Datalography: Scaling Datalog Graph Analytics on Graph Processing Systems

Vicky Papavasileiou @ UCSD
Walaa Elding Moustafa @LinkedIn
Alin Deutsch, Ken Yocum @UCSD

CNS - October 2017
UCSD
Graphs are important

- Every-day activities: Look for a job, connect with friends, check news, browse internet
Graphs are important

- **Research and business**: personalized health, drug interaction discovery, personalized ads and recommendations
Graphs are important

- Graph analytics are complex and need to process huge amounts of data

**Social scale**
- Facebook
- 1.4B active users
- 1T edges

**Web scale**
- 1B pages
- 1T links

**Brain scale**
- Human connectome
- 100B vertices
- 100T edges
Graphs processing systems

• Abstract synchronization and distribution from developer

• Graph is distributed across machines

• Computation happens in parallel

• Mature, deployed by industry and scale to trillions of edges
Current limitations

Graph analytics

Graph processing systems
Current limitations

Graph analytics → Manual implementation → Graph processing systems
Current limitations

- Use imperative code (Java, C++, Scala)
- Manual optimization (for each analytic)
- Not portable (to other Graph processing systems)
Imperative graph analytics

```java
public class SimplePageRankComputation extends BasicComputation<LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {
    public static final int MAX_SUPERSTEPS = 30;

    @Override
    public void compute(
        Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
        Iterable<DoubleWritable> messages) throws IOException {
        if (getSuperstep() >= 1) {
            double sum = 0;
            for (DoubleWritable message : messages) {
                sum += message.get();
            }
            DoubleWritable vertexValue =
                new DoubleWritable((0.15f / getTotalNumVertices()) + 0.85f * sum);
            vertex.setValue(vertexValue);
        }
        if (getSuperstep() < MAX_SUPERSTEPS) {
            long edges = vertex.getNumEdges();
            sendMessageToAllEdges(vertex,
                new DoubleWritable(vertex.getValue().get() / edges));
        } else {
            vertex.voteToHalt();
        }
    }
```
Datalography

💡 Declarative: Automatically optimize queries on logical level (**Portable optimizations**)

💡 Recursive: Naturally expresses iterative graph analytics (**Concise**)

💡 Performance: Outperform imperative code
Declarative graph analytics

1. \texttt{Degree(x, count<y>) := Edge(x,y).}
2. \texttt{PR(x,0,1).}
3. \texttt{PR(x,i+1,sum<p/d>) := Edge(y,x), PR(y,i,p), Degree(y,d), i<20.}

Why write 30 lines of code when you can write 3?!
Background on Datalog
Datalog
Back from the dead
Datalog
Back from the dead

Used in industry and academia
Datalog
Back from the dead

Used in industry
and academia
Datalog
Back from the dead

Used in industry
and
academia

Declarative Networking
Program Analysis
Distributed Programming
Relational graph representation

The input graph is represented as Vertex and Edge tables
Relational graph representation

- The input graph is represented as **Vertex** and **Edge** tables
Relational graph representation

Input graph

Relational representation

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>a</td>
</tr>
</tbody>
</table>

...    ...

a      0.1
b      0.3
.....  ....
d      0.2
...    ...

a      0.1
b      0.3
.....  ....
c      0.5
d      0.2
.....  ....
A datalog program consists of a set of rules.

Each rule defines a new table. Evaluation adds tuples to table.

Evaluate each rule iteratively until convergence.
Datalog evaluation: PageRank

Input graph

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td>a</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Datalog evaluation: PageRank

Input graph

\[ \text{Degree}(x, \text{count}<y>) \leftarrow \text{Edge}(x, y). \]
Datalog evaluation: PageRank

Degree(x, count < y) ← Edge(x, y).

New table  Aggregation  Input graph
# Datalog evaluation: PageRank

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>a</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

Input graph

<table>
<thead>
<tr>
<th>Degree(x,count&lt;y&gt;) ← Edge(x,y).</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(x,0,1).</td>
</tr>
</tbody>
</table>

Tables created by rule evaluation

<table>
<thead>
<tr>
<th>Degree</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>0.3</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>0.5</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>0.2</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

...
# Datalog evaluation: PageRank

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>a</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input graph

<table>
<thead>
<tr>
<th>Degree(x,count&lt;y&gt;)←Edge(x,y).</th>
<th>PR(x,0,1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR(x,i+1,sum&lt;p/d&gt;)←Edge(y,x), PR(y,i,p), Degree(y,d),i&lt;20.</td>
<td></td>
</tr>
</tbody>
</table>

Tables created by rule evaluation

<table>
<thead>
<tr>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
</tbody>
</table>

Vertices

| a      | 0.1 |
| b      | 0.3 |
| c      | 0.5 |
| d      | 0.2 |
|        |     |

Friday, October 20, 17
## Datalog evaluation: PageRank

**Input graph**

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>c→a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>d→a</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c→d</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>e→a</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Tables created by rule evaluation**

\[
\text{Degree}(x, \text{count}<y>) \leftarrow \text{Edge}(x,y).
\]

\[
\text{PR}(x,0,1).
\]

\[
\text{PR}(x,i+1, \text{sum}<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}(y,i,p), \text{Degree}(y,d), i<20.
\]

**Aggregation**

**Tables created by previous rule evaluations**

\[
\text{PR}.
\]
# Datalog evaluation: PageRank

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Input graph**

**Tables created by rule evaluation**

\[
\text{Degree}(x, \text{count}<y>) \leftarrow \text{Edge}(x, y).
\]

\[
\text{PR}(x, 0, 1).
\]

\[
\text{PR}(x, i+1, \text{sum}<p/d>) \leftarrow \text{Edge}(y, x), \text{PR}(y, i, p), \text{Degree}(y, d), i<20.
\]

**Rule is recursive**
### Datalog evaluation: PageRank

#### Input graph

<table>
<thead>
<tr>
<th>Edge</th>
<th>Vertices</th>
<th>Degree</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>a</td>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

#### Tables created by rule evaluation

<table>
<thead>
<tr>
<th>Degree(x,count&lt;y&gt;)</th>
<th>Edge(x,y)</th>
<th>PR(x,0,1)</th>
<th>PR(x,i+1,sum&lt;p/d&gt;)</th>
<th>Edge(y,x)</th>
<th>PR(y,i,p)</th>
<th>Degree(y,d),i&lt;20</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>a 1</td>
<td></td>
<td></td>
<td>b 1</td>
<td>a 0.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b 0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c 0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d 0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**Rule is recursive**
Distributed setting

- Graph is distributed across workers
- Edges span workers
“Think like a vertex”

- Computation happens in **supersteps** followed by global synchronization

- Vertex function:
  - **Receive** messages from neighbors
  - **Apply** computation
  - **Send** messages to neighbors
“Think like a vertex”

- Computation happens in **supersteps** followed by global synchronization

- Vertex function:
  - **Receive** messages from neighbors
  - **Apply** computation
  - **Send** messages to neighbors
“Think like a vertex”

- Computation happens in **supersteps** followed by global synchronization

- Vertex function:
  - **Receive** messages from neighbors
  - **Apply** computation
  - **Send** messages to neighbors
“Think like a vertex”

- Computation happens in **supersteps** followed by global synchronization

- Vertex function:
  - Receive messages from neighbors
  - Apply computation
  - Send messages to neighbors

![Graph with nodes and edges](image)

Compute rank
“Think like a vertex”

- Computation happens in supersteps followed by global synchronization

- Vertex function:
  - Receive messages from neighbors
  - Apply computation
  - Send messages to neighbors
Datalog on “Think like a vertex”

- Rule evaluated by every vertex
- Tables with rule results created by every vertex
Datalog on “Think like a vertex”

- Tables are exchanged between vertices as messages

⚠️ A lot of computation and communication overhead
Our contributions

• **Super-vertices** = Reduce number of messages

• Query Optimizer
  
  • **Rule locality** = Reduce number of tables sent

  • **Eager-aggregation** = Reduce size of tuples and size of tables

  • **Semi-join** = Reduce number of joins
Super-vertices

- Partition the graph into **super-vertices** minimizing the edge-cut

- A super-vertex contains a group of vertices of the input graph and their edges

- GPS is **agnostic** of the internal vertices and only considers super-vertices for the computation
Every super-vertex evaluates a rule and creates a table

Tables sent as messages between super-vertices

Less computation and communication overhead
Super-vertices

• Every super-vertex evaluates a rule and creates a table

• Tables sent as messages between super-vertices

• Less computation and communication overhead
Super-vertices

Developer is agnostic of super-vertices

Datalog program written at level of vertices

Super-vertices portable to other GPS without change to programming model or underlying architecture
Rule locality: Problem

\[
\begin{align*}
\text{Degree}(x, \text{count}<y>) & \leftarrow \text{Edge}(x, y). \\
\text{PR}(x, 0, 1). \\
\text{PR}(x, i+1, \text{sum}<p/d>) & \leftarrow \text{Edge}(y, x), \ \text{PR}(y, i, p), \ \text{Degree}(y, d), i < 20.
\end{align*}
\]

- Location specifiers: Location of rule evaluation and tables

- \(\text{PR}(x, i+1, \text{sum}<p/d>)\) evaluated at vertex \(x\) but requires tables from neighbors \(y\)

- Cannot be evaluated locally
Rule locality: Problem

\[\text{Degree}(x, \text{count}<y>) \leftarrow \text{Edge}(x,y).\]
\[\text{PR}(x,0,1).\]
\[\text{PR}(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}(y,p), \text{Degree}(y,d), i < 20.\]

- Location specifiers: Location of rule evaluation and tables
- \(\text{PR}(x,i+1,\text{sum}<p/d>)\) evaluated at vertex \(x\) but requires tables from neighbors \(y\)
- Cannot be evaluated locally
Optimization: Rule locality

Initial rule

\[
PR(x,i+1,sum<p/d>) \leftarrow Edge(y,x), PR(y,i,p), Degree(y,d), i<20.
\]

Rewritten rule

\[
PR_2(y,i,p,d) \leftarrow PR(y,i,p), Degree(y,d).
PR(x,i+1,sum<p/d>) \leftarrow Edge(y,x), PR_2(y,i,p,d), i<20.
\]

- Split \( PR \) into two rules, \( PR_2 \) and \( PR \)
- \( PR_2 \) refers only to \( y \) and can be evaluated locally
Optimization: Rule locality

**Initial rule**

\[ \text{PR}(x,i+1,sum<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}(y,i,p), \text{Degree}(y,d), i < 20. \]

**Rewritten rule**

\[ \text{PR}_2(y,i,p,d) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d). \]

\[ \text{PR}(x,i+1,sum<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,p,d), i < 20. \]

- Split \text{PR} into two rules, \text{PR}_2 and \text{PR}
- \text{PR}_2 refers only to \text{y} and can be evaluated locally
Optimization: Rule locality

Initial rule

\[ PR(x, i+1, \text{sum}<p/d>) \leftarrow \text{Edge}(y, x), PR(y, i, p), Degree(y, d), i<20. \]

Rewritten rule

\[ PR_2(y, i, p, d) \leftarrow PR(y, i, p), Degree(y, d). \]
\[ PR(x, i+1, \text{sum}<p/d>) \leftarrow \text{Edge}(y, x), PR_2(y, i, p, d), i<20. \]

- Split \( PR \) into two rules, \( PR_2 \) and \( PR \)
- \( PR_2 \) refers only to \( y \) and can be evaluated locally
Optimization: Rule locality

- Only one table sent as message instead of two
Optimization: Rule locality

- Only one table sent as message instead of two
Optimization: Eager-aggregation

- Eager-aggregation first used in relational query optimization
  - Perform aggregation before joins to minimize the input to the join
  - Aggregation function must be decomposable

- Datalography performs aggregation before and after messages are sent
  - **Reduce size and number of messages**
Reducing super-vertex communication

- SV_2 and SV_3 send a PR_2 table with a tuple per vertex with degree and rank
Reducing super-vertex communication

- $SV_2$ and $SV_3$ send a $PR_2$ table with a tuple per vertex with degree and rank

\[
\begin{array}{|c|c|c|c|}
\hline
PR_2 & \text{SuperVertex 1} \\
\hline
\text{c} & 1 & 0.5 & 2 \\
\text{d} & 1 & 0.2 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
PR_2 & \text{SuperVertex 3} \\
\hline
\text{e} & 1 & 0.4 & 1 \\
\hline
\end{array}
\]
Reducing super-vertex communication

- SV_2 and SV_3 send a PR_2 table with a tuple per vertex with degree and rank
Reduce size of tuples

Instead of sending both rank and degree, send their quotient.

<table>
<thead>
<tr>
<th>PR2</th>
<th>c</th>
<th>1</th>
<th>0.5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1</td>
<td>0.2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PR2</th>
<th>e</th>
<th>1</th>
<th>0.4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Reduce size of tuples

Instead of sending both rank and degree, send their quotient
Reduce number of tuples

Since sum is decomposable, sum the rank at the sender and the receiver

SV₂ sends one tuple that is the sum of ranks
Reduce number of tuples

Since sum is decomposable, sum the rank at the sender and the receiver

SV₂ sends one tuple that is the sum of ranks
Eager-aggregation: Rewriting

Initial rule

\[ \text{PR}_2(y,i,p,d) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d) \]
\[ \text{PR}(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,p,d), i<20. \]

Rewritten rule

\[ \text{PR}_2(y,i,\text{sum}<p/d>) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d) \]
\[ \text{PR}(x,i+1,\text{sum}<r>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,r), i<20. \]

- Send one value per tuple
- Send one tuple per destination vertex
Eager-aggregation: Rewriting

Initial rule

\[ \text{PR}_2(y,i,p,d) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d) \]
\[ \text{PR}(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,p,d), i<20. \]

Rewritten rule

\[ \text{PR}_2(y,i,\text{sum}<p/d>) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d) \]
\[ \text{PR}(x,i+1,\text{sum}<r>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,r), i<20. \]

- Send **one value** per tuple
- Send **one tuple** per destination vertex
Eager-aggregation: Rewriting

Initial rule

\[ PR_2(y,i,p,d) \leftarrow PR(y,i,p), \text{Degree}(y,d) \]
\[ PR(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), PR_2(y,i,p,d), i<20. \]

Rewritten rule

\[ PR_2(y,i,\text{sum}<p/d>) \leftarrow PR(y,i,p), \text{Degree}(y,d) \]
\[ PR(x,i+1,\text{sum}<r>) \leftarrow \text{Edge}(y,x), PR_2(y,i,r), i<20. \]

- Send **one value** per tuple
- Send **one tuple** per destination vertex
Eager-aggregation: Rewriting

Initial rule

\[ PR_2(y,i,p,d) \leftarrow PR(y,i,p), \text{Degree}(y,d) \]
\[ PR(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), PR_2(y,i,p,d), i<20. \]

Rewritten rule

\[ PR_2(y,i,\text{sum}<p/d>) \leftarrow PR(y,i,p), \text{Degree}(y,d) \]
\[ PR(x,i+1,\text{sum}<r>) \leftarrow \text{Edge}(y,x), PR_2(y,i,r), i<20. \]

- Send one value per tuple
- Send one tuple per destination vertex
Semi-join: Example

- **Send** messages: Join with *Edges* to find neighbors
Semi-join: Example

*Send* messages: Join with *Edges* to find neighbors
Semi-join: Example

Input graph

<table>
<thead>
<tr>
<th>In Edge</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

Input graph

<table>
<thead>
<tr>
<th>Out Edge</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Message

<table>
<thead>
<tr>
<th>PR₂</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c,d</td>
<td>1</td>
</tr>
<tr>
<td>c,d</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Message

- **Receive** messages: Join with **Edges** to find destination vertex
Semi-join: Example

- Two joins needed for every message
Reduce number of joins

- Include destination vertex in message
- Eliminate join needed at destination
Reduce number of joins

- Include destination vertex in message
- Eliminate join needed at destination
Reduce number of joins

<table>
<thead>
<tr>
<th></th>
<th>PR$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>

Include destination vertex in message

Eliminate join needed at destination
Semi-join: Rewriting

Initial rule

\[ PR_2(y,i,p,d) \leftarrow PR(y,i,p), \ Degree(y,d) \]
\[ PR(x,i+1,sum<\frac{p}{d}>) \leftarrow Edge(y,x), \ PR_2(y,i,p,d), \ i<20. \]

Rewritten rule

\[ PR_2(x,i,\frac{p}{d}) \leftarrow PR(y,i,p), \ Degree(y,d), \ Edge(y,x) \]
\[ PR(x,i+1,sum<r>) \leftarrow PR_2(x,i,r), \ i<20. \]

- One join per message
Semi-join: Rewriting

Initial rule

\[
\begin{align*}
& \text{PR}_2(y,i,p,d) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d) \\
& \text{PR}(x,i+1,\text{sum}<p/d>) \leftarrow \text{Edge}(y,x), \text{PR}_2(y,i,p,d), i<20.
\end{align*}
\]

Rewritten rule

\[
\begin{align*}
& \text{PR}_2(x,i,p/d) \leftarrow \text{PR}(y,i,p), \text{Degree}(y,d), \text{Edge}(y,x) \\
& \text{PR}(x,i+1,\text{sum}<r>) \leftarrow \text{PR}_2(x,i,r), i<20.
\end{align*}
\]

- One join per message
Experiments
• Extended Giraph 1.1

• Open source, used by industry, fits well into Hadoop ecosystem

• A super-vertex is mapped to a Giraph vertex

• Worker evaluates rules, Master checks convergence
Experiments: Setup

- 7 nodes, 32GB RAM, 4 vCPUs
- Metis for partitioning the graph into super-vertices
- Algorithms: Single Source Shortest Paths (SSSP), PageRank (PR)

Datasets from [http://law.di.unimi.it/datasets.php](http://law.di.unimi.it/datasets.php)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>V</th>
<th>E</th>
<th>Avg Degree</th>
<th>Avg Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK-2</td>
<td>18.5M</td>
<td>298M</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>AR</td>
<td>22.7M</td>
<td>640M</td>
<td>28</td>
<td>22</td>
</tr>
<tr>
<td>UK-5</td>
<td>39.4M</td>
<td>936M</td>
<td>24</td>
<td>23</td>
</tr>
</tbody>
</table>
SSSP: Running time

- Speed-up over Giraph: 4x
- Speed-up over Giraph-Metis: 1.3x-3x
PageRank: running time

Running time in sec

UK-2

AR

UK-5

Speed-up over Giraph: 4.6x - 5.8x
Speed-up over Giraph-Metis: 1.5x - 2x

Friday, October 20, 17
Upper bound (UK-2): \#edges \times \#iterations = 5,962,275,240
Giraph: 5,844,873,260
Datalog: 6,715,819
Upper bound: \#edges x \#iterations x 16 bytes = 95,396,403,840
Giraph: 93,571,709,017
Datalog: 13,391,803,945
Conclusion

• Datalog evaluation engine on “think like a vertex”

• Logical level optimizations (super-vertices, query rewritings)

  • Reduce *communication* cost

  • Portable (Independent of underlying infrastructure)

  • Performance *gains*