# HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings

Wolfgang Fischl
TU Wien
wolfgang.fischl@tuwien.ac.at

Georg Gottlob
TU Wien & University of Oxford
georg.gottlob@cs.ox.ac.uk

Davide Mario Longo
TU Wien
davide.longo@tuwien.ac.at

Reinhard Pichler
TU Wien
reinhard.pichler@tuwien.ac.at

## ABSTRACT

To cope with the intractability of answering Conjunctive Queries (CQs) and solving Constraint Satisfaction Problems (CSPs), several notions of hypergraph decompositions have been proposed – giving rise to different notions of width, noticeably, plain, generalized, and fractional hypertree width (hw, ghw, and fhw). Given the increasing interest in using such decomposition methods in practice, a publicly accessible repository of decomposition software, as well as a large set of benchmarks, and a web-accessible workbench for inserting, analysing, and retrieving hypergraphs are called for.

We address this need by providing (i) concrete implementations of hypergraph decompositions (including new practical algorithms), (ii) a new, comprehensive benchmark of hypergraphs stemming from disparate CQ and CSP collections, and (iii) HyperBench, our new web-interface for accessing the benchmark and the results of our analyses. In addition, we describe a number of actual experiments we carried out with this new infrastructure.

## KEYWORDS

hypergraph decomposition methods, query answering, constraint satisfaction

**ACM Reference Format:**
Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. 2019. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3294052.3319683


ACM ISBN 978-1-4503-6227-6/19/06…$15.00
https://doi.org/10.1145/3294052.3319683

## 1 INTRODUCTION

In this work we study computational problems on hypergraph decompositions which are designed to speed up the evaluation of Conjunctive Queries (CQs) and the solution of Constraint Satisfaction Problems (CSPs). Hypergraph decompositions have meanwhile found their way into commercial database systems such as LogicBlox [5, 8, 32, 33, 40] and advanced research prototypes such as EmptyHeaded [1, 2, 41, 48]. Hypergraph decompositions have also been successfully used in the CSP area [4, 29, 34]. In theory, the pros and cons of various notions of decompositions and widths are well understood (see [23] for a survey). However, from a practical point of view, many questions have remained open.

We want to analyse the hypertree width ($hw$) of hypergraphs from different application contexts. The investigation of millions of CQs [12, 42] posed at various SPARQL endpoints suggests that these real-world CQs with atoms of arity $\leq 3$ have very low $hw$: the overwhelming majority is acyclic; almost all of the rest has $hw = 2$. It is, however, not clear if CQs with arbitrary arity and CSPs also have low hypertree width, say, $hw \leq 5$. Ghionna et al. [21] gave a positive answer to this question for a small set of TPC-H benchmark queries. We significantly extend their collection of CQs.

Answering CQs and solving CSPs are fundamental tasks in Computer Science. Formally, they are the same problem, since both correspond to the evaluation of first-order formulae over a finite structure, such that the formulae only use $\{\exists, \wedge\}$ as connectives but not $\{\forall, \vee, \neg\}$. Both problems, answering CQs and solving CSPs, are NP-complete [14]. Consequently, the search for tractable fragments of these problems has been an active research area in the database and artificial intelligence communities for several decades.

The most powerful methods known to date for defining tractable fragments are based on various decompositions of the hypergraph structure underlying a given CQ or CSP. The most important forms of decompositions are *hypertree decompositions (HDs)* [24], *generalized hypertree decompositions (GHDs)* [24], and *fractional hypertree decompositions (FHDs)* [27]. These decomposition methods give rise to three notions of width of a hypergraph $H$: the *hypertree width* $hw(H)$, *generalized hypertree width* $ghw(H)$, and *fractional hypertree width* $fhw(H)$, where, $fhw(H) \leq ghw(H) \leq hw(H)$ holds for every hypergraph $H$. For definitions, see Section 2.

Both, answering CQs and solving CSPs, become tractable if the underlying hypergraphs have bounded *hw*, *ghw*, or, *fhw* and an appropriate decomposition is given. This gives rise to the problem of recognizing if a given CQ or CSP has *hw*, *ghw*, or, *fhw* bounded by some constant $k$. Formally, for *decomposition* $\in$ {HD, GHD, FHD} and $k \geq 1$, we consider the following family of problems:

Check(*decomposition*, $k$)
**input** hypergraph $H = (V, E)$;
**output** *decomposition* of $H$ of width $\leq k$ if it exists and answer 'no' otherwise.

Clearly, bounded *fhw* defines the largest tractable class while bounded *hw* defines the smallest one. On the other hand, the problem Check(HD, $k$) is feasible in polynomial time [24] while the Check(GHD, $k$) [25] and Check(FHD, $k$) [19] problems are NP-complete even for $k = 2$.

Systems to solve the Check(HD, $k$) problem exist [26, 44]. In contrast, for Check(GHD, $k$) and Check(FHD, $k$), apart from exhaustive search over possible decomposition trees (which only works for small hypergraphs), no implementations have been reported yet [1] – with one exception: very recently, an interesting approach is presented in [17], where SMT-solving is applied to the Check(FHD, $k$) problem. In [26], tests of the Check(HD, $k$) system are presented. However, a benchmark for systematically evaluating systems for the Check(*decomposition*, $k$) problem with *decomposition* $\in$ {HD, GHD, FHD} and $k \geq 1$ were missing so far. This motivates our first research goals.

**Goal 1:** Create a comprehensive, easily extensible benchmark of hypergraphs corresponding to CQs or CSPs for the analysis of hypergraph decomposition algorithms.

**Goal 2:** Use the benchmark from Goal 1 to find out if the hypertree width is, in general, small enough (say $\leq 5$) to allow for efficient evaluation of CQs of arbitrary arity and of CSPs.

Recently, in [19], the authors have identified classes of CQs for which the Check(GHD, $k$) and Check(FHD, $k$) problems become tractable (from now on, we only speak about CQs; of course, all results apply equally to CSPs). To this end, the Bounded Intersection Property (BIP) and, more generally, the Bounded Multi-Intersection Property (BMIP) have been introduced. The maximum number $i$ of attributes shared by two (resp. $c$) atoms is referred to as the intersection width (resp. $c$-multi-intersection width) of the CQ, which is similar to the notion of cutset width from the CSP literature [15]. We say that a class of CQs satisfies the BIP (resp. BMIP) if the number of attributes shared by two (resp. by a constant number $c$ of) query atoms is bounded by some constant $i$.

A related property is that of bounded degree, i.e., each attribute only occurs in a constant number of query atoms. Clearly, the BMIP is an immediate consequence of bounded degree. It has been shown in [19] that Check(GHD, $k$) is solvable in polynomial time for CQs whose underlying hypergraphs satisfy the BMIP. For CQs, the BMIP and bounded degree seem natural restrictions. For CSPs, the situation is not so clear. This yields the following research goals.

**Goal 3:** Use the hypergraph benchmark from Goal 1 to analyse how realistic the restrictions to low (multi-)intersection width, or low degree of CQs and CSPs are.

**Goal 4:** Verify that for hypergraphs of low intersection width, the Check(GHD, $k$) problem indeed allows for efficient algorithms that work well in practice.

The tractability results for Check(FHD, $k$) [18, 19] are significantly weaker than for Check(GHD, $k$): they involve a factor which is at least double-exponential in some "constant" (namely $k$, the bound $d$ on the degree and/or the bound $i$ on the intersection-width). Hence, we want to investigate if (generalized) hypertree decompositions could be "fractionally improved" by taking the integral edge cover at each node in the HD or GHD and replacing it by a fractional edge cover. We will thus introduce the notion of *fractionally improved* HD which checks if there exists an HD of width $\leq k$, such that replacing each integral cover by a fractional cover yields an FHD of width $\leq k'$ for given bounds $k, k'$ with $0 < k' < k$.

**Goal 5:** Explore the potential of fractionally improved HDs, i.e., investigate if the improvements achieved are significant.

In cases where Check(GHD, $k$) and Check(FHD, $k$) are intractable, we may have to settle for good approximations of *ghw* and *fhw*. For GHDs, we may thus use the inequality $ghw(H) \leq 3 \cdot hw(H) + 1$, which holds for every hypergraph $H$ [3]. In contrast, for FHDs, the best known general, polynomial-time approximation is cubic. More precisely, in [38], a polynomial-time algorithm is presented which, given a hypergraph $H$ with $fhw(H) = k$, computes an FHD of width $O(k^3)$. In [19], it is shown that a polynomial-time approximation up to a logarithmic factor is possible for any class of hypergraphs with bounded Vapnik–Chervonenkis dimension (VC-dimension; see Section 2 for a precise definition). The problem of efficiently approximating the *ghw* and/or *fhw* leads us to the following goals.

**Goal 6:** Use the benchmark from Goal 1 to analyse if, in practice, *hw* and *ghw* indeed differ by factor 3 or, if *hw* is typically much closer to *ghw* than this worst-case bound.

**Goal 7:** Use the benchmark from Goal 1 to analyse how realistic the restriction to small VC-dimension of CQs and CSPs is.

**Results.** Our main results are as follows:

- We provide *HyperBench*, a comprehensive hypergraph benchmark of initially over 3,000 hypergraphs (see Section 3). This benchmark is exposed by a web interface, which allows the user to retrieve the hypergraphs or groups of hypergraphs together with a broad spectrum of properties of these hypergraphs, such as lower/upper bounds on *hw* and *ghw*, (multi-)intersection width, degree, etc.
- We extend the software for HD computation from [26] to also solve the Check(GHD, $k$) problem. For a given hypergraph $H$, our system first computes the intersection width of $H$ and then applies the *ghw*-algorithm from [19], which is parameterized by the intersection width. We implement several improvements and we further extend the system to compute also "fractionally improved" HDs.
- We carry out an empirical analysis of the hypergraphs in the HyperBench benchmark. This analysis demonstrates, especially for real-world instances, that the restrictions to BIP, BMIP, bounded degree, and bounded VC-dimension are astonishingly realistic. Moreover, on all hypergraphs in the HyperBench benchmark, we run our *hw*- and *ghw*-systems to identify (or at least bound) their *hw* and *ghw*. An interesting observation of our empirical study is that apart from the CQs also a significant portion of CSPs in our benchmark has small hypertree width (all non-random CQs have $hw \leq 3$ and over 60% of CSPs stemming from applications have $hw \leq 5$). Moreover, for $hw \leq 5$, in all of the cases where the *ghw*-computation terminates, *hw* and *ghw* have identical values.
- In our study of the *ghw* of the hypergraphs in the HyperBench benchmark, we observed that a straightforward implementation of the algorithm from [19] for hypergraphs of low intersection width is too slow in many cases. We therefore present a new approach (based on so-called "balanced separators") with promising experimental results. It is interesting to note that the new approach works particularly well in those situations which are particularly hard for the straightforward implementation, namely hypergraphs $H$ where the test if $ghw \leq k$ for given $k$ gives a "no"-answer. Hence, combining the different approaches is very effective.

**Structure.** This paper is structured as follows: In Section 2, we recall some basic notions. In Section 3, we present our system and test environment as well as our HyperBench benchmark. First results of our empirical study of the hypergraphs in this benchmark are presented in Section 4. In Section 5, we describe our algorithms for solving the Check(GHD, $k$) problem. A further extension of the system to allow for the computation of fractionally improved HDs is described in Section 6. Finally, in Section 7 we summarize related work and conclude in Section 8 by highlighting the most important lessons learned from our empirical study and by identifying some appealing directions for future work.

Due to lack of space, some of the statistics presented in the main body contain aggregated values (for instance, for different classes of CSPs). Figures and tables with more fine-grained results (for instance, distinguishing the 3 classes of CSPs to be presented in Section 4) will be provided in the full version of this paper.

## 2 PRELIMINARIES

Let $\phi$ be a CQ or CSP (i.e., an FO-formula with connectives $\{\exists, \wedge\}$). The *hypergraph corresponding to* $\phi$ is defined as $H = (V(H), E(H))$, where the set of vertices $V(H)$ is defined as the set of variables in $\phi$ and the set of edges $E(H)$ is defined as $E(H) = \{e \mid \phi$ contains an atom $A$, s.t. $e$ equals the set of variables occurring in $A\}$.

**Hypergraph decompositions and width measures.** We consider here three notions of hypergraph decompositions with associated notions of width. To this end, we first need to introduce the notion of (fractional) edge covers:

Let $H = (V(H), E(H))$ be a hypergraph and consider a function $\gamma \colon E(H) \to [0, 1]$. Then, we define the set $B(\gamma)$ of all vertices covered by $\gamma$ and the weight of $\gamma$ as

$$\begin{aligned} B(\gamma) &= \left\{ v \in V(H) \mid \sum_{e \in E(H), v \in e} \gamma(e) \geq 1 \right\}, \\ \mathit{weight}(\gamma) &= \sum_{e \in E(H)} \gamma(e). \end{aligned}$$

The special case of a function with values restricted to $\{0, 1\}$, will be denoted by $\lambda$, i.e., $\lambda \colon E(H) \to \{0, 1\}$. Following [24], we can also treat $\lambda$ as a set with $\lambda \subseteq E(H)$ (namely, the set of edges $e$ with $\lambda(e) = 1$) and the weight as the cardinality of such a set of edges.

We now introduce three notions of decompositions.

*Definition 2.1.* A *generalized hypertree decomposition* (for short GHD) of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, (B_u)_{u \in N(T)}, (\lambda_u)_{u \in N(T)} \rangle$, such that $T = \langle N(T), E(T) \rangle$ is a rooted tree and the following conditions hold:

(1) $\forall e \in E(H)$: there exists a node $u \in N(T)$ with $e \subseteq B_u$;
(2) $\forall v \in V(H)$: the set $\{u \in N(T) \mid v \in B_u\}$ is connected in $T$;
(3) $\forall u \in N(T)$: $\lambda_u$ is defined as $\lambda_u \colon E(H) \to \{0, 1\}$ with $B_u \subseteq B(\lambda_u)$.

We use the following notational conventions throughout this paper. To avoid confusion, we will consequently refer to the elements in $V(H)$ as *vertices* of the hypergraph and to the elements in $N(T)$ as the *nodes* of the decomposition. For a node $u$ in $T$, we write $T_u$ to denote the subtree of $T$ rooted at $u$. By slight abuse of notation, we will often write $u' \in T_u$ to denote that $u'$ is a node in the subtree $T_u$ of $T$. Finally, we define $V(T_u) := \bigcup_{u' \in T_u} B_{u'}$.

*Definition 2.2.* A *hypertree decomposition* (for short HD) of a hypergraph $H = (V(H), E(H))$ is a GHD, which in addition also satisfies the following condition:

(4) $\forall u \in N(T)$: $V(T_u) \cap B(\lambda_u) \subseteq B_u$

*Definition 2.3.* A *fractional hypertree decomposition* (for short FHD) [27] of a hypergraph $H = (V(H), E(H))$ is defined as a tuple $\langle T, (B_u)_{u \in N(T)}, (\gamma_u)_{u \in N(T)} \rangle$, where conditions (1) and (2) of Definition 2.1 plus the following condition (3') hold:

(3') $\forall u \in N(T)$: $\gamma_u$ is defined as $\gamma_u : E(H) \to [0, 1]$ with $B_u \subseteq B(\gamma_u)$.

The width of a GHD, HD, or FHD is the maximum weight of the functions $\lambda_u$ or $\gamma_u$, over all nodes $u$ in $T$. The generalized hypertree width, hypertree width, and fractional hypertree width of $H$ (denoted $ghw(H)$, $hw(H)$, $fhw(H)$) is the minimum width over all GHDs, HDs, and FHDs of $H$, respectively. Condition (2) is called the "connectedness condition", and condition (4) is referred to as "special condition" [24]. The set $B_u$ is often referred to as the "bag" at node $u$. The functions $\lambda_u$ and $\gamma_u$ are referred to as the $\lambda$-*label* and $\gamma$-*label* of node $u$. Strictly speaking, only HDs require that the underlying tree $T$ be rooted. We assume that also the tree underlying a GHD or an FHD is rooted where the root is *arbitrarily* chosen.

**Favourable properties of hypergraphs.** In [19], the following properties of hypergraphs were identified to allow for the definition of tractable classes of Check(GHD, $k$) and for an efficient approximation of Check(FHD, $k$), respectively.

*Definition 2.4.* The *intersection width iwidth*($H$) of a hypergraph $H$ is the maximum cardinality of any intersection $e_1 \cap e_2$ of two edges $e_1 \neq e_2$ of $H$. We say that a hypergraph $H$ has the *i-bounded intersection property (i-BIP)* if $iwidth(H) \leq i$. A class $\mathscr{C}$ of hypergraphs has the *bounded intersection property (BIP)* if there exists some constant $i$ such that every hypergraph $H$ in $\mathscr{C}$ has the $i$-BIP.

*Definition 2.5.* For positive integer $c$, the *c-multi-intersection width c-miwidth(H)* of a hypergraph $H$ is the maximum cardinality of any intersection $e_1 \cap \cdots \cap e_c$ of $c$ distinct edges $e_1, \ldots, e_c$ of $H$. We say that a hypergraph $H$ has the *i-bounded c-multi-intersection property (ic-BMIP)* if *c-miwidth(H)* $\leq i$ holds. We say that a class $\mathscr{C}$ of hypergraphs has the *bounded multi-intersection property (BMIP)* if there exist constants $c$ and $i$ such that every hypergraph $H$ in $\mathscr{C}$ has the *ic*-BMIP.

There are two more relevant properties of (classes of) hypergraphs: bounded degree and bounded Vapnik–Chervonenkis dimension (VC-dimension). It is easy to verify [19] that bounded degree implies the BMIP, which in turn implies bounded VC-dimension.

*Definition 2.6.* The *degree* deg($H$) of a hypergraph $H$ is defined as the maximum number $d$ of hyperedges in which a vertex occurs, i.e., $d = \max_{v \in V(H)} |\{e \in E(H) \mid v \in E(H)\}|$. We say that a class $\mathscr{C}$ of hypergraphs has bounded degree, if there exists $d \geq 1$, such that every hypergraph $H \in \mathscr{C}$ has degree $\leq d$.

*Definition 2.7 ([49]).* Let $H = (V(H), E(H))$ be a hypergraph, and $X \subseteq V$ a set of vertices. Denote by $E(H)|_X = \{X \cap e \mid e \in E(H)\}$. $X$ is called *shattered* if $E(H)|_X = 2^X$. The *Vapnik-Chervonenkis dimension (VC dimension)* of $H$ is the maximum cardinality of a shattered subset of $V$. We say that a class $\mathscr{C}$ of hypergraphs has bounded VC-dimension, if there exists $v \geq 1$, such that every hypergraph $H \in \mathscr{C}$ has VC-dimension $\leq v$.

The above four properties help to solve or approximate the Check(GHD, $k$) and Check(FHD, $k$) problems as follows:

Theorem 2.8 ([18, 19]). *Let $\mathscr{C}$ be a class of hypergraphs.*

- *If $\mathscr{C}$ has the BMIP, then the Check(GHD, $k$) problem is solvable in polynomial time for arbitrary $k \geq 1$. Consequently, this tractability holds if $\mathscr{C}$ has bounded degree or the BIP (which each imply the BMIP) [19].*
- *If $\mathscr{C}$ has bounded degree, then the Check(FHD, $k$) problem is solvable in polynomial time for arbitrary $k \geq 1$ [18].*
- *If $\mathscr{C}$ has bounded VC-dimension, then the fhw can be approximated in polynomial time up to a log-factor [19].*

## 3 HYPERBENCH BENCHMARK AND TOOL

In this section, we introduce our system, test environment, and HyperBench – our new benchmark and web tool.

**System and Test Environment.** In [26], an implementation (called `DetKDecomp`) of the hypertree decomposition algorithm from [24] was presented. We have extended this implementation and built our library (called `NewDetKDecomp`) upon it. This library includes the original $hw$-algorithm from [26], the tool `hg-stats` to determine properties described in Section 4 and the algorithms to be presented in Sections 5 and 6. The library is written in C++ and comprises around 8,500 lines of code. The code is available in GitHub at http://github.com/TUfischl/newdetkdecomp.

All the experiments reported in this paper were performed on a cluster of 10 workstations each running Ubuntu 16.04. Every workstation has the same specification and is equipped with two Intel Xeon E5-2650 (v4) processors each having 12 cores and 256-GB main memory. Since all algorithms are single-threaded, we were allowed to compute several instances in parallel. For all upcoming runs of our algorithms we set a timeout of 3600s.

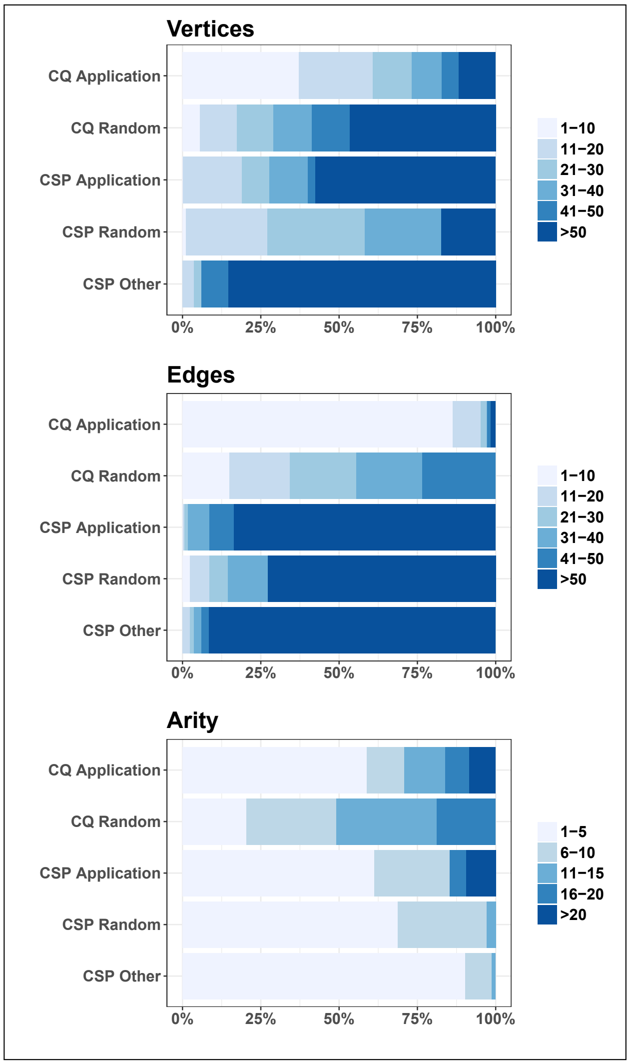


**Figure 1: Hypergraph Sizes**

**Hypergraph benchmark.** Our benchmark contains 3,070 hypergraphs, which have been converted from CQs and CSPs collected from various sources. Out of these 3,070 hypergraphs, 2,918 hypergraphs have never been used in a hypertree width analysis before. The hypertree width of 70 CQs and of 82 CSPs has been analysed in [26], [11], and/or

**Table 1: Overview of benchmark instances**

| | *Benchmark* | *No. instances* | *$hw \geq 2$* |
|---|---|---|---|
| CQs | SPARQL[12] | 70 (out of 26,157,880) | 70 |
| | LUBM[10, 28] | 14 | 2 |
| | iBench[6, 10] | 40 | 0 |
| | Doctors[10, 20] | 14 | 0 |
| | Deep[10] | 41 | 0 |
| | JOB (IMDB) [36] | 33 | 7 |
| | TPC-H [9, 47] | 33 | 1 |
| | SQLShare [30] | 290 (out of 15,170) | 1 |
| | Random [43] | 500 | 464 |
| CSPs | Application [7] | 1,090 | 1,090 |
| | Random [7] | 863 | 863 |
| | Other [11, 26] | 82 | 82 |
| | *Total:* | *3,070* | *2,580* |

[12]. An overview of all instances of CQs and CSPs is given in Table 1. They have been collected from various publicly available benchmarks and repositories of CQs and CSPs. In the first column, the names of each collection of CQs and CSPs are given together with references where they were first published. In the second column we display the number of hypergraphs extracted from each collection. The *hw* of the CQs and CSPs in our benchmark will be discussed in detail in Section 4. To get a first feeling of the *hw* of the various sources, we mention the number of cyclic hypergraphs (i.e., those with $hw \geq 2$) in the last column. When gathering the CQs, we proceeded as follows: of the huge benchmark reported in [12], we have only included CQs, which were detected as having $hw \geq 2$ in [12]. Of the big repository reported in [30], we have included those CQs, which are not trivially acyclic (i.e., they have at least 3 atoms). Of all the small collections of queries, we have included all.

Below, we describe the different benchmarks in detail:

- *CQs*: Our benchmark contains 535 CQs from four main sources [9, 10, 12, 30] and a set of 500 randomly generated queries using the query generator of [43]. In the sequel, we shall refer to the former queries as *CQ Application*, and to the latter as *CQ Random*. The CQs analysed in [12] constitute by far the biggest repository of CQs – namely 26,157,880 CQs stemming from SPARQL queries. The queries come from real-users of SPARQL endpoints and their hypertree width was already determined in [12]. Almost all of these CQs were shown to be acyclic. Our analysis comprises 70 CQs from [12], which (apart from few exceptions) are essentially the ones in [12] with $hw \geq 2$. In particular, we have analysed all 8 CQs with highest *hw* among the CQs analysed in [12] (namely, $hw = 3$).

The LUBM [28], iBench [6], Doctors [20], and Deep scenarios have been recently used to evaluate the performance of chase-based systems [10]. Their queries were especially tailored towards the evaluation of query answering tasks of such systems. Note that the LUBM benchmark [28] is a widely used standard benchmark for the evaluation of Semantic Web repositories. Its queries are designed to measure the performance of those repositories over large datasets. Strictly speaking, the iBench is a tool for generating schemas, constraints, and mappings for data integration tasks. However, in [10], 40 queries were created for tests with the iBench. We therefore refer to these queries as iBench-CQs here. In summary, we have incorporated all queries that were either contained in the original benchmarks or created/adapted for the tests in [10].

The goal of the Join Order Benchmark (JOB) [36] was to evaluate the impact of a good join order on the performance of query evaluation in standard RDBMS. Those queries were formulated over the real-world dataset Internet Movie Database (IMDB). All of the queries have between 3 and 16 joins. Clearly, as the goal was to measure the impact of a good join order, those 33 queries are of higher complexity, hence 7 out of the 33 queries have $hw \geq 2$.

The 33 TPC-H queries in our benchmark are from the GitHub repository originally provided by Michael Benedikt and Efthymia Tsamoura [9] for the work on [10]. Out of the 33 CQs based on the TPC-H benchmark [47], 13 queries were handcrafted and 20 randomly generated. The TPC-H benchmark has been widely used to assess multiple aspects of the capabilities of RDBMS to process queries. They reflect common workloads in decision support systems and were chosen to have broad industry-wide relevance.

From SQLShare [30], a multi-year SQL-as-a-service experiment with a large set of real-world queries, we extracted 15,170 queries by considering all CQs (in particular, no nested SELECTs). After eliminating trivial queries (i.e., queries with $\leq 2$ atoms, whose acyclicity is immediate) and duplicates, we ended up with 290 queries.

The random queries were generated with a tool that stems from the work on query answering using views in [43]. The query generator allows 3 options: chain/star/random queries. Since the former two types are trivially acyclic, we only used the third option. Here it is possible to supply several parameters for the size of the generated queries. In terms of the resulting hypergraphs, one can thus fix the number of vertices, number of edges and arity. We have generated 500 CQs with 5 – 100 vertices, 3 – 50 edges and arities from 3 to 20. These values correspond to the values observed for the *CQ Application* hypergraphs. However, even though these size values have been chosen similarly, the structural properties of the hypergraphs in the two groups *CQ Application* and *CQ Random* differ significantly, as will become clear from our analysis in Section 4.

- *CSPs*: In total, our benchmark currently contains 2,035 hypergraphs from CSP instances, out of which 1,953 instances were obtained from xcsp.org (see also [7]). We have selected all CSP instances from xcsp.org with less than 100 constraints such that all constraints are extensional. These instances are divided into CSPs from concrete applications, called *CSP Application* in the sequel (1,090 instances), and randomly generated CSPs, called *CSP Random* below (863 instances). In addition, we have included 82 CSP instances from previous hypertree width analyses provided at https://www.dbai.tuwien.ac.at/proj/hypertree/; all of these stem from industrial applications and/or further CSP benchmarks. We refer to these instances as *other CSPs*.

Our HyperBench benchmark consists of these instances converted to hypergraphs. In Figure 1, we show the number of vertices, the number of edges and the arity (i.e., the maximum size of the edges) as three important metrics of the size of each hypergraph. The smallest are those coming from *CQ Application* (at most 10 edges), while the hypergraphs coming from CSPs can be significantly larger (up to 2993 edges). Although some hypergraphs are very big, more than 50% of all hypergraphs have maximum arity less than 5. In Figure 1 we can easily compare the different types of hypergraphs, e.g. hypergraphs of arity greater than 20 only exist in the *CSP Application* class; the *other CSPs* class contains the highest portion of hypergraphs with a big number of vertices and edges, etc.

The hypergraphs and the results of our analysis can be accessed through our web tool, available at http://hyperbench.dbai.tuwien.ac.at.

## 4 FIRST EMPIRICAL ANALYSIS

In this section, we present first empirical results obtained with the HyperBench benchmark. On the one hand, we want to get an overview of the hypertree width of the various types of hypergraphs in our benchmark (cf. Goal 2 in Section 1). On the other hand, we want to find out how realistic the restriction to low values for certain hypergraph invariants is (cf. Goal 3 stated in Section 1).

**Hypergraph Properties.** In [18, 19], several invariants of hypergraphs were used to make the Check(GHD, $k$) and Check(FHD, $k$) problems tractable or, at least, easier to approximate. We thus investigate the following properties (cf. Definitions 2.4 – 2.7):

- *Deg*: the degree of the underlying hypergraph
- *BIP*: the intersection width
- *c-BMIP*: the $c$-multi-intersection width for $c \in \{3, 4\}$
- *VC-dim*: the VC-dimension

**Table 2: Properties of all benchmark instances**

*CQ Application*

| $i$ | Deg | BIP | 3-BMIP | 4-BMIP | VC-dim |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 118 | 173 | 10 |
| 1 | 2 | 421 | 348 | 302 | 393 |
| 2 | 176 | 85 | 59 | 50 | 132 |
| 3 | 137 | 7 | 5 | 5 | 0 |
| 4 | 87 | 5 | 5 | 5 | 0 |
| 5 | 35 | 17 | 0 | 0 | 0 |
| 6 | 98 | 0 | 0 | 0 | 0 |

*CQ Random*

| $i$ | Deg | BIP | 3-BMIP | 4-BMIP | VC-dim |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 16 | 49 | 0 |
| 1 | 1 | 17 | 77 | 125 | 20 |
| 2 | 15 | 53 | 90 | 120 | 133 |
| 3 | 38 | 62 | 103 | 74 | 240 |
| 4 | 31 | 63 | 62 | 42 | 106 |
| 5 | 33 | 71 | 47 | 28 | 1 |
| 6 | 382 | 233 | 105 | 62 | 0 |

*CSP Application & Other*

| $i$ | Deg | BIP | 3-BMIP | 4-BMIP | VC-dim |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 597 | 603 | 0 |
| 1 | 0 | 1037 | 495 | 525 | 0 |
| 2 | 597 | 95 | 57 | 23 | 1115 |
| 3 | 6 | 29 | 21 | 21 | 52 |
| 4 | 20 | 10 | 2 | 0 | 0 |
| 5 | 6 | 0 | 0 | 0 | 0 |
| >5 | 543 | 1 | 0 | 0 | 0 |

*CSP Random*

| $i$ | Deg | BIP | 3-BMIP | 4-BMIP | VC-dim |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 200 | 200 | 238 | 0 |
| 2 | 0 | 224 | 312 | 407 | 220 |
| 3 | 0 | 76 | 147 | 95 | 515 |
| 4 | 12 | 181 | 161 | 97 | 57 |
| 5 | 8 | 99 | 14 | 1 | 71 |
| >5 | 843 | 83 | 29 | 25 | 0 |

The results obtained from computing *Deg*, *BIP*, 3-*BMIP*, 4-*BMIP*, and *VC-dim* for the hypergraphs in the HyperBench benchmark are shown in Table 2.

Table 2 has to be read as follows: In the first column, we distinguish different values of the various hypergraph metrics. In the columns labelled "Deg", "BIP", etc., we indicate for how many instances each metric has a particular value. For instance, by the last row in the second column, only 98 non-random CQs have degree $> 5$. Actually, for most CQs, the degree is less than 10. Moreover, for the BMIP, already with intersections of 3 edges, we get 3-*miwidth(H)* $\leq 2$ for almost all non-random CQs. Also the VC-dimension is $\leq 2$.

For CSPs, all properties may have higher values. However, we note a significant difference between randomly generated CSPs and the rest: For hypergraphs in the groups *CSP Application* and *CSP Other*, 543 (46%) hypergraphs have a high degree (>5), but nearly all instances have BIP or BMIP of less than 3. And most instances have a VC-dimension of at most 2. In contrast, nearly all random instances have a significantly higher degree (843 out of 863 instances with a degree >5). Nevertheless, many instances have small BIP and BMIP. For nearly all hypergraphs (838 out of 863) we have 4-*miwidth(H)* $\leq 4$. For 5 instances the computation of the VC-dimension timed out. For all others, the VC-dimension is $\leq 5$ for random CSPs. Clearly, as seen in Table 2, the random CQs resemble the random CSPs a lot more than the CQ and CSP Application instances. For example, random CQs have similar to random CSPs high degree (382, corresponding to 76%, with degree $> 5$), higher BIP and BMIP. Nevertheless, similar to random CSPs, the values for BIP and BMIP are still small for many random CQ instances.

To conclude, for the proposed properties, in particular BIP/BMIP and VC-dimension, most of the hypergraphs in our benchmark indeed have low values.

**Hypertree Width.** We have systematically applied the *hw*-computation from [26] to all hypergraphs in the benchmark. The results are summarized in Figure 2. In our experiments, we proceeded as follows. We distinguish between *CQ Application*, *CQ Random*, and all three groups of CSPs taken together. For every hypergraph $H$, we first tried to solve the CHECK(HD, $k$) problem for $k = 1$. In case of *CQ Application*, we thus got 454 yes-answers and 81 no-answers. The number in each bar indicates the average runtime to find these yes- and no-instances, respectively. Here, the average runtime was "0" (i.e., less than 1 second). For *CQ Random* we got 36 yes- and 464 no-instances with an average runtime below 1 second. For all CSP-instances, we only got no-answers.

In the second round, we tried to solve the CHECK(HD, $k$) problem for $k = 2$ for all hypergraphs that yielded a no-answer for $k = 1$. Now the picture is a bit more diverse: 73 of the remaining 81 CQs from *CQ Application* yielded a yes-answer in less than 1 second. For the hypergraphs stemming from *CQ Random* (resp. CSPs), only 68 (resp. 95) instances yielded a yes-answer (in less than 1 second on average), while 396 (resp. 1932) instances yielded a no-answer in less than 7 seconds on average and 8 CSP instances led to a timeout (i.e., the program did not terminate within 3,600 seconds).

This procedure is iterated by incrementing $k$ and running the *hw*-computation for all instances, that either yielded a

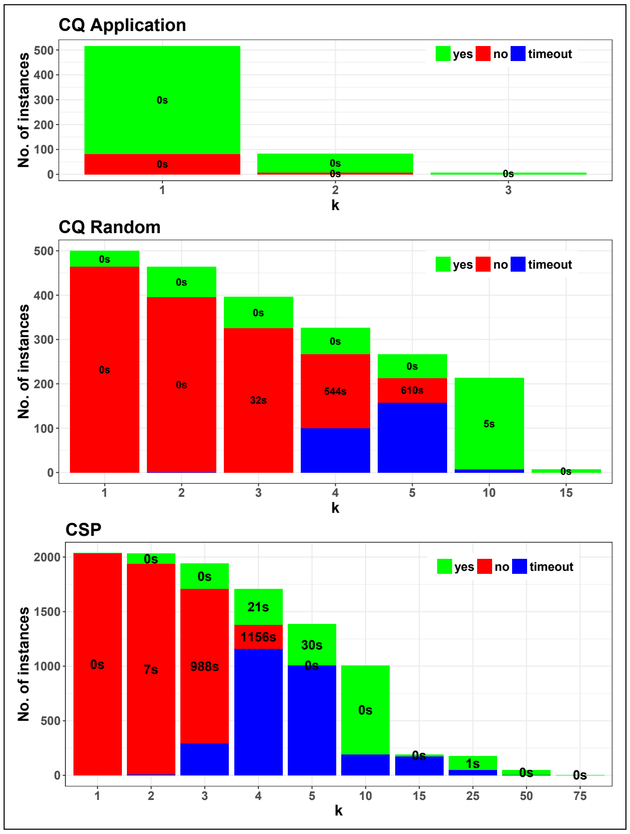


**Figure 2: HW analysis (labels are avg. runtimes in s)**

no-answer or a timeout in the previous round. For instance, for queries from *CQ Application*, one further round is needed after the second round. In other words, we confirm the observation of low $hw$, which was already made for CQs of arity $\leq 3$ in [12, 42]. For the hypergraphs stemming from *CQ Random* (resp. CSPs), 396 (resp. 1940) instances are left in the third round, of which 70 (resp. 232) yield a yes-answer in less than 1 second on average, 326 (resp. 1415) instances yield a no-answer in 32 (resp. 988) seconds on average and no (resp. 293) instances yield a timeout. Note that, as we increase $k$, the average runtime and the percentage of timeouts first increase up to a certain point and then they decrease. This is due to the fact that, as we increase $k$, the number of combinations of edges to be considered in each $\lambda$-label (i.e., the function $\lambda_u$ at each node $u$ of the decomposition) increases. In principle, we have to test $O(n^k)$ combinations, where $n$ is the number of edges. However, if $k$ increases beyond a certain point, then it gets easier to “guess” a $\lambda$-label since an increasing portion of the $O(n^k)$ possible combinations leads to a solution (i.e., an HD of desired width).

To answer the question in *Goal 2*, it is indeed the case that for a big number of instances, the hypertree width is small enough to allow for efficient evaluation of CQs or CSPs: all instances of non-random CQs have $hw \leq 3$ no matter whether their arity is bounded by 3 (as in case of SPARQL queries) or not; and a large portion (at least 1027, i.e., ca. 50%) of all 2035 CSP instances have $hw \leq 5$. In total, including random CQs, 1,849 (60%) out of 3,070 instances have $hw \leq 5$, for which we could determine the exact hypertree width for 1,453 instances; the others may even have lower $hw$.

**Correlation Analysis.** Finally, we have analysed the pairwise correlation between all properties. Of course, the different intersection widths (BIP, 3-BMIP, 4-BMIP) are highly correlated. Other than that, we only observe quite a high correlation of the arity with the number of vertices and the hypertree width and of the number of vertices with the arity and the hypertree width. Clearly, the correlation between arity and hypertree width is mainly due to the CSP instances and the random CQs since, for non-random CQs, the $hw$ never increases beyond 3, independently of the arity.

A graphical presentation of all pairwise correlations is given in Figure 3. Here, large, dark circles indicate a high correlation, while small, light circles stand for low correlation. Blue circles indicate a positive correlation while red circles stand for a negative correlation. In [19], we have argued that Deg, BIP, 3-BMIP, 4-BMIP and VC-dim are non-trivial restrictions to achieve tractability. It is interesting to note that, according to the correlations shown in Figure 3, these properties have almost no impact on the hypertree width of our hypergraphs. This underlines the usefulness of these restrictions in the sense that (a) they make the GHD computation and FHD approximation easier [19] but (b) low values of degree, (multi-)intersection-width, or VC-dimension do not pre-determine low values of the widths.

## 5 GHW COMPUTATION

In this section, we report on new algorithms and implementations to solve the CHECK(GHD, $k$) problem and on new empirical results.

**Background.** In [19], it is shown that the CHECK(GHD, $k$) problem becomes tractable for fixed $k \geq 1$, if we restrict ourselves to a class of hypergraphs enjoying the BIP. As our first empirical analysis with the HyperBench has shown (see Section 4), it is indeed realistic to assume that the intersection width of a given hypergraph is small. We have therefore extended the $hw$-computation from [26] by an implementation of the CHECK(GHD, $k$) algorithm from [19], referred to as the “$ghw$-algorithm” in the sequel. This algorithm is parameterized, so to speak, by two integers: $k$ (the desired width of a GHD) and $i$ (the intersection width of $H$).

The key idea of the $ghw$-algorithm is to add a polynomial-time computable set $f(H, k)$ of subedges of edges in $E(H)$ to the hypergraph $H$, such that $ghw(H) = k$ iff $hw(H') = k$

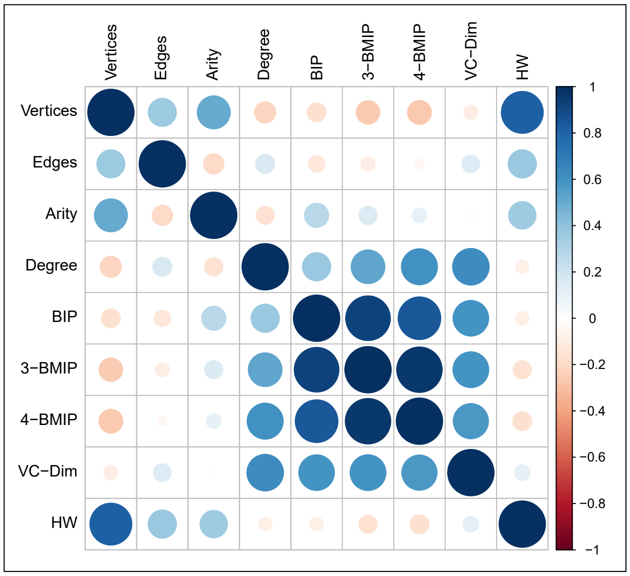


**Figure 3: Correlation analysis.**

with $H = (V(H), E(H))$ and $H' = (V(H), E(H) \cup f(H,k))$. Tractability of Check(GHD, $k$) follows immediately from the tractability of Check(HD, $k$). The set $f(H,k)$ is defined as

$$f(H,k) = \bigcup_{e \in E(H)} \Bigg( \bigcup_{e_1,\ldots,e_j \in (E(H)\setminus\{e\}),\, j \le k} 2^{(e \cap (e_1 \cup \cdots \cup e_j))} \Bigg),$$

i.e., $f(H,k)$ contains all subsets of intersections of edges $e \in E(H)$ with unions of $\le k$ edges of $H$ different from $e$. By the BIP, the intersection $e \cap (e_1 \cup \cdots \cup e_j)$ has at most $i \cdot k$ elements. Hence, for fixed constants $i$ and $k$, $|f(H,k)|$ is polynomially bounded.

**“Global” implementation.** In a straightforward implementation of this algorithm, we compute $f(H,k)$ and from this $H'$ and call the $hw$-computation from [26] for the Check(HD, $k$) problem as a “black box”. A coarse-grained overview of the results is given in Table 3 in the column labelled as ‘GlobalBIP”. We call this implementation of the $ghw$ algorithm of [19] “global” to indicate that the set $f(H,k)$ is computed “globally”, once and for all, for the *entire* hypergraph. We have run the program on each hypergraph from the HyperBench up to hypertree width 6, trying to get a smaller $ghw$ than $hw$. We have thus run the $ghw$-algorithm with the following parameters: for all hypergraphs $H$ with $hw(H) = k$ (or $hw \le k$ and, due to timeouts, we do not know if $hw \le k-1$ holds), where $k \in \{3,4,5,6\}$, try to solve the Check(GHD, $k-1$) problem. In other words, we just tried to improve the width by 1. Clearly, for $hw(H) \in \{1,2\}$, no improvement is possible since, in this case, $hw(H) = ghw(H)$ holds.

**Table 3: GHW algorithms with avg. runtimes in s**

| $hw \to$ | | GlobalBIP | | LocalBIP | | BalSep |
|---|---|---|---|---|---|---|
| $ghw$ | *total* | *yes* | *no* | *yes* | *no* | *no* |
| $3 \to 2$ | 310 | - | 128 (537) | - | 195 (162) | 307 (12) |
| $4 \to 3$ | 386 | - | 137 (2809) | - | 54 (2606) | 249 (54) |
| $5 \to 4$ | 427 | - | - | - | - | 148 (13) |
| $6 \to 5$ | 459 | 13 (162) | - | 13 (60) | - | 180 (288) |

In Table 3, we report on the number of “successful” attempts to solve the Check(GHD, $k-1$) problem for hypergraphs with $hw = k$. Here “successful” means that the program terminated within 1 hour. For instance, for the 310 hypergraphs with $hw = 3$ in the HyperBench, the “global” computation terminated in 128 cases (i.e., 41%) when trying to solve Check(GHD, 2). The average runtime of these “successful” runs was 537 seconds. For the 386 hypergraphs with $hw = 4$, the “global” computation terminated in 137 cases (i.e., 35%) with average runtime 2809 when trying to solve the Check(GHD, 3) problem. For the 886 hypergraphs with $hw \in \{5,6\}$, the “global” computation only terminated in 13 cases (i.e., 1.4%). Overall, it turns out that the set $f(H,k)$ may be very big (even though it is polynomial if $k$ and $i$ are constants). Hence, $H'$ can become considerably bigger than $H$. This explains the frequent timeouts in the GlobalBIP column in Table 3.

**“Local” implementation.** Looking for ways to improve the $ghw$-algorithm, we closely inspect the role played by the set $f(H,k)$ in the tractability proof in [19]. The definition of this set is motivated by the problem that, in the top down construction of a GHD, we may want to choose at some node $u$ the bag $B_u$ such that $x \notin B_u$ for some variable $x \in B(\lambda_u) \cap V(T_u)$. This violates condition (4) of Definition 2.2 (the “special condition”) and is therefore forbidden in an HD. In particular, there exists an edge $e$ with $x \in e$ and $\lambda_u(e) = 1$. The crux of the $ghw$-algorithm in [19] is that for every such “missing” variable $x$, the set $f(H,k)$ contains a subedge $e' \subseteq e$ with $x \notin e'$. Hence, replacing $e$ by $e'$ in $\lambda_u$ (i.e., setting $\lambda_u(e) = 0$, $\lambda_u(e') = 1$ and leaving $\lambda_u$ unchanged elsewhere) eliminates the special condition violation. By the connectedness condition, it suffices to consider the intersections of $e$ with unions of edges that may possibly occur in bags of $T_u$ rather than with arbitrary edges in $E(H)$. In other words, for each node $u$ in the decomposition, we may restrict $f(H,k)$ to an appropriate subset $f_u(H,k) \subseteq f(H,k)$.

The results obtained with this enhanced version of the $ghw$-computation are shown in Table 3 in the column labelled “LocalBIP”. We call this $ghw$-computation “local” because the set $f_u(H,k)$ of subedges of $H$ to be added to the hypergraph is computed separately for each node $u$ of the decomposition. Recall that in this table, the "successful" calls of the program are recorded. Interestingly, for the hypergraphs with $hw = 3$, the "local" computation performs significantly better (namely 63% solved with average runtime 162 seconds rather than 41% with average runtime 537 seconds). In contrast, for the hypergraphs with $hw = 4$, the "global" computation is significantly more successful. For $hw \in \{5, 6\}$, the "global" and "local" computations are equally bad. A possible explanation for the reverse behaviour of "global" and "local" computation in case of $hw = 3$ as opposed to $hw = 4$ is that the restriction of the "global" set $f(H, k)$ of subedges to the "local" set $f_u(H, k)$ at each node $u$ seems to be quite effective for the hypergraphs with $hw = 3$. In contrast, the additional cost of having to compute $f_u(H, k)$ at each node $u$ becomes counter-productive, when the set of subedges thus eliminated is not significant. It is interesting to note that the sets of solved instances of the global computation and the local computation are incomparable, i.e., in some cases one method is better, while in other cases the other method is better.

**New alternative approach: "balanced separators".** We now propose a completely new approach, based on so-called "balanced separators". The latter are a familiar concept in graph theory [16, 46] – denoting a set $S$ of vertices of a graph $G$, such that the subgraph $G'$ induced by $V(G) \setminus S$ has no connected component larger than some given size, e.g., $\alpha \cdot |V|$ for some given $\alpha \in (0, 1)$. In our setting, we may consider the label $\lambda_u$ at some node $u$ in a GHD as *separator* in the sense that we can consider connected components of the subhypergraph $H'$ of $H$ induced by $V(H) \setminus B_u$. Clearly, in a GHD, we may consider any node as the root. So suppose that $u$ is the root of some GHD. Moreover, as is shown in [19] in the proof of tractability of Check(GHD, $k$) in case of the BIP, we may choose $\lambda_u$ such that $B(\lambda_u) = B_u$ if the subedges in $f(H, k)$ have been added to the hypergraph.

By the HD-algorithm from [24], we know that an HD of $H'$ (and, hence, a GHD of $H$) can be constructed in such a way that every subtree rooted at a child node $u_i$ of $u$ contains only one connected component $C_i$ of the subhypergraph of $H'$ induced by $V(H) \setminus B_u$. For our purposes, it is convenient to define the size of a component $C_i$ as the number of edges that have to be covered at some node in the subtree rooted at $u_i$ in the GHD. We thus call a separator $\lambda_u$ "balanced", if the size of each component $C_i$ is at most $|E(H')|/2$. The following observation is immediate:

Proposition 5.1. *In every GHD, there exists a node $u$ (which we may choose as the root) such that $\lambda_u$ is a balanced separator.*

This property allows us to design the algorithm sketched in Figure 4 to compute a GHD of $H'$. Actually, as will become clear below, we assume that the input to this recursive

**ALGORITHM** Find_GHD_via_balancedSeparators
// high-level description

**Input:** hypergraph $H'$, integer $k \geq 0$.
**Output:** a GHD $\langle T, B_u, \lambda_u \rangle$ of width $\leq k$ if exists,
"Reject", otherwise.

**Procedure** Find_GHD ($H$: Hypergraph,
*Sp*: Set of special edges)
**begin**

1. Base Case: if there are only two edges or special edges left then stop and return a GHD with one node for each edge or special edge, respectively.
2. Find a balanced separator: for all $\lambda\colon E(H') \to \{0, 1\}$ check if $\lambda_u$ is a balanced separator for $H$; if none is found then return Reject.
3. Split $H$ into connected components $C_1, \ldots, C_\ell$ w.r.t. $\lambda_u$: $C_i \subseteq V(H) \setminus B(\lambda_u)$ for every $i$ and $C_i$ is connected in $H$ and each $C_i$ is maximal with this property.
4. Build the pair $\langle H_i, Sp_i \rangle$ (the subhypergraph based on $C_i$ and the special edges in $C_i$) for each connected component $C_i$; add $B(\lambda_u)$ as one more special edge to each set $Sp_i$.
5. Call Find_GHD($H_i$, $Sp_i$) for each pair $\langle H_i, Sp_i \rangle$; each successful call returns a GHD $T_i$ for $H_i$ if one call returns Reject then return Reject.
6. Create and return a new GHD for $H$ having $\lambda_u$ as root: each $T_i$ has one leaf node labelled $B(\lambda_u)$; the new GHD is obtained by gluing together all subtrees $T_i$ at the node with label $B(\lambda_u)$.

**end**

**begin** (* Main *)
**return** Find_GHD ($H'$, $\emptyset$);
**end**

**Figure 4: GHD-algorithm via balanced separators**

algorithm consists of a hypergraph plus a set $Sp$ of "special edges" and we request that the GHD to be constructed contains "special nodes", which (a) have to be leaf nodes in the decomposition and (b) the $\lambda$-label of such a leaf node consists of a single special edge only. Each special edge contains the set of vertices $B_u$ of some balanced separator $\lambda_u$ further up in the hierarchy of recursive calls of the decomposition algorithm. The special edges are propagated to the recursive calls for subhypergraphs in order to determine how to assemble the overall GHD from the GHDs of the subhypergraphs. This will become clearer in the proof sketch of Theorem 5.2 which, for space reasons, is given in the appendix.

Theorem 5.2. *Let $H$ be a hypergraph, let $k \geq 1$, and let $H'$ be obtained from $H$ by adding the subedges in $f(H, k)$ to $E(H)$. Then the algorithm Find_GHD_via_balancedSeparators*

**Table 4: GHW of instances with average runtime in s**

| $hw \rightarrow ghw$ | yes | no | timeout |
|---|---|---|---|
| $3 \rightarrow 2$ | 0 | 309 (10) | 1 |
| $4 \rightarrow 3$ | 0 | 262 (57) | 124 |
| $5 \rightarrow 4$ | 0 | 148 (13) | 279 |
| $6 \rightarrow 5$ | 18 (129) | 180 (288) | 261 |

*given in Figure 4 outputs a GHD of width $\leq k$ if one exists and rejects otherwise.*

If we look at the number of solved instances in Table 3, we see that the recursive algorithm via balanced separators (reported in the last column labelled `BalSep`) has the least number of timeouts due to the fast identification of negative instances (i.e., those with no-answer), where it often detects quite fast that a given hypergraph does not have a balanced separator of desired width. As $k$ increases, the performance of the balanced separators approach deteriorates. This is due to $k$ in the exponent of the running time of our algorithm, i.e. we need to check for each of the possible $O(n^{k+1})$ combinations of $\leq k$ edges if it constitutes a balanced separator. Note that the balanced separators approach only terminated in case of no-answers. We have therefore omitted the yes-column for this algorithm in Table 3.

**Empirical results.** We now look at Table 4, where we report for all hypergraphs with $hw \leq k$ and $k \in \{3, 4, 5, 6\}$, whether $ghw \leq k - 1$ could be verified. To this end, we run our three algorithms ("global", "local", and "balanced separators") in parallel and stop the computation, as soon as one terminates (with answer "yes" or "no"). The number in parentheses refers to the average runtime needed by the fastest of the three algorithms in each case. A timeout occurs if none of the three algorithms terminates within 3,600 seconds. It is interesting to note that in the vast majority of cases, no improvement of the width is possible when we switch from $hw$ to $ghw$: in 98% of the solved cases and 57% of all instances with $hw \leq 6$, $hw$ and $ghw$ have identical values. Actually, we think that the high percentage of the *solved cases* gives a more realistic picture than the percentage of *all cases* for the following reason: our algorithms (in particular, the "global" and "local" computations) need particularly long time for negative instances. This is due to the fact that in a negative case, "all" possible choices of $\lambda$-labels for a node $u$ in the GHD have to be tested before we can be sure that no GHD of $H$ (or, equivalently, no HD of $H'$) of desired width exists. Hence, it seems plausible that the timeouts are mainly due to negative instances. This also explains why our new GHD algorithm in Figure 4, which is particularly well suited for negative instances, has the least number of timeouts.

We conclude this section with a final observation: in Figure 2, we had many cases, for which only some upper bound $k$ on the $hw$ could be determined, namely those cases, where the attempt to solve Check(HD, $k$) yields a yes-answer and the attempt to solve Check(HD, $k - 1$) gives a timeout. In several such cases, we could get (with the balanced separator approach) a no-answer for the Check(GHD, $k - 1$) problem, which implicitly gives a no-answer for the problem Check(HD, $k - 1$). In this way, our new $ghw$-algorithm is also profitable for the $hw$-computation: for 827 instances with $hw \leq 6$, we were not able to determine the exact hypertree width. Using our new $ghw$-algorithm, we closed this gap for 297 instances; for these instances $hw = ghw$ holds.

To sum up, we now have a total of 1,778 (58%) instances for which we determined the exact $hw$ and a total of 1,406 instances (46%) for which we determined the exact $ghw$. Out of these, 1,390 instances had identical values for $hw$ and $ghw$. In 16 cases, we found an improvement of the width by 1 when moving from $hw$ to $ghw$, namely from $hw = 6$ to $ghw = 5$. In 2 further cases, we could show $hw \leq 6$ and $ghw \leq 5$, but the attempt to check $hw = 5$ or $ghw = 4$ led to a timeout. Hence, in response to *Goal 6*, $hw$ is equal to $ghw$ in 45% of the cases if we consider all instances and in 60% of the cases (1,390 of 2,308) with small width ($hw \leq 6$). However, if we consider the fully solved cases (i.e., where we have the precise value of $hw$ and $ghw$), then $hw$ and $ghw$ coincide in 99% of the cases (1,390 of 1,406).

# 6 FRACTIONALLY IMPROVED DECOMPOSITIONS

The algorithms proposed in the literature for computing FHDs are very expensive. For instance, even the algorithm used for the tractability result in [18] for hypergraphs of low degree is problematical since it involves a double-exponential factor in the degree. Therefore, we investigate the potential of a simplified method to compute approximated FHDs. Below, we present two algorithms for such approximated FHD computations – with a trade-off between computational cost and quality of the approximation.

- The simplest way to obtain a fractionally improved (G)HD is to take either a GHD or HD as input and compute a fractionally improved (G)HD. To this end, an algorithm (which we refer to as SimpleImproveHD) visits each node $u$ of a given GHD or HD and computes an optimal fractional edge cover $\gamma_u$ for the set $B_u$ of vertices. This algorithm is simple and computationally inexpensive, provided that we can start off with a GHD or HD that was computed before. In our case, we simply took the HD resulting from the $hw$-computation reported in Figure 2. Clearly, this approach is rather naive and the dependence on a concrete HD is unsatisfactory. We therefore move to a more sophisticated algorithm.

**Table 5: Instances solved with `SimpleImproveHD`**

| $hw$ | $\geq 1$ | $[0.5, 1)$ | $[0.1, 0.5)$ | no | timeout |
|---|---|---|---|---|---|
| 2 | 0 | 41 | 25 | 172 | 0 |
| 3 | 12 | 104 | 25 | 169 | 0 |
| 4 | 9 | 55 | 11 | 311 | 0 |
| 5 | 20 | 14 | 11 | 382 | 0 |
| 6 | 12 | 60 | 80 | 309 | 0 |

**Table 6: Instances solved with `FracImproveHD`**

| $hw$ | $\geq 1$ | $[0.5, 1)$ | $[0.1, 0.5)$ | no | timeout |
|---|---|---|---|---|---|
| 2 | 0 | 46 | 29 | 160 | 1 |
| 3 | 14 | 116 | 21 | 135 | 24 |
| 4 | 11 | 81 | 2 | 8 | 284 |
| 5 | 18 | 126 | 59 | 2 | 222 |
| 6 | 28 | 149 | 95 | 4 | 183 |

- The algorithm FracImproveHD has as input a hypergraph $H$ and numbers $k, k' \geq 1$, where $k$ is an upper bound on the $hw$ and $k'$ the desired fractionally improved $hw$. We search for an FHD $\mathcal{D}'$ with $\mathcal{D}' = SimpleImproveHD(\mathcal{D})$ for some HD $\mathcal{D}$ of $H$ with $width(\mathcal{D}) \leq k$ and $width(\mathcal{D}') \leq k'$. In other words, this algorithm searches for the best fractionally improved HD over all HDs of width $\leq k$. Hence, the result is independent of any concrete HD.

The experimental results with these algorithms for computing fractionally improved HDs are summarized in Tables 5 and 6. We have applied these algorithms to all hypergraphs for which $hw \leq k$ with $k \in \{2, 3, 4, 5\}$ is known from Figure 2. The various columns of the Tables 5 and 6 are as follows: the first column (labelled $hw$) refers to the (upper bound on the) $hw$ according to Figure 2. The next 3 columns, labelled $\geq 1$, $[0.5, 1)$, and $[0.1, 0.5)$ tell us, by how much the width can be improved (if at all) if we compute an FHD by one of the two algorithms. We thus distinguish the 3 cases if, for a hypergraph of $hw \leq k$, we manage to construct an FHD of width $k-c$ for $c \geq 1$, $c \in [0.5, 1)$, or $c \in [0.1, 0.5)$. The column with label "no" refers to the cases where no improvement at all or at least no improvement by $c \geq 0.1$ was possible. The last column counts the number of timeouts.

For instance, in the first row of Table 5, we see that (with the SimpleImproveHD algorithm and starting from the HD obtained by the $hw$-computation of Figure 2) out of 238 hypergraphs with $hw = 2$, no improvement was possible in 172 cases. In the remaining 66 cases, an improvement to a width of at most $2 - 0.5$ was possible in 25 cases and an improvement to $k - c$ with $c \in [0.1, 0.5)$ was possible in 41 cases. For the hypergraphs with $hw = 3$ in Figure 2, almost half of the hypergraphs (141 out of 310) allowed at least some improvement, in particular, 104 by $c \in [0.5, 1)$ and 12 even by at least 1. The improvements achieved for the hypergraphs with $hw \leq 4$ and $hw \leq 5$ are less significant.

The results obtained with our FracImproveHD implementation are displayed in Table 6. We see that the number of hypergraphs which allow for a fractional improvement of the width by at least 0.5 or even by 1 is often bigger than with SimpleImproveHD – in particular in the cases where $k' \leq k$ with $k \in \{4, 5\}$ holds. In the other cases, the results obtained with the naive SimpleImproveHD algorithm are not much worse than with the more sophisticated FracImproveHD algorithm.

## 7 RELATED WORK

We distinguish several types of works that are highly relevant to ours. The works most closely related are the descriptions of HD, GHD and FHD algorithms in [19, 24] and the implementation of HD computation by the `DetKDecomp` program reported in [26]. We have extended these works in several ways. Above all, we have incorporated our analysis tool (reported in Sections 3 and 4) and the GHD and FHD computations (reported in Sections 5 and 6) into the `DetKDecomp` program – resulting in our `NewDetKDecomp` library, which is openly available on GitHub. For the GHD computation, we have added heuristics to speed up the basic algorithm from [19]. Moreover, we have proposed a novel approach via balanced separators, which allowed us to significantly extend the range of instances for which the GHD computation terminates in reasonable time. We have also introduced a new form of decomposition method: the fractionally improved decompositions (see Section 6), which allow for a practical, lightweight form of FHDs.

The second important input to our work comes from the various sources [6, 9–11, 20, 26, 30, 35, 47] which we took our CQs and CSPs from. Note that our main goal was not to add further CQs and/or CSPs to these benchmarks. Instead, we have aimed at taking and combining existing, openly accessible benchmarks of CQs and CSPs, convert them into hypergraphs, which are then thoroughly analysed. Finally, the hypergraphs and the analysis results are made openly accessible again.

The third kind of works highly relevant to ours are previous analyses of CQs and CSPs. To the best of our knowledge, Ghionna et al. [21] presented the first systematic study of HDs of benchmark CQs from TPC-H. However, Ghionna et al. pursued a research goal different from ours in that they primarily wanted to find out to what extent HDs can actually speed up query evaluation. They achieved very positive results in this respect, which have recently been confirmed by the work of Perelman et al. [41], Tu et al. [48] and Aberger et al. [1] on query evaluation using FHDs. As a side result, Ghionna et al. also detected that CQs tend to have low hypertree width (a finding which was later confirmed in [12, 42] and also in our study). In a pioneering effort, Bonifati, Martens, and Timm [12] have recently analysed an unprecedented, massive amount of queries: they investigated 180,653,910 queries from (not openly available) query logs of several popular SPARQL endpoints. After elimination of duplicate queries, there were still 56,164,661 queries left, out of which 26,157,880 queries were in fact CQs. The authors thus significantly extend previous work by Picalausa and Vansummeren [42], who analysed 3,130,177 SPARQL queries posed by humans and software robots at the DBPedia SPARQL endpoint. The focus in [42] is on structural properties of SPARQL queries such as keywords used and variable structure in optional patterns. There is one paragraph devoted to CQs, where it is noted that 99.99% of ca. 2 million CQs considered in [42] are acyclic.

Many of the CQs (over 15 million) analysed in [12] have arity 2 (here we consider the maximum arity of all atoms in a CQ as the arity of the query), which means that all triples in such a SPARQL query have a constant at the predicate-position. Bonifati et al. made several interesting observations concerning the shape of these graph-like queries. For instance, they detected that exactly one of these queries has $tw = 3$, while all others have $tw \leq 2$ (and hence $hw \leq 2$). As far as the CQs of arity 3 are concerned (for CQs expressed as SPARQL queries, this is the maximum arity achievable), among many characteristics, also the hypertree width was computed by using the original `DetKDecomp` program from [26]. Out of 6,959,510 CQs of arity 3, only 86 (i.e. 0.01‰) turned out to have $hw = 2$ and 8 queries had $hw = 3$ , while all other CQs of arity 3 are acyclic. Our analysis confirms that, also for non-random CQs of arity $> 3$, the hypertree width indeed tends to be low, with the majority of queries being even acyclic.

For the analysis of CSPs, much less work has been done. Although it has been shown that exploiting (hyper-) tree decompositions may significantly improve the performance of CSP solving [4, 29, 31, 34], a systematic study on the (generalized) hypertree width of CSP instances has only been carried out by few works [26, 34, 45]. To the best of our knowledge, we are the first to analyse the $hw$, $ghw$, and $fhw$ of ca. 2,000 CSP instances, where most of these instances have not been studied in this respect before.

It should be noted that the focus of our work is different from the above mentioned previous works: above all, we wanted to test the practical feasibility of various algorithms for HD, GHD, and FHD computation (including both, previously presented algorithms and new ones developed as part of this work). As far as our repository of hypergraphs (obtained from CQs and CSPs) is concerned, we emphasize open accessibility. Thus, users can analyse their CQs and CSPs (with our implementations of HD, GHD, and FHD algorithms) or they can analyse new decomposition algorithms (with our hypergraphs, which cover quite a broad range of characteristics). In fact, in the recent work on FHD computation via SMT solving [17], the Hyperbench benchmark has already been used for the experimental evaluation. In [17] a novel approach to $fhw$ computation via an efficient encoding of the check-problem for FHDs to SMT (SAT modulo Theory) is presented. The tests were carried out with 2,191 hypergraphs from the initial version of the HyperBench. For all of these hypergraphs we have established at least some upper bound on the $fhw$ either by our $hw$-computation or by one of our new algorithms presented in Sections 5 and 6. In contrast, the exact algorithm in [17] found FHDs only for 1.449 instances (66%). In 852 cases, both our algorithms and the algorithm in [17] found FHDs of the same width; in 560 cases, an FHD of lower width was found in [17]. By using the same benchmark for the tests, the results in [17] and ours are comparable and have thus provided valuable input for future improvements of the algorithms by combining the different strengths and weaknesses of the two approaches.

The use of the same benchmark has also allowed us to provide feedback to the authors of [17] for debugging their system: in 9 out of 2,191 cases, the "optimal" value for the $fhw$ computed in [19] was apparently erroneous, since it was higher than the $hw$ found out by our analysis; note that upper bounds on the width are, in general, more reliable than lower bounds since it is easy to verify if a given decomposition indeed has the desired properties, whereas ruling out the existence of a decomposition of a certain width is a complex and error-prone task.

## 8 CONCLUSION

In this work, we have presented HyperBench, a new and comprehensive benchmark of hypergraphs derived from CQs and CSPs from various areas, together with the results of extensive empirical analyses with this benchmark.

**Lessons learned.** The empirical study has brought many insights. Below, we summarize the most important lessons learned from our studies.

- The finding of [12, 42] that non-random CQs have low hypertree width has been confirmed by our analysis, even if (in contrast to SPARQL queries) the arity of the CQs is not bounded by 3. For random CQs and CSPs, we have detected a correlation between the arity and the hypertree width, although also in this case, the increase of the $hw$ with increased arity is not dramatic.
- In [19], several hypergraph invariants were identified, which make the computation of GHDs and the approximation of FHDs tractable. We have seen that, at least for non-random instances, these invariants indeed have low values.

- The reduction of the *ghw*-computation problem to the *hw*-computation problem in case of low intersection width turned out to be more problematical than the theoretical tractability results from [19] had suggested. Even the improvement by "local" computation of the additional subedges did not help much. However, we were able to improve this significantly by presenting a new algorithm based on "balanced separators". In particular for negative instances (i.e., those with a no-answer), this approach proved very effective.

- An additional benefit of the new *ghw*-algorithm based on "balanced separators" is that it allowed us to also fill gaps in the *hw*-computation. Indeed, in several cases, we managed to verify $hw \le k$ for some $k$ but we could not show $hw \nleq k-1$, due to a timeout for $\textsc{Check}(\mathrm{HD}, k-1)$. By establishing $ghw \nleq k-1$ with our new GHD-algorithm, we have implicitly showed $hw \nleq k-1$. This allowed us to compute the exact *hw* of many further hypergraphs.

- Most surprisingly, the discrepancy between *hw* and *ghw* is much lower than expected. Theoretically, only the upper bound $hw \le 3 \cdot ghw + 1$ is known. However, in practice, when considering hypergraphs of $hw \le 6$, we could show that in 53% of all cases, *hw* and *ghw* are simply identical. Moreover, in *all* cases when one of our implementations of *ghw*-computation terminated on instances with $hw \le 5$, we got identical values for *hw* and *ghw*.

**Future work.** Our empirical study has also given us many hints for future directions of research. We find the following tasks particularly urgent and/or rewarding.

- So far, we have only implemented the *ghw*-computation in case of low intersection width. In [19], tractability of the $\textsc{Check}(\mathrm{GHD}, k)$ problem was also proved for the more relaxed bounded multi-intersection width. Our empirical results in Table 2 show that, apart from the random CQs and random CSPs, the 3-multi-intersection is $\le 2$ in almost all cases. It seems therefore worthwhile to implement and test also the BMIP-algorithm from [19].

- The three approaches for *ghw*-computation presented here turned out to have complementary strengths and weaknesses. This was profitable when running all three algorithms in parallel and taking the result of the first one that terminates (see Table 4). In the future, we also want to implement a more sophisticated combination of the various approaches: for instance, one could try to apply our new "balanced separator" algorithm recursively only down to a certain recursion depth (say depth 2 or 3) to split a big given hypergraph into smaller subhypergraphs and then continue with the "global" or "local" computation from Section 5.

- Our new approach to *ghw*-computation via "balanced separators" proved quite effective in our experiments. However, further theoretical underpinning of this approach is missing. The empirical results obtained for our new GHD algorithm via balanced separators suggest that the number of balanced separators is often drastically smaller than the number of arbitrary separators. We want to determine a realistic upper bound on the number of balanced separators in terms of $n$ (the number of edges) and $k$ (an upper bound on the width). This will then allow us to compute also a realistic upper bound on the runtime of this new algorithm.

- Finally, we want to further extend the HyperBench benchmark and tool in several directions. We will thus incorporate further implementations of decomposition algorithms from the literature such as the GHD- and FHD computation in [39] or the polynomial-time FHD computation for hypergraphs of bounded degree in [18]. Moreover, we will continue to fill in hypergraphs from further sources of CSPs and CQs. For instance, in [1, 13, 21, 22] a collection of CQs for the experimental evaluations in those papers is mentioned. We will invite the authors to disclose these CQs and incorporate them into the HyperBench benchmark.

- Very recently, a new, huge, publicly available query log has been reported in [37]. It contains over 200 million SPARQL queries on Wikidata. In the paper, the anonymisation and publication of the query logs is mentioned as future work. However, on their web site, the authors have meanwhile made these queries available. At first glance, these queries seem to display a similar behaviour as the SPARQL queries collected by Bonifatti et al. [12]: there is a big number of single-atom queries and again, the vast majority of the queries is acyclic. A detailed analysis of the query log in the style of [12] constitutes an important goal for future research.

## ACKNOWLEDGMENTS

This work was supported by the Austrian Science Fund (FWF) project P30930-N35. Davide Mario Longo's work was supported by the Austrian Science Fund (FWF) project W1255-N23. We would like to thank Angela Bonifati, Wim Martens, and Thomas Timm for sharing most of the hypergraphs with $hw \ge 2$ from their work [12] and for their effort in anonymising these hypergraphs, which was required by the license restrictions.

# Appendix

Below, we give a proof a proof sketch of Theorem 5.2.

Proof Sketch. Steps 1–5 of the algorithm in Figure 4 essentially correspond to the computation of $\lambda_u$ and $B_u$ for the root node $u$ in the HD-computation of [24]. The most significant modifications here are due to the handling of "special edges" in parameter $Sp$. A crucial property of the construction in [24] and also of our construction here is that each subtree below node $u$ in the decomposition only contains vertices from a single connected component $C_i$ w.r.t. $V(H) \setminus B_u$ (see Steps 3 and 4). Since the special edges come from such bags $B_u$, special edges can never be used as separators in recursive calls below. Hence, we can exclude special edges from the search for a balanced separator in Step 2. The base case (in Step 1) is reached for $|E(H) \cup Sp| \leq 2$.

The correctness of assembling a GHD (in Step 6) from the results of the recursive calls can be shown by structural induction on the tree structure of a GHD: suppose that the recursive calls in the algorithm for each hypergraph $H_i$ with set $Sp_i$ of special edges are correct, i.e., they yield for each hypergraph $H_i$ a GHD $\mathcal{D}_i$ such that each special edge $s$ in $Sp_i$ is indeed covered by a leaf node in $\mathcal{D}_i$ whose $\lambda$-label consists of $s$ only. In particular, since $s = B(\lambda_u)$ is a special edge contained in $Sp_i$ for each $i$, there exists a leaf node $t_i$ in $\mathcal{D}_i$ with $\lambda_{t_i} = \{s\}$. In a GHD, any node can be taken as the root. We thus choose $t_i$ as the root node in each GHD $\mathcal{D}_i$. By construction, we have $B_u = B_{t_1} = \cdots = B_{t_\ell}$. Moreover, any two subhypergraphs $H_i$, $H_j$ contain the vertices from two different connected components. Hence, apart from the vertices contained in the special edge $s$, any two GHDs $\mathcal{D}_i$, $\mathcal{D}_j$ with $i \neq j$ have no vertices in common. We can therefore construct a GHD $\mathcal{D}$ of $H'$ by deleting the root node $t_i$ from each GHD $\mathcal{D}_i$ and by appending the child nodes of each $t_i$ directly as child nodes of $u$. Clearly, the connectedness condition is satisfied in the resulting decomposition. □

## REFERENCES


[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44. https://doi.org/10.1145/3129246

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. Old Techniques for New Join Algorithms: A Case Study in RDF Processing. *CoRR* abs/1602.03557 (2016). arXiv:1602.03557 http://arxiv.org/abs/1602.03557

[3] Isolde Adler, Georg Gottlob, and Martin Grohe. 2007. Hypertree width and related hypergraph invariants. *Eur. J. Comb.* 28, 8 (2007), 2167–2181. https://doi.org/10.1016/j.ejc.2007.04.013

[4] Kamal Amroun, Zineb Habbas, and Wassila Aggoune-Mtalaa. 2016. A compressed Generalized Hypertree Decomposition-based solving technique for non-binary Constraint Satisfaction Problems. *AI Commun.* 29, 2 (2016), 371–392. https://doi.org/10.3233/AIC-150694

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1371–1382. https://doi.org/10.1145/2723372.2742796

[6] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The iBench Integration Metadata Generator. *PVLDB* 9, 3 (2015), 108–119. https://doi.org/10.14778/2850583.2850586

[7] Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. 2016. XCSP3: an XML-based format designed to represent combinatorial constrained problems. http://www.xcsp.org/

[8] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[9] Michael Benedikt. 2017. CQ benchmarks. Personal Communication.

[10] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 37–52. https://doi.org/10.1145/3034786.3034796

[11] Jeremias Berg, Neha Lodha, Matti Järvisalo, and Stefan Szeider. 2017. MaxSAT Benchmarks based on Determining Generalized Hypertree-width. *MaxSAT Evaluation 2017: Solver and Benchmark Descriptions* B-2017-2 (2017), 22.

[12] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *PVLDB* 11, 2 (2017), 149–161. https://doi.org/10.14778/3149193.3149196

[13] Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. 2017. Efficiently Enumerating Minimal Triangulations. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 273–287. https://doi.org/10.1145/3034786.3056109

[14] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison (Eds.). ACM, 77–90. https://doi.org/10.1145/800105.803397

[15] Rina Dechter. 2003. *Constraint Processing*. Elsevier. https://doi.org/10.1016/b978-1-55860-890-0.x5000-2

[16] Uriel Feige and Mohammad Mahdian. 2006. Finding small balanced separators. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, Jon M. Kleinberg (Ed.). ACM, 375–384. https://doi.org/10.1145/1132516.1132573

[17] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. 2018. An SMT Approach to Fractional Hypertree Width. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science)*, John N. Hooker (Ed.), Vol. 11008. Springer, 109–127. https://doi.org/10.1007/978-3-319-98334-9_8

[18] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. 2017. Tractable Cases for Recognizing Low Fractional Hypertree Width. *viXra.org e-prints* viXra:1708.0373 (2017). http://vixra.org/abs/1708.0373

[19] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. 2018. General and Fractional Hypertree Decompositions: Hard and Easy Cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*,

Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 17–32. https://doi.org/10.1145/3196959.3196962

[20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2014. Mapping and cleaning. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 232–243. https://doi.org/10.1109/ICDE.2014.6816654

[21] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. 2007. Hypertree Decompositions for Query Optimization. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 36–45. https://doi.org/10.1109/ICDE.2007.367849

[22] Lucantonio Ghionna, Gianluigi Greco, and Francesco Scarcello. 2011. H-DB: a hybrid quantitative-structural sql optimizer. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 2573–2576. https://doi.org/10.1145/2063576.2064023

[23] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *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*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 57–74. https://doi.org/10.1145/2902251.2902309

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627. https://doi.org/10.1006/jcss.2001.1809

[25] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM* 56, 6 (2009), 30:1–30:32. https://doi.org/10.1145/1568318.1568320

[26] Georg Gottlob and Marko Samer. 2008. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics* 13 (2008), 1:1.1–1:1.19. https://doi.org/10.1145/1412228.1412229

[27] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1 (2014), 4:1–4:20. https://doi.org/10.1145/2636918

[28] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3, 2-3 (2005), 158–182. https://doi.org/10.1016/j.websem.2005.06.005

[29] Zineb Habbas, Kamal Amroun, and Daniel Singer. 2015. A Forward-Checking algorithm based on a Generalised Hypertree Decomposition for solving non-binary constraint satisfaction problems. *J. Exp. Theor. Artif. Intell.* 27, 5 (2015), 649–671. https://doi.org/10.1080/0952813X.2014.993507

[30] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 281–293. https://doi.org/10.1145/2882903.2882957

[31] Shant Karakashian, Robert J. Woodward, and Berthe Y. Choueiry. 2011. Reformulating R(*, m)C with Tree Decomposition. In *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona, Catalonia, Spain, July 17-18, 2011.*, Michael R. Genesereth and Peter Z. Revesz (Eds.). AAAI, 62–69. http://www.aaai.org/ocs/index.php/SARA/SARA11/paper/view/4234

[32] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.* 41, 4 (2016), 22:1–22:45. https://doi.org/10.1145/2967101

[33] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *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*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. https://doi.org/10.1145/2902251.2902280

[34] Mohammed Lalou, Zineb Habbas, and Kamal Amroun. 2009. Solving Hypertree Structured CSP: Sequential and Parallel Approaches. In *Proceedings of the 16th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA@AI*IA 2009, Reggio Emilia, Italy, December 11-12, 2009 (CEUR Workshop Proceedings)*, Marco Gavanelli and Toni Mancini (Eds.), Vol. 589. CEUR-WS.org. http://ceur-ws.org/Vol-589/paper11.pdf

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[36] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[37] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl (Eds.), Vol. 11137. Springer, 376–394. https://doi.org/10.1007/978-3-030-00668-6_23

[38] Dániel Marx. 2010. Approximating fractional hypertree width. *ACM Trans. Algorithms* 6, 2 (2010), 29:1–29:17. https://doi.org/10.1145/1721837.1721845

[39] Lukas Moll, Siamak Tazari, and Marc Thurley. 2012. Computing hypergraph width measures exactly. *Inf. Process. Lett.* 112, 6 (2012), 238–242. https://doi.org/10.1016/j.ipl.2011.12.002

[40] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[41] Adam Perelman and Christopher Ré. 2015. DunceCap: Compiling Worst-Case Optimal Query Plans. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 2075–2076. https://doi.org/10.1145/2723372.2764945

[42] François Picalausa and Stijn Vansummeren. 2011. What are real SPARQL queries like?. In *Proceedings of the International Workshop on Semantic Web Information Management, SWIM 2011, Athens, Greece, June 12, 2011*, Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca (Eds.). ACM, 7. https://doi.org/10.1145/1999299.1999306

[43] Rachel Pottinger and Alon Y. Halevy. 2001. MiniCon: A scalable algorithm for answering queries using views. *VLDB J.* 10, 2-3 (2001), 182–198. https://doi.org/10.1007/s007780100048

[44] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. 2007. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.* 73, 3 (2007), 475–506. https://doi.org/10.1016/j.jcss.2006.10.010

[45] Werner Schafhauser. 2006. *New heuristic methods for tree decompositions and generalized hypertree decompositions*. Master's thesis. Technische Universität Wien.

[46] Aaron Schild and Christian Sommer. 2015. On Balanced Separators in Road Networks. In *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings (Lecture Notes in Computer Science)*, Evripidis Bampis (Ed.), Vol. 9125. Springer, 286–297. https://doi.org/10.1007/978-3-319-20086-6_22

[47] Transaction Processing Performance Council (TPC). 2014. TPC-H decision support benchmark. http://www.tpc.org/tpch/default.asp

[48] Susan Tu and Christopher Ré. 2015. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 2077–2078. https://doi.org/10.1145/2723372.2764946

[49] V. N. Vapnik and A. Ya. Chervonenkis. 1971. On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. *Theory of Probability & Its Applications* 16, 2 (jan 1971), 264–280. https://doi.org/10.1137/1116025