EmptyHeaded: A Relational Engine for Graph Processing

Jason Priest
2018-02-28
6.886

Key Contributions

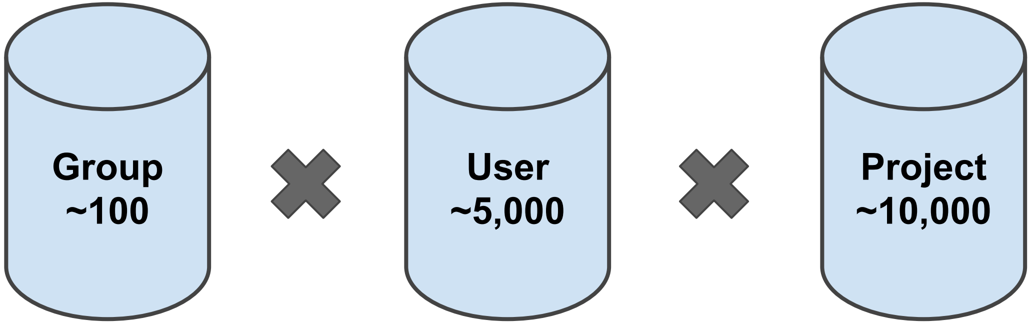
- Join optimization based on *generalized hypertree decomposition*
- Data and algorithm optimization based on local graph skew

Traditional Relational Query

How many github organizations (groups) have a C++ developer who registered this year?

```
select count(*)
    from group
    join user     on   group.users contains user.id
    join project on project.owner is       user.id
    where user.joined  after “2018-01-01”
    and   project.lang is    “C++”;
```

Join Order Matters



Cyclic Queries

$$Q_{\triangle} = R(A,B) \bowtie S(B,C) \bowtie T(A,C).$$

- Joins usually implemented pairwise, or between two sets at a time
- For cyclic queries such as above, this leads to suboptimal behavior

Pairwise Joins Insufficient

$$R = \{a_0\} \times \{b_0, \ldots, b_m\} \cup \{a_0, \ldots, a_m\} \times \{b_0\}$$

$$S = \{b_0\} \times \{c_0, \ldots, c_m\} \cup \{b_0, \ldots, b_m\} \times \{c_0\}$$

$$T = \{a_0\} \times \{c_0, \ldots, c_m\} \cup \{a_0, \ldots, a_m\} \times \{c_0\}$$

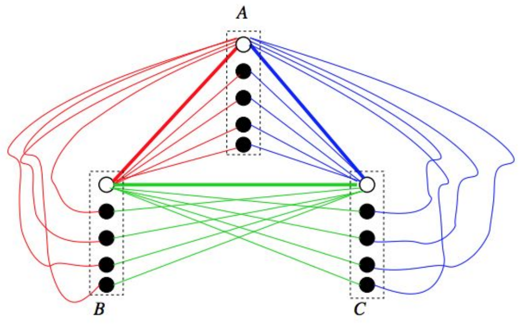
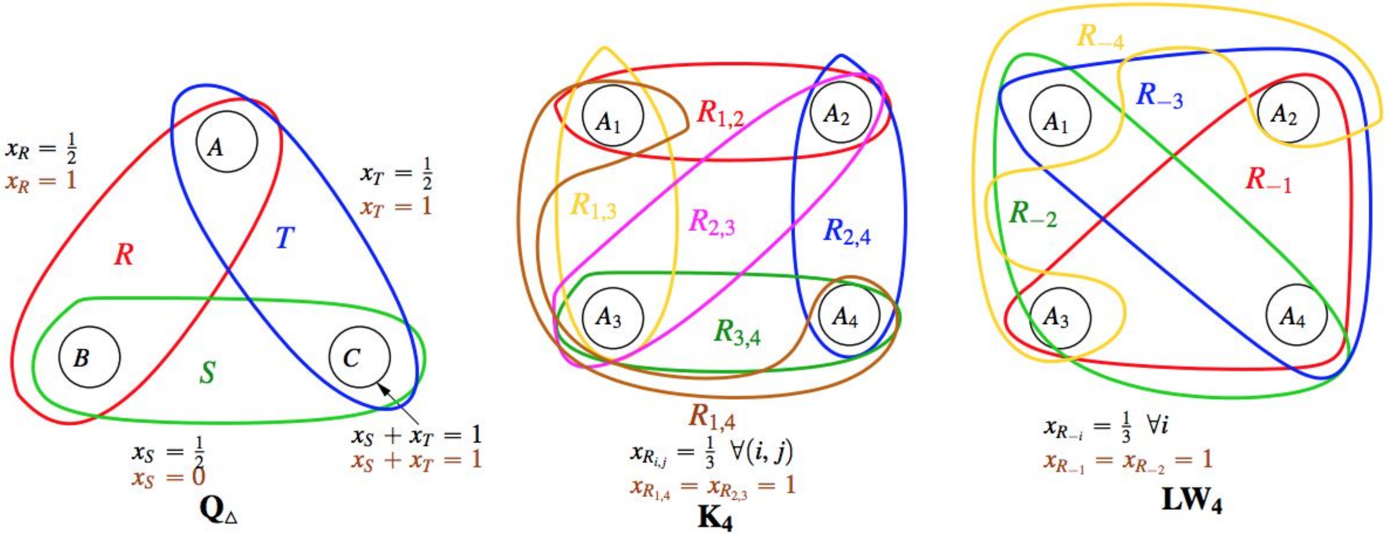


Figure 2: Counter-example for join-project only plans for the triangles (left) and an illustration for $m = 4$ (right). The pairs connected by the red/green/blue edges form the tuples in the relations $R/S/T$ respectively. Note that the in this case each relation has $N = 2m + 1 = 9$ tuples and there are $3m + 1 = 13$ output tuples in $Q_\triangle$. Any pair-wise join however has size $m^2 + m = 20$.

Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: New developments in the theory of join algorithms. SIGMOD Record 42, 4 (2013), 5–16.

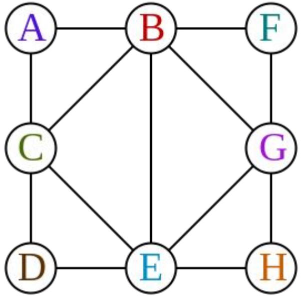
Worst-Case Optimal Join Algorithm

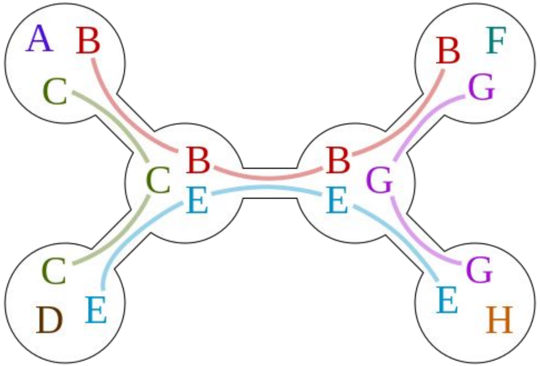
Graph Covers



$$|Q| = |\bowtie_{F\in\mathcal{E}} R_F| \leqslant \prod_{F\in\mathcal{E}} |R_F|^{x_F}. \qquad (6)$$

Graph Decomposition





https://en.wikipedia.org/wiki/Tree_decomposition

Worst-Case Optimal Join Algorithm

- Brute force all decompositions of relations
- Find minimum width decomposition

- Use decomposition to inform much order of joins, order of comparing fields, etc
- Resulting plan for joins is optimal intermediate output sizes

ALGORITHM 2: Enumerating all GHDs via Brute Force Search

```
// Input: Hypergraph H = (V, E) and a set of parent edges P.
// Output: A list of GHDs in the form of (GHD node, subtree) pairs.
GHD-Enumeration(V, E, P):
  GHDs = []
  // Iterate over all subset combinations of edges.
  for {C | C ⊆ E} do
    // The remaining edges, not in C.
    R = E − C
    // If the running intersection property is broken, the GHD is
    // not valid. The check makes sure that all attributes in the
    // parent and subtree of a specified GHD node also appear
    // within the specified GHD node. Here we use ∪ on a set
    // of edges to indicate the union of their attributes.
    if not ( (∪P) ∩ (∪R) ⊆ ∪C ) then continue
    // Consider each subgraph of the remaining edges. For each
    // subgraph, recursively enumerate all possible GHDs.
    PartitionChildren = []
    for Pr in Partition(R) do
      PartitionChildren += [GHD-Enumeration(∪Pr, Pr, C)]
    // Consider all possible combinations of subtrees by calling the
    // method below. For each, construct a GHD with C as the root.
    for Ch in Subtree-Combinations(PartitionChildren) do
      GHDs += [(C, Ch)]
  return GHDs

// Input: A list of lists of GHDs: for each partition of a hypergraph
// (the outer list), all possible decompositions for that partition
// (the inner lists).
// Output: A list where each member of this list is a list that
// contains one subtree from each partition.
Subtree-Combinations(PartitionChildren):
  ChildrenCombinations = []
  if |PartitionChildren| > 0 then
    if |PartitionChildren| == 1 then
      // If there is only one partition, for each of the possible GHDs
      // of this partition, add a combination with just this GHD.
      for Ch in PartitionChildren[0] do
        ChildrenCombinations += [[Ch]]
    else
      // Recursively generate combinations for the partitions after
      // the first one.
      RemainingCombinations = Subtree-Combinations(PartitionChildren[1:])
      // If there is more than one partition, each subtree in the
      // first partition is combined with each list of subtrees in
      // the recursively generated combinations for the remaining
      // partitions.
      for Ch in PartitionChildren[0] do
        for C in RemainingCombinations do
            FinalCombination = [Ch] + C
            ChildrenCombinations += [FinalCombination]
  return ChildrenCombinations
```

Execution Engine

Skew

- Density skew
  - some values are much more common
  - some relations are much more selective
- Cardinality skew
  - some tables are much larger
  - some nodes have much greater degree

Set Layout

- bitsets: bitvectors with offsets to first element
- pshort: nearby values may have similar prefix, thus store repeated high 16 bits
- varint: difference encoding with continue bits
- uint: sorted array with binary search

Set Intersection Algorithm

- uint & uint
  - 5 SIMD techniques, chosen based on density skew and cardinality
- bitset & bitset
  - Just SIMD AND comparison

Evaluation

Simplicity

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.`

Performance

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.

Performance

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

"-" indicates the engine does not process over 70 million edges. The other engines include Galois, PowerGraph, a commercial graph tool (CGT-X), SociaLite, and LogicBlox. "x" indicates the engine did not compute the query properly.