Large-Scale Debugging for Datalog

David Zhao, Pavle Subotic, Bernhard Scholz
Introduction
Introduction

- Logic programming (e.g. Datalog) is popular [Aref et al., 2015]
  - Static program analysis
  - Declarative networking
  - Security analysis
- Evaluate at large scale, e.g. hundreds of millions of tuples
- Current debugging approaches do not scale well

We present a new approach to debugging that scales to super large sizes
Datalog

Declarative programming language - logical rules define computation

Example
path(x, y) :- edge(x, y).
path(x, z) :- edge(x, y), path(y, z).

Example Input
edge(1, 2), edge(2, 2), edge(2, 3)

Example Output
path(1, 2), path(2, 2), path(2, 3), path(1, 3)
Debugging in Datalog

Debugging Example
Program produces unexpected output `path(1, 4)`
Where does output come from?

- Debugging in Datalog is difficult
- Imperative language debugging:
  - Inspect values of variables at certain points in program
- In Datalog, we only get the output
  - No notion of variables
  - No notion of time
Provenance as a Debugging Tool

The answer is provenance!

Data Provenance
A way to explain the origins and derivations of data

- Previous approaches for provenance are expensive
  [Deutch et al., 2015, Köhler et al., 2012]

How do we compute provenance efficiently?
Provenance Computation
Proof Trees

A form of provenance - a complete explanation for a tuple

Definition (Proof Trees)

A *proof tree* for a tuple describes how that tuple is derived. The root is the tuple itself, the tree explains which rules are applied and which tuples are used.

Proof trees for $\text{path}(1, 3)$

\[
\frac{\text{edge}(1, 2) \quad \text{edge}(2, 3)}{\text{path}(1, 3)} (r_1) \\
\frac{\text{path}(2, 3)}{} (r_2)
\]

\[
\frac{\text{edge}(1, 2) \quad \text{edge}(2, 2) \quad \text{edge}(2, 3)}{\text{path}(1, 3)} (r_1) \\
\frac{\text{path}(2, 3)}{} (r_2)
\]

\[
\frac{\text{edge}(1, 2) \quad \text{path}(2, 3)}{\text{path}(1, 3)} (r_2)
\]
Fundamental Question

How do we compute a proof tree?
Apply one step of computation repeatedly

One step of computation

- Given a concrete tuple $R(a)$ and rule $R(X) :- R_1(X_1), \ldots R_k(X_k)$
- Want subproof for $R(a)$ - tuples for each atom $R_i(X_i)$ which generate $R(a)$

If we can do one step of computation, we can apply it recursively to get the full proof tree
Naïve Encoding

Directly store the subproof and rule for each tuple

Path program

\[
\text{path}(x, y) :- \text{edge}(x, y).
\]

\[
\text{path}(x, z) :- \text{edge}(x, y), \text{path}(y, z).
\]

<table>
<thead>
<tr>
<th>Path</th>
<th>Subproof</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2)</td>
<td>\text{edge}(1, 2)</td>
<td>\text{r}_1</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>\text{edge}(2, 3)</td>
<td>\text{r}_1</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>\text{edge}(1, 2), \text{path}(2, 3)</td>
<td>\text{r}_2</td>
</tr>
</tbody>
</table>
Naïve Encoding

Directly store the subproof and rule for each tuple

- Can directly query for a subproof
- Storing full provenance is expensive
Guided SLD

What information do we actually need for a subproof?

- Tuples matching the body of a rule
- Form the next level up in a proof tree
What information do we actually need for a subproof?

- Tuples matching the body of a rule
- Form the next level up in a proof tree

So, we need

- The rule generating the tuple
- Its level in the proof tree
Guided SLD

A better method - generate annotations for each tuple

- Rule which generated tuple
- Level in proof tree for tuple

Path program

\[
\begin{align*}
\text{path}(x, y) & :\!- \, \text{edge}(x, y). \\
\text{path}(x, z) & :\!- \, \text{edge}(x, y), \text{path}(y, z).
\end{align*}
\]

<table>
<thead>
<tr>
<th>Path</th>
<th>Rule</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 2)</td>
<td>(r_1)</td>
<td>1</td>
</tr>
<tr>
<td>(2, 3)</td>
<td>(r_1)</td>
<td>1</td>
</tr>
<tr>
<td>(1, 3)</td>
<td>(r_2)</td>
<td>2</td>
</tr>
</tbody>
</table>

Finding a subproof

Search for tuples matching the rule with lower level number
Guided SLD

Advantages:

- Only store 2 extra numbers per tuple
- Finds minimum height proof tree - optimality
Guided SLD

Figure: Diagram of guided SLD provenance system
Implementation in Soufflé
Soufflé

- Soufflé [Jordan et al., 2016] is a high-performance, compilation based Datalog engine - used in large-scale real-world applications

Implementation

- Datalog-to-Datalog transformation
- Guided SLD
  - Soufflé evaluation modification - standard set enforcement fails with annotations
  - Modified existing Soufflé machinery for subproof search
Provenance Query System

On-demand query interface

Figure: Provenance Query Interface
Experiments and Results
Overhead vs Normal Soufflé on Doop

Industry standard Doop DaCapo benchmarks

- Points-to analysis framework for Java
- Hundreds of millions of output tuples
Overhead vs Normal Soufflé on Doop

Industry standard Doop DaCapo benchmarks

- Points-to analysis framework for Java
- Hundreds of millions of output tuples

Figure: Runtime overhead of guided SLD

Figure: Memory usage overhead of guided SLD
Comparisons

Compared to state-of-the-art method (top-k [Deutch et al., 2015])

- Instrument Datalog for single query, and run on Soufflé
Comparisons

Compared to state-of-the-art method (top-k [Deutch et al., 2015])

- Instrument Datalog for single query, and run on Soufflé

Figure: Results of Datalog evaluation time

Figure: Results of Datalog evaluation memory usage
Proof Construction Time

**Figure**: Distribution of proof tree heights for DaCapo

**Figure**: Proof tree construction time vs. size
Conclusion
Conclusion

- Debugging in Datalog is difficult
- Developed a solution to efficiently generate provenance information
- Demonstrated viability with large-scale real world data

Future Work

- Optimise Soufflé for guided SLD
- Provenance for negated Datalog
The End

References


