GYM: A Multiround Join Algorithm In MapReduce And Its Analysis

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

Manas Joglekar
Stanford University
California, USA
manasrj@stanford.edu

Christopher Re
Stanford University
California, USA
chrismre@stanford.edu

Semih Salihoglu
Stanford University
California, USA
semih@stanford.edu

Jeffrey Ullman
Stanford University
California, USA
ullman@gmail.com

ABSTRACT

We study the problem of computing the join of n relations in multiple rounds of MapReduce. We introduce a distributed and generalized version of Yannakakis's algorithm, called GYM. GYM takes as input any generalized hypertree decomposition (GHD) of a query of width w and depth d, and computes the query in $O(d+\log(n))$ rounds and $O(n\frac{(\text{IN}^w+\text{OUT})^2}{M})$ communication cost, where M is the memory available per machine in the cluster and IN and OUT are the sizes of input and output of the query, respectively. M is assumed to be $\text{IN}^{\frac{1}{\epsilon}}$, for some constant $\epsilon > 1$. Using GYM we achieve two main results: (1) Every width-w query can be computed in $O(n)$ rounds of MapReduce with $O(n\frac{(\text{IN}^w+\text{OUT})^2}{M})$ cost; (2) Every width-w query can be computed in $O(\log(n))$ rounds of MapReduce with $O(n\frac{(\text{IN}^{3w}+\text{OUT})^2}{M})$ cost. We achieve our second result by showing how to construct a $O(\log(n))$-depth and width-$3w$ GHD of a query of width w. We describe another general technique to construct GHDs with even shorter depth and longer widths, effectively showing a spectrum of tradeoffs one can make between communication and the number of rounds of MapReduce.

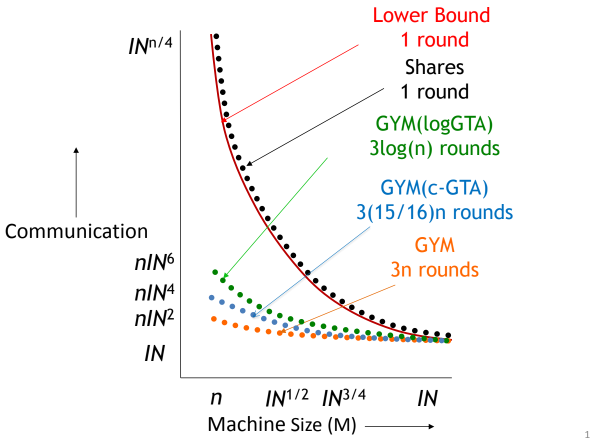


Figure 1: Memory Per Processor vs Total Communication for C_n.

1. INTRODUCTION

The problem of evaluating joins efficiently in distributed environments has gained importance since the advent of Google's MapReduce [11] and the emergence of a series of distributed systems with relational operators, such as Pig [20], Hive [22], SparkSQL [21], and Myria [18]. The costs of join algorithms in such systems can be broken down to: (1) local computation of machines; (2) communication between the machines; and (3) the number of global synchronizations that need to take place between the machines, e.g. the number of rounds of MapReduce jobs that need to be executed. The computation cost is generally subsumed by the communication cost, so we focus on communication cost and the number of rounds.

We consider an equijoin query Q on n relations. Since it is impossible to perform a join using less communication than the sum of the input size, IN, and the output size, OUT, we use those sizes as the parameters for measuring complexity. It has recently been shown that when evaluating joins in a single round of MapReduce, there is a natural tradeoff between the machine size, i.e., the memory available on each machine, and the total communication cost. When machine sizes are smaller, we need a larger number of machines to evaluate the join, which increases parallelism, but also increases the total communication. Figure 1 shows this tradeoff for the *chain query* $C_n = R_1(A_0, A_1) \bowtie R_2(A_1, A_2) \bowtie \ldots \bowtie R_n(A_{n-1}, A_n)$, along with the costs of several algorithms. The red curve in the figure is the lower bound on communication cost of any one-round algorithm for different cluster machine sizes. The black points slightly above the lower bound curve is the performance of the one-round Shares algorithm [5], which matches this lower bound almost optimally for all machine sizes [4, 8]. Interestingly, reference [8] has recently shown that this lower bound is robust; it holds even when the query output is known to be small, e.g., OUT = O(IN), implying that designing multi-round algorithms is the only way to compute joins more efficiently.

We describe a multiround MapReduce 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 (explained momentarily) [25]. The performance of GYM depends on two important structural properties of the input query: *depths* and *widths* of its *generalized hypertree decompositions (GHDs)*. 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. GYM takes as input a GHD of a query Q of width w and depth d, and executes $O(d)$ semijoins and joins in $O(d + \log(n))$ rounds, where M is the memory available per machine in the cluster, and a communication cost that increases as a function of w. Therefore, GYM can be highly efficient on GHDs with low widths and execute for a small number of rounds on GHDs with short depths. We also present two algorithms, *Log-GTA* (Section 6.4) and *C-GTA* (Section 7.1), for constructing GHDs of different depths and widths of a given query, exposing a spectrum of tradeoffs one can make between the number of rounds and communication. The green, blue, and orange curves in Figure 1 show the performance of GYM on three different GHDs for the chain query when OUT is assumed to be $O(\text{IN})$.

We note that all of our results hold under any amount of skew in the input data, i.e., the frequency of attribute values. Specifically, we state our results assuming that there may be heavy hitter values on the columns of the input tables. For example, when joining two tables, $R(A, B)$ and $S(B, C)$, column B might contain a single heavy hitter value b_i. Our results only get better if we assume that the input is skew-free (see Section 7.3).

Our approach to modifying Yannakakis's algorithm to run in logarithmic rounds might surprise database researchers, who 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 [16, 17, 12], acyclic queries have been described as being polynomial-time sequentially solvable by Yannakakis's algorithm, but highly "parallelizable" by the *ACQ* algorithm [13], where parallelizability refers to executing for a small number of PRAM steps. By simulating MapReduce in the PRAM model, our results show that unlike previously thought, we can easily parallelize Yannakakis's algorithm with simple optimizations. Moreover, by proving combinatorial lemmas about GHDs, we can match ACQ's performance on all queries in terms of rounds and also outperform it on certain classes of queries in terms of efficiency. In the remainder of this section, we first summarize our main results and then compare GYM's performance to Shares and ACQ.

1.1 Summary of Main Results

The original algorithm of Yannakakis runs on a single machine. It takes as input a width-1 GHD of an acyclic query involving n relations and always executes a sequence of $\Theta(n)$ semijoins and joins in $\Theta(\text{IN} + \text{OUT})$ time. As we show in Section 4.3, Yannakakis' algorithm can easily be mapped to the MapReduce setting. The resulting algorithm, which we call *DYM* (for **D**istributed **Y**annakakis in **M**apReduce), has a communication cost of $O(n\frac{(\text{IN}+\text{OUT})^2}{M})$ (recall that M is the machine size, i.e., memory available per machine). Throughout the paper, we let $\text{B}(X, M)$ denote the communication cost of a binary join, i.e., join of two relations, where the the total size of the relations is X and machine size is M. We show in Section 4.2 that $\text{B}(X, M) = \frac{X^2}{M}$. The communication cost of DYM can thus be written as $O(n\text{B}(\text{IN} + \text{OUT}, M))$. Throughout the paper, we assume that $M = \text{IN}^{\frac{1}{\epsilon}}$ for some constant $\epsilon > 1$. For practical values of input and machine size, ϵ is a small constant. For instance, if IN is in terabytes and the machine size is in megabytes, then $\epsilon \approx 2$. Even if the machine size is in kilobytes, $\epsilon \approx 4$.

GYM takes as input any GHD of any query. Let D be a width-w, depth-d GHD of a query Q. Our first main result is the following:
Main Result 1: *GYM computes Q in $O(d + \log(n))$ rounds of MapReduce with $O(n\text{B}(\text{IN}^w + \text{OUT}, M))$ communication cost.*
GYM is based on three observations:

1. Each join between two input relations can be executed in one round with a communication cost in $O(B(\text{IN} + \text{OUT}, M)) = O(\frac{(\text{IN}+\text{OUT})^2}{M})$. Each semijoin can be computed in a constant number of rounds with the same cost.
2. We can further parallelize the algorithm from step 1 by executing some of the semijoins or joins in parallel reducing the number of rounds to $O(d+\log(n))$ without affecting the communication cost of the algorithm from step 1.
3. We can generalize the algorithm from step 2 to take as input any GHD D of any (possibly cyclic) query by first running the Shares algorithm on each vertex of D in parallel. This preprocessing step takes a constant number of rounds and at most $O(\text{IN}^w)$ cost and generates a set of acyclic intermediate relations over which the algorithm from step 2 can be run.

We observe that when $M = \text{IN} + \text{OUT}$ for an acyclic query, the entire join can be computed on a single machine, and hence the cost of GYM becomes $\text{IN} + \text{OUT}$, which is same as the cost of running Yanakakkis' algorithm on a single machine. On acyclic queries with constant-depth GHDs, such as the star query (see Section 1.2), GYM executes for only $O(\log(n))$ rounds; while incurring communication cost in $O(n\text{B}(\text{IN} + \text{OUT}, M))$. However there are acyclic queries, such as the chain query (see Section 1.2), whose width-1 GHDs have a depth of $\Theta(n)$. GYM executes such queries in $\Theta(n)$ rounds.

Our second main result shows how to execute such queries by GYM in 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 2: *Let Q be an equijoin query between n relations. Given a width-w, depth-d GHD D of Q, we can construct a GHD D' of Q of depth $O(\log(n))$ and width at most $3w$.*
We describe a GHD transformation algorithm called *Log-GTA* to achieve our second main result. This result implies that by increasing the communication cost from $O(n\text{B}(\text{IN}^w + \text{OUT}, M))$ to $O(n\text{B}(\text{IN}^{3w} + \text{OUT}, M))$, we can decrease the number of rounds from $O(n)$ to $O(\log(n))$ for width-w queries with long depth GHDs. Interestingly, as we discuss momentarily, our second main result recovers the result proven by ACQ [13] that constant-width queries are in the complexity class NC. However, we can also state tighter efficiency results by leveraging existing theory about GHDs (Section 7). We also describe another GHD transformation algorithm called *C-GTA*, using which one can further reduce the depths of the GHDs of queries to $O(\log((\frac{15}{16})^i n))$ at the cost of increasing their widths to $2^i 3w$, exposing a layer of other tradeoffs that are possible between number of rounds and communication.

1.2 GYM, Shares, and ACQ

We next compare GYM to Shares and *ACQ-MR* (explained momentarily). For reference, Tables 1 and 2 compare the performance of GYM to Shares and ACQ-MR on two acyclic queries: (1) the star query of n relations S_n: $S(A_1, ..., A_n) \bowtie R_1(A_1, B_1) \bowtie ... \bowtie R_n(A_n, B_n)$; and (2) the chain query of n relations C_n, defined for Figure 1 earlier.

GYM vs Shares: Shares is a one-round MapReduce algorithm. Shares is parameterized algorithm, whose communication cost is different for different queries and machine sizes. For example, the communication cost of Shares for C_n can be expressed as $O(\frac{\text{IN}^{\frac{n}{4}}}{M^{\frac{n}{2}}}+$

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

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

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

Table 2: Worst-Case Complexity of Algorithms on the Chain Query C_n. D_{C_n} is a $\theta(n)$-depth GHD of C_n.

OUT) [4, 8]. We note that this cost reflects the easiest setting when there is no skew in the input tables; if the input contains skew, the cost goes up to $O(\frac{\mathrm{IN}^{\frac{n}{2}}}{M^{\frac{n}{2}}} + \mathrm{OUT})$ [9]. The computation cost of Shares is trivially lower bounded by its communication cost but can be much larger for some queries. It has been proven that for each parallelism level and skew level, the Shares algorithm can be configured to incur the optimal communication cost possible among one-round algorithms [4, 8]. However, Shares can be prohibitively expensive when computing queries with small outputs at small values of M. Even when M is large, say $\sqrt{\mathrm{IN}}$, the cost of Shares on C_n is exponential in n. However, at the same parallelism levels, the cost of GYM for C_n is $O(n\frac{\mathrm{IN}+\mathrm{OUT}}{\sqrt{\mathrm{IN}}})$. In general, GYM's cost on a width-w query; so is $O(n\mathrm{B}(\mathrm{IN} + \mathrm{OUT}, M))$. Therefore, GYM significantly outperforms Shares in terms of communication cost when executing low-width queries, such as the S_n and C_n, at high parallelism levels.

GYM vs ACQ: The ACQ algorithm [13] is the most efficient known $O(\log(n))$-step PRAM algorithm for computing constant-width queries. By inventing ACQ, Gottlob et al. have proved that constant-width queries are in the complexity class NC, i.e., computable in $O(\log(n))$ PRAM steps. Because MapReduce can simulate the PRAM model, the ACQ algorithm can easily be mapped to MapReduce. We call this algorithm *ACQ-MR*. Given a width-w query Q, ACQ-MR executes for $\Theta(\log(n))$ rounds, when $M = \mathrm{IN}^{\frac{1}{\epsilon}}$, with $O(n\mathrm{B}(\mathrm{IN}^{3w} + \mathrm{OUT}, M))$ communication cost. If Q has short-depth GHDs, such as the star query, GYM outperforms ACQ-MR in communication cost while using a comparable number of rounds (Table 1). On the other hand, if Q has long-depth GHDs, say of $\Theta(n)$-depth, such as the chain query, then ACQ-MR can execute for exponentially fewer number of rounds than GYM, though at a higher communication cost. For such queries, we can also match the performance of ACQ-MR exactly with GYM as follows: we first apply our Log-GTA transformation on a $\Theta(n)$-depth width-w GHD D of Q and construct a $O(\log(n))$-depth width-$3w$ GHD D' and then execute GYM on D'. We refer to this combined algorithm as GYM(Log-GTA). GYM(Log-GTA) matches ACQ-MR in terms of both rounds and communication cost. In addition, by using our C-GTA transformation algorithm on D, we can also execute queries in fewer rounds than ACQ-MR but with higher communication cost. Finally, we note that because PRAM can also simulate MapReduce, our GYM(Log-GTA) method also proves that constant-width queries are in NC. We believe this result 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 a sequential algorithm.

1.3 Outline of the Paper

Here is the outline and the specific contributions of this paper:

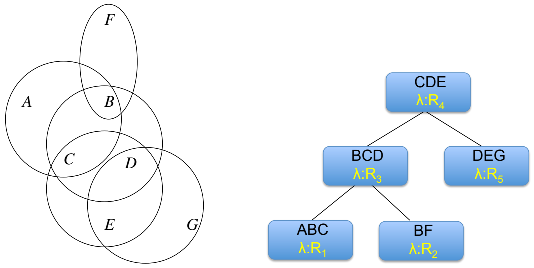
- In Section 4, we describe two distributed versions of Yannakakis's algorithm, DYM-n and DYM-d, as stepping stones to GYM. DYM-n and DYM-d take as input width-1 GHDs of acyclic queries.
- In Section 5, we describe the GYM algorithm, which generalizes DYM-d to any width-w, depth-d GHD of any query and runs in $O(d + \log(n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}, M))$ communication.
- In Section 6, we describe our Log-GTA algorithm for transforming any width-w, depth-d *GHD* D of a query Q, with $d \in \Omega(\log(n))$ into another GHD D' of Q, whose depth is $O(\log(n))$ and width is at most $3w$. By giving D' as input to GYM, we can compute width-w queries in $O(\log((n))$ rounds and $O(n\mathrm{B}(\mathrm{IN}^{3w} + \mathrm{OUT}, M))$ cost.
- In Section 7, we describe our C-GTA algorithm, which transforms a width-w *GHD* D of a query with n vertices into width-$2w$ *GHD* D' with at most $\frac{15n}{16}$ vertices. We can use C-GTA along with Log-GTA to construct GHDs with shorter depths but higher widths. We also describe an optimization called *materialization-before-transformation* that further decrease the communication cost of GYM(Log-GTA).

Section 2 discusses related work. Section 3 covers the necessary background and Section 8 concludes and discusses future work.

2. RELATED WORK

We have reviewed the related work on the Shares and the ACQ algorithms. We next review other work in processing joins in MapReduce and in generalized hypertree decompositions.

MapReduce Join Algorithms: The only other work that theoretically studies multiround join algorithms in MapReduce-related models is reference [8]. This work proves lower bounds on the number of rounds required to compute queries when the amount of data that each reducer is allowed to receive in each round is at most $\frac{\mathrm{IN}}{p^{\epsilon}}$, where p is the number of processors, and ϵ is a parameter called the space exponent. The authors provide an algorithm that matches these lower bounds on a limited set of inputs, called *matching databases*. The property of matching databases is that the size of the output and any intermediate output is at most the size of the input. On non-matching databases however, the algorithm can produce intermediate results of size $\mathrm{IN}^{\theta(n)}$ for any width-w query, where IN is the input size, and n is the number of relations in the query. On matching databases, our algorithm asymptotically matches these lower bounds in terms of rounds and efficiency. On arbitrary databases, our algorithm can violate their space exponent. However, with a modification, we can be within their space requirements and only a $\log(n)$ factor away from their lower bounds in term of the number of rounds, while keeping intermediate relation sizes bounded by $\mathrm{IN}^{3w} + \mathrm{OUT}$. The authors of reference [8] also cover the topic of handling skew in a single round of computation in a follow up work [9]. The same skew-handling methods based on broadcasting can be applied to each round of GYM to get an even workload balance across reducers in each round.



(a) Hypergraph (b) GHD

Figure 2: The hypergraph and the GHD for the join in Example 3.1.

Reference [24] designs a query optimizer for a MapReduce based query processing system. The authors consider breaking a multi-way join into multiple MapReduce rounds consisting of smaller multi-way joins, which can also potentially generate IN$^{\Theta(n)}$ size intermediate relations irrespective of the actual output size. Reference [14] focuses on query planning and optimization for massively parallel RDF queries. RDF data can be thought of as a set of binary relations. The authors try to decompose a query over these binary relations into as flat join plans as possible where each join is a star join. They only experimentally analyze their plans but their join plans could generate IN$^{\Theta(n)}$ size intermediate relations, irrespective of the actual output size. We focus on equijoins in our work. There has been a body of work studying other types of joins in MapReduce, such as theta joins [19] and fuzzy joins under different metrics [2, 3, 23], all of which study constant-round algorithms.

Generalized Hypertree Decompositions: *Generalized Hypertree Decompositions* (GHDs) [15] are very powerful mathematical tools to characterize the difficulty of computational problems that can be represented as hypertrees, such as joins or constraint satisfaction problems. We review the formal definition of GHDs in Section 3. In the context of joins, several prior work study the problem of computing GHDs of queries with small, sometimes minimum, width [10, 15]. Our join algorithms take as input different GHDs of a query; so we assume that a minimum width GHD of a query has already been computed by one of the existing methods. Reference [6] characterizes the complexity of finding a logarithmic depth GHD of a query. However, it does not contain an algorithm for finding such a GHD. In contrast, our work provides a transformation algorithm that takes as input any GHD D of a query Q and constructs a log-depth GHD D' of Q but with larger width than D. In addition, we provide another algorithm for constructing even shorter depth GHDs of Q but also with larger widths.

3. PRELIMINARIES

We review the notions related to generalized hypertree decompositions of queries [15] and then describe 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 3.1. *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 corresponding to 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;
- $\chi : V(T) \to 2^{V(H)}$ is a function associating a set of vertices $\chi(t) \subseteq V(H)$ to each vertex t of T;
- $\lambda : V(T) \to 2^{E(H)}$ 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 $t \in V(T)$, $\chi(t) \subseteq \bigcup \lambda(t)$.

Consider a query Q that joins a set of n relations $R_0, ..., R_{n-1}$, where the schemas of the relations contain m attributes $A_0, ..., A_{m-1}$. We could rephrase these definitions and properties as follows. A GHD of Q is a triple $D = (T, \chi, \lambda)$, where:

- $T(V(T), E(T))$ is a tree;
- $\chi : V(T) \to 2^{V(H)}$ is a function assigning a set of attributes to each vertex t of T; we refer to $\chi(t)$ as the attributes of t.
- $\lambda : V(T) \to 2^{E(H)}$ is a function assigning a set of relations to each vertex t of T; we refer to $\lambda(t)$ as the relations of t.

such that the following properties hold:

1. For each relation R_i, the attributes of R_i are contained within at least one vertex t's attributes.
2. For any attribute A_i, let T_{A_i} be the subgraph in T containing only the vertices containing A_i. Then T_{A_i} must be connected.
3. For every $t \in V(T)$, each attribute of t is contained in at least one of the relations of t.

EXAMPLE 3.2. *Figure 2b shows a GHD of the query from Example 3.1. In the figure, the attribute values on top of each vertex t is the χ assignments for t and the λ assignments are explicitly shown.*

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 (ghw)*** of a hypergraph H is the minimum width of all hypertree decompositions of H. With some abuse of notation, when we say the "width" of a query Q is w, we will mean that the ghw of the hypergraph corresponding to Q is w. The width of a query captures its degree of cyclicity. In general, the larger the width of a query, the more "cyclic" it is. By definition, a query is acyclic if and only if its hypergraph is acyclic. Equivalently, acyclic queries are exactly the queries with width 1 [10].

We will introduce a new property of GHDs of queries called ***fractional generalized hypertree widths (fghw)***, which is a slight variation of a known property called ***fractional hypertree width (fhw)***. Let Q be a query and D be a GHD of Q. For any $v \in V(T)$, we can assign a weight w_R to each relation $R \in \lambda(v)$ such that for each attribute $A \in \chi(v)$, the sum of weights of relations having attribute A is at least one. That is, $\sum_{A \in R} w_R \geq 1$. This weighting is called a *fractional edge cover*. We choose the fractional edge cover that minimizes the sum $\sum_{R \in \lambda(v)} w_R$. Let the minimum value of this sum be w_v. Then the maximum value of w_v over all nodes v is the fghw w^* of D. Then by the Atserias-Grohe-Marx bounds [7], the result of the join of relations in $\lambda(v)$ for any v must be $\leq$ IN$^{w^*}$. It is known that $w^* \leq w$, thus this gives us a tighter bound on intermediate relation sizes than we could obtain by simply considering

IN^w, which would be the cartesian product of all relations in $\lambda(v)$. The difference between fhw and fghw is that fhw is obtained by assigning a weight to each relation R whose attributes are all in $\chi(v)$, instead of to each relation in $\lambda(v)$.

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

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

3.2 Cost Model

We measure the complexity of MapReduce algorithms in terms of their communication cost and the number of rounds of MapReduce they use. Communication in our model consists of two costs: (1) the cost of shuffling data from mappers to reducers; (2) costs of writing the outputs of the MapReduce jobs. We count the cost of writing the outputs as communication because the outputs of MapReduce jobs are usually written to distributed network file systems or databases, which involve communication between machines. Given a query $Q = R_1 \bowtie R_2 \bowtie .... \bowtie R_n$ over n relations, we will refer to the sum of the input sizes as IN, and the output size of the query as OUT. Given these definitions, an optimal algorithm would have communication cost in $O(\text{IN} + \text{OUT})$. As we noted before, we omit the computation cost as it is usually subsumed by communication cost.

4. DISTRIBUTED YANNAKAKIS

We first review the serial version of Yannakakis's algorithm for acyclic queries (Section 4.1). We then show that the algorithm can be parallelized in a straightforward fashion to yield an $O(n)$-round MapReduce algorithm with $O(n\text{B}(\text{IN} + \text{OUT}, M))$ communication cost (Section 4.2). Recall that $\text{B}(X, M)$ is the communication cost of joining two relations of size X using machines of size M and is equal to $\frac{X^2}{M}$ (Section 4.2). Finally, we show that we can reduce the number of rounds of the algorithm to $O(d + \log(n))$, where d is the depth of a width-1 GHD of the input acyclic query (Section 4.3).

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*. Therefore D is effectively a join tree (also called a *parse tree*) for Q that can be joined in any bottom-up fashion. However, instead of directly joining the relations of Q, Yannakakis's algorithm first 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 consecutive phases: (1) a **semijoin phase**; and (2) a **join phase**. The dangling tuple elimination in the semijoin phase guarantees that the sizes of all intermediate tables during the join phase are smaller than or equal to the final output [25]. We next discuss the details of each phase.

Semijoin Phase: Consider a 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 (3); that is, $R := R \ltimes S$.

The executions of step (1) in this recursive algorithm form the *upward* phase, and the executions of step (4) form the *downward* phase. In total, this version of the algorithm performs 2(n-1) semijoin operations. For example, for the GHD in Figure 2b, the 8 semijoins could be: (1) $BCD \ltimes ABC$; (2) $BCD \ltimes BF$, (3) $CDE \ltimes BCD$; (4) $CDE \ltimes DEG$; (5) $DEG \ltimes CDE$; (6) $BCD \ltimes CDE$; (7) $BF \ltimes BCD$; and (8) $ABC \ltimes BCD$. As argued in [25], the semijoin phase guarantees that all dangling tuples are eliminated.

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

EXAMPLE 4.1. *One possible choice of bottom-up join executions for the GHD of Figure 2b could be:*

(1) $Int_1 = R_1 \bowtie R_3$

(2) $Int_2 = R_2 \bowtie Int_1$

(3) $Int_3 = R_5 \bowtie R_4$

(4) $O = Int_2 \bowtie Int_3$

where O is the final output of the join.

4.2 DYM-n

We now show that Yannakakis' algorithm can be distributed in MapReduce in a straightforward manner. We refer to this algorithm as ***DYM-n***. Yannakakis' algorithm involves multiple semijoins or joins of two relations at a time. We use the Shares algorithm as a sub-routine to perform each pairwise join or semijoin.

We first show that two relations R and S can be joined or semijoined in $O(1)$ rounds and $\frac{(|R|+|S|)^2}{M}$ communication below. We first consider the case where R and S have no attributes in common. Then we show that when R and S have an attribute in common, a high level of skew effectively reduces it to the former case with no common attributes.

R and S have no common attributes: This join is a cartesian product and the Shares algorithm performs the join in one round with a cost of $\text{B}(|R| + |S|, M) = \frac{(|R|+|S|)^2}{M}$ 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 $g_r g_s$ machines and send a distinct pair of groups to each machine, which joins its groups locally. This gives a communication cost of $O(\frac{|R||S|}{M}) = O(\frac{(|R|+|S|)^2}{M})$.

A semijoin is simply a join followed by a projection. So for computing $S \ltimes R$, we first join S and R in a single round. Then each reducer locally projects its tuples onto attributes of S. But each tuple of S may join with multiple R tuples, creating duplicates on up to $g_r = \frac{2|R|}{M}$ machines (upto one per group of R). We can eliminate the duplicates in $O(1)$ rounds as follows: each machine hashes its $\leq M$ output tuples into $\frac{|S|}{\sqrt{M}}$ buckets, on the attributes on S. Then it sends each bucket to a distinct machine. Now each hash bucket contains about $\sqrt{M}$ unique tuples. For each such bucket, there are up to g_r machines containing outputs for that bucket. In each subsequent round, $\sqrt{M}$ machines corresponding to the same bucket send their ($\approx \sqrt{M}$) outputs to one machine (so as to not exceed machine size M), which locally de-duplicates them. Since we combine outputs from $\sqrt{M}$ machines in each round, the deduplication requires up to $1 + \log_{\sqrt{M}}(g_r)$ rounds. Since $|R| \leq$ IN and $\epsilon = \log_M(\text{IN})$, the number of rounds required for the semijoin is $O(\epsilon) = O(1)$.

R and S have a common attribute: An example of this case would be the join of $R(A, B) \bowtie S(B, C)$. In this case we can simply hash the tuples on their common attribute B, which would have a communication cost of IN. However, this strategy is prone to skew. If there's a heavy hitter B value b_i in the input, then a single machine can get all of the tuples and violate its machine capacity M. The worst-case scenario is when there is a single heavy hitter value on B [9], making the computation equivalent to a cartesian product. In that case, the best strategy is again to apply the grouping technique from the no common attribute case. Since we make no assumptions about skew, we adopt this worst-case performance technique in our results. In Section 7.3 we discuss in detail how our results improve if we assume the input is skew-free. We now state the following theorem:

THEOREM 4.2. *DYM-n can compute every acyclic query $Q = R_1 \bowtie \ldots \bowtie R_n$ in $O(n)$ rounds of MapReduce 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. The algorithm therefore executes a total of 3(n-1) pairwise joins and semijoins, in a total of $O(n)$ MapReduce rounds. The communication cost of each MapReduce semijoin job $R \ltimes S$ is in $O(\text{B}(|R| + |S|, M))$. Since there are 2($n$-1) semijoins and the largest input to any semijoin operation is the largest relation size, i.e., $\max_i\{|R_i|\} \leq \text{IN}$, the total cost of the semijoin phase is $O(n\text{B}(\text{IN}, M))$. During the join phase, the sizes of the semijoined input relations and the intermediate relations generated are less than or equal to the size of the final output because there are no dangling tuples. Since there are n-1 join operations, the total cost of the join phase is $O(n\text{B}(\text{OUT}, M))$. Therefore 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 modified version of DYM-n 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 in each step, reducing the total number of rounds. We now give a recursive procedure for performing the semijoin phase. Our input is 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.

Steps (1) and (2) above can be performed in $O(1)$ rounds, in parallel for all leaves. Moreover, the number of recursive calls made by the above procedure is $O(d + \log(n))$, as we next prove.

Let $X = \sum_{l \in L(T)} 2^{d(l)}$, where $L(T)$ stands for the leaves in the (remaining) T during the above procedure, and $d(l)$ is the depth of leaf l. Then, X is at most $n2^d$ in the beginning, as there are initially at most n leaves, with depth at most d. Now consider what happens to X 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. Since its starting value is at most $n2^d$, the number of recursive calls (and hence number of rounds), is $O(\log(n2^d)) = O(d + \log(n))$.

Additionally, 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.

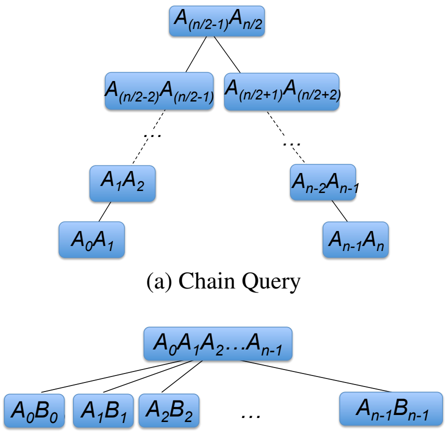
Downward Semijoin Phase in $O(d)$ Rounds: In the downward phase, the algorithm semijoins each child relation with its parent. Note however that the semijoins of the children relations with the same parent are independent and can be done in parallel in $O(1)$ rounds. Thus we can perform the downward phase 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 may be 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 4.3. *DYM-d can compute every acyclic query $Q = R_1 \bowtie \ldots \bowtie R_n$ in $O(d + \log(n))$ rounds of MapReduce 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.*

We note that in general, the depth of any minimum width GHD for some acyclic queries can be $\Theta(n)$. As an example, recall the chain query C_n from Section 1.2, whose lowest depth width-1 GHD has a depth of $\frac{n}{2}$ as shown in Figure 3a. It can be shown that there is no shorter depth width-1 GHD for C_n. However, other acyclic queries, such as the star query S_n can have constant depth width-1 GHDs, as shown in Figure 3b, and therefore be computed in $O(\log(n))$ rounds with optimal communication cost.

In Section 6, we will show that no matter what the depth of the GHD of an acyclic query is, we can always compute its join in $O(\log(n))$ rounds with $O(n\text{B}(\text{IN}^3 + \text{OUT}, M))$ communication



(a) Chain Query

(b) Star Query

Figure 3: Acyclic queries with different depth GHDs.

```
1  Input: GHD $D(T,\chi,\lambda)$ of a query $Q$
2  Materialization Stage:
3  foreach vertex $v$ in D (in parallel):
4  Compute IDB$_v$ = $\bowtie_{R_i \in \lambda(v)} R_i$ by the Shares algorithm.
5  foreach $R_j$ that is not in the $\lambda(v)$ for any $v$ (in parallel):
6  Let tempAssign($R_j$) be a vertex $u$ s.t $\chi(u) \supseteq$ attributes of $R_j$.
7  Compute IDB$_{uj}$=IDB$_u \bowtie R_j$
8  foreach $v$ s.t $v =$ tempAssign($R_j$) for any $R_j$ (in parallel):
9  Compute $IDB_v = \bowtie IDB_{vj}$ in $O(1)$ rounds of MR
10 Yannakakis Stage:
11 Let $Q' = \bowtie_v IDB_v$.
12 Execute DYM$-$d on $Q'$.
```

Figure 4: GYM.

(Corollary 6.10). This result will be a special case of a more general result that states that we can compute any width-w query in $O(\log(n))$ rounds with $O(n\mathrm{B}(\mathrm{IN}^{3w}+\mathrm{OUT}, M))$ communication (Theorem 6.9), essentially showing that we can trade off number of rounds with communication for any query.

5. GYM

Consider a GHD $D(T,\chi,\lambda)$ of a query Q where the width and depth of D are w and d, respectively, and $|V(T)| = O(n)$. Note that the width of Q may be strictly less than w as it may have other GHDs with smaller width. In this section, we show that Q can be computed in $O(n\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}, M))$ communication and $O(d + \log(n))$ rounds of MapReduce using an extension of DYM-d, which we call ***GYM***.

5.1 Overview of GYM

Assume for simplicity that each relation R_i is assigned to some vertex $v \in V(T)$ of D and 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' with communication in $O(\mathrm{B}(\sum |IDB| + \mathrm{OUT}, M))$ in the sizes of Q''s input and output. Based on this observation, GYM, shown in Figure 4, consists of two stages:

Materialization Stage: Materializes each vertex $v \in V(T)$ using the Shares algorithm.

Yannakakis Stage: Executes DYM-d from Section 4 on the materialized GHD D.

The algorithm does not assume that each R_i is assigned to some v and ensures that each R_i appears in the transformed Q' during the materialization stage. We will discuss this technicality in our analysis of GYM in the next section.

5.2 Analysis of GYM

We start this section by stating our first main result:

THEOREM 5.1. *Given a GHD $D(T,\chi,\lambda)$ of a query Q where the width and depth of D are w and d, respectively, GYM executes Q in $O(d + \log(n))$ rounds and $O(|V(T)|\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}, M))$ communication cost.*

PROOF. We start with the materialization stage. First, for each v, the algorithm computes an initial IDB_v by joining the relations assigned to v (line 4). Now, there may be some relations that have not been assigned to any v. So the algorithm next ensures that each such R_i appears in the final join. Let R_j be such a relation. We know by the definition of a GHD, that there is a vertex v whose attributes contain the attributes of R_j. The algorithm then joins each such R_j, in parallel, with its IDB_v to get IDB_{vj} (lines 5-7). Finally, if there are multiple IDB_{vj} relations for a particular v, the algorithm joins them together to compute the final version of IDB_v (lines 8-9). We next calculate the cost of each step of the materialization stage.

1. **Computing Initial $\boldsymbol{IDB_v}$'s**: Since the width of D is w, we join at most w relations for each v. We assume the worst case scenario when the relations do not have any common attributes and the computation is a Cartesian product of the w relations. In this case, no matter what the parallelism level is, Shares will have a cost of IN^w. Therefore, we can perform this step in one round with a total cost of $O(|V(T)|\mathrm{IN}^w)$.
2. **Computing $\boldsymbol{IDB_{vj}}$'s**: Note that this is essentially a semijoin operation filtering some tuples from IDB_v, since each attribute of R_j is contained in the attributes of IDB_v. Therefore each IDB_{vj} can be computed in $O(1)$ rounds in $O(\mathrm{B}(|IDB_v| + |R_j|, M))$ communication cost. The size of each IDB is IN^w, so a loose bound on the cumulative cost of computing all IDB_{vj}'s is $O(|V(T)|\mathrm{B}(\mathrm{IN}^w, M))$.
3. **Computing Final $\boldsymbol{IDB_v}$'s**: Since each IDB_{vj} has the same attributes, this is essentially an intersection operation, which can be performed in one round with a communication cost of again $O(\mathrm{B}(|IDB_{vj}|, M))$'s. Cumulatively, we can also bound this cost as $O(|V(T)|\mathrm{B}(\mathrm{IN}^w, M))$.

Therefore, the materialization stage takes $O(1)$ rounds and $O(|V(T)|\mathrm{B}(\mathrm{IN}^w, M))$ communication cost. Executing DYM-d on the IDB_vs takes $O(d+\log(n))$ rounds and $O(|V(T)|\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}), M))$ cost by Theorem 4.3 and the fact that the size of each IDB_v is at most IN^w. Therefore GYM takes $O(d + \log(n))$ total rounds of MapReduce, and $O(|V(T)|\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}), M))$ cost. □

An immediate corollary to Theorem 5.1 is the following:

COROLLARY 5.2. *Any width-w query can be computed in $O(n)$ rounds of MapReduce and $O(n\mathrm{B}(\mathrm{IN}^w + \mathrm{OUT}, M))$ communication cost.*

PROOF. The proof is immediate from Theorem 5.1 and Lemma 3.3 that states that any width-w query has a GHD D with at most n vertices, which implies that D has $O(n)$-depth. □

We finish this subsection with two notes. First, one can show that there are queries with width w whose GHDs have depth $\Theta(\frac{n}{w})$, therefore causing GYM to execute for a large number of rounds. Second, in practice, when we compute the IDB_v's, we might in the last step of the materialization stage, do a projection onto the attributes assigned to vertex v, i.e., $\chi(v)$, to save communication.

5.3 Example Execution of GYM

We finish this section by describing 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 5a shows a width 3 GHD for this query. GYM on this GHD would first compute the IDBs in each vertex of Figure 5a. The materialized GHD, shown in Figure 5b, 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 5b 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 $O(\mathrm{B}(\mathrm{IN}^3 + \mathrm{OUT}, M))$ communication cost. For comparison, Figure 5c shows a width-1 GHD of the original chain query. Executing GYM directly on this GHD would take $32c + 16$ rounds and $O(\mathrm{B}(\mathrm{IN} + \mathrm{OUT}, M))$ communication cost.

6. CONSTRUCTING $O(\log(N))$ DEPTH GHDS

We now describe our ***Log-GTA*** algorithm (for **Log**-depth **G**HD **T**ransformation **A**lgorithm) which can take any hypergraph H and its GHD $D(T, \chi, \lambda)$ with width w and construct another GHD D^* of H that has $O(\log(|V(T)|))$ depth and whose width is at most $3w$. This result implies that we can construct a $\log(n)$-depth GHD for any query, which has at most three times the width of the query (recall the width of the query is defined to be the minimum width of any GHD of the query). For an acyclic query Q, this result implies that we can construct a $O(\log(n))$-depth GHD of Q, whose width is at most 3. Therefore using GYM, we can execute Q in $O(\log(n))$ rounds with $O(\log(n)\mathrm{B}(\mathrm{IN}^3 + \mathrm{OUT}, M))$ communication.

Given a GHD $D(T, \chi, \lambda)$ of a hypergraph H, Log-GTA iteratively transforms it into a GHD $D^*(T^*, \chi^*, \lambda^*)$. For simplicity, we refer to all GHDs during the transformation as $D'(T', \chi', \lambda')$. In other words, $D' = D$ in the beginning and $D' = D^*$ at the end of the transformation.

Here is the outline of this section. In Section 6.1, we describe some additional metadata that are assigned to the vertices and edges of T' by Log-GTA. In Section 6.2, we define *unique-child-and-grandchild* vertices. These are one of the two types of vertices that will be modified in each iteration of Log-GTA. In Section 6.3, we describe the two transformation operations of Log-GTA: *leaf and unique-child-grandchild inactivations*, which will be iteratively performed to modify D'. Finally, in Section 6.4, we describe the entire Log-GTA algorithm.

6.1 Extending D′

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

1. **Active/Inactive**: Indicates whether v will be modified in later iterations of Log-GTA. Every vertex is active in the beginning and inactive in the end. Once a vertex becomes inactive, it remains inactive until the end of the transformation. We will refer to the subtree of T' that consists only of active vertices and the edges between them as ***active(T')***. We will prove that active(T') is indeed a tree (Lemma 6.4).
2. **Height**: Assigned to each inactive vertex v when v becomes inactive. The value of v's height will be v's height in T' in the iteration that v becomes inactive and, as we prove, all future iterations (Corollary 6.5). In particular, it will be v's height in T^*, i.e., the final tree at the end of the transformation. That is, if v is a leaf in T^*, then v's height will be 0. Similarly, if v's longest path to a leaf in T^* has length l, then $v's$ height will be l.

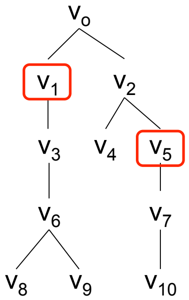


Figure 6: Unique-child-and-grandchild vertices.

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 words, the cc($u$, v) is a subset of the hyperedges of H whose vertices cover the vertices that are shared by both $\chi(u)$ and $\chi(v)$. In query terms, the cc(u, v) is a set of relations whose attributes cover the common attributes between u and v. Initially, in the original $D(T, \chi, \lambda)$, for each (u, v), we set cc(u, v) simply to $\lambda(v)$[1]. Therefore, the size of each cc(u, v) is equal to w. We will show that throughout Log-GTA, the size of common covers will be w for each edge between active vertices, including those that Log-GTA introduces (Lemma 6.4).

6.2 Unique-Child-And-Grandchild Vertices

Consider a tree T of n vertices with a very long depth, say, $\Theta(n)$. Intuitively, such long depths are caused by long chains of vertices, where vertices in the chain have only a single child. Log-GTA will try to shorten long-depth GHDs by identifying and "branching out" such chains. At a high-level, Log-GTA will find a vertex v with a single child c (for child), which also has single child gc (for grandchild), and put v, c, and gc under a new vertex s. We call vertices like v as *unique-child-grand-child (unique-c-gc)* vertices. Figure 6 shows an example tree and two unique-c-gc vertices of the tree, which are drawn inside boxes.

In each iteration Log-GTA will identify a set of nonadjacent unique-c-gc vertices and some leaves of active(T'), and inactivate them (along with shortening the chains of unique-c-gc vertices). We next state an important theorem that lower bounds the number of leaves and nonadjacent unique-c-gc vertices in any tree. This theorem will be used to bound the number of iterations that Log-GTA executes in Section 6.4:

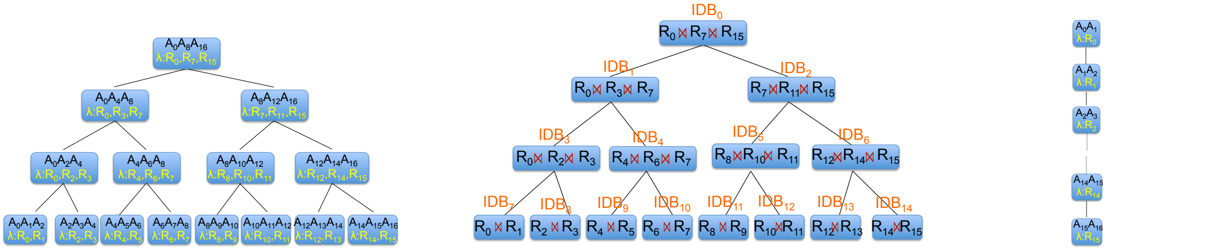
THEOREM 6.1. *If a tree has N vertices, then at least $\frac{N}{4}$ vertices are either leaves or non adjacent unique-c-gc vertices.*

We first discuss two lemmas that the proof of Theorem 6.1 depends on and then prove Theorem 6.1.

LEMMA 6.2. *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. □

[1] We can set cc(u, v) to $\lambda(u)$ without affecting any of our results.



(a) Width 3 GHD of C_{16}

(b) After Computing IDBs of the GHD of Figure 5a

(c) Width-1 GHD of C_{16}

Figure 5: GHDs at various steps of GYM(Log-GTA)

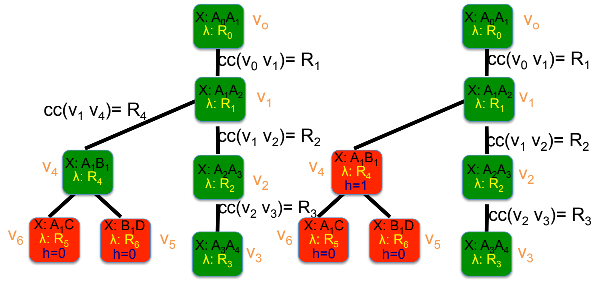


Figure 7: Leaf Inactivation.

LEMMA 6.3. *Suppose a tree has N nodes, L of which are leaves, and U of which are unique-child nodes. Then $4L + U \geq N + 2$.*

The proof of Lemma 6.3 is by an induction on the height h of the tree and is given in appendix A.2. Using Lemmas 6.2 and 6.3, we can now prove Theorem 6.1.

PROOF OF THEOREM 6.1. Let the tree have N nodes, L leaves, and U unique-c-gc nodes. Lemma 6.3 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 6.2, 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. □

6.3 Two Transformation Operations

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

Leaf Inactivation: Takes a leaf l of the active(T') and (1) changes its label to inactive; and (2) sets its 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 of the edge between l and l's parent is removed. Figure 7 shows the effect of this operation on vertex v_4 of an extended GHD. In the figure, green and red indicate that the vertex is active or inactive, respectively. In the figure the attributes of each R_i are the χ values on the nodes that R_i is assigned to.

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

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

(2) Inactivates u and c. Similar to leaf inactivation, sets their heights to 0 if they have no inactive childre, 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 8 visually shows the effect of this operation when inactivating the unique-c-gc vertex v_1 from Figure 7. We next prove a key lemma about these two operations:

LEMMA 6.4. *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 five 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. *$|cc(u, v)| \leq w$ between any two active vertices u and v and does indeed cover the shared attributes of u and v.*
5. *D' is a GHD with width at most $3w$.*

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

PROOF. Let $D'(T', \chi', \lambda')$ be a GHD that satisfies these five 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) and (5) 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 the 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

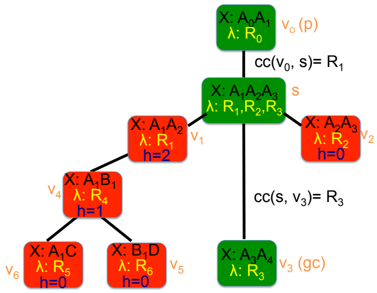


Figure 8: Unique-c-gc Inactivation.

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 w because the sizes of $cc(p, u)$ and $cc(c, gc)$ are at most w 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). Therefore $\chi(p) \cap \chi(s)$ is exactly $\chi(p) \cap \chi(u)$, which is covered by $cc(p, u)$ by assumption. Therefore $cc(p, s)$, which includes $cc(p, u)$, covers $\chi(p) \cap \chi(s)$.
5. For property (5), we need to prove that the three properties of GHDs hold and also verify that the width of the modified D' is at most $3w$.
   - **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 w, therefore $|\lambda(s)|$ is at most $3w$, showing that the width of GHD is still at most $3w$.
   - **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.

□

```
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 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.

COROLLARY 6.5. *Consider any GHD $D(T, \chi, \lambda)$ of a hypergraph H with width w, extending it 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 of H with width at most $3w$, where the height of each inactive vertex v is v's actual height in T'.*

PROOF. Notice that extending D as described in Section 6.1 trivially satisfies the five initial properties. Therefore by Lemma 6.4, the resulting D$'$ is a valid GHD with width at most $3w$ and correct height assignments. □

6.4 Log-GTA

Finally, we present our Log-GTA algorithm. Figure 9 shows the 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 $\frac{1}{4}$ of the remaining active vertices in T', until all vertices are inactive. By Theorem 6.1, we know that in each tree we can always select $\frac{1}{4}$ of the vertices that are either leaves or a set of nonadjacent unique-c-gc vertices. Notice that because the unique-c-gc vertices are nonadjacent, activating any one of them, say u, does not increase the number of active children of other unique-c-gc vertices in U', so other vertices in U' remain as unique-c-gc vertices. As a result, the inactivation of all vertices in U' is a well-defined procedure. Therefore the algorithm essentially performs a sequence of leaf and unique-c-gc inactivations on D', and by Corallary 6.5 we know that the final D' is a valid GHD of the input hypergraph H and has width $3w$. Figure 10 shows a simulation of Log-GTA on the width-1 GHD of the chain query $R_0(A_0, A_1) \bowtie ... \bowtie R_6(A_6, A_7)$ which has depth 6. Log-GTA produces a width-3 GHD with depth 2. In the figure, we label the selected leaves and unique-c-gc vertices with L and U, respectively and omit the common-cover labels.

We next prove that our algorithm takes $O(\log(|V(T)|))$ iterations to finish. We then prove the height of each inactive vertex v is at most equal to the iteration number at which v was inactivated.

LEMMA 6.6. *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 in each iteration the number of active vertices decreases by $\frac{1}{4}$. Initially there are $|V(T)|$ active vertices, so the algorithm terminates in $O(\log(|V(T)|))$ iterations. □

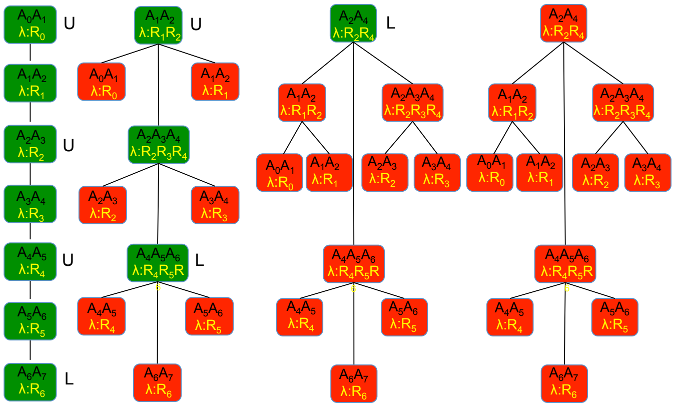


Figure 10: Log-GTA simulation.

LEMMA 6.7. *The height of each inactive vertex v is at most the iteration number at which v was inactivated.*

PROOF. By Corollary 6.5, 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. □

Finally we can state our second main result:

THEOREM 6.8. *Given any GHD $D(T, \chi, \lambda)$ of a hypergraph H with width w, we can construct a GHD $D'(T', \chi', \lambda')$ of H where $w' \leq 3w$, $\text{depth}(T') = \min\{\text{depth}(T), O(\log(|V(T)|))\}$, and $|V(T')| \leq 2|V(T)|$.*

PROOF. By Corollary 6.5 the width of D' is at most $3w$. By Lemmas 6.4, 6.6 and 6.7, the height of each vertex v is v's true height in the graph and is $O(\log(|V(T)|))$, implying that 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 also at most $\text{depth}(T)$. 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, $|V(T')| \leq 2|V(T)|$. □

Theorems 6.8, 5.1, and Lemma 3.3 imply that we can execute any width-w query in $O(\log(n))$ rounds and $O(n\text{B}(\text{IN}^{3w}+\text{OUT}, M))$ communication, which we state as a theorem.

THEOREM 6.9. *Any query Q with width w can be executed in $O(\log(n))$ rounds of MapReduce and $O(n\text{B}(\text{IN}^{3w} + \text{OUT}, M))$ communication.*

PROOF. By Lemma 3.3, any width-w Q has a GHD D with width w and $O(n)$ vertices. By Theorem 6.8, we can transform D into a D' with depth $O(\log(n))$ and width at most $3w$. Finally, by Theorem 5.1, we can run GYM on D' and compute Q in $O(\log(n))$-rounds in $O(n\text{B}(\text{IN}^{3w} + \text{OUT}, M))$ communication. □

A corollary of Theorem 6.9 for acyclic queries is the following:

COROLLARY 6.10. *Any acyclic query can be executed in $O(\log(n))$ rounds of MapReduce and $O(n\text{B}(\text{IN}^3 + \text{OUT}, M))$ communication.*

7. EXTENSIONS

We first describe another GHD transformation algorithm called C-GTA. C-GTA takes a GHD D of width-w and n vertices and transforms it into a GHD D' of width-$2w$ and $\leq \frac{15n}{16}$ vertices. Therefore, it can potentially shorten the depths of $\Theta(n)$-depth GHDs by a constant fraction. We then show two optimizations we can make to GYM when executing on GHDs constructed by Log-GTA and C-GTA.

7.1 C-GTA

C-GTA (for **C**onstant-depth **GHD** **T**ransformation **A**lgorithm) transforms a width-w GHD $D(T, \chi, \lambda)$ into a width-$2w$ GHD having fewer nodes, using a series of merges. 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. And as long as t_1 and t_2 have not been obtained from the previous merge, the width of the tree stays $\leq 2w$.

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 all but one of the leaves into pairs and merge them, and merge the remaining leaf with v.
3. For each vertex u that has a unique child c, if c does not have an odd 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 half of $\max(L, U)$ nodes from T. Since $L + U \geq \frac{n+2}{4}$ by Lemma 6.3, the procedure removes at least $\frac{n+2}{16}$ nodes, and so after all the merges, the resulting tree T' has at most $\frac{15n}{16}$ nodes left, justifying the following lemma:

LEMMA 7.1. *If a query Q has a width-w GHD $D = (T, \chi, \lambda)$, then Q also has a GHD $D' = (T', \chi', \lambda')$ with width $\leq 2w$ and $|V(T')| \leq \frac{15|V(T)|}{16}$.*

Lemma 7.1 implies that we can reduce the number of rounds of some queries with $\Theta(n)$-depth GHDs by a constant fraction, at the expense of increasing their communication costs from $O(n(\text{IN}^w + \text{OUT}))$ to $O(n(\text{IN}^{2w} + \text{OUT}))$ We can also repeatedly use C-GTA and then combine it with Log-GTA and obtain the following lemma:

LEMMA 7.2. *If a query Q has a width-w GHD $D = (T, \chi, \lambda)$, then for any i, there exists a GHD $D' = (T', \chi', \lambda')$ with width $\leq 3 \times 2^i \times w$ and depth $\leq \log((\frac{15}{16})^i n)$ $|V(T')| \leq \frac{15|V(T)|}{16}$.*

The proof of Lemma 7.2 follows directly from Lemma 7.1 and Theorem 6.8 and is omitted. By Lemma 7.2, we can further tradeoff communication by constructing even lower depth trees than a single invocation of Log-GTA. However the depth decrease due to C-GTA invocations incurs significantly more width increases compared to the decrease due to the single Log-GTA invocation.

7.2 Materialization-Before-Transformation

Let D be width-w GHD of a query Q with depth $\Theta(n)$. One way to compute Q in $O(\log(n))$ rounds is to first apply Log-GTA to D and get back a D' with depth $\log(n)$ and width potentially equal to $3w$; and then run GYM on D'. As we have seen this incurs a cost of $O(n\text{B}(\text{IN}^{3w} + \text{OUT}, M))$. Our *Materialization-Before-Transformation* (MBT) optimization instead takes the following steps:

(1) We first materialize each IDB_v on D and incur a cost of $O(\text{IN}^w)$ and now get a width-1 D''. Now, the size of the IDB_v's in D'' is of size $O(\text{IN}^{w^*})$, where $w^* \leq w$ is the fghw of D (recall the definition of fghw from Section 3).

(2) We then perform Log-GTA on D'' and get back a width-3 GHD D'''; and (3) We execute GYM on D''', incurring a cost of $O(n\text{B}(\text{IN}^{3w^*}+\text{OUT}, M))$, which can be lower than $O(n\text{B}(\text{IN}^{3w}+\text{OUT}, M))$ when $w^* < w$.

7.3 Discussion on Skew

Skew in a database input refers to variation in the frequency of different attribute values. A table without skew would be one where there are no heavy hitter values in the table. 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, (as opposed to max-skew cost of $O(\text{B}(\text{IN} + \text{OUT}, M)) = O(\frac{(\text{IN}+\text{OUT})^2}{M}))$. This implies that GYM's cost improves from $O(n\text{B}(\text{IN}^w + \text{OUT}, M))$ to $O(n(\text{IN}^w + \text{OUT}))$.

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. For example, consider the join phase of GYM, when joining a node with its k children in a GHD. When skew is high, we have to perform this join in $O(\log(k))$ rounds to limit the computations to pairwise joins. If we did not limit the computation to pairwise joins, the communication cost of Shares could go up to $\text{IN}^{\theta(k)}$. However, when there is no skew, we can compute this 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 a another round by hashing each table on the attributes of R (which 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 $O(d)$.

8. 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 tradeoff communication against number of rounds of MapReduce computations. We believe our approach of discovering such tradeoffs without designing new MapReduce algorithms, but by only proving different combinatorial properties of GHDs 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 short depths, as depth determines the number of rounds in our context. As a first step, in appendix B we describe an algorithm for constructing GHDs with minimum possible depths for acyclic queries. We also plan to study the interplay between the depths and different notions of widths further to discover more tradeoffs one can make between number of rounds and communication.

Our efficiency bounds of $O(n\text{B}(\text{IN}^{3w} + \text{OUT}, M))$ and $O(n\text{B}(\text{IN}^{3w^*} + \text{OUT}, M))$ are tight for some queries, such as the chain query, and tighter than previous literature. However, it seems from some examples that they are not always tight. For example, it is easy to construct width-2 GHDs which have $\Theta(n)$ depths, that are transformed into width-3 GHDs of $O(\log(n))$-depth by Log-GTA, as opposed to a width-6 one. We believe our results can be improved for some queries, possibly by considering other structural properties we have not focused on.

9. REFERENCES

[1] F. Afrati, M. Dodlekar, C. Ré, S. S., and J. D. Ullman. Gym: A multiround join algorithm in mapreduce and its analysis. Technical report, Stanford University, January 2015.

[2] F. N. Afrati, A. D. Sarma, D. M., A. G. Parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. In *ICDE*, 2012.

[3] F. N. Afrati, A. D. Sarma, A. Rajaraman, P. Rule, S. Salihoglu, and J. D. Ullman. Anchor-Points Algorithms for Hamming and Edit Distances Using MapReduce. In *ICDT*, 2014.

[4] 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.

[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE TKDE*, 23, 2011.

[6] Akatov, Dmitri and Gottlob, Georg. Balanced Queries: Divide and Conquer. In *International Conference on Mathematical Foundations of Computer Science*, 2010.

[7] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42, 2013.

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

[9] P. Beame, P. Koutris, and D. Suciu. Skew in Parallel Query Processing. In *PODS*, 2014.

[10] C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. *TCS*, 239, 2000.

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

[12] A. Durand and E. Grandjean. The Complexity of Acyclic Conjunctive Queries Revisited. *CoRR*, abs/cs/0605008, 2006.

[13] G. Gottlob, N. Leone, and F. Scarcello. Advanced Parallel Algorithms for Acyclic Conjunctive Queries. Technical report, Vienna University of Technology, 1998. http://www.dbai.tuwien.ac.at/staff/gottlob/parallel.ps.

[14] Z. Goasdoué, F.and Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. Technical report, French Institute for Research in Computer Science and Automation, 2014.

[15] G. Gottlob, M. Grohe, M. Nysret, S. Marko, and F. Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG*, 2005.

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

[17] G. Gottlob, N. Leone, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. *J. ACM*, 48, 2001.

[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] A. Okcan and M. Riedewald. Processing Theta-joins Using MapReduce. In *SIGMOD*, 2011.

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

[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*, 2, 2009.

[23] R. Vernica, M. J. Carey, and C. Li. Efficient Parallel Set-similarity Joins Using MapReduce. In *SIGMOD*, 2010.

[24] S. Wu, F. Li, S. Mehrotra, and B. Ooi. Query Optimization for Massively Parallel Data Processing. In *SoCC*, 2011.

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

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

APPENDIX

A. PROOFS OF LEMMAS

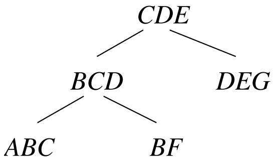


Figure 11: A parse tree for the join in Example 3.1

A.1 Lemma 3.3

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 Lemma 6.3

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.

B. 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, *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.

B.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 [26] 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 B.1. *Recall the join from Example 3.1. The GYO reduction we do (and that corresponds to the parse tree in Figure 11) 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.

B.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 B.2. *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'.[2]
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 [3] Then the set E_R in the first iteration contains all leaves of $T_{\min}$. If the depth is greater than 3, then 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 $T_{\min}$[4]. 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 B.3. *Let T be a parse tree for connected acyclic hypergraph H. If an attribute A[5] 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 B.3

LEMMA B.4. *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 B.5. *Any parse tree T of acyclic hypergraph H is a spanning tree[6] 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 B.6. *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 B.7. *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. □

LEMMA B.8. *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 B.8, 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 B.6 and B.7, 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[7]. 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:

[2] According to Lemma B.6, 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.

[3] Depth 2 means a root and its children.

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

[5] 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.

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

[7] 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.).

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 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.