# Removing the Haystack to Find the Needle(s): Minesweeper, an Adaptive Join Algorithm

Hung Q. Ngo
Computer Science and Engineering
University at Buffalo, SUNY

Dung T. Nguyen
Computer Science and Engineering
University at Buffalo, SUNY

Christopher Ré
Computer Science
Stanford University

Atri Rudra
Computer Science and Engineering
University at Buffalo, SUNY

**Abstract**

We describe a new algorithm, Minesweeper, that is able to satisfy stronger runtime guarantees than previous join algorithms (colloquially, 'beyond worst-case guarantees') for data in indexed search trees. Our first contribution is developing a framework to measure this stronger notion of complexity, which we call *certificate complexity*, that extends notions of Barbay et al. and Demaine et al.; a certificate is a set of propositional formulae that certifies that the output is correct. This notion captures a natural class of join algorithms. In addition, the certificate allows us to define a strictly stronger notion of runtime complexity than traditional worst-case guarantees. Our second contribution is to develop a dichotomy theorem for the certificate-based notion of complexity. Roughly, we show that Minesweeper evaluates $\beta$-acyclic queries in time linear in the certificate plus the output size, while for any $\beta$-cyclic query there is some instance that takes superlinear time in the certificate (and for which the output is no larger than the certificate size). We also extend our certificate-complexity analysis to queries with bounded treewidth and the triangle query.

## 1 Introduction

Efficiently evaluating relational joins is one of the most well-studied problems in relational database theory and practice. Joins are a key component of problems in constraint satisfaction, artificial intelligence, motif finding, geometry, and others. This paper presents a new join algorithm, called Minesweeper, for joining relations that are stored in order data structures, such as B-trees. Under some mild technical assumptions, Minesweeper is able to achieve stronger runtime guarantees than previous join algorithms.

The Minesweeper algorithm is based on a simple idea. When data are stored in an index, successive tuples indicate *gaps*, i.e., regions in the output space of the join where no possible output tuples exist. Minesweeper maintains gaps that it discovers during execution and infers where to look next. In turn, these gaps may indicate that a large number of tuples in the base relations cannot contribute to the output of the join, so Minesweeper can efficiently skip over such tuples without reading them. By using an appropriate data structure to store the gaps, Minesweeper guarantees that we can find at least one point in the output space that needs to be explored, given the gaps so far. The key technical challenges are the design of this data structure, called the *constraint data structure*, and the analysis of the join algorithm under a more stringent runtime complexity measure.

To measure our stronger notion of runtime, we introduce the notion of a *certificate* for an instance of a join problem: essentially, a certificate is a set of comparisons between elements of the input relations that certify that the join output is exactly as claimed. We use the certificate as a measure of the difficulty of a particular instance of a join problem. That is, our goal is to find algorithms whose running times can be bounded by some function of the *smallest certificate size* for a particular input instance. Our notion has two key properties:

- *Certificate complexity captures the computation performed by widely implemented join algorithms.* We observe that the set of comparisons made by any join algorithm that interacts with the data by comparing elements of the

input relations (implicitly) constructs a certificate. Examples of such join algorithms are index-nested-loop join, sort-merge join, hash join,[1] grace join, and block-nested loop join. Hence, our results provide a lower bound for this class of algorithms, as any such algorithm must take at least as many steps as the number of comparisons in a smallest certificate for the instance.

- *Certificate complexity is a strictly finer notion of complexity than traditional worst-case data complexity*. In particular, we show that there is always a certificate that is no larger than the input size. In some cases, the certificate may be much smaller (even constant-sized for arbitrarily large inputs).

These two properties allow us to model a common situation in which indexes allow one to answer a query *without* reading all of the data—a notion that traditional worst-case analysis is too coarse to capture. We believe ours is the first *beyond worst-case analysis* of join queries.

Throughout, we assume that all input relations are indexed consistently with a particular ordering of all attributes called the *global attribute order* (GAO). In effect, this assumption means that we restrict ourselves to algorithms that compare elements in GAO order. This model, for example, excludes the possibility that a relation will be accessed using indexes with multiple search keys during query evaluation.

With this restriction, our main technical results are as follows. Given a $\beta$-acyclic query we show that there is some GAO such that Minesweeper runs in time that is essentially optimal in the certificate-sense, i.e., in time $\tilde{O}(|C| + Z)$, where $C$ is a smallest certificate for the problem instance, $Z$ is the output size, and $\tilde{O}$ hides factors that depend (perhaps exponentially) on the query size and at most logarithmically on the input size.[2] Assuming the 3SUM conjecture, this boundary is tight, in the sense that any $\beta$-cyclic query (and any GAO) there are some family of instances that require a run-time of $\Omega(|C|^{4/3-\epsilon}+Z)$ for any $\epsilon > 0$ where $Z = O(|C|)$. For $\alpha$-acyclic join queries, which are the more traditional notion of acyclicity in database theory and a strictly larger class than $\beta$-acyclic queries, Yannakakis's seminal join algorithm has a worst-case running time that is linear in the input size plus output size (in data complexity). However, we show that in the certificate world, this boundary has changed: assuming the exponential time hypothesis, the runtime of any algorithm for $\alpha$-acyclic queries cannot be bounded by any polynomial in $|C|$.[3]

We also describe how to extend our results to notions of treewidth. Recall that any ordering of attributes can be used to construct a tree decomposition. Given a GAO that induces a tree decomposition with an (induced) *treewidth* $w$, Minesweeper runs in time $\tilde{O}(|C|^{w+1} + Z)$. In particular, for a query with *treewidth* $w$, there is always a GAO that achieves $\tilde{O}(|C|^{w+1} + Z)$. Moreover, we show that no algorithm (comparison-based or not) can improve this exponent by more than a constant factor in $w$. However, our algorithm does not have an optimal exponent: for the special case of the popular triangle query, we introduce a more sophisticated data structure that allows us to run in time $\tilde{O}(|C|^{3/2} + Z)$, while Minesweeper runs in time $\tilde{O}(|C|^2 + Z)$.

**Outline of the Remaining Sections** In Section 2, we describe the notion of a certificate and formally state our main technical problem and results. In Section 3, we give an overview of the main technical ideas of Minesweeper, including a complete description of our algorithm and its associated data structures. In Section 4, we describe the analysis of Minesweeper for $\beta$-acyclic queries. In Section 5, we then describe how to extend the analysis to queries with low-treewidth and the triangle query. In Section 6, we discuss related work. Most of the technical details are provided in the appendix.

# 2 Problem and Main Result

Roughly, the main problem we study is:

> Given a natural join query $Q$ and a database instance $I$, compute $Q$ in time $f(|C|, Z)$, where $C$ is the smallest "certificate" that certifies that the output $Q(I)$ is as claimed by the algorithm and $Z = |Q(I)|$.

[1] Within a log-factor, an ordered tree can simulate a hash table.

[2] The exponential dependence on the query is similar to traditional data complexity; the logarithmic dependence on the data is an unavoidable technical necessity (see Appendix C).

[3] In Appendix J, we show that both worst-case optimal algorithms [40, 53] and Yannakakis's algorithm run in time $\omega(|C|)$ for $\beta$-acyclic queries on some family of instances.

We will assume that all relations in the input are already indexed. Ideally, we aim for $f(|C|,Z) = O(|C|+Z)$. We make this problem precise in this section.

## 2.1 The inputs to Minesweeper

We assume a set of attributes $A_1,\ldots,A_n$ and denote the domain of attribute $A_i$ as $\mathbf{D}(A_i)$. Throughout this paper, without loss of generality, we assume that all attributes are on domain $\mathbb{N}$. We define three items: (1) the global attribute order; (2) our notation for order; and (3) our model for how the data are indexed.

**The Global Attribute Order** Minesweeper evaluates a given natural join query $Q$ consisting of a set atoms$(Q)$ of relations indexed in a way that is consistent with an ordering $A_1,\ldots,A_n$ of all attributes occurring in $Q$ that we call the *global attribute order* (GAO). To avoid burdening the notation, we assume that the GAO is simply the order $A_1,\ldots,A_n$. We assume that all relations are stored in ordered search trees (e.g., B-trees) where the search key for this tree is consistent with this global order. For example, $(A_1,A_3)$ is consistent, while $(A_3,A_2)$ is not.

**Tuple-Order Notation** We will extensively reason about the relative order of tuples and describe notation to facilitate the arguments. For a relation $R(A_{s(1)},\ldots,A_{s(k)})$ where $s:[k]\to[n]$ is such that $s(i)<s(j)$ if $i<j$, we define an *index tuple* $\mathbf{x}=(x_1,\cdots,x_j)$ to be a tuple of positive integers, where $j\leqslant k$. Such tuples index tuples in the relation $R$. We define their meaning inductively. If $\mathbf{x}=(x_1)$, then $R[\mathbf{x}]$ denotes the $x_1$'th smallest value in the set $\pi_{A_{s(1)}}(R)$. Inductively, define $R[\mathbf{x}]$ to be the $x_j$'th smallest value in the set

$$R[x_1,\ldots,x_{j-1},*] := \pi_{A_j}\Big(\sigma_{A_{s(1)}=R[x_1],\cdots,A_{s(j-1)}=R[x_1,\ldots,x_{j-1}]}(R)\Big).$$

For example, if $R(A_1,A_2)=\{(1,1),(1,8),(2,3),(2,4)\}$ then $R[*]=\{1,2\}$, $R[1,*]=\{1,8\}$, $R[2]=2$, and $R[2,1]=3$.

We use the following convention to simplify the algorithm's description: for any index tuple $(x_1,\ldots,x_{j-1})$,

$$R[x_1,\ldots,x_{j-1},0] = -\infty \quad (1)$$
$$R[x_1,\ldots,x_{j-1},|R[x_1,\ldots,x_{j-1},*]|+1] = +\infty. \quad (2)$$

**Model of Indexes** The relation $R$ is indexed such that the values of various attributes of tuples from $R$ can be accessed using index tuples. We assume appropriate size information is stored so that we know what the correct ranges of the $x_j$'s are; for example, following the notation described above, the correct range is $1\leqslant x_j\leqslant |R[x_1,\ldots,x_{j-1},*]|$ for every $j\leqslant \text{arity}(R)$. With the convention specified in (1) and (2), $x_j=0$ and $x_j=|R[x_1,\ldots,x_{j-1},*]|+1$ are *out-of-range* coordinates. These coordinates are used for the sake of brevity only; an index tuple, by definition, cannot contain out-of-range coordinates.

The index structure for $R$ supports the query $R.\textsc{FindGap}(\mathbf{x},a)$, which takes as input an index tuple $\mathbf{x}=(x_1,\ldots,x_j)$ of length $0\leqslant j<k$ and a value $a\in\mathbb{Z}$, and returns a pair of coordinates $(x_-,x_+)$ such that

- $0\leqslant x_-\leqslant x_+\leqslant |R[(\mathbf{x},*)]|+1$
- $R[(\mathbf{x},x_-)]\leqslant a\leqslant R[(\mathbf{x},x_+)]$, and
- $x_-$ (resp. $x_+$) is the maximum (resp. minimum) index satisfying this condition.

Note that it is possible for $x_-=x_+$, which holds when $a\in R[(\mathbf{x},*)]$. Moreover, we assume throughout that FindGap runs in time $O(k\log|R|)$. This model captures widely used indexes including a B-tree [45, Ch.10] or a Trie [53].

## 2.2 Certificates

We define a *certificate*, which is a set of comparisons that certifies the output is exactly as claimed. We do not want the comparisons to depend on the specific values in the instance, only their order. To facilitate that, we think of $R[\mathbf{x}]$

as a variable that can be mapped to specific domain value by a database instance.[4] These variables are only defined for valid index tuples as imposed by the input instance described in the previous section.

A *database instance* $I$ instantiates all variables $R[\mathbf{x}]$, where $\mathbf{x} = (x_1, \ldots, x_j)$, $1 \leqslant j \leqslant \text{arity}(R)$, is an index tuple in relation $R$. (In particular, the input database instance described in the previous section is such a database instance.) We use $R^I[\mathbf{x}]$ to denote the instantiation of the variable $R[\mathbf{x}]$. Note that each such variable is on the domain of some attribute $A_k$; for short, we call such variable an $A_k$-*variable*. A database instance $I$ fills in specific values to the nodes of the search tree structures of the input relations.

**Example 2.1.** Consider the query $Q = R(A) \bowtie T(A,B)$ on the input instance $I(N)$ defined by $R^{I(N)} = [N]$ and $T^{I(N)} = \{(1,2i) \mid i \in [N]\} \cup \{(2,3i) \mid i \in [N]\}$. This instance can be viewed as defining the following variables: $R[i]$, $i \in [N]$, $T[1]$, $T[2]$, $T[1,i]$, and $T[2,i]$, $i \in [N]$. Another database instance $J$ can define the same index variables but using different constants, in particular, set $R^J[i] = \{2i \mid i \in [N]\}$, $T^J[1] = 2$, $T^J[2] = 4$, $T^J[1,i] = i$, and $T^J[2,i] = 10i$, $i \in [N]$.

We next formalize the notion of certificates. Consider an input instance to Minesweeper, consisting of the query $Q$, the GAO $A_1, \ldots, A_n$, and a set of relations $R \in \text{atoms}(Q)$ already indexed consistently with the GAO.

**Definition 2.2** (Argument)**.** An *argument* for the input instance is a set of symbolic comparisons of the form

$$R[\mathbf{x}] \;\theta\; S[\mathbf{y}], \quad \text{where } R, S \in \text{atoms}(Q) \tag{3}$$

and $\mathbf{x}$ and $\mathbf{y}$ are two index tuples[5] such that $R[\mathbf{x}]$ and $S[\mathbf{y}]$ are both $A_k$-variables for some $k \in [n]$, and $\theta \in \{<, =, >\}$. Note that we allow $R = S$.[6] A database instance $I$ *satisfies an argument* $\mathcal{A}$ if $R^I[\mathbf{x}] \;\theta\; S^I[\mathbf{y}]$ is true for every comparison $R[\mathbf{x}] \;\theta\; S[\mathbf{y}]$ in the argument $\mathcal{A}$.

An index tuple $\mathbf{x} = (x_1, \ldots, x_r)$ for a relation $S$ is called a *full index tuple* if $r = \text{arity}(S)$. Let $I$ be a database instance for the problem. Then, the full index tuple $\mathbf{x}$ is said to *contribute* to an output tuple $\mathbf{t} \in Q(I) = \bowtie_{R \in \text{atoms}(Q)} R^I$ if the tuple $(S[x_1], S[x_1, x_2], \ldots, S[\mathbf{x}])$ is exactly the projection of $\mathbf{t}$ onto attributes in $S$. A collection $X$ of full index tuples is said to be a *witness* for $Q(I)$ if $X$ has exactly one full index tuple from each relation $R \in \text{atoms}(Q)$, and all index tuples in $X$ contribute to the same $\mathbf{t} \in Q(I)$.

**Definition 2.3** (Certificate)**.** An argument $\mathcal{A}$ for the input instance is called a *certificate* iff the following condition is satisfied: if $I$ and $J$ are two database instances of the problem both of which satisfy $\mathcal{A}$, then *every* witness for $Q(I)$ is a witness for $Q(J)$ and vice versa. The *size* of a certificate is the number of comparisons in it.

**Example 2.4.** Continuing with Example 2.1. Fix an $N$, the argument $\{R[1] = T[1],\ R[2] = T[2]\}$ is a certificate for $I(N)$. For every database, such as $I = I(N)$ and $J$ in the example, that satisfies the two equalities, the set of witnesses are the same, i.e., the sets $\{1, (1,i)\}$ and $\{2, (2,i)\}$ for $i \in [N]$. Notice we do not need to spell out all of the outputs in the certificate.

Consider the instance $K$ in which $R^K = [N]$, $T^K = \{(1,2i) \mid i \in [N]\} \cup \{(3,3i) \mid i \in [N]\}$. While $K$ is very similar to $I$, $K$ does *not* satisfy the certificate since $R^K[2] \neq T^K[2]$. The certificate also does not apply to $I(N+1)$ from Example 2.1, since $I(N+1)$ defines a different set of variables from $I(N)$, e.g., $T[1, N+1]$ is defined in $I(N+1)$, but not in $I(N)$.

**Properties of optimal certificates** We list three important facts about $C$, a minimum-sized certificate:

(i) The set of comparisons issued by a very natural class of (non-deterministic) comparison-based join algorithms *is* a certificate; this result not only justifies the definition of certificates, but also shows that $|C|$ is a lowerbound for the runtime of any comparison-based join algorithm.

[4] We use variables as a perhaps more intuitive, succinct way to describe the underlying morphisms.

[5] Note again that the index tuples are constructed from the input instance as described in the previous section.

[6] Equality constraints between index tuples from same relation is *required* to guarantee that certificates are no longer than the input, see property (ii) below.

(ii) $|C|$ can be shown to be at most linear in the input size *no matter what the data and the GAO are*, and in many cases $|C|$ can even be of constant size. Hence, running time measured in $|C|$ is a strictly finer notion of runtime complexity than input-based runtimes; and

(iii) $|C|$ depends on the data and the GAO.

We explain the above facts more formally in the following two propositions. The proofs of the propositions can be found in Appendix B.

**Proposition 2.5** (Certificate size as run-time lowerbound of comparison-based algorithms)**.** *Let $Q$ be a join query whose input relations are already indexed consistent with a GAO as described in Section 2.1. Consider any comparison-based join algorithm that only does comparisons of the form shown in* (3)*. Then, the set of comparisons performed during execution of the algorithm* is *a certificate. In particular, if $C$ is an optimal certificate for the problem, then the algorithm must run in time at least $\Omega(|C|)$.*

**Proposition 2.6** (Upper bound on optimal certificate size)**.** *Let $Q$ be a general join query on $m$ relations and $n$ attributes. Let $N$ be the total number of tuples from all input relations. Then, no matter what the input data and the GAO are, we have $|C| \leqslant r \cdot N$, where $r = \max\{arity(R) \mid R \in \text{atoms}(Q)\} \leqslant n$.*

In Appendix B, we present examples to demonstrate that $|C|$ can vary any where from $O(1)$ to $\Theta(|\text{input-size}|)$, that the input data or the GAO can change the certificate size, and that same-relation comparisons are needed.

## 2.3 Main Results

Given a set of input relations already indexed consistent with a fixed GAO, we wish to compute the natural join of these relations as quickly as possible. As illustrated in the previous section, a runtime approaching $|C|$ is optimal among comparison-based algorithms. Furthermore, runtimes as a function of $|C|$ can be sublinear in the input size. Ideally, one would like a join algorithm running in $\tilde{O}(|C|)$-time. However, such a runtime is impossible because for many instances the output size $Z$ is *super*linear in the input size, while $|C|$ is at most linear in the input size. Hence, we will aim for runtimes of the form $\tilde{O}(g(|C|) + Z)$, where $Z$ is the output size and $g$ is some function; a runtime of $\tilde{O}(|C| + Z)$ is essentially optimal.

Our algorithm, called Minesweeper, is a general-purpose join algorithm. Our main results analyze its runtime behavior on various classes of queries in the certificate complexity model. Recall that $\alpha$-acyclic (often just acyclic) is the standard notion of (hypergraph) acyclicity in database theory [1, p. 128]. A query is $\beta$-acyclic, a stronger notion, if every subquery of $Q$ obtained by removing atoms from $Q$ remains $\alpha$-acyclic. For completeness, we include these definitions and examples in Appendix A.

Let $N$ be the input size, $n$ the number of attributes, $m$ the number of relations, $Z$ the output size, $r$ the maximum arity of input relations, and $C$ any optimal certificate for the instance. Our key results are as follows.

**Theorem 2.7.** *Suppose the input query is $\beta$-acyclic. Then there is some GAO such that Minesweeper computes its output in time $O\left(2^n m^2 n (4^r |C| + Z) \log N\right)$.*

As is standard in database theory, we ignore the dependency on the query size, and the above theorem states that Minesweeper runs in time $\tilde{O}(|C| + Z)$.[7]

What about *$\beta$-cyclic* queries? The short answer is *no*: we cannot achieve this guarantee. It is obvious that any join algorithm will take time $\Omega(Z)$. Using 3SUM*-hardness*, a well-known complexity-theoretic assumption [44], we are able to show the following.

**Proposition 2.8.** *Unless the* 3SUM *problem can be solved in sub-quadratic time, for any $\beta$-*cyclic *query $Q$ in any GAO, there does not exist an algorithm that runs in time $O(|C|^{4/3-\epsilon} + Z)$ for any $\epsilon > 0$ on all instances.*

We extend our analysis of Minesweeper to queries that have bounded treewidth and to triangle queries in Section 5. These results are technically involved and we only highlight the main technical challenges.

[7] For $\beta$-acyclic queries with a fixed GAO, our results are loose; our best upper bound the complexity uses the treewidth from Section 5.

# 3 The Minesweeper Algorithm

We begin with an overview of the main ideas and technical challenges of the Minesweeper algorithm. Intuitively, Minesweeper probes into the space of all possible output tuples, and explores the gaps in this space where there is no output tuples. These gaps are encoded by a technical notion called constraints, which we describe next. (For illustration, we present complete end-to-end results for set intersection and the *bow-tie* query in Appendix H and I.)

## 3.1 Notation for Minesweeper

We need some notation to describe our algorithm. Define the *output space* $\mathcal{O}$ of the query $Q$ to be the space $\mathcal{O} = \mathbf{D}(A_1) \times \mathbf{D}(A_2) \times \cdots \times \mathbf{D}(A_n)$, where $\mathbf{D}(A_i)$ is the domain of attribute $A_i$.[8] By definition, a tuple $\mathbf{t}$ is an *output tuple* if and only if $\mathbf{t} = (t_1, \ldots, t_n) \in \mathcal{O}$, and $\pi_{\bar{A}(R)}(\mathbf{t}) \in R$, for all $R \in \text{atoms}(Q)$, where $\bar{A}(R)$ is the set of attributes in $R$.

**Constraints** A *constraint* $\mathbf{c}$ is an $n$-dimensional vector of the following form: $\mathbf{c} = \langle c_1, \cdots, c_{i-1}, (\ell, r), \{*\}^{n-i} \rangle$, where $c_j \in \mathbb{N} \cup \{*\}$ for every $j \in [i-1]$. In other words, each constraint $\mathbf{c}$ is a vector consisting of three types of components:

(1) *open-interval* component $(\ell, r)$ on the attribute $A_i$ (for some $i \in [n]$) and $\ell, r \in \mathbb{N} \cup \{-\infty, +\infty\}$,

(2) *wildcard* or $*$ component, and

(3) *equality* component of the type $p \in \mathbb{N}$.

In any constraint, there is exactly one interval component. All components after the interval component are wildcards. Hence, we will often not write down the wildcard components that come after the interval component. The prefix that comes before the interval component is called a *pattern*, which consists of any number of wildcards and equality components. The equality components encode the coordinates of the axis parallel affine planes containing the gap. For example, in three dimensions the constraint $\langle *, (1, 10), * \rangle$ can be viewed as the region between the affine hyperplanes $A_2 = 1$ and $A_2 = 10$; and the constraint $\langle 1, *, (2, 5) \rangle$ can be viewed as the strip inside the plane $A_1 = 1$ between the line $A_3 = 2$ and $A_3 = 5$. We encode these gaps syntactically to facilitate efficient insertion, deletion, and merging.

Let $\mathbf{t} = (t_1, \ldots, t_n) \in \mathcal{O}$ be an arbitrary tuple from the output space, and $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$ be a constraint. Then, $\mathbf{t}$ is said to *satisfy* constraint $\mathbf{c}$ if for every $i \in [n]$ one of the following holds: (1) $c_i = *$, (2) $c_i \in \mathbb{N}$ and $t_i = c_i$, or (3) $c_i = (\ell, r)$ and $t_i \in (\ell, r)$. We say a tuple $\mathbf{t}$ is *active* with respect to a set of constraints if $\mathbf{t}$ does not satisfy any constraint in the set (Geometrically, no constraint covers the point $\mathbf{t}$).

## 3.2 A High-level Overview of Minesweeper

We break Minesweeper in two components: (1) a special data structure called the *constraint data structure* (CDS), and (2) an algorithm that uses this data structure. Algorithm 1 gives a high-level overview of how Minesweeper works, which we will make precise in the next section.

**The CDS stores the constraints already discovered during execution**. For example, consider the query $R(A, B), S(B)$. If Minesweeper determines that $S[4] = 20$ and $S[5] = 28$, then we can deduce that there is no tuple in the output that has a $B$ value in the open interval $(20, 28)$. This observation is encoded as a constraint $\langle *, (20, 28) \rangle$. A key challenge with the CDS is to efficiently find an active tuple $\mathbf{t}$, given a set of constraints already stored in the CDS.

**The outer algorithm queries the CDS to find active tuples and then probes the input relations.** If there is no active $\mathbf{t}$, the algorithm terminates. Given an active $\mathbf{t}$, Minesweeper makes queries into the index structures of the input relations. These queries either report that $\mathbf{t}$ is an output tuple, in which case $\mathbf{t}$ is output, or they discover constraints that are then inserted into the CDS. Intuitively, the queries into the index structures are crafted so that at least one of the constraints that is returned is responsible for ruling out $\mathbf{t}$ in any optimal certificate.

We first describe the interface of the CDS and then the outer algorithm which uses the CDS.

[8] Recall, we assume $\mathbf{D}(A_i) = \mathbb{N}$ for simplicity.

**Algorithm 1** High-level view: Minesweeper algorithm

CDS $\leftarrow \varnothing$ ▷ No gap discovered yet
**While** CDS can find $\mathbf{t}$ not in any stored gap **do**
  **If** $\pi_{\bar{A}(R)}(\mathbf{t}) \in R$ for every $R \in \text{atoms}(Q)$ **then**
    Report $\mathbf{t}$ and tell CDS that $\mathbf{t}$ is ruled out
  **else**
    Query all $R \in \text{atoms}(Q)$ for gaps around $\mathbf{t}$
    Insert those gaps into CDS

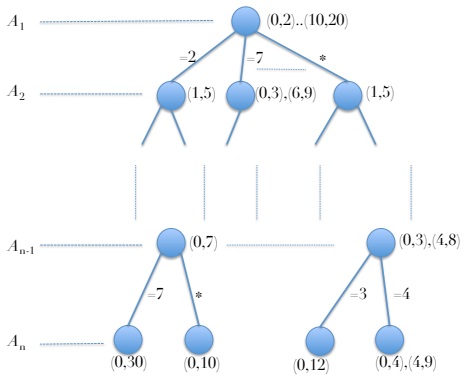



Figure 1: Example of ConstraintTree data structure

## 3.3 The CDS

The CDS is a data structure that implements two functions as efficiently as possible: (1) InsConstraint($\mathbf{c}$) takes a new constraint $\mathbf{c}$ and inserts it into the data structure, and (2) getProbePoint() returns an active tuple $\mathbf{t}$ with respect to all constraints that have been inserted into the CDS, or null if no such $\mathbf{t}$ exists.

**Implementation** To support these operations, we implement the CDS using a tree structure called ConstraintTree, which is a tree with at most $n$ levels, one for each of the attributes following the GAO. Figure 1 illustrates such a tree. More details are provided in Appendix E. Each node $v$ in the CDS corresponds to a prefix (i.e. pattern) of constraints; each node has two data structures:

**(1)** $v$.equalities is a sorted list with one entry per child of $v$ in the underlying tree. Each entry in the sorted list is labeled with an element of $\mathbb{N}$ and has a pointer to the subtree rooted at the corresponding child. There are two exceptions: (1) if $v$ is a leaf then $v$.equalities $= \varnothing$, and (2) each $v$ has at most one additional child node labeled with $*$.

**(2)** $v$.intervals is a sorted list of disjoint open intervals under that corresponding attribute. A key property is that *given a value $u$ we can, in logarithmic time, output the smallest value $u' \geqslant u$ that is not covered by any interval in $v$.intervals (via the* Next *function)*. We will maintain the invariant that, for every node $v$ in a ConstraintTree, none of the labels in $v$.equalities is contained in an interval in $v$.intervals.

The following lemma is straightforward hence we omit the proof. Note that when we insert a new interval that overlaps existing intervals and/or contains values in equalities, we will have to merge them and/or remove the entries in equalities; and hence the cost is amortized.

**Proposition 3.1.** *The operation* InsConstraint($\mathbf{c}$) *can be implemented in amortized time $O(n \log W)$, where $W$ is total number of constraint vectors already inserted.*

The key challenge is to design an efficient implementation of getProbePoint(); the heart of Sections 4 and 5 is to analyze getProbePoint() using properties of the query $Q$.

**Algorithm 2** Minesweeper for evaluating the query $Q = \bowtie_{R \in \text{atoms}(Q)} R(\bar{A}(R))$

**Input:** We use the conventions defined in (1) and (2)

Initialize the constraint data structure CDS $= \varnothing$
**While** $((\mathbf{t} \leftarrow \text{CDS.}\textsc{getProbePoint}()) \neq \text{NULL})$ **do**
  Denote $\mathbf{t} = (t_1, \ldots, t_n)$
  **For** each $R \in \text{atoms}(Q)$ **do**
    $k \leftarrow \text{arity}(R)$;
    Let $\bar{A}(R) = (A_{s(1)}, \ldots, A_{s(k)})$ be $R$'s attributes, where $s : [k] \to [n]$ is such that $s(i) < s(j)$ for $i < j$.
    **For** $p = 0$ **to** $k-1$ **do** ▷ Explore around $\mathbf{t}$ in $R$
      **For** each vector $\mathbf{v} \in \{\ell, h\}^p$ **do** ▷ $\ell, h$ are just symbols, and $\{\ell, h\}^0$ has only the empty vector
        Let $\mathbf{v} = (v_1, \ldots, v_p)$ ▷ $v_j \in \{\ell, h\}, \forall j \in [p]$
        $(i_R^{(\mathbf{v},\ell)}, i_R^{(\mathbf{v},h)}) \leftarrow R.\textsc{FindGap}\left(\left(i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}\right), t_{s(p+1)}\right)$ ▷ Gap around $(R[\mathbf{i}_R^{(\mathbf{v})}], t_{s(p+1)})$ in $R$.
    **If** $R\left[i_R^{(h)}, i_R^{(h,h)}, \ldots, i_R^{\{h\}^p}\right] = t_{s(p)}$ for all $p \in [\text{arity}(R)]$ and for all $R \in \text{atoms}(Q)$ **then**
      **Output** the tuple $\mathbf{t}$
      CDS.$\textsc{InsConstraint}(\langle t_1, t_2, \ldots, t_{n-1}, (t_n - 1, t_n + 1)\rangle)$
    **else**
      **For** each $R \in \text{atoms}(Q)$ **do**
        $k \leftarrow \text{arity}(R)$
        **For** $p = 0$ **to** $k-1$ **do**
          **For** each vector $\mathbf{v} \in \{\ell, h\}^p$ **do**
            **If** (all the indices $i_R^{(v_1)}, \ldots, i_R^{(v_1,\ldots,v_p)}$ are **not** out of range) **then**
              $\text{CDS.}\textsc{InsConstraint}\left(\left\langle R\left[i_R^{(v_1)}\right], \ldots, R\left[i_R^{(v_1)}, \cdots, i_R^{(v_1,\ldots,v_p)}\right], \left(R[i_R^{(\mathbf{v},\ell)}], R[i_R^{(\mathbf{v},h)}]\right)\right\rangle\right)$
              ▷ Note that the constraint is empty if $R[i_R^{(\mathbf{v},\ell)}] = R[i_R^{(\mathbf{v},h)}]$

## 3.4 The outer algorithm

Algorithm 2 contains all the details that were missing from the high-level view of Algorithm 1. Appendix D.1 has a complete run of Minesweeper on a specific query. Appendices H and I have the complete end-to-end descriptions of two specific queries, which help clarify the general algorithm. We prove the following result.

**Theorem 3.2.** *Let $N$ denote the input size, $Z$ the number of output tuples, $m = |\text{atoms}(Q)|$, and $r = \max_{R \in \text{atoms}(Q)} arity(R)$. Let $C$ be any optimal certificate for the input instance. Then, the total runtime of Algorithm 2 is*

$$O((4^r|C| + rZ)m\log(N)) + T(CDS),$$

*where $T(CDS)$ is the total time taken by the constraint data structure. The algorithm inserts $O(m4^r|C| + Z)$ constraints to CDS and issues $O(2^r|C| + Z)$ calls to* getProbePoint().

Our proof strategy bounds the number of iterations of the algorithm using an amortized analysis. We pay for each probe point $\mathbf{t}$ returned by the CDS by either charging a comparison in the certificate $C$ *or* by charging an output tuple. If $\mathbf{t}$ is an output tuple, we charge the output tuple. If $\mathbf{t}$ is not an output tuple, then we observe that at least one of the constraints we discovered must rule out $\mathbf{t}$. Recall that each constraint is essentially a pair of elements from some base relation. If one element from each such pair is not involved in any comparison in $C$, then we can perturb the instance slightly by moving the comparison-free element to align with $\mathbf{t}$. This means $C$ does not have enough information to rule out $\mathbf{t}$ as an output tuple, reaching a contradiction. Hence when $\mathbf{t}$ is not an output tuple, essentially some gap must map to a pair of comparisons. Finally, using the geometry of the gaps, we show that each comparison is charged at most $2^r$ times and each output tuple is charged $O(1)$ times. Thus, in total the number of iterations is $O(2^r|C| + Z)$.

When $C$ is an optimal-size certificate, the runtime above is about linear in $|C| + Z$ *plus* the total time the CDS takes. Note, however, that $|C|$ can be very small, even constant. Hence, we basically shift all of the burden of join evaluation to the CDS. Thus, one should not hope that there is an efficient CDS for general queries:

**Theorem 3.3** (Limitation of any CDS)**.** *Unless the exponential time hypothesis is wrong, no constraint data structure can process the constraints and the probe point accesses in time polynomial (independent of the query) in the number of constraints inserted and probe points accessed.*

Complete proofs of the above theorems are included in Appendix D. In the next sections, we analyze the CDS, specifically the function getProbePoint(). Our analysis exploits properties of the query and the GAO for $\beta$-acyclic and bounded treewidth queries.

# 4 $\beta$-acyclic queries

We describe how to implement getProbePoint for $\beta$-acyclic queries. In particular, we show that there is some GAO that helps implement getProbePoint in amortized logarithmic time. Hence, by Theorem 3.2 our running time is $\tilde{O}(|C| + Z)$, which we argued previously is essentially optimal.

## 4.1 Overview

Recall that given a set of intervals, getProbePoint returns an active tuple $\mathbf{t} = (t_1, \ldots, t_n) \in O$, i.e., a tuple $\mathbf{t}$ that does not satisfy any of the constraints stored in the CDS. Essentially, during execution there may be a large number of constraints, and getProbePoint needs to answer an alternating sequence of constraint satisfaction problems and insertions. The question is: how do we split this work between insertion time and querying time?

In Minesweeper, we take a lazy approach: we insert all the constraints without doing any cleanup on the CDS. Then, when getProbePoint is called Minesweeper might have to do hard work to return a new active tuple, applying memoization along the way so the heavy labor does not have to be repeated in the future. When the GAO has a special structure, this strategy helps keep every CDS operation at amortized logarithmic time. We first give an example to build intuition about how our lazy approach works.

**Example 4.1.** Consider a query with three attributes $(A, B, C)$, and suppose the constraints that are inserted into the CDS are

(i) $\langle a, b, (-\infty, 1)\rangle$ for all $a, b \in [N]$,

(ii) $\langle *, b, (2i-2, 2i)\rangle$ for all $b, i \in [N]$,

(iii) $\langle *, *, (2i-1, 2i+1)\rangle$ for $i \in [N]$,

(iv) and $\langle *, *, (2N, +\infty)\rangle$.

There are $O(N^2)$ constraints, and there is no active tuple of the form $(a, b, c)$ for $a, b \in [N]$. Without memoization, the brute-force strategy will take time $\Omega(N^3)$, because for every pair $(a, b) \in [N]^2$, the algorithm will have to verify in $\Omega(N)$ time that the constraints $(ii)$ forbid all $c = 2i-1, i \in [N]$, the constraints $(iii)$ forbid all $c = 2i, i \in [N]$, and the constraint $(iv)$ forbid $c > 2N$.

But we can do better by remembering inferences that we have made. Fix a value $a = 1, b = 1$. Minesweeper recognizes in $O(N)$-time that there is no $c$ for which $(a, b, c)$ is active. Minesweeper is slightly smarter: it looks at constraints of the type $(ii), (iii), (iv)$ (for $b = 1$) and concludes in $O(N)$-time that every tuple satisfying those constraints also satisfies the constraint $\langle *, 1, (0, +\infty)\rangle$. Minesweeper remembers this inference by inserting the inferred constraint into the CDS. Then, for $a \geqslant 2$, it takes only $O(1)$-time to conclude that no tuple of the form $(a, 1, c)$ can be active. It does this inference by inserting constraint $\langle a, 1, (0, +\infty)\rangle$, which is merged with $(i)$ to become $\langle a, 1, (-\infty, +\infty)\rangle$. Overall, we need only $O(N^2)$-time to reach the same conclusion as the $\Omega(N^3)$ brute-force strategy.

## 4.2 Patterns

Recall that getProbePoint returns a tuple $\mathbf{t} = (t_1, \ldots, t_n) \in O$ such that $\mathbf{t}$ does not satisfy any of the constraints stored in the CDS. We find $\mathbf{t}$ by computing $t_1, t_2, \ldots, t_n$, one value at a time, backtracking if necessary. We need some notation to describe the algorithm and the properties that we exploit.

Let $0 \leqslant k \leqslant n$ be an integer. A vector $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$ for which $p_i \in \mathbb{N} \cup \{*\}$ is called a *pattern*. The number $k$ is the *length* of the pattern. If $p_i \in \mathbb{N}$ then it is an *equality component* of the pattern, while $*$ is a *wildcard component* of the pattern.

A node $u$ at depth $k$ in the tree ConstraintTree can be identified by a pattern of length $i$ corresponding naturally to the labels on the path from the root of ConstraintTree down to node $u$. The pattern for node $u$ is denoted by $P(u)$. In particular, $P(\text{root}) = \epsilon$, the empty pattern.

Let $\mathbf{p} = \langle p_1, \ldots, p_k \rangle$ be a pattern. Then, a *specialization* of $\mathbf{p}$ is another pattern $\mathbf{p}' = \langle p'_1, \ldots, p'_k \rangle$ *of the same length* for which $p'_i = p_i$ whenever $p_i \in \mathbb{N}$. In other words, we can get a specialization of $\mathbf{p}$ by changing some of the $*$ components into equality components. If $\mathbf{p}'$ is a specialization of $\mathbf{p}$, then $\mathbf{p}$ is a *generalization* of $\mathbf{p}'$. For two nodes $u$ and $v$ of the CDS, if $P(u)$ is a specialization of $P(v)$, then we also say that node $u$ is a specialization of node $v$.

The specialization relation defines a partially ordered set. When $\mathbf{p}'$ is a specialization of $\mathbf{p}$, we write $\mathbf{p}' \preceq \mathbf{p}$. If in addition we know $\mathbf{p}' \neq \mathbf{p}$, then we write $\mathbf{p}' \prec \mathbf{p}$.

Let $G(t_1, \ldots, t_i)$ be the *principal filter* generated by $(t_1, \ldots, t_i)$ in this partial order, i.e., it is the set of all nodes $u$ of the CDS such that $P(u)$ is a generalization of $\langle t_1, \ldots, t_i \rangle$ and that $u.\text{intervals} \neq \varnothing$. The key property of constraints that we exploit is summarized by the following proposition.

**Proposition 4.2.** *Using the notation above, for a $\beta$-acyclic query, there exists a GAO such that for each $t_1, \ldots, t_i$ the principal filter $G(t_1, \ldots, t_i)$ is a chain.*

Recall that a chain is a totally ordered set. In particular, $G = G(t_1, \ldots, t_i)$ has a smallest pattern $\bar{\mathbf{p}}$ (or bottom pattern). Note that these patterns in $G$ might come from constraints inserted from relations, constraints inserted by the outputs of the join, or even constraints inserted due to backtracking. Thinking of the constraints geometrically, this condition means that the constraints form a collection of axis-aligned affine subspaces of $O$ where one is contained inside another.

In Appendix F, we prove Proposition 4.2 using a result of Brouwer and Kolen [15]. The class of GAOs in the proposition is called a *nested elimination order*. We show that there exists a GAO that is a nested elimination order if and only if the query is $\beta$-acyclic. We also show that $\beta$-acyclicity and this GAO can be found in polynomial time.

## 4.3 The getProbePoint Algorithm

Algorithm 3 describes getProbePoint algorithm specialized to $\beta$-acyclic queries. In turn, this algorithm uses Algorithm 4, which is responsible for efficiently inferring constraints imposed by patterns above this level. We walk through the steps of the algorithm below.

Initially, let $v$ be the root node of the CDS. We set $t_1 = v.\text{intervals.Next}(-1)$. This is the smallest value $t_1$ that does not belong to any interval stored in $v.\text{intervals}$. We work under the implicit assumption that any interval inserted into ConstraintTree that contains $-1$ must be of the form $(-\infty, r)$, for some $r \geqslant 0$. This is because the domain values are in $\mathbb{N}$. In particular, if $t_1 = +\infty$ then the constraints cover the entire output space $O$ and null can be returned.

Inductively, let $(t_1, \ldots, t_i)$, $i \geqslant 1$, be the *prefix* of $\mathbf{t}$ we have built thus far. Our goal is to compute $t_{i+1}$. What we need to find is a value $t_{i+1}$ such that $t_{i+1}$ does not belong to the intervals stored in $u.\text{intervals}$ *for every* node $u \in G(t_1, \ldots, t_i)$. For this, we call algorithm 4 that uses Prop. 4.2 to efficiently find $t_{i+1}$ or return that there is no such $t_{i+1}$. We defer its explanation for the moment. We only note that if such a $t_{i+1}$ cannot be found (i.e. if $t_{i+1} = +\infty$ is returned after the search), then we have to *backtrack* because what that means is that every tuple $\mathbf{t}$ that begins with the prefix $(t_1, \ldots, t_i)$ satisfies some constraint stored in ConstraintTree. Line 15 of Algorithm 3 shows how we backtrack. In particular, we save this information (by inserting a new constraint into the CDS) in Line 15 to avoid ever exploring this path again.

**Next Chain Value.** The key to Algorithm 4 is that such a $t_{i+1}$ can be found efficiently since one only needs to look through a chain of constraint sets. We write $\mathbf{p} \lessdot \mathbf{p}'$ if $\mathbf{p} \prec \mathbf{p}'$ and there is no pattern $\mathbf{p}''$ such that $\mathbf{p} \prec \mathbf{p}'' \prec \mathbf{p}'$. Every interval from a node $u \in G$ higher up in the chain infers an interval at a node lower in the chain. For instance, in Example 4.1, the chain $G$ consists of three nodes $\langle a, b \rangle$, $\langle *, b \rangle$, and $\langle *, * \rangle$. Further, every constraint of the form $\langle *, *, (2i-1, 2i+1) \rangle$ infers a more specialized constraint of the form $\langle *, b, (2i-1, 2i+1) \rangle$, which in turns infers a constraint of the form $\langle a, b, (2i-1, 2i+1) \rangle$. Hence, if we infer every single constraint downward from the top pattern to the bottom pattern, we will be spending a lot of time. The idea of Algorithm 4 is to infer as large of an interval

**Algorithm 3** CDS.GETPROBEPOINT() for $\beta$-acyclic queries

**Input:** A CONSTRAINTTREE CDS

1: $i \leftarrow 0$
2: **While** $i < n$ **do**
3: $\quad G \leftarrow \{u \in \text{CDS} \mid (t_1,\ldots,t_i) \preceq P(u) \text{ and } u.\text{INTERVALS} \neq \varnothing\}$
4: $\quad$ **If** $(G = \varnothing)$ **then**
5: $\quad\quad t_{i+1} \leftarrow -1$
6: $\quad\quad i \leftarrow i+1$
7: $\quad$ **else**
8: $\quad\quad$ Let $\bar{\mathbf{p}} = \langle \bar{p}_1,\ldots,\bar{p}_i\rangle$ be the *bottom* of $G$
9: $\quad\quad$ Let $\bar{u} \in$ CDS be the node for which $P(\bar{u}) = \bar{\mathbf{p}}$
10: $\quad\quad t_{i+1} \leftarrow \text{CDS.NEXTCHAINVAL}(-1, \bar{u}, G)$
11: $\quad\quad i_0 \leftarrow \max\{k \mid k \leqslant i, \bar{p}_k \neq *\}$
12: $\quad\quad$ **If** $(t_{i+1} = +\infty)$ and $i_0 = 0$ **then**
13: $\quad\quad\quad$ **Return** NULL $\quad\triangleright$ No tuple $\mathbf{t}$ found
14: $\quad\quad$ **else If** $(t_{i+1} = +\infty)$ **then**
15: $\quad\quad\quad \text{CDS.INSCONSTRAINT}(\langle \bar{p}_1,\ldots,\bar{p}_{i_0-1}, (\bar{p}_{i_0}-1, \bar{p}_{i_0}+1)\rangle)$
16: $\quad\quad\quad i \leftarrow i_0 - 1$ $\quad\triangleright$ Back-track
17: $\quad\quad$ **else**
18: $\quad\quad\quad i \leftarrow i+1$ $\quad\triangleright$ Advance $i$
19: **Return** $\mathbf{t} = (t_1,\ldots,t_n)$

as possible from a node higher in the chain before specializing it down. Our algorithm will ensure that whenever we infer a new constraint (line 13 of Algorithm 4), this constraint subsumes an old constraint which will never be charged again in a future inference.

## 4.4 Runtime Analysis

The proofs of the following main results are in Appendix F.

**Lemma 4.3.** *Suppose the input query $Q$ is $\beta$-acyclic. Then, there exists a GAO such that each of the two operations* GETPROBEPOINT *and* INSCONSTRAINT *of* CONSTRAINTTREE *takes amortized time $O(n2^n \log W)$, where $W$ is the total number of constraints ever inserted.*

The above lemma and Theorem 3.2 leads directly to one of our main results.

**Corollary 4.4** (Restatement of Theorem 2.7)**.** *Suppose the input query is $\beta$-acyclic then there exists a GAO such that Minesweeper computes its output in time*

$$O\Big(2^n m^2 n\,(4^r|C| + Z)\log N\Big).$$

*In particular, its data-complexity runtime is essentially optimal in the certificate complexity world: $\tilde{O}(|C| + Z)$.*

Beyond $\beta$-acyclic queries, we show that we cannot do better modulo a well-known complexity theoretic assumption.

**Proposition 4.5** (Re-statement of Proposition 2.8)**.** *Unless the* 3SUM *problem can be solved in sub-quadratic time, for any $\beta$-*cyclic *query $Q$ in any GAO, there does not exist an algorithm that runs in time $O(|C|^{4/3-\epsilon} + Z)$ for any $\epsilon > 0$ on all instances.*

**Algorithm 4** CDS.NEXTCHAINVAL$(x,u,G)$, where $G$ is a chain

**Input:** A CONSTRAINTTREE CDS, a node $u \in G$
**Input:** A chain $G$ of nodes, and a starting value $x$
**Output:** the smallest value $y \geqslant x$ not covered by *any* $v$.INTERVALS, for all $v \in G$ such that $P(u) \preceq P(v)$

```
1:  If there is no v ∈ G for which P(u) ≺ P(v) then                ▷ At the top of the chain G
2:      Return u.intervals.Next(x)
3:  else
4:      y ← x
5:      repeat
6:          Let v ∈ G such that P(u) ≺ P(v)
7:                                                                 ▷ Next node up the chain
8:          z ← CDS.nextChainVal(y, v, G)
9:                                                                 ▷ first "free value" ⩾ y at all nodes up the chain
10:         y ← u.intervals.Next(z)
11:                                                                ▷ first "free value" ⩾ z at u
12:     until y = z
13:     CDS.InsConstraint(⟨P(u), (x − 1, y)⟩)
14:     Return y
```

**Comparison with Worst-Case Optimal Algorithms** It is natural to wonder if Yannakakis' worst-case optimal algorithm for $\alpha$-acyclic queries or the worst-case optimal algorithms of [40] (henceforth, NPRR) or [53] (henceforth LFTJ) can achieve runtimes of $\tilde{O}(|C|+Z)$ for $\beta$-acyclic queries. We outline the intuition about why this cannot be the case.

Yannakakis' algorithm performs pairwise semijoin reducers. If we pick an instance where $|C| = o(N)$ such that there is a relation pair involved each with size $\Omega(N)$, then Yannakakis's algorithm will exceed the bound. For NPRR and LFTJ, consider the family of instances in which one computes all paths of length $\ell$ (some constant) in a directed graph $G = (V,E)$ (this can be realized by a "path" query of length $\ell$ where the relations are the edge set of $G$). Now consider the case where the longest path in $G$ has size at most $\ell - 1$. In this case the output is empty and since each relation is $E$, we have $|C| \leqslant O(|E|)$ and by Corollary 4.4, we will run in time $\tilde{O}(|E|)$. Hence, when $G$ has many paths (at least $\omega(|E|)$) of length at most $\ell$, then both NPRR and LFTJ will have to explore all $\omega(|E|)$ paths leading to an $\omega(|C|)$ runtime.

Appendix J presents a rich family of $\beta$-acyclic queries and a family of instances that combines both of the ideas above to show that all the three worst-case optimal algorithms can have arbitrarily worse runtime than Minesweeper. In particular, even running those worst-case algorithms in parallel is not able to achieve the certificate-based guarantees.

# 5 Extensions

We extend in two ways: queries with bounded tree width and we describe faster algorithms for the triangle query.

## 5.1 Queries with bounded tree-width

While Proposition 2.8 shows that $O(|C|^{4/3-\epsilon}+Z)$-time is not achievable for $\beta$-cyclic queries, we are able to show the following analog of the treewidth-based runtime under the traditional worst-case complexity notion [6, 19].

**Theorem 5.1** (Minesweeper for bounded treewidth queries)**.** *Suppose the GAO has* elimination width *bounded by w, Then, Minesweeper runs in time*

$$O\Big(m^3n^3 4^n\Big(nm^{w+1}8^{n(w+1)}|C|^{w+1}+Z\Big)\log N\Big).$$

*In particular, if we ignore the dependence on the query size, the runtime is $\tilde{O}\big(|C|^{w+1}+Z\big)$. Further, if the input query $Q$ has treewidth bounded by $w$, then there exists a GAO for which Minesweeper runs in the above time.*

The overall structure of the algorithm remains identical to the $\beta$-acyclic case, the only change is in getProbePoint. The getProbePoint algorithm for general queries remains very similar in structure to that of the $\beta$-acyclic case (Algorithm 3), and if the input query is $\beta$-acyclic (with a nested elimination order as the GAO), then the general getProbePoint algorithm is *exactly* Algorithm 3. The new issue we have to deal with is the fact that the poset $G$ at each depth is not necessarily a chain. Our solution is simple: we mimic the behavior of Algorithm 3 on a shadow of $G$ that *is* a chain and make use of both the algorithm and the analysis for the $\beta$-acyclic case. Appendix G contains all the algorithm details, and the proofs of the above theorem, along with the following negative result.

It is natural to wonder if Theorem 5.1 is tight. In addition to the obvious $\Omega(Z)$ dependency, the next result indicates that the dependence on $w$ also cannot be avoided, *even if* we just look at the class of $\alpha$-acyclic queries.

**Proposition 5.2.** *Unless the* exponential time hypothesis *is false, for every large enough constant $k > 0$, there is an $\alpha$-acyclic query $Q_k$ for which there is no algorithm with runtime $|C|^{o(k)}$. Further, $Q_k$ has treewidth $k-1$.*

Our analysis of Minesweeper is off by at most 1 in the exponent.

**Proposition 5.3.** *For every $w \geqslant 2$, there exists an ($\alpha$-acyclic) query $Q_w$ with treewidth $w$ with the following property. For every possible global ordering of attributes, there exists an (infinite family of) instance on which the Minesweeper algorithm takes $\Omega(|C|^w)$ time.*

## 5.2 The Triangle Query

We consider the triangle query $Q_\triangle = R(A,B) \bowtie S(B,C) \bowtie T(A,C)$ that can be viewed as enumerating triangles in a given graph. Using the CDS described so far, Minesweeper computes this query in time $\tilde{O}(|C|^2+Z)$, and this analysis is tight.[9] The central inefficiency is that the CDS wastes time determining that many tuples with the same prefix $(a,b)$ have been ruled out by existing constraints. In particular, the CDS considers all possible pairs $(a,b)$ (of which there can be $\Omega(|C|^2)$ of them). By designing a smarter CDS, our improved CDS explores $O(|C|)$ such pairs. We can prove the following result. (The details are in Appendix L.)

**Theorem 5.4.** *We can solve the triangle query, $Q_\Delta$ in time*
$O\Big(\big(|C|^{3/2}+Z\big)\log^{7/2} N\Big)$.

# 6 Related Work

Our work touches on a few different areas, and we structure the related work around each of these areas: join processing, certificates for set intersection, and complexity measures that are finer than worst-case complexity.

## 6.1 Join Processing

Many positive and negative results regarding conjunctive query evaluation also apply to natural join evaluation. On the negative side, both problems are NP-hard in terms of expression complexity [16], but are easier in terms of data complexity [50] (when the query is assumed to be of fixed size). They are W[1]-complete and thus unlikely to be fix-parameter tractable [33, 43].

On the positive side, a large class of conjunctive queries (and thus natural join queries) are tractable. In particular, the classes of acyclic queries and bounded treewidth queries can be evaluated efficiently [17, 27, 31, 54, 55]. For example, if $|q|$ is the query size, $N$ is the input size, and $Z$ is the output size, then Yannakakis' algorithm can evaluate acyclic natural join queries in time $\tilde{O}(\text{poly}(|q|)(N\log N+Z))$. Acyclic conjunctive queries can also be evaluated efficiently in the I/O model [42], and in the RAM model even when there are inequalities [54]. For queries with treewidth $w$, it was recognized early on that a runtime of about $\tilde{O}(N^{w+1}+Z)$ is attainable [19, 28]. our result strictly

[9] A straightforward application of our more general analysis given in Theorem 5.1, which gives $\tilde{O}(|C|^3+Z)$.

generalizes these results. In Appendix J, we show that Yannakakis' algorithm does not meet our notion of certificate optimality.

The notion of treewidth is loose for some queries. For instance, if we replicate each attribute $x$ times for every attribute, then the treewidth is inflated by a factor of $x$; but by considering all duplicate attributes as one big compound attribute the runtime should only be multiplied by a polynomial in $x$ and there should not be a factor of $x$ in the exponent of the runtime. Furthermore, there is an inherent incompatibility between treewidth and acyclicity: an acyclic query can have very large treewidth, yet is still tractable. A series of papers [2, 17, 27, 31, 32] refined the treewidth notion leading to generalized hyper treewidth [31] and ultimately *fractional hypertree width* [39], which allows for a unified view of tractable queries. (An acyclic query, for example, has fractional hypertree width at most 1.)

The fractional hypertree width notion comes out of a recent tight worst-case output size bound in terms of the input relation sizes [7]. An algorithm was presented that runs in time matching the bound, and thus it is worst-case optimal in [40]. Given a tree decomposition of the input query with the minimum fractional edge cover over all bags, we can run this algorithm on each bag, and then Yannakakis algorithm [55] on the resulting bag relations, obtaining a total runtime of $\tilde{O}(N^{w^*} + Z)$, where $w^*$ is the fractional hyper treewidth. The *leap-frog triejoin* algorithm [53] is also worst-case optimal and runs fast in practice; it is based on the idea that we can efficiently skip unmatched intervals. The indices are also built or selected to be consistent with a chosen GAO. In the Appendix J, we show that neither Leapfrog nor the algorithm from [40] can achieve the certificate guarantees of Minesweeper for $\beta$-acyclic queries.

**Notions of acyclicity** There are at least five notions of acyclic hypergraphs, four of which were introduced early on in database theory (see e.g, [24]), and at least one new one introduced recently [22]. The five notions are *not* equivalent, but they form a strict hierarchy in the following way:

$$\text{Berge-acyclicity} \subsetneq \gamma\text{-acyclicity} \subsetneq \text{jtdb} \subsetneq \beta\text{-acyclicity} \subsetneq \alpha\text{-acyclicity}$$

Acyclicity or $\alpha$-acyclicity [11, 12, 26, 29, 38] was recognized early on to be a very desirable property of data base schemes; in particular, it allows for a data-complexity optimal algorithm in the worst case [55]. However, an $\alpha$-acyclic hypergraph may have a sub-hypergraph that is not $\alpha$-acyclic. For example, if we take *any* hypergraph and add a hyperedge containing all vertices, we obtain an $\alpha$-acyclic hypergraph. This observation leads to the notion of $\beta$-acyclicity: a hypergraph is $\beta$-acyclic if and only if every one of its sub-hypergraph is ($\alpha$-) acyclic [24]. It was shown (relatively) recently [41] that SAT is in P for $\beta$-acyclic CNF formulas and is NP-complete for $\alpha$-acyclic CNF formulas. Extending the result, it was shown that negative conjunctive queries are poly-time solvable if and only if it is $\beta$-acyclic [14]. The separation between $\gamma$-acyclicity and $\beta$-acyclicity showed up in logic [21], while Berge-acyclicity is restrictive and, thus far, is of only historical interest [13].

**Graph triangle enumeration** In social network analysis, computing and listing the number of triangles in a graph is at the heart of the clustering coefficients and transitivity ratio. There are four decades of research on computing, estimating, bounding, and lowerbounding the number of triangles and the runtime for such algorithms [5, 36, 37, 49, 51, 52]. This problem can easily be reduced to a join query of the form $Q = R(A,B) \bowtie S(B,C) \bowtie T(A,C)$.

## 6.2 Certificates for Intersection

The problem of finding the union and intersection of two sorted arrays using the fewest number of comparisons is well-studied, dated back to at least Hwang and Lin [35] since 1972. In fact, the idea of skipping elements using a binary-search jumping (or leap-frogging) strategy was already present in [35]. Demaine et al. [20] used the leap-frogging strategy for computing the intersection of $k$ sorted sets. They introduced the notion of proofs to capture the intrinsic complexity of such a problem. Then, the idea of gaps and certificate encoding were introduced to show that their algorithm is average case optimal. (See Appendix K for a more technical discussion.)

DLM's notion of proof inspired another adaptive complexity notion for the set intersection problem called partition certificate by Barbay and Kenyon in [8, 9], where instead of a system of inequalities essentially a set of gaps is used to encode and verify the output. Barbay and Kenyon's idea of a partition certificate is very close to the set of intervals that Minesweeper outputs. In the analysis of Minesweeper in Appendix H for the set intersection problem, we (implicitly) show a correspondence between these partition certificates and DLM's style proofs. In addition to the fact that join

queries are more general than set intersection, our notion of certificate is value-oblivious; our certificates do not depend on specific values in the domain, while Barbay-Kenyon's partition certificate does.

It should be noted that these lines of inquiries are not only of theoretical interest. They have yielded good experimental results in text-datamining and text-compression [10].[10]

### 6.3 Beyond Worst-case Complexity

There is a fairly large body of work on analyzing algorithms with more refined measures than worst-case complexity. (See, e.g., the excellent lectures by Roughgarden on this topic [46].) This section recalls the related works that are most closely related to ours.

A fair amount of work has been done in designing *adaptive* algorithms for sorting [23], where the goal is to design a sorting algorithm whose runtime (or the number of comparisons) matches a notion of difficulty of the instance (e.g. the number of inversions, the length of longest monotone subsequence and so on – the survey [23] lists at least eleven such measures of *disorder*). This line of work is similar to ours in the sense that the goal is to run in time proportional to the difficulty of the input. The major difference is that in these lines of work the main goal is to avoid the logarithmic factor over the linear runtime whereas in our work, our potential gains are of much higher order and we ignore log-factors.

Another related line of work is on self-improving algorithms of Ailon et al. [4], where the goal is to have an algorithm that runs on inputs that are drawn i.i.d. from an *unknown* distribution and in expectation converge to a runtime that is related to the entropy of the distribution. In some sense this setup is similar to online learning while our work requires worst-case per-instance guarantees.

The notion of instance optimal join algorithms was (to the best of our knowledge) first explicitly studied in the work of Fagin et al. [25]. The paper studies the problem of computing the top-$k$ objects, where the ranking is some aggregate of total ordering of objects according to different attributes. (It is assumed that the algorithm can only iterate through the list in sorted order of individual attribute scores.) The results in this paper are stronger than ours since Fagin et al. give $O(1)$-optimality ratio (as opposed to our $O(\log N)$-optimality ratio). On the other hand the results in the Fagin et al. paper are for a problem that is arguably narrower than the class we consider of join algorithms.

The only other paper with provable instance-optimal guarantees that we are aware of is the Afshani et al. results on some geometric problems [3]. Their quantitative results are somewhat incomparable to ours. On the one hand their results get a constant optimality ratio: on the other hand, the optimality ratio is only true for *order oblivious* comparison algorithms (while our results with $O(\log N)$ optimality ratio hold against all comparison-based algorithms).

## 7 Conclusion and Future Work

We described the Minesweeper algorithm for processing join queries on data that is stored ordered in data structures modeling traditional relational databases. We showed that Minesweeper can achieve stronger runtime guarantees than previous algorithms; in particular, we believe Minesweeper is the first algorithm to offer beyond worst-case guarantees for joins. Our analysis is based on a notion of certificates, which provide a uniform measure of the difficulty of the problem that is independent of any algorithm. In particular, certificates are able to capture what we argue is a natural class of comparison-based join algorithms.

Our main technical result is that, for $\beta$-acyclic queries there is some GAO such that Minesweeper runs in time that is linear in the certificate size. Thus, Minesweeper is optimal (up to an $O(\log N)$ factor) among comparison-based algorithms. Moreover, the class of $\beta$-acyclic queries is the boundary of complexity in that we show no algorithm for *$\beta$-cyclic* queries runs in time linear in the certificate size. And so, we are able to completely characterize those queries that run in linear time for the certificate and hence are optimal in a strong sense. Conceptually, certificates change the complexity landscape for join processing as the analogous boundary for traditional worst-case complexity are $\alpha$-acyclic queries, for which we show that there is no polynomial bound in the certificate size (assuming the strong form of the exponential time hypothesis). We then considered how to extend our results using treewidth. We showed

[10] We thank Jérémy Barbay for bringing these references to our attention.

that our same Minesweeper algorithm obtains $\tilde{O}(|C|^{w+1} + Z)$ runtime for queries with treewidth $w$. For the triangle query (with treewidth 2), we presented a modified algorithm that runs in time $\tilde{O}(|C|^{3/2} + Z)$.

**Future Work** We are excited by the notion of certificate-based complexity for join algorithms; we see it as contributing to an emerging push beyond worst-case analysis in theoretical computer science. We hope there is future work in several directions for joins and certificate-based complexity.

**Indexing and Certificates** The interplay between indexing and certificates may provide fertile ground for further research. For example, the certificate size depends on the order of attributes. In particular, a certificate in one order may be smaller than in another order. We do not yet have a handle on how the certificate-size changes for the same data in different orders. Ideally, one would know the smallest certificate size for any query and process in that order. Moreover, we do not know how to use of multiple access paths (eg. Btrees with different search keys) in either the analysis or the algorithm. These indexes may result in dramatically faster algorithms and new types of query optimization.

**Fractional Covers** A second direction is that join processing has seen a slew of powerful techniques based on increasingly sophisticated notions of covers and decompositions for queries. We expect that such covers (hypergraph, fractional hypergraph, etc.) could be used to tighten and improve our bounds. For the triangle query, we have the fractional cover bound, i.e., $\tilde{O}(|C|^{3/2})$. But is this possible for all queries?

# Acknowledgments

We thank LogicBlox, Mahmoud Abo Khamis, Semih Salihoglu and Dan Suciu for many helpful conversations.

HQN's work is partly supported by NSF grant CCF-1319402 and a gift from Logicblox. CR's work on this project is generously supported by NSF CAREER Award under No. IIS-1353606, NSF award under No. CCF-1356918, the ONR under awards No. N000141210041 and No. N000141310129, Sloan Research Fellowship, Oracle, and Google. AR's work is partly supported by NSF CAREER Award CCF-0844796, NSF grant CCF-1319402 and a gift from Logicblox.

# References

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, 1995.

[2] I. Adler, G. Gottlob, and M. Grohe, *Hypertree width and related hypergraph invariants*, European J. Combin., 28 (2007).

[3] P. Afshani, J. Barbay, and T. M. Chan, *Instance-optimal geometric algorithms*, in FOCS, 2009, pp. 129–138.

[4] N. Ailon, B. Chazelle, K. L. Clarkson, D. Liu, W. Mulzer, and C. Seshadhri, *Self-improving algorithms*, SIAM J. Comput., 40 (2011), pp. 350–375.

[5] N. Alon, *On the number of subgraphs of prescribed type of graphs with a given number of edges*, Israel J. Math., 38 (1981).

[6] S. Arnborg and A. Proskurowski, *Linear time algorithms for NP-hard problems restricted to partial k-trees*, Discrete Appl. Math., 23 (1989), pp. 11–24.

[7] A. Atserias, M. Grohe, and D. Marx, *Size bounds and query plans for relational joins*, 2008, pp. 739–748.

[8] J. Barbay and C. Kenyon, *Adaptive intersection and t-threshold problems*, in SODA, 2002, pp. 390–399.

[9] ——, *Alternation and redundancy analysis of the intersection problem*, ACM Transactions on Algorithms, 4 (2008).

[10] J. Barbay and A. López-Ortiz, *Efficient algorithms for context query evaluation over a tagged corpus*, in SCCC, M. Arenas and B. Bustos, eds., IEEE Computer Society, 2009, pp. 11–17.

[11] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis, *Properties of acyclic database schemes*, in STOC, New York, NY, USA, 1981, ACM, pp. 355–362.

[12] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, *On the desirability of acyclic database schemes*, J. ACM, 30 (1983), pp. 479–513.

[13] C. Berge, *Graphs and Hypergraphs*, Elsevier Science Ltd, 1985.

[14] J. Brault-Baron, *A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic*, in CSL 12, vol. 16, 2012, pp. 137–151.

[15] A. Brouwer and A. Kolen, *A super-balanced hypergraph has a nest point*, (1980). Tech. Report.

[16] A. K. Chandra and P. M. Merlin, *Optimal implementation of conjunctive queries in relational data bases*, in STOC, 1977.

[17] C. Chekuri and A. Rajaraman, *Conjunctive query containment revisited*, Theor. Comput. Sci., 239 (2000), pp. 211–229.

[18] J. Chen, S. Lu, S.-H. Sze, and F. Zhang, *Improved algorithms for path, matching, and packing problems*, in SODA, 2007, pp. 298–307.

[19] R. Dechter and J. Pearl, *Tree clustering for constraint networks.*, Artificial Intelligence, 38 (1989), pp. 353–366.

[20] E. D. Demaine, A. López-Ortiz, and J. I. Munro, *Adaptive set intersections, unions, and differences*, in SODA, 2000, pp. 743–752.

[21] D. Duris, *Hypergraph acyclicity and extension preservation theorems*, in LICS, 2008, pp. 418–427.

[22] ——, *Some characterizations of $\gamma$ and $\beta$-acyclicity of hypergraphs.*, Information Processing Letters, 112 (2012).

[23] V. Estivill-Castro and D. Wood, *A survey of adaptive sorting algorithms*, ACM Comput. Surv., 24 (1992), pp. 441–476.

[24] R. Fagin, *Degrees of acyclicity for hypergraphs and relational database schemes*, J. ACM, 30 (1983), pp. 514–550.

[25] R. Fagin, A. Lotem, and M. Naor, *Optimal aggregation algorithms for middleware*, J. Comput. Syst. Sci., 66 (2003), pp. 614–656.

[26] R. Fagin, A. O. Mendelzon, and J. D. Ullman, *A simplied universal relation assumption and its properties*, TODS, 7 (1982).

[27] J. Flum, M. Frick, and M. Grohe, *Query evaluation via tree-decompositions*, J. ACM, 49 (2002), pp. 716–752.

[28] E. C. Freuder, *Complexity of k-tree structured constraint satisfaction problems*, in AAAI, AAAI'90, AAAI Press, 1990, pp. 4–9.

[29] N. Goodman and O. Shmueli, *Tree queries: a simple class of relational queries*, ACM Trans. Database Syst., 7 (1982).

[30] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello, *Hypertree decompositions: Structure, algorithms, and applications*, in WG, D. Kratsch, ed., vol. 3787 of LNCS, Springer, 2005.

[31] G. Gottlob, N. Leone, and F. Scarcello, *Hypertree decompositions and tractable queries*, J. Comput. Syst. Sci., 64 (2002), pp. 579–627.

[32] G. Gottlob, Z. Miklós, and T. Schwentick, *Generalized hypertree decompositions: Np-hardness and tractable variants*, J. ACM, 56 (2009), pp. 30:1–30:32.

[33] M. Grohe, *The parameterized complexity of database queries*, in PODS, 2001, pp. 82–92.

[34] P. Heggernes and B. W. Peyton, *Fast computation of minimal fill inside a given elimination ordering*, SIAM J. Matrix Anal. Appl., 30 (2008), pp. 1424–1444.

[35] F. K. Hwang and S. Lin, *A simple algorithm for merging two disjoint linearly ordered sets*, SIAM J. Comput., 1 (1972), pp. 31–39.

[36] A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM J. Comput., 7 (1978), pp. 413–423.

[37] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, *Efficient triangle counting in large graphs via degree-based vertex partitioning*, Internet Mathematics, 8 (2012), pp. 161–185.

[38] D. Maier and J. D. Ullman, *Connections in acyclic hypergraphs: extended abstract*, in PODS, ACM, 1982, pp. 34–39.

[39] D. Marx, *Approximating fractional hypertree width*, ACM Transactions on Algorithms, 6 (2010).

[40] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, *Worst-case optimal join algorithms: [extended abstract]*, in PODS, 2012, pp. 37–48.

[41] S. Ordyniak, D. Paulusma, and S. Szeider, *Satisfiability of Acyclic and Almost Acyclic CNF Formulas*, in FSTTCS 2010, vol. 8, 2010.

[42] A. Pagh and R. Pagh, *Scalable computation of acyclic joins*, in PODS, 2006, pp. 225–232.

[43] C. H. Papadimitriou and M. Yannakakis, *On the complexity of database queries*, in PODS, 1997, pp. 12–19.

[44] M. Pătraşcu, *Towards polynomial lower bounds for dynamic problems*, in STOC, 2010, pp. 603–610.

[45] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, McGraw-Hill, Inc., New York, NY, USA, 3 ed., 2003.

[46] T. Roughgarden, *Lecture notes for CS369N "beyond worst-case analysis"*. http://theory.stanford.edu/ tim/f09/f09.html, 2009.

[47] ——, *Problem set #1 (CS369N: Beyond worst-case analysis)*. http://theory.stanford.edu/ tim/f11/hw1.pdf, 2011.

[48] W. Schafhauser, *New Heuristic Methods for Tree Decompositions and Generalized Hypertree Decompositions*, Master's thesis, 2006.

[49] S. Suri and S. Vassilvitskii, *Counting triangles and the curse of the last reducer*, in WWW, 2011, pp. 607–614.

[50] M. Y. Vardi, *The complexity of relational query languages (extended abstract)*, in STOC, 1982, pp. 137–146.

[51] V. Vassilevska and R. Williams, *Finding a maximum weight triangle in $n^{3-\delta}$ time, with applications*, in STOC, 2006, pp. 225–231.

[52] V. Vassilevska and R. Williams, *Finding, minimizing, and counting weighted subgraphs*, in STOC, ACM, 2009, pp. 455–464.

[53] T. L. Veldhuizen, *Leapfrog triejoin: a worst-case optimal join algorithm*, ICDT, (2014). To Appear.

[54] D. E. Willard, *An algorithm for handling many relational calculus queries efficiently*, J. Comput. Syst. Sci., 65 (2002), pp. 295–331.

[55] M. Yannakakis, *Algorithms for acyclic database schemes*, in VLDB, 1981, pp. 82–94.

# A The GAO and query's structure

The input query $Q$ can be represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of all attributes, and $\mathcal{E}$ is the collection of input relations' attribute sets. Any global attribute order (GAO) is just a permutation of vertices of $\mathcal{V}$. In the logic, constraint satisfaction, and databases [48], graphical models, and sparse matrix computation [34] literature, any permutation of vertices of a hypergraph is called an *elimination order*, which can be used to characterize many important properties of the hypergraph.

Our algorithm is no different: its performance intimately relates to properties of the GAO, which characterizes structural properties of the hypergraph $\mathcal{H}$. In this section we state some relevant known results and derive two slightly new results regarding the relationship between elimination orders and notions of widths and acyclicity of hypergraphs.

## A.1 Basic concepts

There are many definitions of acyclic hypergraphs. A hypergraph $(\mathcal{V}, \mathcal{E})$ is $\alpha$-acyclic if the GYO procedure returns empty [1, p. 128]. Essentially, in GYO one iterates two steps: (1) remove any edge that is empty or contained in another hyperedge, or (2) remove vertices that appear in at most one hyperedge. If the result is empty, then the hypergraph is $\alpha$-acyclic. A query is $\beta$-acyclic if the graph formed by any subset of hyperedges is $\alpha$-acyclic. Thus, the requirement that a hypergraph be $\beta$-acyclic is (strictly) stronger than $\alpha$-acyclic. We illustrate this with an example.

**Example A.1.** We map freely between hypergraphs and queries. The query $Q_\Delta = R(A,B) \bowtie S(A,C) \bowtie T(B,C)$ is both $\alpha$-cyclic and $\beta$-cyclic. However, if one adds the relation $U(A,B,C)$ to form $Q_{\Delta+U} = R(A,B) \bowtie S(A,C) \bowtie T(B,C) \bowtie U(A,B,C)$ this query is $\alpha$-acyclic, but it is still $\beta$-cyclic.

We can also define these concepts via a notion of tree decomposition.

**Definition A.2** (Tree decomposition)**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. A *tree-decomposition* of $\mathcal{H}$ is a pair $(T, \chi)$ where $T = (V(T), E(T))$ is a tree and $\chi : V(T) \to 2^{\mathcal{V}}$ assigns to each node of the tree $T$ a set of vertices of $\mathcal{H}$. The sets $\chi(t)$, $t \in V(T)$, are called the *bags* of the tree-decomposition. There are two properties the bags must satisfy

(a) For every hyperedge $F \in \mathcal{E}$, there is a bag $\chi(T)$ such that $F \subseteq \chi(t)$.

(b) For every vertex $v \in \mathcal{V}$, the set $\{t \mid t \in T, v \in \chi(t)\}$ is not empty and forms a connected subtree of $T$.

There are at least five notions of acyclic hypergraphs, four of which were introduced very early on (see e.g, [24]), and at least one new one introduced recently [22]. The five notions are *not* equivalent, but they form a strict hierarchy as discussed in Section 6. Of interest to us in this paper are $\beta$-acyclicity and acyclicity.

**Definition A.3** (Acyclicity)**.** A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is *$\alpha$-acyclic* or just *acyclic* if and only if there exists a tree decomposition $(T = (V(T), E(T)), \{\chi(t) \mid t \in V(T)\})$ in which every bag $\chi(t)$ is a hyperedge of $\mathcal{H}$. When $\mathcal{H}$ represents a query $Q$, the tree $T$ is also called the *join tree* of the query. A query is acyclic if and only if its hypergraph is acyclic.

**Definition A.4** ($\beta$-acyclicity)**.** A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is *$\beta$-acyclic* if and only if there is **no** sequence

$$(F_1, u_1, F_2, u_2, \cdots, F_m, u_m, F_{m+1} = F_1)$$

with the following properties

- $m \geqslant 3$
- $u_1, \ldots, u_m$ are distinct vertices of $\mathcal{H}$
- $F_1, \ldots, F_m$ are distinct hyperedges of $\mathcal{H}$
- for every $i \in [m]$, $u_i \in F_i \cap F_{i+1}$, and $u_i \notin F_j$ for every $j \in [m+1] - \{i, i+1\}$.

A query is $\beta$-acyclic if and only if its hypergraph is $\beta$-acyclic.

The rest of this section roughly follows the definitions given in [39]. For a more detailed discussion of (generalized) hypertree decomposition, the reader is referred to [30].

The *width* of a tree-decomposition is the quantity

$$\max_{t \in V(T)} |\chi(t)| - 1.$$

The *treewidth* of a hypergraph $\mathcal{H}$, denoted by $\mathrm{tw}(\mathcal{H})$, is the minimum width over all tree decompositions of the hypergraph.

## A.2 Elimination orders, prefix posets, acyclicity, and hypergraph widths

An *elimination order* of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is simply a total order $v_1, \ldots, v_n$ of all vertices in $\mathcal{V}$. Fix an elimination order $\rho = v_1, \ldots, v_n$ of $\mathcal{H}$, for $j = n, n-1, \ldots, 1$ we recursively define $n$ hypergraphs $\mathcal{H}_n, \mathcal{H}_{n-1}, \ldots, \mathcal{H}_1$, and $n$ set collections $\mathcal{P}_n, \mathcal{P}_{n-1}, \ldots, \mathcal{P}_1$, as follows.

(a) Let $\mathcal{H}_n = \mathcal{H} = (\mathcal{V}_n, \mathcal{E}_n = \mathcal{E})$ and define

$$\begin{aligned} \partial(v_n) &= \{F \in \mathcal{E}_n \mid v_n \in F\}, \\ \mathcal{P}_n &= \{F - \{v_n\} \mid F \in \partial(v_n)\}, \\ U(\mathcal{P}_n) &= \bigcup_{F \in \mathcal{P}_n} F. \end{aligned}$$

In other words, $\partial(v_n)$ is the collection of hyperedges of $\mathcal{H}_n$ each of which contains $v_n$. (The notation $\partial(v)$ is relatively standard in graph theory, denoting the set of edges incident to the vertex $v$.) Next, $\mathcal{P}_n$ is the same set of hyperedges in $\partial(v_n)$ with $v_n$ removed. Note that the empty set might be a member of $\mathcal{P}_n$. Finally, $U(\mathcal{P}_n)$ is the "universe" of sets in $\mathcal{P}_n$.

(b) For each $j = n-1, n-2, \ldots, 1$, define $\mathcal{H}_j = (\mathcal{V}_j, \mathcal{E}_j)$ as follows.

$$\begin{aligned} \mathcal{V}_j &= \{v_1, \ldots, v_j\} \\ \mathcal{E}_j &= \{F - \{v_{j+1}\} \mid F \in \mathcal{E}_{j+1}\} \cup \{U(\mathcal{P}_{j+1})\} \\ \partial(v_j) &= \left\{F \in \mathcal{E}_j \mid v_j \in F\right\} \\ \mathcal{P}_j &= \{F - v_j \mid F \in \partial(v_j)\} \\ U(\mathcal{P}_j) &= \bigcup_{F \in \mathcal{P}_j} F. \end{aligned}$$

In other words, let $\mathcal{H}_j = (\mathcal{V}_j, \mathcal{E}_j)$ be the hypergraph obtained from $\mathcal{H}_{j+1}$ by removing $v_{j+1}$ from $\mathcal{H}_{j+1}$ from all hyperedges, adding a new hyperedge which is the union of all sets in $\mathcal{P}_{j+1}$. Finally, let $\mathcal{P}_j$ be the collection of all hyperedges of $\partial(v_j)$ with $v_j$ removed.

In particular, the hypergraph $\mathcal{H}_j$ is on vertex set $\{v_1, \ldots, v_j\}$, and the hypergraph $\mathcal{H}_1$ has only $\{v_1\}$ as a hyperedge. The universe $U(\mathcal{P}_k)$ of $\mathcal{P}_k$ is a subset of $\{v_1, \ldots, v_{k-1}\}$, and in particular $U(\mathcal{P}_1) = \varnothing$.

**Prefix posets** For each $k \in [n]$, the set collection $\mathcal{P}_k$ is a collection of subsets of $\{v_1, \ldots, v_{k-1}\}$. We will view $\mathcal{P}_k$ as a *partially ordered set* (poset) using the *reversed inclusion order*. In particular, for any $S_1, S_2 \in \mathcal{P}_k$, we write $S_1 \preceq S_2$ if and only if $S_2 \subseteq S_1$.

These posets $\mathcal{P}_k$ are called the *prefix posets* with respect to the elimination order $v_1, \ldots, v_n$ of $\mathcal{H}$. The *bottom element* of a poset $\mathcal{P}$ is an element $F \in \mathcal{P}$ such that $F \preceq F'$ for every $F' \in \mathcal{P}$. The poset $\mathcal{P}$ is a *chain* if all members of $\mathcal{P}$ can be linearly ordered using the $\preceq$ relation. In other words, $\mathcal{P}$ is a chain if its members form a nested inclusion collection of sets. It turns out that we can characterize $\beta$-acyclicity and treewidth of $\mathcal{H}$ using the prefix posets of some elimination order.

**Definition A.5** (Nested elimination order)**.** For any $\beta$-acyclic hypergraph $\mathcal{H}$, a vertex ordering $v_1,\ldots,v_n$ of $\mathcal{H}$ is called a *nested elimination order* if and only if every prefix poset $\mathcal{P}_k$ is chain.

**Proposition A.6** ($\beta$-acyclicity and the GAO)**.** *A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{E})$ is* $\beta$-acyclic if and only if *there exists a vertex ordering $v_1,\ldots,v_n$ which is a nested elimination order for $\mathcal{H}$.*

*Proof.* For the forward direction, suppose $\mathcal{H}$ is $\beta$-acyclic. A *nest point* of $\mathcal{H}$ is a vertex $v\in\mathcal{H}$ such that the collection of hyperedges containing $v$ forms a nested sequence of subsets, one contained in the next. In 1980, Brouwer and Kolen [15] proved that any $\beta$-acyclic hypergraph $\mathcal{H}$ has at least two nest points. Let $v_n$ be a nest point of $\mathcal{H}$. Then, from the definition of nest point, $\partial(v_n)$ is a chain each of whose members contains $v_n$. The set $\mathcal{P}_n$ is thus also a chain as it is the same as $\partial(v_n)$ with $v_n$ removed, and $\mathcal{P}_n$'s bottom element is precisely $U(\mathcal{P}_n)$. Consequently, $\mathcal{H}_{n-1}$ is precisely $\mathcal{H}-\{v_n\}$. The graph $\mathcal{H}-\{v_n\}$ is $\beta$-acyclic because $\mathcal{H}$ is $\beta$-acyclic. By induction there exists an elimination order $v_1,\ldots,v_{n-1}$ such that every prefix poset $\mathcal{P}_k$, $k\in[n-1]$, is a chain. Thus, the elimination order $v_1,\ldots,v_n$ satisfies the desired property.

Conversely, suppose there exists an ordering $v_1,\ldots,v_n$ of all vertices of $\mathcal{H}$ such that every poset $\mathcal{P}_k$ is a chain. Assume to the contrary that $\mathcal{H}$ is not $\beta$-acyclic. Then, there is a sequence

$$(F_1,u_1,F_2,u_2,\ldots,F_m,u_m,F_{m+1}=F_1)$$

satisfying the conditions stated in Definition A.4. Without loss of generality, suppose $u_m$ comes last in the elimination order $v_1,\ldots,v_n$, and that $u_m=v_k$ for some $k$. Then, the poset $\mathcal{P}_k$ contains the set $F_m\cap\{v_1,\ldots,v_{k-1}\}$ and the set $F_1\cap\{v_1,\ldots,v_{k-1}\}$. Since both $u_2$ and $u_{m-1}$ come before $u_m$ in the ordering, we have

$$\begin{aligned} u_2 &\in (F_1\cap\{v_1,\ldots,v_{k-1}\})\setminus(F_m\cap\{v_1,\ldots,v_{k-1}\}) \\ u_{m-1} &\in (F_m\cap\{v_1,\ldots,v_{k-1}\})\setminus(F_1\cap\{v_1,\ldots,v_{k-1}\}). \end{aligned}$$

Consequently, $\mathcal{P}_k$ is not a chain. □

We have just characterized $\beta$-acyclicity with a polynomial-time verifiable property of the GAO. We next characterize the treewidth of a hypergraph using the best "elimination width" of its GAO. This result is well-known in the probabilistic graphical model literature.

**Proposition A.7** (Treewidth and the GAO)**.** *Let $\mathcal{H}=(\mathcal{V},\mathcal{E})$ be a hypergraph with treewidth $w$. Then there exists an elimination order $v_1,\ldots,v_n$ of all vertices of $\mathcal{H}$ such that for every $k\in[n]$ we have $\left|U(\mathcal{P}_k)\right|\leqslant w$.*

*Proof.* This follows from the well-known fact that the smallest induced treewidth (over all elimination orders) of $\mathcal{H}$ is the same as the treewidth of $\mathcal{H}$ (see, e.g., [6, 19]). The maximum size of the universes $U(\mathcal{P}_k)$, $k\in[n]$, is precisely the induced treewidth of the Gaifman graph of $\mathcal{H}$ with respect to the given elimination order. □

# B Certificates

The notion of certificate is subtle. In this section we give a series of examples and proofs of propositions exploring its properties.

## B.1 Illustrations and basic examples

To understand the notion of certificates, it is important to understand the input to Minesweeper and how relations are accessed. Figure 2 gives an illustration of index tuples to access a relation $R$. In this example, the nodes (except for the root) are the variables $R[\mathbf{x}]$ whose contents have been filled out by a database instance.

Next, we start with an extremely simple join query to illustrate the notion of certificates, showing that certificates can have constant size and they can be a lot smaller than the output size.

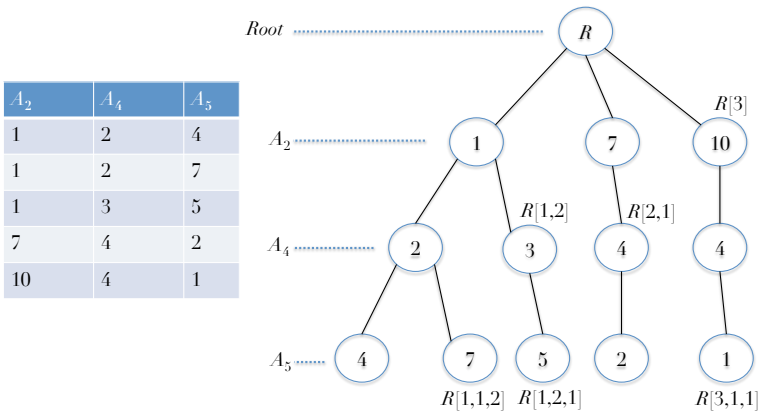



Figure 2: The (unbounded fanout) Search-Tree data structure. Here $R[x_1, x_2]$ is the value of the node where we take the $x_1$th branch of the first level, then the $x_2$th branch at the second level of $R$'s Search-Tree. For this example, $|R[*]| = 3$, $|R[1,*]| = 2$, $|R[2,*]| = 1$.

**Example B.1** (Constant size certificates)**.** Consider the query $R(A) \bowtie S(A,B)$ where

$$\begin{aligned} R &= [N] \\ S &= \{(N+1, i+N) \mid i \in [N]\}. \end{aligned}$$

In this case, $\{R[N] < S[1]\}$ is a certificate showing that the output is empty: for every database $I$ in which $R^I[N] < S^I[1]$ there is no tuple in the output.

**Example B.2** ($|C| \ll Z$)**.** Next, consider the following instance of the same query as above.

$$\begin{aligned} R &= [N] \\ S &= \{(N, 10i), i \in [N]\}. \end{aligned}$$

In this case, $\{R[N] = S[1]\}$ is a certificate because, for every input database $I$ for which $R^I[N] = S^I[1]$, the outputs are tuples of the form $(R^I[N], S^I[1,i]), i \in [N]$. And, the witnesses are pairs of index tuples $\{N, (1,i)\}$. These two examples show that certificates can be of constant size, and they can be arbitrarily smaller than the output size.

Extrapolating from the above example, it is not hard to show that for any join query we can construct an instance whose optimal certificate size is only a function of the query size and not the data. In essence, such certificates are of constant size in data complexity. Consequently, algorithms whose runtime is a function of the optimal certificate can be extremely fast!.

## B.2 Certificate subtleties

The comparisons of the forms shown in (3) allow for comparisons between tuples of the same relation. Comparisons between tuples from the same relation and the equalities can help tremendously in reducing the size of the overall certificate. This fact will make the job of the algorithm designer more difficult if we aim for a runtime proportional to the optimal certificate size. Consider the following example.

**Example B.3** (Equalities and same-relation comparisons are important)**.** Consider the following query, where the global attribute order is $A, B, C$

$$Q = R(A,C) \bowtie S(B,C),$$

where

$$\begin{aligned} R(A,C) &= [N] \times \{2k \mid k \in [N]\} \\ S(B,C) &= [N] \times \{2k-1 \mid k \in [N]\} \end{aligned}$$

The join is empty, and there is a certificate of size $O(N^2)$ showing that the output is empty. Note that both the relations have size $N^2$. The certificate consists of the following comparisons:

$$\begin{aligned} R[1,c] &= R[a,c], \text{ for } a,c \in [N], a > 1 \\ S[1,c] &= S[b,c], \text{ for } b,c \in [N], b > 1, \end{aligned}$$

$$S[1,1] < R[1,1] < S[1,2] < R[1,2] < \cdots < S[1,N] < R[1,N].$$

If we don't use any equality, or if we only compare tuples from different relations, any certificate will have to be of size $\Omega(N^3)$ because it will have to show for each pair $a,b$ that $R[a,*] \cap S[b,*] = \varnothing$ which takes $2N-1$ inequalities, for a grand total of $N^2(2N-1) = \Omega(N^3)$ comparisons.

A certificate is a function of the GAO (and of course, the data). For the same input data, changing the GAO can dramatically change the optimal certificate size, and for non-trivial queries we cannot predict the dramatic difference between optimal certificate sizes of different GAOs without examining the data values.

**Example B.4** (Certificate's dependency on the GAO)**.** Consider the same query as in Example B.3 but with the global attribute order of $C,A,B$. In this case,

$$\begin{aligned} R(C,A) &= \{2k \mid k \in [N]\} \times [N] \\ S(C,B) &= \{2k-1 \mid k \in [N]\} \times [N] \end{aligned}$$

The following is an $O(N)$-sized certificate proving that the output is empty:

$$S[1] < R[1] < S[2] < R[2] < \cdots < S[N] < R[N].$$

This GAO is a nested elimination order for this query, and thus Minesweeper runs in time $\tilde{O}(N)$ on this instance, thanks to Theorem 2.7.

Examples B.3 and B.4 indicate a trend that we can prove rigorously.

**Proposition B.5.** *Let $\rho$ be any GAO. Let $B$ be any private attribute of some relation $R$, i.e. $B$ does not belong to any other relation. Let $\rho'$ be an attribute order obtained from $\rho$ by removing $B$ from $\rho$ and adding it to the end of $\rho$. Let $C(\rho)$ denote an optimal certificate with respect to the GAO $\rho$. Similarly, define $C(\rho')$. Then, $|C(\rho')| \leqslant |C(\rho)|$.*

*Proof.* First, we observe that in an optimal certificate $C$ (for any GAO), there is no comparison involving $B$-variables. If $C$ does contain such comparison, let $\mathcal{A}$ be the argument obtained from $C$ by removing all comparisons involving $B$-variables. We want to show that $\mathcal{A}$ remains a certificate, still, contradicting the optimality of $C$. Let $K$ be any database instance satisfying $C$. (If there is no such $K$, then there is no database instance satisfying $\mathcal{A}$, and hence $\mathcal{A}$ is vacuously a certificate!) Let $I$ and $J$ be two database instances satisfying the argument $\mathcal{A}$. Let $I'$ and $J'$ be obtained from $I$ and $J$ by filling in the $B$-variables using values from the corresponding $B$-variables from $K$. Then, $I'$ and $J'$ satisfy $C$. Consequently, every witness for $Q(I')$ is a witness for $Q(J')$ and vice versa. But every witness for $Q(I')$ is *also* a witness for $Q(I)$, and every witness for $Q(J')$ is also a witness for $Q(J)$, and vice versa, because $B$ is a private attribute! Hence, $\mathcal{A}$ is a certificate as desired.

Second, we can now assume that $C(\rho)$ has no comparison between $B$-variables. Note that, a variable on a relation $R$ is simply a node on its search tree. When we change $\rho$ to $\rho'$, some nodes on a variable coming after $B$ in a relation might collapse into one node because their values are equal. Call the new node an *image* of the old node. Let $\mathcal{A}$ be an argument for the $\rho'$ GAO obtained from $C(\rho)$ by replacing every comparison in $C(\rho)$ with the comparison between their images in $\rho'$. Every database satisfying $\mathcal{A}$ also satisfies $C$, from which we can infer the set of witnesses. Thus $\mathcal{A}$ is a certificate, which can be smaller than $C(\rho)$ because of the collapsing of nodes. □

From the above proposition, we know that better certificates can be obtained by having GAOs in which all private attributes come at the end of the order. Unfortunately, that is as far as the GAO can tell us about the optimal certificate size. If there were more than one non-private attribute, then the optimal certificate size is highly data dependent. The following example illustrates this point further.

**Example B.6** (Certificate's dependency on the GAO even without private attributes)**.** Consider the join query

$$Q = R(A,B) \bowtie S(A,B).$$

Suppose

$$\begin{aligned} R &= \{(i,i) \mid i \in [N]\} \\ S &= \{(N+i,i) \mid i \in [N]\}. \end{aligned}$$

Then, the optimal certificate for the $(A,B)$ order has size $O(1)$:

$$R[N] < S[1],$$

while the optimal certificate for the $(B,A)$ order has size $\Omega(N)$:

$$R[i,N] < S[i,1], \text{ for all } i \in [N].$$

And, we can't tell which is which by just looking at the shape of the search trees for $R$ and $S$.

The following example illustrates that the runtime of $O(|C(\rho)| + Z)$ in the GAO $\rho$ may not be better than the runtime of, say, $O(|C(\rho')|^{w+1} + Z)$ in another GAO $\rho'$ for *the same* data. The notion of *nested elimination order* was defined earlier in Section A.

**Example B.7** (Nested elimination order may have large certificate)**.** It is easy to construct a query and the data so that a nested elimination order has a much larger optimal certificate than a non-nested elimination order. Consider the following query

$$Q = R(A,B,C) \bowtie S(A,C) \bowtie T(B,C).$$

This query is $\beta$-acyclic, and $\rho = (C,A,B)$ is a nested elimination order while $\rho' = (A,B,C)$ is not. Minesweeper runs in time $\tilde{O}(|C(C,A,B)| + Z)$ for the former order, and in time $\tilde{O}(|C(A,B,C)|^3 + Z)$ for the latter. However, it is entirely possible that $|C(A,B,C)|^3 \ll |C(C,A,B)|$. For example, consider

$$\begin{aligned} R(A,B,C) &= \{(i,i,i) \mid i \in [N]\} \\ S(A,C) &= \{(N+i,i) \mid i \in [N]\} \\ T(B,C) &= \{(i,i) \mid i \in [N]\}. \end{aligned}$$

In this case, similar to the previous example $|C(A,B,C)| = 1$ (where it says $R[N] < S[1]$), while $|C(C,A,B)| = \Omega(N)$.

## B.3 Proof of Proposition 2.5

*Proof.* To prove this proposition, it is sufficient to show that the set of comparisons issued by an execution of a comparison-based algorithm is a certificate. To be concrete, we model a comparison-based join algorithm by a decision tree. Every branch in the tree corresponds to a comparison of the form (3). An execution of the join algorithm is a path through this decision tree, reaching a leaf node. At the leaf node, the result $Q(I)$ is labeled. The label at a leaf is the set of tuples the algorithm deems the output of the query applied to database instance $I$. The collection of comparisons down the path is an argument $\mathcal{A}$ which we want to prove a certificate.

First, note that for every tuple $\mathbf{t} = (t_1, \ldots, t_n) \in Q(I)$, the values $t_i$ have to be one of the values $R^I[\mathbf{x}]$ for some $R \in \text{atoms}(Q)$. If this is not the case, then we can perturb the instance $I$ as follows: for every attribute $A_i$ let $M_i$ be the maximum value occurring in any $A_i$-value overall tuples in the input relations. Now, add $M_i + 1$ to every $A_i$-value. Then, all $A_i$-values are shifted the same positive amount. In this new database instance $J$, all of the comparisons in the argument have the same Boolean value, and hence the output has to be the same. Hence, if there was a value $t_i$ in some output tuple not equal to $R[\mathbf{x}]$, the output would be wrong.

Second, we show that every output tuple can be uniquely identified with a witness, independent of the input instance $I$. Recall that a collection $X$ of (full) index tuples is said to be a *witness* for $Q(I)$ if $X$ has exactly one full index tuple from each relation $R \in \text{atoms}(Q)$, and all index tuples in $X$ contribute to the same $\mathbf{t} \in Q(I)$.

Fix an input instance $I$ and an output tuple $\mathbf{t} = (t_1, \ldots, t_n)$. Note as indicated above that the $t_i$ can now be thought of as a variable $R[\mathbf{x}]$ for some index tuple $\mathbf{x}$ (not necessarily full) and some relation $R \in \text{atoms}(Q)$. By definition of the natural join operator, there has to be a witness $X$ for this output tuple $\mathbf{t}$.

Consider, for example, a full index tuple $\mathbf{y} = (y_1, \ldots, y_k)$ from some relation $S$ which is a member of the witness $X$. Suppose the relation $S$ is on attributes $(A_{s(1)}, A_{s(2)}, \ldots, A_{s(k)})$. We show that, for every $j \in [k]$, the argument $\mathcal{A}$ must imply via the transitivity of the equalities in the argument that $S[y_1, \ldots, y_j] = t_{s(j)}$.

Suppose to the contrary that this is not the case. Let $V$ be the set of all variables transitively connected to the variable $S[y_1, \ldots, y_j]$ by the equality comparisons in $\mathcal{A}$.

Now, construct an instance $J$ from instance $I$ by doing the following

- set $R^J[\mathbf{x}] = 2R^I[\mathbf{x}] + 1$ for all variables $R[\mathbf{x}]$ appearing in the argument $\mathcal{A}$ but $R[\mathbf{x}]$ is not in $V$.
- set $R^J[\mathbf{x}] = 2R^I[\mathbf{x}] + 2$ for all variables $R[\mathbf{x}]$ appearing in $V$.

Then, any comparison between a pair of variables both not in $V$ or both in $V$ have the same outcome in both databases $I$ and $J$. For a pair of variables $R[\mathbf{x}] \in V$ and $T[\mathbf{y}] \notin V$ the comparison cannot be an equality from the definition of $V$, and hence the $<$ or $>$ relationship still holds true. This is because if $a$ and $b$ are natural numbers, then $a < b$ implies $2a + 2 < 2b + 1$ and $2a + 1 < 2b + 2$. Consequently, the instance $J$ also satisfies all comparisons in the argument $\mathcal{A}$. However, at this point $S[\mathbf{y}]$ can no longer be contributing to $\mathbf{t}$. More importantly, **no** full index tuple from $S$ can contribute to $\mathbf{t}$ in $Q(J)$. Because,

$$\begin{aligned} S^J[y_1, \ldots, y_{j-1}, y_j - 1] &\leqslant 2S^I[y_1, \ldots, y_{j-1}, y_j - 1] + 1 \\ &\leqslant 2(S^I[y_1, \ldots, y_{j-1}, y_j] - 1) + 1 \\ &= 2S^I[y_1, \ldots, y_{j-1}, y_j] - 1 \\ &= 2t^I_{s(j)} - 1 \\ &< t^J_{s(j)}. \end{aligned}$$

(The first inequality is an equality except when $y_j = 1$.) Similarly,

$$\begin{aligned} S^J[y_1, \ldots, y_{j-1}, y_j + 1] &\geqslant 2S^I[y_1, \ldots, y_{j-1}, y_j + 1] + 1 \\ &\geqslant 2(S^I[y_1, \ldots, y_{j-1}, y_j] + 1) + 1 \\ &= 2S^I[y_1, \ldots, y_{j-1}, y_j] + 3 \\ &= 2t^I_{s(j)} + 3 \\ &> t^J_{s(j)}. \end{aligned}$$

(Except when $y_j = |S[y_1, \ldots, y_{j-1}, *]|$, the first inequality is an equality.) □

## B.4 Proof of Proposition 2.6

*Proof.* We construct a certificate $C$ as follows. For each attribute $A_i$, let $v_1 < v_2 < \cdots < v_p$ denote the set of *all* possible $A_i$-values present in any relations from $\text{atoms}(Q)$ which has $A_i$ as an attribute. More concretely,

$$\{v_1, v_2, \ldots, v_p\} := \bigcup_{R \in \text{atoms}(Q), A_i \in \bar{A}(R)} \pi_{A_i}(R).$$

For each $k \in [p]$, let $T_k$ denote the set of all tuples from relations containing $A_i$ such that the tuple's $A_i$-value is $v_k$. Note that the tuples in $T_k$ can come from the same or different relations in $\text{atoms}(Q)$. Next, add to $C$ at most $|T_k| - 1$ equalities connecting all tuples in $T_k$ asserting that their $A_i$-values are equal. (The reason we may not need exactly $|T_k| - 1$ equalities is because there might be many tuples from the same relation $R$ that share the $A_i$-value, and $A_i$ comes earlier than other attributes of $R$ in the total attribute order.)

Then, for each $k \in [p]$, pick an arbitrary tuple $\mathbf{t}_k \in T_k$ and add $p-1$ inequalities stating that $\mathbf{t}_1.A_i < \mathbf{t}_2.A_i < \cdots < \mathbf{t}_p.A_i$. (Depending on which relation $\mathbf{t}_k$ comes from, the actual syntax for $\mathbf{t}_k.A_i$ is used correspondingly. For example, if $\mathbf{t}_k$ is from the relation $R[A_j, A_i, A_\ell]$, then $\mathbf{t}_k.A_i$ is actually $R[x_j, x_i]$.)

Overall, for each $A_i$ the total number of comparisons we added is at most the number of tuples that has $A_i$ as an attribute. Hence, there are at most $rN$ comparisons added to the certificate $C$, and they represent all the possible relationships we know about the data. The set of comparisons is thus a certificate for this instance. □

# C Running Time Analysis

In this paper, we use the following notion to benchmark the runtime of join algorithms.

**Definition C.1.** We say a join algorithm $\mathcal{A}$ for a join query $Q$ to be instance optimal for $Q$ with optimality ratio $\alpha$ if the following holds. For every instance for $Q$, the runtime of the algorithm is bounded by $O_{|Q|}(\alpha \cdot |C|)$, where $O_{|Q|}(\cdot)$ ignores the dependence on the query size and $C$ be the certificate of the smallest size for the given input instance. We allow $\alpha$ to depend on the input size $N$. Finally, we refer to an instance optimal algorithm for $Q$ with optimality ratio $O(\log N)$ simply as near instance optimal[11] for $Q$.

Next, we briefly justify our definition above. First note that we are using the size of the optimal certificate as a benchmark to quantify the performance of join algorithms. We have already justified this as a natural benchmark to measure the performance of join algorithms in Section 2.2. In particular, recall that Proposition 2.5 says that $|C|$ is a valid lower bound on the number of comparisons made by any comparison-based algorithm that "computes" the join $Q$. Even though this choice makes us compare performance of algorithms in two different models (the RAM model for the runtime and the comparison model for certificates), this is a natural choice that has been made many times in the algorithms literature: most notably, the claim that algorithms to sort $n$ numbers that run in $O(n \log n)$ time are optimal in the comparison model. (This has also been done recently in other works, e.g., in [3, 4].)

Second, the choice to ignore the dependence on the query size is standard in database literature. In particular, in this work we focus on the data complexity of our join algorithms.

Perhaps the more non-standard choice is to call an algorithm with optimality ratio $O(\log N)$ to be (near) instance optimal. We made this choice because this is *unavoidable* for comparison-based algorithm. In particular, there exists a query $Q$ so that every (deterministic) comparison-based join algorithm for $Q$ needs to make $\Omega(\log N \cdot |C|)$ many comparisons on *some* input instance. This follows from the easy-to-verify fact for the selection problem (given $N$ numbers $a_1, \ldots, a_N$ in sorted order, check whether a given value $v$ is one of them), every comparison-based algorithm needs to make $\Omega(\log N)$ many comparisons while every instance can be "certified" with constant many comparisons [47, Problem 1(a)]. For the sake of completeness we sketch the argument below.

Consider the query $Q = R(A) \bowtie S(A)$. Now consider the instance where $R(A) = \{a_1, \ldots, a_N\}$ and $S(A) = \{v\}$. Note that for this instance, we have $|C| \leqslant O(1)$ (and that the output of $Q$ is empty if and only if $v$ does not belong to $\{a_1, \ldots, a_N\}$). However, given any sequence of $\lfloor \log N \rfloor - 1$ comparisons between (the only) element of $S$ and some element of $R$, there always exists two instantiation of $a_1, \ldots, a_N$ and $v$ such that in one case the output of $Q$ is empty and is non-empty in the other case. (Basically, the adversary will always answer the comparison query in a manner that forces $v$ to be in the larger half of the "unexplored" numbers.)

Finally, we remark that even though this $\Omega(\log N)$ lower bound on the optimality ratio is stated for the specific join query $Q$ above, it can be easily extended to any join query $Q'$ where at least two relations share an attribute (by "embedding" the above simple set intersection query $Q$ into $Q'$).

[11] Technically we should be calling such algorithms as near instance optimal for certificate-based complexity but for the sake of brevity we drop the qualification. Further, we use the term near instance optimal to mirror the usage of the term near linear to denote runtimes of $O(N \log N)$.

# D The outer algorithm

## D.1 Worked Example of Minesweeper

**Example D.1** (Minesweeper in action)**.** Let $Q_2$ join the following relations:

$$\begin{aligned} R(A_1) &= [N], \\ S(A_1,A_2) &= [N] \times [N], \\ T(A_2,A_3) &= \{(2,2),(2,4)\}, \\ U(A_3) &= \{1,3\}, \end{aligned}$$

where $(A_1,A_2,A_3)$ is the global attribute order.

In this example, the value domain of every attribute is $[N]$. The algorithm to compute $Q_2$ will run as follows:

- First the constraint set CDS is empty.
- WLOG, assume $\mathbf{t} = (-1,-1,-1)$ is the first tuple returned by CDS.GETPROBEPOINT().
- Step 1, the following constraints will be added to CDS:

  $$\begin{aligned} \langle(-\infty,1),*,*\rangle &: \text{ from } R \text{ and } S \\ \langle 1,(-\infty,1),*\rangle &: \text{ from } S \\ \langle *,(-\infty,2),*\rangle &: \text{ from } T \\ \langle *,=2,(-\infty,2)\rangle &: \text{ from } T \\ \langle *,*,(-\infty,1)\rangle &: \text{ from } U \end{aligned}$$

  Then CDS returns, say, $\mathbf{t} = (1,2,2)$ which does not satisfy any of the above constraints.
- Step 2, the following constraint will be added to CDS:

  $$\langle *,*,(1,3)\rangle \quad : \quad \text{from } U$$

  Then CDS returns, say, $\mathbf{t} = (1,2,3)$ which does not satisfy any of the above constraints.
- Step 3, the following constraint will be added to CDS:

  $$\langle *,=2,(2,4)\rangle \quad : \quad \text{from } T$$

  Then CDS returns, say, $\mathbf{t} = (1,2,4)$ which does not satisfy any of the above constraints.
- Step 4, the following constraint will be added to CDS:

  $$\langle *,*,(3,+\infty)\rangle \quad : \quad \text{from } U$$

  Then CDS returns, say, $\mathbf{t} = (1,3,1)$ which does not satisfy any of the above constraints.
- Step 5, the following constraint will be added to CDS:

  $$\begin{aligned} \langle *,(3,+\infty),*\rangle &: \text{ from } T \\ \langle *,=2,(4,+\infty)\rangle &: \text{ from } T \end{aligned}$$

  At this point no $\mathbf{t} \in O$ is free from the constraints and the algorithm stops, reporting that the output is empty.

## D.2 Proof of Theorem 3.2

*Proof.* We account for the maximum number of iterations to be $O(2^r|C| + Z)$ as follows. We give each comparison in the optimal certificate $O(2^r)$ credits and each output tuple $O(1)$ credits. Every iteration is represented by a distinct probe point (or active tuple) $\mathbf{t}$. Hence, instead of counting the number of iterations we count the number of probe points $\mathbf{t}$ returned by the CDS.

Consider a probe point

$$\mathbf{t} = (t_1, t_2, \ldots, t_n)$$

returned by the CDS in some iteration of Algorithm 2. If $\mathbf{t}$ is an output tuple, then we use a credit from the output tuple to pay for this iteration. Hence, the hard part is to account for the probe points $\mathbf{t}$ that are not part of the query's output. In these cases we will use the credits from the comparisons of $C$.

**Case 1.** First, let us assume that no input relation has a private attribute[12]. (Intuitively, private attributes should not be a factor in any join decision, so this is the harder case.)

Consider a relation $R \in \text{atoms}(Q)$ with $\text{arity}(R) = k$. Let the attributes of $R$, in accordance with the GAO, be

$$\bar{A}(R) = (A_{s(1)}, \ldots, A_{s(k)}).$$

(Strictly speaking, the function $s : [k] \to [n]$ depends on $R$, but we will implicitly assume this dependency to simplify notation.) Let $p$ be an integer such that $p \in \{0, 1, \ldots, k-1\}$. For any vector

$$\mathbf{v} = (v_1, \ldots, v_p) \in \{\ell, h\}^p,$$

the variable

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

is said to be $\mathbf{t}$-*alignable* if all variables

$$R\left[i_R^{(v_1)}\right], \cdots, R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}\right]$$

are already $\mathbf{t}$-alignable **and** if

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

is either equal to $t_{s(p+1)}$ or it is not involved in any comparison in the certificate $C$. Similarly, we define $\mathbf{t}$-alignability for the variable

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right].$$

The semantic of $\mathbf{t}$-alignability is as follows. For any $p \in [k]$, if a $\mathbf{t}$-alignable variable

$$e = R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}\right]$$

is not already equal to $t_{s(p)}$, setting $e = t_{s(p)}$ will transform the input instance into another database instance satisfying all comparisons in $C$ without violating the relative order in the relation that $e$ belongs to. Following this semantic, any element whose index is out of range is *not* $\mathbf{t}$-alignable.

**Claim:** Since $\mathbf{t}$ is not an output tuple, we claim that there must be a relation $R \in \text{atoms}(Q)$ with arity $k$, some $p \in \{0, \ldots, k-1\}$ and a vector $\mathbf{v} \in \{\ell, h\}^p$ for which both variables

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right]$$

and

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

[12] An attribute is private if it only appears in one relation.

are **not** **t**-alignable.

To see the claim, suppose for every relation $R$ the above claim does not hold. Then, for every relation $R \in \text{atoms}(Q)$ there is a vector

$$\mathbf{v}^R = (v_1^R, \ldots, v_k^R) \in \{\ell, h\}^k$$

with $k = \text{arity}(R)$ such that the variable

$$R\left[i_R^{(v_1^R)}, i_R^{(v_1^R, v_2^R)}, \ldots, i_R^{(v_1^R, \ldots, v_k^R)}\right]$$

is **t**-alignable. By definition of **t**-alignability, *all* the variables

$$R\left[i_R^{(v_1^R)}, i_R^{(v_1^R, v_2^R)}, \ldots, i_R^{(v_1^R, \ldots, v_j^R)}\right] \tag{4}$$

are also **t**-alignable for every $j \in [k]$.

Now, to reach a contradiction we construct two database instances $I$ and $J$ satisfying all comparisons in $C$ yet there is a witness for $Q(I)$ which is *not* a witness for $J$.

- *The database instance I.* Keep all variables the same except for the following: for each $j \in [k]$, we set

  $$R\left[i_R^{(v_1^R)}, i_R^{(v_1^R, v_2^R)}, \ldots, i_R^{(v_1^R, \ldots, v_j^R)}\right] = t_{s(j)},$$

  for every $R \in \text{atoms}(Q)$. Then, clearly the following set of full index tuples

  $$\left\{\left(i_R^{(v_1^R)}, i_R^{(v_1^R, v_2^R)}, \ldots, i_R^{(v_1^R, \ldots, v_k^R)}\right) \mid R \in \text{atoms}(Q)\right\} \tag{5}$$

  is a witness for $Q(I)$.

- *The database instance J.* Note that we are in the case where **t** is not an output tuple. Hence, of the variables specified in (4), there must be at least one relation $R$ of arity $k$ and one index $j \in [k]$ for which

  $$R\left[i_R^{(v_1^R)}, i_R^{(v_1^R, v_2^R)}, \ldots, i_R^{(v_1^R, \ldots, v_j^R)}\right] \neq t_{s(j)}.$$

  (Note again that $s$ is a function of $R$ too, but we dropped the subscript for clarity.) Now, we set all of the alignable variables in (4) to be equal to corresponding coordinate in **t**, except for the above. Then, the set defined in (5) is no longer a witness for $Q(J)$.

This is a contradiction and the claim is thus proved.

Now, fix a relation $R$ for which the pair of variables in the claim exists. Let $p$ be the smallest integer in the set $\{0, 1, \ldots, k-1\}$ for which the pair of variables are not **t**-alignable. In particular, for this value of $p$ the pair

$$R\left[i_R^{(v_1)}, i_R^{(v_1, v_2)}, \ldots, i_R^{(v_1, \ldots, v_p)}, i_R^{(v_1, \ldots, v_p, \ell)}\right]$$

and

$$R\left[i_R^{(v_1)}, i_R^{(v_1, v_2)}, \ldots, i_R^{(v_1, \ldots, v_p)}, i_R^{(v_1, \ldots, v_p, h)}\right]$$

are not **t**-alignable due to the fact that both of them are not equal to $t_{s(p+1)}$, not because a prefix variable wasn't alignable. In particular, $t_{s(p+1)}$ falls strictly in the open interval between these two variables.

For this pair, the constraint added in line 20 is not empty. And, each variable in this non-**t**-alignable pair is involved in a comparison in $C$. We will pay for **t** by charging this pair of comparisons. (If one end of this pair is out of range, we will only charge the non-out-of-range end. The other end is either $-\infty$ or $+\infty$.)

Finally, we want to upper bound how many times a pair of comparisons is charged. Consider a pair of non-**t**-alignable variables

$$e^\ell = R\left[i_R^{(v_1)}, i_R^{(v_1, v_2)}, \ldots, i_R^{(v_1, \ldots, v_p)}, i_R^{(v_1, \ldots, v_p, \ell)}\right]$$

and

$$e^h = R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right].$$

Each of $e^\ell$ and $e^h$ is involved in a comparison in $C$, and we need to bound the total charge for these pairs of comparisons. We think of the pair of comparisons as an interval between $e^\ell$ and $e^h$ in a high dimensional space.

To see the charging argument, let us consider a few simple cases. When $p = 0$, then the interval between $e^h = R[i_R^h]$ and $e^\ell = R[i_R^\ell]$ is a band from one hyperplane $H_1$ to another hyperplane $H_2$ of the output space $O$. This band consists of all points in $O$ whose $A_{s(1)}$-values are between $R[i_R^\ell]$ and $R[i_R^h]$. We call such an interval an $n$-dimensional interval. Due to the constraint added in line 20, a probe point $\mathbf{t}$ from a later iteration cannot belong to the band. However, $\mathbf{t}$ might belong to the "left" of $H_1$ or the "right" of $H_2$, in which case a new $n$-dimensional interval might be created that is charged to the comparison involving $H_1$ or involving $H_2$. Consequently, each comparison from a $n$-dimensional interval can be charged twice.

When $p = 1$, the interval between $e^\ell$ and $e^h$ is an $(n-1)$-dimensional interval which is a band lying inside the hyperplane whose $A_{s(1)}$-value is equal to $R[i_R^{(v_1)}]$. In this case, each comparison might be charged 4 times: one from one side of the hyperplane, one from the other side, and twice from the two sides *inside* the hyperplane itself.

It is not hard to formally generalize the above reasoning to show that the comparison involving

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,\ell)}\right]$$

or

$$R\left[i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}, i_R^{(v_1,\ldots,v_p,h)}\right]$$

might be charged $2^{p+1}$ times. Hence, the total number of iterations is at most $O(2^r|C| + Z)$.

The total number of constraints inserted into the data structure CDS is at most $O(m4^r|C| + Z)$, because if the probe point $\mathbf{t}$ is an output tuple then only one constraint is inserted, and when $\mathbf{t}$ is not an output tuple then at most $m2^r$ constraints are inserted.

As for the total run-time, when $\mathbf{t}$ is an output tuple the iteration does about $O(mr \log N)$ amount of work. When $\mathbf{t}$ is not an output tuple, the amount of work is $O(m2^r \log N)$. Hence, the total runtime is $O((4^r|C| + rZ)m \log N)$, not counting the total time CDS takes.

**Case 2.** Now, suppose some input relation has some private attribute. Although this was supposed to be the easier case, and it is, it still needs to be handled with delicate care to rigorously go through. Let us see where the above proof might fail.

The proof fails at the main claim above. When we construct the database instance $J$, in order to use the fact that

$$R\left[i_R^{(v_1^R)}, i_R^{(v_1^R,v_2^R)}, \ldots, i_R^{(v_1^R,\ldots,v_j^R)}\right] \neq t_{s(j)}$$

to conclude that the set (5) is no longer a witness for $Q(J)$, we crucially use the assumption that there is another relation having the attribute $A_{s(j)}$. But, it might be the case that in the alignable variables in (4), all the variables on non-private attributes already aligns perfectly with $\mathbf{t}$, and only private attributes give us the gap (i.e. $\neq t_{s(j)}$ above) we wanted. In this case, the set (5) actually *is* a witness for $Q(J)$ too. What happens really is that the probe point $\mathbf{t}$ is in between output tuples. All of the non-private attributes already align! In this case, we actually need to charge one of the output tuples that align with all the non-private attributes (and align with $\mathbf{t}$). It is not hard to see that each output tuple is still charged only a constant number of times this way. □

## D.3 Proof of Theorem 3.3

*Proof.* Recall from Theorem 3.2 that the total runtime of Algorithm 2 is

$$O((4^r|C| + rZ)m \log(N) + T(\text{CDS})),$$

where $T(\text{CDS})$ is the total time taken by the constraint data structure. And that the algorithm inserts a total of $O(m4^r|C| + Z)$ constraints to the CDS and issues $O(2^r|C| + Z)$ calls to getProbePoint(). Recall also from Proposition 2.6 that the optimal certificate size is only $O(N)$. From this, we show the "negative" result stated.

We prove this theorem by using the reduction from UNIQUE-CLIQUE to the natural join evaluation problem. The reduction is standard [43]. The UNIQUE-$m$-CLIQUE input instance ensures that the output size is at most 1. In the reduction the clique size $m$ will become the number of relations. The result of Chen et al. [18] implied that if there was an $O(N^{o(m)})$-time algorithm solving UNIQUE-$m$-CLIQUE, then the exponential time hypothesis is wrong, and many NP-complete problems have sub-exponential running time. Consequently, if there was a constraint data structure satisfying the stated conditions, the exponential time hypothesis would not hold. □

# E Details on the CDS

This section provides more details on how the CDS is built, and analyzes some of its basic operations. The ConstraintTree data structure is an implementation of the CDS and we use the two terms interchangeably. We first need two basic building blocks called SortedList and IntervalList.

## E.1 The SortedList building block

A SortedList data structure $L$ can store $N$ numbers in *sorted* order with the following operations:

1. $L$.Find($v$) returns TRUE if $v \in L$, FALSE otherwise.
2. $L$.FindLub($v$) returns the smallest $v' \geqslant v$ in $L$. Return FALSE if no such $v'$ exists.
3. $L$.insert($v$) inserts the value $v$ into $L$
4. $L$.Delete($v$) deletes value $v$ from $L$
5. $L$.DeleteInterval($\ell, r$) deletes all the values stored in $L$ that are in the interval $(\ell, r)$, where $\ell, r \in \mathbb{Z} \cup \{-\infty, +\infty\}$.

**Remark E.1.** Even though we defined the SortedList data structure for numbers one can of course store more complex elements as long as there is a key value whose domain is totally ordered.

**Proposition E.2.** *There exists a data structure that implements a* SortedList $L$ *with $N$ elements such that the first four operations above can be performed in time $O(\log N)$ in the worst-case and* DeleteInterval *can be implemented in $O(\log N)$ amortized time.*

*Proof.* By using any balanced binary search trees such as AVL or Red Black trees, the claim on the first four operations follow immediately. For the claim on DeleteInterval note that this implies figuring out the index $i'$ of the smallest $\ell' \geqslant \ell$ and the index $j'$ of the smallest $r' \geqslant r$ in $L$. This can be done by calling FindLub. Then we perform Delete operations on elements from index $i'$ (and $i'+1$ if SortedList$(i') = \ell$) to index $j'-1$. There might be many of these Delete operations but since each of those deleted elements must be added at some earlier point, leading to an overall $O(\log N)$ amortized runtime. □

## E.2 The IntervalList building block

The next building block is called an IntervalList. This data structure stores open intervals $(\ell, r)$, where $\ell, r \in \mathbb{Z} \cup \{-\infty, +\infty\}$, and supports the following operations (where $I$ is the IntervalList):

- $I$.Next($v$) returns the smallest integer $v'$ such that (i) $v \leqslant v'$ and (ii) $v' \notin (\ell, r)$ for every $(\ell, r) \in I$.
- $I$.covers($v$) returns whether the integer $v$ is covered by some interval in $I$
- $I$.insert($\ell, r$) inserts the interval $(\ell, r)$ into $I$.

It is not hard to show that one can use a variation of any *segment tree* or *interval tree* data structure to construct an IntervalList with the above operations taking logarithmic *amortized* cost. In the following proposition, for completeness we describe a simple implementation of IntervalList using SortedList.

**Proposition E.3.** *An* IntervalList *can be implemented on N intervals such that the* Next *and* covers *work in* $O(\log N)$ *worst-case time, and* insert *operates in* $O(\log N)$ *amortized time.*

*Proof.* The main idea is to store the $N$ intervals as *disjoint* intervals. The end points are then stored in a SortedList and then we use the various operations of an SortedList to implement the operations of IntervalList. Details follow.

At any point of time we maintain $m \leqslant N$ *disjoint* intervals $(s_i, t_i)$ $(i \in [m])$ such that

$$\bigcup_{i=1}^{N} (\ell_i, r_i) = \bigcup_{j=1}^{m} (s_j, t_j).$$

We then store all unique numbers $s_1 \leqslant t_1 \leqslant s_2 \leqslant t_2 \leqslant \cdots \leqslant t_m$ in a SortedList $L$ (with two extra bits of information saying whether the number of the left end point of an interval, right end point of an interval, or both left end point of some interval and right end point of another interval). For example, with 2 intervals $(2,5),(5,9)$, elements in SortedList $L$ will be stored as follows: $(2,L),(5,M),(9,R)$, where 2 is the left end point of $(2,5)$, 5 is the mixed point (i.e. the right end point of $(2,5)$ and the left end point of $(5,9)$), and 9 is the right end point of $(5,9)$. Next we show how we implement the three operations needed on $I$.

We begin with $I$.Next$(v)$. Let $u = L$.FindLub$(v)$. If there is no such $u$, then return $v$. Otherwise, if $u$ is the right end point of an interval or the mixed point, then $v$ is in an interval whose right end point is $u$ or $v = u$. In this case, return $u$. Finally, if $u$ is the left end point, then $v$ is not covered by any interval in $L$. And so, return $v$. By Proposition E.2, all this can be implemented in $O(\log m) = O(\log N)$ time.

Next, we consider $I$.covers$(v)$. Let $u = L$.FindLub$(v)$. If there is no such $u$, then return false. Otherwise, if $u$ is the right end point or the mixed point, and $u \neq v$, then $v$ is covered by some interval. In this case, it returns true. Finally, if $u = v$ or $u$ is the left end point, then $v$ is not covered by any interval; and so it returns false. It is easy to check that only FindLub is used and by Proposition E.2 this takes $O(\log N)$ time.

We consider $I$.insert$(\ell, r)$ operation. The main idea is to delete all elements between $\ell$ and $r$ in SortedList $L$ and then adjust the left end point and the right end point. We now present the details. First by covers operation, we determine if $l$ and $r$ are covered by some intervals in $I$. Now run $L$.DeleteInterval$(l, r)$ that will delete all elements in SortedList $L$ that are strictly between $\ell$ and $r$. Finally, we need to adjust the end points $\ell$ and $r$. If $\ell$ is covered by some interval in $I$, then no action needs to be taken on this side; the newly inserted interval $(\ell, r)$ will be merged with the existing interval in $I$ on the left side. Consider the case when there is an entry $(\ell, b)$ in SortedList $L$, where b indicates whether $\ell$ is a left end point, right end point, or a mixed point. This can be checked by $L$.Find$(\ell)$. If $\ell$ is the right end point, then update this entry in SortedList $L$ by $(\ell, M)$. In the final case, if $\ell$ is not covered by any interval and no entry $(\ell, b)$ exists in SortedList $L$, then insert $(\ell, L)$ into $L$. That is all about handling the left side. A similar argument handles the argument for the right side $r$. To analyze the running time, note that all operations used are $I$.covers,$L$.insert, $L$.Find, and $L$.DeleteInterval. By Proposition E.2 and the above reasoning, this leads to an overall $O(\log N)$ amortized runtime. □

### E.3 The ConstraintTree and the InsConstraint operation

As described in Section 3.3, a ConstraintTree is a tree with $n$ levels, one for each of the attributes with the root as the first attribute in the GAO. (See also Figure 1.) The two key data structures associated with each node are implemented using SortedList and IntervalList: $v$.equalities is a SortedList and $v$.intervals is a IntervalList.

We next describe how the CDS supports InsConstraint. The operation InsConstraint is supported by a member function of ConstraintTree called InsertTree that takes as parameter a constraint vector $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$. Algorithm 5 inserts a constraint vector into a ConstraintTree.

From the description above Proposition 3.1 follows straightforwardly. Note again that when we insert a new interval that covers a lot of existing intervals we will have to remove existing intervals; hence the cost is amortized.

## F $\beta$-acyclic queries

This section analyzes getProbePoint algorithm for $\beta$-acyclic queries. The key assumption is that the GAO has to be a nested elimination order, which as shown in Section A precisely characterizes $\beta$-acyclic queries. Since we deal

**Algorithm 5** CDS.InsertTree($\mathbf{c}$)

**Input:** A ConstraintTree $T$ and a constraint vector $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$.
**Output:** Update the data structure with $\mathbf{c}$.

```
1:  i ← 1
2:  v ← root(T)
3:  While c_i is not an interval component do                    ▷ c_i ∈ ℕ ∪ {∗}
4:      If c_i ∈ ℕ and v.intervals.covers(c_i) then
5:          Return                                               ▷ c is subsumed by an exiting constraint
6:      else If (v.equalities.Find(c_i) = false) then            ▷ search even if c_i = ∗
7:          v.equalities.insert(c_i)
8:          Create a new node in T and point v.equalities(c_i) to it
9:      v ← v.equalities(c_i)
10:     i ← i + 1
11: Suppose c_i = (ℓ, r)                                         ▷ c_i is an interval component
12: v.intervals.insert(ℓ, r)                                     ▷ Insert the interval into interval list
13: v.equalities.DeleteInterval(ℓ, r)                            ▷ Update the v.equalities
```

extensively with the partially ordered sets formed by patterns, Figure 3 should help visualizing these posets.

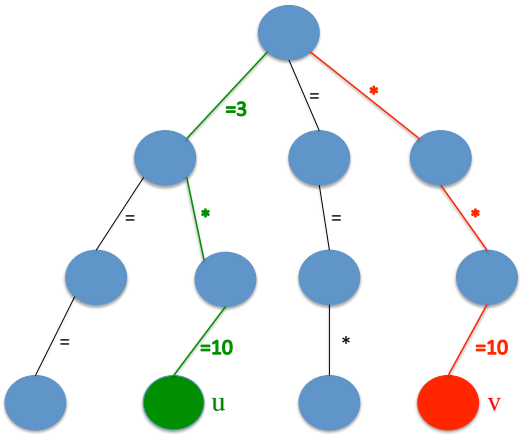



Figure 3: Patterns of nodes and the notion of specialization

The meaning of the terms "specialization" and "generalization" are as follows. Suppose $P(u)$ is a specialization of $P(v)$. Then, the constraints stored in $v$ are of "higher-order" than the constraints stored in $u$. To be more concrete, suppose $P(u) = \langle 3,5 \rangle$ and $P(v) = \langle *,5 \rangle$. Then, for a tuple $\mathbf{t} = (t_1, t_2, t_3)$ to satisfy a constraint stored in $P(u)$, it must be the case that $t_1 = 3$, $t_2 = 5$, and $t_3$ belongs to some interval stored in $u$.intervals. On the other hand, for the tuple to satisfy $P(v)$ we only need $t_2 = 5$ and $t_3 \in v$.intervals.

## F.1 Proof of Proposition 4.2

*Proof.* Let $t_1, \ldots, t_i$ be an arbitrary prefix. Recall that the principal filter $G = G(t_1, \ldots, t_i)$ is a set of nodes $u$ – or equivalently the set of patterns $P(u)$ – which are above the pattern $\langle t_1, \ldots, t_i \rangle$ in the partial order defined in Section 4.2. In particular, for every pattern $P(u)$ in $G$, its equality component comes from one of $\{t_1, \ldots, t_i\}$. It follows that $G$ is isomorphic to a sub-poset of the prefix poset $\mathcal{P}_{i+1}$, which is a chain by Proposition A.6.

Note the important fact that, strictly speaking, the patterns in $G(t_1,\ldots,t_i)$ might come from constraints inserted from relations, or constraints inserted by the outputs of the join. The constraints corresponding to the outputs of the joins always match every entry in a prefix $\langle t_1,\ldots,t_i\rangle$, hence even though Proposition A.6 only infers that the patterns from input relations form a chain, we can safely conclude that the entire poset $G(t_1,\ldots,t_i)$ is a chain. □

## F.2 Proof of Lemma 4.3

This section analyzes the overall run-time of Minesweeper for $\beta$-acyclic queries. Let us first summarize what we know so far. Theorem 3.2 showed that the total runtime of Minesweeper (Algorithm 2) is

$$O((4^r|C|+rZ)\,m\log(N)+T(\text{CDS})),$$

where $T(\text{CDS})$ is the total time it takes the constraint data structure. Minesweeper inserts a total of $O(m4^r|C|+Z)$ constraints to CDS and issues $O(2^r|C|+Z)$ calls to getProbePoint(), where $C$ is any certificate, $Z$ is the output size, and $r$ is the maximum arity over all relations. What we will prove in this section is, provided that the global attribute order is the nested elimination order, we have

$$T(\text{CDS}) = O((4^r|C|+Z)mn2^n\log N).$$

This means the overall runtime is $O(mn2^n(4^r|C|+Z)\log(N))$. Hence, in terms of data complexity the runtime is nearly optimal: $\tilde{O}(|C|+Z)$.

**Lemma F.1** (Re-statement of Lemma 4.3)**.** *Suppose the input query $Q$ is $\beta$-acyclic, and the global attribute order $A_1,\ldots,A_n$ is a nested elimination order, then each of the two operations* getProbePoint() *and* InsConstraint() *of* ConstraintTree *takes amortized time $O(n2^n\log W)$, where $W$ is the total number of constraints ever inserted into* ConstraintTree.

*Proof.* For each node $u\in$ CDS, let $|P(u)|$ denote the number of equality components in the pattern $P(u)$. For example, if $P(u)=\langle *,*,*\rangle$ then $|P(u)|=0$; and if $P(u)=\langle *,3,2\rangle$ then $|P(u)|=2$. Note that $|P(u)|\leqslant n-1$, for all $u\in$ CDS.

Our proof strategy is as follows. We equip each of the InsConstraint and getProbePoint operations with $O(n2^n\log W)$ "credits." We then show that those credits are sufficient to account for the runtime of each operation, *and* at the same time maintain the following *interval-credit invariant*.

**Interval credit invariant:** for every node $u\in$ CDS, and for every interval in the list $u$.intervals, there is always a reserve of at least $\left(2^{|P(u)|+1}-2\right)c\log W$ credits at any point in time, where $c$ is a constant to be specified later. (Note that, by definition, if $|P(u)|=0$ then the intervals in $u$.intervals do not need any reserve credits to maintain the invariant.)

First, for the InsConstraint operation, the interval-credit invariant is easy to maintain. From Proposition 3.1, $O(n\log W)$ credits per operation is already sufficient; furthermore, we have up to $O(n2^n\log W)$ credits to spend. Hence, we have more than enough to give $(2^n-2)c\log W$ credits to the interval of the new constraint for a large enough constant $c$. In fact, we will be very generous by assigning credits as follows.

- We give the interval component of the newly inserted constraint $(2^n-2)c\log W$ credits to maintain the interval-credit invariant. Note that $2^n\geqslant 2^{|P(u)|+1}$ for any node $u$ in the tree.
- We give each component $c_i$ (equality or wildcard) that comes *before* the interval component of the new constraint $5\cdot 2^n c\log W$ credits. How these credits will be used is explained below.

Overall, each InsConstraint operation requires at most

$$n2^{n+3}c\log W = O(n2^n\log W)$$

credits as desired. Note again that $n2^{n+3}c\log W$ credits is a lot more than what is required for the InsConstraint operation by itself. We need the extra credit to pay for something else down the line.

Next, we consider a getProbePoint operation. We iterate through the depth $i$ of the CDS, for $i$ goes from 0 (the root) to $n-1$ (a leaf). At each depth $i$, we try to compute the value $t_{i+1}$, backtracking if necessary. The crucial

observation is the following: at each depth $i$ of the algorithm, thanks to Proposition A.6, the set $G$ forms a *totally ordered set* because the global attribute order is a nested elimination order. Note that Proposition A.6 only considers input relations. In the constraint tree there might be constraints inserted due to the output tuples. However, those constraints are always the most specific (i.e. they are at the bottom of any poset they participate in), and thus in any poset $G$ at any depth $i$ the pattern coming from an output-initiated constraint is a specialization of any input-initiated pattern. Furthermore, and this is a slightly subtle point, there are also intervals inserted due to backtracking; but luckily due to the chain property of the prefix poset $\mathcal{P}_k$, for $\beta$-acyclic queries the backtracking intervals have patterns which are just the same as the patterns from input-generated intervals. Since $G$ is a totally ordered set, it has a bottom element $\bar{u} \in G$.

The basic idea is to show that at each depth $i$ the algorithm takes $O(2^n \log W)$-amortized time, accounted for by using newly infused $O(n2^n \log W)$ credits from GETPROBEPOINT *and* the reserved credits from existing intervals guaranteed by the invariant. At the same time, we need to still maintain the interval credit invariant and thus we cannot abuse the banked reserve of the data structure. In particular, intervals whose reserved credits have been used up have to somehow "disappear" or be infused with fresh credits to maintain the invariant.

Specifically, we will equip the GETPROBEPOINT operation exactly $n2^{n+1}c\log W$ credits, distributing precisely $2^{n+1}c\log W$ credits to each depth $i$ of the tree. These credits will be called the *depth-$i$* credits of GETPROBEPOINT.

Fix an iteration at depth $i \in \{0,\dots,n-1\}$ of the CDS. If $G = \varnothing$ (line 5 of Algorithm 3), then we move on to the next depth and hence depth-$i$ credits of GETPROBEPOINT is more than sufficient to spend here, assuming $c$ is sufficiently large. Henceforth, suppose $|G| \geqslant 1$. Note that we are still considering depth $i$.

**Case 1.** Let us first assume that there is no backtracking at this depth. Let $\bar{u} = u_k \prec u_{k-1} \prec \cdots \prec u_1$ be the members of the poset $G$, which as explained above is a total order. We will show by induction the following claim.

**Claim.** For every $j \in [k]$, the call CDS.NEXTCHAINVAL$(x, u_j, G)$ takes amortized time

$$(2^{|P(u_j)+2|} - 3)c\log W,$$

while maintaining the interval credit invariant. In other words, we need to use $(2^{|P(u_j)+2|} - 3)c\log W$ credits from somewhere to pay for this call.

From the claim, and from the fact that

$$n - 1 \geqslant |P(u_k)| > |P(u_{k-1})| > \cdots > P(u_1) \geqslant 0$$

the initial call CDS.NEXTCHAINVAL$(-1, u_k, G)$ (line 10 of Algorithm 3) takes time at most $(2^{n+1}-3)c\log W = O(2^n \log W)$. Consequently, the depth-$i$ credits of GETPROBEPOINT is sufficient to pay for the call.

We next prove the claim by induction. The base case is when $j = 1$, i.e. when we are calling $u_1$.INTERVALS.NEXT$(x)$. Line 2 of Algorithm 4 takes $O(\log W)$-time, thanks to Proposition E.3. Note that,

$$(2^{|P(u_1)|+2} - 3)c\log W \geqslant c\log W.$$

Hence, with $c$ large we have enough credits to pay for the call.

Next, consider $j \geqslant 2$ and assume the claim holds for $j-1$. Consider a call to CDS.NEXTCHAINVAL$(x, u_j, G)$. An iteration of Algorithm 4 has two steps: (a) line 8 takes $(2^{|P(u_{j-1})|+2} - 3)c\log W$-credits by the claim's induction hypothesis, and (b) line 10 takes $c\log W$-time for $c$ large. In total, each iteration takes time at most

$$\Big(2^{|P(u_{j-1})|+2} - 3\Big)c\log W + c\log W \leqslant \Big(2^{|P(u_j)|+1} - 2\Big)c\log W.$$

If we had to continue on with the next iteration ($y \neq z$), then it must be the case that $z \in (\ell, y)$ for some interval $(\ell, y) \in u_j$.INTERVALS. By the interval credit invariant, this interval $(\ell, y)$ has a credit reserve of $\Big(2^{|P(u_j)|+1} - 2\Big)c\log W$, which by the above inequality is sufficient to pay for the next iteration! This process repeats itself.

Consequently, the reserves of credits at $u_j$-intervals that $z$ hits pay for subsequent iterations. We are left to pay for $(i)$ the first iteration, $(ii)$ the insertion of the new interval in line 13, *and* $(iii)$ fresh credits to deposit to the newly inserted interval to maintain the invariant. It is crucial to notice that the newly inserted interval "consumes" all intervals whose credits we have used up to pay for subsequent iterations. Hence, by paying for $(i)$, $(ii)$, and $(iii)$ above we are done.

With large $c$, the insertion in line 13 takes time at most $c \log W$. Hence, the missing amount in all these three unpaid operations is at most

$$\Big((2^{|P(u_{j-1})|+2} - 3)c \log W + c \log W\Big) + c \log W + (2^{|P(u_j)|+1} - 2)c \log W \leqslant (2^{|P(u_j)|+2} - 3)c \log W.$$

This proves the claim. Figure 4 illustrates the induction reasoning and where all the credits go.

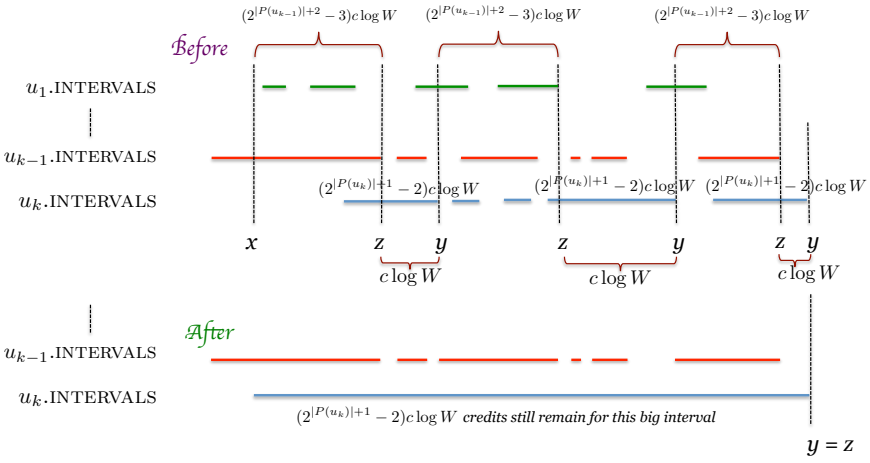



Figure 4: Illustration of the analysis of Algorithm 4

**Case 2.** Next, we consider the case when there is some backtracking. When backtracking occurs in line 15, we have $5 \cdot 2^n c \log W$ credits for *each* of the components of the constraint $\bar{u}$ from $\bar{p}_{i_0}$ to $\bar{p}_i$. We will use these credits as follows.

- $2^n c \log W$ credits of $\bar{p}_{i_0}$ is used for the insertion in line 15 itself,
- $2 \cdot 2^n c \log W$ credits of $\bar{p}_{i_0}$ is now considered fresh depth-$(i_0 - 1)$ credits of the GETPROBEPOINT operation.
- $2 \cdot 2^n c \log W$ credits of $\bar{p}_{i'}$ for every $i'$ from $i_0$ to $i$ are now considered fresh depth-$i'$ credits of GETPROBEPOINT. We need these fresh credits because the depth-$i'$ credits from GETPROBEPOINT have been used up when we visited depth up to $i$ before this backtracking step. Luckily when we backtrack we will never visit the points $\bar{p}_{i_0}, \ldots, \bar{p}_i$ again due to the constraint inserted in line 15, and hence their credits can be used freely.

□

## F.3 Proof of Proposition 2.8

To prove Proposition 2.8 we need an auxiliary lemma.

**Lemma F.2** ( [44]). *If 3-SUM problem does not have a sub-quadratic algorithm, then for every $c \geqslant 3$, there exist $c$-partite graphs $G$ such that listing all $O(|E|)$ $c$-cycles in $G$ needs to take time $\Omega\Big(|E|^{4/3-\epsilon}\Big)$ for any $\epsilon > 0$. Further, the graph $G$ can be written as $(V_1, V_2, \ldots, V_c; E)$ where $E$ can be written as the disjoint union of edge sets $E_{i,i+1 \mod c+1} \subseteq V_i \times V_{i+1 \mod c+1}$.*

*Proof.* The result for $c = 3$ appears in [44]. The extension to the case of $c > 3$ is simple: pick any two partitions and replace each edge (between the two partitions) by a path of length $c-2$. Note that the resulting graph is $c$-partite (with the claimed special structure on the edge set), has $O(c|E|)$ edges and has a $c$-cycle if and only if the original tri-partite graph has a triangle. □

*Proof of Proposition 2.8.* Consider an arbitrary $\beta$-cyclic query $Q$ (with attribute set $\{A_1,\ldots,A_n\}$ and relations/hyperedges $R_1,\ldots,R_m$). Note that this implies that $Q$ has a $\beta$-cycle of length $c \geqslant 3$. W.l.o.g. assume that the cycle involves relations $R_1,\ldots,R_c$ and the attributes $A_1,\ldots,A_c$. In other words, for every $1 < i \leqslant c$, $\{A_{i-1},A_i\} = R_i \cap \{A_1,\ldots,A_c\}$ and $\{A_1,A_c\} = R_1 \cap \{A_1,\ldots,A_c\}$. The idea is to embed the hard instance for listing $c$-cycles from Lemma F.2 into this cycle. Details follow.

Define

$$Q' = \bowtie_{i=1}^{c} R_i.$$

Let $G = (V_1,V_2,\ldots,V_c;E)$ be the hard instance for listing $c$-cycles from Lemma F.2, where $E$ is the disjoint union of $E_{i,i+1 \mod c+1} \subseteq V_i \times V_{i+1 \mod c+1}$. Further, define the instance for $Q$ as follows:

$$\begin{aligned}
R_1 &= \{(u,1,\ldots,1,v) | (u,v) \in E_{1,c}\} \times \left( \bigtimes_{c<j\leqslant m : A_j \in R_1} \{1\} \right), \\
R_i &= \left( \bigtimes_{j<i-1 : A_j \in R_i} \{1\} \right) \times E_{i-1,i} \times \left( \bigtimes_{j>i : A_j \in R_i} \{1\} \right) \text{ for } 1 < i \leqslant c, \\
R_i &= \left( \bigtimes_{j\in[c] : A_j \in R_i} V_j \right) \times \left( \bigtimes_{c<j\leqslant m : A_j \in R_i} \{1\} \right) \text{ for } c < i \leqslant m.
\end{aligned}$$

Note that the size of the output of $Q$ in the instance above is exactly the same as the size of the output of $Q'$. Further, there is a one-to-one correspondence between an output tuple of $Q'$ (and hence $Q$) and a $c$-cycle of $G$. (Indeed for each $c$-cycle $(v_1,\ldots,v_c,v_{c+1}=v_1)$ in $G$ (where $v_j \in V_j$), the output tuple of $Q$ assigns $v_j$ to attribute $A_j$ for every $j \in [c]$ and every other attribute is assigned 1.)

Since the relations $R_i$ for $i \in [c]$ are of size $O(|E|)$, Proposition 2.6 implies that the optimal certificate for $Q'$ has size $O(|E|)$. We claim that such a certificate can be extended to a certificate for $Q$ also of size $O(|E|)$ as we did in the proof of Proposition 5.2. Indeed, the certificate for $Q'$ is enough to pinpoint which tuples are the output tuples with $O(|E|)$ comparisons (or certify that the join is empty). Then $O(n\log(|E|))$ more comparisons can verify whether each of the output tuples of $Q'$ can be extended to an output tuple of $Q'$. (The formal argument is pretty much the same as the argument for the proof of Proposition 5.2. The only difference is that $Q$ can have attributes that are not in $Q'$ and for such attributes we have to check that they have the same values in the projections to $R_j$ for appropriate $c < j \leqslant m$ but this can be done in the claimed time.) Since $G$ has $O(|E|)$ $c$-cycles, this will produce an $O(|E| + c \cdot n \cdot |E| \cdot \log(|E|)) = \tilde{O}_n(|E|)$ sized certificate for $Q$. Thus, an $O(|C|^{4/3-\epsilon})$ time algorithm to solve $Q$ for any $\epsilon > 0$ would (by Lemma F.2) imply a sub-quadratic time algorithm for 3SUM, which is a contradiction. □

# G General queries

In this section, we show that Minesweeper runs in time roughly $\tilde{O}(C^{w+1} + Z)$ for general queries ($\beta$-acyclic or not) where $w$ is the elimination width of the GAO. In particular, if the query has treewidth $w$, then there exists a GAO for which the above runtime holds, thanks to Proposition A.7.

The algorithm for general query is the same as that of the $\beta$-acyclic case; the only (slight) difference is in getProbePoint, which is described next.

## G.1 Algorithms

The getProbePoint algorithm for general queries remains very similar in structure to that of the $\beta$-acyclic case (Algorithm 3), and if the input query is $\beta$-acyclic with the nested elimination order as the global attribute order, then the general getProbePoint algorithm is exactly Algorithm 3. The new issue we have to deal with lies in the fact that the poset $G$ at each depth is not necessarily a chain. Our solution shown in Algorithm 6 is very simple and quite natural: we mimic the behavior of Algorithm 3 on a "shadow" of $G$ that *is* a chain and make use of both the algorithm and the analysis for the $\beta$-acyclic case.

The "shadow" of $G$ is constructed as follows. Let $u_1, \ldots, u_k$ be an arbitrary linearization of nodes in $G$, i.e. if $1 \leqslant i < j \leqslant k$, then either $P(u_i) \preceq P(u_j)$ or $P(u_i)$ and $P(u_j)$ are incomparable using the relation $\preceq$. A linearization always exists because $\preceq$ is a partial order. Now, for $j \in [k]$, define the patterns

$$\bar{P}(u_j) = \bigwedge_{i=j}^{k} P(u_i).$$

Here $\wedge$ denotes "meet" under the partial order $\preceq$. Then, obviously the shadow patterns form a chain:

$$\bar{P}(u_1) \preceq \bar{P}(u_2) \preceq \cdots \preceq \bar{P}(u_k).$$

Note that it is possible for $\bar{P}(u_i) = \bar{P}(u_j)$ for $i \neq j$. For example, suppose the patterns of nodes in $G$ are

$$\langle a, b, * \rangle, \langle *, b, * \rangle, \langle *, *, * \rangle, \langle a, *, c \rangle, \langle *, b, c \rangle;$$

and suppose we pick the following linearization of these patterns:

$$\langle a, *, c \rangle, \langle *, b, c \rangle, \langle a, b, * \rangle, \langle *, b, * \rangle, \langle *, *, * \rangle.$$

Then, the $\bar{P}$ patterns are as follows.

| | | | | | |
|---|---|---|---|---|---|
| Linearization: | $\langle a, *, c \rangle$ | $\langle *, b, c \rangle$ | $\langle a, b, * \rangle$ | $\langle *, b, * \rangle$ | $\langle *, *, * \rangle$ |
| The $\bar{P}$ patterns: | $\langle a, b, c \rangle$ | $\langle a, b, c \rangle$ | $\langle a, b, * \rangle$ | $\langle *, b, * \rangle$ | $\langle *, *, * \rangle$. |

It should be apparent from the above example the two claims we made earlier: the shadow patterns form a chain, and some shadow patterns are the same. To continue with the above example, GETPROBEPOINT is supposed to return a free value $d$ on attribute $D$ which does not belong to any interval in the interval lists of the nodes

$$\langle a, *, c \rangle, \langle *, b, c \rangle, \langle a, b, * \rangle, \langle *, b, * \rangle, \langle *, *, * \rangle.$$

For each node $u$, we will operate as if its pattern was actually $\bar{P}(u)$. Algorithm 6 has the details.

## G.2 Proof of Theorem 5.1

*Proof.* Let us first go through the skeleton of Algorithm 6. We encourage the reader to view Algorithm 3 and Algorithm 6 side-by-side. Their structures are identical except in two places. First, lines 9 to 14 of Algorithm 6 build a shadow chain because $G$ itself is not necessarily a chain as is the case in Algorithm 3. Second, the call to NEXTCHAINVAL on line 10 in Algorithm 3 is replaced by the call on line 17 of Algorithm 6 to NEXTSHADOWCHAINVAL.

For the moment, suppose the calls to NEXTCHAINVAL and NEXTSHADOWCHAINVAL take the same amount of time, then the only extra work that Algorithm 6 does compared to Algorithm 3 per depth of the CDS comes from building up the shadow poset. It takes time $O(mn \log W)$ for each shadow poset. (Recall that $W$ is the number of intervals ever inserted into CDS by Minesweeper.) And, if we also want to maintain the interval credit invariant then it takes $O(mn2^n \log W)$-time extra per depth per GETPROBEPOINT operation. Up to this point, we can mimic the proof from the $\beta$-acyclic case and assign each GETPROBEPOINT operation $O(mn^2 2^n \log W)$ credits. The only difference (so far) from the analysis of the $\beta$-acyclic case is that the amount of credits per depth assigned to GETPROBEPOINT is blown up by a factor of $mn$ due to the shadow poset construction and the extra credits needed for the shadow intervals. Note also that, modulo the difference between the NEXTCHAINVAL call and the NEXTSHADOWCHAINVAL call, if the poset $G$ is a chain, then $\bar{G}$ is exactly $G$ and every node is a shadow of itself! In this case, we do not need to do the extra work of building up the shadow poset and indeed we "get back" Algorithm 3.

Next, let us look at Algorithm 4 and Algorithm 7 side by side. The key difference is in the calls to NEXT on an interval list, we make a call to NEXTCHAINVAL on the chain $\{\bar{u}, u\}$, where $\bar{u}$ is the shadow of $u$. (Since $\bar{u}$ is the meet of all nodes from $u$ and above in the linearization, $\bar{u} \preceq u$ and hence $\{\bar{u}, u\}$ is a chain.) If we maintain the interval credit invariant, then each of these calls to NEXTCHAINVAL$(z, \bar{u}, \{\bar{u}, u\})$ takes amortized time $O(2^n \log W)$, a $2^n$-blowup compared to NEXT. So far, we have an $mn2^n$ blowup factor (with a very loose analysis), relative to the $\beta$-acyclic case.

**Algorithm 6** CDS.GETPROBEPOINT() for general queries

**Input:** A CONSTRAINTTREE CDS
**Output:** Returns a tuple $\mathbf{t} = (t_1, \ldots, t_n)$ that does not satisfy any stored constraint

$i \leftarrow 0$
**While** $i < n$ **do**
  $G \leftarrow \{u \in \text{CDS} \mid (t_1, \ldots, t_i) \preceq P(u)$ and $u$.INTERVALS $\neq \varnothing\}$
  **If** $(G = \varnothing)$ **then** ▷ $G$ will be empty for all later values of $i$
    $t_{i+1} \leftarrow -1$
    $i \leftarrow i + 1$
  **else**
    Let $G = \{u_1, \ldots, u_k\}$, where $u_1, \ldots, u_k$ is a linearization of $G$
    $\bar{G} \leftarrow \varnothing$ ▷ start constructing the shadow chain
    **For** $j \leftarrow 1$ **to** $k$ **do**
      $\bar{P}(u_j) \leftarrow \bigwedge_{\ell=j}^{k} P(u_\ell)$ ▷ $\wedge$ denotes *meet* under partial order $\preceq$
      **If** CDS has no node with pattern $\bar{P}(u_j)$ **then** ▷ Create the shadow nodes
        CDS.INSCONSTRAINT$(\langle \bar{P}(u_j), (-\infty, 0)\rangle)$
      Add the pattern $\bar{P}(u_j)$ to $\bar{G}$ ▷ $\bar{G}$ is a **multiset**, also a chain, break ties arbitrarily
    Let $\bar{\mathbf{p}} = \langle \bar{p}_1, \ldots, \bar{p}_i \rangle$ be the *bottom* element of the poset $\bar{G}$
    Let $\bar{u} \in$ CDS be the node for which $P(\bar{u}) = \bar{\mathbf{p}}$
    $t_{i+1} \leftarrow$ CDS.NEXTSHADOWCHAINVAL$(-1, \bar{u}, \bar{G})$ ▷ Algorithm 7
    $i_0 \leftarrow \max\{k \mid k \leqslant i, \bar{p}_k \neq *\}$
    **If** $(t_{i+1} = +\infty)$ and $i_0 = 0$ **then**
      **Return** NULL ▷ No tuple $\mathbf{t}$ found
    **else If** $(t_{i+1} = +\infty)$ **then**
      CDS.INSCONSTRAINT$(\langle \bar{p}_1, \ldots, \bar{p}_{i_0 - 1}, (\bar{p}_{i_0} - 1, \bar{p}_{i_0} + 1)\rangle$
      $i \leftarrow i_0 - 1$ ▷ Back-track
    **else**
      $i \leftarrow i + 1$ ▷ Advance $i$
**Return** $\mathbf{t} = (t_1, \ldots, t_n)$

**Algorithm 7** CDS.NEXTSHADOWCHAINVAL$(x, u, \bar{G})$, where $\bar{G}$ is a chain

**Input:** A CONSTRAINTTREE $T$, a node $\bar{u} \in \bar{G}$ to start the recursion with
**Input:** A (multiset) chain $\bar{G}$ of nodes, and a starting value $x$
**Output:** the smallest value $y \geqslant x$ not covered by *any* $v$.INTERVALS, for all $v \in \bar{G}$ such that $P(\bar{u}) \preceq P(v)$ ▷ note that $v$ could be an original node or a shadow node

Let $u$ be the node that $\bar{u}$ is a shadow of ▷ $u$ could be the same as $\bar{u}$
**If** there is no $v \in \bar{G}$ for which $P(\bar{u}) \prec P(v)$ **then** ▷ At the top of the chain $\bar{G}$
  **Return** CDS.NEXTCHAINVAL$(x, \bar{u}, \{\bar{u}, u\})$ ▷ Algorithm 4
**else**
  $y \leftarrow x$
  **repeat**
    Let $v \in \bar{G}$ such that $P(u) \prec P(v)$ ▷ Next node up the shadow chain
    $z \leftarrow$ CDS.NEXTSHADOWCHAINVAL$(y, v, \bar{G})$ ▷ first "free value" $\geqslant y$ up the chain
    $y \leftarrow$ CDS.NEXTCHAINVAL$(z, \bar{u}, \{\bar{u}, u\})$ ▷ first "free value" $\geqslant z$ at $u$
  **until** $y = z$
  CDS.INSCONSTRAINT$(\langle P(u), (x - 1, y)\rangle)$ ▷ All values from $x$ to $y - 1$ are not available
  **Return** $y$

Note again that, if the poset $G$ is a chain, implying $\bar{G} = G$, and every node is a shadow of itself, then the calls to INTERVALS.NEXT and to NEXTCHAINVAL$(u, z, \{u\})$ are identical. In this case NEXTSHADOWCHAINVAL is the same as NEXTCHAINVAL and we get back to the $\beta$-acyclic case.

In general, however, we cannot maintain the interval credit invariant by simply giving each inserted interval $O(n2^n \log W)$ credits (blown up by $mn2^n$ more) as we have done in the $\beta$-acyclic case because the same node $u$ might have different shadows depending on the prefix we are working on. For example, the node $u = \langle *, b, * \rangle$ might have the shadows $\bar{u}^1 = \langle a, b, * \rangle$, $\bar{u}^2 = \langle a', b, c' \rangle$, $\bar{u}^3 = \langle *, b, c'' \rangle$, and so forth. In this example, the number of credits we give to an interval in the list $u$.INTERVALS has to be at least three times as much as that in the $\beta$-acyclic case because $u$ might participate in the NEXTCHAINVAL calls with each of its shadows. Consequently, we will have to give each inserted interval many more credits than $O(mn^2 4^n \log W)$. The key question is: *how many more credits*?

The number of credits assigned to each interval depends on the size of its pattern, and on what type of interval it is. From Theorem 3.2, we know that the number of intervals inserted into the CDS is $O(m4^r|C| + Z)$. In fact, there are two types of intervals inserted: $Z$ intervals inserted by the output called the *output-generated intervals*, and $O(m4^r|C|)$ intervals inserted by the input relations, called the *input-generated intervals*. There are also *backtracking intervals* created by the algorithm. Hence, overall we have three types of intervals.

The overall credit-assignment scheme is intimately tied to the size of interval pattern under consideration. Recall that the *size* of a pattern is defined to be the number of equality components of the pattern. For example, the pattern $P(u) = \langle a, b, * \rangle$ has size 2 and the pattern $P(u) = \langle *, *, * \rangle$ has size 0.

Consider the simplest case when $P(u)$ has size 0, such as $P(u) = \langle *, *, * \rangle$. In this case, the node $u$ is always on top of the linearization $G$ and thus in $\bar{G}$ it is the shadow of itself. In other words, it does not really have any shadow. The symmetric situation is when $P(u)$ has **no** wildcard pattern. All of the intervals that come from the outputs are of this type. This includes backtracking intervals which are created from a prefix of output-generated intervals. In this case, $u$ is always at the *bottom* of the linearization $G$ and thus in $\bar{G}$ it is also a shadow of itself. For these types of intervals – intervals in $u$.INTERVALS where $u$ is self-shadowed – we can give them the same credits as they get in the $\beta$-acyclic case (multiplied by the blowup factor).

Next, we analyze how many credits we need for intervals whose patterns are not self-shadowed at some point in the execution of Minesweeper. (A pattern might be self-shadowed at one point, but then not self-shadowed at another time due to a different prefix.) Let $P(u)$ be one such pattern with size $s$; let $k-1$ be the length of the pattern $P(u)$, i.e. intervals of $u$ are on attribute $A_k$ in the global attribute order. Then, the support of $P(u)$ (the positions of equality components) is precisely a subset of the universe $U(\mathcal{P}_k)$ of the prefix poset $\mathcal{P}_k$ as defined in Section A.2. From Proposition A.7 we know that $s \leqslant w$, and hence $1 \leqslant s \leqslant w-1$. (If $s \in \{0, w\}$ then $P(u)$ is self-shadowed.) Since $s$ out of $w$ components in $P(u)$ are already fixed, the number of different shadows $\bar{P}(u)$ of $u$ is at most $(m4^r|C|)^{w-s}$: it has $w-s$ degrees of freedom, each of which can be attributed to some interval in the set of input-generated intervals. Consequently, we can give each interval in $u$.INTERVALS the following number of credits to pay for all operations it's involved in: $(m4^r|C|)^{w-s} \cdot O(mn2^n \log W)$. In the above accounting we did not need to distinguish between input-generated intervals or backtracking intervals, namely $P(u)$ can be the prefix of a backtracking interval too.

Let us summarize what we know thus far:

- Intervals whose patterns are self-shadowed get the same credits as in the $\beta$-acyclic case.
- Other intervals with size-$s$ patterns get $(m4^r|C|)^{w-s} \cdot O(mn2^n \log W)$ credits.

(Again, the credits are supposed to be multiplied by $mn2^n$.) So our next task is to sum up all the credits we need and that will be the final (amortized) runtime of Minesweeper.

- Each output-generated interval gets $O(mn^2 4^n \log W)$ credits. (We already multiplied in the blowup factor.) The total contribution of output-generated intervals to the overall runtime is thus $O(mn^2 4^n Z \log W)$.
- Each input-generated interval with pattern of size $s \in \{0, w\}$ gets $O(mn^2 4^n \log W)$ credits. Each input-generated interval with pattern of size $s \in [w-1]$ receives

$$O\Big((m4^r|C|)^{w-s} \cdot mn^2 4^n \log W\Big)$$

credits. Since there were at most $m4^r|C|$ input-generated intervals, the total number of credits infused is at most

$$O\Big(m4^r|C| \cdot (m4^r|C|)^{w-1} \cdot mn^2 4^n \log W\Big) = O\Big(m^{w+1} n^2 4^{n+rw} |C|^w \log W\Big).$$

- Lastly, we account for the backtracking intervals whose patterns are not self-shadowed. The number of such intervals with a size-$s$ pattern can be upperbounded by $n\binom{w}{s}(m4^r|C|)^{s+1}$, because every backtracking interval must have come from a pattern (of size $s+1$) and each equality component of the pattern can be attributed to an input-generated interval. Each such interval, as analyzed above, gets $O\Big((m4^r|C|)^{w-s} \cdot mn^2 4^n \log W\Big)$ credits. Hence, overall we need

$$O\left(\sum_{s=1}^{w-1} n\binom{w}{s}(m4^r|C|)^{s+1} \cdot (m4^r|C|)^{w-s} \cdot mn^2 4^n \log W\right) = O\Big(n2^w (m4^r|C|)^{w+1} mn^2 4^n \log W\Big).$$

Overall, over-estimating by a lot, we need to pump in

$$\begin{aligned} & mn^2 4^n Z \log W + m^{w+1} n^2 4^{n+rw} |C|^w \log W + n2^w (m4^r|C|)^{w+1} mn^2 4^n \log W \\ \leqslant\ & 2mn^2 4^n (nm^{w+1} 8^{r(w+1)} |C|^{w+1} + Z) \log W \\ =\ & O\Big(m^3 n^3 4^n \Big(nm^{w+1} 8^{r(w+1)} |C|^{w+1} + Z\Big) \log N\Big). \end{aligned}$$

To get the last inequality, we bound $W$ – the number of intervals ever inserted in to the CDS – as follows. $W$ is at most the number of input-generated intervals plus the number of output generated intervals plus the number of backtracking intervals:

$$W \leqslant m4^r|C| + N^m + n2^w|C|^w \leqslant 3mn4^n N^m.$$

□

## G.3 Proof of Proposition 5.2

*Proof.* We prove this result by using the reduction from UNIQUE-CLIQUE to the natural join evaluation problem. The UNIQUE-$k$-CLIQUE input instance ensures that the output size is at most 1. Let's say the input graph is $G = (V,E)$ (with no self loops), which is guaranteed to either have no clique or exactly one clique. Then consider the following query

$$Q_k = \Big(\bowtie_{i \neq j \in [k]} R_{i,j}(v_i, v_j)\Big) \bowtie U(v_1, \ldots, v_k),$$

where the domain of each $v_i$ for $i \in [k]$ is $V$,

$$R_{i,j} = \bigcup_{(u,v) \in E} \{(u,v),(v,u)\},$$

and $U = V^k$. Note that $Q_k$ is empty if $G$ has no clique otherwise $Q_k$ has exactly $k!$ tuples (corresponding to each of the $k!$ assignments of the vertices of the clique in $G$ to $v_1, \ldots, v_k$). Further, $Q_k$ is $\alpha$-acyclic because of the presence of $U$. Finally, it is easy to verify that $Q_k$ has treewidth $k-1$.

Next, we argue that the certificate size of the query above is $O(|E|)$. To see this, first consider the sub-query

$$Q'_k = \bowtie_{i \neq j \in [k]} R_{i,j}(v_i, v_j).$$

Since all the relations are of size $2|E|$, Proposition 2.6 implies that the optimal certificate for $Q'_k$ has size $O(|E|)$. We claim that such a certificate can be extended to a certificate for $Q_k$ also of size $O(|E|)$. Indeed, the certificate for $Q'_k$ is enough to pinpoint which tuples are the $k!$ output tuples with $O(|E|)$ comparisons (or certify that the join is empty). Then $O(k \log(|E|))$ more comparisons can verify whether each of the $k!$ tuples is in $U$ or not. This will produce an

$O(|E| + k \cdot k! \cdot \log(|E|)) = O_k(|E|)$ sized certificate for $Q_k$.[13] Thus, if there were an $|C|^{o(k)}$ algorithm for $Q_k$, it would determine if $G$ has a clique or not in time $\tilde{O}_k(|E|^{o(k)}) = \tilde{O}_k(|V|^{o(k)})$.

However, Chen et al. [18] showed that if there was an $O(|V|^{o(k)})$-time algorithm solving UNIQUE-$k$-CLIQUE, then the exponential time hypothesis is wrong, and many NP-complete problems have sub-exponential running time. This implies that for large enough $k$, the above $|C|^{o(k)}$ time algorithm will be a contradiction. □

## G.4 Proof of Proposition 5.3

*Proof.* We will in fact prove this result using the same query family as in Proposition 5.2. In particular, we define

$$Q_w = \left(\bowtie_{i \neq j \in [w+1]} R_{i,j}(v_i, v_j)\right) \bowtie U(v_1, \ldots, v_{w+1}).$$

As was observed in the proof of Proposition 5.2, $Q_w$ is both $\alpha$-acyclic and has treewidth $w$.

W.l.o.g. assume that the global attribute order is $v_1, \ldots, v_{w+1}$. Now consider the following input instance:

$$U = [m]^{w+1},$$

$$R_{i,j} = [m] \times [m] \text{ for every } (i,j) \in [w] \times [w],$$
$$R_{i,w+1} = [m] \times \{1\} \text{ for every } i \in [w-1],$$

and

$$R_{w,w+1} = [m] \times \{2\}.$$

It is easy to check that the output of $Q_w$ on the input above is empty and that $|C| \leqslant O(wm)$. To see why the latter is true note that with $m-1$ equalities one can certify $\pi_{v_{w+1}}(R_{i,w+1})$ for every $i \in [w]$. Further with $w-1$ further equalities and one comparison one can certify that the output is empty. Thus, we need overall $O(wm)$ comparisons. To complete the proof, we will show that Minesweeper on the input above runs in time $\Omega(m^w)$, which would prove the result (since we are ignoring the query complexity). In fact, we will prove this claim by showing that Line 17 in Algorithm 6 is executed $\Omega(m^w)$ times.

For simplicity, we will assume that Minesweeper always has the interval $(-\infty, 0]$ inserted in all branches of its CDS.

We will argue Minesweeper has to consider all possible prefixes of size $w$: $(t_1, \ldots, t_w) \in [m]^w$. In particular, for each such prefix Algorithm 6 executes Line 17. One can show (e.g. by induction) that for any such prefix $(t_1, \ldots, t_w)$, the only constraints in the CDS that can rule them out are of the form $\langle *, *, \cdots, *, t_i, *, \cdots, *, (1, \infty)\rangle$ for $i \in [w-1]$ and $\langle *, \cdots, *, t_w, (0,2)\rangle$. However, this implies that to rule this prefix (specifically the potential tuple $(t_1, \ldots, t_w, 1)$) out, Algorithm 6 has to "merge" at least two of these constraints, which means that Line 17 has to be executed at least once[14], as desired. □

# H end-to-end results for the set intersection query

This section describes our results specialized to intersection queries, which have been discussed by previous work. The purpose is for the interested reader to both see all the tools used in this simple example and be able to more directly compare our results with previous results on set intersection. Also, this query and the bowtie query in the next section are both $\beta$-acyclic (with any GAO); these two sections illustrate many of the key ideas in our outer algorithm analysis, the design and analysis of CDS and GETPROBEPOINT, without too much abstraction.

[13] More formally, let $\mathbf{t} \in Q_k'$ and define $\mathbf{t}_i = \pi_{R_{i,i+1 \mod k+1}}(\mathbf{t})$. Further, let $\mathbf{t}_i = (R_{i,i+1 \mod (k)+1}[x_i], R_{i,i+1 \mod (k)+1}[x_i, y_i])$ for $i \in [k]$. Then to "pinpoint" whether $\mathbf{t} \in U$, we run the following $k$ binary searches: for $i \in [k-1]$, perform binary search to compute a $z_i$ such that $R_{i,i+1 \mod (k)+1}[x_i] = U[z_1, \ldots, z_{i-1}, z_i]$. Then perform the binary search to compute $z_k$ such that $R_{k,1}[x_k, y_k] = U[z_1, \ldots, z_k]$. If any of the $z_i$'s do not exist, then $\mathbf{t}$ is not in $Q_k$ otherwise it does. It is easy to check that the above set of comparisons (along with the comparisons in the certificate for $Q_k'$) constitute a valid certificate for $Q_k$ (in the sense of Definition 2.3). Finally, note that each binary search can be done with $O(\log(|E|))$ comparisons, which implies the claimed certificate size of $Q_k$.

[14] Note that all the constraints listed above might not exist in which case Line 17 might not be able to rule the tuple $(t_1, \ldots, t_w, 1)$ out but that is fine since the outer algorithm will rule this tuple out.

## H.1 The set intersection query

**Definition H.1** (Set intersection query)**.** The *set intersection query* $Q_\cap$ is defined as

$$Q_\cap = S_1(A) \bowtie S_2(A) \bowtie \cdots \bowtie S_m(A).$$

In this query, each input relation $S_i$ is unary over the same attribute $A$. So each input relation $S_i$ can be viewed as a set of (distinct) values over the domain $\mathbf{D}(A)$. The output $Q_\cap$ is simply the intersection of all input relations $S_i$. In this case, $\text{atoms}(Q_\cap) = \{S_1, \ldots, S_m\}$, $\bar{A} = \bar{A}(S_i) = \{A\}$, for all $i \in [m]$, and an output "tuple" is a one-dimensional vector of the form $\mathbf{t} = (t)$, where $t \in S_1 \cap \cdots \cap S_m$.

In this section, we present Minesweeper specialized to the intersection query $Q_\cap$. To recap, consider the following problem. We want to compute the intersection of $m$ sets $S_1, \cdots, S_m$. Let $n_i = |S_i|$. We assume that the sets are sorted, i.e.

$$S_i[1] < S_i[2] < \cdots < S_i[n_i], \forall i \in [m].$$

The set elements belong to the same domain $\mathbf{D}$, which is a totally ordered domain. Without loss of generality, we will assume that $\mathbf{D} = \mathbb{N}$.

In "practice" it might be convenient to think of $\mathbf{D}$ as the *index set* to another data structure that stores the real domain values. For example, suppose the domain values are `string`s and there are only 3 strings `this`, `is`, `interesting` in the domain. Then, we can assume that those strings are stored in a 3-element array, and the value $a \in \mathbf{D}$ is one of the three indices $0, 1, 2$ into the array.

## H.2 The CDS for $Q_\cap$

The CDS for $Q_\cap$ is a data structure that stores a collection of *open* intervals of the form $(a, b)$, where $a$ and $b$ are in the set $\mathbb{N} \cup \{-\infty, +\infty\}$. When two intervals overlap they are automatically merged. We overload notation and refer to both the data structure and the set of intervals stored as CDS. The data structure supports two operations: InsConstraint and getProbePoint.

- The InsConstraint operation takes an open interval and inserts it into the CDS.
- The getProbePoint operation either returns an integer $t$ that does not belong to any stored interval, or returns null if no such $t$ exists.
- If CDS is empty, then CDS.getProbePoint() returns an arbitrary integer. We use $-1$ as the default.

**Two options for implementing the constraint data structure CDS** If we implement the data structure CDS straightforwardly, then we can do the following. We give each input interval one credit, $1/2$ to each end of the interval. When two intervals are merged, say $(a_1, b_1)$ is merged with $(a_2, b_2)$ to become $(a_1, b_2)$, we use $1/2$ credit from $b_1$ and $1/2$ credit from $a_2$ to pay for the merge operation. If an interval is contained in another interval, only the larger interval is retained in the data structure. By maintaining the intervals in sorted order, in $O(1)$-time the data structure can either return a probe point $t$ that does not belong to any stored interval, or correctly report (return null) that no such $t$ exists. In other words, each call to getProbePoint takes amortized constant time. Inserting a new interval into CDS takes $O(\log W)$-amortized time where $W$ is the maximum number of intervals ever inserted into CDS, using the credit scheme described above.

On the other hand, if we apply the strategy of always returning the least value of $t$ that does not belong to any stored interval, then it is easy to see that CDS essentially only needs to maintain *one* interval $(-\infty, t)$. Initially when CDS is empty $t = -1$ is returned. After that – referring forward to the outer algorithm presented in the next section – the newly inserted intervals always contain $t$ and the new single interval maintained in CDS becomes $(-\infty, t')$ for some $t' > t$. Insertion of a new interval only takes constant time because we only need to compare the high-end of the interval with the current $t$ value. In this case, the algorithm becomes the minimum-comparison method in [20] and it is the same as a typical $m$-way merge join algorithm.

## H.3 The outer algorithm for intersection

The outer algorithm for Minesweeper specialized to evaluate $Q_\cap$ is presented in Algorithm 8. In this case, each *constraint* is an open interval of the form $(a,b)$, where $a$ and $b$ are *integers*. An interval $(a,b)$ is inserted into the constraint data structure CDS if the algorithm has determined that the interval $(a,b)$ contains **no** output. Note that $a$ and/or $b$ themselves might be part of the output, but any value in between is not. In particular, the constraint data structure CDS stores a set of constraints and thus we will use the term *constraint set* to refer to the set of intervals stored in CDS.

**Algorithm 8** Minesweeper for computing the intersection of $m$ sets

**Input:** $m$ sorted sets $S_1,\cdots,S_m$, where $|S_i| = n_i$, $i \in [m]$
**Input:** Elements of $S_i$ are $S_i[1],\cdots,S_i[n_i]$
**Input:** Implicitly $S_i[0] = -\infty$, $S_i[n_i+1] = +\infty$, following the conventions stipulated in (1) and (2)

```
1:  Initialize the constraint data structure CDS ← ∅
2:  While ((t ← CDS.getProbePoint()) ≠ null) do
3:      For i = 1,...,m do
4:          x_i^h ← min{j | S_i[j] ⩾ t}
5:          x_i^ℓ ← max{j | S_i[j] ⩽ t}                    ▷ It is possible that x_i^h = x_i^ℓ
6:      If S_i[x_i^h] = t for all i ∈ [m] then              ▷ Then all S_i[x_i^ℓ] = t too
7:          Output t
8:          CDS.InsConstraint(t − 1, t + 1)
9:      else
10:         For each i ∈ [m] such that S_i[x_i^h] > t do    ▷ S_i[x_i^ℓ] < t and x_i^ℓ = x_i^h − 1 for such index i
11:             CDS.InsConstraint(S_i[x_i^ℓ], S_i[x_i^h])
```

We next run through the elements of the argument and the algorithm for the case of $Q_\cap$.

## H.4 Analysis

We specialize notions of argument and certificate to $Q_\cap$ in order to illustrate these concepts.

**Definition H.2** (Argument for $Q_\cap$)**.** An *argument* is a finite set of symbolic equalities and inequalities, or *comparisons*, of the following forms: (1) $(S_s[i] < S_t[j])$ or (2) $S_s[i] = S_t[j]$ for $i,j \geqslant 1$ and $s,t \in [m]$. An instance satisfies an argument if all the comparisons in the argument hold for that instance.

**Definition H.3** (Certificate for $Q_\cap$)**.** An argument $\mathcal{A}$ is called a *certificate* if any collection of input sets $S_1,\ldots,S_m$ satisfying $\mathcal{A}$ must have the "same" output, in the following sense. Let $R_1,\ldots,R_m$ be an arbitrary set of unary relations such that $|R_j| = |S_j|$, for all $j \in [m]$, and that $R_1,\ldots,R_m$ satisfy all comparisons in the certificate $\mathcal{A}$, then the following must hold:

$$S_1[i_1] = S_2[i_2] = \cdots = S_m[i_m]$$

if and only if

$$R_1[i_1] = R_2[i_2] = \cdots = R_m[i_m].$$

The tuple $(i_1,\ldots,i_m)$ is called a *witness* for this instance of the query. Another way to state the definition is that, an argument is a certificate iff all instances satisfying the argument must have the same set of witnesses.

The *size* of a certificate is the number of comparisons in it. The *optimal certificate* for an input instance is the smallest-size certificate that the instance satisfies. The optimal certificate size measures the information-theoretic lowerbound on the number of comparisons that any comparison-based join algorithm has to discover. Hence, if there was an algorithm that runs in linear time in the optimal certificate size, then that algorithm would be instance-optimal.

The following theorem along with Proposition 2.5 imply that Algorithm 8 has a near instance-optimal run-time. up to an $m \log N$ factor. Since $m$ is part of the query size and the output has to be reported, Minesweeper is instance-optimal in terms of data complexity up to a log factor for this query.

**Theorem H.4** (Minesweeper is near instance optimal for $Q_\cap$)**.** *Algorithm 8 runs in time $O((|C|+Z)m\log N)$, where $C$ is* **any** *certificate for the instance, $N=\sum_{i=1}^{m} n_i$ is the input size, and $Z$ is the output size.*

*Proof.* We show that the number of iterations of Algorithm 8 is $O(|C|+Z)$, and that each iteration takes time $O(m\log N)$.

To upperbound the number of iterations, the key idea is to "charge" each iteration of the main while loop to either a distinct output value *or* a pair of comparisons in the certificate $C$ such that no comparison will ever be charged more than a constant number of times. Each iteration in the loop is represented by a distinct probe value $t$. Hence, we will find an output value or a pair of comparisons to "pay" for $t$ instead of paying for the iteration itself.

Let $t$ be a probe value in an arbitrary iteration of Algorithm 8. Let $x_i^h$ and $x_i^\ell$ be defined as in lines 4 and 5 of the algorithm.

First, consider the case when $S_i[x_i^h]=t$ for all $i\in[m]$, i.e. $t$ is an output value. Note that in this case $x_i^\ell=x_i^h$ for all $i\in[m]$. We pay for $t$ by charging the output $t$. The new constraint inserted in line 8 ensures that we will never charge an output twice.

Second, suppose $S_i[x_i^h]>t$ for some $i$, i.e. $t$ is not an output. (Note that by definition it follows that $S_i[x_i^\ell]<t$.) For each $i\in[m]$, the variable $S_i[x_i^h]$ is said to be *$t$-alignable* if $S_i[x_i^h]$ is already equal to $t$ (in which case $x_i^\ell=x_i^h$) or if $S_i[x_i^h]$ is not part of *any* comparison $(=,<,>)$ in the certificate $C$. Similarly, we define the notion of *$t$-alignability* for $S_i[x_i^\ell]$, $i\in[m]$.

When $S_i[x_i^h]$ is $t$-alignable, setting $S_i[x_i^h]=t$ will not violate any of the comparisons in the certificate $C$. Similarly, we can transform the input instance to another input instance satisfying $C$ by setting $S_i[x_i^\ell]=t$, provided $S_i[x_i^\ell]$ is $t$-alignable.

**Claim:** if $t$ is not an output, then there must exist some $\bar{i}\in[m]$ for which both $S_{\bar{i}}[x_{\bar{i}}^\ell]$ and $S_{\bar{i}}[x_{\bar{i}}^h]$ are *not* $t$-alignable. In particular, in that case

$$S_{\bar{i}}[x_{\bar{i}}^\ell]<t<S_{\bar{i}}[x_{\bar{i}}^h]$$

and both $S_{\bar{i}}[x_{\bar{i}}^\ell]$ and $S_{\bar{i}}[x_{\bar{i}}^h]$ are involved in comparisons in the certificate.

Before proving the claim, let us assume it is true and finish off the charging argument. We will pay for $t$ using any comparison involving $S_{\bar{i}}[x_{\bar{i}}^h]$ and any comparison involving $S_{\bar{i}}[x_{\bar{i}}^\ell]$. Because they are not $t$-alignable, each of them must be part of some comparison in $C$. Since we added the interval $(S_i[x_i^\ell],S_i[x_i^h])$ to the constraint data structure CDS, in later iterations $t$ will never hit the same interval again. Each comparison involving one variable will be charged at most 3 times: one from below the variable, one from the above the variable, and perhaps one when the variable is output.

**Proof of claim.** Suppose to the contrary that at least one member of every pair $S_i[x_i^\ell]$ and $S_i[x_i^h]$, $i\in[m]$, *is* $t$-alignable. Let $v(i)\in\{\ell,h\}$ such that $S_i[x_i^{v(i)}]$ is $t$-alignable, $i\in[m]$. Let $i_0$ be such that $S_i[x_i^{v(i_0)}]\neq t$. The value $i_0$ must exist because $t$ is not an output. First, by assigning $S_i[x_i^{v(i)}]=t$ for all $i$, we obtain an instance satisfying the certificate for which

$$S_1[x_1^{v(1)}]=S_2[x_2^{v(2)}]=\cdots S_m[x_m^{v(m)}]. \tag{6}$$

Second, by assigning $S_i[x_i^{v(i)}]=t$ for all $i\neq i_0$, we obtain an instance also satisfying the certificate for which (6) *does not* hold! This contradicts the certificate definition; hence, the claim holds.

We have already discussed how the constraint data structure CDS can be implemented so that insertion takes amortized constant time in the number of intervals inserted, and querying (for a new probe point $t$) takes constant time. Given a probe point $t$, searching for the values $x_i^h$ and $x_i^\ell$ takes $O(\log N)$-time, for each $i\in[m]$. Hence, each iteration of the algorithm takes time at most $O(m\log N)$. □

**Remark H.5.** In fact, if we implement Minesweeper using the galloping/leapfrogging strategy shown in [20] and [53], then we can speed up the search for the values $x_i^h$ and $x_i^\ell$ of Algorithm 2 slightly in terms of asymptotic runtime. Those ideas in fact work very well in practice! However, they are regarded as "implementation details" in this paper and will not be discussed further. We are happy with a log-factor loss.

# I End-to-end results for the bowtie query

To illustrate the key ideas of the analysis of Minesweeper, we present in this section the second end-to-end set of results on a query that is slightly more complex than the intersection query from Section H. The hope is, without burdening the reader with the heavy notation from the general algorithm, the so-called *bowtie query* is able to illustrate many key ideas. We will define what the query is, what are arguments and certificates for this query, what are the constraints and the CDS, the outer algorithm, and finally the analysis. It turns out that the CDS for this query is very simple. Additionally, the bowtie query *is* $\beta$-acyclic, and *any* GAO is a nested elimination order!

## I.1 The bowtie query, arguments, and certificates

**Definition I.1** (Bow-tie query)**.** The *bow-tie query* is defined as

$$Q_{\bowtie} = R(X) \bowtie S(X,Y) \bowtie T(Y).$$

In this case, $\text{atoms}(Q_{\bowtie}) = \{R,S,T\}$, $\bar{A} = (X,Y)$, and a tuple $\mathbf{t} = (x,y)$ is in the output if and only if $x \in R$, $(x,y) \in S$, and $y \in T$.

Due to symmetry, the global attribute order (GAO), without loss of generality, can be assumed to be $(X,Y)$. The relations $R$, $S$, and $T$ are assumed to be already indexed, allowing for the following kind of access.

- $R[*]$ is the set of all values in $R$.
- $R[i]$ is the $i$th smallest value in $R$, where $i$ is the index and the value $R[i]$ belongs to the domain $\mathbf{D}(X)$ of attribute $X$.
- Similarly, $T[*] = T$, and $T[j] \in \mathbf{D}(Y)$ is the $j$th value in $T$.
- $S[*]$ is the set of all $X$-values in $S$
- $S[i]$ is the $i$th smallest $X$-value in $S$
- $S[i,*]$ is the set of all $Y$-values among tuples $(x,y) \in S$ with $x = S[i]$.
- $S[i,j]$ is the $j$th $Y$-value among all tuples $(x,y) \in S$ with $x = S[i]$.

In the above, when we say $i$th smallest value we use the set semantic. There is no duplicate value and thus no need to break ties. We next specialize notions of argument and certificate to this particular query in order to illustrate these concepts.

**Definition I.2** (Argument for $Q_{\bowtie}$)**.** An *argument* for the bow-tie query $Q_{\bowtie} = R(X) \bowtie S(X,Y) \bowtie T(Y)$ is a set of comparisons in one of the following three formats:

$$\begin{aligned}
R[i_r] &\;\theta\; S[i_s], && \text{(a comparison on } X\text{-value)}\\
S[i_s,j_s] &\;\theta\; T[j_t], && \text{(a comparison on } Y\text{-value)}\\
S[i_s,j_s] &\;\theta\; S[i'_s,j'_s]. && \text{(a comparison on } Y\text{-value between } S\text{-tuples)}
\end{aligned}$$

where $\theta \in \{<,=,>\}$ is called a *comparison*.

Since the $X$-values in $R$ and $Y$-values in $T$ are distinct, there was no need to allow for comparisons between tuples in $R$ or between tuples in $T$. Allowing for such comparisons does not change our analysis.

**Definition I.3** (Certificate for $Q_{\bowtie}$)**.** For the bow-tie query, an *argument* $\mathcal{A}$ is called a *certificate* if the following conditions hold. Let $R'(X), S'(X,Y), T'(Y)$ be three arbitrary relations such that

$$\begin{aligned}
|R'| &= |R|\\
|T'| &= |T|\\
|S'[*]| &= |S|\\
|S'[i,*]| &= |S[i,*]|,\ \forall i, 1 \leqslant i \leqslant |S[*]|
\end{aligned}$$

and that the $R', S', T'$ and $R, S, T$ satisfy all the comparisons in $\mathcal{A}$. Then, for any triple $\{i, (j,k), \ell\}$,

$$R[i] = S[j] \text{ and } S[j,k] = T[\ell]$$

if and only if

$$R'[i] = S'[j] \text{ and } S'[j,k] = T'[\ell].$$

Such a triple is called a *witness* for the instance $R, S, T$; and, it is also a witness for the instance $R', S', T'$.

Following the lead from Example B.3, it is not hard to construct an example showing that comparisons between $Y$-variables between tuples in $S$ are sometimes crucial to reduce the overall certificate size.

## I.2 Constraints and the CDS

For the bow-tie query $Q_{\bowtie}$ every constraint is of one of the following three forms:

- $\langle (a,b), * \rangle$,
- $\langle p, (a,b) \rangle$,
- or $\langle *, (a,b) \rangle$.

where $p \in \mathbb{N}$, $a, b \in \{-\infty, +\infty\} \cup \mathbb{N}$.

A tuple $\mathbf{t} = (x,y)$ satisfies the constraint $\langle (a,b), * \rangle$ if $x \in (a,b)$; it satisfies the constraint $\langle p, (a,b) \rangle$ if $x = p$ and $y \in (a,b)$; and it satisfies the constraint $\langle *, (a,b) \rangle$ if $y \in (a,b)$.

Each constraint can be thought of as an "interval" in the following sense. The first form of constraints consists of all two-dimensional (integer) points whose $X$-values are between $a$ and $b$. We think of this region as a 2D-interval (a vertical strip). Similarly, the second form of constraints is a 1D-interval, and the third form of constraints is a 2D-interval (a horizontal strip).

We store these constraints using a two-level tree data structure (which is a ConstraintTree specialized to the two attribute case). Figure 5 illustrates the data structure.

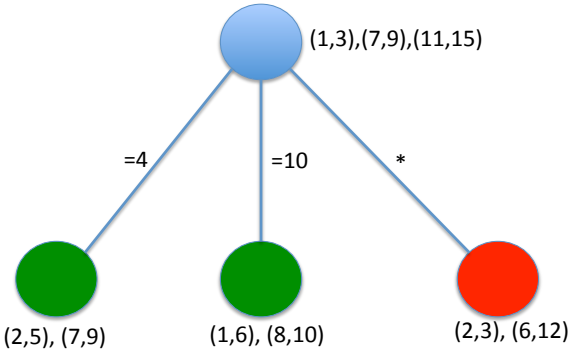



Figure 5: Constraint tree data structure for the bow-tie query

In the first level (the root node), there is a collection of intervals indicating the ruled out $X$-values. Then there are branches to the second level. Each branch is marked with an $*$, or $= p$ for some integer $p$ that does not belong to any interval stored at the root.

In the second level of the tree, every node has a collection of open intervals. Intervals belonging to the same node are merged when they overlap. So, the collection of intervals at each node are disjoint too. To analyze the cost of merging we use a credit based argument. Every inserted interval is given one credit. We use the trick of giving the low end and the high end of an interval half a credit to pay for the merging of two intervals. If the second level of a $[= p]$-branch covers the entire domain, then the $[= p]$-branch is turned into a $\langle (p-1, p+1), * \rangle$ constraint that can

further be merged (at the root level). Effectively, the interval $(p-1, p+1)$ is inserted into the interval list of the root node. We will give an extra credit to the $[= p]$-branch so that when the branch is turned into a $\langle (p-1, p+1), * \rangle$ interval both of the end points has half a credit as any other interval of the root.

Inserting a new constraint takes amortized logarithmic time, as we keep the branches sorted, and the new constraint might "consume" existing intervals. (This logarithmic factor can be improved to constant time if the Minesweeper algorithm and the constraint data structure work in concert, but we will not dig deeper into this detail at this point. We rather keep the description generic, and separate as much as possible the inner workings of the algorithm from the data structure.) In amortized constant time, the data structure CDS is able to report a new tuple $\mathbf{t} = (x, y)$ that does not satisfy any of its constraints, or correctly report that no such $\mathbf{t}$ exists. To find $\mathbf{t}$, we apply the following strategy:

- We first find $x$ such that $x$ does not belong to any root-level interval. This value of $x$, if it exists, can easily be found in constant time by taking the right end point of the lowest interval from the interval list at the root. (Recall the invariant that intervals are disjoint!). If there is no first level interval, then we can set $x = -1$
- If $x$ is found and there is no $[= x]$ branch, then we find a value $y$ that does not belong to any second level interval on the $*$-branch. If there is no $*$ branch, then set $y = -1$. If no $y$ exists then no such $\mathbf{t}$ exists, the algorithm terminates.
- If $[= x]$ is a first-level branch, we find a value $y$ under the $[= x]$-branch that does not belong to any interval under that branch. We call such a $y$ a "free value" $y$. The tuple $\mathbf{t} = (x, y)$ might still violate a $\langle *, (a, b) \rangle$ constraint in the $*$-branch. In that case, we insert the constraint $\langle x, (a, b) \rangle$ into the tree. Then we find the next smallest "free" value $y$ under the $[= x]$-branch again and continue with the "ping-pong" with the $*$-branch until a good value of $y$ is found. The intervals under $[= x]$-branch might be merged with an interval taken from the $*$-branch, but if we give each constraint $\langle x, (a, b) \rangle$ a constant number of credits, we can pay for all the merging operations.

To summarize, insertion and querying for a new probe point into the above data structure takes at most logarithmic time in the amortized sense (over all operations performed).

## I.3 The outer algorithm

We next describe the outer algorithm of Minesweeper specialized to the bowtie query. The key of Algorithm 9 is the loop. We begin with a point $\mathbf{t} = (x, y)$ that has not been ruled out by the constraint data structure at this point of the algorithm. Intuitively, our goal is to determine if $\mathbf{t}$ is in the output; if $\mathbf{t}$ it is not in the output, then our goal is to find some gap that could rule out $\mathbf{t}$. We find the gaps by probing each of the relations $R$, $S$, and $T$ "around" the point $\mathbf{t}$.

The probes of $R$ and $T$ are straightforward: we find gaps around the value of $x$ and $y$ in each relation. With respect to $S$, if we only needed to verify that $\mathbf{t}$ is not an output tuple then a gap around the value $(x, y)$ in $S$ would suffice. However, we do a bit more work in Lines 5 to 7; the reason for that is explained below. Before then, observe that if $\mathbf{t}$ is in the output then $S[i_S^h, i_S^{hh}] = (x, y)$ (in fact, $S[i_S^z, i_S^{zz'}] = (x, y)$ for all $z, z' \in \{l, h\}$). Thus, the condition in Line 8 is a sound and complete check for $\mathbf{t}$ to be in the output.

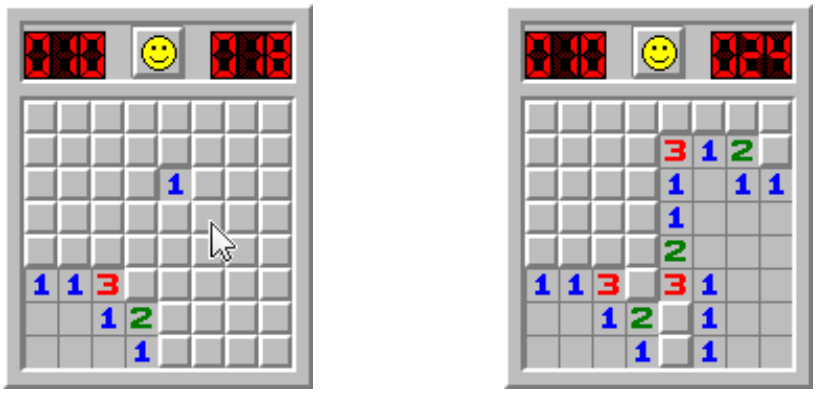
As we noted, Minesweeper does more work than is necessary to certify that $\mathbf{t}$ is not an output tuple. This is because Minesweeper is not only searching for a gap that contains $\mathbf{t}$ to rule $\mathbf{t}$ out, but it also "looks for" a variable that is involved in *any* certificate (including the optimal certificate); this is done so that the algorithm does not explore too deep into – and thus spend too much time in – regions that the optimal certificate wastes very few comparisons to rule out. At the same time, we also have to ensure that we don't explore too many gaps just to capture one single comparison in the optimal certificate. Minesweeper steps on this fine line roughly as follows.

Since $\mathbf{t}$ could have been excluded by a gap that does not have $(x, y)$ on its boundary, Minesweeper– analogously to the eponymous windows game when one chooses a square without a bomb (see Figure 6) – finds the biggest gaps around the point $\mathbf{t}$. In this case, we observe that for each $\mathbf{t}$, Minesweeper examines a constant number of gaps. From Lines 11 onward, we simply insert all the gaps that we found above. Later, we will reason that *some* comparison in any optimal certificate is found by at least one of the FindGap searches. In this example, we explored a constant number of gaps (here 5) for every $\mathbf{t}$; for more complex queries, the number of gaps may grow, but it will grow with the number of attributes in $Q$, i.e., the number of gaps we explore does *not* depend on the data.

**Algorithm 9** Minesweeper for evaluating the bow-tie query $R(X) \bowtie S(X,Y) \bowtie T(Y)$.

**Input:** Following the conventions stipulated in (1) and (2), the following are implicit:
**Input:** $R[0] = S[0] = T[0] = -\infty$ ▷ out-of-range indices
**Input:** $R[|R|+1] = T[|T|+1] = S[|S[*]|+1] = +\infty$ ▷ out-of-range indices
**Input:** $S[i,0] = -\infty, S[i,|S[i,*]|+1] = +\infty, \forall 1 \leqslant i \leqslant |S[*]|$ ▷ out-of-range indices

**While** $((\mathbf{t} \leftarrow \text{CDS.GETPROBEPOINT}()) \neq \text{NULL})$ **do**
  Say $\mathbf{t} = (x,y)$
  $(i_R^\ell, i_R^h) \leftarrow R.\text{FINDGAP}((), x)$
  $(i_T^\ell, i_T^h) \leftarrow T.\text{FINDGAP}((), y)$
  $(i_S^\ell, i_S^h) \leftarrow S.\text{FINDGAP}((), x)$
  $(i_S^{\ell\ell}, i_S^{\ell h}) \leftarrow S.\text{FINDGAP}((i_S^\ell), y)$
  $(i_S^{h\ell}, i_S^{hh}) \leftarrow S.\text{FINDGAP}((i_S^h), y)$
  **If** $(R[i_R^h] = S[i_S^h] = x$ **and** $S[i_S^h, i_S^{hh}] = T[i_T^h] = y)$ **then** ▷ Not true if any index is out of range
    **Output** the tuple $\mathbf{t} = (x,y)$
    $\text{CDS.INSCONSTRAINT}(\langle x, (y-1, y+1)\rangle)$
  **else**
    $\text{CDS.INSCONSTRAINT}\left(\left\langle \left(R[i_R^\ell], R[i_R^h]\right), *\right\rangle\right)$ ▷ Interval on $X$
    $\text{CDS.INSCONSTRAINT}\left(\left\langle \left(S[i_S^\ell], S[i_S^h]\right), *\right\rangle\right)$ ▷ Interval on $X$
    $\text{CDS.INSCONSTRAINT}\left(\left\langle *, \left(T[i_T^\ell], T[i_T^h]\right)\right\rangle\right)$; ▷ Interval on $Y$
    **If** ($i_S^h$ is not out of range) **then**
      $\text{CDS.INSCONSTRAINT}\left(\left\langle S[i_S^h], \left(S[i_S^h, i_S^{h\ell}], S[i_S^h, i_S^{hh}]\right)\right\rangle\right)$
    **If** ($i_S^\ell$ is not out of range) **then**
      $\text{CDS.INSCONSTRAINT}\left(\left\langle S[i_S^\ell], \left(S[i_S^\ell, i_S^{\ell\ell}], S[i_S^\ell, i_S^{\ell h}]\right)\right\rangle\right)$



(a) (b)

Figure 6: In the classic game Minesweeper, when you click on a square without a bomb (a), it can reveal many other squares (b), which is analogous to our gap exploration. See `http://en.wikipedia.org/wiki/Minesweeper_(video_game)` for a history of this classic game.

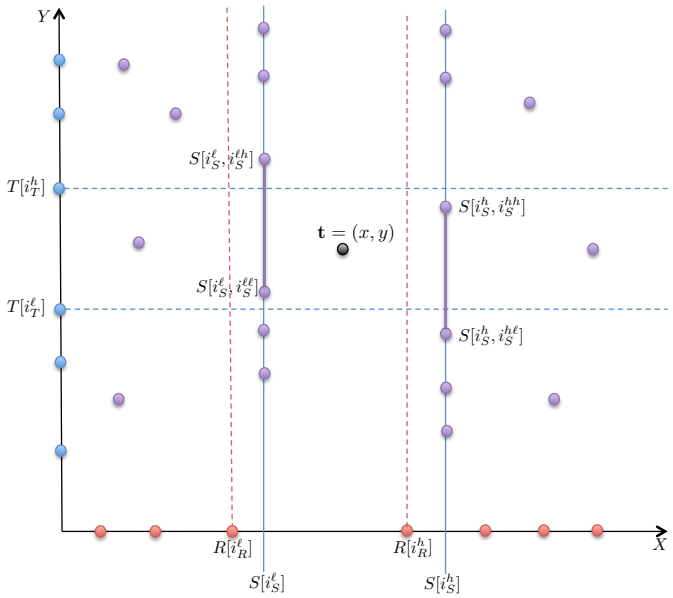



Figure 7: Illustration for Algorithm 9

Algorithm 9 presents the formal details of the algorithm. Notations are redefined here for completeness. Let $|R|, |S|, |T|$ denote the number of tuples in the corresponding relations, $S[*]$ the set of $X$-values in $S$, $S[i]$ the $i$'th $X$-value, $S[i,*]$ the set of $y$'s for which $(S[i],y) \in S$, and $S[i,j]$ the $j$'th $Y$-value in the set $S[i,*]$.

Figure 7 illustrates the choices of various parameters in the algorithm. Some of the constraints might look unintuitive and perhaps redundant at first. The picture shown in Figure 7 should give the reader the correct geometric intuition behind the constraints: all points in $O$ satisfying the constraints are guaranteed to be not part of the output.

To convey the subtlety in gap exploration, let us consider a simple idea. The first candidate gap from $S$ that comes to mind is perhaps the gap between $(x_-, y_-)$ and $(x_+, y_+)$ where $(x_-, y_-)$ is the largest tuple in $S$ that is smaller than $(x,y)$ lexicographically, and $(x_+, y_+)$ is the smallest tuple in $S$ that is greater than $(x,y)$ lexicographically. The problem with this simple idea is that this gap might actually fail to capture *any* variable involved in an optimal certificate comparison at all. Consider, for example, the following input:

$$\begin{aligned} R[X] &= \{2\}, \\ T[Y] &= \{N+1\}, \\ S[X,Y] &= \{(1, N+1+i) \mid i \in [N]\} \cup \{(3,i) \mid i \in [N]\}. \end{aligned}$$

The certificate $C = \{S[1,1] > T[1], S[2,N] < T[1]\}$ is an optimal certificate for this instance. Suppose the probe point is $\mathbf{t} = (2, N+1)$, then $(x_-, y_-) = (1, 2N+1)$ and $(x_+, y_+) = (3,1)$ both of which do not have anything to do with the optimal certificate above.

## I.4 Analysis

We next show Algorithm 9 is near instance optimal (modulo the time spent in the CDS). This Theorem parallels the content of Theorem 3.2 so the reader can map back this special case to the general statement.

**Theorem I.4** (runtime of Minesweeper on $Q_{\bowtie}$)**.** *Let $N$ denote the total number of tuples from the input relations, $Z$ the total number of output tuples. Let $C$ be an arbitrary certificate for the input query. Then, the total runtime of Algorithm 9 is*

$$O((|C| + Z)\log(N) + T(CDS)),$$

*where $T(CDS)$ is the total time it takes the constraint data structure. The algorithm inserts a total of $O(|C|+Z)$ constraints to CDS and issues $O(|C|+Z)$ calls to* getProbePoint.

*Proof.* We show that the number of iterations of Algorithm 9 is $O(|C|+Z)$. Since the amount of work done in each iteration is $O(\log N)$ and the number of calls to getProbePoint() and the number of inserted constraints are linear in the number of iterations, the proof is complete.

We "pay" for each iteration of the algorithm, represented by the tuple $\mathbf{t} = (x,y)$ that the CDS returns in the while predicate for that iteration, by "charging" either an output tuple or a pair of comparisons in the certificate $C$. We show that each output tuple and each comparison will be charged at most $O(1)$ times. To this end, we define a couple of terms.

Any variable

$$e \in \left\{R[i_R^h], R[i_R^\ell], S[i_S^h], S[i_S^\ell]\right\}$$

is said to be **t**-*alignable* if either $e$ is already equal to $x$ or $e$ is not involved in any comparison in the certificate $C$. By convention, any variable whose index is out of range is **not** **t**-alignable. The semantic of **t**-alignability is as follows. If a **t**-alignable variable $e$ is not already equal to $x$, setting $e = x$ will transform the input into another database instance satisfying all comparisons in $C$ without violating the relative order in the relation that $e$ belongs to.

A variable

$$e \in \left\{T[i_T^h], T[i_T^\ell]\right\}$$

is said to be **t**-*alignable* if either $e$ is already equal to $y$ or $e$ is not involved in any comparison in the certificate $C$. A variable

$$e \in \left\{S[i_S^h, i_S^{h\ell}], S[i_S^h, i_S^{hh}]\right\}$$

is **t**-alignable if $S[i_S^h]$ is **t**-alignable and either $e$ is already equal to $y$ or $e$ is not part of any comparison in the certificate $C$. Similarly, we define **t**-alignability for a variable

$$e \in \{S[i_S^\ell, i_S^{\ell\ell}], S[i_S^\ell, i_S^{\ell h}]\}.$$

Next, we describe how to "pay" for the tuple $\mathbf{t} = (x,y)$.

**Case 1** Line 9 is executed. We pay for **t** by charging the output tuple **t**. The constraint added in line 10 ensures that we won't have to pay for the same output **t** again.

**Case 2** The else part (line 11) is executed, i.e. **t** is not an output tuple. We claim that one of the following five cases must hold:

(1) both $R[i_R^h]$ and $R[i_R^\ell]$ are not **t**-alignable,

(2) both $S[i_S^h]$ and $S[i_S^\ell]$ are not **t**-alignable,

(3) both $S[i_S^h, i_S^{h\ell}]$ and $S[i_S^h, i_S^{hh}]$ are not **t**-alignable,

(4) both $S[i_S^\ell, i_S^{\ell\ell}]$ and $S[i_S^\ell, i_S^{\ell h}]$ are not **t**-alignable,

(5) both $T[i_T^h]$ and $T[i_T^\ell]$ are not **t**-alignable.

Suppose otherwise that at least one member in each of the five pairs above is **t**-alignable. For example, suppose the following variables are **t**-alignable:

$$R[i_R^h], S[i_S^\ell], S[i_S^\ell, i_S^{\ell\ell}], T[i_T^h].$$

Then, we construct two database instances as follows.

- *Database instance $I$.* In this instance, we keep all current variable values except that we set $R[i_R^h] = S[i_S^\ell] = x$ and $S[i_S^\ell, i_S^{\ell\ell}] = T[i_T^h] = y$. Then, clearly in this instance the set of index tuples $\left\{i_R^h, (i_S^\ell, i_S^{\ell\ell}), i_T^h\right\}$ is a witness for $Q_{\bowtie}(I)$.

- *Database instance J*. This instance requires a little bit more care. Recall that we are in the case when **t** is *not* an output tuple. Hence, it cannot possibly be the case that $R[i_R^h] = S[i_S^\ell] = x$ and $S[i_S^\ell, i_S^{\ell\ell}] = T[i_T^h] = y$ already. Assume, for example, that $S[i_S^\ell, i_S^{\ell\ell}] \neq y$. Then, the database instance $J$ is constructed by setting $R[i_R^h] = S[i_S^\ell] = x$ and $T[i_T^h] = y$. Then, in this case $\left\{i_R^h, (i_S^\ell, i_S^{\ell\ell}), i_T^h\right\}$ is **not** a witness for $Q_{\bowtie}(J)$.

Note that both $I$ and $J$ satisfy $C$. Hence, we reach a contradiction because $C$ is a certificate. The claim is thus proved.

The key idea is, by definition each non-out-of-range variable $e$ that is not **t**-alignable must be involved in a comparison in the certificate $C$. There is an exception, something like $S[i_S^\ell, i_S^{\ell\ell}]$ might be non-**t**-alignable because its prefix variable $S[i_S^\ell]$ is not alignable. But in that case $S[i_S^\ell]$ must be involved in a comparison in the certificate, and that's all we need for the reasoning below.

Instead of charging a comparison, we will charge a non-out-of-range non-**t**-alignable variable. **If** each non-out-of-range non-**t**-alignable variable is charged $O(1)$ times, then each comparison will be charged $O(1)$-times.

We pay for **t** by charging any pair of non-out-of-range non-**t**-alignable variables out of the five pairs above that are involved in comparisons in $C$. We call each of those five pairs an "interval." The pairs of the type (1), (2), and (5) are 2D-intervals, and the pairs of the type (3), (4) are 1D-intervals.

Due to the constraints added on lines 14, 16, and 18, the 2D-intervals are charged at most once. Since two 2D-intervals might share an end point, each non-out-of-range non-**t**-alignable variable from a 2D-interval might be charged twice. The 1D-intervals are charged at most twice. Each non-out-of-range non-**t**-alignable variable from a 1D-interval might be charged at most four times. Consequently, each comparison is charged $O(1)$ times. □

# J Counter examples

In this section, we present a family of $\beta$-acyclic join queries and instances on which none of Leapfrog-Triejoin [53] (LFTJ henceforth), the algorithm of [40] (NPRR henceforth) or Yannakakis' algorithm [55] are instance optimal (i.e. they don't run in $O(|C|+Z)$-time). Furthermore, the gap between those algorithms and Minesweeper can be arbitrarily large.

To simplify the argument, let us first consider a simpler family of instances. The query is the following.

$$Q = \bowtie_{i=1}^{m} R_i(A_i, A_{i+1}).$$

The above query is $\beta$-acyclic, and the GAO $A_1, \ldots, A_{m+1}$ is a nested elimination order. The main idea behind the instance is to "hide" the certificate along a long path in the query. All three algorithms NPRR, LFTJ, and Yannakakis do not explore the attributes globally as Minesweeper does, and hence they will get stuck looking for *many* partial tuples that do not contribute to the output.

The relations are constructed as follows. Each attribute $A_i$ will have as its domain the set $[mM]$, where $m \geqslant 5$ and $M$ is a large positive integer. Each relation $R_i$ will have $m$ "chunks," where the $j$th chunk ($1 \leqslant j \leqslant m$) is a subset of the set

$$[(j-1)M+1, jM] \times [(j-1)M+1, jM].$$

More precisely, relation $R_i$ is defined as follows.

- For every $j \in [m] - \{i, i-1\}$, the $j$th chunk of $R_i$ is exactly exactly

$$[(j-1)M+2, jM] \times [(j-1)M+2, jM].$$

- The $i$th chunk consists of one single tuple $\Big((i-1)M+1, (i-1)M+1\Big)$.
- And the $(i-1)$'th chunk is empty.

If $i = 1$, then we interpret $i-1$ as $m$. Namely, the $m$th chunk of $R_1$ is empty. Note that every relation is of size $N = \Theta(mM^2)$.

It is not hard to see that the output of the above instance is empty. Furthermore, there is a certificate of size $O(mM)$. Hence, by Theorem 2.7 Minesweeper takes $O(mM \log M)$ time since $A_1, \ldots, A_{m+1}$ is a nested elimination order.

This certificate consists of the following comparisons

$$\begin{aligned}
R_1[1,1] &< R_2[1] \\
R_1[i,1] &> R_2[1], \text{ for } i > 1 \\
R_2[i,1] &> R_3[M+1], \text{ for } i > 1 \\
R_3[i,1] &> R_4[2M+1], \text{ for } i > M+1 \\
R_4[i,1] &> R_5[3M+1], \text{ for } i > 2M+1 \\
&\vdots \quad \vdots \quad \vdots \\
R_{m-1}[i,1] &> R_m[(m-2)M+1], \text{ for } i > (m-3)M+1.
\end{aligned}$$

To see why this is a certificate that the output is empty, consider an arbitrary witness for this instance: $X = \{(i^{(1)}, j^{(1)}), \ldots, (i^{(m)}, j^{(m)})\}$. From the first two (sets of) inequalities above, we know $i^{(2)} > 1$. Then, from the next inequality we know $i^{(3)} > M+1$. This inference goes on until the last inequality, which does not leave any room for $j^{(m)}$. Hence, such a witness cannot exist.

Now, for every $i \in [m]$, the semijoin $R_i \ltimes R_{i+1}$ has size $\Omega(mM^2)$. Hence, Yannakakis algorithm runs in time at least $\Omega(mM^2)$.

For LFTJ and NPRR, it takes slightly more work to be rigorous, but the key ideas are as follows. Consider *any* attribute ordering that LFTJ adopts. Say the attribute ordering is $A_{i_1}, \ldots, A_{i_{m+1}}$, for some permutation $\{i_1, \ldots, i_{m+1}\}$ of $[m+1]$. LFTJ will compute the intersection on $A_{i_1}$, and for each value $a$ in the intersection it will compute the join on $A_{i_2}$ using $a$ as an anchor, and so on. If $|i_1 - i_2| > 1$, then clearly the runtime is $\Omega(mM^2)$ because the intersection on each attribute is $\Omega(mM)$. If $|i_1 - i_2| = 1$, then LFTJ will go through every tuple in the input relation $R(A_{i_1}, A_{i_2})$ after it is semijoin-reduced on $A_{i_1}$ and $A_{i_2}$. And even after two such reductions, the size of the relation is still $\Omega(mM^2)$. The algorithm NPRR suffers the same drawback.

There are two potential unsatisfactory aspects of the instance above: (i) The gap between Minesweeper and the worst-case algorithms is only quadratic and (ii) the example only considers path type queries. Next, we handle these two shortcomings.

We can increase the gap in the above example by considering the following join query:

$$Q = R_1(A_1, \ldots, A_k) \bowtie R_2(A_2, \ldots, A_{k+1}) \bowtie \cdots \bowtie R_m(A_m, \ldots, A_{m+k-1}).$$

Then, each relation $R_i$ still has $m$ blocks like before, where the $j$th block for $j \in [m] - \{i, i-1\}$ is $[(j-1)M+2, jM]^k$. For $j = i$ the block has only one tuple $((i-1)M+1)^{(k)}$, and the $(i-1)$th block is empty. It is not hard to see that there is a certificate of size $O(mM)$, and all three algorithms Yannakakis, NPRR, and LFTJ run in time at least $\Omega(mM^k)$. The reasoning is basically identical to the previous example.

Finally, we tackle the class of $\beta$-acyclic queries for which our quadratic gap holds. We note that our argument holds for any $\beta$-acyclic query into which we can embed the 5-path query. In other words, as long as a $\beta$-acyclic query $Q$ has attributes $A_{i_1}, \ldots, A_{i_6}$ and relations $R_{i_1}, \ldots, R_{i_5}$ such that $A_{i_j}$ is only present in $R_{i_{j-1}}$ and $R_{i_j}$ (except for the cases $j = 1$ in which case $A_{i_1}$ only exists in $R_{i_1}$ and the case of $j = 6$ in which case $A_{i_6}$ only exists in $R_{i_5}$), we can embed the hard instance above into a hard instance for $Q$, where we extend the values for other attributes and relations as we did in the proof of Proposition 2.8 (in Appendix F.3). Recall that the proof in Appendix F.3 shows that this does not change the certificate size (which implies that the runtime of Minesweeper remains the same) while it is not hard to check that the worst-case optimal algorithms still are quadratically slower. We note that we did not try to optimize the length of the shortest path for which our hard instance still works. However, we note that the argument cannot work for path of length 3 (since LFTJ is instance optimal for the query $R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$, which is essentially the bowtie query). Note that the class of $\beta$-acyclic queries that has a 5-path embedded in it as above is a fairly rich subset of $\beta$-acyclic queries.

# K Our certificate vs. the notion of proof from DLM.

It is perhaps instructive to compare and contrast our notion of certificate from the similar notion of "proof" from DLM. The obvious difference is that our certificate is defined for a natural join query, while DLM's "proof" is only defined for

the $Q_\cap$ query. The difference, however, is subtler than that. Consider only the $Q_\cap$ case. DLM's proof is output-specific while our certificate does not need any specific mentioning of the output at all. For some subset $B$ of *domain values*, they defined a *B-proof* to be an argument for which each value $b \in B$ is certified with a spanning tree of equalities, and each of the set of values in between consecutive values in $B$ must be "emptiness-certified" with an $\varnothing$-proof. An $\varnothing$-proof in DLM is defined in terms of "eliminating" elements by the $<$ comparisons. (See DLM's Lemmas 2.1 and 2.2 and Theorem 2.1.) Our notion of certificate is stronger. For example, our certificate does not need *any* $<$ comparison to certify that the output is empty. For example, consider intersection of three sets $R,S,T$ of the same size $N$. An emptiness certificate may consist of two *equalities*: $\{(S[N] = R[1]), (R[2] = T[1])\}$. It is obvious that any instance satisfying those two inequalities must have an empty intersection because we can infer that $S[N] < T[1]$. Subtler than that, Example B.3 points to the fact that the use of equalities can asymptotically reduce the certificate size even when the output is empty.

# L The Triangle Query

The goal of this section is to prove Theorem 5.4. We begin with the data structure CDS.

## L.1 The CDS

We begin by collecting some properties of a data structure that maintain interval lists that will be useful in our new definition of CDS.

### L.1.1 Dyadic Tree for Intervals

Let $N = 2^d$ for ease throughout.

- The data structure consists of nodes that contain an interval list. We index the nodes of the tree by binary strings of length less than or equal to $d$, i.e,.
$$x \in \{0,1\}^{\leqslant d} = \cup_{j=0}^{d} \{0,1\}^j$$
e.g., $x = ()$ is the root, $x = (0)$ is the left child, and while $(1,1,\ldots,1)$ is the rightmost leaf. We can think of string of length $j$ as denoting the interval each as a binary expansion of a value, i.e., $[b(x)2^{d-j}, (b(x)+1)2^{d-j})$.
- Each node $x \in \{0,1\}^{\leqslant d}$ is associated with an interval list on domain $[N]$ denoted $I(x)$.
- Given an interval $[a_1,a_2]$, we will need its dyadic decomposition, i.e., one of the intervals above and we denote it
$$d([a_1,a_2]) = \{J_1,\ldots,J_k\} \text{ and } [a_1,a_2] = \bigcup_{i=1}^{k} J_i \text{ where } J_i = [b(x)2^{d-j}, (b(x)+1)2^{d-j}) \text{ for } x \in \{0,1\}^{\leqslant d}, j \in [d]$$
For any interval, in $[N]$ we have $k \leqslant 2\log N$. Let $x(J_i)$ be the string associated with the dyadic interval $J_i$.
- For every dyadic interval node $x$, there is an IntervalList $I(x)$. We also insist that any consecutive interval $[b_1,b_2]$ is stored as the collection $d([b_1,b_2])$. (This makes the upcoming arguments simpler.)
- insert$([a_1,a_2],[b_1,b_2])$: For each $J \in d([a_1,a_2])$, set
$$I(x(J)) \leftarrow I(x(J)) \cup [b_1,b_2]$$
This takes $O(\log^2 N)$ since there are $O(\log N)$ dyadic intervals $J$ and insertion into an IntervalList takes $O(\log N)$ time.

- INTERSECT$(x,[b_1,b_2])$ for dyadic $[b_1,b_2]$: Returns the interval list for $I(x) \cap [b_1,b_2]$. This can be done in $O(y \cdot \log N)$ time, where $y$ is the total number of (dyadic) intervals that need to be output. Note that to perform this task one has to find the correct position of $b_1$ and $b_2$ in $I(x)$ (which can be done in $O(\log N)$ time) and then returning the corresponding intervals. There are two cases: (i) The output is $[b_1,b_2]$, in which case we "charge" the intersection to this interval (note that we have $y = 1$ in this case) or (ii) The output is a subset of intervals from $I(x)$: in this case we "charge" the intersection to these set of intervals. (Note that we can only have $y > 1$ in case (ii).) This charging scheme will be useful in the proof of Proposition L.1 below.

### L.1.2 The New CDS

For notational convenience we will denote the interval lists for prefixes $\epsilon, (= a), (*), (= a, *), (*, = b)$ and $(= a, = b)$ simply as $I(), I(*), I(= a, *), I(*, = b)$ and $I(= a, = b)$.

Our new CDS would be very similar to our earlier CDS in Appendix E except in the following way:

The $*$-branch for variable $A$ would be replaced by a dyadic tree containing all the intervals of the form $\langle *, b, [c_1,c_2]\rangle$. In other words in addition to maintaining the interval lists $I(*, = b)$, it will also maintain for dyadic interval $x$ of $[N]$ the interval list $I(*, = x)$, which will always satisfy the following invariant:

$$I(*, x) = I(*, = x \circ 0) \cap I(*, = x \circ 1). \tag{7}$$

Not surprisingly, we will maintain the interval lists $I(*, = x)$ as a dyadic tree as outlined in the previous section. For the rest of the argument we will show that the INSERT operation on the new CDS can be done in amortized $O(\log^3 N)$ time. For all constraints except of the form $\langle *, b, [c_1,c_2]\rangle$ the INSERT can be done in amortized $O(\log N)$ time by Proposition 3.1. So we only need to show the following:

**Proposition L.1.** *Given a dyadic tree on domain $N$ with $M$ insertions of the form $\langle *, b, [c_1,c_2]\rangle$, then* (7) *can be ensured in time $O(M \log^3 N)$.*

*Proof.* We use the natural algorithm to implement the insert of constraints of the form $\langle *, b, [c_1,c_2]\rangle$, which we outlined next. Let $L$ be $[c_1,c_2] \setminus I(*, = b)$. Then perform our original $I(*, = b)$.INSERT$([c_1,c_2])$. Then do the following for every $J \in L$. Let $L' \leftarrow$ INTERSECT(SIBLING$(b), J)$, where SIBLING$(b)$ is the sibling of $b$ in the dyadic tree. If $L'$ is empty we stop otherwise we recurse with this algorithm at the parent $x$ of $b$ (where we want to insert all intervals in $L'$ into $I(*, = x)$).

We first assume that the algorithm only deals with dyadic intervals throughout. We claim that under this assumption we would be done by assigning $O(\log^2 N)$ credits to each inserted interval. To see this first consider the simple case, where we always have $|L| = |L'| = 1$, i.e. we always need to insert one interval. In this case, for each recursive level we do $O(\log N)$ amounts of work and we have $O(\log N)$ recursive calls (up the path in the dyadic tree from the leaf corresponding to $b$ to the root) overall. We expand a bit on the $O(\log N)$ work on each recursive call. Recall that the analysis of the INTERSECT procedure: we first need $O(\log N)$ work to figure out the correct position of $J$ in the INTERVALLIST of SIBLING$(b)$. To pay for this we use up credits from $J$. We still have to pay for the computation of $L'$. We pay for this by charging $J$ or $L'$ as appropriate. The important point to note that is that once an interval stops "floating" up, it can only be pushed by fresh intervals that arrive at its sibling node at a later point of time.

For the more general case (but still with dyadic interval), note that any interval that "floats" up is either one of the $M$ inserted intervals or is "sandwiched" between two such inserted intervals. Adjusting the constant for the number of tokens appropriately takes care of this issue.

Finally to handle the general case, we can replace any interval by $O(\log N)$ dyadic intervals. To deal with this we need to increase the number of credits to $O(\log^3 N)$ from the previous $O(\log^2 N)$ credits. This completes the proof. □

Finally, we will also ensure the following:

For every constraint $\langle *, [b_1,b_2], *\rangle$ that is inserted into $I(*)$, we also insert $[N]$ into the interval list $I(*, = J)$ for every $J \in d([b_1,b_2])$.[15]

[15] Note that Proposition L.1 only talks about singleton $b$, it can easily be checked to see that it can handle this more general case. Basically we can make all such $J$ to be the leaves in the dyadic tree and the argument in Proposition L.1 can handle insertions into leaves.

## L.2 The Algorithm

The outer algorithm for $Q_\triangle$ will be the same as in Algorithm 2. We will have to change the GETPROBEPOINT algorithm, which is formally presented in Algorithm 10.

**Algorithm 10** GETPROBEPOINT for evaluating the triangle query $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$.

**Input:** CDS as outlined earlier.

```
1:  i ← 0
2:  While i < 3 do
3:      If i = 0 then                                              ▷ Handling A
4:          a ← I().NEXT(−1)
5:          If a = ∞ then
6:              Return NULL
7:          i ← i + 1
8:      If i = 1 then                                              ▷ Handling B
9:          b ← NEXTUNION(I(= a), I(∗), −1).
10:         If b = ∞ then
11:             i ← 0
12:         else
13:             i ← i + 1
14:     If i = 2 then                                              ▷ Handling C
15:         If I(= a, ∗).NEXT(−1) = ∞ then
16:             CDS.INSCONSTRAINT(⟨(a − 1, a + 1), ∗, ∗⟩)
17:             i ← 0
18:         else                                   ▷ There is a probe point with prefix (a, b)
19:             x ← ϵ                                  ▷ Initializing x to the root
20:             While x is not a leaf do
21:                 z ← GETCACHE(a, x)
22:                 c ← NEXTUNION(I(= a, ∗), I(∗, x), z)
23:                 CACHE(a, x, c)
24:                 If c = ∞ then                          ▷ No viable b in interval x
25:                     CDS.INSCONSTRAINT(⟨= a, x, ∗⟩)
26:                     y ← NEXTSIBLING(x)
27:                     If y = NULL then
28:                         CDS.INSCONSTRAINT(⟨(a − 1, a + 1), ∗, ∗⟩)
29:                         i ← 0
30:                         Exit While loop
31:                     x ← y
32:                 else
33:                     x ← x ∘ 0
34:             If x is a leaf then                            ▷ Found the probe point
35:                 Return (a, b, c)
```

Algorithm 10 uses the following helper functions:

- NEXTUNION$(I_1, I_2, v)$: Finds the smallest value $v' \geqslant v$ that is not covered by $I_1 \cup I_2$. We implement this algorithm by the MERGE algorithm.
- NEXTSIBLING$(x)$: Returns the next node in the dyadic tree by the pre-order traversal (returns NULL if the traversal is done). It is simple to implement: let $x = (x_1, \ldots, x_j)$. Let $1 \leqslant i \leqslant j$ be the largest index such that $x_i = 0$. Then return $(x_1, \ldots, x_{i-1}, 1)$. If no such $i$ exists, return NULL.

- We maintain a data structure, which keeps track of the last "uncovered" value considered by the algorithm in the union of $I(=a,*)$ and $I(*,x)$. The function GetCache$(a,x)$ returns this cached value while Cache$(a,x,c)$ updates the cached value to $c$.

  Further, we will assume that when the outer algorithm outputs a tuple $(a,b,c)$ and adds the constraint $\langle a,b,(c-1,c+1)\rangle$, there is an accompanying call to Cache$(a,b,c+1)$.

It is not hard to see that Algorithm 10 is correct. We state this fact without proof:

**Lemma L.2.** *Algorithm 10 correctly returns a tuple $(a,b,c)$ that is not covered by any existing constraints. If no such tuple exists then it correctly outputs* NULL.

**Lemma L.3.** *Time spent (except those involving backtracking intervals from later part of the algorithm) in Steps 2-9 is $O(|C|\log N)$.*

*Proof.* This follows from the fact that the time spent is in some sense running Algorithm 3 on $R(A,B)\bowtie S(B)\bowtie T(A)$, which is a $\beta$-acyclic query. So our earlier proof can be easily adopted to prove the lemma. □

**Lemma L.4.** *Time spent over prefixes that satisfy the condition in Step 15 is upper bounded by $O(|C|\log N)$.*

*Proof.* This just follows from the fact that the total time spent is bounded by (up to constants):

$$\sum_a |I(=a,*)|\log N \leqslant |C|\log N,$$

as desired. □

The next couple of lemmas need the following definition:

**Definition L.5.** For any $A$ value $a$ define

$$B(a) = \{b | I(=a,*)\cup I(*,=b)\subset [N]\},$$

where $I(=a,*)$ and $I(*,b)$ were the interval list the first time Algorithm 10 deals (i.e. it reaches Step 19) with the prefix $(a,b)$. If Algorithm 10 never reaches Step 19 for some $a$, then define $B(a)=\varnothing$.

We first argue that the number of pairs $(a,b)$ with $b\in B(a)$ is bounded:

**Lemma L.6.**

$$\sum_a |B(a)| \leqslant O(|C|).$$

*Proof.* Note that we only need to consider the values $a$ for which $B(a)\neq\varnothing$. Fix such an arbitrary $a$ and consider an arbitrary $b\in B(a)$. Now when the Algorithm gets to Step 19 we know the following:

- $a\notin I()$
- $b\notin I(=a)\cup I(*)$.
- $I(=a,*)\cup I(*,=b)\subset [N]$

All of the above imply that there exists a $c$ such that the tuple $(a,b,c)$ is not ruled out by the current set of constraints. This implies that Algorithm 10 will return such a tuple $(a,b,c)$ (by Lemma L.2). This probe point will then be used by the outer algorithm to either (i) discover a new constraint or (ii) recognize it as an output tuple.

If there is even one $c$ such that the returned tuple $(a,b,c)$ fall in category (i) above, then note that we can assign a unique inserted constraint to the prefix $(a,b)$ (among all such prefixes that have $c$ such that $(a,b,c)$ falls in category (i)). This by the argument for the runtime of the outer algorithm implies that the number of such prefixes is bounded by $O(|C|)$.

Thus, we only have to consider prefixes $(a,b)$ such that every tuple $(a,b,c)$ returned by Algorithm 10 turns out to be an output tuple. We now claim that each such pair $(a,b)$ must be certified via equalities in the certificate to be present as a tuple in the relation $R(A,B)$. If this were not the case then one can come up with two database instances that satisfy all the comparisons in the certificate but in one instance $(a,b,c)$ is in the output while in the other it is not. This contradicts the definition of a certificate. Thus, we can assign each such prefix with a unique pair (one for $a$ and one for $b$) of equalities in the certificate. Further since each pair involves the tuple $(a,b) \in R$, these assignments are unique and thus, we have the number of prefixes $(a,b)$ such that all its extensions lead to output tuples is upper bounded by $|C|$. This completes the proof. □

**Lemma L.7.** *Total time spent by Algorithm 10 on prefixes $(a,b)$ for which it reached Step 19 is bounded by*

$$O\left(\sum_{(a,x):\ x\in B'(a)} \min(|I(=a,*)|, I(*,=x)|)\log^3 N\right) + O\left(\sum_a |I(=a,*)|\log N\right) + O(Z\log^2 N), \tag{8}$$

*where $B(a) \subseteq B'(a)$ is a set of disjoint dyadic intervals and $|B'(a)| \leqslant O(|B(a)|\log N)$.*

*Proof.* Let us first consider the values $a$ for which we have $I(=a,*) \cup I(*,\epsilon) = [N]$. In this case there is only one iteration of the While loop and Step 28 is executed. Other than Step 22 all the other steps take $O(\log N)$ time. Since NextUnion is implemented as the MERGE algorithm, Step 22 runs in time at most (up to constants):

$$\min(|I(=a,*)|, |I(*,\epsilon)|) \leqslant |I(=a,*)|,$$

since the MERGE algorithm will run till it has skipped over at least one of the two interval lists. Summing up the above run-time for all $a$ such that $I(=a,*) \cup I(*,\epsilon) = [N]$, gives the second term in the claimed runtime.

For the rest of the proof we consider the $a$'s such that $I(=a,*) \cup I(*,\epsilon) \subset [N]$: fix such an arbitrary $a$. Now consider all the possible $B$ values. Mark a $b$ as a *comparison-probe* if there exists a $c$ such that Algorithm 10 returns the tuple $(a,b,c)$, which is used by the outer algorithm to discover a new constraint. We will mark $b$ as an *output-probe* if for *every* $c$ such that Algorithm 10 returns the tuple $(a,b,c)$, it is used by the outer algorithm to discover a new output tuple. For notational convenience we will call $b$ a probe value if it is marked either as a comparisons-probe or an output-probe. Note that there are exactly $|B(a)|$ probe values. Now sort the $B$ values and consider two probe values $b < b'$ such that there are no probe values in $(b.b')$. For the time being assume that $(b,b')$ is dyadic. Note that in this case we execute Step 25 for $x = (b,b')$ and no children of $x$ in the dyadic tree is explored. Now denote certain nodes in the dyadic tree as $\ell_1,\ldots,\ell_m$ for some $m \leqslant O(|B(a)|\log N)$ as follows. Each probe value $b$ (which corresponds to a singleton interval in the dyadic tree) gets its own $\ell_i$. For any two consecutive probe values $b < b'$ (in sorted order of $B$ values) each of $O(\log N)$ dyadic intervals in $(b,b')$ gets its own $\ell_i$. Since there are $|B(a)|$ probe values, there are at most $|B(a)|+1$ intervals of consecutive non-probe values. Further, each such interval gets partitioned into $O(\log N)$ dyadic intervals, which means that we will have $m = O(|B(a)|\log N)$ nodes $\ell_i$ overall, as desired.

Consider the subtree of the dyadic tree whose leaves are $B'(a) \stackrel{\text{def}}{=} \{\ell_1,\ldots,\ell_m\}$. (We will overload notation by referring to the dyadic interval corresponding to $\ell_i$ as just $\ell_i$.) We will show that the total time spent by Algorithm 10 on pairs $(a,x)$ on Step 19 and beyond is bounded by

$$O\left(\sum_{x\in B'(a)} \min(|I(=a,*)|, I(*,=x)|)\log^3 N\right) + O(Z_a\log^2 N), \tag{9}$$

where $Z_a$ is the number of output tuples with $A$ value as $a$. Summing above the above bound over all values of $a$ proves the claimed runtime bound.

To complete the proof, we prove (9). First we note that for a given pair $(a,x)$, the total time spent by Algorithm 10 on Step 22 where $z \neq c$ is upper bounded by (up to constants)

$$\min(|I(=a,*)|, |I(*,x)|)\log N. \tag{10}$$

The above follows from the fact that when $z \neq c$, it means that the MERGE algorithm actually made an advance and that the total number of times we can advance is upper bounded by the size of the shorter list. (Recall that each advance needs a binary search and thus takes $O(\log N)$ time.)

Notice that all Steps other than Step 22 can be implemented in $O(\log N)$ time.[16] Now note that Algorithm 10 (for the given value of $a$) only considers pairs $(a,x)$ such that $x$ contains at least one $\ell_i$. This implies two things. First, due to (10) the total time spent on Step 22 where $z \neq c$ is upper bounded by

$$\sum_{x:\ell_i \subseteq x \text{ for some } i} \min(|I(=a,*)|, |I(*,x)|) \log N. \tag{11}$$

Second, we have that Algorithm 10 exactly traces all paths from the root to one of the $\ell_i$'s. Let us consider the different cases of $\ell_i$:

1. ($\ell_i$ is an interval for which we run Step 25.) If we exclude the time spent from the root to $\ell_i$ that is accounted for in (11), then we essentially go along a path of length $O(\log N)$ doing $O(\log N)$ amount of work at each interval in the path. Thus, the overall time spent on these paths (excluding time spent in (11)) is bounded by (up to constant factors):
$$\sum_{x \in B'(a)} \log^2 N.$$

2. ($\ell_i \in B(a)$ and is marked output-probe.) For such cases note that we will always have $z = c$ on all intervals in the path. So again with an argument as in the last case we spend time at most (up to constant factors)
$$Z_a \log^2 N.$$

3. ($\ell_i \in B(a)$ and is marked comparison-probe.) Note that the number of $c$ values for which we get $(a,b,c)$ probe points is upper bounded by $O(\min(|I(=a,*)|, |I(*,=\ell_i)|))$. Thus, total time spent in this case outside of the time accounted for in (11), by an argument similar to the earlier cases is upper bounded by (up to constants)
$$\sum_{b \in B(a)} \min(|I(=a,*)|, I(*,=b)|) \log^2 N.$$

Adding up the bounds above with (11) implies that the time bound we are after is at most (up to constants)

$$\sum_{x:\ell_i \subseteq x \text{ for some } i} \min(|I(=a,*)|, |I(*,x)|) \log^2 N + Z_a \log^2 N.$$

To complete the proof, we will argue that

$$\sum_{x:\ell_i \subseteq x \text{ for some } i} \min(|I(=a,*)|, |I(*,x)|) \leqslant O\left( \sum_{y \in B'(a)} \min(|I(=a,*)|, |I(*,y)|) \log N \right), \tag{12}$$

since the above will imply (9). To see why the above is true, note that by (7), we have

$$I(*,y) = \cap_{i:\ell_i \subseteq y} I(*,\ell_i),$$

where the intersection is over the set of points covered by the interval lists. This in turn implies that

$$|I(*,y)| \leqslant \sum_{i:\ell_i \subseteq y} |I(*,\ell_i)|.$$

The above in turn implies that

$$\min(|I(=a,*)|, |I(*,y)|) \leqslant \sum_{i:\ell_i \subseteq y} \min(|I(=a,*)|, |I(*,\ell_i)|).$$

Noting that for intervals $y \neq y'$ of the same size, the set of $\ell_i$'s contained in them are disjoint and that there are $O(\log N)$ distinct sizes for dyadic intervals, the above implies (12), as desired. □

[16] The time bound is amortized for Steps 25 and 28. However, the number of times these steps are run is bounded by $\sum_a |B'(a)|$. So the overall time spent on these steps will be bounded by $O(\sum_a |B'(a)| \log N)$, which is subsumed by the bound in (8).

We are finally ready to prove the runtime for Algorithm 10:

**Theorem L.8.** *Over all calls to Algorithm 10 from the outer algorithm, the total time spent is bounded by*

$$O\Big(|C|^{3/2}\log^{7/2}N + Z\log^2 N\Big).$$

*Proof.* The total time spent is bounded by the sum of the time bounds in Lemmas L.3, L.4 and L.7. The first two terms are subsumed by the bound in this lemma. Thus, we only need to bound

$$O\left(\sum_{(a,x):\ x\in B'(a)} \min(|I(=a,*)|, I(*,=x)|)\log^3 N\right) + O\left(\sum_a |I(=a,*)|\log N\right) + O(Z\log^2 N).$$

Since $\sum_a |I(=a,*)| \leqslant |C|$, the last two terms in the sum above are subsumed by the bound in this lemma. So we are left with the bound

$$\sum_a \sum_{x\in B'(a)} \min(|I(=a,*)|, I(*,=x)|). \tag{13}$$

Next we note the following:

$$\sum_a |I(=a,*)| \leqslant |C|,$$

and

$$\sum_{x\in B'(a)} |I(*,=x)| \leqslant \sum_b |I(*,=b)| \leqslant |C|.$$

In the above the first inequality follows from the fact that every $x \neq x' \in B'(a)$ are disjoint and by the argument used in proof of Lemma L.7, $|I(*,x)| \leqslant \sum_{b\in x} |I(*,=b)|$. Then Lemmas L.9, L.6 and L.7 imply that (13) is bounded by $|C|^{3/2}\sqrt{\log N}$, as desired. □

**Lemma L.9.** *For any two vectors $u, v \in \mathbb{R}^M_{\geqslant 0}$ and a set $J \subseteq [M]\times[M]$, we have*

$$\sum_{(i,j)\in J} \min(u_i, v_i) \leqslant \sqrt{|J|\cdot\|u\|_1\cdot\|v\|_1}.$$

*Proof.* For notational convenience, define

$$J[i] = \{j | (i,j)\in J\}.$$

Now consider the following sequence of relationships:

$$\begin{aligned}
\sum_{(i,j)\in J} \min\{u_i, v_j\} &\leqslant \sum_{(i,j)\in J} \sqrt{u_i v_j} \\
&= \sum_i \sqrt{u_i} \sum_{j\in J[i]} \sqrt{v_j} \\
&\leqslant \sum_i \sqrt{u_i}\cdot\sqrt{|J[i]|}\cdot\sqrt{\|v\|_1} \\
&= \sqrt{\|v\|_1}\cdot\sum_i \sqrt{u_i}\cdot\sqrt{|J[i]|} \\
&\leqslant \sqrt{\|v\|_1}\cdot\sqrt{\|u\|_1}\cdot\sqrt{\sum_i |J[i]|} \\
&= \sqrt{\|v\|_1}\cdot\sqrt{\|u\|_1}\cdot\sqrt{|J|},
\end{aligned}$$

where the inequalities follow from Cauchy-Schwarz inequality. □

## L.3 Wrapping it up

It is easy to see that Theorem L.8, Proposition L.1, Proposition 3.1 and Theorem 3.2 prove Theorem 5.4.