SQL: Programming

Introduction to Database Management

CS348 Fall 2022

SQL

- Basic SQL (queries, modifications, and constraints)

- Intermediate SQL
  - Triggers
  - Views
  - Indexes

- Advanced SQL
  - Programming
  - Recursion (Optional, video only)

Motivation

- 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

☞ Solution: cursor
  - Open (a result table), Get next, Close
  - ☞ Found in virtually every database language/API
    - With slightly different syntaxes

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
```

Declare local variables

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;
```

Function body

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
  - https://www.postgresql.org/docs/9.6/plpgsql.html

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

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))
except Exception as e:
    print(e)
```

Perform passing, semantic analysis, optimization, compilation, and finally execution

More psycopg2 examples

```
....
while true:
# Input bar, beer, price…
    cur.execute('''
        UPDATE Serves
        SET price = %s
        WHERE bar = %s AND beer = %s''', (price, bar, beer))
    ....
    # Check result…
```

Perform passing, semantic analysis, optimization, compilation, and finally execution

Execute many times

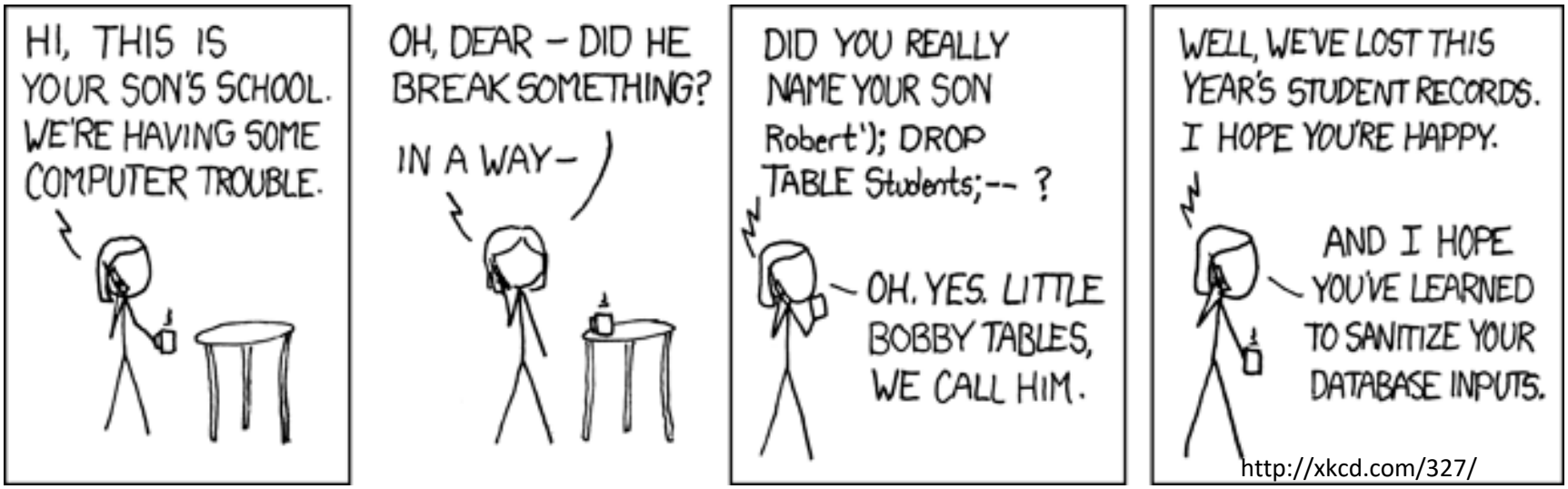
Can we reduce this overhead?

Prepared statements: example

```
cur.execute('''              # Prepare once (in SQL).
        PREPARE update_price AS        # Name the prepared plan,
        UPDATE Serves
        SET price = $1                 # and note the $1, $2, … notation for
        WHERE bar = $2 AND beer = $3''') # parameter placeholders.
while true:
# Input bar, beer, price…
    cur.execute('
        EXECUTE update_price(%s, %s, %s)',\ # Execute many times.
            (price, bar, beer))….
    # Check result…
```

Prepare only once

“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

Augmenting SQL vs. API

- Pros of augmenting SQL:
  - More processing features for DBMS
  - More application logic can be pushed closer to data

- Cons of augmenting SQL:
  - SQL is already too big
  - 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
  - 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;
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;
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;
```

Declare variables to be "shared" between the application and DBMS

Specify a handler for NOT FOUND exception

Embedded SQL v.s. API

- Pros of embedded SQL:
  - Be processed by a preprocessor prior to compilation → may catch SQL-related errors at preprocessing time
  - API: SQL statements are interpreted at runtime

- Cons of embedded SQL:
  - New host language code → complicate debugging

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
  - 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)

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

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)

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

Summary

- Basic SQL (queries, modifications, and constraints)
- Intermediate SQL(triggers, views, indexes)
- Programming
  - Augment SQL, e.g., SQL/PSM
  - Through an API, e.g., Python psycopg2, JDBC
  - Embedded SQL, e.g., in C
  - Automatic object-relational mapping, e.g.: Python SQLAlchemy
  - Extending programming languages with SQL-like constructs, e.g.: LINQ
- Recursion (Optional, video only)