Recursive Query Evaluation and Datalog

Instructor: Sudeepa Roy
Announcements

• HW3 due Monday 11/26
• Next week
  – practice pop-up quiz on transactions (all lectures)
• Project presentation in last class, but final report due 2 days before final
Where are we now?

We learnt

✓ Relational Model and Query Languages
  ✓ SQL, RA, RC
  ✓ Postgres (DBMS)
    ▪ HW1
✓ Database Normalization
✓ DBMS Internals
  ✓ Storage
  ✓ Indexing
  ✓ Query Evaluation
  ✓ Operator Algorithms
  ✓ External sort
  ✓ Query Optimization
✓ Map-reduce and spark
  ▪ HW2

• Transactions
  – Basic concepts
  – Concurrency control
  – Recovery
• Distributed DBMS
• NOSQL
• Parallel DBMS
Today

• Semantic of recursion in databases

• Datalog
  – for recursion in database queries

• Views
Recursion!

http://xkcdsw.com/1105
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
Recursion in Databases

• Consider a graph $G(V, E)$. Can you find out all “ancestor” vertices that can reach “x” using Relational Algebra/Calculus?

• NO! – ANCESTOR cannot be defined using a constant-size union of select-project-join queries (conjunctive queries)

• No RA/RC expressions can express ANCESTOR or REACHABILITY (TRANSITIVE CLOSURE) (Aho-Ullman, 1979)

• A limitation of RA/RC in expressing recursive queries
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)

• SQL:1999 (SQL3) and later versions support “linear Datalog”
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++
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 following slides have been borrowed from slides by Prof. Jun Yang
Recursive Query in SQL
Recursion in SQL

• **SQL2** had no recursion

• **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';
Fixed point of a function

• If \( f: T \to T \) 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 \textbf{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

\( \text{Starting from } \emptyset \text{ produces the unique minimal fixed point (assuming } q \text{ is monotone) } \)
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)
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*
    • *Homework: 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
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
  - 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** $R_1$ **AS** $Q_1$, ..., **RECURSIVE** $R_n$ **AS** $Q_n$
  
  $Q$;
  
  - $Q$ and $Q_1, ..., Q_n$ may refer to $R_1, ..., R_n$

- **Semantics**

  1. $R_1 \leftarrow \emptyset$, ..., $R_n \leftarrow \emptyset$

  2. Evaluate $Q_1, ..., Q_n$ using the current contents of $R_1, ..., R_n$:
     
     $R_1^{\text{new}} \leftarrow Q_1$, ..., $R_n^{\text{new}} \leftarrow Q_n$

  3. If $R_i^{\text{new}} \neq R_i$ for some $i$
     
     3.1. $R_1 \leftarrow R_1^{\text{new}}$, ..., $R_n \leftarrow R_n^{\text{new}}$
     
     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))))

- \( Even = \emptyset, Odd = \emptyset \)
- \( Even = \emptyset, Odd = \{1\} \)
- \( Even = \{2\}, Odd = \{1\} \)
- \( Even = \{2\}, Odd = \{1, 3\} \)
- \( Even = \{2, 4\}, Odd = \{1, 3\} \)
- \( Even = \{2, 4\}, Odd = \{1, 3, 5\} \)
- ...
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 ($\text{pop} \geq 0.8$) join either Jessica’s Circle or Tommy’s (but not both)
  – Those not in Jessica’s Circle should be in Tom’s
  – Those not in Tom’s Circle should be in Jessica’s

• 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))
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))

<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>
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))

<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>

Problem: What do we answer if someone asks whether 121 belongs to JessicaCircle?
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)),
    Person(person) AS
    ((SELECT parent FROM Parent) UNION
     (SELECT child FROM Parent)),
    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 \( R \) in the dependency graph
  
  – \textit{Ancestor}: stratum 0
  – \textit{Person}: stratum 0
  – \textit{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: \textit{Ancestor} and \textit{Person}
    * Stratum 1: \textit{NoCommonAnc}

\( \Rightarrow \) Intuitively, there is no negation within each stratum
Summary so far

• 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 $\emptyset$
• 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
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
Recall our drinker example in RC (Lecture 4)

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Drinker example is from slides by Profs. Balazinska and Suciu and the [GUW] book
Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

### RC:
\[
Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y,z) \land \text{Likes}(x,z)
\]

### Datalog:
\[
Q(x) :- \text{Frequents}(x, y), \text{Serves}(y,z), \text{Likes}(x,z)
\]
Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

RC:
\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Datalog:
\[ Q(x) \leftarrow \text{Frequents}(x, y), \text{Serves}(y, z), \text{Likes}(x, z) \]

• Quick differences:
  – Uses “:-” not =
  – no need for \( \exists \) (assumed by default)
  – Use “,” on the right hand side (RHS)
  – Anything on RHS the of :- is assumed to be combined with \( \land \) by default
  – \( \forall, \Rightarrow \), not allowed – they need to use negation \( \neg \)
  – Standard “Datalog” does not allow negation
  – Negation allowed in datalog with negation

• How to specify disjunction (OR / \( \lor \))?
Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer “BestBeer”

RC:
Q(x) = [∃y. ∃z. Frequents(x, y) ∧ Serves(y, z) ∧ Likes(x, z)] ∨ [Likes(x, “BestBeer”)]

Datalog:
Q(x) :- Frequents(x, y), Serves(y, z), Likes(x, z)
Q(x) :- Likes(x, “BestBeer”)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)
Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer “BestBeer”, (c) or, frequent bars that “Joe” frequents

RC:
Q(x) = [∃y. ∃z. Frequents(x, y) ∧ Serves(y,z) ∧ Likes(x,z)] ∨ [Likes(x, “BestBeer”)]
∨ [∃w Frequents(x, w) ∧ Frequents(“Joe”, w)]

Datalog:
JoeFrequents(w) :- Frequents(“Joe”, w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, “BestBeer”)
Q(x) :- Frequents(x, w), JoeFrequents(w)

• To specify “OR”, write multiple rules with the same “Head”
• Next: terminology for Datalog
Datalog Rules

- Each rule is of the form  \( \text{Head} :- \text{Body} \)

- Each variable in the head of each rule must appear in the body of the rule

\[ \begin{align*}
\text{JoeFrequents}(w) & :- \text{Frequents(“Joe”, w)} \\
Q(x) & :- \text{Frequents(x, y), Serves(y,z), Likes(x,z)} \\
Q(x) & :- \text{Likes(x, “BestBeer”) } \\
Q(x) & :- \text{Frequents(x, w), JoeFrequents(w)} \\
\end{align*} \]
Termination of a Datalog Program

Q. A Datalog program always terminates – why?
Unsafe/Safe Datalog Rules

Find drinkers who like beer “BestBeer”

\[
Q(x) :\text{ Likes}(x, \text{“BestBeer”})
\]

Find drinkers who DO NOT like beer “BestBeer”

\[
Q(x) :\neg\text{ Likes}(x, \text{“BestBeer”})
\]

• **What is the problem with this rule?**
• **What should this rule return?**
  – names of all drinkers in the world?
  – names of all drinkers in the USA?
  – names of all drinkers in Durham?
Domain-dependency is bad

Find drinkers who like beer “BestBeer”

Q(x) :- Likes(x, “BestBeer”)

Find drinkers who **DO NOT** like beer “BestBeer”

Q(x) :- ¬Likes(x, “BestBeer”)

• What is the problem with this rule?
• Dependent on “domain” of drinkers
  – domain-dependent
  – infinite answers possible too..
    • keep generating “names”
  – Unsafe rule
Safe Datalog Rules

Find drinkers who like beer “BestBeer”

Find drinkers who DO NOT like beer “BestBeer”

• Solution:
• Restrict to “active domain” of drinkers from the input Likes (or Frequent) relation
  – “domain-independence” – same finite answer always
• Becomes a “safe rule”
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

• 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