Tractable Conjunctive Queries over Static and Dynamic Relations

Ahmet Kara[1]
ahmet.kara@uzh.ch

Zheng Luo[1]
zheng.luo@uzh.ch

Milos Nikolic[2]
milos.nikolic@ed.ac.uk

Dan Olteanu[1]
dan.olteanu@uzh.ch

Haozhe Zhang[1]
haozhe.zhang@uzh.ch

[1]University of Zurich [2]University of Edinburgh

Abstract

We investigate the evaluation of conjunctive queries over static and dynamic relations. While static relations are given as input and do not change, dynamic relations are subject to inserts and deletes.

We characterise syntactically three classes of queries that admit constant update time and constant enumeration delay. We call such queries *tractable*. Depending on the class, the preprocessing time is linear, polynomial, or exponential (under data complexity, so the query size is constant).

To decide whether a query is tractable, it does not suffice to analyse separately the sub-query over the static relations and the sub-query over the dynamic relations. Instead, we need to take the interaction between the static and the dynamic relations into account. Even when the sub-query over the dynamic relations is not tractable, the overall query can become tractable if the dynamic relations are sufficiently constrained by the static ones.

Acknowledgements. The authors would like to acknowledge the UZH Global Strategy and Partnerships Funding Scheme and EPSRC grant EP/T022124/1.

1 Introduction

Incremental view maintenance, also known as fully dynamic query evaluation, is a fundamental task in data management, e.g., [21, 13, 10, 15, 28, 18, 29, 9]. A natural question is to understand which queries are tractable, i.e., which queries admit constant time per single-tuple update (insert or delete) and also constant delay for the enumeration of the result tuples. The problem setting also allows for some one-off preprocessing phase to construct a data structure that supports the updates and the enumeration. Prior work [4] showed that the q-hierarchical queries are the conjunctive queries that are tractable; this already holds even if we only allow for linear time preprocessing. All other queries cannot admit both constant update time and constant enumeration delay, even when we allow arbitrary time for preprocessing. If we only allow inserts (and no deletes), then every free-connex α-acyclic conjunctive query becomes tractable [29, 20]. The tractable queries with free access patterns, where the free variables are partitioned into input and output, naturally extend the q-hierarchical queries, which are queries without input variables [17]. Further works investigated classes of intractable queries for which the update time and enumeration delay are shown to be not constant yet worst-case optimal, e.g., triangle queries [14, 15] and hierarchical queries with arbitrary free variables [16, 19]. Further works consider restrictions to the data or to the update sequence: Intractable queries become tractable when the update sequence has a small enclosureness parameter [29] or when the database satisfies functional dependencies [18], bounded degrees [5], or more general integrity constraints [6].

All aforementioned works consider the *all-dynamic setting*, where all relations are updateable. In this work, we extend the tractability frontier by considering a *mixed setting*, where the input database can have

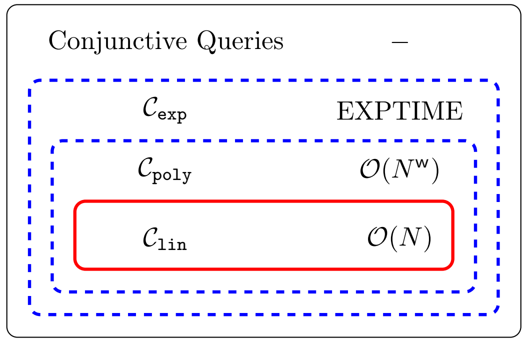


Figure 1: Classes $\mathcal{C}_{\texttt{lin}} \subset \mathcal{C}_{\texttt{poly}} \subset \mathcal{C}_{\texttt{exp}}$ of tractable queries over static and dynamic relations and the corresponding preprocessing time (data complexity). N is the current database size and w is the preprocessing width. The solid (red) border around the class $\mathcal{C}_{\texttt{lin}}$ states that there is a dichotomy between the queries inside and outside the class. The dashed (blue) border around a class states that no dichotomy is known for queries inside and outside the class.

both dynamic relations, which are subject to change, and static relations, which do not change. The mixed setting appears naturally in real-world applications at RelationalAI (personal communication). For instance, in the retailer domain, the Inventory and Sales relations are updated very frequently, whereas the Stores and Demographics relations are updated very infrequently and can be considered static for a large time period. We show that by differentiating between static and dynamic relations, we can design efficient query maintenance strategies tailored to the mixed setting.

Main Contributions

We characterise syntactically three classes of tractable conjunctive queries depending on their preprocessing time, cf. Figure 1: $\mathcal{C}_{\texttt{lin}} \subset \mathcal{C}_{\texttt{poly}} \subset \mathcal{C}_{\texttt{exp}}$. These classes are defined in Section 4.

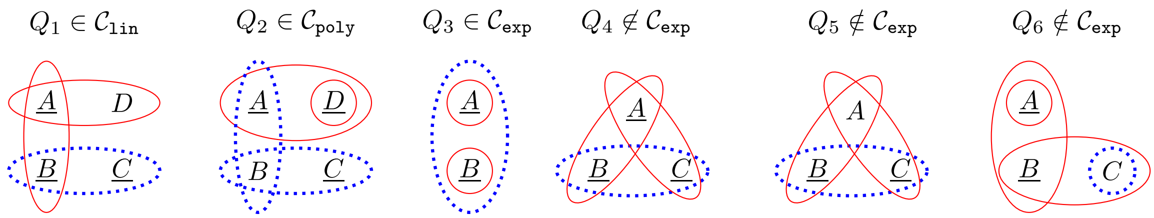
The class $\mathcal{C}_{\texttt{lin}}$ defines the tractable queries with linear time preprocessing:

Theorem 1. *Let a CQ Q and a database of size N.*

- *If Q is in $\mathcal{C}_{\texttt{lin}}$, then it can be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.*
- *If Q is not in $\mathcal{C}_{\texttt{lin}}$ and has no self-joins, then it cannot be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay, unless the Online Matrix-Vector Multiplication or the Boolean Matrix-Matrix Multiplication conjecture fail.*

The upper bound in Theorem 1 relies on a rewriting of a given query using multiple strata of views, where the views are defined by projecting or joining input relations or other views (Section 3). We call such rewritings *safe* if the views can be maintained in constant time under single-tuple updates to the input relations and support constant-delay enumeration of the query result. We show that every $\mathcal{C}_{\texttt{lin}}$ query has a safe rewriting and the views can be computed in linear time (Section 5). The lower bound in Theorem 1 relies on two widely used conjectures: the Online Matrix-Vector Multiplication [12] and the Boolean Matrix-Matrix Multiplication [3]. The proof of the lower bound is outlined in Section 7.

Example 2. *Let the query $Q_1(A, B, C) = R^d(A, D), S^d(A, B), T^s(B, C)$ in Figure 2. The dynamic relations R and S are adorned with the superscript d, while the static relation T is adorned with s. The query is not tractable in the all-dynamic setting (as it is not q-hierarchical, cf. Section 2), yet it is in $\mathcal{C}_{\texttt{lin}}$, so it is tractable and uses linear time preprocessing.*



$$Q_1(A,B,C) = R^d(A,D), S^d(A,B), T^s(B,C) \qquad Q_2(A,C,D) = R^d(A,D), S^s(A,B), T^s(B,C), U^d(D)$$
$$Q_3(A,B) = R^d(A), S^s(A,B), T^d(B) \qquad Q_4(A,B,C) = R^d(A,B), S^d(A,C), T^s(B,C)$$
$$Q_5(B,C) = R^d(A,B), S^d(A,C), T^s(B,C) \qquad Q_6(A,B) = R^d(A), S^d(A,B), T^d(B,C), U^s(C)$$

Figure 2: Examples of queries inside and outside our tractability classes. The static and dynamic relations are adorned with the superscripts s and respectively d. In the hypergraphs, there is one node per variable and one hyperedge per relation. Underlined variables are free. Solid (red) hyperedges denote the dynamic relations, the dotted (blue) hyperedges denote the static relations.

In the all-static setting, $\mathcal{C}_{\texttt{lin}}$ becomes the class of free-connex α-acyclic queries, which are those conjunctive queries that admit constant enumeration delay after linear time preprocessing [3]. In the all-dynamic setting, $\mathcal{C}_{\texttt{lin}}$ becomes the class of q-hierarchical queries, which are those conjunctive queries that admit constant enumeration delay and constant update time after linear time preprocessing. Every query in $\mathcal{C}_{\texttt{lin}}$ is free-connex α-acyclic and its dynamic sub-query, which is obtained by removing the static relations, is q-hierarchical. Yet the queries in $\mathcal{C}_{\texttt{lin}}$ need to satisfy further syntactic constraints on the connections between their static and dynamic relations: For instance, Q_3 and Q_4 from Figure 2 are not in $\mathcal{C}_{\texttt{lin}}$ even though they are free-connex α-acyclic and their dynamic sub-queries are q-hierarchical.

Queries in $\mathcal{C}_{\texttt{poly}} \setminus \mathcal{C}_{\texttt{lin}}$ are tractable but require super-linear time preprocessing that depends on a new width measure w, which we call the *preprocessing width* of the query:

Theorem 3. *Every query in $\mathcal{C}_{\texttt{poly}}$ can be evaluated with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay, where N is the database size and w is the preprocessing width of the query.*

Like the queries in $\mathcal{C}_{\texttt{lin}}$, every query in $\mathcal{C}_{\texttt{poly}}$ also admits a safe rewriting (Section 5).

Example 4. *The query $Q_2(A,C,D) = R^d(A,D), S^s(A,B), T^s(B,C), U^d(D)$ from Figure 2 is contained in $\mathcal{C}_{\texttt{poly}} \setminus \mathcal{C}_{\texttt{lin}}$: It is tractable but requires quadratic time preprocessing. The quadratic blowup is due to the creation of a view that joins the static relations S and T on the bound variable B.*

The preprocessing width is not captured by previous width notions, such as the fractional hypertree width (fhtw) of the static sub-query or of the entire query [22]. Let us take the free-connex α-acyclic query $Q_7(A,B,C) = R^s(A,B), S^s(B,C), T^s(A,C), U^d(A,B,C)$, whose fhtw is 1. For its static sub-query, which is the triangle join, fhtw is 3/2. The preprocessing width is 1, so Q_7 is in $\mathcal{C}_{\texttt{lin}}$. The triangle join $Q_8(A,C) = R^s(A,B), S^s(B,C), T^d(A,C)$ has fhtw = 3/2 and its static sub-query has fhtw = 2. The preprocessing width is 2: We need to materialize the view $V^s(A,C) = R^s(A,B), S^s(B,C)$, which is the static sub-query. We may reduce the preprocessing width to 3/2 for the static sub-query by also joining with the dynamic relation $T^d(A,C)$, yet the modified view becomes dynamic and needs to be maintained under updates to T. However, this maintenance cannot be achieved with constant update time, while allowing for constant enumeration delay [15].

The class $\mathcal{C}_{\texttt{exp}}$ characterises tractable queries that can use exponential time preprocessing:

Theorem 5. *Every query in $\mathcal{C}_{\texttt{exp}}$ can be evaluated with $2^p \cdot p^2$ time preprocessing, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay, where N is the database size, $p = \mathcal{O}(N^{\rho^*(stat(Q))})$, $stat(Q)$ is the static sub-query of Q, and ρ^* is the fractional edge cover number.*

The class $\mathcal{C}_{\texttt{exp}}$ is merely of theoretical interest since it comes with preprocessing time that is exponential in the size of the input database. The reason for this high preprocessing time is to ensure constant enumeration delay; if we would allow the enumeration delay to become linear, then the preprocessing time would collapse to linear for the acyclic queries in $\mathcal{C}_{\texttt{exp}}$. We use the fractional edge cover number as it characterises the worst-case optimal size of the static sub-query result as a function of the size N of the input relations [2].

Example 6. *The query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ from Figure 2 is in $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$. The possible results of Q_3 are completely predetermined by its static sub-query: They are the subsets of the materialized static sub-query, while the updates to the dynamic relations only act as selectors in this powerset. Such queries require exponential time preprocessing as there are 2^N many possible subsets of the relation S^s of size N. Its dynamic sub-query, i.e., the sub-query over the dynamic relations, is q-hierarchical. Its static sub-query, i.e., the sub-query over the static relation, is trivially free-connex α-acyclic. This means that,* when taken in isolation*, the dynamic sub-query can be evaluated with constant update time and enumeration delay after linear time preprocessing, while the static sub-query can be evaluated with constant enumeration delay after linear time preprocessing. Yet Q_3 is not in $\mathcal{C}_{\texttt{lin}}$: It does not admit constant update time and enumeration delay* after linear time preprocessing*. The queries Q_4, Q_5, and Q_6 from Figure 2 are not in $\mathcal{C}_{\texttt{exp}}$: Their dynamic sub-queries are not covered by the static sub-queries. They are discussed in Section 8.*

Queries in $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$ may not admit safe rewritings that rely solely on joins and projections. Take again $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ from Figure 2. There is no safe rewriting of this query that solely relies on projections and joins. Any rewriting that supports constant-delay enumeration of the query result must contain a view that either joins: (i) R^d and S^s; (ii) S^s and T^d; or (iii) R^d and T^d. None of these views can be maintained with constant update time. Consider the view $V(A, B) = R^d(A), S^s(A, B)$. An insert of a value a to R requires to iterate over all B-values paired with a in S^s in order to propagate the change to the view V. The number of such B-values can be linear, which implies that the update time is linear. Section 6 gives our evaluation strategy for queries in $\mathcal{C}_{\texttt{exp}}$.

2 Preliminaries

We use $\mathbb{N}$ to denote the set of natural numbers including 0. For $n \in \mathbb{N}$, we define $[n] = \{1, 2, \ldots, n\}$. In case $n = 0$, we have $[n] = \emptyset$.

Databases with Static and Dynamic Relations A *schema* is a tuple of variables. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. The domain of a variable X is denoted by $\mathsf{Dom}(X)$. A value tuple $\boldsymbol{t}$ over schema $\boldsymbol{X} = (X_1, \ldots, X_n)$ is an element from $\mathsf{Dom}(\boldsymbol{X}) = \mathsf{Dom}(X_1) \times \cdots \times \mathsf{Dom}(X_n)$. A *relation* over schema $\boldsymbol{X}$ is a finite set of value tuples over the same schema. The size $|R|$ of a relation R is the number of its tuples. The relation R is called *dynamic* if it is subject to changes; otherwise, it is called *static*. To emphasize that R is static or dynamic, we write R^s or respectively R^d. A database is a finite set of relations. The size of a database is the sum of the sizes of its relations.

Conjunctive Queries A conjunctive query (CQ) is of the form

$$Q(\boldsymbol{X}) = R_1^d(\boldsymbol{X}_1), \ldots, R_k^d(\boldsymbol{X}_k), S_1^s(\boldsymbol{Y}_1) \ldots, S_\ell^s(\boldsymbol{Y}_\ell)$$

where each $R_i^d(\boldsymbol{X}_i)$ is a dynamic atom; each $S_j^s(\boldsymbol{Y}_j)$ is a static atom; $\mathit{vars}(Q) = \bigcup_{i\in[k]} \boldsymbol{X}_i \cup \bigcup_{j\in[\ell]} \boldsymbol{Y}_j$ is the set of variables of Q; $\mathit{free}(Q) = \boldsymbol{X} \subseteq \mathit{vars}(Q)$ is the set of *free* variables, while all others are *bound* variables; $\mathit{atoms}(Q)$ is the set of all atoms of Q; $\mathit{atoms}(X)$ is the set of the atoms containing the variable X in their schema. The static (dynamic) sub-query $\mathit{stat}(Q)$ ($\mathit{dyn}(Q)$) is obtained from Q by omitting all dynamic (static) atoms and their variables. We say that Q is without self-joins if every relation symbol appears in at most one atom. We visualise queries as hypergraphs where nodes are query variables (free variables are underlined), solid red hyperedges represent dynamic atoms, and dotted blue hyperedges represent static atoms.

Example 7. *Consider the query $Q_2(A,C,D) = R^d(A,D), S^s(A,B), T^s(B,C), U^d(D)$ and its hypergraph from Figure 2. Its static and dynamic sub-queries are $Q^s(A,C) = S^s(A,B), T^s(B,C)$ and respectively $Q^d(A,D) = R^d(A,D), U^d(D)$.*

A CQ is *(α)-acyclic* if we can construct a tree, called join tree, such that: (1) the nodes of the tree are the atoms of the query, and (2) for each variable, it holds: if the variable appears in two atoms, then it appears in all atoms on the unique path between the two atoms in the tree [8]. A CQ is called *free-connex acyclic* if it is acyclic and stays acyclic after adding a new atom whose schema consists of the free variables [7].

A CQ is *hierarchical* if for any two variables X and Y, it holds $\mathit{atoms}(X) \subseteq \mathit{atoms}(Y)$, $\mathit{atoms}(Y) \subseteq \mathit{atoms}(X)$, or $\mathit{atoms}(X) \cap \mathit{atoms}(Y) = \emptyset$ [27]. A hierarchical query is q-hierarchical if for any two variables X and Y with $\mathit{atoms}(X) \supset \mathit{atoms}(Y)$, it holds: if Y is free, then X is free [4].

A *path* in a query Q is a sequence $X_1, \ldots, X_n$ of non-repeating variables from Q such that each two adjacent variables X_i and X_{i+1} are contained in an atom from Q, $\forall i \in [n-1]$. We sometimes see a path as the set of its variables. Two variables X_1 and X_n are connected if there is a path $X_1, \ldots, X_n$. An atom $R(\boldsymbol{Y})$ and a variable X_n are connected if there is a path $X_1, \ldots, X_n$ for some variable $X_1 \in \boldsymbol{Y}$. Two atoms, $R(\boldsymbol{Y})$ and $S(\boldsymbol{Z})$, are connected if there are $X_1 \in \boldsymbol{Y}$ and $X_n \in \boldsymbol{Z}$ such that X_1 and X_n are connected. A set $\mathcal{C}$ of atoms in Q is called a *connected component* of Q if any two atoms in $\mathcal{C}$ are connected and this does not hold for any superset of $\mathcal{C}$.

Example 8. *The query $Q_1(A,B,C) = R^d(A,D), S^d(A,B), T^s(B,C)$ has the path D, A, B, C that connects (i) the variables D and C and (ii) the atoms $R^d(A,D)$ and $T^s(B,C)$.*

Dynamic Query Evaluation The problem of dynamic query evaluation is as follows: In a database containing both static and dynamic relations, when presented with a query, our goal is to maintain the query result under updates to the dynamic relations and to allow for the enumeration of tuples in the query result following an update.

A *single-tuple update* to a relation R is an insert or a delete of a tuple to R. We denote an insert of $\boldsymbol{t}$ by $+\boldsymbol{t}$ and its delete by $-\boldsymbol{t}$. In this paper, we consider set semantics: A tuple is either in or not in the database; the results of this paper can be generalised to bag semantics, or $\mathbb{Z}$-sets, where tuples are associated with (positive or negative) multiplicities.

Following prior work, e.g., [4, 15, 18], we decompose the time complexity of dynamic query evaluation into *preprocessing time*, *update time*, and *enumeration delay*. The preprocessing time is the time to compute a data structure that represents the query result before receiving any update. The update time is the time to update the data structure under an insert or delete of a single tuple to the database. The enumeration delay is the maximum of three times: the time between the start of the enumeration process and outputting the first tuple, the time between outputting any two consecutive tuples, and the time between outputting the last tuple and the end of the enumeration process [11].

Computation Model and Data Complexity We use the RAM model of computation where database schemas and data values are of constant size. To address the elements in a set of N values, where N is the input size, we need an index of constant size. Looking up the value in the set at a given index takes constant time. We further assume that each relation R over schema $\boldsymbol{X}$ is implemented by a data structure of size $O(|R|)$ that can: (1) look up, insert, and delete entries in constant time, and (2) enumerate all stored entries in R with constant delay. For a schema $\boldsymbol{S} \subset \boldsymbol{X}$, we use an index data structure that for any $\mathbf{t} \in \mathsf{Dom}(\boldsymbol{S})$ can: (3) enumerate all tuples in $\sigma_{\boldsymbol{S}=\mathbf{t}}R$ with constant delay, and (4) insert and delete index entries in constant time. All aforementioned lookup times, update times, and enumeration delays are amortised.

We report time complexities as functions of the database size N at the update time and using data complexity (the query is fixed and has constant size). Constant update time and constant delay therefore mean that they do not depend on the database size.

Due to space constraints, Appendix A introduces the fractional edge cover number $\rho^*_Q(\boldsymbol{S})$ of a variable set $\boldsymbol{S}$, the Online Matrix-Vector Multiplication (OMv), Online Vector-Matrix-Vector Multiplication (OuMv), and the Boolean Matrix-Multiplication (BMM) conjectures.

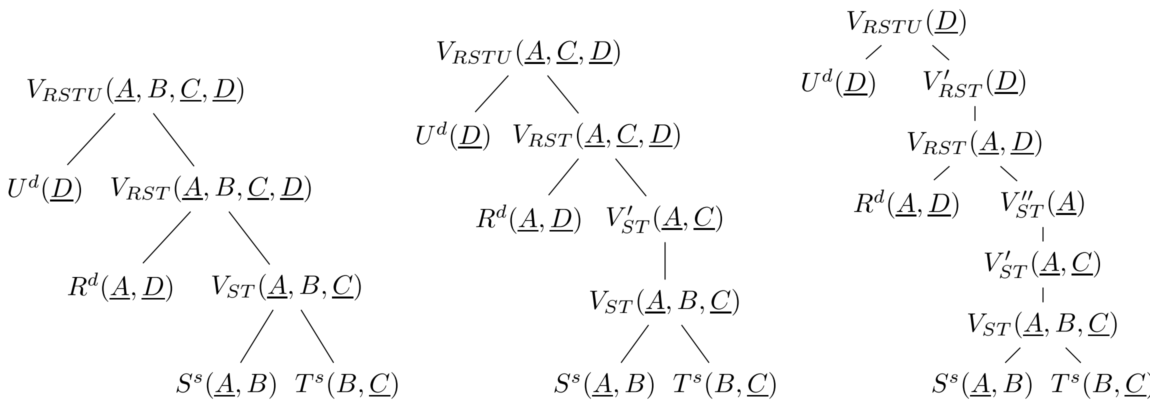


Figure 3: Three rewritings of the query $Q_2(A, C, D) = R^d(A, D), S^s(A, B), T^s(B, C), U^d(D)$ from Figure 2. The first two rewritings are not safe, while the last one is safe.

3 Safe Query Rewriting

A rewriting of a query is a project-join plan for the query. In the context of dynamic query evaluation, query rewritings have been previously used under the term *view trees* due to their natural tree-shaped graphical depiction [19, 18]. In this paper, we introduce a special type of query rewritings called *safe*, which ensures tractable evaluation.

A *rewriting* of a CQ Q using views (rewriting of Q in short) is a forest $\mathcal{T} = \{T_i\}_{i\in[n]}$ of trees T_i where the leaves are the atoms of Q and each inner node is a view V such that:

- If V has a single child, then V results from its child by projecting away some variables; we call V a *projection view*.
- If V has several children, then V is the join of its children; we call V a *join view*.

A view V is *dynamic* if the subtree rooted at V contains a dynamic atom. For convenience, we also refer to the atoms in a rewriting as views.

Example 9. *Figure 3 gives three (out of many possible) rewritings of the query $Q_2(A, C, D) = R^d(A, D)$, $S^s(A, B)$, $T^s(B, C), U^d(D)$ from Figure 2. Each rewriting is depicted as a tree. In all rewritings, the view V_{ST} is static, while the view V_{RST} is dynamic since it contains the dynamic relation R^d in its subtree.*

Next, we define *safe* query rewritings, which have four properties. The two *correctness* properties ensure that the views correctly encode the query result as a factorised representation [25]. The *update* property guarantees that the dynamic views can be maintained in constant time under single-tuple updates to any dynamic relation. The *enumeration* property ensures that the tuples in the query result can be listed from the views with constant delay.

Definition 10 (Safe Query Rewriting)**.** *A rewriting $\mathcal{T} = \{T_i\}_{i\in[n]}$ is safe for a CQ Q if:*

Correctness *(1) For each connected component $\mathcal{C}$ of Q, there is a tree T_i that contains all atoms in $\mathcal{C}$. (2) For any projection view $V'(\boldsymbol{Y})$ with child view $V(\boldsymbol{X})$, it holds that each atom of Q containing a variable from $\boldsymbol{X} \setminus \boldsymbol{Y}$ is contained only in the subtree rooted at V.*

Update *The schema of each dynamic view covers the schema of each of its sibling views.*

Enumeration *Each tree T_i has a subtree T'_i containing the root of T_i such that:*

$$\bigcup_{i\in[n]} vars(T'_i) = free(Q),$$

where $vars(T'_i)$ is the set of variables of the views in the subtree T'_i.

The computation time for a query rewriting is the time to materialise all its views.

Proposition 11. *Let a CQ Q and a database of size N. If Q has a safe rewriting with $f(N)$ computation time for some function f, then Q can be evaluated with $f(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.*

Example 12. *The first rewriting in Figure 3 is not safe: It violates the enumeration property because the root view V_{RSTU} contains the bound variable B, thus there is no guarantee of reporting distinct (A, C, D)-values with constant delay, as per our computational model. One possibility is to project away B before starting the enumeration but this requires time linear in the size of V_{RSTU}. The rewriting also violates the update property: for instance, the schema of the dynamic view $R^d(A, D)$ does not cover the schema of its sibling $V_{ST}(A, B, C)$, thus computing the change in V_{RST} for an update to R^d requires iterating over linearly many (B, C)-values from V_{ST} that are paired with the A-value from the update.*

The second rewriting is also not safe: It satisfies the enumeration property as the root view encodes the query result but fails on the update property for both dynamic atoms.

The third rewriting is safe and admits $\mathcal{O}(1)$ update time and $\mathcal{O}(1)$ enumeration delay, per Proposition 11. We next show how to handle updates and enumerate from this rewriting.

We can propagate an update to R^d or U^d to their ancestor views in constant time. Consider an insert of a new tuple (a, d) to relation R^d. To compute the change in V_{RST}, we check if a exists in V''_{ST} via a constant-time lookup. If not, we stop as no further propagation is needed; otherwise, we insert (a, d) into V_{RST} in constant time. We maintain V'_{RST} by inserting d into that view. We compute the change in the root V_{RSTU} by checking if d exists in U^d via a constant-time lookup, and if so, we insert d into the root. Propagating an insert to U^d requires a lookup in V'_{RST} and a insert into V_{RSTU}, both in constant time. Deletes to R^d and U^d are handled analogously. Thus, updates in this rewriting take constant time.

We can enumerate the distinct tuples in the query result using three nested for-loops. The first loop iterates over the D-values in V_{RSTU}; the second loop iterates over the A-values in V_{RST} paired with a D-value from the first loop; and the third loop iterates over the C-values in V'_{ST} paired with an A-value from the second loop. Hence, each distinct output tuple can be constructed in constant time.

The time to compute the view V_{ST} is quadratic because in the worst case each A-value in S^s is paired with each C-value in T^s. All other views in the rewriting are either projection views or semi-joins of one child view with another child view. Thus, the overall computation time for the rewriting is $\mathcal{O}(N^2)$, where N is the database size.

4 New Query Classes

In this section we introduce the query classes $\mathcal{C}_{\mathtt{lin}}$, $\mathcal{C}_{\mathtt{poly}}$, and $\mathcal{C}_{\mathtt{exp}}$. We first define two syntactic properties that underlie the classes $\mathcal{C}_{\mathtt{lin}}$ and $\mathcal{C}_{\mathtt{poly}}$ and ensure that any query in these two classes has a safe rewriting using views. The class $\mathcal{C}_{\mathtt{exp}}$ contains queries that do not satisfy these properties.

A query has *safe atom-to-atom paths* if any path $\boldsymbol{P}$ connecting two dynamic atoms goes through a variable that is common to the two atoms:

Definition 13 (Safe Atom-to-Atom Paths)**.** *A query has safe atom-to-atom paths if for every path $\boldsymbol{P}$ connecting two dynamic atoms $R(\boldsymbol{X})$ and $S(\boldsymbol{Y})$, it holds $\boldsymbol{X} \cap \boldsymbol{Y} \cap \boldsymbol{P} \neq \emptyset$.*

Example 14. *The queries $Q_3(A, B, C) = R^d(A), S^s(A, B), T^d(B)$ and $Q_4(A, B, C) = R^d(A, B), S^d(A, C), T^s(B, C)$ from Figure 2 do not have safe atom-to-atom paths. The path A, B connects the two dynamic atoms $R^d(A)$ and $T^d(B)$ in Q_3, which do not share any variable. The path B, C connects the two dynamic atoms $R^d(A, B)$ and $S^d(A, C)$ in Q_4 but does not go through their only common variable A.*

A query has *safe atom-to-variable paths* if any path $\boldsymbol{P}$ connecting a dynamic atom with a free variable goes through a free variable of the atom:

Definition 15 (Safe Atom-to-Variable Paths)**.** *A query Q has safe atom-to-variable paths if for every path $\boldsymbol{P}$ connecting a dynamic atom $R(\boldsymbol{X})$ with a free variable, it holds $\boldsymbol{X} \cap free(Q) \cap \boldsymbol{P} \neq \emptyset$.*

Example 16. *The query $Q_5(B,C) = R^d(A,B), S^d(A,C), T^s(B,C)$ from Figure 2 does not have safe atom-to-variable paths since the path A, B connects the dynamic atom $S^d(A,C)$ with the free variable B but does not go through C, which is the free variable of $S^d(A,C)$.*

A query has safe paths if it has safe atom-to-atom paths and safe atom-to-variable paths. We can check efficiently whether a query has safe paths:

Proposition 17. *For any CQ Q with n variables and m atoms, it can be decided in $O(n^2 \cdot m^2)$ time whether Q has safe paths.*

Safe paths ensure that the dynamic sub-query is q-hierarchical:

Proposition 18.

- *Any CQ without static relations is q-hierarchical if and only if it has safe paths.*
- *The dynamic sub-query of any CQ with safe paths is q-hierarchical.*

In Section 5, we show that every query with safe paths admits a safe rewriting, that is, it can be rewritten into a forest of views that support constant update time and constant enumeration delay. To obtain linear preprocessing time, we further require that the query is free-connex α-acyclic.

Definition 19 ($\mathcal{C}_{\texttt{lin}}$)**.** *A CQ is in $\mathcal{C}_{\texttt{lin}}$ if it is free-connex α-acyclic and has safe paths.*

For the class $\mathcal{C}_{\texttt{poly}}$, we skip the condition that the query is free-connex acyclic.

Definition 20 ($\mathcal{C}_{\texttt{poly}}$)**.** *A CQ is in $\mathcal{C}_{\texttt{poly}}$ if it has safe paths.*

Example 21. *Consider the queries from Figure 2. The query Q_1 is in $\mathcal{C}_{\texttt{lin}}$ since it is free-connex α-acyclic and has safe paths. The query Q_2 has safe paths but is not free-connex since adding its head atom $Q_2(A,C,D)$ to its body yields a cyclic query. Hence, Q_2 is in $\mathcal{C}_{\texttt{poly}}$. The queries Q_3-Q_6 are not in $\mathcal{C}_{\texttt{poly}}$ since they have no safe paths.*

The class $\mathcal{C}_{\texttt{exp}}$ requires that dynamic atoms are covered by static atoms in case the safe path property is violated:

Definition 22. *A CQ is in $\mathcal{C}_{\texttt{exp}}$ if it has safe paths or every variable contained in a dynamic atom is also contained in a static atom.*

Example 23. *The query Q_3 does not have safe atom-to-atom paths but is in $\mathcal{C}_{\texttt{exp}}$ since the two dynamic atoms are covered by a static atom. The queries Q_4-Q_6 are not contained in $\mathcal{C}_{\texttt{exp}}$ because they have variables in dynamic atoms that are not covered by static atoms.*

5 Evaluation of Queries in $\mathcal{C}_{\texttt{lin}}$ and $\mathcal{C}_{\texttt{poly}}$

We describe our evaluation strategy for queries in the class $\mathcal{C}_{\texttt{poly}}$, which includes the class $\mathcal{C}_{\texttt{lin}}$. In Section 5.1, we introduce variable orders, which guide the construction of safe rewritings for such queries. We also define the preprocessing width of a query based on its variable orders. We show that for queries in $\mathcal{C}_{\texttt{lin}}$, the preprocessing width is 1. In Section 5.2, we show how we can construct a safe rewriting for queries in $\mathcal{C}_{\texttt{poly}}$ following variable orders in time that is given by their preprocessing width. This proves the complexity upper bounds in Theorems 1 and 3.

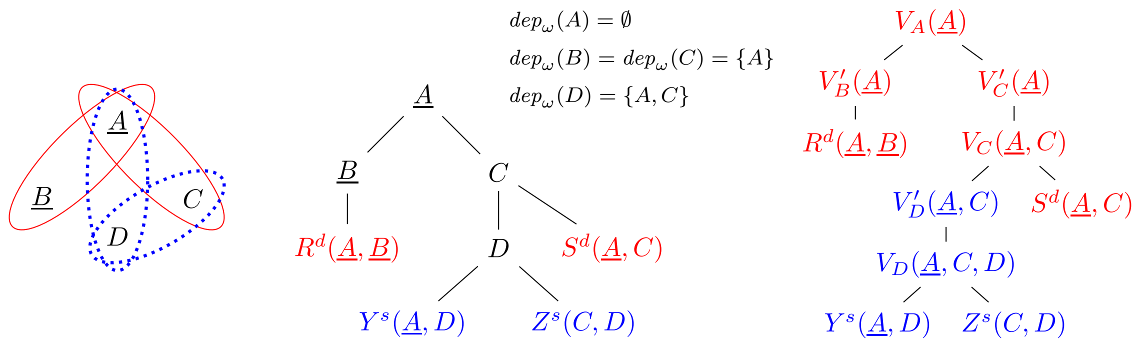


Figure 4: (left to right) The hypergraph of $Q(A, B) = R^d(A, B), S^d(A, C), Y^s(A, D), Z^s(C, D)$, a VO ω for Q, and the view tree for Q constructed by the procedure $\textsc{Rewrite}(\omega)$ from Figure 8. Dynamic views are in red, static views are in blue. The query Q has safe paths and the preprocessing width of 2, thus $Q \in \mathcal{C}_{\texttt{poly}}$.

5.1 Variable Orders

We say that two variables in a query are *dependent* if they appear in the same atom. A *variable order* (VO) for a query Q in $\mathcal{C}_{\texttt{poly}}$ is a forest $\omega = \{\omega_i\}_{i \in [n]}$ of trees ω_i such that: (1) The nodes of ω are the variables in Q; (2) the variables of each atom in Q lie on a root-to-leaf path in ω [26, 25]. We denote by $vars(\omega)$ the set of variables in ω and by ω_X the subtree of ω rooted at X. We associate each VO ω with a dependency function dep_ω that maps each variable X in ω to the subset of its ancestors on which the variables in ω_X depend on: $dep_\omega(X) = \{Y | Y \text{ is an ancestor of } X \text{ and } \exists Z \in \omega_X \text{ s.t. } Y \text{ depends on } Z\}$.

An *extended* VO results from a VO ω by adding the atoms in Q as new leaves: Each atom becomes the child of the lowest variable in ω that is contained in the schema of the atom. A VO is *canonical* if the variables of each atom are the inner nodes of a root-to-leaf path in the extended VO. It is called free-top if no bound variable is on top of a free variable. Every q-hierarchical query admits a free-top canonical VO [19]. We say that the *dynamic part of a VO* $\omega = \{\omega_i\}_{i \in [n]}$ *is canonical* if each tree ω_i has a connected subtree ω'_i including the root such that $\{\omega'_i\}_{i \in [n]}$ is a canonical VO for $dyn(Q)$.

Example 24. *Consider the query Q from Figure 4. The query has safe paths but is not free-connex α-acyclic, thus $Q \in \mathcal{C}_{\texttt{poly}}$. The dynamic subquery $dyn(Q)$ is q-hierarchical, per Proposition 18. The figure shows an extended free-top VO for the query where the dynamic part consisting of the variables A, B, and C is canonical.*

We can generalise the above example to all queries in $\mathcal{C}_{\texttt{poly}}$:

Proposition 25. *Any query in $\mathcal{C}_{\texttt{poly}}$ has a free-top VO whose dynamic part is canonical.*

In the following, we use VO to refer to an extended free-top VO whose dynamic part is q-hierarchical. Next, we define the *preprocessing width* w of VOs and queries in $\mathcal{C}_{\texttt{poly}}$. For a variable X in a VO, Q_X denotes the query that joins all atoms in the subtree of the VO rooted at X:

$$\mathsf{w}(\omega) = \max_{X \in vars(\omega)} \rho^*_{Q_X}(\{X\} \cup dep_\omega(X))$$

$$\mathsf{w}(Q) = \min_{\omega \in \mathsf{VO}(Q)} \mathsf{w}(\omega)$$

The preprocessing width of any query in $\mathcal{C}_{\texttt{lin}}$ is 1:

Proposition 26. *For any query Q in $\mathcal{C}_{\texttt{lin}}$, it holds $\mathsf{w}(Q) = 1$.*

Example 27. *Figure 4 shows the dependency set for each variable in the VO ω. We have $dep_\omega(D) = \{A, C\}$ and $\rho^*_{Q_D}(\{A, C, D\}) = 2$ since we need two atoms to cover these variables. For all other sets, the fractional edge cover number is 1. Hence, the preprocessing width of the overall query is 2.*

Rewrite(VO ω) : rewriting using views	
switch ω:	
$\{\omega_i\}_{i\in[n]}$	1 **return** $\{$ Rewrite(ω_i) $\}_{i\in[n]}$
$R(\boldsymbol{Y})$	2 **return** $R(\boldsymbol{Y})$
$\begin{array}{c} X \\ / \quad \backslash \\ \omega_1 \cdots \omega_k \end{array}$	3 **if** $k = 1$ and ω_1 is atom $R(\boldsymbol{Y})$ **return** $R(\boldsymbol{Y})$ 4 **let** $T_i =$ Rewrite(ω_i) with root view $V_i(\boldsymbol{S}_i)$, $\forall i \in [k]$ 5 **let** $\boldsymbol{S} = \{X\} \cup dep_\omega(X)$ 6 **let** $V_X(\boldsymbol{S}) =$ join of $V_1(\boldsymbol{S}_1), \ldots, V_k(\boldsymbol{S}_k)$, and $V'_X(\boldsymbol{S} \setminus \{X\}) = V_X(\boldsymbol{S})$ 7 **return** $\begin{cases} \begin{array}{c} V_X(\boldsymbol{S}) \\ / \quad \backslash \\ T_1 \cdots T_k \end{array} & \text{if } dep_\omega(X) = \emptyset \\ \begin{array}{c} V'_X(\boldsymbol{S} \setminus \{X\}) \\ \mid \\ V_X(\boldsymbol{S}) \\ / \quad \backslash \\ T_1 \cdots T_k \end{array} & \text{otherwise} \end{cases}$

Figure 5: Rewriting a query using views following its VO ω.

5.2 Safe Rewriting of Queries in $\mathcal{C}_{\texttt{poly}}$

We show that every query in $\mathcal{C}_{\texttt{poly}}$ has a safe rewriting using views. The time to compute the views is $\mathcal{O}(N^{\mathsf{w}})$, where N is the database size and w is the preprocessing width of the query.

Prior work uses view trees following variable orders to evaluate queries in the all-dynamic setting [17, 1, 18]. We adapt the view tree construction to the setting over static and dynamic relations and show that for queries in $\mathcal{C}_{\texttt{poly}}$, the resulting view trees are safe rewritings.

Given a VO ω for a query Q in $\mathcal{C}_{\texttt{poly}}$, the procedure Rewrite in Figure 5 rewrites Q into a forest of views following ω. It proceeds recursively on the structure of ω. If ω consists of a set of trees, it creates a view tree for each tree in ω (Line 1). If ω is a single tree with root X, it proceeds as follows. If X is an atom or its only child is an atom, it returns the atom (Lines 2-3). Otherwise, it creates a join view V_X that joins the created child views and puts on top a projection view V'_X that projects away X, unless X has no ancestor (Lines 5-7).

Example 28. *Figure 4 shows the view tree constructed by the procedure* Rewrite *following the VO ω depicted in the figure. Observe that we obtain the view tree from the VO by replacing each variable X either by a single projection view $V'_X(dep_\omega(X))$ or by a join view $V_X(\{X\} \cup dep_\omega(X))$ and a projection view $V'_X(dep_\omega(X))$ on top.*

We can verify that the view tree satisfies all four properties of safe rewritings as specified in Definition 10. The schema of each dynamic view in the view tree covers the schema of each of its siblings; for instance, the schema of the view $V'_B(A)$ covers the schema of the sibling view $V'_C(A)$. The views $V_A(A)$ and $R^d(A, B)$ encode the query result.

The time to compute the view $V_D(A, C, D)$ is quadratic in the database size. All other views compute semi-joins or project away a variable in linear time. Thus, the computation time for this rewriting is quadratic. From Proposition 11 follows that this query admits constant update time and constant enumeration delay after quadratic preprocessing time.

The procedure Rewrite outputs a safe rewriting. The computation time of the rewriting is given by the preprocessing width of Q.

Proposition 29. *For any query Q in $\mathcal{C}_{\mathtt{poly}}$, VO ω for Q, and database of size N, $\textsc{Rewrite}(\omega)$ is a safe rewriting for Q with $\mathcal{O}(N^{\mathsf{w}})$ computation time, where N is the database size and w is the preprocessing width of Q.*

6 Evaluation of Queries in $\mathcal{C}_{\mathtt{exp}}$

In this section, we first exemplify our evaluation strategy for one simple query in $\mathcal{C}_{\mathtt{exp}}$ and then describe it for arbitrary queries in $\mathcal{C}_{\mathtt{exp}}$.

Consider the query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ from Figure 2. It does not admit a safe rewriting, thus the evaluation strategy described in Section 5 cannot achieve constant update and constant enumeration delay. At a first glance, Q_3 does not seem tractable as a single-tuple update can lead to linearly many changes to the query result. For example, an insert $+a$ to R^d may be paired with linearly many B-values in S^s. Yet if we allow for more preprocessing time, we can resolve this difficulty by precomputing the effect of each possible update and store it in a transition system that allows us to efficiently fetch the updated query result after each single-tuple update.

A transition system is a tuple $(\mathcal{S}, s_{init}, \mathcal{U}, \delta)$, where $\mathcal{S}$ is a set of states, $s_{init} \in \mathcal{S}$ is the initial state, $\mathcal{U}$ is a set of single-tuple updates, and $\delta : \mathcal{S} \times \mathcal{U} \to \mathcal{S}$ is a function that maps a state and a single-tuple update to another state. In our context, each state corresponds to a database instance and the materialised query result over that database; since the static relations do not change and are the same across all states, we only record in the states the content of the dynamic relations and the query result. Each transition corresponds to a single-tuple update to a dynamic relation. In the preprocessing step, we build such a transition system. For each update, we use the transition system to move from the current state to another state. To enumerate the query result, we enumerate the tuples in the current state. We next show our evaluation strategy for query Q_3.

Example 30. *We create one state for each possible snapshot of the dynamic relations R^d and T^d, that is, for every combination of the possible A-values in R^d and B-values in T^d. We then build the transitions between these states that correspond to the insertions and deletions of A- and B-values. Even though the domain of R^d and T^d can be unbounded, the result of Q_3 is nevertheless guarded by the static relation S^s. This means that only the A- and B-values from S^s can appear in the query result. Therefore, we can restrict the domain of the variables in the dynamic relations R^d and T^d to the A-values and B-values from S^s.*

Let $\mathcal{A} = \pi_A S^s$ and $\mathcal{B} = \pi_B S^s$. For each pair $\alpha \subseteq \mathcal{A}$ and $\beta \subseteq \mathcal{B}$, we create a state (α, β) and the corresponding query result for $R^d = \alpha$ and $T^d = \beta$. That is, the state (α, β) has the result $Q_3(A, B) = \alpha(A), S^s(A, B), \beta(B)$. The transition system has the states $\mathcal{S} = \{(\alpha, \beta) \mid \alpha \subseteq \mathcal{A}, \beta \subseteq \mathcal{B}\}$. The initial state $s_{init} = (\alpha_0, \beta_0)$ is the state corresponding to the input database with $\alpha_0 = R^d \cap \mathcal{A}$ and $\beta_0 = T^d \cap \mathcal{B}$. Each transition corresponds to an insertion $+a$ (deletion $-a$) of an A-value $a \in \mathcal{A}$ to (from) R^d or the insertion $+b$ (deletion $-b$) of a B-value $b \in \mathcal{B}$ to (from) T^d. Consider a state $(\alpha, \beta) \in \mathcal{S}$. For each $a \in \mathcal{A} \setminus \alpha$, we create the transition $\delta((\alpha, \beta), +a) = (\alpha \cup \{a\}, \beta)$. For each $a \in \alpha$, we create the transition $\delta((\alpha, \beta), -a) = (\alpha \setminus \{a\}, \beta)$. We ignore transitions from a state to itself without loss of generality. We create similar transitions for inserting and deleting B-values.

Figure 6 shows an input database (left) and the transition system built for query Q_3 on the input database (right). The A- and B-values in S^s are $\mathcal{A} = \{a_1\}$ and $\mathcal{B} = \{b_1, b_2\}$, respectively. The transition system contains eight states (boxes), one for each combination of the subsets $\alpha \subseteq \mathcal{A}$ and $\beta \subseteq \mathcal{B}$. Each state (α, β) defines the result of Q_3 for $R^d = \alpha$ and $T^d = \beta$. Since the input dynamic relations are $R^d = \{a_1\}$ and $T^d = \{b_3\}$, the initial state has $\alpha = \{a_1\}$ and $\beta = \emptyset$, since $b_3 \notin \mathcal{B}$ and does not match any transition.

The input database contains $\mathcal{O}(N)$ A-value and B-values, so the transition system contains $2^{\mathcal{O}(N)}$ states. The query result at any state is a subset of S^s and can be computed in $\mathcal{O}(N)$ time. The number of transitions from any state is bounded by twice the number of distinct A-values and B-values, which is $\mathcal{O}(N)$. For each transition, we take $O(N)$ time to find the landing state (α, β) of size $O(N)$ among exponentially many such pairs. Overall, it takes $\mathcal{O}(N^2)$ time to build the index to allow transitions from each state. Once built, we

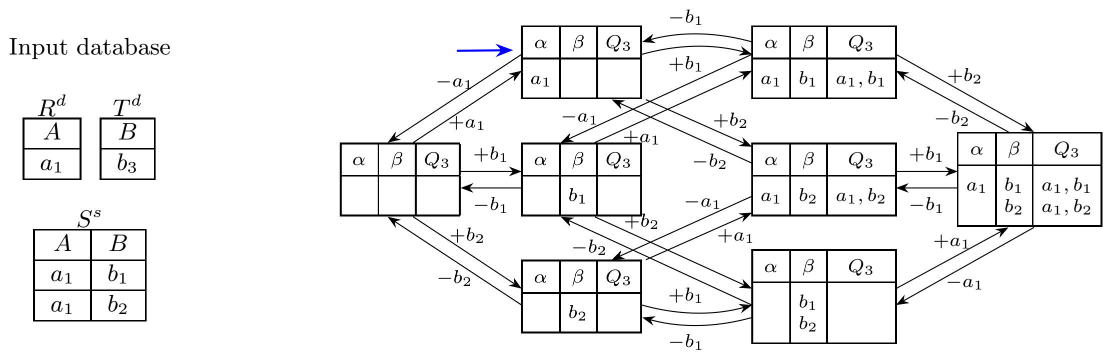


Figure 6: Left: Input database with two dynamic relations $R^d(A)$ and $T^d(B)$ and one static relation $S^s(A, B)$. Right: The transition system built from the input database and query Q_3. Each state (box) corresponds to a pair of subsets α and β of A-values and B-values in S^s (first two columns), and the corresponding materialized result of Q_3 if R^d and T^d contain the values in α and β (third column). Each transition (arrow) corresponds to an insertion $+a$ (deletion $-a$) of an A-value a to (from) R^d or to insertion $+b$ (deletion $-b$) of a B-value b to (from) T^d. The initial state has $\alpha = \{a_1\}$ and $\beta = \emptyset$ (blue incoming arrow).

can move between states in constant time. Overall, we can create each state and its transitions in $\mathcal{O}(N^2)$ time. The general case is detailed in Appendix E.

The preprocessing time is given by the number of states times the creation time per state, so $\mathcal{O}(2^N \cdot N^2)$. The update time is constant since for each single-tuple update to the dynamic relation, we find in constant time the corresponding transition and move to the target state or ignore the update if it does not match any transition. We enumerate the tuples in the current state trivially with constant delay, as the query result is already materialised.

We generalize the evaluation strategy from Example 30 to arbitrary queries in $\mathcal{C}_{\texttt{exp}}$. The transition system can be built in time $\mathcal{O}(2^p \cdot p^2)$, where $p = \mathcal{O}(N^{\rho^*(stat(Q))})$, as stated in Theorem 5. Finding the initial state takes time $\mathcal{O}(N)$.

Consider a query Q in $\mathcal{C}_{\texttt{exp}}$. Let k be the number of dynamic relations in Q. We create one state for each possible snapshot of the dynamic relations. Since Q is in $\mathcal{C}_{\texttt{exp}}$, every variable in a dynamic atom $R^d(\mathcal{X})$ also occurs in a static atom. Similar to Example 30, we restrict the possible values for $\mathcal{X}$ in R^d to the projection of the result of the static sub-query $stat(Q)$ of Q onto $\mathcal{X}$. Let us denote this projection by $\mathcal{A}_{R^d}$. Similarly, we create the sets $\mathcal{A}_{R_1^d}, \ldots, \mathcal{A}_{R_k^d}$ for all dynamic relations $R_1^d, \ldots, R_k^d$. The size of each such set is upper bounded by the size of the result of the static sub-query, which is $\mathcal{O}(N^{\rho^*(stat(Q))})$.

The transition system has the states $\mathcal{S} = \{(\alpha_1, \ldots, \alpha_k) \mid \alpha_1 \subseteq \mathcal{A}_{R_1^d}, \ldots, \alpha_k \subseteq \mathcal{A}_{R_k^d}\}$, i.e., we create a state for each combination of the subsets of $\mathcal{A}_{R_1^d}, \ldots, \mathcal{A}_{R_k^d}$. The overall number of states is $(2^{\mathcal{O}(N^{\rho^*(stat(Q))})})^k = 2^{\mathcal{O}(k \cdot N^{\rho^*(stat(Q))})}$.

Consider an arbitrary state $(\alpha_1, \ldots, \alpha_k) \in \mathcal{S}$. We materialise the corresponding query result, that is, the result of Q where $R_i^d = \alpha_i$, for $i \in [k]$. The size of the result is bounded by $\mathcal{O}(N^{\rho^*(stat(Q))})$, and thus can be computed in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time.

We next build the transitions for each state. Consider a dynamic relation R_i^d. For each $a \in \mathcal{A}_{R_i^d} \setminus \alpha_i$, we create a transition $\delta((\alpha_1, \ldots, \alpha_k), +a) = (\alpha_1, \ldots, \alpha_i \cup \{a\}, \ldots, \alpha_k)$ that moves to the state where a is included, and conversely, for each $a \in \alpha_i$, we create a transition $\delta((\alpha_1, \ldots, \alpha_k), -a) = (\alpha_1, \ldots, \alpha_i \setminus \{a\}, \ldots, \alpha_k)$ that moves to the state where a is removed. The state has $\mathcal{O}(N^{\rho^*(stat(Q))})$ transitions per dynamic relation. Appendix E explains how to build an index for these transitions in $\mathcal{O}(N^{2\rho^*(stat(Q))})$ time, which allows us to move from the current state to another state for an update in constant time. We build the transitions and compute the index for all k dynamic relations. Overall, we can create the state in $\mathcal{O}(k \cdot N^{2\rho^*(stat(Q))})$ time.

Building the transition system takes time proportional to the number of states times the time to build one state: $\mathcal{O}(2^{k\cdot N^{\rho^*(stat(Q))}} \cdot k \cdot N^{2\rho^*(stat(Q))})$ in total. We simplify it to $2^p \cdot p^2$, where $p = \mathcal{O}(N^{\rho^*(stat(Q))})$. This matches the complexity in Theorem 5.

7 Lower Bound for Queries Outside $\mathcal{C}_{\texttt{lin}}$

In this section, we outline the proof of the lower bound result in Theorem 1. It consists of two parts. In the first part, we give lower bounds on the evaluation complexity of the simple queries $Q_{RST}() = R^d(A), S^s(A,B), T^d(B)$ and $Q_{ST}(A) = S^s(A,B), T^d(B)$. None of these queries is contained in $\mathcal{C}_{\texttt{lin}}$: the first one does not have safe atom-to-atom paths and the second one does not have safe atom-to-variable paths. In the second part, the argument is as follows. Consider a query $Q \notin \mathcal{C}_{\texttt{lin}}$ that has no self-joins. By definition of $\mathcal{C}_{\texttt{lin}}$, this means that (1) Q is not free-connex acyclic, or (2) it does not have safe atom-to-atom paths, or (3) it does not have safe atom-to-variable paths. In case Q is not free-connex acyclic, we cannot achieve constant-delay enumeration of the result after linear preprocessing time (even without processing any update), unless the BMM conjecture fails [3]. If Q does not have atom-to-atom or atom-to-variable paths, we reduce the evaluation of Q_{RST} or respectively Q_{ST} to the evaluation of Q, which transfers the lower bound for these queries to Q. The latter reduction is standard (see, e.g., [4]). In the following, we outline the lower bound proofs for Q_{RST} and Q_{ST} and defer the details to Appendix F.

The lower bound for Q_{RST} is conditioned on the OuMv conjecture, which is implied by the OMv conjecture [12]:

Proposition 31. *The CQ $Q_{RST}() = R^d(A), S^s(A,B), T^d(B)$ cannot be evaluated with $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, $\mathcal{O}(N^{1/2-\gamma})$ update time, and $\mathcal{O}(N^{1/2-\gamma})$ enumeration delay for any $\gamma > 0$, where N is the database size, unless the OuMv conjecture fails.*

Proof Sketch. Prior work reduces the OuMv problem to the evaluation of the variant of Q_{RST} where all relations are dynamic [4]. The reduction starts with an empty database and encodes the matrix of the OuMv problem into the relation S using updates. In our case, it is not possible to do this encoding using updates, since the relation is static. Instead, we do the encoding before the preprocessing stage of the evaluation algorithm for Q_{RST}.

We explain the reduction in more detail. Assume that there is an algorithm A that evaluates Q_{RST} with $\mathcal{O}(N^{1/2-\gamma})$ update time and enumeration delay after $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, for some $\gamma > 0$. Consider an $n \times n$ matrix M and n pairs (u_r, v_r) of n-dimensional vectors that serve as input to the OuMv problem. We first encode M into the relation S in time $\mathcal{O}(n^2)$, which leads to a database of size $\mathcal{O}(n^2)$. Then, in each round $r \in [n]$, we encode the vectors u_r and v_r into R and respectively T using $\mathcal{O}(n)$ updates and trigger the enumeration procedure of A to obtain from Q_{RST} the result of $u_r M v_r$. This takes $\mathcal{O}(n(n^2)^{1/2-\gamma}) = \mathcal{O}(n^{2-2\gamma})$ time. After n rounds, we use overall $\mathcal{O}(n^{3-2\gamma})$ time. This means that we solve the OuMv problem in sub-cubic time, which contradicts the conjecture. □

The reduction of the OMv problem to the evaluation of the query $Q_{ST}(A) = S^s(A,B)$, $T^d(B)$ is similar to the above reduction. We encode the matrix M into the relation S before the preprocessing stage and encode each incoming vector v_r into T using updates.

8 Outlook: Tractability Beyond $\mathcal{C}_{\texttt{exp}}$

This work explores the tractability of conjunctive queries over static and dynamic relations. The largest class of tractable queries put forward is $\mathcal{C}_{\texttt{exp}}$. Yet a characterisation of *all* tractable queries remains open. In the following, we discuss the evaluation of some queries outside the class $\mathcal{C}_{\texttt{exp}}$.

Let a query Q. The *reduced dynamic sub-query* of Q is Q without its static atoms and the variables in these static atoms. An immediate observation is that queries whose reduced dynamic sub-query is not q-hierarchical are not tractable. This is implied by the intractability of non-q-hierarchical queries in the all-dynamic setting [4]. One example query whose reduced sub-query is not q-hierarchical is the query $Q_6(A,B)$

$= R^d(A),\ S^d(A,B),\ T^d(B,C),\ U^s(C)$ from Figure 2. A question is whether the queries, whose reduced dynamic sub-queries are q-hierarchical, are tractable. We discuss next two such queries that are not in $\mathcal{C}_{\mathtt{exp}}$: $Q_4(A,B,C) = R^d(A,B),\ S^d(A,C),\ T^s(B,C)$ and $Q_5(B,C) = R^d(A,B),\ S^d(A,C),\ T^s(B,C)$ from Figure 2.

The evaluation strategy illustrated in Example 30 can be easily extended to Q_4. We create the same transition system as in Example 30 and assign each A-value a that is common to R^d and S^d to the state that stores the (B,C)-values paired with a in the result. Any update to R^d or S^d changes the assignment of at most one A-value. At any time, we can enumerate the query result by iterating over the A-values and enumerating for each of them the tuples in their corresponding state.

The query $Q_5(B,C)$ differs from Q_4 in that the variable A is bound. The above approach for Q_4 does not allow for constant-delay enumeration of the result of Q_5 since distinct A-values might be assigned to distinct states that share (B,C)-tuples. Filtering out duplicates can however incur a non-constant enumeration delay.

References

[1] A. Abboud and V. V. Williams. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*, pages 434–443, 2014.

[2] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[3] G. Bagan, A. Durand, and E. Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*, pages 208–222, 2007.

[4] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017.

[5] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering FO+MOD Queries under Updates on Bounded Degree Databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018.

[6] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *ICDT*, pages 8:1–8:19, 2018.

[7] J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

[8] J. Brault-Baron. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.*, 49(3):54:1–54:26, 2016.

[9] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.*, 16(7):1601–1614, 2023.

[10] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018.

[11] A. Durand and E. Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic*, 8(4):21, 2007.

[12] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015.

[13] M. Idris, M. Ugarte, and S. Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017.

[14] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT*, pages 4:1–4:18, 2019.

[15] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Maintaining Triangle Queries under Updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020.

[16] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. In *PODS*, pages 375–392, 2020.

[17] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Conjunctive Queries with Free Access Patterns Under Updates. In *ICDT*, volume 255 of *LIPIcs*, pages 17:1–17:20, 2023.

[18] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. F-IVM: Analytics over Relational Databases under Updates. *The VLDB Journal*, 2023.

[19] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *Log. Methods Comput. Sci.*, 19(3), 2023.

[20] M. A. Khamis, A. Kara, D. Olteanu, and D. Suciu. Insert-Only versus Insert-Delete in Dynamic Query Evaluation. *CoRR*, abs/2312.09331, 2023.

[21] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *The VLDB Journal*, 23(2):253–278, 2014.

[22] D. Marx. Approximating Fractional Hypertree Width. *ACM Trans. Algorithms*, 6(2):29:1–29:17, 2010.

[23] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[24] H. Q. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[25] D. Olteanu and M. Schleich. Factorized Databases. *SIGMOD Rec.*, 45(2):5–16, 2016.

[26] D. Olteanu and J. Závodný. Size Bounds for Factorised Representations of Query Results. *ACM TODS*, 40(1):2:1–2:44, 2015.

[27] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[28] Y. Tao and K. Yi. Intersection Joins under Updates. *J. Comput. Syst. Sci.*, 124:41–64, 2022.

[29] Q. Wang, X. Hu, B. Dai, and K. Yi. Change Propagation Without Joins. *Proc. VLDB Endow.*, 16(5):1046–1058, 2023.

A Further Preliminaries

We recall some notions that were omitted in Section 2. We start with well-known width measure for queries.

Definition 32 (Fractional Edge Cover [2]). *Given a CQ Q and $\mathcal{F} \subseteq vars(Q)$, a* fractional edge cover *of $\mathcal{F}$ is a solution $\boldsymbol{\lambda} = (\lambda_{R(\mathcal{X})})_{R(\mathcal{X}) \in atoms(Q)}$ to the following linear program:*

$$\begin{array}{lll}
\textit{minimize} & \displaystyle\sum_{R(\mathcal{X}) \in\, atoms(Q)} \lambda_{R(\mathcal{X})} & \\
\textit{subject to} & \displaystyle\sum_{R(\mathcal{X}) \in\, atoms(Q)\ \ s.t.\ \ X \in \mathcal{X}} \lambda_{R(\mathcal{X})} \geq 1 & \textit{for all } X \in \mathcal{F} \textit{ and} \\
& \lambda_{R(\mathcal{X})} \in [0, 1] & \textit{for all } R(\mathcal{X}) \in atoms(Q)
\end{array}$$

The optimal objective value of the above program is called the fractional edge cover number *of the variable set $\mathcal{F}$ and is denoted as $\rho^*_Q(\mathcal{F})$. If Q is clear from the context, we omit the index Q and write $\rho^*(\mathcal{F})$. When all Q's variables are considered, namely $\mathcal{F} = vars(Q)$, we write $\rho^*(Q)$. For a database of size N, the result of a query Q without bound variables can be computed in time $\mathcal{O}(N^{\rho^*(Q)})$ [23].*

Definition 33 (Tree Decomposition)**.** *A tree decomposition of a CQ Q is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree with vertices $V(\mathcal{T})$ and $\chi : V(\mathcal{T}) \to 2^{vars(Q)}$ maps each node t of the tree $\mathcal{T}$ to a subset $\chi(t)$ of variables of Q such that the following properties hold:*

1. *for every atom $R(X) \in atoms(Q)$, the schema X is a subset of $\chi(t)$ for some $t \in V(\mathcal{T})$,*
2. *for every variable $X \in vars(Q)$, the set $\{t | X \in \chi(t)\}$ is a non-empty connected subtree of $\mathcal{T}$. The sets $\chi(t)$ are called the bags of the tree decomposition.*

A tree decomposition is free-connex if it has a connected sub-tree such that the union of the bags of this sub-tree are the free variables. We use $TD(Q)$ to denote the set of free-connex tree decompositions of Q.

Definition 34 (Fractional Hypertree Width)**.** *Given a CQ Q and a free-connex tree decomposition $(\mathcal{T}, \chi)$ of Q, the fractional hypertree width of $(\mathcal{T}, \chi)$ and of Q are defined as follows:*

$$w(\mathcal{T}, \chi) = \max_{t \in V(\mathcal{T})} \rho^*_Q(\chi(t))$$
$$w(Q) = \min_{(\mathcal{T}, \chi) \in TD(Q)} w(\mathcal{T}, \chi)$$

Next, we overview the widely believed complexity-theoretic conjectures.

Definition 35 (Boolean Matrix Multiplication (BMM) Problem [3])**.** *Given two $n \times n$ Boolean matrices A and B, represented as lists of their non-zero entries, the task is to output the product AB.*

Conjecture 36 (BMM conjecture [3])**.** *The BMM problem cannot be solved in time $m^{1+o(1)}$, where m is the number of non-zero entries in A, B and AB.*

Definition 37 (Online Matrix-Vector Multiplication (OMv) Problem [12])**.** *We are given an $n \times n$ Boolean matrix M and receive n Boolean column vectors $v_1, \ldots, v_n$ of size n, one by one. After seeing each vector v_i, the task is to output the product Mv_i before seeing the next vector.*

Conjecture 38 (OMv Conjecture [12])**.** *For any constant $\gamma > 0$, there is no algorithm that solves the OMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

Definition 39 (Online Vector-Matrix-Vector Multiplication (OuMv) Problem [12])**.** *We are given an $n \times n$ Boolean matrix M and receive n pairs of Boolean column vectors $(u_1, v_1), \ldots, (u_n, v_n)$ of size n, one by one. After seeing each pair of vectors (u_i, v_i), the task is to output the product $u_i M v_i$ before seeing the next pair.*

The following OuMv conjecture is implied by the OMv conjecture.

Conjecture 40 (OuMv Conjecture [12])**.** *For any constant $\gamma > 0$, there is no algorithm that solves OuMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

B Missing Details in Section 3

B.1 Proof of Proposition 11

Proposition 11 *Let a CQ Q and a database of size N. If Q has a safe rewriting with $f(N)$ computation time for some function f, then Q can be evaluated with $f(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.*

Let $\mathcal{T}$ be a safe rewriting for Q. The preprocessing time is the computation time of $\mathcal{T}$.

To enumerate the result of Q, we nest the enumeration procedures for the connected components of Q, concatenating their output tuples. For each connected component C, the enumeration procedure traverses the tree $T \in \mathcal{T}$ containing all atoms from C in a top-down manner. The enumeration property of $\mathcal{T}$ guarantees that there exists a subtree T' of T having the same root as T that covers exactly the free variables of C.

We enumerate the result of C by traversing the subtree T' in preorder. At each view $V(\boldsymbol{X})$, we fix the values of variables in $\boldsymbol{X} \setminus \boldsymbol{Y}$, where $\boldsymbol{Y}$ is the set of variables of the ancestor views of V. We retrieve in constant time a tuple of values over $\boldsymbol{X} \setminus \boldsymbol{Y}$ from V for the given $\boldsymbol{Y}$-value. After visiting all views once, we construct the first complete output tuple for C and report it. We continue iterating over the remaining distinct values over $\boldsymbol{X} \setminus \boldsymbol{Y}$ in the last visited view V, reporting new tuples with constant delay. After exhausting all values from V, we backtrack and repeat the enumeration procedure for the next $\boldsymbol{Y}$-value. The enumeration stops once all views from the subtree T' are exhausted. Given that all views are calibrated bottom-up but the enumeration proceeds top-down, the procedure only visits those tuples that appear in the output, thus ensuring constant enumeration delay.

We propagate a constant-sized update through a tree in a bottom-up manner, maintaining each view on the path from the affected relation to the root. From the update property of the safe rewriting $\mathcal{T}$, computing the delta of any join view involves only constant-time lookups in the sibling views of the child carrying the update. The size of the delta also remains constant. Computing the delta of a projection view also requires a constant-time projection of its incoming update. Since an update to one relation affects one tree of $\mathcal{T}$, propagating an update through $\mathcal{T}$ takes constant time.

C Missing Details in Section 4

C.1 Proof of Proposition 17

Proposition 17. *For any CQ with n variables and m atoms, it can be decided in $\mathcal{O}(n^2 \cdot m^2)$ time whether Q has safe paths.*

A CQ Q has safe atom-to-atom paths if every path between any two dynamic atoms includes at least one of their common variables. For each pair $(R(\boldsymbol{X}), S(\boldsymbol{Y}))$ of dynamic atoms of Q, we first construct the Gaifman graph of the hypergraph of Q without the common variables $\boldsymbol{C} = \boldsymbol{X} \cap \boldsymbol{Y}$. The graph contains $\mathcal{O}(n)$ nodes and $\mathcal{O}(n^2)$ edges. We next choose any $x \in \boldsymbol{X} \setminus \boldsymbol{C}$ and $y \in \boldsymbol{Y} \setminus \boldsymbol{C}$ and check if x is reachable from y using the Breadth-First Search algorithm in $\mathcal{O}(n^2)$ time; if so, Q has unsafe atom-to-atom paths. We repeat this procedure for every pair of dynamic atoms, which gives the total cost of $\mathcal{O}(n^2 \cdot m^2)$.

A CQ Q has safe atom-to-variable paths if every path between a dynamic atom and a free variable includes at least one free variable from that atom. For each dynamic atom $R(\boldsymbol{X})$, we construct the Gaifman graph Q without the free variables from $R(\boldsymbol{X})$. For each atom containing a free variable y, we choose any $x \in \boldsymbol{X} \setminus \mathit{free}(Q)$ and check if x is reachable from y in the graph; if so, Q has unsafe atom-to-variable paths. The total time is $\mathcal{O}(n^2 \cdot m^2)$.

C.2 Proof of Proposition 18

Proposition 18.

- *Any CQ without static relations is q-hierarchical if and only if it has safe paths.*
- *The dynamic sub-query of any CQ with safe paths is q-hierarchical.*

We start with the proof of the second statement in Proposition 18. Consider a CQ Q with safe paths. We denote by $\mathit{dynAtoms}(X)$ the set of dynamic atoms that contain a variable X. Assume for the sake of contradiction that $\mathit{dyn}(Q)$ is not hierarchical. This means that Q contains two variables X and Y such that $\mathit{atoms}(X) \not\subseteq \mathit{atoms}(Y)$, $\mathit{atoms}(Y) \not\subseteq \mathit{atoms}(X)$, and $\mathit{atoms}(X) \cap \mathit{atoms}(Y) \neq \emptyset$. This implies that Q has three dynamic atoms $R^d(\boldsymbol{X})$, $S^d(\boldsymbol{Y})$, and $T^d(\boldsymbol{Z})$ such that $X \in \boldsymbol{X}$, $X \in \boldsymbol{Y}$, $X \notin \boldsymbol{Z}$, $Y \in \boldsymbol{Z}$, $Y \in \boldsymbol{Y}$, and

$Y \notin \boldsymbol{X}$. The path $\boldsymbol{P} = (X, Y)$ connects the two dynamic atoms $R^d(\boldsymbol{X})$ and $T^d(\boldsymbol{Z})$ such that $\boldsymbol{P} \cap \boldsymbol{X} \cap \boldsymbol{Z} = \emptyset$. This means that Q does not have safe atom-to-atom paths, which is a contradiction.

Assume now that $dyn(Q)$ is hierarchical, but not q-hierarchical. This implies that Q contains two dynamic atom $R^d(\boldsymbol{X})$ and $S^d(\boldsymbol{Y})$, a bound variable X with $X \in \boldsymbol{X}$ and $X \in \boldsymbol{Y}$, and a free variable Y with $Y \in \boldsymbol{Y}$ and $Y \notin \boldsymbol{X}$. The path $\boldsymbol{P} = (X, Y)$ connects the the dynamic atoms $R^d(\boldsymbol{X})$ with the free variable Y such that $\boldsymbol{P} \cap free(Q) \cap \boldsymbol{X} = \emptyset$. This means that Q does not have safe atom-to-variable paths, which is again a contradiction.

Next, we prove the first statement in Proposition 18. The "$\Leftarrow$"-direction follows directly from the second statement. It remains to show the "$\Rightarrow$"-direction. Consider a q-hierarchical CQ Q without static atoms.

We first show that Q has safe atom-to-atom paths. Consider a path $\boldsymbol{P}$ that connects two atoms $R^d(\boldsymbol{X})$ and $S^d(\boldsymbol{Y})$. It follows from the structural properties of q-hierarchical queries that $\boldsymbol{P}$ must contain a variable X that is contained in $\boldsymbol{X}$ and in $\boldsymbol{Y}$. Hence, $\boldsymbol{P} \cap \boldsymbol{X} \cap \boldsymbol{Y} \neq \emptyset$. This mean that Q has safe atom-to-atom paths.

Now, we show that Q has safe atom-to-variable paths. Let $\boldsymbol{P}$ be a path that connects an atom $R^d(\boldsymbol{X})$ with a free variable Y. Without loss of generality, assume that Y is not included in $\boldsymbol{X}$. Otherwise, $\boldsymbol{P} \cap free(Q) \cap \boldsymbol{X} \neq \emptyset$ and the property holds. Let $X \in \boldsymbol{X}$ be the starting point of $\boldsymbol{P}$ in $R^d(\boldsymbol{X})$. The structural properties of hierarchical queries imply that $atoms(X) \supset atoms(Y)$. Since Y is free, X must be free. This implies however that $\boldsymbol{P} \cap free(Q) \cap \boldsymbol{X} \neq \emptyset$. This means that Q has safe atom-to-variable paths.

D Missing Details in Section 5

We introduce two notions that will be useful for the proofs in the following sections: *static parts of queries* and *VOs with neck*.

A *static part* of a query Q is constructed as follows. Without loss of generality, assume that Q does not have repeating relation symbols. In case a relation symbol R appears $k > 1$ times in the query, we can create k copies of R in the database and refer to each copy by a distinct relation name. We call an atom $R(\boldsymbol{X}')$ the *reduced version* of a static atom $R(\boldsymbol{X})$ in Q if $\boldsymbol{X}'$ results from $\boldsymbol{X}$ by skipping variables that appear in dynamic atoms of Q. Let Q' result from $stat(Q)$ by replacing all atoms with their reduced versions. Consider a connected component $\mathcal{C}$ of Q'. Let $\mathcal{A}$ be the set of all static atoms in Q whose reduced versions are in $\mathcal{C}$. Let $\boldsymbol{Y}$ be the set of variables that appear in the atoms in $\mathcal{A}$ and in $dyn(Q)$ and $\boldsymbol{F}$ the free variables of Q that appear in the atoms in $\mathcal{A}$. We call $Q_p(\boldsymbol{F}) = (R(\boldsymbol{Z}))_{R(\boldsymbol{Z}) \in \mathcal{A}}$ a static part of Q with intersection set $\boldsymbol{Y}$.

Example 41. *Figure 7 (top) shows a $\mathcal{C}_{\texttt{poly}}$-query Q, its dynamic sub-query $dyn(Q)$, and the static parts of Q with their intersection sets below. For instance, the third static part $Q_p(C) = R(G, C), S(G, F)$ in the figure has the intersection set $\{C, F\}$.*

An important observation, which will be used in the following proofs, is that the intersection set of each static part is covered by a single dynamic atom:

Proposition 42. *For any query Q in $\mathcal{C}_{\texttt{poly}}$ and any static part Q_p of Q with intersection set $\boldsymbol{Y}$, there is a single dynamic atom $R^d(\boldsymbol{X})$ in Q such that $\boldsymbol{Y} \subseteq \boldsymbol{X}$.*

Proof. For the sake of contradiction, assume that we need least two distinct dynamic atoms $R_1^d(\boldsymbol{X}_1)$ and $R_2^d(\boldsymbol{X}_2)$ from Q to cover the variables in $\boldsymbol{Y}$. This means that there are variables $X_1 \in \boldsymbol{X}_1$ and $X_2 \in \boldsymbol{X}_2$ such that $X_1 \notin \boldsymbol{X}_1 \cap \boldsymbol{X}_2$ and $X_2 \notin \boldsymbol{X}_1 \cap \boldsymbol{X}_2$. By construction of Q_p and its intersection set, there must be a path $\boldsymbol{P}$ in Q of the form $(X_1 = Y_1, \dots, Y_n = X_2)$, where all variables Y_i with $i \in \{2, \dots, n-1\}$ appear only in static atoms. Hence, none of the variables Y_i with $i \in \{2, \dots, n-1\}$ can be included in the intersection $\boldsymbol{X}_1 \cap \boldsymbol{X}_2$. Since X_1 and X_2 are not included in $\boldsymbol{X}_1 \cap \boldsymbol{X}_2$ either, it holds $\boldsymbol{P} \cap \boldsymbol{X}_1 \cap \boldsymbol{X}_2 = \emptyset$. This means that Q does not have safe atom-to-atom paths. Thus, Q is not in $\mathcal{C}_{\texttt{poly}}$, which is a contradiction. $\square$

Given a VO ω and a subset $\mathcal{N}$ of its variables, we say that ω has *neck* $\mathcal{N}$ if its upper part is a path consisting of the variables in $\mathcal{N}$. More formally, there is an ordering $X_1, \dots, X_n$ of the variables in $\mathcal{N}$ such that X_1 is the root of ω and for each $i \in [n-1]$, X_{i+1} is the only child of X_i.

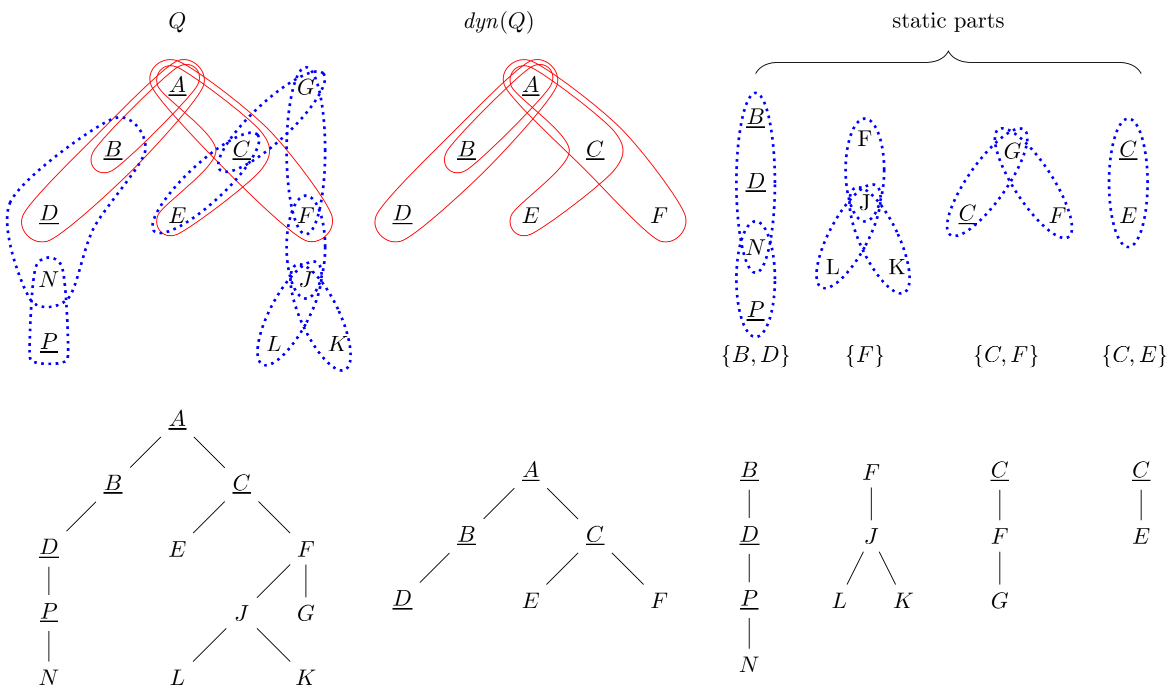


Figure 7: (top) Query Q, its dynamic sub-query $dyn(Q)$, and its static parts with the intersection sets. (bottom) Free-top VO for Q, free-top canonical VO for $dyn(Q)$, and free-top VOs with neck $\boldsymbol{Y}$ for each static part with intersection set $\boldsymbol{Y}$. For simplicity, the atoms in the VOs are omitted.

Example 43. *Figure 7 (bottom right) shows for each static part with intersection set $\boldsymbol{Y}$, a VO with neck $\boldsymbol{Y}$. For simplicity, we omit the atoms in the VOs. In each VO, the intersection variables form a path and are on top of all the other variables. For instance, the VO for the third static part $Q_p(C) = R(G, C), S(G, F)$ has the intersection variables C and F above G.*

D.1 Proof of Proposition 25

Proposition 25. *Any query in $\mathcal{C}_{\mathtt{poly}}$ has a free-top VO whose dynamic part is canonical.*

Given a query Q in $\mathcal{C}_{\mathtt{poly}}$, we show how to construct a free-top VO for Q whose dynamic part is q-hierarchical. The high-level idea of the construction is as follows. By Proposition 18, the dynamic sub-query $dyn(Q)$ of Q is q-hierarchical, hence, it admits a canonical VO. First, we construct a canonical VO ω^d for $dyn(Q)$. Then, we split the static sub-query $stat(Q)$ into several parts for which we create free-top VOs. Finally, we combine ω^d with the VOs for the static parts and show that the resulting structure is a free-top VO for Q.

We first show that for any static part Q_p of Q with intersection set $\boldsymbol{Y}$, we can construct a free-top VO with neck $\boldsymbol{Y}$.

Example 44. *Figure 7 shows under each static part of the query Q with intersection set $\boldsymbol{Y}$, a free-top VO with neck $\boldsymbol{Y}$ for the static part. Observe that in the VO for the third static part $Q_p(C) = R(G, C), S(G, F)$, the free variable C is on top of F and G, and the intersection variables C and F are on top of G.*

First, we show:

Proposition 45. *Let Q be a CQ in $\mathcal{C}_{\mathtt{poly}}$ and Q_p a static part of Q with intersection set $\boldsymbol{Y}$. If free$(Q_p) \setminus \boldsymbol{Y} \neq \emptyset$, then all variables in $\boldsymbol{Y}$ are free.*

Proof. For the sake of contradiction, assume that $\boldsymbol{Y}$ contains a bound variable Y and Q_p has a free variable X that is not contained in $\boldsymbol{Y}$. By Proposition 42, Y is included in a dynamic atom $R(\boldsymbol{X})$ of Q. Since $\boldsymbol{Y}$ is

$\textsc{create}\text{VO}(\mathcal{C}_{\texttt{poly}}\text{-query } Q)$: free-top VO for Q with canonical dynamic part

let ω^d be the canonical VO for $dyn(Q)$
let $Q_1, \ldots, Q_n$ be the static parts of Q with intersection sets $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$
let $\omega = \omega^d$
foreach $i \in [n]$
  let ω_i be a VO for Q_i with neck $\boldsymbol{Y}_i$
  add ω_i to ω
return ω

Figure 8: Given a $\mathcal{C}_{\texttt{poly}}$-query Q, constructing a free-top VO whose dynamic part is canonical.

the intersection set of Q_p, the variable X cannot be covered by any dynamic atom of Q. This means that Q contains a path $\boldsymbol{P}$ of the form $Y = X_1, \ldots, X_n = X$ that connects $R(\boldsymbol{X})$ with the free variable X such that $\mathit{free}(Q) \cap \boldsymbol{X} \cap \boldsymbol{P} = \emptyset$. This implies that Q does not have safe atom-to-variable paths. Hence, Q cannot be in $\mathcal{C}_{\texttt{poly}}$, which is a contradiction. ☐

Using Proposition 45, we show:

Proposition 46. *Let Q be a CQ in $\mathcal{C}_{\texttt{poly}}$ and Q_p a static part of Q with intersection set $\boldsymbol{Y}$. There is a free-top VO for Q_p with neck $\boldsymbol{Y}$.*

Proof. It follows from Proposition 45 that all free variables in Q_p are in $\boldsymbol{Y}$. Hence, we can construct the following free-top VO ω for Q with neck $\boldsymbol{Y}$. We construct a path $\boldsymbol{N}$ out of the variables in $\boldsymbol{Y}$ such that the free variables in $\boldsymbol{Y}$ are on top of all other variables in $\boldsymbol{Y}$. This path becomes the neck of ω. In the rest of ω, we again make sure that the free variables are on top of the other variables. Observe that this is always possible, since in the worst case, ω is path, in which case the variables can be ordered arbitrarily. ☐

We describe now how to construct from the free-top VOs for the static parts of Q, a free-top VO ω for the overall query such that the dynamic part of ω is canonical. The construction is described by the procedure $\textsc{create}$VO in Figure 8. The procedure first constructs a canonical VO ω^d for $dyn(Q)$. The existence of such a canonical VO is guaranteed by the fact that $dyn(Q)$ is q-hierarchical (Proposition 18). Then, for each static part Q_i with intersection set $\boldsymbol{Y}_i$, it constructs a free-top VO ω_i with neck $\boldsymbol{Y}_i$. The existence of such a VO is guaranteed by Proposition 46. We obtain the final VO by attaching the VOs for the static parts to ω^d. We explain next how a VO ω_i for a static part is attached to ω^d.

By Proposition 42, the neck of ω_i is included in a single atom of $dyn(Q)$. Hence, all neck variables are on a root-to-leaf path of ω^d. If $\boldsymbol{Y}_i$ is empty, we add ω_i as a separate tree to ω. Otherwise, let X be the lowest variable in ω_i that is contained in $\boldsymbol{Y}_i$ and let Y be the lowest variable in ω^d that is contained in $\boldsymbol{Y}_i$. We make the child trees of X to child trees of Y.

Example 47. *Figure 7 shows under the query Q the free-top VO obtained by putting together the canonical VO for $dyn(Q)$ and the free-top VOs for the static parts.*

It remains to show:

Proposition 48. *For any query Q in $\mathcal{C}_{\texttt{poly}}$, $\textsc{create}VO(Q)$ is a free-top VO for Q whose dynamic part is canonical.*

Proof. Consider a query Q in $\mathcal{C}_{\texttt{poly}}$ with static parts $Q_1, \ldots, Q_n$ and corresponding intersection sets $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$ and let $\omega = \mathit{createVO}(Q)$. Let ω^d be the canonical variable order of $dyn(Q)$ and $\omega_1, \ldots, \omega_n$ the free-top VOs of the static parts $Q_1, \ldots, Q_n$ with necks $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$, as guaranteed by Proposition 46.

We show that the variables of any atom $R(\boldsymbol{X})$ of Q is on a root-to-leaf path of ω. The atom $R(\boldsymbol{X})$ must be in $dyn(Q)$ or in a static part Q_i. In the former case, the variables in $\boldsymbol{X}$ lie on a root-to-leaf path in ω^d.

In the latter case, they lie on a root-to-leaf-path of ω_i. In both cases, they must lie on a root-to-leaf path in ω, since ω^d and ω_i are subtrees in ω.

For the sake of contradiction, assume that ω contains a bound variable Y above a free variable X. Since ω^d and $\omega_1, \ldots, \omega_n$ are free-top, it must hold $Y \in \mathit{vars}(\omega^d)$, $X \notin \mathit{vars}(\omega^d)$ and $X \in \mathit{vars}(\omega_i)$, for some $i \in [n]$. Let $R(\boldsymbol{X})$ be a dynamic atom with $Y \in \boldsymbol{X}$. It follows from the construction of ω that there is a path $\boldsymbol{P}$ from Y to X such that $\boldsymbol{X} \cap \mathit{free}(Q) \cap \boldsymbol{P} = \emptyset$. This implies that Q does not have have safe atom-to-variable paths. Hence, Q is not a query from the class $\mathcal{C}_{\texttt{poly}}$, which is a contradiction.

Finally, by construction, the dynamic part ω^d of ω is canonical. □

D.2 Proof of Proposition 26

Proposition 26. *For any query Q in $\mathcal{C}_{\texttt{lin}}$, it holds* $\mathsf{w}(Q) = 1$.

Consider a query Q in $\mathcal{C}_{\texttt{lin}}$. We will show that we can construct for Q a free-top VO ω with preprocessing width 1 such that the dynamic part of ω is canonical. First, we will construct for every static part Q_p of Q with intersection set $\boldsymbol{Y}$, a free-top VO with neck $\boldsymbol{Y}$ and preprocessing width 1. Then, we use the procedure createVO from Figure 8 to build a free-top VO for Q with preprocessing width 1, whose dynamic part is canonical.

Constructing VOs for Static Parts Consider a static part Q_p of Q with intersection set $\boldsymbol{Y}$. We explain how to construct for Q_p a free-top VO with neck $\boldsymbol{Y}$ and preprocessing width 1. By definition, Q is free-connex acyclic. It follows from prior work that a query is free-connex acyclic if and only if it has a free-top VO ω that has preprocessing width 1 but whose dynamic part is not necessarily canonical [26]. Let Q' and Q'_p be the queries obtained from Q and respectively Q_p by adding a fresh atom $R(\boldsymbol{Y})$. The query Q' must be free-connex acyclic either. Hence, it has a free-top VO ω with preprocessing width 1. We start with such a VO ω for Q' and eliminate one-by-one all variables that do not appear in Q'_p. When removing a variable X with parent variable Y, the children of X become the children of Y. If a removed variable X does not have a parent, the child trees of X become independent trees. Observe that after all variables that are not contained in Q'_p are removed, we are left with a valid free-top VO for Q'_p whose preprocessing width is 1. Hence, it is free-connex acyclic. This means that it has a free-connex tree decomposition where each bag is covered by single atom [3]. We root the tree decomposition at the bag that contains $R(\boldsymbol{Y})$. This decomposition can be transformed into a free-top VO of width 1 following a construction in prior work [26]. The construction proceeds top-down and an leaves the variables of each bag on a root-to-leaf path. hence, the resulting VO has neck $\boldsymbol{Y}$.

Constructing a VO for Q Let $Q_1, \ldots, Q_n$ be the static parts of Q with intersection sets $\boldsymbol{Y}_1, \ldots \boldsymbol{Y}_n$. For each $i \in [n]$, let Q'_i be the extension of Q_i by a fresh atom $R_i(\boldsymbol{Y}_i)$. The purpose of using $R_i(\boldsymbol{Y}_i)$ in the query Q_i is to guarantee that in any VO for Q_i, the variables in $\boldsymbol{Y}_i$ are on a root-to-leaf path. Let ω^d be the canonical VO of $\mathit{dyn}(Q)$ and $\omega_1, \ldots \omega_n$ the free-top VOs for $Q'_1, \ldots, Q'_n$ constructed as described above. We use the procedure createVO in Figure 8 to combine the VO ω^d with the VOs $\omega_1, \ldots, \omega_n$. The same arguments as in the proof of Proposition 48 imply that we obtain a valid free-top VO ω for Q whose dynamic part is canonical. It remains to explain that the preprocessing width of ω is 1. Consider a variable X that is contained in a dynamic atom. Since the dynamic part of ω is canonical, all ancestor variables of X are contained in each dynamic atom below X. This implies $\rho^*_{Q_X}(\{X\} \cup \mathit{dep}_\omega(X)) = 1$. Consider now a variable X that does not appear in $\mathit{dyn}(Q)$ in some static part Q_i. Since ω_i has preprocessing width 1, it holds $\rho^*_{Q_X}(\{X\} \cup \mathit{dep}_{\omega_i}(X)) = 1$ within ω_i. Since all variables that depend on X must be included in Q_i, we obtain $\rho^*_{Q_X}(\{X\} \cup \mathit{dep}_\omega(X)) = 1$ within ω. Overall, we derive that the preprocessing width of ω and, hence, of Q is 1.

D.3 Proof of Proposition 29

Proposition 29. *For any query Q in $\mathcal{C}_{\texttt{poly}}$, VO ω for Q, and database of size N, $\text{REWRITE}(\omega)$ is a safe rewriting for Q with $\mathcal{O}(N^{\mathsf{w}})$ computation time, where w is the preprocessing width of Q.*

Given a query Q in $\mathcal{C}_{\texttt{poly}}$, a VO ω for Q, and database of size N, let $\mathcal{T} = \text{REWRITE}(\omega)$. We show that $\mathcal{T}$ satisfies all properties for safe rewritings, as specified in Definition 10.

We first show that the *correctness properties* for safe rewritings hold. By the definition of VOs, the variables of each atom must be on a root-to-leaf path and each atom is placed under its lowest variable in the VO. This implies that for each connected component $\mathcal{C}$ of Q, all atoms in $\mathcal{C}$ must be in a single tree of $\mathcal{T}$ (first correctness property). Since all atoms containing a variable X must be below X in the VO, all atoms that contain X must be in the subtree rooted at a projection view V_X (second correctness property).

Now we show that the *update property* holds for $\mathcal{T}$. By definition, the dynamic part of ω is canonical. Hence, the schema of each dynamic atom covers all ancestor variables. This implies that also each dynamic view V contains in its schema all its ancestor variables. Consider a dynamic view V_X and its sibling view V_Y. By construction, we need to consider two cases. In the first case, V_Y is an atom. In this case, the schema of V_Y is included in its ancestor variables. Hence, V_X covers the schema of V_Y. In the second case, V_Y is a projection view at some variable Y that does not include Y. In this case, the view V_Y can only contain variables that are ancestors of V_X. Hence also in this case, V_X covers all variables of V_Y.

The *enumeration property* follows simply from the fact that the $\mathcal{T}$ follows a free-top VO.

It remains to show that all views can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time. The proof follows closely prior work using view trees [17]. Let T be a tree in $\mathcal{T}$. We show by induction on the structure of T that every view in T can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time.

Base Case: The base case states that each leaf atom can be computed in $\mathcal{O}(N^{\mathsf{w}})$, which is obviously true.

Induction Step: Consider a projection view a $V'_X(\mathit{dep}(X))$ in T. Such a view results from its single child view by projecting away X. By induction hypothesis, the view V_X can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time. Hence, it is of size $\mathcal{O}(N^{\mathsf{w}})$. The view V'_X can be constructed from V_X by a single scan, which takes $\mathcal{O}(N^{\mathsf{w}})$ time.

Consider now a join view $V_X(\{X\} \cup \mathit{dep}(X))$ in T. Let $V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ be the child views of V_X. By induction hypothesis, each child view can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time. By construction of T, any variable that appears in at least two of the child views must be contained in the schema of V_X. This mean that variables that do not appear in $\{X\} \cup \mathit{dep}(X)$ cannot be join variables among the child views of V_X. In each child view we project away all non-join variables that do not appear in $\{X\} \cup \mathit{dep}(X)$, using $\mathcal{O}(N^{\mathsf{w}})$ time. Let $V'_1(\mathcal{S}'_1), \ldots, V'_k(\mathcal{S}'_k)$ be the resulting child views. Using a worst-case optimal join algorithm [24], we then compute the view V_X from its child views in $\mathcal{O}(|V_X|)$ time. The size of V_X is upper-bounded by $\mathcal{O}(N^p)$ where $p = \rho^*_{Q_X}(\{X\} \cup \mathit{dep}(X))$ and Q_X is the query that joins all atoms in the subtree rooted at V_X. Since $p \leq \mathsf{w}$, the view V_X can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time.

E Missing Details in Section 6

In Section 6, we described how to evaluate a $\mathcal{C}_{\texttt{exp}}$-query using a transition system. We explain here how to build in $\mathcal{O}(N^{2\rho^*(\mathit{stat}(Q))})$ time an index that allows us to move from the current state in the transition system to another state for a single-tuple update in constant time.

We first describe how to build an index that allows fast lookups of a state in the transition system for a given snapshot of the dynamic relations. Consider a dynamic relation R^d in Q. As we discussed in Section 6, we restrict the domain of R^d to those values appearing in the static relations in Q, which is denoted by $\mathcal{A}_{R^d}$. The size of $\mathcal{A}_{R^d}$ is $\mathcal{O}(N^{\rho^*(\mathit{stat}(Q))})$. We index the values in $\mathcal{A}_{R^d}$ from 0 to $|\mathcal{A}_{R^d}| - 1$. Each instance of R^d is a subset of the set $\{0, \ldots, |\mathcal{A}_{R^d}| - 1\}$. We could represent this using a bitset that needs $\log |\mathcal{A}_{R^d}|$ bits, where each bit represents whether the corresponding tuple is in the instance of R^d. In other words, we encode each instance of R^d as a bitset, i.e., an integer in $\{0, \ldots, 2^{|\mathcal{A}_{R^d}|} - 1\}$. For a state $(\alpha_1, \ldots, \alpha_k)$, where α_i is an instance of the dynamic relation R^d_i, we encode $(\alpha_1, \ldots, \alpha_k)$ as a tuple $(i_1, \ldots, i_k)$, where i_j is the bitset that represents α_j. It takes $\mathcal{O}(N^{\rho^*(\mathit{stat}(Q))})$ time to encode a state.

We create an index T that is a k-dimensional tensor, where each dimension corresponds to the bitsets representing the instances of a dynamic relation. The entry $T(i_1, \dots, i_k)$ with the indices $(i_1, \dots, i_k)$ is a pointer pointing to the state that corresponds to the instances of the dynamic relations represented by the bitsets $(i_1, \dots, i_k)$. Each dimension has $2^{\mathcal{O}(N^{\rho^*(stat(Q))})}$ entries. For a given bitset, we can look it up in the corresponding dimension in $\log 2^{\mathcal{O}(N^{\rho^*(stat(Q))})}$ time, which is $\mathcal{O}(N^{\rho^*(stat(Q))})$. Therefore, for a tuple $(i_1, \dots, i_k)$ of bitsets, we can look up the corresponding pointer in T in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time. In the preprocessing step, we create all states without transitions as described in Section 6, and then we fill in the entries of T.

We now create the transitions between the states. Consider a state $(\alpha_1, \dots, \alpha_k)$ and an insertion of a tuple $a \in \mathcal{A}_{R_i^d} \setminus \alpha_i$ to a dynamic relation R_i^d. We want to move to the state that contains additionally a, i.e., $(\alpha_1, \dots, \alpha_i \cup \{a\}, \dots, \alpha_k)$. We encode $(\alpha_1, \dots, \alpha_k)$ as a tuple $(i_1, \dots, i_k)$ of bitsets in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time and look up the pointer $p = T(i_1, \dots, i_k)$ to this state from the tensor in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time. We then create the transition: $\delta((\alpha_1, \dots, \alpha_k), +a) = p$. We do the same for the deletion of a tuple $a \in \alpha_i$, i.e., $\delta((\alpha_1, \dots, \alpha_k), -a) = p'$, where $p' = T(i_1, \dots, i_k)$, and then the similar process for the other dynamic relations. Overall, within one state, there can be $\mathcal{O}(N^{\rho^*(stat(Q))})$ transitions, so we spend $\mathcal{O}(N^{\rho^*(stat(Q))}) \cdot \mathcal{O}(N^{\rho^*(stat(Q))}) = \mathcal{O}(N^{2\rho^*(stat(Q))})$ time to find all pointers for the transitions in the state.

To allow for moving from one state to another state for a single-tuple update in constant time, we build a hash table that maps each value $a \in \mathcal{A}_{R_i^d}$ to the corresponding pointer to the next state. The size of the hash table is $\mathcal{O}(N^{\rho^*(stat(Q))})$, and we can build it in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time.

Overall, we can build the index for the transitions in $\mathcal{O}(N^{2\rho^*(stat(Q))})$ time.

F Missing Details in Section 7

We prove the lower bound in Theorem 1, which states:

Proposition 49. *Any $Q \notin \mathcal{C}_{\texttt{lin}}$ without self-joins cannot be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay, where N is the database size, unless the OMv or the BMM conjecture fails.*

We first show a lower bound for the simple queries $Q_{RST}() = R^d(A),\ S^s(A,B),\ T^d(B)$ and $Q_{ST}(A) = S^s(A,B),\ T^d(B)$, which are both not included in $\mathcal{C}_{\texttt{lin}}$.

F.1 Lower Bound for Q_{RST}

The lower bound for the query $Q_{RST}() = R^d(A),\ S^s(A,B),\ T^d(B)$ relies on the OuMv conjecture, which is implied by the OMv conjecture [12]:

Proposition 31. *The CQ $Q_{RST}() = R^d(A),\ S^s(A,B),\ T^d(B)$ cannot be evaluated with $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, $\mathcal{O}(N^{1/2-\gamma})$ update time, and $\mathcal{O}(N^{1/2-\gamma})$ enumeration delay for any $\gamma > 0$, where N is the database size, unless the OuMv conjecture fails.*

Assume that we have an Algorithm $\mathcal{A}$ that maintains the query Q_{RST} with $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, $\mathcal{O}(N^{1/2-\gamma})$ update time, and $\mathcal{O}(N^{1/2-\gamma})$ enumeration delay for some $\gamma > 0$. We show that we can use the Algorithm $\mathcal{A}$ to solve the OuMv problem given by Definition 39 in subcubic time, which contradicts Conjecture 40.

Considering an n-by-n matrix M and n pairs of vectors $(u_1, v_1), \dots, (u_n, v_n)$ that serve as the input to the OuMv problem, we construct an Algorithm $\mathcal{B}$ that uses the static relation $S^s(A,B)$ to encode M and the two dynamic relations $R^d(A)$ and $T^d(B)$ to encode the vector pairs $(u_r, v_r)_{r \in [n]}$. Next, we explain the encoding in detail.

Algorithm $\mathcal{B}$ starts with the empty relations $R^d(A)$, $S^s(A,B)$ and $T^d(B)$. First, it populates relation $S^s(A,B)$ such that $(i,j) \in S^s(A,B)$ if and only if $M(i,j) = 1$. After the initial population, the static relation $S^s(A,B)$ does not accept any update. The two dynamic relations, $R^d(A)$ and $T^d(B)$, are empty before the first vector pair is given. Then, it executes the preprocessing mechanism of Algorithm $\mathcal{A}$. In round 1, Algorithm $\mathcal{B}$ receives the vector pair (u_1, v_1) and updates $R^d(A)$ and $T^d(B)$ so that $i \in R(A)$ if and only if $u_1[i] = 1$ and $i \in T(B)$ if and only if $v_1[i] = 1$. We observe that $u_1^T M v_1 = 1$ if and only if the (Boolean) result of Q_{RST} is true. Algorithm $\mathcal{B}$ triggers the enumeration mechanism of $\mathcal{A}$ and outputs 1 if the result of Q_{RST} is true. Otherwise, it outputs 0. In each round, this procedure is repeated.

We analyse the overall time used by Algorithm $\mathcal{B}$. Given that M is an $n \times n$ matrix M, the size and construction time of relation S^s are both $\mathcal{O}(n^2)$. It results in a database D of size $|D| = \mathcal{O}(n^2)$. The preprocessing time is

$$\mathcal{O}((n^2)^{3/2-\gamma}) = \mathcal{O}(n^{3-2\gamma})$$

In each round $r \in [n]$, Algorithm $\mathcal{B}$ executes at most $4n$ updates to $R^d(A)$ and $T^d(B)$ and enumerates the first result of Q_{RST}. Since the database size remains $\mathcal{O}(n^2)$, the time to update the relations is

$$\mathcal{O}(2n \cdot (n^2)^{0.5-\gamma}) = \mathcal{O}(n^{2-2\gamma})$$

It is sufficient to enumerate the first result tuple to check if Q_{RST} is empty. The time to check emptiness is,

$$\mathcal{O}((n^2)^{0.5-\gamma}) = \mathcal{O}(n^{1-2\gamma})$$

Hence, for n rounds, the overall time is $\mathcal{O}(n^{3-2\gamma})$. We conclude that the overall time Algorithm $\mathcal{B}$ takes to solve the OuMv problem is $\mathcal{O}(n^{3-2\gamma})$, thus contradicting Conjecture 40 which rules out sub-cubic solutions for the OuMv problem.

F.2 Lower Bound for the Query Q_{ST}

Proposition 50. *The CQ $Q_{ST}(A) = S^s(A, B), T^d(B)$ cannot be evaluated with $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, $\mathcal{O}(N^{1/2-\gamma})$ update time, and $\mathcal{O}(N^{1/2-\gamma})$ enumeration delay for any $\gamma > 0$, where N is the database size, unless the OMv conjecture fails.*

The proof is similar to that of Proposition 31. The main difference is the direct reduction from the OMv problem given by Definition 37.

Assume there is an Algorithm $\mathcal{A}$ that maintains the query Q_{ST} with $\mathcal{O}(N^{3/2-\gamma})$ preprocessing time, $\mathcal{O}(N^{1/2-\gamma})$ update time, and $\mathcal{O}(N^{1/2-\gamma})$ enumeration delay for some $\gamma > 0$, We show that the existence of Algorithm $\mathcal{A}$ contradicts Conjecture 38.

We construct an Algorithm $\mathcal{B}$ that uses the static relation $S^s(A, B)$ to encode the n-by-n matrix M and a dynamic relation $T^d(B)$ to encode each vector $v_{r\in[n]}$, where M and $v_{r\in[n]}$ are the input to the OMv problem.

Algorithm $\mathcal{B}$ first populates relation S^s to reflect the non-empty entries in M. Similar to the proof of Proposition 31, Algorithm $\mathcal{B}$ executes the preprocessing mechanism of Algorithm $\mathcal{A}$ in the beginning and $\mathcal{O}(n)$ updates to $T^d(B)$ when a vector $v_{r\in[n]}$ arrives. To obtain the result of $Mv_{r\in[n]}$, Algorithm $\mathcal{B}$ enumerates the result of Q_{ST}.

The static relation S^s dominates the size of the database $|D| = \mathcal{O}(n^2)$. The preprocessing time of Algorithm $\mathcal{B}$ is,

$$\mathcal{O}((n^2)^{1.5-\gamma}) = \mathcal{O}(n^{3-2\gamma})$$

The total update and enumeration time over n rounds is,

$$\mathcal{O}(n \cdot n \cdot (n^2)^{0.5-\gamma}) = \mathcal{O}(n^{3-2\gamma})$$

The subcubic processing time contradicts Conjecture 38.

F.3 Proof of Proposition 49

Consider a CQ $Q \notin \mathcal{C}_{\texttt{lin}}$. By definition of the class $\mathcal{C}_{\texttt{lin}}$, one of the following cases holds: (1) Q is not free-connex acyclic, or (2) Q does not have safe atom-to-atom paths, or (3) Q does not have safe atom-to-variable paths. If Case (1) holds, then Q does not admit constant-delay enumeration after liner time preprocessing even without executing any updates [3]. In the sequel we show how we can reduce the the evaluation of one of the queries Q_{RST} and Q_{ST} if Case (2) or Case (3) holds for Q. This reduction transfers the lower bound for Q_{RST} and Q_{ST} to Q.

Violation of the Atom-to-Atom Path Property Consider a CQ Q that does not have safe atom-to-atom paths. This means that the query contains at least one path $\boldsymbol{P}$ of the form $X_1, \ldots, X_n$ that connects two dynamic atoms $\hat{R}^d(\boldsymbol{A})$ and $\hat{T}^d(\boldsymbol{B})$ such that $\boldsymbol{P} \cap \boldsymbol{A} \cap \boldsymbol{B} = \emptyset$. This implies $n \geq 2$. The path connecting the two dynamic atoms is illustrated in Figure 9.

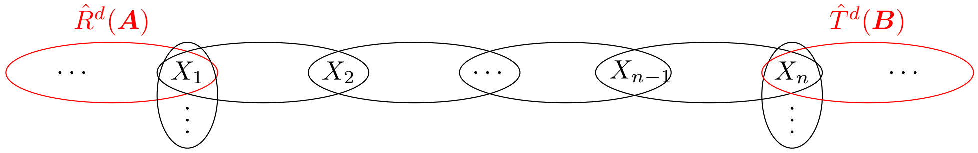


Figure 9: Illustration of a path in a query that does not have safe atom-to-atom paths.

The idea of the reduction of the evaluation of the query Q_{RST} to the evaluation of the query Q is as follows. We use the relations on the path $\boldsymbol{P}$ to encode the static relation S^s and use the dynamic relations $\hat{R}^d$ and $\hat{T}^d$ to encode the dynamic relations R^d and T^d in Q. In particular, we use the variables $X_1, \ldots, X_{n-1}$ to simulate the variable A and the variable X_n to simulate the variable B in Q_{RST}. This means, the variables $X_1, \ldots, X_n$ are assigned the same values. All other variables in Q are assigned a fixed dummy value.

In the preprocessing stage, we construct the relations constituting the path $\boldsymbol{P}$ such that the above encoding is satisfied. Each update to R^d or T^d is translated into an update to $\hat{R}^d$ or respectively $\hat{T}^d$. Observe that the property $\boldsymbol{P} \cap \boldsymbol{A} \cap \boldsymbol{B} = \emptyset$ ensures that the simulations of the variables A and B do not interfere with each other. Each enumeration request to Q_{RST} is translated into an enumeration request to Q. The answer of Q_{RST} is true if and only if the result of Q is non-empty.

Violation of the Atom-to-Variable Property Assume that the query Q does not have safe atom-to-variable paths. This means that the query contains a path $\boldsymbol{P}$ of the form $X_1, \ldots, X_n$ that connects a dynamic atom $\hat{T}^d(\boldsymbol{B})$ with the free variable X_1 such that $\boldsymbol{P} \cap \boldsymbol{B} \cap \mathit{free}(Q) = \emptyset$. The path in Q is illustrated in Figure 10.

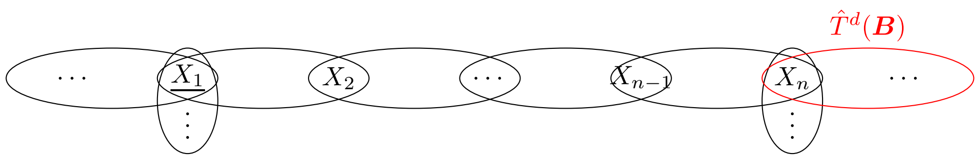


Figure 10: Illustration of a path in a query that does not have safe atom-to-variable paths.

The idea of the reduction is similar to the previous case. We use the relations constituting the path to encode the static relation S and use the dynamic relation $\hat{T}^d$ to simulate the dynamic relation T^d In particular, we use the variables $X_2, \ldots, X_n$ to simulate the variable B in Q_{ST} and use the free variable X_1 to simulate the free variable A in Q_{ST}.