# Fast Hypertree Decompositions via Linear Programming: Fractional and Generalized

VAISHALI SURIANARAYANAN*, University of California, Santa Barbara, USA
ANIKAIT MUNDHRA*, University of California, Santa Barbara, USA
AJAYKRISHNAN E S*, University of California, Santa Barbara, USA
DANIEL LOKSHTANOV, University of California, Santa Barbara, USA

Efficient query evaluation in databases and solving constraint satisfaction problems (CSPs) are crucial for improving performance in many real-world applications, from large-scale data management to decision-making systems. These problems naturally admit hypergraph representations, and are efficiently solvable using hypertree decomposition techniques, when the decomposition width is small. However, these techniques require finding small-width decompositions efficiently. This remains a significant challenge despite research from both the database and theory communities.

In this work we present *Ralph (Randomized Approximation using Linear Programming for Hypertree-Decompositions)*, a fast algorithm to compute low width fractional and generalized hypertree decompositions for input hypergraphs, as well as lower bounds for these widths. We build on the recent breakthrough by Korchemna et al. [FOCS 2024], which introduced the first polynomial time approximation algorithm for fractional (generalized) hypertree width. Our approach combines this theoretical advancement with practical heuristic improvements utilizing (mixed-integer) linear programs. Along the way, we present new algorithms with strong theoretical guarantees.

Through empirical evaluation on the nearly 3700 instances of HyperBench, a well established benchmark suite for hypertree decompositions, we find near optimal decompositions for all previously solved instances and low width decompositions for all 500 previously unsolved instances, effectively pushing state-of-the-art.

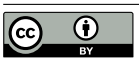
CCS Concepts: • **Theory of computation → Database theory**; **Database query processing and optimization (theory)**; **Parameterized complexity and exact algorithms**; **Constraint and logic programming**; • **Information systems → Data management systems**; **Database query processing**; **Query optimization**; **Join algorithms**; • **Computing methodologies → Artificial intelligence**; **Search methodologies**; • **Mathematics of computing → Graph theory**; **Hypergraphs**; **Graph algorithms**; **Approximation algorithms**; **Linear programming**; **Mathematical optimization**; **Discrete mathematics**;

Additional Key Words and Phrases: Hypertree decompositions; fractional hypertree width; generalized hypertree width; hypertree width; query optimization; conjunctive queries; constraint satisfaction; algorithms; MILP; linear programming; HyperBench; benchmark; hypergraphs; query evaluation; acyclic queries; tractability; structural decompositions; tree decompositions

**ACM Reference Format:**
Vaishali Surianarayanan*, Anikait Mundhra*, Ajaykrishnan E S*, and Daniel Lokshtanov. 2025. Fast Hypertree Decompositions via Linear Programming: Fractional and Generalized. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 159 (June 2025), 27 pages. https://doi.org/10.1145/3725296

*These authors contributed equally to this work

Authors' Contact Information: Vaishali Surianarayanan*, vaishalisurianarayanan@gmail.com, University of California, Santa Barbara, USA; Anikait Mundhra*, anikait@mundhra.com, University of California, Santa Barbara, USA; Ajaykrishnan E S*, es.ajaykrishnan@gmail.com, University of California, Santa Barbara, USA; Daniel Lokshtanov, daniello@ucsb.edu, University of California, Santa Barbara, USA.




ACM 2836-6573/2025/6-ART159
https://doi.org/10.1145/3725296

## 1 Introduction

Efficient query evaluation is a fundamental challenge in modern database systems, particularly as data grows in scale and complexity. Queries in relational databases can be modelled as *hypergraphs*, where nodes represent attributes and hyperedges correspond to the relationships (or joins) between those attributes. This structural representation has been beneficial for optimizing query evaluation in both theoretical and practical contexts.

For instance, for *acyclic queries*, an important special class of queries whose hypergraph is acyclic, the *Yannakakis' algorithm* [50], introduced in the 1980s, provides an elegant solution for join evaluation. This algorithm efficiently processes acyclic queries by reducing the size of intermediate results through *semi-joins*, ensuring *polynomial-time evaluation*. Yannakakis' algorithm has long been central to the theory of query processing and has gained momentum in practice as evidenced by several recent extensions and applications [26–28, 46–48]. However, *cyclic queries* require more involved techniques and this is where hypergraph decompositions play an important role. A Yannakakis' style algorithm can be performed over tree decompositions of the query hypergraph to speed up query evaluation. In fact, there is a growing list of other algorithms, such as join aggregation [30, 40], join ranked enumeration [10], and join sampling [8, 33] that use hypergraph decompositions.

Hypergraph tree decomposition based algorithms have also been applied both in commercial database systems such as LogicBlox [7, 31, 32, 45] and advanced research prototypes such as EmptyHeaded [1–3]. For example, to process SQL queries, EmptyHeaded uses generalized hypertree decompositions [15] to select query plans [29] and a worst-case optimal join algorithm [41] to execute query plans.

Hypergraph decomposition-based methods for query evaluation have been well studied. But for all such methods, one needs to compute a hypertree decomposition of *low width* (defined in Section 2) for the query hypergraph. *Despite extensive research, computing such decompositions remains a challenge [18].* Our work deals with this exact problem by building on the recent breakthrough theoretical work by Korchemna et al. [34] for computing low-width hypertree decompositions, bridging the gap between theory and practice. We now provide an example of a cyclic query.

. **Example:** *Cyclic Query.* Consider a relational database schema with the following relations:

- `Person(person_id, name, age)`
- `Lives_In(person_id, city_id)`
- `Works_For(person_id, company_id)`
- `Company(company_id, company_name)`
- `Located_In(company_id, city_id)`
- `City(city_id, city_name)`.

Now consider the following query – *"Find the names of people who work for a company located in a specific city which is also the city they live in"*. It can be expressed in SQL as follows:

```
SELECT P.name
FROM Person P, Works_For W, Company C, Located_In L,
City Y, Lives_In H
WHERE P.person_id = W.person_id
  AND W.company_id = C.company_id
  AND C.company_id = L.company_id
  AND L.city_id = Y.city_id
  AND Y.city_name = 'New York'
  AND H.person_id = P.person_id
  AND H.city_id = Y.city_id;
```

The *hypergraph* corresponding to the above cyclic query along with a hypertree decomposition is shown in Figure 1. The exact definition of hypertree decompositions is provided in Section 2. Note that we have chosen a simple example for easy illustration purposes. It might be possible to transform this instance into an acyclic query by considering factors like functional dependencies.

Cyclic queries present significant challenges for evaluation, as the intermediate results can grow exponentially if not handled efficiently. To handle cyclic queries efficiently, as mentioned earlier, we can use decomposition-based techniques [15, 16]. The complexity of hypertree decompositions are captured by the notion of width of the decomposition. *The smaller the width, the faster the algorithms one can design.* There are several related width measures for hypergraph decompositions, the most commonly used ones to date are *hypertree width (htw)* [21], *generalized hypertree width (ghtw)* [21], and *fractional hypertree width (fhtw)* [23] – we define these formally in Section 2. They are related as $fhtw \leq ghtw \leq htw \leq 3 \cdot ghtw$. A hypergraph is acyclic if and only if $ghtw = 1$.

There has been a large body of work trying to compute hypertree decompositions with small width for all three notions of width both in theory [6, 18–21, 21, 23, 36, 38, 39] and practice [5, 13, 15, 17, 18, 22, 35, 43, 44]. In fact $htw$ was introduced mainly because it is easier to compute when compared to $ghtw$ and $fhtw$. Of the three, $fhtw$ seems the hardest to compute but is the smallest and thus the most effective. In this work we deal with computing $fhtw$ and $ghtw$ both quickly and efficiently.

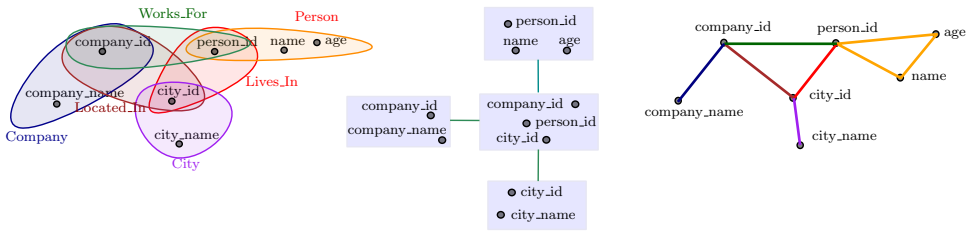



Fig. 1. (Top Left) Hypergraph $H$ corresponding to query, vertices are attributes, edges are relations. (Right) Hypertree decomposition of $H$ - generalized hypertree width = 2, fractional hypertree width = 1.5, treewidth = 2. (Bottom Left) Gaifman graph corresponding to $H$.

*Even more motivations for hypertree decompositions:* Algorithms based on hypergraph tree decompositions have been designed for problems that are motivated by different application areas, such as constraint satisfaction problems [19, 23, 38], combinatorial auctions [14], and automated selection of Web services based on recommendations from social networks [25] in addition to databases.

Despite the significant success of hypergraph decomposition methods, more widespread adoption is being held back by how computationally expensive it currently is to compute tree decompositions of bounded width [18]. This paper aims to bridge this gap. Our Contribution is three-fold:

- **New approximation and heuristic algorithms** for efficiently computing *fractional hypertree width (fhtw)* and *generalized hypertree width (ghtw)*
- **Highly optimized LP & MILP based implementations**, designed to handle complex datasets efficiently.
- **Comprehensive experimental evaluation**, demonstrating that our algorithms outperform existing state-of-the-art methods on the benchmark dataset *HyperBench*.

Our approach builds on the work of Korchemna et.al. [34] and the *Robertson-Seymour framework*. It employs advanced techniques such as *linear programming* with *rounding schemes* to compute $fhtw$ and $ghtw$ efficiently. We conduct empirical evaluation on HyperBench, a benchmark dataset

containing 3682 hypergraphs [12] that were created from instances of CQs and CSPs found in both industry and academia.

By integrating theoretical advances with practical optimization techniques, this work brings *fractional hypertree decompositions* closer to adoption in modern database systems, enabling more efficient evaluation of cyclic queries.

## 2 Preliminaries

As discussed earlier, our work deals with finding low width decompositions of hypergraphs. In this section we provide some concrete definitions and state our problem.

A ***graph*** $G = (V(G), E(G))$ is a pair consisting of a set $V(G)$ of vertices and a set $E(G)$ of edges. We always talk about undirected and unweighted graphs. $N(C)$ is the 1-hop neighborhood from C not including $C$, formally defined as $N(C) := \{v \in V(H) \setminus C \mid \exists e \in E(H), u \in C$ which satisfies $\{u, v\} \subseteq e\}$, and $N[C] := N(C) \cup C$. A *tree* is a connected graph with no cycles. Many graph problems become simple to solve (even polytime solvable) when the input graph is a tree. The *treewidth of a graph, is an integer which measures, informally, how far the graph is from being a tree.* The smallest treewidth is 1 and the graphs with treewidth 1 are exactly trees and forests (disconnected graph with each component being a tree). The lower the treewidth, the faster one can solve the problem at hand. We now define the notion of tree decompositions of a graph that we will need to define treewidth of a graph. We then discuss the limitations of treewidth for hypergraphs and introduce other width measures for hypergraphs.

*Decompositions for graphs.* Let $G$ be a graph. A ***tree decomposition*** of $G$ is a pair $(T, \beta)$ where $T$ is a tree and $\beta$ is a function from the vertex set of $T$ to subsets of vertices of $G$. We shall refer to the vertices of $T$ using the term "nodes" and sets in the image of $\beta$ using "bags". For a tree decomposition to be valid, it needs to satisfy three properties.

- Each vertex $v$ in $G$ must belong to at least one bag of $T$.
- Each edge $e$ in $G$ must be a subset of at least one bag of $T$.
- For each vertex $v$ in $G$, the set $S_v$ of nodes of $T$ whose bag contains $v$ must induce a connected subgraph in $T$.

A ***tree decomposition rooted at*** $Z$ of $G$ is a triple $(T, \beta, Z)$, where $(T, \beta)$ is a tree decomposition of $G$ and $Z$ is a subset of vertices of $G$ such that $Z = \beta(x)$ for some node $x$ in $T$.

The ***width of a tree decomposition*** $(T, \beta)$ with respect to a function $f : 2^{V(G)} \to \mathbb{R}$ on the subset of vertices of $G$ is defined to be $\max_{x \in V(T)} f(\beta(x))$, that is the maximum value of $f(\beta(x))$ over all nodes $x$ in $T$. The ***width*** of $G$ with respect to $f$ is the maximum over the width of all tree decompositions of $G$. The ***treewidth*** of $G$ is then simply the width of $G$ when for each subset of vertices $S$, $f(S) = |S| - 1$, that is $f(S)$ represents the size of the set minus 1.

*Decompositions for Hypergraphs.* A ***hypergraph*** $H = (V(H), E(H))$ is a pair consisting of a set $V(H)$ of vertices and a set $E(H) \subseteq 2^{V(H)}$ of hyperedges. The ***Gaifman graph*** $G$ of a hypergraph $H$ is defined as: $V(G) = V(H)$ and $(u, v) \in E(G)$ if and only if $\{u, v\} \subseteq e$ for some $e \in E(H)$. A ***tree decomposition of a hypergraph*** $H$, is defined to be a tree decomposition of its Gaifman graph. So the ***width*** of $H$ with respect to a function $f : 2^{V(H)} \to \mathbb{R}$ is the width of its Gaifman graph with respect to $f$. In particular, the ***treewidth*** of $H$ is the treewidth of the Gaifman graph of $H$.

Observe that if the hypergraph contains only one hyperedge covering all vertices in $H$, then the treewidth of $H$ is $|V(H)| - 1$ which is large. But $H$ itself is a very trivial hypergraph. So we need other width measures that capture hypergraphs well. For this we introduce the notion of edge cover and fractional edge cover.

A set $C \subseteq E(H)$ is an ***edge cover*** of a set $S \subseteq V(H)$, if for every vertex in $S$ there exists at least one edge in $C$ that contains it. The ***edge cover number*** $\mathbf{cov}(S)$ of a set $S$ is the minimum cardinality of an edge cover of $S$. A function $h : E(H) \rightarrow [0, 1]$ is a ***fractional edge cover*** of $S \subseteq V(H)$, if for every vertex $v$ in $S$, $\sum_{e|v \in e} h(e) \geq 1$. The ***fractional edge cover number*** $\mathbf{fcov}(S)$ of a set $S$ is the minimum $\sum_{e \in E(H)} h(e)$ over all fractional edge covers $h$ of $S$. It can be shown that $fcov(S) \leq cov(S)$ and in fact an optimum fractional edge cover $h$ of $S$ with cost $fcov(S)$ can be computed in polynomial time by solving a linear program (LP).

The ***generalized hypertree width*** ($ghtw$) of $H$ is the width of $H$ with respect to the function $f : 2^{V(H)} \rightarrow \mathbb{R}$ with $f(U) = cov(U)$ for each $U \subseteq V(H)$ and the ***fractional hypertree width*** ($fhtw$) of $H$ is the width of $H$ with respect to $f$ when $f(U) = fcov(U)$ for each $U \subseteq V(H)$. See Figure 1 for an example of an hypergraph along with a tree decomposition and the different width measures for it. We now relate acyclic hypergraphs to $ghtw$.

Proposition 2.1. *A hypergraph is acyclic if and only if $ghtw = 1$.*

Acyclic hypergraphs can be recognized in linear time [50] but it turns out that $fhtw$ and $ghtw$ are NP-hard to compute even when they are equal to 2. Thus another common notion of width called ***hypertree width*** ($htw$) which is at most $3 \cdot ghtw$ was introduced by Gottolob et.al. in 2002 [21] for easy computation. It is similar to $ghtw$ but it does not use $cov$ instead it only uses edge covers with some special property. We now state our main objective below.

**Problem definition:** Given a hypergraph $H$ as input, the task is to compute a hypertree decomposition with minimum fractional (generalized) hypertree width. Since determining an optimal decomposition is NP-complete, a more practical objective is to find one that achieves approximately optimal width. Another goal is to find lower bounds for these width measures, to quantify how close the output decomposition is to the optimal one.

## 3 Polytime Approximation for *fhtw* - Overview

In this section we review the polynomial time approximation algorithm for computing fractional hypertree width by Korchemna et al. [34]. Their algorithm uses the now well known framework of Robertson and Seymour's [42], which we will refer to as *R&S*. We provide a quick overview of this framework and present the polynomial time algorithm of [34] in Algorithm 1.

*R&S Framework.* It was initially introduced in the context of computing treewidth of a graph $G$. The crux of this framework is –

- Begin by creating an empty set $Z$ which will become the root bag of the tree decomposition that we output, and add arbitrary vertices to $Z$ until it becomes "large."
- Find a "small" set $S$ such that every connected component $C$ of $G - S$ has at most half the vertices of $Z$.
- Recursively compute the decomposition of $C \cup N(C)$ for each *connected component $C$ of $G - S$*, with the new root bag being $(Z \cap C) \cup N(C)$.
- Glue together the decompositions of the subgraphs using $Z \cup S$ as an intermediate bag and finally glue this decomposition via $Z$ as shown in figure 2, to construct a decomposition for the input graph $G$.

A now standard proof by induction [9] can be used to show that the output constructed by the gluing procedure is indeed a valid tree decomposition of the input graph $G$. Note that during the recursive step, we set the new root bags to $(Z \cap C) \cup N(C)$, to ensure that the subtree of the tree decomposition induced by each vertex in $S$ is connected.

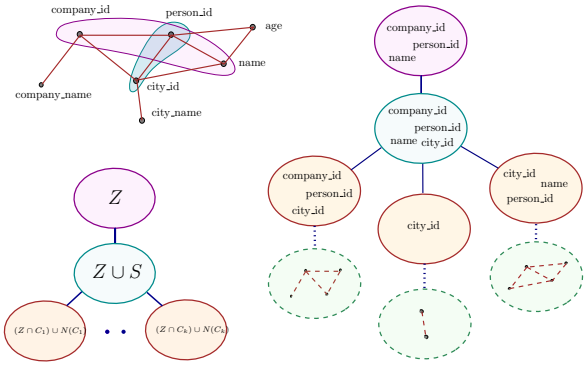



Fig. 2. (Top Left) An example graph $G$ where $Z$ = {company_id, person_id, name} and $S$ = {person_id, city_id}. (Bottom Left) "Glue" in Robertson and Seymour framework. (Right) "Glue" procedure implemented on $G$

*Vanilla R&S for fhtw computation from [34].* One of the key ideas needed to establish the approximation guarantee of the R&S algorithm for treewidth computation is that, for any graph $G$ of treewidth $k$ and any vertex subset $Z$, there exists a set $S$ of size at most $k+1$ such that every connected component in $G - S$ has at most half the vertices of $Z$. We maintain the invariant that the set $Z$ (root bag) under consideration satisfies $|Z| \leq 2k+2$. Thus we ensure that the new root bags $((Z \cap C) \cup N(C))$ in the recursive step have size at most $\frac{|Z|}{2} + |S| \leq 2k+2$. Hence when we recursively construct the decomposition for the smaller graph, the root bag will not keep getting bigger. The analogous statement for fractional hypertree width is given below as Lemma 3.1 and the proof of a slightly more general statement is presented in Lemma 6.1, [34].

Lemma 3.1 ([34], Lemma 6.1). *Let $H$ be a hypergraph of fractional hypertree width at most $\omega$ and $Z$ be a subset of vertices of $H$. Then, there exists a vertex subset $S$ with fractional cover number at most $\omega$ such that the fractional cover of $Z \cap C$ is at most half the fractional cover of $Z$ for every connected component $C$ of $H - S$.*

In [34], they also provide an algorithm to construct the set $S$ approximately. Specifically for $\phi \in (1/2, 1]$, they provide a polynomial time algorithm that takes as input a hypergraph $H$ with $fhtw = \omega$, a vertex set $Z$ and a fractional edge cover $\gamma : E(H) \rightarrow [0,1]$ of $Z$, and returns a $(\gamma, \phi)$-*balanced separator* $S$ of fractional cover number at most $O(\omega \cdot \log n \cdot \log \omega)$. Here a $(\gamma, \phi)$-**balanced separator** is a set $S$ for which $\sum_{e|e\cap C\neq\emptyset} \gamma(e)$ is at most $\phi \cdot \sum_{e|e\cap Z\neq\emptyset} \gamma(e)$ and consequently $fcov(Z \cap C) \leq 3\phi \cdot fcov(Z)$, for every connected component $C$ of $H - S$.

This algorithm can be used in conjunction with the R&S framework to construct a tree decomposition with fractional hypertree width at most $O(\omega \cdot \log n \cdot \log \omega)$. The pseudocode for a simple recursive version of such an algorithm is given in Algorithm 1 as Direct R&S FHTW.

Here, the BalancedSeparator function denotes the constructive algorithm mentioned above and Glue function executes the process depicted in Figure 2. Note that the decomposition for $H$ will be the output of Direct R&S FHTW$(H, \emptyset)$.

## 4 Our Algorithm - RALPH

In subsection 4.1, we provide an overview of our algorithm *Ralph* (**R**andomized **A**pproximation using **L**inear **P**rogramming for **H**yper-tree decompositions), which is presented in two parts as Algorithm 2 and Algorithm 3. Subsection 4.2 describes in detail, the methods that we use in Algorithm 2. Finally, in Subsection 4.3 we describe the computation of generalized hypertree width and lowerbounds for both fractional and generalized hypertree width.

**Algorithm 1** DIRECT R&S FHTW(H, Z)

```
1: if V(H) = Z then
2:     return tree decomposition of H[Z] with single bag Z
3: else
4:     if fcov(Z) < 3 ω log n log ω then
5:         v ← arbitrary vertex in V(H)
6:         T ← DIRECT R&S FHTW(H, Z ∪ {v})
7:         return GLUE ([T], Z)
8:     else
9:         S ← BALANCEDSEPARATOR (H, Z)
10:        for C_i in connected components of (H − (Z ∪ S)) do
11:            T_i ← DIRECT R&S FHTW (N_H[C_i], N_H(C_i))
12:        T ← GLUE ([T_1, ..., T_k], Z ∪ S)
13:        return GLUE ([T], Z)
```

### 4.1 Overview of *Ralph*.

A direct implementation of Algorithm 1, leaves room for improvement on multiple fronts. Firstly, note that for this approach, the algorithm needs prior information on the value of fractional hypertree width $\omega$ of the input $H$. Secondly, it always returns a tree decomposition of fractional hypertree width exactly $\min\{ fcov(H),\ 3\omega \cdot \log n \cdot \log \omega\}$. This is due to the step where we add an arbitrary vertex to $Z$ as long as $fcov(Z)$ is less than $3\omega \cdot \log n \cdot \log \omega$.

Hence, in our implementation we first try to find a balanced separator $S$ for $Z$ where $S$ is not a subset of $Z$ and $fcov(S \cup Z)$ is small. We implement the step of adding a vertex to $Z$ and recurse on $(H, Z)$ if and only if we cannot find such a separator. Otherwise we recurse on $(N[C], N(C))$ for every connected component $C$ of $H - (Z \cup S)$ and glue their outputs via $Z \cup S$ to get $(T, \beta, Z \cup S)$. This addresses both drawbacks of the direct implementation. Note that the output of the algorithm will be the decomposition obtained by gluing $(T, \beta, Z \cup S)$ via $Z$. The gluing procedure here is identical to the one in a classical R&S algorithm, and is depicted in Figure 2. ***The main component of the algorithm both theoretically and in terms of implementation is the step of finding such a separator.*** We implement this via a BALANCEDSEPARATOR function which has four different versions, each with a different advantage and use case. Before we get into the details of the function, we provide an overview of our implementation and the heuristics we apply.

The core of our implementation takes the form of a recursive function IMPROVED R&S FHTW which takes as input a hypergraph $H$ and a vertex subset $Z$, and returns a rooted tree decomposition $(T, \beta, Z)$ of $H$. The pseudocode of IMPROVED R&S FHTW(H, Z) is presented as Algorithm 2 below.

***Some simple optimizations.*** Similar to the direct implementation, IMPROVED R&S FHTW(H, Z) begins by checking whether $Z$ spans the whole hypergraph. If that is the case, we return the trivial decomposition of $H$ which is a single node whose bag is $Z$. If not, we check whether $H - Z$ has multiple connected components. If it does, we divide the instance into smaller ones each corresponding to a connected component $C$, by running IMPROVED R&S FHTW on $(N_H[C], N_H(C))$ and return the decomposition formed by gluing their output together via $Z$. If the previous cases do not occur, we check whether there are vertices of $Z$ that do not have any neighbors outside of $Z$. If so, we run IMPROVED R&S FHTW on $(N_H[C], N_H(C))$ where $C$ is the only connected component of $H - Z$ and return the decomposition formed by gluing its output via $Z$. Note that all of these

additional steps decrease the number of vertices and potentially lower the fractional cover number of $Z$ without incurring much computational overhead.

After these initial steps (1-10), we call the FracEdgeCover which uses an LP solver to find and return $\gamma : E(H) \to \mathbb{R}_{\geq 0}$, a fractional edge cover of $Z$ in $H$. Following this, we try to add vertices to $Z$ without increasing its fractional cover number. The motivation behind this heuristic is the observation that a set with more vertices has a higher chance of triggering the previous heuristics. Furthermore, since we only care about the fractional cover number of the bags, a vertex which does not increase this quantity can be freely added to $Z$. If a vertex $v$ with this property exists, the Expand function finds it and we recurse by setting the new root bag to be $Z \cup \{v\}$. Otherwise, we run the BalancedSeparator function on $(H, Z, \gamma)$ that tries to return, either a small $(\gamma, \frac{1}{2})$-balanced separator which is not entirely contained within $Z$, or a single vertex. We run the Improved R&S FHTW function on $(N_H[C_i], N_H(C_i))$ for each connected component of $H - (Z \cup S)$ and construct a single tree decomposition by gluing its output via $Z \cup S$. The tree decomposition that we output is constructed by gluing this to $Z$.

**Algorithm 2** Improved R&S FHTW(H, Z)

1: **if** $V(H) = Z$ **then return** trivial tree decomposition of $H[Z]$
2: **else**
3: $C_1, \ldots, C_k \leftarrow$ connected components of $(H - Z)$
4: **if** $k > 1$ **then**
5: **for** $C_i$ in $[C_1, \ldots, C_k]$ **do**
6: $T_i \leftarrow$ Improved R&S FHTW $(N_H[\, C_i],\ N_H(C_i))$
7: **return** Glue $([T_1, \ldots, T_k\,],\ Z)$
8: **if** $N_H(C_1) \neq Z$ **then**
9: $T \leftarrow$ Improved R&S FHTW $(N_H[\, C_1],\ N_H(C_1))$
10: **return** Glue $([T],\ Z)$
11: $\gamma \leftarrow$ FracEdgeCover $(H, Z)$
12: $Z' \leftarrow$ Expand $(H, Z, \gamma)$
13: **if** $Z' \neq Z$ **then**
14: $T \leftarrow$ Improved R&S FHTW $(H,\ Z')$
15: **return** Glue $([T],\ Z)$
16: $S \leftarrow$ BalancedSeparator $(H, Z, \gamma)$
17: **for** $C_i$ in connected components of $(H - (Z \cup S))$ **do**
18: $T_i \leftarrow$ Improved R&S FHTW $(N_H[\, C_i],\ N_H(C_i))$
19: $T \leftarrow$ Glue $([T_1, \ldots, T_k\,],\ Z \cup S)$
20: **return** Glue $([T],\ Z)$

***Warm start:*** We have *one additional heuristic at the beginning, where we set the initial root bag to be a separator with low fractional edge cover that splits the hypergraph into small instances.* For this, we arbitrary choose a small subset $U$ of $V(H)$, compute its fractional edge cover $\gamma$ and try to find a balanced separator $S$ for it. If the largest connected component of $H - S$ has size more than $0.7|V(H)|$, we pick $U$ again to be double in size and retry. It can be shown that, in any tree decomposition there exists a bag $B$ such that the largest connected component of $H - B$ has at most half the number of vertices in $H$. Consequently, using standard probability tail bounds, we can show that the expected number of vertices sampled from a component $C$ of $H - B$ is proportional to the number of vertices in $C$. As the size of the sample increases, with good probability, the bag $B$ will be

**Algorithm 3** WARM START(H)

```
k ← 5
while True do
    U ← k arbitrary vertices of H
    γ ← FRACEDGECOVER (H, U)
    S ← BALANCEDSEPARATOR (H, U, γ)
    C ← largest connected component of H − S
    if |C| ≤ 0.7|V(H)| then
        return IMPROVED R&S FHTW(H, S)
    else
        k ← 2 × k
```

a $(\gamma, \frac{1}{2})$-balanced separator where $\gamma$ is the fractional edge cover of the sample. Combining the above observations and using standard probability analysis, we can show that when $fcov(U)$ is within $poly(\log n)$ factors of $\omega$. Further we can successfully find such a separator with high probability, and consequently, the number of times we have to double the size of $U$ will be small. Furthermore, since there exists a $(\gamma, \frac{1}{2})$-balanced separator of fractional cover at most $\omega$, the one that we find will also have a small fractional cover. The pseudocode for this step is given in Algorithm 3.

### 4.2 Discussion of Methods in Algorithm 2.

***FRACEDGECOVER:*** The function takes as input a set $Z$, a hypergraph $H$ and returns the solution to the standard edge cover LP that we define below. Let us denote $E_v = \{e \in E(H) \mid v \in e\}$ and define a variable $y_e$ for each $e \in E(H)$ with domain $[0, 1]$. With this, we can formulate the following LP,

$$\begin{aligned} \text{Minimize :} \quad & \sum_{e \in E(H)} y_e \\ \text{Subject To :} \quad & \sum_{e \in E_v} y_e \geq 1 \qquad \forall v \in Z \end{aligned} \tag{1}$$

FRACEDGECOVER returns the mapping $\gamma : E(H) \to \mathbb{R}_{\geq 0}$, where $\gamma(e) = y_e$ for every edge $e$ in $E(H)$. A close examination of the constraints suffices to see that $\gamma$ is a min fractional edge cover of $Z$.

***EXPAND:*** The function takes as input a set $Z$, a hypergraph $H$ and if possible, adds vertices to $Z$ without increasing its fractional edge cover. A brute force search for such a vertex is expensive, since it involves re-solving the fractional cover LP corresponding to $Z \cup \{v\}$ for every vertex $v$ in $H$. Hence, in order to reduce the number of calls to the LP solver, *we first formulate and solve the fractional packing LP which is the dual of LP 1.* We define a variable $x_v$ for every vertex $v \in V(H)$ with domain $[0, 1]$ and formulate the following,

$$\begin{aligned} \text{Maximize :} \quad & \sum_{v \in Z} x_v \\ \text{Subject To :} \quad & \sum_{v \in e} x_v \leq 1 \qquad \forall e \in E(H),\ e \cap Z \neq \emptyset \end{aligned} \tag{2}$$

Now, consider a vertex $v'$ which does not belong to any edge corresponding to tight constraints. Observe that if $x_{v'}$ is added to the objective function, the optimum of LP 2 increases since we can set $x_{v'}$ to be non-zero without breaking any constraints. Since adding $v'$ to $Z$ increases the objective function of the dual LP, doing so would also increase the objective function of the primal LP. Consequently, only vertices belonging edges that correspond to tight constraints in the dual,

need to be considered. We iterate over all such vertices and check if adding it to $Z$ increases the fractional edge cover. If we find a vertex $v$ that can be added without increasing the fractional edge cover, we return $Z \cup \{v\}$, otherwise we return $Z$.

***BalancedSeparator:*** The function takes as input a set $Z$, a hypergraph $H$, an edge cover $\gamma$ of $Z$ and tries to find a $(\gamma, \frac{1}{2})$-balanced separator $S$ which is not a subset of $Z$, that minimises $fcov(Z \cup S)$. The key lemma here is that once $Z$ is "large" enough, BalancedSeparator always finds a separator which satisfies most of these requirements. The precise definition of "large," the time complexity of the algorithm, and the approximation guarantee for the fractional hypertree width all depend on the version of BalancedSeparator used, which is detailed in Section 5.

***Glue:*** The function takes as input a set of rooted tree decompositions $\mathcal{T}$, a set $U$ and returns a rooted decomposition $(T, \beta, U)$. Here, $(T, \beta, U)$ is the tree decomposition with a root node whose bag is $U$, which is adjacent to the root nodes of every $T_i \in \mathcal{T}$.

### 4.3 Lower Bounds and Generalized Hypertree width

***Lower Bounds:*** During the run of the algorithm if we find a set $Z$ and a fractional cover $\gamma$, for which the smallest $(\gamma, \frac{1}{2})$-balanced separator has fractional cover at least $k$, we can conclude that the fractional hypertree width of the hypergraph is at least $k$ (see Lemma 3.1). Hence, in our implementation, we also keep track of the maximum value of this lower bound over all fractional edge cover $\gamma$ the algorithm tries to separate, and outputs this value and its corresponding function $\gamma$ as a witness for the lower bound.

***Generalized Hypertree width:*** We can get an upper bound on the generalized hypertree width of the graph $H$ by computing an upper bound for the generalized hypertree width of the tree decomposition that we obtain. Note that this is not easy to implement exactly, since it involves finding the minimum edge cover of a bag which is equivalent to the NP-complete problem Set Cover. To overcome this, *we employ a heuristic method where we compute the fractional edge cover of a bag and pick edges, which has weight equal to one, into the edge cover. If there are vertices which remain uncovered after this step, we run the greedy algorithm [49] for* Set Cover *to compute an edge cover for it.* Observe that this edge cover serves as an upper bound on the minimum edge cover number of the bag. Furthermore, the maximum value of this bound over all bags, gives us an upper bound on the generalized hypertree width of $H$. Also, since fractional hypertree width is always smaller than generalized hypertree width [4], the lower bound that we obtain during our algorithm run can be rounded up to obtain a lower bound for generalized hypertree width.

## 5 Different Implementations of BalancedSeparator

*The* BalancedSeparator *function includes four versions, each with varying guarantees, resource requirements, and thus, different use cases.* The first and most computationally intensive version is one which formulates and solves a Mixed Integer Linear Program (MILP), equivalent to the LP by Korchemna et al. [34]. The main advantages of this approach are that it provides better upper and lower bounds, compared to the other versions, and outputs a decomposition whose fractional hypertree width is at most a factor of three away from the optimal value. Then, we have two versions that formulate a Linear Program (LP), solve it and feed the solution to a rounding algorithm which outputs an integral solution. Both LPs are equivalent to, and have the same approximation guarantee, as their algorithm. However, they differ in size, with the larger one, referred to as the Boosted LP, being more accurate yet more computationally intensive than the smaller version. Note that the formulation of Boosted LP is novel and was constructed to give better accuracy compared to the LP from [34]. Finally we have a heuristic version, which does not

formulate an LP but rather, provides a near uniform weight function to the rounding algorithm and uses its output. The output decomposition of an algorithm using this version does not have any theoretical approximation guarantees; however, it provides meaningful upper bounds for large instances where none were previously known. We remark that Korchemna et al. claim a runtime of $n^{O(1)}$. A naive analysis of our implementation gives an upper bound of $n^{10}$ for the LP versions and $n^5$ for the heuristic. It is important to note that this bound does not reflect the practical performance of *Ralph*, as LP solvers perform significantly better in practice than their worst-case bounds. Moreover, many factors of $n$ in the bound can be replaced with terms polynomial in the number of non-zero variables in the LP, which is often much smaller.

In this section, we go over the MILP and LP relaxation based versions in Subsections 5.1 and 5.2 respectively. We then describe the rounding algorithm used by the LP, Boosted LP and heuristic versions in Subsection 5.3, and conclude with a summary of the heuristic version in Subsection 5.4. *Throughout this section, we frequently use the following notations and definitions. Let,* $\gamma : E(H) \to \mathbb{R}_{\geq 0}$ *be the edge cover of* $Z$ *in* $H$*, given as input to* BalancedSeparator *and* $G$ *be the Gaifman graph of* $H$*. We denote by* $F(H)$ *the edges* $e \in E(H)$ *for which* $\gamma(e)$ *is positive. Finally, for ease of notation, we use* $\gamma(U)$ *to refer to* $\sum_{e|e\cap U\neq\emptyset} \gamma(e)$ *where* $U$ *is a subset of vertices of* $G$*.*

### 5.1 MILP based Balanced Separator

In this version of the function, we check for the existence of a $(\gamma, \frac{1}{2})$-balanced separator which is not a subset of $Z$ by constructing and solving a Mixed Integer Linear Program (MILP). We define a variable $x_v$ that can take a value of zero or one, for each vertex $v \in V(H)$, indicating whether $v$ is included in the separator or not. We have $y_e$ for every edge $e \in F(H)$, taking values in $[0, 1]$, which represents the fractional edge cover of the separator. Finally, we have $d_{e,v}$, $d_{e,e'}$ for vertices in $V(H)$ and edges in $F(H)$, taking values in $[0, 1]$, which serve as proxies for the separation between a vertex-edge pair and an edge-edge pair once the separator is deleted. The MILP formulated by the algorithm is presented in MILP 3,

The output of this version is the set of vertices corresponding to $x_v$ variables that are assigned a value one by the MILP solver. We can prove that this set satisfies the desired properties by observing the relation between the constraints and desired properties. This proof is similar to the argument used in Observation 5.15 of [34].

$$\text{Minimize :} \quad \sum_{e\in E(H)} y_e \tag{3}$$

$$\begin{aligned}
\text{Subject To :} \quad & x_v \leq \sum_{e\in E_v} y_e && \forall v \in V(H) \\
& d_{e,v} \leq x_v && \forall e \in F(H),\ v \in e \\
& d_{e,v} \leq 1 && \forall v \in V(H),\ e \in F(H) \\
& d_{e,v} \leq d_{e,u} + x_v && \forall e \in F(H),\ (u, v) \in E(G) \\
& d_{e,e'} \leq d_{e,v} && \forall e, e' \in F(H),\ v \in e' \\
& \sum_{e'\in F(H)} \gamma(e')\, d_{e,e'} \geq \frac{\gamma(Z)}{2} && \forall e \in F(H) \\
& \sum_{v\in V(H)\setminus Z} x_v \geq 1 && (*)
\end{aligned}$$

*Comparison with LP by Korchemna et al. [34]* Observe that once we remove constraint $(*)$ from the above MILP, a standard inspection will be enough to prove that the LP version of the above MILP is equivalent to theirs. The major differences between the two are twofold. First, the size of this MILP is often significantly smaller, enabling it to be solved much more quickly in many practical situations. Second, the $(*)$ constraint ensures that at least one vertex outside $Z$ is included in the separator, which empirically helps the algorithm make more progress. All these empirical improvements are theoretically supported by Lemma 5.1.

Lemma 5.1. *Let $H$ be a hypergraph of fractional hypertree width $\omega$ and $\gamma : V(H) \to \mathbb{R}_{\geq 0}$ be a fractional edge cover of $Z$, where every vertex of $Z$ has a neighbor outside of $Z$, and $H - Z$ has a single connected component. Then there exists a $(\gamma, \frac{1}{2})$-balanced separator $S$ of $Z$ such that the fractional edge cover of $S$ is at most $\omega$ and $S$ is not a subset of $Z$.*

Furthermore, observe that if such a separator $S$ exists then the MILP has a solution where $x_v$ is set to be one for every vertex in $S$, $y_e$ is an optimum fractional edge cover of $S$ and $d_{e,v}, d_{e,e'}$ is zero if they belong to the same component of $H - S$ and one otherwise. Hence, in this version of the BalancedSeparator function we never have to resort to returning an arbitrary vertex since we always find a separator with the desired properties. Furthermore, since the MILP directly finds an integral balanced separator without resorting to rounding algorithms, we can show that it outputs a tree decomposition with fractional hypertree width at most three times the optimum value.

Theorem 5.2. *The output decomposition of the algorithm which uses the above version of* BalancedSeparator *will have fractional hypertree width at most $3\omega$, where $\omega$ is the minimum fractional hypertree width of the input hypergraph $H$.*

Note that although the approximation guarantee of this version is the smallest, it is computationally expensive (exponential time).

### 5.2 LP based Balanced Separators

Both the normal LP and Boosted LP versions of the algorithm have the same structure. They both formulate an LP, solve it and use this solution to construct a separator $S$ to output. Unlike the MILP, since every variable can take fractional value, the solution does not immediately give us a separator. Instead, we provide the value assigned to variables as input to a Round function which produces the set $S$ that we output as our separator. The procedure that we use to round the fractional solution uses a ball growing argument similar to that of Leighton and Rao [37] and is explained in more detailed in Subsection 5.3. Also, note that the optimum value to the LP both normal and boosted acts as a lower bound to the fractional edge cover of an integral $(\gamma, \frac{1}{2})$-balanced separator. Finally, the approximation guarantees for both of these versions can be derived, with some additional work, from the proof of Theorem 1.1 by Korchemna et al. [34].

***Normal LP relaxation.*** In this version, we run an LP solver on MILP 3, which we refer to in the following sections as the *normal LP*, formulated corresponding to the input $(H, Z, \gamma)$. We run the Round function multiple times on this LP solution, and output the one for which fractional edge cover of $Z \cup S$ is minimized. Note that, even though the fractional solution assigns non-zero values for some vertices outside of $Z$, the Round function might return a set which is contained inside $Z$, and consequently, the separator $S$ that we pick might end up being a subset of $Z$. If so, we pick a vertex from $V(H) \setminus Z$ with probability proportional to the value of the LP variable $x_v$ and return it, instead of $S$. The motivation behind this heuristic is the observation that the value assigned to $x_v$ variables reflects the importance of a vertex in separating $Z$. Since separating $Z$ and reducing its fractional cover is always beneficial, we obtain empirical advantages by favoring vertices that help

separate $Z$. Note that this version of the BalancedSeparator function takes polynomial time and consequently runs a lot faster in practice when compared to the MILP version.

**Boosted *LP relaxation*.** In this version, we formulate a new LP, which we refer to as the *boosted LP*, corresponding to the input $(H, Z, \gamma)$ and solve it. Similar to the normal LP version, we run the Round function multiple times on this LP solution, and pick the one for which fractional edge cover of $Z \cup S$ is minimized. Finally, if the separator $S$ that we pick ends up being a subset of $Z$, we select a vertex $v$ from $V(H) \setminus Z$ at random, with probability proportional to the value of the LP variable $x_{v,v}$ and return it, instead of $S$.

$$
\begin{aligned}
\text{Minimize :} \quad & \sum_{e \in E(H)} y_e && && (4)\\
\text{Subject To :} \quad & x_{v,v} = \sum_{e \in E_v} y_e && \forall v \in V(H)\\
& x_{u,v} = \sum_{e \in E_v \setminus E_u} y_e && \forall (u,v) \in E(G)\\
& d_{e,v} \leq x_{v,v} && \forall e \in F(H),\ v \in e\\
& d_{e,v} \leq 1 && \forall v \in V(H),\ e \in F(H)\\
& d_{e,v} \leq d_{e,u} + x_{u,v} && \forall e \in F(H),\ (u,v) \in E(G)\\
& d_{e,e'} \leq d_{e,v} && \forall e, e' \in F(H),\ v \in e'\\
& \sum_{e' \in F(H)} \gamma(e')\, d_{e,e'} \geq \frac{\gamma(Z)}{2} && \forall e \in F(H)\\
& \sum_{v \in V(H) \setminus Z} x_{v,v} \geq 1
\end{aligned}
$$

Now, let us discuss the LP formulation. Recall that, $F(H)$ is the subset of edges $e \in E(H)$ for which $\gamma(e)$ is positive and $G(H)$ is the Gaifman graph of $H$. We define a variable $x_{v,v}$ for each vertex $v \in V(H)$, taking values in $[0, 1]$ indicating whether $v$ is included in the separator or not. We have $y_e$ for every edge $e \in F(H)$, taking values in $[0, 1]$, which represents the fractional edge cover of the separator. We define $x_{u,v}$ for every edge $(u, v)$ in $E(G)$, taking values in $[0, 1]$ indicating the "cost" of going from $u$ to $v$. Finally, we have $d_{e,v}$, $d_{e,e'}$ for vertices in $V(H)$ and edges in $F(H)$, taking values in $[0, 1]$, which serve as proxies for the separation between a vertex-edge pair and an edge-edge pair once the separator is deleted. The LP formulated by the algorithm, is given in LP 4,

***Comparison with normal LP.*** Observe that in both the normal LP and the boosted LP, the $y$ variables are the free variables and all of the rest are dependant variables. This means that fixing the value assigned to $y$ variables determines the assignment to all the variables. Using this, we note that any feasible solution $(\hat{y})$ to the above LP can be converted into a feasible solution $(\hat{y}')$ to the normal LP by setting $\hat{y}'_e = \hat{y}_e$. Hence the set of feasible solutions to the boosted LP is a subset of the set of feasible solutions to the normal LP, which implies that the boosted LP has considerably less fractional solutions. It also holds that they have the same number of solutions where $x$ variables takes a value of zero or one, as any such solution to the normal LP, is also a solution to the boosted one. Hence, the solution found by the boosted LP is expected to be more representative of the separator we are searching for. This is the main advantage that the boosted LP has over the normal LP that helps it find better lower and upper bounds. It is computationally more expensive than

normal LP due to its larger size, but remains significantly less expensive than the MILP, as it runs in polytime.

### 5.3 Round function

The rounding algorithm takes as input the hypergraph $H$, a set $Z$, a function $\gamma : E(H) \rightarrow \mathbb{R}_{\geq 0}$ which is a fractional edge cover of $Z$, the values $(\hat{y})$ assigned to variables of LP 3 or 4, and outputs a $(\gamma, \frac{1}{2})$-balanced separator $S$ for $H$. The rounding procedure that we implement is inspired from [34] but is more optimized for practice. We implement heuristical improvements which leads to gains in both the computational requirements and output quality. Also we note that our implementation with slight modifications has the same theoretical guarantee as the rounding procedure described in [34]. Our implementation has an outer layer and an inner layer:

*Outer Layer.* We define any component $C$ with $\gamma(C) > \frac{1}{2}\gamma(Z)$ to be a *bad component* and begin the outer layer by setting $S$ to be the empty set. In the outer layer, we check if $H - S$ has a bad component $C$, and if it does, set $S$ to be $N(C)$. We then run the inner layer of the algorithm on $C$, which returns a small separator $S'$ for $C$, and recursively call the outer layer after setting $S$ to be $S' \cup S$. Note that the recursion has the property that any bad component in a recursive call is a strict subset of the current bad component. Now, when we find a set $S$, such that $H - S$ has no bad components, we create a new set $C$ and initialize it as the empty set. We greedily add small components of $H - S$ to $C$, until we reach a point where addition of one more component results in $\gamma(C)$ exceeding $\frac{1}{2}\gamma(Z)$. The final output of the Round function will be $N(C)$. Note that the algorithm always terminates in $O(n)$ steps, since the bad component in the outer loop gets smaller every iteration and it always returns a $(\gamma, \frac{1}{2})$-balanced separator.

*Inner Layer.* To explain the details of the inner layer, need a few notations and definitions. We define $E_1(U)$ as the set of edges $e \in E(H)$ contained within $U$, and $E_2(U)$ as the set of edges that intersect $U$, where $U$ is a subset of vertices of $H$. Let $H_C$ be the subhypergraph of $H$ with vertex set $C$ and edge set $E_2(C)$, and $G$ be the the Gaifman graph of $H_C$. We convert $G$ to an edge weighted digraph by replacing every undirected edge $(u, v) \in E(G)$ with directed edges $(u, v)$ and $(v, u)$. The edge $(u, v)$ in $G$ is assigned weight $\hat{x}_v$ when using the solution to the normal LP and $\hat{x}_{u,v}$ when using the boosted LP. The inner layer picks an edge $e$ in $C$ with probability proportional to $\gamma(e)$ and runs Dijkstra's algorithm on $G$ to compute the distance $d_v$ of every vertex $v$ to the edge $e$. Note that $d_v$ will be at most one for every vertex $v$. Now, corresponding to every vertex $v$, we assign the interval $[d_{u^\star}, d_v]$, where $u^\star$ is the in-neighbour of $v$ with minimum $d_u$ value. Let $S_i$ for some $i \in [0, 1]$ be the set of vertices whose interval contains $i$. We iterate over the set of all distinct $S_i$ sets and assign a *gain* and *loss* value for each of them. Finally, we return the separator $S'$ which minimizes the $loss(S') : gain(S')$ ratio.

To discuss the details of how *gain* and *loss* values are computed, we introduce some notations. For a vertex $v$ in $H$, let $\gamma(v)$ denote $\sum_{e|v \in e} \gamma(e)$. Also, for an edge $e$ and a set $U \subseteq V(H)$, let $\tilde{y}_e(U)$ be the maximum value of $\hat{y}_e(1 - \gamma(v))/\hat{x}_v$, over every vertex $v$ in $e \cap U$, if $e$ has a non-empty intersection with $U$, and zero otherwise.

***Gain:*** Consider the quantity $\sum_{e \in E_2(C)} \hat{y}_e$, where $C$ is some connected component of $H - S$. We claim that, if it equals zero, then $C$ cannot be a bad component and consequently, the change in this quantity is a good measure of how closer we are to removing all bad components when adding a specific set $S_i$ to $S$. Motivated by this, we define $gain(S_i)$ to be $\sum_{e \in E_2(C)} \hat{y}_e - \sum_{e \in E_2(C')} \hat{y}_e$.

***Loss:*** The *loss* that we incur while adding $S'$ to the separator $S$ is the increase in its fractional edge cover. We claim that, setting $\tilde{y}_e(S)$ to be maximum of $(1 - \gamma(v))\hat{y}_e/\hat{x}_v$, over vertices of $e \cap S$,

will ensure that every vertex of $S \cup Z$ gets covered by the function $\gamma'(e) = \gamma(e) + \tilde{y}_e(S)$. This motivates our definition of $loss(S_i)$ as $\sum_{e \in E_1(S_i)} \tilde{y}_e(S \cup S_i) - \sum_{e \in E_1(S)} \tilde{y}_e(S)$, since it is an upper bound for the actual quantity we want to compute.

### 5.4 Heuristic based Balanced Separator

Observe that, the rounding algorithm by virtue of its terminating condition, always returns a $(\gamma, \frac{1}{2})$-balanced separator in $H$, irrespective of the vector $(\hat{y})$ given as input. The only way the rounding algorithm depends on $(\hat{y})$ is in terms of approximation guarantee. Specifically, if $(\hat{y})$ is the solution to a balanced separator LP, such as LP 4 or 3, then we have a $O(\log n \cdot \log \omega)$ upper bound. Hence, in this version of the algorithm we provide a vector $(\hat{y})$ as input to the rounding function, which may or may not even be a feasible solution to the balanced separator LPs, and use the $(\gamma, \frac{1}{2})$-balanced separator it outputs. Even though this does not have any theoretical guarantees, we were able to establish reasonable upper bounds for all instances that previous algorithms had no insights on. To go into the details of the specific $(\hat{y})$ that we provide as input to the rounding algorithm, let us define $\gamma'$ as a fractional edge cover of $G$ computed using LP 1. Observe that, in an LP solution, the value taken by a $y_e$ variable is an indicator for how important the edge $e$ is in covering the output $(\gamma, \frac{1}{2})$-balanced separator. Also, it seems reasonable to believe that, in practice, if an edge plays an important role in covering the whole graph or the set we are trying to separate, then it also plays a significant role in covering the separator. Motivated by this, we set $\hat{y}_e$ to be $\frac{\gamma(e)}{2}$ + $\frac{\gamma(G)}{2}(\frac{10\gamma'(e)}{\gamma'(G)})$. The constants are chosen to work well in practice, as discussed in Subsection 6.2.

## 6 Experimental Setup

### 6.1 HyperBench and Previous Benchmarks

We tested the performance of our algorithm *Ralph* against the HyperBench dataset [12], which contains 3682 hypergraphs reduced from instances of CQs and CSPs found in both industry and academia. The majority of these instances can be found on the HyperBench website [1], while the rest are from a recent benchmark [18]. We have compiled these instances into our experimental data [2]. A quick summary of these hypergraphs is shown in Table 1.

Since the time of publication (2019), there have been a number of algorithms benchmarked on HyperBench. These algorithms find fractional hypertree width (*fhtw*), generalized hypertree width (*ghtw*), and hypertree width (*htw*) of hypergraphs. The current state-of-the-art algorithms are:

- *FraSMT* (2018) [11] Exact *fhtw*, exact *ghtw*
- *Triangulator* (2020) [35] Exact *fhtw*, exact *ghtw*
- *HtDLeo* (2022) [44] Exact *htw*
- *log-k-decomp* (2022) [18] Parameterized *ghtw*

All of these algorithms find decompositions for a different subset of the HyperBench instances, within a time limit of 60 minutes for each instance. Amongst them, *log-k-decomp* is unique in that it returns a lower bound, even when it is unable to find the optimal solution. This is due to the parameterized nature of the underlying algorithm, where it takes an integer $k$ and a hypergraph $H$ as input and checks if the *ghtw* of $H$ is equal to $k$. *log-k-decomp* initially sets $k = 1$ and runs the algorithm, incrementing $k$ until a decomposition is found or the time limit is hit. If a run terminates without finding a decomposition, we obtain a lower bound, and if it succeeds, we get an upper bound for *ghtw*. But, note that if a run times out, then we do not get any information on whether

[1] http://hyperbench.dbai.tuwien.ac.at
[2] https://github.com/strongani/Ralph-hypertree-width

a decomposition of *ghtw k* exists or not. Additionally, we observe that each run (instead of each instance) of *log-k-decomp* with a fixed $k$ is allocated 60 minutes.

### 6.2 Experimental Design

As the nature of our algorithm is both randomized and approximate, every run on the same instance will give a different output. Therefore, to test the effectiveness of *Ralph*, we ran it on each instance multiple times. For a fair comparison, we stopped our algorithm if either $(i)$ we already ran it 60 times or $(ii)$ we hit 60 minutes of compute power while running it. Notice that these constraints only allow for a proportionate time to be spent on each instance compared to how large the instance is. The upper bound of 60 runs is chosen to spend a reasonable amount of time on small instances: if the upper bound of 60 runs is hit before the timeout of 60 minutes, we know the instance, on average, runs within a minute of compute, and is therefore relatively small/easy. The timeout of 60 minutes is consistent with the four benchmarks we compare against.

As explained in Section 5, we have four possible versions of the algorithm to run: *Heuristic, Normal LP, Boosted LP, and MILP (Mixed ILP)*. So, we split our 60 possible runs for each instance into the following: Runs 1-10 using the Heuristic, 11-30 using the Normal LP, 31-50 using the Boosted LP, and 51-60 using the MILP. Notice that the runs of each new variation increase in both time and accuracy. The decision to do this "mixed" run of each instance is motivated by the preliminary tests we conducted. In particular, we observed that many small graphs were able to not only finish 60 runs of the Normal LP, but also finish this "mixed" run, including 10 MILP runs. On the other hand, large graphs could finish 60 Heuristic runs, yet were able to do only a couple of Normal LP runs before hitting the 60 minutes time limit. We discuss and analyze the different versions of our algorithm, as well as our run distribution in Subsection 7.4.

*Hyperparameters.* In the code, there are many hyperparameters, such as the stopping condition for the warm start (Algorithm 3), the values of $\hat{y}_e$ in the Heuristic version (Subsection 5.4) and the number of rounding attempts (Subsection 5.2). All of these can be tuned in practice to get better empirical performance, but their choice does not impact the algorithm's theoretical guarantee. We chose values for our hyperparameters after analyzing preliminary runs on a various graphs. We observed that moderate changes to the parameters did not significantly impact results, and the large hyperparameter space made grid search infeasible and distracting from our original focus. That being said, an important hyperparameter we tweak later in Subsection 7.3 is the number of rounding attempts. Given variable assignments to the balanced separator LP, this value determines the number of times we round before taking the best separator. This allows a trade-off between compute time and solution quality, making it important in various settings.

### 6.3 Code and Environment

Our codebase is publicly available [1]. We ran these instances in batches, running 16 instances multithreaded at a time on the HPC server at the UCSB Center for Scientific Computing at CNSI. Each batch had 16 physical cores and 16 logical processors allocated to it, using an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz. We had a limit of 20 GB RAM. That being said, the runs of our algorithm were noticed to have minimal RAM usage. Our implementation is in Python 3.12.3 for simplicity. When profiling the code, we noticed that the bulk of the running time is consumed by library calls to an LP solver. As such, implementing in a relatively lower-level language, such as C++, is unlikely to boost performance significantly. Finally, for solving both the LP and MILP we used the Gurobi Optimization [24], with access through the Python library PuLP for convenience [2].

[1] https://github.com/strongani/Ralph-hypertree-width
[2] https://github.com/coin-or/pulp

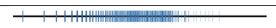
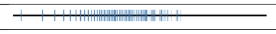
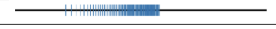
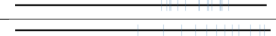
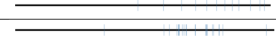
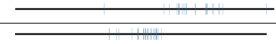
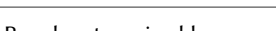
| Source Type<br>*Subtype* | Num<br>Graphs | Num Vertices<br>1 3 10 30 100 300 1000 3000 |
|---|---|---|
| CQ | 1146 |  |
| *SPARQL* | 70 |  |
| *SQLShare* | 290 |  |
| CQ Random | 500 |  |
| CSP Application | 1090 |  |
| CSP Random | 863 |  |
| CSP Other | 82 |  |
| *DaimlerChrysler* | 15 |  |
| *Grid2D* | 12 |  |
| *ISCAS89* | 24 |  |
| *MaxSAT* | 31 |  |
| Other | 1 |  |
| **Total** | **3682** |  |

Table 1. Instances in HyperBench categorized by source type.

## 7 Experimental Results

### 7.1 Raw Data of our Results

The raw data of our results is publicly available [1]. *Format of data:* for each run of an instance, we store an output file containing the following: *fhtw* lower bound, *fhtw* upper bound, *ghtw* upper bound, hypertree decomposition, balanced separator version, a witness for the lower bound, and some associated statistics.

Recall that, our *ghtw* upper bound is from heuristic rounding of our fractional edge covers for each bag and the lower bound is obtained by rounding up our *fhtw* lower bound. The witness for the lower bound is a set $Z \subseteq V(H)$ whose optimal Balanced Separator LP/MILP value achieves the lower bound. Finally, throughout the section, when we mention the *fhtw*/*ghtw* upper/lower bound we get for an instance, we are referring to the best bound over the multiple runs of the instance within the run/time limit.

### 7.2 Main Results: Comparison

Now, our algorithm *Ralph* is randomized and approximate, while all the other algorithms we compare to are exact and deterministic. Thus, for ease of comparison, we partition the instances of HyperBench into four categories, for each algorithm:

- *solved*. We say an instance is *solved* if the exact value of the parameter (either fractional or generalized hypertree width) is found by the algorithm of interest. For *Ralph* and *log-k-decomp*, this corresponds to an instance for which they find equal upper and lower bounds within the time limit. For the exact algorithms, this corresponds to an instance which terminates within the time limit.
- *almost solved*. We say an instance is *almost solved* if the algorithm of interest (*Ralph* or *log-k-decomp*) finds an upper bound and lower bound of that instance with difference at most 2.
- *bounded*. We say an instance is *bounded* if the algorithm of interest finds any non-trivial upper bound of that instance. For *Ralph*, we specifically differentiate bounded LP instances, where the algorithm had at least one Normal LP run, and bounded Heuristic instances, where it only had Heuristic runs. This differentiation is important as the algorithm provides theoretical guarantees for the approximation ratio only in non-Heuristic runs.
- *failed*. We say an instance *failed* when no bound is found by the algorithm of interest.

[1] https://github.com/strongani/Ralph-hypertree-width

The results of *Ralph* with this terminology are shown in the first plots of Figures 3 and 5. Notice we find some non-trivial upper bound for **all** instances, and almost solve a majority of the instances.

We now compare the performance of *Ralph* with previous algorithms. As we run *Ralph* 60 times, while the exact algorithms run only once and the parameterized algorithms rerun until a decomposition is found, all of these within a 60 minute time limit, we do not compare running times. Instead, as the 60 minute time limit is for each algorithm, we compare the quality of widths found by each.

*Fractional Hypertree Width.* Out of the four benchmarks we compare against, only *FraSMT* and *Triangulator* compute fractional hypertree width. Both of these algorithms are exact. We use the results from [35] for both algorithms, and as such, both algorithms have been ran on the same set of instances.

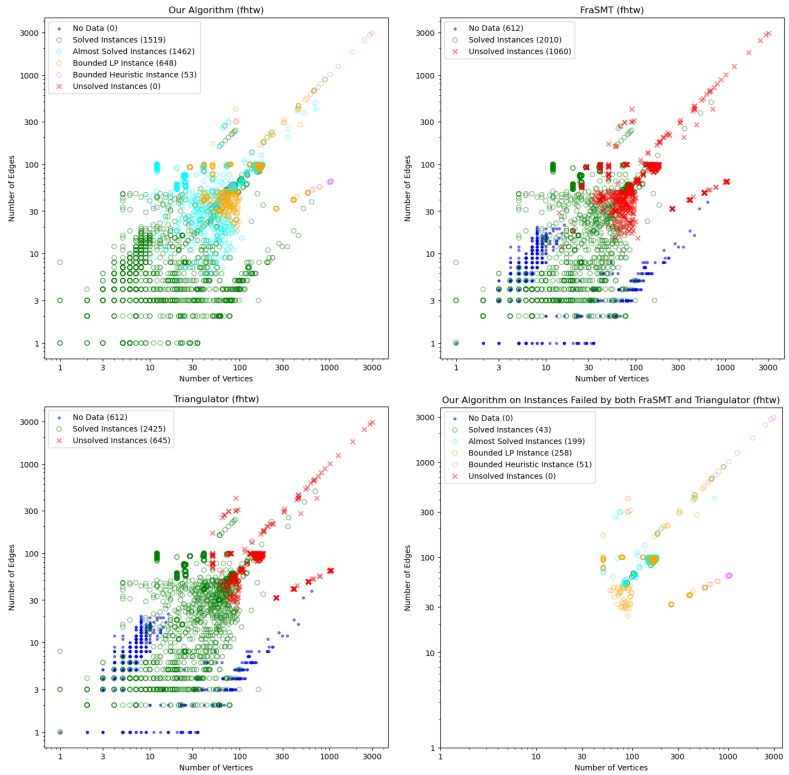



Fig. 3. Performance of *FraSMT* and *Triangulator* in comparison to *Ralph* for *fhtw*. The instances are categorized as discussed in Subsection 7.2. *Ralph* obtains upper bounds for 551 instances failed by both *FraSMT* and *Triangulator*.

For *fhtw*, *FraSMT* solves 2010 instances, and fails on 1060. *Triangulator* solves 2425 instances, and fails on 645. Note that *Ralph* has bounds for all of these instances, as shown in the first plots of Figures 3 and 5. Furthermore, for each instance solved by either *FraSMT* or *Triangulator*, we calculate the difference between our upper bound for that instance and the exact value of *fhtw* for that instance. Interestingly, this difference never exceeds 2: See Table 2.

This data suggests that *Ralph* performs close to state-of-the-art for the solved instances, solving more than half of them, and getting upper bounds at most one away for the rest of them, barring a small percentage. For the instances which both *FraSMT* and *Triangulator* failed, we bound all of them, and (almost) solve half. Notice we are obtaining non-trivial upper bounds for 551 instances which have **not** had non-trivial upper bounds prior to this work: See Figure 3.

| Algorithm | # Solved | Our *fhtw* ub - exact | | |
|---|---|---|---|---|
| | | 0 | $(0, 1]$ | $(1, 2]$ |
| *FraSMT* | 2010 | 1304 | 702 | 4 |
| *Triangulator* | 2425 | 1317 | 1100 | 8 |
| **Total** | 2519 | 1380 | 1130 | 9 |

Table 2. Number of Instances and their *fhtw* upper bound (ub) difference which is (our ub - exact value)

In summary, our algorithm finds near-optimal fhtw upper bounds for the 2519 solved instances by *FraSMT* and *Triangulator*, while bounding all of the 551 unsolved instances.

*Generalized Hypertree Width.* All four algorithms compute either generalized hypertree width or hypertree width. It is known that $htw \leq 3 \cdot ghtw$ [4] and hence, *HtDLeo*, which computes exact *htw*, can be regarded a 3-approximation algorithm for *ghtw*. Furthermore, comparing *FraSMT* and *Triangulator*, we see that *Triangulator* solves all 1494 instances solved by *FraSMT*, and solves 1050 out of 1576 instances it fails on. Thus, *FraSMT* can be directly omitted from our comparisons. As such, we will compare *Triangulator* (exact *ghtw*), *log-k-decomp* (parameterized *ghtw*), and *HtDLeo* (3-approx *ghtw*) with *Ralph* (randomized approx *ghtw*).

Since *Triangulator* is an exact algorithm, we will start by comparing the other algorithms to it: See Table 3. Note *Triangulator* solves 2544 instances and fails on the other 526 instances.

- *Triangulator* vs *log-k-decomp:* Out of the for 2511 instances solved by *Triangulator* and attempted by *log-k-decomp*, it solves 2492 instances, fails on 13 and bounds the remaining 6. Furthermore, *log-k-decomp* solves 304 instances that *Triangulator* fails. So, while *Triangulator* is exact, *log-k-decomp* performs better in practice.

- *Triangulator* vs *HtDLeo:* Even though *HtDLeo* computes *htw*, the results show that all the 1910 solved by both *Triangulator* and *HtDLeo* have equal *ghtw* and *htw*. Nonetheless, *Triangulator* outperforms *HtDLeo*, solving *601* instances that *HtDLeo* fails on, while *HtDLeo* only solves 57 instances that *Triangulator* fails on.

- *Triangulator* vs *Ralph*: For 2466 out of the 2544 instances solved by *Triangulator*, *Ralph* either solves or is one away from the exacts value for each. It is also never more than 3 away from the exact value for the instances solved by *Triangulator*. Furthermore, *Ralph* bounds all instances, including all which *Triangulator* fails.

| Algorithm | B | F | B & *T* F | F & *T* S | Alg *ghtw* ub - exact | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | 0 | 1 | 2 | 3 |
| *log-k-decomp* | 3414 | 235 | 304 | 13 | 2492 | 5 | 1 | 0 |
| *HtDLeo* | 2545 | 1104 | 57 | 601 | 1910 | 0 | 0 | 0 |
| *Ralph* | 3682 | 0 | 526 | 0 | 1591 | 876 | 71 | 6 |

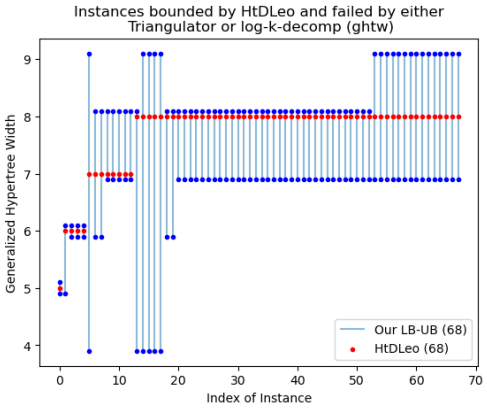
Table 3. *ghtw* algorithms compared to *Triangulator*. B stands for bounded, F stands for failed, S stands for solved, and *T* stands for *Triangulator*.

We now consider *HtDLeo* and see how it compares to *Ralph*. Consider, all instances bounded by *HtDLeo* but failed by either *Triangulator* or *log-k-decomp*. There are only 68 of these instances, and we solve or almost solve 62 of them: See Figure 4a. So, *Ralph* with either *Triangulator* or *log-k-decomp* outperform *HtDLeo*.

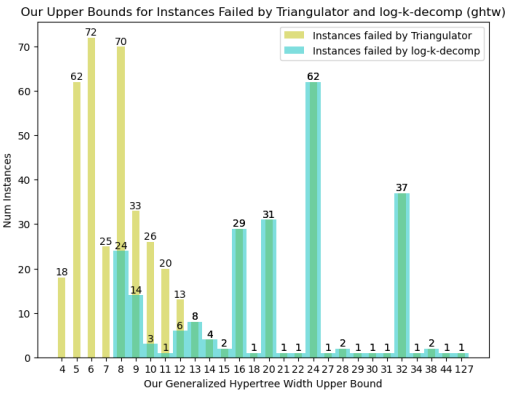
Now, considering *Triangulator* and *log-k-decomp*, if we look at the instances failed by either, we see that all are bounded by our algorithm, with the majority being LP bounded: See Figure 5.

Let's now compare *log-k-decomp* with our algorithm. From Figure 5, we see *log-k-decomp* solves 3251, almost solves 139, bounds 24 and fails on 235. In comparison, we only solve 1653, almost solve 1344, LP bound 632 and Heuristic bound 53. So, for the majority of graphs, *log-k-decomp* outperform our algorithm and have tigher upper bounds. However, we bound all the 235 instances failed by *log-k-decomp*. We further note that all these instances have large *ghtw*: See Figure 4b.

Therefore, even though *Ralph* is outperformed by both *Triangulator* and *log-k-decomp* in known instances, it still computes near-optimal *ghtw* for those instances, while bounding all of the 222 large failed *ghtw* instances. We argue that our algorithm has comparable performance for small/known instances, while clearly pushing state-of-the-art for large/unknown instances.




(a) *Ralph* on instances bounded by *HtDLeo* but failed by either *Triangulator* or *log-k-decomp*.




(b) *Ralph*'s Upper Bounds for Instances Failed by *Triangulator* and *log-k-decomp*.

Fig. 4. Comparison of *Ralph*'s performance on instances bounded or failed by other algorithms.

### 7.3 Main Results: Fine-Grained Analysis

So far, we have highlighted the performance of *Ralph* on large instances and lengthy timeouts. This raises the question of how fast and accurately *Ralph* can perform on small (CQ) instances, where split-second differences in speed are as important as low-width decompositions. In this section, we discuss the high potential of *Ralph* to be incredibly fast in these fine-grained settings, by presenting its ability to keep up with other algorithms without much optimization. That said, we emphasize that optimizing *Ralph* to gain the order of seconds for small instances was not the original focus of our paper, and something we explore to provide background for a common use-case and potential future project.

*Fine-Grained Experimental Setup.* To analyze *Ralph* in a fine-grained manner, we chose to compare to *Triangulator* [35] on CQ instances. As *fhtw* can be used interchangeably with *ghtw* and *htw* in applications, while being a smaller parameter, we see it more useful to compute in these faster settings. So we picked *Triangulator*, which we found to be faster than *FraSMT*. Secondly, the

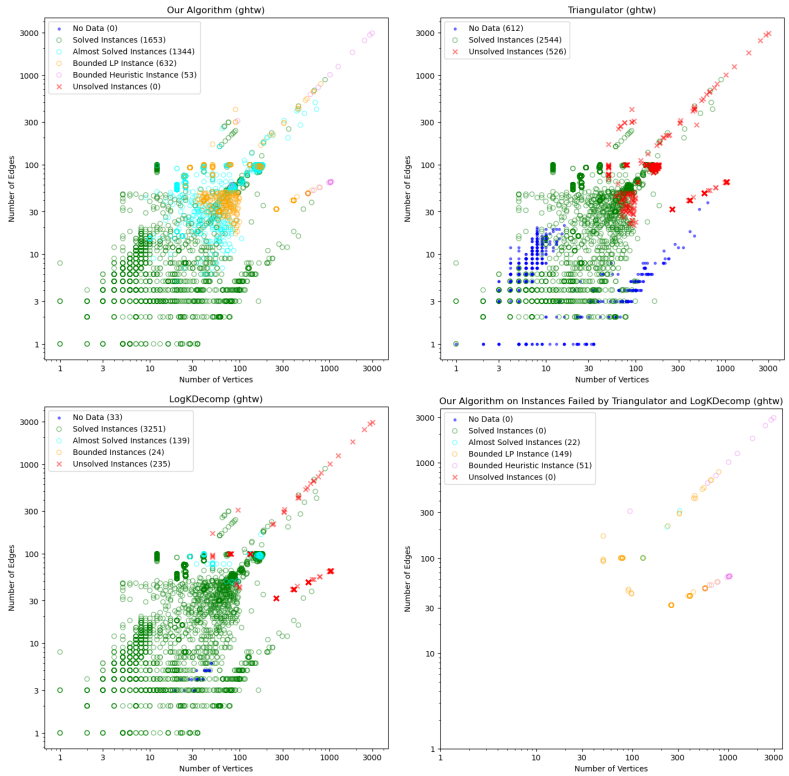


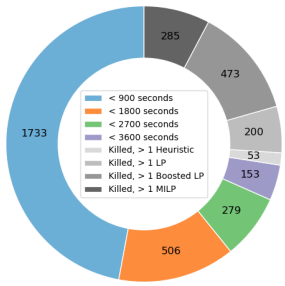

Fig. 5. Performance of *Triangulator* and *log-k-decomp* in comparison to *Ralph* for *ghtw*. The instances are categorized as discussed in Subsection 7.2. *Ralph* obtains upper bounds for 222 instances failed by both *Triangulator* and *log-k-decomp*.

applications for split-second fast computations of small instances lie heavily in query optimizations, so we decided to compare over the CQ instances. Furthermore, for the fine-grained analysis, we only compare runs of the Heuristic with the *Triangulator* runs, as the other versions will be likely be too slow for split-second gain.

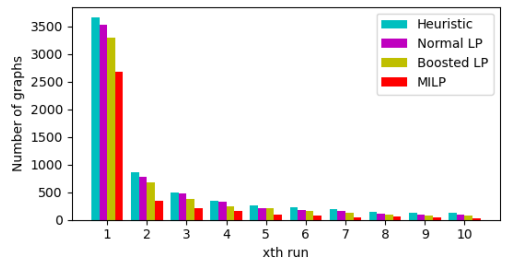
| # | % instances where {type} | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| R.A. | approx ratio ≤ 1.1 | | | runtime ratio ≤ 1 | | | both ineq. | |
| type: | best | avg | worst | best | avg | worst | best | worst |
| 1 | 95 | 82 | 79 | 65 | 57 | 50 | 62 | 48 |
| 2 | 97 | 86 | 80 | 63 | 54 | 48 | 62 | 47 |
| 4 | 98 | 91 | 81 | 60 | 52 | 46 | 59 | 46 |
| 8 | 98 | 92 | 83 | 58 | 51 | 47 | 58 | 47 |
| 16 | 99 | 92 | 84 | 57 | 52 | 49 | 57 | 49 |

Table 4. Results for 8 Heuristic runs per value of rounding attempts (R. A.) on small CQ instances solved by *Triangulator*. The approximation ratio is our *fhtw* upper bound over the true *fhtw*, and the runtime ratio is our running time over *Triangulator*'s running time. For both ineq., we look at the % of instances where the best (worst) approximation ratio and best (worst) runtime ratio was at most 1.1 and 1 respectively.

*Fine-Grained Optimizations.* Out of the 881 CQ instances solved by *Triangulator* within a second of compute, 490 of them have *fhtw* 1. These instances are acyclic, and can be recognized within




(a) Time taken (seconds) by *Ralph* on each instance. We finish 60 runs on 2671/3682 instances within the 3600 seconds time limit. *Killed,* > 1 *[version]* indicates at least one run finished before the time limit.




(b) Number of graphs in which the $x$th run found a better *fhtw* upper bound than the previous $x-1$ runs of the same version.

Fig. 6. Analysis of *Ralph*'s performance: (a) time taken across instances, and (b) improvement in *fhtw* upper bounds over multiple runs.

milliseconds. In the same vein, current algorithms like *Triangulator* use preprocessing steps, such as cutting all clique separators, that either fully solve these small instances or reduce their complexity to trivial levels. We have specifically added a simple preprocessing rule in *Ralph* for this Fine-Grained analysis, which cuts across single edge separators. This handles simple graphs, but many slightly more complex graphs preprocessed by *Triangulator* are left out. Further work would be to implement strong preprocessing rules.

Additionally, *Ralph* is written in Python, leaving potential for heavy optimization for split-second gains. There are many other potential places *Ralph* could be losing milliseconds such as Gurobi checking the validity of the license (network api call). One final tweak we make for fine-grained optimizations is reducing the number of rounding attempts we do for a run, as described in Subsection 6.2. We explore different values for this hyperparameter, as it gives us a trade-off between speed and solution quality.

*Fine-Grained Results* We try the values $\{1, 2, 4, 8, 16\}$ for the number of rounding attempts per run. For each value, we run our Heuristic version 8 times on the 881 CQ instances solved by Triangulator in under 1 second. We find promising results, shown in Table 4.

In particular, we see that increases rounding attempts increases solution quality and decreases compute time, as expected. Depending on the use case, one may pick a value for the number of rounding attempts to for a time and quality trade-off. The main takeaway from these results is that, on average, we run as fast as or even better than *Triangulator* for at least half of the graphs, and, on average, our *fhtw* upper bound for these small graphs is only slightly worse than the true *fhtw*. Furthermore, we notice on almost half the instances the worst approximation ratio and the worst runtime ratio out of the 8 Heuristic runs is still bounded by 1.1 and 1 respectively. We conclude that *Ralph* is as strong as *Triangulator* in fine-grained settings, and more importantly, has large potential to be even faster by optimizing in the various directions discussed earlier.

### 7.4 Main Results: Internal Analysis

As *Ralph* is randomized, approximate and is ran with four different versions of BalancedSeparator (see Section 5.3), there are many interesting stats about *Ralph* that can be gleaned from our results.

*Time Analysis.* First, we do a time analysis on *Ralph*. This analysis is independent of the parameter computed (*fhtw*, *ghtw*), as each run of our algorithm computes both *fhtw* and *ghtw*. The analysis is shown in Figure 6a. Note that only 53 of the instances complete without any theoretical guarantee.

*Balanced Separator Versions and Run Distribution.* We currently run *Ralph* with a 10/20/20/10 run distribution until we hit our 3600 second time limit. We picked this run distribution through analysis of preliminary results with each variation of the balanced separator, as discussed in Subsection 6.2. There is a great potential for growth by varying the run distribution per instance size, time budget and quality of solution desired. However, finding such an adaptive run distribution without overfitting is hard and time consuming. Yet, there are cases where one can easily choose a good run distribution, such as only running the Heuristic in the Fine-Grained case.

Another point to discuss when considering run distributions is: how beneficial is it to run a version multiple times? As in, how does the upper bound we get improve with more runs of the algorithm. For each version of our algorithm, we looked at graphs on which ran we at least 10 runs of that version (so some relatively large graphs were left out), and checked if the $x$th run of each version improved the fhtw upper bound found by the previous $x-1$ runs. The results (Figure 6b) show that consecutive runs improve the upper bound for many graphs; note that the majority of graphs are small and trivial, so for bigger graphs consecutive runs are more important.

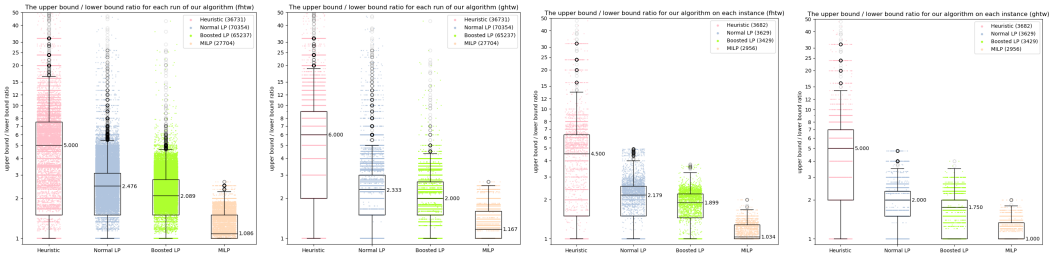



Fig. 7. Upper bound / lower bound ratio for each run of *Ralph*, and for each instance *Ralph* is ran on (where the instance ratio is the best upper over the best lower bound found on that instance using *Ralph* variation).

*Robustness of Upper and Lower Bounds.* For each run of *Ralph* on an instance, we get upper and lower bounds for *fhtw* and *ghtw*. We conduct an analysis on how close how upper and lower bounds are. From our results above (Figures 3 and 5), we see that the large majority of our instances are either solved (upper bound = lower bound) or almost solved (upper bound - lower bound $\leq 2$). Let us now look at how close our upper and lower bounds are for each run and variation. Note that for the Heuristic runs, we get no lower bounds besides the trivial lower bound of 1. To start, we plot the upper bound / lower bound ratio for each run, bucketed by the variation of BalancedSeparator we use. This can be seen in Figure 7. Unsurprisingly, each consecutive variation of *Ralph* is better than the last, with MILP reaching a median ratio very close to 1, and not more than 2 (when comparing the best upper and lower bounds achieved by MILP for each instance). Furthermore, while the ratio for each run has a lot more outliers than the ratio for each instance, the box plots only have a slightly larger spread. We argue that this shows our best upper/lower bound ratios are not due to one good run while the rest are bad, but moreso that the ratio for each run is evenly spread. So, while the probability of an "amazing" run might still be low, the non-"amazing" runs will be evenly distributed, and not far off from the best possible run. Therefore, *Ralph* is stable in both expectation and variance.

## 8 Related Work

Hypertree decomposition based techniques have applications in numerous domains in addition to databases such as in constraint satisfaction [19, 23, 38], combinatorial auctions [14], and automated selection of Web services based on recommendations from social networks [25]. Thus hypergraph tree decompositions have been extensively studied both from theoretical and practical viewpoints.

Here are some works on algorithms with provable guarantees in terms of time [19–21, 23, 34, 36, 38, 39], space [21] and parallellism [6, 18, 21]. A recent survey by Gottob et al. [15] has a complete picture of hypertree decompositions and its use in databases.

On another front, numerous variants of join processing have been proposed and hundreds of algorithms have been presented. We refer readers to [40] for a survey on join processing. On the experimental side, since the time of publication of HyperBench (2019), there have been a number of algorithms [5, 13, 17, 18, 22, 35, 43, 44] evaluated upon it. Out of these algorithms, there are four which we believe to be state-of-the-art, by analysis of the comparisons conducted in their papers. These four algorithms are *FraSMT* [11], *Triangulator* [35], *HtDLeo* [44], and *log-k-decomp* [18].

Finally, it is important to discuss the techniques used in these four algorithms. As suggested by the name, *FraSMT* smartly encodes the fractional hypertree decomposition constraints into a SMT problem. For *ghtw*, it simply strengthens some variables in the SMT encoding to be integral. On the other hand, *Triangulator* uses the enumeration of Potential Maximal Cliques. *HtDLeo* encodes hypertree decompositions into SAT using linear elimination orderings. Finally, *log-k-decomp* uses balanced separators to do parallel pruning of the search space, requiring only log recursion-levels.

## 9 Conclusion

In this work, we presented *Ralph* (Randomized Approximation using Linear Programming for Hypertree-Decompositions), an algorithm for computing upper bounds and lower bounds for both fractional and generalized hypertree width of an input hypergraph. In addition to the bounds, our algorithm also outputs a hypertree decomposition with width equal to the upper bound. In our algorithm, we utilized (mixed integer) linear programs, and used an approach consisting of four different schemes. This allows us to solve small instances with strong theoretical guarantees and deliver non-trivial upper bounds on larger instances within a 60-minute time limit using heuristics.

In our empirical evaluation on the HyperBench benchmark, we find our decompositions to be near optimal, compared to the previous state-of-the-art. In addition, for all previously unsolved instances, we compute non-trivial upper and lower bounds. Fine-Grained analysis shows both success and potential for growth in split-second computes for small graphs, while internal analysis reveals our algorithm to be stable in both expectation and variance. Many avenues for future work lie in optimizing *Ralph* for different settings through hyperparameter and run distribution tuning.

Two important applications of this work include CQ evaluation and CSP solving. As *Ralph* is randomized, we output many different tree decompositions for each input, these can be used in query optimization to generate multiple query plans.

Moving forward, this work demonstrates the practicality of computing fractional hypertree width efficiently, particularly showing that *fhtw* is as easy to compute as *ghtw* and *htw*. Since *fhtw* is the smallest of these parameters and is used interchangeably in applications, we believe these findings motivate future work to be focused on the computation and utility of *fhtw*, as opposed to *ghtw* and *htw*. Finally, we hope that our work will lead to the wider adoption of tree decomposition based methods in practice.

## Acknowledgments

Vaishali Surianarayanan, Anikait Mundhra, and Ajaykrishnan E S contributed equally to this work. The project was supervised by Daniel Lokshtanov. Use was made of computational facilities purchased with funds from the National Science Foundation (CNS-1725797) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 2308708) at UC Santa Barbara. We acknowledge the use of Gurobi Optimization [24] and the Python library PuLP in our code. Anikait Mundhra thanks the UCSB CCS SURF program for supporting his work.

## References

[1] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Mind the gap: Bridging multi-domain query workloads with emptyheaded. *Proc. VLDB Endow.*, 10(12):1849–1852, 2017.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 431–446. ACM, 2016.

[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old techniques for new join algorithms: A case study in RDF processing. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pages 97–102. IEEE Computer Society, 2016.

[4] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007.

[5] Nabil Adrar, Philippe Jégou, and Cyril Terrioux. Computing partial hypergraphs of bounded width. *Discret. Appl. Math.*, 329:1–22, 2023.

[6] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, volume 68 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382. ACM, 2015.

[8] Yu Chen and Ke Yi. Random sampling and size estimation over cyclic joins. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[9] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.

[10] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *Proc. VLDB Endow.*, 15(5):1024–1037, 2022.

[11] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2018.

[12] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, page 464–480, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Robert Ganian, André Schidler, Manuel Sorge, and Stefan Szeider. Threshold treewidth and hypertree width. *J. Artif. Intell. Res.*, 74:1687–1713, 2022.

[14] Georg Gottlob and Gianluigi Greco. Decomposing combinatorial auctions and set packing problems. *J. ACM*, 60(4):24:1–24:39, 2013.

[15] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 57–74. ACM, 2016.

[16] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In Dieter Kratsch, editor, *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[17] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, and Cem Okulmus. Incremental updates of generalized hypertree decompositions. *ACM J. Exp. Algorithmics*, 27:1.16:1–1.16:28, 2022.

[18] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Fast parallel hypertree decompositions in logarithmic recursion depth. *ACM Trans. Database Syst.*, 49(1):1:1–1:43, 2024.

[19] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *J. ACM*, 68(5):38:1–38:50, 2021.

[20] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Fractional covers of hypergraphs with bounded multi-intersection. *Theor. Comput. Sci.*, 979:114204, 2023.

[21] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[22] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Fast and parallel decomposition of constraint satisfaction problems. *Constraints An Int. J.*, 27(3):284–326, 2022.

[23] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[25] Khayyam Hashmi, Zaki Malik, Erfan Najmi, and Abdelmounaam Rezgui. SNRNeg: A social network enabled negotiation service. *Inf. Sci.*, 349-350:248–262, 2016.

[26] Zeyuan Hu and Daniel P. Miranker. Treetracker join: Turning the tide when a tuple fails to join. *CoRR*, abs/2403.01631, 2024.

[27] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1259–1274. ACM, 2017.

[28] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. General dynamic yannakakis: conjunctive queries with theta joins under updates. *VLDB J.*, 29(2-3):619–653, 2020.

[29] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. Aggregations over generalized hypertree decompositions. *CoRR*, abs/1508.07532, 2015.

[30] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. On functional aggregate queries with additive inequalities. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 414–431. ACM, 2019.

[31] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst-case and beyond. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 213–228. ACM, 2015.

[32] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 13–28. ACM, 2016.

[33] Kyoungmin Kim, Jaehyun Ha, George Fletcher, and Wook-Shin Han. Guaranteeing the $\tilde{O}$(agm/out) runtime for uniform sampling and size estimation over joins. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, page 113–125. Association for Computing Machinery, 2023.

[34] Viktoriia Korchemna, Daniel Lokshtanov, Saket Saurabh, Vaishali Surianarayanan, and Jie Xue. Efficient approximation of fractional hypertree width, 2024.

[35] Tuukka Korhonen. *Finding optimal tree decompositions.* PhD thesis, Master's thesis, University of Helsinki, 2020., 2020.

[36] Matthias Lanzinger and Igor Razgon. FPT approximation of generalised hypertree width for bounded intersection hypergraphs. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12-14, 2024, Clermont-Ferrand, France*, volume 289 of *LIPIcs*, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[37] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, 1999.

[38] Dániel Marx. Approximating fractional hypertree width. *ACM Trans. Algorithms*, 6(2):29:1–29:17, 2010.

[39] Lukas Moll, Siamak Tazari, and Marc Thurley. Computing hypergraph width measures exactly. *Inf. Process. Lett.*, 112(6):238–242, 2012.

[40] Hung Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124. ACM, 2018.

[41] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[42] Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

[43] André Schidler and Stefan Szeider. Computing optimal hypertree decompositions. In Guy E. Blelloch and Irene Finocchi, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 1–11. SIAM, 2020.

[44] André Schidler and Stefan Szeider. Computing optimal hypertree decompositions with SAT. *Artif. Intell.*, 325:104015, 2023.

[45] Susan Tu and Christopher Ré. Duncecap: Query plans using generalized hypertree decompositions. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2077–2078. ACM, 2015.

[46] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. Change propagation without joins. *Proc. VLDB Endow.*, 16(5):1046–1058, 2023.
[47] Qichen Wang and Ke Yi. Conjunctive queries with comparisons. *SIGMOD Rec.*, 52(1):54–62, 2023.
[48] Yilei Wang and Ke Yi. Secure yannakakis: Join-aggregate queries over private data. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1969–1981. ACM, 2021.
[49] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.
[50] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

Received October 2024; revised January 2025; accepted February 2025