Tractable Conjunctive Queries over Static and Dynamic Relations

Ahmet Kara
OTH Regensburg, Germany

Zheng Luo
University of California, Los Angeles, CA, USA

Milos Nikolic
University of Edinburgh, UK

Dan Olteanu
University of Zurich, Switzerland

Haozhe Zhang
University of Zurich, Switzerland

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-queries over the static relations and over the dynamic relations, respectively. 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.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory); Information systems → Database views; Information systems → Data streams

Keywords and phrases fully dynamic algorithm, constant enumeration delay, constant update time

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.12

Related Version *Extended technical report*: `https://arxiv.org/abs/2404.16224` [14]

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

Acknowledgements Olteanu would like to thank the IVM team at RelationalAI and in particular Niko Göbel for discussions that motivated this work. Luo's contribution was done while he was with the University of Zurich.

1 Introduction

Incremental view maintenance, also known as fully dynamic query evaluation, is a fundamental task in data management [22, 13, 10, 16, 28, 19, 29, 9]. A natural question is to understand which conjunctive 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 showed that the q-hierarchical queries are the conjunctive queries without repeating relation symbols that are tractable; such queries admit constant update time and enumeration delay, even

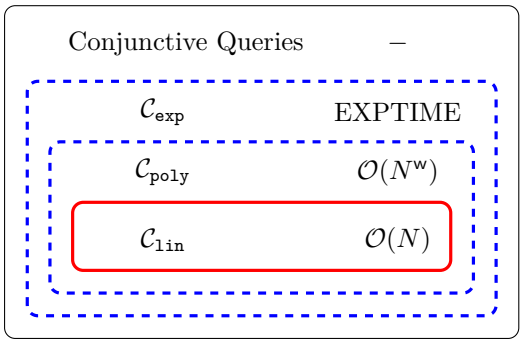


Figure 1 Classes $\mathcal{C}_{\texttt{lin}} \subset \mathcal{C}_{\texttt{poly}} \subset \mathcal{C}_{\texttt{exp}}$ of tractable conjunctive queries over static and dynamic relations and the corresponding preprocessing time (data complexity). N is the current database size and $\texttt{w}$ is the preprocessing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

if we only allow for linear time preprocessing [4]. All other conjunctive queries without repeating relation symbols cannot admit both constant update time and constant enumeration delay, even if 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, 21]. 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 [18]. Beyond this well-behaved query class, there are known intractable queries such as the triangle queries [15, 16] and hierarchical conjunctive queries with arbitrary free variables [17, 20] for which algorithms are known to achieve worst-case, yet not constant, optimal update time and enumeration delay. Restrictions to the data and the update sequence can turn intractable queries into tractable ones: for instance, when the update sequence has a bounded enclosureness values [29] or when the data satisfies functional dependencies [19], degree bounds [5], or more general integrity constraints [6].

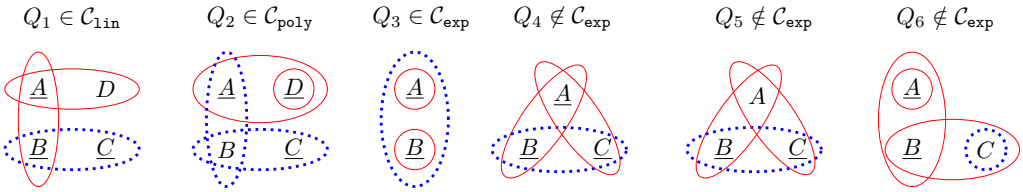
All aforementioned works consider the *all-dynamic setting*, where all relations are updatable. In this work, we extend the tractability frontier by considering a *mixed setting*, where the input database can have both dynamic relations, which are subject to change, and static relations, which do not change. The mixed setting appears naturally in practice, e.g., in the retail domain where some relations (e.g., Inventory, Sales) are updated very frequently while others (e.g., Stores, Demographics) are only updated sporadically if at all. 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: $\mathcal{C}_{\texttt{lin}} \subset \mathcal{C}_{\texttt{poly}} \subset \mathcal{C}_{\texttt{exp}}$. The queries are categorised based on their structure and on the structure of their static and dynamic sub-queries. These classes are defined in Section 4.

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

▶ **Theorem 1.** *Let a conjunctive query Q and a database of size N.*
If Q is in $\mathcal{C}_{lin}$, then it can be maintained with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.



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

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.

- *If Q is not in $\mathcal{C}_{lin}$ and does not have repeating relation symbols, then it cannot be maintained 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 the query Q using strata of views, where the views are defined using projections or joins of input relations and possibly other views (Section 3). We call such rewritings *safe* if the views can be maintained in constant time for single-tuple updates 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 in the mixed setting.

In the all-static setting, $\mathcal{C}_{\texttt{lin}}$ becomes the class of free-connex α-acyclic conjunctive 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 conjunctive 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.

The queries in $\mathcal{C}_{\texttt{poly}} \setminus \mathcal{C}_{\texttt{lin}}$ are also tractable, yet they require super-linear preprocessing time, unless the Online Matrix-Vector Multiplication or the Boolean Matrix-Matrix Multiplication conjectures fail. We introduce a new width measure w for conjunctive queries, called the *preprocessing width*, to characterise the preprocessing time of queries in $\mathcal{C}_{\texttt{poly}}$:

▶ **Theorem 3.** *Let a conjunctive query $Q \in \mathcal{C}_{poly}$, the preprocessing width* w *of Q, and a database of size N. The query Q can be maintained with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.*

Like the queries in $\mathcal{C}_{\mathtt{lin}}$, every query in $\mathcal{C}_{\mathtt{poly}}$ 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 $\mathcal{C}_{\mathtt{poly}} \setminus \mathcal{C}_{\mathtt{lin}}$: It is tractable but requires quadratic time preprocessing. The quadratic blowup is due to the creation of a view that is the join of 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 either the static sub-query or of the entire query [23], as exemplified next. Consider 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. Its static sub-query is the triangle join and has $\mathsf{fhtw} = 3/2$. The preprocessing width of Q_7 is 1, so it is in $\mathcal{C}_{\mathtt{lin}}$. The triangle join $Q_8(A, C) = R^s(A, B), S^s(B, C), T^d(A, C)$ has $\mathsf{fhtw} = 3/2$ and its static sub-query has $\mathsf{fhtw} = 2$. Yet the preprocessing width of Q_8 is 2: We materialise the static sub-query $V^s(A, C) = R^s(A, B), S^s(B, C)$. 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 [16].

The class $\mathcal{C}_{\mathtt{exp}}$ contains a broad set of tractable queries that may use up to exponential preprocessing time in the size of the input database:

▶ **Theorem 5.** *Let a conjunctive query Q, the static sub-query $stat(Q)$ of Q, the fractional edge cover number $\rho^*(Q)$ of Q, $p = \mathcal{O}(N^{\rho^*(stat(Q))})$, and a database whose static relations have size N. If Q is in $\mathcal{C}_{exp}$, then it can be maintained with $2^p \cdot p^2$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.*

The class $\mathcal{C}_{\mathtt{exp}}$ 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 α-acyclic conjunctive queries in $\mathcal{C}_{\mathtt{exp}}$. The fractional edge cover number characterises the worst-case optimal result size of the static sub-query of Q [2].

▶ **Example 6.** The query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ from Figure 2 is in $\mathcal{C}_{\mathtt{exp}} \setminus \mathcal{C}_{\mathtt{poly}}$. This query can be maintained with constant update time and enumeration delay at the expense of exponential preprocessing time: The possible results of Q_3 are any of the 2^N possible subsets of the static relation S^s of size N, while the updates to the dynamic relations only act as selectors in this powerset. The dynamic sub-query of Q_3, which is defined as the sub-query over the dynamic relations, is q-hierarchical. The static sub-query of Q_3, which is defined as the sub-query over the only 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 $\mathcal{C}_{\mathtt{lin}}$: It does not admit constant update time and enumeration delay *after linear time preprocessing*. As we show in Section 8, the queries Q_4, Q_5, and Q_6 from Figure 2 are not in $\mathcal{C}_{\mathtt{exp}}$, and we do not know whether they are tractable.

The 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 materialised 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. Hence, the update time is linear. Section 6 sketches our evaluation strategy for the queries in $\mathcal{C}_{\texttt{exp}}$.

Due to lack of space, the proofs of the formal statements in Sections 5–7 are deferred to the extended technical report [14].

2 Preliminaries

For a natural number n, we define $[n] = \{1, 2, \ldots, n\}$. In case $n = 0$, we have $[n] = \emptyset$.

Databases with Static and Dynamic Relations

Following standard terminology, a relation is a finite set of tuples and a database is a finite set of relations [1]. The size $|R|$ of a relation R is the number of its tuples. The size $|D|$ of a database D is given by the sum of the sizes of its relations. The relation R is *dynamic* if it is subject to updates; otherwise, it is *static*. To emphasize that R is static or dynamic, we write R^s or respectively R^d.

Conjunctive Queries

A conjunctive query (CQ or query for short) has 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 Q, R_i, and S_j are *relation symbols*; $R_i^d(\boldsymbol{X}_i)$ are dynamic body atoms; $S_j^s(\boldsymbol{Y}_j)$ are static body atoms; $Q(\boldsymbol{X})$ is the head atom; $vars(Q) = \bigcup_{i\in[k]} \boldsymbol{X}_i \cup \bigcup_{j\in[\ell]} \boldsymbol{Y}_j$ is the set of variables of Q; $free(Q) = \boldsymbol{X} \subseteq vars(Q)$ is the set of *free* variables; $vars(Q) \setminus free(Q)$ is the set of *bound* variables; $atoms(Q)$ is the set of body atoms; $atoms(X)$ is the set of body atoms with variable X. The static (dynamic) sub-query $stat(Q)$ ($dyn(Q)$) is obtained from Q as follows: Its body is the conjunction of all static (dynamic) atoms of Q and its free variables are the free variables of Q that appear in static (dynamic) atoms of Q. We visualise queries as hypergraphs where nodes are query variables (with the free variables 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 query is *α-acyclic* if we can construct a tree, called join tree, such that: (1) the nodes of the tree are the body atoms of the query, and (2) for each variable, the following holds: if the variable appears in two body atoms, then it appears in all body atoms on the unique path between these two body atoms in the tree [8]. A query is *free-connex α-acyclic* if it is α-acyclic and remains α-acyclic after adding the head atom $Q(free(Q))$ to its body [7].

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

A *path* in a query Q is a sequence $(X_1, \ldots, X_n)$ of variables in Q such that each variable appears at most once in the sequence and each two adjacent variables X_i and X_{i+1} are contained in a body atom from Q, $\forall i \in [n-1]$. We use set operations on paths as on sets of their variables. Two variables X_1 and X_n are *connected* if there is a path $(X_1, \ldots, X_n)$. Two atoms $R(\boldsymbol{Y})$ and $S(\boldsymbol{Z})$ are *connected* if there are connected variables $X_1 \in \boldsymbol{Y}$ and $X_n \in \boldsymbol{Z}$. A set $\mathcal{C}$ of body atoms in Q is 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 $Q_1(A, B, C) = R^d(A, D), S^d(A, B), T^s(B, C)$ from Figure 2 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: Given a query Q and a database with static and dynamic relations, the problem is to maintain the query result under updates to the dynamic relations and to enumerate the tuples in the query result.

A single-tuple update to a relation R is an insert of a tuple into R or a delete of a tuple from R. We denote the insert of a tuple $\boldsymbol{t}$ by $+\boldsymbol{t}$ and the delete of $\boldsymbol{t}$ by $-\boldsymbol{t}$. In this paper, we consider the set semantics, where a tuple is in or out of the database; in particular, we do not consider the setting, where each tuple is associated with a multiplicity that may be positive or negative as in prior work, e.g., [22, 16].

Following prior work, e.g., [4, 16, 19], 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 input database and data structure under a single-tuple insert or delete. 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 consider the RAM model of computation. We assume that each relation R 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 set $\boldsymbol{S} \subsetneq \boldsymbol{X}$, where $\boldsymbol{X}$ is the set of attributes of R, we use an index data structure that, for any tuple $\mathbf{t}$ over the attributes in $\boldsymbol{S}$, 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. We further require the following assumption in Section 6: Given a relation T, whose size is polynomial in the input database size N, we can lookup for any tuple in T in constant time.

We report time complexities as functions of the database size N under data complexity (so 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.

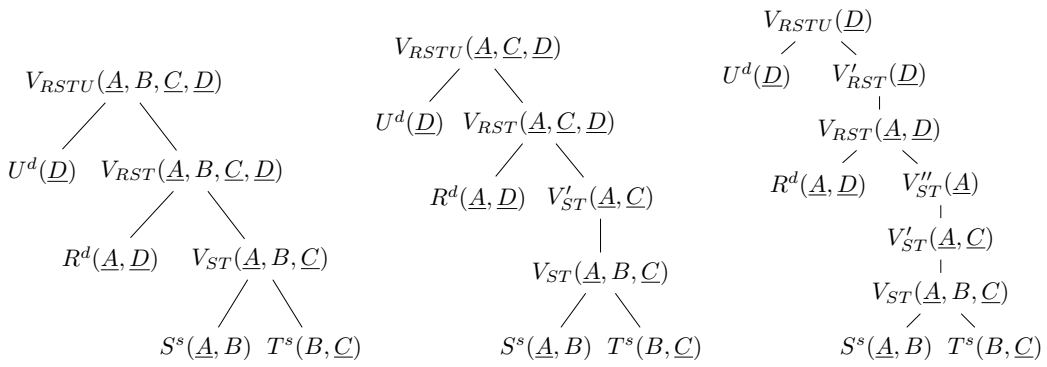


Figure 3 Three rewritings of the query $Q_2(A, C, D) = R^d(A, D), S^s(A, B), T^s(B, C), U^d(D)$ from Figure 2. The first two rewritings are not safe, while the last one is safe.

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, 20]. In this paper, we use so-called *safe* query rewritings, which ensure tractable dynamic evaluation.

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

- If V has a single child, then V is a projection of the child onto 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 conjunctive query 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 set of free variables of each dynamic view includes the set of free variables of each of its sibling views.*

Enumeration *Each tree T_i has a connected set $\mathcal{V}_i$ of views in T_i that contains the root of T_i such that $\bigcup_{i\in[n]} \bigcup_{V(\boldsymbol{X})\in\mathcal{V}_i} \boldsymbol{X} = free(Q)$.*

The *materialisation time* for a query rewriting is the time to materialise all its views (including both static and dynamic views).

▶ **Proposition 11.** *Let a conjunctive query Q and a database of size N. If Q has a safe rewriting with $f(N)$ materialisation 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. 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 set of free variables of the dynamic view $R^d(A, D)$ does not include the set of free variables 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 an 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 result 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 $\mathcal{C}_{\texttt{lin}}$, $\mathcal{C}_{\texttt{poly}}$, and $\mathcal{C}_{\texttt{exp}}$. We first define syntactic properties of the classes $\mathcal{C}_{\texttt{lin}}$ and $\mathcal{C}_{\texttt{poly}}$ that guarantee the existence of safe rewritings using views. The class $\mathcal{C}_{\texttt{exp}}$ contains queries that do not satisfy these properties.

▶ **Definition 13.** *A path $\boldsymbol{P}$ connecting two atoms $R(\boldsymbol{X})$ and $S(\boldsymbol{Y})$ in a conjunctive query Q is* safe *if $\boldsymbol{P} \cap \boldsymbol{X} \cap \boldsymbol{Y} \neq \emptyset$. In particular, the path is* body-safe *if the two atoms are body atoms and it is* head-safe *if one atom is a body atom and the other atom is the head atom. A conjunctive query Q is* well-behaved *if: (1) all paths connecting dynamic body atoms are body-safe; and (2) all paths connecting a dynamic body atom with the head atom are head-safe.*

▶ **Example 14.** The queries Q_3–Q_6 from Figure 2 are not well-behaved. The path (A, B) in $Q_3(A, B, C) = R^d(A), S^s(A, B), T^d(B)$ connects the two dynamic body atoms $R^d(A)$ and $T^d(B)$, but it is not body-safe, since $\{A, B\} \cap \{A\} \cap \{B\} = \emptyset$. The path (B, C) in $Q_4(A, B, C) = R^d(A, B), S^d(A, C), T^s(B, C)$ connects the two dynamic body atoms $R^d(A, B)$ and $S^d(A, C)$, but the path is not body-safe, since $\{B, C\} \cap \{A, B\} \cap \{A, C\} = \emptyset$. The path (A, B) in $Q_5(B, C) = R^d(A, B), S^d(A, C), T^s(B, C)$ connects the dynamic body atom $S^d(A, C)$ with the head atom $Q_5(B, C)$, but it is not head-safe, since $\{A, B\} \cap \{A, C\} \cap \{B, C\} = \emptyset$. The path (A, B) in $Q_6(A, B) = R^d(A), S^d(A, B), T^d(B, C), U^s(C)$ connects the two dynamic body atoms $R^d(A)$ and $T^d(B, C)$, but it is not body-safe, since $\{A, B\} \cap \{A\} \cap \{B, C\} = \emptyset$.

We can check efficiently whether a query is well-behaved.

▶ **Proposition 15.** *For any conjunctive query Q with n variables and m atoms, we can decide in $O(n^2 \cdot m^2)$ time whether Q is well-behaved.*

The well-behavedness property of a query implies the q-hierarchical property of its dynamic sub-query.

▶ **Proposition 16.**

- *Any conjunctive query without static relations is q-hierarchical if and only if it is well-behaved.*
- *The dynamic sub-query of any well-behaved conjunctive query is q-hierarchical.*

In Section 5, we show that every well-behaved query admits a safe rewriting, that is, it can be rewritten into a forest of view trees 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 17** ($\mathcal{C}_{\texttt{lin}}$). *A conjunctive query is in $\mathcal{C}_{lin}$ if it is free-connex α-acyclic and well-behaved.*

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

▶ **Definition 18** ($\mathcal{C}_{\texttt{poly}}$). *A conjunctive query is in $\mathcal{C}_{poly}$ if it is well-behaved.*

▶ **Example 19.** Consider the queries from Figure 2. The query Q_1 is in $\mathcal{C}_{\texttt{lin}}$, since it is free-connex α-acyclic and well-behaved. The query Q_2 is well-behaved but not free-connex α-acyclic, 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 are not well-behaved, as explained in Example 14.

We now define our most permissive class of tractable queries:

▶ **Definition 20** ($\mathcal{C}_{\texttt{exp}}$). *A conjunctive query is in $\mathcal{C}_{exp}$ if it is well-behaved or every variable occurring in a dynamic atom also occurs in a static atom.*

▶ **Example 21.** The query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ in Figure 2 is not well-behaved but is in $\mathcal{C}_{\texttt{exp}}$, since the variables of the two dynamic atoms $R^d(A)$ and $T^d(B)$ also occur in the static atom $S^s(A, B)$.

The queries Q_4–Q_6 are not contained in $\mathcal{C}_{\texttt{exp}}$, since they are not well-behaved (as explained in Example 14) and have variables in dynamic atoms that do not occur in static atoms: The variable A in Q_4 and Q_5 does not appear in the only static atom $S^d(B, C)$, and the variables A and B in Q_6 do not appear in the only static atom $U^s(C)$.

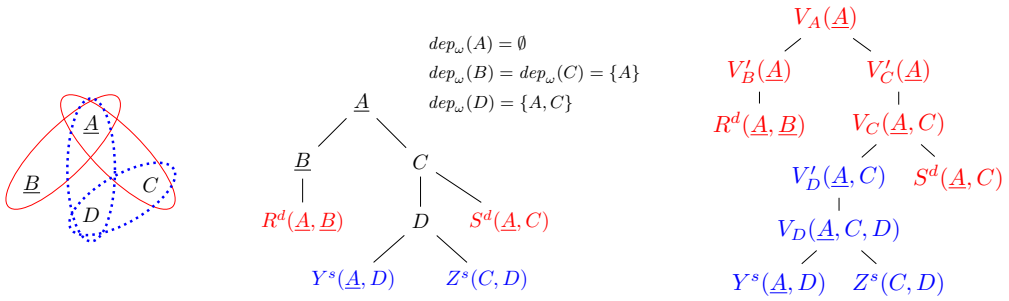


Figure 4 (left to right) The hypergraph of $Q(A, B) = R^d(A, B), S^d(A, C), Y^s(A, D), Z^s(C, D)$, a well-structured VO ω for Q, and the view tree for Q constructed by the procedure REWRITE(ω) from Figure 5. Dynamic views are in red, static views are in blue. The query Q is well-behaved and the preprocessing width is 2, thus $Q \in \mathcal{C}_{\texttt{poly}}$.

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

This section introduces our evaluation strategy for the $\mathcal{C}_{\texttt{poly}}$ and $\mathcal{C}_{\texttt{lin}}$ queries. Section 5.1 introduces variable orders, which guide the construction of safe rewritings for such queries, and the preprocessing width w of a query based on its variable orders. Section 5.2 shows that the construction of a safe rewriting for queries in $\mathcal{C}_{\texttt{poly}}$ can be done in $\mathcal{O}(N^{\mathsf{w}})$ time, where N is the size of the input database.

5.1 Variable Orders

We say that two variables in a query are *dependent* if they appear in the same body 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) There is a one-to-one mapping between the nodes of ω and the variables in Q; (2) the variables of each body 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: $dep_\omega(X) = \{Y | Y \text{ is an ancestor of } X \text{ and } \exists Z \in \omega_X \text{ s.t. } Y \text{ and } Z \text{ are dependent}\}$. We further extend a VO by adding each body atom of the query as the child of the lowest variable in the VO that belongs to the atom.

We adapt the notion of canonical VOs from the all-dynamic setting [20] to the setting with both static and dynamic relations. A VO is *canonical* if the set of variables of each dynamic body atom A is the set of inner nodes of the root-to-leaf path that ends with A. It is *free-top* if no bound variable is an ancestor of a free variable. It is *well-structured* if it is canonical and free-top. We denote by $\mathsf{WVO}(Q)$ the set of well-structured VOs of Q.

▶ **Example 22.** The query Q from Figure 4 is in $\mathcal{C}_{\texttt{poly}}$ as it is well-behaved. It is not in $\mathcal{C}_{\texttt{lin}}$ as it is not free-connex α-acyclic. The dynamic sub-query $Q^d(A, B) = R^d(A, B), S^d(A, C)$ is q-hierarchical, per Proposition 16. The figure shows a well-structured VO for the query.

The following proposition generalises the above example to all well-behaved queries:

▶ **Proposition 23.** *Any well-behaved conjunctive query has a well-structured VO.*

Next, we define the *preprocessing width* w of a VO ω and a query Q in $\mathcal{C}_{\texttt{poly}}$. For a variable X in a VO ω, let Q_X denote the query, whose body is the conjunction of all atoms in ω_X and whose free variables are all its variables. The preprocessing width of ω is defined next using the fractional edge cover number[1] for such queries Q_X, whereas the preprocessing width of a query is the minimum over the preprocessing widths of its well-structured VOs:

$$\mathsf{w}(\omega) = \max_{X \in vars(\omega)} \rho^*_{Q_X}(\{X\} \cup dep_\omega(X)) \qquad \text{and} \qquad \mathsf{w}(Q) = \min_{\omega \in \mathsf{WVO}(Q)} \mathsf{w}(\omega)$$

▶ **Example 24.** Figure 4 depicts a well-structured VO ω for the query Q. We have $dep_\omega(D) = \{A, C\}$ and $\rho^*_{Q_D}(\{A, C, D\}) = 2$. Overall, the preprocessing width of Q is 2.

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

▶ **Proposition 25.** *For any conjunctive query Q in $\mathcal{C}_{lin}$, it holds* $\mathsf{w}(Q) = 1$.

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 to maintain queries in the all-dynamic setting [18, 19]. We adapt the view tree construction to the setting over both 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 using views following a top-down traversal of ω. Initially, the parameter VO ν is ω. If ν is a set of trees, the procedure creates a view tree for each tree in ν (Line 1). If ν is a single atom, the procedure returns the atom (Line 2). If ν is a single tree with root X, it proceeds as follows. If X has only one child and this child is an atom, then the procedure returns the atom (Line 3). Otherwise, X has several child views. In this case, the procedure creates a join view V_X with free variables $\{X\} \cup dep_\omega(X)$. If X has a parent node, the procedure also adds on top of the join view V_X a projection view V'_X that projects away X (Line 8).

▶ **Example 26.** Figure 4 shows a well-structured VO ω for the query Q and the view tree constructed from ω using the procedure Rewrite. Observe that we obtain the view tree from ω 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 from Definition 10. The set of free variables of each dynamic view includes the set of free variables of each of its siblings; for instance, the sibling views $V'_C(A)$ and $V'_B(A)$ have the same set of free variables. 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 only need linear time to compute semi-joins or project away a variable. Thus, the materialisation time for this rewriting is quadratic. By Proposition 11, this query thus admits constant update time and constant enumeration delay after quadratic preprocessing time.

Given a well-structured VO ω for a query in $\mathcal{C}_{\texttt{poly}}$, the procedure Rewrite outputs a safe rewriting for Q whose materialisation time is a function of the preprocessing width of ω.

[1] The fractional edge cover number is used to define an upper bound on both the size of the query result and the time to compute it (Appendix A).

REWRITE(VO ν) : rewriting using views

switch ν:

$\{\nu_i\}_{i\in[n]}$ — 1 **return** $\{\,\text{REWRITE}(\nu_i)\,\}_{i\in[n]}$

$R(\boldsymbol{Y})$ — 2 **return** $R(\boldsymbol{Y})$

X / $\nu_1 \ \ldots \ \nu_k$

3 **if** $k = 1$ and ω_1 is atom $R(\boldsymbol{Y})$ **return** $R(\boldsymbol{Y})$

4 **let** $T_i = \text{REWRITE}(\nu_i)$ 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 $V_1(\boldsymbol{S}_1), \ldots, V_k(\boldsymbol{S}_k)$

7 **let** $V'_X(\boldsymbol{S} \setminus \{X\}) = V_X(\boldsymbol{S})$

8 **return**
- $V_X(\boldsymbol{S})$ over $T_1 \ \cdots \ T_k$ — if X has no parent in ω
- $V'_X(\boldsymbol{S} \setminus \{X\})$ over $V_X(\boldsymbol{S})$ over $T_1 \ \cdots \ T_k$ — otherwise

Figure 5 Rewriting a query using views following its VO ω by calling REWRITE(ω).

▶ **Proposition 27.** *For any conjunctive query Q in $\mathcal{C}_{\texttt{poly}}$, VO ω in $\mathsf{WVO}(Q)$, and database of size N, REWRITE(ω) is a safe rewriting for Q with $\mathcal{O}(N^{\mathsf{w}})$ materialisation time, where N is the database size and w is the preprocessing width of ω.*

Using a VO ω from $\mathsf{WVO}(Q)$ with $\mathsf{w}(\omega) = \mathsf{w}(Q)$, Proposition 27 immediately implies Theorem 3. Together with Proposition 25, which states that the queries in $\mathcal{C}_{\texttt{lin}}$ have preprocessing width 1, Proposition 27 also implies the complexity upper bound in Theorem 1.

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

In this section, we introduce our evaluation strategy for queries in $\mathcal{C}_{\texttt{exp}}$, specifically targeting those in $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$ for which the approach from previous sections is not applicable. We begin by demonstrating our strategy with a simple query from this class.

▶ **Example 28.** 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. However, the result of Q_3 is always a subset of the static relation S^s. With additional preprocessing time, we can precompute the effect of each update: We can construct a transition system whose states are the possible subsets of S^s and the updates to the dynamic relations may cause state transitions. In this system, each state enables the query result to be enumerated with constant delay, and transitioning between states occurs in constant time.

Each state corresponds to a database instance and the materialised query result for that instance. Since the static relations are the same across all states, the states only record the content of the query result and of the dynamic relations without any dangling tuples.

For clarity in this section, we interpret a database as a finite set of facts and apply standard set operations to databases. The static and dynamic parts of a database D, denoted D^s and D^d, consist of the facts from the static and dynamic relations in D, respectively, with $D = D^s \cup D^d$. Before explaining how to construct a transition system, we first define maximum dynamic databases.

▶ **Definition 29.** *Let a conjunctive query Q in $\mathcal{C}_{exp} \setminus \mathcal{C}_{poly}$ and a database $D = D^s \cup D^d$, where $D^d = \{R_1^d, \ldots, R_k^d\}$. For each dynamic body atom $R_i^d(\boldsymbol{X})$ in Q, let S_i^d be the result of the sub-query Q_{R_i} of Q whose body is the conjunction of the static body atoms of Q and whose free variables are $\boldsymbol{X}$. The* maximum dynamic database $D^d_{\max}$ *for query Q and database D is $D^d_{\max} = \{S_1^d, \ldots, S_k^d\}$.*

By the definition of $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$, each variable from a dynamic atom $R^d(\boldsymbol{X})$ appears in at least one static atom of Q_R. The sub-query Q_R identifies the complete set of R^d-tuples that might contribute to the result of Q; any other R^d-tuples have no impact on the result of Q.

▶ **Example 30.** Consider the query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$ and the database D from Figure 6 (left). The maximum dynamic database is obtained by evaluating the queries $Q_R(A) = S^s(A, B)$ and $Q_T(B) = S^s(A, B)$ on D. As a result, the maximum dynamic database for query Q_3 and database D consists of $R^d = \{a_1\}$ and $T^d = \{b_1, b_2\}$.

We next establish the upper bound on the size of a maximum dynamic database.

▶ **Proposition 31.** *For any conjunctive query Q in $\mathcal{C}_{exp} \setminus \mathcal{C}_{poly}$ and database D of size N, the maximum dynamic database $D^d_{\max}$ for query Q and database D has $\mathcal{O}(N^{\rho^*(stat(Q))})$ size.*

We next define the transition system for a query in $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$ and a database D.

▶ **Definition 32.** *Consider a conjunctive query Q in $\mathcal{C}_{exp} \setminus \mathcal{C}_{poly}$ and a database $D = D^s \cup D^d$. Let $D^d_{\max}$ denote the maximum dynamic database for query Q and database D.*

A transition state for query Q and database D is a pair (I, R), where $I \subseteq D^d_{\max}$ and R is the result of Q on $D^s \cup I$. The set $\mathcal{S}$ of transition states for query Q and database D is: $\mathcal{S} = \{\, (I, R) \mid I \subseteq D^d_{\max}, R = Q(D^s \cup I) \,\}$.

A transition system for query Q and database D is a tuple $(\mathcal{S}, s_{init}, \mathcal{U}, \delta)$, where $\mathcal{S}$ is the set of transition states for Q and D, $s_{init} \in \mathcal{S}$ is the initial state, $\mathcal{U}$ is the set of single-tuple updates to D^d, and $\delta : \mathcal{S} \times \mathcal{U} \to \mathcal{S}$ is a transition function that determines the next state based on the current state and an update. The initial state s_{init} pairs the database $D^d \cap D^d_{\max}$ and the result of Q on $D^s \cup (D^d \cap D^d_{\max})$.

We construct this transition system during the preprocessing step. For each single-tuple update to a dynamic relation, the transition system is used to move from the current state to another. If the update does not belong to the maximum dynamic database, it has no effect, and the system remains in the same state. The query result is then enumerated from the current state. Next, we demonstrate our evaluation strategy for the query Q_3.

▶ **Example 33.** Figure 6 shows the states and the transition system constructed for the query Q_3 and the database $D = \{R^d, T^d, S^s\}$. The maximum dynamic database $D^d_{\max}$ for Q_3 and D, consisting of $R^d = \{a_1\}$ and $T^d = \{b_1, b_2\}$, has 8 possible subsets. Each state corresponds to one of these subsets and the result of Q_3 evaluated on that subset and the static relation S^s.

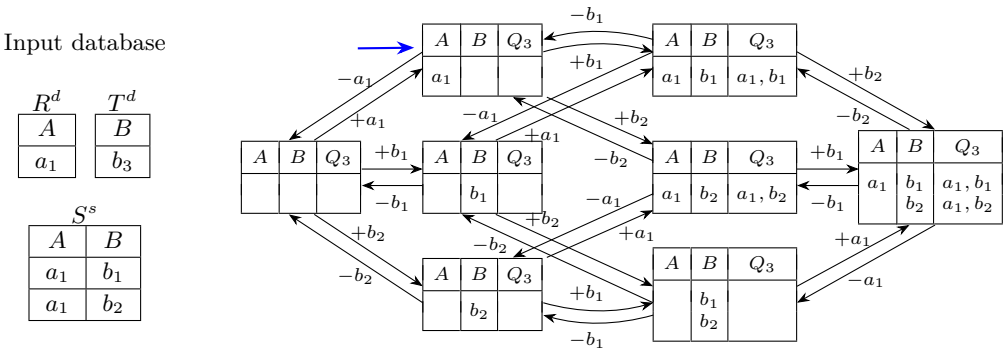


Figure 6 Left: Input database with dynamic relations $R^d(A)$ and $T^d(B)$ and static relation $S^s(A, B)$. Right: The transition system built from the input database and the query $Q_3(A, B) = R^d(A), S^s(A, B), T^d(B)$. Each state (box) consists of a dynamic database (first two columns), showing the possible contents of R^d and T^d, restricted to the A- and B-values from S^s, respectively; and the materialised result of Q_3 on that dynamic database and S^s (third column). The initial state is derived from the input database (blue arrow). Each transition (arrow) denotes an insertion $(+a)$ or deletion $(-a)$ of an A-value a in R^d, or an insertion $(+b)$ or deletion $(-b)$ of a B-value b in T^d.

The initial state s_{init}, denoted by the blue arrow, corresponds to the database $D^d \cap D^d_{max}$, which includes $R^d = \{a_1\}$ and $T^d = \emptyset$, along with the associated empty query result.

The state transitions correspond to single-tuple updates (inserts or deletes) to the dynamic relations R^d and T^d. Only updates involving tuples from $D^d_{\max}$ are relevant to the result of Q_3, specifically updates involving the value a_1 in R^d and the values b_1 and b_2 in S^d. All other updates have no effect on the query result because the updated values do not appear in S^s. For simplicity, we show only the state transitions for relevant updates in Figure 6.

For the same query and a database of size N, the size of $D^d_{\max}$ is the total number of distinct A- and B-values in S^s, thus $\mathcal{O}(N)$. The query result at any state is a subset of S^s and can be computed in $\mathcal{O}(N)$ time. The transition system captures all possible subsets of $D^d_{\max}$, thus it has $2^{|D^d_{\max}|}$ states in total. The number of transitions from any state is bounded by $2 \cdot |D^d_{\max}|$ when considering only updates relevant to the query result.

When constructing the transition system in this case, we create a global index for $\mathcal{O}(2^N)$ states, with a lookup time of $\mathcal{O}(N)$. For each state, we also build a local index to support transitions to $\mathcal{O}(N)$ neighbouring states. The local index is constructed by performing a lookup in $\mathcal{O}(N)$ time in the global index for each state transition. With $\mathcal{O}(N)$ possible transitions, constructing the local index takes $\mathcal{O}(N^2)$ time. The local index is defined over $\mathcal{O}(N)$ states and provides constant-time lookups, see the computational model in Section 2.

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 a 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.

Consider now an arbitrary query from $\mathcal{C}_{\texttt{exp}} \setminus \mathcal{C}_{\texttt{poly}}$ and a database D of size N. The constructed transition system consists of 2^p states, where $p = |D^d_{\max}| = \mathcal{O}(N^{\rho^*(stat(Q))})$, per Definition 32 and Proposition 31. For each state, the corresponding query result can be computed in $\mathcal{O}(N^{\rho^*(stat(Q))})$ time using a worst-case optimal join algorithm [24].

We next construct the transitions for each state. Each state has at most $2p$ transitions to other states when considering only updates relevant to the query result. The global index over the exponentially many states takes 2^p time to construct and requires time proportional to p for a lookup. For each state, it takes time quadratic in p to construct a local index that comprises at most $2p$ neighbouring states using the global index with lookup time proportional to p. Overall, constructing the transition system takes time $2^p \cdot p^2$, where $p = \mathcal{O}(N^{\rho^*(stat(Q))})$. This matches the complexity in Theorem 5.

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

In this section, we outline the proof of the lower bound in Theorem 1. The proof consists of two parts. In the first part, we give a lower bound on the complexity of evaluating 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 well-behaved and, therefore, not contained in $\mathcal{C}_{\mathtt{lin}}$: the first one has the path (A,B) that connects the dynamic body atoms $R^d(A)$ and $T^d(B)$ but is not body-safe; the second one has the path (B,A) that connects the dynamic body atom $T^d(B)$ with the head atom $Q_{ST}(A)$ but is not head-safe. In the second part of the proof, we show a lower bound on the complexity of evaluating arbitrary conjunctive queries that are not contained in $\mathcal{C}_{\mathtt{lin}}$ and do not have repeating relation symbols. The argument is as follows. Consider a conjunctive query $Q \notin \mathcal{C}_{\mathtt{lin}}$ that does not have repeating relation symbols. By definition of $\mathcal{C}_{\mathtt{lin}}$, this means that (1) Q is not free-connex α-acyclic, or (2) it has a path connecting two dynamic body atoms that is not body-safe, or (3) it has a path connecting a dynamic body atom with the head atom that is not head-safe. In Case (1), we cannot achieve constant-delay enumeration of the result after linear preprocessing time (even without processing any update), unless the Boolean Matrix Multiplication conjecture fails [3]. In Case (2), we reduce the evaluation of Q_{RST} to the evaluation of Q. In Case (3), we reduce the evaluation of Q_{ST} to the evaluation of Q. The latter two reductions transfer the lower bound for Q_{RST} and Q_{ST} to Q. In the following, we outline the lower bound proofs for Q_{RST} and Q_{ST} and defer further details to the technical report [14].

The lower bound for Q_{RST} is conditioned on the Online Vector-Matrix-Vector Multiplication (OuMv) conjecture, which is implied by the Online Matrix-Vector Multiplication (OMv) conjecture [12] (Appendix A). The proof of the lower bound is inspired by prior work, which reduces the OuMv problem to the evaluation of the variant of Q_{RST} where all relations are dynamic [4]. We explain the differences of our reduction to prior work in terms of construction and implication. The reduction in prior work 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}. Since the preprocessing procedure of the evaluation algorithm for Q_{RST} is executed on a non-empty database, we need to put a bound on the preprocessing time. Hence, while prior work establishes a lower bound on the update time and enumeration delay regardless of the preprocessing time, our reduction implies a lower bound on the combination of preprocessing time, update time, and enumeration delay:

▶ **Proposition 34.** *The conjunctive query $Q_{RST}() = R^d(A), S^s(A,B), T^d(B)$ 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. Assume that there is an algorithm $\mathcal{A}$ 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 $\mathcal{A}$ 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 OuMv 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 queries outside $\mathcal{C}_{\texttt{exp}}$. Let a conjunctive query Q. The *reduced dynamic sub-query* of Q is obtained from Q by omitting all static atoms and all of their variables.[2] 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 $Q_6(A, B) = R^d(A), S^d(A, B), T^d(B, C), U^s(C)$ from Figure 2. Its reduced dynamic sub-query is $Q_6'(A, B) = R^d(A), S^d(A, B), T^d(B)$. Any dynamic evaluation strategy for Q_6, where the variable C is fixed to an arbitrary constant, translates into a dynamic evaluation strategy for Q_6'. This means that any complexity lower bound for the dynamic evaluation of Q_6' is also a lower bound for Q_6.

A question is whether all queries, whose reduced dynamic sub-queries are q-hierarchical, are tractable. We discuss two such queries from Figure 2 that are not in $\mathcal{C}_{\texttt{exp}}$: $Q_4(A, B, C) = R^d(A, B), S^d(A, C), T^s(B, C)$ and $Q_5(B, C) = R^d(A, B), S^d(A, C), T^s(B, C)$. The evaluation strategy from Example 33 can be easily extended to Q_4. Similar to Example 33, we create a transition system where each state stores (B, C)-values 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. So Q_4 is tractable, albeit not in $\mathcal{C}_{\texttt{exp}}$.

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)-values. Filtering out duplicates can however incur a non-constant enumeration delay. Therefore, our approach for $\mathcal{C}_{\texttt{exp}}$ cannot evaluate Q_5 tractably.

[2] In contrast, the dynamic sub-query of Q as defined in Section 2 retains all variables that appear in at least one dynamic atom.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2 Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. doi:10.1137/110859440.

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

4 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017. doi:10.1145/3034786.3034789.

5 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD Queries under Updates on Bounded Degree Databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018. doi:10.1145/3232056.

6 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *ICDT*, pages 8:1–8:19, 2018. doi:10.4230/LIPICS.ICDT.2018.8.

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

8 Johann Brault-Baron. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.*, 49(3):54:1–54:26, 2016. doi:10.1145/2983573.

9 Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.*, 16(7):1601–1614, 2023. doi:10.14778/3587136.3587137.

10 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018. doi:10.1145/3212685.

11 Arnaud Durand and Etienne Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic*, 8(4):21, 2007. doi:10.1145/1276920.1276923.

12 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

13 Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017. doi:10.1145/3035918.3064027.

14 Ahmet Kara, Zheng Luo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Tractable Conjunctive Queries over Static and Dynamic Relations. *CoRR*, arXiv:2404.16224, 2024. doi:10.48550/arXiv.2404.16224.

15 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT*, pages 4:1–4:18, 2019. doi:10.4230/LIPICS.ICDT.2019.4.

16 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining Triangle Queries under Updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020. doi:10.1145/3396375.

17 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. In *PODS*, pages 375–392, 2020. doi:10.1145/3375395.3387646.

18 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive Queries with Free Access Patterns Under Updates. In *ICDT*, volume 255 of *LIPIcs*, pages 17:1–17:20, 2023. doi:10.4230/LIPICS.ICDT.2023.17.

19 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. F-IVM: Analytics over Relational Databases under Updates. *The VLDB Journal*, 2023. doi:10.1007/S00778-023-00817-W.

20 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *Log. Methods Comput. Sci.*, 19(3), 2023. doi:10.46298/LMCS-19(3:11)2023.

21 Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. Insert-Only versus Insert-Delete in Dynamic Query Evaluation. *Proc. ACM Manag. Data*, 2(5):219:1–219:26, 2024. doi:10.1145/3695837.

22 Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *The VLDB Journal*, 23(2):253–278, 2014. doi:10.1007/S00778-013-0348-4.

23 Dániel Marx. Approximating Fractional Hypertree Width. *ACM Trans. Algorithms*, 6(2):29:1–29:17, 2010. doi:10.1145/1721837.1721845.

24 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. doi:10.1145/3180143.

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

26 Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM TODS*, 40(1):2:1–2:44, 2015. doi:10.1145/2656335.

27 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. doi:10.2200/S00362ED1V01Y201105DTM016.

28 Yufei Tao and Ke Yi. Intersection Joins under Updates. *J. Comput. Syst. Sci.*, 124:41–64, 2022. doi:10.1016/J.JCSS.2021.09.004.

29 Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. Change Propagation Without Joins. *Proc. VLDB Endow.*, 16(5):1046–1058, 2023. doi:10.14778/3579075.3579080.

A Further Preliminaries

▶ **Definition 35** (Fractional Edge Cover [2]). *Given a conjunctive query Q and a set $\boldsymbol{F} \subseteq vars(Q)$, a* fractional edge cover *of Q for $\boldsymbol{F}$ is a solution $\boldsymbol{\lambda} = (\lambda_{R(\boldsymbol{X})})_{R(\boldsymbol{X}) \in atoms(Q)}$ to the following linear program:*

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

The optimal objective value of the above program is called the fractional edge cover number *of Q for the variable set $\boldsymbol{F}$ and denoted as $\rho^*_Q(\boldsymbol{F})$. We abbreviate $\rho^*_Q(vars(Q))$ by $\rho^*(Q)$.*

For a query Q without bound variables, a set $\boldsymbol{F} \subseteq vars(Q)$, and a database of size N, N^e with $e = \rho^*_Q(\boldsymbol{F})$ is an upper bound on the worst-case size of the result of Q projected onto the variables in $\boldsymbol{F}$ [2]. The result of Q can be computed in time $\mathcal{O}(N^{\rho^*(Q)})$ [24].

In this work, we establish complexity lower bounds based on the following widely accepted complexity-theoretic conjectures.

▶ **Definition 36** (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 multiplication Mv_i before seeing the next vector.*

▶ **Conjecture 37** (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 38** (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 multiplication $u_i M v_i$ before seeing the next pair.*

The following OuMv conjecture is implied by the OMv conjecture.

▶ **Conjecture 39** (OuMv Conjecture [12]). *For any constant $\gamma > 0$, there is no algorithm that solves OuMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

B Proof of Proposition 11

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

To enumerate the result of Q, we nest the enumeration procedures for the connected components of Q, concatenating their result 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 such that the root of T' is the same as the root of T and the set of free variables of the views in T' consists of 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 the variables in $\boldsymbol{Y}$, where $\boldsymbol{Y}$ is the set of free 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 result 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 result, 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 15

We first explain how we can check that every path connecting two dynamic body atoms in Q is body-safe. For each pair $(R(\boldsymbol{X}), S(\boldsymbol{Y}))$ of dynamic body atoms of Q, we first construct the Gaifman graph of the hypergraph of Q without the 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 in $\mathcal{O}(n^2)$ time if x is reachable from y using the Breadth-First Search algorithm; if so, Q has a path connecting $R(\boldsymbol{X})$ and $S(\boldsymbol{Y})$ that is not body-safe. We repeat this procedure for every pair of dynamic body atoms, which gives the total cost of $\mathcal{O}(n^2 \cdot m^2)$.

We now show how to check that every path connecting a dynamic body atom with the head atom is head-safe. For each dynamic body atom $R(\boldsymbol{X})$, we construct the Gaifman graph of Q without the free variables in $\boldsymbol{X}$. 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 path connecting $R(\boldsymbol{X})$ with the head atom that is not head-safe. The total time is $\mathcal{O}(n^2 \cdot m^2)$.

C.2 Proof of Proposition 16

We start with the proof of the second statement in Proposition 16. Consider a well-behaved query Q. For the sake of contradiction, assume that $\mathit{dyn}(Q)$ is not q-hierarchical. This means that one of the following two cases holds: (1) $\mathit{dyn}(Q)$ is not hierarchical or (2) $\mathit{dyn}(Q)$ is hierarchical but not q-hierarchical. We show that both cases lead to a contradition.

Assume that $\mathit{dyn}(Q)$ is not hierarchical. In the following, we denote by $\mathit{dynAtoms}(X)$ the set of dynamic body atoms in Q that contain the variable X. The query Q must have two variables X and Y such that $\mathit{dynAtoms}(X) \not\subseteq \mathit{dynAtoms}(Y)$, $\mathit{dynAtoms}(Y) \not\subseteq \mathit{dynAtoms}(X)$, and $\mathit{dynAtoms}(X) \cap \mathit{dynAtoms}(Y) \neq \emptyset$. This implies that Q has three dynamic body 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 $\boldsymbol{P}$ is not body-safe, which is a contradiction to our assumption that Q is well-behaved.

Assume now that $\mathit{dyn}(Q)$ is hierarchical, but not q-hierarchical. This implies that Q contains two dynamic body atoms $R^d(\boldsymbol{X})$ 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 dynamic body atom $R^d(\boldsymbol{X})$ with the head atom of Q such that $\boldsymbol{P} \cap \mathit{free}(Q) \cap \boldsymbol{X} = \emptyset$. This means that the path $\boldsymbol{P}$ is not head-safe, which is again a contradiction.

Next, we prove the first statement in Proposition 16. The "if"-direction follows directly from the second statement of the proposition. It remains to show the "only-if"-direction. We prove the contraposition of this direction. Consider a query Q without static relations that is not well-behaved. We show that Q is not q-hierarchical.

Assume that Q has a path $\boldsymbol{P} = (X_1, \ldots, X_n)$ that connects two body atoms $R^d(\boldsymbol{X})$ and $T^d(\boldsymbol{Z})$ such that $\boldsymbol{P} \cap \boldsymbol{X} \cap \boldsymbol{Y} = \emptyset$, i.e., $\boldsymbol{P}$ is not body-safe. Assume that $\boldsymbol{P}$ is the shortest such path. Since $\boldsymbol{P}$ is not body-safe, it must hold $n > 1$. Furthermore, every pair (X_i, X_{i+1}) with $i \in [n-1]$ is contained in a body atom i.e., $\mathit{atoms}(X_i) \cap \mathit{atoms}(X_{i+1}) \neq \emptyset$. The following claim implies that Q is not hierarchical, hence, not q-hierarchical:

Claim 1: $\exists i \in [n-1]$ *such that* $\mathit{atoms}(X_i) \not\subseteq \mathit{atoms}(X_{i+1})$ *and* $\mathit{atoms}(X_{i+1}) \not\subseteq \mathit{atoms}(X_i)$.

We prove Claim 1. Assume that for each $i \in [n-1]$, it holds $\mathit{atoms}(X_i) \subseteq \mathit{atoms}(X_{i+1})$ or $\mathit{atoms}(X_{i+1}) \subseteq \mathit{atoms}(X_i)$. Consider an $i \in [n-1]$ such that $\mathit{atoms}(X_i) \subseteq \mathit{atoms}(X_{i+1})$ (the case for $\mathit{atoms}(X_{i+1}) \subseteq \mathit{atoms}(X_i)$ is treated analogously). If $i = 1$, let $\boldsymbol{P'} = (X_{i+1}, \ldots, X_n)$. Otherwise, let $\boldsymbol{P'} = (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)$. The path $\boldsymbol{P'}$ is shorter than $\boldsymbol{P}$ and still not body-safe. This contradicts our assumption on the length of $\boldsymbol{P}$.

Assume now that Q has a path $\boldsymbol{P} = (X_1, \ldots, X_n)$ connecting a dynamic body atom $R^d(\boldsymbol{X})$ with the head atom of Q such that $\boldsymbol{P} \cap \boldsymbol{X} \cap \mathit{free}(Q) = \emptyset$, i.e., $\boldsymbol{P}$ is not head-safe. Let $\boldsymbol{P}$ be the shortest such path. Without loss of generality, we assume that Q is hierarchical. Since X_n is free, the following claim implies that Q is not q-hierarchical:

Claim 2: X_{n-1} *is bound and* $\mathit{atoms}(X_{n-1}) \supset \mathit{atoms}(X_n)$.

We prove Claim 2. Assume that X_{n-1} is free. Since $\boldsymbol{P}$ is not head-safe, X_{n-1} cannot be included in $\boldsymbol{X}$. Hence, $\boldsymbol{P'} = (X_1, \ldots, X_{n-1})$ is a path that is shorter than $\boldsymbol{P}$ and not head-safe, which contradicts our assumption on the length of $\boldsymbol{P}$. Assume now that

$atoms(X_{n-1}) \not\supset atoms(X_n)$. Since Q is hierarchical and there must be at least one body atom including both X_{n-1} and X_n, it must hold $atoms(X_{n-1}) \subseteq atoms(X_n)$. This implies that $n > 2$, since otherwise the free variable X_n is included in $\boldsymbol{X}$, which contradicts our assumption that $\boldsymbol{P}$ is not head-safe. This means however that the path $\boldsymbol{P}'' = (X_1, \ldots, X_{n-2}, X_n)$ is sorter than $\boldsymbol{P}$ and not head-safe, which contradicts our assumption on the length of $\boldsymbol{P}$.