EmptyHeaded: A Relational Engine for Graph Processing

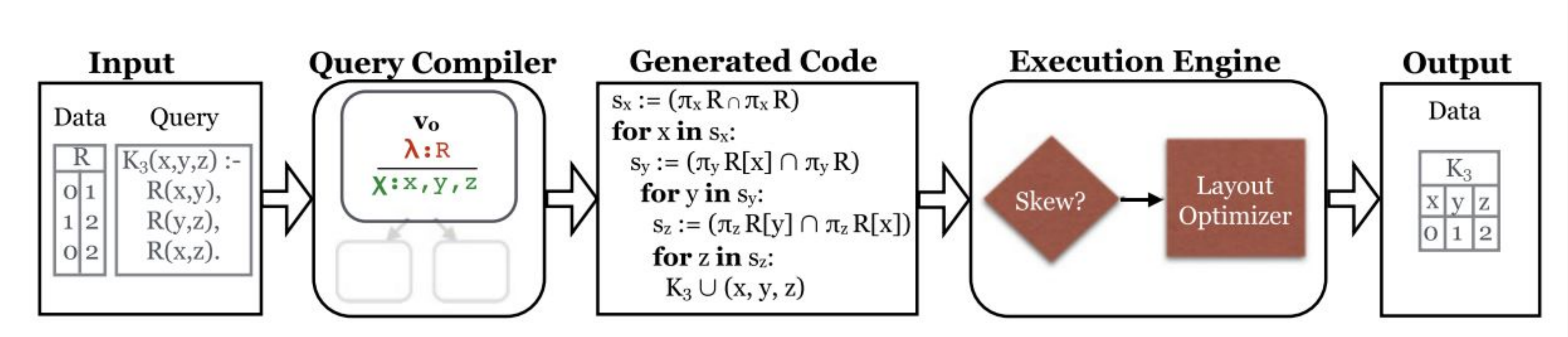
CHRISTOPHER R. ABERGER, ANDREW LAMB, SUSAN TU, ANDRES NÖTZLI, KUNLE OLUKOTUN, and CHRISTOPHER RÉ

PRESENTED BY SAI SAMEER PUSAPATY

Goals at a High Level

- Low-level graph engines
  - Fast performance (domain-spec primitives, optimized data layouts…)
  - Require users to write non-trivial code
- High-level graph engines
  - Slow performance
  - Easy to write queries
- We want the best of both worlds

System Overview



Terminology

- SIMD – Single Instruction Multiple Data (hardware that can apply same op on multiple data concurrently)
- GHD – Generalized Hypertree Decomposition
- Multiway Join - join multiple tables at same time
- Worst Case Optimal Join – optimal algorithm with worst case usage (output size of join)

Preliminaries

Worst-Case Optimal Join

- Hypergraph H = (V,E)
  - V = query attribute
  - E = relation
- Join queries can be represented as hypergraphs

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 ∩_{e∈E} R_e[t].
  Let I = {v_1} // the first attribute.
  Q ← ∅ // the return value
  // Intersect all relations that contain v_1
  // Only those tuples that agree with t.
  for every t_v ∈ ∩_{e∈E:e∋v_1} π_I(R_e[t]) do
    Q_t ← Generic-Join(V - I, E, t :: t_v)
    Q ← Q ∪ {t_v} × Q_t
  return Q
```

Feasible Cover/Bounding OUT Size

- AGM Paper creates a way to bound worst case size of join query
- Consider a hypergraph H(V,E), and vector x, which has a component for each edge such that each component is >= 0
- Feasible cover if

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

- If x is feasible, then

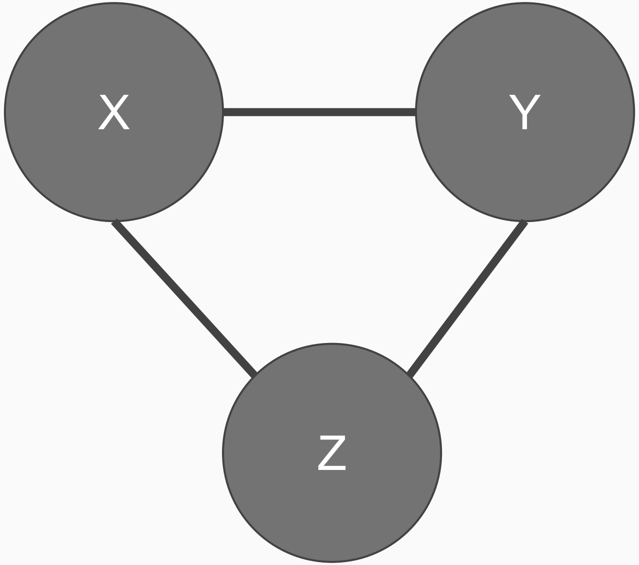
$$|\text{OUT}| \leq \prod_{e \in E} |R_e|^{x_e}.$$

Using AGM bound

X = <1,1,0>

By AGM we get...

O(N*N*1) = O(N^2)



X = <½,½,½>

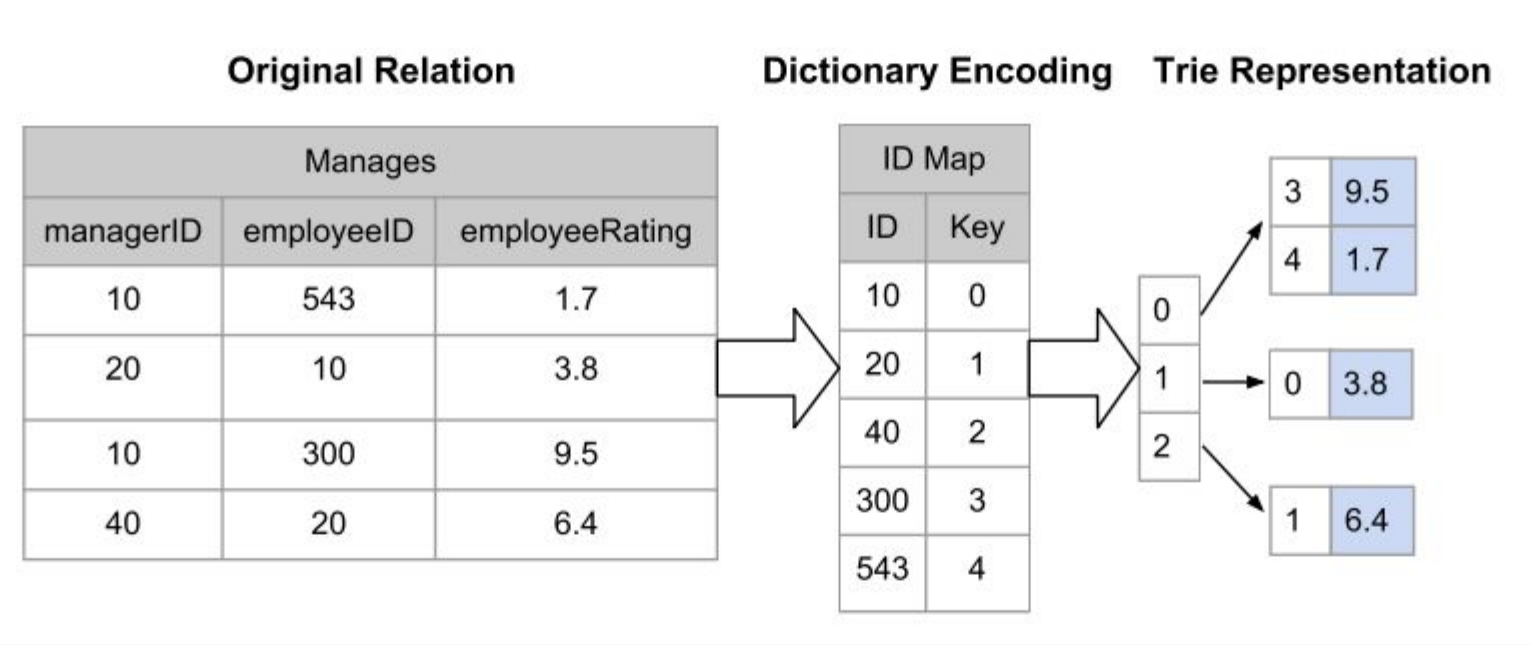
By AGM we get...

O(N^(3*.5)) = O(N^(3/2))

Turns out this is a tight bound if we consider a graph with sqrt(N) vertices

Input

Input Data Transformation



Query Language

- Aggregation (MIN, SUM, COUNT, matrix multiplication, etc…)
  - Annotations on trie
- Recursion
- Easy syntax for queries ->

Table 1. Example Graph Queries in EmptyHeaded

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	CntTriangle(;w:long):-R(x,y),S(x,z),T(x,z); w=<<COUNT(*)>>.
4-Clique-Selection	S4Clique(x,y,z,w):-R(x,y),S(y,z),T(x,z),U(x,w), V(y,w),Q(z,w),P(x,'node').
Barbell-Selection	SBarbell(x,y,z,x',y',z'):-R(x,y),S(y,z),T(x,z),U(x,'node'), V('node',x'),R'(x',y'),S'(y',z'),T'(x',z').
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.

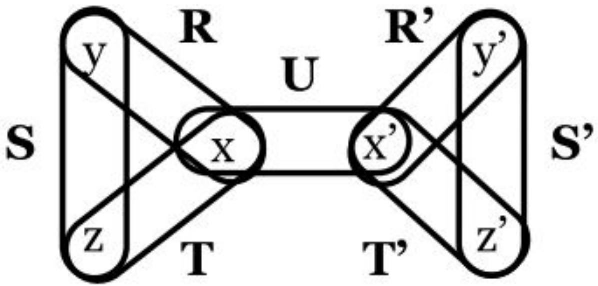
Query Compiler

Generalized HyperTree Decompositions

- Previously relational algebra used to represent query plans
- We now have multi-joins
  - Either extend relational algebra
  - Use GHDs so optimizations can be applied
- “A GHD is a tree similar to the abstract syntax tree of a relational algebra expression: nodes represent a join and projection operation, and edges indicate data dependencies. A node v in a GHD captures which attributes should be retained (projection with $\chi(v)$) and which relations should be joined (with $\lambda(v)$)”

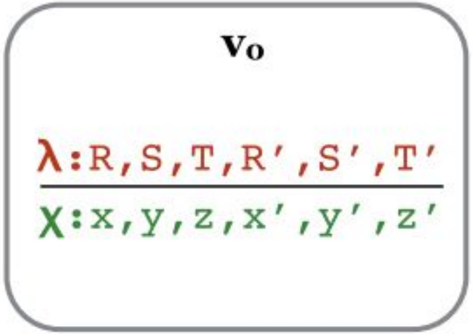
GHD Example

<½,½,½,0,½,½,½>



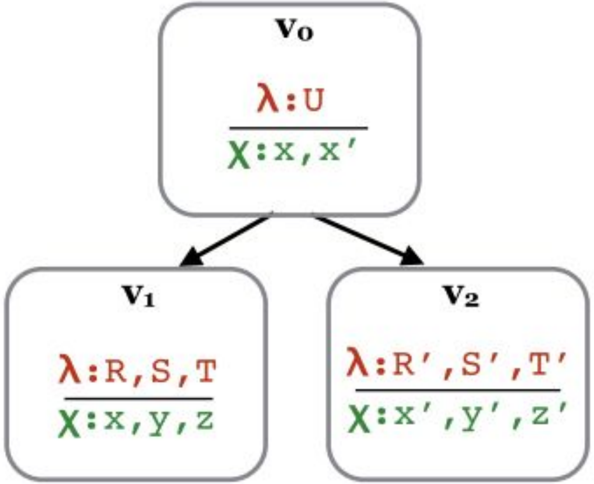
(a) Hypergraph

O(N^3)



(b) LogicBlox GHD

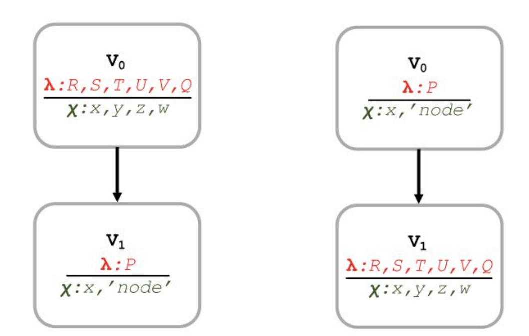
O(N^(3/2)+|OUT|)



(c) EmptyHeaded GHD

Pushing down Selections

- High selectivity operations should process ASAP
- Within a Node
  - Rearrange attribute order for WCOJ algorithm
  - Potential for early termination
- Across Nodes
  - Push low selectivity/low cardinality nodes down as far in GHD
- 4 orders of magnitude improvement in runtime

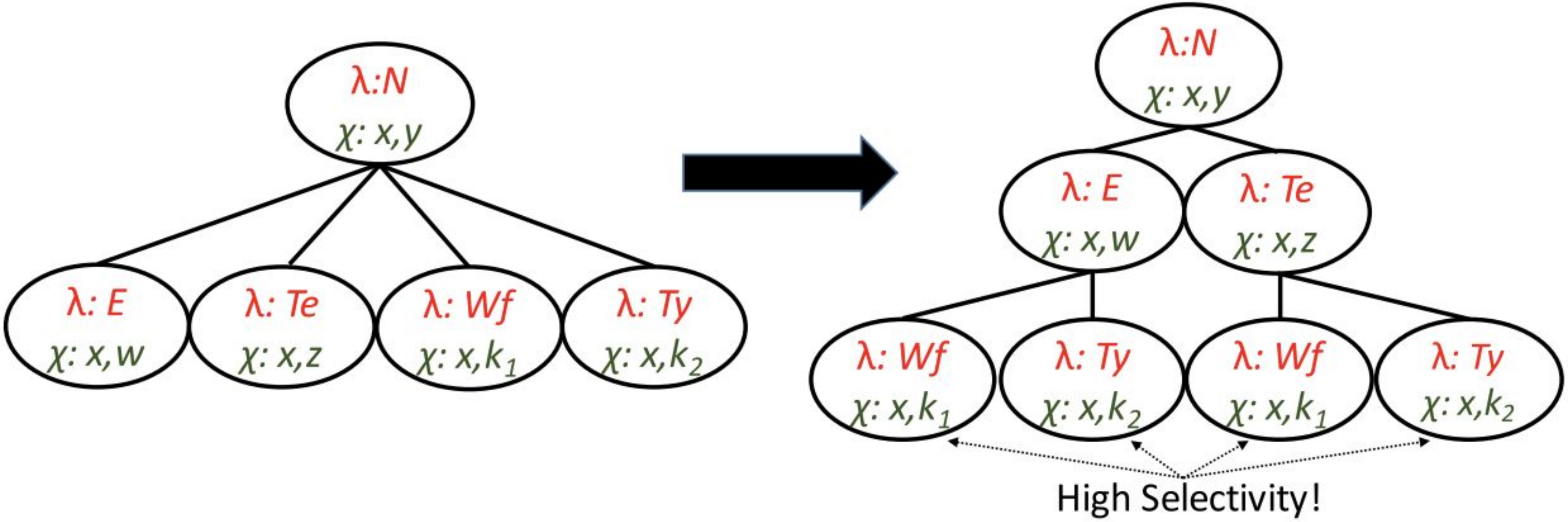


(a) GHD without pushing down (b) GHD with pushing down

Pushing Down Example

Q4	`out(x,y,z,w) :- worksFor(x,`Univ0Dept0'), name(x,y),emailAddress(x,w),telephone(x,z), type(x,`AssociateProfessor').`

Pushing Down Example



Code Generation

Generating Code

	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.

Generating Code

- Convert GHD into optimized C++ code
- standard API for trie traversals and set intersections
- EmptyHeaded provides optimized iterator interface for trie
  - Find which values match specific tuple predicate
- Within each node WCOJ algo is used as shown before
- Across Nodes
  - First a bottom up pass to compute Q and pass it to parent
  - Then top-down pass to build the result
- Recursion ends up just unrolling the join algorithm (GHD child points to parent)

Reducing Redundant Work

- Its possible to have two identical nodes
- Two nodes are equivalent if
  - They have same join patterns on same input
  - Same aggregations, selections, and projections
  - Result of each subtree is identical
- Extra work is removed during the bottom-up pass
  - List of previously computed GHD nodes is maintained
- Top-down pass can also be removed sometimes (COUNT query)

Execution Engine

Getting SIMD parallelism

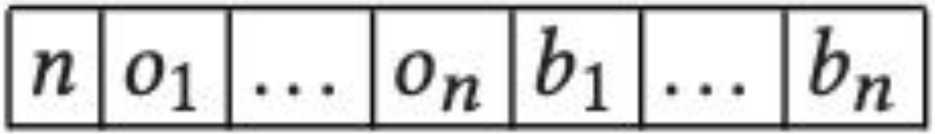
- Skews exist
  - Density of data vals is not constant
  - Cardinality of data vals is highly varied
- SIMD parallelism while dealing with these skews are achieved via data layouts and intersection algorithms

Layouts

- **uint** - (32 bit) great for representing sparse data (bad for SIMD parallelism)
- **bitset** - (bit vector) great for dense data and SIMD parallelism
- pshort - groups vals with common upper 16 bit prefix together (stores prefix once)
- varint - variable byte encoding for compression
- Bitpacked - partitions set into blocks and compresses each block

bitset

- Stores (offset, bit vector)
- Offset stores index of smallest val in bit vector
- Offsets are packed contiguously (allowing for uint layout)
  - Allows for easy intersection of offsets to find block match



pshort

- Exploits the fact that close by vals share common prefix
- Grouped by 16 bit prefix

$$S = \{65536, 65636, 65736\}$$

0 15	16 31	32 47	48 63	64 79
$v_1[31..16]$	length	$v_1[15..0]$	$v_2[15..0]$	$v_3[15..0]$
1	3	0	100	200

varint

- Variable byte encoding
  - Encode differences between data vals int bytes
  - Lower 7 bits store the data, 8th but indicates extension or not
  - If 8th bit is 0, output, otherwise append next byte

$$S = \{0, 2, 4\} \qquad Diff = \{0, 2, 2\}$$

0–31	32–38	39	40–46	47	48–54	55
$\lvert S\rvert$	$\delta_1[6..0]$	c	$\delta_2[6..0]$	c	$\delta_3[6..0]$	c
3	0	0	2	0	2	0

bitpacked

- Partitions set into blocks
- Blocks compressed
- Minds maximum bits of entropy for block, b
- Uses b bits to encode value
- Past work shows that encoding and decoding values happen efficiently at the granularity of SIMD registers

$$S = \{0, 2, 8\}, \qquad Diff = \{0, 2, 6\}$$

0 – 31	32 – 39	40 – 42	43 – 45	46 – 48
$\|S\|$	bits/elem	$\delta_1[2..0]$	$\delta_2[2..0]$	$\delta_3[2..0]$
3	3	0	2	6

Which layouts to use?

- Density Skew
  - Using uint and bitset layouts were enough
  - Varint and bitpacking are never the best
  - Pshort offers marginal benefits
  - Real world data has large amt of density skew

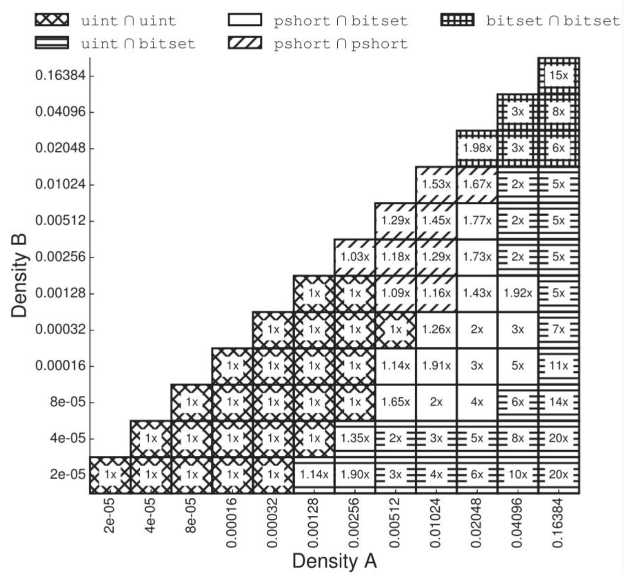


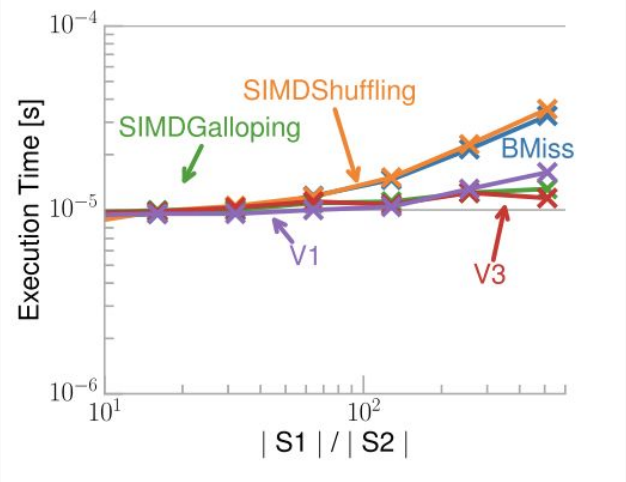
Fig. 7. Best performing layouts for set intersections with relative performance over uint.

Intersection Algorithms

- uint ∩ uint -
  - SIMDShuffling basic block-wise equality checking using SIMD shuffles and comparisons
  - V1 iterates through smaller set one by one and checks in larger set
  - V3 same as V1 but binary search on 4 blocks of data (each in a SIMD register)
  - SIMD Galloping uses exponential search on larger set to find potential match, then normal search
  - BMiss uses SIMD instructions to find partial matches then normal comps to check
- bitset ∩ bitset -
  - Common blocks found by intersection of offsets-> SIMD AND to intersect matching blocks
- Others described in paper

How well do the Intersection Algorithms work

- Cardinality Skew
  - SIMDGalloping and V3 algos do the best
  - Especially when size diff of two sets is very large
  - Use Shuffling until 1:32 threshold switch to Galloping



Node Orderings

- Maybe ordering of Nodes in Dictionary Encoding can make a big diff?
  - Can affect cardinality/density skew of data
- Try a variety of orderings: (Random, BFS, Strong_Runs, Degree)
- Turns out not really:
  - Effects of node ordering are mitigated by intersection and layout optimizations

Results

Experiment

- Dataset
  - Low Density Skew - LiveJournal, Orkut, Patents
  - Med Density Skew - Twitter, Higgs
  - High Density Skew - Google+
- Low Lvl Engines Considered: PowerGraph, CGT-X, Snap-R
- High Lvl Engines Considered: LogicBlox, SocialLite

Results

Ran Triangle Counting

Table 9. Triangle Counting Runtime (in Seconds) for EmptyHeaded and Relative Slowdown for Other Engines Including PowerGraph, a Commercial Graph Tool (CGT-X), Snap-Ringo, SociaLite, and LogicBlox

		Low-Level			High-Level	
Dataset	EmptyHeaded	PowerGraph	CGT-X	Snap-Ringo	SociaLite	LogicBlox
Google+	**0.31**	8.40×	62.19×	4.18×	1390.75×	83.74×
Higgs	**0.15**	3.25×	57.96×	5.84×	387.41×	29.13×
LiveJournal	**0.48**	5.17×	3.85×	10.72×	225.97×	23.53×
Orkut	**2.36**	2.94×	-	4.09×	191.84×	19.24×
Patents	**0.14**	10.20×	7.45×	22.14×	49.12×	27.82×
Twitter	**56.81**	4.40×	-	2.22×	t/o	30.60×

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.

Results

Ran with PageRank

Table 10. Runtime for Five Iterations of PageRank (in Seconds) Using 48 Threads for All Engines

		Low-Level				High-Level	
Dataset	EmptyHeaded	Galois	PowerGraph	CGT-X	Snap-Ringo	SociaLite	LogicBlox
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

Results

Ran for Single Source Shortest Path

Table 11. SSSP Runtime (in Seconds) Using 48 Threads for All Engines

		Low-Level			High-Level	
Dataset	EmptyHeaded	Galois	PowerGraph	CGT-X	SociaLite	LogicBlox
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

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

Results

Dataset	-SIMD	-Representation	-SIMD & Representation
Google+	1.0×	3.0×	7.5×
Higgs	1.5×	3.9×	4.8×
LiveJournal	1.6×	1.0×	1.6×
Orkut	1.8×	1.1×	2.0×
Patents	1.3×	0.9×	1.1×

"-SIMD" is EmptyHeaded without SIMD. "-Representation" is EmptyHeaded using `uint` at the graph level.

Conclusion

- First WCOJ processing engine that also...
  - Can compete with low level engines
  - Has simple high level querying
- Use GHDs (10^3x improvement)
- Use layouts to get SIMD parallelism
- Outperform other popular engines by 4-60x
- Extend to Resource Description Framework Engines
  - More complex join queries
  - Specialized
  - (Subject, Object, Predicate) triples form massive graph

Table 17. Runtime in Milliseconds for Best Performing System and Relative Runtime for Each Engine on the LUBM Benchmark with 133 Million Triples

Query	Best	EmptyHeaded	TripleBit	RDF-3X	MonetDB	LogicBlox
Q1	4.00	1.51×	3.45×	**1.00**×	174.58×	8.62×
Q2	973.95	**1.00**×	2.38×	1.92×	8.79×	1.52×
Q3	0.47	**1.00**×	92.61×	8.44×	283.37×	83.41×
Q4	3.39	4.62×	**1.00**×	1.77×	2093.78×	116.32×
Q5	0.44	**1.00**×	99.21×	9.15×	303.11×	81.44×
Q7	6.00	3.18×	8.53×	**1.00**×	573.33×	6.52×
Q8	78.50	9.83×	**1.00**×	3.07×	206.62×	5.03×
Q9	581.37	**1.00**×	3.53×	6.63×	24.29×	1.35×
Q11	0.45	**1.00**×	6.07×	11.03×	58.63×	73.76×
Q12	3.05	2.22×	**1.00**×	7.86×	118.94×	50.23×
Q13	0.87	**1.00**×	48.90×	35.49×	86.18×	102.77×
Q14	3.00	1.90×	54.47×	**1.00**×	313.47×	325.02×

Table 18. Relative Speedup of Each Optimization on Selected LUBM Queries with 133 Million Triples

Query	+Layout	+Attribute	+GHD	+Pipelining
Q1	2.10×	129.85×	-	-
Q2	8.22×	1.03×	-	-
Q4	2.02×	12.88×	69.94×	-
Q7	4.35×	95.01×	-	-
Q8	2.24×	1.99×	1.5×	4.67×
Q14	7.92×	234.49×	-	-

`+Layout` refers to EmptyHeaded when using multiple layouts versus solely an unsigned integer array (index layout). `+Attribute` refers to reordering attributes with selections within a GHD node. `+GHD` refers to pushing down selections across GHD nodes in our query plan. `+Pipelining` refers to pipelining intermediate results in a given query plan. "-" means the optimization has no impact on the query.

Discussion Questions

- Have there been any advancements or competitors to EmptyHeaded in its goal to balance low-level performance and high-level simplicity?

- What merits does extending relational algebra to multi-way joins have?