Open access • Proceedings Article • DOI:10.1145/2594538.2594547

Beyond worst-case analysis for joins with minesweeper — Source link

Hung Q. Ngo, Dung Nguyen, Christopher Ré, Atri Rudra

Institutions: University at Buffalo, Stanford University

Published on: 18 Jun 2014 - Symposium on Principles of Database Systems

Topics: Certificate

Beyond Worst-case Analysis for Joins with Minesweeper*

Hung Q. Ngo
SUNY at Buffalo
hungngo@buffalo.edu

Dung T. Nguyen
SUNY at Buffalo
dtn3@buffalo.edu

Christopher Ré
Stanford University
chrismre@stanford.edu

Atri Rudra
SUNY at Buffalo
atri@buffalo.edu

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 β-acyclic queries in time linear in the certificate plus the output size, while for any β-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. We present empirical results that certificates can be much smaller than the input size, which suggests that ideas in minesweeper might lead to faster algorithms in practice.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*

*We thank LogicBlox, Mahmoud Abo Khamis, Semih Salihoglu and Dan Suciu for many helpful conversations. We thank Jérémy Barbay for bringing helpful references on set intersection to our attention. HQN's work is partly supported by NSF grant CCF-1319402 and a gift from Logicblox. DTN's work is partly supported by NSF grant CCF-0844796 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.

PODS'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2594538.2594547.

General Terms

Algorithms, Theory

Keywords

Join Algorithms; Adaptive Algorithm; Instance Optimality; Certificate; Beta-acyclic queries; Bounded Treewidth; 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, 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 β-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.[1] Assuming the 3SUM conjecture, this boundary is tight, in the sense that any β-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 α-acyclic join queries, which are the more traditional notion of acyclicity in database theory and a strictly larger class than β-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 α-acyclic queries cannot be bounded by any polynomial in $|C|$. In the full version of this paper [37], we show that both worst-case optimal algorithms [38, 50] and Yannakakis's algorithm run in time $\omega(|C|)$ for β-acyclic queries on some family of instances.

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 β-acyclic queries. In Section 5, we then describe how to extend the analysis to queries with low-treewidth and the triangle query. We also present some empirical results to indicate that the certificate size may be much smaller than the input size for some graph queries. In Section 6, we discuss related work. Most of the technical details are provided in the appendix.

[1]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 B).

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 $Z = |Q(I)|$.

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 $a \in R[(\mathbf{x},*)]$. We assume throughout that FindGap runs in time $O(k \log |R|)$. This model captures widely used indexes including a B-tree [43, Ch.10] or a Trie [50].

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. We use variables as a perhaps more intuitive, succinct way to describe the underlying morphisms. 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 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 $R = S$. In fact, we need to allow equality constraints between index tuples from the same relation to guarantee that certificates are no larger than the input, which is property (ii) below. 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 $(S[x_1], S[x_1,x_2], \ldots, S[\mathbf{x}])$ 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.

(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 A.

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 A, 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 α-acyclic (often just acyclic) is the standard notion of (hypergraph) acyclicity in database theory [1, p. 128]. A query is β-acyclic, a stronger notion, if every subquery of Q obtained by removing atoms from Q remains α-acyclic.

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 β-acyclic. Then there is some GAO such that Minesweeper computes its output in time $O\big(2^n m^2 n(4^r|C|+Z)\log N\big)$.*

As is standard in database theory, we ignore the dependency on the query size, and the above theorem states that Minesweeper runs in time $\tilde{O}(|C|+Z)$. For β-acyclic queries with a fixed GAO, our results are loose; our best upper bound the complexity uses the treewidth from Section 5.

What about *β-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 [42], we are able to show the following.

Proposition 2.8. *Unless the* 3SUM *problem can be solved in subquadratic time, for any β-*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.

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.

3.1 Notation for Minesweeper

We need some notation to describe our algorithm. Define the *output space* O of the query Q to be the space $O = \mathbf{D}(A_1) \times \mathbf{D}(A_2) \times \cdots \times \mathbf{D}(A_n)$, where $\mathbf{D}(A_i)$ is the domain of attribute A_i. Recall, we assume $\mathbf{D}(A_i) = \mathbb{N}$ for simplicity. By definition, a tuple $\mathbf{t}$ is an *output tuple* if and only if $\mathbf{t} = (t_1,\ldots,t_n) \in O$, 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 (ℓ,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 $\mathbf{t} = (t_1,\ldots,t_n) \in O$ 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.

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.

Algorithm 1 High-level view: Minesweeper algorithm

```
1: CDS ← ∅                                          ▷ No gap discovered yet
2: While CDS can find t not in any stored gap do
3:     If π_{Ā(R)}(t) ∈ R for every R ∈ atoms(Q) then
4:         Report t and tell CDS that t is ruled out
5:     else
6:         Query all R ∈ atoms(Q) for gaps around t
7:         Insert those gaps into CDS
```

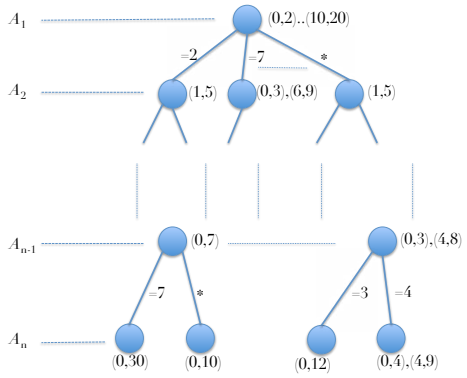

Figure 1: Example of ConstraintTree data structure

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. 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(). In Sections 4 and 5, we analyze getProbePoint() using properties of the query Q.

3.4 The outer algorithm

Algorithm 2 contains all the details that were missing from the high-level view of Algorithm 1. The full version contains a complete run of Minesweeper on a specific query along with complete end-to-end descriptions of two specific queries, which may 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.*

In the next sections, we analyze the CDS, specifically the function getProbePoint(). Our analysis exploits properties of the query and the GAO for β-acyclic and bounded treewidth queries.

4. β-ACYCLIC QUERIES

We describe how to implement getProbePoint for β-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 the Minesweeper calls getProbePoint, Minesweeper might have to do hard work to return a new active tuple, applying memoization along

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** ▷ ℓ, h are just symbols, and $\{\ell,h\}^0$ has only the empty vector
        Let $\mathbf{v} = (v_1, \ldots, v_p)$ ▷ $v_j \in \{\ell, h\}, \forall j \in [p]$
        $(i_R^{(\mathbf{v},\ell)}, i_R^{(\mathbf{v},h)}) \leftarrow R.\textsc{FindGap}\Big(\big(i_R^{(v_1)}, i_R^{(v_1,v_2)}, \ldots, i_R^{(v_1,\ldots,v_p)}\big), t_{s(p+1)}\Big)$ ▷ Gap around $(R[\mathbf{i}_R^{(\mathbf{v})}], t_{s(p+1)})$ in R.
    If $R\Big[i_R^{(h)}, i_R^{(h,h)}, \ldots, i_R^{\{h\}^p}\Big] = t_{s(p)}$ for all $p \in [\text{arity}(R)]$ and for all $R \in \text{atoms}(Q)$ **then**
      Output the tuple $\mathbf{t}$
      $\text{CDS.}\textsc{InsConstraint}\Big(\langle t_1, t_2, \ldots, t_{n-1}, (t_n - 1, t_n + 1)\rangle\Big)$
    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**
              $\text{CDS.}\textsc{InsConstraint}\Big(\Big\langle R\Big[i_R^{(v_1)}\Big], \ldots, R\Big[i_R^{(v_1)}, \cdots, i_R^{(v_1,\ldots,v_p)}\Big], \Big(R[i_R^{(\mathbf{v},\ell)}], R[i_R^{(\mathbf{v},h)}]\Big)\Big\rangle\Big)$
              ▷ Note that the constraint is empty if $R[i_R^{(\mathbf{v},\ell)}] = R[i_R^{(\mathbf{v},h)}]$

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 k 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 $u.\textsc{intervals} \neq \varnothing$. The key property of constraints that we exploit is summarized by the following proposition.

Proposition 4.2. *Using the notation above, for a β-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.

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 β-acyclic. We also show that β-acyclicity and this GAO can be found in polynomial time.

4.3 The getProbePoint Algorithm

Algorithm 3 describes getProbePoint algorithm specialized to β-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 to the smallest value that does not belong to any interval stored in v.intervals, i.e., $t_1 = v$.intervals.Next(-1). 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.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 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

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

Algorithm 3 CDS.getProbePoint() for β-acyclic queries

Input: A ConstraintTree CDS

```
1: i ← 0
2: While i < n do
3:     G ← {u ∈ CDS | (t_1,...,t_i) ≤ P(u) and u.intervals ≠ ∅}
4:     If (G = ∅) then
5:         t_{i+1} ← −1
6:         i ← i + 1
7:     else
8:         Let p̄ = ⟨p̄_1,...,p̄_i⟩ be the bottom of G
9:         Let ū ∈ CDS be the node for which P(ū) = p̄
10:        t_{i+1} ← CDS.nextChainVal(−1, ū, G)
11:        i_0 ← max{k | k ⩽ i, p̄_k ≠ *}
12:        If (t_{i+1} = +∞) and i_0 = 0 then
13:            Return null                          ▷ No tuple t found
14:        else If (t_{i+1} = +∞) then
15:            CDS.InsConstraint(⟨p̄_1,...,p̄_{i_0−1}, (p̄_{i_0} − 1, p̄_{i_0} + 1)⟩)
16:            i ← i_0 − 1                          ▷ Back-track
17:        else
18:            i ← i + 1                            ▷ Advance i
19: Return t = (t_1,...,t_n)
```

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 β-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 β-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 β-*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.*

Comparison with Worst-Case Optimal Algorithms. It is natural to wonder if Yannakakis' worst-case optimal algorithm for α-acyclic queries or the worst-case optimal algorithms of [38] (henceforth, NPRR) or [50] (henceforth LFTJ) can achieve runtimes of $\tilde{O}(|C| + Z)$ for β-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 ℓ (some constant) in a directed graph $G = (V,E)$ (this can be realized by a "path" query of length ℓ 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

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
```

length at most ℓ, then both NPRR and LFTJ will have to explore all $\omega(|E|)$ paths leading to an $\omega(|C|)$ runtime.

In the full version of this paper, we exhibit a family of β-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 β-cyclic queries, we are able to show the following analog of the treewidth-based runtime under the traditional worst-case complexity notion [6, 18].

Theorem 5.1 (Minesweeper for bounded treewidth queries)**.** *Suppose that the GAO has an* 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 $\tilde{O}\big(|C|^{w+1}+Z\big)$*. Furthermore, if the treewidth of the input query Q is 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 β-acyclic case, the only change is in GETPROBEPOINT. For general queries, the GETPROBEPOINT algorithm remains similar in structure to that of the β-acyclic case (Algorithm 3), and if the input query is β-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 β-acyclic case.

Query	com-Orkut		soc-Epinions1		soc-LiveJournal1	
	N	$\lvert C\rvert$	N	$\lvert C\rvert$	N	$\lvert C\rvert$
Star	352M	214K	1.5M	1.1K	207M	172K
3-path	352M	119K	1.5M	0.8K	207M	138K
Tree	469M	2.8M	2M	3.4K	276M	2.7M

Table 1: Input size (N) versus Certificate size ($|C|$). Units are Million(M) and Thousand(K). The three graph datasets are from Orkut, Epinions, and LiveJournal network `http://snap.stanford.edu/data/`.

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 α-acyclic queries.

Proposition 5.2. *Unless the* exponential time hypothesis *is false, for every large enough constant $k > 0$, there is an α-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 (α-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 An implementation of Minesweeper

With the help of LogicBlox, we implemented Minesweeper inside the LogicBlox engine. Our results are preliminary: it is implemented for main memory data and all experiments are run in a multi-threaded mode. We run three queries: a star query, a small path query, and a tree query, which are described below, on three data sets Orkut online social network, Who-trusts-whom network of Epinions.com, and LiveJournal online social network.

- Star query: $Q = R_1(A) \bowtie S(A,B) \bowtie S(A,C) \bowtie S(A,D) \bowtie R_2(B) \bowtie R_3(C) \bowtie R_4(D)$.
- 3-path query: $Q = S(A,B) \bowtie S(B,C) \bowtie S(C,D) \bowtie R_5(A) \bowtie R_6(B) \bowtie R_7(C) \bowtie R_8(D)$.
- Tree query: $Q = S(A,B) \bowtie S(B,C) \bowtie S(B,D) \bowtie S(D,E) \bowtie R_9(A) \bowtie R_{10}(C) \bowtie R_{11}(D) \bowtie R_{12}(E)$.

For each query and each dataset, relation S is a graph dataset, while every R_i relation contains a subset of vertices from that graph dataset, where every vertex is chosen with a probability 0.001. Table 1 shows the input size versus certificate size on different queries and different graph datasets. The upper bound of the certificate size is measured by counting the number of FINDGAP operations during computing join queries. These numbers show that certificate size is very small compared to input size and so it indicates that a practical implementation might be obtained.

5.3 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. (A straightforward application of our more general analysis given in Theorem 5.1, which gives $\tilde{O}(|C|^3+Z)$.) 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 $\Omega(|\mathcal{C}|^2)$ of them). By designing a smarter CDS, our improved CDS explores $O(|\mathcal{C}|)$ such pairs. We can prove the following result.

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

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 [47] (when the query is assumed to be of fixed size). They are W[1]-complete and thus unlikely to be fix-parameter tractable [31,41].

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 [17,26,29,51,52]. 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 [40], and in the RAM model even when there are inequalities [51]. For queries with treewidth w, it was recognized early on that a runtime of about $\tilde{O}(N^{w+1}+Z)$ is attainable [18,27]. our result strictly generalizes these results. We able to 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, 26, 29, 30] refined the treewidth notion leading to generalized hyper treewidth [29] and ultimately *fractional hypertree width* [36], 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 [38]. 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 [52] 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 [50] 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. We are able to show that neither Leapfrog nor the algorithm from [38] can achieve the certificate guarantees of Minesweeper for β-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, [23]), and at least one new one introduced recently [21]. 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 α-acyclicity [11, 12, 25, 28, 35] 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 [52]. However, an α-acyclic hypergraph may have a sub-hypergraph that is not α-acyclic. For example, if we take *any* hypergraph and add a hyperedge containing all vertices, we obtain an α-acyclic hypergraph. This observation leads to the notion of β-acyclicity: a hypergraph is β-acyclic if and only if every one of its sub-hypergraph is (α-) acyclic [23]. It was shown (relatively) recently [39] that SAT is in P for β-acyclic CNF formulas and is NP-complete for α-acyclic CNF formulas. Extending the result, it was shown that negative conjunctive queries are poly-time solvable if and only if it is β-acyclic [14]. The separation between γ-acyclicity and β-acyclicity showed up in logic [20], 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 lower-bounding the number of triangles and the runtime for such algorithms [5, 33, 34, 46, 48, 49]. 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 [32] since 1972. In fact, the idea of skipping elements using a binary-search jumping (or leap-frogging) strategy was already present in [32]. Demaine et al. [19] 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.

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 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].

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 [44].) 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 [22], 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 [22] 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. [24]. 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 β-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 β-acyclic queries is the boundary of complexity in that we show no algorithm for *β-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 α-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 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?

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

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

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

[21] ———, *Some characterizations of γ and β-acyclicity of hypergraphs.*, Information Processing Letters, 112 (2012).

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

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

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

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

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

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

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

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

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

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

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

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

[34] 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.

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

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

[37] H. Q. Ngo, D. T. Nguyen, C. Ré, and A. Rudra, *Beyond worst-case analysis for joins with minesweeper*, CoRR, abs/1302.0914 (2014).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. CERTIFICATES

The full version of this paper contains a handful of examples to illustrate the subtlties of certificates. For completeness, we include the following proofs.

A.1 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.) □

A.2 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 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 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. □

B. RUNNING TIME ANALYSIS

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

Definition B.1. We say a join algorithm $\mathcal{A}$ for a join query Q to be instance optimal for Q with optimality ratio α 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 α 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 for Q.

Technically we should be call such algorithms *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 $O(N\log N)$.

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 [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 [45, 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').