GYM: A Multiround Distributed Join Algorithm

Foto N. Afrati[1], Manas R. Joglekar[2], Christopher M. Re[3], Semih Salihoglu[4], and Jeffrey D. Ullman[5]

1 **National Technical University of Athens, Athens, Greece**
afrati@gmail.com
2 **Stanford University, Stanford, CA, USA**
manasrj@stanford.edu
3 **Stanford University, Stanford, CA, USA**
chrismre@cs.stanford.edu
4 **University of Waterloo, Waterloo, ON, Canada**
semih.salihoglu@uwaterloo.ca
5 **Stanford University, Stanford, CA, USA**
ullman@gmail.com

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 several rounds for the problem of computing the equijoin of n relations. Given any query Q with width w, *intersection width* iw, input size IN, output size OUT, and a cluster of machines with $M = \Omega(\text{IN}^{\frac{1}{\epsilon}})$ memory available per machine, where $\epsilon > 1$ and $w \geq 1$ are constants, 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 with high probability.
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 with high probability.

Intersection width is a new notion we introduce for queries and generalized hypertree decompositions (GHDs) of queries that captures how connected the adjacent 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 $O(\mathsf{d} + \log(n))$ 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 lower depths, demonstrating other tradeoffs one can make between communication and the number of rounds.

1998 ACM Subject Classification H.2.4 [Systems] Query Processing

Keywords and phrases Joins, Yannakakis, Bulk Synchronous Processing, GHDs

Digital Object Identifier 10.4230/LIPIcs.ICDT.2017.4

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 [24], Hive [28], SparkSQL [27], and Myria [18]. These systems are conceptually based on Valiant's *bulk synchronous parallel* (BSP) computational model [29]. 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 [19, 20].

This paper studies the problem of evaluating an equijoin query Q in multiple rounds of computation in a distributed cluster. We restrict ourselves to queries that are full, i.e. do not contain projections, but queries can contain self-joins. We let n be the number of relations, IN the input size, OUT the output size of Q, and $M = o(\text{IN})$ the memory available per machine in the cluster. Memory sizes of the machines intuitively capture different parallelism levels: when memory sizes are smaller, we need a larger number of machines to evaluate the join, which increases parallelism. We assume throughout the paper that $M = \Omega(\text{IN}^{\frac{1}{\epsilon}})$ for some constant $\epsilon > 1$. For practical values of input and memory 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 [3, 6]. For example, for the *chain query*, $C_n = R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie \cdots \bowtie R_n(A_{n-1}, A_n)$, the lower bound on the communication cost of any one-round algorithm is $\geq (\frac{\text{IN}}{M})^{n/4}$. For example, if the input is one petabyte, i.e., IN=10^{15}, even when we have machines with ten gigabytes of memory, i.e., M=10^{10}, the communication cost of any one-round algorithm to evaluate the C_{16} query is 100000 petabytes. Moreover, this lower bound holds even when the query output is known to be small, e.g., OUT $= O(\text{IN})$, and the input has no skew [6], implying that designing multiround algorithms is the only way to compute such joins more efficiently.

Second, the cost of running a new round of computation has decreased from several minutes in the early systems (e.g., Hadoop [5]) to milliseconds in some recent systems (e.g., Spark [31]), making it practical to run algorithms that consist of a large number of rounds. Although multiround algorithms are becoming commonplace, how much algorithms can benefit from running more rounds is not well understood. In this paper, we answer this question for equijoin queries.

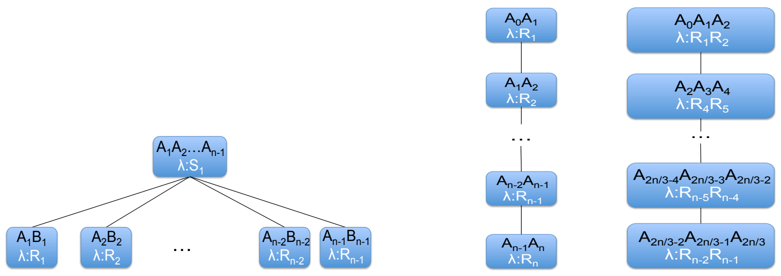
We describe a multiround algorithm, called *GYM*, for **G**eneralized **Y**annakakis in **M**apReduce (Sections 4-5), which is a distributed and generalized version of Yannakakis's algorithm for acyclic queries [30]. The performance of GYM depends on two important structural properties of the input query: *depths* and *widths* of its *generalized hypertree decompositions* (GHDs). We then present two algorithms, *Log-GTA* (Section 6) and *C-GTA* (Section 7), 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. In the remainder of this section, we give an overview of our results.

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. In this paper, we focus on bounded width queries in this paper, i.e. those whose widths are a constant.

Table 1 Example Queries S_n, C_n, and TC_n.

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



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

Figure 1 Example GHDs.

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

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 $O(n\frac{(\text{IN}^{\text{w}}+\text{OUT})^2}{M})$ communication cost using GYM.

Table 1 lists three example queries and their widths w, minimum depths of their width-w GHDs, and intersection widths (explained momentarily). Figure 1 shows example GHDs of these queries. The labels on the vertices of the GHDs in Figure 1 are the λ and χ values, following the notation in Section 3.

▶ **Example 2.** The star query S_n is an acyclic query. As shown in Figure 1, 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 3.** The chain query C_n is also an acyclic query. Figure 1 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 $O(n\frac{(\text{IN}+\text{OUT})^2}{M})$.

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. We discuss the improvements to our results when the inputs to queries are skew-free in the longer version of our paper [1].

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 more communication cost by proving a combinatorial lemma about GHDs, which may be of independent interest to readers:

▶ **Main Result 4.** *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. We present an algorithm *Log-GTA*, for ***Log****-depth* ***G****HD* ***T****ransformation* ***A****lgorithm*, to achieve our second main result. Using Log-GTA, we can tradeoff rounds and communication for queries with high depth GHDs as follows:

▶ **Example 5.** The TC_n query has a width of 2 and intersection width of 1. Figure 1c shows an example width-2 GHD D of TC_n which has a depth of $\frac{n}{3}$-1. One option to evaluate TC_n is to use D directly. On D, GYM will execute $\Theta(n)$ rounds and have a communication cost of $O(n\frac{(\mathrm{IN}^2+\mathrm{OUT})^2}{M})$. Another option is to 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 $O(n\frac{(\mathrm{IN}^3+\mathrm{OUT})^2}{M})$.

We end this section by discussing two interesting consequences of Log-GTA and GYM.

Log-depth Decompositions. GHDs [12] 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) [26], and *hypertree decompositions* (HDs) [17]. Two previous results by Bodlaender [7] and Akatov [4] have proved the existence of log-depth TDs of hypergraphs 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. Interestingly, neither of these results (including ours) imply each other. However, we show in the longer version of our paper [1] that Log-GTA also recovers Bodlaender's result. That is, Log-GTA also transforms a given TD into log-depth one with thrice its treewidth. In addition, we show that a modification of Log-GTA recovers both Akatov's and Bodlaender's results and a weaker version of our second main result. We also 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. 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, 15, 16], acyclic queries have been described as being polynomial-time sequentially solvable by Yannakakis's algorithm, but highly parallelizable by the *ACQ* algorithm [14], where parallelizability refers to being in the complexity class NC. By constructing log-depth GHDs of queries and simulating GYM in the PRAM model, we show that unlike previously thought, with simple modifications Yannakakis's algorithm can run in logarithmic rounds, implying that bounded-width queries are in the complexity class

NC, recovering a result that was prove using the ACQ algorithm [14]. We note that BSP models can also simulate PRAM and a comparison of GYM against a distributed simulation of ACQ, which we call *ACQ-MR*, is given in the longer version of our paper [1]. We also show in the longer version of our paper [1] that GYM can match ACQ-MR's performance on every query using an appropriate GHD for the query and on some queries GYM strictly outperforms ACQ-MR.

2 Related Work

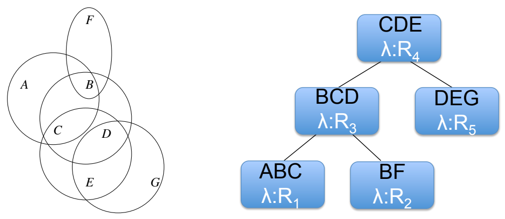
We provide a brief overview of related work here. A comprehensive coverage of related work is in the full version of this paper [1].

Shares. Shares [3] is the optimal one-round join algorithm. References [2] and [6] have shown that for every query Q, and value of M, and skew level, the Shares algorithm can be configured to incur the lowest possible communication cost among one-round algorithms that send at most M input tuples to each machine. However, as we discussed in Section 1, for some queries, these costs can be prohibitively expensive.

Other Distributed Join Algorithms. Reference [6] studies multiround distributed join algorithms in the *Massively Parallel Computing* (MPC) model. Reference [6] proves lower bounds on the number of rounds required to compute queries when $M = \frac{\mathrm{IN}}{p^{1-\epsilon'}}$, where p is the number of machines, and ϵ' is a constant $\in [0, 1)$ called the *space exponent*. They show that when evaluating a query whose GHDs are of depth d on an arbitrary database, any algorithm with limited memory, where the limitation is defined as space exponent being a constant, will have to run $O(\log(\mathsf{d}))$ rounds. 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*, which represent skew-free inputs. On arbitrary databases, their iterative Shares algorithm can produce intermediate data of size $\mathrm{IN}^{\Theta(n)}$ for any query irrespective of its width. By our second result, GYM evaluates these queries on any database instance in $O(\log(n))$ rounds. So when d i constant but n is unbounded, GYM runs $O(\log(n))$ rounds whereas their lower bound is $O(1)$. However, GYM keeps intermediate relation sizes bounded by $\mathrm{IN}^{\max(w, 3\mathsf{iw})} + \mathrm{OUT}$. On matching databases, their algorithms matches these lower bounds exactly. In the longer version of our paper [1], we show that our GYM algorithm matches these lower bounds exactly using several optimizations.

Reference [22] describes worst case optimal constant-round join algorithms for several classes of conjunctive queries when the frequencies of each value in the attributes of the relations are known. The authors relate the communication cost of algorithms on a query to a structural property of the query called *edge quasi-packing number*, which can be smaller than the width of the query. In contrast, we do not assume any prior knowledge of frequencies.

Generalized Hypertree Decompositions. Structural decomposition methods, such as GHDs [13], query decompositions (QDs) [8], tree decompositions (TDs) [26], and hypertree decompositions (HDs) [17], 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 be smaller than HDs and QDs, giving us stronger results in terms of communication cost.



(a) Hypergraph. **(b)** GHD.

Figure 2 Hypergraph and GHD of Example 6.

3 Preliminaries

We review GHDs, describe our model, and specify the assumptions we make in this paper.

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 6.** 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 2a.

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

- $T(V(T), E(T))$ is a tree;
- $\chi : V(T) \rightarrow 2^{V(H)}$ is a function associating a set of vertices $\chi(t) \subseteq V(H)$ to each vertex t of T;
- $\lambda : V(T) \rightarrow 2^{E(H)}$ is a function associating a set of hyperedges 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.
3. For every $t \in V(T)$, $\chi(t) \subseteq \bigcup \lambda(t)$, i.e., hyperedges of $\lambda(t)$ must "cover" the vertices of $\chi(t)$.

For any $t \in V(T)$, we refer to $\chi(t)$ as the attributes of t and $\lambda(t)$ as the relations on t. A GHD of a join query Q is defined to be a GHD on the hypergraph of Q.

▶ **Example 7.** Figure 2b shows a GHD of the query from Example 6. 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:

- The ***depth*** of a GHD $D = (T, \chi, \lambda)$ is the depth of the tree T.
- The ***width*** of a GHD D is $\max_{t \in V(T)}\{|\lambda(t)|\}$, i.e., the maximum number of relations assigned to any vertex t.

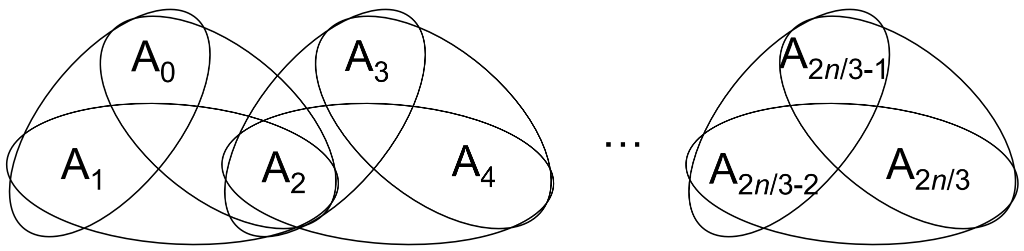


Figure 3 Hypergraph of TC_n.

- 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 1 [8]. 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 $t, t' \in V(T)$, let $\mathsf{iw}(t, t')$ denote the size of the smallest set $S \subseteq E(H)$ such that $\chi(t) \cap \chi(t') \subseteq \bigcup_{s \in S} s$. In other words, $\mathsf{iw}(t, t')$ is the size of the smallest set of relations whose attributes cover the common attributes between t and t'. The intersection width iw of a GHD is the maximum $\mathsf{iw}(t, t')$ over all adjacent $t, t' \in V(T)$.

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

▶ **Example 8.** Consider the TC_n example from Table 1. TC_n is a width-2 query. The hypergraph of TC_n, shown in Figure 3, visually is a chain of triangles, where any two consecutive triangles are connected by a single attribute. Figure 1c 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 common attribute between each triangle can be covered by one relation (e.g., A_2, which is the common attribute between the first two triangles, can be covered by R_2).

In the rest of this paper we restrict ourselves, for simplicity of presentation, to queries whose hypergraphs are connected. All of our results generalize to queries with disconnected hypergraphs. A GHD $D(T, \chi, \lambda)$ of a hypergraph H is called *complete* if each hyperedge $e \in E(H)$ occurs in $\lambda(t)$ of some vertex $t \in V(T)$. That is, each relation is assigned to the λ-label of some vertex $t \in V(T)$. We assume throughout the paper that the GHDs we use are rooted, i.e., one of the vertices (arbitrarily) in T is picked as a root. This ensures that there is a well defined notion of height of vertices and parent-child relationships between the vertices in T. We end this section by stating a lemma about complete GHDs of queries:

▶ **Lemma 9.** *If a query Q has a width-w, intersection width-iw GHD $D = (T, \chi, \lambda)$ of depth d, then Q has a complete GHD $D' = (T', \chi', \lambda')$ with depth $\leq d + 1$, width w, intersection width iw, and $|V(T')| \leq 4n$.*

We prove this lemma in the longer version of our paper [1]. Using this lemma, we will assume w.l.o.g. that the GHDs of queries that we use in our algorithms are complete and have $O(n)$ size.

3.2 MapReduce and Cost Model

Our MapReduce (MR) model is equivalent to the MR model in reference [25] except we use tuples instead of bits as our cost unit. In the tuple-based MR model, the unit of memory and communication cost is a base or intermediate tuple consisting of any set of attributes in Q. There is a set of distributed machines on a networked file system, each with memory $M = o(\text{IN})$.

Map Stage: Each machine, referred to as a *mapper*, reads a set of base or intermediate tuples over any set of attributes in the query from the networked file system. Mappers can send tuples to one or more machines, called *reducers*[1], deterministically or by hashing them on any set of their attributes. We assume that mappers have access to the same random bits and families of universal hash functions.[2] Suppose there are k reducers. Using an appropriate family of hash functions, mappers can hash tuples of an input or intermediate relation R, using any subset of the tuples' attributes, to one of the k reducers. Therefore two tuples with the same attributes that were used in the hashing will go to the same reducer. The total number of tuples received by a reducer from all mappers should not exceed memory size M. Otherwise the computation aborts. We note that we use randomization for load-balancing only. Specifically, GYM might send a machine more than M tuples, exceeding the memory capacity of the machine, resulting in the computation to abort. However, this will happen with exponentially small probability.

Reduce Stage: Each reducer locally performs any computation on the $\leq M$ tuples it receives, produces a set of output tuples, and streams the output tuples 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 the total communication cost and the number of rounds. In the longer version of our paper [1], we compare our model to existing models of modern distributed BSP systems.

3.3 Assumptions

We next specify three assumptions we make throughout the paper.

1. As in many MapReduce and distributed BSP models [6, 11, 21, 23, 25], we constrain M to be $o(\text{IN})$. This ensures that machines cannot store the entire input. Otherwise we can send the entire input to a single machine and incrementally evaluate the join using any binary join plan, without exceeding memory $O(M)$, in a single round and without any communication (except to write the output to the networked file system).
2. We assume $M = \Theta(\text{IN}^{\frac{1}{\epsilon}})$ for some constant $\epsilon > 1$.
3. We assume queries have constant widths, i.e., the term w is a constant ($O(1)$).

As we discuss in Section 5, the complexities of our algorithms become slightly worse when we drop assumptions (2) and (3).

[1] The machines we refer to as reducers are equivalent to *reduce keys* in the original description of MR [9]. These are separate groups of data on which the *reduce()* function is executed in the original system.

[2] Access to these random bits do not require any synchronization. In practice this would achieved by using a pseudorandom number generator with the same seed.

3.4 Basic Relational Operations in MR

We next state four lemmas characterizing the costs of joins, duplicate elimination, semijoins, and intersections in our model.

▶ **Lemma 10.** *Any z relations $R_1, \ldots, R_z$ can be joined in 1 round with $O(\frac{z^z(\Sigma_i|R_i|)^z}{M^{z-1}} + |R_1 \bowtie \cdots \bowtie R_z|)$ communication. When z is a constant, the join can be performed in $O(\frac{(\Sigma_i|R_i|)^z}{M^{z-1}} + |R_1 \bowtie \cdots \bowtie R_z|)$ communication.*

Proof. We perform the join as follows. We divide each R_i in $g_{r_i} = \frac{z|R_i|}{M}$ disjoint groups of size $\frac{M}{z}$ each. Then we use a total of $g_{r_1} \times \cdots \times g_{r_z}$ reducers and map a distinct set of z groups, one from each relation, to each reducer. Each reducer joins its z groups locally. Thus we use $\frac{z^z|R_1|\ldots|R_z|}{M^z}$ reducers and each reducer gets an input equal to M, and the total output size is $|R_1 \bowtie \cdots \bowtie R_z|$. Therefore, the total communication cost is $O(\frac{z^z(\Sigma_i|R_i|)^z}{M^{z-1}} + |R_1 \bowtie \cdots \bowtie R_z|)$. ◀

We note that Lemma 10 holds even if the given relations contain self-joins.

▶ **Lemma 11.** *Let S be a multiset such that each tuple $t \in S$ has at most k duplicates. We can remove the duplicates in S w.h.p. in $O(\log_M(k))$ rounds and $O(\log_M(k)|S|)$ communication.*

Proof. We cannot send the duplicates of each tuple to a separate reducer because k might be greater than M, exceeding the memory of machines. Let h be a hash function mapping the tuples of S into $|S|^2$ buckets randomly, so w.h.p. each bucket $h(i)$ gets $O(1)$ unique tuples. In the first round of duplicate elimination, we use $|S|^2k^2$ reducers indexed with two numbers, $(1,1), \ldots, (1,k^2), \ldots, (|S|^2,1), \ldots, (|S|^2,k^2)$, and each tuple t is mapped to the reducer with the first index $h(t)$ and a uniformly random second index. Therefore, w.h.p., each reducer gets $O(1)$ tuples. Reducers do not perform any computation on their tuples. Note that every duplicate of tuple t is mapped to a reducer with the 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 $O(1)$ unique S tuples to each $h(i)$. In the second round, 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 tuples to the same reducer, and eliminate the duplicates across them.[3] This reduces the number of reducers that can contain duplicates to $\frac{k^2}{\sqrt{M}}$. We then repeat this procedure in parallel for each group of reducers with the same first index until all duplicates within each group are eliminated. In each round, each reducer gets $O(\sqrt{M})$ tuples and outputs $O(1)$ tuples. This computation takes $O(\log_{\sqrt{M}}(k))$ rounds. Since the communication in each round is $|S|$, this computation takes $O(\log_{\sqrt{M}}(k)|S|)$ communication.[4] ◀

▶ **Lemma 12.** *Let $B(X,M) = \frac{X^2}{M}$. Given two relations R and S, the semijoin $S \ltimes R$ can be computed w.h.p. in $O(\log_M(|R|))$ rounds with $O(\log_M(|R|)B(|R|+|S|,M))$ communication. When $M = \Omega((|R|)^{1/\epsilon})$, for some $\epsilon > 1$, the semijoin can be performed in $O(1)$ rounds and $O(B(|R|+|S|,M))$ communication.*

[3] We can also group them into $\frac{M}{c}$ for a constant c that is larger than the (O(1)) tuples any reducer has.
[4] As is standard, we use the term w.h.p. in this paper to refer to probabilities that are exponentially small in the input size IN. The probabilities mentioned in Lemma 11 are exponentially small in the size of S, which as we will see, in Yannakakis's algorithm can be smaller than IN. However when an input relation S on which we perform duplicate elimination is small, we can make this probability exponentially small in IN by simply increasing the number of our buckets to IN^2.

Proof. The first round of the semijoin $S \ltimes R$ is similar to the join. Let $g_r = \frac{2|R|}{M}$ and $g_s = \frac{2|S|}{M}$ be disjoint groups of size $\frac{M}{2}$. Each of the $g_r g_s$ reducers 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 reducers, there may be up to g_r duplicates of each tuple. So the size of the multiset S' with duplicates of S is $g_r|S|$. Using Lemma 11, we can eliminate these tuples w.h.p. in $O(\log_M(g_r))$ rounds and $O(\log_M(g_r)g_r|S|) = O(\frac{\log_M(|R|)|R||S|}{M})$ communication. Together with the costs of the join operation, which takes 1 round and $O(\frac{(|R|+|S|)^2}{M})$ communication, we conclude that the semijoin can be performed w.h.p. in $O(\log_M(|R|))$ rounds and $O(\log_M(|R|)B(|R|+|S|,M))$ communication cost. When $M = \Omega((|R|)^{1/\epsilon})$, the semijoin can be computed in $O(1)$ rounds and $O(B(|R|+|S|,M))$ communication. ◀

▶ **Lemma 13.** *Two relations R and S can be intersected w.h.p in 1 round with $O(|R|+|S|)$ communication.*

Proof. Suppose w.l.o.g. that $|R| > |S|$. We simply hash each tuple of R and S using all of their attributes into $|R|^2$ machines, so w.h.p. each machine gets $O(1)$ tuples from both R and S and performs a local intersection, without exceeding M, and writes the at most $|R|+|S|$ output. ◀

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 $O(n\mathrm{B}(\mathrm{IN}+\mathrm{OUT},M))$ communication cost. In Section 4.3, we reduce the number of rounds to $O(\mathsf{d}+\log(n))$ rounds without affecting the communication cost.

4.1 Serial Yannakakis Algorithm

The serial version of Yannakakis's algorithm takes as input an acyclic query $Q = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, 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. We will refer to relations that are assigned to leaf (non-leaf) vertices in T as *leaf (non-leaf) relations*. Yannakakis's algorithm first eliminates all tuples 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**.

Semijoin Phase: 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* sub-phase, and the executions of step (3) form the *downward* sub-phase. In total, this version of the algorithm performs $2(n-1)$ semijoin operations.

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

An important property of Yannakakis's algorithm is that the semijoin phase removes all of the "dangling" tuples in the input, i.e., those that will not contribute to the final output. This guarantees that the sizes of all intermediate tables during the join phase are no larger than the final output size OUT [30].

4.2 DYM-n

If we simply execute each semijoin and join operation of Yannakakis's algorithm by the algorithms in Lemmas 10 and 12, we get a distributed algorithm which we refer to as ***DYM-n***:

▶ **Theorem 14.** *DYM-n computes every acyclic query Q in $O(n)$ rounds and in $O(n\mathrm{B}(\mathrm{IN}+\mathrm{OUT}, M))$ communication cost.*

Proof. The algorithm executes a total of $2(n-1)$ pairwise semijoins and $n-1$ joins, in a total of $O(n)$ rounds. The largest input to any semijoin operation is the largest relation size, which is at most IN. By Lemma 12, the communication cost of the semijoin phase is $O(n\mathrm{B}(\mathrm{IN}, M))$. In each round of the join phase, the input and outputs are at most the final output size OUT. By Lemma 10, the cost of each round is $O(\frac{\mathrm{OUT}^2}{M}+\mathrm{OUT})$. Therefore, the total cost of both phases is $O(n(\mathrm{B}(\mathrm{IN}, M)+\mathrm{B}(\mathrm{OUT}, M)+\mathrm{OUT}))$, which is $O(n\mathrm{B}(\mathrm{IN}+\mathrm{OUT}, M))$ when $M=O(\mathrm{IN})$, as we assume in this paper (recall Section 3.3). ◀

4.3 DYM-d

DYM-d parallelizes Yannakakis's algorithm further by executing multiple semijoins and joins in parallel, reducing the number of rounds to $O(\mathsf{d}+\log(n))$, where d is the depth of the GHD $D(T, \chi, \lambda)$, without asymptotically affecting DYM-n's communication cost.

Upward Semijoin Sub-phase in $O(\mathsf{d}+log(n))$ Rounds: During the upward semijoin sub-phase, the algorithm from Section 4.1 picks one leaf t that is assigned relation R and processes R by replacing it's parent S with $S \ltimes R$ in $O(1)$ rounds. Instead we can pick and process all leaves in parallel. 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. We will replace step (1) of the algorithm from Section 4.1 with two steps, which will be performed in parallel.

1.1. For each R in L_1 in parallel, replace R's parent S with $S \ltimes R$.
1.2. Divide the leaves in L_2 into disjoint pairs of siblings, and up to one triple of siblings per parent, if there is an odd number of siblings with the same parent. Then in parallel perform the following computation. Suppose R_1 and R_2 form such a pair with parent S. Replace R_1 with $(S \ltimes R_1) \cap (S \ltimes R_2)$ and remove R_2. 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.

▶ **Lemma 15.** *The above procedure runs in $O(\mathsf{d}+\log(n))$ rounds.*

The proof of this lemma is provided in the longer version of our paper [1]. Since we perform $O(n)$ intersection or semijoin operations in total and all of the initial and intermediate relations involved have size at most IN, by Lemmas 12 and 13, the total communication cost of the upward semijoin sub-phase is $O(n\mathrm{B}(\mathrm{IN}, M))$.

Downward Semijoin Sub-phase in $O(\mathsf{d})$ Rounds: Note that in the downward semijoin sub-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 sub-phase in $O(\mathsf{d})$ rounds and in $O(n\mathrm{B}(\mathrm{IN}, M))$ communication.

Join Phase in $O(\mathsf{d} + log(n))$ Rounds: The join phase is similar to the upward semijoin sub-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 $O(\mathsf{d} + \log(n))$. The total communication cost of each pairwise join is $O(n\mathrm{B}(\mathrm{OUT}, M))$, since the intermediate relations being joined are at most as large as OUT. Therefore, both the semijoin and join phases can be performed in $O(\mathsf{d}+\log(n))$ rounds with a total communication cost of $O(n\mathrm{B}(\mathrm{IN} + \mathrm{OUT}, M))$, justifying the following theorem:

▶ **Theorem 16.** *DYM-d evaluates an acyclic query Q in $O(\mathsf{d}+\log(n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}+\mathrm{OUT}, M))$ 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-w, depth-d GHD $D(T, \chi, \lambda)$ of a query Q. By Lemma 9, we assume w.l.o.g. that D is complete. 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 and D is now a width-1 GHD for Q'. Therefore we can directly run DYM-d to compute Q'.

▶ **Theorem 17** (First Main Result). *Given a width-w, depth-d GHD $D(T, \chi, \lambda)$ of a query Q over n relations, GYM executes Q in $O(\mathsf{d} + \log(n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}^{\mathsf{w}} + \mathrm{OUT}, M))$ communication cost.*

Proof. For the materialization stage, for each vertex v of D, joining the w relations inside $\lambda(v)$ takes 1 round and $O(\frac{\mathrm{IN}^{\mathsf{w}}}{M^{w-1}} + |IDB_v|)$ communication cost by Lemma 10. In the worst case when the relations constitute a Cartesian product, $|IDB_v|$ is IN^{w}, so evaluating IDB_v takes $O(\mathrm{IN}^{\mathsf{w}})$ cost. By Lemma 9 there are at most $4n$ vertices in $V(T)$, so the materialization stage takes $O(n\mathrm{IN}^{\mathsf{w}})$ communication cost. Since the size of each IDB_v is at most IN^{w}, executing DYM-d on the IDB_v's takes $O(\mathsf{d} + \log(n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}^{\mathsf{w}} + \mathrm{OUT}, M))$ communication, which dominates the cost of materialization phase, completing the proof. ◀

In the longer version of our paper [1], we present an example execution of GYM on a query. We note that if we drop our assumptions that $M = \Omega(\mathrm{IN}^{\frac{1}{\epsilon}})$ and w is a constant, the number of rounds that GYM takes on a width-w, depth-d GHD increases by a factor of $\log_M(\mathrm{IN}^{\mathsf{w}}) = \mathsf{w}\log_M(\mathrm{IN})$. This is because the semijoin operations on $O(\mathrm{IN}^{\mathsf{w}})$ size inputs will execute $O(w\log_M(\mathrm{IN}))$ rounds instead of $O(1)$. Similarly, the communication cost of GYM will increase by at most a factor of $\max\{\mathsf{w}\log_M(\mathrm{IN}), \mathsf{w}^{\mathsf{w}}\}$. The $\mathsf{w}\log_M(\mathrm{IN})$ and w^{w} factors are due to the communication cost increases in the semijoin (Lemma 12) and join operations (Lemma 10), respectively.

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. By running GYM on D^*, we can execute Q in $O(\log(n))$ rounds with $O(n\mathrm{B}(\mathrm{IN}^{\max(\mathsf{w}, 3\mathsf{iw})} + \mathrm{OUT}, M))$ communication.

6.1 Extending to 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')***. We prove that active(T') is indeed a tree in Lemma 19.
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.

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 at most 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 high depth, say, $\Theta(n)$. Intuitively, such high depths are caused by long chains of vertices, where vertices in the chain have only a single child. Log-GTA reduces the depth of high-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.

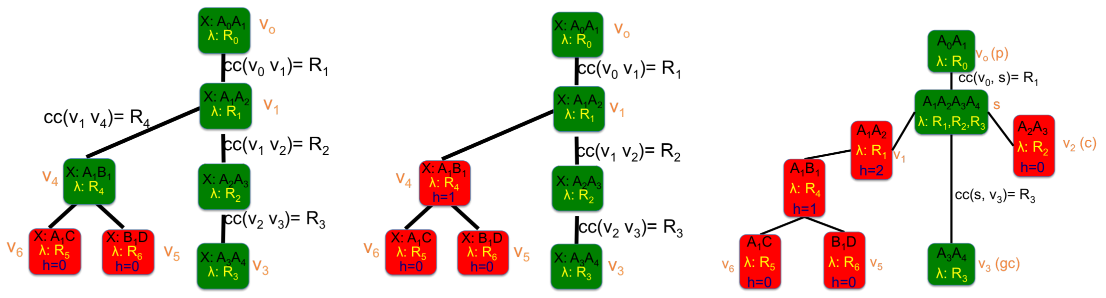
In each iteration, Log-GTA identifies a set of nonadjacent unique-c-gc vertices and leaves of active(T'), and inactivates them (while shortening the chains of unique-c-gc vertices). We next state an important lemma that will help bound the number of iterations of Log-GTA (proved in the longer version of our paper [1]):

▶ **Lemma 18.** *In a tree with N vertices, we can find two sets L' and U' such that $|L'|+|U'| > \lceil \frac{N}{4} \rceil$ and vertices in L' are leaves, and vertices in U' are (1) unique-c-gc vertices; and (2) pairwise non-adjacent.*

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 $\max\{0, \max_c\{\mathrm{height}(c)\} + 1\}$, 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.



(a) Original GHD. **(b)** Leaf Inactivation. **(c)** Unique-c-gc Inactivation.

Figure 4 Effects of leaf inactivation and unique-c-gc inactivation.

Unique-c-gc (And Child) Inactivation: Let u be a unique-c-gc vertex in active(T'). Note that u is not necessarily a unique-c-gc vertex in T'. Let u's parent be p (if one exists), u's child be c, and u's grandchild be gc. Unique-c-gc inactivation 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 4b shows the effect of leaf inactivation on vertex v_4 of the extended GHD in Figure 4a. In the figure, green and red indicate that the vertex is active and inactive, respectively. The attributes of each R_i are the χ values on the nodes that R_i is assigned to. Figure 4c shows the effect of Unique-c-gc Inactivation on a unique-c-gc vertex v_1 from Figure 4b. We next state a key lemma about these two operations:

▶ **Lemma 19.** *Assume that an extended GHD $D'(T', \chi', \lambda')$ of a hypergraph H with active/inactive labels on $V(T')$, and common covers on $E(T')$ initially satisfies the following five 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. *$|cc(u, v)| \leq$ iw between any two active vertices u and v and does indeed cover the shared attributes of u and v.*
5. *D' is a GHD of H with width at most* max(w, 3iw).

Performing any sequence of leaf and unique-c-gc inactivations maintains these five properties.

We prove this lemma in the longer version of our paper [1]. We next state an immediate corollary to Lemma 19.

▶ **Corollary 20.** *Let $D(T, \chi, \lambda)$ be a GHD of a hypergraph H with width* w*, intersection width* iw*. 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(w, 3iw) *and the height of each inactive vertex v is v's actual height in T'.*

```
Input: GHD $D(T, \chi, \lambda)$ for hypergraph $H$
Extend $D$ into $D'(T', \chi', \lambda')$ as described in Section 6.1.
while(there are active 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 $l \in L'$, each $u \in U'$ and the child of $u$
return $D'$
```

Figure 5 Log-GTA.

6.3 Log-GTA

Finally, we present our Log-GTA algorithm. Log-GTA takes a GHD D and extends it into D' by following the procedure 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' by Lemma 18, until all vertices are inactive. Figure 5 shows the pseudocode of Log-GTA. We next state two lemmas about Log-GTA and then prove our second main result.

▶ **Lemma 21.** *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. In each iteration the number of active vertices decreases by a factor of $\frac{1}{4}$. Therefore the algorithm terminates in $O(\log(|V(T)|)$ iterations. ◀

▶ **Lemma 22.** *The height of each inactive vertex v is at most the iteration number at which v was inactivated.*

Proof. By Corollary 20, 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, because in each iteration, the height of each inactivated vertex v is set to the maximum of v's inactive children plus one. Therefore the height numbers assigned in iteration i are less than i, completing the proof. ◀

▶ **Theorem 23** (Second Main Result). *Given any GHD $D(T, \chi, \lambda)$ with width w, intersection width iw, we can construct a GHD $D'(T', \chi', \lambda')$ where width $w' \leq \max(\mathsf{w}, 3\mathsf{iw})$, $\mathrm{depth}(T') = \min\{\mathrm{depth}(T), O(\log(|V(T)|))\}$.*

Proof. By Corollary 20 the width of D' is at most $\max(\mathsf{w}, 3\mathsf{iw})$. By Lemmas 19, 21 and 22, the height of each vertex v is v's true height in the tree 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 $\min\{\mathrm{depth}(T), O(\log(|V(T)|))\}$. ◀

Theorems 17 and 23 and Lemma 9 imply the following two results:

▶ **Corollary 24.** *Given a hypergraph H with n hyperedges, width w, intersection width iw, we can construct a $\log(n)$ depth GHD of H with width at most $\max(\mathsf{w}, 3\mathsf{iw})$.*

▶ **Theorem 25.** *Any query Q with width w can be executed in $O(\log(n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}^{\max(\mathsf{w},3\mathsf{iw})} + \mathrm{OUT}, M))$ communication.*

We note that Corollary 24 shows that, similar to Bodlaender's and Akatov's results about log-depth TDs and HDs, a similar and stronger property also holds for GHDs. In the longer version of our paper [1] we further show: (1) Log-GTA without any modifications recovers Bodlaender's result about TDs; and (2) a modification of Log-GTA recovers Bodlaender's and Akatov's results and a weaker version of our result.

Surprisingly, if we simulate GYM on PRAM using a log-depth GHD generated by Log-GTA, we show that any bounded-width query can be evaluated in logarithmic PRAM steps using polynomial number of processors, i.e., bounded-width queries are in the complexity class NC—a result that was proven by the ACQ algorithm [14]. This is interesting in 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.

7 C-GTA (Constant-depth GHD Transformation Algorithm)

Our C-GTA algorithm is based on the following observation. For any two adjacent nodes $t_1, t_2 \in V(T)$, we can "merge" them and replace them with a new node $t \in V(T)$ and set $\chi(t) = \chi(t_1) \cup \chi(t_2)$, $\lambda(t) = \lambda(t_1) \cup \lambda(t_2)$ and set t's neighbors to the union of neighbors of t_1 and t_2. As long as t_1 and t_2 were either neighbors, or both leaves with the same parent, T remains a valid GHD tree after this 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 a set U of pairwise non-adjacent 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 the longer version of our paper [1].

▶ **Lemma 26.** *Suppose a tree has N nodes, L of which are leaves, and U of which are unique-c-gc nodes. Then $4L + U \geq N + 2$.*

By Lemma 26, $\max(L, U)$ 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 27.** *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 $\leq \log((\frac{15}{16})^i n)$.*

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

8 Conclusions and Future Work

We have 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 using different combinatorial properties of GHDs is a promising direction for future work. An important open area is to explore other GHD construction algorithms that output GHDs with different depths and widths. Specifically, we believe it is plausible that an algorithm can generate polynomially lower depth GHDs with twice their widths (instead of the constant depth reduction of C-GTA). We also plan to investigate the lower bounds on the communication costs of algorithms that run $O(\log(n))$ or $O(n)$ rounds.

Acknowledgements. We would like to thank the anonymous ICDT reviewers whose comments and suggestions about many parts of this paper were critical when preparing the final version of this paper.

References

1 F. Afrati, M. Joglekar, C. Ré, Salihoglu S., and J. D. Ullman. GYM: A Multiround Join Algorithm in MapReduce and Its Analysis. *CoRR*, abs/1410.4156, 2014. URL: `http://arxiv.org/abs/1410.4156`.

2 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.

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

4 D. Akatov. *Exploiting Parallelism in Decomposition Methods for Constraint Satisfaction*. PhD thesis, University of Oxford, 2010.

5 Apache Hadoop. `http://hadoop.apache.org/`.

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

7 H. L. Bodlaender. NC-Algorithms for Graphs with Small Treewidth. In *Graph-Theoretic Concepts in Computer Science*, 1988.

8 C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. *TCS*, 2000.

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

10 A. Durand and E. Grandjean. The Complexity of Acyclic Conjunctive Queries Revisited. *CoRR*, abs/cs/0605008, 2006. URL: `http://arxiv.org/abs/cs/0605008`.

11 M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, Searching, and Simulation in the Mapreduce Framework. In *ISAAC*, 2011.

12 G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. In *J. Comput. Syst. Sci*, 2003.

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

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

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

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

17 G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions and Tractable Queries. In *J. Comput. Syst. Sci.*, 2002.

18 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.

19 S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *New Frontiers in Information and Software as Services*. Springer Berlin Heidelberg, 2011.

20 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.

21 H. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *SODA*, 2010.

22 P. Koutris, P. Beame, and D. Suciu. Worst-Case Optimal Algorithms for Parallel Query Processing. In *ICDT*, 2016.

23 P. Koutris and D. Suciu. Parallel Evaluation of Conjunctive Queries. In *PODS*, 2011.

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

25 A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round Tradeoffs for MapReduce Computations. In *ICS*, 2012.

26 N. Robertson and P. D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-width. *Journal of Algorithms*, 1986.

27 Spark SQL. `https://spark.apache.org/sql/`.

28 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.

29 L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, August 1990.

30 M. Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, 1981.

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