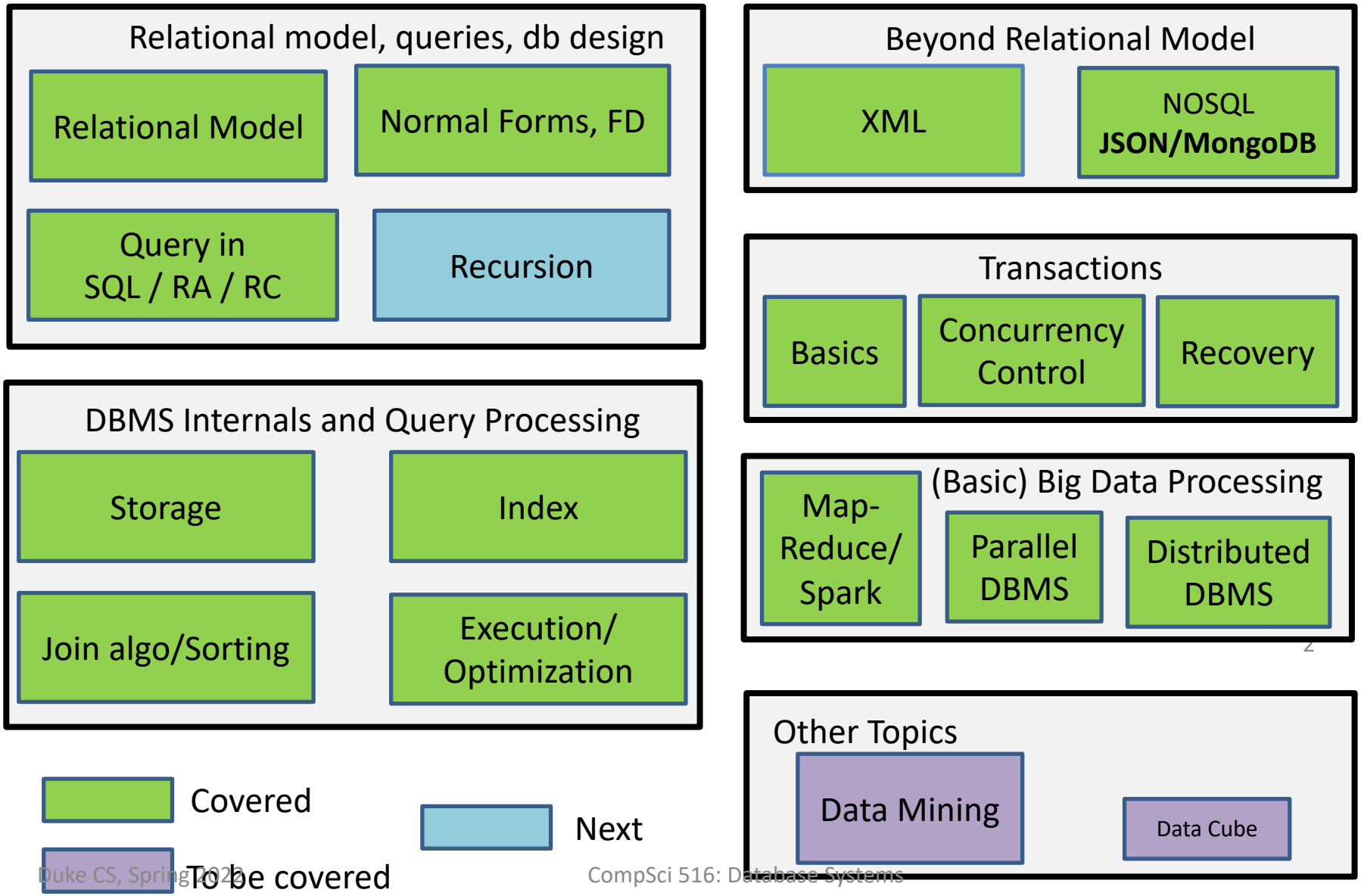
CompSci 516
Database Systems

Lecture 25
Recursive Query Evaluation
Datalog
And Views

Instructor: Sudeepa Roy

Where are we now?



Today

- Semantic of recursion in databases
- Recursion in SQL
- Datalog
  - Another language for recursion in database queries
- Views

Reading Material: Datalog

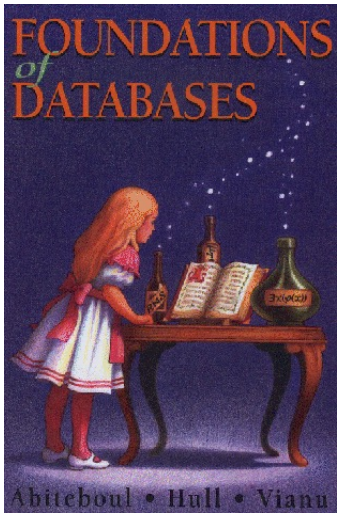
Optional:

1. The datalog chapters in the "Alice Book"

Foundations of Databases

Abiteboul-Hull-Vianu

Available online: http://webdam.inria.fr/Alice/



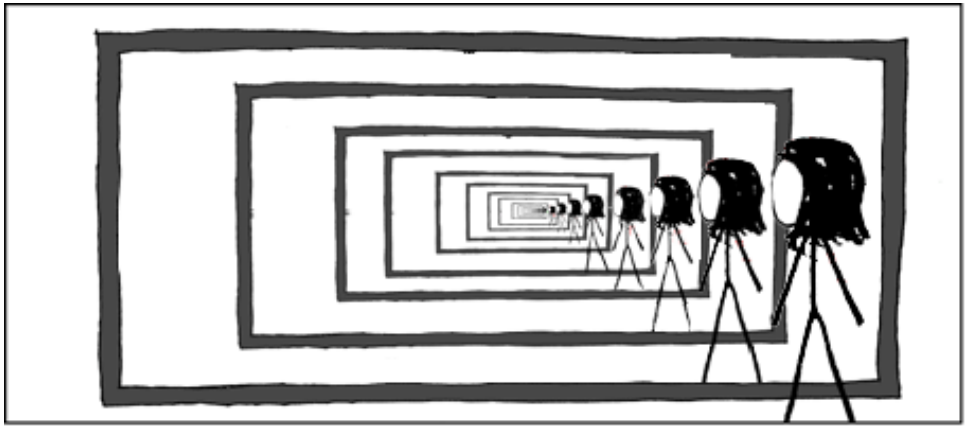
2. Datalog tutorial

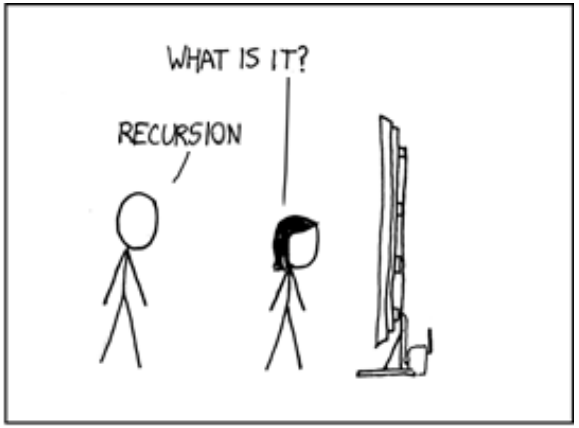
SIGMOD 2011

"Datalog and Emerging Applications: An Interactive Tutorial"

Acknowledgement:
Some of the slides have been borrowed from slides by Prof. Jun Yang

Recursion!



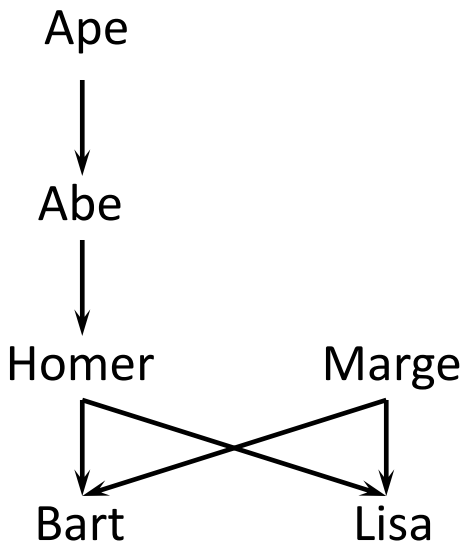


http://xkcdsw.com/1105

A motivating example

Parent (*parent*, *child*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	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
  - No RA/RC expressions can express ANCESTOR or REACHABILITY (TRANSITIVE CLOSURE in a graph) [Aho-Ullman, 1979]

Recursion in Databases

- What can we do to overcome the limitation?

1. Embed SQL in a high-level language supporting recursion
   - (-) destroys the high-level declarative characteristic of SQL
2. Augment RC with a high-level declarative mechanism for recursion
   - Datalog (Chandra-Harel, 1982)

- SQL2 had no recursion
- SQL:1999 (SQL3) and later versions support "linear Datalog"
  - WITH Clause
  - In Postgres

Brief History of Datalog

- Motivated by Prolog – started back in 80's – then quiet for a long time

- A long argument in the Database community whether recursion should be supported in query languages
  - *"No practical applications of recursive query theory ... have been found to date"*—Michael Stonebraker, 1998
    Readings in Database Systems, 3rd Edition Stonebraker and Hellerstein, eds.
  - Recent work by Hellerstein et al. on Datalog-extensions to build networking protocols and distributed systems. [Link]

Datalog is resurging

- Number of papers and tutorials in DB conferences
- Applications in
  - data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing
- Systems supporting datalog in both academia and industry:
  - Lixto (information extraction)
  - LogicBlox (enterprise decision automation)
  - Semmle (program analysis)
  - BOOM/Dedalus (Berlekey)
  - Coral
  - LDL++

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

recursion step

Define a relation recursively

Query using the relation defined in WITH clause

Fixed point of a function

- If $f: T \rightarrow T$ is a function from a type T to itself, a fixed point of f is a value x such that $f(x) = x$

- Example: What is the fixed point of $f(x) = x/2$?
  - 0, because $f(0) = 0/2 = 0$

To compute fixed point of a function f

- Start with a "seed": $x \leftarrow x_0$
- Compute $f(x)$
  - If $f(x) = x$, stop; x is fixed point of f
  - Otherwise, $x \leftarrow f(x)$; repeat

- Example: compute the fixed point of $f(x) = x/2$
  - With seed 1: 1, 1/2, 1/4, 1/8, 1/16, … → 0

☞Doesn't always work, but happens to work for us!

Fixed point of a query

- A query q is a function that maps an input table to an output table
- so a fixed point of q is a table T such that $q(T) = T$
- i.e., if you run the query again on the result, it does not change

To compute fixed point of q

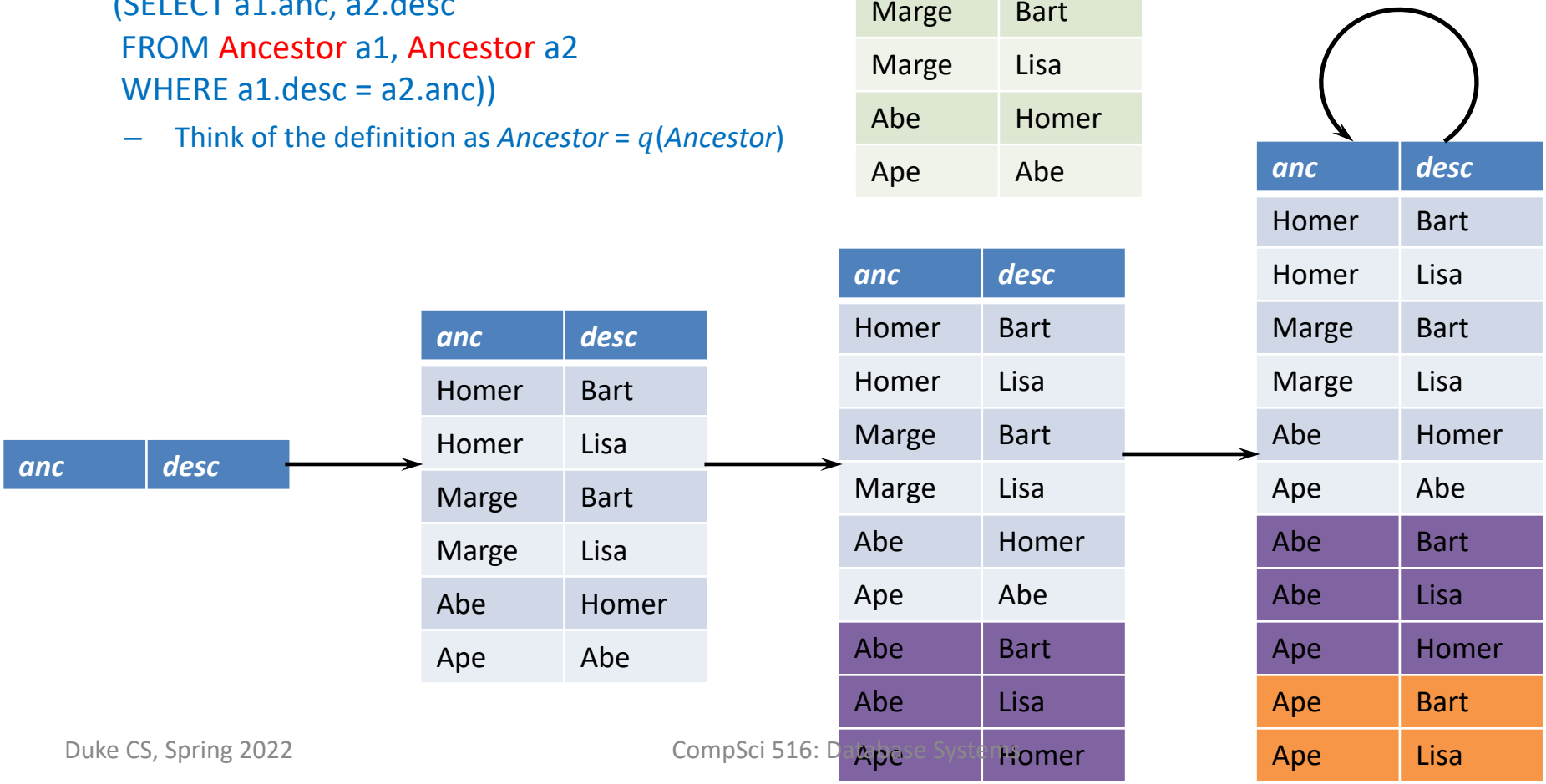
- Start with an empty table: $T \leftarrow \emptyset$
- Evaluate q over T
  - If the result is identical to T, stop; T is a fixed point
  - Otherwise, let T be the new result; repeat

☞ Starting from $\emptyset$ produces the unique minimal fixed point (assuming q is monotone)

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))
  - Think of the definition as *Ancestor* = *q*(*Ancestor*)

parent	child
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe



Linear recursion

- With linear recursion, a recursive definition can make only one reference to itself
- Non-linear
  - 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))
- Linear
  - WITH RECURSIVE Ancestor(anc, desc) AS
    ((SELECT parent, child FROM Parent)
     UNION
     (SELECT anc, child
      FROM Ancestor, Parent
      WHERE desc = parent))

Linear vs. non-linear recursion

- Linear recursion is easier to implement
  - For linear recursion, just keep joining "newly generated" *Ancestor* rows with *Parent*
    - *try to figure out why it should work*
  - For non-linear recursion, need to join newly generated *Ancestor* rows with all existing *Ancestor* rows

- Non-linear recursion may take fewer steps to converge, but perform more work
  - Example: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
  - Linear recursion takes 4 steps
  - Non-linear recursion takes 3 steps
    - More work: e.g., $a \rightarrow d$ has two different derivations

Datalog

Datalog: Another query language for recursion

- Ancestor(x, y) :- Parent(x, y)
- Ancestor(x, y):- Parent(x, z), Ancestor(z, y)

- Like logic programming
- Multiple rules
- Same "head" = union
- "," = AND

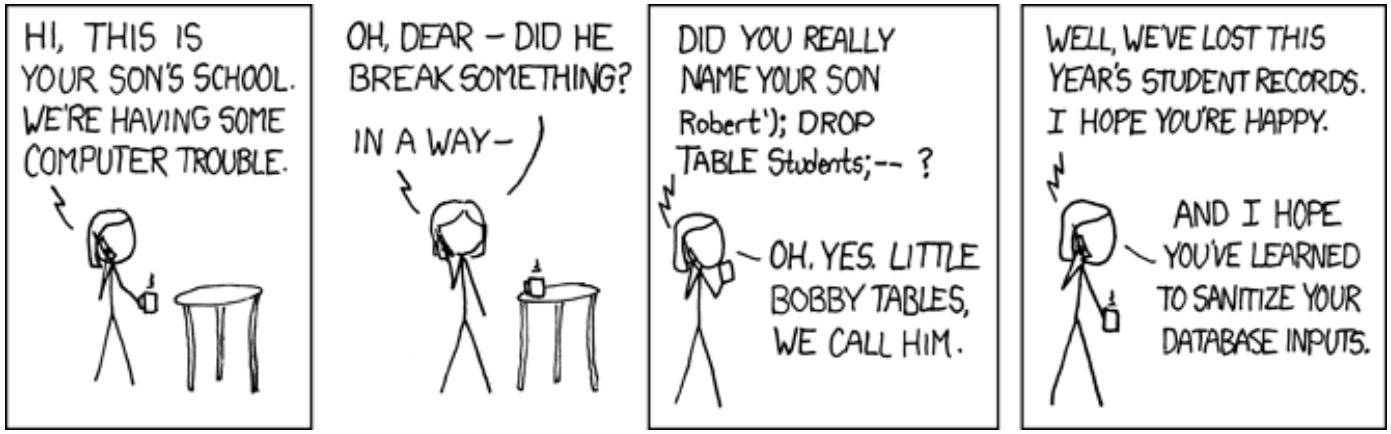
- Same semantics that we discussed so far

Practice Datalog

- Write Datalog program for reachability:
  - R(x, y) :- E(x, y)
  - R(x, y) :- E(x, z), R(z, y)
- E(u, v, c): an edge exists from u to v of color "c"
  - e.g. E(1, 2, 'blue'),E(2, 3, 'red'), ….
- Find node pairs x, y such that x can reach y by a blue path
  - BR(x, y) :- E(x, y, 'blue')
  - BR(x, y) :- BR(x, z), E(x, y, 'blue')
- Try reachable by odd number of edges, by odd number of blue edges, by alternating blue and red paths etc.

Optional reading for SQL programming (from Lecture 3)

SQL Injection Attack

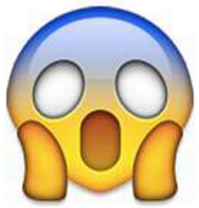


- The school probably had something like:

  http://xkcd.com/327/

  ```
  cur.execute("SELECT * FROM Students " + \
          "WHERE (name = '" + name + "')")
  ```

  where name is a string input by user
- Suppose name = Robert'; DROP TABLE Students



  - Drop deletes a table
  - -- starts a comment
  - Becomes SELECT * FROM Students WHERE (name = 'Robert'; DROP TABLE Students; -- ')

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
- Check out Ashley Madison data breach story or https://medium.com/five-guys-facts/sql-injection-98199af86c9

Optional slide

Prepared statements: motivation

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

- Every time we send an SQL string to the DBMS, it must perform parsing, semantic analysis, optimization, compilation, and finally execution
- A typical application issues many queries with a small number of patterns (with different parameter values)
- 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))
                                # Note the switch back to %s for parameter placeholders.
    # Check result...
```

- The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it "prepares" the statement
- At execution time, the DBMS only needs to check parameter types and validate the compiled plan
- Most other API's have better support for prepared statements than psycopg2
  - E.g., they would provide a cur.prepare() method

Views

Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the view definition query instead of view contents
  - Can be used in queries just like a regular table

Creating and dropping views

User(uid, name, pop)
Member(gid, uid)

- Example: members of Jessica's Circle
  - CREATE VIEW JessicaCircle AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
    WHERE gid = 'jes');
  - Tables used in defining a view are called "base tables"
    - *User* and *Member* above
- To drop a view
  - DROP VIEW JessicaCircle;

Using views in queries

- Example: find the average popularity of members in Jessica's Circle
  - SELECT AVG(pop) FROM JessicaCircle;
  - To process the query, replace the reference to the view by its definition
  - SELECT AVG(pop)
    FROM (SELECT * FROM User
      WHERE uid IN
        (SELECT uid FROM Member
         WHERE gid = 'jes'))
      AS JessicaCircle;

Why use views?

- To hide data from users
- To hide complexity from users

- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications

- To provide a uniform interface for different implementations or sources

☞ Real database applications use tons of views