

# SQL: Programming

Introduction to Databases

CompSci 316 Spring 2019



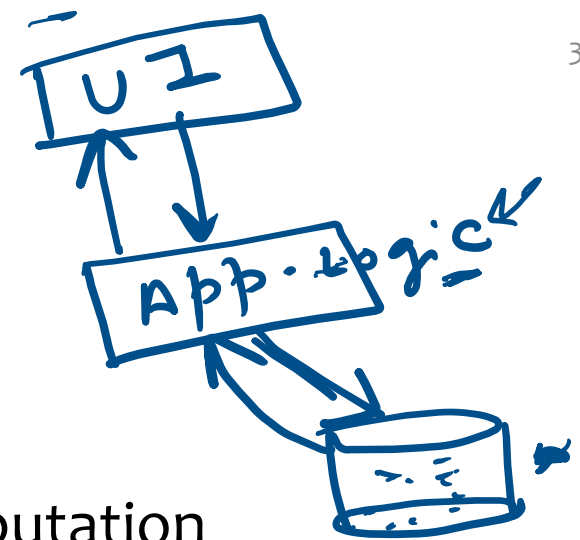
**DUKE**  
COMPUTER SCIENCE

# Announcements (Thu., Feb 21)

- Homework 2 Problem 1 due today
- Homework 2 Problems 2 due tomorrow
- Homework 2 Problems 4, 5, X1 due next Thu.
- Non-gradiance problems: 5% per hour late penalty
- Project milestone #1 due on Tuesday
  - Only one member per team needs to submit
  - Remember members.txt
- Zhengjie's office hour 2:30-3:30 tomorrow In LSRC D316 (next to elevator)

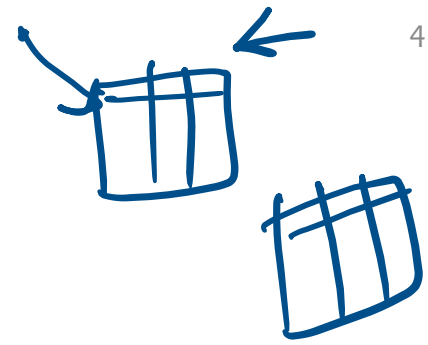
# Motivation

~ Sanitize



- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Not intended for general-purpose computation
- Solutions
  - Augment SQL with constructs from general-purpose programming languages
    - E.g.: SQL/PSM
  - Use SQL together with general-purpose programming languages: many possibilities
    - Through an API, e.g., Python psycopg2
    - Embedded SQL, e.g., in C
    - Automatic object-relational mapping, e.g.: Python SQLAlchemy
    - Extending programming languages with SQL-like constructs, e.g.: LINQ

# An “impedance mismatch”



- SQL operates on **a set of records at a time**
- Typical low-level general-purpose programming languages operate on **one record at a time**
  - Less of an issue for functional programming languages

## 👉 Solution: **cursor**

- **Open** (a result table): position the cursor before the first row
- **Get next**: move the cursor to the next row and return that row; raise a flag if there is no such row
- **Close**: clean up and release DBMS resources

👉 Found in virtually every database language/API

- With slightly different syntaxes

👉 Some support more positioning and movement options, modification at the current position, etc.

# Augmenting SQL: SQL/PSM

- PSM = Persistent Stored Modules
- CREATE PROCEDURE *proc\_name*(*param\_decls*)  
    *local\_decls*  
    *proc\_body*;
- CREATE FUNCTION *func\_name*(*param\_decls*)  
    RETURNS *return\_type*  
    *local\_decls*  
    *func\_body*;
- CALL *proc\_name*(*params*);
- Inside procedure body:  
    SET *variable* = CALL *func\_name*(*params*);

# SQL/PSM example

```
CREATE FUNCTION SetMaxPop(IN newMaxPop FLOAT)
  RETURNS INT
  -- Enforce newMaxPop; return # rows modified.
BEGIN
  DECLARE rowsUpdated INT DEFAULT 0;
  DECLARE thisPop FLOAT;

  -- A cursor to range over all users:
  DECLARE userCursor CURSOR FOR
    SELECT pop FROM User
  FOR UPDATE;

  -- Set a flag upon "not found" exception:
  DECLARE noMoreRows INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET noMoreRows = 1;

  ... (see next slide) ...

  RETURN rowsUpdated;
END
```

# SQL/PSM example continued

-- Fetch the first result row:

OPEN userCursor;

FETCH FROM userCursor INTO thisPop;

-- Loop over all result rows:

WHILE noMoreRows <> 1 DO

IF thisPop > newMaxPop THEN

-- Enforce newMaxPop:

UPDATE User SET pop = newMaxPop

WHERE CURRENT OF userCursor;

-- Update count:

SET rowsUpdated = rowsUpdated + 1;

END IF;

-- Fetch the next result row:

FETCH FROM userCursor INTO thisPop;

END WHILE;

CLOSE userCursor;

# Other SQL/PSM features

- Assignment using scalar query results
  - SELECT INTO
- Other loop constructs
  - FOR, REPEAT UNTIL, LOOP
- Flow control
  - GOTO
- Exceptions
  - SIGNAL, RESIGNAL

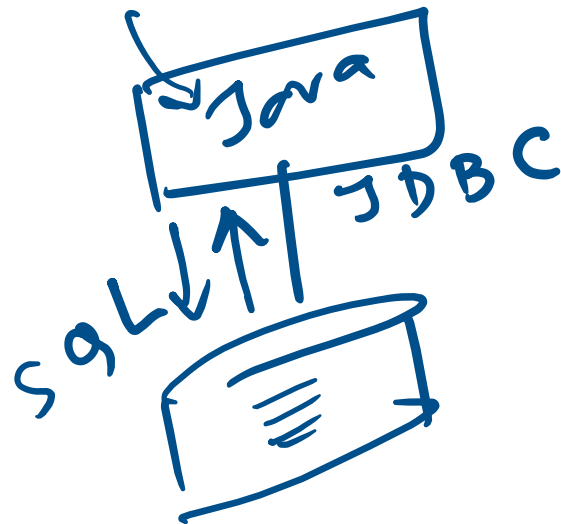
...

- For more PostgreSQL-specific information, look for “PL/pgSQL” in PostgreSQL documentation
  - Link available from course website (under **Help: PostgreSQL Tips**)



# Working with SQL through an API

- E.g.: Python psycopg2, JDBC, ODBC (C/C++/VB)
  - All based on the SQL/CLI (Call-Level Interface) standard
- The application program sends SQL commands to the DBMS at runtime
- Responses/results are converted to objects in the application program



# Example API: Python psycopg2

```
import psycopg2
conn = psycopg2.connect(dbname='beers')
cur = conn.cursor()
# list all drinkers:
cur.execute('SELECT * FROM Drinker')
for drinker, address in cur:
    print(drinker + ' lives at ' + address)
# print menu for bars whose name contains "a":
cur.execute('SELECT * FROM Serves WHERE bar LIKE %s', ('%a%',))
for bar, beer, price in cur:
    print('{} serves {} at ${:,.2f}'.format(bar, beer, price))
cur.close()
conn.close()
```

You can iterate over cur  
one tuple at a time

Placeholder for  
query parameter

Tuple of parameter values,  
one for each %s

(note that the trailing "," is needed when  
the tuple contains only one value)

# More psycopg2 examples

# "commit" each change immediately—need to set this option just once at the start of the session

```
conn.set_session(autocommit=True)
```

```
# ...
```

```
bar = input('Enter the bar to update: ').strip()
```

```
beer = input('Enter the beer to update: ').strip()
```

```
price = float(input('Enter the new price: '))
```

```
try:
```

```
    cur.execute('''
UPDATE Serves
SET price = %s
WHERE bar = %s AND beer = %s''', (price, bar, beer))
```

```
    if cur.rowcount != 1:
```

```
        print('{} row(s) updated: correct bar/beer?' \
              .format(cur.rowcount))
```

# of tuples modified

```
except Exception as e:
```

```
    print(e)
```

Exceptions can be thrown

(e.g., if positive-price constraint is violated)

# Prepared statements: motivation

```
while True:
```

```
    # Input bar, beer, price...
```

```
        cur.execute(''  
UPDATE Serves  
SET price = %s  
WHERE bar = %s AND beer = %s'', (price, bar, beer))
```

```
    # Check result...
```

- Every time we send an SQL string to the DBMS, it must perform parsing, semantic analysis, optimization, compilation, and finally execution
- A typical application issues many queries with a small number of patterns (with different parameter values)
- Can we reduce this overhead?

# Prepared statements: example

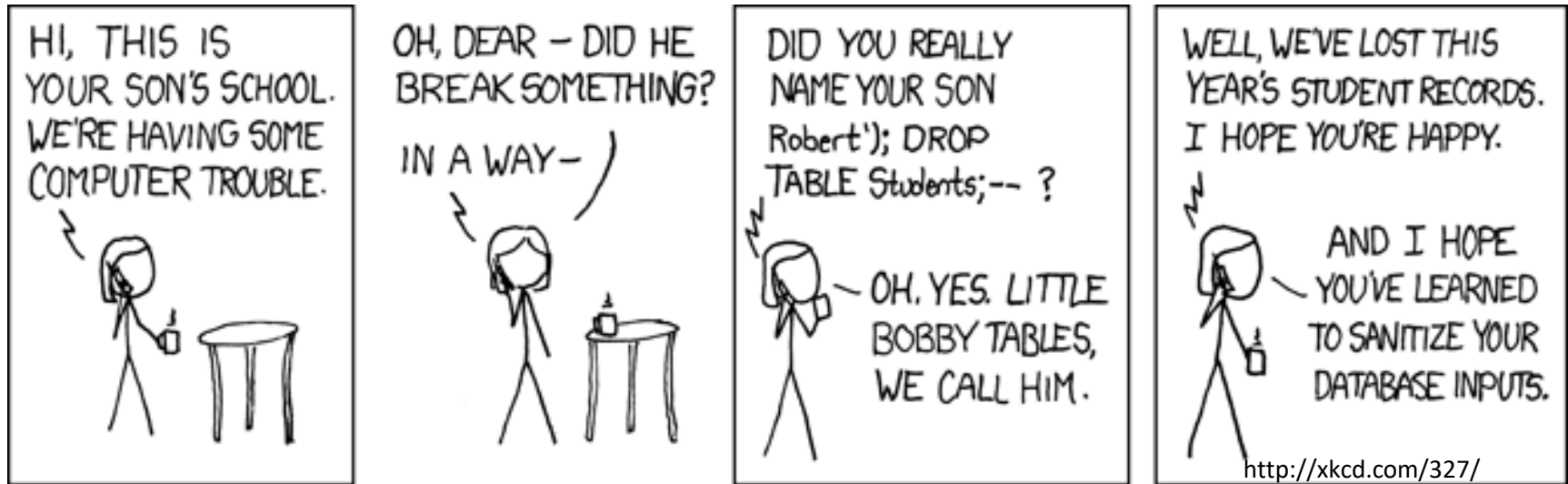
See `/opt/dbcourse/examples/psycopg2/`  
on your VM for a complete code example

```
cur.execute('''
PREPARE update_price AS
UPDATE Serves
SET price = $1
WHERE bar = $2 AND beer = $3''') # parameter placeholders.

while True:
    # Input bar, beer, price...
    cur.execute('EXECUTE update_price(%s, %s, %s)', \
                (price, bar, beer)) # Execute many times.
    # Note the switch back to %s for parameter placeholders.
    # Check result...
```

- The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it “prepares” the statement
- At execution time, the DBMS only needs to check parameter types and validate the compiled plan
- Most other API’s have better support for prepared statements than `psycopg2`
  - E.g., they would provide a `cur.prepare()` method

# “Exploits of a mom”



- The school probably had something like:

```
cur.execute("SELECT * FROM Students " + \  
            "WHERE (name = '" + name + "')")
```

where **name** is a string input by user

- Called an **SQL injection attack**

# Guarding against SQL injection

- Escape certain characters in a user input string, to ensure that it remains a single string
  - E.g., ' , which would terminate a string in SQL, must be replaced by " (two single quotes in a row) within the input string
- Luckily, most API's provide ways to “sanitize” input automatically (if you use them properly)
  - E.g., pass parameter values in psycopg2 through %s's

# If one fails to learn the lesson...



*... P.S. To Ashley Madison's Development Team:  
You should be embarrassed [sic] for your train  
wreck of a database (and obviously security), not  
sanitizing your phone numbers to your database  
is completely amateur, it's as if the entire site was  
made by Comp Sci 1XX students.*

*— Creators of CheckAshleyMadison.com*



# Augmenting SQL vs. API

- Pros of augmenting SQL:
  - More processing features for DBMS
  - More application logic can be pushed closer to data
    - Less data “shipping,” more optimization opportunities ⇒ more efficient
    - Less code ⇒ easier to maintain multiple applications
- Cons of augmenting SQL:
  - SQL is already too big—at some point one must recognize that SQL/DBMS are not for everything!
  - General-purpose programming constructs complicate optimization and make it impossible to guarantee safety

# A brief look at other approaches

- “Embed” SQL in a general-purpose programming language
  - E.g.: embedded SQL
- Support database features through an object-oriented programming language
  - By automatically storing objects in tables and translating methods to SQL
  - E.g., object-relational mappers (ORM) like Python SQLAlchemy
- Extend a general-purpose programming language with SQL-like constructs
  - E.g.: LINQ (Language Integrated Query for .NET)

# Embedding SQL in a language

## Example in C

```
EXEC SQL BEGIN DECLARE SECTION;
int thisUid; float thisPop;
EXEC SQL END DECLARE SECTION;
```

} Declare variables to be “shared”  
between the application and DBMS

```
EXEC SQL DECLARE ABCMember CURSOR FOR
  SELECT uid, pop FROM User
  WHERE uid IN (SELECT uid FROM Member WHERE gid = 'abc')
  FOR UPDATE;
```

```
EXEC SQL OPEN ABCMember;
EXEC SQL WHENEVER NOT FOUND DO break;
```

→ Specify a handler for  
NOT FOUND exception

```
while (1) {
  EXEC SQL FETCH ABCMember INTO :thisUid, :thisPop;
  printf("uid %d: current pop is %f\n", thisUid, thisPop);

  printf("Enter new popularity: ");
  scanf("%f", &thisPop);
  EXEC SQL UPDATE User SET pop = :thisPop
    WHERE CURRENT OF ABCMember;
}

EXEC SQL CLOSE ABCMember;
```

# Object-relational mapping

- Example: Python SQLAlchemy

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    password = Column(String)
```

```
class Address(Base):
    __tablename__ = 'addresses'
    id = Column(Integer, primary_key=True)
    email_address = Column(String, nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'))
```

```
Address.user = relationship("User", back_populates="addresses")
User.addresses = relationship("Address", order_by=Address.id, back_populates="user")
```

```
jack = User(name='jack', password='gjffdd')
jack.addresses = [Address(email_address='jack@google.com'),
                  Address(email_address='j25@yahoo.com')]
session.add(jack)
session.commit()
```

```
session.query(User).join(Address).filter(Address.email_address=='jack@google.com').all()
```

- Automatic data mapping and query translation
- But syntax may vary for different host languages
- Very convenient for simple structures/queries, but quickly get complicated and less intuitive for more complex situations

# Deeper language integration

- Example: LINQ (Language Integrated Query) for Microsoft .NET languages (e.g., C#)

```
int someValue = 5;
var results = from c in someCollection
               let x = someValue * 2
               where c.SomeProperty < x
               select new {c.SomeProperty, c.OtherProperty};
foreach (var result in results) {
    Console.WriteLine(result);
}
```

- Again, automatic data mapping and query translation
- Much cleaner syntax, but it still may vary for different host languages