SQL:
Recursion, Programming

Introduction to Database Management
CS348 Spring 2021
SQL

• Basic SQL (queries, modifications, and constraints)

• Intermediate SQL
  • Triggers
  • Views
  • Indexes

• Advanced SQL
  • Recursion
  • Programming
A motivating example

- 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 introduces recursion
  • WITH clause
  • Implemented in PostgreSQL (common table expressions)
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
((SELECT parent, child FROM Parent)
UNION
(SELECT a1.anc, a2.desc
FROM Ancestor a1, Ancestor a2
WHERE a1.desc = a2.anc))
.....;
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 \( f \)
    • 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 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
Non-linear v.s. linear recursion

• Non-linear
  
  ```sql
  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: a recursive definition can make only one reference to itself

  ```sql
  WITH RECURSIVE Ancestor2(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
  • 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: Given $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$, i.e., a is parent of b, b is parent of c, ..., d is parent of e.
    • The linear recursion takes 4 steps to find $(a, e)$ is an ancestor-descendant pair (slide 9, Ancestor2)
    • Question: How about non-linear recursion? (slide 9, Ancestor)
Mutual recursion example

• Table *Natural*  \((n)\) contains 1, 2, ..., 100
• Which numbers are even/odd?
  • An even number plus 1 is an odd number
  • An odd number plus 1 is an even number
  • 1 is an odd number

WITH RECURSIVE Even(n) AS
 (SELECT n FROM Natural
 WHERE n = ANY(SELECT n+1 FROM Odd)),
RECURSIVE Odd(n) AS
 (SELECT n FROM Natural WHERE n = 1)
 UNION
 (SELECT n FROM Natural
 WHERE n = ANY(SELECT n+1 FROM Even)
Computing mutual recursion

WITH RECURSIVE Even(n) AS
    (SELECT n FROM Natural
     WHERE n = ANY(SELECT n+1 FROM Odd)),
RECURSIVE Odd(n) AS
    ((SELECT n FROM Natural WHERE n = 1)
     UNION
     (SELECT n FROM Natural
      WHERE n = ANY(SELECT n+1 FROM Even))
     )

• Even = ∅, Odd = ∅
• Even = ∅, Odd = {1}
• Even = {2}, Odd = {1}
• Even = {2}, Odd = {1, 3}
• Even = {2, 4}, Odd = {1, 3}
• Even = {2, 4}, Odd = {1, 3, 5}
• ...

• Even = {2, 4}, Odd = {1, 3, 5}
Semantics of WITH

- WITH RECURSIVE \( R_1 \) AS \( Q_1 \),
  ...
  RECURSIVE \( R_n \) AS \( Q_n \)

\( Q \);

- \( Q \) and \( Q_1, \ldots, Q_n \) may refer to \( R_1, \ldots, R_n \)

- Semantics

  1. \( R_1 \leftarrow \emptyset, \ldots, R_n \leftarrow \emptyset \)
  2. Evaluate \( Q_1, \ldots, Q_n \) using the current contents of \( R_1, \ldots, R_n \):
     \[ R_1^{\text{new}} \leftarrow Q_1, \ldots, R_n^{\text{new}} \leftarrow Q_n \]
  3. If \( R_i^{\text{new}} \neq R_i \) for some \( i \)
     3.1. \( R_1 \leftarrow R_1^{\text{new}}, \ldots, R_n \leftarrow R_n^{\text{new}} \)
     3.2. Go to 2.
  4. Compute \( Q \) using the current contents of \( R_1, \ldots, R_n \) and output the result

13
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)

.....;
Fixed points are not unique

- If \( q \) is monotone, then starting from \( \emptyset \) produces the unique minimal fixed point
  - All these fixed points must contain this fixed point
    \( \rightarrow \) the unique minimal fixed point is the “natural” answer

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))

Note how the bogus tuple reinforces itself!

<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>Abe</td>
<td>Abe</td>
</tr>
<tr>
<td>Abe</td>
<td>Bart</td>
</tr>
<tr>
<td>Abe</td>
<td>Lisa</td>
</tr>
<tr>
<td>Ape</td>
<td>Homer</td>
</tr>
<tr>
<td>Ape</td>
<td>Bart</td>
</tr>
<tr>
<td>Ape</td>
<td>Lisa</td>
</tr>
<tr>
<td>Bogus</td>
<td>Bogus</td>
</tr>
</tbody>
</table>
Mixing negation with recursion

• If $q$ is non-monotone
  • The fixed-point iteration may never converge
  • There could be multiple minimal fixed points

• Example: popular users ($\text{pop} \geq 0.8$) join either SGroup or PGroup
  • Those not in SGroup should be in PGroup
  • Those not in GGroup should be in SGroup

WITH RECURSIVE PGroup(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM SGroup)),
RECURSIVE SGroup(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM PGroup))
Fixed-point iter may not converge

WITH RECURSIVE `PGroup`(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM `SGroup`)),
RECURSIVE `SGroup`(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM `PGroup`))

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>121</td>
<td>Allison</td>
<td>8</td>
<td>0.85</td>
</tr>
</tbody>
</table>
WITH RECURSIVE PGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM SGroup)),
RECURSIVE SGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM PGroup))

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>121</td>
<td>Allison</td>
<td>8</td>
<td>0.85</td>
</tr>
</tbody>
</table>
Legal mix of negation and recursion

• Construct a dependency graph
  • One node for each table defined in WITH
  • A directed edge $R \rightarrow S$ if $R$ is defined in terms of $S$
  • Label the directed edge “−” if the query defining $R$ is not monotone with respect to $S$

• Legal SQL3 recursion: no cycle with a “−” edge
  • Called stratified negation

• Bad mix: a cycle with at least one edge labeled “−”
Stratified negation example

• Find pairs of persons with no common 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)),
RECURSIVE Person(person) AS
((SELECT parent FROM Parent) UNION
 (SELECT child FROM Parent)),
RECURSIVE NoCommonAnc(person1, person2) AS
((SELECT p1.person, p2.person
  FROM Person p1, Person p2
  WHERE p1.person <> p2.person)
EXCEPT
(SELECT a1.desc, a2.desc
 FROM Ancestor a1, Ancestor a2
 WHERE a1.anc = a2.anc))

SELECT * FROM NoCommonAnc;
Evaluating stratified negation

• The stratum of a node $R$ is the maximum number of “−” edges on any path from $R$
  • Ancestor: stratum 0
  • Person: stratum 0
  • NoCommonAnc: stratum 1

• Evaluation strategy
  • Compute tables lowest-stratum first
  • For each stratum, use fixed-point iteration on all nodes in that stratum
    • Stratum 0: Ancestor and Person
    • Stratum 1: NoCommonAnc

☞ Intuitively, there is no negation within each stratum
SQL features covered so far

- Basic SQL (queries, modifications, and constraints)
- Intermediate SQL (triggers, views, indexes)
- Recursion
  - SQL3 WITH recursive queries
  - Solution to a recursive query (with no negation)
  - Mixing negation and recursion is tricky
- Programming
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: many possibilities
    • Through an API, e.g., Python psycopg2
    • Embedded SQL, e.g., in C
    • Automatic object-relational mapping, e.g.: Python SQLAlchemy
    • Extending programming languages with SQL-like constructs, e.g.: LINQ
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), Get next, Close

Found in virtually every database language/API

• With slightly different syntaxes
Augmenting SQL: SQL/PSM

- 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);
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
SQL/PSM example continued

```sql
-- 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;
```
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
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
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()
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)
# ...
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

```python
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

```python
cur.execute(''')
# Prepare once (in SQL).
PREPARE update_price AS
UPDATE Serves
SET price = $1
WHERE bar = $2 AND beer = $3''')
# Name the prepared plan,
# parameter placeholders.
while true:
    # Input bar, beer, price...
    cur.execute('''
    EXECUTE update_price(%s, %s, %s),
    (price, bar, beer))....
    # Execute many times.
    # Check result...
```

Prepare only once
“Exploits of a mom”

• The school probably had something like:
  
  ```python
  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

• Cons of augmenting SQL:
  • SQL is already too big
  • Complicate optimization and make it impossible to guarantee safety
A brief look at other approaches

• “Embed” SQL in a general-purpose programming language
  • E.g.: embedded SQL

• Support database features through an object-oriented programming language
  • E.g., object-relational mappers (ORM) like Python SQLAlchemy

• Extend a general-purpose programming language with SQL-like constructs
  • E.g.: LINQ (Language Integrated Query for .NET)
Embedding SQL in a language

Example in C

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')
FOR UPDATE;
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;
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
A brief look at other approaches

• “Embed” SQL in a general-purpose programming language
  • E.g.: embedded SQL

• Support database features through an object-oriented programming language
  • E.g., object-relational mappers (ORM) like Python SQLAlchemy

• Extend a general-purpose programming language with SQL-like constructs
  • E.g.: LINQ (Language Integrated Query for .NET)
Object-relational mapping

- Example: Python SQLAlchemy

```python
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    password = Column(String)

class Address(Base):
    __tablename__ = 'addresses'
    id = Column(Integer, primary_key=True)
    email_address = Column(String, nullable=False)
    user_id = Column(Integer, ForeignKey('users.id'))

    # relationship declarations
    User.addresses = relationship("Address", order_by=Address.id, back_populates="user")
    Address.user = relationship("User", back_populates="addresses")

jack = User(name='jack', password='gjjfdd')
jack.addresses = [Address(email_address='jack@gmail.com'), Address(email_address='j25@yahoo.com')]
session.add(jack)
session.commit()

session.query(User).join(Address).filter(Address.email_address=='jack@gmail.com').all()
```

- Automatic data mapping and query translation
- But syntax may vary for different host languages
- Very convenient for simple structures/queries, but quickly get complicated and less intuitive for more complex situations
A brief look at other approaches

• “Embed” SQL in a general-purpose programming language
  • E.g.: embedded SQL

• Support database features through an object-oriented programming language
  • By automatically storing objects in tables and translating methods to SQL
  • E.g., object-relational mappers (ORM) like Python SQLAlchemy

• Extend a general-purpose programming language with SQL-like constructs
  • E.g.: LINQ (Language Integrated Query for .NET)
Deeper language integration

• Example: LINQ (Language Integrated Query) for Microsoft .NET languages (e.g., C#)

```csharp
int someValue = 5;
var results = from c in someCollection
    let x = someValue * 2
    where c.SomeProperty < x
    select new {c.SomeProperty, c.OtherProperty};
foreach (var result in results) {
    Console.WriteLine(result);
}
```

• Again, automatic data mapping and query translation
• Much cleaner syntax, but it still may vary for different host languages
Summary

• Basic SQL (queries, modifications, and constraints)
• Intermediate SQL (triggers, views, indexes)
• Recursion
• Programming
  • Augment SQL, e.g., SQL/PSM
  • Through an API, e.g., Python psycopg2, JDBC
  • Embedded SQL, e.g., in C
  • Automatic object-relational mapping, e.g.: Python SQLAlchemy
  • Extending programming languages with SQL-like constructs, e.g.: LINQ