Recursive Computation of Regions and Connectivity in Networks

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

To appear in ICDE 2009
Motivation

- Recursive computations used to be expressed in procedural languages.
- Declarative queries:
  - Router states: shortest path, etc.
    - Declarative networking [Loo^+06]
  - Sensors: contiguous regions, etc.
    - Macroprogramming abstract regions [Welsh^+04]
- Recursive queries
- How do we incrementally maintain the state as data sources are distributed streams?
Outline

• Example queries and execution model
• Stream view maintenance
• Absorption provenance
• Minimizing propagation
• Experimental evaluation
• Conclusion and future work
Example queries

- **Network reachability**
  
  \[
  \text{reachable}(x, y) :- \text{link}(x, y).
  \]
  
  \[
  \text{reachable}(x, y) :- \text{link}(x, z), \text{reachable}(z, y).
  \]

- **Network shortest path**

- **Sensing contiguous regions**
  
  \[
  \text{activeRegion}(\text{rid}, x) :- \text{sensor}(x, \text{posx}), \text{isTriggered}(x), \text{mainSensorInRegion}(\text{rid}, x).
  \]
  
  \[
  \text{activeRegion}(\text{rid}, y) :- \text{sensor}(x, \text{posx}), \text{sensor}(y, \text{posy}), \text{isTriggered}(x), \text{activeRegion}(\text{rid}, x), \text{distance}(\text{posx}, \text{posy})<k.
  \]
Query execution model

- Distributed continuous query processing over streams

Assumptions:
- Input: streams of insertions and deletions
- Sliding windows on base data
- Set semantics
- Communication among nodes: in-order delivery
- Tuples are horizontally partitioned
Outline

- Example queries and execution model
- **Stream view maintenance**
- Absorption provenance
- Minimizing propagation
- Experimental evaluation
- Conclusion and future work
Stream View Maintenance

- Problem: *Incremental View Maintenance* in the presence of distributed *streams* of update tuples

- Challenges
  - Recursive
  - Distributed
  - Dynamic
  - Aggregates

- First question: can we use existing centralized schemes? Such as DRed[Gupta+ 93]?
DRed-style incremental insertion

Reachable Query:

reachable(x,y) :- link(x,y).
reachable(x,y) :- link(x,z), reachable(z,y).

Delta rules:

reachable⁺(x,y) :- link⁺(x,y).
reachable⁺(x,y) :- link⁺(x,z), reachable(z,y).
reachable⁺(x,y) :- link⁺(x,z), reachable⁺(z,y).
reachable⁺(x,y) :- link⁺(x,z), reachable⁺(z,y).

<table>
<thead>
<tr>
<th>Link</th>
<th>Reachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>tuple at - to tuple at - to</td>
<td></td>
</tr>
<tr>
<td>(A,B) A (A,B) A</td>
<td></td>
</tr>
<tr>
<td>(B,C) B (B,C) B</td>
<td></td>
</tr>
<tr>
<td>(A,C) B - A (A,C) B - A</td>
<td></td>
</tr>
<tr>
<td>(C,A) C (C,A) C</td>
<td></td>
</tr>
<tr>
<td>(C,B) A - C (C,B) A - C</td>
<td></td>
</tr>
<tr>
<td>(C,C) A - C (C,C) A - C</td>
<td></td>
</tr>
<tr>
<td>(B,A) C - B (B,A) C - B</td>
<td></td>
</tr>
<tr>
<td>(B,B) C - B (B,B) C - B</td>
<td></td>
</tr>
<tr>
<td>(B,C) C - B (B,C) C - B</td>
<td></td>
</tr>
<tr>
<td>(A,A) B - A (A,A) B - A</td>
<td></td>
</tr>
<tr>
<td>(A,B) B - A (A,B) B - A</td>
<td></td>
</tr>
<tr>
<td>(C,A) A - C (C,A) A - C</td>
<td></td>
</tr>
<tr>
<td>(C,B) B - C (C,B) B - C</td>
<td></td>
</tr>
<tr>
<td>(C,C) B - C (C,C) B - C</td>
<td></td>
</tr>
</tbody>
</table>
**DRed-style incremental deletion**

Reachable Query:

\[
\text{reachable}(x,y) :- \text{link}(x,y).
\]

\[
\text{reachable}(x,y) :- \text{link}(x,z), \text{reachable}(z,y).
\]

**Delta rules:**

**Step 1 (Over-Delete):**

\[
\text{reachable}^+(x,y) :- \text{link}^-(x,y).
\]

\[
\text{reachable}^+(x,y) :- \text{link}^-(x,z), \text{reachable}^+(z,y).
\]

\[
\text{reachable}^+(x,y) :- \text{link}^-(x,z), \text{reachable}^-(z,y).
\]

**Step 2 (Re-Derive):**

\[
\text{reachable}^+(x,y) :- \text{link}^+(x,z), \text{reachable}^+(z,y), \text{reachable}^-(x,y).
\]

<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></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>

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

**Insertions: 16 tuples shipped.**

**Deletion on a single tuple: 22 tuples shipped!**
Intuition for a solution

- Major challenge is in handling deletions.
- When is a derived tuple’s existence dependent on base tuples?
- A kind of provenance, which is Boolean PosBool semiring in [Green+ 07]
- Our goal: build a query model around this (including aggregate functions such as min, max, count, sum, etc), and minimize message propagations.
Outline

- Example queries and execution model
- Stream view maintenance
- Absorption provenance
- Minimizing propagation
- Experimental evaluation
- Conclusion and future work
Absorption Provenance

- Annotate tuples with Boolean expressions: the tuple is in the view iff the expression evaluates to true.
- For each base tuple \( t \), we annotate it with \( P(t) \). If \( t \) is an insertion, \( P(t)=\text{true} \); if \( t \) is an deletion, \( P(t)=\text{false} \).
- Rules:
  - \( \sigma_\theta(R) \): If tuple \( t \) in \( R \) satisfies \( \theta \), annotate \( t \) with \( P(t) \).
  - \( R_1 \bowtie R_2 \): For each tuple \( t_1 \) in \( R_1 \) and tuple \( t_2 \) in \( R_2 \), annotate the output join tuple with \( P(t_1) \land P(t_2) \).
  - \( R_1 \cup R_2 \): For each tuple \( t \) output by \( R_1 \cup R_2 \), annotate \( t \) with \( P(t_1) \lor P(t_2) \).
  - \( \pi_A(R) \): Given tuples \( t_1, t_2, \ldots t_n \) that project to the same tuple \( t \), annotate \( t \) with \( P(t_1) \lor P(t_2) \ldots \lor P(t_n) \).
- Absorption Law:
  \[
  a \land (a \lor b) \equiv a \lor (a \land b) \equiv a
  \]
Our Approach: Absorption Provenance

Insertion:

Reachable Query:

reachable(x,y) :- link(x,y).
reachable(x,y) :- link(x,z), reachable(z,y).

Here we introduce absorption provenance!

We reach a fixpoint when we can no longer derive any new results that affect the absorption provenance of any tuple in the result!
Our Approach: Absorption Provenance

Deletion:

Reachable Query:

reachable(x,y) :- link(x,y).
reachable(x,y) :- link(x,z), reachable(z,y).

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

Insertions: 20 tuples shipped.
Deletion on a single tuple: 2 tuples shipped!

<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>
Implementing Absorption Provenance

- Binary Decision Diagrams
  - Rooted, direct, acyclic graph
  - Ordered and Reduced
  - Highly optimized libraries available: e.g. JavaBDD.
Provenance-aware Stateful Operators

- Fixpoint operator
  - Pipelined semi-naive evaluation
  - Add an inner loop to the query plan tree
- Join operator
  - Pipelined hash join
- Aggregate operator
- Ship operator
Reachable Query:

reachable(x,y) :- link(x,y).
reachable(x,y) :- link(x,z), reachable(z,y).
Outline

- Example queries and execution model
- Stream view maintenance
- Absorption provenance
- Minimizing propagation
- Experimental evaluation
- Conclusion and future work
Minimizing 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 metadata
  - Insertions: only propagates the \textit{first} derivation, and buffers the subsequent derivations
  - Deletions: forward propagating the combined buffered derivations (usually stored in a single BDD)
  - If the buffer size goes to infinite, we call it \textit{Lazy provenance propagation}
Minimizing propagation of tuples

- **Aggregate selection** is a good idea: push down selections before aggregates.
- But how to deal with deletions? We introduce a provenance-aware Aggregate Selection module.
  - Insertions: If it affects the aggregate value, propagate it; otherwise, store it in the buffer.
  - Deletions: If it results in the aggregate value, propagate this deletion, and fetch the next best value, propagate it; otherwise, store it in the buffer.
  - We can also optimize multi-aggregates using this scheme!
Outline

- Example queries and execution model
- Stream view maintenance
- Absorption provenance
- Minimizing propagation
- Experimental evaluation
- Conclusion and future work
Experimental Evaluation

- **Workload 1: Declarative networks**
  - GT-ITM: simulate Internet topologies
  - *Reachable query* and *shortest-path query*

- **Workload 2: Sensor networks**
  - Simulate 100m * 100m grid of sensors
  - *Continuous region query* and *largest-region query*

- **Evaluation Metrics:**
  - Per-tuple provenance overhead (B)
  - Communication overhead (MB)
  - Per-node states of operators (MB)
  - Convergence time (s)

- **Comparison Methods:**
  - Absorption provenance/Relative provenance/Dred
  - Eager/Lazy propagation
  - Dense/Sparse graph
Evaluation: Incremental View Maintenance

Insertions-only workload

Communication Overhead (MB)

Execution Time (s)

- DRed
- Absorption Lazy
- Absorption Eager
- Relative Lazy
- Relative Eager

Insertion Ratio

Insertion Ratio
Evaluation: Incremental View Maintenance

Insertions-followed-by-deletion workload
Evaluation: Scaling data sets

- Lazy Dense
- Lazy Sparse
- Eager Dense
- Eager Sparse

Graphs showing communication overhead and execution time as functions of total links in the network.
Evaluation: Aggregate Selection

Communication Overhead (MB)

- Dense
- Sparse

Execution Time (s)

- Multi AggSel
- Single AggSel
- No AggSel

Overhead:
- >25 MB
- >300 MB
Outline

- Example queries and execution model
- Stream view maintenance
- Absorption provenance
- Minimizing propagation
- Experimental evaluation
- Conclusion and future work
Conclusion and Future work

- Distributed stream view maintenance
  - Absorption Provenance
  - MinShip and Lazy propagation
  - Multi-aggregate selection

- Future work
  - Adaptive cost-based optimizations
Thank you!