SociaLite: A Datalog-based Language for Large-Scale Graph Analysis

Jiwon Seo & Monica Lam

MOBISOCIAL RESEARCH LABORATORY

Overview

1. Introduction to SociaLite
2. Datalog extensions for sequential execution
3. Datalog extensions for parallel execution
4. Evaluation on a single-core machine
5. Evaluation on parallel machines
1. Motivation & Challenges

- Analysis of large-scale graphs are important
  - IT Industry – Twitter, LinkedIn, FaceBook, Bio-informatics, etc

- Challenges
  - Difficulty of distributed programming
  - Complexity of graph algorithms

State of the Art Technology

- MapReduce
  - MapReduce based graph systems (HaLoop)
- Pregel
  - vertex-centric programming

→ Too low-level programming model
Distributed Graph Language

- SociaLite
  - Abstractions for graph algorithms
  - Datalog-based query language
  - Graph algorithms in high-level language, and compiled to parallel/distributed code

Single-Source Shortest Paths (SSSP)

- Shortest distances from a single source node to the rest of the nodes in a graph
- Core graph algorithm & a running example in this talk
SSSP in Datalog

- Three tables to store the graph and path distances
  - Edge(src, target, length).
  - Path(target, distance).
  - MinPath(target, minimum-distance).

Path(t, d) :- Edge(1, t, d).  \hspace{1cm} \text{(1)}
Path(t, d) :- Path(s, d₁), Edge(s, t, d₂), d=d₁+d₂.  \hspace{1cm} \text{(2)}
MinPath(t, $\min(d)$) :- Path(t,d).  \hspace{1cm} \text{(3)}

- Does not terminate in the presence of cycles!

Datalog Performance

- Execution times for SSSP

<table>
<thead>
<tr>
<th></th>
<th>Exec Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlog</td>
<td>24.9</td>
</tr>
<tr>
<td>IRIS</td>
<td>12.0</td>
</tr>
<tr>
<td>LogicBlox</td>
<td>3.4</td>
</tr>
<tr>
<td>Java**</td>
<td>0.1</td>
</tr>
</tbody>
</table>

- Datalog 30~250 times slower

* synthetic graph with 100K nodes, 1M edges
** Java program implemented in Dijkstra’s algorithm
Why is Datalog Slow?

- Execution time wasted in sub-optimal paths

\[
\begin{align*}
\text{Path}(t, d) & : \text{Edge}(1, t, d). \quad (1) \\
\text{Path}(t, d) & : \text{Path}(s, d_1), \text{Edge}(s, t, d_2), d=d_1+d_2. \quad (2) \\
\text{MinPath}(t, \min(d)) & : \text{Path}(t,d). \quad (3)
\end{align*}
\]

- For acyclic graphs, generating all possible paths is inefficient
- Dijkstra’s algorithm:
  Prioritize the evaluation: closest node first

Why is Datalog Slow?

- Inefficient data structure for graphs

Java, adjacency list vs Datalog, flat table
2. Extensions for Sequential Execution

- Recursive aggregate functions
- Tail-nested tables (data layout extension)

---

Recursive Aggregate Functions

- Aggregate functions inside recursion
  → Pruning suboptimal answers
- Syntax and Semantics

\[
\text{Path}(t, \min(d)) \leftarrow \text{Edge}(1, t, d); \\
\quad \text{Path}(s, d_1), \text{Edge}(s, t, d_2), d = d_1 + d_2.
\]

Shortest Paths in SociaLite

- \(\min\) is a “meet” operator → partial order on Path
- Naïve evaluation generates the greatest fixed point
- Prioritize naïve evaluation using smallest Paths → Dijkstra’s algorithm!
Tail-Nested Tables

- Tables with nesting at the last column
- Annotation in table declarations
  e.g. Edge(int s, (int t)).
- Data as index
  - Range annotation (N1..N2) → array index
  e.g. Edge(int s:1..10, (int t)).

Tail-Nested Tables (cont.)

- Edge(int s, int t).
- Edge(int s, (int t)).
- Edge(int s:0..10, (int t)).
3. Extensions for Parallel Execution
Distributed Tables

- Partitioning (sharding) by first column
- Hash-based partitioning

\[
\text{Edge}(\text{int src}, (\text{int target})).
\]

Machine 1

\[
\begin{array}{c}
1 \\
7 \\
9 \\
10 \\
11 \\
\end{array}
\]

Machine 2

\[
\begin{array}{c}
2 \\
5 \\
7 \\
9 \\
\end{array}
\]

Distributed Tables (cont.)

- Range-based partitioning

\[
\text{Edge}(\text{int src:0..10}, (\text{int target})).
\]

Machine 1

\[
\begin{array}{c}
1 \\
2 \\
5 \\
7 \\
9 \\
10 \\
\end{array}
\]

Machine 2

\[
\begin{array}{c}
7 \\
11 \\
\end{array}
\]
Distributed Execution

```
Foo(int a, int b).
Bar(int a, int b).
Qux(int a, int b).
Foo(a, c) :- Bar(a, b), Qux(b, c).
```

Parallel Prioritization of Recursive Aggregation

```
Path(t, $min(d)) :- t=0, d=0;
    :- Path(s, d_1), Edge(s, t, d_2), d=d_1+d_2.
```

\[ \Delta \text{Path} = \begin{cases} 
\text{Path}(1, 1.1) \\
\text{Path}(2, 1.5) \\
\text{Path}(1, 1.2) \\
(\text{d < 2.0}) \\
\text{Path}(3, 2.1) \\
\text{Path}(4, 2.7) \\
\text{Path}(0, 2.3) \\
(2.0 < \text{d < 3.0}) \\
\text{Path}(9, 3.1) \\
\text{Path}(7, 3.9) \\
\text{Path}(4, 3.3) \\
(3.0 < \text{d < 4.0}) 
\end{cases} \]

\[ \rightarrow \text{Delta-stepping algorithm generalized,} \\
\rightarrow \text{applied to recursive aggregate functions} \\
\rightarrow \text{Apply to the relations in each bucket in parallel} \]
Approximate Computation

- Bloom Filter:
  - Over-approximate membership
    “May be a member”
  - Fast approximate member count
  - Store large intermediate results approximately

Other Optimizations

- Condition Folding
  - Binary search for sorted columns
  e.g. \( \text{Foo}(a,b) :- \text{Bar}(a,b), b > 10. \)

- Pipelining
  - Evaluate multiples rules at once (locality)
  e.g. \( \text{Foo}(a,b) :- \text{Bar}(a,b), b > 10. \)
    \( \text{Baz}(a,b) :- \text{Foo}(a,b), c=b*b. \)
Benchmark algorithms
- Shortest-Paths
- PageRank
- Mutual Neighbors
- Connected Components
- Finding Triangles
- Clustering Coefficients

→ Evaluated on single-core multi-core/distributed cluster

### 4. Input Graph for Single-Core

<table>
<thead>
<tr>
<th>Source</th>
<th>Type</th>
<th>Size</th>
<th>Used for</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>directed</td>
<td>4.8M nodes 68M edges</td>
<td>Shortest Paths PageRank</td>
<td>Intel Xeon 2.80GHz 32GB memory</td>
</tr>
<tr>
<td>Last.fm</td>
<td>undirected</td>
<td>1.7M nodes 6.4M edges</td>
<td>Mutual Neighbors Connected Components Triangle Clustering Coefficients</td>
<td>Intel Core 2 2.66GHz 3GB memory</td>
</tr>
</tbody>
</table>
SociaLite (w/ optimizations)

Optimized Java vs SociaLite
5. Input Graph for Multi-Core

<table>
<thead>
<tr>
<th>Source</th>
<th>Size</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster</td>
<td>120M nodes</td>
<td>Intel Xeon E5-2670 16 cores(8+8)</td>
</tr>
<tr>
<td></td>
<td>2.5B edges</td>
<td>2.60GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20MB last-level cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256GB memory</td>
</tr>
</tbody>
</table>

Parallel Performance (Multi-Core)
**Input Graph for Distributed Evaluation**

<table>
<thead>
<tr>
<th>Source</th>
<th>Size</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic Graph*</td>
<td>up to 268M nodes, 4.3B edges</td>
<td>64 Amazon EC2 Instances Intel Xeon X5570, 8 cores, 23GB memory</td>
</tr>
</tbody>
</table>

*RMAT algorithm, Graph 500 Generator

**Distributed Performance**

- **Shortest Paths**
- **PageRank**
- **Mutual Neighbors**
- **Connected Components**
- **Triangle**
- **Clustering Coefficients**
In collaboration with Intel Parallel Research Lab

- Compared frameworks
  - SociaLite
  - Giraph
  - GraphLab
  - Combinatorial BLAS
- Native Implementation in C, assembly – optimal

**Comparisons of Graph Frameworks**

- Benchmark Algorithms
  - BFS (Breadth First Search)
  - PageRank
  - Collaborative Filtering
  - Triangle

- Evaluation on Intel cluster
  - Intel Xeon E5-2697, 24 cores 2.7GHz, 64GB memory, InfiniBand network
  - Input graph: up to 500M vertices, 16B edges
### Programmability

- **BFS (Breadth First Search)**

<table>
<thead>
<tr>
<th></th>
<th>Lines of Code</th>
<th>Development Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SociaLite</td>
<td>4</td>
<td>10min</td>
</tr>
<tr>
<td>Giraph</td>
<td>200</td>
<td>1~2 hours</td>
</tr>
<tr>
<td>GraphLab</td>
<td>180</td>
<td>1~2 hours</td>
</tr>
<tr>
<td>Combinatorial BLAS</td>
<td>450</td>
<td>a few hours</td>
</tr>
<tr>
<td>Native</td>
<td>&gt; 1000</td>
<td>&gt; A few months</td>
</tr>
</tbody>
</table>
Distributed Execution – Comparison

**Graphs:**
- **Breadth First Search**
- **PageRank**
- **Collaborative Filtering**
- **Triangle**

The graphs show the execution time and time per iteration for each of the considered algorithms (Native, Combblas, Graphlab, Giraph) with varying numbers of machines. The algorithms and tasks include:
- **Breadth First Search**
- **PageRank**
- **Collaborative Filtering**
- **Triangle**
Summary

- Large-scale graph analysis made easy with SociaLite
  - Succinct code (1/10th)
  - High-level abstraction w/ compiler optimizations
- Competitive performance to other graph frameworks