EmptyHeaded: A Relational Engine for Graph Processing

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

EMPTYHEADED

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, ...

V.S.

High-level Graph Engines

Supports tasks using query languages

Drawbacks:
1) Performance gap

Examples:
SociaLite, LogicBlox, Grail

Introduction

EmptyHeaded

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

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, ...

V.S.

High-level Graph Engines

Supports tasks using query languages

Drawbacks:
1) Performance gap

Examples:
SociaLite, LogicBlox, Grail

Preliminaries: Datalog

Facts: tuples in the database

Rules: queries

StudentID	GPA	Course
0	4.5	6
1	4.8	2

A tuple

StudentID	Dorm	Class
0	Ashdown	2019
1	SidPac	2020

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

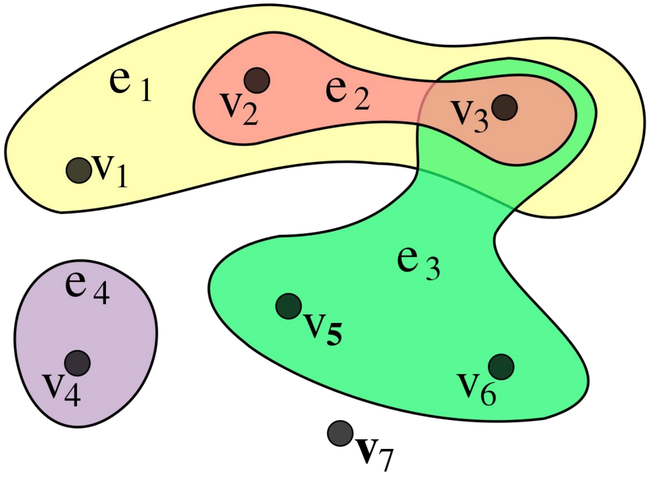
In general:

$$Q(x_1, x_2, \ldots, x_n) := R_1(\text{ args }), R_2(\text{ args }), \ldots$$

- 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
  - $Q(x_1, x_2, \ldots, x_n) := R_1(\text{ args }) \bowtie R_2(\text{ args }) \bowtie \ldots$

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)}



A hypergraph
(https://commons.wikimedia.org/wiki/File:Hypergraph-wikipedia.svg)

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:

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

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

Definition of the AGM bound:

$$|\mathrm{OUT}| \leq \prod_{e \in E} |R_e|^{x_e}$$

where R_e is the size of the relation represented by edge e

Preliminaries: Worst-Case Optimal Join

Example: triangle query

$$R(x,y) \bowtie S(y,z) \bowtie T(x,z)$$

Feasible cover: (1,1,0)

AGM bound: N^2

Another feasible cover: (½, ½, ½)

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:

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

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

Definition of the AGM bound:

$$|\mathrm{OUT}| \leq \prod_{e \in E} |R_e|^{x_e}$$

where R_e is the size of the relation represented by edge e

Preliminaries: Worst-Case Optimal Join

- 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

//*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 $\leftarrow \emptyset$ // *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**
    $Q_t \leftarrow$ Generic−Join($V - I$, E, $t :: t_v$)
    Q $\leftarrow Q \cup \{t_v\} \times Q_t$
  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

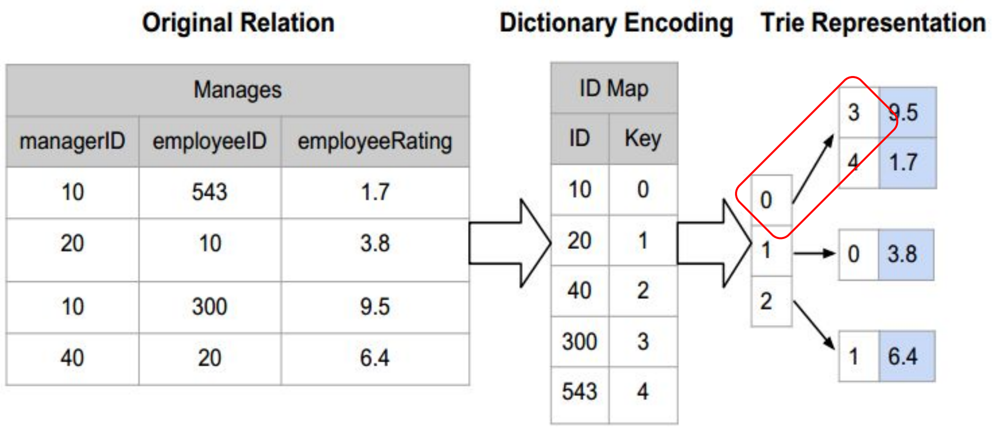
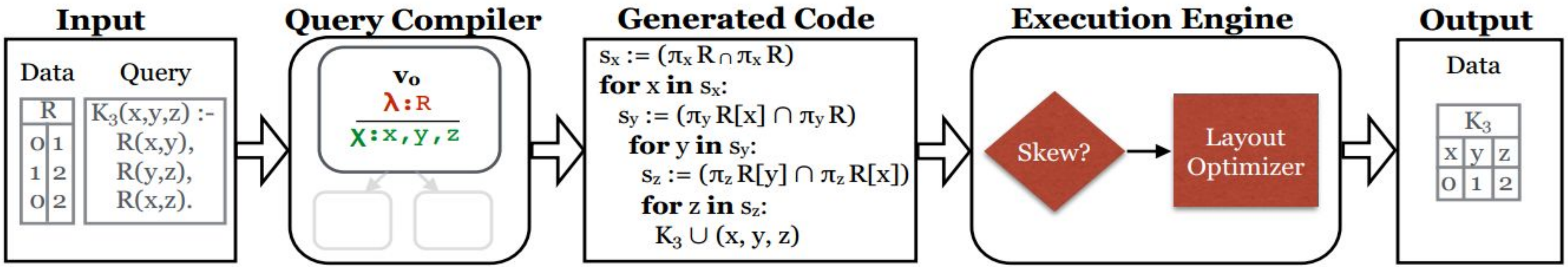


Figure 2: EmptyHeaded transformations from a table to trie representation using attribute order (*managerID,employerID*) and *employerID* attribute annotated with *employeeRating*.

EmptyHeaded: Overview



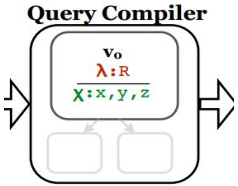
High-level Query

GHD

Generated C++ Code

Execution Engine with Optimization

Query Optimizer: Sample Queries



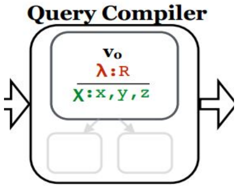
Name	Query Syntax
Triangle	`Triangle(x,y,z) :- R(x,y),S(y,z),T(x,z).`
4-Clique	`4Clique(x,y,z,w) :- R(x,y),S(y,z),T(x,z),U(x,w),V(y,w),Q(z,w).`
Lollipop	`Lollipop(x,y,z,w) :- R(x,y),S(y,z),T(x,z),U(x,w).`
Barbell	`Barbell(x,y,z,x',y',z') :- R(x,y),S(y,z),T(x,z),U(x,x'),R'(x',y'),S'(y',z'),T'(x',z').`
Count Triangle	`CountTriangle(;w:long) :- R(x,y),S(x,z),T(x,z); w=<<COUNT(*)>>.`

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 1: Example Queries in EmptyHeaded

Query Optimizer: Using Queries



Join:

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

Project:

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

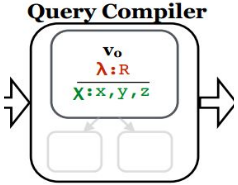
Selection:

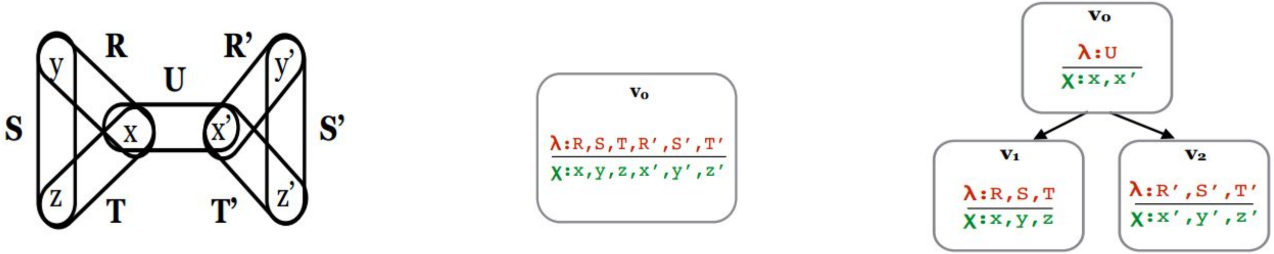
```
db.eval("""FliqueSel(a,b,c,d) :- x=0,
 Edge(a,b),Edge(b,c),Edge(a,c),
 Edge(a,d),Edge(b,d),Edge(c,d),Edge(a,x).""")
```

You can also use SQL:

```
db.eval("""
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)





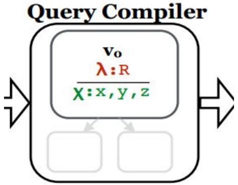
(a) Hypergraph (b) LogicBlox GHD (c) EmptyHeaded GHD

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 $\chi(v)$.

- 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

Query Optimizer: Generalized Hypertree Decomposition (GHD)



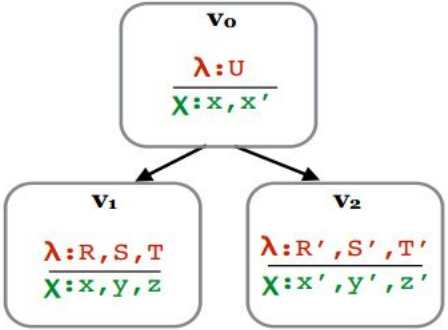
Formal definition:

Let H be a hypergraph. A generalized hypertree decomposition (GHD) of H is a triple D = (T; χ; λ), where:

T(V(T), E(T)) is a tree

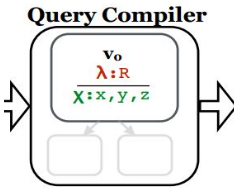
$\chi : V(T) \to 2^{V(H)}$ is a function associating a set of vertices $\chi(v) \subseteq V(H)$ to each node v of T;

$\lambda : V(T) \to 2^{E(H)}$ is a function associating a set of hyperedges to each vertex v of T;



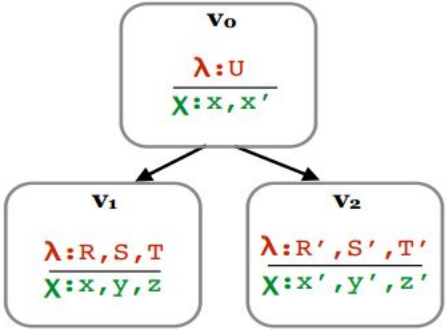
The following properties hold:

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 $v \in V(T), \chi(v) \subseteq \cup\lambda(v)$.



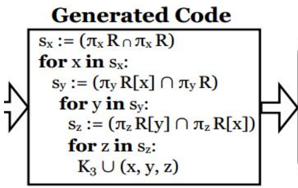
Query Optimizer: Compute outputs from GHDs

- Define Q_v as the query formed by joining the relations in $\lambda(v)$.
- Width w of a GHD:
  - AGM(Q_v)
- Given a GHD with width w, there is a simple algorithm to run in time $O(N^w + OUT)$.
  - 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 (OUT).



EmptyHeaded brute force all GHDs of all possible widths, because number of relations and attributes is typically small.

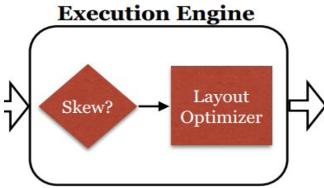
Query Optimizer: Code Generation from GHD



- 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

	Operation	Description
Trie (R)	$R[t]$	Returns the set matching tuple $t \in R$.
	$R \leftarrow R \cup t \times xs$	Appends elements in set xs to tuple $t \in R$.
Set (xs)	for x in xs	Iterates through the elements x of a set xs.
	$xs \cap ys$	Returns the intersection of sets xs and ys.

Table 2: Execution Engine Operations



Execution Engine Optimizer: Layouts

n	o_1	...	o_n	b_1	...	b_n

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 bitvector.

Two layouts

UINT (for sparse data)

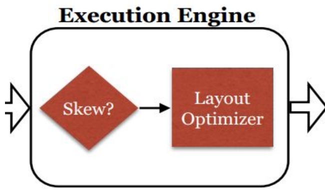
- Just an array of 32-bit unsigned integers

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.

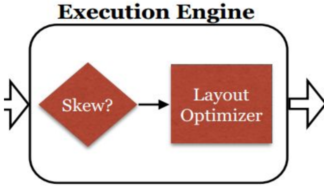
BITSET ∩ BITSET

- 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.

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.

Execution Engine Optimizer: Layout Selection Granularity



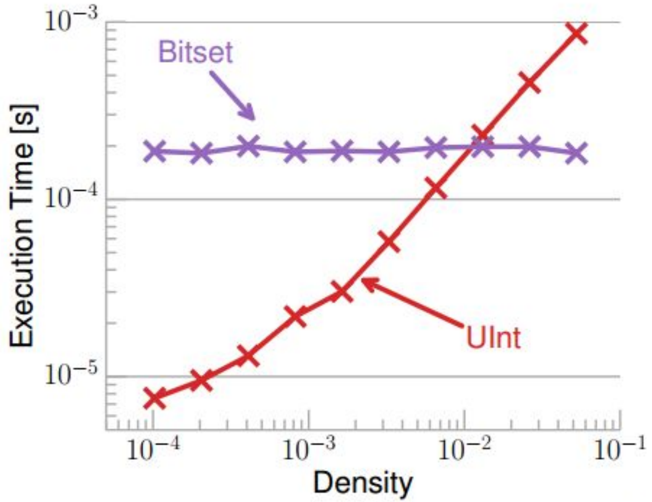


Figure 5: Intersection time of **uint** and **bitset** layouts for different densities.

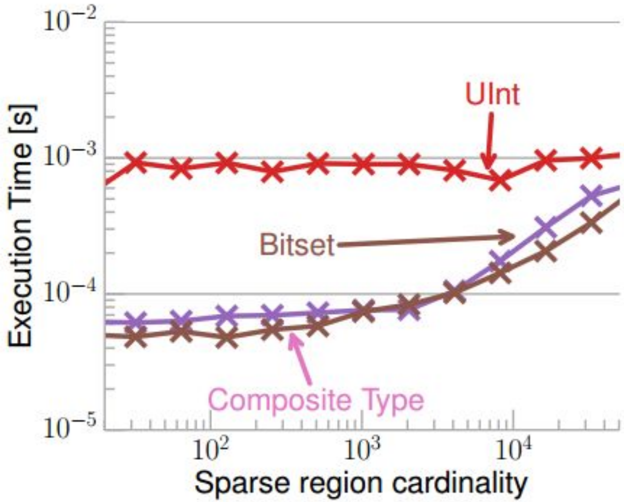
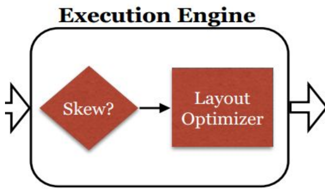


Figure 6: Intersection time of layouts for sets with different sizes of dense regions.

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

Dataset	Relation level	Set level	Block level
Google+	7.3x	1.1x	3.2x
Higgs	1.6x	1.4x	2.4x
LiveJournal	1.3x	1.4x	2.0x
Orkut	1.4x	1.4x	2.0x
Patents	1.2x	1.6x	1.9x

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.

Dataset	Nodes [M]	Dir. Edges [M]	Undir. Edges [M]	Density Skew	Description
Google+ [42]	0.11	13.7	12.2	1.17	User network
Higgs [42]	0.4	14.9	12.5	0.23	Tweets about Higgs Boson
LiveJournal [23]	4.8	68.5	43.4	0.09	User network
Orkut [4]	3.1	117.2	117.2	0.08	User network
Patents [1]	3.8	16.5	16.5	0.09	Citation network
Twitter [17]	41.7	1,468.4	757.8	0.12	Follower network

Table 3: Graph datasets presented in Section 5.1.1 that are used in the experiments.

Experiments: Results

Triangle Counting:

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

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

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

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

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)

		Low-Level			High-Level	
Dataset	EH	G	PG	CGT-X	SL	LB
Google+	0.024	**0.008**	0.22	0.51	0.27	41.81
Higgs	0.035	**0.017**	0.34	0.91	0.85	58.68
LiveJournal	0.19	**0.062**	1.80	-	3.40	102.83
Orkut	0.24	**0.079**	2.30	-	7.33	215.25
Patents	0.15	**0.054**	1.40	4.70	3.97	159.12
Twitter	7.87	**2.52**	36.90	-	x	379.16

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

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

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

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

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?