SQL: Index and Recursion
Introduction to Databases
CompSci 316 Spring 2017

Announcements (Wed., Feb. 15)
• Homework #2 due Friday 02/17
• Except Problem 6 (Gradience) and Problem X1 (non-Gradience): Thursday 02/23
• Please submit on time – solutions will be posted by Saturday morning
• Midterm next Wednesday 02/22 in class
• Up to lecture 9 included
• Will review some concepts/practice problems on Monday

Indexes
• An index is an auxiliary persistent data structure • Search tree (e.g., B-tree), lookup table (e.g., hash table), etc.
  • More on indexes later in this course!
• An index on $R.A$ can speed up accesses of the form
  • $R.A = \text{value}$
  • $R.A > \text{value}$ (sometimes; depending on the index type)
• An index on $(R.A_1, ..., R.A_n)$ can speed up
  • $R.A_1 = \text{value}_1 \land ... \land R.A_n = \text{value}_n$
  • $(R.A_1, ..., R.A_n) > (\text{value}_1, ..., \text{value}_n)$ (again depends)
  • Ordering or 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 $R.B$?

Examples of using indexes: 1/2
• SELECT * FROM User WHERE name = 'Bart';
• Without an index on User.name:
  • With an index on User.name:

Examples of using indexes: 2/2
• SELECT * FROM User, Member
  WHERE User.uid = Member.uid
  AND Member gid = 'jes';
• With an index on Member.gid or (gid, uid):
• With an index on User.uid:
• Without an index:

Creating and dropping indexes in SQL
CREATE [UNIQUE] INDEX indexname ON
tablename(columnname_1, ..., columnname_n);
• With UNIQUE, the DBMS will also enforce that
  {columnname_1, ..., columnname_n} is a key of
  tablename
DROP INDEX indexname;
• Typically, the DBMS will automatically create
  indexes for PRIMARY KEY and UNIQUE
  constraint declarations
Choosing indexes to create

More indexes = better performance?

Optimal index selection depends on both query and update workload and the size of tables
- Automatic index selection is now featured in some commercial DBMS

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)

Ancestor query in SQL3

```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.child = a2.parent AND a1.desc = 'Bart',)
SELECT desc FROM Ancestor WHERE desc = 'Bart';
```

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 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

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

Intuition behind fixed-point iteration

- Initially, we know nothing about ancestor-descendent relationships
- In the first step, we deduce that parents and children form ancestor-descendent relationships
- In each subsequent step, we use the facts deduced in previous steps to get more ancestor-descendent relationships
- We stop when no new facts can be proven

Linear vs. non-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)
Mutual recursion example

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

Semantics of WITH

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

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))

Fixed points are not unique

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!

Mixing negation with recursion

- If \( q \) is non-monotone
  - The fixed-point iteration may flip-flop and never converge
  - There could be multiple minimal fixed points—we wouldn't know which one to pick as answer!

- Example: popular users (pop \( \geq 0.8 \)) join either Jessica's Circle or Tommy's (but not both)
  - Those not in Jessica's Circle should be in Tommy's
  - Those not in Tommy's Circle should be in Jessica's

- WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop \( \geq 0.8 \)
  AND uid NOT IN (SELECT from JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop \( \geq 0.8 \)
  AND uid IN (SELECT from TommyCircle))
Fixed-point iter may not converge

• WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM TommyCircle))

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 “−”

Evaluating stratified negation

• The stratum of a node \( R \) is the maximum number of
  “−” edges on any path from \( R \) in the dependency graph
  • Ancestor: stratum 0
  • Person: stratum 0
  • NoCommonAnc: stratum 1

• 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

Multiple minimal fixed points

• WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM TommyCircle))

Problem: What do we answer if someone asks whether 121 belongs to JessicaCircle?

Stratified negation example

• Find pairs of persons with no common ancestors

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
    • not covered in detail in this class
Summary

• SQL3 WITH recursive queries
• Solution to a recursive query (with no negation): unique minimal fixed point
• Computing unique minimal fixed point: fixed-point iteration starting from ∅
• Mixing negation and recursion is tricky
  • Illegal mix: fixed-point iteration may not converge; there may be multiple minimal fixed points
  • Legal mix: stratified negation (compute by fixed-point iteration stratum by stratum)
• Another language for recursion: Datalog