EmptyHeaded

Aaron Huang

Motivation

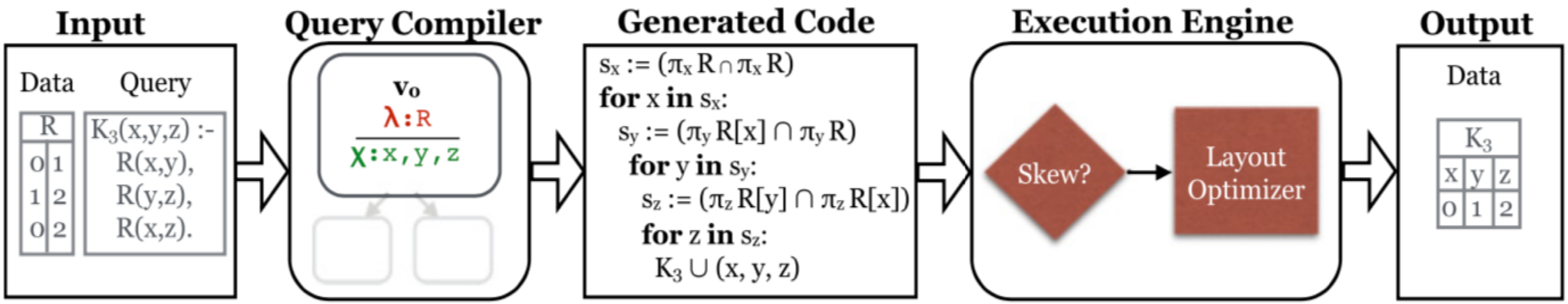
- High Level Engines – SQL like, easy to use
- Low Level Engines – Faster/optimized, harder to write/use

- EmptyHeaded – Create an engine with the simplicity of high level engine yet the speed of low level engine

Definitions

- SIMD – Single Instruction Multiple Data
- 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)

Overview


Input
Data
Query
R
0 1
1 2
0 2
K3(x,y,z) :-
R(x,y),
R(y,z),
R(x,z).
Query Compiler
v0
λ:R
χ:x,y,z
Generated Code
sx := (πx R ∩ πx R)
for x in sx:
sy := (πy R[x] ∩ πy R)
for y in sy:
sz := (πz R[y] ∩ πz R[x])
for z in sz:
K3 ∪ (x, y, z)
Execution Engine
Skew?
Layout Optimizer
Output
Data
K3
x y z
0 1 2

Preliminaries

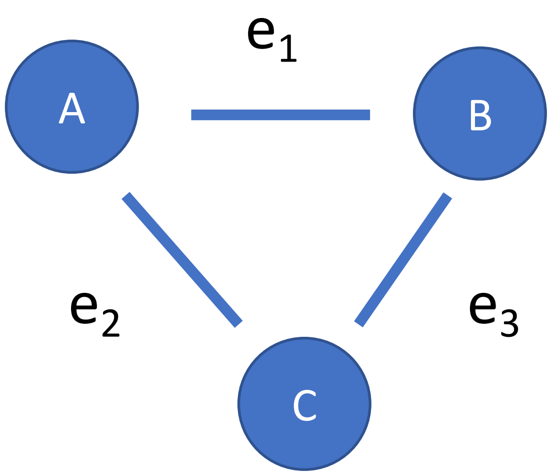
Compiler

Execution

Results

Worst Case Optimal Join – Fractional Cover

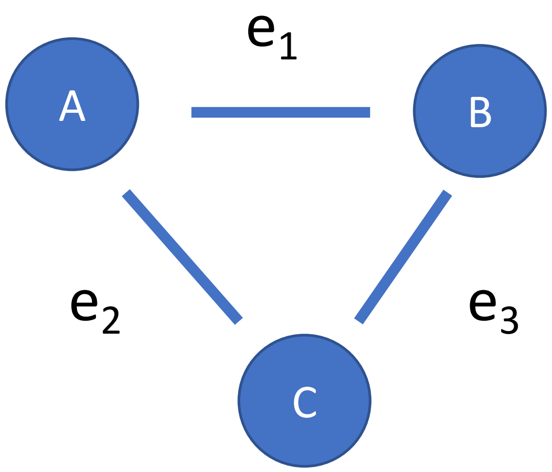
- Theoretical tight bound of worst case optimal join
- Hypergraph (V,E)
  - V – attribute of query
  - E – relation
- Define vector X with a component for each edge in the graph



$X = \langle e_1, e_2, e_3 \rangle$

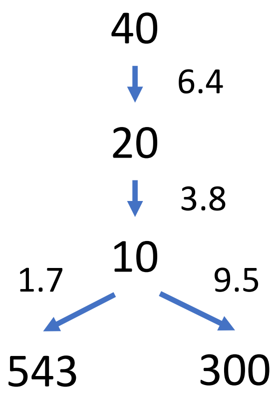
Feasible Cover

- Feasible Cover - Each vertex *v*, $\sum_{e \in E} X_e \geq 1$
- Upper bound , $OUT \leq \prod R_e^{x_e}$
- Triangle Query – A,B,C
  - (1,1,0) -> $O(N^2)$
  - (½, ½, ½) -> $O(N^{3/2})$



X = <e_1, e_2, e_3 >

Input Data



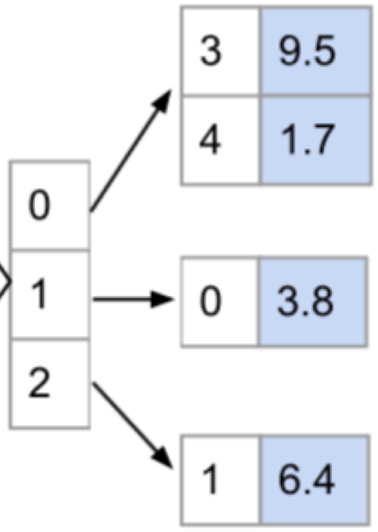
Original Relation

Manages		
managerID	employeeID	employeeRating
10	543	1.7
20	10	3.8
10	300	9.5
40	20	6.4

Dictionary Encoding

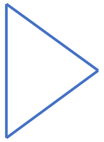
ID Map	
ID	Key
10	0
20	1
40	2
300	3
543	4

Trie Representation

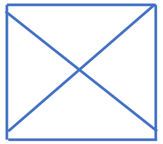


Example Queries

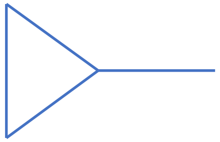
- Triangle



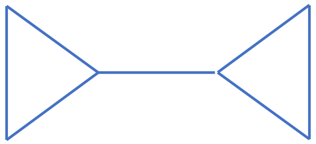
- 4 Clique



- Lollipop



- Barbell



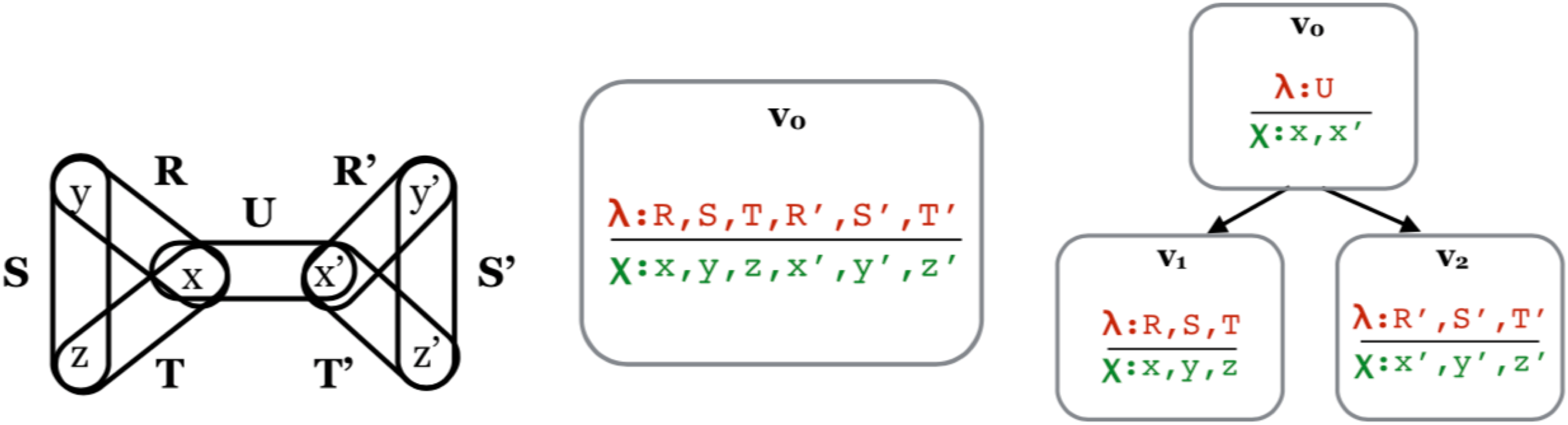
Preliminaries

Compiler

Execution

Results

GHD



(a) Hypergraph

(b) LogicBlox GHD

(c) EmptyHeaded GHD

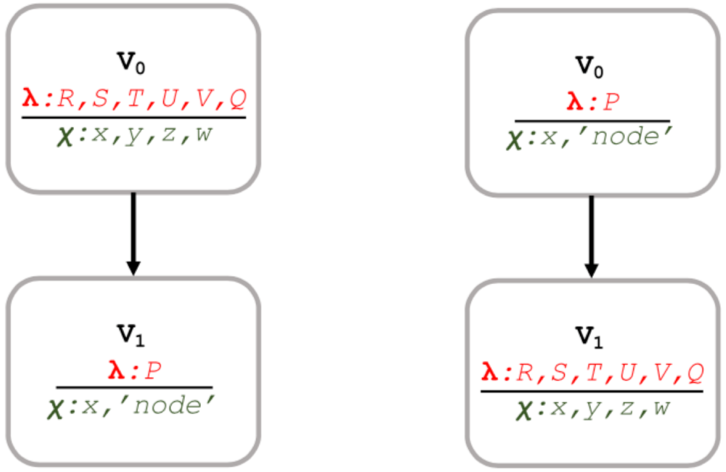
(½, ½, ½, 0, ½, ½, ½)

$O(N^3)$

$O(N^{3/2} + OUT)$

Push Down

- Within Node
  - Reorder attributes to allow early termination in trie (x,x') -> (x',x)
- Across Node
  - High selectivity nodes at bottom
  - Choose lowest width GHD's – fractional hypertree width
    - $O(N^{fhw})$
  - If A covers unselected attributes of B, add B as child of A
  - Maximize the depth (sum of heights) of fhw GHD trees



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

- Up to around 10^4 speedup

Redundancies

- 2 Nodes equivalent if
  - do the same join on same input
  - do same aggregation, selection, projection
  - have same subtree result

- 2x increase in Barbell Query

Preliminaries

Compiler

Execution

Results

Layouts

- Uint – efficient sparse data representation
- Bitset – good parallelism for dense data

- Pshort
- Varint
- Bitpacked

Bitset

- Set of pairs (offset, bit vector)
- Offset is index of the smallest value in the vector

- High parallelism
  - Intersect 2 bitsets
    - uint intersection of offsets to find potential block match
    - SIMD intersection of blocks
  - Instead of 4 element in SIMD reg, up to 256 elements

n	o_1	$\dots$	o_n	b_1	$\dots$	b_n

Pshort

- Values close to each other share prefix

- 3 values share prefix 0x10000
  - 96 bits vs 80 bits

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

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

- Value differences encoded
  - Bottom 7 bits: store data
  - 8th bit: data extends to next byte or not
- Good for dense, large data

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

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

Bitpacked

- Partition into blocks, compress each block
- Can compute differences in parallel SIMD
- Pack the difference into minimum block width

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

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

Density Skew

- Varint and Bitpacked decoding takes too long
- Pshort hard to convert/not compatible with other representation

- Relation (Graph) Level
  - Sparse – uint
- Set Level (Vertex)
  - Sparse – uint, dense – bitset
- Block Level (Blocks in set)
  - Sparse – uint, dense – bitset

Density Skew – Optimize

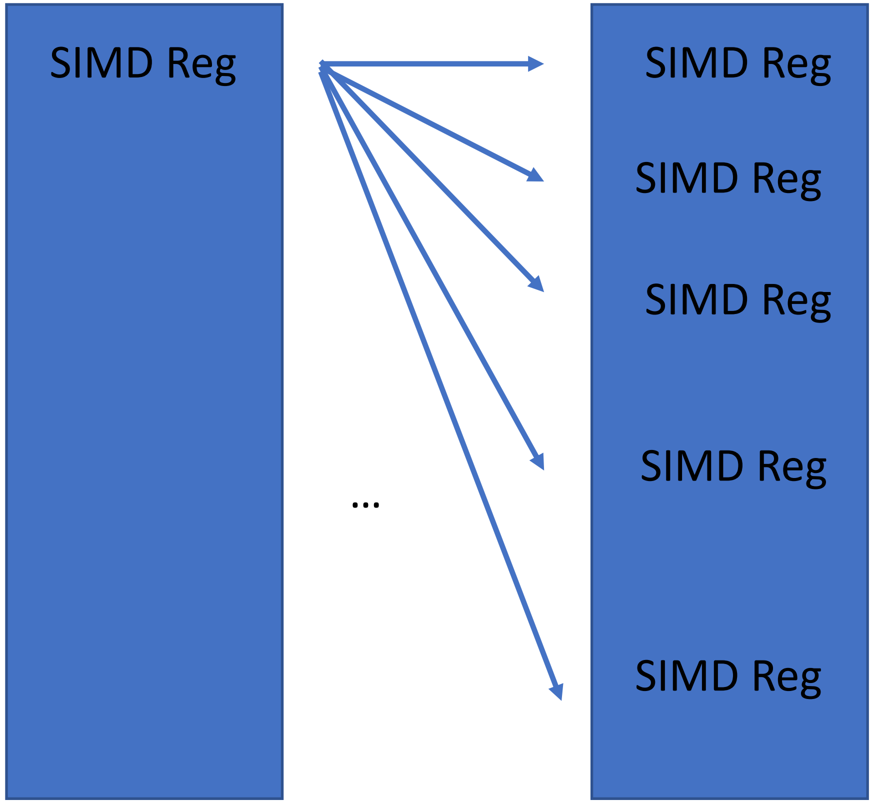
- Relation Level – doesn't optimize for density at all, 7.3x slower
- Set Level – at most 1.6x slower than optimal
- Block Level – at most 3.2x slower
    - Need to call more intersections and merge
    - 2.5x overhead

- Set Optimizer
    - Dense – Bitset if each value fits into SIMD register space
    - Sparse – Uint if values greater than that

Intersections – uint*uint

- SIMDShuffling – compare pairs of blocks in sets
- V1 – iterate through smaller set, SIMD comparison with larger
- V3 – V1 but do binary search on 4 blocks
- SIMDGalloping – V1 but do scalar binary search
- BMiss – SIMD to compare parts, then scalar comparison for full match

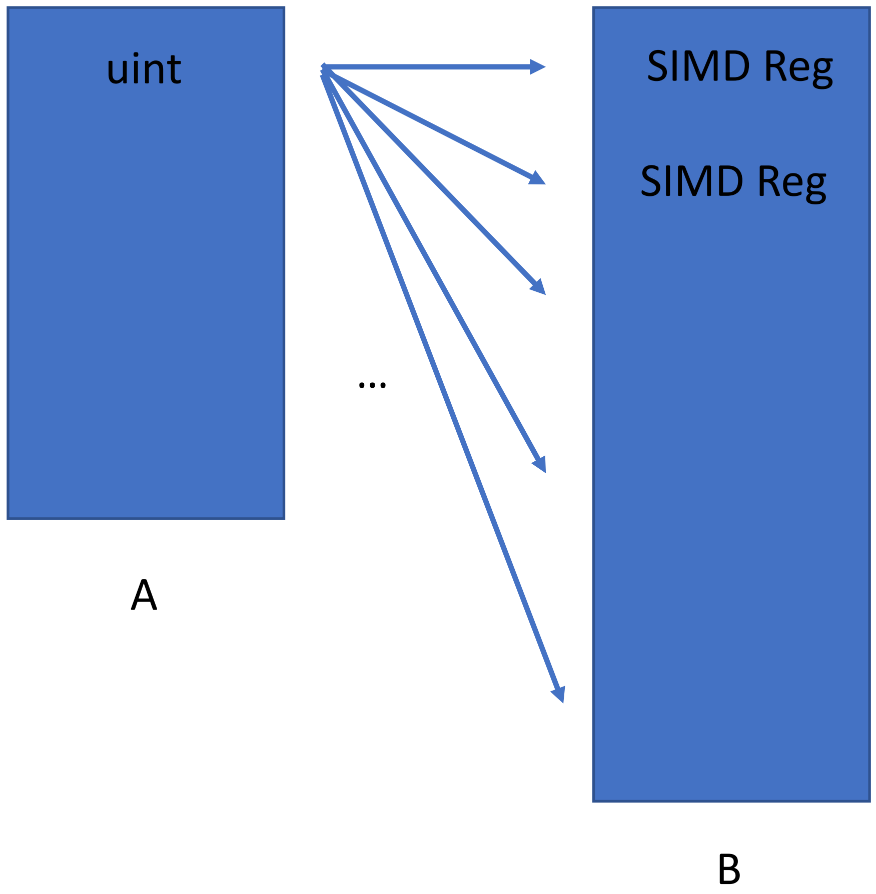
SIMD Shuffling



$O(N^2)$

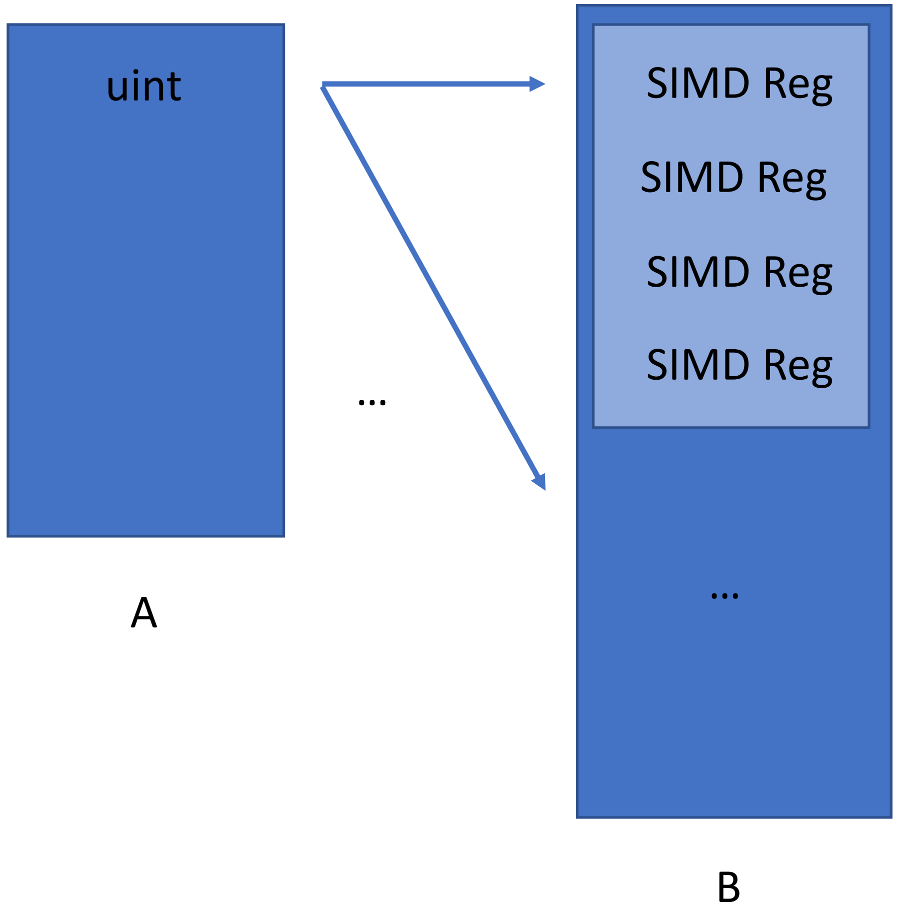
V1

- A,B sorted
- Uint a, Block b
- Find block where last element in b greater than a
- SIMD Comparison to find match



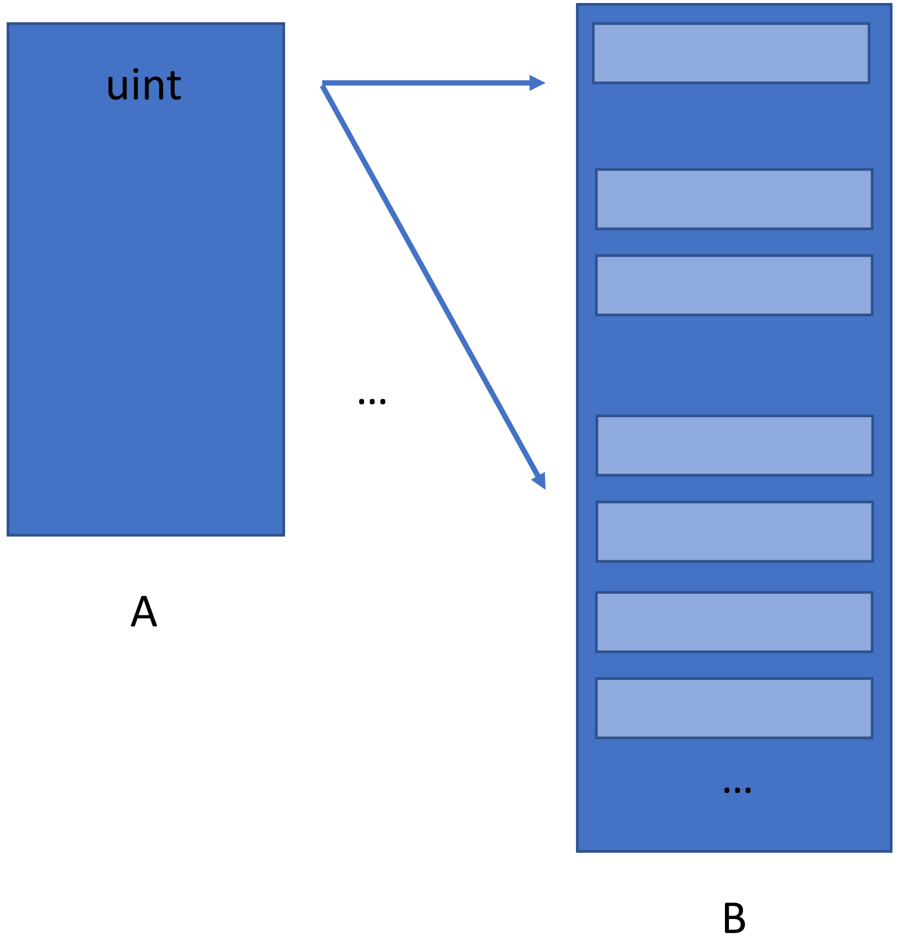
V3

- A,B sorted
- Uint a, Block b
- Find group of 4 blocks where last element greater than a
- Binary Search the 4 blocks
- SIMD Comparison to find match



SIMD Galloping

- A,B sorted
- Uint a, Block b
- Check block groups of exponential size (1,2,4,…)
- Binary search group
- SIMD Comparison



Cardinality Skew

- Set Cardinality difference – difference in size between sets
- Galloping algorithms work well when one set much smaller than the other
- Have inherent overhead over normal algorithms
- Use SIMDShuffling by default, and SIMDGalloping if cardinality ratio over 1:32

Node Ordering

- Random
- BFS
- Strong Runs – BFS starting at highest degree node
- Degree
- Rev-Degree
- Shingle – order by similar neighbors

- Selecting Intersection and Layout has greater effect, don’t care about ordering

Summary of EmptyHeaded Optimizations

- GHD Ordering
  - Attribute (within GHD node)
  - GHD (across GHD node)
- Layout (Dense vs. Sparse)
- Intersection (Shuffling vs. Galloping)

Preliminaries

Compiler

Execution

Results

Experiment

- Dataset
  - Low Density Skew – LiveJournal, Orkut, Patents
  - Medium Density – Twitter, Higgs
  - High Density – Google+
- Low-Level Engines – PowerGraph, CGT-X, Snap-R
- High Level Engines – LogicBlox, SocialLite

Results – Triangle Counting

		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×

Results – Optimizations

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×

Galois Results – PageRank, SSSP

- Galois
  - PageRank
    - Around 2-3x faster, 5x on Google+
    - 271 lines vs. EmptyHeaded 3
  - SSSP
    - 2-30x faster
    - 172 lines vs. EmptyHeaded 2

RDF

- Subject -> Predicate -> Object
- Extra Optimization - Pipelining
  - Since triples may share many common subject prefixes etc.
  - Can Pipeline GHD

Performance

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×

Optimizations

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×	-	-