GYM: A Multiround Join Algorithm In MapReduce

Foto N. Afrati\textsuperscript{1}, Manas Joglekar\textsuperscript{1}, Christopher Ré\textsuperscript{2}, Semih Salihoglu\textsuperscript{1}, Jeffrey D. Ullman\textsuperscript{2}
\textsuperscript{1}National Technical University of Athens, \textsuperscript{2}Stanford University

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' 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 $O(d)$ rounds and $O(n)$ communication and computation cost. Using GYM we achieve two main results: (1) Every width-$w$ query can be computed in $O(n)$ rounds of MapReduce with $O(n\log(n)+\text{OUT})$ cost; (2) Every width-$w$ query can be computed in $O(\log(n))$ rounds of MapReduce with $O(n\log(n)+\text{OUT})$ 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 even shorter depth GHDs with longer widths, effectively showing a spectrum of tradeoffs one can make between communication and computation and the number of rounds of MapReduce. By simulating MapReduce in the PRAM model, our second main result also implies the result of Gottlob et al. \cite{gottlob2013} that computing acyclic and constant-width queries are in NC. In fact, for certain queries, our approach yields significantly fewer PRAM steps than does the construction of the latter paper. However, we achieve our results using only Yannakakis' algorithm, which has been perceived to have a sequential nature. Instead, we surprisingly show that Yannakakis' algorithm can be parallelized significantly by giving it as input short-depth GHDs of queries.

1. INTRODUCTION

The problem of evaluating joins efficiently in distributed environments has gained importance since the advent of Google’s MapReduce \cite{google2004} and the emergence of a series of distributed systems with relational operators, such as Pig \cite{pig2008}, Hive \cite{hive2008}, SparkSQL \cite{sparksql2010}, and Myria \cite{myria2013}. 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. It has been recently shown that the most efficient one-round algorithm to evaluate joins is the Shares algorithm, introduced by Afrati et al. \cite{afrati2014} in the MapReduce context. However, much less is known about multiround algorithms, which constitute the topic of this paper.

The specific problem we consider is the evaluation of a join query $Q$ of $n$ relations. Since it is impossible to perform a join using less communication or computation than the sum of the input size, IN, and the output size, OUT, we use those sizes as the parameters for measuring complexity. We consider three important structural properties of queries in this paper: depths, widths, and fractional generalized hypertree widths of their generalized hypertree decompositions (GHDs). In particular, the width of a query, i.e., the minimum width of any of its GHDs, will be one of the factors determining how efficiently we can compute it, and the depths of its GHDs will determine the number of rounds of MapReduce we use. The width of a query characterizes its degree of cyclicity, where acyclic queries are equivalent to width-1 queries.

We describe a multiround MapReduce algorithm, called GYM, for Generalized Yannakakis in MapReduce, which is a distributed and generalized version of Yannakakis’ algorithm for acyclic queries \cite{yannakakis1990}. GYM takes as input a GHD of a query $Q$ of width $w$ and depth $d$, and executes $O(d)$ rounds of semijoins and joins with a total communication and computation cost that increases as a function of $w$. GYM can be highly efficient on GHDs with low widths and execute for a small number of rounds, say $O(1)$ or $O(\log(n))$, on GHDs with short depths.

We then show how to construct GHDs of different depths and widths and expose a spectrum of tradeoffs one can make between the number of rounds and communication and computation cost. Figure 1 shows the tradeoff for a width-$w$ query, whose GHDs have depth $\Omega(n)$. In the figure, $D$ is a width-$w$ GHD for query $Q$, and Log-GTA and C-GTA are our GHD construction algorithms. In the figure and throughout the paper, we assume that algorithms run at high parallelism levels. That is, we assume there is a sufficient...
number of reducers so that each reducer gets a small number of tuples in each round.

Our approach to modifying Yannakakis’s algorithm to run in $O(1)$ or $O(\log(n))$ rounds might surprise database researchers, who have often thought of Yannakakis’s algorithm to have a sequential nature, executing for $\Theta(n)$ steps in the PRAM model. In the PRAM literature [11, 15, 10], acyclic queries are described as being polynomial-time sequentially solvable by Yannakakis’s algorithm, but highly “parallelizable” by the ACQ algorithm [12], 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 and also outperform it on certain classes of queries in terms of both number of rounds and 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 takes as input a width-1 GHD of an acyclic query and executes a sequence of $O(n)$ semijoins and joins in $O(n(IN + OUT))$ time. In contrast, 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 $O(d)$ rounds of MapReduce with $O(n(IN^w + OUT))$ communication and computation cost.**

GYM is based on three simple observations:
1. Each semijoin and join of Yannakakis can be computed in a straightforward fashion in one round of MapReduce with communication and computation cost that is linear in $IN + OUT$.
2. We can further parallelize the algorithm from step 1 by executing all the semijoins or joins on the same level of $D$ in two rounds with communication and computation cost that is linear in $IN + OUT$.
3. We can generalize the algorithm from step 2 to take as input any GHD $D$ of any query by first running Shares on each vertex of $D$ in parallel. This preprocessing step takes one round and at most $O(IN^w)$ cost and generates a set of acyclic intermediate relations over which the algorithm from step 2 can be run.

On acyclic queries with constant-depth GHDs, such as the star query (see Section [12]), GYM is optimal both in terms of number of rounds: executing for a constant number of rounds; and efficiency: incurring communication and computation cost that is linear in $IN$ and $OUT$. However there are acyclic queries, such as the chain query (see Section [12]), whose width-1 GHDs have a depth of $\Theta(n)$, GYM executes such queries in $\Theta(n)$ rounds.

We next show how to execute such queries by GYM in fewer number of rounds but with more communication and computation cost by proving a combinatorial lemma about GHDs, which may be of independent interest to readers:

**Main Result 2: Given a width-$w$, depth-$d$ GHD $D$ of a query, we can construct a GHD $D'$ of $Q$ of depth $O(\min\{\log(d), \log(n)\})$ 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 and computation cost from $O(n(IN^w + OUT))$ to $O(n(IN^{3w} + OUT))$, we can decrease the number of rounds from $\Theta(n)$ 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 [12] that constant-width queries are in the complexity class $NC$ but we can also state tighter efficiency results by leveraging existing theory about GHDs.

We also describe another GHD transformation algorithm called C-GTA, using which one can further reduce the depths of the GHDs of queries to $O(\log((\frac{1}{\sqrt{w}})n))$ at the cost of increasing their widths to $2^3w$, exposing a layer of other tradeoffs that are possible between number of rounds and communication and computation.

### 1.2 Shares, ACQ-MR, and GYM

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 relations $S_n$: $S(A_1, ..., A_n)$ $\bowtie$ $R_1(A_1, B_1) \bowtie$ ... $\bowtie R_w(A_w, B_w)$; and (2) the chain query of relations $C_n$: $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie$ ... $\bowtie R_n(A_n, A_{n+1})$.

**GYM vs Shares**

Shares is a one-round MapReduce algorithm, whose communication and computation cost can be expressed as $O(f(p)IN + OUT)$, where $f(p)$ is a non-decreasing function of the parallelism level $p$. The exact form of $f(p)$ depends on the sizes of the input relations and the schemas of the queries. The cost numbers shown in Tables 1 and 2 assume each input relation has the same size. 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, Shares incurs the optimal communication cost possible among one-round algorithms [3, 6]. However, Shares can be prohibitively expensive when computing queries with small outputs at large parallelism levels $p$. In particular, at the parallelism levels that ensure only a small number of tuples to be mapped to each reducer, the cost of Shares can be as high as $IN^{n-1}$. However, at the same parallelism levels, the cost of GYM is $O(n(IN^w + OUT))$. Therefore, GYM significantly outperforms Shares in terms of communication and computation cost when executing low-width queries, such as the star and chain queries, at large parallelism levels. Moreover, GYM can almost match Shares’s number of rounds when executing queries which have constant-depth GHDs, such as the star query (Table 1), but can execute for significantly larger number of rounds for queries

| # Rounds | Shares($S_n$) | ACQ-MR($S_n$) | GYM($D_{S_n}$) |
|----------|--------------|---------------|----------------|
| Communication | $O(p^{-1}IN + OUT)$ | $O(n\min\{pIN^2, IN^3\} + OUT))$ | $O(n(IN + OUT))$ |
| Computation | $O(p^{-1}IN + OUT)$ | $O(n(3IN^3 + OUT))$ | $O(n(IN + OUT))$ |

Table 1: Performances of Algorithms on the Star Query $S_n$. $D_{S_n}$ is a GHD of $S_n$.

| # Rounds | Shares($C_n$) | ACQ-MR($C_n$) | GYM(Log-GTA($D_{C_n}$)) | GYM($D_{C_n}$) |
|----------|--------------|---------------|--------------------------|----------------|
| Communication | $O(p^{-1}IN + OUT)$ | $O(n\min\{pIN^2, IN^3\} + OUT))$ | $O(n(3IN^3 + OUT))$ | $O(n(IN + OUT))$ |
| Computation | $O(p^{-1}IN + OUT)$ | $O(n(3IN^3 + OUT))$ | $O(n(IN + OUT))$ | $O(n(IN + OUT))$ |

Table 2: Performances of Algorithms on the Chain Query $C_n$. $D_{C_n}$ is a GHD of $C_n$. 
with Θ(n)-depth GHDs, such as the chain query (Table 2).

We note that in the original description of Shares, there is a limit to how much the algorithm can parallelize. It has been noted [7] that even at its highest parallelism level, Shares is prone to skew and does not guarantee that a constant number of tuples are sent to each reducer. For clarity of presentation, our description of GYM will also be prone to skew (Section 5). However, we show in Appendix A how to go to even higher parallelism levels to guarantee that a constant number of tuples are sent to each reducer in each round of GYM and Shares. The performance comparison of the algorithms at those parallelism levels depict the same exact tradeoff between rounds and total communication and computation.

GYM vs ACQ

The ACQ algorithm [12], 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 with \(O(n(3^w + \text{OUT}))\) communication and computation cost.

If \(Q\) has short-depth GHDs, such as the star query, GYM outperforms ACQ-MR both in terms of rounds and communication and computation cost (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. However, 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-3\(w\) GHD \(D'\) and then execute GYM on \(D'\). We refer to this combined algorithm as GYM(Log-GTA). In addition, by using our C-GTA transformation algorithm on \(D\), we can also execute queries in fewer number of rounds than ACQ-MR but with more communication and computation cost.

In our work, we also describe optimizations that leverage existing theory about GHDs and maximum output sizes of queries and reduce the communication cost of GYM(Log-GTA). Our first optimization, Materialization-Before-Transformation, reduces the communication cost of GYM(Log-GTA) to \(O(n (\max\{IN^w, \min\{pIN^{2w}, IN^{3w}\} + \text{OUT})\))\), where \(w^*\) is the fractional generalized hypertreewidth (defined in Section 3) of a GHD of \(Q\) and is at most \(w\). Our second optimization, On-The-Fly-Materialization, can be applied when executing at low parallelism levels \(p\) and reduce the communication cost further down to \(O(n (\max\{IN^w, \min\{pIN^{2w}, IN^{3w}\} + \text{OUT})\)). Similar optimizations can also be used on the ACQ-MR algorithm to improve its efficiency.

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, DY-M-n and DY-M-d, as stepping stones to GYM. DY-M-n and DY-M-d algorithms take as input width-1 GHDs of acyclic queries.
- In Section 5, we describe our GYM algorithm, which generalizes DY-M-d to any width-\(w\), depth-\(d\) GHD of any query and runs in \(O(d)\) rounds and \(O(n(3^w + \text{OUT}))\) communication and computation.
- In Section 6, we describe our Log-GTA algorithm for transforming any width-\(w\), depth-\(d\) GHD \(D\) of a query \(Q\) into another GHD \(D'\), whose depth is \(O(\min\{d, \log(n)\})\) and width is at most \(3w^*\). By giving \(D'\) as input to GYM, we can compute width-\(w\) queries in \(O(n(\text{IN}^{3w} + \text{OUT}))\) 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 \(\frac{n}{w}\) vertices. We can use C-GTA along with Log-GTA to construct GHDs with shorter depths but higher widths.

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

2. RELATED WORK

The only other work that theoretically studies multiround join algorithms on MapReduce-related models is reference [9]. This work proves lower bounds on the number of rounds required to compute connected queries when the amount of data that each reducer is allowed to receive in each round is at most \(\frac{n}{p}\), where \(p\) is the number of processors, and \(\epsilon\) is a parameter called the space exponent. They 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, their algorithm can produce intermediate results of size \(In^{\Theta(n)\epsilon}\) 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 also matches their efficiency. On arbitrary databases, our algorithm can violate their space exponent but with a modification we describe in Appendix A we can be within their space requirements and only a \(\log(n)\) factor away from their lower bounds in term of number of rounds, while keeping intermediate relation sizes bounded by \(\text{IN}^{3w} + \text{OUT}\). The authors of [6] also cover the topic of handling skew in a single round of computation in a follow up work [7]. 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.

Reference [13] 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. Reference [24] also designs a query optimizer for a MapReduce based query processing system. The authors consider breaking a multi-way join into multiple map-reduce rounds consisting of smaller multi-way joins, which again can potentially generate \(IN^{\Theta(n)}\) size intermediate relations irrespective of the actual output size.

We focus on equi-joins 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 [1, 2, 23], all of which study constant-round algorithms.

3. PRELIMINARIES

In this section, we review the notions related to generalized hypertree decompositions of queries [14] and then describe our cost model.
Figure 2: The hypergraph and the GHD for the join in Example 3.1.

### 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 $t \in V(T)$, $\chi(t) \subseteq \cup \lambda(t)$.

Consider a query $Q$ that joins a set of $n$ relations $R_0, \ldots, R_{n-1}$, where the schemas of the relations contain $m$ attributes $A_0, \ldots, A_{m-1}$. We can 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 $T$;
- $\lambda : V(T) \to 2^{E(H)}$ is a function assigning a set of relations to each vertex $t$ 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. Let $T_{A_i}$ be the subgraph in $T$ containing only the vertices that are assigned $A_i$. For each $A_i, T_{A_i}$ is connected.
3. For every $t \in V(T)$, the attributes assigned to $t$ are contained within the relations assigned to $t$.

Figure 2b shows the GHD of the query from Example 3.1. In the figure, the attribute values on top of each vertex $t$ is the $\chi$ assignments for $t$ and the $\lambda$ 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 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 hypertree is acyclic. Equivalently, acyclic queries are exactly the queries with width $1$.

In this paper, we introduce a new property of GHDs of queries called **fractional generalized hypertree widths (fghw)**. 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, $\sum_{R \in \lambda(v)} w_R \geq 1$. 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 $\omega$ of $D$. Then by the Atserias-Grohe-Marx bounds, the result of the join of relations in $\lambda(v)$ for any $v$ must be $\leq IN^{\omega}$. It is known that $\omega \leq w$, 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 $\lambda(v)$.

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 two lemmas about connected hypergraphs and GHDs of queries that will be used in later sections:

**Lemma 3.2.** If $D(T, \chi, \lambda)$ is a GHD of a connected hypergraph $H$, then for each edge $(p, c) \in E(T)$, $\chi(p) \cap \chi(c) \neq \emptyset$.

The proof of this lemma follows from property (2) of GHDs and is omitted. In query terms, this lemma states that if the hypergraph of a query $Q$ is connected and $D(T, \chi, \lambda)$ is a GHD of $Q$, then each parent and child in $T$ share at least one attribute. Our second lemma states that if a query has a GHD $D(T, \chi, \lambda)$ of width $w$ and depth $d$, then we can w.l.o.g. assume that there are at most $n$ vertices in $V(T)$.

**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 the Appendix.

### 3.2 Cost Model

We measure the complexities of MapReduce algorithms in terms of their communication cost, the total computation 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 $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ 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 and computation cost that is linear in $IN + OUT$.

### 4. DISTRIBUTED YANNAKAKIS

In this section 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 rather straightforward fashion to yield an $O(n)$-round MapReduce algorithm with $O(n(IN + OUT))$ communication and computation (Section 4.2). Notice that this is optimal up to a factor that only depends on the number of relations $n$. We also note that this cost is equal to the run-time of the serial version of the algorithm and no serial algorithm with better run-time is known. Finally, we show that we can reduce the number of rounds of the algorithm to $O(d)$, 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 $Q = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, 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 \cite{Yannakakis}. 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 $R_i$, and let $S$ be the relation assigned to $t$’s parent.

1. Replace $S$ by the semijoin of $S$ with $R$, $S \bowtie R = S \bowtie \pi_{R \bowtie S}(R)$.
2. Remove the leaf for $R$ from the tree.
3. Recursively process the resulting tree.
4. Compute the final value of $R$ by computing its semijoin with the value of $S$ that results from step (3); that is, $R := R \bowtie S$.

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 28, the 8 semijoins could be: (1) $ABC \bowtie BCD$; (2) $BF \bowtie BCD$; (3) $BCD \bowtie CDE$; (4) $DEG \bowtie CDE$; (5) $CDE \bowtie DEG$; (6) $CDE \bowtie BCD$; (7) $BCD \bowtie BF$; and (8) $BCD \bowtie ABC$. As argued in \cite{Yannakakis}, the semijoin phase guarantees that all dangling tuples are eliminated.

Join Phase

Next, the Yannakakis algorithm performs a series of (n-1) joins, in any bottom-up order on $T$. For example, one possible choice of bottom-up join executions for the GHD of Figure 28 could be:

1. $\text{Int}_1 = R_1 \bowtie R_3$
2. $\text{Int}_2 = R_2 \bowtie \text{Int}_1$
3. $\text{Int}_3 = R_3 \bowtie \text{Int}_1$
4. $O = \text{Int}_2 \bowtie \text{Int}_3$

where $O$ is the final output of the join.

4.2 DYM-n

In this section, we show that Yannakakis’s algorithm can be distributed in MapReduce in a straightforward manner. We refer to this algorithm as DYM-n. Note that each semijoin and join operation is between two relations $R$ and $S$ that contain at least one common attribute (recall Lemma 4.2). Therefore, we can compute $R \bowtie S$ in a single round of MapReduce: we map the tuples of the relations on their common attributes, and then the reducers output the tuples of $S$ that have at least one matching tuple from $R$. The communication and computation cost of this distributed semijoin operation would be linear in $|R| + |S|$, since the output of the semijoin is smaller than or equal to the size of $|S|$.

Similarly we can perform a join operation, say $\text{Int} = R \bowtie S$, by mapping the tuples of the relations on their common attributes and then performing a local join at each reducer. The communication and computation cost of the join operation would be linear in $|R| + |S| + |\text{Int}|$. Summing up the costs of the two phases, we can state the following theorem:

**Theorem 4.1.** DYM-n can compute every acyclic query $Q = R_1 \bowtie \ldots \bowtie R_n$ in $O(n)$ rounds of MapReduce in $O(n(max\{|R_i|\} + \text{OUT}))$ communication and computation cost.

**Proof.** For each edge in $T$, there is exactly two semijoin operations, one in the upward phase and one in the downward phase, and one join operation. The algorithm therefore executes a total of $2(n-1)$ MapReduce jobs. The communication and computation cost of each MapReduce semijoin job is linear in the sizes of its input relations. Since there are $2(n-1)$ semijoins and the largest input to any semijoin operation is the largest relation size, i.e., $\text{max\{|R_i|\}}$, the total cost of the semijoin phase is $O(n \times \text{max\{|R_i|\}})$. During the join phase, the sizes of the semijoinned 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 $O(n\text{OUT})$. Therefore the sum of the costs of both phases is $O(n(max\{|R_i|\} + \text{OUT}))$, completing the proof.

Since $\text{max\{|R_i|\}}$ can be as large as $|\text{IN}|$, a corollary to Theorem 4.1 that uses simpler terminology would be that acyclic queries can be computed in $O(n)$ rounds and $O(n(|\text{IN}) + \text{OUT}))$ communication and computation cost.

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 in a single round of MapReduce. With this extra parallelism, we can reduce the number of rounds required to be $O(d)$, where $d$ is the depth of the GHD $D(T, \chi, \lambda)$, without asymptotically affecting the communication and computation cost of the algorithm. We refer to this modified version of DYM-n as DYM-d.

Semijoin Phase in $O(d)$ Rounds

Consider any non-leaf relation $R$ in $T$. Notice that during the upward phase of the semijoins, $R$ is semijoined with all of its children in some order before $R$’s parent (if one exists) is semijoined with $R$. We can replace $R$’s semijoins with all of its children with two rounds of MapReduce as follows. Suppose $R$ has $k$ children $P_1, P_2, \ldots, P_k$, and assume w.l.o.g. that we want to semijoin $R$ with its children in the order of $P_1, P_2, \ldots, P_k$. In other words, we want to compute $(P_k \bowtie (P_{k-1} \bowtie (\ldots \bowtie (P_1 \bowtie R)))$.

Round 1: In parallel we produce $k$ relations $P_i' = P_i \bowtie R$.

Round 2: We take the intersection of the $P_i'$’s, which gives us the semijoin of $R$ with each $P_i'$ in the order we want. Our MapReduce job takes all the $P_i'$’s as input. The Map function generates
a key-value pair $(t, 1)$ for each tuple $t$ in $P'$. The value actually does not matter. The Reduce function checks, for a given key $t$ that we have received $k$ values, and if so, $t$ becomes an output – part of the new value of relation $R$. If we receive fewer than $k$ values for a given $t$, then $t$ is missing from at least one of the $P'$'s and therefore will not appear in the semijoin-reduced $R$.

Moreover, for all non-leaf relations in the same level of $T$, we can execute these two MapReduce jobs in parallel. Therefore, for each level of $T$, we can perform two MapReduce jobs in the upward semijoin phase, finishing the phase in 2d rounds. We still perform $n - 1$ upward semijoin operations each taking as input two relations, whose sizes are at most $\max_i(|R_i|)$ (recall that $R_i$ are the input relations). Additionally we perform one intersection operation for each non-leaf relation $R$. The intersection operation for $R$ takes one intermediate relation $P'_i$ for each child $P_i$ of $R$, where $|P'_i| \leq |R| \leq \max_i(|R_i|)$. Therefore, across all intersection operations, one intermediate relation $P'_i$ of size at most $\max_i(|R_i|)$ is processed for each edge of $T$. Since there are $n - 1$ edges in $T$, the total cost of the intersection operations and the entire upward semijoin phase is $O(n \times \max_i(|R_i|))$.

In the downward phase, the algorithm semijoins each child relation with its parent. Note however that the semijoin of each child relation of the same parent are independent and can be done in parallel in a single round. Therefore, we can perform the downward phase in $d$ rounds and in $O(n \times \max_i(|R_i|))$ communication and computation. In summary, the entire semijoin phase can be performed in $O(d)$ rounds and in $O(n \times \max_i(|R_i|))$ communication and computation.

**Join Phase in $O(d)$ Rounds**

Similar to the upward semijoin phase, during the join phase, each non-leaf relation $R$ is joined with all of its children in some order before the result of these joins is joined with $R$’s parent (if one exists). Similar to our trick for the upward semijoin phase, we can replace these join operations with two rounds of MapReduce as follows. Assume w.l.o.g. that we want to compute $((R \bowtie P_1) \bowtie P_2) \ldots \bowtie P_k$.

Round 1: In parallel we produce $k$ relations $P'_i = P_i \bowtie R$.

Round 2: We join all the $P'_i$’s on $R$’s attributes, which gives us exactly $(((R \bowtie P_1) \bowtie P_2) \ldots \bowtie P_k)$.

Similar to the upward semijoin phase, the join of each non-leaf relation $R$ in the same level can be executed in parallel. By the same arguments, the join phase can be performed in 2d rounds and $O(n \times \max_i(|R_i|))$ cost with these modifications. Therefore, both the semijoin and join phases of DYM-$n$ can be performed in $O(d)$ rounds without asymptotically affecting its communication and computation cost, justifying the following theorem:

**Theorem 4.2.** DYM-$d$ can compute every acyclic query $Q = (R_1 \bowtie \ldots \bowtie R_n$ in $O(d)$ rounds of MapReduce and $O(n \times \max_i(|R_i|) + \text{OUT})$ communication and computation 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 GHD $D$ for some acyclic queries can be $\Theta(n)$. As an example, recall the chain query $C_n$ from Section 1.2 whose lowest depth GHD has a depth of $\frac{n}{2}$ as shown in Figure 3a. It can be shown that there is no shorter depth GHD for $C_n$. However, other acyclic queries, such as the star query $S_n$, can have constant depth GHDs, as shown in Figure 3b and therefore be computed in constant rounds with optimal communication and computation cost.

In Section 5 we will show that no matter how long the depth of the GHDs of acyclic queries, we can always compute them in $O(\log(n))$ rounds with $O(n(\text{IN}^3 + \text{OUT}))$ communication and computation (Corollary 5.10). This result will be a special case of a more general result that states that we can compute any width-$w$ query $Q$ in $O(\log(n))$ rounds with $O(n(\text{IN}^w + \text{OUT}))$ communication and computation (Theorem 5.9), essentially showing that we can trade off number of rounds with communication and computation for any bounded-width 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 will show that we can compute $Q$ in $O(n(\text{IN}^w + \text{OUT}))$ communication and computation and $d$ rounds of MapReduce using an extension of DYM-$d$, which we call GYM. We start by giving an overview of GYM (Section 5.1). Then, we analyze the cost of GYM (Section 5.2) and finish this section with an example execution of GYM (Section 5.3).

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 $IB_{B_i} = \exists R_i \in \chi(v) R_i$. Now, consider the query $Q' = \exists R_i \in V(T) IB_{B_i}$. 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 $IB_{B_i}$, $D$ is now a width-1 GHD for $Q'$. Therefore we can directly run DYM-
d to compute $Q'$ in linear communication and computation in the sizes of $Q'$'s input and output. Based on this observation, our GYM algorithm, 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 Section 5.2.

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 $d$ rounds and $O(|V(T)||IN^w + OUT|)$ communication and computation cost.

**Proof.** We start with the materialization stage, which takes three rounds of MapReduce. In the first round, 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 $v$. In the next two rounds, the algorithm ensures that each such $R_i$ appears in the final join. Let $R_i$ be such a relation. We know by definition of GHD, that there is however a vertex $v$ whose attributes contain the attributes of $R_i$. In the second round, the algorithm joins each such $R_i$, in parallel, with its $IDB_v$ to get $IDB_{vij}$ (lines 5-7). In the third round, if there are multiple $IDB_{vij}$ relations for a particular $v$, the algorithm joins them together to compute the final version of $IDB_v$ (lines 8-9). We calculate the cost of each round.

1. **Computing Initial $IDB_v$’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 $O(w)$. Therefore, the total cost of this round is $O(|V(T)||IN^w)$.

2. **Computing $IDB_{vij}$’s:** Note that this is essentially a semijoin operation filtering some tuples from $IDB_v$, since each attribute of $R_i$ is contained in the attributes of $IDB_v$. Therefore the cost of computing each $IDB_{vij}$ is linear in the sizes of $IDB_v$ and $R_i$. A loose bound on the cumulative cost of computing all $IDB_{vij}$’s could therefore be $O(|V(T)||IN^w)$.

3. **Computing Final $IDB_v$’s:** Since each $IDB_{vij}$ has the same attributes, this is essentially an intersection operation, whose cost is again linear in the sizes of $IDB_{vij}$’s. Cumulatively, we can also bound this cost as $O(|V(T)||IN^w)$.

Therefore, the materialization stage takes three rounds and $O(|V(T)||IN^w)$ communication and computation cost. Executing DYM-d on the $IDB_v$’s takes $O(d)$ rounds and $O(|V(T)||IN^w + OUT|)$ cost by Theorem 4.2 and the fact that the size of each $IDB_v$ is at most $IN^w$. Therefore GYM takes $O(d)$ total rounds of MapReduce, and $O(|V(T)||IN^w + OUT|)$ 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 $O(n(IN^w + OUT))$ communication and computation cost.

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

5.3 Example Execution of GYM

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 round of the three rounds of the materialization stage, do a projection onto the attributes assigned to vertex $v$, i.e., $\chi(v)$, to save communication and computation.

5.4 Constructing $O(\log(N))$ Depth GHDS

In this section, we describe our Log-GTA algorithm (for Log-depth GHD Transformation Algorithm) 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 $O(\log(n))$ MapReduce rounds with $O(n(IN^3 + OUT))$ communication and computation.

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 on 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 6.3). 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 $l$.

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

- **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 hyperedges whose common 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 $\chi(v)$. Therefore, the size of each cc($u$, $v$) is equal to $v$.

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.

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 $\max\{0, \max_{c \mid \text{height}(c)} + 1\}$, 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 8 shows the effect of this operation on vertex $v_2$ 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 $\chi$ 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 $gc$ 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 children, and one plus the maximum height of their inactive children otherwise.
3. Removes the edges $(p, u)$ and $(u, c)$ and adds an edge from $s$ to both $u$ and $c$.
4. Adds an edge from $p$ to $s$ with $cc(p, s) = cc(p, u)$ and $s$ to $gc$ with $cc(s, gc) = cc(c, gc)$.

Figure 10 visually shows the effect of this operation when inactivating the unique-c-gc vertex $v_l$ from Figure 9.

We next prove a key lemma about these two operations:

**Lemma 6.4.** Assume an extended GHD $D'(T', x', \lambda')$ 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 $v$.
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 $D'(T', x', \lambda')$ 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 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, $\chi$, and $\lambda$ 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 also has one active child $gc$. So $u$ and $c$ are part of a chain of active($T'$). We effectively merge $u$ and $c$ together into another active vertex $s$ on this chain without affecting the acyclicity or connectedness of active($T'$). Notice that we also add two edges from $s$ to $u$ and $v$ but $u$ and $v$ are inactive.
2. For property (2) observe that the only two subtrees we need to consider are the subtrees rooted at $u$ and $c$, which we call $S_u$ and $S_c$, respectively. Notice that all of the edges that go down the tree from $u$ and $c$ after removing $(c, gc)$ were in 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 $v$, $\chi(v) \subseteq \bigcup \{v\}$. 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 $3w$. 

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 11: Log-GTA.

- **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 \(p, s, u, c\), 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 \(\chi(\text{c})\). 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 \(gc\).

**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-vertex, common-edges, and heights as described in Section 6.7 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 6.6 trivially satisfies the five initial properties. Therefore by Lemma 6.1 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 11 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 Corollary 6.5 we know that the final \(D'\) is a valid GHD of the input hypergraph \(H\) and has width \(3w\). Figure 12 shows a simulation of Log-GTA on the width-1 GHD of the chain query \(R_0(A_0, A_1) \bowtie \cdots \bowtie R_0(A_6, A_7)\) 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-vertex 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.
7. EXTENSIONS

In this section 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 $\Theta(\eta)$-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 Constant-depth GHD Transformation Algorithm) 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 $n \in V(T)$ and setting $\chi(n) = \chi(t_1) \cup \chi(t_2)$, $\lambda(n) = \lambda(t_1) \cup \lambda(t_2)$ and setting the neighbors of $n$ 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-cgc nodes, then the above procedure removes at least half of $max(L, U)$ nodes from $T$. Since $L + U \geq \frac{2w^2}{4}$ by Lemma 6.3 the procedure removes at least $\frac{w^2}{4}$ 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 and computation costs from $O(n(IN^{3w} + OUT))$ to $O(n(IN^{2w} + OUT))$. 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 $D_i = (T_i, \chi_i, \lambda_i)$ with width $\leq 3 \times 2^i \times w$ and $|V(T_i)| \leq \log((\frac{16}{15})^i n) |V(T)| \leq \frac{15|V(T)|}{16}$.

The proof of Lemma 7.2 follows directly from Lemma 7.1 and Theorem 6.3 and is omitted. By Lemma 7.2 we can further tradeoff communication and computation 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 On-The-Fly Materialization of IDBs

The communication (and computation) cost of executing GYM on a width-3w GHD $D'$ that was transformed by Log-GTA from a width-$w$ GHD $D$ is $O(n(IN^{3w} + OUT))$. We next show that, when running at low parallelism levels $p$, we can decrease this communication cost to $O(n(pIN^{2w} + OUT))$. In practice, $p$ can be significantly smaller than $IN$. This reduction in communication relies on the following observation, the proof of which is given in Appendix E.

**Lemma 7.3.** The size of the common-cover between any two vertices in $D'$ at the end of the transformation is at most $2w$.

Therefore although some vertices might have $3w$ relations assigned to them what they need to communicate with their parents in the upward semijoin phase is limited to the attributes that are contained in at most $2w$ relations. As a result, the number of tuples they need to communicate with their parents have size at most $IN^{2w}$. Based on this observation, our optimization omits the materialization stage of GYM completely. Instead, it materializes $IDB_v$’s on-the-fly in the upward semijoin phase and then fully materializes them in the downward semijoin phase of the Yannakakis stage. Specifically we make the following changes to GYM:

- **Materialization Stage:** We skip this stage completely.
- **Modified Upward Semijoin:** First, we compute $\pi_{cc(v,p_a)}IDB_v$ for each leaf $v$ in two rounds, where $pa$ is $v$’s parent: (1) We execute Shares on $\lambda(v)$ to materialize $IDB_v$ on-the-fly. However, each reducer now projects its portion of $IDB_v$ onto $cc(v, pa)$ and locally eliminates duplicates. (2) We then eliminate possible duplicate tuples across reducers by a second round of MapReduce. Later, when a non-leaf $v$ is to be semijoined with its children $c_1, \ldots, c_k$, we materialize $IDB_v$ on the fly and compute the semijoin in two rounds: (1) We now run Shares on $\lambda(v)$ and $\pi_{cc(c,v)}IDB_{c_i}$ for each reducer $r$ first materializes its portion of $IDB_v$ on-the-fly and then semijoins $IDB_v$ with the groups that it has received from $IDB_{c_i}$. Finally, $r$ projects the results onto $cc(c, pa)$, where $pa$ is again $v$’s parent. (2) Similarly, we eliminate global duplicates across reducers.
- **Modified Downward Semijoin:** We execute the same modified Shares algorithm from the modified upward semijoin phase, now communicating $\pi_{cc(p,v)}IDB_p$ to $v$, but now we also materialize $IDB_v$ fully after the semijoin.

The join phase of the Yannakakis stage does not change. Since each $IDB_v$ can be of size $IN^{3w}$ the computation cost of the on-the-fly materialization of $IDB_v$’s is still $O(IN^{3w})$. However as we next calculate, the communication cost for the semijoin operations is now $O(pIN^{2w})$.

- **Modified Upward Semijoin:** There are two rounds: (1) The inputs to the first round are the relations $\lambda(v)$, which are of size at most $IN$ and possibly some $\pi_{cc(c,v)}IDB_{c_i}$, which are of size at most $IN^{3w}$. Since Shares can replicate each tuple to at most $p$ reducers, the communication cost of the map phase of the first round is at most $pIN^{3w}$. The output of the first round of MapReduce before global duplicate elimination is also at most $pIN^{3w}$, since each tuple of (potentially semijoin) $\pi_{cc(c,v)}IDB_{c_i}$ can be duplicated at most $p$ times. (2) The second round simply eliminates the duplicates from the output of the first round and also has communication cost of at most $IN^{3w}$.
- **Modified Downward Semijoin:** The downward phase has the exact cost analysis since we again communicate only $\pi_{cc(p,v)}IDB_p$ from $v$’s parent to $v$ during the semijoin. We ignore the cost of fully materializing $IDB_v$ because the size of $IDB_v$ is less than $OUT$ since all dangling tuples are removed at this point.

Finally we analyze the number of rounds of computation. The number of rounds of the join phase remains the same. Due to the extra round for duplicate elimination, we double the number of rounds of the semijoin phase. Summing up, the number of rounds remains at $O(d)$ and the following theorem is justified.

**Theorem 7.4.** Any query $Q$ with width $w$ can be executed in $O(\log((n))$ rounds of MapReduce and $O(n(pIN^{2w} + OUT))$ com-
7.3 Materialization-Before-Transformation

Let $D$ be width-$w$ GHD of a query $Q$ with depth $\Theta(n)$. One way to compute $Q$ in $\log(n)$ 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 $O(\text{IN}^{3w} + \text{OUT})$. Our Materialization-Before-Transformation (MBT) optimization instead takes the following steps: (1) We first materialize each $ID_{Ds}$ on $D$ and incur a cost of $O(\text{IN}^{w})$ and now get a width-$1$ $D''$. Now, the size of the $ID_{Ds}$’s in $D''$ is of size $O(\text{IN}^{w})$, where $w^* \leq w$ is the fgwh of $D$ (recall the definition of fgwh 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 $O(n(\text{IN}^{3w} + \text{OUT}))$. Thus the total cost of GYM with MBT would be $O(n(\text{IN}^{\max(w, 3w^*)} + \text{OUT}))$, which is lower than $O(n(\text{IN}^{3w} + \text{OUT}))$. When used with the on-the-fly-materialization optimization, we can also reduce the communication to $O(n(\max\{\text{IN}^w, \min\{\text{IN}^{2w^*}, \text{IN}^{3w}\}\} + \text{OUT}))$.

Finally, MBT can also be applied before a C-GTA transformation to decrease the cost from $O(n(\text{IN}^{2w} + \text{OUT}))$ to $O(n(\text{IN}^{\max(\text{w, 2w^*})} + \text{OUT}))$.

8. CONCLUSIONS AND FUTURE WORK

We have shown 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 and computation 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. However, no prior work has studied depths of GHDs, which determine the number of rounds in our context. We are interested in exploring methods to construct GHDs with minimum depth. As a first step, in Appendix C 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 and computation.

Our efficiency bounds of $O(n(\text{IN}^{3w} + \text{OUT}))$ and $O(n(\text{IN}^{\max(w, 3w^*)} + \text{OUT}))$ 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 in this paper.

On the practical side, we have an implementation of GYM [16] on Spark [27]. Our preliminary experiments comparing GYM to Shares shows, as expected, that when joining more than several relations, GYM can be much more efficient than Shares even at very small parallelism levels. We plan to compare its performance against existing systems, such SparkSQL or Hive, which generally evaluate joins with pairwise binary join plans. Finally, in the serial setting, a set of new join algorithms with worst-case and instance optimality guarantees have been developed [18]. We are interested in designing either distributed versions of those algorithms or new algorithms with similar optimality guarantees, that can go beyond the guarantees of GYM.

9. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. M., A. G. Parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. In ICDE, 2012.
[2] 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.
[3] 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.
[4] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. IEEE TKDE, 23, 2011.
[5] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. SIAM J. Comput., 42, 2013.
[6] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. In PODS, 2013.
[7] P. Beame, P. Koutris, and D. Suciu. Skew in Parallel Query Processing. In PODS, 2014.
[8] C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. TCS, 239, 2000.
[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004.
[10] A. Durand and E. Grandjean. The Complexity of Acyclic Conjunctive Queries Revisited. CoRR, abs/cs/0605008, 2006.
[11] G. G., N. Leone, and F. Scarcello. On Tractable Queries and Constraints. In DEXA, 1999.
[12] 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.
[13] Z. Goasdufè, Fand 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.
[14] G. Gottlob, M. Grohe, M. N., M. S., and F. Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In WG, 2005.
[15] G. Gottlob, M. Grohe, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. J. ACM, 48, 2001.
[16] GYM Source Code. https://github.com/mrjosins/mrjosins.
[17] D. Halperin, V. Teixeira de Almeida, L. Choo, S. Chai, P. Koutris, D. Moritz, J. Oriz, V. Ruanviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria Big Data Management Service. In SIGMOD, 2014.
[18] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. SIGMOD, 42, 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 COMPASS, 1979.
[27] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.
APPENDIX

A. COMPARISON OF ALGORITHMS AT - MAXIMUM LEVEL OF PARALLELISM

Measuring the total communication cost of an algorithm can be misleading in distributed settings. For example, if Shares uses only one reducer, then for any query its communication cost can be given as IN + OUT. However this parallelism level corresponds to serial computation when the entire query is computed at a single machine. Therefore, we are interested in the behavior of algorithms at very high parallelism levels. In particular it can be insightful to compare algorithms at their “maximum parallelism levels” (MPL), which is the level that causes processor to get a constant number of input tuples in each round, and hence require a constant amount of computation and memory (constant here means independent of IN). Notice that by definition, when an algorithm is executing at its MPL, each processor does constant amount of work, so in general the lower the MPL of an algorithm, the better it is, since fewer servers with constant memories are needed to perfectly parallelize the algorithm. Assume that the MPL of an algorithm is \( P \), then notice also that when the algorithm is using \( P \) reducers, the total communication and computation cost of the algorithm at each round is exactly equal to \( P \). This is because, there are \( P \) processors and each one does \( O(1) \) amount of work in each round.

Figure 13 shows the MPL of each algorithm, and the number of rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm. We see that our algorithms require fewer rounds of the algorithm.

B. PROOFS OF LEMMAS

B.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: If \( |V(T)| = 1 \leq n \), 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 \( |V(T)| \leq k \), 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 \( Q_2 \), \( Q_2 \) 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.

B.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 $c_1, \ldots, c_k$. 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 $4L + U \geq (N - 1) + 2k$.

Since $k \geq 2$, we conclude that $4L + U \geq N + 2$, which completes the induction in this case.

Case 3: The root has $k \geq 2$ children $c_1, \ldots, c_k$. 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.3 Lemma 7.3

Lemma 6.4 proves that the common-cover between any two active vertices is at most $w$ during the execution of Log-GTA. We now show that it is at most $2w$ between any two (inactive) vertices at the end of the transformation. Consider the common cover between any vertex $x$ and its parent $y$ at the end of Log-GTA. We first observe that after $x$ gets inactivated, $x$’s parent never changes. Therefore $y$ was also $x$’s parent when $x$ got inactivated. There are two possibilities. If $x$ got inactivated by a leaf inactivation, then we know by Lemma 6.4 that the common cover between $x$ and $y$ is at most $w$, since both $x$ and $y$ were active vertices before $x$ was inactivated. Otherwise, $x$ got inactivated because it was a unique-c-gc vertex $y$ or the child of a unique-c-gc vertex $c$. We show the case when $x$ was the unique-c-gc vertex $u$. The proof for when $v$ is the child of a unique-c-gc vertex is similar and omitted. In this case $y$ is the new vertex $s$ that was created by Log-GTA and $\chi(s) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(v)) \cup (\chi(v) \cap \chi(ge)))$. Therefore $\chi(s) \cap \chi(u) = (\chi(p) \cap \chi(u)) \cup (\chi(u) \cap \chi(v))$, which are covered by $cc(u, p)$ and $cc(u, c)$, each of which contain at most $w$ relations. Therefore the common cover between $x$ and $y$ is of size at most $2w$, completing the proof.

C. MINIMIZING THE PARSE TREE DEPTH FOR ACYCLIC QUERIES

In this section we show how to construct a width-1 GHD of an acyclic query with the minimum depth possible. We will refer to width-1 GHGs 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.

C.1 GYO Reduction

We say that a hyperedge $e$ of a hypergraph is consumed by a hyperedge $e'$ if $e$ contains nodes that are either unique to $e$ (i.e., not in any other hyperedge of 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 $V' = V$ 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 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 edge that consumes $e$.

Example C.1. Recall the join from Example 3.1.

The GYO reduction we do (and that corresponds to the parse tree in Figure 7) is the following:

1. $R_3(A, B, C)$ is consumed by $R_3(\{B, C, D\})$ because $A$ appears only in $R_3(A, B, C)$, 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 $R_3(A, B, C)$.

2. For the next step of the GYO reduction, we are left with a hypergraph that has four hyperedges (since $R_3(A, B, C)$ 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, $R_3(D, E, G)$ is consumed by $R_4(C, D, E)$.

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

C.2 Minimum Depth Parse Trees

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

Example C.2. Consider

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

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, $R_1$ is the root. Moreover if we choose the first step, $B$ is only in the schema of $R_3$, since $R_1$ and $R_2$ have been deleted from the hypergraph.

Moreover if we choose the first (long) parse tree then, if we have such relations (instead of 5), we will need $O(\log n)$ 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 $H$:

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 $H'$.
3. We repeat using the hypergraph $H'$ which is previous $H'$ with $E_R$ deleted.

According to Lemma 6.6 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.
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 \((u, v)\) 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 \( T_{\text{min}} \) be a minimum depth parse tree. Suppose the depth of \( T_{\text{min}} \) is greater than \( 2^i \). Then the set \( E_R \) in the first iteration contains all leaves of \( T_{\text{min}} \). If the depth is greater than 3, then the set \( E_R \) in the second iteration contains all parents of the leaves of \( T_{\text{min}} \). 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 \( T_{\text{min}} \). 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 C.3.** Let \( T \) be a parse tree for connected acyclic hypergraph \( H \). If an attribute \( A \) 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 C.3.

**Lemma C.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 C.5.** Any parse tree \( T \) of acyclic hypergraph \( H \) is a spanning tree 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 \( G_0 \).

3Depth 2 means a root and its children.
4We 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.
6with the root having ingoing edges and all edges go from child to parent.

**Lemma C.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 \( A_2 \). This means that the attributes in \( A_2 \) 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 C.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 C.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 \( u \) to \( v \). 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 C.3 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 C.6 and C.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.

Now we need to prove that the rest of the algorithm builds a tree of minimum depth.

When the algorithm builds a tree of depth one or two then the argument about how we choose the root proves that the algorithm correctly constructs the minimum-depth parse tree. We have the following cases for \( G_0 \) for tree built by the algorithm which is of depth one or two:

1. \( G_0 \) is a single node. This is trivial.

2. 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.).
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.