;
COMMIT TRANSACTION;
```

- **Transaction Isolation Level**
  - **Tradeoff:** isolation (and related guarantees) vs performance
  - READ UNCOMMITTED (~~lost update~~, dirty read, unrepeatable read, phantom R)
  - READ COMMITTED (~~lost update~~, ~~dirty read~~, unrepeatable read, phantom R)
  - REPEATABLE READ (~~lost update~~, ~~dirty read~~, ~~unrepeatable read~~, phantom R)
  - SERIALIZABLE (~~lost update~~, ~~dirty read~~, ~~unrepeatable read~~, ~~phantom R~~)

JDBC Transaction Handling

- **JDBC Transaction Handling**
  - Isolation levels (incl NONE) and (auto) commit option
  - Savepoint and rollback (undo till begin or savepoint)
  - **Note:** TX handling on connection not statements

- **Beware of Defaults**
  - DBMS-specific default isolation levels

(SQL Standard: **SERIALIZABLE**, PostgreSQL: **READ COMMITTED**)

```
conn.setTransactionIsolation(
   TRANSACTION_SERIALIZABLE);
conn.setAutoCommit(false);

PreparedStatement pstmt = conn
   .prepareStatement("UPDATE Account
   SET Balance=Balance+? WHERE AID = ?");

Savepoint save1 = conn.setSavepoint();

pstmt.setInt(1,-100); pstmt.setInt(107);
pstmt.executeUpdate();

if( rand()<0.1 )
   conn.rollback(save1);

pstmt.setInt(1,100); pstmt.setInt(999);
pstmt.executeUpdate();

conn.commit();
```

JDBC Transaction Handling, cont.

- **Batching of Inserts**
  - Batching multiple inserts in one transaction can improve performance
  - **Example**

```
conn.setAutoCommit(false);
PreparedStatement pstmt = conn.prepareStatement(
  "INSERT INTO Persons(AKey, Name, Website, IKey) VALUES(?,?,?,?)");
for( String[] p : tmp ) {
  pstmt.setInt(1, Integer.valueOf(p[0].substring(1)));
  pstmt.setString(2, p[1]);
  pstmt.setString(3, p[5].isEmpty()? null : p[5]);
  pstmt.setObject(4, orgs.get(p[3]+"_"+p[4]),Types.INTEGER);
  pstmt.executeUpdate();
}
conn.commit();
```

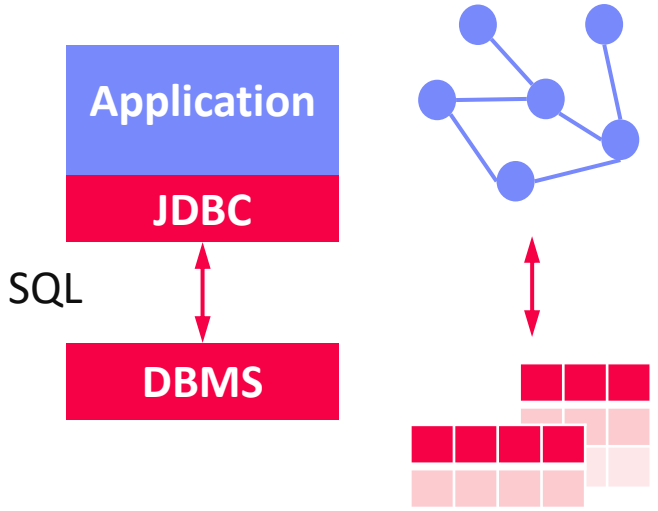
Performance Ref Implementation SS2020:
(36K authors, 28K papers, 101K author-papers)

```
* Auto Commit:          23.7s
* Batched Commits:      12.5s
```

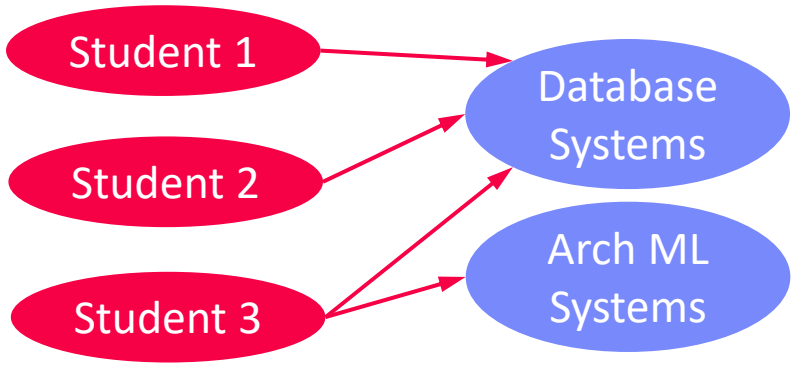
Object-Relational Mapping Frameworks

The "Impedance Mismatch" Argument

- **Problem Description**
  - Applications rely on **object-oriented programming languages** with hierarchies or graphs of objects
  - Data resides in **normalized "flat" tables** (note: ~~OODBMS~~, object-relational)
  - Application is responsible for **bridging this structural/behavioral gap**



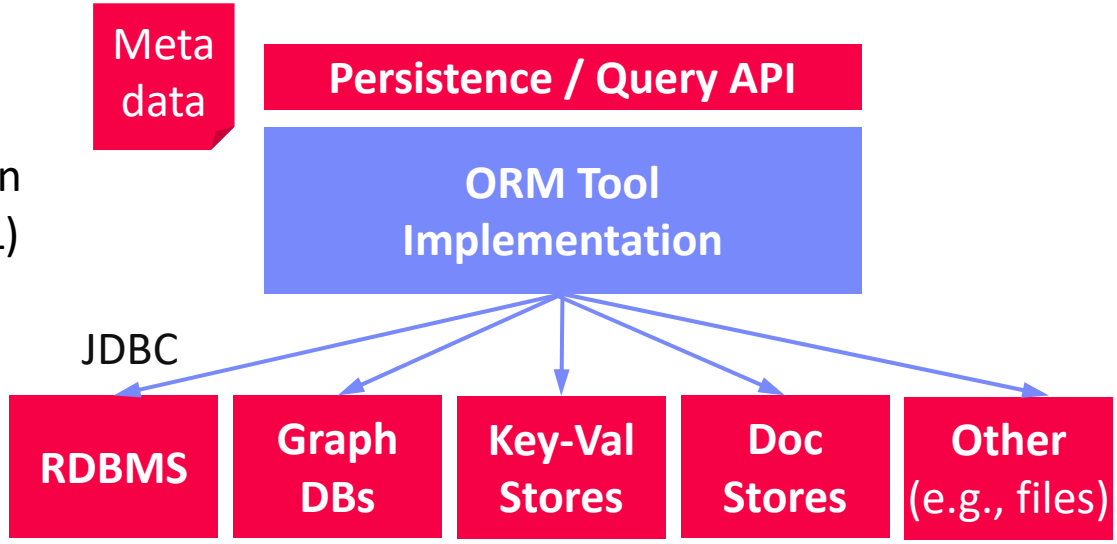
- **Example**
  - `SELECT * FROM Students`
  - ```
    SELECT C.Name, C.ECTS FROM
      Courses C, Attendance A
      WHERE C.CID = A.CID
        AND A.SID = 7;
    ```
  - `… A.SID = 8;`


    ```

Overview Object-Relational Mapping

- **Goals of ORM Tools**
  - Automatic **handling of object persistence lifecycle** and querying of the underlying data stores (e.g., RDBMS)
  - Reduced development effort ➔ **developer productivity**
  - Improved testing and independence of DBMS

- **Common High-Level Architecture**
  - **#1** Persistence definition (meta data → e.g., XML)
  - **#2** Persistence API
  - **#3** Query language / query API



History and Landscape

- **History of ORM Tools** (aka persistence frameworks)
  - Since 2000 J2EE EJB **Entity Beans** (automatic persistence and TX handling)
  - Since 2001 **Hibernate** framework (close to ODMG specification)
  - Since 2002 **JDO** (Java Data Objects) via class enhancement
  - 2006 **JPA** (**Java Persistence API**), reference implementation **TopLink**
  - 2013 JPA 2, reference implementation **EclipseLink**
  - Late 2000s/early 2010s: **explosion of ORM alternatives, but criticism**
  - **2012 - today:** ORM tools just part of a much more diverse eco system

- **Example Frameworks**
  - http://java-source.net/open-source/persistence
  - Similar lists for .NET, Python, etc

JPA – Class Definition and Meta Data

- **Entity Classes**
  - **Define persistent classes** via annotations
  - Add details for IDs, relationship types, and specific behavior on updates
  - Some JPA implementations require enhancement process as post compilation step

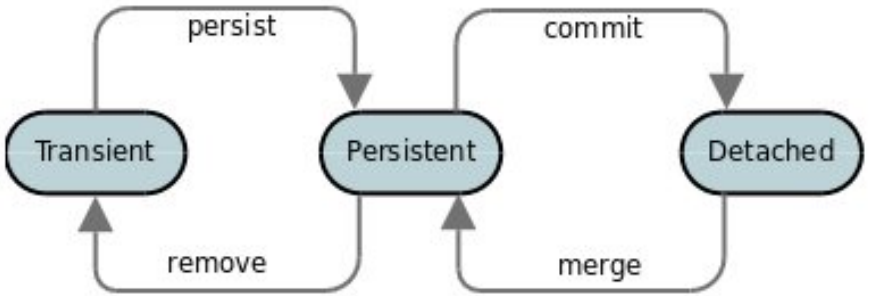
```
@Entity
public class Student {
   @Id
   private int SID = -1;
   private String Fname;
   private String Lname;
   @ManyToMany
   private List<Course> ...
}
```

- **Persistence Definition**
  - **Separate XML meta data** `META-INF/persistence.xml`
  - Includes connection details

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence
      xmlns="http://xmlns.jcp.org/xml/ns/persistence"
      xmlns:xsi=... xsi:schemaLocation=...>
   <persistence-unit name="UniversityDB">
      <class>org.tugraz.Student</class>
      <class>org.tugraz.Course</class>
      <exclude-unlisted-classes/>
      <properties> ... </properties>
   </persistence-unit>
</persistence>
```

JPA – Object Modification

- **CRUD Operations**
  - Insert by making objects persistent
  - Update and delete objects according to object lifecycle states
- **Lifecycle States**
  - Lifecycle state transitions via specific persistence contexts
  - Explicit and implicit transitions




```
EntityManager em = factory
   .createEntityManager();

tx.begin();

Student s = new
   Student(7,"Jane","Smith");
s.addCourse(new Course(...));
s.addCourse(new Course(...));

em.persist(s);

tx.commit();
em.close
```

[**Credit:** Data Nucleus, JPA Persistence Guide (v5.2), http://www.datanucleus.org/products/accessplatform/jpa/persistence.html#lifecycle]

JPA – Query Languages

- **JPQL: Java Persistence Query Language**
  - SQL-like object-oriented query language
  - Parameter binding similar to embedded SQL

```
EntityManager em = factory
   .createEntityManager();
Query q = pm.createQuery(
   "SELECT s FROM Student s
      WHERE s.age > :age");
q.setParameter("age", 35);

Iterator iter = q
   .getResultList().iterator();
while( iter.hasNext() )
   print((Student)iter.next());
```

- **JPQL Criteria API**
  - JPQL syntax and semantics with a programmatic API

```
CriteriaQuery<Student> q = bld.createQuery(Student.class);
Root<Student> c = q.from(Student.class);
q.select(c).where(bld.gt(c.get("age"), bld.parameter(...)));
```

- **Native SQL Queries**
  - Run native SQL queries if necessary
  - Designed as "leaky abstraction"

```
em.createNativeQuery("SELECT *
   FROM Students WHERE Age > ?1");
```

Jdbi (Java Database Interface)

[http://jdbi.org/]

- **Jdbi Overview**
  - Fluent API built on top of JDBC w/ same functionality exposed
  - Additional simplifications for row to object mapping

- **Example**

```
Jdbi jdbi = Jdbi.create("jdbc:postgresql://.../db1234567");
Handle handle = jdbi.open();

jdbi.registerRowMapper(Student.class, (rs, ctx)
  -> new Student(rs.getInt("sid"), rs.getString("lname"));

List<Student> ret = handle
  .createQuery("SELECT * FROM Students WHERE LName = :name")
  .bind(0, "Smith")
  .map(Student.class)
  .list();
```

A Critical View on ORM

- **Advantages**
  - **Simple CRUD operations** (insert/delete/update) and simple queries
  - **Application-centric development** (see boundary crossing)
- **Disadvantages**
  - **Unnecessary indirections** and complexity (meta data, mapping)
  - **Performance problems** (hard problem and missing context knowledge)
  - **Application-centric development** (schema ownership, existing data)
  - **Dependence** on evolving framework APIs
- **Sentiments** (additional perspectives)
  - Omar Rayward: Breaking Free From the ORM: Why Move On?, **2018**
    medium.com/building-the-system/**dont-be-a-sucker-and-stop-using-orms**-190add65add4
  - Vedra Bilopavlović: Can we talk about ORM Crisis?, **2018**
    linkedin.com/pulse/**can-we-talk-orm-crisis**-vedran-bilopavlovi%C4%87
  - Martin Fowler: ORM Hate, **2012** martinfowler.com/bliki/**OrmHate**.html

➔ **Awareness of strength and weaknesses / hybrid designs**

Conclusions and Q&A

- **Summary**
  - **Recap Exercise 2:** Query Languages and APIs
  - **Call-level Interfaces (ODBC/JDBC)** as fundamental access technology
  - **Object-Relational Mapping (ORM)** frameworks existing (**pros and cons**)

- **Next Lectures**
  - **07 Physical Design and Tuning** [Nov 23]
  - **08 Query Processing** [Nov 30]
  - **09 Transaction Processing and Concurrency** [Dec 07]