

# Compressed Representations of Conjunctive Query Results

Shaleen Deep

University of Wisconsin-Madison  
shaleen@cs.wisc.edu

Paraschos Koutris

University of Wisconsin-Madison  
paris@cs.wisc.edu

## ABSTRACT

Relational queries, and in particular join queries, often generate large output results when executed over a huge dataset. In such cases, it is often infeasible to store the whole materialized output if we plan to reuse it further down a data processing pipeline. Motivated by this problem, we study the construction of space-efficient *compressed representations* of the output of conjunctive queries, with the goal of supporting the efficient access of the intermediate compressed result for a given access pattern. In particular, we initiate the study of an important tradeoff: minimizing the *space* necessary to store the compressed result, versus minimizing the *answer time* and *delay* for an access request over the result. Our main contribution is a novel parameterized data structure, which can be tuned to trade off space for answer time. The tradeoff allows us to control the space requirement of the data structure precisely, and depends both on the structure of the query and the access pattern. We show how we can use the data structure in conjunction with query decomposition techniques in order to efficiently represent the outputs for several classes of conjunctive queries.

## KEYWORDS

Query enumeration; Compressed representation; Space delay trade-off; Join algorithms; Constant delay enumeration

### ACM Reference Format:

Shaleen Deep and Paraschos Koutris. 2018. Compressed Representations of Conjunctive Query Results. In *PODS'18: 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, June 10–15, 2018, Houston, TX, USA*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3196959.3196979>

## 1 INTRODUCTION

In this paper, we study the problem of constructing space-efficient *compressed representations* of the output of conjunctive query results, with the goal of efficiently supporting a given access pattern directly over the compressed result, instead of the original input database. In many data management tasks, the data processing pipeline repeatedly accesses the result of a conjunctive query (CQ) using a particular access pattern. In the simplest case, this access pattern can be to enumerate the full result (e.g., in a multiquery optimization context). Generally, the access pattern can specify,

or *bound*, the values of some variables, and ask to enumerate the values of the remaining variables that satisfy the query.

Currently, there are two extremal solutions for this problem. In one extreme, we can materialize the full result of the CQ and index the result according to the access pattern. However, since the output result can often be extremely large, storing this index can be prohibitively expensive. In the other extreme, we can service each access request by executing the CQ directly over the input database every time. This solution does not need extra storage, but can lead to inefficiencies, since computation has to be done from scratch and may be redundant. In this work, we explore the design space between these two extremes. In other words, we want to compress the query output such that it can be stored in a space-efficient way, while we can support a given access pattern over the output as fast as possible.

**Example 1.** *Suppose we want to perform an analysis about mutual friends of users in a social network. The friend relation is represented by a symmetric binary relation  $R$  of size  $N$ , where a tuple  $R(a, b)$  denotes that user  $a$  is a friend of user  $b$ . The data analysis involves accessing the database through the following pattern: given any two users  $x$  and  $z$  who are friends, return all mutual friends  $y$ . We formalize this task through an adorned view  $V^{\text{bfb}}(x, y, z) = R(x, y), R(y, z), R(z, x)$ . The above formalism says that the view  $V$  of the database will be accessed as follows: given values for the bound (b) variables  $x, z$ , we have to return the values for the free (f) variable  $y$  such that the tuple is in the view  $V$ . The sequence bfb is called the access pattern for the adorned view.*

*One option to solve this problem is to satisfy each access by evaluating a query on the input database. This approach is space-efficient, since we work directly on the input and need space  $O(N)$ . However, we may potentially have to wait  $\Omega(N)$  time to even learn whether there is any returned value for  $y$ . A second option is to materialize the view  $V(x, y, z)$  and build a hash index with key  $(x, z)$ : in this case, we can satisfy any access optimally with constant delay  $\tilde{O}(1)$ .<sup>1</sup> On the other hand, the space needed for storing the view can be  $\Omega(N^{3/2})$ .*

*In this scenario, we would like to construct representations that trade off between space and delay (or answer time). As we will show later, for this particular example we can construct a data structure for any parameter  $\tau$  that needs space  $O(N^{3/2}/\tau)$ , and can answer any access request with delay  $\tilde{O}(\tau)$ .*

The idea of efficiently compressing query results has recently gained considerable attention, both in the context of factorized databases [23], as well as constant-delay enumeration [5, 27]. In these settings, the focus is to construct compressed representations that allow for enumeration of the full result with constant delay: this means that the time between outputting two consecutive tuples is  $O(1)$ , independent of the size of the data. Using factorization techniques, for any input database  $D$ , we can construct a

<sup>1</sup>the  $\tilde{O}$  notation includes a poly-logarithmic dependence on  $N$ .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

PODS'18, June 10–15, 2018, Houston, TX, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-4706-8/18/06...\$15.00

<https://doi.org/10.1145/3196959.3196979>

compressed data structure for any CQ without projections, called a  $d$ -representation, using space  $O(|D|^{\text{fhw}})$ , where  $\text{fhw}$  is the *fractional hypertree width* of the query [23]. Such a  $d$ -representation guarantees constant delay enumeration of the full result. In [5, 26], the compression of CQs with projections is also studied, but the setting is restricted to  $O(|D|)$  time preprocessing –which also restricts the size of the compressed representation to  $O(|D|)$ .

In this work, we show that we can dramatically decrease the space for the compressed representation by both (i) taking advantage of the access pattern, and (ii) tolerating a possibly increased delay. For instance, a  $d$ -representation for the query in Example 1 needs  $O(N^{3/2})$  space, while no linear-time preprocessing can support constant delay enumeration (under reasonable complexity assumptions [5]). However, we show that if we are willing to tolerate a delay of  $\tilde{O}(N^{1/2})$ , we can support the access pattern of Example 1 using only  $\tilde{O}(N)$  space, linear in the input size.

**Applications.** We illustrate the applicability of compressed representations of conjunctive queries on two practical problems: (i) processing graph queries over relational databases, and (ii) scaling up statistical inference engines.

In the context of graph analytics, the graph to be analyzed is often defined as a *declarative* query over a relational schema [2, 29–31]. For instance, consider the DBLP dataset, which contains information about which authors write which papers through a table  $R(\text{author}, \text{paper})$ . To analyze the relationships between co-authors, we will need to extract the *co-author graph*, which we can express as the view  $V(x, y) = R(x, p), R(y, p)$ . Most graph analytics algorithms typically access such a graph through an API that asks for the set of neighbors of a given vertex, which corresponds to the adorned view  $V^{\text{bf}}(x, y) = R(x, p), R(y, p)$ . Since the option of materializing the whole graph (here defined as the view  $V$ ) may require prohibitively large space, it is desirable to apply techniques that compress  $V$ , while we can still answer any access request efficiently. Recent work [29] has proposed compression techniques for this particular domain, but these techniques are limited to adorned views of the form  $V^{\text{bf}}(x, y)$ , rely on heuristics, and do not provide any formal analysis on the tradeoff between space and runtime.

The second application of query compression is in statistical inference. For example, FELIX [21] is an inference engine for Markov Logic Networks over relational data, which provides scalability by optimizing the access patterns of logical rules that are evaluated during inference. These access patterns over rules are modeled exactly as adorned views. FELIX groups the relations in the body of the view in partitions (optimizing for some cost function), and then materializes each partition (which corresponds to materializing a subquery). In the one extreme, it will eagerly materialize the whole view, and in the other extreme it will lazily materialize nothing. The materialization in FELIX is discrete, in that it is not possible to partially materialize each subquery. In contrast, we consider materialization strategies that explore the full continuum between the two extremes.

**Our Contribution.** In this work, we study the design space for compressed representations of conjunctive queries in the full continuum between optimal space and optimal runtime, when our goal is to optimize for a specific access pattern.

Our main contribution is a novel data structure that (i) *can compress the result for every CQ without projections according to the access pattern given by an adorned view*, and (ii) *can be tuned to tradeoff space for delay and answer time*. At the one extreme, the data structure achieves constant delay  $O(1)$ ; At the other extreme it uses linear space  $O(|D|)$ , but provides a worst delay guarantee. Our proposed data structure includes as a special case the data structure developed in [11] for the *fast set intersection* problem.

To construct our data structure, we need two technical ingredients. The first ingredient (Theorem 1) is a data structure that trades space with delay with respect to the worst-case size bound of the query result. As an example of the type of tradeoffs that can be achieved, for any CQ  $Q$  without projections and any access pattern, the data structure needs space  $\tilde{O}(|D|^{\rho^*} / \tau)$  to achieve delay  $\tilde{O}(\tau)$ , where  $\rho^*$  is the fractional edge cover number of  $Q$ , and  $|D|$  the size of the input database. In many cases and for specific access patterns, the data structure can substantially improve upon this tradeoff. To prove Theorem 1, we develop novel techniques on how to encode information about expensive sub-instances of the problem in a balanced way.

However, Theorem 1 by its own gives suboptimal tradeoffs, since it ignores structural properties of the query (for example, for constant delay it materializes the full result). Our second ingredient (Theorem 2) combines the data structure of Theorem 1 with a type of tree decomposition called *connex tree decomposition* [5]. This tree decomposition has the property of restricting the tree structure such that the bound variables in the adorned view always form a connected component at the top of the tree.

Finally, we discuss the complexity of choosing the optimal parameters for our two main theorems, when we want to optimize for delay given a space constraint, or vice versa.

**Organization.** We present our framework in Section 2, along with the preliminaries and basic notation. Our two main results (Theorems 1 and 2) are presented in Section 3. We then present the detailed construction of the data structure of Theorem 1 in Section 4, and of Theorem 2 in Section 5. Finally, in Section 6, we discuss some complexity results for optimizing the choice of parameters.

## 2 PROBLEM SETTING

In this section we present the basic notions and terminology, and then discuss in detail our framework.

### 2.1 Conjunctive Queries

In this paper we will focus on the class of *conjunctive queries* (CQs), which are expressed as

$$Q(\mathbf{y}) = R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \dots, R_n(\mathbf{x}_n)$$

Here, the symbols  $\mathbf{y}, \mathbf{x}_1, \dots, \mathbf{x}_n$  are vectors that contain *variables* or *constants*, the atom  $Q(\mathbf{y})$  is the *head* of the query, and the atoms  $R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \dots, R_n(\mathbf{x}_n)$  form the *body*. The variables in the head are a subset of the variables that appear in the body. A CQ is *full* if every variable in the body appears also in the head, and it is *boolean* if the head contains no variables, *i.e.* it is of the form  $Q()$ . We will typically use the symbols  $x, y, z, \dots$  to denote variables, and  $a, b, c, \dots$  to denote constants. If  $D$  is an input database, we denote by  $Q(D)$  the result of running  $Q$  over  $D$ .

**Natural Joins.** If a CQ is full, has no constants and no repeated variables in the same atom, then we say it is a *natural join query*. For instance, the triangle query  $\Delta(x, y, z) = R(x, y), S(y, z), T(z, x)$  is a natural join query. A natural join can be represented equivalently as a *hypergraph*  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ , where  $\mathcal{V}$  is the set of variables, and for each hyperedge  $F \in \mathcal{E}$  there exists a relation  $R_F$  with variables  $F$ . We will write the join as  $\bowtie_{F \in \mathcal{E}} R_F$ . The size of relation  $R_F$  is denoted by  $|R_F|$ . Given a set of variables  $I \subseteq \mathcal{V}$ , we define  $\mathcal{E}_I = \{F \in \mathcal{E} \mid F \cap I \neq \emptyset\}$ .

**Valuations.** A *valuation*  $v$  over a subset  $V$  of the variables is a total function that maps each variable  $x \in V$  to a value  $v(x) \in \text{dom}$ , where  $\text{dom}$  is a domain of constants. Given a valuation  $v$  of the variables  $(x_{i_1}, \dots, x_{i_\ell})$ , we denote  $R_F(v) = R_F \times \{v(x_{i_1}, \dots, x_{i_\ell})\}$ .

**Join Size Bounds.** Let  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$  be a hypergraph, and  $S \subseteq \mathcal{V}$ . A weight assignment  $\mathbf{u} = (u_F)_{F \in \mathcal{E}}$  is called a *fractional edge cover* of  $S$  if (i) for every  $F \in \mathcal{E}$ ,  $u_F \geq 0$  and (ii) for every  $x \in S$ ,  $\sum_{F: x \in F} u_F \geq 1$ . The *fractional edge cover number* of  $S$ , denoted by  $\rho_{\mathcal{H}}^*(S)$  is the minimum of  $\sum_{F \in \mathcal{E}} u_F$  over all fractional edge covers of  $S$ . We write  $\rho^*(\mathcal{H}) = \rho_{\mathcal{H}}^*(\mathcal{V})$ .

In a celebrated result, Atserias, Grohe and Marx [4] proved that for every fractional edge cover  $\mathbf{u}$  of  $\mathcal{V}$ , the size of a natural join is bounded using the following inequality, known as the *AGM inequality*:

$$|\bowtie_{F \in \mathcal{E}} R_F| \leq \prod_{F \in \mathcal{E}} |R_F|^{u_F} \quad (1)$$

The above bound is constructive [18, 20]: there exist worst-case algorithms that compute the join  $\bowtie_{F \in \mathcal{E}} R_F$  in time  $O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$  for every fractional edge cover  $\mathbf{u}$  of  $\mathcal{V}$ .

**Tree Decompositions.** Let  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$  be a hypergraph of a natural join query  $Q$ . A *tree decomposition* of  $\mathcal{H}$  is a tuple  $(\mathcal{T}, (\mathcal{B}_t)_{t \in \mathcal{T}})$  where  $\mathcal{T}$  is a tree, and every  $\mathcal{B}_t$  is a subset of  $\mathcal{V}$ , called the *bag* of  $t$ , such that

- (1) each edge in  $\mathcal{E}$  is contained in some bag  $\mathcal{B}_t$ ; and
- (2) for each  $x \in \mathcal{V}$ , the set of nodes  $\{t \mid x \in \mathcal{B}_t\}$  is connected in  $\mathcal{T}$ .

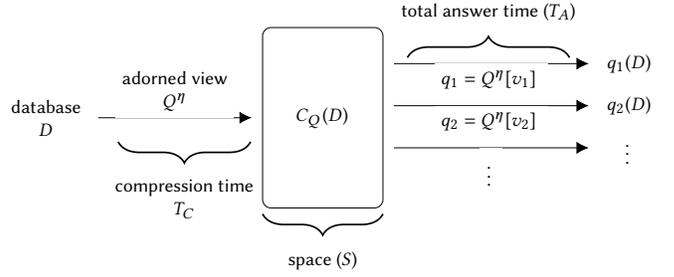
The *fractional hypertree width* of a tree decomposition is defined as  $\max_{t \in \mathcal{T}} \rho^*(\mathcal{B}_t)$ , where  $\rho^*(\mathcal{B}_t)$  is the minimum fractional edge cover of the vertices in  $\mathcal{B}_t$ . The fractional hypertree width of a query  $Q$ , denoted  $\text{fhw}(Q)$ , is the minimum fractional hypertree width among all tree decompositions of its hypergraph.

**Computational Model.** To measure the running time of our algorithms, we will use the uniform-cost RAM model [17], where data values as well as pointers to databases are of constant size. Throughout the paper, all complexity results are with respect to data complexity (unless explicitly mentioned), where the query is assumed fixed.

We use the notation  $\tilde{O}$  to hide a polylogarithmic factor  $\log^k |D|$  for some constant  $k$ , where  $D$  is the input database.

## 2.2 Adorned Views

In order to model access patterns over a view  $Q$  defined over the input database, we will use the concept of *adorned views* [28]. In an adorned view, each variable in the head of the view definition is associated with a binding type, which can be either *bound* (b) or *free* (f). A view  $Q(x_1, \dots, x_k)$  is then written as  $Q^\eta(x_1, \dots, x_k)$ ,



**Figure 1: Depiction of the compression framework along with the parameters.**

where  $\eta \in \{b, f\}^k$  is called the *access pattern*. We denote by  $\mathcal{V}_b$  (resp.  $\mathcal{V}_f$ ) the set of bound (resp. free) variables from  $\{x_1, \dots, x_k\}$ .

We can interpret an adorned view as a function that maps a valuation over the bound variables  $\mathcal{V}_b$  to a relation over the free variables  $\mathcal{V}_f$ . In other words, for each valuation  $v$  over  $\mathcal{V}_b$ , the adorned view returns the answer for the query  $Q^\eta[v] = \{V_f \mid Q(x_1, \dots, x_k) \wedge \forall x_i \in \mathcal{V}_b : x_i = v(x_i)\}$ , which we will also refer to as an *access request*.

**Example 2.**  $\Delta^{\text{bbf}}(x, y, z) = R(x, y), S(y, z), T(z, x)$  captures the following access pattern: given values  $x = a, y = b$ , list all the  $z$ -values that form a triangle with the edge  $R(a, b)$ . As another example,  $\Delta^{\text{fff}}(x, y, z) = R(x, y), S(y, z), T(z, x)$  simply captures the case where we want to perform a full enumeration of all the triangles in the result. Finally,  $\Delta^{\text{b}}(x) = R(x, y), S(y, z), T(z, x)$  expresses the access pattern where given a node with  $x = a$ , we want to know whether there exists a triangle that contains it or not.

An adorned view  $Q^\eta(x_1, \dots, x_k)$  is *boolean* if every head variable is bound, it is *non-parametric* if every head variable is free, and it is *full* if the CQ is full (i.e., every variable in the body also appears in the head). Of particular interest is the adorned view that is full and non-parametric, which we call the *full enumeration view*, and simply asks to output the whole result.

## 2.3 Problem Statement

Given an adorned view  $Q^\eta(x_1, \dots, x_k)$  and an input database  $D$ , our goal is to answer any access request  $Q^\eta[v]$  that conforms to the access pattern  $\eta$ . The view  $Q$  can be expressed through any type of query, but in this work we will focus on the case where  $Q$  is a conjunctive query.

There are two extremal approaches to handle this problem. The first solution is to answer any such query directly on the input database  $D$ , without materializing  $Q(D)$ . This solution is efficient in terms of space, but it can lead to inefficient query answering. For instance, consider the adorned view  $\Delta^{\text{bbf}}(x, y, z) = R(x, y), S(y, z), T(z, x)$ . Then, every time we are given new values  $x = a, y = b$ , we would have to compute all the nodes  $c$  that form a triangle with  $a, b$ , which can be very expensive.

The second solution is to materialize the view  $Q(D)$ , and then answer any incoming query over the materialized result. For example, we could choose to materialize all triangles, and then create an

appropriate index over the output result. The drawback of this approach is that it requires a lot of space, which may not be available.

We propose to study the solution space between these two extremal solutions, that is, instead of materializing all of  $Q(D)$ , we would like to store a *compressed representation*  $C_Q(D)$  of  $Q(D)$ . The compression function  $C_Q$  must guarantee that the compression is lossless, *i.e.*, there exists a decompression function  $D_Q$  such that for every database  $D$ , it holds that  $D_Q(C_Q(D)) = Q(D)$ . We compute the compressed representation  $C_Q(D)$  during a *preprocessing phase*, and then answer any access request in an *online phase*.

**Parameters.** Our goal is to construct a compression that is as space-efficient as possible, while it guarantees that we can efficiently answer any access query. In particular, we are interested in measuring the tradeoff between the following parameters, which are also depicted in Figure 1:

**Compression Time** ( $T_C$ ): the time to compute  $C_Q(D)$  during the preprocessing phase.

**Space** ( $S$ ): the size of  $C_Q(D)$ .

**Answer Time:** this parameter measures the time to enumerate a query result, where the query is of the form  $Q^\eta[v]$ . The enumeration algorithm must (i) enumerate the query result without any repetitions of tuples, and (ii) use only  $O(\log |D|)$  extra memory<sup>2</sup>. We will measure answer time in two different ways.

- (1) **delay** ( $\delta$ ): the maximum time to output any two consecutive tuples (and also the time to output the first tuple, and the time to notify that the enumeration has completed).
- (2) **total answer time** ( $T_A$ ): the total time to output the result.

In the case of a boolean adorned view, the delay and the total answer time coincide. In an ideal situation, both the compression time and the space are linear to the input size and any query can be answered with constant delay  $O(1)$ . As we will see later, this is achievable in certain cases, but in most cases we have to tradeoff space and preprocessing time for delay and total answer time.

## 2.4 Some Basic Results

We present here some basic results that set up a baseline for our framework. We will study the case where the given view definition  $Q$  is a conjunctive query.

Our first observation is that if we allow the compression time to be at least  $\Omega(|D|)$ , we can assume without loss of generality that the adorned view  $Q^\eta$  has no constants or repeated variables in a single atom. Indeed, we can first do a linear time computation to rewrite the adorned view  $Q^\eta$  to a new view where constants and repeated variables are removed, and then compute the compressed representation for this new view (with the same adornment).

**Example 3.** Consider  $Q^{\text{fb}}(x, z) = R(x, y, a), S(y, y, z)$ . We can first compute in linear time  $R'(x, y) = R(x, y, a)$  and  $S'(y, z) = S(y, y, z)$ , and then rewrite the adorned view as  $Q^{\text{fb}}(x, z) = R'(x, y), S'(y, z)$ .

Hence, whenever the adorned view is a full CQ, we can w.l.o.g. assume that it is a natural join query. We now state a simple result

<sup>2</sup>Memory requirement also depends on the memory required for executing the join algorithm. Note that worst case optimal join algorithms such as NPRR [18] can be executed using  $\log |D|$  memory assuming query size is constant and all relations are sorted and indexed.

for the case where the adorned view is full and every variable is bound.

**Proposition 1.** *Suppose that the adorned view is a natural join query with head  $Q^{\text{b}\cdots\text{b}}(x_1, \dots, x_k)$ . Then, in time  $T_C = O(|D|)$ , we can construct a data structure with space  $S = O(|D|)$ , such that we can answer any access request over  $D$  with constant delay  $\delta = O(1)$ .*

Next, consider the full enumeration view  $Q^{\text{f}\cdots\text{f}}(x_1, \dots, x_k)$ . A first observation is that if we store the materialized view, we can enumerate the result in constant delay. From the AGM bound, to achieve this we need space  $|D|^{\rho^*(\mathcal{H})}$ , where  $\mathcal{H}$  is the hypergraph of  $Q$ . However, it is possible to improve upon this naive solution using the concept of a factorized representation [23]. Let  $\text{fhw}(Q)$  denote the *fractional hypertree width* of  $Q$ . Then, the result from [23] can be translated in our terminology as follows.

**Proposition 2** ([23]). *Suppose that the adorned view is a natural join query with head  $Q^{\text{f}\cdots\text{f}}(x_1, \dots, x_k)$ . Then, in compression time  $T_C = \tilde{O}(|D|^{\text{fhw}(Q)})$ , we can construct a data structure with space  $S = O(|D|^{\text{fhw}(Q)})$ , such that we can answer any access request over  $D$  with constant delay  $\delta = O(1)$ .*

Since every acyclic query has  $\text{fhw}(Q) = 1$ , for acyclic CQs without projections both the compression time and space become linear,  $O(|D|)$ . In the next section, we will see how we can generalize the above result to an arbitrary adorned view that is full.

## 3 MAIN RESULTS AND APPLICATION

In this section we present our two main results, and show how they can be applied. The first result (Theorem 1) is a compression primitive that can be used with any full adorned view. The second result (Theorem 2) builds upon Theorem 1 and query decomposition techniques to obtain an improved tradeoff between space and delay.

### 3.1 First Main Result

Consider a full adorned view  $Q^\eta(x_1, \dots, x_k)$ , where  $Q$  is a natural join query expressed by the hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ . Recall that  $\mathcal{V}_b, \mathcal{V}_f$  are the bound and free variables respectively. Since the query is a natural join and there are no projections, we have  $\mathcal{V}_b \cup \mathcal{V}_f = \mathcal{V}$ . We will denote by  $\mu = |\mathcal{V}_f|$  the number of free variables. We also impose a lexicographic order on the enumeration order of the output tuples. Specifically, we equip the domain  $\text{dom}$  with a total order  $\leq$ , and then extend this to a total order for output tuples in  $\text{dom}^\mu$  using some order  $x_f^1, x_f^2, \dots, x_f^\mu$  of the free variables.<sup>3</sup>

**Example 4.** *As a running example, consider*

$$Q^{\text{ffffbb}}(x, y, z, w_1, w_2, w_3) = R_1(w_1, x, y), R_2(w_2, y, z), \\ R_3(w_3, x, z).$$

*We have  $\mathcal{V}_f = \{x, y, z\}$  and  $\mathcal{V}_b = \{w_1, w_2, w_3\}$ . To keep the exposition simple, assume that  $|R_1| = |R_2| = |R_3| = N$ .*

*If we materialize the result and create an index with composite key  $(w_1, w_2, w_3)$ , then in the worst case we need space  $S = O(N^3)$ , but we will be able to enumerate the output for every access request with constant delay. On the other hand, if we create three indexes, one for each  $R_i$  with key  $w_i$ , we can compute each access request with*

<sup>3</sup>There is no restriction imposed on the lexicographic ordering of the free variables.

worst-case running time and delay of  $O(N^{3/2})$ . Indeed, once we fix the bound variables to constants  $c_1, c_2, c_3$ , we need to compute the join  $R_1(c_1, x, y) \bowtie R_2(c_2, y, z) \bowtie R_3(c_3, x, z)$ , which needs time  $O(N^{3/2})$  using any worst-case optimal join algorithm.

For any fractional edge cover  $\mathbf{u}$  of  $\mathcal{V}$ , and  $S \subseteq \mathcal{V}$ , we define the slack of  $\mathbf{u}$  for  $S$  as:

$$\alpha(S) = \min_{x \in S} \left( \sum_{F: x \in F} u_F \right) \quad (2)$$

Intuitively, the slack is the maximum positive quantity such that  $(u_F/\alpha(S))_{F \in \mathcal{E}}$  is still a fractional edge cover of  $S$ . By construction, the slack is always at least one,  $\alpha(S) \geq 1$ . For our running example, suppose that we pick a fractional edge cover for  $\mathcal{V}$  with  $u_{R_1} = u_{R_3} = 1$ . Then, the slack of  $\mathbf{u}$  for  $\mathcal{V}_f$  is  $\alpha(\mathcal{V}_f) = 2$ .

**Theorem 1.** *Let  $Q^\eta$  be an adorned view over a natural join query with hypergraph  $(\mathcal{V}, \mathcal{E})$ . Let  $\mathbf{u}$  be any fractional edge cover of  $\mathcal{V}$ . Then, for any input database  $D$  and parameter  $\tau > 0$  we can construct a data structure with*

$$\begin{aligned} \text{compression time } T_C &= \tilde{O}(|D| + \prod_{F \in \mathcal{E}} |R_F|^{u_F}) \\ \text{space } S &= \tilde{O}(|D| + \prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^{\alpha(\mathcal{V}_f)}) \end{aligned}$$

such that for any access request  $q = Q^\eta[v]$ , we can enumerate its result  $q(D)$  in lexicographic order with

$$\text{delay } \delta = \tilde{O}(\tau)$$

$$\text{answer time } T_A = \tilde{O}(|q(D)| + \tau \cdot |q(D)|^{1/\alpha(\mathcal{V}_f)})$$

**Example 5.** *Let us apply Theorem 1 to our running example for  $\mathbf{u} = (1, 1, 1)$  and  $\tau = N^{1/2}$ . The slack for the free variables is  $\alpha(\mathcal{V}_f) = 2$ . The theorem tells us that we can construct in time  $\tilde{O}(N^3)$  a data structure with space  $\tilde{O}(N^2)$ , such that every access request  $q$  can be answered with delay  $\tilde{O}(N^{1/2})$  and answer time  $\tilde{O}(|q(D)| + \sqrt{N} \cdot |q(D)|)$ .*

We prove Theorem 1 in Section 4. We next show how to apply the theorem to obtain several results on space-efficient compressed representations for CQs.

**Applying Theorem 1.** We start with the observation that we can always apply Theorem 1 by choosing  $\mathbf{u}$  to be the fractional edge cover with optimal value  $\rho^*(\mathcal{H})$ . Since the slack is always  $\geq 1$ , we obtain the following result.

**Proposition 3.** *Let  $Q^\eta$  be an adorned view over a natural join query with hypergraph  $\mathcal{H}$ . Then, for any input database  $D$  and parameter  $\tau > 0$ , we can construct a data structure with*

$$\text{space } S = \tilde{O}(|D| + |D|^{\rho^*(\mathcal{H})} / \tau)$$

such that for any access request  $q$ , we can enumerate its result  $q(D)$  in lexicographic order with

$$\delta = \tilde{O}(\tau), \quad T_A = \tilde{O}(\tau \cdot |q(D)|)$$

Proposition 3 tells us that the data structure has a linear tradeoff between space and delay. Also, to achieve (almost) constant delay  $\delta = \tilde{O}(1)$ , the space requirement becomes  $\tilde{O}(|D|^{\rho^*})$ ; in other words, the data structure will essentially materialize the whole result. Our

second main result will allow us to exploit query decomposition to avoid this case.

**Example 6.** *Consider the following adorned view over the Loomis-Whitney join:*

$$LW_n^{\text{b}\cdots\text{bf}}(x_1, \dots, x_n) = S_1(x_2, \dots, x_n), S_2(x_1, x_3, \dots, x_n), \dots, S_n(x_1, \dots, x_{n-1})$$

The minimum fractional edge cover assigns weight  $1/(n-1)$  to each hyperedge and has  $\rho^* = n/(n-1)$ . Then, Proposition 3 tells us that for  $\tau > 0$ , we can construct a compressed representation with space  $S = \tilde{O}(|D| + |D|^{n/(n-1)}/\tau)$  and delay  $\delta = \tilde{O}(\tau)$ . Notice that if we aim for linear space, we can choose  $\tau = |D|^{1/(n-1)}$  and achieve a small delay of  $\tilde{O}(|D|^{1/(n-1)})$ .

Proposition 3 ignores the effect of the slack for the free variables. The next example shows that taking slack into account is critical in obtaining better tradeoffs.

**Example 7.** *Consider the adorned view over the star join*

$$S_n^{\text{b}\cdots\text{bf}}(x_1, \dots, x_n, z) = R_1(x_1, z), R_2(x_2, z), \dots, R_n(x_n, z)$$

The star join is acyclic, which means that the  $d$ -representation of the full result takes only linear space. This  $d$ -representation can be used for any adornment of  $S_n$  where  $z$  is a bound variable; hence, in all these cases we can guarantee  $O(1)$  delay using linear compression space. However, we cannot get any guarantees when  $z$  is free, as is in the adornment used above.

If we apply Proposition 3, we get space  $\tilde{O}(|D| + |D|^n/\tau)$  with delay  $\tilde{O}(\tau)$ . However, we can improve upon this by noticing that for the fractional edge cover where  $u_1 = \dots = u_n = 1$ , the slack is  $\alpha(\mathcal{V}_f) = n$ . Hence, Theorem 1 tells us that with space  $\tilde{O}(|D|^n/\tau^n)$  we get delay  $\tilde{O}(\tau)$  and answer time  $\tilde{O}(|Q(D)| + \tau \cdot |Q(D)|^{1/n})$ .

We should note here that our data structure strictly generalizes the data structure proposed in [11] for the problem of *fast set intersection*. Given a family of sets  $S_1, \dots, S_n$ , the goal in this problem is to construct a space-efficient data structure, such that given any two sets  $S_i, S_j$  we can compute their intersection  $S_i \cap S_j$  as fast as possible. It is easy to see that this problem is captured by the adorned view  $S_2^{\text{b}\text{bf}}(x_1, x_2, z) = R(x_1, z), R(x_2, z)$ , where  $R$  is a relation that describes set membership ( $R(S_i, a)$  means that  $a \in S_i$ ).

### 3.2 Second Main Result

The direct application of Theorem 1 can lead to suboptimal tradeoffs between space and time/delay, since it ignores the structural properties of the query. In this section, we show how to overcome this problem by combining Theorem 1 with tree decompositions.

We first need to introduce a variant of a tree decomposition of a hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ , defined with respect to a given subset  $C \subseteq \mathcal{V}$ .

**DEFINITION 1 (CONNEX TREE DECOMPOSITION [5]).** *Let  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$  be a hypergraph, and  $C \subseteq \mathcal{V}$ . A  $C$ -connex tree decomposition of  $\mathcal{H}$  is a tuple  $(\mathcal{T}, A)$ , where:*

- (1)  $\mathcal{T} = (\mathcal{T}, (\mathcal{B}_t)_{t \in V(\mathcal{T})})$  is a tree decomposition of  $\mathcal{H}$ ; and
- (2)  $A$  is a connected subset of  $V(\mathcal{T})$  such that  $\bigcup_{t \in A} \mathcal{B}_t = C$ .

In a  $C$ -connex tree decomposition, the existence of the set  $A$  forces the set of nodes that contain some variable from  $C$  to be connected in the tree.

**Example 8.** Consider the hypergraph  $\mathcal{H}$  in Figure 2. The decomposition depicted on the left is a  $C$ -connex tree decomposition for  $C = \emptyset$ . The  $C$ -connex tree decomposition on the right is for  $C = \{v_1, v_5, v_6\}$ . In both cases,  $A$  consists of a single bag (colored grey) which contains exactly the variables in  $C$ .

In [5],  $C$ -connex decompositions were used to obtain compressed representations of CQs with projections (where  $C$  is the set of the head variables). In our setting, we will choose  $C$  to be the set of bound variables in the adorned view, i.e.,  $C = \mathcal{V}_b$ . Additionally, we will use a novel notion of width, which we introduce next.

Given a  $\mathcal{V}_b$ -connex tree decomposition  $(\mathcal{T}, A)$ , we orient the tree  $\mathcal{T}$  from some node in  $A$ . For any node  $t \in V(\mathcal{T}) \setminus A$ , we denote by  $\text{anc}(t)$  the union of all the bags for the nodes that are the ancestors of  $t$ . Define  $\mathcal{V}_b^t = \mathcal{B}_t \cap \text{anc}(t)$  and  $\mathcal{V}_f^t = \mathcal{B}_t \setminus \mathcal{V}_b^t$ . Intuitively,  $\mathcal{V}_b^t$  (resp.  $\mathcal{V}_f^t$ ) are the bound (resp. free) variables for the bag  $t$  as we traverse the tree top-down. Figure 2 depicts each bag  $\mathcal{B}_t$  as  $\mathcal{V}_f^t \mid \mathcal{V}_b^t$ .

Given a  $\mathcal{V}_b$ -connex tree decomposition, a *delay assignment* is a function  $\delta : V(\mathcal{T}) \rightarrow [0, \infty)$  that maps each bag to a non-negative number, such that  $\delta(t) = 0$  for  $t \in A$ . Intuitively, this assignment means that we want to achieve a delay of  $|D|^{\delta(t)}$  for traversing this particular bag. For a bag  $t$ , define

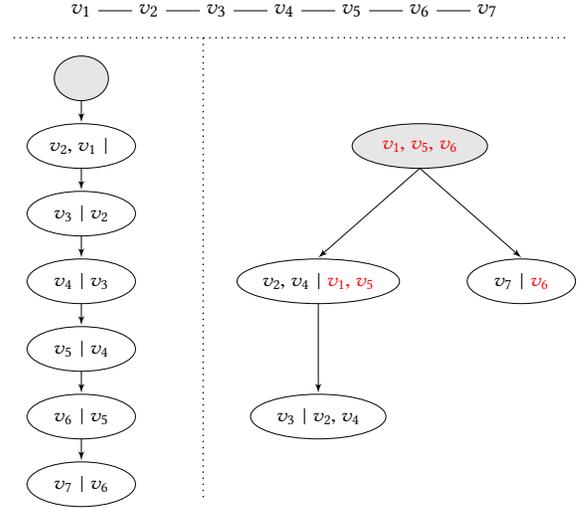
$$\rho_t^+ = \min_{\mathbf{u}} \left( \sum_F u_F - \delta(t) \cdot \alpha(\mathcal{V}_f^t) \right) \quad (3)$$

where  $\mathbf{u}$  is a fractional edge cover of the bag  $\mathcal{B}_t$ . The  $\mathcal{V}_b$ -connex fractional hypertree  $\delta$ -width of  $(\mathcal{T}, A)$  is defined as  $\max_{t \in V(\mathcal{T}) \setminus A} \rho_t^+$ . It is critical that we ignore the bags in the set  $A$  in the max computation. We also define  $u_t^+ = \sum_F u_F^+$  where  $\mathbf{u}^+$  is the fractional edge cover of bag  $\mathcal{B}_t$  that minimizes  $\rho_t^+$ .

When  $\delta(t) = 0$  for every bag  $t$ , the  $\delta$ -width of any  $\mathcal{V}_b$ -connex tree decomposition becomes  $\max_{t \in V(\mathcal{T}) \setminus A} \rho^*(\mathcal{B}_t)$ , where  $\rho^*(\mathcal{B}_t)$  is the fractional edge cover number of  $\mathcal{B}_t$ . Define  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$  as the smallest such quantity among all  $\mathcal{V}_b$ -connex tree decompositions of  $\mathcal{H}$ . When  $\mathcal{V}_b = \emptyset$ , then  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) = \text{fhw}(\mathcal{H})$ , thus recovering the notion of fractional hypertree width. Appendix E shows the relationship between  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$  and other hypergraph related parameters.

Finally, we define the  $\delta$ -height of a  $\mathcal{V}_b$ -connex tree decomposition to be the maximum weight root-to-leaf path, where the weight of a path  $P$  is defined as  $\sum_{t \in P} \delta(t)$ .

**Example 9.** Consider the decomposition on the right in Figure 2, and a delay assignment  $\delta$  that assigns  $1/3$  to node  $t_1$  with  $\mathcal{B}_{t_1} = \{v_2, v_4, v_1, v_5\}$ ,  $1/6$  to the bag  $t_2$  with  $\mathcal{B}_{t_2} = \{v_2, v_3, v_4\}$ , and  $0$  to the node  $t_3$  with  $\mathcal{B}_{t_3} = \{v_6, v_7\}$ . The  $\delta$ -height of the tree is  $h = \max\{1/3 + 1/6, 0\} = 1/2$ . To compute the fractional hypertree  $\delta$ -width, observe that we can cover the bag  $\{v_2, v_4, v_1, v_5\}$  by assigning weight of  $1$  to the edges  $\{v_1, v_2\}, \{v_4, v_5\}$ , in which case  $\rho_{t_1}^+ = (1 + 1) - 1/3 \cdot 1 = 5/3$ . We also have  $\rho_{t_2}^+ = (1 + 1) - 1/6 \cdot 2 = 5/3$ , and  $\rho_{t_3}^+ = 1$ . Hence, the fractional hypertree  $\delta$ -width is  $5/3$ . Also, observe that  $u_{t_1}^+ = u_{t_2}^+ = 2$  and  $u_{t_3}^+ = 1$ .



**Figure 2: The hypergraph  $\mathcal{H}$  for a path query of length 6, along with two  $C$ -connex tree decompositions. The decomposition on the left has  $C = \emptyset$ , and the decomposition on the right  $C = \{v_1, v_5, v_6\}$ . The variables in  $C$  are colored red, and the grey nodes are the ones in the set  $A$ .**

**Theorem 2.** Let  $q = Q^\eta$  be an adorned view over a natural join query with hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ . Suppose that  $\mathcal{H}$  admits a  $\mathcal{V}_b$ -connex tree decomposition. Fix any delay assignment  $\delta$ , and let  $f$  be the  $\mathcal{V}_b$ -connex fractional hypertree  $\delta$ -width,  $h$  the  $\delta$ -height of the decomposition, and  $u^* = \max_{t \in V(\mathcal{T}) \setminus A} u_t^+$ .

Then, for any input database  $D$ , we can construct a data structure in compression time  $T_C = \tilde{O}(|D| + |D|^{u^* + \max_t \delta(t)})$  with space  $S = \tilde{O}(|D| + |D|^f)$ , such that we can answer any access request with delay  $\tilde{O}(|D|^h)$ .

If we write the delay in the above result as  $\tilde{O}(\prod_{t \in P} |D|^{\delta(t)})$ , where  $P$  is the maximum-weight path, Theorem 2 tells us that the delay is essentially multiplicative in the same branch of the tree, but additive across branches. Unlike Theorem 1, the lexicographic ordering of the result  $q(D)$  for Theorem 2 now depends on the tree decomposition.

For our running example, Theorem 2 implies a data structure with space  $\tilde{O}(|D| + |D|^{5/3})$  and delay  $\tilde{O}(|D|^{1/2})$ . This data structure can be computed in time  $\tilde{O}(|D| + |D|^{7/3})$ . Notice that this is much smaller than the  $O(|D|^4)$  time required to compute the worst case output. We prove the theorem in detail in Section 5, and we discuss the complexity of choosing the optimal parameters in Section 6. Next, we delve deeper into Theorem 2 and how to apply it.

**Applying Theorem 2.** We first give an example where Theorem 2 can substantially improve upon the space/delay tradeoff of Theorem 1.

**Example 10.** Consider the following adorned view:

$$P_n^{\text{bf} \cdots \text{fb}}(x_1, \dots, x_{n+1}) = R_1(x_1, x_2), R_2(x_2, x_3), \dots, R_n(x_n, x_{n+1}).$$

A direct application of Theorem 1 results in a tradeoff of space  $\tilde{O}(|D| + |D|^{\lceil n/2 \rceil / \tau})$  with delay  $\tilde{O}(\tau)$ . On the other hand, we can construct

a connex tree decomposition where  $A$  has a single bag  $\{x_1, x_{n+1}\}$ , which is connected to  $\{x_1, x_2, x_n, x_{n+1}\}$ , which is in turn connected to  $\{x_2, x_3, x_{n-1}, x_n\}$ , and so on. Consider the delay assignment that assigns to each bag  $\delta(t) = \log_{|D|} \tau$ . The  $\delta$ -width of this decomposition is  $2 - \log_{|D|} \tau$ , while the  $\delta$ -height is  $\lfloor n/2 \rfloor \cdot \log_{|D|} \tau$ . Hence, Theorem 2 results in a tradeoff of space  $\tilde{O}(|D| + |D|^2/\tau)$  with delay  $\tilde{O}(\tau^{\lfloor n/2 \rfloor})$ .

Suppose now that our goal is to achieve constant delay. From Theorem 2, in order to do this we have to choose the delay assignment to be 0 everywhere. In this case, we have the following result (which slightly strengthens Theorem 2 in this special case by dropping the polylogarithmic dependence).

**Proposition 4.** *Let  $Q^\eta$  be a full adorned view over a hypergraph  $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ . Then, for any input database  $D$ , we can construct a data structure in compression time and space  $S = O(|D|^{\text{fhw}(\mathcal{H} | \mathcal{V}_b)})$ , such that we can answer any access request with delay  $O(1)$ .*

Observe that when all variables are free, then  $\mathcal{V}_b = \emptyset$ , in which case  $\text{fhw}(\mathcal{H} | \mathcal{V}_b) = \text{fhw}(\mathcal{H})$ , thus recovering the compression result of a  $d$ -representation. Moreover, since the delay assignment is 0 for all bags, the compression time  $T_C = \tilde{O}(|D| + |D|^{\text{fhw}(\mathcal{H} | \mathcal{V}_b)})$ .

**Beyond full adorned views.** Our work provides compression strategies for queries that do not admit out-of-the-box factorization (such as Loomis-Whitney joins), and can also recover the result of compressed  $d$ -representations as a special case when all variables are free (Proposition 4). On the other hand, factorized databases support a much richer set of queries such as projections, aggregations [6, 7] and analytical tasks such as learning regression models [22, 25]. One possible approach to handling projections in our setting is to force a variable ordering in the  $\mathcal{V}_b$ -connex decomposition: more precisely, we can force projection variables to appear first in any root to leaf path. This idea of fixing variable ordering would be similar to how order-by clauses are handled in  $d$ -tree query plans [6]. Remarkably, the  $\mathcal{V}_b$ -connex decomposition in our setting also corresponds to the tree decompositions used to compute aggregations and orderings with group-by attributes as  $\mathcal{V}_b$  [22]. This points to a deeper connection between our compressed representation and  $d$ -tree representations used to compute group-by aggregates. We defer the study of these connections and extension of our framework to incorporate more expressive queries to future work.

### 3.3 A Remark on Optimality

So far we have not discussed the optimality of our results. We remark why proving tight lower bounds might be a hard problem.

The problem of  $k$ -SetDisjointness is defined as follows. Given a family of sets  $S_1, \dots, S_m$  of total size  $N$ , we want to ask queries of the following form: given as input a subset  $I \subseteq \{1, \dots, m\}$  of size  $k$ , is the intersection  $\bigcap_{i \in I} S_i$  empty? The goal is to construct a space-efficient data structure such that we can answer as fast as possible. Note that  $k$ -SetDisjointness corresponds to the following adorned view:  $Q^{b \cdots b}(x_1, \dots, x_k) = R(x_1, z), R(x_2, z), \dots, R(x_k, z)$ , where  $R$  has size  $N$ . One can see that we can use the data structure for the corresponding full view with head  $Q^{b \cdots bf}(x_1, \dots, x_k, z)$  (see Example 7) to answer  $k$ -SetDisjointness queries in time  $\tilde{O}(\tau)$ , using space  $\tilde{O}(N^k/\tau^k)$ .

In a recent work, Goldstein et al. [15] conjecture the following lower bound:

**Conjecture 1.** (due to [15]) *Consider a data structure that preprocesses a family of sets  $S_1, \dots, S_m$  of total size  $N$ . If the data structure can answer  $k$ -SetDisjointness queries in time (or delay)<sup>4</sup>  $T$ , then it must use  $S = \tilde{\Omega}(N^k/T^k)$  space.*

The above conjecture is a generalization of a conjecture from [13] for the case  $k = 2$ , which in turn generalizes a folklore conjecture of Patrascu and Roditty [24], which was stated only for the case where  $\tau = 1$  and  $k = 2$ . Applied in our setting, Conjecture 1 implies that for the adorned view  $Q^{b \cdots b}(x_1, \dots, x_k) = R_1(x_1, z), R_2(x_2, z), \dots, R_k(x_k, z)$ , the tradeoff between space and delay (or answer time) is essentially optimal when all relations have equal size. Unfortunately, proving even the weaker conjecture of [24] is considered a hard open problem.

## 4 A COMPRESSION PRIMITIVE

In this section, we describe the detailed construction of our data structure for Theorem 1. Due to space constraints, we defer the proofs to the full version of the paper [14].

### 4.1 Intervals and Boxes

Before we present the compression procedure, we first introduce two important concepts in our construction,  $f$ -intervals and  $f$ -boxes, both of which describe subspaces of the space of all possible tuples in the output.

**Intervals.** The *active domain*  $\mathbf{D}[x]$  of each variable  $x$  is equipped with a total order  $\leq$  induced from the order of  $\mathbf{dom}$ . We will use  $\perp, \top$  to denote the smallest and largest element of the active domain respectively (these will always exist, since we assume finite databases). An *interval* for variable  $x$  is any subset of  $\mathbf{D}[x]$  of the form  $\{u \in \mathbf{D}[x] \mid a \leq u \leq b\}$ , where  $a, b \in \mathbf{D}[x]$ , denoted by  $[a, b]$ . We adopt the standard notation for closed and open intervals and write  $[a, b) = \{u \in \mathbf{D}[x] \mid a \leq u < b\}$ , and  $(a, b] = \{u \in \mathbf{D}[x] \mid a < u \leq b\}$ . The interval  $[a, a]$  is called the *unit interval* and represents a single value. We will often write  $a$  for the interval  $[a, a]$ , and the symbol  $\square$  for the interval  $\mathbf{D}[x]$ .

By lifting the order from a single domain to the lexicographic order of tuples in  $\mathbf{D}_f = \mathbf{D}[x_1^1] \times \dots \times \mathbf{D}[x_1^\mu]$ , we can also define intervals over  $\mathbf{D}_f$ , which we call  *$f$ -intervals*. For instance, if  $\mathbf{a} = \langle a_1, \dots, a_\mu \rangle$  and  $\mathbf{b} = \langle b_1, \dots, b_\mu \rangle$ , the  $f$ -interval  $\mathbf{I} = [\mathbf{a}, \mathbf{b})$  represents all valuations  $v_f$  over  $\mathcal{V}_f$  that are lexicographically at least  $\mathbf{a}$ , but strictly smaller than  $\mathbf{b}$ .

**Boxes.** It will be useful to consider another type of subsets of  $\mathbf{D}_f$ , which we call  $f$ -boxes.

**DEFINITION 2 (f-BOX).** *An  $f$ -box is defined as a tuple of intervals  $\mathbf{B} = \langle I_1, \dots, I_\mu \rangle$ , where  $I_i$  is an interval of  $\mathbf{D}[x_1^i]$ . The  $f$ -box represents all valuations  $v_f$  over  $\mathcal{V}_f$ , such that  $v_f(x_1^i) \in I_i$  for every  $i = 1, \dots, \mu$ .*

We say that an  $f$ -box is *canonical* if whenever  $I_i \neq \square$ , then every  $I_j$  with  $j < i$  is a unit interval. A canonical  $f$ -box is always of the form  $\langle a_1, \dots, a_{i-1}, I_i, \square, \dots \rangle$ . For ease of notation, we will omit the  $\square$  intervals in the end of a canonical  $f$ -box, and simply write  $\langle a_1, \dots, a_{i-1}, I_i \rangle$ .

An  $f$ -box  $\mathbf{B}$  satisfies the following important property:

<sup>4</sup>For boolean queries, answer time and delay coincide.

**Proposition 5.**  $(\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{B} = \bowtie_{F \in \mathcal{E}} (R_F \times \mathbf{B})$ .

In other words, if we want to compute the restriction of an output to tuples in  $\mathbf{B}$ , it suffices to first restrict each relation to  $\mathbf{B}$  and then perform the join. We denote this restriction of the relation as  $R_F(\mathbf{B}) = R_F \times \mathbf{B}$ .

Unfortunately, Proposition 5 does not extend to  $f$ -intervals. Example 15 in Appendix A shows that it is generally not possible to first restrict each relation to  $R_F \times \mathbf{I}$  and then perform the join. As we will see next, we can partition each  $f$ -interval to a set of  $f$ -boxes of constant size.

**Box Decomposition.** It will be useful to represent an  $f$ -interval  $\mathbf{I} = (\mathbf{a}, \mathbf{b})$  as a union of canonical  $f$ -boxes. Let  $j$  be the first position such that  $a_j \neq b_j$ . Then, we define the *box decomposition* of  $\mathbf{I}$ , denoted  $\mathcal{B}(\mathbf{I})$ , as the following set of canonical  $f$ -boxes:

$$\begin{aligned} \mathbf{B}_\mu^\ell &= \langle a_1, \dots, a_{\mu-1}, (a_\mu, \top] \rangle \\ &\dots \\ \mathbf{B}_{j+1}^\ell &= \langle a_1, \dots, a_j, (a_{j+1}, \top] \rangle \\ \mathbf{B}_j &= \langle a_1, \dots, a_{j-1}, (a_j, b_j] \rangle \\ \mathbf{B}_{j+1}^r &= \langle b_1, \dots, b_j, [\perp, b_{j+1}) \rangle \\ &\dots \\ \mathbf{B}_\mu^r &= \langle b_1, \dots, b_{\mu-1}, [\perp, b_\mu) \rangle \end{aligned}$$

Intuitively, a box decomposition divides an interval into a set of disjoint, lexicographically ordered intervals. We give next an example of an  $f$ -interval and its decomposition into canonical  $f$ -boxes.

**Example 11.** For our running example (Example 4), let the active domain be  $\mathbf{D}[w_i] = \{1, 2, \dots, 1000\}$  for  $i = 1, 2, 3$ . Consider an open  $f$ -interval  $\mathbf{I} = (\langle 10, 50, 100 \rangle, \langle 20, 10, 50 \rangle)$ . The box decomposition of  $\mathbf{I}$  consists of the following 5 canonical  $f$ -boxes:

$$\begin{aligned} \mathbf{B}_3^\ell &= \langle 10, 50, (100, \top] \rangle, & \mathbf{B}_2^\ell &= \langle 10, (50, \top] \rangle \\ \mathbf{B}_1 &= \langle (10, 20) \rangle \\ \mathbf{B}_2^r &= \langle 20, [\perp, 10) \rangle & \mathbf{B}_3^r &= \langle 20, 10, [\perp, 50) \rangle \end{aligned}$$

For another  $f$ -interval  $\mathbf{I}' = [\langle 10, 50, 100 \rangle, \langle 10, 50, 200 \rangle)$ , where the first two positions coincide, the box decomposition consists of one  $f$ -box:  $\mathbf{B}_3 = \langle 10, 50, [100, 200) \rangle$ .

The following lemma summarizes several important properties of the box decomposition:

**Lemma 1.** Let  $\mathbf{I}$  be an  $f$ -interval and  $\mathcal{B}(\mathbf{I})$  be its box decomposition. Then:

- (1) The  $f$ -boxes in  $\mathcal{B}(\mathbf{I})$  form an order,  $\mathbf{B}_\mu^\ell \leq \dots \leq \mathbf{B}_{j+1}^\ell \leq \mathbf{B}_j \leq \mathbf{B}_{j+1}^r \leq \dots \leq \mathbf{B}_\mu^r$ , such that two tuples from different  $f$ -boxes are ordered according to the order of their  $f$ -boxes.
- (2) The non-empty  $f$ -boxes of  $\mathcal{B}(\mathbf{I})$  form a partition of  $\mathbf{I}$ .
- (3)  $|\mathcal{B}(\mathbf{I})| \leq 2\mu - 1$ , where  $\mu = |\mathcal{V}_f|$ .

The above lemma implies the following corollary:

**Corollary 1.** Let  $\mathbf{I}$  be an  $f$ -interval and  $\mathcal{B}(\mathbf{I})$  be its box decomposition. Then:

$$\bigcup_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \bowtie_{F \in \mathcal{E}} (R_F \times \mathbf{B}) = (\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{I}$$

## 4.2 Two Key Ingredients

We describe the intuition behind the compression representation. Our data structure is parametrized by an integer  $\tau > 0$ , which can be viewed as a threshold parameter that works as a knob. We seek to compute the result  $(\bowtie_{F \in \mathcal{E}} R_F(v_b)) \times \mathbf{I}$ , where  $\mathbf{I}$  is initially the  $f$ -interval that represents all possible valuations over  $\mathcal{V}_f$ . We can upper bound the running time for this instance using the AGM bound. If the bound is less than  $\tau$ , we can compute the answer in time and delay at most  $\tau$ .

Otherwise, we do two things: (i) we store a bit (1 if the answer is nonempty, and 0 if it is empty), and (ii) we split the  $f$ -interval into two smaller  $f$ -intervals. Then, we recursively apply the same idea for each of the two  $f$ -intervals. Since we need to store one bit for every valuation that exceeds the given threshold for a given  $f$ -interval, we need to bound the number of such valuations: this bound will be our first ingredient. Second, we split each  $f$ -interval in the same way for every valuation over bound variables; we do it such that we can balance the information we need to store for each smaller  $f$ -interval. The method to split the  $f$ -intervals in a balanced way is our second key ingredient.

**Bounding the Heavy Valuations.** Given a valuation  $v_b$  for the bound variables, suppose we are asked to compute the result restricted in some  $f$ -interval  $\mathbf{I}$ , in other words  $(\bowtie_{F \in \mathcal{E}} R_F(v_b)) \times \mathbf{I}$ . Let  $R_F(v, \mathbf{B}) = R_F(v) \times \mathbf{B} = (R_F \times v) \times \mathbf{B}$ .

Let us denote the slack  $\alpha = \alpha(\mathcal{V}_f)$ , and also define  $\hat{\mathbf{u}} = (\hat{u}_F)_{F \in \mathcal{E}}$ , where  $\hat{u}_F = u_F / \alpha$ . For an  $f$ -box  $\mathbf{B}$  and valuation  $v$  over any variables, we define:

$$T(\mathbf{B}) = \prod_{F \in \mathcal{E}} |R_F(\mathbf{B})|^{\hat{u}_F}, \quad T(v, \mathbf{B}) = \prod_{F \in \mathcal{E}} |R_F(v, \mathbf{B})|^{\hat{u}_F}$$

We overload  $T$  to apply to an  $f$ -interval  $\mathbf{I}$  and valuation  $v$  over any variables as follows:

$$T(\mathbf{I}) = \sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} T(\mathbf{B}), \quad T(v, \mathbf{I}) = \sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} T(v, \mathbf{B})$$

**Proposition 6.** The output  $(\bowtie_{F \in \mathcal{E}} R_F(v_b)) \times \mathbf{I}$  can be computed in time  $O(T(v_b, \mathbf{I}))$ .

We will use the above bound on the running time as a threshold of when it means that a particular interval is expensive to compute.

**DEFINITION 3.** A pair  $(v_b, \mathbf{I})$  is  $\tau$ -heavy for a fractional edge cover  $\mathbf{u}$  if  $T(v_b, \mathbf{I}) > \tau$ .

Observe that if a pair is not  $\tau$ -heavy, this means that we can compute the corresponding subinstance over  $\mathbf{I}$  in time at most  $O(\tau)$ . The following proposition provides an upper bound for the number of such  $\tau$ -heavy pairs.

**Proposition 7.** Given an  $f$ -interval  $\mathbf{I}$  and integer  $\tau$ , let  $\mathcal{H}(\mathbf{I}, \tau)$  be the valuations  $v_b$  such that the pair  $(v_b, \mathbf{I})$  is  $\tau$ -heavy for  $\mathbf{u}$ . Then,

$$|\mathcal{H}(\mathbf{I}, \tau)| \leq \left( \frac{T(\mathbf{I})}{\tau} \right)^\alpha$$

**Example 12.** Consider the following instance for our running example.

| w <sub>1</sub> | x | y |
|----------------|---|---|
| 1              | 1 | 1 |
| 1              | 1 | 2 |
| 1              | 2 | 1 |
| 2              | 1 | 1 |
| 3              | 1 | 1 |

| w <sub>2</sub> | y | z |
|----------------|---|---|
| 1              | 1 | 2 |
| 1              | 2 | 1 |
| 1              | 2 | 2 |
| 2              | 1 | 1 |
| 2              | 1 | 2 |

| w <sub>3</sub> | x | z |
|----------------|---|---|
| 1              | 1 | 1 |
| 1              | 1 | 2 |
| 1              | 2 | 1 |
| 2              | 1 | 1 |
| 2              | 1 | 2 |

R<sub>1</sub>
R<sub>2</sub>
R<sub>3</sub>

We will use  $\mathbf{u} = (1, 1, 1)$  as the fractional edge cover for  $\mathcal{V}$ . Recall that the slack is  $\alpha = 2$ , and thus  $\hat{\mathbf{u}} = (1/2, 1/2, 1/2)$ . Observe that  $\mathbf{D}[x] = \mathbf{D}[y] = \mathbf{D}[z] = \{1, 2\}$ ,  $\mathbf{D}[w_1] = \{1, 2, 3\}$ ,  $\mathbf{D}[w_2] = \{1, 2\}$ ,  $\mathbf{D}[w_3] = \{1, 2, 3\}$ . Consider the root interval  $\mathbf{I}(r) = [\langle 1, 1, 1 \rangle, \langle 2, 2, 2 \rangle]$ . The box decomposition  $\mathcal{B}(\mathbf{I}(r))$  is:

$$\mathbf{B}_3^\ell = \langle 1, 1, [1, 2] \rangle, \quad \mathbf{B}_2^\ell = \langle 1, (1, 2) \rangle$$

$$\mathbf{B}_2^r = \langle 2, [1, 2] \rangle, \quad \mathbf{B}_3^r = \langle 2, 2, [1, 2] \rangle$$

We can then compute  $T(\mathbf{I}(r)) = \sqrt{3|3|4|} + \sqrt{1|2|4|} + \sqrt{1|3|1|} + 0 \approx 10.56$ . Consider  $v_b(w_1, w_2, w_3) = (1, 1, 1)$ . One can compute  $T(v_b, \mathbf{I}(r)) = \sqrt{2} + 2 + 1 = 4.414$ . If we pick  $\tau = 4$ , then  $(v_b, \mathbf{I}(r))$  is  $\tau$ -heavy.

**Splitting an Interval.** We next discuss how we perform a balanced splitting of an  $f$ -interval  $\mathbf{I}$ . The first lemma tells us how we can arbitrarily split a canonical  $f$ -box.

**Lemma 2.** Consider the canonical  $f$ -box

$$\mathbf{B} = \langle a_1, \dots, a_{i-1}, [\beta_L, \beta_U] \rangle.$$

Then, for any  $t \geq 0$ , there exists  $\beta \in \mathbf{D}[x_i^t]$  such that

- (1)  $T(\langle a_1, \dots, a_{i-1}, [\beta_L, \beta] \rangle) \leq t$
- (2)  $T(\langle a_1, \dots, a_{i-1}, [\beta, \beta_U] \rangle) \leq \max\{0, T(\mathbf{B}) - t\}$ .

Moreover, we can compute  $\beta$  in time  $\tilde{O}(1)$ .

We now present Algorithm 1, an algorithm that allows for balanced splitting of an  $f$ -interval  $\mathbf{I}$ .

---

**Algorithm 1:** Splitting an  $f$ -interval  $\mathbf{I}$

---

```

1  $\mathcal{B}(\mathbf{I}) = \{\mathbf{B}_1, \dots, \mathbf{B}_k\}$  in lexicographic order
2  $T \leftarrow \sum_{i=1}^k T(\mathbf{B}_i)$ 
3  $s \leftarrow \arg \min_j \{ \sum_{i=1}^j T(\mathbf{B}_i) > T/2 \}$ 
  /* let  $\mathbf{B}_s = \langle c_1, \dots, c_{k-1}, I_k, \dots, I_\mu \rangle$  */
4  $\gamma_{k-1} \leftarrow \sum_{i=1}^{s-1} T(\mathbf{B}_i), \quad \Delta_{k-1} \leftarrow T(\mathbf{B}_s)$ 
5 for  $j=k$  to  $\mu$  do
6   find min  $c_j$  s.t.  $T(\langle c_1, \dots, c_{j-1}, I_j \cap [\perp, c_j] \rangle) \geq$ 
   min $\{\Delta_{j-1}, T/2 - \gamma_{j-1}\}$ 
7    $\Delta_j \leftarrow T(\langle c_1, \dots, c_j \rangle)$ 
8    $\gamma_j \leftarrow \gamma_{j-1} + T(\langle c_1, \dots, c_{j-1}, I_j \cap [\perp, c_j] \rangle)$ 
9 end
10 return  $(c_1, \dots, c_\mu)$ 

```

---

**Proposition 8.** Let  $\mathbf{I} = [a, b]$  be an  $f$ -interval. Then, Algorithm 1 returns  $\mathbf{c} \in \mathbf{D}_f$  that splits  $\mathbf{I}$  into  $\mathbf{I}^< = [a, \mathbf{c}]$  and  $\mathbf{I}^> = (\mathbf{c}, b]$  such that  $T(\mathbf{I}^<) \leq T(\mathbf{I})/2$  and  $T(\mathbf{I}^>) \leq T(\mathbf{I})/2$ . Moreover, it terminates in time  $\tilde{O}(1)$ .

### 4.3 The Basic Structure

We now have all the necessary pieces to describe how we construct the compressed representation. Recall that our data structure is parametrized by a threshold parameter  $\tau$ , and a fractional edge cover  $\mathbf{u} = (u_F)_{F \in \mathcal{E}}$  of the variables in  $\mathcal{V}$ . The construction consists of two steps.

**1) The Delay-Balanced Tree.** In the first step, we construct an annotated binary tree  $\mathcal{T}$ . Each node  $w \in V(\mathcal{T})$  is annotated with an  $f$ -interval  $\mathbf{I}(w)$  and a value  $\beta(w) \in \mathbf{D}_f$ , which is chosen according to Algorithm 1. The tree is constructed recursively.

Initially, we create a root  $r$  with interval  $\mathbf{I}(r) = \mathbf{D}_f$ . Let  $w$  be a node at level  $\ell$  with interval  $\mathbf{I}(w) = [a, c]$ , and define the threshold at level  $\ell$  to be  $\tau_\ell = \tau/2^{\ell(1-1/\alpha)}$ . In the case where  $T(\mathbf{I}(w)) < \tau_\ell$ ,  $w$  is a leaf of the tree. Otherwise, using  $\beta(w)$  as splitting point, we construct two sub-intervals of  $\mathbf{I}$ :

$$\mathbf{I}^< = [a, \beta(w)] \text{ and } \mathbf{I}^> = (\beta(w), c].$$

If  $\mathbf{I}^< \neq \emptyset$ , we create a new node  $w_l$  as the left child of  $w$ , with interval  $\mathbf{I}(w_l) = \mathbf{I}^<$ . Similarly, if  $\mathbf{I}^> \neq \emptyset$ , we create a new node  $w_r$  as the right child of  $w$ , with interval  $\mathbf{I}(w_r) = \mathbf{I}^>$ . We call the resulting tree  $\mathcal{T}$  a *delay-balanced tree*.

**Lemma 3.** Let  $\mathcal{T}$  be a delay-balanced tree. Then:

- (1) For every node  $w \in V(\mathcal{T})$  at level  $\ell$ , we have  $T(\mathbf{I}(w)) \leq T(\mathbf{I}(r))/2^\ell$ .
- (2) The depth of  $\mathcal{T}$  is at most  $O(\log T)$  and its size at most  $O(T)$ , where  $T = \prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^\alpha$ .

**Example 13.** Continuing our running example, we will construct the delay-balanced tree. Since  $\ell = 0$  for the root node,  $\tau_\ell = \tau$ . We begin by finding the split point  $\beta(r)$  for the root node. We start with the unit interval  $\mathbf{I}(r)^\leftarrow = [\langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle]$  and keep increasing the interval range until the join evaluation cost  $T(\mathbf{I}(r)^\leftarrow) > T(\mathbf{I}(r))/2$ . For interval  $\mathbf{I}(r)^\leftarrow = [\langle 1, 1, 1 \rangle, \langle 1, 1, 1 \rangle]$ , the box decomposition is  $\mathcal{B}(\mathbf{I}(r)^\leftarrow) = \mathbf{B}_3^\ell = \langle 1, 1, 1 \rangle$ .

The reader can verify that  $T(\mathbf{I}(r)^\leftarrow) = \sqrt{3|1|1|2|} \approx 2.44$  and changing the interval to  $\mathbf{I}'(r)^\leftarrow = [\langle 1, 1, 1 \rangle, \langle 1, 1, 2 \rangle]$  gives  $T(\mathbf{I}'(r)^\leftarrow) = \sqrt{3|3|4|} = 6 > T(\mathbf{I}(r))/2$ . Thus,  $\beta(r) = (1, 1, 2)$  and  $\mathbf{I}(r)^\leftarrow = [\langle 1, 2, 1 \rangle, \langle 2, 2, 2 \rangle]$  with  $T(\mathbf{I}(r)^\leftarrow) = \sqrt{1|1|2|4|} + \sqrt{1|1|3|1|} \approx 4.56$ .

For the next level  $\ell = 1$ , the threshold  $\tau_\ell = \tau/\sqrt{2} = 4/\sqrt{2} \approx 2.82$ . Since  $T(\mathbf{I}(r)^\leftarrow) \leq 2.82$ , it is a leaf node. We recursively split  $\mathbf{I}(r_r) = \mathbf{I}^>(r) = [\langle 1, 2, 1 \rangle, \langle 2, 2, 2 \rangle]$  into  $\mathbf{I}^<(r_r)$  and  $\mathbf{I}^>(r_r)$ . Fixing  $\beta(r_r) = (1, 2, 2)$ , we get  $T(\mathbf{I}^<(r_r)) = \sqrt{1|1|2|1|} \approx 1.414$  and  $\mathbf{I}^>(r_r) = [\langle 2, 1, 1 \rangle, \langle 2, 2, 2 \rangle]$ ,  $T(\mathbf{I}^>(r_r)) = \sqrt{3}$ . Since both worst case running times are smaller than  $\tau_2 = \tau/2 = 2$ , our tree construction is complete. We demonstrate the final delay-balanced tree  $\mathcal{T}$  in Figure 3.

**2) Storing Auxiliary Information.** The second step is to store auxiliary information for the heavy valuations at each node of the tree  $\mathcal{T}$ . Recall that the threshold for a heavy valuation at a node in level  $\ell$  is  $\tau_\ell = \tau/2^{\ell(1-1/\alpha)}$ . We will construct a dictionary  $\mathcal{D}$  that takes as arguments a node  $w \in V(\mathcal{T})$  at level  $\ell$  and a valuation  $v_b$  such that  $(v_b, \mathbf{I}(w))$  is  $\tau_\ell$ -heavy and returns in constant time:

$$\mathcal{D}(w, v_b) = \begin{cases} 0, & \text{if } (\boxtimes_{F \in \mathcal{E}} R_F(v_b)) \boxtimes \mathbf{I}(w) = \emptyset, \\ 1, & \text{otherwise.} \end{cases}$$

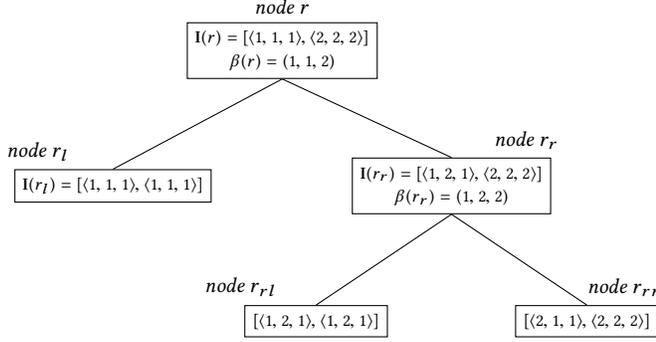


Figure 3: Delay balanced tree for running example

If  $(v_b, \mathbf{I}(w))$  is not  $\tau_\ell$ -heavy, then there is no entry for this pair in the dictionary and it simply returns  $\perp$ . In other words,  $\mathcal{D}$  remembers for the pairs that are heavy whether the answer is empty or not for the restriction of the result to the f-interval  $\mathbf{I}(w)$ . We next provide an upper bound on the size of  $\mathcal{D}$ .

**Lemma 4.**  $|\mathcal{D}| = \tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^\alpha)$ .

Appendix B shows the detailed construction that allows us to build the dictionary  $\mathcal{D}$  in time  $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ , using at most  $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^\alpha)$  space, i.e., no more space than the size of the dictionary. The final compressed representation consists of the pair  $(\mathcal{T}, \mathcal{D})$ , along with the necessary indexes on the base relations (that need only linear space).

**Example 14.** The last step for our running example is to construct the dictionary for all  $\tau_\ell$ -heavy valuations. Consider the valuation  $v_b(w_1, w_2, w_3) = (1, 1, 1)$ , which we have shown to be  $\tau$ -heavy. Next, we store a bit in the dictionary at each node for  $v_b$  denoting if the join output is non-empty for the restriction of result to interval  $\mathbf{I}$ . The reader can verify that  $(v_b, \mathbf{I}(r))$  and  $(v_b, \mathbf{I}(r_r))$  are  $\tau_0$ - and  $\tau_1$ -heavy respectively. Thus, the dictionary will store two entries for  $v_b$ :  $\mathcal{D}(\mathbf{I}(r), v_b) = 1$ ,  $\mathcal{D}(\mathbf{I}(r_r), v_b) = 1$ .

#### 4.4 Answering a Query

We now explain how we can use the data structure to answer an access request  $q = Q^\eta[v]$  given by a valuation  $v$ . The detailed algorithm is depicted in Algorithm 2.

We start traversing the tree from the root  $r$ . For a node  $w$ , if  $\mathcal{D}(w, v_b) = \perp$ , we compute the corresponding subinstance using a worst-case optimal algorithm for every box in the box decomposition. If  $\mathcal{D}(w, v_b) = 0$ , we do nothing. If  $\mathcal{D}(w, v_b) = 1$ , we recursively traverse the left child (if it exists), compute the instance for the unit interval  $[\beta(w), \beta(w)]$ , then recursively traverse the right child (if it exists). This traversal order guarantees that the tuples are output in lexicographic order.

We prove in full version of the paper [14] the following result about the performance of Algorithm 2:

**Lemma 5.** Algorithm 2 enumerates  $q(D)$  in lexicographic order with delay  $\delta = \tilde{O}(\tau)$  and answer time  $T_A = \tilde{O}(|q(D)| + \tau \cdot |q(D)|^{1/\alpha})$ .

---

#### Algorithm 2: Answering a query $q = Q^\eta[v_b]$

---

**INPUT** : tree  $\mathcal{T}$ , dictionary  $\mathcal{D}$ , valuation  $v$

**OUTPUT**: query answer  $q(D)$

```

1 eval( $r, v_b$ ) /* start from the root */
2 return
3 PROCEDURE eval( $w, v_b$ )
4   if  $\mathcal{D}(w, v_b) = \perp$  then
5     forall  $\mathbf{B} \in \mathcal{B}(\mathbf{I}(w))$  do
6       | output  $\bowtie_{F \in \mathcal{E}} R_F(v_b, \mathbf{B})$ 
7     end
8   else if  $\mathcal{D}(w, v_b) = 1$  then
9     if  $w$  has left child  $w_\ell$  then
10      | eval( $w_\ell, v_b$ )
11     output  $\bowtie_{F \in \mathcal{E}} R_F(v_b, [\beta(w), \beta(w)])$ 
12     if  $w$  has right child  $w_r$  then
13      | eval( $w_r, v_b$ )
14   return

```

---

## 5 QUERY DECOMPOSITIONS

In this section, we prove Theorem 2. Consider an adorned view over a natural join query (with hypergraph  $\mathcal{H}$ ), with bound variables  $\mathcal{V}_b$ . Fix a  $\mathcal{V}_b$ -connex tree decomposition  $(\mathcal{T}, A)$ . Observe that, since the bags in  $A$  do not play any role in the width or height, we can assume w.l.o.g. that  $A$  consists of a single bag  $t_b$ . We consider the example in Figure 2 as the running example for this section.

### 5.1 Constant Delay Enumeration

As a warm-up, we first show how to construct the data structure for Proposition 4, when our goal is to achieve constant delay for the enumeration of every access query.

In the full enumeration case, where  $\mathcal{V}_b = \emptyset$ , we can take any optimal tree decomposition with fractional hypertree width  $\text{fhw}(\mathcal{H})$  (so a  $\mathcal{V}_b$ -connex decomposition), materialize the tuples in each bag by running the query restricted on the vertices of the bag, and finally apply a sequence of semi-joins in a bottom-up order to remove any tuples from the bags that do not participate in the final result. Additionally, for each node  $t \in V(\mathcal{T})$ , we construct a hash index over the materialized result with key  $\mathcal{V}_b^t = \mathcal{B}_t \cap \mathcal{V}_b^t$ , and output values for the variables in  $\mathcal{V}_f^t = \mathcal{B}_t \setminus \mathcal{V}_b^t$ . For example, the node with bag  $\{v_2, v_3\}$  constructs an index with key  $v_2$  that returns all the matching values of  $v_3$ . Given such indexes, we can perform a full enumeration in constant delay starting from the root (which will be the empty bag), and visiting the nodes of the tree  $\mathcal{T}$  in a pre-order fashion by following the indexes at each bag. This construction uses the same idea as  $d$ -representations [23], and requires space  $O(|D|^{\text{fhw}(\mathcal{H})})$ .

When  $\mathcal{V}_b \neq \emptyset$ , the standard tree decomposition may not be useful to achieve constant delay enumeration. For instance, for the example hypergraph of Figure 2, if the adorned view has  $\mathcal{V}_b = \{v_1, v_5, v_6\}$ , then the pre-order traversal of the decomposition on the left will fail to achieve constant delay. However, we can use a  $\mathcal{V}_b$ -connex decomposition to successfully answer any access request (e.g., the decomposition on the right in Figure 2). We first materialize

all the bags, except for the bag of  $t_b$ . Then, we run a sequence of semi-joins in a bottom-up manner, where we stop right before the node  $t_b$  (since it is not materialized). For each node  $t \in V(\mathcal{T}) \setminus \{t_b\}$ , we construct a hash index over the materialized result with key  $\mathcal{V}_b^t$ . Finally, for the root node  $t_b$ , we construct a hash index that tests membership for every hyperedge of  $\mathcal{H}$  that is contained in  $\mathcal{V}_b$ . For the example in Figure 2, we construct one such index for the hyperedge  $\{v_5, v_6\}$ .

Given a valuation  $v_b$  over  $\mathcal{V}_b$ , we answer the access request as follows. We start by checking in constant time whether  $v_b(v_5, v_6)$  is in the input. Then, we use the hash index of the node with bag  $\{v_2, v_4, v_1, v_5\}$  to find the matching values for  $v_2, v_4$  (since  $v_1, v_5$  are bound by  $v_b$ ), and subsequently use the hash index of  $\{v_3, v_2, v_4\}$  to find the matching values for  $v_3$ ; similarly, we also traverse the right subtree starting of the root node to find the matching values of  $v_7$ . We keep doing this traversal until all tuples are enumerated. We describe next how this algorithm generalizes for any adorned view and beyond constant delay.

## 5.2 Beyond Constant Delay

We will now sketch the construction and query answering for the general case. The detailed construction and algorithms are presented in Appendix C, D. Along with the tree decomposition, let us fix a delay assignment  $\delta$ .

**Construction Sketch.** The first step is to apply for each node  $t$  in  $\mathcal{T}$  (except  $t_b$ ) the construction of the data structure from Theorem 1, with the following parameters: (i) hypergraph  $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$  where  $\mathcal{V}' = \mathcal{B}_t$  and  $\mathcal{E}' = \mathcal{E}_{\mathcal{B}_t}$ , (ii) bound variables  $\mathcal{V}'_b = \mathcal{V}_b^t$ , (iii)  $\tau = |D|^{\delta(t)}$ , and (iv)  $\mathbf{u}$  the fractional edge cover that minimizes  $\rho_t^+$ . For the root node  $t_b$ , we simply construct a hash index that tests membership for every hyperedge of  $\mathcal{H}$  that is contained in  $\mathcal{V}_b$ . This construction uses for each bag space  $\tilde{O}(|D| + |D|^{\rho_t^+ - \delta(t) \cdot \alpha(\mathcal{V}_b^t)})$ , which means that the compressed representation has size  $\tilde{O}(|D| + |D|^f)$ .

The second step is to run a sequence of semi-joins in bottom-up fashion. However, since the bags are not fully materialized anymore, this operation is not straightforward. Instead, we flip all entries of the dictionary  $\mathcal{D}_t(w, v_b)$  of the data structure at node  $t$  from 1 to 0 if no valuation in the interval  $I(w)$  joins with its child bag. This step is necessary to guarantee that if we visit an interval in the delay-balanced tree with entry 1, we are certain to produce an output for the full query (and not only the particular bag). To perform this check, we do not actually materialize the bag of the child, but we simply use its dictionary (hence costing an extra factor of  $\max_t \delta(t)$  during preprocessing time).<sup>5</sup>

**Query Answering.** To answer an access query with valuation  $v_b$ , we start from the root node  $t_b$  of the decomposition and check using the indexes of the node whether  $v_b$  belongs to all relations  $R_F$  such that  $F \subseteq \mathcal{V}_b$ . Then, we invoke Algorithm 2 on the left-most child  $t_0$  of  $t_b$ , which outputs a new valuation in time at most  $\tilde{O}(|D|^{\delta(t_0)})$ , or returns nothing. As soon as we obtain a new output, we recursively proceed to the next bag in pre-order traversal of  $\mathcal{T}$ , and find valuations for the (still free) variables in the bag. If there

<sup>5</sup>Recall that since we are applying Theorem 1 to each bag, there will be a separate data structure created for each bag in the decomposition.

are no such valuations returned by Algorithm 2 for the node under consideration, this means that the last valuation outputted by the parent node does not lead to any output. In this case, we resume the enumeration for the parent node. Finally, when Algorithm 2 finishes the enumeration procedure, then we resume the enumeration for the pre-order predecessor of the current node (and not the parent). Intuitively, we go the predecessor to fix our next valuation, in order to enumerate the cartesian product of all free variables in the subtree rooted at the least common ancestor of current node and predecessor node.

The delay guarantee of  $\tilde{O}(|D|^h)$  comes from the fact that, at every node  $t$  in the tree, we will output in time  $\tilde{O}(|D|^{\delta(t)})$  at most  $\tilde{O}(|D|^{\delta(t)})$  valuations, one of which will produce a final output tuple. Moreover, when a node has multiple children, then for a fixed valuation of the node, the traversal of each child is independent of the other children: if one subtree produces no result, then we can safely exit all subtrees and continue the enumeration of the node. The full details and analysis of the algorithm are in Appendix D.

## 6 THE COMPLEXITY OF MINIMIZING DELAY

In this section, we study the computational complexity of choosing the optimal parameters for Theorem 1 and Theorem 2. We identify two objectives that guide the parameter choice: (i) given a space constraint, minimize the delay, and (ii) given a delay constraint, minimize the necessary space.

We start with the following computational task, which we call **MINDELAYCOVER**. We are given as input a full adorned view  $Q^\eta$  over a CQ, the sizes  $|R_F|$  of each relation  $F$ , and a positive integer  $\Sigma$  as a space constraint. The size of  $Q^\eta$ , denoted  $|Q^\eta|$ , is defined as the length of  $Q^\eta$  when viewed as a word over alphabet that consists of variable set  $\mathcal{V}$ , **dom** and atoms in the body of the query. The goal is to output a fractional edge cover  $\mathbf{u}$  that minimizes the delay in Theorem 1, subject to the space constraint  $S \leq \Sigma$ .

We observe that we can express **MINDELAYCOVER** as a linear fractional program with a bounded and non-empty feasible region (see [14]). Such a program can always be transformed to an equivalent linear program [9], which means that the problem can be solved in polynomial time.

**Proposition 9.** *MINDELAYCOVER can be solved in polynomial time in the size of the adorned view, the relation sizes, and the space constraint.*

We also consider the inverse task, called **MINSPACECOVER**: given as input a full adorned view  $Q^\eta$  over a CQ, the sizes  $|R_F|$  of each relation  $F$ , and a positive integer  $\Delta$  as a delay constraint, we want to output a fractional edge cover  $\mathbf{u}$  that minimizes the space  $S$  in Theorem 1, subject to the delay constraint  $\tau \leq \Delta$ .

To solve **MINSPACECOVER**, observe that we can simply perform a binary search over the space parameter  $S$ , from  $|D|$  to  $|D|^k$ , where  $k$  is the number of atoms in  $Q$ . For each space, we then run **MINDELAYCOVER** and check whether the minimum delay returned satisfies the delay constraint.

**Proposition 10.** *MINSPACECOVER can be solved in polynomial time in the size of the adorned view, the relation sizes, and the delay constraint.*

We next turn our attention to how to optimize for the parameters in Theorem 2. Suppose we are given a full adorned view  $Q^\eta$  over a CQ, the database size  $|D|$ , and a space constraint  $\Sigma$ , and we want to minimize the delay. If we are given a fixed  $\mathcal{V}_b$ -connex tree decomposition, then we can compute the optimal delay assignment  $\delta$  and optimal fractional edge cover for each bag as follows: we iterate over every bag in the tree decomposition, and then solve MINDELAYCOVER for each bag using the space constraint. It is easy to see that the delay that we obtain in each bag must be the delay of an optimal delay assignment. For the inverse task where we are given a  $\mathcal{V}_b$ -connex tree decomposition, a delay constraint, and our goal is to minimize the space, we can apply the same binary search technique as in the case of MINSPACECOVER (observe that the  $\delta$ -height is also easily computable in polynomial time). In other words, we can compute the optimal parameters for our given objective in polynomial time, as long as we are provided with a tree decomposition.

In the case where the tree decomposition is not given, then the problem of finding the optimal data structure according to Theorem 2 becomes intractable. Indeed, we have already seen that if we want to achieve constant delay  $\tau = 1$ , then the tree decomposition that minimizes the space  $S$  is the one that achieves the  $\mathcal{V}_b$ -connex fractional hypertree width,  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$ . Since for  $\mathcal{V}_b = \emptyset$  we have  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) = \text{fhw}(\mathcal{H})$ , and finding the optimal fractional hypertree width is NP-hard [16], finding the optimal tree decomposition for our setting is also NP-hard.

## 7 RELATED WORK

There has been a significant amount of literature on data compression; a common application is to apply compression in column-stores [1]. However, such compression methods typically ignore the logical structure that governs data that is a result of a relational query. The key observation is that we can take advantage of the underlying logical structure in order to design algorithms that can compress the data effectively. This idea has been explored before in the context of *factorized databases* [23], which can be viewed as a form of logical compression. Our approach builds upon the idea of using query decompositions as a factorized representation, and we show that for certain access patterns it is possible to go below  $|D|^{\text{fhw}}$  space for constant delay enumeration. In addition, our results also allow trading off delay for smaller space requirements of the data structure. A long line of work has also investigated the application of a broader set of queries with projections and aggregations [6, 7], as well as learning linear regression models over factorized databases [22, 25]. Closely related to our setting is the investigation of join-at-a-time query plans, where at each step, a join over one variable is computed [10]. The intermediate results of these plans are partial factorized representations that compress only a part of the query result. Thus, they can be used to tradeoff space with delay, albeit in a non-tunable manner.

Our work is also connected to the problem of constant-delay enumeration [5, 26, 27]: in this case, we want to enumerate a query result with constant delay after a linear time preprocessing step. We can view the linear time preprocessing step as a compression algorithm, which needs space only  $O(|D|)$ . It has been shown that

the class of *connex-free acyclic conjunctive queries* can be enumerated with constant delay after a linear-time preprocessing. Hence, in the case of connex-free acyclic CQs, there exists an optimal compression/decompression algorithm. However, many classes of queries are not factorizable to linear size, and can not be enumerated with constant-delay after linear-time preprocessing. Such examples are the triangle query  $\Delta^{\text{fff}}(x, y, z) = R(x, y), R(y, z), R(z, x)$ , or the 2-path query  $P_2^{\text{ff}}(x, y) = R(x, y), R(y, z)$ .

Query compression is also a central problem in graph analytics. Many applications involve extracting insights from relational databases using graph queries. In such situations, most systems load the relational data in-memory and expand it into a graph representation that can become very dense. Analysis of such graphs is infeasible, as the graph size blows up quickly. Recent work [29–31] introduced the idea of controlled graph expansion by storing information about high-degree nodes and evaluating acyclic CQs over light sub-instances. However, this work is restricted only to binary views (*i.e.*, graphs), and does not offer any formal guarantees on delay or answer time. It also does not allow the compressed representation to grow more than linear in the size of the input.

Finally, our work is closely related to the problem of set intersection. Set intersection has applications in problems related to document indexing [3, 11] and proving hardness results and lower bounds for space/approximation tradeoff of distance oracles for graphs [12, 24]. Previous work [11] has looked at creating a data structure for fast set intersection reporting and its corresponding boolean version. Our main data structure is a strict generalization of the one from [11].

## 8 CONCLUSION

In this paper we propose a novel and tunable data structure that allows us to compress the result of a conjunctive query, so that we can answer efficiently access requests over the query output.

This work initiates an exciting direction on studying compression tradeoffs for query results, and thus there are several open problems. The main challenge is to show whether our proposed data structure achieves optimal tradeoffs between the various parameters. Recent work [3] makes it plausible to look for lower bounds in the pointer machine model. A second open problem is to explore how our data structures can be modified to support efficient updates of the base tables. Recent results [8] indicate that efficient maintenance of CQ results under updates is in general a hard problem. This creates a new challenge for designing data structures and algorithms that provide theoretical guarantees and work well in practice. A third challenge is to extend our algorithms to support views with projections: projections add the additional challenge that we have to deal with duplicate tuples in the output. Finally, an interesting question is whether it is possible to build our data structure *on-the-fly* without a preprocessing step.

## 9 ACKNOWLEDGMENTS

This work is partially supported by Google and the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation.

## REFERENCES

- [1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.
- [2] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. Spartex: A vertex-centric framework for rdf data analytics. *Proceedings of the VLDB Endowment*, 8(12):1880–1883, 2015.
- [3] P. Afshani and J. A. S. Nielsen. Data structure lower bounds for document indexing problems. In *ICALP 2016 Automata, Languages and Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik GmbH, 2016.
- [4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
- [5] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, pages 208–222, 2007.
- [6] N. Bakibayev, T. Kočický, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.
- [7] N. Bakibayev, D. Olteanu, and J. Závodný. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.
- [8] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318. ACM, 2017.
- [9] A. Charnes and W. W. Cooper. Programming with linear fractional functionals. *Naval Research Logistics (NRL)*, 9(3-4):181–186, 1962.
- [10] R. Ciucanu and D. Olteanu. Worst-case optimal join at a time. Technical report, Technical report, Oxford, 2015.
- [11] H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.
- [12] H. Cohen and E. Porat. On the hardness of distance oracle for sparse graph. *arXiv preprint arXiv:1006.1117*, 2010.
- [13] P. Davoodi, M. Smid, and F. van Walderveen. Two-dimensional range diameter queries. In *Proceedings of the 10th Latin American International Conference on Theoretical Informatics, LATIN’12*, pages 219–230, Berlin, Heidelberg, 2012. Springer-Verlag.
- [14] S. Deep and P. Koutris. Compressed representations of conjunctive query results. *arXiv preprint arXiv:1709.06186*, 2017.
- [15] I. Goldstein, T. Kopelowitz, M. Lewenstein, and E. Porat. Conditional lower bounds for space/time tradeoffs. In *Workshop on Algorithms and Data Structures*, pages 421–436. Springer, 2017.
- [16] G. Gottlob, G. Greco, and F. Scarcello. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems*, 1, 2014.
- [17] J. E. Hopcroft, J. D. Ullman, and A. Aho. The design and analysis of computer algorithms, 1975.
- [18] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.
- [19] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.
- [20] F. Niu, C. Zhang, C. Ré, and J. Shavlik. Felix: Scaling Inference for Markov Logic with an Operator-based Approach. *ArXiv e-prints*, Aug. 2011.
- [21] D. Olteanu and M. Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.
- [22] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2, 2015.
- [23] M. Patrascu and L. Roditty. Distance oracles beyond the thorup-zwick bound. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 815–823. IEEE, 2010.
- [24] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.
- [25] L. Segoufin. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT ’13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 10–20, 2013.
- [26] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015.
- [27] J. D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, Sept. 1985.
- [28] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 897–912. ACM, 2017.
- [29] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proceedings of the VLDB Endowment*, 8(12):2032–2035, 2015.

- [30] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES’17*, pages 9:1–9:7, New York, NY, USA, 2017. ACM.

## A BOX DECOMPOSITION

**Example 15.** Consider the adorned view

$$\mathcal{V}^{\text{fbbf}}(x, y, z, w) = R_1(x, y), R_2(y, z), R_3(z, w), R_4(w, x).$$

Assume that the active domain is  $\mathbf{D}[x] = \mathbf{D}[y] = \mathbf{D}[z] = \mathbf{D}[w] = \{1, 2\}$ . Since  $\mathcal{V}_f = \{x, z, w\}$ , consider the  $f$ -interval  $\mathbf{I} = [\mathbf{a}, \mathbf{b}]$  where  $\mathbf{a} = \langle 1, 2, 1 \rangle$  and  $\mathbf{b} = \langle 2, 1, 2 \rangle$ . In other words, interval  $\mathbf{I}$  contains the following valuations for  $\mathcal{V}_f$ :  $(1, 2, 1)$ ,  $(1, 2, 2)$ ,  $(2, 1, 1)$ ,  $(2, 1, 2)$ . It is easy to verify that  $R_i \times \mathbf{I} = R_i$  for every  $i = 1, 2, 3, 4$  and that  $(1, 1, 1, 1)$  is an output tuple. However,  $(\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{I}$  filters out  $(1, 1, 1, 1)$  as  $(1, 1, 1)$  does not lie in the interval  $\mathbf{I}$ .

Next, we state an important lemma that will be useful subsequently to bound the running time of the dictionary construction. We defer its proof to the full version of the paper [14].

**Lemma 6.** Let  $\mathbf{B} = \langle I_1, \dots, I_i, \dots \rangle$  be an  $f$ -box, and  $J_1, \dots, J_p$  a partition of the interval  $I_i$ . Denote  $\mathbf{B}_k = \langle I_1, \dots, J_k, \dots \rangle$ . Then,  $\sum_{k=1}^p T(\mathbf{B}_k) \leq T(\mathbf{B})$ .

## B DICTIONARY CONSTRUCTION

In this section, we show how to efficiently construct the dictionary  $\mathcal{D}$  from the input database  $D$ . In particular, we prove the following:

**Lemma 7.** The dictionary  $\mathcal{D}$  can be constructed in time  $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ , using at most  $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^\alpha)$  space.

We construct the dictionary  $\mathcal{D}$  as follows:

**a) Find Heavy Valuations.** The first step of the algorithm is to compute the list of heavy valuations  $v_b$  for any interval  $\mathbf{I}$ .

**Proposition 11.** Let  $\mathcal{L}_1$  denote the sorted list of all valuations of  $\mathcal{V}_b$  such that  $(v_b, \mathbf{I})$  is  $\tau$ -heavy. Then,  $\mathcal{L}_1$  can be constructed in time  $\tilde{O}(\sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \prod_{F \in \mathcal{E}_{\mathcal{V}_b}} |R_F \times \mathbf{B}|^{u_F})$  using space  $O((T(\mathbf{I})/\tau)^\alpha)$ .

**PROOF.** The first observation is that for all heavy  $(v_b, \mathbf{I})$  valuations, since  $T(v_b, \mathbf{I}) > \tau$ , there exists  $\mathbf{B} \in \mathcal{B}(\mathbf{I})$  such that  $R_F(v_b) \times \mathbf{B}$  is non-empty for each  $F \in \mathcal{E}_{\mathcal{V}_b}$ . This implies that  $\pi_{F \cap \mathcal{V}_b}(v_b) \in \pi_{F \cap \mathcal{V}_b}(R_F \times \mathbf{B})$  (otherwise the relation will be empty and  $T(v_b, \mathbf{I}) = 0$ ). Thus, it is sufficient to compute  $\pi_{\mathcal{V}_b}((\bowtie_{F \in \mathcal{E}_{\mathcal{V}_b}} R_F) \times \mathbf{I})$  to find all heavy valuations.

We can construct the list  $\mathcal{L}_1$  by running a worst case join algorithm in time  $O(\sum_{\mathbf{B} \in \mathcal{B}(\mathbf{I})} \prod_{F \in \mathcal{E}_{\mathcal{V}_b}} |R_F \times \mathbf{B}|^{u_F})$ . Additionally, as soon as the worst case join algorithm generates an output  $v_b$ , we check if  $v_b$  is  $\tau$ -heavy in  $\tilde{O}(1)$  time. This can be done by using linear sized indexes on base relations to count the number of tuples in each relation  $R_{F \in \mathcal{E}}(v_b, \mathbf{I})$  and using  $u$  as cover to check whether the execution time is greater than the threshold  $\tau$ . Since we need only heavy valuations, we only retain those in memory. Proposition 7 bounds the space requirement of  $\mathcal{L}_1$  to at most  $O((T(\mathbf{I})/\tau)^\alpha)$ . Sorting  $\mathcal{L}_1$  introduces at most an additional  $O(\log |D|)$  factor.  $\square$

**b) Using join output to create  $\mathcal{D}$ .** Consider the delay balanced tree  $\mathcal{T}$  as constructed in the first step. Without loss of generality, assume that the tree is full. We bound the time taken to create  $\mathcal{D}(w, v_b)$

for all nodes  $w_L$  at some level  $L$  (applying the same steps to other levels introduces at most  $\log |D|$  factor). The detailed algorithm is presented below.

---

**Algorithm 3:** Create Dictionary  $\mathcal{D}(w, v_b)$  for level  $L$  nodes in  $\mathcal{T}$

---

**INPUT** : tree  $\mathcal{T}$   
**OUTPUT**:  $\mathcal{D}(w, v_b)$  for all leaf nodes

```

1 forall  $w$  in  $w_L$  do
  /* Run NPRR on  $(\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{I}(w)$  to compute  $\mathcal{L}_{\mathbf{I}(w)}$  */
2 forall  $v_b \in \mathcal{L}_{\mathbf{I}(w)}$  do
3   |  $\mathcal{D}(w, v_b) = 0$  /* initializing  $\mathcal{D}$  with all heavy pairs */
4 end
  /* Run NPRR on  $(\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{I}(w)$  */
5 forall  $j \leftarrow$  output tuple from NPRR /* requires  $\log |D|$ 
   main memory */
6 do
7   |  $v_b \leftarrow \Pi_{\mathcal{V}_b}(j)$ 
8   | if  $v_b \in \mathcal{L}_{\mathbf{I}(w)}$  then /* binary search over  $\mathcal{L}_{\mathbf{I}(w)}$  */
9     |  $\mathcal{D}(w, v_b) = 1$ 
10  | end
11 end
12 end

```

---

**Algorithm Analysis.** We first bound the running time of the algorithm.

**Proposition 12.** *Algorithm 3 runs in time  $\tilde{O}(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ .*

**PROOF.** We will first compute the time needed to construct the list  $\mathcal{L}_{\mathbf{I}(w)}$  for all nodes  $w$  at level  $L$ . Proposition 11 tells us that to find the heavy valuations for an interval  $\mathbf{I}(w)$  we need time  $\tilde{O}(\sum_{B \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \times \mathbf{B}|^{u_F})$ . We will apply Lemma 6 to show that  $\sum_{w \in w_L} \sum_{B \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \times \mathbf{B}|^{u_F} = O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ . Consider all the  $f$ -boxes in the box decomposition of  $\mathbf{I}(w)$ ,  $\forall w \in w_L$ . All  $f$ -boxes that have the first  $\mu - 1$  variables fixed are of the form  $\mathbf{B}_\mu^k = \langle a_1, \dots, a_{\mu-1}, (a_\mu^k, b_\mu^k) \rangle$ . We apply Lemma 6 with  $i = \mu$  to all such boxes. Thus,  $\sum_k T(\mathbf{B}_\mu^k) \leq T(\langle a_1, \dots, a_{\mu-1}, \square \rangle)$ .

After this step, all  $f$ -boxes have unit interval prefix of length at most  $\mu - 1$  and have the domain of  $x_\mu^f$  as  $\square$ . We repeatedly apply lemma 6 to all boxes with  $i = \mu - 1, \mu - 2, \dots, 1$  sequentially. Each application merges the boxes and fixes the domain of  $x_i^f = \square$ . The last step merges all  $f$ -boxes of the form  $\langle a \rangle$  to  $\mathbf{I}(r) = \langle \square, \dots, \square \rangle$ . This gives us,

$$\begin{aligned} \sum_{w \in w_L} \sum_{B \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \times \mathbf{B}|^{u_F} &\leq \prod_{F \in \mathcal{E}} |R_F \times \mathbf{I}(r)|^{u_F} \\ &= O\left(\prod_{F \in \mathcal{E}} |R_F|^{u_F}\right) \end{aligned}$$

The second step is to bound the running time of the worst case join optimal algorithm to compute  $(\bowtie_{F \in \mathcal{E}} R_F) \times \mathbf{I}(w)$  in line 5. Observe that this join can also be computed in worst case time  $\sum_{w \in w_L} \sum_{B \in \mathcal{B}(\mathbf{I}(w))} \prod_{F \in \mathcal{E}} |R_F \times \mathbf{B}|^{u_F} = O(\prod_{F \in \mathcal{E}} |R_F|^{u_F})$ . Finally, note that all steps in line 7-line 10 are  $\tilde{O}(1)$  operations.  $\square$

Next, we analyze the space requirement of Algorithm 3.

**Proposition 13.** *Algorithm 3 requires space  $O(\prod_{F \in \mathcal{E}} |R_F|^{u_F} / \tau^\alpha)$*

**PROOF.** Lines 2-3 take  $|D|$  amount of space. The NPRR algorithm requires  $\log |D|$  amount of memory to keep track of pointers.<sup>6</sup> Since we are only streaming through the join output, there is no additional memory overhead in this step. Thus, the bound on memory required follows from Proposition 11 (bounding the size of  $\mathcal{L}_{\mathbf{I}(r)}$ ) and Lemma 4.  $\square$

## C CONSTRUCTING DATA STRUCTURE FOR QUERY DECOMPOSITION

In this section, we present the detailed construction of data structure and time required for Theorem 2. Without loss of generality, we assume that all bound variables are present in a single bag  $t_b$  in the  $\mathcal{V}_b$ -connex tree decomposition of the hypergraph  $\mathcal{H}$ . This can be achieved by simply merging all the bags  $t$  with  $\mathcal{B}_t \subseteq \mathcal{V}_b$  into  $t_b$  which is also designated as the root. Note that the delay assignment for root node is  $\delta_{t_b} = 0$ . Let  $\lambda(\mathcal{T})$  denote the set of all root to leaf paths in  $\mathcal{T}$ ,  $h$  be the  $\delta$ -height of the decomposition and  $f$  be the  $\mathcal{V}_b$ -connex fractional hypertree  $\delta$ -width of the decomposition. We also define the quantity  $u^* = \max_{t \in V(\mathcal{T}) \setminus t_b} \sum_F u'_{t,F}$  where  $u'_t$  is the fractional edge cover for bag  $\mathcal{B}_t$  that minimizes  $\rho_t^+$ . We will show that in time  $T_C = \tilde{O}(|D| + |D|^{u^* + \max_t \delta_t})$ , we can construct required data structure using space  $S = \tilde{O}(|D| + |D|^f)$  for a given  $\mathcal{V}_b$ -connex tree decomposition  $\mathcal{T}$  of  $\delta$ -width  $f$ . The construction will proceed in two steps:

**(i) Apply Theorem 1 to decomposition.** We apply Theorem 1 to each bag (except  $t_b$ ) in the decomposition with the following parameters: (i)  $\mathcal{H}^t = (\mathcal{V}^t, \mathcal{E}^t)$  where  $\mathcal{V}^t = \mathcal{B}_t$  and  $\mathcal{E}^t = \mathcal{E}_{\mathcal{B}_t}$ , (ii)  $\mathcal{V}_b^t = \text{anc}(t) \cap \mathcal{B}_t$  and  $\mathcal{V}_f^t = \mathcal{B}_t \setminus \text{anc}(t)$ , and (iii) the edge cover  $\mathbf{u}$  is the cover of the node  $t$  in the decomposition corresponding to  $\rho_t^+$ . Thus, in time  $\tilde{O}(|D| + |D|^{u^*})$  we can construct delay-balanced tree  $\mathcal{T}_t$  and the corresponding dictionary  $\mathcal{D}_t$  for each bag other than the root. The space requirement for each bag is no more than  $\tilde{O}(|D| + |D|^f)$ .

However, the dictionary  $\mathcal{D}_t$  needs to be modified for each bag since there can be *dangling tuples* in a bag that may participate only in the join output of the bag but not in join output of the branch containing  $t$ . In the following description, we will use the notation  $v_b^t$  to denote a valuation over variables  $\mathcal{V}_b^t$ . Note that  $v_b$  is the valuation over  $\mathcal{V}_b$ .

**(ii) Modify  $\mathcal{D}_t$  using semijoins.** Algorithm 4 shows the construction of the modified dictionary  $\mathcal{D}_t$  to incorporate the semijoin result. The goal of this step is to ensure that if  $\mathcal{D}_t(w, v_b^t) = 1$ , then there exists a set of valid valuations for all variables in the subtree rooted at  $t$ . We will apply a sequence of semijoin operations in a bottom up fashion which we describe next.

**Bottom Up Semijoin.** In this phase, the bags are processed according to *post-order* traversal of the tree in bottom-up fashion. The key idea is to stream over all heavy valuations of a node in  $\mathcal{T}$  and ensure that they join with some tuple in the child bags. Let

<sup>6</sup>If we have at least  $|D|$  memory, i.e, all relations can fit in memory, then we require no subsequent I/O's

**Algorithm 4:** Modifying  $(\mathcal{D})_{t \in V(\mathcal{T})}$ 


---

**INPUT** :  $\mathcal{V}_b$ -bound decomposition  $\mathcal{T}, (\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T})}$

---

```

1 forall  $t \in \mathcal{T} \setminus \{t_b \cup \text{children of } t_b\}$  in post-order fashion do
2    $\text{parent} \leftarrow \text{parent of } t$ 
3   for  $\ell = 0$  to  $L$  /*  $L$  is the number of levels in  $\mathcal{T}_{\text{parent}}$  */
4   do
5     forall  $w \in w_\ell$  of  $\mathcal{T}_{\text{parent}}$  /*  $w_\ell$  represents all nodes at
6       level  $\ell$  in  $\mathcal{T}_{\text{parent}}$  */
7     do
8       forall heavy  $v_b^{\text{parent}} \in w$  and
9          $\mathcal{D}_{\text{parent}}(w, v_b^{\text{parent}}) = 1$  do
10        forall  $k \leftarrow Q_{\text{parent}}(v_b^{\text{parent}}, D) \times \mathbf{I}(w)$ 
11          /* computing  $(\bowtie_{F \in \mathcal{E}_{\mathcal{V}^t}} R_F)$  via box
12            decomposition */
13          do
14            if Algorithm 2 on  $t$  with
15               $v_b^t = \pi_{\mathcal{B}_{\text{parent}} \cap \mathcal{B}_t}(k)$  is empty for all  $k$ 
16            then
17               $\mathcal{D}_t(w, \pi_{\mathcal{V}_b^{\text{parent}}}(k)) = 0$ 
18            end
19          end
20        end
21      end
22    end
23  end

```

---

$Q_t(v_b^t, D)$  denote the NPRR join instance on the relations covering variables  $\mathcal{V}^t$  where bound variables are fixed to  $v_b^t$ . When processing a non-root (or non-child of root) node  $t_j$ , a semijoin is performed with its parent  $t_i$  to flip all dictionary entries of  $t_i$  from 1 to 0 if the entry does not join with any tuple in  $t_j$  on their common attributes  $\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}$ . To perform this operation, we stream over all tuples  $k \leftarrow Q_{t_i}(v_b^{t_i}, D)$  and check if  $\pi_{\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}}(k)$  is present in the join output of relations covering  $t_j$ . This check in bag  $t_j$  can be performed by invoking Algorithm 2 with bound valuation  $\pi_{\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j}}(k)$  in time  $\tilde{O}(|D|^{\delta_{t_j}})$ .

**Algorithm Analysis.** We will show that Algorithm 4 can be executed in time  $\tilde{O}(|D|^{u^* + \max_t \delta_t})$ .

**Proposition 14.** *Algorithm 4 has running time  $\tilde{O}(|D|^{u^* + \max_t \delta_t})$ .*

**PROOF.** The main observation is that the join  $Q_{\text{parent}}(v_b^t, D) \times \mathbf{I}(w)$  for all nodes at level  $L$  in  $\mathcal{T}_{\text{parent}}$  can be computed in time at most  $O(|D|^{u^*})$  as shown in Proposition 12. Since the operation in Line 10 can be performed in time  $\tilde{O}(|D|^{\delta_t})$  for each  $k$  and  $\mathcal{T}_{\text{parent}}$  has at most logarithmic number of levels, the total overhead of the procedure is dominated by the semijoin operation where delay for bag  $t$  is the largest. This implies a running time of  $\tilde{O}(|D|^{u^* + \max_t \delta_t})$  for the procedure.  $\square$

**Proposition 15.** *If  $\mathcal{D}_t(w, v_b^t) = 1$ , then there exists a set of valuations for all variables in the subtree rooted at  $t$  for  $v_b^t$ .*

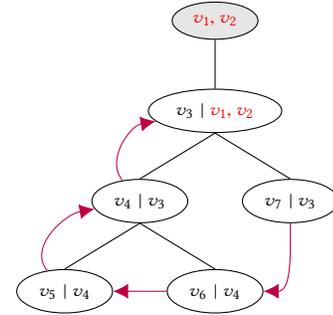
**PROOF.** Consider a valuation such that  $\mathcal{D}_t(w, v_b^t) = 1$ . If  $v_b^t$  does not join with the relations of any child bag  $c$ , then Line 10 would be true and Algorithm 4 would have flipped the dictionary entry to 0. Thus, there exists a valuation for  $\mathcal{V}_f^c$ . Applying the same reasoning inductively to each child bag  $c$  till we reach the leaf nodes gives us the desired result.  $\square$

The modified dictionary, along with the enumeration algorithm, guarantees that any valuation of free variables that is output by Algorithm 2 for a particular bag will also produce an output for the subtree of that bag. Note that the main memory requirement of Algorithm 4 is only  $O(1)$  pointers and the data structures for each bag which takes  $\tilde{O}(|D| + |D|^f)$  space.

## D ANSWERING USING QUERY DECOMPOSITION

In this section, we present the query answering algorithm for the given  $\mathcal{V}_b$ -connex decomposition. We will first add some metadata to each bag in the decomposition and then invoke algorithm 2 for each bag in pre-order fashion.

**Adding pointers for each bag.** Consider the decomposition  $\mathcal{T}$  along with  $(\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T}) \setminus t_b}$  for each bag. We will modify  $\mathcal{T}$  as follows: for each node of the tree, we fix a pointer *predecessor*( $t$ ), that will point to the *pre-order predecessor* of the node. Intuitively, the pre-order predecessor of a node is the last node where the valuation for a free variable will be fixed in the pre-order traversal of the tree just before visiting the current node. Figure 4 shows an example of a modified decomposition. This transformation can be done in  $O(1)$  time.



**Figure 4:** Example of the modified tree decomposition: the arrows in color are the predecessor pointers

For ease of description of the algorithm, we assume that Algorithm 2 answering  $Q^{\eta}[v_b^t]$  for any bag  $t$  is accessible using the procedure  $\text{next}_t(v_b^t)$ . Let  $\mathcal{V}_{\text{pred}}^t$  represent all bound variables encountered in the pre-order traversal of the tree from  $t_b$  to  $t$  (including bound variables of  $t$ ).

**Algorithm Description.** The algorithm begins from the root node and fixes the valuation for all free variables in the root bag. Then, it proceeds to the next bag recursively considering all ancestor variables as bound variables and finds a valuation for  $\mathcal{B}_t \setminus \text{anc}(t)$ . At the first visit to any bag, if the bound variables  $v_b^t$  do not produce an output in delay  $\tilde{O}(|D|^{\delta_t})$ , then we proceed to the next valuation

in the parent bag. However, if the enumeration for some  $v_b^t$  did produce output tuples but the procedure  $\text{next}_t(v_b^t)$  has finished, we proceed to the predecessor of the bag to fix the next valuation for variables in predecessor bag. In other words, the ancestor variables remain fixed and we enumerate the cartesian product of the remaining variables.

**Lemma 8.** *Algorithm 5 enumerates the answers with delay at most  $\tilde{O}(|D|^h)$  where  $h$  is the  $\delta$ -height of the decomposition tree. Moreover, it requires at most  $O(\log |D|)$  memory*

**PROOF.** Since the size of the decomposition is a constant, we require at most  $O(1)$  pointers for predecessors and  $O(1)$  pointers for storing the valuations of each free variable. Let  $n_\ell$  be the set of nodes at depth  $\ell$ . We will express the delay of the algorithm in terms of the delay of the subtrees of every node. The delay at the root  $t_b$  after checking whether valuation  $v_b$  is in the base relations is  $d_{t_b} = O(\sum_{t \in n_1} d_t)$ . This is because the enumeration of each subtree rooted at depth  $\ell = 1$  depends only on its ancestor variables and is thus independent of the other subtrees at that depth. Since each node in the tree can produce at most  $|D|^{\delta_\ell}$  valuations in  $\tilde{O}(|D|^{\delta_\ell})$  time, the recursive expansion of  $d_{t_b}$  gives  $\delta_{t_b} = O(\sum_{p \in \lambda(\mathcal{T})} \tilde{O}(|D|^{\sum_{t \in p} \delta_t}))$ . The largest term over all root to leaf paths is  $\tilde{O}(|D|^h)$  which gives us the desired delay guarantee.  $\square$

---

**Algorithm 5:** Query Answering using  $C$ -bound decomposition

---

**INPUT** : tree  $\mathcal{T}$ ,  $(\mathcal{T}, \mathcal{D})_{t \in V(\mathcal{T})}, v_b$

**OUTPUT**: query answer  $Q(D)$

---

```

1 Initialize  $t_{\text{visited}} \leftarrow 0$  for all nodes,  $v \leftarrow v_b, t \leftarrow$  left child of  $t_b$ 
2 Check if  $R_F(v_b) \neq \emptyset, F \in \mathcal{E}, F \subseteq C$ 
3 forall nodes in pre-order traversal starting from  $t$  do
4    $v \leftarrow \pi_{\mathcal{V}_F^t}^{\text{pred}}(v)$ 
5    $v_f^t \leftarrow \text{next}_t(\pi_{\mathcal{V}_F^t}(v))$ 
6   if  $v_f^t$  is empty and  $t_{\text{visited}} = 0$  then
7      $t \leftarrow \text{parent}(t)$ 
8     continue
9   end
10  if  $v_f^t$  is empty and  $t_{\text{visited}} = 1$  then
11     $t_{\text{visited}} \leftarrow 0$ 
12     $t \leftarrow \text{predecessor}(t)$ 
13    continue
14  end
15   $t_{\text{visited}} \leftarrow 1$ 
16   $v \leftarrow (v, v_f^t)$ 
17  if  $t$  is last node in the tree then
18    if  $R_F(v) \neq \emptyset, F \in \mathcal{E}$  then
19      emit  $v$ 
20    end
21    go to line 4 /* If  $t$  is last node in tree, find next
                valuation for  $\mathcal{V}_F^t$  */
22 end
23 end

```

---

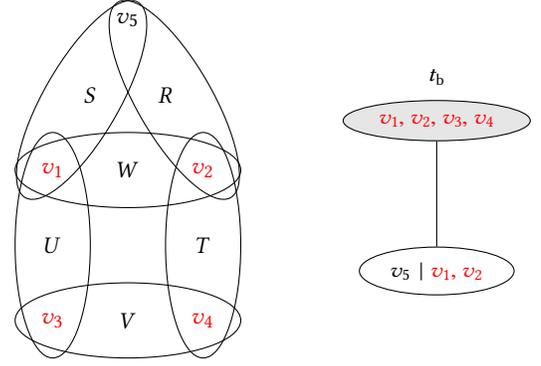
## E COMPARING WIDTH NOTIONS

We briefly discuss the connection of  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$  for a  $\mathcal{V}_b$ -connex decomposition with other related hypergraph notions.

The first observation is that the minimum edge cover number  $\rho^*$  is always an upper bound on  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$ . On the other hand, the  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b)$  is incomparable with  $\text{fhw}(\mathcal{H})$ . Indeed, Example 17 shows that  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) < \text{fhw}(\mathcal{H})$ , and the example below shows that the inverse situation can happen as well.

**Example 16.** *The query  $R(x, y), S(y, z)$  is acyclic and has  $\text{fhw}(\mathcal{H}) = 1$ . Let  $\mathcal{V}_b = \{x, z\}$ . The only valid  $\mathcal{V}_b$ -bound decomposition is the one with two bags,  $\{x, z\}, \{x, y, z\}$ , and hence  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) = 2$ . In this scenario,  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) > \text{fhw}(\mathcal{H})$ .*

**Example 17.** *Figure 5 shows an example hypergraph and a  $\mathcal{V}_b$ -bound tree decomposition (the variables in  $\mathcal{V}_b$  are colored red). For this example,  $\text{fhw}(\mathcal{H}) = 2$ , but  $\text{fhw}(\mathcal{H} \mid \mathcal{V}_b) = 3/2$ . Indeed, observe that we can cover the lower bag of the tree decomposition with a fractional edge cover of value only  $3/2$ .*



**Figure 5: Query hypergraph and corresponding  $C$ -bound tree decomposition with  $C = \{v_1, v_2, v_3, v_4\}$**