DunceCap: Query Plans Using Generalized Hypertree Decompositions

[Extended Abstract]

Susan Tu [*]
Stanford University
sctu@stanford.edu

Christopher Ré [†]
Stanford University
chrismre@stanford.edu

ABSTRACT

Joins are central to data processing. However, traditional query plans for joins, which are based on choosing the order of pairwise joins, are provably suboptimal. They often perform poorly on cyclic graph queries, which have become increasingly important to modern data analytics. Other join algorithms exist: Yannakakis', for example, operates on acyclic queries in runtime proportional to the input size plus the output size [7]. More recently, Ngo et al. published a join algorithm that is optimal on worst-case inputs [5]. My contribution is to explore query planning using these join algorithms. In our approach, every query plan can be viewed as a generalized hypertree decomposition (GHD). We score each GHD using the minimal fractional hypertree width, which Ngo et al. show allows us to bound its worst-case runtime. We benchmark our plans using datasets from the Stanford Large Network Dataset Collection [4] and find that our performance compares favorably against that of LogicBlox, a commercial system that implements a worst-case optimal join algorithm.

1. PRELIMINARIES

We begin by describing Yannakakis' and a worst-case optimal join algorithm.

1.1 Yannakakis

Yannakakis' algorithm consists of constructing the tree T_{GYO} implicit in performing the Graham-Yu-Ozsoyoglu (GYO) reduction. For every leaf v in T_{GYO} with relation R, compute $R := R \ltimes S$ where S is the relation in the leaf's parent. Now recursively fully reduce T_{GYO} without v. Then compute $S := S \ltimes R$. Finally, join each leaf in the tree with its parent. Since the semijoins "filter out" input tuples that will not appear in the final result, they prevent wasted work in the final join phase.

1.2 Worst-Case Optimal Join Algorithms

We implemented the worst-case optimal join algorithm described in Ngo, Rudra, and Ré's survey [5]., which we reproduce below as Algorithm 1. To see a situation in which this algorithm is optimal, consider the triangle query $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,A)$. As a concrete example (taken from [6]), consider having all three relations contain $(1,2)$, $(1,3)$, $(1,4)$, $(2,1)$, $(3,1)$, $(4,1)$. The maximum number of tuples that could result from a triangle query is $O(M^{3/2})$. Algorithm 1 is $O(M^{3/2})$; any pairwise join is $\Omega(M^2)$.

Algorithm 1 Worst-case optimal join algorithm

```
Input: Query Q, hypergraph H = (V, E)
def WorstCaseOptimalJoin(⋈_{F∈E} R_F)
    Q = ∅
    If |V| = 1 then
        return ⋂_{F∈E} R_F
    Let I = {v}, J = V \ I for some v ∈ V.
    A = WorstCaseOptimalJoin(⋈_{F∈E | I∩F≠∅} π_I(R_F))
    For every a ∈ A:
        partialQ{a} = WorstCaseOptimalJoin(
            ⋈_{F∈E | J∩F≠∅} π_J(R_F ⋉ {a}))
        Q = Q ∪ {a} × partialQ{a}
    return Q
```

2. APPROACH

DunceCap (DC), our query compiler, uses a hybrid algorithm to perform joins. Algorithm 1 is guaranteed to be optimal only for *worst-case* data but can be used for any query, whereas Yannakakis can only be used on acyclic queries. We might ask, given a cyclic query, how can we separate the query into subqueries such that the results of the subqueries can be joined using Yannakakis? (The subqueries would be computed using Algorithm 1.) The subquery that takes the most time in such a partition of work would contribute the dominant term in the worst case analysis, and if we parallelize the computation of the subqueries, the largest subquery will be the bottleneck computation. We therefore seek to minimize the size of the largest subquery output.

[*]Thanks to Adam Perelman, with whom I worked to implement the two join algorithms and who implemented the count query. Thanks to Chris Aberger and Andres Nötzli for implementation advice, and to Rohan Puttagunta and Manas Joglekar for theory help.

[†]This project is supported by the National Science Foundation CAREER Award (No. IIS-1353606).

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2764946.

2.1 Generalized Hypertree Decomposition

A theory problem that maps well to this problem of allocating subqueries to Algorithm 1 is finding a minimal width generalized hypertree decomposition (GHD). A GHD of a join query $\bowtie_{i\in\{1,\ldots,d\}} R_i$ is a tree T where each node contains at least one relation, each relation appears in at least one node (we also refer to these nodes as *bags*), and if a bag v and a bag u both contain an attribute A, all bags on the path between v and u contain A. Where $\alpha(v)$ denotes the number of attributes in node v and V is the set of all nodes in the GHD, the generalized hypertree width is $max_{v\in V}(\alpha(v)-1)$. While finding a minimal width GHD is NP-hard [3], our inputs are so small (our inputs are relations and their attributes, not tuples) that we can use Algorithm 2 to enumerate a GHD of every possible width.

Algorithm 2 Algorithm for enumerating GHDs

```
def enumerateGHDs(ℰ, attributes)
For every n in 1 to |ℰ|:
  For every possible subset S of size n in ℰ:
    For any R ∈ ℰ \ S:
      If R.attributes ∩ attributes ⊈ S.attributes
        return []
    Find connected components in ℰ \ S,
    ignoring attributes in S
    For every connected component 𝒞_i:
      subtrees{𝒞_i} = enumerateGHDs(
        𝒞_i, S.attributes)
    subtrees = {{c_1, c_2, ...} | each c_i ∈ subtrees{𝒞_i}}
    result = []
    For each s in subtrees:
      result.append(S with s as children)
    return result
```

2.1.1 Fractional Hypertree Width

The Atserias, Grohe, and Marx bound tells us that the output size is upper bounded by $\prod_{F\in\mathcal{E}}|R_F|^{x_F}$, under the constraints $\forall v\in\mathcal{V}, \sum_{F:v\in F} x_F \geq 1$ and $x_F \geq 0$ [2]. We would therefore like to minimize $\prod_{F\in\mathcal{E}}|R_F|^{x_F}$, subject to those constraints. If we assume that $|R_F| = N$ for all $F\in\mathcal{E}$ (which is true for graph queries), we can take the log of our objective and divide by $log(N)$. Then our linear program is: minimize $\sum_{F\in\mathcal{E}} x_F$, subject to $\forall v\in\mathcal{V}, \sum_{F:v\in F} x_F \geq 1$ and $x_F \geq 0$. Its solution is known as the fractional hypertree width (FHW). We can score all the GHDs according to the FHW instead of the generalized hypertree width.

2.2 Attribute Ordering

Because we represent our tables using tries, it is advantageous, for the joins in Yannakakis, to have the join attributes at or near the first level of the trie. We therefore compute a global attribute ordering by performing a pre-order traversal of our plan's GHD. If attribute A comes before B in the global attribute ordering, in the tries we index A before B, and we perform intersections on A before B in Algorithm 1.

3. RESULTS

We benchmarked our query plans on the Arxiv GR-QC collaboration network (5242 nodes, 14496 edges) and Facebook friend lists (4039 nodes, 88234 edges) [4]. (Each edge is represented as a tuple in our input tables.) Execution times are presented in Tables 1 and 2. Our numbers compare favorably with LogicBlox's despite their use of all 48 cores. We currently use 1 thread but are faster on all queries except for the 4-clique query on Facebook.

Query	DC 1 bag	DC min FHW	LogicBlox	Count
Triangle	0.051	0.051	0.56	290E3
(3,1)-Lollipop	0.133	0.069	0.92	~10E6
(4,1)-Lollipop	5.576	0.817	6.06	~342E6
4-Clique	0.538	0.538	1.37	~8E6

Table 1: Time (s) to run queries that count the occurrences of various graph structures on the Arxiv dataset. Note that for triangle and 4-clique, the 1-bag plan is the minimal FHW plan. Datasets are not pruned, so each triangle is counted 6x, etc. Benchmarks were run on a machine with 48 cores on 4 Intel Xeon E5-4657L v2 CPUs and 1 TB of RAM.

Query	DC 1 bag	DC min FHW	LogicBlox	Count
Triangle	0.281	0.281	0.88	~10E6
(3,1)-Lollipop	3.427	0.529	21.47	~1.4E9
(4,1)-Lollipop	416.753	48.408	1084.83	~121E9
4-Clique	29.437	29.437	26.23	~720E6

Table 2: Time (s) to run queries on the Facebook dataset. Queries and experimental setup are the same as in Table 1.

Note that there can be quite a drastic difference in execution times between the minimal FHW and any other arbitrary plan: For example, the query plan for the (4,1)-lollipop count query that puts a single edge from the 4-clique part of the structure into a bag and the rest in another bag takes over 14 seconds to run on the Arxiv dataset. The optimal plan, which places the 4-clique in a bag and the remaining edge in a bag (FHW=2), takes 817 ms. We also benchmarked all the 1 or 2-bag plans we generate for the (3,1)-Lollipop count query. The 2 plans with FHW=1.5 take 69 and 69 ms, the plans with FHW=2 take 137 to 10866 ms, and the plans with FHW=3 take 641 and 11967 ms. These times show that plans with lower FHW are indeed faster.

4. FUTURE WORK

EmptyHeaded is an graph pattern engine that uses SIMD parallelism to implement fast set intersections [1], which we plan to leverage in the base case of Algorithm 1. We plan to more thoroughly test our system on different queries (including non-graph queries where the relations have more attributes) and datasets.

5. REFERENCES

[1] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: Boolean Algebra Based Graph Processing. *ArXiv e-prints*, Mar. 2015.

[2] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins, 2008.

[3] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: Np-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, Sept. 2009.

[4] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[5] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. PODS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[6] H. Q. Ngo, C. Re, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *CoRR*, abs/1310.3314, 2013.

[7] M. Yannakakis. Algorithms for acyclic database schemes. VLDB '81, pages 82–94. VLDB Endowment, 1981.