Recursive Computation of Regions and Connectivity in Networks

Mengmeng Liu, Nicholas E. Taylor, Wenchao Zhou, Zachary G. Ives, Boon Thau Loo

To appear in International Conference on Data Engineering (ICDE 2009)
Presented at System Lunch, Jan 22, 2009
Motivation

Traditional Networks
- Network state
- Network protocol
- Network messages

Declarative Networking
- Distributed table
- Distributed recursive query
- Query execution results
Network queries

2-hop query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), link(Z,D).

reachability query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), reachable(Z,D).

Base: link(source, destination)
Derived: reachable(source, destination)
What if network states change a lot?

2-hop query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), link(Z,D).

reachability query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), reachable(Z,D).

link state

Base: link(source, destination)
Derived: reachable(source, destination)
A lot of interesting applications

- Networks
  - Research efforts:
    - Declarative networking [Loo+ 06]
  - Example: computing shortest-path in a network

- Sensor networks
  - Research efforts:
    - Macroprogramming abstract regions [Welsh+04]
    - Declarative sensor networks [Chu+07]
  - Example: sensing fired regions

- Things in common: declarative, distributed, dynamic
Why it is hard...

- Maintain correct state under dynamic networks
- Understand how to partition data and process queries in a distributed fashion
- Handle interesting aggregates (e.g., min, max, count, sum...)
- Propose a general framework for any type of recursive query in any domain
- Always remember scalability and performance!
- We term this problem: distributed recursive stream view maintenance!
Incremental insertion

link state

reachability query:

R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), reachable(Z,D).

Note:
(1) Set semantics: one tuple can only appear once in the result
(2) Fixpoint: it reaches a fixpoint when no longer any more tuples are derived
How about incremental deletion?

• For non-recursive queries (such as 2-hop query), counting scheme makes incremental deletion the same as incremental insertion!

• But, how about recursive queries? If you count how many ways a tuple is derived, it soon becomes infinite!
DRed algorithm [Gupta+ 93] for incremental deletion

Delta rules for incremental deletion:
Step 1(Over-Delete):
reachable(S,D) :- link(S,D).
reachable(S,D) :- link(S,Z), reachable(Z,D).
reachable(S,D) :- link(S,Z), reachable¬(Z,D).

Step 2(Re-Derive):
reachable+(S,D) :- link+(S,Z), reachable+(Z,D), reachable¬(S,D).

In this example, this approach basically deletes and then restores the whole table! Can it be worse?

Reachability query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), reachable(Z,D).

<table>
<thead>
<tr>
<th>Link</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple</td>
<td>at - to</td>
</tr>
<tr>
<td>tuple</td>
<td>at - to</td>
</tr>
<tr>
<td>-(C,B)</td>
<td>C</td>
</tr>
<tr>
<td>-(C,A)</td>
<td>B - C</td>
</tr>
<tr>
<td>-(C,B)</td>
<td>B - C</td>
</tr>
<tr>
<td>-(C,C)</td>
<td>B - C</td>
</tr>
<tr>
<td>-(B,A)</td>
<td>C - B</td>
</tr>
<tr>
<td>-(B,B)</td>
<td>C - B</td>
</tr>
<tr>
<td>-(B,C)</td>
<td>C - B</td>
</tr>
<tr>
<td>-(C,A)</td>
<td>B - A</td>
</tr>
<tr>
<td>-(C,B)</td>
<td>B - A</td>
</tr>
<tr>
<td>-(C,C)</td>
<td>B - A</td>
</tr>
</tbody>
</table>
Intuition For a Solution

• Book-keeping the condition under which a derived tuple’s existence depends on base tuples might be very useful for deletions
• Related work: semiring provenance [Green+ 07]
• What kind of provenance do we need here?
• Our goal: build a query engine around this (including aggregate functions such as min, max, count, sum, etc), minimize message propagations and scale to large networks.
### Insertions

**Link State**

- **A**
- **B**
- **C**

**Reachability Query:**

**R1:** reachable(S,D) : - link(S,D).

**R2:** reachable(S,D) : - link(S,Z), reachable(Z,D).

Note a different fixpoint notion here: now we reach a fixpoint when we can no longer derive any new results that affect the absorption provenance of any tuple in the result!

<table>
<thead>
<tr>
<th>Tuple</th>
<th>At</th>
<th>To</th>
<th>PV</th>
<th>Tuple</th>
<th>At</th>
<th>To</th>
<th>PV</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A,B)</td>
<td>A</td>
<td></td>
<td>p₁</td>
<td>(A,B)</td>
<td>A</td>
<td></td>
<td>p₁</td>
</tr>
<tr>
<td>(B,C)</td>
<td>B</td>
<td></td>
<td>p₂</td>
<td>(B,C)</td>
<td>B</td>
<td></td>
<td>p₂</td>
</tr>
<tr>
<td>(A,C)</td>
<td>B - A</td>
<td></td>
<td>p₁ p₂</td>
<td>(C,A)</td>
<td>C</td>
<td></td>
<td>p₃</td>
</tr>
<tr>
<td>(C,B)</td>
<td>A - C</td>
<td></td>
<td>p₁ p₃</td>
<td>(C,B)</td>
<td>A - C</td>
<td></td>
<td>p₁ p₂ p₃</td>
</tr>
<tr>
<td>(B,A)</td>
<td>C - B</td>
<td></td>
<td>p₂ p₃</td>
<td>(B,B)</td>
<td>C - B</td>
<td></td>
<td>p₁ p₂ p₃</td>
</tr>
<tr>
<td>(B,C)</td>
<td>C - B</td>
<td></td>
<td>p₁ p₂ p₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A,A)</td>
<td>B - A</td>
<td></td>
<td>p₁ p₂ p₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A,B)</td>
<td>B - A</td>
<td></td>
<td>p₁ p₂ p₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C,A)</td>
<td>A - C</td>
<td></td>
<td>p₁ p₂ p₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C,B)</td>
<td>C</td>
<td></td>
<td>p₄</td>
<td>(C,B)</td>
<td>C</td>
<td></td>
<td>p₄</td>
</tr>
<tr>
<td>(C,A)</td>
<td>B - C</td>
<td></td>
<td>p₂ p₃ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C,B)</td>
<td>B - C</td>
<td></td>
<td>p₁ p₂ p₃ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C,C)</td>
<td>B - C</td>
<td></td>
<td>p₁ p₂ p₃ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>(B,B)</em></td>
<td>C - B</td>
<td></td>
<td>p₂ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>(B,C)</em></td>
<td>C - B</td>
<td></td>
<td>p₁ p₂ p₃ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>(A,B)</em></td>
<td>B - A</td>
<td></td>
<td>p₁ p₂ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>(C,B)</em></td>
<td>A - C</td>
<td></td>
<td>p₁ p₃ p₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What’s magic about deletions here?

For incremental deletions, we just need to zero out the appropriate tokens in the provenance expressions of all reachable tuples!

reachability query:
R1: reachable(S,D) : - link(S,D).
R2: reachable(S,D) : - link(S,Z), reachable(Z,D).

<table>
<thead>
<tr>
<th>Link</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple</td>
<td>at - to</td>
</tr>
<tr>
<td>(A,A)</td>
<td>A</td>
</tr>
<tr>
<td>(A,B)</td>
<td>A</td>
</tr>
<tr>
<td>(A,C)</td>
<td>A</td>
</tr>
<tr>
<td>(B,A)</td>
<td>B</td>
</tr>
<tr>
<td>(B,B)</td>
<td>B</td>
</tr>
<tr>
<td>(B,C)</td>
<td>B</td>
</tr>
<tr>
<td>(C,A)</td>
<td>C</td>
</tr>
<tr>
<td>(C,B)</td>
<td>C</td>
</tr>
<tr>
<td>(C,C)</td>
<td>C</td>
</tr>
<tr>
<td>-(C,B)</td>
<td>C</td>
</tr>
<tr>
<td>(B,B)</td>
<td>B</td>
</tr>
<tr>
<td>(C,B)</td>
<td>C</td>
</tr>
<tr>
<td>(C,C)</td>
<td>C</td>
</tr>
</tbody>
</table>
Formal representation

• Annotate tuples with Boolean expressions: the tuple is in the result iff the expression evaluates to true.
• For each base tuple t, we annotate it with P(t). If t is an insertion, P(t)=true; if t is an deletion, P(t)=false.
• Annotating Rules:
  – Joins: For each tuple t₁ in R₁ and tuple t₂ in R₂, annotate t₁ join t₂ with P(t₁) ∧ P(t₂).
  – Unions: For each tuple t output by R₁ ∪ R₂, annotate t with P(t₁) ∨ P(t₂).
• Both operations are idempotent.
• Absorption law:
  \[ a \land (a \lor b) \equiv a \lor (a \land b) \equiv a \]
How to store absorption provenance?

• Binary Decision Diagrams
  – Rooted, direct, acyclic graph
  – Ordered and Reduced
  – Highly optimized libraries available: e.g. JavaBDD.
Distributed query processing
More optimizations: lazy propagation of provenance

• Good effects of absorption provenance
  – Derivations are combined and absorbed.
  – Avoid infinite number of derivations in recursive queries.

• Bad effects of absorption provenance:
  – A single tuple may be processed and shipped multiple times.

• We introduce a *MinShip* operator!
  – Idea: separate data with provenance
  – Insertion: only propagate the *first* derivation, and buffer the subsequent derivations
  – Deletion: trigger propagating those buffered, not yet sent derivations
How to make aggregates efficient?

- **Aggregate selection** is always a good idea: push down selections before aggregates

Example (Shortest path query):

R1: `path(S,D,C) :- link(S,D,C).`
R2: `path(S,D,C) :- link(S,Z,C1), path(Z,D,C2), C=C1+C2.`
R3: `shortestpath(S,D,min<C>) :- path(S,D,C)`

- But again, don’t forget deletion situations...
Experimental Evaluation: Incremental View Maintenance

- **Communication Overhead (MB)** vs. Deletion Ratio
  - DRed
  - Absorption Lazy
  - Absorption Eager

- **Execution Time (s)** vs. Deletion Ratio
  - DRed
  - Absorption Lazy
  - Absorption Eager
Conclusion

• Distributed recursive stream view maintenance
  – Absorption provenance
  – MinShip and lazy propagation
  – Multi-aggregate selection

• If you are interested in our paper or more technical details, check out my homepage: http://www.cis.upenn.edu/mengmeng
Thanks!