COMP 430
Intro. to Database Systems

SQL from application code

Some issues

- How to connect to database
  - Where, what type, user credentials, …

- How to send SQL commands
- How to get communicate data to/from DB
  - Object-Relational Impedance Mismatch
- What code goes where

Connecting to DB

Connecting to database – Java example

```
public class Example
{
    public static void main(String[] args)
    {
        Connection connection = null;
        Class.forName("com.Microsoft.jdbc.sqlserver.SQLServerDriver");
        String url = "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=MYDB";
        connection = DriverManager.getConnection(url, "USERNAME", "PASSWORD");
        …
    }
}
```

Imports and error handling omitted for brevity.

Need sqljdbc4.jar in CLASSPATH.

Key pieces:

- Driver
- Host
- DB name
- User credentials

Connecting to database – Python + SQLAlchemy example

String split for readability.

```
engine = create_engine("mssql+pyodbc://USERNAME:PASSWORD@localhost/MYDB" +
"driver=SQL+Server+Native+Client+10.0")
```

<u>Key pieces:</u>

- Driver
- Host
- DB name
- User credentials

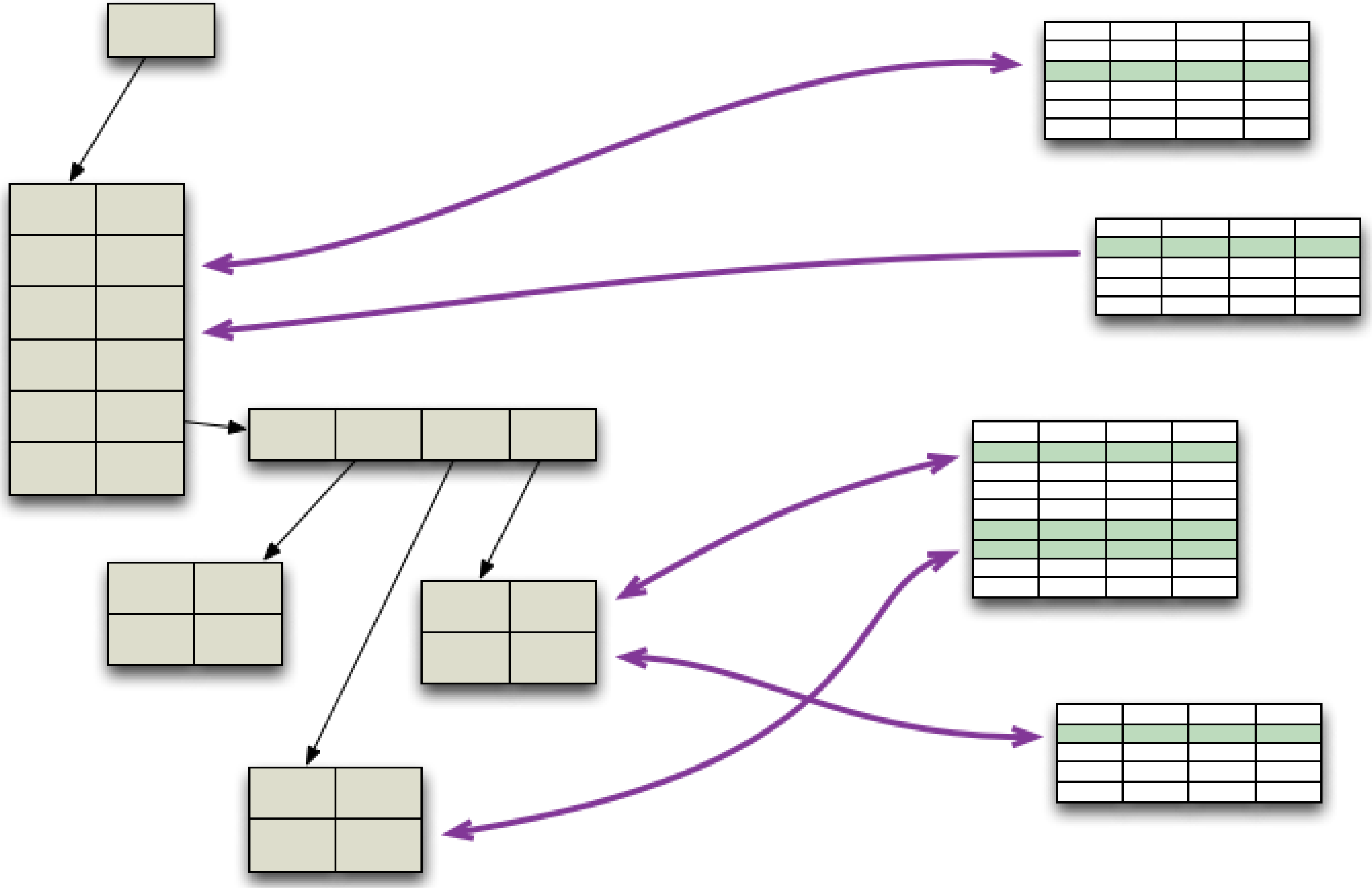
Connecting to database

Those *connection strings* share a standard syntax, although some arguments can be specified separately by library.

DB/Application Interface Overview

Object-Relational Impedance Mismatch

Mapping data is **HARD**



Mapping data is **HARD**

Under the hood, often involves <u>two</u> mappings:

- application $\rightarrow$ string
- string $\rightarrow$ DB

Communicating & (un)parsing data is relatively expensive.

Strategy: Plain SQL

```
connection = …;
Statement stmt1 = connection.createStatement();
stmt1.executeUpdate("CREATE TABLE Student" +
                                  "(id INT, first VARCHAR(50), last VARCHAR(50))");
…
Statement stmt2 = connection.createStatement();
ResultSet result = stmt2.executeQuery("SELECT id FROM Student");
…
```

Plain SQL Summary

Minimal API

☺ SQL commands not restricted by API

☹ Requires knowing SQL

☹ Limited static type checking possible

☹ Dynamic SQL strings allow injection attacks

What are some bad input strings?

```
studentId = getRequestString(“StudentID”);
String query = “SELECT first, last FROM Student WHERE id = “ + studentId;
Statement stmt = connection.createStatement(query);
```

Satisfy WHERE clause: “123456789 OR 1=1”

Cause destructive behavior: “123456789; DROP TABLE Student”

Expose implementation details: “1 AND userid IS NULL”

Many variations on these themes.

Techniques for preventing injection attacks

- Validate input used in SQL command strings
  - E.g., numeric input only contains digits
- Build SQL commands via parameterized APIs (example)
- Only call DB functions; no permission to access tables directly
- Tightly manage user permissions
- Don’t expose DB error messages

```
studentId = getRequestString(“StudentID”);
Statement stmt = connection.preparedStatement(
   “SELECT first, last FROM Student WHERE id=?”);
Stmt.setInt(1, Integer.parseInt(studentId));
ResultSet result = Stmt.executeQuery(stmt);
while (result.next()) {
   String first = result.getString(“first”);
   String last = result.getString(“last”);
   …
}
```

Strategy: Object-Relational Mapping Library

- Think in terms of application & persistent objects
  - Specify which objects persistent
  - Interaction with DB largely(?) hidden

- Basic abstraction: table = class, table row = ob
- Generates both from specification
  - Generates CRUD code
  - Maps between application/DB types
- Manages data movement

HIBERNATE

SQLAlchemy

django

Sequelize

LINQ

Microsoft® .NET

ORM: Table/object specification

Hibernate

```
<hibernate-mapping>
    <class name=“Student” table=“Student”>
        <id name=“id” type=“int” column=“id”>
            <generator class=“native”/>
        </id>
        <property name=“firstName” column=“first_name” type=“string” />
        <property name=“lastName” column=“last_name” type=“string” />
    </class>
</hibernate-mapping>
```

SQLAlchemy

```
class Student(Base):
    __tablename__ = ‘Student’
    id = Column(Integer, primary_key=True, autoincrement=True)
    first_name = Column(String)
    last_name = Column(String)
```

ORM: Data movement

- Granularity:
  - Application-level traditionally accesses an attribute at a time.
  - DB-level accesses one or more records at a time.
- Consistency
  - Are application & DB data guaranteed to be consistent?
- Eager vs. lazy loading

ORM Summary: Pros

☺ Simplifies application programming

- Avoids SQL syntax
- Fits in style of language
- Allows static type checking

```
resultset = session.query(User, Document, DocumentPermission)
                   .join(Document)
                   .join(DocumentPermission)
                   .filter(User.email == ‘john_smith@foo.com’)
```

ORM Summary: Cons

☹ Doesn't really simplify learning curve
- Often largely SQL in disguise
- Still need to understand SQL concepts
- Complex library
- Every ORM is different.

☹ Efficiency overhead

☹ May encourage bad usage
- SELECTing all fields
- n+1 SELECTs problem

"ORMs make the easy even easier, and the difficult impossible. … writing good, high-performing SQL is hard enough as it is. … Add an ORM … and then you have to figure out how to say what you want … through a mapping layer."

http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

n+1 SELECTs Problem

ORM pseudo-code:

```
foreach emp in select_all_employees()
   print select_salary_of_employee(emp)
```

Avoid looping over SELECT!

Corresponds to:

```
SELECT * FROM Employee;

SELECT salary FROM Salary WHERE emp_id = 1
SELECT salary FROM Salary WHERE emp_id = 2
…
```

Instead of:

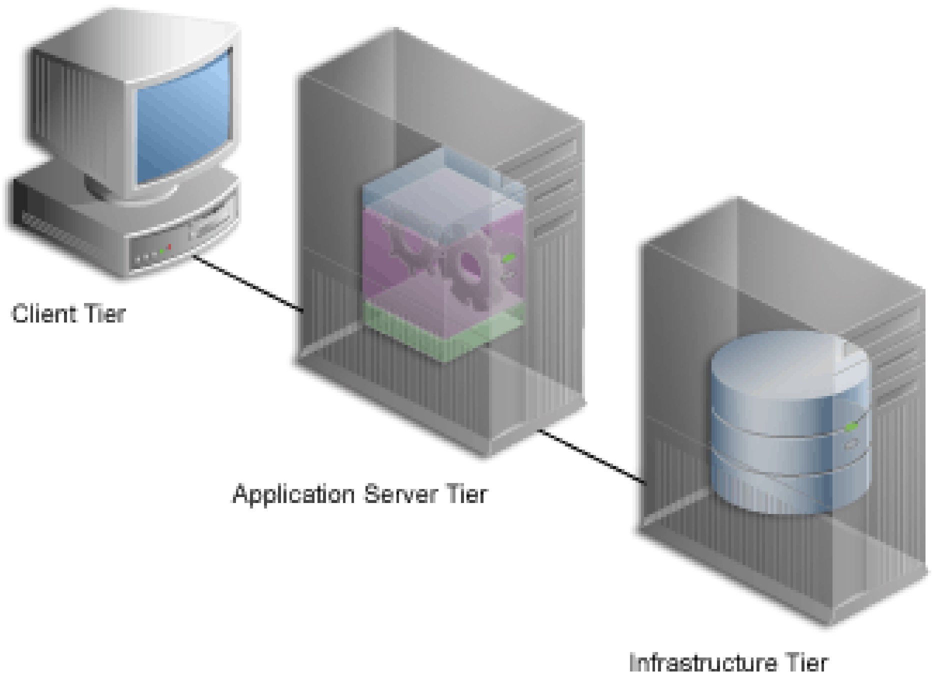
```
SELECT salary
FROM Salary s
INNER JOIN Employee e ON e.emp_id = s.emp_id
```

Strategy: Write your own ORM

☺ Specialized to what you need

☹ Lots of code

☹ Error-prone



Where does “business logic” go?

Application | Application & DB | DB

What’s this “business logic”?

Common term for the core computation.

Example for a bank:

Business logic:

- Creating accounts
- Computing interest
- Transactions between accounts

Application logic:

- Website/mobile UI
- User log-in

Logic in application only

DB as dumb storage. Even data-integrity rules in application.

- Add application servers to handle load.

- Support multiple DBMS's (at once or over time)
  - E.g., creating application s/w that is DB-agnostic
  - DBMS's are commodities. Use whichever is currently best.
  - Put logic in the single application, rather than replicating in multiple DBMS's.
- SQL language & tools (e.g., unit testing) aren't as good.
- Decision-makers & software engineers are application-focused.
  - Few good SQL programmers.
  - DB programmers often in a separate DB team.

Logic in DB only

DB in charge of data – including data integrity, data computation

- Use functions, triggers extensively
- Application accesses DB via DB-level function API
- Add DB servers & replication to handle load

- Application only gets the data it really needs
  - Minimize communication.
  - Expose minimum of DB data & structure. Maximize security.
- Support multiple applications (at once or over time) using same DB
  - Applications change more frequently than DB architectures.
- Application programmers shouldn’t write SQL or pseudo-SQL (ORM)
  - Protect data from bad programmers.
  - Maximize query optimization.

Logic trade-off summary

Application:

- **Convenience** – More Java/C++/… programmers than for SQL. Better language, libraries, & tools.
- **Flexibility** – Can use the DB however you want.

DB:

- **Encapsulation** – Application programmer only uses API, and doesn't need details, including much SQL.
- **Safety** – Protect DB from application programmer. Only expose desired API.
- **Performance** – Less I/O. Better DBMS optimization.

Between the extremes

Application & DB each used for their best features

- DB – CRUD, data integrity, data replication
- Business logic in either application or DB, **depending on situation**.
- Application – Everything else

http://www.vertabelo.com/blog/notes-from-the-lab/business-logic-in-the-database-yes-or-no-it-depends