EmptyHeaded: A Relational Engine for Graph Processing

Authors: Christopher R. Aberger, Susan Tu, Kunle Olukotun, Christopher Ré

Presenter: Jingnan Shi
Outline

- Introduction
- Preliminaries
- Query Compiler
- Code Generation
- Execution Engine Optimizer
- Experiments
- Questions

https://github.com/HazyResearch/EmptyHeaded
Introduction

Low-level Graph Engines

1) Iterators and domain-specific primitives
2) Optimized data layouts

**Drawbacks:**
Require users to write code imperatively

**Examples:**
Powergraph, Galois, SNAP, Ligra, ...

---

High-level Graph Engines

Supports tasks using query languages

**Drawbacks:**
1) Performance gap

**Examples:**
SociaLite, LogicBlox, Grail
Introduction

Low-level Graph Engines

1) Iterators and domain-specific primitives
2) Optimized data layouts

**Drawbacks:**
Require users to write code imperatively

**Examples:**
Powergraph, Galois, SNAP, Ligra, ...

High-level Graph Engines

Supports tasks using query languages

**Drawbacks:**
1) Performance gap

**Examples:**
SociaLite, LogicBlox, Grail

V.S.

- SQL/Datalog query interface
- Worst-case Optimal
- Optimized data layout and code generation
Preliminaries: Datalog

Facts: tuples in the database

Rules: queries

<table>
<thead>
<tr>
<th>StudentID</th>
<th>GPA</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>4.8</td>
<td>2</td>
</tr>
</tbody>
</table>

Schemas:
academic(studentID, GPA, Course)
info(studentID, dorm, class)

Facts:
academic(0, 4.5, 6)
academic(1, 4.8, 2)
info(0, Ashdown, 2019)
info(1, SidPac, 2020)

Rules/Queries:
q(x) :- academic(x,y,z), z=6
Find students in course 6

q(x) :- academic(x,y,2), info(x, ‘SidPac’, w)
Find student in course 2 who lives in SidPac
Preliminaries: Datalog

In general:

\[ Q(x_1, x_2, ..., x_n) := R_1(\text{args}), R_2(\text{args}), ... \]

- LHS: head
- RHS: body
- This is a conjunctive query:
  - \( R_i \) returns true if the relation contains the tuple described by the input arguments
  - Each of the \( R_i \) is called a subgoal, and the query results / tuples returned have to satisfy all of them
- Also can be expressed as natural join queries
  - \( Q(x_1, x_2, ..., x_n) := R_1(\text{args}) \bowtie R_2(\text{args}) \bowtie ... \)
Preliminaries: Query as a Hypergraph

- $H = (V, E)$
- $V$: non empty set of vertices
- $E$: hyperedges
  - Can connect more than two vertices
- Each vertex represents an attribute/variable in the body of the query
  - The “args”
- Each hyperedge represents a relation
- Eg: $q :- R(x,y), S(y,z), T(z,x)$
  - $V = \{x,y,z\}$
  - $E = \{(x,y), (y,z), (z,x)\}$
Preliminaries: Worst-Case Optimal Join

- Evaluating conjunctive queries are **NP-complete** in terms of combined complexity
  - Combined complexity: Query + Input database
- Thus, we want to consider algorithms with respect to both input and output sizes
- The AGM (Atserias, Grohe, and Marx) bound tightly bounds the worst-case size of a join query using a notion called a **fractional (edge) cover**.

Definition of a fractional cover:

For each \( v \in V \) we have \( \sum_{e \in E : e \ni v} x_e \geq 1 \)

where \( x_e \) is a weight vector indexed by edges, and \( H = (V, E) \) is a fixed hypergraph.

Definition of the AGM bound:

\[
|\text{OUT}| \leq \prod_{e \in E} |R_e|^{x_e}
\]

where \( R_e \) is the size of the relation represented by edge \( e \).
**Preliminaries: Worst-Case Optimal Join**

Example: triangle query

\[ R(x, y) \Join S(y, z) \Join T(x, z) \]

Feasible cover: \((1,1,0)\)

AGM bound: \(N^2\)

Another feasible cover: \((\frac{1}{2}, \frac{1}{2}, \frac{1}{2})\)

AGM bound: \(N^{3/2}\)

This bound is tight: consider a complete graph with \(\sqrt{N}\) vertices. On it this query produces \(\Omega(N^{3/2})\) tuples.

Definition of a fractional cover:

for each \(v \in V\) we have \(\sum_{e \in E : e \ni v} x_e \geq 1\)

where \(x_e\) is a weight vector indexed by edges, and \(H = (V, E)\) is a fixed hypergraph.

Definition of the AGM bound:

\[ |\text{OUT}| \leq \prod_{e \in E} |R_e|^{x_e} \]

where \(R_e\) is the size of the relation represented by edge \(e\)
We define worst-case optimal join algorithms as those that evaluate a full conjunctive query in time that is proportional to the worst-case output size of the query.

- The NPRR algorithm is one of them.
- NPRR has the so called min property:
  - the running time of the intersection algorithm is upper bounded by the length of the smaller of the two input sets.

**Algorithm 1** Generic Worst-Case Optimal Join Algorithm

```plaintext
// Input: Hypergraph H = (V, E), and a tuple t.
Generic-Join(V, E, t):
    if |V| = 1 then return \( \cap_{e \in E} R_e[t] \).
    Let I = \{v_1\} // the first attribute.
    Q ← \emptyset // the return value
    // Intersect all relations that contain v_1
    // Only those tuples that agree with t.
    for every \( t_v \in \cap_{e \in E: e \ni v_1} \pi_I(R_e[t]) \) do
        \( Q_t \leftarrow \text{Generic-Join}(V - I, E, t :: t_v) \)
        Q ← Q ∪ \{t_v\} × Q_t
    return Q
```
Preliminaries: Input and Output Data Structures

- Dictionary encoding maps original data values to 32 bit unsigned integer keys
- Sets of values can be annotated with data values for aggregations
  - For example, a two-level trie annotated with a float value represents a sparse matrix or graph with edge properties.
- Depth of the trie equals to the arity of the relation
- A tuple can be obtained by simply getting the path from root to leaf

Figure 2: EmptyHeaded transformations from a table to trie representation using attribute order (managerID, employerID) and employerID attribute annotated with employeeRating.
EmptyHeaded: Overview

**Input**

<table>
<thead>
<tr>
<th>Data</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>K_3(x,y,z) := R(x,y), R(y,z), R(x,z).</td>
</tr>
</tbody>
</table>

**Query Compiler**

\[
\begin{align*}
\mathcal{V}_0 & : \lambda : R \\
\mathcal{X} & : x, y, z \\
\end{align*}
\]

**Generated Code**

\[
\begin{align*}
\mathcal{S}_x & := (\pi_x R \cap \pi_x R) \\
\text{for } x \text{ in } \mathcal{S}_x: \\
\mathcal{S}_y & := (\pi_y R[x] \cap \pi_y R) \\
\text{for } y \text{ in } \mathcal{S}_y: \\
\mathcal{S}_z & := (\pi_z R[y] \cap \pi_z R[x]) \\
\text{for } z \text{ in } \mathcal{S}_z: \\
K_3 & \cup (x, y, z)
\end{align*}
\]

**Execution Engine**

Skew? \rightarrow Layout Optimizer

**Output**

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>K_3</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
<tr>
<td>012</td>
</tr>
</tbody>
</table>
## Query Optimizer: Sample Queries

<table>
<thead>
<tr>
<th>Name</th>
<th>Query Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>Triangle(x,y,z) :- R(x,y),S(y,z),T(x,z).</td>
</tr>
<tr>
<td>4-Clique</td>
<td>4Clique(x,y,z,w) :- R(x,y),S(y,z),T(x,z),U(x,w),V(y,w),Q(z,w).</td>
</tr>
<tr>
<td>Lollipop</td>
<td>Lollipop(x,y,z,w) :- R(x,y),S(y,z),T(x,z),U(x,w).</td>
</tr>
<tr>
<td>Barbell</td>
<td>Barbell(x,y,z,x’,y’,z’) :- R(x,y),S(y,z),T(x,z),U(x,x’),R’(x’,y’),S’(y’,z’),T’(x’,z’).</td>
</tr>
<tr>
<td>Count Triangle</td>
<td>CountTriangle(;w:long) :- R(x,y),S(x,z),T(x,z); w=&lt;COUNT(*)&gt;.</td>
</tr>
</tbody>
</table>
| PageRank   | N(;w:int) :- Edge(x,y); w=<COUNT(x)>.
|            | PageRank(x;y:float) :- Edge(x,z); y = 1/N.
|            | PageRank(x;y:float)*[i=5] :- Edge(x,z),PageRank(z),InvDeg(z); y=0.15+0.85*<SUM(z)>.
| SSSP       | SSSP(x;y:int) :- Edge("start",x); y=1.
|            | SSSP(x;y:int)* :- Edge(w,x),SSSP(w); y=<MIN(w)>+1.                        |

<table>
<thead>
<tr>
<th>Name</th>
<th>Query Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Example Queries in EmptyHeaded
Query Optimizer: Using Queries

Join:

```sql
db.eval("Triangle(a,b,c) :- Edge(a,b),Edge(b,c),Edge(a,c).")
```

Project:

```sql
db.eval("Triangle(a,b) :- Edge(a,b),Edge(b,c),Edge(a,c).")
```

Selection:

```sql
db.eval("""FliqueSel(a,b,c,d) :- x=0, Edge(a,b),Edge(b,c),Edge(a,c), Edge(a,d),Edge(b,d),Edge(c,d),Edge(a,x).""")
```

You can also use SQL:

```sql
CREATE TABLE Triangle AS (  SELECT e1.a, e2.a, e3.a  FROM Edge e1  JOIN Edge e2 ON e1.b = e2.a  JOIN Edge e3 ON e2.b = e3.a  AND e3.b = e1.a  )""", useSql=True)```
Query Optimizer: Generalized Hypertree Decomposition (GHD)

- Nodes represent a join and projection operation
- Edges represent data dependencies
- Given a query, there exists many different GHDs
- Need to find the GHD with the lowest cost

Figure 3: We show the Barbell query hypergraph and two possible GHDs for the query. A node \( v \) in a GHD captures which relations should be joined with \( \lambda(v) \) and which attributes should be retained with projection with \( \chi(v) \).
Query Optimizer: Generalized Hypertree Decomposition (GHD)

Formal definition:

Let $H$ be a hypergraph. A generalized hypertree decomposition (GHD) of $H$ is a triple $D = (T; \chi; \lambda)$, where:

- $T(V(T), E(T))$ is a tree
- $\chi : V(T) \rightarrow 2^{V(H)}$ is a function associating a set of vertices $\chi(v) \subseteq V(H)$ to each node $v$ of $T$;
- $\lambda : V(T) \rightarrow 2^{E(H)}$ is a function associating a set of hyperedges to each vertex $v$ of $T$;

The following properties hold:

1. For each $e \in E(H)$, there is a node $v \in V(T)$ such that $e \subseteq \chi(v)$ and $e \in \lambda(v)$
2. For each $t \in V(H)$, the set $\{v \in V(T) \mid t \in \chi(v)\}$ is connected in $T$.
3. For every $v \in V(T)$, $\chi(v) \subseteq \cup \lambda(v)$. 
Query Optimizer: Compute outputs from GHDs

- Define $Q_v$ as the query formed by joining the relations in $\lambda(v)$.
- Width $w$ of a GHD:
  - $\text{AGM}(Q_v)$
- Given a GHD with width $w$, there is a simple algorithm to run in time $O(N^w + \text{OUT})$.
  - First, run any worst-case optimal algorithm on $Q_v$ for each node $v$ of the GHD; each join takes time $O(N^w)$ and produces at most $O(N^w)$ tuples.
  - Then, run Yannakakis’ algorithm which enables us to compute the output in linear time in the input size ($O(N^w)$) plus the output size ($\text{OUT}$).

EmptyHeaded brute force all GHDs of all possible widths, because number of relations and attributes is typically small.
Query Optimizer: Code Generation from GHD

- The goal is to translate GHDs into operations listed on the right.
- For each node, generate the code using the worst-case optimal join algorithm.
- The nodes are access first in a bottom up pass, then the result is constructed by walking down the tree in a top-down pass.
- Handles recursion through both naive evaluation and semi-naive evaluation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trie ((R))</td>
<td>(R[t]) Returns the set matching tuple (t \in R).</td>
</tr>
<tr>
<td></td>
<td>(R \leftarrow R \cup t \times xs) Appends elements in set (xs) to tuple (t \in R).</td>
</tr>
<tr>
<td>Set ((xs))</td>
<td>for (x) in (xs) Iterates through the elements (x) of a set (xs).</td>
</tr>
<tr>
<td></td>
<td>(xs \cap ys) Returns the intersection of sets (xs) and (ys).</td>
</tr>
</tbody>
</table>

Table 2: Execution Engine Operations
Execution Engine Optimizer: Layouts

Figure 4: Example of the bitset layout that contains \( n \) blocks and a sequence of offsets \( (o_1-o_n) \) and blocks \( (b_1-b_n) \). The offsets store the start offset for values in the bitvectors.

Two layouts

UINT (for sparse data)

- Just an array of 32-bit unsigned integers

BITSET (for dense data)

- Stores a set of pairs (offset, bitvector).
- Offsets are indices of the smallest values in the bitvectors.
- Offsets are packed contiguously.

Associated Values:

- Layouts depend on the layouts of the set
- For the bitset layout:
  - store the associated values as a dense vector (where associated values are accessed based upon the data value in the set).
- For the UINT layout:
  - store the associated values as a sparse vector (where the associated values are accessed based upon the index of the value in the set)
Execution Engine Optimizer: Intersection Algorithms

**UINT ∩ UINT**

- Sizes of the two sets might be drastically different
  - Cardinality skew
- A simple hybrid algorithm that selects a SIMD **galloping algorithm** when the ratio of cardinalities is greater than 32:1, and a SIMD **shuffling algorithm** otherwise.

**BITSET ∩ UINT**

- First intersect the uint values with the offsets in the bitset.
- For each matching uint and bitset block we check whether the corresponding bitset blocks contain the uint value by probing the block.

**BITSET ∩ BITSET**

- Intersect offset firsts
- Then intersect blocks using SIMD AND
- The best case:
  - all bits in the register are 1, a single hardware instruction computes the intersection of 256 values.
Figure 5: Intersection time of \texttt{uint} and \texttt{bitset} layouts for different densities.

Figure 6: Intersection time of layouts for sets with different sizes of dense regions.
Execution Engine Optimizer: Layout Selection Granularity

Relation level:
- Force the data in all relations to be stored using the same layout
  - Does not address density skew
- UINT provides the best performance

Set level:
- Decide on a per-set level if the entire set should be represented using a UINT or a BITSET layout.

Block level:
- Regards the domain as a series of fixed-sized blocks; we represent sparse blocks using the UINT layout and dense blocks using the BITSET layout

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Relation level</th>
<th>Set level</th>
<th>Block level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>7.3x</td>
<td>1.1x</td>
<td>3.2x</td>
</tr>
<tr>
<td>Higgs</td>
<td>1.6x</td>
<td>1.4x</td>
<td>2.4x</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>1.3x</td>
<td>1.4x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Orkut</td>
<td>1.4x</td>
<td>1.4x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Patents</td>
<td>1.2x</td>
<td>1.6x</td>
<td>1.9x</td>
</tr>
</tbody>
</table>

Table 4: Relative time of the level optimizers on triangle counting compared to the oracle.

- Selecting layouts on a set level works best on real-world graphs.
- It selects the BITSET layout when each value in the set consumes at most as much space as a SIMD (AVX) register and the UINT layout otherwise.
Experiments: Setup

- 5 datasets are used in tests.
- Low-level Engines Tested:
  - PowerGraph, CGT-X, Snap-R
  - No Ligra :(
- High-level Engines Tested:
  - LogicBlox, SociaLite
- Run on a single machine with 48 cores on four Intel Xeon E5-4657L v2 CPUs and 1 TB of RAM.
Experiments: Results

Triangle Counting:

- Outperforms other baselines by 2x - 60x
- Speedups most significant on datasets with large density skew

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EH</th>
<th>PG</th>
<th>CGT-X</th>
<th>SR</th>
<th>SL</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>0.31</td>
<td>8.40x</td>
<td>62.19x</td>
<td>4.18x</td>
<td>1390.75x</td>
<td>83.74x</td>
</tr>
<tr>
<td>Higgs</td>
<td>0.15</td>
<td>3.25x</td>
<td>57.96x</td>
<td>5.84x</td>
<td>387.41x</td>
<td>29.13x</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>0.48</td>
<td>5.17x</td>
<td>3.85x</td>
<td>10.72x</td>
<td>225.97x</td>
<td>23.53x</td>
</tr>
<tr>
<td>Orkut</td>
<td>2.36</td>
<td>2.94x</td>
<td>-</td>
<td>4.09x</td>
<td>191.84x</td>
<td>19.24x</td>
</tr>
<tr>
<td>Patents</td>
<td>0.14</td>
<td>10.20x</td>
<td>7.45x</td>
<td>22.14x</td>
<td>49.12x</td>
<td>27.82x</td>
</tr>
<tr>
<td>Twitter</td>
<td>56.81</td>
<td>4.40x</td>
<td>-</td>
<td>2.22x</td>
<td>t/o</td>
<td>30.60x</td>
</tr>
</tbody>
</table>

Table 5: Triangle counting runtime (in seconds) for Empty-Headed (EH) and relative slowdown for other engines including PowerGraph (PG), a commercial graph tool (CGT-X), Snap-Ringo (SR), SociaLite (SL) and LogicBlox (LB). 48 threads used for all engines. "-" indicates the engine does not process over 70 million edges. "t/o" indicates the engine ran for over 30 minutes.
Experiments: Results

PageRank:

- 2x - 4x faster than compared
- An order of magnitude faster than high-level graph engines compared

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EH</th>
<th>G</th>
<th>PG</th>
<th>CGT-X</th>
<th>SR</th>
<th>SL</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>0.10</td>
<td>0.021</td>
<td>0.24</td>
<td>1.65</td>
<td>0.24</td>
<td>1.25</td>
<td>7.03</td>
</tr>
<tr>
<td>Higgs</td>
<td>0.08</td>
<td>0.049</td>
<td>0.5</td>
<td>2.24</td>
<td>0.32</td>
<td>1.78</td>
<td>7.72</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>0.58</td>
<td>0.51</td>
<td>4.32</td>
<td>-</td>
<td>1.37</td>
<td>5.09</td>
<td>25.03</td>
</tr>
<tr>
<td>Orkut</td>
<td>0.65</td>
<td>0.59</td>
<td>4.48</td>
<td>-</td>
<td>1.15</td>
<td>17.52</td>
<td>75.11</td>
</tr>
<tr>
<td>Patents</td>
<td>0.41</td>
<td>0.78</td>
<td>3.12</td>
<td>4.45</td>
<td>1.06</td>
<td>10.42</td>
<td>17.86</td>
</tr>
<tr>
<td>Twitter</td>
<td>15.41</td>
<td>17.98</td>
<td>57.00</td>
<td>-</td>
<td>27.92</td>
<td>367.32</td>
<td>442.85</td>
</tr>
</tbody>
</table>

Table 6: Runtime for 5 iterations of PageRank (in seconds) using 48 threads for all engines. “-” indicates the engine does not process over 70 million edges. EH denotes EmptyHeaded and the other engines include Galois (G), PowerGraph (PG), a commercial graph tool (CGT-X), Snap-Ringo (SR), SociaLite (SL), and LogicBlox (LB).
Experiments: Results

SSSP:

- Slower than Galois, still competitive against other baseline methods
- Require significantly fewer lines of code (2 vs. 172 for Galois)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>EH</th>
<th>G</th>
<th>PG</th>
<th>CGT-X</th>
<th>SL</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>0.024</td>
<td>0.008</td>
<td>0.22</td>
<td>0.51</td>
<td>0.27</td>
<td>41.81</td>
</tr>
<tr>
<td>Higgs</td>
<td>0.035</td>
<td>0.017</td>
<td>0.34</td>
<td>0.91</td>
<td>0.85</td>
<td>58.68</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>0.19</td>
<td>0.062</td>
<td>1.80</td>
<td>-</td>
<td>3.40</td>
<td>102.83</td>
</tr>
<tr>
<td>Orkut</td>
<td>0.24</td>
<td>0.079</td>
<td>2.30</td>
<td>-</td>
<td>7.33</td>
<td>215.25</td>
</tr>
<tr>
<td>Patents</td>
<td>0.15</td>
<td>0.054</td>
<td>1.40</td>
<td>4.70</td>
<td>3.97</td>
<td>159.12</td>
</tr>
<tr>
<td>Twitter</td>
<td>7.87</td>
<td>2.52</td>
<td>36.90</td>
<td>-</td>
<td>x</td>
<td>379.16</td>
</tr>
</tbody>
</table>

Table 7: SSSP runtime (in seconds) using 48 threads for all engines. “-” indicates the engine does not process over 70 million edges. EH denotes EmptyHeaded and the other engines include Galois (G), PowerGraph (PG), a commercial graph tool (CGT-X), and SociaLite (SL). “x” indicates the engine did not compute the query properly.
Experiments: Micro-Benchmarking

Setups:

- Run three different queries:
  - 4-clique ($K_4$)
  - Lollipop ($L_{3,1}$)
  - Barbell ($B_{3,1}$)

- Run COUNT(*) aggregate queries to test GHD

- Did not benchmark against low-level graph engines

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>EH</th>
<th>-R</th>
<th>-RA</th>
<th>-GHD</th>
<th>SL</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>$K_4$</td>
<td>4.12</td>
<td>10.01x</td>
<td>10.01x</td>
<td>-</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>3.11</td>
<td>1.05x</td>
<td>1.10x</td>
<td>8.93x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>3.17</td>
<td>1.05x</td>
<td>1.14x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Higgs</td>
<td>$K_4$</td>
<td>0.66</td>
<td>3.10x</td>
<td>10.69x</td>
<td>-</td>
<td>666x</td>
<td>50.88x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.93</td>
<td>1.97x</td>
<td>7.78x</td>
<td>1.28x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>0.95</td>
<td>2.53x</td>
<td>11.79x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>$K_4$</td>
<td>2.40</td>
<td>36.94x</td>
<td>183.15x</td>
<td>-</td>
<td>t/o</td>
<td>141.13x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>1.64</td>
<td>45.30x</td>
<td>176.14x</td>
<td>1.26x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>1.67</td>
<td>88.03x</td>
<td>344.90x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Orkut</td>
<td>$K_4$</td>
<td>7.65</td>
<td>8.09x</td>
<td>162.13x</td>
<td>-</td>
<td>t/o</td>
<td>49.76x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>8.79</td>
<td>2.52x</td>
<td>24.67x</td>
<td>1.09x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>8.87</td>
<td>3.99x</td>
<td>47.81x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Patents</td>
<td>$K_4$</td>
<td>0.25</td>
<td>328.77x</td>
<td>1021.77x</td>
<td>-</td>
<td>20.05x</td>
<td>21.77x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.46</td>
<td>104.42x</td>
<td>575.83x</td>
<td>0.99x</td>
<td>318x</td>
<td>62.23x</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>0.48</td>
<td>200.72x</td>
<td>1105.73x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
</tbody>
</table>

- t/o indicates the engine ran for over 30 minutes.
- -R is EH without layout optimizations.
- -RA is EH without both layout (density skew) and intersection algorithm (cardinality skew) optimizations.
- -GHD is EH without GHD optimizations (single-node GHD).
Experiments: Micro-Benchmarking

Observations:

- GHD optimizations help significantly
  - Faster than LogicBlox, which doesn't have GHD optimizations
- GHDs enable early aggregation, eliminating computation on datasets with high density skew
  - 8.93x speed up on Google+ vs. other datasets
- SIMD parallelism significantly improve EmptyHeaded's performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>EH</th>
<th>-R</th>
<th>-RA</th>
<th>-GHD</th>
<th>SL</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>$K_4$</td>
<td>4.12</td>
<td>10.01x</td>
<td>10.01x</td>
<td>-</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>3.11</td>
<td>1.05x</td>
<td>1.10x</td>
<td>8.93x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>3.17</td>
<td>1.05x</td>
<td>1.14x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Higgs</td>
<td>$K_4$</td>
<td>0.66</td>
<td>3.10x</td>
<td>10.69x</td>
<td>-</td>
<td>666x</td>
<td>50.88x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.93</td>
<td>1.97x</td>
<td>7.78x</td>
<td>1.28x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>0.95</td>
<td>2.53</td>
<td>11.79x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>$K_4$</td>
<td>2.40</td>
<td>36.94x</td>
<td>183.15x</td>
<td>-</td>
<td>t/o</td>
<td>141.13x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>1.64</td>
<td>45.30x</td>
<td>176.14x</td>
<td>1.26x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>1.67</td>
<td>88.03x</td>
<td>344.90x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Orkut</td>
<td>$K_4$</td>
<td>7.65</td>
<td>8.09x</td>
<td>162.13x</td>
<td>-</td>
<td>t/o</td>
<td>49.76x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>8.79</td>
<td>2.52x</td>
<td>24.67x</td>
<td>1.09x</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>8.87</td>
<td>3.99x</td>
<td>47.81x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
<tr>
<td>Patents</td>
<td>$K_4$</td>
<td>0.25</td>
<td>328.77x</td>
<td>1021.77x</td>
<td>-</td>
<td>20.05x</td>
<td>21.77x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.46</td>
<td>104.42x</td>
<td>575.83x</td>
<td>0.99x</td>
<td>318x</td>
<td>62.23x</td>
</tr>
<tr>
<td></td>
<td>$B_{3,1}$</td>
<td>0.48</td>
<td>200.72x</td>
<td>1105.73x</td>
<td>t/o</td>
<td>t/o</td>
<td>t/o</td>
</tr>
</tbody>
</table>

t/o indicates the engine ran for over 30 minutes.
-R is EH without layout optimizations.
-RA is EH without both layout (density skew) and intersection algorithm (cardinality skew) optimizations. -GHD is EH without GHD optimizations (single-node GHD).
Drawbacks

- Some of the concepts are not clearly defined in the paper.
- Did not compare against Ligra.

Questions

- EmptyHeaded applies the paradigm of relational algebra / databases to graph processing. Has anyone tried the inverse: treat traditional relational databases as graphs?
- Are there graph engines that treat these queries as mathematical programs instead of relational queries?