Announcements

• Assignment 1 due by 11:59PM tonight!
  • Submit via CrowdMark
SQL

• Basic SQL (queries, modifications, and constraints)

• Intermediate SQL
  • Triggers
  • Views
  • Indexes

• Advanced SQL
  • Programming
  • Recursion
Motivation

• Pros and cons of SQL
  • Very high-level, possible to optimize
  • Not intended for general-purpose computation

• Can SQL and general-purpose programming languages (PL) interact with each other?
  YES!!

  Dynamic SQL
  Build SQL statements at runtime using APIs provided by DBMS

  Embedded SQL
  SQL statements embedded in general-purpose PL; identified at compile time
A mismatch b/w SQL and PLs

• 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
_solution: Found in virtually every database language/API
  • With slightly different syntaxes
Dynamic SQL: 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

```python
import psycopg2
conn = psycopg2.connect(host="db.uwaterloo.ca", port=5432,
dbname="membership", user='u1', password='passwd1'))
cur = conn.cursor()
.....
```

Connect to the database

An object used to query db & get results
import psycopg2
conn = psycopg2.connect(host="db.uwaterloo.ca", port=5432,
dbname="membership", user='u1', password='passwd1')
cur = conn.cursor()
# list all groups:
cur.execute('SELECT * FROM Group')
for gid, name in cur:
    print('Group ' + gid + ' has name ' + name)
# print users whose name contains “a”:
cur.execute('SELECT name, pop FROM User WHERE name LIKE %s', ('a%'))
for name, pop in cur:
    print('{} has a popularity of {}'.format(gid, name))
cur.close()
conn.close()
More psycopg2 examples

```python
# "commit" each change immediately—need to set this option just once at
# the start of the session
conn.set_session(autocommit=True)
# ...
uid = input('Enter the user id to update: ').strip()
name = input('Enter the name to update: ').strip()
pop = float(input('Enter new pop: '))
try:
cur.execute("UPDATE User
    SET pop = %s
    WHERE uid = %s AND name = %s",
    (pop, uid, name))
print('{} row(s) updated'.format(cur.rowcount))
except Exception as e:
    print(e)
```
Perform parsing, semantic analysis, optimization, compilation, and finally execution
while true:
    # Input uid, name, pop...
    cur.execute('''
        UPDATE User
        SET pop = %s
        WHERE uid = %s AND name = %s''', (pop, uid, name))

    # Check result...

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

Execute many times
Can we reduce this overhead?
cur.execute('''
# Prepare once (in SQL).  Prepare only once
PREPARE update_pop AS
UPDATE User
SET pop = $1
WHERE uid = $2 AND name = $3''')  # Name the prepared plan,
# and note the $1, $2, ... notation for
# parameter placeholders.

while true:
# Input uid, name, pop

cur.execute(''
EXECUTE update_pop(%s, %s, %s),'',
(pop, uid, name))....

# Check result...
Prepared statements: example (JDBC)

Specific API provided by the driver

```java
PreparedStatement pStmt = conn.prepareStatement("insert into user values(?,?,?,?)");
pStmt.setInt(1, 678);
pStmt.setString(2, "Bart");
pStmt.setFloat(3, 0.6);
pStmt.setInt(4, 10);
pStmt.executeUpdate();
```
“Exploits of a mom”

• The school probably had something like:

```
SELECT * FROM Students
WHERE (name = 'Bart')
```

```
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

• Luckily, most API’s provide ways to “sanitize” input automatically when using prepared statements (%s)
  • E.g., user input for name= " Robert’);Drop table students; ”
    • SELECT * FROM Students WHERE (name =‘Robert\’;Drop table students;’)
    • Returns empty relation

• Some systems limit only one SQL query per API call
So far in programming

- Dynamic SQL
- Augmented SQL
- Embedded SQL
Augmenting SQL: functions & procedures

• Procedures and functions allow business logic to be stored in db and executed from SQL statements

• 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);
Creating function in SQL

create function dept_count(dept_name varchar(20))
returns integer
begin
  declare d_count integer;
  select count( *) into d_count
  from instructor
  where instructor.dept_name= dept_name
  return d_count;
end

Declaring variables and defining the function

Writing an SQL query to get desired results

select dept_name, budget
from department
where dept_count(dept_name) > 12;

Invoking the function: returns dept. names & budgets for all depts with > 12 instructors
Creating a procedure in SQL

• Functions used to calculate something based on inputs; procedure are precompiled statements to perform some tasks in a specified order

```
cREATE PROCEDURE dept_count_proc
(IN dept_name VARCHAR(20),
OUT d_count INTEGER)
BEGIN
  SELECT COUNT(*) INTO d_count
  FROM instructor
  WHERE instructor.dept_name = dept_count_proc.dept_name
END
```

Invoking the procedure (either from another procedure or embedded SQL)

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```
Other SQL features

• Conditional constructs
  • IF, IF ELSIF ELSE

• Loop constructs
  • FOR, REPEAT UNTIL, LOOP

• Flow control
  • GOTO

• Exceptions
  • SIGNAL, RESIGNAL

... Read DMBS manual for more details!
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

• Augmented SQL is not commonly used
Embedded SQL (optional)

• “Embed” SQL in a general-purpose programming language
• A language in which SQL queries are embedded is referred to as a host language
• The SQL structures permitted in the host language constitute embedded SQL
• To identify embedded SQL requests to the preprocessor, we use the “exec SQL” statements.
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')
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;

Example in C
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
  • Need a preprocessor s/w
So far

• Basic SQL (queries, modifications, and constraints)
• Intermediate SQL (triggers, views, indexes)
• Programming

• Recursion
A motivating example

**Parent** (*parent, child*)

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Orville</td>
<td>Abe</td>
</tr>
</tbody>
</table>

- Example: find Bart’s ancestors
- “Ancestor” has a recursive definition
  - $X$ is $Y$’s ancestor if
    - $X$ is $Y$’s parent, or
    - $X$ is $Z$’s ancestor and $Z$ is $Y$’s ancestor
Recursion in SQL

• SQL2 had no recursion
  • You can find Bart’s parents, grandparents, great grandparents, etc.

```sql
SELECT p1.parent AS grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent
AND p2.child = 'Bart';
```

• But you cannot find all his ancestors with a single query

• SQL3 introduced recursion
  • `WITH RECURSIVE` clause
  • Many systems support recursion but limited functionality
Ancestor query in SQL3

WITH RECURSIVE
Ancestor(anc, desc) AS
((SELECT parent, child FROM Parent)
UNION
(SELECT a1.anc, a2.desc
FROM Ancestor a1, Ancestor a2
WHERE a1.desc = a2.anc))
SELECT anc
FROM Ancestor
WHERE desc = 'Bart';
WITH RECURSIVE
Ancestor(anc, desc) AS
base case
(SELECT parent, child FROM Parent)
UNION
recursive step
(SELECT a1.anc, a2.desc
FROM Ancestor a1, Ancestor a2
WHERE a1.desc = a2.anc)
.....;

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Orville</td>
<td>Abe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Orville</td>
<td>Abe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abe</td>
<td>Bart</td>
</tr>
<tr>
<td>Abe</td>
<td>Lisa</td>
</tr>
<tr>
<td>Orville</td>
<td>Homer</td>
</tr>
<tr>
<td>Orville</td>
<td>Lisa</td>
</tr>
</tbody>
</table>
Fixed point of a function

• If \( f: D \rightarrow D \) is a function from a type \( D \) to itself, a
  **fixed point** of \( f \) is a value \( x \) such that \( f(x) = x \)
  • Example: what is the fixed point of \( f(x) = x/2? \)
  • Ans: 0, as \( f(0) = 0 \)

• To compute a fixed point of \( f \)
  • Start with a “seed”: \( x \leftarrow x_0 \)
  • Compute \( f(x) \)
    • If \( f(x) = x \), stop; \( x \) is fixed point of \( f \)
      • (Similar to base case in recursive prog.)
    • Otherwise, \( x \leftarrow f(x) \); repeat
Fixed point of a query

- A query $q$ is just a function that maps an input table to an output table, so a **fixed point** of $q$ is a table $T$ such that $q(T) = T$

- To compute fixed point of $q$
  - Start with executing the base query: $T ← \text{base query}$
  - Evaluate $q$ over $T$
    - If the result is identical to $T$, stop; $T$ is a fixed point
    - Otherwise, let $T$ be the new result; repeat

- **Fixed point**: there is no further change in the result of the recursive query evaluation
- **Fixed point indicates when the evaluation of the recursive query terminates**
Restrictions on recursive queries

• A recursive query \( q \) must be monotonic
  • If input changes, old output should still be valid
• If more tuples are added to the recursive relation, \( q \) must return at least the same set of tuples as before, and possibly return additional tuples

• The following is not allowed in \( q \):
  • Aggregation on the recursive relation
  • NOT EXISTS in generating the recursive relation
  • Set difference (EXCEPT) whose right-hand side uses the recursive relation
Summary

• Basic SQL (queries, modifications, and constraints)
• Intermediate SQL (triggers, views, indexes)
• Programming

• Recursion

• Next 2 lectures: DB design (E/R diagrams)