SQL: Indexes, Programming, Recursion

CS348 Spring 2024

Instructor: Sujaya Maiyya

Sections: **002 & 003 only**

Announcements

- Assignment 1 due by 11:59PM tonight!
  - Submit via CrowdMark/Marmoset

SQL features covered so far

- Basic SQL

- Intermediate SQL
  - Triggers
  - Views
  - Indexes

- Advanced SQL
  - Programming
  - Recursion

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 *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*($columnname_1$,...,$columnname_n$);
  - With UNIQUE, the DBMS will also enforce that $\{columnname_1, \dots, columnname_n\}$ is a key of *tablename*
- 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
  - $R.A \ = \ value$
  - $R.A \ > \ value$ (sometimes; depending on the index type)

- 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 depends)

Questions (lecture 12):

☞ 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

- 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

☞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++)
  - 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

Connect to the database

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

An object used to query db & get results

Example API: Python psycopg2

```
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(name,pop))
conn.commit()
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

Commit the changes, if any, and close the cursor and the DB connection

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)
# ...
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

More psycopg2 examples

```
....
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

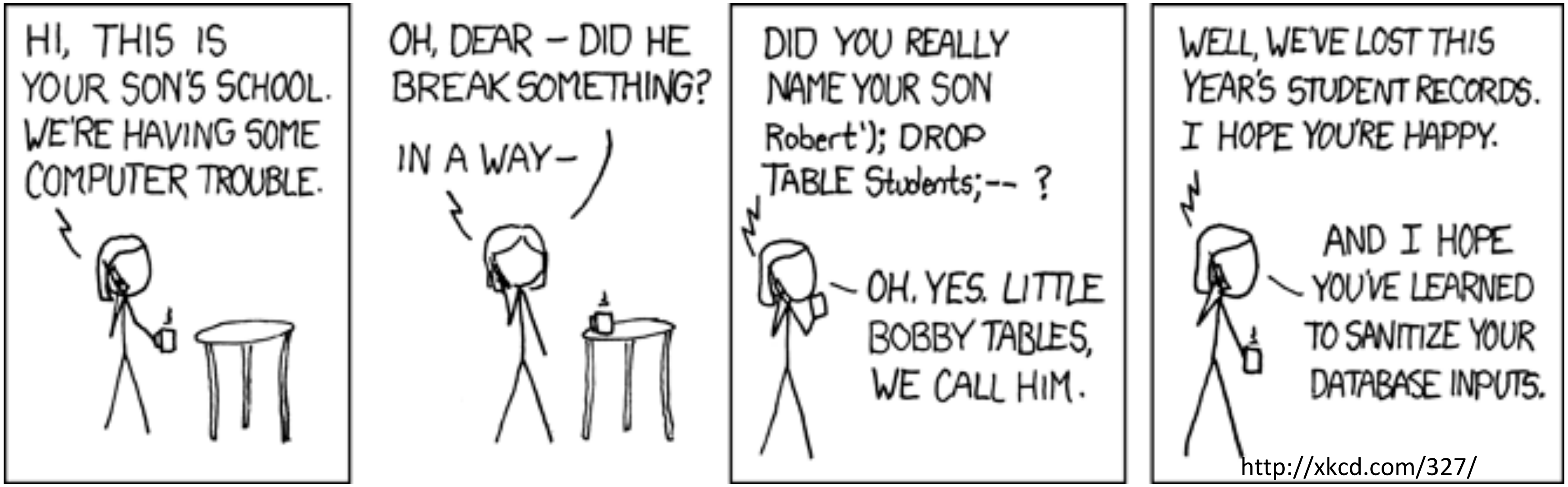
Execute many times

Can we reduce this overhead?

Prepared statements: example

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

"Exploits of a mom"



http://xkcd.com/327/

- 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

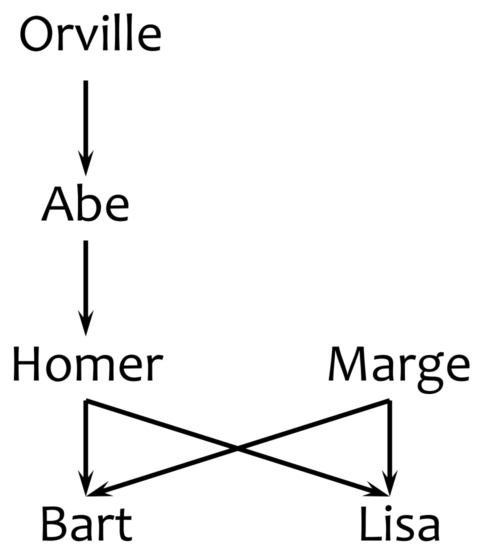
- Basic SQL (queries, modifications, and constraints)
- Intermediate SQL(triggers, views, indexes)
- Programming
  - (Optional slides on course website on Embedded and Augmented SQL)

- Recursion

A motivating example

Parent (*parent*, *child*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe



- 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.

```
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';
```

base case

a1.anc (X) → a1.desc(Z)
a2.anc (Z) → a2.desc (Y)

recursion step

Define a relation recursively

Query using the relation defined in WITH clause

Finding ancestors

```
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))
.....;
```

base case (SELECT parent, child FROM Parent)

recursive step (SELECT a1.anc, a2.desc FROM Ancestor a1, Ancestor a2 WHERE a1.desc = a2.anc)

Parent table

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe

Ancestor table

anc	*desc*

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe
Abe	Bart
Abe	Lisa
Orville	Homer

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Orville	Abe
Abe	Bart
Abe	Lisa
Orville	Homer
Orville	Bart
Orville	Lisa

Fixed point of a function

- If $f: D \to D$ 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: $T \leftarrow base\ query$
  - 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

Lecture 3

- 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/NOT IN 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)