EmptyHeaded: Worst-case Optimal Join Processing on Graphs

Chris Aberger, Andres Nötzli, Kunle Olukotun, and Christopher Ré
Stanford University
Specialized Graph Engines

- Green-Marl
- PowerGraph
- Graph-X
- Snap
- FlockDB
- StingerGraph
- Pregel
- Galois
- Ligra
- Neo4j
- Socialite
- Tao
- PGX
- Giraph

**EmptyHeaded** is not another graph engine!
Shock: For Joins and Graph Patterns, Relational Algebra is suboptimal, but Boolean Algebra can be optimal.

Part 1. Theory. Optimality Claim
- Worst-case optimal Join Processing. [PODS12,SIGMORDR14]

- Modern processors love Boolean Algebra
  - SIMD is the new Moore’s Law!
  - 8x over last 4 generations, more soon.
    - Linear (hardware) scaling.
- Challenge: Cope with Skew!
Joins Joins Joins!

\[ R(A, B) \text{ join } T(B, C) = \]

\[
\begin{array}{c|c}
A & B \\
0 & 1 \\
2 & 1 \\
\end{array}
\quad \begin{array}{c|c}
B & C \\
1 & 3 \\
1 & 4 \\
\end{array}
\]

Reminder: The worst case output size a join on two tables of size \( N \) is \( N^2 \).
Join Queries on Graphs

Data: \( R(A,B), S(A,C), \) and \( T(B,C) \)

Today: graph where edges colored \( R, S, T \).

Nodes are data values.

\[ Q_1 = \text{Join}(R, S) \]

"triples of nodes on a path of length 2 that goes via \( R \) then \( S \)"

\[ Q_2 = \text{Join}(R, S, T) \]

"triples of nodes that form \( R-S-T \) triangles"

Acyclic query!

Cyclic query!
Joins Since System R

\[ \text{Join}(R, S, T) = \{ (a, b, c) : (a, b) \text{ in } R, (a, c) \text{ in } S, (b, c) \text{ in } T \} \]

System R searches through \textbf{pairwise} joins.

For 40+ years, major commercial database use System-R style optimizer.
Nugget: “DBs have been asymptotically suboptimal for the last 4 decades…”
Data: R(A,B), S(A,C), and T(B,C)
Let Q be Join(R,S,T) = “R-S-T triangles”

If R, S, T contain \( \leq N \) tuples, how big can \(|Q|\) be?

|Join(R,S)| \(\leq N^2\) 

Correct asymptotic: \(|Q|\) in \(\Theta(N^{3/2})\)

Can we compute Q in time \(O(N^{3/2})\)?
Pairwise Joins are Suboptimal

Let $R(A,B), S(B,C), T(A,C)$ be relations.

- $R = [N] \times \{1\}$
- $S = \{1\} \times [N]$  
- $T = \{1\} \times [N]$

Data in $R$ and $S$

$|R| = |S| = |T| = N$

$[N] = \{1, \ldots, N\}$

$\text{JOIN}(R,S) = [N] \times \{1\} \times [N]$

$|\text{Join}(R,S)| = N^2$

DB is toast!

**Panic!** A simple modification: any pairwise join plan takes $\Omega(N^2)$.
Simple algorithm.

For each $x$ in $V$
For each $y$ in $E[x,-]$

$Q = Q \cup \{(x,y)\}$ \times \text{Intersect}(E[x,-],E[y,-])$

Intersect($X,Y$) **can be done** in time $\min\{|X|,|Y|\}$.

Running time can be bounded by...

$$\sum \sum \sum \min\{|E[x,-]|,|E[y,-]|\}|E[x,y]|$$

Let’s upper bound this by $|E|^{3/2}=N^{3/2}$
This proof is simple!

\[
\sum_{x \in V} \sum_{y \in V} \min\{|E[x, -]|, |E[y, -]|\} |E[x, y]|
\]

Use geometric mean inequality.

\[
\min\{|X|, |Y|\} \leq |X|^{1/2} |Y|^{1/2}
\]

\[
\leq \sum_{x \in V} \sum_{y \in V} |E[x, -]|^{1/2} |E[y, -]|^{1/2} |E[x, y]|
\]

Just rearrange terms...

\[
= \sum_{x \in V} |E[x, -]|^{1/2} \sum_{y \in V} |E[y, -]|^{1/2} |E[x, y]|
\]
Show no larger than $N^{3/2}$ in which $N = |E|$

\[\sum_{x \in V} |E[x, -]|^{1/2} \sum_{y \in V} |E[y, -]|^{1/2} |E[x, y]|\]

Using Cauchy-Schwarz

\[\sum_{i=1}^{n} x_i y_i \leq \sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{j=1}^{n} y_j^2}\]

\[\leq \sum_{x \in V} |E[x, -]|^{1/2} \sqrt{\sum_{y \in V} |E[y, -]|} \sqrt{\sum_{y \in V} |E[x, y]|}\]

Definitions of $E[x, -]$ and $|E| = N$

\[= \sqrt{N} \sum_{x \in V} |E[x, -]|^{1/2} |E[x, -]|^{1/2}\]
For each node $x$ in $V$
For each $y$ in $E[x,-]$
$Q = Q \cup \{(x,y)\} \times \text{Intersect}(E[x,-],E[y,-])$

Optimality: Complete graph on $N^{1/2}$ Nodes has $\sim N$ edges and $\sim N^{3/2}$ triangles
Remarkable Observation

For the Triangle query,

Boolean algebra+ “map” is faster than relational algebra (theory now, empirical next)

Optimality bound followed as long as “min property”:

\[
\text{Intersect}(X,Y) \text{ can be done in time } \min\{|X|, |Y|\}
\]

This is the key issue in each one of these algorithms Galloping in the 70s, or LeapFrog Trie Join (LogicBlox) in the 2010s.
How do we generalize to joins?
Fractional Hypergraph Covers

Given a hypergraph $H=(V,E)$ a **fractional edge cover** is $x : E \rightarrow \mathbb{R}$ such that $x \geq 0$ and for each $v$ in $V$ we have $\Sigma_{e : v \text{ in } e} x(e) \geq 1$

**Ex:** $R(A,B), S(B,C), T(A,C)$.

$x(R,S,T) = (1,0,1)$ ... or...

$x(R,S,T) = (0.5, 0.5, 0.5)$

We think of a **query** as hypergraph to cover.
Size bounds [GM05, AGM08]

Fix a query $Q=(V,E)$.
Let $\mathbf{N}$ be a tuple of $|E|$ positive integers.

Define $S(Q, \mathbf{N})$ be the maximum size of $Q$ subject to $|R_e| \leq N_e$

**Thm [Atserias, Grohe, Marx FOCS08]:** Given any hypergraph cover $x$ for $(V,E)$ then

$$S(Q, \mathbf{N}) \leq \prod_{e \in E} |R_e|^{x(e)}$$

Triangle: $|R| = |S| = |T| \leq N$, $x(R)=x(S)=x(T)=0.5 \Rightarrow N^{1.5}$
One more example.

\[ R(A,B,C), S(A,B,D), T(A,C,D), U(B,C,D) \]

\[ x(R) = x(S) = x(T) = x(U) = 1/3 \]

Output size is \( O(N^{4/3}) \). More joins needs to be done faster?!?

Known since Loomis-Whitney (1940s Geometers!)
AGM’s result.

Atserias, Grohe, and Marx (AGM) allow one to write a linear program that tightly bounds the output size of any join query.

Proof using Han/Shearer’s lemma (non constructive)

Compute the output in upper bound time?

We would call this worst-case optimal
We show AGM’s fractional cover inequality is equivalent to the Bollabás-Thomason inequality from geometry.

Algorithm: Only Map and Boolean Algebra. [Ngo, Ré, Rudra SIGMOD Record14]
One Example

\(R(A,B), S(B,C), T(A,C), U(C,D)\).

\(x(R,S,T,U) = (1,0,0,1)\) is optimal \(N^2\) bound

Can “factor the Ds” to get: \(N^{3/2} + \text{Output}\). (i.e., Generalized Hypertree Decomposition)

\(|\text{Output}|\) could be \(N^2\) but could be much smaller!

EmptyHeaded uses this optimization.
Relational algebra is suboptimal but Boolean algebra can be optimal.
- “min property” for intersections.

Generalized hyper-graph decompositions (GHDs) yield even better runtimes.
- A generalization of tree decompositions.
  - “Run NPRR on Bags, Yannakakis between them”
- With Afrati, Ullman, Joglekar, and Salihoglu recover some “optimal” parallelism results in multiround setting.
Part 2: The engine. It’s fast.
EmptyHeaded Engine

Datalog Query
triangle(x,y,x) :-
node(x,'engineer'),
node(y, 'leader'),
nodex(z, 'suspect '),
uedge(x, y, 'phone'),
uedge(y, z, 'email'),
uedge(x, z, 'text'),
x<y, y<z

for x in V :
ys := Edge[x, -]
for y in ys:
for z in ys ∩ Edge[y, -]:
triangle.insert(x, y, z)

EmptyHeaded Engine

Set Operations

EmptyHeaded Engine

Storage Engine (Choose Set Representation)
Graph Sparse?
Set Density Skew?
Block Density Skew?
Cardinality Skew?

Query Engine (Choose Boolean Operator)
Set Intersection Algorithm

Execution

Query Result

EmptyHeaded Engine

Graph
CSR or Dense Matrix
Bitset or U-Int
Composite Type

Set Operations

EmptyHeaded Engine

Query Compiler

EmptyHeaded Engine

Set Operations

EmptyHeaded Engine

Query Compiler
EmptyHeaded

EH uses the following operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge[x,-]</td>
<td>{ y</td>
</tr>
<tr>
<td>Edge[-,y]</td>
<td>{ x</td>
</tr>
<tr>
<td>for x in xs</td>
<td>Iterates through the elements x of set xs</td>
</tr>
<tr>
<td>Intersect(xs,ys)</td>
<td>The intersection of xs and ys</td>
</tr>
</tbody>
</table>

**Example Code**

```plaintext
For x in V
    For y in E[x,-]
        For z in Intersect(E[x,-],E[y,-])
            Out += (x,y,z)
```
SIMD, and the trend behind it.

Coping with skew is key challenge.

Performance #s. It’s fast.
## SIMD Processing

### Single Instruction Multiple Data (SIMD)

Multiple processing elements that perform the same operation on multiple data items *simultaneously*:

- Exploits **data-level** parallelism, not concurrency.
- Happens in one clock cycle (more or less).

---

#### Scalar Register Addition

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_2$</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R_1 + R_2$</td>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### SIMD Register Addition – width 4

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$R_2$</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>$R_1 + R_2$</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

Almost 4x faster!
Evolution of multi-core:
- Single Core
- Hyper Threading (2002)
- Dual Core (2006)
- Multi Core (2008)
- Many Core (2011)

Future processors: many cores, wider SIMD.

8x FLOPS over last 4 generations due to SIMD!

Image courtesy of Intel Corporation
Why is SIMD likely to be a trend?

Wider SIMD = Linear increase in area and power
Wider superscalar = Quadratic increase in a&p
Higher Frequency = Cubic increase in a&p

SIMD performance is linear in # transistors, and likely to continue with Moore’s law!
Representing Graphs
Graph Representation

“Compressed Sparse Row”
• Standard in-memory representation for graphs
• Each neighborhood is a set of dictionary encoded id’s

Dictionary
Encoded ID’s for each node

Neighbor sets could be Dense, sparse, or in between
Dictionary
Encoded ID’s for each node
EmptyHeaded

Design a Boolean algebra execution engine that exploits SIMD parallelism.

**Challenge**: cope with **skew** in data.

- **Cardinality Skew**
  - Skew in the sizes of the sets.
  - EH selects SIMD algorithms for set intersection

- **Density Skew**
  - Skew in the density of the sets.
    - *(subtle property of encoding)*
  - EH selects different representations

- **Block Skew**
  - Skew within a set!

Skew calculated using Pearson's first coefficient of skew:
$$3 \frac{\text{mean-mode}}{\text{stddev}}$$
Two types of Set Representation

Unsigned Integer (uint)
- Standard CSR representation
- Works well with sparse data
  - 4 comparisons per cycle using SSE intrinsic
    - Next processor generations even faster!

Bitvector
- Works well with dense data
  - Can provide compression or increase memory usage
  - 256 comparisons per cycle
    - Intersections to use wide AVX AND instructions
    - Next processor generation 2x faster

Full paper contains fancier types…
Unsigned Integer Intersections

Uint Intersection Algorithms

**Shuffling**: shuffle through arrays SIMD comparing blocks. **No min property**

**Galloping**: gallop over large gaps that do not contribute to the output. **Has Min property**

**Bmiss**: filter out unnecessary comparisons with a SIMD check.

To cope with cardinality skew, EH **dynamically** selects galloping when

$$\max\{ |S_1|/|S_2|, |S_2|/|S_1| \} > 32$$

Preserves min property.
Bitset

EH uses a variant of bitvectors called bitsets.

Stores a set of pairs \((\text{offset}, \text{bitvector})\)

- Offset is the index of the smallest value in the bitvector

**Block size:** number of bits in the bitvector

Example: \(S = \{0, 1, 2, 5\}\) with a block size of 2.
**Bitset Intersections**

- Use SIMD unsigned integer intersection to intersect offsets.
  - Produces blocks with potential matches

- Use an AVX AND instruction to intersect the bitvectors
  - Can compute the intersection of up to 256 elements in one cycle

**Diagram:**
- Bitset = \{0, 1, 2, 5\}
- Block size = 2
- Offsets: 0, 2, 4
- Bitvectors: 11, 10, 01

**Graph:**
- Execution Time [s] vs. Density
- Bitset
- U-Int
Introduce a composite type which pushes the decision of representation to a block (within each set) level.

- Each block represented using a bitset or unsigned integer representation.
- Each set contains a bitset with all the bitset blocks and a unsigned integer with the unsigned integer blocks.

Intersection: the cross product of the two representations intersected using previously described techniques, add a merge for the unsigned integer outputs.
**Representation Decisions**

At what granularity do we make the representation decision?

- **Graph Level** *Edge set Density is Skewed*
  - All sets in graph stored using the same representation
  - No Overhead!

- **Set Level** *Set Densities are Skewed*
  - Each set stored using a fixed representation
  - Overhead to check set type

- **Block Level** *Block Densities are Skewed*
  - Each block uses a different representation.
  - Overhead of merging results of blocks within sets
Scene Missing

See paper for many more experiments and layouts...
High-level Experiments
## Triangle Counting

Skew calculated using Pearson's first coefficient of skew: \(\frac{3(\text{mean}-\text{mode})}{\text{stddev}}\)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>0.11</td>
<td>12.2</td>
<td>1.17</td>
<td>1.17</td>
</tr>
<tr>
<td>Higgs</td>
<td>0.4</td>
<td>12.5</td>
<td>0.23</td>
<td>0.46</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4.8</td>
<td>43.4</td>
<td>0.09</td>
<td>0.97</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.1</td>
<td>117.2</td>
<td>0.08</td>
<td>1.46</td>
</tr>
<tr>
<td>Patents</td>
<td>3.8</td>
<td>16.5</td>
<td>0.09</td>
<td>2.22</td>
</tr>
<tr>
<td>Twitter</td>
<td>41.7</td>
<td>757.8</td>
<td>0.12</td>
<td>0.07</td>
</tr>
</tbody>
</table>

**Recall:**  
- **Cardinality Skew** handled by Set Algorithms,  
- **Density Skew** handled by Representations  
  - Google+ Highly Skewed in density & cardinality.  
  - Patents no Density Skew, High Cardinality Skew.
Triangle Counting

<table>
<thead>
<tr>
<th></th>
<th>-SIMD</th>
<th>-Rep</th>
<th>-Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>1.0x</td>
<td>3.0x</td>
<td>7.5x</td>
</tr>
<tr>
<td>Higgs</td>
<td>1.5x</td>
<td>3.9x</td>
<td>4.8x</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>1.6x</td>
<td>1.0x</td>
<td>1.6x</td>
</tr>
<tr>
<td>Orkut</td>
<td>1.8x</td>
<td>1.1x</td>
<td>2.0x</td>
</tr>
<tr>
<td>Patents</td>
<td>1.3x</td>
<td>0.9x</td>
<td>1.1x</td>
</tr>
</tbody>
</table>

SIMD and representation optimizations

High Skew

Low Skew
## Triangle Counting

<table>
<thead>
<tr>
<th></th>
<th>EmptyHeaded</th>
<th>PowerGraph</th>
<th>Snap-R</th>
<th>LogicBlox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>0.055s</td>
<td>55x</td>
<td>23.7x</td>
<td>862.7x</td>
</tr>
<tr>
<td>Higgs</td>
<td>0.057s</td>
<td>8.9x</td>
<td>15.4x</td>
<td>126.1x</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>0.24s</td>
<td>10.8x</td>
<td>21.6x</td>
<td>92.4x</td>
</tr>
<tr>
<td>Orkut</td>
<td>1.36s</td>
<td>5.5x</td>
<td>7.1x</td>
<td>46.4x</td>
</tr>
<tr>
<td>Patents</td>
<td>0.048s</td>
<td>30.0x</td>
<td>63.8x</td>
<td>88.7x</td>
</tr>
<tr>
<td>Twitter</td>
<td>24.67s</td>
<td>11.0x</td>
<td>5.1x</td>
<td>129.2x</td>
</tr>
</tbody>
</table>

48 used threads for each system.

Against highly tuned systems, EH approach is faster.
# 4-clique and Lollipops!

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Graph Type</th>
<th>EmptyHeaded</th>
<th>-SR</th>
<th>-Y</th>
<th>LogicBlox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>$K_4$</td>
<td>1.66s</td>
<td>19.1x</td>
<td>-</td>
<td>t/o</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>6.03s</td>
<td>3.5x</td>
<td>653.9x</td>
<td>t/o</td>
</tr>
<tr>
<td>Higgs</td>
<td>$K_4$</td>
<td>0.25s</td>
<td>1.7x</td>
<td>-</td>
<td>230.4x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.89s</td>
<td>19.7x</td>
<td>834.9x</td>
<td>t/o</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>$K_4$</td>
<td>1.04s</td>
<td>1.5x</td>
<td>-</td>
<td>1276.4x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>1.76s</td>
<td>1.3x</td>
<td>79.9x</td>
<td>t/o</td>
</tr>
<tr>
<td>Orkut</td>
<td>$K_4$</td>
<td>3.40s</td>
<td>1.4x</td>
<td>-</td>
<td>299.9x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>8.88s</td>
<td>1.5x</td>
<td>216.9x</td>
<td>t/o</td>
</tr>
<tr>
<td>Patents</td>
<td>$K_4$</td>
<td>0.10s</td>
<td>0.8x</td>
<td>-</td>
<td>37.2x</td>
</tr>
<tr>
<td></td>
<td>$L_{3,1}$</td>
<td>0.31s</td>
<td>0.9x</td>
<td>1.8x</td>
<td>52.5x</td>
</tr>
</tbody>
</table>
## Similarity Queries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>EmptyHeaded</th>
<th>-SR</th>
<th>LogicBlox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>CN</td>
<td>0.01s</td>
<td>6.1x</td>
<td>851.9x</td>
</tr>
<tr>
<td></td>
<td>Symb</td>
<td>4e-5s</td>
<td>0.9x</td>
<td>6.6e4x</td>
</tr>
<tr>
<td>Higgs</td>
<td>CN</td>
<td>0.01s</td>
<td>44.6x</td>
<td>255x</td>
</tr>
<tr>
<td></td>
<td>Symb</td>
<td>0.01s</td>
<td>1.0x</td>
<td>834.7x</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>CN</td>
<td>0.03s</td>
<td>39.7x</td>
<td>37.9x</td>
</tr>
<tr>
<td></td>
<td>Symb</td>
<td>0.03s</td>
<td>0.9x</td>
<td>604.0x</td>
</tr>
<tr>
<td>Orkut</td>
<td>CN</td>
<td>0.04s</td>
<td>51.1x</td>
<td>141.4x</td>
</tr>
<tr>
<td></td>
<td>Symb</td>
<td>0.03s</td>
<td>0.9x</td>
<td>742.1x</td>
</tr>
<tr>
<td>Patents</td>
<td>CN</td>
<td>0.04s</td>
<td>1.6x</td>
<td>10.0x</td>
</tr>
<tr>
<td></td>
<td>Symb</td>
<td>0.01s</td>
<td>0.9x</td>
<td>291.6x</td>
</tr>
</tbody>
</table>
More Complex Queries

- On K4 or Lollipop, only LogicBlox runs between 37x to 1276x+ improvement.
  - Part of this due to the GHD trick I mentioned
    - 80x-850x in the cases when it helps.
    - Up to 20x due to SIMD across datasets.

- Also ran BFS and some other workloads
  - Single threaded performance beats all systems
  - Ligra scales better on BFS
  - “Similarity” workloads also much faster.
Conclusion

**Shock:** For Joins and Graph Patterns, **Relational Algebra** is suboptimal, but **Boolean Algebra** can be optimal.

**Part 1. Theory. Optimality Claim**
- Simplified worst-case optimal compilation

**Part 2. Hardware. Optimize Boolean Algebra [Arxiv15]**
- Modern processors **love** Boolean Algebra
  - SIMD is the new Moore’s Law!
- Challenge: Cope with **Skew**!
To Infinity and Beyond

The Team

- Adam Perelman + Susan Tu
  - DunceCap Query Compiler
  - REPL

- Rohan Puttagunta
  - Incremental algorithms
  - Worst-case theory and beyond!

Future Work

Is everything a join? Taking advantage of pockets of density certainly extends beyond joins.

- Applications - Matrix Multiply, Inference, Fast-Fourier transform
- Distributed computation - "If your system doesn’t perform well on one billion things, you don’t yet have the credibility to claim it’s going to work much better on a trillion things."
One System to Rule Them All?

Operators
- Map
- Reduce
- Filter
- GroupBy
- Intersect
- Union
- Difference
Lets collaborate!
## Triangle Counting

<table>
<thead>
<tr>
<th>Service</th>
<th>Graph 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>

Relative time of the EmptyHeaded optimizers compared to an oracle. **Smaller is better.**
## Triangle Counting

<table>
<thead>
<tr>
<th></th>
<th>Set Optimizer</th>
<th>Block Optimizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google+</td>
<td>4%</td>
<td>5%</td>
</tr>
<tr>
<td>Higgs</td>
<td>1%</td>
<td>6%</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4%</td>
<td>12%</td>
</tr>
<tr>
<td>Orkut</td>
<td>3%</td>
<td>8%</td>
</tr>
<tr>
<td>Patents</td>
<td>10%</td>
<td>24%</td>
</tr>
</tbody>
</table>

Set and block level optimizer overheads.
Graph Representation

- Logically regard each neighborhood as sets.
- Not fundamentally different from Compressed Sparse Row but now an abstraction is in place to choose representations at will.
  - $S_0$ is sparse...storing the 32-bit ID probably works OK
  - $S_5$ is dense...maybe we can do better than storing the 32-bit ID's here?
Whole graph encoded using unsigned integer or bitset representations

\{S_0, S_1, S_2, S_3, S_4, S_5\} = unsigned integers (CSR)

or

\{S_0, S_1, S_2, S_3, S_4, S_5\} = bitsets
Each set encoded using unsigned integer or bitset representations

\{S_0, S_2, S_3, S_4\} = \text{unsigned integers (CSR)}

\{S_1, S_5\} = \text{bitsets}
Shift the unsigned integer values right by \( \log_2(\text{block size}) \)
- Restrict ourselves to block sizes which are powers of 2

Intersect offsets with shifted values, keeping track of the positions in both sets that match
- e.g. yields \{0,2,4\}

Probe the bitvector blocks yielded from the intersection with the corresponding unsigned integer values that matched
- e.g. probe blocks with offsets of \{0,2,4\} with values of \{1,3,5\}

Output result in a unsigned integer representation
- e.g. result = \{1,5\}

Bitset = \{0,1,2,5\}
- block size = 2

<table>
<thead>
<tr>
<th>offsets</th>
<th>bitvectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 4 11 10 01</td>
<td></td>
</tr>
</tbody>
</table>

Unsigned Integer = \{0,1,3,5\}

| 1 3 5 7 |