CS 348 Lecture 7

SQL Part 4

Semih Salihoğlu

Jan 27th, 2025

Announcements

- Assignment 1: Due January 31st
- Assignment 2: Out January 31st (due Feb 14)
- Project Milestone 1: See the Piazza note on the ER model background.

SQL features to cover in this lecture

- Views: Virtual tables
- WITH statement: Temporary tables
- Indexes
- Programming Applications With SQL

SQL features to cover in this lecture

- Views: Virtual tables
- WITH statement: Temporary tables
- Indexes
- Programming Applications With SQL

Views

- A view is like a “virtual” table
  - Contrasts with “base” tables, i.e., those added through CREATE TABLE statements.
  - Defined by a query, which describes how to compute the view contents on the fly
  - Stored as a query by DBMS instead of query contents
  - Can be used in queries just like a regular table

```
CREATE VIEW PopGroup AS
      SELECT * FROM User
      WHERE uid IN (SELECT uid
                         FROM Member
                         WHERE gid = ‘popgroup');
```

Base tables

```
SELECT AVG(pop)
FROM (SELECT * FROM User
         WHERE uid IN
         (SELECT uid FROM Member
          WHERE gid = ‘popgroup'))
          AS popGroup;
```

```
SELECT AVG(pop) FROM PopGroup;
```

```
SELECT MIN(pop) FROM PopGroup;
```

```
SELECT … FROM PopGroup;
```

```
DROP VIEW popGroup;
```

Why use views?

- To hide complexity from users

- To hide data from users

- Logical data independence

- To provide a uniform interface

Exercises

Consider this db instance:

User

uid	*name*	*age*	*pop*
142	Bart	10	0.9
123	Milhouse	10	0.2
857	Lisa	8	0.7
456	Ralph	7	0.3

Member

uid	*gid*
857	dps
123	gov
857	abc
857	gov
456	abc
456	gov

- What is the output of these queries?

```
CREATE VIEW ageGroups(age,cnt) AS
            (SELECT age, COUNT(*) FROM User GROUP BY age)
```

```
SELECT * FROM ageGroups;
```

```
SELECT age FROM ageGroups
WHERE cnt = (SELECT MAX(cnt) FROM ageGroups);
```

Exercises

User (*uid* int, *name* string, *age* int, *pop* float)
Group (*gid* string, *name* string)
Member (*uid* int, *gid* string)

- Assume there is a CHECK constraint on User table s.t. (*age > 0 and age < 140*)

```
CREATE VIEW youngUsers AS
        (SELECT * FROM User WHERE age < 25) WITH CHECK OPTION;
```

- What happens to the following statements?

```
INSERT INTO youngUsers VALUES (835, 'Alex', 30, 0.2);
```

```
INSERT INTO youngUsers VALUES (923, 'James', 150, 0.3);
```

Storing Views: Materialized views

- Some systems allow view relations to be stored in db
  - If the actual relations used in the view definition change, the view is kept up-to-date
- Such views are called materialized views

- Why? Because of several performance reasons:
  - Views are results of SQL queries

  1. No query is faster than an already computed one: answering the query is equivalent to just scanning the computed "materialized view"
  2. If the query is asked multiple times, we can avoid recomputing views each time

- View maintenance: updating the materialized view upon base table changes
  - Immediately or lazily, up to the DBMS
  - Fascinating, challenging & still active research problem

Can we modify views directly?

- Does it even make sense, since views are virtual?

- It does make sense if we want users to really see views as tables

- Goal: modify the base tables such that the modification would appear to have been accomplished on the view

A simple case

```
CREATE VIEW UserPop AS
            SELECT uid, pop FROM User;
```

```
DELETE FROM UserPop WHERE uid = 123;
```

translates to:

```
DELETE FROM User WHERE uid = 123;
```

An impossible case

```
CREATE VIEW PopularUser AS
        SELECT uid, pop FROM User
        WHERE pop >= 0.8;
```

```
INSERT INTO PopularUser VALUES(987, 0.3);
```

- No matter what we do on *User*, the inserted row will not be in *PopularUser*

A case with too many possibilities

```
CREATE VIEW AveragePop(pop) AS
        SELECT AVG(pop) FROM User;
```

Renamed column

```
UPDATE AveragePop SET pop = 0.5;
```

- Set everybody’s *pop* to 0.5?
- Adjust everybody’s *pop* by the same amount?
- Just lower one user’s *pop*?

SQL92 updateable views

- More or less just single-table selection queries
  - No join
  - No aggregation or group by
  - No subqueries
  - Attributes not listed in SELECT must be nullable

- Arguably somewhat restrictive
- Still might get it wrong in some cases
  - See the slide titled "An impossible case"
  - Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications

SQL features to cover in this lecture

- Views: Virtual tables
- WITH statement: Temporary tables
- Indexes
- Programming Applications With SQL

WITH clause

- WITH clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs
- Think of this as an “on-the-fly” view only for a single query

- Ex: List group ids of users with age > 10 and pop < 0.5

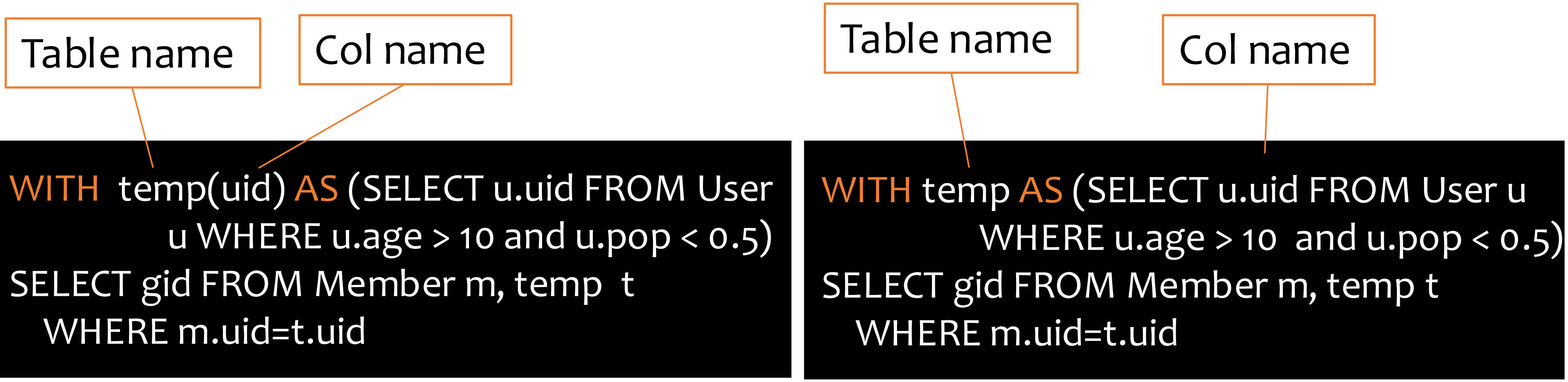


Table name

Col name

```
WITH  temp(uid) AS (SELECT u.uid FROM User
            u WHERE u.age > 10 and u.pop < 0.5)
SELECT gid FROM Member m, temp  t
   WHERE m.uid=t.uid
```

Table name

Col name

```
WITH temp AS (SELECT u.uid FROM User u
              WHERE u.age > 10  and u.pop < 0.5)
SELECT gid FROM Member m, temp t
   WHERE m.uid=t.uid
```

- Supported by many but not all DBMSs
- Can be written using subqueries but can simplify your sub-queries (in some systems can even refer to a not yet defined outer query variabl

WITH clause

```
SELECT *
FROM Users
WHERE EXISTS (SELECT * FROM Members
               WHERE Members.uid = Users.uid)
```

can in many systems equivalently be written as:

```
WITH tmp AS (SELECT * FROM Members
              WHERE Members.uid = Users.uid)

SELECT *
FROM Users
WHERE EXISTS (SELECT * FROM tmp)
```

Note that temporary tables are tables, so you need to use them as tables:
WHERE EXISTS (SELECT * FROM tmp) above.
You cannot do WHERE EXISTS (tmp) => this is not valid SQL syntax, since tmp
is a table; it's not a string substitution for "SELECT * FROM Members
WHERE Members.uid = Users.uid"

SQL features to cover in this lecture

- Views: Virtual tables
- WITH statement: Temporary tables
- Indexes
- Programming Applications With SQL

Motivating examples of using indexes

```
SELECT * FROM User WHERE name = 'Bart';
```

- Can we go “directly” to rows with *name*='Bart’ instead of scanning the entire table?

  → index on *User.name*

```
SELECT * FROM User, Member
WHERE User.uid = Member.uid AND Member.gid = ‘popgroup';
```

- Can we find relevant *Member* rows “directly”?

  → index on *Member.gid*

- For each relevant *Member* row, can we “directly” look up *User* rows with matching *Member.uid*

  → index on *User.uid*

Indexes

- An index is an auxiliary persistent data structure that helps with efficient searches
    - Search tree (e.g., B^+-tree), lookup table (e.g., hash table), etc.
    - ☞ More on indexes later in this course!

- CREATE [UNIQUE] INDEX $indexname$ ON $tablename(columnname_1,\ldots,columnname_n)$;
    - With UNIQUE, the DBMS will also enforce that $\{columnname_1, \ldots, columnname_n\}$ is a key of $tablename$
    - *So it is same behavior as creating an index + a unique constraint on* $\{columnname_1, \ldots, columnname_n\}$
- DROP INDEX $indexname$;
- Typically, the DBMS will automatically create indexes for PRIMARY KEY and UNIQUE constraint declarations

Indexes

- An index on $R.A$ can speed up accesses of the form
  - $R.A \ = \ value$
  - sometimes, if it is tree-based also: $R.A \ > \ value$ (or <, <=, >=)

- An index on $(R.A_1, \dots, R.A_n)$ can speed up
  - $R.A_1 = value_1 \wedge \cdots \wedge R.A_n = value_n$
  - $(R.A_1, \dots, R.A_n) > (value_1, \dots, value_n)$ (again depending on the index type)

Questions (will be discussed in the 2nd half of course):

☞Ordering of index columns is important—is an index on $(R.A, R.B)$ equivalent to one on $(R.B, R.A)$?

☞How about an index on $R.A$ plus another on $R.B$?

☞More indexes = better performance?

SQL features to cover in this lecture

- Views: Virtual tables
- WITH statement: Temporary tables
- Indexes
- Programming Applications With SQL

Programming Applications W/ SQL

- Challenge of using SQL on a real app:
    - Not intended for general-purpose computation
    - E.g.: No while or for loops, standard conditionals, arbitrary functions
- Solutions
    - Augment SQL with constructs from general-purpose programming languages
        - E.g.: SQL/PSM (Persistent Stored Modules)
    - Use SQL together with general-purpose programming languages: many possibilities
        - Through an API ← You will use this in practice.
        - Embedded SQL, e.g., in C
    - SQL generating approaches: Web Programming Frameworks (e.g., Django)

And this

1) Augmenting SQL: SQL/PSM

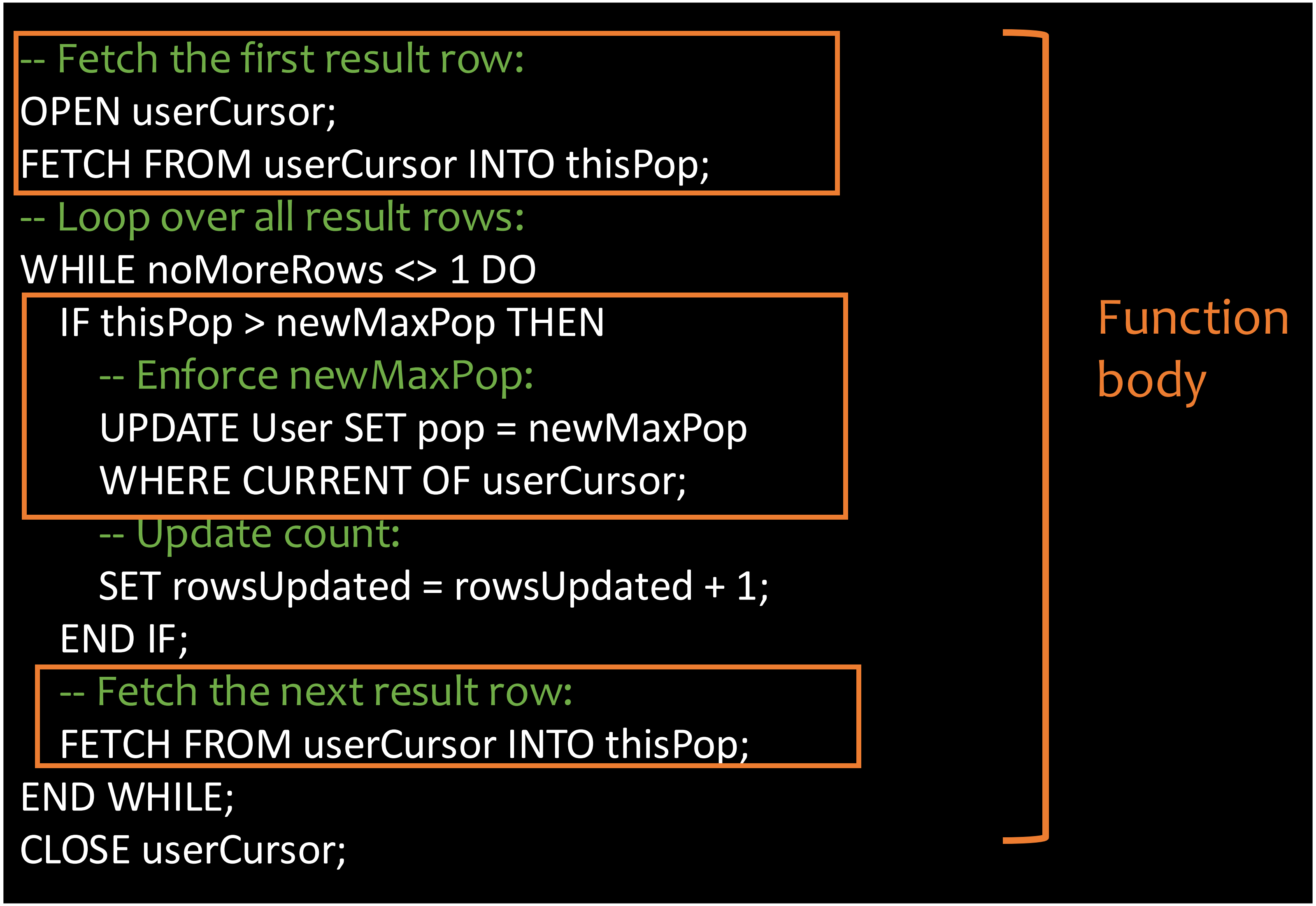
- An ISO standard to extend SQL to an advanced prog. lang.
  - Control flow, exception handling, etc.
- Several systems adopt SQL/PSM partially (e.g. MySQL, PostgreSQL)
- PSM = Persistent Stored Modules
- CREATE PROCEDURE *proc_name*(*param_decls*)
          local_decls
          proc_body;
- CREATE 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



```
-- 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
- Ultimately: Not very popular nowadays.

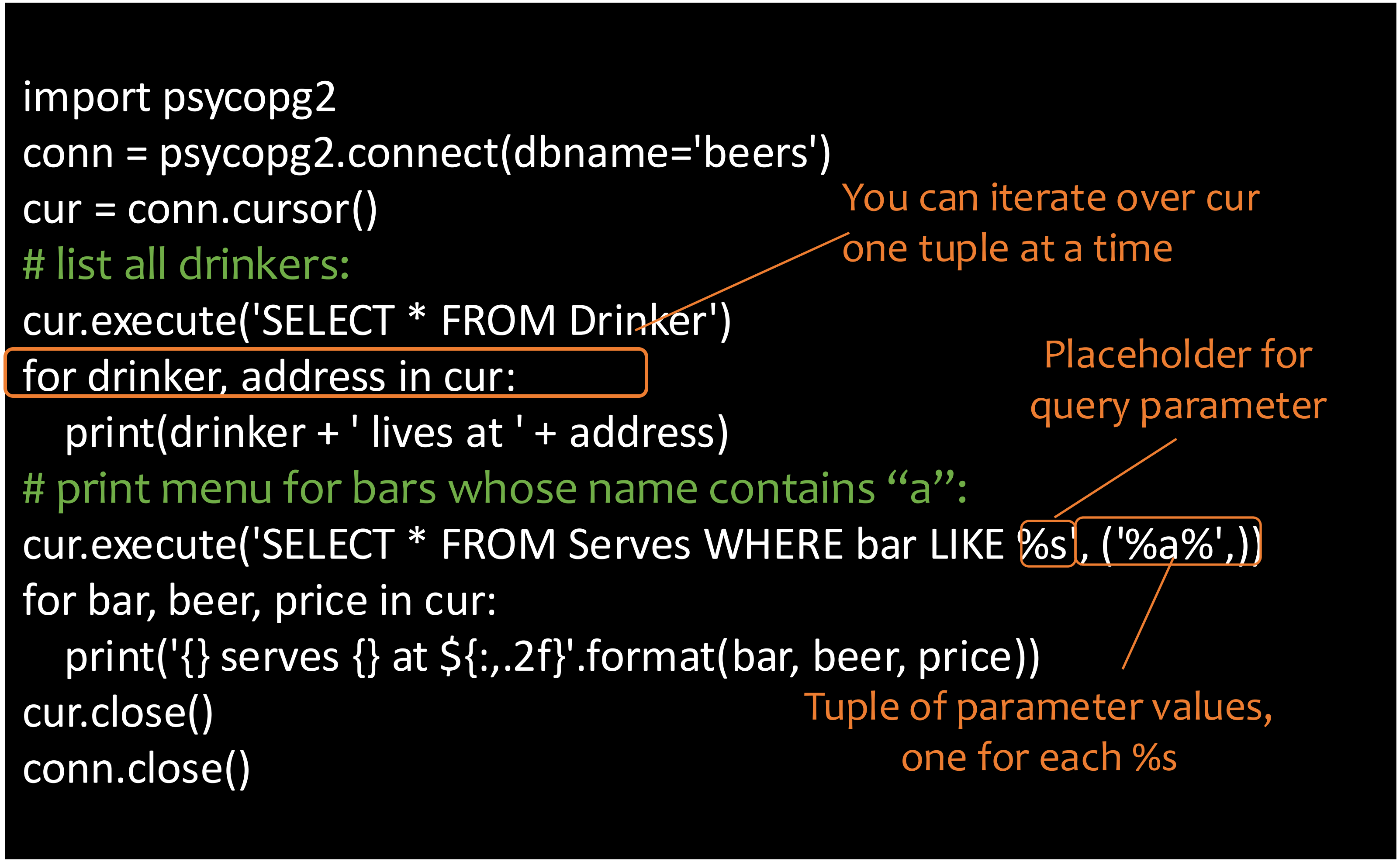
2) Working with SQL through an API

- E.g.: Python psycopg2, JDBC, ODBC (C/C^{++}/VB)
  - Based on the SQL/CLI (Call-Level Interface) standard
- The application program sends SQL commands to the DBMS at runtime. Gets back a “cursor” that can iterate over results.
- Results are converted to objects in the application program. Often you use a cursor to loop through result tuples.
- In Assignment 2: You will work with JDBC API for Java applications (standard for many DBMSs).
- Next we cover an API for Python for PostgreSQL.

2) Working with SQL through an API

- Functionalities provided in these APIs:
  - Connect/disconnect to a DBMS => get a connection object
  - Execute SQL queries
  - Iterate over result tuples (e.g., cursors) and access attributes of tuples
  - Begin/commit/rollback transactions
  - …

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

- ➢Different APIs have different interfaces (e.g,. JDBC), so need to read their documentations.

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

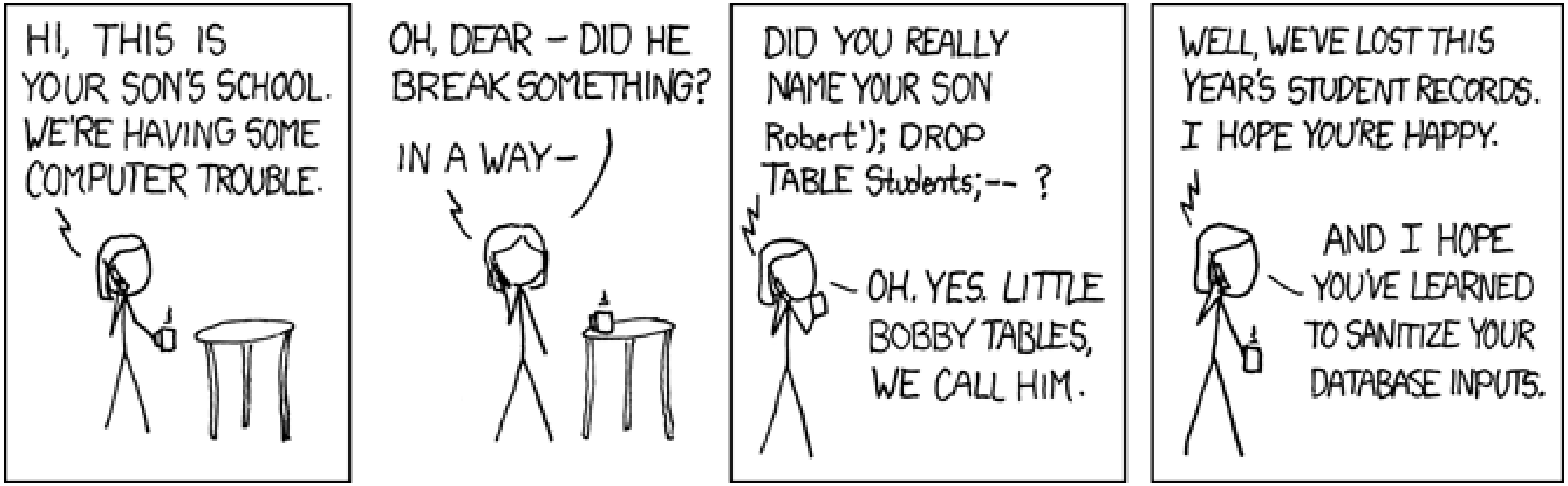
Can we reduce this overhead?

Prepared Statements: Example

```
cur.execute('''              # Prepare once (in SQL).   Prepare only once
        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...
```

- ➢Again: different APIs have different functions to implement prepared statements; so need to read their documentations.

Watch Out For SQL Injection Attacks!


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.

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. Most APIs have ways to sanitize inputs.

Augmenting SQL vs. Programming Through an 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

3) "Embedding" SQL in a host language

- Can be thought of as the opposite of SQL/PSM
- Extends a host language, e.g., C or Java, with SQL-based features
- Can compile host language together with SQL statements and catch SQL errors during *application compilation time*

4) Web Programming Frameworks

- A web development “framework” e.g., Django or Ruby on Rails
- Very frequent approach to web apps that need a DB
- For most parts, no explicitly writing of SQL is needed:
- Example: Django Web App Programming:
  - Define “Models”: python objects and only do oo programming
  - Models will be backed up with Relations in an RDBMS
- E.g.: a Person class/object with first and lastName:

```
from django.db import models

class Person(models.Model):
f_name = models.CharField(max_len=30)
l_name = models.CharField(max_len=30)
```

```
CREATE TABLE myapp_person (
"id" serial NOT NULL PRIMARY KEY,
"f_name" varchar(30) NOT NULL,
"l_name" varchar(30) NOT NULL );
```

- Would lead the “framework” (not the user) to generate the following SQL code somewhere in the web application files:

Thank You