

# SQL: Programming

Introduction to Databases

CompSci 316 Fall 2017



**DUKE**  
COMPUTER SCIENCE

# Announcements (Wed., Mar. 1)

- Keep working on the project!
- TA/UTA assignment to each project soon

# Today

- An overview of programming in SQL (13a)
- Start database internals with physical data organization (13b)

# 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
    - E.g.: Python DB API, JDBC, embedded SQL
  - Extend general-purpose programming languages with SQL-like constructs
    - E.g.: LINQ (Language Integrated Query for .NET)

# 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): 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** = **P**ersistent **S**tored **M**odules
- **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*);

# Parameters in PSM

- Triples mode-name-type
- name and type
  - similar to standard programming languages
- mode in a procedure
  - new
  - IN : parameter is input-only -- default
  - OUT : output only
  - INOUT : both input and output
- mode in a function
  - only can be IN
  - PSM forbids side effect in functions
  - only output through return values

# SQL/PSM example

User(uid, name, age, pop)

```
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 a variable with a name and type – local – value not preserved by the DBMS

Cursor ranges over the tuples of the relation produced by the query



# 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;
```

CURRENT OF refers to the current tuple that has been fetched

# 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**)

# Interfacing SQL with another language

- **API** approach
  - SQL commands are sent to the DBMS at runtime
  - Examples: Python DB API, JDBC, ODBC (C/C++/VB)
  - These API's are all based on the SQL/CLI (Call-Level Interface) standard
- **Embedded SQL** approach
  - SQL commands are embedded in application code
  - A **precompiler** checks these commands at compile-time and converts them into DBMS-specific API calls
  - Examples: embedded SQL for C/C++, SQLJ (for Java)

# 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
```

```
cur.close()
```

```
conn.close()
```

You can iterate over cur  
one tuple at a time



# Transactions in programming

Using psycopg2 as an example:

```
conn = psycopg2.connect(dbname='beers')
```

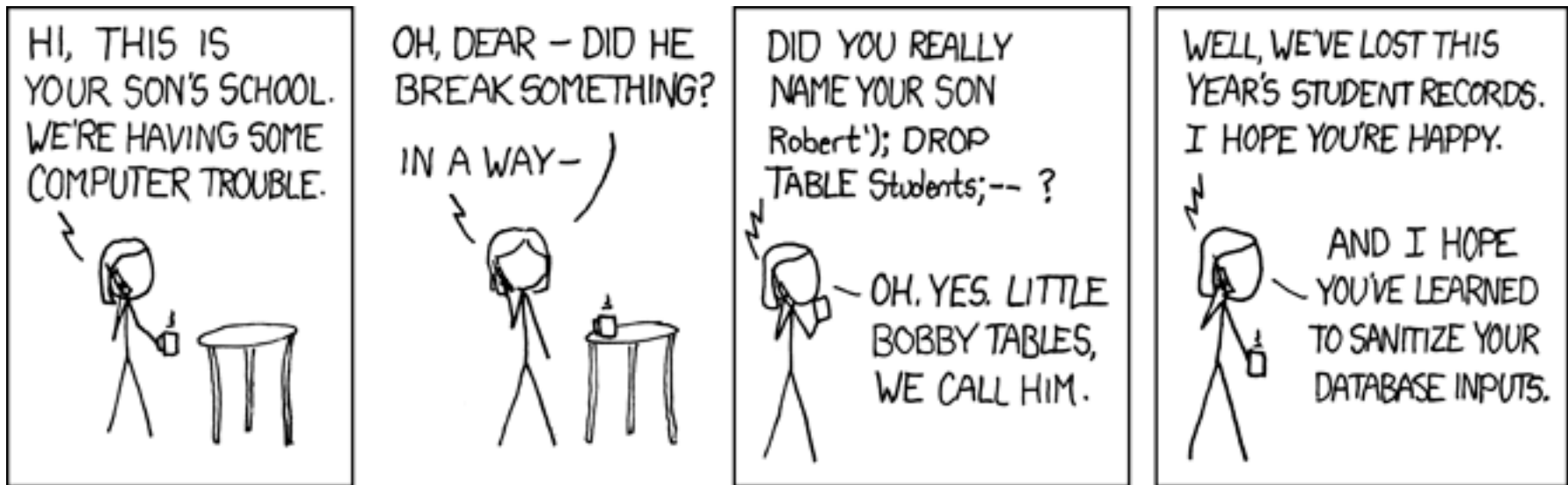
```
conn.set_session(isolation_level='SERIALIZABLE',  
                 ready_only=False,  
                 autocommit=True)
```

- `isolation_level` defaults to READ COMMITTED
- `read_only` defaults to False
- `autocommit` defaults to False
- When `autocommit` is False, `commit/abort` current transaction as follows:

```
conn.commit()
```

```
conn.rollback()
```

# “Exploits of a mom”



<http://xkcd.com/327/>

- 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
    - 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



# Takeaways

- Be aware of all options
- Need to learn them as you need by actually trying and coding