Programming with SQL

Bringing Data to Your Codebase

Section I
Overview

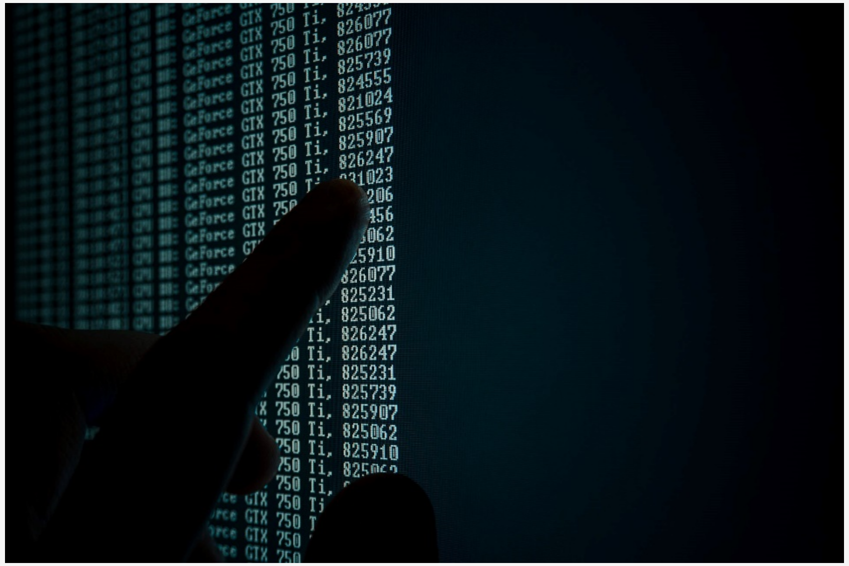
Agenda

Here is what we will do for the next 3 hours:

1. Expectations and Setup
2. Reading Data in Python, R, and Java
3. Writing Data in Python, R, and Java
4. Connection Management and Design Strategy

What to Expect

We are going to learn how to leverage SQL from programming platforms like Python, R, and Java.



- While we will not cover each functionality encyclopedically, we will learn enough functionalities to create fully functional database applications.
- We will also cover connection pooling, design strategy, and other considerations when making a SQL database talk to a coded application.

What Not to Expect

We obviously cannot cover programming with SQL on every platform like Go, Swift, Rust, .NET, Julia, or <*put favorite language here*>

- Hopefully the knowledge we gain from using Python, R, and Java will give a good starting point to transfer knowledge to these other platforms.

- The principles and design philosophy we learn here should be largely translatable from these three platforms.

Setting Up Python

If you want to follow along, have your favorite Python environment set up:

https://www.python.org/

Code files and other resources are here:

https://github.com/thomasnield/oreilly_programming_with_sql

Make sure to have the *thunderbird_manufacturing.db* database file in the working folder.

Have the following libraries set up:

- SQLite (already included with Python library)
- SQLAlchemy
- Pandas

Setting Up R

Have your favorite R environment set up:

https://www.r-project.org/

https://rstudio.com/

Code files and other resources are here:

https://github.com/thomasnield/oreilly_programming_with_sql

Make sure to have the *thunderbird_manufacturing.db* database file in the working folder.

Have the following packages set up:

- DBI
- RSQLite
- pool

Setting Up Java

Have your favorite Java 8 (or later) environment set up:

https://aws.amazon.com/corretto/

https://www.jetbrains.com/idea/

Code files and other resources are here:

https://github.com/thomasnield/oreilly_programming_with_sql

Make sure to have the *thunderbird_manufacturing.db* database file in the working folder.

Have the following libraries set up (Maven or Gradle build system recommended):

- com.zaxxer:HikariCP:3.4.1
- org.xerial:sqlite-jdbc:3.30
- tech.tablesaw:tablesaw-core:0.36.0

Section II
Reading Data in Python, R, and Java

EXERCISE

Use Python, R, or Java to retrieve PRODUCT records with a PRICE of at least 100 and print them.

EXERCISE – Python Solution

```
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///thunderbird_manufacturing.db')
conn = engine.connect()

stmt = text("SELECT * FROM PRODUCT WHERE PRICE >= 100")
results = conn.execute(stmt)

for record in results:
    print(record)
```

EXERCISE – R Solution

```
library(DBI)
library(RSQLite)

db <- dbConnect(SQLite(), dbname='thunderbird_manufacturing.db')

my_query <- dbSendQuery(db, "SELECT * FROM PRODUCT WHERE PRICE >= 100")

my_data <- dbFetch(my_query, n = -1)

dbClearResult(my_query)

print(my_data)

remove(my_query)
dbDisconnect(db)
```

EXERCISE – Java Solution

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JavaLauncher {

    public static void main(String[] args) {

        try {
            Connection conn = DriverManager.getConnection("jdbc:sqlite:/c:/my_folder/thunderbird_manufacturing.db");
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * from PRODUCT WHERE PRICE >= 100");

            while (rs.next()) {
                System.out.println(rs.getString("PRODUCT_NAME") + " " + rs.getString("PRICE"));
            }
            //release connection
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Section III
Writing Data in Python, R, and Java

EXERCISE

Use Python, R, or Java to create a new PRODUCT with a name of “NiteHawk”, a product group of “BETA”, and a price of “41”.

However, do it in a transaction but never complete the transaction. Do a query of the PRODUCT table to see if it was inserted temporarily.

EXERCISE – Python Solution

```
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///thunderbird_manufacturing.db')
conn = engine.connect()

# Create a transaction, but do not commit it
transaction = conn.begin()

# INSERT a new record
stmt = text("INSERT INTO PRODUCT (PRODUCT_NAME,PRODUCT_GROUP,PRICE) VALUES (:productName,:productGroup,:price)")

conn.execute(stmt, productName="NiteHawk",
             productGroup="BETA",
             price=41.0
             )

# Check records to see if last one inserted
for r in conn.execute(text("SELECT * FROM PRODUCT")):
    print(r)

# Close connection, don't commit
conn.close()
```

EXERCISE – R Solution

```
library(DBI)
library(RSQLite)

db <- dbConnect(SQLite(), dbname='thunderbird_manufacturing.db')

my_query <- dbSendQuery(db, "SELECT * FROM PRODUCT WHERE PRICE >= 100")

my_data <- dbFetch(my_query, n = -1)

dbClearResult(my_query)

print(my_data)

remove(my_query)
dbDisconnect(db)
```

EXERCISE – Java Solution

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JavaLauncher {

    public static void main(String[] args) {

        try {
            Connection conn = DriverManager.getConnection("jdbc:sqlite:/c:/my_folder/thunderbird_manufacturing.db");
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * from PRODUCT WHERE PRICE >= 100");

            while (rs.next()) {
                System.out.println(rs.getString("PRODUCT_NAME") + " " + rs.getString("PRICE"));
            }
            //release connection
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Section IV
Pooling and Design Strategy

Design Strategy - Onus of Work

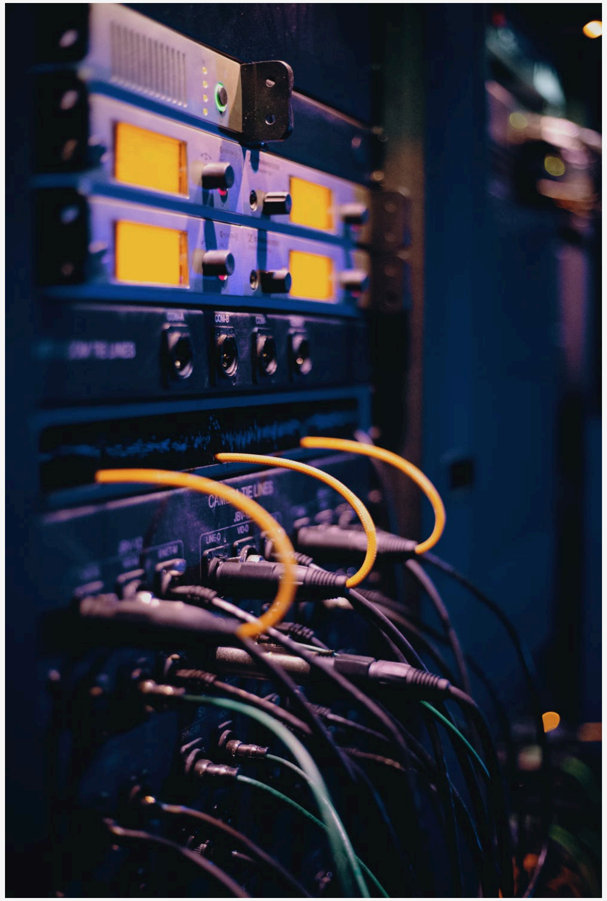
When using SQL with a programming platform like Python, Java, or R, you will constantly be making a decision where the onus of processing will happen.

Should the database engine do the computation work, or the programming platform?

- You can simply pull in data and have your Python/Java/R codebase do the heavy-lifting.
- You can also leverage more complex SQL against the database, and have Python/Java/R consume the results.
- With a very large, expensive and calculated dataset you can save it to a temporary table and use it to support your Python/R/Java application.

A good rule of thumb: start with the simplest solution with minimal code/SQL that liberally hits the database as-needed, and gradually introduce caching strategies as performance starts to warrant it.

Never concatenate parameters, and use established SQL libraries to inject parameters safely to prevent SQL injection.



Design Strategy – Connection Pooling

Modern applications (whether they are client side or server side), can be enormously complex.

- Code design and modules change.
- Several services can be running simultaneously.

Typically multiple threads are executing code simultaneously to parallelize work.

- Having multiple threads accessing a single database connection can be hazardous, unless only one thread accesses and uses the connection at a time.
- It can be helpful to have multiple connections to support multiple threads safely.
- It is more efficient to reuse a connection rather than create/dispose it for each task.

Design Strategy – Connection Pooling

Because most applications are multithreaded, or running several tasks simultaneously, it can be necessary to use a connection pool that maintains several threads.

- Connection pools are set up to maintain a fixed number of connections, and reuse them as needed.
- Of course, threads and database connections are expensive, so we must be conservative in how many we allow.
- If there are more threads than there are connections, then threads may be queued to wait for an available database connection.
- If a connection expires, the thread pool will dispose and replace it.

For example, we can choose to have up to five database connections in a connection pool, and have threads use them as needed.

Connection pools can be set up dynamically to increase and decrease the number of connections based on needs.

Connection Pooling Libraries

Java – HikariCP (fastest and recommended), Vibur, TomCat, C3PO

Python- SQLAlchemy

R – Pool

Design Strategy – Error Handling

One thing we did not talk about is **error handling**, and how this can cause connection leaks.

When a block of code fails using a database connection, it may skip over the line of code releasing the database connection.

To handle this, we can always make sure to dispose the connection in a **try-catch-finally** block so the connection is let go regardless if the operation is successful.

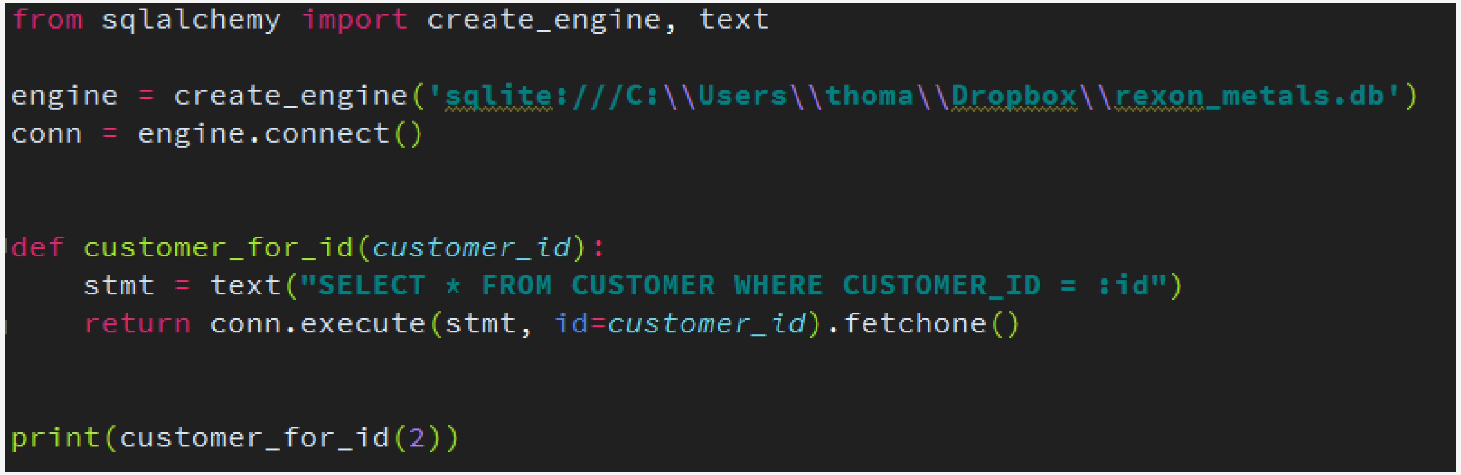
When available, libraries and specialized language features can automatically handle the disposal of connections in a block of code as well (e.g. Java has try-with-resources).

Preventing SQL Injection

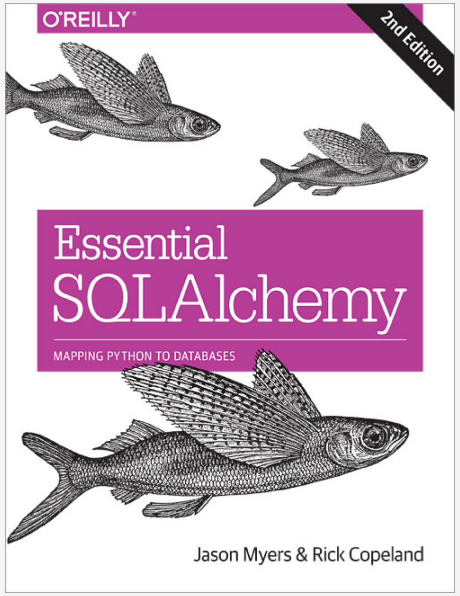
To prevent SQL injection, *never* concatenate a SQL string with parameters

Instead, use the right tools and libraries to safely inject parameters for you

For Python, use SQLAlchemy



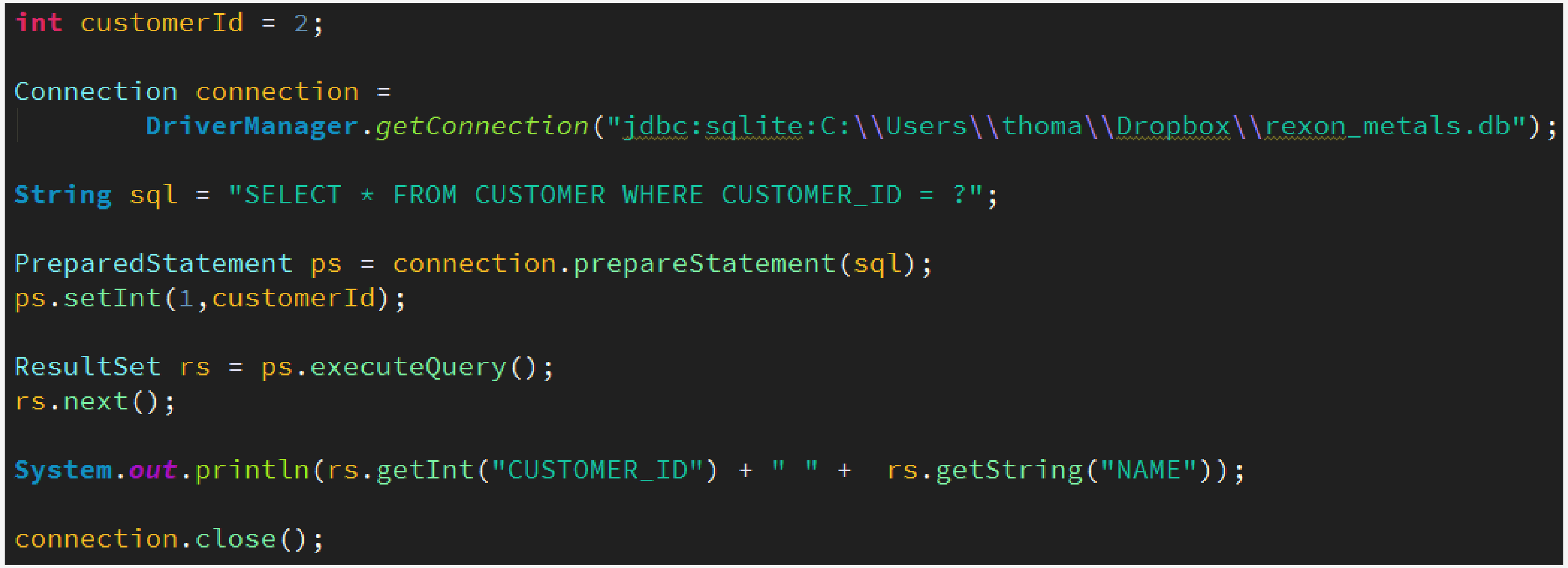
```
from sqlalchemy import create_engine, text

engine = create_engine('sqlite:///C:\\Users\\thoma\\Dropbox\\rexon_metals.db')
conn = engine.connect()


def customer_for_id(customer_id):
    stmt = text("SELECT * FROM CUSTOMER WHERE CUSTOMER_ID = :id")
    return conn.execute(stmt, id=customer_id).fetchone()


print(customer_for_id(2))
```



More info at:
http://www.sqlalchemy.org/

Preventing SQL Injection

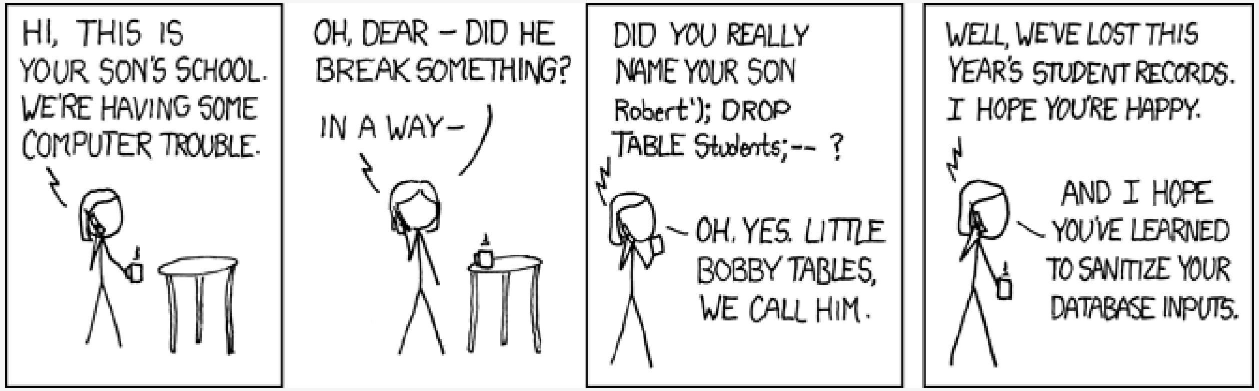
For Java, Scala, Kotlin, and other JVM languages use JDBC's PreparedStatement



```java
int customerId = 2;

Connection connection =
        DriverManager.getConnection("jdbc:sqlite:C:\\Users\\thoma\\Dropbox\\rexon_metals.db");

String sql = "SELECT * FROM CUSTOMER WHERE CUSTOMER_ID = ?";

PreparedStatement ps = connection.prepareStatement(sql);
ps.setInt(1,customerId);

ResultSet rs = ps.executeQuery();
rs.next();

System.out.println(rs.getInt("CUSTOMER_ID") + " " +  rs.getString("NAME"));

connection.close();
```

More info at:
http://tutorials.jenkov.com/jdbc/index.html
http://www.marcobehler.com/make-it-so-java-db-connections-and-transactions

SQL Injection Humor


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY –
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Source: Google Images

Source: https://xkcd.com/327/

SQL Injection in the News

Simple Voice-Command SQL Injection Hack into Alexa Application

https://securityboulevard.com/2019/09/simple-voice-command-sql-injection-hack-into-alexa-application/

How a 'NULL' License Plate Landed One Hacker in Ticket Hell

https://www.wired.com/story/null-license-plate-landed-one-hacker-ticket-hell/

This couple cannot do the simplest things online because their last name is ‘Null’

https://thenextweb.com/insider/2016/03/27/last-name-null-is-tough-for-computers/

EXERCISE

If you intend on having thousands of simultaneous tasks being executed in an application, it is a good idea to create a connection for each task (TRUE/FALSE)

EXERCISE

If you intend on having thousands of simultaneous tasks being executed in an application, it is a good idea to create a connection for each task (TRUE/FALSE)

FALSE! Creating thousands of connections to process thousands of tasks can be computationally expensive and cause crashes on the application side and server side. It is better to pool a limited number of connections (e.g. 12-24 connections) and reuse those connections to process the tasks.

EXERCISE

Which of the following are benefits of connection pools?

A) They reuse connections making the application more efficient

B) Timed out connections will automatically get disposed and replaced

C) Thread safety is enforced with database connection resources

D) They will automatically take threads back from tasks when tasks are complete.

E) Multiple database queries/updates can be executed simultaneously and safely

EXERCISE

Which of the following are benefits of connection pools?

✓ A) They reuse connections making the application more efficient

✓ B) Timed out connections will automatically get disposed and replaced

✓ C) Thread safety is enforced with database connection resources

✗ D) They will automatically take threads back from tasks when tasks are complete.

✓ E) Multiple database queries/updates can be executed simultaneously and safely

All the above are benefits of connection pools except for item “D”. The coded task is responsible for releasing a connection back to the connection pool so another task can use it.

HOMEWORK

Use your knowledge of SQL, SQLite, and your favorite programming language to build a personal banking app.

A) Be able to enter bank transactions (with a date, description, and currency amount) and save them to a SQLite database.

B) Create a table of categories (grocery, utilities, mortgage, etc) and be able to attach them to the transactions

C) Create spending reports by category across day/week/month/year.

Depending on your comfort, you can choose to make this application completely in a command-line environment, a desktop application, or an HTML frontend.

Other Online Trainings by Thomas Nield

SQL Fundamentals for Data

Intermediate SQL for Data Analytics

Intro to Mathematical Optimization

Machine Learning from Scratch