Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries

Nofar Carmeli
snofca@cs.technion.ac.il
Technion, Israel

Nikolaos Tziavelis
tziavelis.n@northeastern.edu
Northeastern University, USA

Wolfgang Gatterbauer
w.gatterbauer@northeastern.edu
Northeastern University, USA

Benny Kimelfeld
bennyk@cs.technion.ac.il
Technion, Israel

Mirek Riedewald
m.riedewald@northeastern.edu
Northeastern University, USA

ABSTRACT

We study the question of when we can provide logarithmic-time direct access to the k-th answer to a Conjunctive Query (CQ) with a specified ordering over the answers, following a preprocessing step that constructs a data structure in time quasilinear in the size of the database. Specifically, we embark on the challenge of identifying the tractable answer orderings that allow for ranked direct access with such complexity guarantees.

We begin with *lexicographic orderings* and give a decidable characterization (under conventional complexity assumptions) of the class of tractable lexicographic orderings for every CQ without self-joins. We then continue to the more general *orderings by the sum of attribute weights* and show for it that ranked direct access is tractable only in trivial cases. Hence, to better understand the computational challenge at hand, we consider the more modest task of providing access to only a single answer (i.e., finding the answer at a given position) — a task that we refer to as *the selection problem*. We indeed achieve a quasilinear-time algorithm for a subset of the class of full CQs without self-joins, by adopting a solution of Frederickson and Johnson to the classic problem of selection over sorted matrices. We further prove that none of the other queries in this class admit such an algorithm.

CCS CONCEPTS

• **Theory of computation** → **Database theory**; *Complexity classes*; *Database query languages (principles)*; *Database query processing and optimization (theory)*.

KEYWORDS

conjunctive queries, direct access, ranking function, answer orderings, query classification

ACM Reference Format:

Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2021. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3452021.3458331

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

PODS '21, June 20–25, 2021, Virtual Event, China

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8381-3/21/06…$15.00
https://doi.org/10.1145/3452021.3458331

1 INTRODUCTION

When can we allow for direct access to a ranked list of answers to a database query without (and considerably faster than) materializing all answers? To illustrate the concrete instantiation of this question, assume the following simple relational schema for information about pandemic spread and relevant activity of residents:

$$\text{Visits}(\text{person}, \text{age}, \text{city}) \qquad \text{Cases}(\text{city}, \text{date}, \#\text{cases})$$

Here, Visits mentions, for each person, the cities that the person visits regularly (e.g., for work and relatives) and the age of the person (for risk assessment); the relation Cases specifies the number of new infection cases in specific cities at specific dates (a measure that is commonly used for spread assessment albeit being sensitive to the amount of testing).

Suppose that we wish to efficiently compute the natural join Visits ⋈ Cases based on equality of the city attribute, so that we have all combinations of people (with their age), the cities they regularly visit, and the city's daily new cases. For example,

$$(\texttt{Anna}, 72, \texttt{Boston}, 12/7/2020, 179)\,.$$

While the number of such answers could be quadratic in the size of the database, the seminal work of Bagan, Durand, and Grandjean [3] has established that it can be evaluated using an enumeration algorithm with a constant delay between consecutive answers, after a linear-time preprocessing phase. This is due to the fact that this join is a special case of a *free-connex* Conjunctive Query (CQ). In the case of CQs without self-joins, being free-connex is a sufficient and necessary condition for such efficient evaluation [3, 8]. The necessity requires conventional assumptions in fine-grained complexity[1] and it holds even if we multiply the preprocessing time and delay by a logarithmic factor in the size of the database.[2]

To realize the constant (or logarithmic) delay, the preprocessing phase builds a data structure that allows for efficient iteration over the answers in the enumeration phase. Brault-Baron [8] showed that in the linear-time preprocessing phase, we can build a structure with better guarantees: not only log-delay enumeration, but even log-time *direct access*: a structure that, given k, allows to directly retrieve

[1] For the sake of simplicity, throughout this section we make all of these complexity assumptions. We give their formal statements in Section 2.4.

[2] We refer to those as *quasilinear preprocessing* and *log delay*, respectively.

the k^{th} answer in the enumeration without needing to enumerate the preceding $k-1$ answers first.[3] Later, Carmeli et al. [10] showed how such a structure can be used for enumerating answers in a random order (random permutation)[4] with the statistical guarantee that the order is uniformly distributed. In particular, in the above example we can enumerate the answers of Visits $\bowtie$ Cases in a provably uniform random permutation (hence, ensuring statistical validity of each prefix) with logarithmic delay, after a linear-time preprocessing phase. Their direct-access structure also allows for *inverted access*: given an answer, return the index k of that answer (or determine that it is not a valid answer). Recently, Keppeler [23] proposed another direct-access structure with the additional ability to allow efficient database updates, but at the cost of only supporting a limited subset of free-connex CQs.

All known direct-access structures [8, 10, 23] allow the answers to be *sorted* by some lexicographic order (even if they do not explicitly state it). For instance, in our Visits $\bowtie$ Cases the structure could be such that the tuples are in the (descending or ascending) order of #cases and then by date, or in the order of city and then by age. Hence, in logarithmic time we can evaluate quantile queries (find the k^{th} answer in order) and determine the position of a tuple inside the sorted list. From this we can also conclude (fairly easily) that we can enumerate the answers ordered by age where ties are broken randomly, again provably uniformly. Carmeli et al. [10] have also shown how the order of the answers can be useful for generalizing direct-access algorithms from CQs to UCQs. Note that direct access to the sorted list of answers is a stronger requirement than *ranked enumeration* that has been studied in recent work [7, 11, 31, 32, 34].

Yet, the choice of which lexicographic order is taken is an artefact of the structure construction (e.g., the elimination order [8], the join tree [10], or the q-tree [8]). If the application desires a specific lexicographic order, we can only hope to find a matching construction. However, this is not necessarily possible. For example, could we construct in (quasi)linear time a direct-access structure for Visits $\bowtie$ Cases ordered by #cases and then by age? Interestingly, we will show that the answer is negative: it is impossible to build in quasilinear time a direct-access structure with logarithmic access time for that lexicographic order.

Getting back to the question posed at the beginning of this section, in this paper we embark on the challenge of identifying, for each CQ, the orders that allow for efficiently constructing a direct-access structure. We adopt the tractability yardstick of quasilinear construction (preprocessing) time and logarithmic access time. In addition, we focus on two types of orders: lexicographic orders, and scoring by the sum of attribute weights.

Contributions. Our first main result is an algorithm for direct access for lexicographic orders, including ones that are not achievable by past structures. We further show that within the class of CQs without self-joins, our algorithm covers all the tractable cases (in the sense adopted here), and we establish a decidable and easy to test classification of the lexicographic orders over the free variables into tractable and intractable ones. For instance, in the case of Visits $\bowtie$ Cases the lexicographic order (#cases, age, city, date, person) is intractable. It is classified as such because #cases and age are non-neighbours (i.e., do not co-occur in the same atom), but city, which comes after #cases and age in the order, is a neighbour of both. This is what we call a *disruptive trio*. The lexicographic order (#cases, age) is also intractable since the query Visits $\bowtie$ Cases is not {#cases, age}-connex. In contrast, the lexicographic order (#cases, city, age) is tractable. We also show that within the tractable side, the structure we construct allows for inverted access in constant time.

Our classification is proved in two steps. We begin by considering the complete lexicographic orders (that involve all free variables). We show that for free-connex CQs without self-joins, the absence of a disruptive trio is a sufficient and necessary condition for tractability. We then generalize to partial lexicographic orders L where the ordering is determined only by a *subset of the free variables*. There, the condition is that there is no disruptive trio *and* that the query is L-connex (a similar condition to being free-connex, but for the subset of the variables that appear in L instead of the free ones). Interestingly, it turns out that a partial lexicographic order is tractable if and only if it is the prefix of a complete tractable lexicographic order.

A lexicographic order is a special case of an ordering by the *sum of attribute weights*, where every database value is mapped to some number. Hence, a natural question is which CQs have a tractable direct access by the order of sum. For example, what about Visits $\bowtie$ Cases with the order ($\alpha\cdot$#cases + $\beta\cdot$age)? It is easy to see that this order is intractable because the lexicographic order (#cases, age) is intractable. In fact, it is easy to show that an order by sum is intractable whenever *any* lexicographic order is intractable (e.g., there is a disruptive trio). However, the situation is worse: the only tractable case is the one where the CQ is acyclic and there is an atom that contains all of the free variables. In particular, ordering by sum is intractable already for the Cartesian product $Q(c_1, d, x, p, a, c_2) \text{ :- } \text{Visits}(p, a, c_1), \text{Cases}(c_2, d, x)$, even though *every* lexicographic order is tractable (according to our aforementioned classification). This daunting hardness also emphasizes how ranked direct access is fundamentally harder than *ranked enumeration* where, in the case of the sum of attributes, the answers of every full acyclic CQ can be enumerated with logarithmic delay after a linear preprocessing time [31].

To understand the root cause of the hardness of sum, we narrow our question to a considerably weaker guarantee. Our notion of tractability so far requires the construction of a structure in quasilinear time and a direct access in logarithmic time. In particular, if our goal is to compute just a single quantile, say the k^{th} answer, then it takes quasilinear time. Computing a single quantile is known as the *selection problem* [6]. The question we ask is to what extent is selection a weaker requirement than direct access in the case of CQs. That is, how much larger is the class of CQs with quasilinear selection than that of CQs with a quasilinear construction of a logarithmic-access structure?

We answer the above question for the class of full CQs without self-joins by establishing the following dichotomy for the order by sum (again assuming fine-grained hypotheses): the selection problem can be solved in $O(n \log n)$ time, where n is the size of the database, if and only if the hypergraph of the CQ contains at most two maximal hyperedges (w.r.t. containment). The tractable side is

[3]"Direct access" is also widely known as "random access." We prefer to use "direct access" to avoid confusion with the problem of answering "in random order."
[4]Not to be confused with "random access."

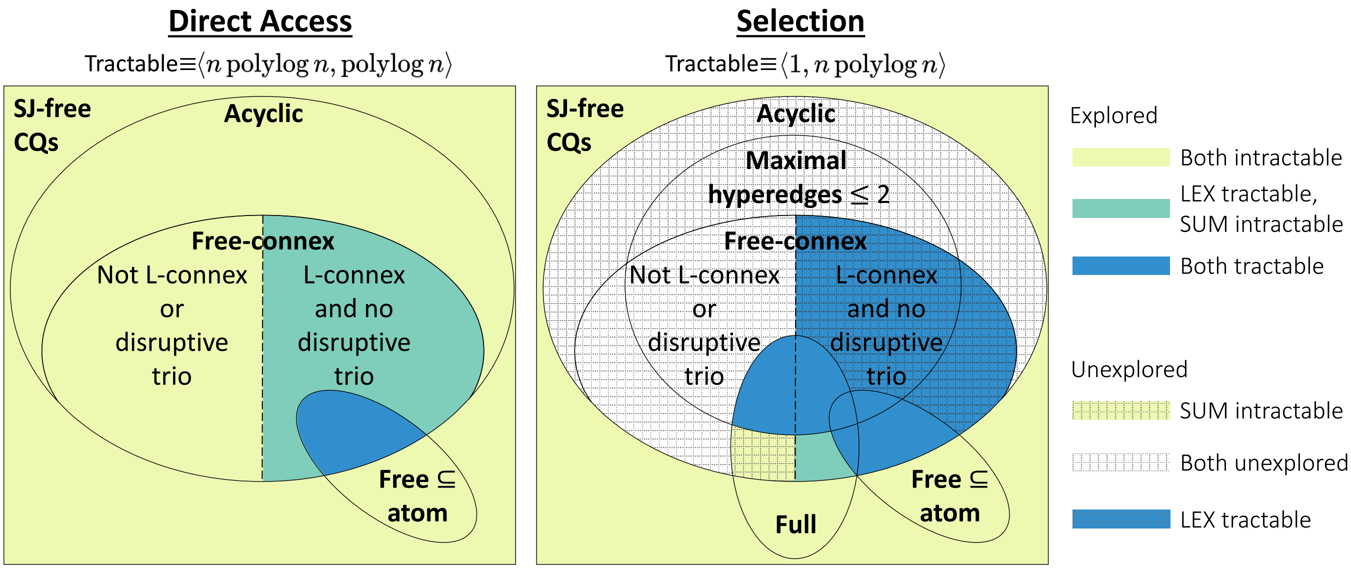


Figure 1: Overview of our results for lexicographic (LEX) orders and sum-of-weights (SUM) orders. CQs without self-joins (SJ-free) are classified based on the tractability of the *direct access problem* (left) and the *selection problem* (right). Some regions of the right figure are not explored in this paper. The L-connex property applies only to lexicographic orders L (the precise definitions are given in Section 2). All tractable cases extend to CQs with self-joins. The sizes of the ellipses are arbitrary and do not correspond to the size or importance of the classes.

applicable even in the presence of self-joins, and it is achieved by adopting an algorithm by Frederickson and Johnson [15] originally developed for selection on sorted matrices. For illustration, the selection problem is solvable in quasilinear time for the query Visits $\bowtie$ Cases ordered by sum.

Overview of results. We summarize our results in Figure 1 with different colors indicating the tractability of the studied orderings. For direct access, we obtain the complete picture of the orders and CQs without self-joins that admit $O(n \operatorname{polylog} n)$ preprocessing and $O(\operatorname{polylog} n)$ per access (conveniently denoted as $\langle n \operatorname{polylog} n, \operatorname{polylog} n\rangle$ for ⟨preprocessing, access⟩). For selection, the present paper explores only some of the possible orders and CQs that admit an $O(n \operatorname{polylog} n)$ solution (denoted as $\langle 1, n \operatorname{polylog} n\rangle$). We depict the unexplored regions (when our results cover only one or none of the problems) with a grid background pattern. Since sum orderings are harder than any lexicographic order, there are only three "Unexplored" cases: (1) SUM is known to be intractable but LEX is yet unexplored, (2) LEX is known to be tractable but SUM is yet unexplored, and (3) neither problem has been explored. Beyond the cases directly covered in the formal statements of our paper, we additionally infer (in)tractability for some other cases based on the fact that selection is an easier problem than direct access.

Applicability. It is important to note that while our results are stated over a limited class of queries (a fragment of acyclic CQs), there are some implications beyond this class that are immediate yet significant. In particular, we can use known techniques that reduce other CQs to a tractable form and then apply our direct-access solutions. An example is the common case where the relations are associated with *functional dependencies*; in this case, some queries become easier since we can make assumptions on the internal structure of the input relations.[5] More specifically, *FD-extensions* can be used to transform CQs with an otherwise intractable structure into queries with a tractable structure [9]. As another example, a *hypertree decomposition* can be used to transform a cyclic CQ to an acyclic form by paying a non-linear overhead during preprocessing [20].

Outline. The remainder of the paper is organized as follows. Section 2 gives the necessary background. In Section 3 we consider direct access by lexicographic orders that include all the free variables, and Section 4 extends the results to partial ones. We move on to the (for the most part) negative results for direct access by sum orderings in Section 5 and then study the selection problem in Section 6. Section 7 concludes and gives some directions for future work. Due to space constraints, some proofs are in the Appendix.

2 PRELIMINARIES

2.1 Basic Notions

Database. A *schema* $\mathcal{S}$ is a set of relational symbols $\{R_1, \dots, R_m\}$. We use $\mathtt{ar}(R)$ for the arity of a relational symbol R. A *database instance* I contains a finite relation $R^I \subseteq \mathtt{dom}^{\mathtt{ar}(R)}$ for each $R \in \mathcal{S}$, where $\mathtt{dom}$ is a set of constant values called the *domain*. We use n for the size of the database, i.e., the total number of tuples.

Queries. A *conjunctive query* (CQ) Q over schema $\mathcal{S}$ is an expression of the form $Q(\vec{X}_f) \coloneq R_1(\vec{X}_1), \dots, R_\ell(\vec{X}_\ell)$, where the tuples $\vec{X}_f, \vec{X}_1, \dots, \vec{X}_\ell$ hold variables, every variable in $\vec{X}_f$ appears in some

[5] Our Visits $\bowtie$ Cases example may also involve functional dependencies, such as person $\longrightarrow$ age, which could invalidate the lower bounds. Yet, all hardness statements mentioned about this example in this section can be shown to follow from the results of this paper.

$\vec{X}_1, \dots, \vec{X}_\ell$, and $R_1, \dots, R_\ell \subseteq \mathcal{S}$. Each $R_i(\vec{X}_i)$ is called an *atom* of the query Q, and $\mathtt{atoms}(Q)$ denotes the set of all atoms. We use $\mathtt{var}(e)$ or $\mathtt{var}(Q)$ for the set of variables that appear in an atom e or query Q, respectively. The variables $\vec{X}_f$ are called *free* and are denoted by $\mathtt{free}(Q)$. A CQ is *full* if $\mathtt{free}(Q) = \mathtt{var}(Q)$ and *Boolean* if $\mathtt{free}(Q) = \emptyset$. Sometimes, we say that CQs that are not full have *projections*. A repeated occurrence of a relational symbol is a *self-join* and if no self-joins exist, a CQ is called *self-join-free*. A homomorphism μ from a CQ Q to a database I is a mapping of $\mathtt{var}(Q)$ to constants from $\mathtt{dom}$, such that every atom of Q maps to a tuple in the database I. A *query answer* q is such a homomorphism followed by a projection of μ on the free variables, denoted by $\pi_{\mathtt{free}(Q)}(\mu)$. The answer to a Boolean CQ is whether such a homomorphism exists. The set of query answers is $Q(I)$.

Hypergraphs. A *hypergraph* $\mathcal{H} = (V, E)$ is a set V of *vertices* and a set E of subsets of V called *hyperedges*. Two vertices in a hypergraph are *neighbors* if they appear in the same edge. A *path* of $\mathcal{H}$ is a sequence of vertices such that every two succeeding vertices are neighbors. A *chordless path* is a path in which no two non-succeeding vertices appear in the same hyperedge (in particular, no vertex appears twice). A *join tree* of a hypergraph $\mathcal{H} = (V, E)$ is a tree T where the nodes[6] are the hyperedges of $\mathcal{H}$ and the *running intersection* property holds, namely: for all $u \in V$ the set $\{e \in E \mid u \in e\}$ forms a (connected) subtree in T. An equivalent phrasing of the running intersection property is that given two vertices e_1, e_2 of the tree, for any vertex e_3 on the simple path between them, we have that $e_1 \cap e_2 \subseteq e_3$. A hypergraph $\mathcal{H}$ is *acyclic* if there exists a join tree for $\mathcal{H}$. We associate a hypergraph $\mathcal{H}(Q) = (V, E)$ to a CQ Q where the vertices are the variables of Q, and every atom of Q corresponds to a hyperedge with the same set of variables. Stated differently, $V = \mathtt{var}(Q)$ and $E = \{\mathtt{var}(e) | e \in \mathtt{atoms}(Q)\}$. With a slight abuse of notation, we identify atoms of Q with hyperedges of $\mathcal{H}(Q)$. A CQ Q is *acyclic* if $\mathcal{H}(Q)$ is acyclic, otherwise it is *cyclic*.

Free-connex CQs. A hypergraph $\mathcal{H}'$ is an *inclusive extension* of $\mathcal{H}$ if every edge of $\mathcal{H}$ appears in $\mathcal{H}'$, and every edge of $\mathcal{H}'$ is a subset of some edge in $\mathcal{H}$. Given a subset S of the vertices of $\mathcal{H}$, a tree T is an *ext-S-connex tree* (i.e., extension-S-connex tree) for a hypergraph $\mathcal{H}$ if: (1) T is a join tree of an inclusive extension of $\mathcal{H}$, and (2) there is a subtree[7] T' of T that contains exactly the vertices S [3]. We say that a hypergraph is S-connex if it has an ext-S-connex tree [3]. A hypergraph is S-connex iff it is acyclic and it remains acyclic after the addition of a hyperedge containing exactly S [8]. Given a hypergraph $\mathcal{H}$ and a subset S of its vertices, an *S-path* is a chordless path $(x, z_1, \dots, z_k, y)$ in $\mathcal{H}$ with $k \geq 1$, such that $x, y \in L$, and $z_1, \dots, z_k \notin L$. A hypergraph is S-connex iff it has no S-path [3]. A CQ Q is *free-connex* if $\mathcal{H}(Q)$ is $\mathtt{free}(Q)$-connex [3]. Note that a free-connex CQ is necessarily acyclic.[8] An implication of the characterization given above is that it suffices to find a join-tree for an inclusive extension of a hypergraph $\mathcal{H}$ to infer that $\mathcal{H}$ is acyclic. To simplify notation, we also say that a CQ is L-connex for a (partial) lexicographic order L if the CQ is S-connex for the variables S that appear in L. Generalizing the notion of an inclusive extension, we say that a hypergraph $\mathcal{H}'$ is *inclusion equivalent* to $\mathcal{H}$ if every hyperedge of $\mathcal{H}$ is a subset of some hyperedge of $\mathcal{H}'$ and vice versa.

[6] To make a clear distinction between the vertices of a hypergraph and those of its join tree, we call the latter nodes.

[7] By subtree, we mean any connected subgraph of the tree.

[8] Free-connex CQs are sometimes called in the literature *free-connex acyclic* CQs [3]. As free-connexity is not defined for cyclic CQs, we choose to omit the word *acyclic* and simply call these CQs *free-connex*.

2.2 Problem Definitions

Orders of Answers. For a CQ Q and database instance I, a *ranking function* $\mathtt{rank} : Q(I) \times Q(I) \to Q(I)$ compares two query answers and returns the smaller one according to some underlying total order.[9] We consider two types of orders in this paper. Assuming that the domain values are ordered, a *lexicographic order* L is an ordering of $\mathtt{free}(Q)$ such that $\mathtt{rank}(q_1, q_2)$ first compares q_1, q_2 on the value of the first L variable, and if they are equal on the value of the second L variable, and so on. A lexicographic order is called *partial* if the variables in L are a subset of $\mathtt{free}(Q)$.

The second type of order assumes a given weight function that assigns a real-valued weight to the domain values of each variable. More precisely, for a variable x, we define $w_x : \mathtt{dom} \to \mathbb{R}$ and then the weight of a query answer is computed by aggregating the weights of the assigned values of free variables. In a *sum-of-weights order*, denoted by Σw, we have $w_Q(q) = \sum_{x \in \mathtt{free}(Q)} w_x(q(x)), q \in Q(I)$ and $\mathtt{rank}(q_1, q_2)$ compares $w_Q(q_1)$ with $w_Q(q_2)$. To simplify notation, we refer to all w_x and w_Q together as one weight function w. If two query answers have the same weight, then we break ties arbitrarily but consistently, e.g., according to a lexicographic order on their assigned values.

Attribute Weights vs. Tuple Weights. Notice that in the definition above, we assume that the input weights are assigned to the domain values of the attributes. Alternatively, the input weights could be assigned to the relation tuples, a convention that has been used in past work on ranked enumeration [31]. Since there are several reasonable semantics for interpreting a tuple-weight ranking for CQs with projections and/or self-joins [30], we elect to present our results for the case of attribute weights. For self-join-free CQs, attribute weights can easily be transformed to tuple weights in linear time such that the weights of the query answers remain the same. This works by assigning each variable to one of the atoms that it appears in, and computing the weight of a tuple by aggregating the weights of the assigned attribute values. Therefore, our hardness results for sum-of-weights orders directly extend to the case of tuple weights. Moreover, note that our positive results on direct access (Section 5) and selection (Section 6.2) rely on algorithms that innately operate on tuple weights, thus we cover that case too.

Direct Access vs. Selection. In the problem of *direct access* by an underlying order, we are given as an input a query Q, and a database I, and the goal is to construct a data structure which then allows us to support accesses on the sorted array of query answers. Specifically, an access asks for the query answer at index k on the (implicit) array containing $Q(I)$ sorted via $\mathtt{rank}$ comparisons, for a given integer k. This data structure is built in a preprocessing phase, after which we have to be able to support multiple such accesses. Our goal is to achieve efficient access (in polylogarithmic time) with a preprocessing phase that is significantly smaller than $Q(I)$ (quasilinear in the database size).

[9] WLOG, we assume that the order is ascending but all results hold if we $\mathtt{rank}$ returns the bigger (max) instead of the smaller (min).

The problem of *selection* [6, 13, 14] is a computationally easier task that requires only a single direct access, hence does not make a distinction between preprocessing and access phases. A special case of the problem is to find the median query result.

2.3 Complexity Framework and Sorting

We measure asymptotic complexity in terms of the size of the database n, while the size of the query is considered constant. If the time for preprocessing is $O(f(n))$ and the time for each access is $O(g(n))$, we denote that as $\langle f(n), g(n)\rangle$, where f, g are functions from $\mathbb{N}$ to $\mathbb{R}$. Note that by definition, the problem of selection asks for a $\langle 1, g(n)\rangle$ solution.

The model of computation is the RAM model with uniform cost measure. In particular, it allows for linear time construction of lookup tables, which can be accessed in constant time. We would like to point out that some past works [3, 10] have assumed that in certain variants of the model, sorting can be done in linear time [21]. Since we consider problems related to summation and sorting [15] where a linear-time sort would improve otherwise optimal bounds, we adopt a more standard assumption that sorting is comparison-based and possible only in quasilinear time. As a consequence, some upper bounds mentioned in this paper are weaker than the original sources which assumed linear-time sorting [8, 10].

2.4 Hardness Hypotheses

Denote by sparseBMM the hypothesis that two Boolean matrices A and B, represented as lists of their non-zero entries, cannot be multiplied in time $m^{1+o(1)}$, where m is the number of non-zero entries in A, B, and AB. A consequence of this hypothesis is that we cannot answer the query $Q(x, z) :- R(x, y), S(y, z)$ with quasilinear preprocessing and polylogarithmic delay. In more general terms, any self-join-free acyclic non-free-connex CQ cannot be enumerated with quasilinear[10] preprocessing time and polylogarithmic delay assuming the sparseBMM hypothesis [3, 5].

A $(k+1, k)$-*hyperclique* is a set of $k+1$ vertices in a hypergraph such that every k-element subset is a hyperedge. Denote by Hyperclique the hypothesis that for every $k \geq 2$ there is no $O(m \operatorname{polylog} m)$ algorithm for deciding the existence of a $(k+1, k)$-hyperclique in a k-uniform hypergraph with m hyperedges. When $k = 2$, this follows from the δ-Triangle hypothesis [1] for any $\delta > 0$. When $k \geq 3$, this is a special case of the (ℓ, k)– Hyperclique Hypothesis [25]. A known consequence is that Boolean cyclic and self-join-free CQs cannot be answered in quasilinear[10] time [8]. Moreover, cyclic and self-join-free CQs do not admit enumeration with quasilinear preprocessing time and polylogarithmic delay assuming the Hyperclique hypothesis [8].

In its simplest form, the 3SUM problem asks for three distinct real numbers a, b, c from a set S with n elements that satisfy $a + b + c = 0$. There is a simple $O(n^2)$ algorithm for the problem, but it is conjectured that in general, no truly subquadratic solution exists [29]. The significance of this conjecture has been highlighted by many conditional lower bounds for problems in computational geometry [17] and within the P class in general [33]. Note that the problem remains hard even for integers provided that they are sufficiently large (i.e., in the order of $O(n^3)$) [29]. We denote by 3sum the following equivalent hypothesis [4] that uses three different sets of numbers: Deciding whether there exist $a \in A, b \in B, c \in C$ from three sets of integers A, B, C such that $a + b + c = 0$ cannot be done in time $O(n^{2-\epsilon})$ for any $\epsilon > 0$. This lower bound has been confirmed in some restricted models of computation [2, 12].

[10] Works in the literature typically phrase this as linear, yet any logarithmic factor increase is still covered by the hypotheses.

2.5 Known Results for CQs

Eliminating Projection. We now provide some background that relates to the efficient handling of CQs. For a query with projections, a standard strategy is to reduce it to an equivalent one where techniques for acyclic full CQs can be leveraged. The following proposition, that is widely known and used [5], shows that this is possible for free-connex CQs.

Proposition 2.1 (Folklore). *Given a database instance I, a CQ Q, a join tree T of an inclusive extension of Q, and a subtree T' of T that contains all the free variables, we can compute in linear time a database instance I' over the schema of a CQ Q' that consists of the nodes of T' such that $Q(I) = Q'(I')$ and $|I'| \leq |I|$.*

This reduction is done by first creating a relation for every node in T using projections of existing relations, then performing the classic semi-join reduction by Yannakakis [35] to filter the relations of T' according to the relations of T, and then we can simply ignore all relations that do not appear in T' and obtain the same answers. Afterwards, they can be handled efficiently, e.g. their answers can be enumerated with constant delay [3].

Ranked enumeration. Enumerating the answers to a CQ in ranked order is a special case of direct access where the accessed indexes are consecutive integers starting from 0. In contrast to direct access, ranked enumeration by sum orderings (thus also lexicographic orderings) is possible with logarithmic delay after a linear-time preprocessing phase for all full acyclic CQs [31]. This result has also been extended to free-connex CQs [30]. Existing ranked-enumeration algorithms rely on priority queue structures that compare a minimal number of candidate answers to produce the ranked answers one-by-one in order. There is no straightforward way to extend them to the task of direct access where we may skip over a large number of answers to get to an arbitrary index k.

Direct Access. Past work on direct access identified the tractable queries without guarantees on the order of the query answers.

Theorem 2.2 ([8, 10]). *Let Q be a CQ. If Q is free-connex, then direct access (in some order) is possible in $\langle n \log n, \log n\rangle$. Otherwise, if it is also self-join-free, then direct access (in any order) is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n\rangle$, assuming sparseBMM and Hyperclique.*

Even though these algorithms do not explicitly discuss the order of the answers, a closer look shows that they internally use and produce some lexicographic order.

Recent work by Keppeler [23] suggests another direct-access solution by lexicographic order, which also supports efficient insertion and deletion of input tuples. Given these additional requirements, the supported CQs are more limited, and are only a subset of free-connex CQs called *q-hierarchical*. This is a subclass of the well-known *hierarchical* queries with an additional restriction on the existential variables. As an example, the following CQs are not q-hierarchical even though they are free-connex:

$Q_1(x,y) :- R_1(x), R_2(x,y), R_3(y)$ and $Q_2(x) :- R_1(x,y), R_2(y)$. For these queries, direct access is not supported by the solution of Keppeler [23], even though it is possible without the update requirements.

All previous direct-access solutions of which we are aware have two gaps compared to this work: (1) they do not discuss which lexicographic orders (given by orderings of the free variables) are supported; (2) they do not support all possible lexicographic orders. We conclude this section with a short survey of existing solutions and their supported orders.

All prior direct-access solutions use some underlying component that depends on the query structure and constrains the supported orders. The algorithm of Carmeli et al. [10, Algorithm 3] assumes that a join tree is given with the CQ, and the lexicographic order is *imposed by the join tree*. Specifically, it is an ordering of the variables achieved by a preorder depth-first traversal of the tree. As a result, it does not support any order that requires jumping back-and-forth between different branches of the tree. In particular, it does not support $Q_3(v_1, v_2, v_3, v_4) :- R(v_1, v_3), S(v_2, v_4)$ with the lexicographic order given by the increasing variable indices (we adopt this convention for all the examples below). We show how to handle this CQ and order in detail in Example 3.5. The algorithm of Brault-Baron [8, Algorithm 4.3] assumes that an *elimination order* is given along with the CQ. The resulting lexicographic order is affected by that elimination order, but is not exactly the same. This solution suffers from similar restrictions, and it does not support Q_3 either. The algorithm of Keppeler [23] assumes that a *q-tree* is given with the CQ, and the possible lexicographic orders are affected by this tree. Unlike the previous ones, this algorithm can interleave variables from different atoms, yet cannot support some orders that are possible for the previous algorithms. As an example, it does not support $Q_4(v_1, v_2, v_3) :- R_1(v_1, v_2), R_2(v_2, v_3)$ as v_2 is highest in the hierarchy (the atoms containing it strictly subsume the atoms containing any other variable) and so it is necessarily the first variable in the q-tree and in the ordering produced.

Finally, we should mention that there are query-ordering pairs that require *both* jumping back-and-forth in the join tree *and* visiting the variables in an order different than any hierarchy. As a result, these are not supported by any previous solution. Two such examples are $Q_5(v_1, v_2, v_3, v_4, v_5) :- R_1(v_1, v_3), R_2(v_3, v_4), R_3(v_2, v_5)$ and $Q_6(v_1, v_2, v_3, v_4, v_5) :- R_1(v_1, v_2, v_4), R_2(v_2, v_3, v_5)$. In Section 3, we provide an algorithm that supports both of these CQs.

3 DIRECT ACCESS BY LEXICOGRAPHIC ORDERS

In this section, we answer the following question: for which underlying lexicographic orders can we achieve "tractable" direct access to ranked CQ answers, i.e. with quasilinear preprocessing and polylogarithmic time per answer?

Example 3.1 (No direct access). Consider the lexicographic order $L = \langle v_1, v_2, v_3 \rangle$ for the query $Q(v_1, v_2, v_3) :- R(v_1, v_3), S(v_3, v_2)$. Direct access to the query answers according to that order would allow us to "jump over" the v_3 values via binary search and essentially enumerate the answers to $Q'(v_1, v_2) :- R(v_1, v_3), S(v_3, v_2)$. However, we know that Q' is not free-connex and that is impossible to achieve enumeration with quasilinear preprocessing and polylogarithmic delay (if sparseBMM holds). Therefore, the bounds we are hoping for are out of reach for the given query and order. The core difficulty is that the joining variable v_3 appears *after* the other two in the lexicographic order.

We formalize this notion of "variable in the middle" in order to detect similar situations in more complex queries.

Definition 3.2 (Disruptive Trio). Let Q be a CQ and L a lexicographic order of its free variables. We say that three free variables u_1, u_2, u_3 are a *disruptive trio* in Q with respect to L if u_1 and u_2 are not neighbors (i.e. they don't appear together in an atom), u_3 is a neighbor of both u_1 and u_2, and u_3 appears after u_1 and u_2 in L.

As it turns out, when considering free-connex and self-join-free CQs, the tractable CQs are precisely captured by this simple criterion. Regarding self-join-free CQs that are not free-connex, their known intractability of enumeration implies that direct access is also intractable. This leads to the following dichotomy:

Theorem 3.3. *Let Q be a CQ and L be a lexicographic order.*

- *If Q is free-connex and does not have a disruptive trio with respect to L, then direct access by L is possible in $\langle n \log n, \log n \rangle$.*
- *Otherwise, if Q is also self-join-free, then direct access by L is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n \rangle$ assuming sparseBMM and Hyperclique.*

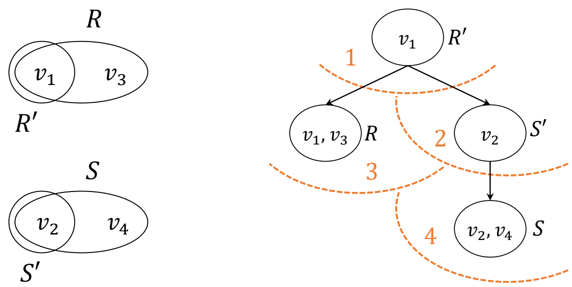
Remark 1. *On the positive side of Theorem 3.3, the preprocessing time is dominated by sorting the input relations, which we assume requires $O(n \log n)$ time. If we assume instead that sorting takes linear time (as assumed in some related work [8, 10, 21]), then the time required for preprocessing is only $O(n)$ instead of $O(n \log n)$.*

In Section 3.1, we provide an algorithm for this problem for full acyclic CQs that have a particular join tree that we call *layered*. Then, we show how to find such a layered join tree whenever there is no disruptive trio in Section 3.2. In Section 3.3, we explain how to adapt our solution for CQs with projections, and in Section 3.4 we prove a lower bound which establishes that our algorithm applies to *all* cases where direct access is tractable.

3.1 Layer-Based Algorithm

Before we explain the algorithm, we first define one of its main components. A *layered join tree* is a join tree where each node belongs to a layer. The layer number matches the position in the lexicographic order of the last variable that the node contains. Intuitively, "peeling" off the outermost (largest) layers must result in a valid join tree (for a hypergraph with fewer variables). To find such a join tree for a CQ Q, we may have to introduce hyperedges that are contained in those of $\mathcal{H}(Q)$ (this corresponds to taking the projection of a relation) or remove hyperedges of $\mathcal{H}(Q)$ that are contained in others (this corresponds to filtering relations that contain a superset of the variables). Thus, we define the layered join tree with respect to a hypergraph that is inclusion equivalent.

Definition 3.4 (Layered Join Tree). Let Q be a full acyclic CQ, and let $L = \langle v_1, \ldots, v_f \rangle$ be a lexicographic order. A *layered join tree* for Q with respect to L is a join tree of a hypergraph that is inclusion equivalent to $\mathcal{H}(Q)$ where (1) every node V of the tree is assigned



(a) A hypergraph that is inclusion equivalent to $\mathcal{H}(Q_3)$.

(b) A layered join tree for Q_3 w.r.t. the lexicographic order.

Figure 2: Constructing a layered join tree for the query $Q_3(v_1, v_2, v_3, v_4) :- R(v_1, v_3), S(v_2, v_4)$ and order $\langle v_1, v_2, v_3, v_4 \rangle$.

to layer $\max\{i \mid v_i \in V\}$, (2) there is exactly one node for each layer, and (3) for all $j \leq f$ the induced subgraph with only the nodes that belong to the first j layers is a tree.

Example 3.5. Consider the CQ

$$Q_3(v_1, v_2, v_3, v_4) :- R(v_1, v_3), S(v_2, v_4)$$

and the lexicographic order $\langle v_1, v_2, v_3, v_4 \rangle$. To support that order, we first find an inclusion equivalent hypergraph, shown in Figure 2a. Notice that we added two hyperegdes that are strictly contained in the existing ones. A layered join tree constructed from that hypergraph is depicted in Figure 2b. There are four layers, one for each node of the join tree. The layer of the node containing $\{v_1, v_3\}$ is 3 because v_3 appears after v_1 in the order and it is the third variable. If we remove the last layer, then we obtain a join tree for the induced hypergraph where the last variable v_4 is removed.

We now describe an algorithm that takes as an input a CQ Q, a lexicographic order L, and a corresponding layered join tree and provides direct access to the query answers after a preprocessing phase. For preprocessing, we leverage a construction from Carmeli et al. [10, Algorithm 2] and apply it to our layered join tree. For completeness, we briefly explain how it works below. Subsequently, we describe the access phase that takes into account the layers of the tree to accommodate the provided lexicographic order. We emphasize that the way we access the structure is different than that of the past work [10], and that this allows support of lexicographic orders that were impossible for the previous access routine (e.g. the order in Example 3.5).

Preprocessing. The preprocessing phase (1) creates a relation for every node of the tree, (2) removes dangling tuples, (3) sorts the relations, (4) partitions the relations into buckets, and (5) uses dynamic programming on the tree to compute and store certain counts. After preprocessing, we are guaranteed that for all i, the node of layer i has a corresponding relation where each tuple participates in at least one query answer; this relation is partitioned into buckets by the assignment of the variables preceding i. In each bucket, we sort the tuples lexicographically by v_i. Each tuple is given a weight that indicates the number of different answers this tuple agrees with when only joining its subtree. The weight of each bucket is the sum of its tuple weights. We denote both by the function `weight`. Moreover, for every tuple t, we compute the sum of

R'	w	s
a_1	8	0
a_2	8	8

S'	w	s
b_1	3	0
b_2	1	3

R		w	s
a_1	c_1	1	0
a_1	c_2	1	1
a_2	c_2	1	0
a_2	c_3	1	1

S		w	s
b_1	d_1	1	0
b_1	d_2	1	1
b_1	d_3	1	2
b_2	d_4	1	0

Figure 3: Example 3.6: The result of the preprocessing phase on Q_3, the layered join tree (Figure 2b) and an example database. The weight and start index for each tuple are abbreviated in the figure as w and s respectively.

weights of the preceding tuples in the bucket, denoted by $\texttt{start}(t)$. We use $\texttt{end}(t)$ for the sum that corresponds to the tuple following t in the same bucket; if t is last, we set this to be the bucket weight. If we think of the query answers in the subtree sorted in the order of v_i values, then `start` and `end` distribute the indices between 0 and the bucket weight to tuples. The number of indices within the range of each tuple corresponds to its weight.

Example 3.6 (Continued). The result of the preprocessing phase on an example database for our query Q_3 is shown in Figure 3. Notice that R has been split into two buckets according to the values of its parent R', one for value a_1 and one for a_2. For tuple $(a_1) \in R'$, we have $\texttt{weight}((a_1)) = 8$ because this is the number of answers that agree on that value in its subtree: the left subtree has 2 such answers which can be combined with any of the 4 possible answers of the right subtree. The start index of tuple $(b_1, d_3) \in S$ is the sum of the previous weights within the bucket: $\texttt{start}((b_1, d_3)) = \texttt{weight}((b_1, d_1)) + \texttt{weight}((b_1, d_2)) = 1 + 1 = 2$. Not shown in the figure is that every bucket stores the sum of weights it contains.

Access. The access phase works by going through the tree layer by layer. When resolving a layer i, we select a tuple from its corresponding relation, which sets a value for the ith variable in L, and also determines a bucket for each child. Then, we erase the node of layer i and its outgoing edges.

The access algorithm maintains a directed forest and an assignment to a prefix of the variables. Each tree in the forest represents the answers obtained by joining its relations. Each root contains a single bucket that agrees with the already assigned values, thus every answer agrees on the prefix. Due to the running intersection property, different trees cannot share unassigned variables. As a consequence, any combination of answers from different trees can be added to the prefix assignment to form an answer to Q. The answers obtained this way are exactly the answers to Q that agree with the already set assignment. Since we start with a layered join tree, we are guaranteed that at each step, the next layer (which corresponds to the variable following the prefix for which we have an assignment) appears as a root in the forest.

Recall that from the preprocessing phase, the weight of each root is the number of answers in its tree. When we are at layer i, we have to take into account the weights of all the other roots in order to compute the number of query answers for a particular tuple. More specifically, the number of answers to Q containing the already selected attributes (smaller than i) and some v_i value contained in a tuple is found by multiplying the tuple weight with the weights of all other roots. That is because the answers from all trees can be combined into a query answer. Let t be the selected tuple when resolving the i^{th} layer. The number of answers to Q that have a value of $L[i]$ smaller than that of t and a value of $L[j]$ equal to that of t for all $j < i$ is then:

$$\sum_{t'} \left(\mathtt{weight}(t') \prod_{r \in \text{roots}} \mathtt{weight}(r) \right)$$

where t' ranges over tuples preceding t in its bucket. Denote by factor the product of all root weights. Then we can rewrite as:

$$\left(\sum_{t'} \mathtt{weight}(t') \right) \left(\prod_{r \in \text{roots}} \mathtt{weight}(r) \right) = \mathtt{start}(t) \cdot \text{factor}\,.$$

Therefore, when resolving layer i we select the last tuple t such that the index we want to access is at least $\mathtt{start}(t) \cdot \text{factor}$.

Algorithm 1 Lexicographic Direct-Access

1: **if** $k \geq \mathtt{weight}(\text{root})$ **then**
2:     return out-of-bound
3: bucket[1] = root
4: factor = $\mathtt{weight}(\text{root})$
5: **for** i=1,...,f **do**
6:     factor = factor/$\mathtt{weight}(\text{bucket}[i])$
7:     pick $t \in \text{bucket}[i]$ s.t. $\mathtt{start}(t) \cdot \text{factor} \leq k < \mathtt{end}(t) \cdot \text{factor}$
8:     $k = k - \mathtt{start}(t) \cdot \text{factor}$
9:     **for** child V of layer i **do**
10:         get the bucket $b \in V$ agreeing with the selected tuples
11:         bucket[$\mathtt{layer}(V)$] = b
12:         factor = factor $\cdot\ \mathtt{weight}(b)$
13: return the answer agreeing with the selected tuples

Algorithm 1 summarizes the process we described where k is the index to be accessed and f is the number of variables. Iteration i resolves layer i. Pointers to the selected buckets from the roots are kept in a bucket array. The product of the weights of all roots is kept in a factor variable. In each iteration, the variable k is updated to the index that should be accessed among the answers that agree with the already selected attribute values. Note that bucket[i] is always initialized when accessed since layer i is guaranteed to be a child of a smaller layer.

Example 3.7 (Continued). We demonstrate how the access algorithm works for index $k = 12$. When resolving R', the tuple (a_2) is chosen since $8 \cdot 1 \leq 12 < 16 \cdot 1$; then, the single bucket in S' and the bucket containing a_2 in R are selected. The next iteration resolves S'. When it reaches line 7, $k = 12 - 8 = 4$ and factor $= 2$. As $0 \cdot 2 \leq 4 < 3 \cdot 2$, the tuple (b_1) is selected. Next, R is resolved, which we depict in Figure 4. The current index is $k = 4 - 0 = 4$. The weights of the other roots (only S here) gives us factor $= 3$. To make our choice in R, we multiply the weights of the tuples by factor $= 3$. Then, we find that the index k we are looking for falls into the range of (a_2, c_3) because $1 \cdot 3 \leq 4 < 2 \cdot 3$. Next, S is resolved, $k = 4 - 1 \cdot 3 = 1$, and factor $= 1$. As $1 \cdot 1 \leq 1 < 2 \cdot 1$, the tuple (b_1, d_2) is selected. Overall, answer number 12 (the 13^{th} answer) is (a_2, b_1, c_3, d_2).

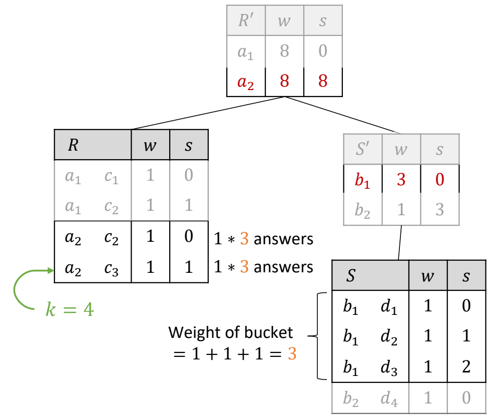


Figure 4: Example 3.7: Illustration of an iteration of the access phase where layer 3 corresponding to R is resolved.

Lemma 3.8. *Let Q be a full acyclic CQ, and $L = \langle v_1, \ldots, v_f \rangle$ be a lexicographic order. If there is a layered join tree for Q with respect to L, then direct access is possible in $\langle n \log n, \log n \rangle$.*

Proof. The correctness of Algorithm 1 follows from the discussion above. For the time complexity, note that it contains a constant number of operations (assuming the number of attributes f is fixed). Line 7 can be done in logarithmic time using binary search, while all other operations only require constant time in the RAM model. Thus, we obtain direct access in logarithmic time per answer after the quasilinear preprocessing (dominated by sorting). $\square$

With minor modifications, the algorithm we presented in this section can be used for the (reverse) task of *inverted access*. We describe this variation in Appendix B.

3.2 Finding Layered Join Trees

We now have an algorithm that can be applied whenever we have a layered join tree. We next show that the existence of such a join tree relies on the disruptive trio condition we introduced earlier. In particular, if no disruptive trio exists, we are able to construct a layered join tree for full acyclic CQs.

Lemma 3.9. *Let Q be a full acyclic CQ, and L be a lexicographic order. If Q does not have a disruptive trio with respect to L, then there is a layered join tree for Q with respect to L.*

Proof. We show by induction on i that there exists a layered join tree for the hypergraph containing the hyperedges $\{V \cap \{v_1, \ldots, v_i\} \mid V \in \mathtt{atoms}(Q)\}$ with respect to the prefix of L containing its first i elements. The induction base is the tree that contains the node $\{v_1\}$ and no edges.

In the inductive step, we assume a layered join tree with $i - 1$ layers for $\{V \cap \{v_1, \ldots, v_{i-1}\} \mid V \in \mathtt{atoms}(Q)\}$, and we build a layer on top of it. Denote by $\mathcal{V}$ the sets of $\{V \cap \{v_1, \ldots, v_i\} \mid V \in \mathtt{atoms}(Q)\}$ that contain v_i (these are the sets that need to be included in the new layer). First note that $\mathcal{V}$ is acyclic. Indeed, by the running intersection property, the join tree for $\mathcal{H}(Q)$ has a subtree with all the nodes that contain v_i. By taking this subtree and projecting out all variables that occur after v_i in L, we get a join tree for an inclusion equivalent hypergraph to $\mathcal{V}$, and its existence proves that $\mathcal{V}$ is acyclic.

We next claim that some set in $\mathcal{V}$ contains all the others; that is, there exists $V_m \in \mathcal{V}$ such that for all $V \in \mathcal{V}$, we have that $V \subseteq V_m$. Consider a join-tree for $\mathcal{V}$. Every variable of $\mathcal{V}$ defines a subtree induced by the nodes that contain this variable. If two variables are neighbors, their subtrees share a node. It is known that every collection of subtrees of a tree satisfies the *Helly property* [19]: if every two subtrees share a node, then some node is shared by all subtrees. In particular, since $\mathcal{V}$ is acyclic, if every two variables of $\mathcal{V}$ are neighbors, then some element of $\mathcal{V}$ contains all variables that appear in (elements of) $\mathcal{V}$. Thus, if, by way of contradiction, there is no such V_m, there exist two non-neighboring variables v_a and v_b that appear in (elements of) $\mathcal{V}$. Since v_i appears in all elements of $\mathcal{V}$, this means that there exist $V_a, V_b \in \mathcal{V}$ with $\{v_a, v_i\} \subseteq V_a$ and $\{v_b, v_i\} \subseteq V_b$. Since v_a and v_b are not neighbors, these three variables are a disruptive trio with respect to L: v_a and v_b are both neighbors of the later variable v_i. The existence of a disruptive trio contradicts the assumption of the lemma we are proving, and so we conclude that there is $V_m \in \mathcal{V}$ such that for all $V \in \mathcal{V}$, we have that $V \subseteq V_m$.

With V_m at hand, we can now add the additional layer to the tree given by the inductive hypothesis. By the inductive hypothesis, the layered join tree with $i-1$ layers contains the hyperedge $V_m \cap \{v_1, \ldots, v_{i-1}\} = V_m \setminus \{v_i\}$. We insert V_m with an edge to the node containing $V_m \setminus \{v_i\}$. This results in the join tree we need: (1) the hyperedges $\{V \cap \{v_1, \ldots, v_i\} \mid V \in \mathtt{atoms}(Q)\}$ are all contained in nodes, since the ones that do not appear in the tree from the inductive hypothesis are contained in the new node; (2) it is a tree since we add one leaf to an existing tree; and (3) the running intersection property holds since the added node is connected to all of its variables that already appear in the tree. □

Lemmas 3.8 and 3.9 give a direct-access algorithm for full acyclic CQs and lexicographic orders without disruptive trios.

3.3 Supporting Projection

Next, we show how to support CQs that have projections. A free-connex CQ can be efficiently reduced to a full acyclic CQ using Proposition 2.1. We next show that the resulting CQ contains no disruptive trio if the original CQ does not.

Lemma 3.10. *Given a database instance I, a free-connex CQ Q, and a lexicographic order L with no disruptive trio with respect to L, we can compute in linear time a database instance I' and a full acyclic CQ Q' with no disruptive trio with respect to L such that $Q'(I') = Q(I)$, $|I'| \le |I|$, and Q' does not depend on I or I'.*

Proof. See Appendix A.1. □

By combining Lemmas 3.8 to 3.10, we conclude an efficient algorithm for CQs and orders with no disruptive trios. The next lemma summarizes our results so far.

Lemma 3.11. *Let Q be a CQ, and L be a lexicographic order. If Q does not have a disruptive trio with respect to L, direct access by L is possible in $\langle n \log n, \log n \rangle$.*

3.4 Lower Bound for Conjunctive Queries

Next, we show that our algorithm supports all tractable cases (for self-join-free CQs); we prove that all unsupported cases are intractable.

Lemma 3.12. *Let Q be a self-join-free CQ, and L be a lexicographic order. If Q has a disruptive trio with respect to L, then direct access by L is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n \rangle$, assuming sparseBMM.*

Lemma 3.12 is a special case of the more general Lemma 4.5 that we prove later when we discuss partial lexicographic orders. Since Q has a disruptive trio, two non-neighboring variables u_1, u_2 are both neighbors of a later variable u_3 in L. Thus, u_1, u_3, u_2 is a chordless path, and Lemma 4.5 implies the correctness of Lemma 3.12.

By combining Lemma 3.11 and Lemma 3.12 together with the known hardness results for non-free-connex CQs (Theorem 2.2), we prove the dichotomy given in Theorem 3.3: direct access by a lexicographic order for a self-join-free CQ is possible with quasilinear preprocessing and polylogarithmic time per answer if and only if the query is free-connex and does not have a disruptive trio with respect to the required order.

4 PARTIAL LEXICOGRAPHIC ORDERS

We now investigate the case where the desired lexicographic order is *partial*, i.e., it contains only some of the free variables. This means that there is no particular order requirement for the rest of the variables. One way to achieve direct access to a partial order is to complete it into a full lexicographic order and then leverage the results of the previous section. If such a completion is impossible, we have to consider cases where tie breaking between the non-ordered variables is done in an arbitrary way. However, we will show in this section that the tractable partial orders are precisely those that can be completed into a full lexicographic order. In particular, we will prove the following dichotomy which also gives an easy-to-detect criterion for the tractability of direct access.

Theorem 4.1. *Let Q be a CQ and L be a partial lexicographic order.*

- *If Q is free-connex and L-connex and does not have a disruptive trio with respect to L, then direct access by L is possible in $\langle n \log n, \log n \rangle$.*
- *Otherwise, if Q is also self-join-free, then direct access by L is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n \rangle$, assuming sparseBMM and Hyperclique.*

Example 4.2. Consider the CQ $Q_7 \coloneq (x, y), S(y, z)$. If the free variables are exactly x and z, then the query is not free-connex, and so it is intractable. Next assume that all variables are free. If $L = \langle x, z \rangle$, then the query is not L-connex, and so it is intractable. If $L = \langle x, z, y \rangle$, then x, z, y is a disruptive trio, thus the query is intractable. However, if $L = \langle x, y, z \rangle$ or $L = \langle z, y \rangle$, then the query is free-connex, L-connex and has no disruptive trio, so it is tractable.

4.1 Tractable Cases

For the positive side, we can solve our problem efficiently if the CQ is free-connex and there is a completion of the lexicographic order to all free variables with no disruptive trio. Lemma 4.4 identifies these cases with a connexity criterion. To prove it, we first need a way to combine two different connexity properties. The proof of the following proposition uses ideas from a proof of the characterization of free-connex CQs in terms of the acyclicity of the hypergraph obtained by including a hyperedge with the free variables [5].

Proposition 4.3. *If a CQ Q is both L_1-connex and L_2-connex where $L_2 \subseteq L_1$, then there exists a join tree T of an inclusive extension of Q with a subtree T_1 containing exactly the variables L_1 and a subtree T_2 of T_1 contains exactly the variables L_2.*

Proof. See Appendix A.2. □

We are now in position to show the following:

Lemma 4.4. *Let Q be a CQ and L be a partial lexicographic order. If Q is free-connex and L-connex and does not have a disruptive trio with respect to L, then there is an ordering L^+ of $\mathtt{free}(Q)$ that starts with L such that Q has no disruptive trio with respect to L^+.*

Proof. According to Proposition 4.3, there is a join tree T (of an inclusive extension of Q) with a subtree $T_{\mathtt{free}}$ containing exactly the free variables, and a subtree T_L of $T_{\mathtt{free}}$ containing exactly the L variables. We assume that T_L contains at least one node; otherwise (this can only happen in case L is empty), we can introduce a node with no variables to all of T, $T_{\mathtt{free}}$ and T_L and connect it to any one node of $T_{\mathtt{free}}$. We describe a process of extending L while traversing $T_{\mathtt{free}}$. Consider the nodes of T_L as handled, and initialize $L^+ = L$. Then, repeatedly handle a neighbor of a handled node until all nodes are handled. When handling a node, append to L^+ all of its variables that are not already there. We prove by induction that Q has no disruptive trio w.r.t any prefix of L^+. The base case is guaranteed by the premises of this lemma since L (hence all of its prefixes) have no disruptive trio.

Let v_p be a new variable added to a prefix $v_1, \ldots, v_{p-1}$ of L^+. Let T^+ be the subtree of $T_{\mathtt{free}}$ with the handled nodes when adding v_p to L^+ and let $V \notin T^+$ be the node being handled. Note that, since v_p is being added, $v_p \in V$ but v_p is not in any node of T^+.

We first claim that every neighbor v_i of v_p with $i < p$ is in V. Our arguments are illustrated in Figure 5. Since v_i and v_p are neighbors, they appear together in a node $V_{i,p}$ outside of T^+. Let V_i be a node in T^+ containing v_i (such a node exists since v_i appears before v_p in L^+). Consider the path from $V_{i,p}$ to V_i. Let V_ℓ be the last node of this path not in T^+. If $V_\ell \neq V$, the path between V_ℓ and V goes only through nodes of T^+ (except for the end-points). Thus, concatenating the path from $V_{i,p}$ to V_ℓ with the path from V_ℓ to V results in a simple path. By the running intersection property, all nodes on this path contain v_p. In particular, the node following V_ℓ contains v_p in contradiction to the fact that v_p does not appear in T^+. Therefore, $V_\ell = V$. By the running intersection property, since V is on the path between V_i and $V_{i,p}$, we have that V contains v_i.

We now prove the induction step. We know by the inductive hypothesis that $v_1, \ldots, v_{p-1}$ have no disruptive trio. Assume by way of contradiction that appending v_p introduces a disruptive trio. Then, there are two variables v_i, v_j with $i < j < p$ such that v_i, v_p are neighbors, v_j, v_p are neighbors, but v_i, v_j are not neighbors. As we proved, since v_i and v_j are neighbors of v_p preceding it, we have that all three of them appear in the handled node V. This is a contradiction to the fact that v_i and v_j are not neighbors. □

The positive side of Theorem 4.1 is obtained by combining Lemma 4.4 with Theorem 3.3.

4.2 Intractable Cases

For the negative part, we prove a generalization of Lemma 3.12. For that, we use the hardness of Boolean matrix multiplication with a construction that is similar to that of Bagan et al. [3] for the hardness of enumeration on acyclic CQs that are not free-connex.

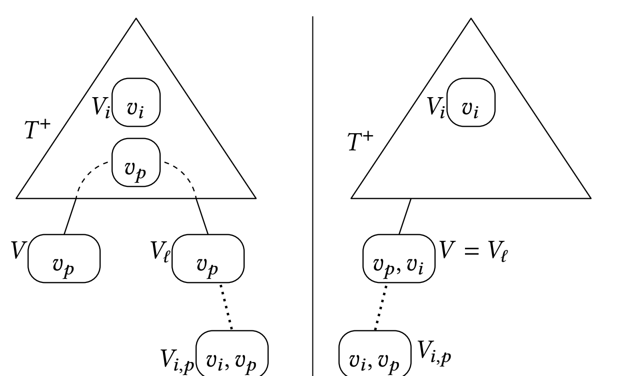
Lemma 4.5. *Let Q be a self-join-free CQ and L be a partial lexicographic order. If there is a chordless path $u_1, z_1, \ldots, z_k, u_2$ such that u_1 and u_2 appear in L and no variable z_i appears in L before any of them, then direct access by L is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n \rangle$, assuming* sparseBMM.

Proof. Let $U_3 = \{z_1, \ldots, z_k\}$. We encode Boolean matrix multiplication with Q such that, in the answers to Q, the assignments to u_1 and u_2 form the answers to the given matrix multiplication instance, the assignments to variables of U_3 can be skipped using binary search (given direct access), and all other variables are assigned a constant value $\perp$.

Let A and B be Boolean $n \times n$ matrices represented as binary relations. That is, $A \subseteq \{1, \ldots, n\}^2$, and $(a, b) \in A$ means that the entry in the ath row and bth column is 1. We define a partition of the atoms of Q where $\mathcal{R}_A$ is the set of all atoms that contain u_1, and $\mathcal{R}_B$ holds all other atoms. Note that no atom in $\mathcal{R}_A$ contains u_2 (since u_1 and u_2 are not neighbors) and no atom in $\mathcal{R}_B$ contains u_1. Given three values (a, b, c), we define a function $\tau_{(a,b,c)} : \mathtt{var}(Q) \to \{a, b, c, \perp\}$ as follows:

$$\tau_{(a,b,c)}(v) = \begin{cases} a & \text{if } v = u_1, \\ b & \text{if } v \in U_3, \\ c & \text{if } v = u_2, \\ \perp & \text{otherwise,} \end{cases}$$

For a vector $\vec{v}$, we denote by $\tau_{(a,b,c)}(\vec{v})$ the vector obtained by element-wise application of $\tau_{(a,b,c)}$. We define a database instance I over Q as follows: For every atom $R(\vec{v})$, if $R(\vec{v}) \in \mathcal{R}_A$ we set $R^I = \{\tau_{(a,b,\perp)}(\vec{v}) \mid (a, b) \in A\}$, and if $R(\vec{v}) \in \mathcal{R}_B$ we set $R^I =$



We get a contradiction in the case where $V \neq V_\ell$.

If v_i is a neighbor of v_p with $i < n$, then $v_i \in V$.

Figure 5: The induction step in Lemma 4.4

$\{\tau_{(\perp,b,c)}(\vec{v}) \mid (b,c) \in B\}$. Note that we do not define relations twice since $\mathcal{R}_A$ and $\mathcal{R}_B$ are disjoint and Q is self-join-free.

Since U_3 is connected, our construction guarantees that in every answer to Q all U_3 variables are assigned the same value. Since u_1 and $z_1 \in U_3$ are neighbors, we are guaranteed that there is an atom that contains them both in $\mathcal{R}_A$. The same holds for $z_k \in U_3$ and u_2 in $\mathcal{R}_B$. Therefore, the answers to $Q(I)$ describe the matrix multiplication. Consider a query answer q. We have that $q(u_1) = a$, $q(z_i) = b$ for all $z_i \in U_3$ and $q(u_2) = c$ for some $(a,b) \in A$ and $(b,c) \in B$. All other variables are mapped to the constant $\perp$. Note that the answers projected to u_1 and u_2 are the answers to the matrix multiplication problem.

Assume, by way of contradiction, that direct access to the answers of Q by a lexicographic order in which no variable of u_3 occurs before any of u_1 and u_2 is possible with $O(n \operatorname{polylog} n)$ preprocessing and $O(\operatorname{polylog} n)$ delay. We show how to find all the unique values of u_1 and u_2 in the answers efficiently. Perform the following starting with $i = 1$ and until there are no more answers. Access answer number i and print its assignment to (u_1, u_2). Then, set i to be the index of the next answer which assigns (u_1, u_2) to different values and repeat. Finding the next index can be done using binary search with a logarithmic number of direct accesses, each taking polylogarithmic time. Overall, we solve Boolean matrix multiplication in $O(n \operatorname{polylog} n)$ time, contradicting sparseBMM. □

The negative part of the dichotomy has three cases. First, if Q is not free-connex, then we know that direct access by any order is intractable according to Theorem 2.2. Next, if Q has a disruptive trio u_1, u_2, u_3 with respect to L, then u_1, u_3, u_2 is a chordless path satisfying the conditions of Lemma 4.5. The last case is that Q is not L-connex. In this case, there is an L-path, and this path satisfies the conditions of Lemma 4.5. Therefore, we obtain that the last two cases are hard too, assuming the sparseBMM hypothesis.

5 DIRECT ACCESS BY SUM OF WEIGHTS

We now consider direct access for the more general orderings based on Σw (the *sum* of attribute weights). As with lexicographic orderings, we are able to exhaustively classify the self-join-free CQs, even those with projections, in terms of tractability. We will show that direct access for Σw is significantly harder and tractable only for a small class of queries.

5.1 Overview of Results

The main result of this section is a dichotomy for direct access by Σw ordering:

Theorem 5.1. *Let Q be a CQ and w be a weight function.*

- *If Q is acyclic and an atom of Q contains all the free variables, then direct access by Σw is possible in $\langle n \log n, 1\rangle$.*
- *Otherwise, if Q is also self-join-free, direct access by Σw is not possible in $\langle n \operatorname{polylog} n, \operatorname{polylog} n\rangle$, assuming 3sum and Hyperclique.*

The proof of the negative part of the above theorem requires the query answers to express certain combinations of weights. If the query contains *independent* free variables, then its answers may contain all possible combinations of their corresponding attribute weights. We will thus rely on this independence measure to identify hard cases.

Definition 5.2 (Independent free variables). A set of vertices $V_I \subseteq V$ of a hypergraph $\mathcal{H}(V,E)$ is called independent iff no pair of these vertices appears in the same hyperedge, i.e., $|V_I \cap e| \leq 1$ for all $e \in E$. For a CQ Q, we denote by $\alpha_{\text{free}}(Q)$ the maximum number of variables among $\mathtt{free}(Q)$ that are independent in $\mathcal{H}(Q)$.

Intuitively, we can construct a database instance where each independent free variable is assigned to n different domain values with n different weights. By appropriately choosing the assignment of the other variables, all possible $n^{\alpha_{\text{free}}(Q)}$ combinations of these weights will appear in the query answers. Providing direct access then implies that we can retrieve these sums in ranked order.

Example 5.3. For $Q_8(x,y,z) :- R(x,z), S(z,y), T(y,u)$, we have $\alpha_{\text{free}}(Q_8) = 2$, namely for variables $\{x,y\}$. If the database instance is $R = [1,n] \times \{0\}$, $S = \{0\} \times [1,n]$, $T = [1,n] \times \{0\}$, then the n^2 query answers are $[1,n] \times [1,n] \times \{0\}$. The n values of x and y can be respectively assigned to any real-valued weights such that direct access on Q_8 retrieves their i^{th} sum in ranked order.

Our independence measure $\alpha_{\text{free}}(Q)$ is related to the classification of Theorem 5.1 in the following way:

Lemma 5.4. *For an acyclic CQ Q, an atom contains all the free variables iff $\alpha_{free}(Q) \leq 1$.*

Proof. See Appendix A.3. □

Therefore, the dichotomy of Theorem 5.1 can equivalently be stated using $\alpha_{\text{free}}(Q) \leq 1$ as a criterion. We chose to use the other criterion (all free variables contained in one atom) in the statement of our theorem statement as it is more straightforward to check. In the next section, we proceed to prove our theorem by showing intractability for all queries with $\alpha_{\text{free}}(Q) > 1$.

5.2 Proofs

For the hardness results, we rely mainly on the 3sum hypothesis. To more easily relate our direct-access problem to 3sum, which asks for the existence of a particular sum of weights, it is useful to define an auxiliary problem:

Definition 5.5 (weight lookup). Given a CQ Q, weight function w, and $\lambda \in \mathbb{R}$, *weight lookup* by Σw returns the first position of a query answer q of weight $w(q) = \lambda$ in the sorted array of answers.

The following lemma associates direct access with weight lookup via binary search on the query answers:

Lemma 5.6. *If the k^{th} query answer according to some ranking function can be directly accessed in $O(T_d(n))$ time for every k, then weight lookup can be performed in $O(T_d(n) \log n)$.*

Proof. See Appendix A.4. □

Lemma 5.6 implies that whenever we are able to support efficient direct access on the sorted array of query answers, weight lookup increases time complexity only by a logarithmic factor, i.e., it is also efficient. The main idea behind our reductions is that via weight lookups on a CQ with an appropriately constructed database, we can decide the existence of a zero-sum triplet over three distinct sets of numbers, thus hardness follows from 3sum. First, we consider the case of three independent variables that are free. These three variables are able to simulate a three-way Cartesian product in the query answers. This allows us to directly encode the 3sum triplets using attribute weights, obtaining a lower bound for direct access.

Lemma 5.7. *If a CQ Q is self-join-free and $\alpha_{free}(Q) \geq 3$, then direct access by Σw is not possible in $\langle n^{2-\epsilon}, n^{2-\epsilon} \rangle$ for any $\epsilon > 0$ assuming 3sum.*

Proof. Assume for the sake of contradiction that the lemma does not hold. We show that this would imply an $O(n^{2-\epsilon})$-time algorithm for 3sum. To this end, consider an instance of 3sum with integer sets A, B, and C of size n, given as arrays. We reduce 3sum to direct access over the appropriate query and input instance by using a construction similar to Example 5.3. Let x, y, and z be free and independent variables of Q, which exist because $\alpha_{\text{free}}(Q) \geq 3$. We create a database instance where x, y, and z take on each value in $[1, n]$, while all the other attributes have value 0. This ensures that Q has exactly n^3 answers—one for each (x, y, z) combination in $[1, n]^3$, no matter the number of atoms and the variables they contain. To see this, note that since x, y, and z are independent, they never appear together in an atom. Also, since Q is self-join-free, each relation appears once in the query, hence contains at most one of x, y, and z. Thus each relation either contains 1 tuple (if neither x, y, nor z is present) or n tuples (if one of x, y, or z is present). No matter on which attributes these relations are joined (including Cartesian products), the output result is always the "same" set $[1, n]^3 \times \{0\}^f$ of size n^3, where f is the number of free variables other than x, y, and z. (We use the term "same" loosely for the sake of simplicity. Clearly, for different values of f the query-result schema changes, e.g., consider Example 5.3 with z removed from the head. However, this only affects the number of additional 0s in each of the n^3 answer tuples, therefore it does not impact our construction.)

For the reduction from 3sum, weights are assigned to the attribute values as $w_x(i) = A[i]$, $w_y(i) = B[i]$, $w_z(i) = C[i]$, $i \in [1, n]$, and $w_u(0) = 0$ for all other attributes u. By our weight assignment, the weights of the answers are $A[i] + B[j] + C[k]$, $i, j, k \in [1, n]$, and thus in one-to-one correspondence with the possible value combinations in the 3sum problem. We first perform the preprocessing for direct access in $O(n^{2-\epsilon})$, which enables direct access to any position in the sorted array of query answers in $O(n^{2-\epsilon})$. By Lemma 5.6, weight lookup for a query result with zero weight is possible in $O(n^{2-\epsilon} \log n)$. Thus, we answer the original 3sum problem in $O(n^{2-\epsilon'})$ for any $0 < \epsilon' < \epsilon$, violating the 3sum hypothesis. □

For queries that do not have three independent free variables we need a slightly different construction. We show next that two variables are sufficient to encode partial 3sum solutions (i.e., pairs of elements), enabling a full solution of 3sum via weight lookups. This yields a weaker lower bound than Lemma 5.7, but still is sufficient to prove intractability according to our yardstick.

Lemma 5.8. *If a CQ Q is self-join-free and $\alpha_{free}(Q) = 2$, then direct access by Σw is not possible in $\langle n^{2-\epsilon}, n^{1-\epsilon} \rangle$ for any $\epsilon > 0$ assuming 3sum.*

Proof. See Appendix A.5. □

A special case of Lemma 5.8 is closely related to the problem of selection in X+Y [22], where we want to access the k^{th} smallest sum of pairs between two sets X and Y. This is equivalent to accessing the answers to $Q_{XY}(x, y) :– R(x), S(y)$ by Σw ordering. It has been shown that if X and Y are given sorted, then selection (a single access) is possible even in *linear* time [15, 26]. Thus, for Q_{XY} direct access by Σw is possible with $O(n \log n)$ preprocessing (where we simply sort the input relations) and $O(n)$ per access.

So far, we have covered all self-join-free CQs with $\alpha_{\text{free}}(Q) > 1$, which, by Lemma 5.4, proves the negative part of Theorem 5.1. Next, we show that the remaining acyclic CQs (those with $\alpha_{\text{free}}(Q) \leq 1$ or equivalently, an atom containing all the free variables) are tractable. For these queries, a single relation contains all the answers, and so direct access can easily be supported by reducing and sorting that relation.

Lemma 5.9. *If a CQ Q is acyclic and an atom contains all the free variables, then direct access by Σw is possible in $\langle n \log n, 1 \rangle$.*

Proof. See Appendix A.6. □

Combining these lemmas with the hardness of Boolean self-join-free cyclic CQs based on Hyperclique, gives a proof of Theorem 5.1.

6 SELECTION BY SUM OF WEIGHTS

Given that direct access by Σw order with quasilinear preprocessing and polylogarithmic delay is possible only in very few cases, we next investigate the tractability of a simpler version of the problem: When is *selection*, i.e., direct access to a *single* query answer, possible in quasilinear time? We further simplify the problem by not allowing any projections in the query, i.e., we limit our attention to *full* CQs. Our main result is a dichotomy theorem that covers all full self-join-free CQs.

6.1 Overview of Results

We show that the simplifications move only a narrow class of queries to the tractable side. For example, the 2-path query $Q_7(x, y, z) :– R(x, y), S(y, z)$ is tractable for selection (a single direct access), even though it is not for direct access. Still, the 3-path query $Q_9(x, y, z, u) :– R(x, y), S(y, z), T(z, u)$ remains intractable. Given that Q_7 and Q_9 both have two free and independent variables, a different criterion than $\alpha_{\text{free}}(Q)$ is needed for classification.

Definition 6.1 (Maximal Hyperedges). For a CQ Q with hypergraph $\mathcal{H}(Q) = (V, E)$, the *maximal number of hyperedges* w.r.t. containment is $\text{mh}(Q)$, i.e., $\text{mh}(Q) = \max |\{e \in E \mid \nexists e' \in R \land e \subseteq e'\}|$.

Note that for full CQs, $\alpha_{\text{free}}(Q) \leq \text{mh}(Q)$ since each independent variable can be associated with a distinct maximal hyperedge. We summarize the results of this section in the following theorem, which classifies full CQs Q based on $\text{mh}(Q)$:

Theorem 6.2. *Let Q be a full CQ and w be a weight function.*

- *If $\text{mh}(Q) \leq 2$, then selection by Σw is possible in $\langle 1, n \log n \rangle$.*
- *Otherwise, if Q is also self-join-free, then selection by Σw is not possible in $\langle 1, n\,\text{polylog}\, n \rangle$. assuming 3sum and Hyperclique.*

We prove the positive part of the theorem in Section 6.2 and the negative part in Section 6.3.

Example 6.3. For the query $Q_7(x, y, z) :– R(x, y), S(y, z)$ we have already shown in Section 5 that direct access by Σw is intractable. However, given that it has two maximal hyperedges, only one access (or a constant number of them) is in fact possible in $O(n \log n)$.

Absorbed atoms. We say that an atom (identified by its hyperedge) V is *absorbed* by an atom V' if $V \subseteq V'$. As evident from Theorem 6.2, adding to a query atoms that are absorbed by existing ones does not affect the complexity of selection. We prove this claim first and use it later in our analysis in order to treat queries that contain absorbed atoms.

A query Q' is a *contraction* of Q if every atom of Q' appears in Q, and all the rest of the atoms of Q are absorbed by some atom of Q'. Q^m is a *maximal contraction* of Q if it is a contraction and there is no Q'' that is a contraction of Q^m except itself. It is easy to see that the number of atoms of Q^m is $\mathrm{mh}(Q)$.

Example 6.4. Consider $Q(x, y, z) :– R(x, y), S(y), T(y, z), U(x, y)$. Here, $S(y)$ is absorbed by $R(x, y)$ and $U(x, y)$, and the latter two absorb each other. There are two maximal contractions that we can obtain from Q: either $Q^m_1 :– R(x, y), T(y, z)$ or $Q^m_2 :– T(y, z), U(x, y)$. The number of maximal hyperdges of Q is $\mathrm{mh}(Q) = 2$.

Lemma 6.5. *Selection on a CQ Q is possible in $O(T_S(n))$ if selection on a maximal contraction Q^m of Q is possible in $O(T_S(n))$. The converse is also true if Q is self-join-free.*

Proof. See Appendix A.7. □

6.2 Tractability Proofs

In this section, we provide tractability results for full CQs with $\mathrm{mh}(Q) \leq 2$. First, we consider the trivial case of $\mathrm{mh}(Q) = 1$ where the maximal contraction of Q has only one atom. The lemma below is a direct consequence of the linear-time array selection algorithm of Blum et al. [6].

Lemma 6.6. *For a full CQ Q with $mh(Q) = 1$, selection by Σw is possible in $\langle 1, n\rangle$.*

Proof. See Appendix A.8. □

For the $\mathrm{mh}(Q) = 2$ case, we rely on an algorithm by Frederickson and Johnson [15], which generalizes selection on the X+Y problem. If the two sets X and Y are given sorted, then the pairwise sums can be represented as a sorted matrix. A *sorted matrix M* contains a sequence of non-decreasing elements in every row and every column. For the $X + Y$ problem, a cell $M[i, j]$ contains the sum $X[i] + Y[j]$. Even though the matrix M has quadratically many cells, there is no need to construct it in advance given that we can compute each cell in constant time. Selection on a union of such matrices $\{M_1, \ldots, M_\ell\}$ asks for the k^{th} smallest cell among the cells of all matrices.

Theorem 6.7 ([15]). *Selection on a union of sorted matrices $\{M_1, \ldots, M_\ell\}$, where M_m has dimension $p_m \times q_m$ with $p_m \geq q_m$, is possible in time $O(\sum_{m=1}^{\ell} q_m \log(2p_m/q_m))$.*

Leveraging this algorithm, we provide our next positive result:

Lemma 6.8. *For a full CQ Q with $mh(Q) = 2$, selection by Σw is possible in $\langle 1, n \log n\rangle$.*

Proof. The maximal contraction of queries with $\mathrm{mh}(Q) = 2$ is $Q(\vec{X}, \vec{Y}) :– R(\vec{X}), S(\vec{Y})$, with $\vec{X} \neq \vec{Y}$, thus by Lemma 6.5, it is enough to prove an $O(n \log n)$ bound for this query. As before, we turn the attribute weights into tuple weights. Since a variable may occur in both atoms, we make sure to assign each attribute weight to only one relation to avoid double-counting. Thus, we compute $w(r) = \sum_{x \in \vec{X}} w_x(r(x))$ and $w(s) = \sum_{y \in (\vec{Y} \setminus \vec{X})} w_y(s(y))$ for all $r \in R$ and $s \in S$, respectively. Since the query is full, the weights of the query answers are in one-to-one correspondence with the pairwise sums of weights of tuples from R and S.

Let $\vec{Z} = \vec{X} \cap \vec{Y}$. We next group the R and S tuples by their Z values: we create ℓ buckets of tuples where all tuples t within a bucket have equal $t(z)$ values, $z \in \vec{Z}$. This can be done in linear time. If $\vec{Z} = \emptyset$, i.e., the query is the Cartesian product, then we place all tuples in a single bucket. For each assignment of $\vec{Z}$ values, the query answers with those values are formed by the Cartesian product of R and S tuples inside that bucket. Also, if the size of bucket m is n_m, then $n_1 + \ldots + n_\ell = |R| + |S|$. We sort the tuples in the buckets according to their weight in $O(n \log n)$ time. Assume R_m and S_m are the partitions of R and S in bucket m and $R_m[i]$ denotes the i^{th} tuple of R_m in sorted order (equivalently for $S_m[j]$). We define a union of sorted matrices $\{M_1, \ldots, M_\ell\}$ as follows: For bucket m, we have $M_m[i, j] = w(R_m[i]) + w(S_m[j])$. Selection on these matrices is equivalent to selection on the query answers of Q. By Theorem 6.7, if matrix M_m has dimension $p_m \times q_m$ with $p_m \geq q_m$, we can achieve selection in $O(\sum_{m=1}^{\ell} q_m \log(2p_m/q_m)) = O(\sum_{m=1}^{\ell} q_m \cdot 2p_m/q_m) = O(\sum_{m=1}^{\ell} p_m) = O(\sum_{m=1}^{\ell} n_m) = O(n)$. Overall, the time spent is $O(n \log n)$ because of sorting. □

6.3 Intractability Proofs

Though selection is a special case of direct access, we show that for most full CQs tractable time complexity $O(n \,\mathrm{polylog}\, n)$ is still unattainable. We start from the cases covered by Lemma 5.7. To extend that result to the selection problem, note that a selection algorithm can be repeatedly applied for solving direct access. For queries with three free and independent variables, an $O(n^{2-\epsilon})$ selection algorithm would imply a $\langle n^{2-\epsilon}, n^{2-\epsilon}\rangle$ direct-access algorithm, which we showed to be impossible. Therefore, the following immediately follows from Lemma 5.7:

Corollary 6.9. *If a full CQ Q is self-join-free and $\alpha_{free}(Q) \geq 3$, then selection by Σw is not possible in $\langle 1, n^{2-\epsilon}\rangle$ for any $\epsilon > 0$ assuming 3sum.*

This leaves only a small fraction of full acyclic CQs to be covered: queries with two or fewer independent variables and three or more maximal hyperedges. We next show that these queries are essentially variants of the general three-path query template where three atoms are organized in a chain.

Lemma 6.10. *The full acyclic CQs Q that satisfy $\alpha_{free}(Q) < 3$ and $mh(Q) > 2$ are $Q_{3g}(\vec{X}, \vec{Y}, \vec{Z}, \vec{U}) :– R(\vec{X}, \vec{Y}), S(\vec{Y}, \vec{Z}), T(\vec{Z}, \vec{U})$ for non-empty $\vec{X}, \vec{Y}, \vec{Z}, \vec{U}$, up to atom absorption.*

Proof. See Appendix A.9. □

Now that we established the precise form of the queries we want to classify, we proceed to prove their intractability. We approach this in a different way than the other hardness proofs: instead of relying on the 3sum hypothesis, we instead show that tractable selection would lead to unattainable bounds for Boolean cyclic queries.

Lemma 6.11. *Selection by* Σw *is not possible in* $\langle 1, n \operatorname{polylog} n\rangle$ *for* $Q_{3g}(\vec{X}, \vec{Y}, \vec{Z}, \vec{U}) :\!- R(\vec{X}, \vec{Y}), S(\vec{Y}, \vec{Z}), T(\vec{Z}, \vec{U})$ *assuming* Hyperclique.

Proof. We will show that if selection for Q_{3g} can be done in $O(n \operatorname{polylog} n)$, then the Boolean triangle query can be evaluated in the same time bound, which contradicts the Hyperclique hypothesis. Let $Q_{\triangle}() :\!- R'(x', y'), S'(y', z'), T'(z', x')$ be a query over a database I. We will construct a database I' for Q_{3g}, and via weight lookups we will be able to answer $Q_{\triangle}$ over I. Let $x \in \vec{X}, y \in \vec{Y}, z \in \vec{Z}, u \in \vec{U}$. For I', we extend relation R' to R by assigning $x = x', y = y'$ and setting the values of all the other attributes $(\vec{X} \cup \vec{Y}) \setminus \{x, y\}$ to a fixed domain value $\perp$. We repeat the same process for the other relations: For S we assign $y = y', z = z'$, and for T we assign $z = z', u = x'$. Consider a query result $q \in Q_{3g}(I')$. If $\pi_u(q) = \pi_x(q)$, then by our construction $\pi_{xyz}(q)$ satisfy R, S and T and thus, $Q_{\triangle}$ over I. We now assign weights as follows: If $\mathsf{dom} \subseteq \mathbb{R}$, then $w_x(i) = i, w_u(i) = -i$, and for all other attributes t, $w_t(i) = 0$. Otherwise, it is also easy to assign w_x and w_u in a way s.t. $w_x(i) = w_x(j)$ if and only if $i = j$ and $w_u(i) = -w_x(i)$. This is done by maintaining a lookup table for all the domain values that we map to some arbitrary real number. Then, we perform a weight lookup for Q_{3g} to identify if a query result with zero weight exists. If it does for some result q, then $w_x(\pi_x(q)) + \ldots + w_u(\pi_u(u)) = 0$ hence $\pi_x(q) = \pi_u(q)$ and $Q_{\triangle}$ is true, otherwise it is false. Since accessing the sorted array of Q_{3g} answers takes $O(n \operatorname{polylog} n)$, by Lemma 5.6, weight lookup also takes $O(n \operatorname{polylog} n)$. □

The negative part of Theorem 6.2 for acyclic queries is proved by combining Corollary 6.9 and Lemma 6.11 together with Lemma 6.10 and Lemma 6.5 that show we cover all cases. For self-join-free cyclic CQs, we once again resort to the hardness of their Boolean version based on Hyperclique.

7 CONCLUSIONS

We investigated the task of constructing a direct-access data structure to the output of a query with an ordering over the answers. We presented algorithms for fragments of the class of CQs for lexicographic orders and sum of weights. In these algorithms, the construction time is quasilinear in the size of the database, and the access time is logarithmic. We showed that within the class of CQs without self-joins, our algorithms cover all the cases where these complexity guarantees are feasible, assuming conventional hypotheses in the theory of fine-grained complexity. In the case of sum, where the tractable fragment is limited, we also studied the restriction of the problem to accessing a single answer (the selection problem) and established a corresponding classification for full CQs.

This work opens up several directions for future work, including the generalization to more expressive queries (CQs with self-joins, union of CQs, negation, etc.), other kinds of orders (e.g., min/max over the tuple entries), and a continuum of complexity guarantees (beyond ⟨quasilinear, logarithmic time⟩). It would also be important to understand how integrity constraints, such as functional dependencies, change the frontier of tractability as they have in the case of enumeration [9], deletion propagation [24], resilience [16], and probabilistic inference [18].

Generalizing the question posed at the beginning of the Introduction, we view this work as part of a bigger challenge that continues the line of research on *factorized representations* in databases [27, 28]: how can we represent the output of a query in a way that, compared to the explicit representation, is fundamentally more compact and efficiently computable, yet equally useful to downstream operations?

ACKNOWLEDGMENTS

Nofar Carmeli and Benny Kimelfeld were supported by the German Research Foundation (DFG) Project 412400621 (DIP program). Nofar Carmeli was also supported by a Google PhD Fellowship. Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald were supported in part by the National Science Foundation (NSF) under award number IIS-1956096. Wolfgang Gatterbauer was also supported by NSF under award number CAREER IIS-1762268.

REFERENCES

[1] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*. 434–443. https://doi.org/10.1109/FOCS.2014.53

[2] Nir Ailon and Bernard Chazelle. 2005. Lower Bounds for Linear Degeneracy Testing. *J. ACM* 52, 2 (2005), 157–171. https://doi.org/10.1145/1059513.1059515

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*. 208–222. https://doi.org/10.1007/978-3-540-74915-8_18

[4] Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. 2005. Subquadratic Algorithms for 3SUM. In *Algorithms and Data Structures*. 409–421. https://doi.org/10.1007/11534273_36

[5] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant delay enumeration for conjunctive queries: a tutorial. *SIGLOG* 7, 1 (2020), 4–33. https://doi.org/10.1145/3385634.3385636

[6] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. 1973. Time bounds for selection. *JCSS* 7, 4 (1973), 448 – 461. https://doi.org/10.1016/S0022-0000(73)80033-9

[7] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. 2021. Ranked Enumeration of MSO Logic on Words. In *ICDT*, Vol. 186. 20:1–20:19. https://doi.org/10.4230/LIPIcs.ICDT.2021.20

[8] Johann Brault-Baron. 2013. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre.* Ph.D. Dissertation. U. de Caen. https://hal.archives-ouvertes.fr/tel-01081392

[9] Nofar Carmeli and Markus Kröll. 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. *TCS* 64, 5 (2020), 828–860. https://doi.org/10.1007/s00224-019-09937-9

[10] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (Unions of) Conjunctive Queries Using Random Access and Random-Order Enumeration. In *PODS*. 393–409. https://doi.org/10.1145/3375395.3387662

[11] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *ICDT*, Vol. 186. 5:1–5:19. https://doi.org/10.4230/LIPIcs.ICDT.2021.5

[12] Jeff Erickson. 1995. Lower Bounds for Linear Satisfiability Problems. In *SODA*. 388–395. https://dl.acm.org/doi/10.5555/313651.313772

[13] Robert W. Floyd and Ronald L. Rivest. 1975. Expected Time Bounds for Selection. *Commun. ACM* 18, 3 (1975), 165–172. https://doi.org/10.1145/360680.360691

[14] Greg N. Frederickson. 1993. An Optimal Algorithm for Selection in a Min-Heap. *Inf. Comput.* 104, 2 (1993), 197–214. https://doi.org/10.1006/inco.1993.1030

[15] Greg N. Frederickson and Donald B. Johnson. 1984. Generalized Selection and Ranking: Sorted Matrices. *SIAM J. Comput.* 13, 1 (1984), 14–30. https://doi.org/10.1137/0213002

[16] Cibele Freire, Wolfgang Gatterbauer, Neil Immerman, and Alexandra Meliou. 2015. The Complexity of Resilience and Responsibility for Self-Join-Free Conjunctive Queries. *Proc. VLDB Endow.* 9, 3 (2015), 180–191. https://doi.org/10.14778/2850583.2850592

[17] Anka Gajentaan and Mark H Overmars. 1995. On a class of O(n2) problems in computational geometry. *Computational Geometry* 5, 3 (1995), 165 – 185. https://doi.org/10.1016/0925-7721(95)00022-2

[18] Wolfgang Gatterbauer and Dan Suciu. 2015. Approximate Lifted Inference with Probabilistic Databases. *Proc. VLDB Endow.* 8, 5 (2015), 629–640. https://doi.org/10.14778/2735479.2735494

[19] Martin Charles Golumbic. 1980. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, Chapter 4, 81 – 104. https://doi.org/10.1016/C2013-0-10739-8

[20] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *PODS*. 57–74. https://doi.org/10.1145/2902251.2902309

[21] Etienne Grandjean. 1996. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence* 16, 1 (1996), 183–236. https://doi.org/10.1007/BF02127798

[22] Donald B Johnson and Tetsuo Mizoguchi. 1978. Selecting the Kth element in X + Y and X_1 + X_2 + ... + X_m. *SIAM J. Comput.* 7, 2 (1978), 147–153. https://doi.org/10.1137/0207013

[23] Jens Keppeler. 2020. *Answering Conjunctive Queries and FO+MOD Queries under Updates.* Ph.D. Dissertation. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät. https://doi.org/10.18452/21483

[24] Benny Kimelfeld. 2012. A dichotomy in the complexity of deletion propagation with functional dependencies. In *PODS*. 191–202. https://doi.org/10.1145/2213556.2213584

[25] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. 2018. Tight Hardness for Shortest Cycles and Paths in Sparse Graphs. In *SODA*. 1236–1252. https://doi.org/10.1137/1.9781611975031.80

[26] A. Mirzaian and E. Arjomandi. 1985. Selection in X + Y and matrices with sorted rows and columns. *Inform. Process. Lett.* 20, 1 (1985), 13 – 17. https://doi.org/10.1016/0020-0190(85)90123-1

[27] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[28] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. In *ICDT*. 285–298. https://doi.org/10.1145/2274576.2274607

[29] Mihai Patrascu. 2010. Towards polynomial lower bounds for dynamic problems. In *STOC*. 603. https://doi.org/10.1145/1806689.1806772

[30] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2019. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *CoRR* abs/1911.05582 (2019). https://arxiv.org/abs/1911.05582

[31] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *PVLDB* 13, 9 (2020), 1582–1597. https://doi.org/10.14778/3397230.3397250

[32] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD*. 2659–2665. https://doi.org/10.1145/3318464.3383132

[33] Virginia Vassilevska Williams. 2015. Hardness of Easy Problems: Basing Hardness on Popular Conjectures such as the Strong Exponential Time Hypothesis (Invited Talk). In *IPEC*, Vol. 43. 17–29. https://doi.org/10.4230/LIPIcs.IPEC.2015.17

[34] Xiaofeng Yang, Mirek Riedewald, Rundong Li, and Wolfgang Gatterbauer. 2018. Any-k Algorithms for Exploratory Analysis with Conjunctive Queries. In *ExploreDB*. 1–3. https://doi.org/doi.org/10.1145/3214708.3214711

[35] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94. https://dl.acm.org/doi/10.5555/1286831.1286840

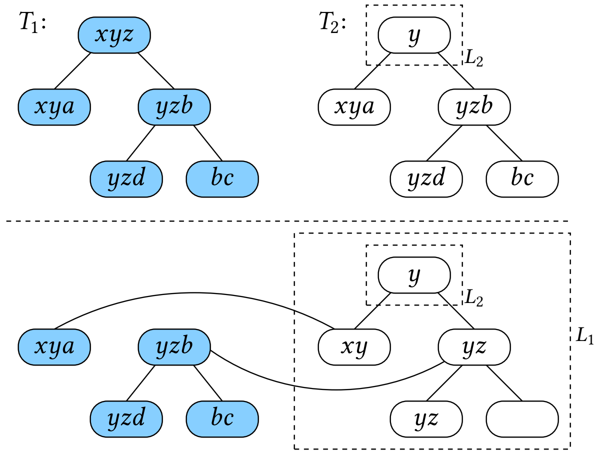


Figure 6: Example for the construction from Proposition 4.3 for the CQ $Q(x, y, z) :- R_1(x, y, a), R_2(y, z, b), R_3(b, c), R_4(y, z, d)$ with $L_1 = \{x, y, z\}$ and $L_2 = \{y\}$.

A ADDITIONAL PROOFS

A.1 Proof of Lemma 3.10

Let Q be a free-connex CQ, and let T be an ext-$\mathtt{free}(Q)$-connex tree for Q where T' is the subtree of T that contains exactly the free variables.

First, we claim that two free variables are neighbors in T iff they are neighbors in T'. The "if" direction is immediate since T' is contained in T. We show the other direction. Let u and v be free variables of Q that are neighbors in T. That is, there is a node V_T in T that contains them both. Consider the unique path from V to any node in T' such that only the last node on the path, which we denote $V_{T'}$, is in T'. Since both variables appear in T' and in V, by the running intersection property, both variables appear in $V_{T'}$. Thus, u and v are also neighbors in T'.

Since the definition of disruptive trios depends only on neighboring pairs of free variables, an immediate consequence of the claim from the previous paragraph is that there is a disruptive trio in T iff there is a disruptive trio in T'. Next, we can simply use Proposition 2.1 to reduce Q to the full acyclic CQ where the atoms are exactly the nodes of T'.

A.2 Proof Sketch of Proposition 4.3

We describe a construction of the required tree. Figure 6 demonstrates our construction. We use two different characterizations of connexity. Since Q is L_2-connex, it has an ext-L_2-connex tree T_2. Since Q is L_1-connex, there is a join-tree T_1 for the atoms of Q and its head. Let $T_2[L_1]$ be T_2 where the variables that are not in L_1 are deleted from all nodes. That is, for every node $V \in T_2$, its variables are replaced with $\mathtt{var}(V) \cap L_1$. Denote by $\mathcal{V}$ all neighbors of the head in T_1, and denote by T_1^- the graph T_1 after the deletion of the head node. Taking both $T_2[L_1]$ and T_1^- and connecting every node $V_1 \in \mathcal{V}$ with a node V_2 of $T_2[L_1]$ such that $\mathtt{var}(V_1) \cap L_1 = \mathtt{var}(V_2)$ gives us the tree we want. Such a node exists in $T_2[L_1]$ since every node of T_1^- represents an atom of Q, and every atom of Q is contained in some node of T_2. The subtree $T_2[L_1]$ contains exactly V_1, and since this subtree comes from an ext-L_2-connex tree, it has a subtree containing exactly L_1. It is easy to verify that the result is a tree, and we can show that the running intersection property holds in the united graph since it holds for T_1 and T_2.

A.3 Proof of Lemma 5.4

The "only if" part is trivial. For $\alpha_{\text{free}}(Q) = 1$ and acyclic query Q, we prove that there is an atom $R_f(\vec{X}_f)$ which contains all the free variables. First note that for $|\mathtt{free}(Q)| = 1$ this is trivially true. For $|\mathtt{free}(Q)| > 1$, let V be a node in the join tree (corresponding to some atom of Q) that contains the maximum number of free variables and assume for the sake of contradiction that there exists a free variable y with $y \notin V$. We use $\mathcal{V}_y$ to denote the set of nodes in the join tree that contain variable y; thus $V \notin \mathcal{V}_y$. From Q being acyclic follows that the nodes in $\mathcal{V}_y$ form a connected graph and there exists a node V' that lies on every path from V to a node in $\mathcal{V}_y$. Since $\alpha_{\text{free}}(Q) = 1$, each variable $x \in V$ must appear together with y in some query atom, implying that x appears in some node $V'' \in \mathcal{V}_y$. From that and the running intersection property follows that x must also appear in V' since V' lies on the path from V to any such V''. Hence V' contains y and all the V variables, violating the maximality assumption for V.

For $\alpha_{\text{free}}(Q) = 0$, Q is a Boolean query and any atom trivially contains the empty set.

A.4 Proof of Lemma 5.6

We use binary search on the sorted array of query answers. Each direct access returns a query answer whose weight can be computed in $O(1)$. Thus, in a logarithmic number of accesses we can find the first occurrence of the desired weight. Since the number of answers is polynomial in n, the number of accesses is $O(\log n)$ and each one takes $O(T_d)$ time.

A.5 Proof of Lemma 5.8

We show that the contrary contradicts the 3SUM hypothesis. Let A, B, and C be three integer arrays of a 3SUM instance of size n. We construct a database instance with attribute weights like in the proof of Lemma 5.7, but now with only 2 free and independent variables x and y. Hence the weights of the n^2 query results are in one-to-one correspondence with the corresponding sums $A[i] + B[j]$, $i, j \in [1, n]$. We run the preprocessing phase for direct access in $O(n^{2-\epsilon})$, which allows us to access the sorted array of query results in $O(n^{1-\epsilon})$. For each value $C[k]$ in C, we perform a weight lookup on Q for weight $-C[k]$, which takes time $O(n^{1-\epsilon} \log n)$ (Lemma 5.6). If that returns a valid index, then there exists a pair (i, j) of A and B with sum $A[i]+B[j] = -C[k]$, which implies $A[i]+B[j]+C[k] = 0$; otherwise no such pair exists. Since there are n values in C, total time complexity is $O(n \cdot n^{1-\epsilon} \log n) = O(n^{2-\epsilon} \log n)$. This procedure solves 3SUM in $O(n^{2-\epsilon'})$ for any $0 < \epsilon' < \epsilon$, violating the 3SUM hypothesis.

A.6 Proof of Lemma 5.9

Since all free variables appear in one atom $R_f(\vec{X}_f)$, we can apply a linear-time semi-join reduction by Yannakakis [35] to remove the dangling tuples, and then compute Q by projecting R on all free variables. Then, we sort the query answers by Σw, which takes

Algorithm 2 Lexicographic Inverted-Access

$k = 0$
bucket$[1]$ = root
factor = weight(root)
for i=1,…,f **do**
  factor = factor/weight(bucket$[i]$)
  select $t \in$ bucket$[i]$ agreeing with the answer
  if no such t exists **then**
    return not-an-answer
  $k = k + \mathtt{start}(t) \cdot$ factor
  for child V of layer i **do**
    get the bucket $b \in V$ agreeing with the answer
    bucket[layer(V)] = b
    factor = factor $\cdot$ weight(b)
return k

total time $O(n \log n)$ for preprocessing and enables constant-time direct access to individual answers in ranked order.

A.7 Proof of Lemma 6.5

For the "if" direction, we can eliminate absorbed atoms from Q to obtain Q^m after making sure that the tuples in the database satisfy those atoms. Thus, to remove an atom $S(\vec{Y})$ which is absorbed by $R(\vec{X})$, we filter the relation R based on the tuples of S. Then, Q^m over the filtered database has the same answers as Q over the original one. For the "only if" direction, each atom $S(\vec{Y})$ that appears in Q but not Q^m is absorbed by some $R(\vec{X})$. We create the relation S by copying $\pi_{\vec{Y}}(R)$ into it, essentially making the atom $S(\vec{Y})$ obsolete. Note that we are allowed to create S without restrictions because Q has no self-joins, hence the database doesn't already contain the relation. Then, Q over the extended database has the same answers as Q^m over the original one. The above reductions take linear time, which is dominated by $T_S(n)$ since $T_S(n)$ is trivially in $\Omega(n)$ for the selection problem.

A.8 Proof of Lemma 6.6

By Lemma 6.5, it suffices to solve selection on the query $Q(\vec{X}) \coloneq R(\vec{X})$, which is a maximal contraction of all queries with $\mathrm{mh}(Q) = 1$. Initially, we turn the attribute weights into tuple weights. For each tuple $r \in R$, we compute its weight as $w(r) = \sum_{x \in \vec{X}} w_x(r(x))$. Thus, the weights $w(r)$ are the weights of the query answers. This takes $O(n)$ for the $O(n)$ tuples of R. Then, applying linear-time selection [6] on R gives us the k^{th} smallest query result.

A.9 Proof of Lemma 6.10

First, for $\alpha_{\text{free}}(Q) = 1$, we have by Lemma 5.4 that an atom contains all free variables, thus $\mathrm{mh}(Q) = 1$ For the case of $\alpha_{\text{free}}(Q) = 2$, let x and u be the two independent variables. Because they do not appear together in the same atom, there exist two different atoms e_R, e_T such that e_R contains x but not u and e_T contains u but not x. Without loss of generality, we can further assume that the variable sets in these atoms are not strictly contained in others (if they are, we can choose those instead). We can also assume that our choice of independent variables x, y and atoms e_R, e_T is such that these two atoms do not have any variables in common, otherwise Q would be cyclic. We also have at least one more maximal atom-hyperedge e_S, that is not absorbed by e_R or e_T because $\mathrm{mh}(Q) > 2$. For the variables of e_S, we claim that $\mathtt{var}(e_S) \subseteq (\mathtt{var}(e_R) \cup \mathtt{var}(e_T))$. Suppose that e_S contains a variable s s.t. $s' \notin (\mathtt{var}(e_R) \cup \mathtt{var}(e_T))$. Then because t cannot be independent, there must exist an atom e_U that contains x and t (or u and t). However, in that case, e_R, e_S, e_U (or e_T, e_S, e_U) create a cycle violating the acylicity of Q. Let $\vec{Y}$ be the variables in $\mathtt{var}(e_R) \cap \mathtt{var}(e_S)$ and $\vec{Z}$ those in $\mathtt{var}(e_S) \cap \mathtt{var}(e_T)$. We have $\vec{Y} \neq \emptyset$ and $\vec{Z} \neq \emptyset$, otherwise e_S would be absorbed by e_R or e_T respectively. Conversely, $\mathtt{var}(e_R) \not\subset \mathtt{var}(e_S)$ because e_R would be absorbed by e_S, and the same is true for e_T. At this point, the other atoms of the query can only be absorbed by the existing ones, otherwise we introduce an independent variable or a cycle.

B INVERTED ACCESS BY LEXICOGRAPHIC ORDER

A straightforward adaptation of Algorithm 1 can be used to achieve *inverted access*: given a query result as the input, we return its index according to the lexicographic order. Algorithm 2 is almost the same algorithm as Algorithm 1 except that the choices in each iteration are made according to the given answer and the corresponding index is constructed (instead of the opposite). The algorithm runs in constant time per answer since every operation can be done within that time (unlike Algorithm 1, there is no need for binary search here).