GYM: A Multiround Distributed Join Algorithm

Foto Afrati
National Technical University of Athens
afrati@gmail.com

Manas Joglekar
Stanford University
manasrj@stanford.edu

Christopher Ré
Stanford University
chrismre@cs.stanford.edu

Semih Salihoglu
University of Waterloo
semihsalihoglu@gmail.com

Jeffrey Ullman
Stanford University
ullman@gmail.com

arXiv:1410.4156v4 [cs.DB] 6 Dec 2015

ABSTRACT

Multiround algorithms are now commonly used in distributed data processing systems, yet the extent to which algorithms can benefit from running more rounds is not well understood. This paper answers this question for a spectrum of rounds for the problem of computing the equijoin of n relations. Specifically, given any query Q with width w, *intersection width* iw, input size IN, output size OUT, and a cluster of machines with M memory available per machine, we show that:

1. Q can be computed in $O(n)$ rounds with $O(n\frac{(\text{IN}^{\mathsf{w}}+\text{OUT})^2}{M})$ communication cost.
2. Q can be computed in $O(\log(n))$ rounds with $O(n\frac{(\text{IN}^{\max(\mathsf{w},3\mathsf{iw})}+\text{OUT})^2}{M})$ communication cost.

Intersection width is a new notion of queries and generalized hypertree decompositions (GHDs) of queries we introduce to capture how connected the adjacent cyclic components of the GHDs are.

We achieve our first result by introducing a distributed and generalized version of Yannakakis's algorithm, called GYM. GYM takes as input any GHD of Q with width w and depth d, and computes Q in $O(d + \log(n))$ rounds and $O(n\frac{(\text{IN}^{\mathsf{w}}+\text{OUT})^2}{M})$ communication cost. We achieve our second result by showing how to construct GHDs of Q with width $\max(\mathsf{w}, 3\mathsf{iw})$ and depth $O(\log(n))$. We describe another technique to construct GHDs with longer widths and shorter depths, demonstrating a spectrum of tradeoffs one can make between communication and the number of rounds.

1. INTRODUCTION

The problem of evaluating joins efficiently in distributed environments has gained importance since the advent of Google's MapReduce [9] and the emergence of a series of distributed systems with relational operators, such as Pig [19], Hive [22], SparkSQL [21], and Myria [15]. These systems are conceptually based on Valiant's *bulk synchronous parallel* (BSP) computational model [23]. Briefly, there are a set of machines that do not share any memory and are connected by a network. The computation is broken into a series of *rounds*. In each round, machines perform some local computation in parallel and communicate messages over the network. Costs of algorithms in these systems can be broken down to: (1) local computation of machines; (2) communication between the machines; and (3) the number of new rounds of computation that are started, which can have large overheads in some systems, e.g. due to reading input from disk or waiting for resources to be available in the cluster. In this paper, we focus on communication and the number of rounds, as for many data processing tasks, the computation cost is generally subsumed by the communication cost [16, 17].

This paper studies the problem of evaluating an equijoin query Q in multiple rounds of computation in a distributed cluster. We let n be the number of relations, IN the input size, OUT the output size of Q, and M the size of the machines in the cluster, i.e., the memory available on each machine of the cluster. Machine sizes intuitively capture different parallelism levels: when machine sizes are smaller, we need a larger number of machines to evaluate the join, which increases parallelism. We assume throughout the paper that $M = \text{IN}^{\frac{1}{\epsilon}}$ for some constant $\epsilon > 1$. For practical values of input and machine sizes, ϵ is a small constant. For instance, if IN is in terabytes, then even when M is in megabytes, $\epsilon \approx 2$.

Our study of multiround join algorithms is motivated by two developments. First, it has been shown recently that there are prohibitively high lower bounds on the communication cost of any one-round algorithm for evaluating some join queries [2, 5]. Figure 1 shows this lower bound for the *chain query*, C_n=$R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie ... \bowtie R_n(A_{n-1}, A_n)$, for different machine sizes. The red curve, with the formula $(\frac{\text{IN}}{M})^{n/4}$, is the lower bound communication cost of any one-round algorithm. The black points depict the optimal one-round Shares algorithm [2], which is prohibitively inefficient. For example, if the input is one petabyte, i.e., IN=10^{15}, even when we have machines with one terabyte memory, i.e., M=10^{12}, the communication cost of Shares or any one-round algorithm to evaluate the C_{32} query is one billion petabytes. Moreover, this lower bound holds even when the query output is known to be small, e.g., OUT = O(IN), and the input has no skew [5], implying that designing multiround algorithms is the only way to compute such joins more efficiently.

Second, running a new round of computation has decreased from tens of minutes in the early systems (e.g., Hadoop [4]) to milliseconds in some recent systems (e.g.,

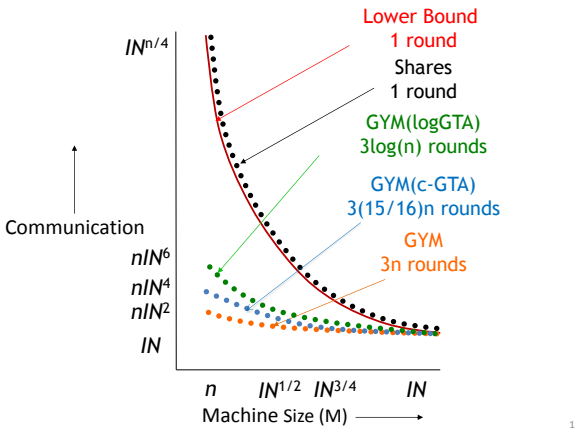


Figure 1: Memory vs Communication for C_n.

Spark [26]), making it practical to run algorithms that consist of a large number of rounds. Although multi-round algorithms are becoming commonplace, the extent to which algorithms can benefit from running more rounds is not well understood. In this paper, we answer this question for equijoin queries.

We describe a multiround algorithm, called *GYM*, for **G**eneralized **Y**annakakis in **M**apReduce, which is a distributed and generalized version of Yannakakis's algorithm for acyclic queries [24]. The performance of GYM depends on two important structural properties of the input query: *depths* and *widths* of its *generalized hypertree decompositions* (GHDs). Given a GHD D of Q, the number of rounds and communication cost of GYM is determined by the depth and width of D, respectively. We then present two algorithms, *Log-GTA* and *C-GTA*, for constructing GHDs of queries with different depths and widths, exposing a spectrum of tradeoffs one can make between the number of rounds and communication using GYM. For example, the green, blue, and orange curves in Figure 1 show the performance of GYM on three different GHDs of the chain query, when OUT is assumed to be $O(\text{IN})$. We note that the performance of GYM on these GHDs significantly outperforms the one-round Shares algorithm.

We present GYM within the context of the MapReduce system, because it is the earliest and one of the simplest modern large-scale data processing systems. However, GYM can easily run on any BSP system, so our results apply to other BSP systems as well. We also note that all of the results presented in this paper hold under any amount of skew in the input data.

Next, we give an overview of our main results. Then, we provide an outline for the rest of the paper.

1.1 GYM: A Multiround Join Algorithm

The width of a query, i.e., the minimum width of any of its GHDs, characterizes its degree of cyclicity, where acyclic queries are equivalent to width-1 queries. The original serial algorithm of Yannakakis takes as input a width-1 GHD of an acyclic query. GYM generalizes Yannakakis's algorithm to take as input any GHD of any query Q and evaluates Q in a distributed fashion. Our first main result is the following:

Main Result 1. *Given a width-*w*, depth-*d *GHD of a query* Q*, GYM computes* Q *in* $O(d + \log(n))$ *rounds with* $O(\frac{(\text{IN}^{\mathsf{w}}+\text{OUT})^2}{M})$ *communication cost.*

Since every width-w query over n relations has a GHD of width w and depth at most n, an immediate corollary to our first main result is that every width-w query can be computed in $O(n)$ rounds and $O(\frac{(\text{IN}^{\mathsf{w}}+\text{OUT})^2}{M})$ communication cost using GYM. GYM is based on three simple observations:

1. Yannakakis's algorithm runs a series of $\Theta(n)$ binary joins and semijoin operations on a width-1 GHD of an acyclic query. We can execute each binary join in one-round and each semijoin in a constant number of rounds, with a communication cost in $O(\frac{(\text{IN}+\text{OUT})^2}{M})$. The semijoin operator can require more than one round to eliminate the duplicate tuples that may exist in the output of its first round. The resulting algorithm takes $\Theta(n)$ rounds, and has a communication cost of $O(n\frac{(\text{IN}+\text{OUT})^2}{M})$.
2. In the algorithm from step 1, we can further execute multiple semijoins or joins simultanously in parallel, reducing the number of rounds to $O(d' + \log(n))$, where d' is the depth of the input width-1 GHD, without affecting the communication cost.
3. We can generalize the algorithm from step 2 to take as input any GHD of any (possibly cyclic) query Q. Let D be width-w GHD of Q. To evaluate Q, we first run the Shares algorithm on each vertex of D in parallel. This preprocessing step takes a constant number of rounds and at most $O(\text{IN}^{\mathsf{w}})$ cost and generates a set of acyclic intermediate relations over which the algorithm from step 2 can be run.

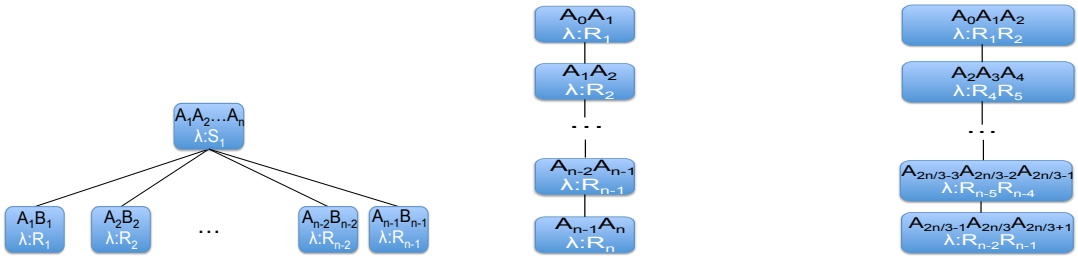
GYM is highly efficient on low width GHDs and executes for a small number of rounds on GHDs with short depths. Table 1 lists three example queries and their widths w, minimum depths of their width-w GHDs, and intersection widths (explained momentarily). Figure 2 shows example GHDs of these queries. The labels on the vertices of the GHDs in Figure 2 are the λ and χ values, which we review in Section 3.

Example 1. *The star query* S_n *is an acyclic query. As shown in Figure 2,* S_n *has a depth-1 and width-1 GHD. Using this GHD, GYM executes* S_n *in* $O(\log(n))$ *rounds with a communication cost of* $O(n\frac{(\text{IN}+\text{OUT})^2}{M})$.

Example 2. *The chain query* C_n *is also an acyclic query. Figure 2 shows an example width-*1 *GHD of* C_n *with depth* $n-1$*. On this GHD, GYM executes* C_n *in* $O(n)$ *rounds with a communication cost of* $O(n\frac{(\text{IN}+\text{OUT})^2}{M})$.

Query	Width	Minimum Depth of a width-w GHD	Intersection Width
$S_n : S(A_1, ..., A_n) \bowtie R_1(A_1, B_1) \bowtie ... \bowtie R_{n-1}(A_{n-1}, B_{n-1})$	1	1	1
$C_n : R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie ... \bowtie R_n(A_{n-1}, A_n)$	1	$\Theta(n)$	1
$TC_n : R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie R_3(A_0, A_2) \bowtie$ $R_4(A_2, A_3) \bowtie R_5(A_2, A_4) \bowtie R_6(A_3, A_4) \bowtie$ ... $R_{n-2}(A_{\frac{2n}{3}-1}, A_{\frac{n}{3}}) \bowtie R_{n-1}(A_{\frac{2n}{3}-1}, A_{\frac{2n}{3}+1}) \bowtie R_n(A_{\frac{2n}{3}}, A_{\frac{2n}{3}+1})$	2	$\Theta(n)$	1

Table 1: Example Queries S_n, C_n, and TC_n.



(a) S_n. (b) C_n. (c) TC_n.

Figure 2: Example GHDs of S_n, C_n, and TC_n.

1.2 Log-GTA: Log-depth GHDs

For some width-w queries, any width-w GHD of the query has a depth of $\Theta(n)$. C_n and the triangle-chain query, TC_n, shown in Table 1, are examples of such queries with widths 1 and 2, respectively. Therefore, on any width-w GHD of such queries, GYM executes $\Theta(n)$ rounds. Our second main result shows how to execute such queries by GYM in exponentially fewer number of rounds but with increased communication cost. To this end, we prove a combinatorial lemma about GHDs, which may be of independent interest to readers:

MAIN RESULT 2. *Given a width-*w*, intersection-width-*iw*, and depth-d GHD D of Q, we can construct a GHD D' of Q of depth $O(\log(n))$ and width at most* $\max(\mathsf{w}, 3\mathsf{iw})$.

Intersection width is a new notion of GHDs we introduce, that captures how connected the adjacent components of a GHD are. Following standard width definitions, we define the intersection width of a width-w query to be the minimum intersection width of any of its width-w GHDs. The intersection width of a query is at most its width, and sometimes strictly smaller, e.g., TC_n (from Figure 2c) has a width of 2 and intersection width 1.

We present an algorithm *Log-GTA*, for **Log**-*depth* **G***HD* **T***ransformation* **A***lgorithm*, to achieve our second main result. Our results imply that we can decrease the number of rounds from $O(n)$ to $O(\log(n))$ for width-w queries with long depth GHDs by increasing the communication cost from $O(n\frac{(\text{IN}^{\mathsf{w}}+\text{OUT})^2}{M})$ to $O(n\frac{(\text{IN}^{\max(\mathsf{w},3\mathsf{iw})}+\text{OUT})^2}{M})$: we take any width-w GHD D of Q, construct a GHD D' of depth $O(\log(n))$ and width $\max(\mathsf{w}, 3\mathsf{iw})$, and run GYM on D'.

EXAMPLE 3. *The TC_n query has a width of 2 and intersection width of 1. Figure 2c shown an example width-2 GHD D of TC_n which has a depth of $n/3$. One option of evaluating TC_n is to use D directly. On D, GYM will execute $\Theta(n)$ rounds and have a communication cost of $O(n\frac{(\text{IN}^2+\text{OUT})^2}{M})$. Another option is construct a new GHD D' from D by Log-GTA, which will have a depth of $O(\log(n))$ and width of 3. On D', GYM will take $O(\log(n))$ rounds and have a communication cost of $O(n\frac{(\text{IN}^3+\text{OUT})^2}{M})$.*

We end this section with a discussion of two interesting consequences of our Log-GTA and GYM algorithms.

Log-depth Decompositions

GHDs are one of several structural decomposition methods that are used to characterize the cyclicity of queries. Each decomposition method represents queries as a graph (if the input relations have arity at most 2) or a hypergraph and has a notion of "width" to measure the cyclicity of queries. Examples include *query decompositions* [8], *tree decompositions* (TDs) [20], and *hypertree decompositions* (HDs) [12]. Two previous results by Bodlaender [6] and Akatov [3] have proved the existence of log-depth TDs of graphs with thrice their *treewidths* and HDs of hypergraphs with thrice their *hypertreewidths*, respectively. Our second main result proves that a similar and stronger property also holds for GHDs of hypergraphs. As we discuss in Section 2, interestingly, neither of these results imply each other. However, we show that Log-GTA also recovers and generalizes Bodlaender's result from graphs to hypergraphs. In addition, we show that a modification of our Log-GTA algorithm recovers both Akatov's and Bodlaender's results and a weaker version of our second main result. Finally, we show that using similar definitions of intersection widths for TDs and HDs, we can improve both Bodlaender's and Akatov's results.

Parallel Complexity of Bounded-width Queries

	Shares(S_n)	**ACQ-MR**(S_n)	**GYM**(D_{S_n})
# Rounds	1	$O(\log(n))$	$O(\log(n))$
Communication	$O(\frac{\text{IN}^{\frac{n}{2}}}{M^{\frac{n}{2}}} + \text{OUT})$	$O(n\frac{(\text{IN}^3+\text{OUT})^2}{M})$	$O(n\frac{(\text{IN}+\text{OUT})^2}{M})$

Table 2: Worst-Case Complexity of Algorithms on S_n. D_{S_n} is a $O(1)$-depth GHD of S_n.

	Shares(TC_n)	**ACQ-MR**(TC_n)	**GYM(Log-GTA**(D_{TC_n}))	**GYM**(D_{TC_n})
# Rounds	1	$O(\log(n))$	$O(\log(n))$	$O(n)$
Communication	$O(\frac{\text{IN}^{\frac{n}{6}}}{M^{\frac{n}{6}}} + \text{OUT}))$	$O(n\frac{(\text{IN}^6+\text{OUT})^2}{M})$	$O(n\frac{(\text{IN}^3+\text{OUT})^2}{M})$	$O(n\frac{(\text{IN}^2+\text{OUT})^2}{M})$

Table 3: Worst-Case Complexity of Algorithms on TC_n. D_{TC_n} is a $\theta(n)$-depth GHD of TC_n.

Database researchers have often thought of Yannakakis's algorithm as having a sequential nature, executing for $\Theta(n)$ steps in the PRAM model. In the PRAM literature [10, 13, 14], acyclic queries have been described as being polynomial-time sequentially solvable by Yannakakis's algorithm, but highly "parallelizable" by the *ACQ* algorithm [11], where parallelizability refers to executing for a small number of PRAM steps. By constructing log-depth GHDs of queries and simulating GYM in the PRAM model, we show that: (1) unlike previously thought, with simple modifications Yannakakis's algorithm can run in logarithmic rounds; and (2) evaluating bounded-width queries is in complexity class NC, which recovers a result proven by reference [11] through the ACQ algorithm.

1.3 Outline of the Paper

- In Section 2, we discuss related work on one-round and multiround join algorithms and structural decomposition methods.
- In Section 3, we provide the background on GHDs, introduce the notion of intersection width, and specify the computational model we use in this paper.
- In Section 4, we describe two distributed versions of Yannakakis's algorithm, *DYM-n* and *DYM-d*, as a stepping stones to GYM. Both algorithms take as input width-1 GHDs of acyclic queries.
- In Section 5, we describe GYM, which generalizes DYM-d to any width-w, depth-d GHD of any query.
- In Section 6, we describe our Log-GTA algorithm for transforming any width-w, intersection width-iw GHD D into another GHD D', whose depth is $O(\log(n))$ and width is at most $\max(\mathsf{w}, 3\mathsf{iw})$.
- In Section 7, we describe another GHD transformation algorithm called *C-GTA*, which transforms a width-w *GHD* into a width-2w *GHD* with fewer vertices. We show that we can use C-GTA along with Log-GTA to construct GHDs with shorter depths and higher widths.
- In Section 8, we discuss the improvements to our results when the inputs to queries are skew-free.
- In Section 9, we conclude and discuss future work.

2. RELATED WORK

We first review related work on distributed equijoin algorithms. Then, we review work in structural decomposition methods.

2.1 Distributed Equijoin Algorithms

In this section, we review the Shares one-round equijoin algorithm and the ACQ algorithm, which is an $O(\log(n))$-step PRAM algorithm, but can easily be modified into a distributed BSP algorithm, which we refer to as ACQ-MR. We then cover related work on other distributed join algorithms. For reference, Tables 2 and 3 compare the performance of GYM, Shares and ACQ-MR on the S_n, and TC_n queries from Table 1. In the tables, *GYM(Log-GTA(D))* refers to our combined algorithm that first runs our Log-GTA algorithm on a GHD D and constructs a new GHD D', and then runs GYM with D' to evaluate the input query.

2.1.1 Shares

The one-round Shares [2] algorithm was introduced within the context of the MapReduce system. Briefly, Shares divides the output space equally across the machines and replicates each input tuple t to the machines that contain an output tuple that is constructed from t, while guaranteeing that each machine gets no more than M input tuples. References [1] and [5] have shown that for every query Q, each parallelism level M, and skew level, the Shares algorithm can be configured to incur the lowest communication cost possible among one-round algorithms. However, as we discussed earlier, for some queries, these costs can be prohibitively expensive. Shares is especially inefficient when computing queries with small outputs using small-size machines, i.e., small values of M. For example, in the easiest setting of no-skew in the input data, the communication cost of Shares for the TC_n query is $O(\frac{\text{IN}^{\frac{n}{6}}}{M^{\frac{n}{6}}} + \text{OUT})$. Therefore, even when M is large, say $\sqrt{\text{IN}}$, the cost of Shares on TC_n is exponential in n. However, at the same parallelism levels, the cost of GYM for TC_n is $O(n\frac{(\text{IN}^2+\text{OUT})^2}{\sqrt{\text{IN}}})$. In general, GYM significantly outperforms Shares in terms of communication cost (while requiring more rounds) when executing low-width queries, such as S_n, C_n, and TC_n, at high parallelism levels.

2.1.2 ACQ

The ACQ algorithm [11] is the most efficient known $O(\log(n))$-step PRAM algorithm for computing bounded-width queries. Because BSP model can simulate the PRAM model, the ACQ algorithm can easily be mapped to distributed BSP systems, such as MapReduce. We call this algorithm *ACQ-MR*. ACQ-MR evaluates a width-w query Q in $\Theta(\log(n))$ rounds and

$O(n\frac{(\text{IN}^{3\mathsf{w}}+\text{OUT})^2}{\text{M}})$ communication. We compare GYM's performance to ACQ-MR under three cases:

1. If Q has a short-depth width-w GHDs, such as the S_n query, GYM outperforms ACQ-MR in communication cost while using a comparable number of rounds (Table 2).
2. If Q has long-depth, say of $\Theta(n)$-depth, GHDs but has an intersection width (see Section 3) that is strictly lower than its width, such as the TC_n query, then: (1) GYM outperforms ACQ-MR in terms of communication but executes for an exponentially larger number of rounds; and (2) GYM(Log-GTA) also incurs less communication cost than ACQ-MR (but more than GYM), while using a comparable number of rounds (Table 2).
3. If Q has long-depth GHDs and has an intersection width that is equal to its width, such as the C_n query, (1) GYM outperforms ACQ-MR in terms of communication but executes for an exponentially larger number of rounds; and (2) GYM(Log-GTA) matches ACQ-MR both in communication cost and the number of rounds.

The ACQ algorithm was used to prove that bounded-width queries are in the parallel complexity class NC. Because PRAM can also simulate MapReduce, our GYM(Log-GTA) method also proves this result. We believe this is interesting within itself, since we recover this positive parallel complexity result by using only a simple variant of Yannakakis's algorithm, which has been thought to be an inherently sequential algorithm.

2.1.3 Other Distributed Join Algorithms

Reference [18] shows that increasing the number of MapReduce rounds from 1 to $\log_M(\text{IN})$ can significantly reduce communication cost. Since $\log_M(\text{IN})$ is a small constant in our setting, we can run this algorithm instead of Shares as a subroutine, to reduce our communication cost. Only one other work [5] studies multiround distributed join algorithms. This work proves lower bounds on the number of rounds required to compute queries when $M = \frac{\text{IN}}{p^\epsilon}$, where p is the number of processors, and ϵ is called the *space exponent*. The authors show that running the Shares algorithm iteratively on sets of the input relations, matches these lower bounds on a limited set of inputs, called *matching databases*. For matching databases, the size of the output and any intermediate output is always at most the size of the input. On non-matching databases, their iterative Shares algorithm can produce intermediate results of size $\text{IN}^{\theta(n)}$ for any query irrespective of its width. On matching databases, our algorithms asymptotically match these lower bounds in terms of rounds and efficiency. On arbitrary databases, our algorithms are a $\log(n)$ factor away from their lower bounds in term of the number of rounds, while keeping intermediate relation sizes bounded by $\text{IN}^{\max(w,3\mathsf{iw})} + \text{OUT}$.

2.2 Generalized Hypertree Decompositions

Structural decomposition methods, such as GHDs, query decompositions (QDs) [8], tree decompositions (TDs) [20], and hypertree decompositions (HDs) [12], are mathematical tools to characterize the difficulty of computational problems that can be represented as graphs or hypergraphs, such as joins or constraint satisfaction problems. GYM can use methods other than GHDs, such as QDs, and HDs, but we use GHDs because the widths of GHDs are known to smaller than HDs and QDs, giving us stronger results in terms of communication cost.

Several prior work study the problem of computing GHDs of hypergraphs of queries with small, sometimes minimum, width [7, 12]. This paper does not study the problem of finding a minimum width GHD a query. Instead, we assume such a GHD has already been computed by one of the existing methods. We show how to use such GHDs to parallelize query evaluation.

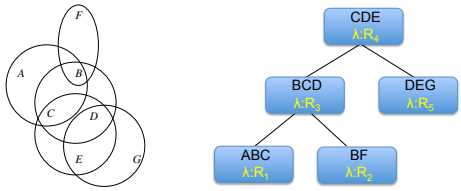
We prove in this paper that width-w and intersection-width-iw hypergraphs have log-depth GHDs with width at most $\max(\mathsf{w}, 3\mathsf{iw})$. Two previous works have shown similar properties to hold for TDs and HDs, which we can generalize.

- **Bodlaender's Result:** Bodlaender [6] shows the existence of log-depth TDs of graphs with at most thrice the optimal *treewidth*. TDs are decompositions of graphs and HDs and GHDs are TD's natural counterparts for hypergraphs. Bodlaender's proof technique contracts a possibly long-depth TD into a log-depth one. Similarly, our Log-GTA algorithm contracts a possibly long-depth GHD into a log-depth one. However, our result does not imply Bodlaender's (and vice versa) because treewidth and (generalized hypertree) width are measured differently. Specifically, the treewidth of a decomposition can be unboundedly higher than its width.
- **Akatov's Result :** Akatov [3] shows the existence of log-depth HDs with at most thrice the optimal *hypertree-width*. HDs are GHDs with an additional "descendant property" (see Section 3). Again, our result does not imply Akatov's result and vice-versa because the hypertree-width (hw) and (generalized hypertree) width (ghw) of a hypergraph may be different. So the existence of log-depth HDs with at most thrice hw only implies that there are GHDs with width 3hw, not 3ghw. Similarly, the existence of log-depth GHDs with ghw $\max(\mathsf{w}, 3\mathsf{iw})$ does not imply that such GHDs are HDs, i.e., satisfy the descendant property.

Interestingly, although these results do not imply each other, we show in Section 6.3 that Log-GTA recovers and generalizes Bodlaender's result from graphs to hypergraphs. Moreover, in Section 6.4, we show that a modification of Log-GTA can transform a TD, HD, or a GHD to recover all three results. We also show in Appendix A.6 that with similar definitions of intersection widths for TDs and GHDs, we can improve Bodlaender and Akatov's results. Finally, neither Bodlaender nor Akatov show how to use decompositions to parallelize query evaluation.

3. PRELIMINARIES

We review the notions related to generalized hypertree decompositions of queries [12] and then describe



(a) Hypergraph. (b) GHD.

Figure 3: Hypergraph and GHD of Example 4.

our cost model.

3.1 Generalized Hypertree Decompositions

A ***hypergraph*** is a pair $H = (V(H), E(H))$, consisting of a nonempty set $V(H)$ of vertices, and a set $E(H)$ of subsets of $V(H)$, the hyperedges of H. Natural join queries can be expressed as hypergraphs, where we have a vertex for each attribute of the query, and a hyperedge for each relation.

EXAMPLE 4. *Consider the query Q:*

$$R_1(A,B,C) \bowtie R_2(B,F) \bowtie R_3(B,C,D) \bowtie R_4(C,D,E) \bowtie R_5(D,E,G)$$

The hypergraph of Q is shown in Figure 3a.

Let H be a hypergraph. A ***tree decomposition (TD)*** of H is a tuple $D = (T, \chi)$, where:

- $T(V(T), E(T))$ is a tree;
- $\chi : V(T) \to 2^{V(H)}$ is a function associating a set of vertices (attributes) $\chi(t) \subseteq V(H)$ to each vertex t of T;

such that the following properties hold:

1. For each $e \in E(H)$, there is a vertex $t \in V(T)$ such that $e \subseteq \chi(t)$.
2. For each $v \in V(H)$, the set $\{t \in V(T) | v \in \chi(t)\}$ is connected in T.

For any $t \in V(T)$, we refer to $\chi(t)$ as the attributes of t. The *treewidth* of TD is $\max_{t \in V(T)}\{|\chi(t)|\} - 1$. The treewidth of hypergraph is the minimum treewidth of any of its TDs. We note that we will not use the notion of treewidth in the paper except when we discuss how Log-GTA recovers Bodlaender's result.

A ***generalized hypertree decomposition (GHD)*** of H is a triple $D = (T, \chi, \lambda)$, where:

- (T, χ) is a TD;
- $\lambda : V(T) \to 2^{E(H)}$ is a function associating a set of hyperedges (relations) to each vertex t of T;

such that the following additional property holds:

3. For every $t \in V(T)$, $\chi(t) \subseteq \bigcup \lambda(t)$.

We refer to $\lambda(t)$ as the relations on t. A GHD of a join query Q is a GHD on the hypergraph of Q.

EXAMPLE 5. *Figure 3b shows a GHD of the query from Example 4. In the figure, the attribute values on top of each vertex t are the χ assignments for t and the λ assignments are explicitly shown.*

We next define several properties of GHDs and hypergraphs:

Figure 4: Hypergraph of TC_n.

- The ***depth*** of a GHD $D = (T, \chi, \lambda)$ is the depth of the tree T.
- The ***width*** of a GHD D is the $\max_{t \in V(T)}\{|\lambda(t)|\}$, i.e., the maximum number of relations assigned to any vertex t.
- The ***generalized hypertree width***, or ***width*** for short, of a hypergraph H is the minimum width of all GHDs of H.

The width of a query captures its degree of cyclicity. In general, the larger the width of a query, the more "cyclic" it is. Acyclic queries are exactly the queries with width one [7]. We next define a new notion called intersection width.

- The ***intersection width*** of a GHD $D = (T, \chi, \lambda)$ is defined as follows: For any adjacent vertices $u, v \in V(T)$, let $\mathsf{iw}(u, v)$ denote the size of the smallest set $S \subseteq E(H)$ such that $\chi(u) \cap \chi(v) \subseteq \bigcup_{s \in S} s$. In other words, $\mathsf{iw}(u, v)$ is the size of the smallest set of relations whose attributes cover the common attributes between u and v. The intersection width iw of a GHD is the maximum $\mathsf{iw}(u, v)$ over all adjacent $u, v \in V(T)$.

Notice that the intersection width of D is never larger than the width of D, because $\forall\ u, v \in V(T) : \mathsf{iw}(u, v) \leq |\lambda(u)|$, since by the 3rd property of GHDs $\lambda(u)$ is one (possibly not smallest) set of relations that covers the attributes of u, and therefore any common attribute that u shares with its neighbors. The intersection width of a GHD can be strictly smaller than the width, as the next example shows.

EXAMPLE 6. *Consider the TC_n example from Table 1. TC_n is a width-2 query. The hypergraph of TC_n, shown in Figure 4, visually is a chain of triangles connected by a single attribute. Figure 2c shows a width-2 GHD of TC_n, where each node covers one of the triangles in the same order they appear in the hypergraph. The intersection width of this GHD is 1, as the single common attribute between each triangle can be covered by a single relation.*

In the rest of this paper we restrict ourselves, for simplicity of presentation, to queries whose hypergraphs are connected. However, all of our results generalize to queries with disconnected hypergraphs. We end this section by stating a lemma about connected hypergraphs and GHDs of queries that will be used in later sections:

LEMMA 1. *If a query Q has a width-w GHD $D = (T, \chi, \lambda)$ of depth d, then Q has a GHD $D' = (T', \chi', \lambda')$ with depth d width w and $|V(T')| \leq n$.*

The proof of this lemma is provided in Appendix A.1.

3.2 MapReduce and our Cost Model

In the MapReduce (MR) model, there are unboundedly many processors on a networked file system. Each

processor has unbounded hard disk space and main memory M. The computation proceeds in two phases.

Step 1: Each processor (referred to as a mapper), reads its tuples from its hard disk and sends each tuple to one or more processors (called reducers). The total number of tuples recieved by each reducer from all mappers should not exceed memory size M.

Step 2: Each reducer locally processes the $\leq M$ tuples it recieves, and streams its output to the network file system. The local computation at a reducer cannot exceed memory size M, but the output of a reducer can exceed M as it is streamed to the file system.

The *communication cost* of each round is defined as the total number of tuples sent from all mappers to reducers plus the number of output tuples produced by the reducers. We measure the complexity of our algorithms in terms of communication cost and number of rounds. In the remainder of the paper, we let $B(X, M) = \frac{X^2}{M}$ to simplify the presentation of results.

4. DISTRIBUTED YANNAKAKIS

We first review the serial version of Yannakakis's algorithm for acyclic queries in Section 4.1. In Section 4.2, we show how to run Yannakakis's algorithm in a distributed setting in $O(n)$ rounds and $O(n\text{B}(\text{IN}+\text{OUT}, M))$ communication cost (Section 4.2). In Section 4.3, we show how to reduce the number of rounds to $O(d + \log(n))$ rounds without affecting the communication cost.

4.1 Serial Yannakakis Algorithm

The serial version of the Yannakakis algorithm takes as input an acyclic query $Q = R_1 \bowtie R_2 \bowtie .... \bowtie R_n$, and constructs a width-1 GHD $D = (T, \chi, \lambda)$ of Q. Since D is a GHD with width 1, each vertex of D is assigned exactly one relation R_i and each R_i is assigned to some vertex of D. We will refer to relations that are assigned to leaf (non-leaf) vertices in T as *leaf (non-leaf) relations*. Before joining the relations of Q, Yannakakis's algorithm eliminates all dangling tuples from the input, i.e., those that will not contribute to the final output, by a series of semijoin operations. The overall algorithm consists of two phases: (1) a **semijoin phase**; and (2) a **join phase**. The dangling tuple elimination guarantees that the sizes of all intermediate tables during the join phase are no larger than the final output [24].

Semijoin Phase: Consider a width-1 GHD $D = (T, \chi, \lambda)$ of an acyclic query Q. The semijoin phase operates recursively as follows.

BASIS: If T is a single node, do nothing.

INDUCTION: If T has more than one node, pick a leaf t that is assigned relation R, and let S be the relation assigned to t's parent.

1. Replace S by the semijoin of S with R, $S \ltimes R = S \bowtie \pi_{R \cap S}(R)$.
2. Recursively process $T \setminus R$.
3. Compute the final value of R by computing its semijoin with the value of S that results from step (2); that is, $R := R \ltimes S$.

The executions of step (1) in this recursive algorithm form the *upward* phase, and the executions of step (3) form the *downward* phase. In total, this version of the algorithm performs 2(n-1) semijoin operations.

Join Phase: Next, the algorithm performs a series of (n-1) joins, in any bottom-up order on T.

4.2 DYM-n

We start this section by a key lemma about performing binary join and semijoin operations in MR.

LEMMA 2. *Any two relations R and S can be joined in 1 round with $O(\frac{(|R|+|S|)^2}{M} + |R \bowtie S|)$ communication, and semijoined in $O(1)$ rounds with $O(\frac{(|R|+|S|)^2}{M})$ communication.*

PROOF. We perform the join as follows: We divide R and S into $g_r = \frac{2|R|}{M}$ and $g_s = \frac{2|S|}{M}$ disjoint groups of size $\frac{M}{2}$ each. Then we use a total of $g_r \cdot g_s$ machines and send a distinct pair of groups to each machine, which joins its groups locally. Thus, each of $g_r \cdot g_s$ machines gets input communication equal to M, and the total output size is $|R \bowtie S|$, resulting in a total communication cost of $O(\frac{|R||S|}{M} + |R \bowtie S|) = O(\frac{(|R|+|S|)^2}{M} + |R \bowtie S|)$.

The first step of the semijoin $S \ltimes R$ is similar to the join: Each of the $g_r g_s$ processors gets $\frac{M}{2}$ tuples from each relation. Then each processor locally computes the semijoin of one S group and one R group it receives. Because each tuple of S is sent to g_r different machines, there may be up to g_r duplicates of each tuple. We next eliminate these duplicates. Let h be a hash function mapping the tuples of S into $|S|^2$ buckets randomly, so with high probability each bucket $h(i)$ gets $O(1)$ tuples. In the second round we use $|S|^2 g_r^2$ reducers indexed with two numbers, $(1,1), ..., (1, g_r^2), \ldots, (|S|^2, 1), ..., (|S|^2, g_r^2)$, and each tuple t is mapped to a reducer with first index $h(t)$ and a uniformly random second index. Therefore, with high probability, each reducer gets $O(1)$ tuples. Note that every duplicate of tuple t is mapped to a reducer with first index $h(t)$. In addition, the number of non-duplicate tuples across all of the reducers with the same first index is $O(1)$, since h maps $O(1)$ different S tuples to each $h(i)$. In later rounds, for each set of reducers with the same first index we do the following in parallel: We group the reducers into groups of $\sqrt{M}$, map their inputs to the same reducer, and eliminate the duplicates across them. This reduces the number of reducers that can contain duplicates to $\frac{g_r^2}{\sqrt{M}}$. We then repeat this procedure until all duplicates within each group of reducers with the same first index are eliminated. In each round, each reducer gets $O(\sqrt{M})$ tuples and outputs $O(1)$ tuples. This computation takes $O(log_{\sqrt{M}}(g_r^2))$ rounds. Since $g_r = \frac{2|R|}{M} = O(\text{IN})$ and we assume $M = \text{IN}^{\frac{1}{\epsilon}}$, we remove the duplicates in $O(\epsilon) = O(1)$ rounds. □

If we simply execute each semijoin and join operation of Yannakakis's algorithm by the algorithms in Lemma 2, we get a distributed algorithm which we refer to as ***DYM-n***:

THEOREM 1. *DYM-n computes every acyclic query Q in $O(n)$ rounds and in $O(n\text{B}(\text{IN} + \text{OUT}, M))$ communication cost.*

PROOF. For each edge in T, there are exactly two semijoin operations, once in the upward phase and once in the downward phase, and one join operation. Therefore, the algorithm executes a total of $3(n\text{-}1)$ pairwise joins and semijoins, in a total of $O(n)$ rounds. Since there are $2(n\text{-}1)$ semijoins and the largest input to any semijoin operation is the largest relation size, which is at most IN, the communication cost of the semijoin phase is $O(n\text{B}(\text{IN}, M))$. In each round of the join phase, the input and outputs are at most the final output size OUT. So by Lemma 2, the cost of each round is $O(\frac{\text{OUT}^2}{M} + \text{OUT})$. Assuming $\text{IN} \times \text{OUT}$ is larger than the size of a single machine, the sum of the costs of both phases is $O(n\text{B}(\text{IN} + \text{OUT}, M))$, completing the proof. □

4.3 DYM-d

In addition to parallelizing each semijoin and join operation, we can parallelize Yannakakis's algorithm further by executing multiple semijoins and joins in parallel. With this extra parallelism, we can reduce the number of rounds to $O(d + \log(n))$, where d is the depth of the GHD $D(T, \chi, \lambda)$, without asymptotically affecting the communication cost of the algorithm. We refer to this algorithm as ***DYM-d***.

Upward Semijoin Phase in $O(d+\log(n))$ Rounds: Consider any leaf node R of T, with parent S. During the upward semijoin phase, we replace S with $S \ltimes R$ in $O(1)$ rounds. But instead of using the rounds to only process R, we can process all leaves in parallel. We now give a recursive procedure for performing the semijoin phase. Given a GHD $D = (T, \chi, \lambda)$:
BASIS: If T is a single node, do nothing.
INDUCTION: If T has more than one node, consider the set L of leaves of T. Let L_1 be the set of leaves that have no siblings, and let L_2 be the remaining leaves.

1. For each R in L_1 with parent S, replace S with $S \ltimes R$, and remove R from the tree for the duration of the upward semijoin phase.
2. Divide the leaves in L_2 into disjoint pairs of siblings, and upto one triple of siblings, if there is an odd number of siblings with the same parent. Suppose R_1 and R_2 form such a pair with parent S. Then replace R_1 with $(S \ltimes R_1) \cap (S \ltimes R_2)$ and remove R_2, for the duration of the upward semijoin phase. If there is a triple R_1, R_2, R_3, replace R_1 with $(S \ltimes R_1) \cap (S \ltimes R_2) \cap (S \ltimes R_3)$ (using two pairwise intersections) and remove R_2 and R_3.
3. Recursively process the resulting T.

LEMMA 3. *The above procedure takes $O(d + \log(n))$ rounds.*

PROOF. Steps (1) and (2) of each recursive call can be performed in $O(1)$ rounds, in parallel for all leaves. So we only need to prove that the number of recursive calls is $O(d + \log(n))$. For any tree T, let $X(T) = \sum_{l \in L(T)} 2^{d(l)}$, where $L(T)$ denotes the leaves in T and $d(l)$ is the depth of leaf l. Then, X of the original join tree T is at most $n2^d$, as there are at most n leaves, with depth at most d each. Now consider what happens to X of the tree in each recursive call. Each leaf l is in either L_1 or L_2. If it is in L_1, it gets deleted. If l's parent has no other children, then the parent becomes a new leaf, of depth $d(l) - 1$. Thus the $2^{d(l)}$ term in X is at least halved for all leaves in L_1. On the other hand, if l_1, l_2 form a pair in L_2, then one of them gets deleted, while the other stays at the same depth. Thus the $2^{d(l_1)} + 2^{d(l_2)}$ term also gets halved. For a triple in L_2, the term becomes one-third. Thus X reduces by at least half in each recursive call. The recursion terminates when T is a single node, i.e. when $X(T) = 1$. Since its starting value is at most $n2^d$, the number of recursive calls, is $O(\log(n2^d)) = O(d + \log(n))$. □

```
1  Input: GHD D(T, χ, λ) of a query Q
2  Materialization Stage:
3  foreach vertex v in D (in parallel):
4  Compute IDB_v = ⋈_{R_i ∈ λ(v)} R_i by the Shares algorithm.
5  Yannakakis Stage:
6  Let Q' = ⋈_v IDB_v.
7  Execute DYM−d on Q'.
```

Figure 5: GYM.

Since we perform $O(n)$ intersection or semijoin operations in total, the total communication cost of the upward semijoin phase is $O(n\text{B}(\text{IN}, M))$, as all initial and intermediate relations involved have size at most IN.

Downward Semijoin Phase in $O(d)$ Rounds: Note that in the downward semijoin phase, the semijoins of the children relations with the same parent are independent and can be done in parallel in $O(1)$ rounds. Thus we can perform the downward phase in $O(d)$ rounds and in $O(n\text{B}(\text{IN}, M))$ communication.

Join Phase in $O(d+\log(n))$ Rounds: The join phase is similar to the upward semijoin phase. The only difference is, we compute $S \bowtie R$ instead of $S \ltimes R$ for $R \in L_1$, and $(R_1 \bowtie S) \bowtie (R_2 \bowtie S)$ for pair $R_1, R_2 \in L_2$. The total number of rounds required is again $O(d + \log(n))$. The total communication cost of each pairwise join is $O(n\text{B}(\text{OUT}, M))$, since the intermediate relations being joined are at most as large as OUT. Therefore, both the semijoin and join phases can be performed in $O(d+\log(n))$ rounds with a total communication cost of $O(n\text{B}(\text{IN}+\text{OUT}, M))$, justifying the following theorem:

THEOREM 2. *DYM-d can compute every acyclic query Q in $O(d + \log(n))$ rounds of MR and $O(n\text{B}(\text{IN} + \text{OUT}), M))$ communication cost, where d is the depth of a width-1 GHD $D(T, \chi, \lambda)$ of Q.*

5. GYM

Our *GYM* algorithm generalizes DYM-d from acyclic queries to any query. Consider a width-$\mathtt{w}$, depth-d GHD $D(T, \chi, \lambda)$ of a query Q. Assume for simplicity that each relation R_i is assigned to some vertex $v \in V(T)$ of D. The algorithm is based on the following observation. Consider "materializing" each $v \in V(T)$ by computing $IDB_v = \bowtie_{R_i \in \lambda(v)} R_i$. Now, consider the query

$Q' = \bowtie_{v \in V(T)} IDB_v$. Note that Q' has the exact same output as Q. This is because Q' is also the join of all R_i, where some R_i might (unnecessarily) be joined multiple times if they are assigned to multiple vertices. However, observe that Q' is now an acyclic query. In particular, after materializing each IDB_v, D is now a width-1 GHD for Q'. Therefore we can directly run DYM-d to compute Q'. Figure 5 shows the pseudo-code of GYM, which consists of a *Materialization Stage* and a *Yannakakis Stage*.

THEOREM 3 (**First Main Result**). *Given a width-*w*, depth-d GHD* $D(T, \chi, \lambda)$ *of a query* Q*, GYM executes* Q *in* $O(d + \log(n))$ *rounds and* $O(|V(T)|\text{B}(\text{IN}^{\mathsf{w}} + \text{OUT}, M))$ *communication cost.*

PROOF. For the materialization stage, we assume the worst case scenario that for each vertex t of D, the join of the w relations inside $\lambda(t)$ is a Cartesian product. In this case, no matter what the parallelism level is, the communication cost of Shares is IN^{w}. Therefore, the materialization stage takes $O(1)$ rounds and $O(|V(T)|\text{B}(\text{IN}^{\mathsf{w}}, M))$ communication cost. Executing DYM-d on the IDB_v's takes $O(d + \log(n))$ rounds and $O(|V(T)|\text{B}(\text{IN}^{\mathsf{w}} + \text{OUT}), M))$ cost by Theorem 2 and the fact that the size of each IDB_v is at most IN^{w}. Therefore, GYM takes $O(d + \log(n))$ rounds and has $O(|V(T)|\text{B}(\text{IN}^{\mathsf{w}} + \text{OUT}), M))$ cost. □

In appendix A.3, we present an example execution of GYM on a query. We note that contrary to our simplifying assumption, in general, each R_i of Q may not be assigned to some vertex $v \in V(T)$ of D. In Appendix A.2, we discuss how to modify GYM to handle this technicality without affecting our results.

COROLLARY 1. *Any width-*w *query can be computed in* $O(n)$ *rounds and* $O(n\text{B}(\text{IN}^{\mathsf{w}} + \text{OUT}, M))$ *communication cost.*

The proof of Corollary 1 is immediate from Theorem 3 and Lemma 1 that states that every width-w query has a width-w GHD D with at most n vertices, which implies that D has $O(n)$-depth.

6. LOG-GTA

We now describe our ***Log-GTA*** algorithm (for **Log**-depth **G**HD **T**ransformation **A**lgorithm) which takes as input a hypergraph H of a query Q, and its GHD $D(T, \chi, \lambda)$ with width w and intersection width iw, and constructs a GHD D^* with depth $O(\log(|V(T)|))$ and width $\leq \max(\mathsf{w}, 3\mathsf{iw})$. For simplicity, we will refer to all GHDs during Log-GTA's transformation as $D'(T', \chi', \lambda')$, i.e., $D' = D$ in the beginning and $D' = D^*$ at the end.

Our Log-GTA algorithm and Lemma 1 prove that $\log(n)$-depth GHDs of hypergraphs with width $\max(\mathsf{w}, 3\mathsf{iw})$ exist. By running GYM on D^*, we can execute Q in $O(\log(n))$ rounds with $O(n\text{B}(\text{IN}^{\max(\mathsf{w}, 3\mathsf{iw})} + \text{OUT}, M))$ communication. Recall from Section 1 that Bodlaender [6] and Akatov [3] have proved that similar properties hold for TDs of graphs and HDs of hypergraphs. As we show Log-GTA also recovers and generalizes Bodlaender's result from graphs to hypergraphs. We also prove that a modification of Log-GTA recovers all three results using a single construction.

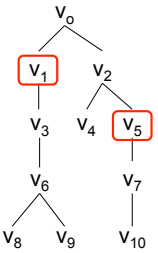


Figure 6: Unique-c-gc vertices.

Here is the outline of this section. In Section 6.1, we describe additional metadata that are assigned to the vertices and edges of T' by Log-GTA, and define a special kind of vertex, called a *unique-child-and-grandchild* vertex. Section 6.2 describes two transformation operations of Log-GTA: *leaf and unique-child-grandchild inactivations*, which we iteratively perform to modify the input D'. Section 6.3 then describes the Log-GTA algorithm. Section 6.4 shows how to modify Log-GTA to recover both Bodlaender's and Akatov's results.

6.1 Extending D′

Log-GTA associates two new labels with the vertices of T':

1. **Active/Inactive**: An 'active' vertex is one that will be modified in later iterations of Log-GTA. Log-GTA starts with all vertices active, and inactivates vertices iteratively until all of them are inactive. At any point, we refer to the subtree of T' consisting of only active vertices as ***active***(T').
2. **Height**: The height of a vertex is its minimum distance from a leaf of the tree. The height of each vertex v is assigned when v is first inactivated, and remains unchanged thereafter (Corollary 2).

In addition, Log-GTA associates a label with each "active" edge $(u, v) \in E(active(T'))$:

- **Common-cover(u, v) (cc(u, v))**: Is a set $S \subseteq E(H)$ such that $(\chi(u) \cap \chi(v)) \subseteq \bigcup_{s \in S} s$. In query terms, cc$(u, v)$ is a set of relations whose attributes cover the common attributes between u and v. In the original $D(T, \chi, \lambda)$, for each (u, v), we set cc(u, v) to any covering subset of size iw. Recall from Section 3 that by definition of iw, such a subset must exist.

Unique-c-gc vertices: Consider a tree T of n vertices with a long depth, say, $\Theta(n)$. Such long depths are caused by long chains of vertices, where vertices in the chain have only a single child. Log-GTA shortens long-depth GHDs by identifying and "branching out" such chains. At a high-level, Log-GTA finds a vertex v with a unique child c (for child), which also has unique child gc (for grandchild), and puts v, c, and gc under a new vertex s. We call vertices like v *unique-c-gc* vertices. Figure 6 shows an example tree and two unique-c-gc vertices v_1 and v_5.

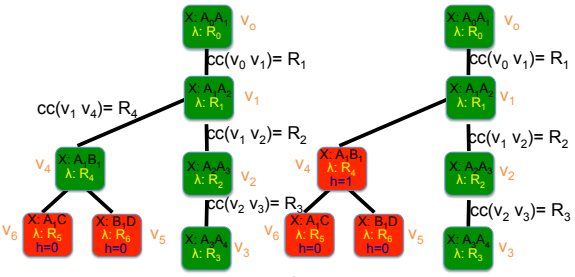


Figure 7: Leaf Inactivation.

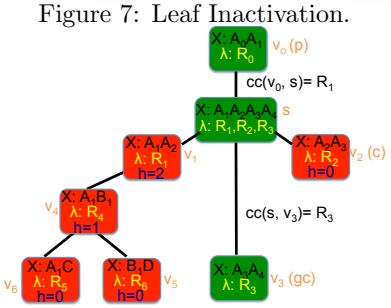


Figure 8: Unique-c-gc Inactivation.

In each iteration, Log-GTA identifies a set of nonadjacent unique-c-gc vertices and leaves of active(T'), and inactivates them (while shortening the chains of unique-c-gc vertices). We next state an important lemma that will help bound the number of iterations of Log-GTA (proved in Appendix A.4):

LEMMA 4. *If a tree has N vertices, then at least $\frac{N}{4}$ vertices are either leaves or non adjacent unique-c-gc vertices.*

6.2 Two Transformation Operations

We next describe the two operations that Log-GTA performs on the nodes of active(T').

Leaf Inactivation: Takes a leaf l of active(T') and (1) sets its label to inactive; and (2) sets height(l) to $\max\{0, \max_c\{\text{height}(c)\}+1\}$, where c is over the "inactive" children of l. $\chi(l)$ and $\lambda(l)$ remain the same. The common-cover between l and l's parent is removed.

Unique-c-gc (And Child) Inactivation: Takes a unique-c-gc vertex u in active(T'), u's parent p (if one exists), u's child c, and u's grandchild gc and does the following:

(1) Creates a new active vertex s, where $\lambda(s) = cc(p,u) \cup cc(u,c) \cup cc(c,gc)$ and $\chi(s) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(c)) \cup (\chi(c) \cap \chi(gc))$.

(2) Inactivates u and c. Similar to leaf inactivation, sets their heights to 0 if they have no inactive children, and one plus the maximum height of their inactive children otherwise.

(3) Removes the edges (p, u) and (u, c) and adds an edge from s to both u and c.

(4) Adds an edge from p to s with $cc(p,s) = cc(p,u)$ and s to gc with $cc(s,gc) = cc(c,gc)$.

Figure 7 shows the effect of Leaf Inactivation on vertex v_4 of an extended GHD. In the figure, green and red indicate that the vertex is active or inactive, respectively. The attributes of each R_i are the χ values on the nodes that R_i is assigned to. Figure 8 shows the effect of Unique-c-gc Inactivation on a unique-c-gc vertex v_1 from Figure 7. We next state a key lemma about these two operations:

LEMMA 5. *Assume an extended GHD $D'(T', \chi', \lambda')$ with active/inactive labels on $V(T')$, and common cover labels on $E(T')$ initially satisfies the following six properties:*

1. *active(T') is a tree.*
2. *The subtree rooted at each inactive vertex v contains only inactive vertices.*
3. *The height of each inactive vertex v is v's correct height in T'.*
4. *$|cc(u, v)| \leq \mathsf{iw}$ between any two active vertices u and v and does indeed cover the shared attributes of u and v.*
5. *$|\chi(u) \cap \chi(v)| \leq k$ between any two active vertices u and v.*
6. *D' is a GHD with width at most $\max(\mathsf{w}, 3\mathsf{iw})$.*

Then performing any sequence of leaf and unique-c-gc inactivations maintains these six properties.

The proof of this lemma is long, and given in Appendix A.5. We next state an immediate corollary to Lemma 5.

COROLLARY 2. *Let $D(T, \chi, \lambda)$ be a GHD with width w, intersection width iw, and treewidth tw. Consider extending D to GHD $D'(T', \chi', \lambda')$ with active/inactive labels, common-covers, and heights as described in Section 6.1, and then applying any sequence of leaf and unique-c-gc inactivations on D'. Then the resulting D' is a GHD with width at most $\max(\mathsf{w}, 3\mathsf{iw})$ and treewidth at most $3\mathsf{tw}+2$, where the height of each inactive vertex v is v's actual height in T'.*

PROOF. The proof that D' is a GHD with width $\max(\mathsf{w}, 3\mathsf{iw})$ is immediate from Lemma 5. We next prove that its treewidth is also $3k+2$. Observe that a series of leaf and unique-c-gc inactivations only introduces new s vertices through unique-c-gc inactivations. The size of $\chi(s)$ for any s is at most $3k$ because $\chi(s)$ is the union of three sets, each with size at most k by assumption. Recall that by definition $\mathsf{tw} = \max_{t \in V(T)}\{|\chi(t)|\} - 1$, which is at least $k-1$. Therefore $3k$ is at most $3\mathsf{tw}+3$, which implies that the treewidth of D' is at most $3\mathsf{tw}+2$. □

6.3 Log-GTA

Finally, we present our Log-GTA algorithm. Log-GTA takes a GHD D and extends it into a D' as described in Section 6.1. Then, Log-GTA iteratively inactivates a set of active leaves L' and nonadjacent unique-c-gc vertices U' (along with the children of U'), which constitute at least $\frac{1}{4}$ fraction of the remaining active vertices in T', until all vertices are inactive. Figure 9 shows the pseudocode of Log-GTA. Figure 10 shows a simulation of Log-GTA on the width-2 and intersection width-1 GHD of the triangle-chain query, TC_{15}, from Table 1, which has depth 6. In the figure, Log-GTA

Input: GHD $D(T, \chi, \lambda)$ for hypergraph H
Extend D into $D'(T', \chi', \lambda')$:
Mark each vertex active
Assign each vertex null heights
For each edge $e = (u, v)$ set $\text{cc}(u, v) = \lambda(v)$
while(there are inactive nodes in T')
Select at least $\frac{1}{4}$ of the active vertices that are either leaves L' or non-adjacent unique-c-gc vertices U'
Inactivate each $l \in L'$
Inactivate each $u \in U'$ and the child of u
return D'

Figure 9: Log-GTA.

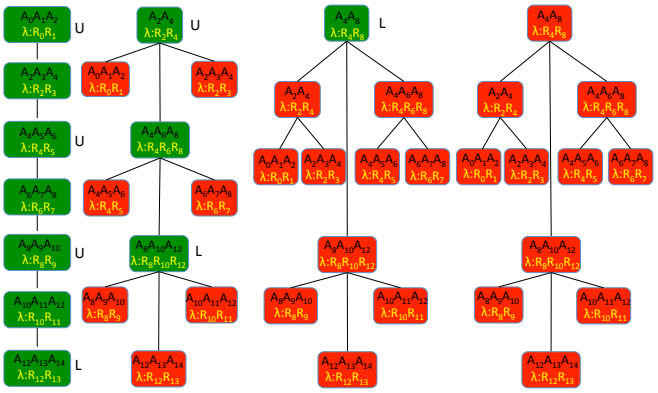


Figure 10: Log-GTA simulation.

produces a width-3 GHD with depth 2. We label the selected leaves and unique-c-gc vertices with L and U respectively, and omit the common-cover labels.

We next state two lemmas about Log-GTA and then prove our second main result.

LEMMA 6. *Log-GTA takes* $O(\log(|V(T)|))$ *iterations.*

PROOF. Observe that both leaf inactivation and unique-c-gc inactivation decrease the number of active vertices in T' by 1. Therefore, by Lemma 4, in each iteration the number of active vertices decreases by a factor of $\frac{1}{4}$. Initially there are $|V(T)|$ active vertices, so the algorithm terminates in $O(\log(|V(T)|)$ iterations. □

LEMMA 7. *The height of each inactive vertex v is at most the iteration number at which v was inactivated.*

PROOF. By Corollary 2, the heights assigned to vertices are their correct heights in the final GHD returned. Moreover the height numbers start at 0 in the first iteration and increase by at most one in each iteration, therefore the height numbers assigned in iteration i are less than i, completing the proof. □

THEOREM 4 **(Second Main Result)**. *Given any GHD $D(T, \chi, \lambda)$ with width* w*, intersection width* iw*, and treewidth* tw*, we can construct a GHD $D'(T', \chi', \lambda')$ where width $w' \leq \max(\mathsf{w}, 3\mathsf{iw})$, treewidth $\mathsf{tw}' \leq 3\mathsf{tw} + 2$, $\text{depth}(T') = \min\{\text{depth}(T), O(\log(|V(T)|))\}$, and $|V(T')| \leq 2|V(T)|$.*

PROOF. By Corollary 2 the width of D' is at most $\max(\mathsf{w}, 3\mathsf{iw})$, and treewidth at most $3\mathsf{tw} + 2$. By Lemmas 5, 6 and 7, the height of each vertex v is v's true height in the graph and is at most the maximum iteration number, which is $O(\log(|V(T)|))$. Therefore, the depth of T' is $O(\log(|V(T)|))$. Also, the leaf and unique-c-gc inactivation operations never increase the depth of the tree, justifying that the depth of the final tree is $\min\{\text{depth}(T), O(\log(|V(T)|))\}$. Finally, Log-GTA increases the number of vertices by one for each unique-c-gc inactivation. Since Log-GTA can make at most $|V(T)|$ unique-c-g-c inactivations, $|V(T')| \leq 2|V(T)|$. □

COROLLARY 3. *Given a hypergraph H with n hyperedges, width* w*, intersection width* iw*, and treewidth* tw*, we can construct a* $\log(n)$ *depth GHD of H with width at most* $\max(\mathsf{w}, 3\mathsf{iw})$ *and treewidth at most* $3\mathsf{tw} + 2$.

Corollary 3 generalizes Bodlaender's result about TDs of graphs to hypergraphs and shows that a similar (and stronger) property also holds for GHDs. The Log-GTA construction we presented in this section does not recover Akatov's result. However, in Section 6.4, we show that a modification of Log-GTA also recovers Akatov's result, thereby providing a single construction that recovers Bodlander's result about TDs, Akatov's result about HDs and (a weaker version) of our new result about GHDs.

Theorem 3 along with Theorem 4 and Lemma 1 imply the following result:

THEOREM 5. *Any query Q with width* w *can be executed in* $O(\log(n))$ *rounds and* $O(n\text{B}(\text{IN}^{\max(\mathsf{w},3\mathsf{iw})} + \text{OUT}, M))$ *communication.*

6.4 Log-depth TDs, GHDs, and HDs

In order to recover both Akatov's and Bodlaender's results, we make two changes to the unique-c-gc inactivation operation. The leaf inactivation operation and the remainder of the algorithm remains exactly the same. We call this algorithm Log-GTA′.

1. In Log-GTA, the common-covers are essentially λ values that are assigned to each active edge of the input D'. Instead, just as is done to the vertices of GHDs, we assign both a λ values and χ values to each active edge (u, v) of D'. To help differentiate these values from λ and χ values assigned to vertices, we will denote them in capital letters, as $\Lambda(u, v)$ and $X(u, v)$. Initially, for each edge (u, v), $\Lambda(u, v) = \lambda(v)$, and $X(u, v) = \chi(v)$, i.e., copies of the child node v's labels.
2. In each unique-c-gc inactivation, the new vertex s is assigned $\lambda(s) = \Lambda(p, u) \cup \Lambda(u, c) \cup \Lambda(c, gc)$, $\chi(s) = X(p, u) \cup X(u, c) \cup X(c, gc)$. The new edge (p, s) is assigned $\Lambda(p, s) = \Lambda(p, u)$, and $X(p, s) = X(p, u)$. Finally, the new edge (s, gc) is assigned $\Lambda(s, gc) = \Lambda(c, gc)$, and $X(s, gc) = X(c, gc)$.

We next state a theorem whose proof is given in Appendix A.6.

THEOREM 6. *Given any GHD $D(T, \chi, \lambda)$ with width* w*, and treewidth* tw*, Log-GTA′ constructs a GHD*

$D'(T', \chi', \lambda')$ *where width* $\mathsf{w}' \leq 3\mathsf{w}$*, treewidth* $\mathsf{tw}' \leq 3\mathsf{tw}+2$*,* $\text{depth}(T') = \min\{\text{depth}(T), O(\log(|V(T)|))\}$*, and* $|V(T')| \leq 2|V(T)|$*. In addition, if the initial GHD is an HD, then* D' *is also an HD.*

Consider a hypergraph H with n hyperedges and (generalized hypertree) width w, hypertree width hw, and treewidth tw. Three corollaries to Theorem 6, and Lemmas 6 and 7, that state that Log-GTA (and Log-GTA$'$) take a logarithmic number of iterations are:

1. A GHD of H with $O(\log(n))$-depth and width $\leq 3\mathsf{w}$ exist.
2. An HD of H with $O(\log(n))$-depth and width $\leq 3\mathsf{hw}$ exist.
3. A TD of H with $O(\log(n))$-depth and treewidth $\leq 3\mathsf{tw}+2$ exist.

Therefore, using Log-GTA$'$, we can recover and generalize Bodlaender's result to hypergraphs, recover Akatov's result, and recover a weaker version of our result that $O(\log(n))$-depth and $\max(\mathsf{w}, 3\mathsf{iw})$ GHDs of hypergraphs exist.

7. C-GTA

We describe another GHD transformation algorithm called C-GTA, (for **C**onstant-depth **G**HD **T**ransformation **A**lgorithm). C-GTA takes a width-w GHD D with n vertices and transforms it into a GHD D' of width-$2\mathsf{w}$ and $\leq \frac{15n}{16}$ vertices. Therefore, it can potentially shorten the depths of $\Theta(n)$-depth GHDs by a constant fraction.

C-GTA is based on the following observation. For any two nodes $t_1, t_2 \in V(T)$, we can "merge" them by replacing them with a new node $t \in V(T)$ and setting $\chi(t) = \chi(t_1) \cup \chi(t_2)$, $\lambda(t) = \lambda(t_1) \cup \lambda(t_2)$ and setting the neighbors of t in T to be the union of neighbors of t_1 and t_2. As long as t_1 and t_2 were either neighbors, or both leaves, T remains a valid GHD tree after the merge operation. C-GTA operates as follows:

1. For each node u that has an even number of leaves as children, divide u's leaves into pairs and merge each pair.
2. For each node u that has an odd number of leaves as children, divide the leaves into pairs and merge them, and merge the remaining leaf with u.
3. For each vertex u that has a unique child c, if c has an even number of leaf children, then merge u and c.

If T has L leaves and non-adjacent U unique-c-gc nodes, then the above procedure removes at least $\frac{\max(L,U)}{2}$ nodes from T. We next state a combinatorial lemma to bound this quantity, which is proved in Appendix A.4.

LEMMA 8. *Suppose a tree has N nodes, L of which are leaves, and U of which are unique-child nodes. Then $4L + U \geq N + 2$.*

By Lemma 8, $\max(L, U)$ is at least $n/8$. Therefore, the resulting tree T' has at most $15n/16$ nodes, and width $\leq 2\mathsf{w}$. We can use this operation repeatedly to reduce the number of vertices while increasing width. We can then apply Log-GTA to get the following theorem:

THEOREM 7. *For any query Q with a width-w, intersection width-iw GHD $D = (T, \chi, \lambda)$, for any i, there exists a GHD $D' = (T', \chi', \lambda')$ with width $\leq 2^i.\max(\mathsf{w}, 3\mathsf{iw})$ and depth $\leq \log(|V(T')|) \leq \log((\frac{15}{16})^i n)\ |V(T)|)$.*

Thus we can further trade off communication by constructing even lower depth trees than a single invocation of Log-GTA.

8. DISCUSSION ON SKEW

Skew in a database input refers to variation in the frequency of different attribute values. A relation without skew would be one without heavy hitter values. All of our results in the paper hold under any amount of skew. If we assume that there are no heavy hitter values in the input and any of the intermediate results generated by GYM, then our results improve both in terms of communication and the number of rounds:

Improvement on Communication: When joining two tables with common attributes, if the input is skew-free, then the join becomes embarrassingly parallelizable and we can simply hash on their common attribute, and perform the join with a communication cost of $O(\text{IN} + \text{OUT})$, irrespective of the machine size M. This implies that GYM's cost improves from $O(n\text{B}(\text{IN}^{\mathsf{w}} + \text{OUT}, M))$ to $O(n(\text{IN}^{\mathsf{w}} + \text{OUT}))$.

Improvement on Number of Rounds: If we assume there are no heavy hitters, we can also join multiple tables at once, as opposed to joining two tables at a time, which is the joining strategy in GYM. When there is no skew, we can process each node R in two rounds: We first join R with each of its children S_i by hashing R and S_i on their common attributes. This generates k intermediate tables T_i. Then we join all T_i in another round by hashing each table on the attributes of R (which each T_i also contains). The same optimization can be done for the semijoin phase, decreasing the number of rounds from $O(d + \log(n))$ to $O(d)$. For example, in absence of skew, we could perform a join on the star query in $O(1)$ rounds.

9. CONCLUSIONS AND FUTURE WORK

We have described a multiround join algorithm GYM, which is a distributed and generalized version of Yannakakis's algorithm. GYM shows that unlike previously thought, Yannakakis's algorithm can be highly parallelized. We have also shown that by using GYM as a primitive and proving different properties of depths and widths of GHDs of queries, we can trade off communication against number of rounds of computations. We believe our approach of discovering such tradeoffs by only proving different combinatorial properties of GHDs and not by designing new distributed algorithms is a promising direction for future work. The theory on GHDs is very rich and there have been many studies focusing on different notions of widths of GHDs. GYM also raises the question of how to construct GHDs with different depths, as depth determines the number of rounds in our context. As a first step, in Appendix C, we describe an algorithm for constructing GHDs with minimum possible depths for acyclic queries.

10. REFERENCES

[1] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. In *VLDB*, 2013.

[2] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE TKDE*, 2011.

[3] Dmitri Akatov. *Exploiting Parallelism in Decomposition Methods for Constraint Satisfaction*. PhD thesis, University of Oxford, 2010.

[4] Apache Hadoop. http://hadoop.apache.org/.

[5] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. In *PODS*, 2013.

[6] Hans L. Bodlaender. NC-Algorithms for Graphs with Small Treewidth. In *Graph-Theoretic Concepts in Computer Science*, 1988.

[7] C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. *TCS*, 2000.

[8] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 2000.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] Arnaud Durand and Etienne Grandjean. The complexity of acyclic conjunctive queries revisited. *CoRR*, abs/cs/0605008, 2006.

[11] G. Gottlob, N. Leone, and F. Scarcello. Advanced Parallel Algorithms for Acyclic Conjunctive Queries. Technical report, Vienna University of Technology, 1998.

[12] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG*, 2005.

[13] G. Gottlob, N. Leone, and F. Scarcello. On Tractable Queries and Constraints. In *DEXA*, 1999.

[14] G. Gottlob, N. Leone, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. *J. ACM*, 2001.

[15] D. Halperin, V. Teixeira de Almeida, L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria Big Data Management Service. In *SIGMOD*, 2014.

[16] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*. 2011.

[17] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *International Conference on Cloud Computing Technology and Science*, 2010.

[18] M. Joglekar and C. Ré. It's all a matter of degree: Using degree information to optimize multiway joins. In *ICDT (to appear)*, 2016.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[20] Neil Robertson and P. D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-width. *Journal of Algorithms*, 1986.

[21] Spark SQL. https://spark.apache.org/sql/.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *VLDB*, 2009.

[23] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, August 1990.

[24] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, 1981.

[25] C.T. Yu and M. Z. Ozsoyoglu. An Algorithm for Tree-Query Membership of a Distributed Query. In *COMPSAC*, 1979.

[26] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

APPENDIX

A. PROOFS OF LEMMAS AND THEOREMS

A.1 Lemma 1

Call a GHD $D = (T, \chi, \lambda)$ *minimal* if for any nodes $u, v \in V(T)$, neither of the sets $\chi(u)$ and $\chi(v)$ is a subset of the other. If $\chi(u) \subset \chi(v)$, then we could simply merge nodes u and v and get another GHD for the same query. Thus if D is not minimal, we can make it minimal by merging some of its nodes iteratively, without increasing its depth or width. We now prove that for any minimal GHD $D = (T, \chi, \lambda)$, if D is a GHD for a query Q having n relations, then $|V(T)| \leq n$. We use induction on $|V(T)|$. Base Case: $|V(T)| = 1 \leq n$, since the query being covered by a non-empty GHD must have at least one relation. Inductive Step: Assume that for all minimal GHDs with $|V(T)| \leq k-1$, any query that they cover must have at least $|V(T)|$ relations. Now consider a GHD $D = (T, \chi, \lambda)$ with $|V(T)| = k$. Let l be a leaf node of tree T. Because D is minimal, $\chi(l)$ contains at least one attribute a that is not contained in $\chi(u)$ for any other $u \in V(T)$. Query Q must have at least one relation R that contains a, and this relation R can only lie in $\lambda(l)$. Now consider the GHD $D_2 = (T_2, \chi_2, \lambda_2)$ obtained by deleting l from T, and query Q_2 obtained by deleting all relations from $\lambda(l)$. Since we deleted at least one relation from Q, Q_2 has $\leq n-1$ relations. Then D_2 is a minimal GHD for Q_2, and by the inductive hypothesis $|V(T_2)| \leq n-1$. And since $|V(T)| = |V(T_2)| + 1$, we have $|V(T)| \leq n$ as required. Therefore we can take any D and make it minimal without affecting its depth or width and get a D' that as at most n vertices.

A.2 Relations Without λ Assignments

In our GYM algorithm, we assume that each relation R_j in the join belongs to $\lambda(v)$ for some v in the GHD D. This may not be the case. We now describe how a simple modification of GYM can be used to handle such R_j's.

From the definition of GHDs, for each R_j in the join, there must exist a $v \in D$ such that attributes$(R_j) \subseteq \chi(v)$. For each R_j let node(j) denote some v such that attributes$(R_j) \subseteq \chi(v)$. Now we modify GYM by adding the following lines after the first IDB_v computation (line 4). (For each $v \in D$ (in parallel):)

- For each j such that node$(j) = v$, compute $IDB_{vj} = IDB_v \bowtie R_j$ (in parallel for all j).
- Compute $IDB_v = \bowtie_{j:\text{node}(j)=v} IDB_{vj}$

Thus, we modify the IDB_v's to ensure that each R_j has been joined with at least one of them. This ensures that the final join of the IDB_v's equals the join of all R's.

Now we show that both of the above steps can be performed in $O(1)$ rounds, with communication cost $O(|V(T)|B(\text{IN}^{\mathsf{w}}, M))$.

- The join $IDB_{vj} = IDB_v \bowtie R_j$ is actually just a semijoin, since the attributes of IDB_v are a superset of the attributes of R_j. Moreover, the sizes of IDB_v and R_j are both $\leq \text{IN}^{\mathsf{w}}$. Thus by Lemma 2, we can perform the semijoin in $O(1)$ rounds at communication cost $O(|V(T)|\frac{(\text{IN}^{\mathsf{w}})^2}{M})$.
- $IDB_v = \bowtie_{j:\text{node}(j)=v} IDB_{vj}$ is simply an intersection of the IDB_{vj}'s, since they all have the same attributes. Moreover, the size of the IDB_{vj}'s is bounded by IN^{w}. This intersection can be performed as follows. We take a hash function that maps tuples of the IDB_{vj}'s to $\text{IN}^{2\mathsf{w}}$ buckets. We have one processor per bucket. Then each tuple of each IDB_{vj}s is sent to the processor corresponding to its hash value. Since the number of hash buckets is the square of the number of tuples per relation, each processor gets $O(1)$ tuples from each relation with high probability. Then each processor can locally perform the intersection, determining which tuples were recived from each of the IDB_{vj}'s. This intersection takes 1 round and has communication cost $O(|V(T)|\text{IN}^{\mathsf{w}})$.

Once the modified IDB's are computed, the rest of the GYM algorithm remains the same. The size of the new IDB's is still bounded by IN^{w}, and the output of their join equals the output of the join of all R's.

A.3 Example Execution of GYM

We now describe how to compute an example query with GYM. Consider the following chain query C_{16}: $R_0(A_0, A_1) \bowtie R_1(A_1, A_2) \bowtie ... \bowtie R_{15}(A_{15}, A_{16})$. Figure 11 shows a width-3 GHD of this query. GYM on this GHD would first compute the IDBs in each vertex of Figure 11. The materialized GHD, shown in Figure 11, is now a width-1 GHD over the IDBs and therefore the join over the IDBs is acyclic. Then the algorithm simply executes DYM-d on the GHD of Figure 11 to compute the final output. Let c be the (constant) number rounds to process the semijoin of two relations. Overall the algorithm would take $12c + 6$ rounds and $O(\text{B}(\text{IN}^3 + \text{OUT}, M))$ communication cost. For comparison, Figure 11 shows a width-1 GHD of the original chain query. Executing GYM directly on this GHD would take $32c + 16$ rounds and $O(\text{B}(\text{IN} + \text{OUT}, M))$ communication cost.

A.4 Lemmas 4 and 8

We first prove Lemma 8 and then prove Lemma 4. First we state and prove another lemma that we will use in the proof of Lemma 8.

LEMMA 9. *If a tree has U unique-c-gc nodes, we can select at least $\lceil \frac{U}{2} \rceil$ of them that are not adjacent to each other.*

PROOF. Partition the U unique-c-gc vertices into disjoint chains; some or all of the chains may be of length two. We can select the first, third, fifth, and so on, of any chain, and thus we select at least $\lceil \frac{U}{2} \rceil$ nodes. □

We next prove Lemma 8. The proof is an induction on the height h of the tree. BASIS: If h=0, then the root is the only node in the tree and is a leaf. Therefore, $4L+U = 4 \geq 1+2 = 3$. If $h = 1$, then the tree is a root plus $N-1$ children of the root, all of which are leaves. Thus, $L = N-1$, and $U = 0$. We must verify $4(N-1)+0 \geq N+2$, or $3N \geq 6$. Since N is necessarily larger than 2 for any tree of height at least 1, we may conclude the bases. INDUCTION: Now, assume $h \geq 1$. There are three cases to consider: *Case 1*: The root has a single child c and c has a single child gc. Then the root is a unique-c-gc node and the tree rooted at c has L leaves, $U-1$ unique-c-gc nodes, and a total of $N-1$ nodes. By the induction hypothesis, $4L + U - 1 \geq (N-1) + 2$, or $4L + U \geq N + 2$, which completes the induction in this case. *Case 2*: The root has a single child, which has $k \geq 2$ children $c_1, ..., c_k$. Let the subtree rooted at c_i have L_i leaves, U_i unique-child nodes, and N_i nodes. By the inductive hypothesis, $4L_i + U_i \geq N_i + 2$. Summing over all i we get $4L + U \geq (N-2) + 2k$. Since $k \geq 2$, we conclude $4L + U \geq N + 2$, which completes the induction in this case. *Case 3*: The root has $k \geq 2$ children $c_1, ..., c_k$. Similarly, if the subtree rooted at c_i has L_i leaves, U_i unique-child nodes, and N_i nodes and we sum over all i, we get $4L + U \geq (N-1) + 2k$. Since $k \geq 2$, we again conclude that $4L + U \geq N + 2$, which completes the proof.

Using Lemmas 9 and 8, we can now prove Lemma 4. Let the tree have N nodes, L leaves, and U unique-c-gc nodes. Lemma 8 says that $4L + U \geq N + 2$. Suppose first that $U = 0$. Then since we can select all leaves, and $4L \geq N + 2$, we can surely select at least $N/4$ nodes. Now, suppose $U \geq 1$, then by Lemma 9, we can select at least $U/2$ of the unique-child nodes. Since we may also select all leaves, and $L + U/2 \geq L + U/4 \geq N/4 + 2/4$, the theorem holds in both cases.

A.5 Lemma 5

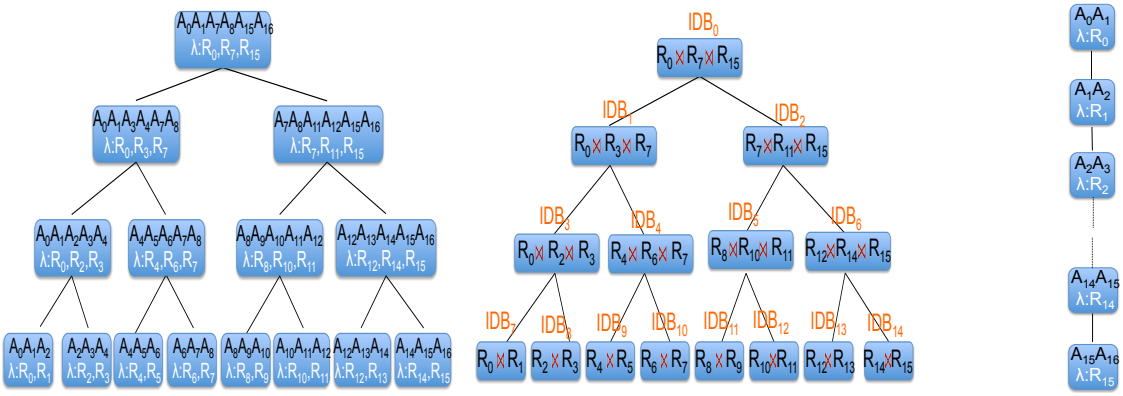


Figure 11: GHDs of C_{16} at various steps of GYM(Log-GTA).

Let $D'(T', \chi', \lambda')$ be a GHD that satisfies these six properties. First, consider inactivating an active leaf l of D'.

1. For property (1), we observe that inactivating l essentially removes a leaf of active(T'), so active(T') remains a tree after the operation.
2. For property (2), we only need to consider the subtree S_l rooted at l. Observe that none of l's children can be active, since this would contradict that l is a leaf of active(T'). In addition, none of l's other descendants can be active because then the subtree rooted at one of l's inactive children would contain an active vertex. This would contradict the assumption that initially all subtrees rooted at inactive vertices contained only inactive vertices.
3. For property (3), notice that the height that is assigned to l is 0 if it has no children, which is its correct height in T'. Otherwise, l's height is one plus the maximum of the heights of l's children, which is also its correct height in T' since all of l's children are inactive and have correct heights by assumption.
4. Properties (4), (5), and (6) hold trivially as leaf inactivation does not affect the common-covers, χ, and λ values and by assumption their properties hold in D$'$.

Now let's consider a unique-c-gc inactivation operation.

1. Property (1) holds because by definition u has one active child c and c also has one active child gc. So u and c are part of a chain of active(T'). We effectively merge u and c together into another active vertex s on this chain without affecting the acyclicity or connectedness of active(T'). Notice that we also add two edges from s to u and v but u and v are inactive.
2. For property (2) observe that the only two subtrees we need to consider are the subtrees rooted at u and c, which we call S_u and S_c, respectively. Notice that all of the edges that go down the tree from u and c after removing (u, c) and (c, gc) were to inactive vertices. Therefore, by the same argument we did for leaf elimination, both S_u and S_c have to consist of only inactive vertices.
3. We assign heights to u and c in the same way as we assigned the height of an inactivated leaf. The exact same argument we made for leaf elimination proves that u and c get assigned their correct heights in T'.
4. We need to consider two common covers: $cc(p, s)$, which is assigned $cc(p, u)$ and $cc(s, gc)$, which is assigned $cc(c, gc)$. The sizes of $cc(p, s)$ and $cc(s, gc)$ are at most iw because the sizes of $cc(p, u)$ and $cc(c, gc)$ are at most iw initially by assumption. We next prove that $cc(p, s)$ indeed covers the common attributes between p and s. The proof for $cc(s, gc)$ is similar and omitted. Notice that since $\chi(s) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(c)) \cup (\chi(c) \cap \chi(gc))$, $\chi(s) \cap \chi(p)$ is exactly equal to $\chi(p) \cap \chi(u)$. This follows from the following observation that p cannot share an attribute with c (or gc), say A_i, that it does not share with u, as this would contradict that the subtree containing A_i in D' is connected (and therefore contradicting that D' is a GHD). By assumption $cc(p, u)$ covers $\chi(p) \cap \chi(u)$. Therefore $cc(p, s)$, which includes $cc(p, u)$, covers $\chi(p) \cap \chi(s)$.
5. For property (5), we only need to look at $|\chi(p) \cap \chi(s)|$ and $|\chi(s) \cap \chi(gc)|$. We argued in the proof of property (4) that $\chi(p) \cap \chi(s) = \chi(p) \cap \chi(u)$, which by assumption has a size of at most k. Similarly, $\chi(s) \cap \chi(gc)$ is exactly equal to $\chi(c) \cap \chi(gc)$, since gc cannot share an attribute with u or p that it does not already with c. Again, by assumption $|\chi(c) \cap \chi(gc)|$ is at most k.
6. For property (6), we need to prove that the three properties of GHDs hold and also verify that the width of the modified D' is at most $\max(\mathsf{w}, 3\mathsf{iw})$.

   - **1st property of GHDs:** The addition of s with two edges to u and c cannot create a cycle or disconnect T', and therefore T' is still a tree.
   - **2nd property of GHDs:** We need to verify that for each vertex v, $\chi(v) \subseteq \cup\lambda(v)$. The unique-c-gc inactivation only inserts the vertex s, and by assumption $\chi(s)$ is the union of three intersections, each of which is covered (respectively) by the three common-covers that comprise $\lambda(s)$.
   - **Width of the modified GHD:** Again by assumption, the sizes of each common cover in $\lambda(s)$

is at most iw, therefore $|\lambda(s)|$ is at most $3\mathsf{iw}$, showing that the width of GHD is still at most $\max(\mathsf{w}, 3\mathsf{iw})$.

- **3rd property of GHDs:** We need to verify that for each attribute X, the vertices that contain X must be connected. It is enough to verify that all attributes among p, s, u, c, and gc are locally connected, since other parts of T' remain unchanged. We need to consider all possible breaks in connectedness between p, u, c, and gc introduced by the insertion of s. The proof of each combination is the same. We only show the proof for attributes between p and gc. Consider any attribute $X \in \chi(p) \cap \chi(gc)$. Then, since the initial D' was a valid GHD, X must have been in $\chi(u)$ (and also $\chi(c)$). Then $\chi(s)$ also includes X because $\chi(s)$ includes $\chi(p) \cap \chi(u)$, proving that the vertices of X are locally connected among p, s, u, c, and gc.

A.6 Theorem 6

We start by proving that a slight modification of Lemma 5 holds about a series of leaf inactivation and the modified unique-c-gc inactivation (modified-unique-c-gc inactivation hereon) operations applied to a GHD D that has been extended with the new Λ and X values on the edges described in Section 6.4.

LEMMA 10. *Assume an extended GHD $D'(T', \chi', \lambda')$ of width w, active/inactive labels on $V(T')$, and common cover labels on $E(T')$ initially satisfies the following six properties:*

1. *The active(T') is a tree.*
2. *The subtree rooted at each inactive vertex v contains only inactive vertices.*
3. *The height of each inactive vertex v is v's correct height in T'.*
4. *For any two adjacent active vertices u and v: (a) $|\Lambda(u, v)| \leq w$; (b) $\Lambda(u, v) \supseteq X(u, v)$; (c) $X(u, v) \supseteq \chi(u) \cap \chi(v)$; and (d) $X(u, v) \subseteq \chi(v)$.*
5. *$|\chi(u) \cap \chi(v)| \leq k$ between any two active vertices u and v.*
6. *D' is a GHD with width at most $3\mathsf{w}$.*

Then performing any sequence of leaf and the modified-unique-c-gc inactivations maintains these six properties. In addition, if the initial GHD is an HD, i.e., satisfies the descendant property (recall Section 3), then the resulting D' is also an HD.

PROOF. We first prove that leaf and unique-c-gc operations maintain the six property for GHDs. We then prove that if D' is an HD, then HD's descendant property is also maintained:

GHD Properties:

The proof that inactivating a leaf vertex maintains these properties except property (4) is the same as the proof provided for leaf inactivation in Lemma 5. Leaf inactivation also maintains property (4) because it does not affect the Λ and X values on the edges or vertices. The proof that modified-unique-c-gc inactivation maintasin properties (1), (2), and (3) are the same as the proof provided for leaf inactivation in Lemma 5. Below we prove that modified-unique-c-gc inactivation also maintains properties (4), (5), and (6).

4. We need to consider two new active edges (p, s) and (s, gc). We provide the proof for (p, s). The proof for (s, gc) is similar and omitted.
   (a) $\Lambda(p, s) = \Lambda(p, u)$, which by assumption has size at most w.
   (b) $\Lambda(p, s) = \Lambda(p, u) \supseteq X(p, u) = X(p, s)$.
   (c) We prove that $X(p, s) \supseteq \chi(p) \cap \chi(s)$. Notice that $\chi(s) = X(p, u) \cup X(u, c) \cup X(c, gc)$ and by assumption $X(p, u) \subseteq \chi(u)$, and $X(u, c) \subseteq \chi(c)$, and $X(c, gc) \subseteq \chi(gc)$. Since p cannot share an attribute with c and gc that it does not already share with u, $\chi(p) \cap \chi(s) \subseteq \chi(p) \cap \chi(u)$ and by assumption $X(p, s) = X(p, u) \supseteq \chi(p) \cap \chi(u)$.
   (d) $X(p, s) \subseteq \chi(s)$ because by construction $X(p, s) = X(p, u)$ and $\chi(s) = X(p, u) \cup X(u, c) \cup X(c, gc)$.
5. For property (5), we only need to look at $|\chi(p) \cap \chi(s)|$ and $|\chi(s) \cap \chi(gc)|$. We argued in the proof of property (4-c) that $\chi(p) \cap \chi(s) \subseteq \chi(p) \cap \chi(u)$, which by assumption has a size of at most k. The proof that $|\chi(s) \cap \chi(gc)| \leq k$ is similar and omitted.
6. For property (6), we need to prove that the three properties of GHDs hold and also verify that the width of the modified D' is at most $3\mathsf{w}$.
   - **1st property of GHDs:** We proved in the proof of Lemma 5 that T' is still a tree.
   - **2nd property of GHDs:** We only need to prove that $\chi(s) \subseteq \cup\lambda(s)$. $\chi(s) = X(p, u) \cup X(u, c) \cup X(c, gc)$, each of these three components are by assumption (4-b) covered (respectively) by the $\Lambda(u, c)$, $\Lambda(u, c)$, and $\Lambda(c, gc)$, which comprise $\lambda(s)$.
   - **Width of the modified GHD:** By assumption (4-a), the sizes of $\Lambda(u, c)$, $\Lambda(u, c)$, and $\Lambda(c, gc)$ are all at most w. Therefore $|\lambda(s)|$ is at most $3\mathsf{w}$, showing that the width of GHD is still at most $3\mathsf{w}$.
   - **3rd property of GHDs:** Similar to our proof of Lemma 5, we need to prove that the modified-unique-c-gc inactivation does not locally disconnect any p, s, u, c, and gc. In addition, now we also need to prove that s does not share any attribute with any other vertex in the tree, that it does not already share with p, u, c, or gc. This is because the values of $\chi(s)$ are now assigned through the X values on the edges, and not the χ values of p, u, c, and gc.
     Similar to our proof of Lemma 5, we again have to consider all possible breaks in connectedness between p, u, c, and gc introduced by the insertion of s. We only show the proof for attributes between p and gc. Consider any attribute $X \in \chi(p) \cap \chi(gc)$. Then, since the initial D' was a valid GHD, X must have been in $\chi(u)$. Then $\chi(s)$ also includes X because $\chi(s)$ by construction contains $X(p, u)$, which by assumption (4-c) contains $\chi(p) \cap \chi(u)$.
     Now assume for the purpose of contradiction that $\chi(s)$ contains an attribute A with a vertex z, and A is not in $\chi(p)$, $\chi(u)$, $\chi(c)$, and $\chi(gc)$. Since

$\chi(s) = X(p,u) \cup X(u,c) \cup X(c,gc)$, and therefore by assumption (4-d) $\chi(s) \subseteq \chi(u) \cup \chi(c) \cup \chi(gc)$, contradicting the assumption that A is not in $\chi(u)$, $\chi(c)$, or $\chi(gc)$, and completing our proof.

HD's Descendant Property:

Leaf inactivation does not modify the χ andr λ assignments, so trivially maintains the descendant property. For modified-unique-c-gc inactivation, we need to prove that the property holds for p, s, u and c:

- u and c: The property holds as u and c's λ assignments remain the same and T_u and T_c can only get smaller.
- p: The only new vertex in T_p is s and $\chi(s) = X(p,u) \cup X(u,c) \cup X(c,gc)$. Recall that we proved in the proof of the 3rd property of GHDs above that $X(p,u) \cup X(u,c) \cup X(c,gc) \subset \chi(u) \cup \chi(c) \cup \chi(gc)$, implying that any attribute in s were already part of T_p.
- s: For s we first need to prove a lemma about the relations in $\Lambda(u,v)$, which we call the ***edge descendant property***.

  LEMMA 11. *Assume the following property holds in the initial GHD D': $\forall e \in \Lambda(u,v)$, $(attr(e) \cap T_v) \subseteq X(u,v)$, then this property is maintained through a series of leaf and modified-unique-c-gc inactivations.*

  PROOF. We only need to consider modified-unique-c-gc inactivation. We need to consider that the property holds for (p,s) and (s,gc). We only provide the proof for (p,s). Consider any edge $e \in \Lambda(p,s) = \Lambda(p,u)$. Notice that $T_s = T_u \cup \chi(s)$. Recall that we proved above that $\chi(s) \subseteq \chi(u) \cup \chi(c) \cup \chi(gc)$. Therefore $T_s = T_u$. Since by assumption $attr(e) \cap T_u \subseteq X(p,u)$, and $X(p,s) = X(p,u)$, $attr(e) \cap T_s \subseteq X(p,s) = X(p,u)$. □

  Now, consider any $e \in \lambda(s)$. e can come from $\lambda(p,u)$, $\lambda(u,c)$, or $\lambda(c,gc)$. We only prove the case when $e \in \lambda(p,u)$. Then we proved in Lemma 11 that $e \cup T_s = T_u \subseteq X(p,u) \subseteq \chi(s)$, completing the proof.

B. IMPROVING AKATOV'S AND BODLAENDER'S RESULTS

We can also improve Akatov's and Bodlaender's results if we modify Log-GTA and use similar notions of intersection width that we used for GHDs.

B.1 Improving Bodlaender's Result

Let the ***tree intersection width*** of a TD to be the maximum size of the number of attributes shared between two adjacent vertices in the TD. Further, let the tree intersection width of a hypergraph H with treewidth tw be the minimum tree intersection width of any of its tw-treewidth TDs. For example, the GHD we showed for TC_n from Figure 2c is also a TD, and has treewidth 2 and tree intersection width of 1.

Let D be a TD with treewidth tw and tree intersection width of tiw. Notice that by definition tiw is at most $\mathsf{tw}+1$. Then using Log-GTA, we can actually improve Bodlaender's result from $3\mathsf{tw}+2$ to $\max(\mathsf{tw}, 3\mathsf{tiw}-1)$. We note that we do not modify Log-GTA. We only modify our analysis using the notion of tree intersection width to get this strictly better result.

We start with a TD of H with tree intersection width tiw (one must exist by definition). Observe that the number of shared attributes between any two vertices during Log-GTA is always at most tiw. To see that this property is maintained throughout Log-GTA, we only need to prove that $|\chi(p) \cap \chi(s)| \leq \mathsf{tiw}$ and $|\chi(s) \cap \chi(gc)| \leq \mathsf{tiw}$. We showed in our proof of Lemma 5 (property 4 of unique-c-gc inactivation case), that $\chi(p) \cap \chi(s) = \chi(p) \cap \chi(u)$, which by assumption is at most tiw. The situation is similar for $\chi(s) \cap \chi(gc)$. Therefore the $|\chi(s)|$, which is the union of three intersections, is at most $3\mathsf{tiw}$. At the end of the transformation, all vertices that have never been an s vertex during the transformation have at most $\mathsf{tw}+1$ attributes, and those that have been s vertices have $3\mathsf{tiw}$ attributes. Therefore the treewidth of the final D' is $\max(\mathsf{tw}, 3\mathsf{tiw}-1)$. For example, we can check that Log-GTA's transformation of TC_{15}, shown in Figure 10, yields a TD with treewidth of $\max(2, 3-1) = 2$, instead of 8.

B.2 Improving Akatov's Result

Similar to our definition of intersection width, let the ***hypertree intersection width*** of a HD $D = (T, \chi, \lambda)$ be defined as follows: For any adjacent vertices $u, v \in V(T)$, let $\mathsf{iw}(u,v)$ denote the size of the smallest set $S \subseteq E(H)$ such that $\chi(u) \cap \chi(v) \subseteq \bigcup_{s \in S} s$. In other words, $\mathsf{iw}(u,v)$ is the size of the smallest set of relations whose attributes cover the common attributes between u and v. The hypertree intersection width hiw of an HD is the maximum $\mathsf{iw}(u,v)$ over all adjacent $u, v \in V(T)$. For example, the GHD we showed for TC_{15} from Figure 2c is also an HD, and has width 2 and hypertree intersection width 1.

To improve Akatov's result, we need to modify the Log-GTA algorithm. Let D be an HD with width w and hypertree intersection width of hiw Now consider making the following two modifications to Log-GTA: (1) When extending the initial HD D to D', we assign the common covers of the edges to be any covering subset of size hiw; and (2) During unique-c-gc inactivation operation, instead of assigning $\chi(s)$ to be $(\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(c)) \cup (\chi(c) \cap \chi(gc))$, we assign $\chi(s) = attr(\lambda(p,u)) \cup attr(\lambda(u,c)) \cup attr(\lambda(c,gc))$. We omit the proof, but it can be shown that this modified Log-GTA returns a $O(\log(n))$-depth D' with width $\max(\mathsf{w}, 3\mathsf{hiw})$, improving Akatov's result from $3\mathsf{w}$ to $\max(\mathsf{w}, 3\mathsf{hiw})$. For example, in the case of TC_{15}, shown in Figure 10, the returned HD would have a width of 3, instead of Akatov's result of 6.

C. MINIMIZING THE PARSE TREE DEPTH FOR ACYCLIC QUERIES

We now show how to construct a width-1 GHD of an acyclic query with the minimum depth possible. We will refer to width-1 GHDs by their conventional names,

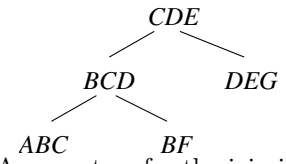


Figure 12: A parse tree for the join in Example 4

parse trees. Parse trees for acyclic queries are constructed from their *GYO* reductions, which we review momentarily. Our algorithm is a set of heuristics to use during the GYO reduction that guarantee the generation of a parse tree of minimum depth for the query.

C.1 GYO Reduction

We say that a hyperedge e of a hypergraph is *consumed* by a hyperedge e' if e contains nodes that are either unique to e (i.e., not in any other hyperedge in the hypergraph) or are shared with e'. In this case, we call edge e an *ear*. A single step of a GYO reduction can replace hypergraph $G(V, E)$ by hypergraph $G'(V', E')$ if there is a hyperedge $e \in E$ that is consumed by another hyperedge e'. In this case $E' = E - \{e\}$ and V' is V minus the nodes that are contained only in e. We say that a hypergraph (and the corresponding join) is *acyclic* if a (multistep) GYO reduction [25] results in a hypergraph with one hyperedge. Given an acyclic hypergraph, we can form a parse tree representing the GYO reduction as follows. The edges of the hypergraph are the nodes of the parse tree. The root is the one edge that is not consumed, and for all other edges e of the hypergraph, its parent in the tree is the hyperedge that consumes e.

Example 7. *Recall the join from Example 4. The GYO reduction we do (and that corresponds to the parse tree in Figure 12) is the following:*

1. *$R_1(A, B, C)$ is consumed by $R_3(B, C, D)$ because A appears only in $R_1(A, B, C)$, while B and C appear in $R_3(B, C, D)$. Hence, in the parse tree $R_3(B, C, D)$ is the parent of $R_1(A, B, C)$.*
2. *For the next step of the GYO reduction, we are left with a hypergraph that has four hyperedges (since $R_1(A, B, C)$ is deleted in the first step). In the new hypergraph, $R_2(B, F)$ is consumed by $R_3(B, C, D)$; hence we delete $R_2(B, F)$. In the parse tree, $R_3(B, C, D)$ is the parent of $R_2(B, F)$.*
3. *Now, $R_3(B, C, D)$ is consumed by $R_4(C, D, E)$. Note that after the first two steps, B is only in the schema of R_3, since R_1 and R_2 have been deleted from the hypergraph.*
4. *In the last step, $R_5(D, E, G)$ is consumed by $R_4(C, D, E)$.*

At this point, we are left with a single hyperedge which represents $R_4(C, D, E)$. We conclude that the hypergraph is acyclic, and its parse tree is complete.

C.2 Minimum Depth Parse Trees

We begin with the observation that certain subgroups of relations in the join may affect largely the depth of the parse tree. The following example makes the point.

Example 8. *Consider*

$$R_1(X, X_1), R_2(X, X_2), R_3(X, X_3), R_4(X, X_4), R_5(X, X_4, Y)$$

Each atom except the last one is an ear and can be consumed by R_5. We could build a parse tree of depth five, where, say, R_2 consumes R_1, then R_3 consumes R_2, and so on. But we can also build a parse tree of depth 2, where R_5 consumes each of the other hyperedges. Moreover if we choose the first (long) parse tree then, if we have i such relations (instead of 5), we will need $O(\log(i)\epsilon)$ rounds, whereas if we choose the short parse tree, then we will need a constant number of rounds.

Here is an algorithm to obtain a parse tree of minimum depth of a connected acyclic hypergraph H: Stage I.

1. H' is H.
2. Find set E_R of all ears in H'. For each ear in E_R we do:
   We define all potential parents of E (i.e., edges that consume E) among all edges of H'.[1]
3. We repeat using the hypergraph H' which is previous H' with E_R deleted.

Thus in the first stage, for each hyperedge we have a list of potential parents. Alternatively, we may imagine that we have built a directed graph G_0 with nodes representing the relations and an edge (u, v_i) showing a potential parent of u. From G_0 we extract a subgraph which is a spanning tree of minimum depth as follows (we will explain shortly why it works): Stage II.

1. Choose as root of the parse tree either a hyperedge with no potential parent or a hyperedge for which each entering edge is on a cycle. Break ties arbitrarily.
2. For all hyperedges with potential parent the root, assign the root as their parent and declare them *parented*.
3. If a hyperedge has at least one parented hyperedge in its list of potential parents, then choose the potential parent closer to the root as its parent and declare it parented.

The following is a critical observation:

- First observe that a minimum depth parse tree has depth at least as large as the number i of iterations in Stage I of the algorithm. We will prove in the following that we construct a tree of depth at most $i + 1$. We will also prove that when the depth is $i + 1$ then it is optimal.

The observation that the minimum depth parse tree has depth at least as large as number of iterations i needs a proof which is as follows. Let $T_{\min}$ be a minimum depth parse tree. Suppose the depth of $T_{\min}$ is greater than 2 [2] Then the set E_R in the first iteration contains all leaves of $T_{\min}$. If the depth is greater than 3, then

[1] According to Lemma 15, the consumed set of E is the same for all parse trees, hence, we assign as potential parents all edges of H' that contain this set.
[2] Depth 2 means a root and its children.

the E_R in the second iteration contains all parents of the leaves of $T_{\min}$. This goes on up until the last iteration, where we may have many ears with potential parents each other. This last iteration may create two levels in $T_{\min}$[3]. If however the last iteration has an E_R that contains only one ear then this ear is the root of a tree of depth equal to i and this is of minimum depth. (We explain more about these two last levels later). We need a series of lemmas (LCA below is short for "lowest common ancestor"):

LEMMA 12. *Let T be a parse tree for connected acyclic hypergraph H. If an attribute A[4] appears in more than one node of T then it appears in their LCA too.*

This lemma is a straightforward consequence of the Bernstein-Goodman result that the nodes of the parse tree that contain attribute A have to be connected. The following is an immediate consequence of Lemma 12

LEMMA 13. *If a node has many potential parents in G_0 then, on any parse tree of H, the LCA of all the potential parents is also a potential parent.*

LEMMA 14. *Any parse tree T of acyclic hypergraph H is a spanning tree*[5] *of G_0 and vice versa.*

PROOF. Any edge of T is also an edge of G_0. Since tree T contains all nodes of G_0, it is a spanning tree of G_0. □

LEMMA 15. *If a relation R has more than one potential parent, then the consumed set (i.e., the attributes that belong both to the parent and R) is the same for all potential parents.*

PROOF. By definition when R becomes an ear, then its attributes are partitioned in two (disjoint) sets: those that belong only to R (denote this subset of attributes by A_1) and those that belong to both R and its parent. Suppose there are two potential parents P_1 and P_2 and suppose there are two different A_1 and A_1' for each potential parent. Then the difference of $A_1 - A_1'$ is a non-empty set A_d. This means that the attributes in A_d have the property: a) they belong to R, b) they do not belong to P_1 and c) they belong to P_2. This is a contradiction because, if so, P_1 does not consume R. □

LEMMA 16. *Suppose there is a path in G_0 from P_2 to P_1 and a path from R to P_2. Then, if an attribute A of R appears in P_1 it appears in P_2 too.*

PROOF. If A of R appears in P_1, this means that A was consumed in each step of the chain from P_1 to R. Hence, it appears in all the relations in this chain. □

[3]We explain more about these two last levels later.

[4]A is a node of H but we will use the term "attribute" to avoid confusion, since the parse tree has nodes too that correspond to relations. We will use the term "node" for nodes of the parse tree.

[5]with the root having ingoing edges and all edges go from child to parent

LEMMA 17. *If edge (u, v) in G_0 is not on a cycle, then u is not an ancestor of v on any parse tree of H.*

PROOF. If u is an ancestor of v in some parse tree, then there is a path in G_0 from v to u. This path together with edge (u, v) form a cycle. □

First we begin to argue about the root and the reason the algorithm in stage II works when it picks the root. According to Lemma 17, the following two cases are left for the root: either a) it is a single node with no potential parent (hence this is the root of the tree) or b) there is in G_0 a strongly connected component whose nodes have only incoming edges from nodes outside this component. The following two observations conclude the case for the root:

1. As a consequence of the Lemmas 15 and 16, when there is a cycle C, all hyperedges/nodes on the cycle share the same set (call it A_C) of attributes. I.e., each hyperedge contains A_C and some other attributes that belong only to this hyperedge (among the hyperedges in the cycle). Moreover, A_C is the consumed set of each node on the cycle.
2. If for a hyperedge E, each entering edge (in G_0) is on a cycle, then whichever (among the hyperedges on this cycle) we choose for root the depth of the tree is not affected. This is shown by observing that a) all the other nodes on the cycle can be children of the root and b) if a node of the cycle is a potential parent of node u, then the root also is a potential parent of u.

An observation of independent interest is put in footnote here[6]. Now we need to prove that the rest of the algorithm builds a tree of minimum depth. When the algorithm builds a tree of depth one or two then the argument about how we choose the root proves that the algorithm correctly constructs the minimum-depth parse tree. We have the following cases for G_0 for tree built by the algorithm which is of depth one or two:

1. G_0 is a single node. This is trivial.
2. G_0 is a single strongly connected component. In this case, as we argued above, all the consumed sets are the equal to each other and we get to choose arbitrarily one node of G_0 for root and the rest are children of the root.
3. G_0 consists of: a "main" strongly connected component and several other strongly connected components whose nodes are consumed by any node of the main strongly connected component (hence they are consumed also by the root which is chosen arbitrarily from the main component). So, in this case we choose the root from the main component (the rest of nodes in this component are children of the root) so that it is the node which is connected

[6]When there is a cycle in G_0 then, there is also a cycle of length two (it is the one among any two hyperedges/nodes of any cycle – because the consumed sets are the same along a cycle, thus we can build a smaller cycle out of any nodes of a larger cycle.).

to the other strongly connected component. We break ties arbitrarily. Here is that the depth can be one more than the number of iterations of stage I of the algorithm but it is easy to see that it is optimal.

The above are the only cases where ears are consumed by the main component of G_0. In all other cases, new ears are consumed by descendants of the root. For the general case we postpone choosing the root till each potential root (ie., node that belongs to the higher component) has built the subtree rooted at it. Then we choose as root the one with the deepest subtree. Again, the depth can be one more that the number of iterations of stage I of the algorithm but it is optimal because each subtree rooted in one of the potential roots is of optimal depth.