Conjunctive Queries with Output Access Patterns under Updates

Ahmet Kara\textsuperscript{1}, Milos Nikolic\textsuperscript{2}, Dan Olteanu\textsuperscript{1}, Haozhe Zhang\textsuperscript{1}

\textsuperscript{1}University of Zurich \quad \textsuperscript{2}University of Edinburgh

Abstract

We study the dynamic evaluation of conjunctive queries with output access patterns. An access pattern is a partition of the free variables of the query into input and output. The query returns tuples over the output variables given a tuple over the input variables.

Our contribution is threefold. First, we give a syntactic characterisation of queries that admit constant time per single-tuple update and whose output tuples can be enumerated with constant delay given an input tuple. Second, we define a class of queries that admit optimal, albeit non-constant, update time and delay. Their optimality is predicated on the Online Matrix-Vector Multiplication conjecture. Third, we chart the complexity trade-off between preprocessing, update time and enumeration delay for such queries. Our results recover prior work on the dynamic evaluation of conjunctive queries without access patterns.

Acknowledgements. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 682588.

1 Introduction

In this paper we consider the problem of answering conjunctive queries with output access patterns under single-tuple updates (inserts and deletes) to the input database. Restricted access to data is commonplace:\cite{22, 23, 21}: For instance, the flight information behind a user interface (query) can only be accessed by providing values for specific input fields such as the departure and destination airports in a flight booking database. We formalise such restricted access by conjunctive queries with output access patterns (CQAP for short):\cite{9}: The free variables of a CQAP query are partitioned into \textit{input} and \textit{output}. The query yields tuples of values over the output variables \textit{given} a tuple of values over the input variables.

We address the following open questions: Which CQAP queries can be maintained with constant time per single-tuple update and with constant delay for the enumeration of the tuples over the output variables given a tuple of values over the input variables? Which CQAP queries admit optimal, albeit non-constant, update time and delay? What is the relationship between the complexities of the update time and the enumeration delay for CQAP queries?

Our answers to these questions draw inspiration from three prior works: (1) the characterisation of the conjunctive queries \textit{without access patterns} that admit constant-time update and delay \cite{6}; (2) the update-delay trade-off for the dynamic evaluation of hierarchical queries \textit{without access patterns} \cite{20}; and (3) the space-delay trade-off for the \textit{static} evaluation of full CQAP queries \cite{9}.

For the first question, we syntactically characterise the maximal class \textbf{CQAP\textsubscript{0}} of queries that admit constant-time update and delay.

\textbf{Theorem 1.} Let any CQAP query $Q$ and database of size $N$.

- If $Q$ is in \textbf{CQAP\textsubscript{0}}, then it admits $O(N)$ preprocessing time, $O(1)$ enumeration delay, and $O(1)$ update time for single-tuple updates.
• If \( Q \) is not in \( \text{CQAP}_0 \) and has no repeating relation symbols, then there is no algorithm that computes \( Q \) with arbitrary preprocessing time, \( \mathcal{O}(N^{\frac{1}{2}-\gamma}) \) enumeration delay, and \( \mathcal{O}(N^{\frac{1}{2}-\gamma}) \) amortized update time, for any \( \gamma > 0 \), unless the Online Matrix-vector conjecture fails.

This dichotomy generalises a similar dichotomy for \( q \)-hierarchical queries without access patterns [6], which are \( \text{CQAP}_0 \) queries where all free variables are output. The class \( \text{CQAP}_0 \) further contains queries with input variables. For instance, the query \( Q(A, D \mid B, C) = R(A, B), S(B, C), T(C, D) \) is not hierarchical but in \( \text{CQAP}_0 \). Note the use of the separator (|) between the output and the input variables. The query reads: Given a pair of values over the input variables \( B, C \), enumerate the distinct pairs of values over the output variables \( A, D \) such that the conjunction expressed in the query body holds. The class \( \text{CQAP}_0 \) trivially contains all queries, whose variables are free and whose join variables are input. The smallest queries that are not in \( \text{CQAP}_0 \) are: \( Q(A \mid \cdot) = R(A, B), S(B) \), where there is no input variable; \( Q(B \mid A) = R(A, B), S(B) \); \( Q(\cdot \mid A) = R(A, B), S(B) \), where there is no output variable; and \( Q(O \mid \cdot) = R(A), S(A, B), T(B) \), where all free variables \( O \subseteq \{A, B\} \) are output.

For the second question, we give a syntactic characterisation of the class \( \text{CQAP}_1 \) of queries whose complexity for both the update time and delay matches the lower bound \( \Omega(N^{\frac{1}{2}}) \) in Theorem 1. These queries can be computed optimally, albeit weakly Pareto: We cannot lower both the update time and delay complexities, yet the lower bound does not preclude lowering only one of them.

**Theorem 2.** Any \( \text{CQAP}_1 \) query \( Q \) admits \( \mathcal{O}(N^{1+\frac{w}{2}-\epsilon}) \) preprocessing time, \( \mathcal{O}(N^\frac{1}{2}) \) enumeration delay, and \( \mathcal{O}(N^{\frac{1}{2}}) \) amortized update time for single-tuple updates, where \( N \) is the database size and \( w \) is the static width of \( Q \).

The class \( \text{CQAP}_1 \) contains both acyclic and cyclic queries, e.g., the 4-cycle query \( Q(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D) \) with input variables \( B, D \) and output variables \( A, C \). Our notion of static width generalises the fractional hypertree width from conjunctive queries, as used in prior work [20], to \( \text{CQAP} \). The static width of the above 4-cycle query is 2. In contrast to \( \text{CQAP}_0 \), \( \text{CQAP}_1 \) is not maximal for the given complexities, e.g., counting triangles needs the same update time and delay \([17, 18]\).

For the third question, we chart the preprocessing time - update time - enumeration delay trade-off for the dynamic evaluation of a class of \( \text{CQAP} \) queries. We introduce a relaxation of \( \text{CQAP} \) queries called fracture to capture the nature of access patterns: Since we expect access by fixing values for the input variables, we can break the joins on the input variables by replacing each of their occurrences with fresh variables. This does not affect the access pattern as each fresh input variable can be set to the corresponding given input value. Yet this may lead to structurally simpler queries whose dynamic evaluation admits lower complexity.

**Theorem 3.** Let a \( \text{CQAP} \) query \( Q \) with static width \( w \) and dynamic width \( \delta \), a database of size \( N \), and \( \epsilon \in [0, 1] \). If \( Q \)'s fracture is hierarchical, then \( Q \) admits \( \mathcal{O}(N^{1+\frac{w-1}{2}-\epsilon}) \) preprocessing time, \( \mathcal{O}(N^{1-\epsilon}) \) enumeration delay, and \( \mathcal{O}(N^{\delta}) \) amortized update time for single-tuple updates.

The aforementioned three main results are obtained using one algorithm, which is parameterised by \( \epsilon \) as in Theorem 2. The upper bound in Theorem 1 is obtained by observing that \( \delta = 0 \) for \( \text{CQAP}_0 \) queries and by setting \( \epsilon = 1 \); for Theorem 2 the \( \text{CQAP}_1 \) queries have \( \delta = 1 \) and we set \( \epsilon = \frac{1}{2} \). Furthermore, our algorithm either recovers or has lower complexity than all prior approaches.

**Example 4.** Let us consider the aforementioned 4-cycle query:

\[
Q(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D). 
\]

Assume that all four relations have size \( N \). Then, the result of the 4-cycle join has size and can be computed in time \( \mathcal{O}(N^2) \).

The complexities for typical eager and lazy approaches can be recovered using our approach by setting \( \epsilon = 1 \) and respectively \( \epsilon = 0 \) (except for preprocessing in the lazy approach):

| Approach | Preprocessing | Update | Delay |
|----------|---------------|--------|-------|
| Eager    | \( \mathcal{O}(N^2) \) | \( \mathcal{O}(N) \) | \( \mathcal{O}(1) \) |
| Lazy     | \( \mathcal{O}(1) \) | \( \mathcal{O}(1) \) | \( \mathcal{O}(N) \) |
| Ours     | \( \mathcal{O}(N^{1+\epsilon}) \) | \( \mathcal{O}(N^\epsilon) \) | \( \mathcal{O}(N^{1-\epsilon}) \) |
Figure 1: Complexities for preprocessing time, update time, and enumeration delay for the query in Example 5 with different access patterns as a function of $\epsilon \in [0, 1]$.

The eager approach precomputes the initial output. On a single-tuple update, it eagerly computes the delta query obtained by fixing the variables of one relation to constants; this delta query can be done in linear time. It can then enumerate the pairs of values over \{A, C\} for any input pair of values over \{B, D\} with constant delay.

The lazy approach has no precomputation and only updates each relation, without propagating the changes to the query output. For enumeration, it first needs to calibrate the relations in the residual query $Q(A, C) = R(A, b), S(b, C), T(C, d), U(A, d)$ under a given pair of values $(b, d)$. This takes linear time. After that, it can enumerate the pairs of values over \{A, C\} with constant delay.

Consider now a sequence of $m$ updates, each followed by one access request to enumerate $k$ out of the maximum possible $O(N^2)$ pairs of values. This sequence takes time (excluding preprocessing) $O(m(N + k))$ in the eager and lazy approaches and $O(m(N^\epsilon + kN^{1-\epsilon}))$ in our general approach. Depending on the values of $m$ and $k$, we can tune our approach to minimise its complexity. For $1 \leq k < N$ and any $m$, our approach has consistently lower complexity than the lazy/eager approaches, while for $k \geq N$ and any $m$ it matches the complexity of the lazy/eager approaches. The complexity of processing the sequence of updates and access requests is shown in the next table for various values of $m$ and $k$ (only the exponents are shown by taking $\log N$ of the complexities):

| $\log_N k$ | 0  | 0.5 | 1   | 1.5 | 2   | 0  | 0.5 |
|-------------|----|-----|-----|-----|-----|----|-----|
| $\log_N m$ | 0  | 0.5 | 1   | 1.5 | 2   | 1  | 1   |
|             | 0.5| 1   | 1.25| 1.5 | 2   | 1.5| 1.5 |
|             | 1  | 1.5 | 1.75| 2   | 2.5 | 2  | 2   |

The middle five columns show the complexities for our general approach for various values of $k$. The last row states the value of $\epsilon$, for which the complexities in the same columns are obtained. The rightmost two columns show the complexities for the lazy/eager approaches for $\log_N k \in \{0, 0.5\}$ only. They are all higher than for our approach: Regardless of $m$, the complexity gap is $O(N^{0.5})$ for $\log_N k = 0$ (with $\epsilon = 0.5$) and $O(N^{0.25})$ for $\log_N k = 0.5$ (with $\epsilon = 0.75$). For $\log_N k \geq 1$, our approach defaults to the eager approach and achieves the lowest complexities for $\epsilon = 1$.

Our approach is sensitive to the access patterns: Its complexity may vary from constant-time update and delay to as much as explicitly computing the query output or to linear delay. This may already happen for acyclic queries, as shown in the next example.

**Example 5.** Consider the query

$$Q(O \mid T) = R(A, B, C), S(A, B, D), T(A, E).$$
The table below gives static \( w \) and dynamic widths \( \delta \) of the query for different access patterns \((O \mid I)\):

| \( O \)         | \( I \)         | \( w \) | \( \delta \) |
|----------------|----------------|--------|------------|
| \( \{A, B, C, D, E\} \) | \( \{\} \)    | 1      | 0          |
| \( \{\} \)    | \( \{A, B, C, D, E\} \) | 1      | 0          |
| \( \{B, C, D, E\} \) | \( \{A\} \)   | 1      | 0          |
| \( \{C, D, E\} \) | \( \{A, B\} \) | 1      | 0          |
| \( \{A, C, D, E\} \) | \( \{B\} \)   | 1      | 1          |
| \( \{A, C, D\} \)  | \( \{B, E\} \) | 2      | 1          |
| \( \{A, E\} \)    | \( \{B, C, D\} \) | 2      | 2          |
| \( \{A, B\} \)    | \( \{C, D, E\} \) | 3      | 2          |

These widths are the exponents of the preprocessing and respectively update time complexities of our algorithm for \( \epsilon = 1 \). When varying \( \epsilon \in [0, 1] \), these three complexities define the lines in the 3D plane in Figure [B].

As shown in Example [B], our approach is tuned for a particular query and access pattern. Yet the same data structure may be used to serve access requests for the same query but different access patterns. This may come however with larger complexity than if it were constructed specifically for these access patterns. Given the data structure for a query with access pattern \((O \mid I)\), it can also support the access patterns \((O \cup I) \cdot \cdot\) and \((I \cdot \cdot)\). Depending on the chosen maintenance strategy, it may also support access patterns for some subsets of input and/or output \( I' \subseteq I \) and \( O' \subseteq O \): \((O' \mid I), (I \cup O' \cdot \cdot)\), and \((I \setminus I') \cup O' \mid I)\). Beyond these, our approach may support arbitrary access patterns yet with linear enumeration delay, which is the largest possible in our setting and the golden standard for arbitrary acyclic queries [B].

2 Preliminaries

We introduce the data model and heavy/light partitioning of relations, as used in this paper. Further details are in Appendix [B].

**Data Model.** A schema \( \mathcal{X} = (X_1, \ldots, X_n) \) is a tuple of distinct variables. Each variable \( X_i \) has a discrete domain \( \text{Dom}(X_i) \). We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple \( x \) of values has schema \( \mathcal{X} = \text{Sch}(x) \) and is an element from \( \text{Dom}(\mathcal{X}) = \text{Dom}(X_1) \times \cdots \times \text{Dom}(X_n) \).

A relation \( R \) over schema \( \mathcal{X} \) is a function \( R : \text{Dom}(\mathcal{X}) \rightarrow \mathbb{Z} \) such that the multiplicity \( R(x) \) is non-zero for finitely many tuples \( x \). A tuple \( x \) is in \( R \), denoted by \( x \in R \), if \( R(x) \neq 0 \). The size \( |R| \) of \( R \) is the size of the set \( \{x \mid x \in R\} \). A database is a set of relations and has size given by the sum of the sizes of its relations.

Given a tuple \( x \) over schema \( \mathcal{X} \) and \( S \subseteq \mathcal{X} \), \( x[S] \) is the restriction of \( x \) onto \( S \). For a relation \( R \) over \( \mathcal{X} \), schema \( S \subseteq \mathcal{X} \), and tuple \( t \in \text{Dom}(S) \): \( \sigma_{S=t} R = \{ x \mid x \in R \land x[S] = t \} \) is the set of tuples in \( R \) that agree with \( t \) on the variables in \( S \); \( \pi_S R = \{ x[S] \mid x \in R \} \) is the set of restrictions of the tuples in \( R \) to the variables in \( S \).

We restrict multiplicities of tuples to be strictly positive. Multiplicity 0 means the tuple is not present. A single-tuple update to a relation \( R \) is expressed as \( \delta R = \{ x \rightarrow m \} \). It is an insert of the tuple \( x \) in \( R \) if the multiplicity \( m \) is strictly positive. It is a delete of the tuple \( x \) from \( R \) if its multiplicity \( m \) is negative. Such a delete is rejected if the existing multiplicity of \( x \) in \( R \) is less than \( |m| \).

**Partitioning.** We partition relations based on the frequencies of their values. For a database \( D \), relation \( R \in D \) over schema \( \mathcal{X} \), schema \( S \subseteq \mathcal{X} \), and threshold \( \theta \), the pair \((R_{S=L}, R_{S=L})\) is a partition of \( R \) on \( S \) with threshold \( \theta \) if it satisfies the conditions:

- (union) \( R(x) = R_{S=L}(x) + R_{S=H}(x) \) for \( x \in \text{Dom}(\mathcal{X}) \)
- (domain partition) \( \pi_S R_{S=L}(x) \cap \pi_S R_{S=H}(x) = \emptyset \)
- (heavy part) \( \forall t \in \pi_S R_{S=H}, \exists K \in D: |\sigma_{S=t} K| \geq \frac{1}{2} \theta \)
- (light part) \( \forall t \in \pi_S R_{S=L} \land \forall K \in D: |\sigma_{S=t} K| < \frac{1}{4} \theta \)
We call \((R^{S\ast_H}, R^{S\ast_L})\) a strict partition of \(R\) on \(S\) with threshold \(\theta\) if it satisfies the union and domain partition conditions and the following strict versions of the heavy and light part conditions:

- (strict heavy part) \(\forall t \in \pi_S R^{S\ast_H}, \exists K \in D: |\sigma_{S=t} K| \geq \theta\)
- (strict light part) \(\forall t \in \pi_S R^{S\ast_L} \text{ and } \forall K \in D: |\sigma_{S=t} K| < \theta\)

The relation \(R^{S\ast_H}\) is called heavy and the relation \(R^{S\ast_L}\) is called light on the partition key \(S\). Due to the domain partition, the relations \(R^{S\ast_H}\) and \(R^{S\ast_L}\) are disjoint. For \(|D| = N\) and a strict partition \((R^{S\ast_H}, R^{S\ast_L})\) of \(R\) on \(S\) with threshold \(\theta = N^\epsilon\) for \(\epsilon \in [0,1]\), we have: \(\forall t \in \pi_S R^{S\ast_L}: |\sigma_{S=t} L| < \theta = N^\epsilon; \) and \(|\pi_S R^{S\ast_H}| \leq N^{1-\epsilon}.

Given schemas \(S_1 \subset \ldots \subset S_n \subset \mathcal{X}\), an HL-signature \(\text{sig}\) for \(R\) is \(\{S_1 \rightarrow s_1, \ldots, S_n \rightarrow s_n\}\), where \(s_i \in \{H, L\}\) for \(i \in [n]\). The relation part \(R^{\text{sig}}\) is defined as \(\bigcap_{i \in [n]} R^{S_i \ast s_i}\). Without loss of information, we do not further partition a relation part on a schema \(S\) if it is already light on a strict subset of \(S\). Let for instance relation \(R\) with schema \((A, B, C)\). One possible partition of \(R\) consists of the relation parts with the HL-signatures \(\{A \rightarrow L\}, \{A \rightarrow H, AB \rightarrow L\}\), and \(\{A \rightarrow H, AB \rightarrow H\}\). The union of these relation parts constitutes relation \(R\).

### 3 Queries with Access Patterns

We introduce the classes of queries investigated in this paper along with several of their properties. A conjunctive query with output access patterns (CQAP for short) has the form

\[Q(\mathcal{O}|\mathcal{I}) = R_1(\mathcal{X}_1), \ldots, R_n(\mathcal{X}_n).\]

We denote by: \((R_i)_{i \in [n]}\) the relation symbols; \((R_i(\mathcal{X}_i))_{i \in [n]}\) the atoms; \(\text{vars}(Q) = \bigcup_{i \in [n]} \mathcal{X}_i\) the set of variables; \(\text{atoms}(X)\) the set of the atoms containing \(X\); \(\text{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}\) the set of all atoms; and \(\text{free}(Q) = \mathcal{O} \cup \mathcal{I} \subseteq \text{vars}(Q)\) the set of free variables, which are partitioned into input variables \(\mathcal{I}\) and output variables \(\mathcal{O}\). An empty set of input or output variables is denoted by a dot (·). Given a tuple \(i\) over \(\mathcal{I}\), \(Q\) returns \(\pi_0 \sigma_{\mathcal{I}=i} Q\). That is, \(Q\) returns each tuple \(o\) over the output variables such that the assignment \(i \circ o\) to the free variables satisfies the body of \(Q\). The result of \(Q\) for an input tuple \(i\) is denoted by \(Q(\mathcal{O}|i)\).

The hypergraph of \(Q\) is \(H = (\mathcal{V}, \mathcal{E})\), where \(\mathcal{V}\) is the set of variables in \(Q\) and the multiset \(\mathcal{E}\) is the set of hyperedges. The schema of each atom in \(Q\) is a hyperedge in \(\mathcal{E}\).

The fracture of a CQAP query \(Q\) is a CQAP query \(Q_{\downarrow}\), defined as follows. We first replace each occurrence of an input variable by a fresh variable. Then, we compute the connected components of the hypergraph of the modified query. Finally, we replace in each connected component of the modified query all new variables originating from the same input variable by one input variable.

Let \(\mathcal{V}\) be a set of variables in \(Q\). For variables \(A\) and \(B\), \(B\) dominates \(A\) if \(\text{atoms}(A) \subseteq \text{atoms}(B)\). The query \(Q\) is \(\mathcal{V}\)-dominant if for any two variables \(A\) and \(B\), it holds: \(A \in \mathcal{V}\) and \(B\) dominates \(A \Rightarrow B \in \mathcal{V}\). The query \(Q\) is almost \(\mathcal{V}\)-dominant if for any variable \(B \notin \mathcal{V}\) and for any atom \(R(\mathcal{X}) \in \text{atoms}(B)\), there is another atom \(S(\mathcal{Y}) \in \text{atoms}(B)\) such that \(\mathcal{X} \cup \mathcal{Y}\) cover all variables in \(\mathcal{V}\) dominated by \(B\); we also require that \(Q\) is not already \(\mathcal{V}\)-dominant. If \(\mathcal{V}\) is the set of free or input variables, we refer to these dominance notions as (almost) free- or input-dominant. A query \(Q\) is hierarchical if for any \(A,B \in \text{vars}(Q)\), either \(\text{atoms}(A) \subseteq \text{atoms}(B)\), \(\text{atoms}(B) \subseteq \text{atoms}(A)\), or \(\text{atoms}(B) \cap \text{atoms}(A) = \emptyset\).

**Example 6.** The query \(Q(F) = R(A), S(A, B), T(B)\) is non-hierarchical for any free variables \(F\): \(\text{atoms}(A) = \{R(A), S(A, B)\}\), \(\text{atoms}(B) = \{S(A, B), T(B)\}\), yet \(\text{atoms}(A) \not\subseteq \text{atoms}(B)\), \(\text{atoms}(B) \not\subseteq \text{atoms}(A)\), and \(\text{atoms}(A) \cap \text{atoms}(B) \neq \emptyset\).

The query \(Q(F) = S(A, B), T(B)\) is hierarchical. It is not \(\{A\}\)-dominant: \(B\) dominates \(A\) yet \(B \not\in \{A\}\). It is \(\{B\}\)-dominant: no other variable (in this case \(A\)) dominates \(B\). It is also \(\{A,B\}\)-dominant.

The query \(Q(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D)\) is \(\mathcal{V}\)-dominant for any \(\mathcal{V}\). Its fracture is \(Q_{\downarrow}(A, C \mid B_1, B_2, D_1, D_2) = R(A, B_1), S(B_2, C), T(C, D_1), U(A, D_2)\). Although \(Q\) is not hierarchical, its fracture \(Q_{\downarrow}\) is. The query \(Q_{\downarrow}\) is not \(\mathcal{V}\)-dominant for \(\mathcal{V} = \{B_1, B_2, D_1, D_2\}\): \(C\) dominates both \(B_2\) and \(D_1\).
and $A$ dominates both $B_1$ and $D_2$, yet $A$ and $C$ are not in $V$. The query $Q_1$ is however almost $V$-dominant: $A \not\in V$ and for any of its atoms $R(A, B_1)$ and $U(A, D_2)$, there is another atom $U(A, D_2)$ and respectively $R(A, B_1)$ such that both $R(A, B_1)$ and $U(A, D_2)$ cover the variables $B_1$ and $D_2$ dominated by $A$; a similar reasoning applies to $C$.

The fracture of $Q(A \mid B) = S(A, B), T(B)$ is $Q_1(A \mid B_1, B_2) = S(A, B_1), T(B_2)$. It is hierarchical and $\{B_1, B_2\}$-dominant. The fracture of $Q(B \mid A) = S(A, B), T(B)$ is the query itself. It is hierarchical, yet not $\{A\}$-dominant since $B$ dominates $A$ and is not in $\{A\}$. It is, however, almost $\{A\}$-dominant: for each atom of $B$, there is one other atom of $B$ such that together these atoms cover $A$. Indeed, atom $S(A, B)$ already covers $A$, and it also does so together with the atom $T(B)$; atom $T(B)$ does not cover $A$, but it does so together with $S(A, B)$.

We next define two query classes investigated in this work.

**Definition 7.** A query with input variables $I$ and output variables $O$ is in $\text{CQAP}_0$ if its fracture is hierarchical, $(O \cup I)$-dominant and $I$-dominant.

In case of no input variables, $\text{CQAP}_0$ corresponds precisely to the class of $q$-hierarchical queries [9]. The query in Example 8 is in $\text{CQAP}_0$ for all access patterns in the table for which the dynamic width $\delta$ is 0 (this is a general property, cf. Proposition 27).

**Definition 8.** A query with input variables $I$ and output variables $O$ is in $\text{CQAP}_1$ if its fracture is hierarchical and has at least one of the following two properties: (1) almost $(O \cup I)$-dominant; and (2) almost $I$-dominant.

**Example 9.** Section 4 gives the smallest three queries that are hierarchical but not in $\text{CQAP}_0$: $Q(A \mid \cdot) = R(A, B), S(B); Q(B \mid A) = R(A, B), S(B); Q(\cdot \mid A) = R(A, B), S(B)$. They are all in $\text{CQAP}_1$. Also, the 4-cycle query in Example 4 is in $\text{CQAP}_1$.

### 3.1 Variable Orders

We can define variable orders [24] for CQAP queries exactly as for conjunctive queries. Given a CQAP query, two variables depend on each other if they occur in the same atom of the query. A variable order $\omega$ for a conjunctive query $Q$ is a pair $(T_\omega, \text{dep}_\omega)$ where:

- $T_\omega$ is a forest with one node per variable or atom in $Q$. The variables of each atom in $Q$ lie along the same root-to-leaf path in $T$. Each atom is a child of its lowest variable.
- The function $\text{dep}_\omega$ maps each variable $X$ to the subset of its ancestor variables in $T$ on which the variables in the subtree rooted at $X$ depend.

The variable order $\omega$ is called canonical if the variables of the leaf atom of each root-to-leaf path are the inner nodes of the path. Hierarchical queries are precisely those conjunctive queries that admit canonical variable orders. The subtree of $\omega$ rooted at $X$ is denoted by $\omega_X$. The sets $\text{vars}(\omega)$, $\text{atoms}(\omega)$, and $\text{anc}_\omega(X)$ consist of all variables of $\omega$, the atoms at the leaves of $\omega$, and the variables on the path from $X$ to the root excluding $X$, respectively.

We introduce classes of variable orders for CQAP queries. A variable order $\omega$ is free-top if no bound variable is an ancestor of a free variable. It is input-top if no output variable is an ancestor of an input variable. The sets of free-top and input-top variable orders for $Q$ are denoted as free-top($Q$) and input-top($Q$), respectively. A variable order is called access-top if it is free-top and input-top:

$$\text{acc-top}(Q) = \text{free-top}(Q) \cap \text{input-top}(Q).$$

**Example 10.** The query $Q(B \mid A) = R(A, B), S(B)$ admits the following variable orders (in term notation; left is above right): $B - \{A - R(A, B), S(B)\}$, where $B$ has as children the variable $A$ and the atom $S(B)$ and $A$ has as child the atom $R(A, B)$. The dependency sets are $\text{dep}(B) = \emptyset$ and $\text{dep}(A) = \{B\}$. This variable
order is free-top, since both variables are free; it is not input-top, since the output variable B is on top of the input variable A. By swapping A and B in the order, it becomes input-top and then also access-top; the dependencies then become: \( \text{dep}(A) = \emptyset \) and \( \text{dep}(B) = \{A\} \).

The fracture of the 4-cycle query in Example 8 admits the access-top variable order consisting of two disconnected paths: \( B_1 - D_2 - A - \{R(A,B_1),U(A,D_2)\} \) and \( B_2 - D_1 - C - \{S(B_2,C),T(C,D_1)\} \), where the dependency sets are: \( \text{dep}(A) = \{B_1, D_2\} \), \( \text{dep}(D_2) = \{B_1\} \), \( \text{dep}(B_1) = \text{dep}(B_2) = \emptyset \), \( \text{dep}(C) = \{B_2, D_1\} \), and \( \text{dep}(D_1) = \{B_2\} \).

### 3.2 Width Measures

We define width measures for CQAP queries in direct analogy to the case of conjunctive queries.

Given a query \( Q \) and \( X \subseteq \text{vars}(Q) \), we denote by \( \rho^*(X) \) the fractional edge cover number of \( X \) in \( Q \). If \( Q \) is clear from the context, we skip it in the notation. The static and dynamic widths of a CQAP query \( Q \) are the respective widths for its fracture \( Q_t \) seen as a plain conjunctive query (so where we disregard the partition of the free variables into input and output). Definition 11 below is thus as in prior work [20], only adjusted to consider query fractures.

**Definition 11.** The static width of a CQAP query \( Q \) is

\[
\omega(Q) = \min_{\omega \in \text{acc-top}(Q_t)} \omega(\omega)
\]

\[
\omega(\omega) = \max_{X \in \text{vars}(Q_t)} \rho^*_Q(\{X\} \cup \text{dep}_\omega(X))
\]

The dynamic width of a CQAP query \( Q \) is

\[
\delta(Q) = \min_{\omega \in \text{acc-top}(Q_t)} \delta(\omega)
\]

\[
\delta(\omega) = \max_{X \in \text{vars}(Q_t)} \max_{\mathcal{Y} \in \text{atoms}(\omega \chi)} \rho^*_Q(\{X\} \cup \text{dep}_\omega(X) - \mathcal{Y})
\]

The fractional edge cover number \( \rho^*(X) \) defines the time needed to compute the (join) subquery induced by the variables \( X \subseteq \text{vars}(Q) \). In the language of fractional hypertree decompositions, \( X \) corresponds to a bag so \( \rho^*(X) \) defines the time to compute the relation represented by that bag. Since we only consider the restricted set of access-top variable orders, \( \rho^*(X) \) may be larger than the fractional hypertree width of the query \( Q_t \).

The dynamic width considers again the static width for the subqueries induced by \( X \), yet under a single-tuple update to any relation \( R \) whose schema contains the variable \( X \), which also means that \( R \) is a descendant of \( X \) in the variable order \( \omega \). Since such updates fix the variables \( \mathcal{Y} \) of \( R \) to constants, we can disregard \( \mathcal{Y} \) in the computation of \( \rho^*(X) \). Appendix C further expands on the width measures with examples and properties; in particular, a query is in CQAP if and only if its dynamic width is \( i \) (for \( i \in \{0,1\} \)).

### 4 Preprocessing

Our dynamic evaluation technique comprises three distinct, yet interdependent stages: preprocessing, updates and enumeration. This section addresses preprocessing, with the following two sections addressing updates and enumeration. Our technique is designed for CQAP queries with hierarchical fractures to support Theorems 1 and 3 in Section B. Whenever we refer to the query in the three stages, we mean the hierarchical fracture of the input CQAP query.

For preprocessing, we construct a succinct data structure that represents the result of the query over both the input and output variables using a set of materialised view trees. Each view tree, which is modeled on a specific variable order, represents a part of the result. This construction exploits the structure of the query and the degree of data values in base relations. We proceed in two steps. First, we construct a set of variable orders corresponding to evaluation strategies for different parts of the query result. Each such
The leaf atoms in $\omega$ remain unchanged. We annotate the leaf atoms in $\omega$ to each tree in the forest. For any variable $X$, we consider two possible variable orders. One variable order is $\omega$ that dominates an input variable. For each such variable, we consider two possible variable orders. One variable order is $\omega$ in which $\omega$ remains unchanged. We annotate the leaf atoms in $\omega$ by $(\{Y\} \cup \text{anc}_\omega(Y) \rightarrow H)$ to signal that $\omega$ will be used for the evaluation of $Q_Y$ over relation parts that are heavy on $\{Y\} \cup \text{anc}_\omega(Y)$, i.e., $Y$ and the path from $Y$ to the root of $\omega$. The second variable order is obtained from $\omega$ by turning $\omega_Y$ into an access-top variable order $\omega'_Y$; this restructuring may increase the static width. The leaf atoms in $\omega'_Y$ are annotated with $((\{Y\} \cup \text{anc}_\omega(Y) \rightarrow L)$ meaning that $\omega'_Y$ will be used for the evaluation over relations that are light on $\{Y\} \cup \text{anc}_\omega(Y)$.

In the second step, we construct from each variable order a view tree by mapping each variable $X$ in the variable order by a view whose free variables are $X$ and its dependency set and whose body is the join of the views created at the children of $X$. If $X$ has a sibling, we also put a view on top of $V_X$ that aggregates away $X$ to enable efficient updates coming from the siblings of $X$ (cf. Section 4). The materialisation of a view tree consists of the strict partitioning of the base relations following the heavy-light annotation the computation of the joins defined by the views. The view trees constructed for the evaluation of CQAP$_0$ queries or queries over heavy relation parts follow canonical variable orders, which means that they can be materialised in linear time. The view trees constructed for the evaluation of queries over light relation parts follow access-top variable orders. Using the degree constraints in base relations, each such view tree can be materialised in $O(N^{1+(\omega-1)\epsilon})$, where $\omega$ is the static width of the query.

We next showcase the view tree construction for a CQAP$_0$ query.

Example 13. Figure 2 shows the hypergraphs of the query

$$Q(B, C, D, E | A) = R(A, B, C), S(A, B, D), T(A, E)$$

and of its fracture

$$Q_1(B, C, D, E | A_1, A_2) = R(A_1, B, C), S(A_1, B, D), T(A_2, E).$$
The fractures is hierarchical, free-dominant and input-dominant. Hence, Q and Q₁ are in CQAP₀. Figure 5 depicts access-top canonical variable orders for the queries whose bodies are the two connected components of the hypergraph of Q₁, i.e., Q₁(B, C, D|A₁) = R(A₁, B, C), S(A₁, B, D) and Q₂(E|A₂) = T(A₂, E), and their corresponding view trees.

Each variable in the variable order is mapped to a view in the view tree, e.g., B is mapped to V₀(A₁, B), where \{B, A₁\} = \{B\} \cup \text{dep}(B). The views V₁, V₂ and V₃ are auxiliary views that allow for efficient maintenance under updates to R and S: they marginalise out one variable from their child views. The view V₂ is the intersection of V₁ and V₃. Hence all views can be computed in linear time.

We next exemplify preprocessing for a CQAP₁ query.

Example 14. Consider the CQAP₁ query

\[ Q(E, D|A, C) = R(A, B, C), S(A, B, D), T(A, E) \]

and its fracture

\[ Q₁(E, D|A₁, A₂, C) = R(A₁, B, C), S(A₁, B, D), T(A₂, E). \]

The hypergraphs of Q and Q₁ are the same as for the CQAP₀ query in Example 13, see Figure 3. We next explain the construction of the view tree for the connected component in Q₁ corresponding to the query Q₁(D|A₁, C) = R(A₁, B, C), S(A₁, B, D). In the canonical variable order of this connected component (Figure 3 left), the bound variable B dominates the free variables C and D. We strictly partition the relations...
$R$ and $S$ on $(A_1, B)$ with threshold $N^*$, where $N$ is the database size. To evaluate the join over the light relation parts, we turn the subtree in the canonical variable order rooted at $B$ into an access-top variable order and construct a view tree following this new variable order. The top row in Figure 4 shows the variable order and view tree constructed in the light case. We compute the view $V_B(A_1, B, C, D)$ in time $O(N^{1+\epsilon})$. For each $(a, b, c)$ in the light part $R^{A_1, B, C}(A_1, B, C)$ of $R$, we fetch the D-values in $S^{A_1, B, C}(A_1, B, D)$ that are paired with $(a, b)$. The iteration in $R^{A_1, B, C}(A_1, B, C)$ takes $O(N)$ time and for each $(a, b)$, there are at most $N^*$ D-values in $S^{A_1, B, C}(A_1, B, D)$. The views $V_D$, $V_C$, and $V_A$ result from $V_B$ by marginalising out one variable at a time. Overall, this takes $O(N^{1+\epsilon})$ time.

To evaluate the join over the heavy parts of $R$ and $S$, we construct a view tree following the canonical variable order of the connected component, as shown in the bottom row of Figure 4. The variable order and view tree are same as those built in Example 13 except that the leaf relations are the heavy parts of $R$ and $S$. As explained in Example 13, the view tree can be materialised in $O(N)$ time. Overall, the two view trees can be computed in $O(N^{1+\epsilon})$ time.

5 Enumeration

The preprocessing stage constructs view trees that represent the result of a CQAP query. In this section, we show how to enumerate the distinct output tuples given a given tuple of values over the input variables. We first discuss the enumeration for CQAP$_0$ queries and then the enumeration for hierarchical CQAP queries in general.

The enumeration relies on iterators with access patterns created over materialized views. We write $\text{it}_V(O|I)$ to denote a view iterator $\text{it}$ over a view $V$ with schema $\{O\} \cup I$, where $O$ is the output variable and $I$ is the context schema. The view iterator implements the standard iterator interface. The open(\text{ctx}) method initialises the iterator using the tuple \text{ctx} over $I$ as context. This method sets the range of the iterator to those $O$-values that are consistent with \text{ctx} in $V$, that is, either paired with or part of \text{ctx} in $V$. The next() method returns an $O$-value consistent with \text{ctx} in $V$. It returns EOF when the $O$-values in the range of the iterator are exhausted. The returned $O$-values are distinct. Both methods operate in constant time, as per our computational model (cf. Appendix B).

We enumerate tuples from the view trees constructed in the preprocessing stage. For each view tree, we create iterators over the views that correspond to the free variables in the variable order of that view tree. We organise the iterators into nested loops based on a pre-order traversal of the view tree. We open the iterators with values from their ancestor views as context, thus ensuring they enumerate only those values guaranteed to be in the query output.

The algorithm for constructing view iterators and its runtime analysis are in Appendix E.

Proposition 15. For any CQAP$_0$ query, its distinct output tuples given an input tuple can be enumerated with $O(1)$ delay.

Example 16. Figure 5 shows the enumeration procedure for the view tree from Figure 3 (second from left) for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$. We create the view iterators for this view tree top-down. At the root view $V_A$, we create $\text{it}_{V_A}(A_1|A_1)$ to check if a given input $A_1$-value exists in $V_{A_1}$. If exists, the iterator returns the same $A_1$-value, which then serves as the context for the iterators created below. The iterator $\text{it}_{V_B}(B|A_1)$ at view $V_B$ enumerates the $B$-values that are paired with a in $V_B$. Such $(A_1, B)$-values serve as the context for $\text{it}_{V_C}(C|A_1, B)$ and $\text{it}_{V_D}(D|A_1, B)$, which enumerate $C$- and respectively $D$-values. We skip creating iterators over auxiliary views $V'_C(A_1, B)$ and $V'_D(A_1, B)$ because we already have iterators for $A_1$ and $B$. The enumeration procedure returns EOF when all the iterators are exhausted, i.e., all tuples have been enumerated.

The time needed to fetch the next value from each iterator is $O(1)$; this is also the enumeration delay of the procedure.

Nesting view iterators, as in Figure 5, is valid when the context schema of each iterator is subsumed by the input variables of the query and the output variables of preceding iterators. The nesting order of
Example 18. Figure 6 shows the enumeration procedure for the view tree from Figure 4 (bottom-right), of a tuple) over schema $I$ serves to check if the given view iterator for each tuple in $ctx$ constant delay.

For any query in CQAP, the corresponding view trees follow access-top variable orders where the free variables are above the bound variables and the input variables are above the output variables. In that case, nesting view iterators according to the access-top variable orders is valid and allows enumeration with constant delay.

For queries not in CQAP, nesting view iterators may be invalid. Assume for instance that the variable $A_1$ is bound in the query from Example 18. The query remains hierarchical but not free-dominant. The view iterators that enumerate $B_1$, $C_1$, and $D_1$-values have $A_1$ in their context schemas, yet there is no iterator for $A_1$-values. We say that such iterators are unsupported.

**Generalised View Iterators.** To support the enumeration for non-CQAP queries, we generalise the above view iterators as follows. The context of a generalised view iterator $git_V (O | I)$ is a relation (instead of a tuple) over schema $I$. The $open(ctx)$ method takes as input a relation $ctx$ over $I$ and instantiates a view iterator for each tuple in $ctx$. The $next()$ method uses the union algorithm [11] to report only distinct $O$-values, with the delay linear in the size of $ctx$. For each reported $O$-value $o$, $next()$ also returns a relation $ctx_o \subseteq ctx$ over schema $I$ with the tuples that are paired with $o$ in $V$. If there are no such tuples in $V$, the method returns $(EOF, \emptyset)$. Appendix E gives more details on generalised view iterators.

**Proposition 17.** For any hierarchical CQAP query $Q$, database of size $N$, and $\epsilon \in [0,1]$, the distinct output tuples given an input tuple can be enumerated with $O(N^{1-\epsilon})$ delay.

**Example 18.** Figure 5 shows the enumeration procedure for the view tree from Figure 4 (bottom-right), created for the connected component $Q_1(D | A_1, C) = R(A_1, B, C), S(A_1, B, D)$.

We construct three generalised view iterators, one for each free variable. The iterator $git_{V_{A_1}} (A_1 | A_1)$ serves to check if the given $A_1$-value exists in $V_{A_1}$ (Lines 2-3). The iterator $git_{V_C} (C | A_1, B, C)$ is unsupported as there is no binding for variable $B$. For this iterator, we provide a relation over schema $(A_1, B, C)$ as context. To avoid enumerating dangling tuples, the context should include only those $B$-values guaranteed to have matching $D$-values in the final output. The ancestor view $V_B (A_1, B)$ provides such $(A_1, B)$-values, which we further restrict to those matching the given input values (Line 4). The $next()$ call on $git_{V_C}$ returns the input $C$-value together with a relation $ctx_c$ containing the matching $(A_1, B, C)$-tuples in $V_C$ if they exist; otherwise, it returns $(EOF, \emptyset)$. The relation $ctx_c$ serves as context for the iterator over $D$-values (Line 6).

The $open$ and $next$ calls take time linear in the size of the context $ctx$ used when opening the iterator. The size of the context for $git_{V_{A_1}}$ is constant, while for $git_{V_C}$ and $git_{V_D}$ is at most the size of $V_B$. Given that $V_B$ is over the heavy part $R^{A_1,B=H}$ of $R$ and the heavy part $S^{A_1,B=H}$ of $S$, the number of distinct $(A_1, B)$-values in $V_B$ is at most $N^{1-\epsilon}$. Thus, the enumeration delay is $O(N^{1-\epsilon})$. 

---

```plaintext
let ctx0 = input A1-value
itv_{A1} (A1|A1).open(ctx0)
while (a := itv_{A1} (A1|A1).next()) \neq EOF do
  itv_B (B|A1).open(a)
  while (b := itv_B (B|A1).next()) \neq EOF do
    itv_C (C|A1,B).open(a,b)
    while (c := itv_C (C|A1,B).next()) \neq EOF do
      itv_D (D|A1,B).open(a,b)
      while (d := itv_D (D|A1,B).next()) \neq EOF do
        output (b,c,d)
  output EOF
```
Figure 7 shows the delta view trees for the view trees in the right column of Figure 4 under Example 20. The blue views in the view trees are the deltas to the corresponding views, computed while propagating single-tuple update \( \delta_R \) (Equation 19). The runtime analysis are in Appendix F.

In this section we consider the problem of how to efficiently maintain the view trees computed in the preprocessing step under a single-tuple update \( \delta_R \) to any base relation \( R \). The update \( \delta_R = \{ x \rightarrow m \} \) maps the tuple \( x \) to the non-zero multiplicity \( m \in \mathbb{Z} \). Inserts and deletes are updates represented as relations in which tuples have positive and negative multiplicities, respectively.

Our approach to effect this update is as follows. We first identify which part of a relation \( R \) is affected by the update: We check the degrees of \( x \) among the keys on which \( R \) is partitioned and find the relation part \( R'^{sig} \) that has the matched degrees. Then, for each view tree that contains \( R'^{sig} \), we update \( R'^{sig} \) with \( \delta_R \) and propagate the change from the leaf \( R'^{sig} \) to the root view of the tree: We update each view on this path using the hierarchy of materialized views and the classical delta rule [7]. The update algorithm and its runtime analysis are in Appendix F.

**Proposition 19.** Given a hierarchical CQAP query \( Q(O|I) \) with dynamic width \( \delta \), a database of size \( N \), and \( \epsilon \in [0, 1] \), the view trees constructed in the preprocessing stage can be maintained under a single-tuple update to any input relation in \( \mathcal{O}(N^{\delta \epsilon}) \) amortised time.

**Example 20.** Figure 4 shows the delta view trees for the view trees in the right column of Figure 4 under the single-tuple update \( \delta_R = \{ (a, b, c) \rightarrow m \} \) to \( R \). The delta view trees for an update to \( S \) are analogous. The blue views in the view trees are the deltas to the corresponding views, computed while propagating \( \delta_R \) from the affected relation part to the root view. The update \( \delta_R \) affects the light part \( R^{sig}_{A_1, B_{sig}^{L}}(A_1, B) \) of \( R \) if the tuple \( (a, b, c) \) is light on the partition key \((A_1, B)\). In this case, we update the relation part \( R^{sig}_{A_1, B_{sig}^{L}}(A_1, B) \) with \( \delta R^{sig}_{A_1, B_{sig}^{L}}(a, b, c) = \delta R(a, b, c) \), and propagate the change up the tree. We update \( V_B(A_1, B, C, D) \) with \( \delta V_B(a, b, c, D) = \delta R^{sig}_{A_1, B_{sig}^{L}}(a, b, c), S^{sig}_{A_1, B_{sig}^{L}}(a, b, D) \) in \( \mathcal{O}(N) \) time, since there are at most \( N^\epsilon \) \( D \)-values paired with \( (a, b) \) in \( S^{sig}_{A_1, B_{sig}^{L}} \). We then update \( V_D(a, c, D) \) with \( \delta V_D(a, c, D) = \delta V_B(a, b, c, D) \) in \( \mathcal{O}(1) \) time and similarly for the views \( V_C(A_1, C) \) and \( V_A_1(A_1) \).

In case \( \delta_R \) affects the heavy part \( R^{sig}_{A, B_{sig}^{H}}(A_1, B, C) \), i.e., \( (a, b, c) \) is heavy on \((A_1, B)\), we update the view \( V_C(A_1, B, C) \) with \( \delta V_C(a, b, c) = \delta R^{sig}_{A, B_{sig}^{H}}(a, b, c) \) in \( \mathcal{O}(1) \) time and then update the other views \( V_C(A_1, B), V_B(A_1, B) \) and \( V_A_1 \) similarly in \( \mathcal{O}(1) \) time.
\[
\begin{align*}
\delta V_A(a) \\
\delta V_C(a, c) \\
\delta V_D(a, c, D) \\
\delta V_B(a, b, c, D) \\
\delta R^{A_1, B \rightarrow L}(a, b, c)
\end{align*}
\]

\[
\begin{align*}
\delta V_A(a) \\
\delta V_B(a, b) \\
\delta V_C(a, b, c) \\
\delta V_D(A_1, B, D) \\
\delta R^{A_1, B \rightarrow H}(A_1, B, D)
\end{align*}
\]

Figure 7: The delta view trees for the view trees in Figure 4 under a single-tuple update to \( R \).

Overall, maintaining the two view trees under a single-tuple update to any relation takes \( O(N^\varepsilon) \) time.

An update may change the degree of values over a partition key from light to heavy or vice versa. In such cases, we need to rebalance the partitioning and possibly recompute some views. Although such rebalancing steps may take time more than \( O(N^\delta \varepsilon) \), they happen periodically and their amortised cost remains the same as for a single-tuple update (Section F.3).

7 Related Work

Our work lies at the intersection of two lines of research: querying under access patterns and dynamic evaluation. No prior work considered dynamic evaluation for queries with access patterns.

**Access Patterns.** The problem of query evaluation in the presence of limited access to base relations is well-investigated in the literature. Deciding whether first-order queries can be answered under access patterns is hard [22, 23, 21]. Access restriction on base relations limits the space of feasible query plans and leads to suboptimal restriction-unaware search algorithms [12, 25] and allows for novel optimisations in the presence of access restrictions and integrity constraints [10, 4, 5]. Our setting does not share the above answerability and optimisation challenges: the query output can only be obtained by setting input free variables to constants; such queries may be seen as sources with access restrictions in the prior setting. To differentiate the two settings, we call our setting output access patterns. For given values over the input variables, CQAP queries become residual queries. As shown in Example 4, our CQAP approach can be however more efficient than the evaluation of residual queries. To support efficient answering, we precompute (subject to a trade-off) some mappings between possible values for input variables and the query output, whereas a residual query is to be computed from scratch for the given input values.

Our work is inspired by a seminal work on the space-delay trade-off for the static evaluation of full conjunctive queries with output access patterns [9]. This work constructs a succinct representation of the query result that allows for the enumeration of those tuples that conform with value bindings of the input variables. The representation relies on a tree decomposition of the query where the input variables form a connected subtree. This work does not support queries with projection nor dynamic evaluation. Follow-up work considers the problem of answering Boolean conjunctive queries with access patterns, where every free variable is fixed to a constant at query time, again in the static setting [8].

**Dynamic evaluation.** Our work draws on and generalises two prior works: the dichotomy for \( q \)-hierarchical queries under single-tuple updates [9, 16] and the complexity trade-offs for queries under updates [17, 18, 20]. We refer the reader to a comprehensive comparison [19] of dynamic query evaluation techniques and how they are recovered by the trade-off [20] extended in our work.

Our hardness result does not follow from that for non-\( q \)-hierarchical queries [6]. It considers several hard patterns as opposed to one and uses different reductions. There are technical differences between the prior trade-off framework [20] and ours. The prior work uses indicator views used to maintain the sets of heavy and light values in base relations. Our work avoids them to achieve simplicity in analysis and the overall approach, e.g., there is no need for a mechanism to maintain indicator views under updates. We partition relations into heavy and light parts based on the frequencies of their values in all input relations.
We modularise the construction of the view trees. Given a canonical variable order for the query, we first create a set of variable orders that represent heavy/light evaluation strategies and then map them to view trees. The advantage is a simpler complexity analysis for the views, since the variables orders and their view trees share the same width measures. It also allows for an extension to new applications. To construct view trees for CQAP, we now use variable orders that are access-top instead of only free-top as in prior work [20]. Finally, the enumeration approach is different: Our framework of iterators allow for efficient enumeration over view trees modelled on any variable orders.

**Dissociation.** Query fractures are central to our access pattern approach. They replace the input variables in a given query with fresh input variables depending on the structure of the query. Dissociation is similar in spirit: It is used to define upper and lower bounds for the probability of Boolean functions by treating multiple occurrences of a random variable as independent and assigning them new individual probabilities [13]. Query dissociation serves the same purpose [14]. It alters both the data, by making multiple independent copies of some tuples in the database and extending relational schemas with attributes, and the query, by extending atoms with variables.

## 8 Conclusion and Future Work

This paper investigates the dynamic evaluation of conjunctive queries with output access patterns. It establishes a dichotomy: CQAP\(_0\) queries are precisely those queries with constant-time update and delay unless the Online Matrix-Vector conjecture fails. This dichotomy is sensitive to the access pattern. We further define the class CQAP\(_1\) of (weakly-Pareto optimal) queries that admit \(O(N^{0.5})\) time for update and delay. Finally, we investigate the trade-off between preprocessing, update, and enumeration for a large class of queries. The main insight into the complexity of queries with access patterns is captured by the notions of query fracture and dominance of input variables. Future work includes the generalisation of our trade-off for all CQAP queries as well as the optimality for the dynamic evaluation of queries beyond the CQAP\(_0\) and CQAP\(_1\) classes.

## References

[1] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

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

[3] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.

[4] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with Access Patterns and Integrity Constraints. *VLDB*, 8(6):690–701, 2015.

[5] M. Benedikt, B. Ten Cate, and E. Tsamoura. Generating Low-cost Plans from Proofs. In *PODS*, pages 200–211, 2014.

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

[7] R. Chirkova and J. Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012.

[8] S. Deep, X. Hu, and P. Koutris. Space-Time Tradeoffs for Answering Boolean Conjunctive Queries. *arXiv*, abs/2109.10889, 2021.

[9] S. Deep and P. Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.
A. Deutsch, B. Ludäscher, and A. Nash. Rewriting Queries using Views with Access Patterns under Integrity Constraints. Theor. Comput. Sci., 371(3):200–226, 2007.

A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In CSL, pages 189–202, 2011.

D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. SIGMOD Rec., 28(2):311–322, 1999.

W. Gatterbauer and D. Suciu. Oblivious bounds on the probability of boolean functions. ACM Trans. Database Syst., 39(1):5:1–5:34, 2014.

W. Gatterbauer and D. Suciu. Dissociation and propagation for approximate lifted inference with standard relational database management systems. VLDB J., 26(1):5–30, 2017.

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.

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

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.

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.

A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. CoRR, abs/1907.01988, 2019.

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.

C. Li and E. Chang. On Answering Queries in the Presence of Limited Access Patterns. In ICDT, pages 219–233, 2001.

A. Nash and B. Ludäscher. Processing First-Order Queries under Limited Access Patterns. In PODS, pages 307–318, 2004.

A. Nash and B. Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In EDBT, pages 422–440, 2004.

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

R. Yerneni, C. Li, J. Ullman, and H. Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In ICDT, pages 348–364, 1999.

A. Nash and B. Ludäscher. Processing First-Order Queries under Limited Access Patterns. In PODS, pages 307–318, 2004.

A. Nash and B. Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In EDBT, pages 422–440, 2004.

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

R. Yerneni, C. Li, J. Ullman, and H. Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In ICDT, pages 348–364, 1999.

A Missing Details in Section [1]

In this appendix we give the missing details and proofs of the formal statements in the introduction. We start with an auxiliary lemma.

**Lemma 21.** If a CQAP query $Q$ can be evaluated with $O(f_p(N))$ preprocessing time, $O(f_e(N))$ enumeration delay, and $O(f_u(N))$ amortised update time, then its fracture $Q^\dagger$ can be evaluated with the same asymptotic complexities, where $N$ is the database size.
Proof. Consider a CQAP query \( Q(\mathcal{O}|\mathcal{I}) \), its fracture \( Q_1(\mathcal{O}|\mathcal{I}_1) \), and a database \( \mathcal{D} \) for \( Q_1 \) of size \( N \). We call a fresh variable \( A \) in \( Q_1 \) that replaces a variable \( A' \) in \( Q \) a representative of \( A \). Let \( C_1, \ldots, C_n \) be the sets of database relations that correspond to the connected components of \( Q_1 \). We construct from \( \mathcal{D} \) the databases \( \mathcal{D}_1, \ldots, \mathcal{D}_n \), where each \( \mathcal{D}_i \) is constructed as follows. The database \( \mathcal{D}_i \) contains each relation in \( \mathcal{D} \) such that:

1. If \( R \in C_i \) and \( R \) has a variable \( A \) in its schema that is a representative of a variable \( A' \), the variable \( A \) is replaced by \( A' \);
2. the values in all relations not contained in \( C_i \) are replaced by a single dummy value \( d_i \).

The overall size of the databases is \( \mathcal{O}(N) \). Given an input tuple \( t \) over \( \mathcal{I} \), we denote by \( (Q(\mathcal{O}|t), \mathcal{D}_i) \) the result of \( Q \) for input \( t \) evaluated on \( \mathcal{D}_i \). The result consists of the tuples over the output variables in \( C_i \) for the given input tuple \( t \), paired with the dummy value \( d_i \) over the output variables not in \( C_i \). Intuitively, the result of \( Q_1 \) on \( \mathcal{D} \) can be obtained from the Cartesian product of the results of \( Q \) on \( \mathcal{D}_1, \ldots, \mathcal{D}_n \). To be more precise, consider a tuple \( t_1 \) over \( \mathcal{I}_1 \). We define for each \( i \in [n] \), a tuple \( t_i \) over \( \mathcal{I} \) such that \( t_i[A] = t_1[A'] \) if \( A' \) is a representative of \( A \). The result of \( Q_1(\mathcal{O}|t_1) \) on \( \mathcal{D} \) is equal to the Cartesian product \( \times_{i \in [n]} \pi_{C_i}(Q(\mathcal{O}|t_i), \mathcal{D}_i) \), where \( \pi_{C_i} \) is the set of output variables of \( Q \) contained in \( C_i \). Now, assume that we want to enumerate the result of \( (Q(t_1), \mathcal{D}) \). We start the enumeration procedure for each \( Q(\mathcal{O}|t_i), \mathcal{D}_i \) with \( i \in [n] \). For each \( t'_i \in Q(\mathcal{O}|t_i), \mathcal{D}_i \), \( \mathcal{I} \), we return the tuple \( t'_i \). This implies that the result of \( Q_1(\mathcal{O}|t_1), \mathcal{D} \) can be enumerated with \( \mathcal{O}(f_e(N)) \) delay if \( Q \) admits \( \mathcal{O}(f_e(N)) \) enumeration delay.

We execute the preprocessing procedure for \( Q \) on each of the databases \( \mathcal{D}_1, \ldots, \mathcal{D}_n \) which takes \( \mathcal{O}(f_p(N)) \) overall time. Consider an update \( \{ t \mapsto m \} \) to a relation \( R \) that is contained in the connected component \( C_i \) for some \( i \in [n] \). We apply the update \( \{ t_x \mapsto m \} \) to relation \( R \) in \( \mathcal{D}_i \), where \( t_x \) is the tuple over \( \mathcal{I} \):

\[
t_x[A] = \begin{cases} t'[A'] & \text{if } A' \text{ is a representative of } A \\ t[A] & \text{otherwise} \end{cases}
\]

The update takes \( \mathcal{O}(f_e(N)) \) amortised update time.

Overall, we obtain an evaluation procedure for \( Q_1 \) with \( \mathcal{O}(f_p(N)) \) preprocessing time, \( \mathcal{O}(f_e(N)) \) enumeration delay, and \( \mathcal{O}(f_o(N)) \) amortised update time.

\[\square\]

A.1 Proof of Theorem 1

A.1.1 Complexity Upper Bound for CQAP\(_0\) Queries

We prove the first statement in Theorem 1. Assume that \( Q \) is in CQAP\(_0\). By definition, the fracture of \( Q \) must be hierarchical. By Proposition 27, \( Q \) has dynamic width 0. It follows from Proposition 25 that the static width of \( Q \) is at most 1. By Theorem 3 and choosing \( \varepsilon = 1 \), \( Q \) admits \( \mathcal{O}(N) \) preprocessing time, \( \mathcal{O}(1) \) enumeration delay, and \( \mathcal{O}(1) \) update time. In case of CQAP queries with dynamic width 0, our approach does not partition relations. Therefore, there is no need for rebalancing between relation parts. Hence, the update time is non-amortised.

A.1.2 Complexity Lower Bound for Non-CQAP\(_0\) Queries

We prove the second statement in Theorem 1. The proof is based on a reduction of the following Online Matrix-Vector Multiplication (OMv) problem [15] to the evaluation of non-CQAP\(_0\) queries.

**Definition 22** (Online Matrix-Vector Multiplication [15]). We are given an \( n \times n \) Boolean matrix \( M \) and receive \( n \) column vectors of size \( n \), denoted by \( v_1, \ldots, v_n \), one by one; after seeing each vector \( v_i \), we output the product \( Mv_i \) before we see the next vector.

It is strongly believed that the OMv problem cannot be solved in subcubic time.

**Conjecture 23** (OMv Conjecture, Theorem 2.4 in [15]). For any \( \gamma > 0 \), there is no algorithm that solves the OMv problem in time \( \mathcal{O}(n^{3-\gamma}) \).
We start with the high-level idea of the proof. Consider the following simple CQAP queries, which are not in CQAP₀.

\[ Q_1(O|·) = R(A), S(A, B), T(B) \quad O \subseteq \{ A, B \} \]
\[ Q_2(A|·) = R(A, B), S(B) \]
\[ Q_3(·|A) = R(A, B), S(B) \]
\[ Q_4(B|A) = R(A, B), S(B) \]

Each query is equal to its fracture, up to variable renaming. Query \( Q_1 \) is not hierarchical; \( Q_2 \) is not free-dominant; \( Q_3 \) and \( Q_4 \) are not input-dominant. It is known that queries that are not hierarchical or free-dominant do not admit constant update time and enumeration delay, unless the OMv conjecture fails [6]. We show that the OMv problem can also be reduced to the evaluation of each of the queries \( Q_3 \) and \( Q_4 \).

Our reduction implies that any algorithm that evaluates the queries \( Q_3 \) or \( Q_4 \) with arbitrary preprocessing time, \( O(N^{2-\gamma}) \) update time, and \( O(N^{2-\gamma}) \) enumeration delay for any \( \gamma > 0 \) can be used to solve the OMv problem in subcubic time, which rejects the OMv conjecture. We then show that the evaluation of one of the queries \( Q_1 \) to \( Q_4 \) can be reduced to the evaluation of any CQAP query that is not in CQAP₀ and does not have repeating relation symbols.

In each of the following two reductions, our starting assumption is that there is an algorithm \( A \) that evaluates the given query with arbitrary preprocessing time, \( O(N^{2-\gamma}) \) amortised update time, and \( O(N^{2-\gamma}) \) enumeration delay for some \( \gamma > 0 \). We then show that \( A \) can be used to design an algorithm \( B \) that solves the OMv problem in subcubic time. The reductions are adaptations of the reduction in the proof of Proposition 1.7 in [20]. The encoding of the matrix and the vector columns in the input to the OMv problem is the same. The difference is in the way we compute the product of the matrix and a column vector.

**Hardness for \( Q_4 \)**

Given \( n \geq 1 \), let \( M, v_1, \ldots, v_n \) be an input to the OMv problem, where \( M \) is an \( n \times n \) Boolean Matrix and \( v_1, \ldots, v_n \) are Boolean column vectors of size \( n \). Algorithm \( B \) uses relation \( R \) to encode matrix \( M \) and relation \( S \) to encode the incoming vectors \( v_1, \ldots, v_n \). The database domain is \( [n] \).

First, algorithm \( B \) executes the preprocessing stage on the empty database. Since the database is empty, the preprocessing stage must end after constant time. Then, it executes at most \( n^2 \) updates to relation \( R \) such that \( R(i, j) = 1 \) if and only if \( M(i, j) = 1 \). Afterwards, it performs a round of operations for each incoming vector \( v_r \), with \( r \in [n] \). In the first part of each round, it executes at most \( n \) updates to relation \( S \) such that \( S(j) = 1 \) if and only if \( v_r(j) = 1 \). Observe that \( Q_4(i|·) \) is true for some \( i \in [n] \) if and only if \( (Mv_r)(i) = 1 \).

Algorithm \( B \) constructs the result vector \( u_r = Mv_r \), as follows. It asks for each \( i \in [n] \), whether \( Q_4(i|·) \) is true, i.e., \( i \) is in the result of \( Q_3 \). If yes, the \( i \)-th entry of the result of \( u_r \) is set to 1, otherwise, it is set to 0.

**Time Analysis**

The size of the database remains \( O(n^2) \) during the whole procedure. Algorithm \( B \) needs at most \( n^2 \) updates to encode \( M \) by relation \( R \). Hence, the time to execute these updates is \( O(n^2(N^{2-\gamma}) = O(n^{3-2\gamma}) \). In each round \( r \) with \( r \in [n] \), algorithm \( B \) executes \( n \) updates to encode vector \( v_r \) into relation \( S \) and asks for the result of \( Q_3(i|·) \) for every \( i \in [n] \). The \( n \) updates and requests need \( O(n(n^3)^{2-\gamma}) = O(n^{2-2\gamma}) \) time. Hence, the overall time for a single round is \( O(n^{2-2\gamma}) \). Consequently, the time for \( n \) rounds is \( O(mn^{2-2\gamma}) = O(n^{3-2\gamma}) \). This means that the overall time of the reduction is \( O(n^{3-2\gamma}) \) in worst-case, which is subcubic.

**Hardness for \( Q_4 \)**

The reduction differs slightly from the case for \( Q_3 \) in the way algorithm \( B \) constructs the result vector \( u_r = Mv_r \) in each round \( r \). For each \( i \in [n] \), it starts the enumeration process for \( Q_1(B|i) \).

If one tuple is returned, it stops the enumeration process and sets the \( i \)-th entry of \( u_r \) to 1. If no tuple is returned, the \( i \)-th entry is set to 0. Thus, the time to decide the \( i \)-th entry of the result of \( u_r \) is the same as in case of \( Q_3 \). Hence, the overall time of the reduction stays subcubic.

**Hardness for Arbitrary Non-CQAP₀-Queries**

Consider now an arbitrary CQAP query \( Q \) that is not in CQAP₀ and does not have repeating relation symbols. Since \( Q \) is not in CQAP₀, this means that its
fracture $Q_1$ is either not hierarchical, not free-dominant, or not input-dominant. If $Q_1$ is not hierarchical or it is not free-dominant and all free variables are output, it follows from prior work that there is no algorithm that evaluates $Q_1$ with $O(N^{2\gamma-\epsilon})$ enumeration delay, and $O(N^{2\gamma-\epsilon})$ amortised update time for any $\gamma > 0$, unless the OMv conjecture fails. By Lemma 21 no such algorithm can exist for $Q$. Hence, we assume that $Q_1$ is hierarchical and consider two cases:

1. $Q_1$ is not free-dominant and all free variables are input
2. $Q_1$ is free-dominant but not input-dominant

Case (1). The query must contain an input variable $A$ and a bound variable $B$ such that $\text{atoms}(A) \subset \text{atoms}(B)$. This mean that there are two atoms $R(\mathcal{X})$ and $S(\mathcal{Y})$ with $\mathcal{Y} \cap \{A, B\} = \{B\}$ and $A, B \in \mathcal{X}$. Assume that there is an algorithm $A$ that evaluates $Q_1$ with arbitrary preprocessing time, $O(N^{2\gamma})$ enumeration delay, and $O(N^{2\gamma})$ amortised update time for some $\gamma > 0$. We will design an algorithm $B$ that evaluates $Q_3$ with the same complexities. This rejects the OMv conjecture. Hence, by Lemma 21 $Q$ cannot be evaluated with these complexities, unless the OMv conjecture fails.

We define $\mathcal{R}_{(A, B)}$ to be the set of atoms that contain both $A$ and $B$ in their schemas and $\mathcal{S}_{(\neg A, B)}$ to be the set of atoms that contain $B$ but not $A$. Note that there cannot be any atom containing $A$ but not $B$, since this would imply that the query is not hierarchical, contradicting our assumption. We use each atom $R'(\mathcal{X}') \in \mathcal{R}_{(A, B)}$ to encode atom $R(A, B)$ and each atom $S'(\mathcal{Y}') \in \mathcal{S}_{(\neg A, B)}$ to encode atom $S(B)$ in $Q_3$. Consider a database $\mathcal{D}$ of size $N$ for $Q_3$ and a dummy value $d$ that is not included in the domain of $\mathcal{D}$. We write $(S, A = a, B = b, d)$ to denote a tuple over schema $S$ that assigns the values $a$ and $b$ to the variables $A$ and respectively $B$ and all other variables in $S$ to $d$. Likewise, $(S, B = b, d)$ denotes a tuple that assigns value $b$ to $B$ and all other variables in $S$ to $d$. Algorithm $B$ first constructs from $\mathcal{D}$ a database $\mathcal{D}'$ for $Q_1$ as follows. For each tuple $(a, b)$ in relation $\mathcal{R}$ and each atom $R'(\mathcal{X}')$ in $\mathcal{R}_{(A, B)}$, it assigns the tuple $(\mathcal{X}', A = a, B = b, d)$ to relation $R'$. Likewise, for each value $b$ in relation $S$ and each atom $S'(\mathcal{Y}')$ in $\mathcal{S}_{(\neg A, B)}$, it assigns the tuple $(\mathcal{Y}', B = b, d)$ to relation $S'$. The size of $\mathcal{D}'$ is linear in $N$. Then, algorithm $B$ executes the preprocessing for $Q_1$ on $\mathcal{D}'$. Each single-tuple update $\{(a, b) \mapsto m\}$ to relation $\mathcal{R}$ is translated to a sequence of single-tuple updates $\{(\mathcal{X}', A = a, B = b, d) \mapsto m\}$ to all relations referred to by atoms in $\mathcal{R}_{(A, B)}$. Analogously, updates $\{b \mapsto m\}$ to $S$ are translated to updates $\{(S', B = b, d) \mapsto m\}$ to all relations $S'$ with $S'(\mathcal{Y}') \in \mathcal{S}_{(\neg A, B)}$. Hence, the amortised update time is $O(N^{0.5-\epsilon})$. Each input tuple $(a)$ for $Q_3$ is translated into an input tuple $(\mathcal{I}, A = a, d)$ for $Q_1$ where $\mathcal{I}$ is the set of input variables for $Q_1$. Recall that all free variables of $Q_1$ are input. The answer of $Q_3(\mathcal{I})$ is true if and only if the answer of $Q_1(\mathcal{I}, A = a, d)$ is true. The answer time is $O(N^{0.5-\epsilon})$. We conclude that $Q_3$ can be evaluated with $O(N^{0.5-\epsilon})$ enumeration delay and $O(N^{0.5-\epsilon})$ amortised update time, a contradiction due to the OMv conjecture.

Case (2). We now consider the case that the query $Q_1$ is free-dominant but not input-dominant. In this case, we reduce the evaluation of $Q_4$ to the evaluation of $Q_1$. The reduction is analogous to Case (1). The way we encode the atoms $R(A, B)$ and $S(B)$, do preprocessing, and translate the updates is exactly the same as in Case (1). The only difference is the way we retrieve the $B$-values in $Q_4(B[a])$ for an input value $a$. We translate $a$ into an input tuple to $Q_1$ where all input variables besides $A$ are assigned to $d$. Recall that $Q_1$ might have several output variables besides $B$. By construction, they can be assigned only to $d$. Hence, all output tuples returned by $Q_1$ have distinct $B$-values. These $B$-values constitute the result of $Q_4(B[a])$. We conclude that $Q_4$ can be evaluated with $O(N^{0.5-\epsilon})$ enumeration delay and $O(N^{0.5-\epsilon})$ amortised update time, which contradicts the OMv conjecture.

A.2 Proof of Theorem 2

By Proposition, $Q$ has dynamic width 1. The theorem follows directly from Theorem 3 by choosing $\epsilon = \frac{1}{2}$.

A.3 Proof of Theorem 3

Consider a CQAP query $Q$ with static width $w$ and dynamic width $\delta$. Assume that the fracture $Q_1$ of $Q$ is hierarchical. In the preprocessing stage, we construct a set of view trees representing the result of $Q_1$. These
view trees can be materialised in $O(N^{1+(w-1)e})$ time (Propositions 12) and can be maintained with $O(N^{de})$ amortised time under single-tuple updates (Proposition 19). Given any input tuple, the view trees allow for the enumeration of the result of $Q$ with $O(N^{1-e})$ enumeration delay (Proposition 17).

B Missing Details in Section 2

Further Notation. Given a query and a variable $X$, we denote by $\text{vars}(\text{atoms}(X))$, $\text{free}(\text{atoms}(X))$, and $\text{in}(\text{atoms}(X))$, the sets of all, free and respectively input variables contained in $\text{atoms}(X)$. For a variable order $\omega$, $\text{bound}(\omega)$ and $\text{out}(\omega)$ are the sets of bound and respectively output variables in $\omega$. Given a variable order $\omega$ and a tuple $p = (X_1, \ldots, X_k)$ of variables, we denote by $(p \circ \omega)$ the variable order defined as follows: $X_1$ is the root, $X_{i+1}$ is the single child of $X_i$ for $i \in [k-1]$, and $\omega$ is the single child tree of $X_k$.

Computational Model. We consider the RAM model of computation. Each relation (or materialized view) $R$ over schema $\mathcal{X}$ is implemented by a data structure that stores key-value entries $(x, R(x))$ for each tuple $x$ with $R(x) \neq 0$ and needs $O(|R|)$ space. This data structure can: (1) look up, insert, and delete entries in constant time, (2) enumerate all stored entries in $R$ with constant delay, and (3) report $|R|$ in constant time. For a schema $S \subseteq \mathcal{X}$, we use an index data structure that for any $t \in \text{Dom}(S)$ can: (4) enumerate all tuples in $\sigma_{S=t} R$ with constant delay, (5) check $t \in \pi_S R$ in constant time; (6) return $|\sigma_{S=t} R|$ in constant time; and (7) insert and delete index entries in constant time.

We give an example data structure that conforms to the computational model. Consider a relation (materialized view) $R$ over schema $\mathcal{X}$. A hash table with chaining stores key-value entries $(x, R(x))$ for each tuple $x$ over $\mathcal{X}$ with $R(x) \neq 0$. The entries are doubly linked to support enumeration with constant delay. The hash table can report the number of its entries in constant time and supports lookups, inserts, and deletes in constant time on average, under the assumption of simple uniform hashing.

To support index operations on a schema $\mathcal{F} \subseteq \mathcal{X}$, we create another hash table with chaining where each tuple entry stores a tuple $t$ of $\mathcal{F}$-values as key and a doubly-linked list of pointers to the entries in $R$ having the $\mathcal{F}$-values $t$ as value. Looking up an index entry given $t$ takes constant time on average under simple uniform hashing, and its doubly-linked list enables enumeration of the matching entries in $R$ with constant delay. Inserting an index entry into the hash table additionally prepends a new pointer to the doubly-linked list for a given $t$; overall, this operation takes constant time on average. For efficient deletion of index entries, each entry in $R$ also stores back-pointers to its index entries (one back-pointer per index for $R$). When an entry is deleted from $R$, locating and deleting its index entries in the doubly-linked lists takes constant time per index.

C Missing Details in Section 3

Width measures. Given a conjunctive query $Q$ and $\mathcal{F} \subseteq \text{vars}(Q)$, a fractional edge cover of $\mathcal{F}$ is a solution $\lambda = (\lambda_R(x))_{R(x) \in \text{atoms}(Q)}$ to the following linear program [1]:

\[
\begin{align*}
\text{minimize} & \quad \sum_{R(x) \in \text{atoms}(Q)} \lambda_{R(x)} \\
\text{subject to} & \quad \sum_{R(x), x \in \mathcal{X}} \lambda_{R(x)} \geq 1 \quad \text{for all } \mathcal{X} \in \mathcal{F} \quad \text{and} \\
& \quad \lambda_{R(x)} \in [0, 1] \quad \text{for all } R(x) \in \text{atoms}(Q)
\end{align*}
\]

The optimal objective value of the above program is called the fractional edge cover number of $\mathcal{F}$ and is denoted as $\rho^*_Q(\mathcal{F})$. An integral edge cover of $\mathcal{F}$ is a feasible solution to the variant of the above program with $\lambda_{R(x)} \in \{0, 1\}$ for each $R(x) \in \text{atoms}(Q)$. The optimal objective value of this program is called the integral edge cover number of $\mathcal{F}$ and is denoted as $\rho_Q(\mathcal{F})$. If $Q$ is clear from the context, we omit the index $Q$ in the expressions $\rho^*_Q(\mathcal{F})$ and $\rho_Q(\mathcal{F})$. 

19
Lemma 26. CQAP queries and uses the following auxiliary lemma.δ

either 
vars
0
difference can be
1.

Proposition 25.

For the bag

Example 24.

We show how to compute the widths for the variable order of the fractured 4-cycle query in Example 10. For the bag at variable A, we have \( \rho^*(\{ A \} \cup \text{dep}(A)) = \rho^*(\{ A, D_1, B_1 \}) = 2 \), which is the largest fractional edge cover number for any variable in the variable order. Further access-top variable orders are possible by swapping \( B_1 \) with \( D_2 \) and \( B_2 \) with \( D_1 \), yielding the same overall cost. The static width of the fractured 4-cycle query is thus 2. To compute the dynamic width of the same variable order, we consider for each atom, the fractional edge cover number of each bag without the variables in this atom.

Proof of Proposition 25.

Let \( Q \) be a hierarchical CQAP query with static width \( w \) and dynamic width \( \delta \). For the sake of contradiction assume that \( \delta - w > \delta \). Let \( \omega \) be an access-top variable order for \( Q_0 \). Let its dynamic width be \( \delta(\omega) = \delta \). Our assumption \( \delta - w > \delta \) implies:

\[ \forall X \in \text{vars}(Q_0), \forall Y \in \text{atoms}(\omega_X) : \rho^*(\{ X \} \cup \text{dep}_\omega(X)) - Y \leq \delta < \delta - w. \] (1)

Since the static width of \( Q_0 \) is \( w \), it holds:

\[ \exists Y \in \text{vars}(Q_0) : \rho^*(\{ Y \} \cup \text{dep}_\omega(Y)) < w. \] (2)

Note: If Statement [2] does not hold, \( \omega \) is an access-top variable order for \( Q_0 \) with static width less than \( w \). This implies that the static width of \( Q_0 \) is less than \( w \), which contradicts our assumption.

We show that Statements [1] and [2] are contradicting, which completes the proof. Let \( X \) be an arbitrary variable in \( \text{vars}(Q_0) \) and \( R(Y) \) any atom in \( \text{atoms}(\omega_X) \). Let \( \lambda = (\lambda_K(X))_{K(X) \in \text{atoms}(Q)} \) be a fractional edge cover of \( \{ X \} \cup \text{dep}_\omega(X) - Y \) such that

\[ \sum_{K(X) \in \text{atoms}(Q)} \lambda_K(X) = \rho^*(\{ X \} \cup \text{dep}_\omega(X)) - Y. \]

Due to Statement [1], it holds

\[ \sum_{K(X) \in \text{atoms}(Q)} \lambda_K(X) < w - 1. \] (3)

Let \( \lambda' = (\lambda'_{K(X)})_{K(X) \in \text{atoms}(Q)} \) be defined as

\[ \lambda'_{K(X)} = \begin{cases} 1, & \text{if } K(X) = R(Y) \\ \lambda_K(X), & \text{otherwise} \end{cases} \]

Clearly, \( \lambda' \) is a fractional edge cover of \( \{ X \} \cup \text{dep}_\omega(X) \). Moreover, due to Inequality [2], it holds that \( \sum_{K(X) \in \text{atoms}(Q)} \lambda'_{K(X)} < w \). Since \( X \) was chosen arbitrarily from \( \text{vars}(Q_0) \), this means that for any \( X \in \text{vars}(Q) \), we have \( \rho^*(\{ X \} \cup \text{dep}_\omega(X)) < w \). However, this contradicts Statement [2].
The classes CQAP\_0 and CQAP\_1 consist of queries with dynamic width 0 and 1, respectively.

**Proposition 27.** A query is in CQAP\_i for \(i \in \{0, 1\}\) if and only if it has dynamic width \(i\).

The proof of Proposition 27 is an extension of the proof of Proposition 1.5 in [19]. The proposition directly follows from the following Lemmas 28 and 29.

**Lemma 28.** Any CQAP\_i query with \(i \in \{0, 1\}\) has dynamic width at least \(i\).

**Lemma 29.** Any CQAP\_i query with \(i \in \{0, 1\}\) has dynamic width at most \(i\).

It remains to prove Lemmas 28 and 29.

**Proof of Lemma 28.** Let \(Q\) be a CQAP\_i query for some \(i \in \{0, 1\}\). Since the dynamic width of a CQAP query must be greater or equal to 0, the case \(i = 0\) is trivial. Assume now that \(Q\) is a CQAP\_1 query and consider its fracture \(Q\_1\). By definition of CQAP\_i queries, \(Q\_1\) must be hierarchical and almost free-dominant or almost input-dominant. Assume first that \(Q\_1\) is almost free-dominant. This means that \(Q\_1\) contains a bound variable \(X\) and an atom \(R(Y) \in \text{atoms}(X)\) such that:

\[
\text{free}(\text{atoms}(X)) \subseteq \mathcal{Y}
\]  

(4)

Let \(\omega = (T\_\omega, \text{dep}\_\omega)\) be an arbitrary access-top variable order for \(Q\_1\). Since the dynamic width of a CQAP query must be greater or equal to 0, the case \(i = 0\) is trivial. Assume now that \(Q\) is a CQAP\_1 query and consider its fracture \(Q\_1\). By definition of CQAP\_i queries, \(Q\_1\) must be hierarchical and almost free-dominant or almost input-dominant. Assume first that \(Q\_1\) is almost free-dominant. This means that \(Q\_1\) contains a bound variable \(X\) and an atom \(R(Y) \in \text{atoms}(X)\) such that:

\[
\text{free}(\text{atoms}(X)) \subseteq \mathcal{Y}
\]

(5)

Consider any access-top variable order \(\omega = (T\_\omega, \text{dep}\_\omega)\) for \(Q\_1\). Since \(X\) is output, the variables in \(\text{in}(\text{atoms}(X))\) must be contained in \(\text{anc}\_\omega(X)\). This means that \(\text{in}(\text{atoms}(X)) \subseteq (\{X\} \cup \text{dep}\_\omega(X))\). By Assumption 28, \(\rho^*(\{(X) \cup \text{dep}\_\omega(X)) \setminus \mathcal{Y}\) must be at least 1. This implies that \(\rho^*((\{X\} \cup \text{dep}\_\omega(X)) \setminus \mathcal{Y}\) must be at least 1 (Lemma 29). It follows that \(\delta(\omega) \geq 1\). Since \(\omega\) is an arbitrary access-top variable order for \(Q\_1\), we derive that the dynamic width of \(Q\) is at least 1.

The case that the fracture \(Q\_1\) is almost input-dominant is handled analogously. The query \(Q\_1\) must contain an output variable \(X\) and an atom \(R(Y) \in \text{atoms}(X)\) such that:

\[
\text{in}(\text{atoms}(X)) \subseteq \mathcal{Y}
\]

(5)

Consider any access-top variable order \(\omega = (T\_\omega, \text{dep}\_\omega)\) for \(Q\_1\). Since \(X\) is output, the variables in \(\text{in}(\text{atoms}(X))\) must be contained in \(\text{anc}\_\omega(X)\). This means that \(\text{in}(\text{atoms}(X)) \subseteq (\{X\} \cup \text{dep}\_\omega(X))\). By Assumption 28, \(\rho^*((\{X\} \cup \text{dep}\_\omega(X)) \setminus \mathcal{Y}\) must be at least 1. It follows that \(\delta(\omega) \geq 1\). Therefore, the dynamic width of \(Q\) must be at least 1.

**Proof of Lemma 29.** Consider a CQAP\_0 query \(Q\) and its fracture \(Q\_1\). The query \(Q\_1\) admits a canonical variable order \(\omega = (T\_\omega, \text{dep}\_\omega)\) where all free variables are above the bound ones and all input variables are output variables. Hence, \(\omega\) is access-efficient. Consider a variable \(X\) in \(\omega\) and an atom \(R(Y) \in \text{atoms}(X)\). Since \(X\) is canonical, it holds \(\text{dep}\_\omega(X) = \text{anc}\_\omega(X)\) and \(R(Y)\) has \(\{X\} \cup \text{anc}\_\omega(X)\) in its schema. Hence, \((\{X\} \cup \text{dep}\_\omega(X)) \setminus \mathcal{Y} = (\{X\} \cup \text{anc}\_\omega(X)) \setminus \mathcal{Y} = \emptyset\). This means that \(\rho^*((\{X\} \cup \text{dep}\_\omega(X)) \setminus \mathcal{Y}\) is 0. Since we chose \(X\) and \(R(Y) \in \text{atoms}(X)\) arbitrarily, this implies that the dynamic width of \(\omega\) is 0. Hence, the dynamic width of \(Q\) and, therefore, of \(Q\) must be 0.

Now, assume that \(Q\) is a CQAP\_1 with fracture \(Q\_1(O, I)\). Let \(\omega\) be the canonical variable order of \(Q\_1\). By Lemma 29, the function Acc-Top(\(\omega, O, I\)) in Figure 8 (Section D.1) constructs an access-top variable order \(\omega^*\) for \(Q\_1\) with dynamic width \(\kappa(\omega, I, O)\), where

\[
\kappa(\omega, I, O) = \max_{Y \subseteq \text{bound}(\omega)} \max_{Z \subseteq \text{out}(\omega)} \{\rho^*((\text{vars}(\omega_Y) \cap F) \setminus \mathcal{Y}, \rho^*((\text{vars}(\omega_Z) \cap I) \setminus \mathcal{Y})\} \]

with \(F = I \cup O\). Recall that \(Q\_1\) is almost free- or almost input-dominant. Consider an arbitrary variable \(X\) in \(\omega\) and an atom \(R(Y)\) containing \(X\). If \(X\) is bound, then \(\rho^*((\text{vars}(\omega_X) \cap F) \setminus \mathcal{Y}\) can be at most 1. Similarly, if \(X\) is output, then \(\rho^*((\text{vars}(\omega_X) \cap I) \setminus \mathcal{Y}\) can be at most 1. It follows that \(\kappa(\omega, I, O)\) is at most 1. This implies that \(\omega^*\) is an access-top variable order for \(Q\_1\) with dynamic width at most 1. We conclude that the dynamic width of \(Q\) must be at most 1.
Acc-Top(variable order \(\omega\), access pattern \((\mathcal{O}|\mathcal{I})\)) : variable order

```
switch \(\omega\):

\[
\begin{array}{ll}
R(\mathcal{Y}) & 1 \text{ return } R(\mathcal{Y}) \\
X & 2 \text{ let } \omega_i' = \text{Acc-Top}(\omega_i, (\mathcal{O}|\mathcal{I})), \forall i \in [k] \\
\omega_1 \ldots \omega_k & 3 \text{ let } \mathcal{D} = \begin{cases} 
\emptyset & \text{if } X \in \mathcal{I} \\
\text{vars}(\omega) \cap \mathcal{I}, & \text{else if } X \in \mathcal{O} \\
\text{vars}(\omega) \cap (\mathcal{I} \cup \mathcal{O}) & \text{otherwise}
\end{cases} \\
\{\hat{\omega}_1^i, \ldots, \hat{\omega}_m^i\} & 4 \text{ let } \{\hat{\omega}_1^i, \ldots, \hat{\omega}_m^i\} = \Delta(\omega_i', \mathcal{D}), \forall i \in [k] \\
(X_1, \ldots, X_{\ell}) & 5 \text{ let } (X_1, \ldots, X_{\ell}) = \mathcal{D} \cap \mathcal{I} \cup \mathcal{D} \cap \mathcal{O} \text{ be an ordering that is compatible with the partial order of } \omega \\
\text{X_1} & 6 \text{ return } \text{X_1} \\
\vdots & \text{X_2} \\
\vdots & \text{X_\ell} \\
\hat{\omega}_1^1 & \hat{\omega}_m^1 \\
\hat{\omega}_1^k & \hat{\omega}_m^k
\end{array}
\]
```

Figure 8: Construction of an access-top variable order from a canonical variable order \(\omega\) of a hierarchical CQAP query with access pattern \((\mathcal{O}|\mathcal{I})\). The function \(\Delta(\omega', \mathcal{D})\), defined in Figure 9, deletes the variables in \(\mathcal{D}\) from the variable order \(\omega'\).

## D  Missing Details in Section 4

Consider a CQAP query \(Q\) with hierarchical fracture \(Q^\dagger\). In the preprocessing stage, we build a set of view trees for each connected component in \(Q^\dagger\). The union of these sets encodes the query result. In the following sections, we focus on building view trees for one connected component.

In Section D.1 we give a function that turns canonical variable orders into optimal access-top ones. In Section D.2 we explain how to obtain different variable orders from the canonical variable order of a hierarchical CQAP query by using the above function. In Section D.3 we describe the construction of view trees from variable orders.

Before we proceed, we introduce some notation used throughout this section. Consider the canonical variable order \(\omega\) of a hierarchical CQAP query and the subtree \(\omega_X\) of \(\omega\) rooted at a variable \(X\). The induced query \(Q_X(\mathcal{O}_X|\mathcal{I}_X)\) is defined over the join of the atoms at the leaves of \(\omega_X\). The \(\mathcal{I}_X\) consists of the input variables in \(\omega_X\) and the variables on the path from \(X\) to a root of \(\omega\). The set \(\mathcal{O}_X\) contains the output variables in \(\omega'\).

### D.1  From Canonical to Access-Top Variable Orders

Given a canonical variable order \(\omega\) of a hierarchical CQAP query \(Q\) with input variables \(\mathcal{I}\) and output variables \(\mathcal{O}\), the function \(\text{Acc-Top}(\omega, (\mathcal{O}|\mathcal{I}))\) in Figure 8 returns an access-top variable order for \(Q\) with optimal static and dynamic width. The function proceeds recursively on the structure of \(\omega\). At a variable \(X\), the function selects a set of \(\mathcal{D}\) of variables from the subtree \(\omega'\) rooted at \(X\) based on the type of \(X\): 1) if \(X\) is an input variable, the function sets \(\mathcal{D} = \emptyset\); 2) if \(X\) is an output variable, the function defined \(\mathcal{D}\) to be the input variables in \(\omega'\), and 3) if \(X\) is bound, the function sets \(\mathcal{D}\) to be the free variables in \(\omega'\) (Line 3). The function then takes out \(\mathcal{D}\) from \(\omega'\) and puts them on top of \(X\) (Lines 4-6). Line 5 makes sure the input
\[ \Delta(\text{variable order } \omega, \text{ variables } D) : \text{ set of variable orders} \]

**switch** \( \omega: \)

1. \( R(Y) \) \hspace{1cm} \text{return } \{R(Y)\} \\
2. \( X \) \hspace{1cm} \text{let } \{\omega^i_1, \ldots, \omega^i_{m_i}\} = \Delta(\omega_i, D), \ \forall i \in [k] \hspace{1cm} \text{return } \left\{ \begin{array}{c} \omega^1_1 \ldots \omega^1_{m_1} \ldots \omega^k_1 \ldots \omega^k_{m_k} \end{array} \right\} \\
3. \text{if } X \notin D \hspace{1cm} \text{return } \left\{ \begin{array}{c} \omega^1_1 \ldots \omega^1_{m_1} \ldots \omega^k_1 \ldots \omega^k_{m_k} \end{array} \right\} \\
4. \text{else if } X \text{ has parent } Y \hspace{1cm} \text{return } \left\{ \begin{array}{c} \omega^1_1 \ldots \omega^1_{m_1} \ldots \omega^k_1 \ldots \omega^k_{m_k} \end{array} \right\} \\
5. \text{else return } \left\{ \omega^1_1, \ldots, \omega^1_{m_1}, \ldots, \omega^k_1, \ldots, \omega^k_{m_k} \right\}

Figure 9: Deletion of a set \( D \) of variables from a variable order \( \omega \). If \( X \in D \) and \( X \) has a parent \( Y \), the child trees of \( X \) are appended to \( Y \). If \( X \in D \) and \( X \) has no parent, the child trees of \( X \) become independent.

Variables are put on top of the output variables.

The deletion of a set \( D \) of variables from a variable order \( \omega \) is implemented by the function \( \Delta(\omega, \Delta) \) in Figure 9. The function traverses recursively over all variables in \( \omega \). If a variable \( X \) is not included in \( D \), the function does not change the structure of \( \omega \) (Lines 3-4). In case \( X \in D \) and \( X \) has a parent \( Y \), it appends the child trees of \( X \) to the variable \( Y \) (Lines 5-6). If \( X \in D \) and \( X \) has no parent, the child trees of \( X \) become independent (Line 7).

**Proposition 30.** Given a CQAP query \( Q \), whose fracture \( Q_\dagger(\mathcal{O}|\mathcal{I}) \) is hierarchical, and a canonical variable order \( \omega \) for \( Q_\dagger \), \( \text{Acc-Top}(\omega, (\mathcal{I}|\mathcal{O})) \) constructs an access-top variable order for \( Q_\dagger \) with static width \( w(Q) \) and dynamic width \( \delta(Q) \).

Before proving Proposition 30, we introduce some useful notation. Let \( \omega \) be a canonical variable order of a hierarchical CQAP query. Let \( \mathcal{F}, \mathcal{I}, \) and \( \mathcal{O} \) be the free, input, and respectively output variables of the query, and \( X \) a variable in \( \omega \). The following measures \( \xi \) and \( \kappa \) express the static and the dynamic width of \( \omega_X \) without referring to access-top variable orders.

\[
\xi(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{Y \in \text{bound}(\omega_X), Z \in \text{out}(\omega_X)} \{\rho^{\omega_X}(\text{vars}(\omega_Y) \cap \mathcal{F}), \rho^{\omega_X}(\text{vars}(\omega_Z) \cap \mathcal{I})\}
\]

\[
\kappa(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{Y \in \text{bound}(\omega_X), Z \in \text{out}(\omega_X)} \max_{R(\gamma) \in \text{atoms}(\omega_Y)} \{\rho^{\omega_X}(\text{vars}(\omega_Y) \cap \mathcal{F}) - \gamma), \rho^{\omega_X}(\text{vars}(\omega_Z) \cap \mathcal{I}) - \gamma)\}
\]

If \( \omega_X \) does not contain any bound or output variable, we have \( \xi(\omega_X, \mathcal{I}, \mathcal{O}) = \kappa(\omega_X, \mathcal{I}, \mathcal{O}) = 0 \).

The next lemma expresses the static and dynamic variable width of the variable orders returned by the function \( \text{Acc-Top} \) in terms of the measures \( \xi \) and \( \kappa \).
Lemma 31. Given a canonical variable order $\omega$ of a hierarchical CQAP query $Q(\mathcal{O}|\mathcal{I})$, a variable $X$ in $\omega$, and the induced query $Q_X$ at variable $X$, $\text{Acc-Top}(\omega_X, (\mathcal{I}|\mathcal{O}))$ constructs a variable order $\omega^t$ such that $\omega^t = (\text{anc}_\omega(X) \circ \omega^t)$ is an access-top variable order for $Q_X$ with $w(\omega^t) = \max\{1, \xi(\omega_X, \mathcal{I}, \mathcal{O})\}$ and $\delta(\omega^t) = \kappa(\omega_X, \mathcal{I}, \mathcal{O})$.

Proof. The function $\text{Acc-Top}$ traverses the given canonical variable order and pulls up free variables such that the resulting variable order becomes access-top. More precisely, if a variable $X$ is bound and contains free variables in its subtree, the function puts all free variables below $X$ on top of $X$ such that the input variables are above the output variables. If the variable $X$ is an output variable and contains input variables in its subtree, it puts all input variables that are under $X$ on top of $X$.

If $\omega$ neither contains a bound variable above a free one nor an output variable above a bound one, the variable order remains unchanged. Since a canonical variable order has static width 1 and dynamic width 0, the statement in the lemma holds in this case.

Assume now that $\omega$ contains at least one bound variable above a free variable or at least one output variable above an input variable. Consider an arbitrary bound variable $X$ in $\omega$ that has free variables in its subtree. Let $\mathcal{F}$ be the set of free variables under $X$. Due to the structure of canonical variable orders, all variables in $\mathcal{F}$ depend on $X$. By moving the variables in $\mathcal{F}$ on top of $X$, the set $\mathcal{F}$ is added to the dependency set of $X$ in the resulting variable order $\omega^t$. Hence, the fractional edge cover number of $\{X\} \cup \text{dep}_\omega(X)$ is $\rho^*(\{X\} \cup \mathcal{F})$. The dependency set of a variable $Y$ in $\mathcal{F}$ can only decrease since the set of the variables from $Y$ to the root decreases. The dependency set of a variable $Y$ below $X$ changes if it contained a variable from $\mathcal{F}$ in its subtree that is now positioned on top of $Y$. However, the fractional edge cover number of $\{Y\} \cup \text{dep}_\omega(Y)$ is upper-bounded by the fractional edge cover number of $\{X\} \cup \text{dep}_\omega(X)$.

In case $X$ is an output variable that has a set $\mathcal{V}$ of input variables in its subtree, the reasoning is similar. The fractional edge cover number of $\{X\} \cup \text{dep}_\omega(X)$ is $\rho^*(\{X\} \cup \mathcal{V})$ and upper-bounds the fractional edge cover numbers at the other variables in the resulting variable order $\omega^t$.

Hence, the static width of $\omega^t$ is determined by the largest set of variables that is moved on top of a single variable by the function $\text{Acc-Top}$.

For the dynamic width of $\omega^t$, the reasoning is completely analogous. The dynamic width of $\omega^t$ is given by the largest set of variables that is moved on top of a single variable $X$ after removing the variables of any atom containing $X$.

We are ready to prove Proposition 30.\[\Box\]

Proof of Proposition 30. Consider a CQAP query $Q$ whose fracture $Q_1(\mathcal{O}|\mathcal{I})$ is hierarchical. Let $\mathcal{F} = \mathcal{I} \cup \mathcal{O}$ and $w$ and $\delta$ be the static and respectively dynamic width of $Q$. By the definition of static and dynamic width, $Q_1$ must have static width $w$ and dynamic width $\delta$. Let $\omega$ be the canonical variable order of $Q_1$. Without loss of generality, assume that $Q_1$ contains at least one atom with non-empty schema. Otherwise, $\text{Acc-Top}$ returns the set of atoms in $Q_1$, which is already an optimal access-top variable order for $Q_1$. Assume also that $\omega$ consists of a single connected component. Otherwise, we apply the same reasoning for each connected component. By Lemma 31, $\text{Acc-Top}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top variable order $\omega^t$ for $Q_1$ with static width $\max\{1, \xi(\omega_X, \mathcal{I}, \mathcal{O})\}$ and dynamic width $\kappa(\omega_X, \mathcal{I}, \mathcal{O})$. We first show:

\[\max\{1, \xi(\omega, \mathcal{I}, \mathcal{O})\} \leq w\]  
(6)

First, assume that $\xi(\omega, \mathcal{I}, \mathcal{O}) = 0$. This means $\max\{1, \xi(\omega, \mathcal{I}, \mathcal{O})\} = 1$. Since $Q_1$ contains at least one atom with non-empty schema, we have $w \geq 1$. Thus, Inequality (6) holds. Now, let $\xi(\omega, \mathcal{I}, \mathcal{O}) = \ell \geq 1$. We show that $w \geq \ell$. It follows from $\xi(\omega, \mathcal{I}, \mathcal{O}) = \ell$ that at least one of the following two cases holds:

- Case (1.1): $\omega$ contains a bound variable $Y$ such that $\rho^*_Q(\mathcal{F'}) = \ell$, where $\mathcal{F'} = \text{vars}(\omega_Y) \cap \mathcal{F}$
- Case (1.2): $\omega$ contains an output variable $Y$ such that $\rho^*_Q(\mathcal{I'}) = \ell$, where $\mathcal{I'} = \text{vars}(\omega_Y) \cap \mathcal{I}$.

We first consider Case (1.1). The inner nodes of each root-to-leaf path of a canonical variable order are the variables of an atom. Hence, for each variable $Z \in \mathcal{F'}$, there must be an atom in $Q_1$ that contains both
Y and Z. This means that Y and Z depend on each other. Let \( \omega' = (T, \text{dep}_\omega) \) be an arbitrary access-top variable order for \( Q_t \). Since all variables in \( \mathcal{F}' \) depend on Y, each of them must be on a root-to-leaf path with Y. Since Y is bound and the variables in \( \mathcal{F}' \) are free, the set \( \mathcal{F}' \) must be included in \( \text{anc}_\omega(Y) \). Thus, \( \mathcal{F}' \subseteq \text{dep}_\omega(Y) \). This means \( \rho^*(\{Y\} \cup \text{dep}_\omega(Y)) \geq \ell \), which implies \( w(\omega') \geq \ell \). It follows \( w \geq \ell \).

The reasoning for Case (1.2) is analogous. In any access-top variable order \( \omega' = (T, \text{dep}_\omega) \) for \( Q_t \), all variables in \( \mathcal{T}' \) must be included in \( \text{anc}_\omega(Y) \). Hence, \( \mathcal{T}' \subseteq \text{dep}_\omega(Y) \), which means \( \rho^*(\{Y\} \cup \text{dep}_\omega(Y)) \geq \ell \). This implies \( w(\omega') \geq \ell \), thus, \( w \geq \ell \).

It follows that the static width of the access-top variable order \( \text{Acc-Top}(\omega, (\mathcal{I}|\mathcal{O})) \) must be \( w(Q) \).

Following similar steps, we can show:

\[
\kappa(\omega, \mathcal{I}, \mathcal{O}) \leq \delta
\]  

Let \( \kappa(\omega, \mathcal{I}, \mathcal{O}) = k \). We show that \( \delta \geq k \). The definition of \( \kappa(\omega, \mathcal{I}, \mathcal{O}) \) implies that one of the following two cases must hold:

- Case (2.1): \( \omega \) contains a bound variable Y and an atom \( R(Y) \) containing Y such that \( \rho^*_Q(\mathcal{F}' - Y) = k \), where \( \mathcal{F}' = \text{vars}(\omega_Y) \cap \mathcal{F} \).
- Case (2.2): \( \omega \) contains an output variable Y and an atom \( R(Y) \) containing Y such that \( \rho^*_Q(\mathcal{T}' - Y) = k \), where \( \mathcal{T}' = \text{vars}(\omega_Y) \cap \mathcal{T} \).

We consider Case (2.1). Let \( \omega' = (T, \text{dep}_\omega) \) be an arbitrary access-top variable order for \( Q_t \). The atom \( R(Y) \) must be included in \( \text{atoms}(\omega'_Y) \), since it contains Y. All variables in \( \mathcal{F}' \) depend on Y. Since Y is bound and the variables in \( \mathcal{F}' \) are free, the set \( \mathcal{F}' - Y \) must be included in \( \text{anc}_\omega(Y) \). Hence, \( \mathcal{F}' - Y \subseteq \text{dep}_\omega(Y) \). This implies that \( \rho^*(((Y) \cup \text{dep}_\omega(Y)) - Y) \geq k \). This means \( \rho^*(((Y) \cup \text{dep}_\omega(Y)) - Y) \geq k \). This implies that \( \delta(\omega') \geq k \). It follows \( \delta \geq k \).

To show Case (2.2), we reason analogously. We just treat the output variables like the bound variables and input variables like the free variables in Case (2.1).

Overall, we conclude that given a CQAP \( Q \) and its fracture \( Q_t|\mathcal{I} \), \( \text{Acc-Top}(\omega, (\mathcal{I}|\mathcal{O})) \) constructs an access-top variable order with static width \( w(Q_t) = w(Q) \) and dynamic width \( \delta(Q_t) = \delta(Q) \).

### D.2 Variable Orders Describing Evaluation Strategies

Each variable order of a CQAP query stands for an evaluation strategy for the query. In this section we show how we can derive from the canonical variable of a query a set of variable orders to evaluate the query result on different parts of the input relations.

We start with a high-level explanation of the construction. Consider the canonical variable order \( \omega \) of a hierarchical CQAP query and a subtree \( \omega' \) of \( \omega \) rooted at a variable X. The induced query \( Q_X(O_X|\mathcal{I}_X) \) is defined over the join of the atoms at the leaves of \( \omega' \). The \( \mathcal{I}_X \) consists of the input variables in \( \omega' \) and the root path of X. The set \( O_X \) contains the output variables in \( \omega' \). Let \( \omega'_{\text{at}} \) be an access-top variable order of \( Q_X(O_X|\mathcal{I}_X) \). If \( Q_X \) is CQAP_0, we use \( \omega'_{\text{at}} \) for the evaluation of \( Q_X \). The view tree following \( \omega'_{\text{at}} \) can be constructed in linear time, can be updated in constant time and allows for constant-delay enumeration of the result of \( Q_X \).

We now consider the case that \( Q_X \) is not CQAP_0. In this case, \( \omega' \) must contain a bound or output variable Y such that \( Q_Y \) is not CQAP_0. If X is this variable Y, we recursively process the subtrees of \( \omega' \), otherwise, i.e., if X is this variable Y, we distinguish two cases based on the degree of values over \( \text{anc}_\omega(X) \cup \{X\} \). In the light case, we construct the view tree following the variable order \( \omega'_{\text{at}} \). This view tree can be constructed and maintained under updates efficiently, since the values over \( \text{anc}_\omega(X) \cup \{X\} \) have bounded degree. In the heavy case, we use the variable order \( \omega' \). The view tree following \( \omega' \) allows for constant update time and an enumeration delay that depends on the number of distinct values over \( \text{anc}_\omega(X) \cup \{X\} \). Since these values have high degree, the number of distinct such values is bounded, which ensure efficient enumeration delay.

Given a canonical variable order \( \omega \) of a hierarchical CQAP query \( Q(O|\mathcal{I}) \), the function \( \Omega(\omega, (O|\mathcal{I})) \) in Figure[10] returns the set of all variable orders for \( Q \) obtained from \( \omega \). The atoms at the leaves of these variable
orders are labelled by HL-signatures. When constructing view trees following these variable orders, these atoms will be materialized with corresponding relation parts. That is, an atom \(\text{key} \rightarrow \omega\) of each atom is extended by some part of the query result. The variable order \(\omega\) will be materialized by a part of relation \(R\) that is heavy on \(S\) if \(s = H\) and light on \(S\) if \(s = L\). We assume that the atoms in the initial canonical variable order \(\omega\) passed as input to the function \(\Omega\) are labelled by the empty HL-signature \(\emptyset\).

We now describe the function \(\Omega(\omega, (O|I))\) in more detail. The function proceeds recursively on the structure of \(\omega\) and considers at each variable \(X\), the induced query \(Q_X(O_X|I_X)\) (Line 4). If \(Q_X\) is CQAP, the function returns an access-top variable order constructed by the function \(\text{Acc-Top}(\omega, (O|I))\) in Figure 10 (Lines 5-6). If \(X\) is an input variable, or it is an output variable and \(\omega\) does not contain any input variable, the query \(Q_X\) can be evaluated efficiently given that the induced queries defined at the children of \(X\) are evaluated efficiently. Hence, the function recursively computes a set of variable orders for each child tree of \(X\). For each combination of these variable orders, it builds a new variable order where \(X\) is on top of the child variable orders (Lines 7-8). Otherwise, if \(X\) is bound or an output variable and \(\omega\) contains input variables, the function creates two evaluation strategies for \(Q_X\) based on the degree of values over \(X \cup \text{anc}(X)\). For the values over \(X \cup \text{anc}(X)\) that are heavy, i.e., the degrees of the values are above a given threshold, the function treats \(X\) as an input variable and proceeds recursively to resolve further variables located below \(X\) in the variable order and to potentially fork into more strategies (Line 10). For the values over \(X \cup \text{anc}(X)\) that are light, the function constructs an access-top variable order for \(\omega\) (Line 10).

\[
\begin{array}{ll}
\text{switch } \omega: & \text{return } \{R^\text{sig}(Y)\} \\
\end{array}
\]

\[
\begin{array}{ll}
R^\text{sig}(Y) & 1 \text{ return } \{R^\text{sig}(Y)\} \\
& 2 \text{ let } \text{key} = \text{anc}_\omega(X) \cup \{X\} \\
& 3 \text{ let } I_X = \text{anc}_\omega(X) \cup (I \cap \text{vars}(\omega)) \\
& 4 \text{ let } O_X = O \cap \text{vars}(\omega) \\
& 5 \text{ let } Q_X(O_X|I_X) = \text{join of } \text{atoms}(\omega) \\
& 6 \text{ if } Q_X(O_X|I_X) \text{ is CQAP}_0 \\
& 7 \text{ return } \{\text{Acc-Top}(\omega, (O|I))\} \\
& 8 \text{ else if } X \in I \text{ or } (X \in O \text{ vars}(\omega) \cap I = \emptyset) \\
& 9 \text{ return } \left\{X \ \begin{array}{l}
\omega_1 \cdots \omega_k
\end{array} \ \begin{array}{l}
\omega_i' \in \Omega(\omega_i, (O|I)), \forall i \in [k]
\end{array} \right\} \\
& 10 \text{ else} \\
& 11 \text{ let } \text{htrees} = \left\{X \ \begin{array}{l}
\omega_1 \cdots \omega_k
\end{array} \ \begin{array}{l}
\omega_i' \in \Omega(\omega_i^{\text{key} \rightarrow H}, (O|I)), \forall i \in [k]
\end{array} \right\} \\
& 12 \text{ let } \text{ltree} = \text{Acc-Top}(\omega^{\text{key} \rightarrow L}, (O|I)) \\
& 13 \text{ return } \text{htrees} \cup \{\text{ltree}\}
\end{array}
\]

Figure 10: Construction of a set of variable orders from a canonical variable order \(\omega\) of a hierarchical CQAP query with access pattern \((O|I)\). Each constructed variable order corresponds to an evaluation strategy of some part of the query result. The variable order \(\omega^{\text{key} \rightarrow s}\) for \(s \in \{H, L\}\) has the structure of \(\omega\) but the HL-signature of each atom is extended by \(\text{key} \rightarrow s\).
\( \tau(\text{variable order } \omega) : \text{view tree} \)

\begin{align*}
\text{switch } \omega: \\
R(Y)^{sig} & \quad \text{return } R^{sig}(Y) \\
X & \\
\omega_1 \ldots \omega_k & \\
2 & \text{let } T_i = \tau(\omega_i), \forall i \in [k] \\
3 & \text{let } S = \{X\} \cup \text{dep}_\omega(X) \\
4 & \text{let } V_X(S) = \text{join of roots of } T_1, \ldots, T_k \\
5 & \text{if } X \text{ does not have a sibling} \\
6 & \quad \text{return } \begin{cases} \\
V_X(S) \\
T_1 \cdots T_k \\
\end{cases} \\
7 & \text{else} \\
8 & \quad \text{let } V'_X(S \setminus \{X\}) = V_X(S) \\
9 & \quad \text{return } \begin{cases} \\
V'_X(S \setminus \{X\}) \\
V_X(S) \\
T_1 \cdots T_k \\
\end{cases}
\end{align*}

Figure 11: Construction of a view tree following a variable order \( \omega \). At each variable \( X \) in \( \omega \), the function creates a view \( V_X \) whose schema consists of \( X \) and the dependency set of \( X \). If \( X \) has siblings, it adds a view on top of \( V_X \) that aggregates away \( X \) to enable efficient updates coming from the siblings of \( X \).

\[ \text{ViewTrees}(\text{canonical variable order } \omega, \text{access pattern } (O|I)) : \text{view trees} \]

\[ 1 \quad \text{return } \{ \tau(\omega') \mid \omega' \in \Omega(\omega, (O|I)) \} \]

Figure 12: Construction of all view trees for a canonical variable order \( \omega \) of a hierarchical CQAP query with access pattern \((O|I)\).

### D.3 View Trees Encoding the Query Result

Given a (not necessarily canonical) variable order \( \omega \), the function \( \tau(\omega) \) in Figure 11 returns a view tree constructed following \( \omega \). The function traverses the variable order recursively and creates at each variable \( X \), a view \( V_X \) defined over the join of the child views of \( X \). The schema of \( V_X \) consists of \( X \) and the dependency set of \( X \) (Lines 3-6). The view enables efficient enumeration of \( X \)-values for a fixed value over the dependency set. If \( X \) has siblings, the function creates a view \( V'_X \) on top of \( V_X \) that results from the latter view by aggregating away \( X \) (Lines 8-9). This view enables efficient maintenance of the ancestor views of \( V_X \) under updates coming from the sibling views of \( X \).

The function \( \text{ViewTrees}(\omega, (O|I)) \) in Figure 12 returns the set of all view trees for a hierarchical CQAP query \( Q(O|I) \) with canonical variable order \( \omega \). For each variable order \( \omega' \) returned by \( \Omega(\omega, (O|I)) \) from Figure 10 the function creates the corresponding view tree by calling \( \tau(\omega') \) from Figure 11.

Materializing a view tree consists of computing the relation parts at the leaves and computing the joins defined by the views in the view tree. The preprocessing phase for a hierarchical CQAP query \( Q(O|I) \) with canonical variable order \( \omega \) consists of materializing all view trees in \( \text{ViewTrees}(\omega, (O|I)) \).

The set of view trees constructed for a hierarchical CQAP query in the preprocessing phase encode exactly the query.

**Proposition 32.** Let \( \{T_1, \ldots, T_k\} \) be the set of view trees in \( \text{ViewTrees}(\omega, (O|I)) \) for a hierarchical CQAP
query \( Q(\mathcal{O}|\mathcal{I}) \) and the canonical variable order \( \omega \) for \( Q \). Let \( Q_{T_i}(\mathcal{O}|\mathcal{I}) \) be the query defined by the conjunction of the leaf atoms in \( T_i \). Then, \( Q(\mathcal{O}|\mathcal{I}) \equiv \bigcup_{i \in [k]} Q_{T_i}(\mathcal{O}|\mathcal{I}) \).

**Proof:** The proof is a simplified version of the proof of Proposition 4.3 in [20]. In [20], the view trees contain additional indicator views that maintain heavy values. For the sake of completeness, we give the full proof.

The procedure VIEWTrees calls \( \Omega \) to construct from the input canonical variable order \( \omega \) a set of variable orders \( \omega_1, \ldots, \omega_k \) and constructs the set of view trees \( T_1, \ldots, T_k \) following the variable orders. The variable order \( \omega_i \) and the corresponding view tree \( T_i \) for \( i \in [k] \) have the same leaf atoms. We define \( Q_{\omega'}(\mathcal{O}|\mathcal{I}) = \bigotimes_{R(X) \in \text{atoms}(\omega')} R(\mathcal{X}) \) to be the query defined by the conjunction of the leaf atoms in \( \omega' \).

The proof is by induction over the structure of the variable order \( \omega \). We show that for any subtree \( \omega' \) rooted at \( X \) of \( \omega \), it holds:

\[
Q_{\omega'}(\mathcal{O}|\mathcal{I}_X) = \bigcup_{\omega'' \in \Omega(\omega', (\mathcal{O}|\mathcal{I}_X))} Q_{\omega''}(\mathcal{O}|\mathcal{I}_X),
\]

where \( \mathcal{O}_X = \mathcal{O} \cap \text{vars}(\omega') \) and \( \mathcal{I}_X = \text{anc}(X) \cup (\mathcal{I} \cap \text{vars}(\omega')) \). This completes the proof.

**Base case:** If \( \omega' \) is an atom, the procedure \( \Omega \) returns that atom and the base case holds trivially.

**Inductive step:** Assume that \( \omega' \) has subtrees \( \omega'_1, \ldots, \omega'_k \). Let \( \text{key} = \text{anc}(X) \cup \{X\} \), \( \mathcal{I}_X = \text{anc}(X) \cup (\mathcal{I} \cap \text{vars}(\omega')) \), and \( \mathcal{O}_X = \mathcal{O} \cap \text{vars}(\omega') \). The procedure \( \Omega \) distinguishes the following cases:

1. **Case 1:** \( Q_X(\mathcal{O}_X|\mathcal{I}_X) \) is CQAP. The procedure \( \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X)) \) constructs an access-top variable order with leaves exactly the atoms of \( \omega' \). This implies Equivalence 8.

2. **Case 1 does not hold and \((X \in \mathcal{O} \text{ or } (X \in \mathcal{O} \text{ and vars}(\omega') \cap \mathcal{I} = \emptyset)\)):** The procedure \( \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X)) \) constructs recursively a set of variable orders for each subtree in \( \omega'_1, \ldots, \omega'_k \). Using the induction hypothesis, we rewrite as follows:

\[
Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) = \bigotimes_{i \in [k]} Q_{\omega'_i}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \equiv \bigcup_{\omega'' \in \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))} Q_{\omega''}(\mathcal{O}_X|\mathcal{I}_X)
\]

where \( X' \) is the root of \( \omega'_i \), \( \mathcal{O}_{X'} = \mathcal{O} \cap \text{vars}(\omega') \) and \( \mathcal{I}_{X'} = \text{anc}(X') \cup (\mathcal{I} \cap \text{vars}(\omega')) \).

**Cases 1 and 2 do not hold:** The procedure \( \Omega \) creates the variable orders \( \text{htrees} \cup \{\text{ltree}\} \) defined as follows:

1. **ltree** = ACC-TOP(\( \omega^{\text{key}=L}, (\mathcal{O}_X|\mathcal{I}_X) \)), where \( \omega^{\text{key}=L} \) has the same structure as \( \omega' \) but each atom is replaced by its part that is light on \( \text{key} \);
2. **htrees** are same as the variable orders built in the previous case except each atom is replace by a part that is heavy on \( \text{key} \).

If a relation is partitioned on a set \( \text{key} \) of variables, then the parts of relation that are light and heavy on \( \text{key} \) are disjoint and together form the relation. This drive the following equivalence. For simplicity, we skip the schemas of queries:

\[
\bigcup_{\forall i \in [k]} Q_{T_i} \equiv Q_{\text{ltree} \cup} \bigcup_{\forall i \in [k]} Q_{T_i} \bigcup_{\forall i \in [k]} Q_{T_i} \bigcup_{\forall i \in [k]} Q_{T_i}
\]
Using the induction hypothesis, we obtain:

\[
Q_{\omega'} = \bigotimes_{\iota \in [k]} Q_{\omega'_\iota} \equiv \bigotimes_{\iota \in [k]} \left( \bigcup_{\omega'' \in \Omega_{\omega'_\iota}, (O \setminus I)} Q_{\omega''} \right)
\]

\[
\equiv \bigcup_{\iota \in [k]} \bigotimes_{\omega'' \in \Omega_{\omega'_\iota}, (O \setminus I)} Q_{\omega''}
\]

\[
\equiv Q_{\text{tree}} \cup \bigcup_{\iota \in [k]} \bigcup_{\omega'' \in \Omega_{\omega'_\iota}, (O \setminus I)} Q_{\omega''}
\]

\[
= Q_{\text{tree}} \cup \bigcup_{T \in \text{hrees}} Q_T = \bigcup_{T \in \Omega_{\omega', (O \setminus I)}} Q_T
\]

□

Given a CQAP \( Q \) with static width \( w \) and hierarchical fracture \( Q_I(O \mid I) \), the preprocessing time of our approach is given by the time to materialise the view trees in \( \text{ViewTrees}(\omega, O \mid I) \). The time to materialise these view tree is \( O(N^{1+(w-1)\epsilon}) \), as stated in Proposition 12.

D.4 Proof of Proposition 12

The proof is an extension of the proof of Proposition 4.4 in [20]. It uses the auxiliary Lemma 33 given below. We first explain how Proposition 12 is implied by Lemma 33. Consider a CQAP query \( Q \) with static width \( w \) and hierarchical fracture \( Q_I \) and an \( \epsilon \in [0, 1] \). In the preprocessing stage, we apply for each connected component \( Q'_I(O \mid I) \) of \( Q_I \) the following steps. Let \( \omega \) be the canonical variable order of \( Q'_I \). First, we call the function \( \Omega(\omega, (O \mid I)) \) in Figure 10 which creates a set of variable orders from \( \omega \). For each variable order \( \omega' \) in this set, we call the function \( \tau(\omega') \) in Figure 11 which creates a view tree \( T \) following \( \omega' \). By Lemma 33 the view tree \( T \) can be materialised in \( O(N^{(w/Q'_I)-1}) \) time. Since \( w(Q'_I) \) is upper-bounded by \( w \), this implies \( O(N^{(w-1)\epsilon}) \) overall preprocessing time.

It remains to prove Lemma 33.

Lemma 33. Let \( \omega \) be a variable order of a CQAP query \( Q(O \mid I) \), \( X \) a variable in \( \omega \), \( Q_X \) the induced query at \( X \) in \( \omega \), \( \omega' \in \Omega(\omega_X, (O \mid I)) \), \( \omega' = (\text{anc}_X(X) \circ \omega) \), \( N \) the size of the leaf relations in \( \omega' \), and \( \epsilon \in [0, 1] \). The view tree \( \tau(\omega') \) can be materialised in \( O(N^{1+(w/Q_X)-1}) \) time.

Proof. The proof is by induction on the structure of \( \omega_X \). We show that for each variable \( Y \) in \( \omega' \), the view \( V_Y \) in \( \tau(\omega') \) as defined in Line 4 of the procedure \( \tau \) can be materialised in \( O(N^{1+(w/Q_X)-1}) \) time. Each auxiliary view defined in Line 8 of the procedure \( \tau \) results from its child view by marginalising a single variable. Materialising these auxiliary views does not increase the overall asymptotic computation time.

Base case: Assume that \( \omega_X \) is a single atom. In this case, the procedure \( \Omega \) returns this atom. The atom can obviously be materialised in \( O(N) \) time. Hence, the statement in the lemma holds.

Inductive step: Assume that the root variable \( X \) in \( \omega_X \) has the child nodes \( X_1, \ldots, X_k \). Let \( \text{key} = \text{anc}_X(X) \cup \{X\}, \mathcal{I}_X = \text{anc}_X(X) \cup (\mathcal{I} \cap \text{vars}(\omega_X)), O_X = \mathcal{O} \cap \text{vars}(\omega) \). The induced query at \( X \) is defined as \( Q_X(O \mid I) = \text{join of atoms}(\omega) \). Following the control flow in \( \Omega \), we distinguish between the following cases.

Case (1): \( Q_X(O \mid I) \) is a CQAP query. In this case, the procedure \( \Omega \) returns the variable order \( \omega' = \text{Acc-Top}(\omega_X, (O \mid I)) \). By Proposition 30 \( \omega' = (\text{anc}_X(X) \circ \omega) \) is an access-top variable order for \( Q_X \) with static width \( w(Q_X) \). Since \( Q_X \) is in CQAP, its static width can be at most 1 (Propositions 27 and 28). This means that for every variable \( Y \in \text{vars}(\omega' \circ X) \), the set \( \{Y\} \cup \text{dep}_\omega(Y) \) can be covered by a single atom in \( Q_X \). Hence, each view \( V_Y (\{Y\} \cup \text{dep}_\omega(Y)) \) can be computed in \( O(N) \) time. This completes the inductive step for Case (1).
Case (2): \( Q_X \) is not in CQAP\(_0\) and \( \{ X \in \mathcal{I} \text{ or } (X \in \mathcal{O} \text{ and } \text{vars}(\omega) \cap \mathcal{I} = \emptyset) \} \)

The set of variable orders returned by \( \Omega \) is defined as follows: For each set \( \{ \omega_i \}_{i \in [k]} \) with \( \omega_i \in \text{vars}(\mathcal{I}) \), the set contains a variable order \( \omega' \) of root node \( X \) and child trees \( \omega_1, \ldots, \omega_k \). Consider for each such variable order \( \omega' \) the variable order \( \omega^i = (\text{anc}_\omega(X) \circ \omega') \). By induction hypothesis, each view tree over \( \omega_i \) can be materialised in \( O(N^{1+(w(Q_X)-1)\epsilon}) \). Since \( w(Q_X) \leq w(\omega_X) \) for any \( i \in [k] \), it follows that each view tree over \( \omega_i \) can be materialised in \( O(N^{1+(w(Q_X)-1)\epsilon}) \). Consider now the view tree \( \tau(\omega^i) \). The view at \( X \) is defined by \( V_X(S) = V_{X_1}(S_1), \ldots, V_{X_k}(S_k) \), where \( S = \{ X \} \cup \text{dep}_\omega(X) \) and \( V_{X_1}, \ldots, V_{X_k} \) are the child views of \( V_X \). By the construction of view trees, \( V_X \) is a free-connex query. Hence, it can be computed by first marginalising the variables in \( V_X \), that are not included in \( S \) for each \( i \in [k] \) and then computing the intersection of the remaining relations. This gives overall \( O(N^{1+(w(Q_X)-1)\epsilon}) \) computation time. This completes the inductive step in this case.

Case (3): \( Q_X \) is not in CQAP\(_0\) and \( X \) is an output variable dominating an input variable or it is a bound variable dominating a free variable.

In this case, the procedure \( \Omega \) constructs a set \( htrees \) of variable orders and a single variable order \( ltree \). The construction of the variable orders in \( htrees \) differs from the variable orders constructed under Case (2) only in that they refer to base relations that are heavy on the variable set \( key \). This does not affect the asymptotic computation time of the view trees. Hence, the view trees over the variable orders \( htrees \) can be computed in \( O(N^{1+(w(Q_X)-1)\epsilon}) \) time. The variable order \( ltree \) is defined as \( ltree = \text{Acc-Top}(\omega^i_{key-L}, \mathcal{O}(\mathcal{I})) \), where \( \omega^i_{key-L} \) indicates that the base relations are light on \( key \). Observe that \( key \) is included in the schemas of the leaf atoms of \( ltree \). By Proposition\[\text{[20]}\] \( ltree \) is an access-top variable order for \( Q_X \) with optimal static width. Then, it follows from Lemma\[\text{[34]}\] that the view tree \( \tau(ltree) \) can be materialised in \( O(N^{1+(w(Q_X)-1)\epsilon}) \) time. This completes the inductive step for Case 3.

We used the following lemma in the above proof.

**Lemma 34.** Let \( \omega \) be a variable order, \( X \) a variable in \( \omega \) such that \( \text{anc}_\omega(X) \) is included in the schemas of all leaf atoms in \( \omega_X \) and \( \omega' = (\text{anc}_\omega \circ \omega_X) \). If the leaf relations in \( \omega_X \) are the light parts of a partition on \( \{ X \} \cup \text{anc}_\omega(X) \) with threshold \( O(N^\epsilon) \) for some \( \epsilon \in [0,1] \), the view tree \( \tau(\omega') \) can be materialised in \( O(N^{1+(w(\omega')-1)\epsilon}) \) time.

**Proof.** Let \( T = \tau(\omega') \) and \( w = w(\omega') \). We show: every view in \( T \) can be computed in \( O(N^{1+(w-1)\epsilon}) \) time.

The leaf atoms can obviously be materialised in \( O(N) \) time. Consider any view \( V_Y(S) \) in \( T \) with \( \text{atoms}(\omega') = \{ R_i(X_i) \}_{i \in [k]} \). The view \( V_Y \) is defined over the join of its child views and it holds \( S = \{ Y \} \cup \text{dep}_\omega(Y) \). By the construction of our view trees, \( V_Y \) can be computed by joining the atoms \( R_i(X_i), \ldots, R_k(X_k) \). Hence, we can write the view as

\[
V_Y(S) = R_1(X_1), \ldots, R_k(X_k).
\]

Let \( \rho^*(S) = m \). By Lemma\[\text{[20]}\] \( \rho(S) = m \). We can find an optimal edge cover for \( S \) by using only atoms from the set \( \{ R_i(X_i) \}_{i \in [k]} \). Let \( \lambda = (\lambda_{R_i(X_i)})_{i \in [k]} \) be an edge cover of \( S \) with \( \sum_{i \in [k]} \lambda_{R_i(X_i)} = m \). Let \( \mathcal{R}_0, \mathcal{R}_1 \subseteq \text{atoms}(\omega_X) \) consist of the atoms in \( \omega_X \) that \( \lambda \) assigns to 0 and 1, respectively. We first compute a view \( V(S) \) over the join of the atoms in \( \mathcal{R}_1 \) as follows. We choose an arbitrary atom from \( \mathcal{R}_1 \) and iterate over its tuples. For each such tuple \( t \), we iterate over the matching tuples in the other atoms in \( \mathcal{R}_1 \). Since each atom in \( \mathcal{R}_1 \) includes \( \text{anc}_\omega(X) \) in its schema and is the light part of a partition on \( \text{anc}_\omega(X) \) with threshold \( O(N^\epsilon) \), it contains \( O(N^\epsilon) \) tuples matching \( t \). This means that the time to materialise \( V \) is \( O(N \cdot N^{(m-1)\epsilon}) = O(N^{1+(m-1)\epsilon}) \). Now, we can rewrite \( V_Y \) using the new view \( V \):

\[
V_Y(S) = V(S), R'_1(X'_1), \ldots, R'_k(X'_k),
\]

where \( R'_1(X'_1), \ldots, R'_k(X'_k) \) are the atoms in \( \mathcal{R}_0 \). The query \[\text{[10]}\] is free-connex \( \alpha \)-acyclic, which means that it can be computed in time linear in the input plus the output size of \( V_Y \), using Yannakakis’s algorithm\[\text{[34]}\]. The input size is upper-bounded by \( |V| = O(N^{1+(m-1)\epsilon}) \). The size of the output is also \( O(N^{1+(m-1)\epsilon}) \). Hence, the overall time to compute \( V_Y \) is \( O(N^{1+(m-1)\epsilon}) \). Since \( m = \rho^*(S) \) is upper-bounded by \( w \), we derive
that the computation time for $V$ is $O(N^{1+\omega-1}e)$. Each of the additional auxiliary views constructed in Line 8 of the procedure $\tau$ is obtained by marginalising away a variable from its child view. This does not blow up the overall asymptotic computation time.

E Missing Details in Section 5

In this section, we provide more details about view iterators, generalised view iterators, and the enumeration procedure, as well as the proofs of the proposition from Section 5.

View Iterators. A view iterator allows the enumeration of values from a materialized view using the standard iterator interface with open and next methods. We write $\text{it}_V(O|I)$ to denote a view iterator $\text{it}$ over a view $V$ with schema $\{O\} \cup I$, where $O$ is the output variable and $I$ is the context schema of the iterator.

The open($ctx$) method takes the tuple $ctx$ as input, requiring that all $O$-values returned via next() are paired with $ctx$ in $V$. We also write $\text{it}_V(O|I).\text{contains}(o)$ to check if the given value $o$ can appear in the output of the $\text{it}_V$ iterator; this is syntactic sugar for the membership test $ctx \circ o \in V$, where $\circ$ denotes tuple concatenation. All the three methods, open, next, and contains, take constant time as per the computational model from Appendix B.

Example 35. Consider a materialized view $V(A, B)$. The iterator $\text{it}_V(B|A)$ enumerates the distinct $B$-values paired with a given $A$-value in $V$. The iterator $\text{it}_V(B|A, B)$ returns the $B$-value in a given $(A, B)$-tuple if the tuple exists in $V$; otherwise, it returns $EOF$. The iterator $\text{it}_V(A)$ is invalid as its output variable $A$ and context schema $\emptyset$ do not match the schema of $V$, i.e., $\{A\} \cup \emptyset \neq \{A, B\}$.

Generalised View Iterators. We next present the open($ctx$) and next() methods of a generalised view iterator $\text{git}_V(O|I)$.

Figures 13 shows the open($ctx$) method, which takes as input a relation $ctx$ over $I$ and creates one view iterator for each tuple in $ctx$. Each view iterator is opened with their corresponding tuple as context. The context tuples and view iterators are stored in the attribute iterators of mapping type. The open($ctx$) method takes time linear in the size of the relation $ctx$, that is, $O(|ctx|)$.

The next() method uses the UNION algorithm from Figure 14 to fetch the next distinct output value from a list of iterators. The algorithm is an adaptation of prior work [11]. It takes as input $n$ iterators with the same output schema, which enumerate values from possibly overlapping sets, and returns a value in the union of these sets, where the value is distinct from all values returned before. Upon each call, the function returns one value. If all iterators are exhausted, the function returns $EOF$.

We first explain the union algorithm on two iterators $\text{it}_1$ and $\text{it}_2$. Given the next value $v_1$ of $\text{it}_1$, the algorithm calls $\text{it}_2.\text{contains}(v_1)$ to check if $v_1$ can be enumerated by $\text{it}_2$. If so, it returns the next value in $\text{it}_2$; otherwise, it returns $v_1$. If $\text{it}_1$ is exhausted, the function returns the next value in $\text{it}_2$ or $EOF$ if $\text{it}_2$ is also exhausted.

For $n > 2$ iterators, the algorithm considers the union of the first $n - 1$ iterators as the next value of one iterator and $\text{it}_n$ as the second iterator, and then reduces the general case to the previous case of two iterators. The algorithm invokes next() and performs a membership test on $n$ iterators, each taking constant time. Thus, fetching the next output value takes $O(n)$ time.

Figure 15 shows the next() method. For each output value $o$ obtained using the UNION algorithm, next() computes a set of tuples over schema $I$ that are paired with $o$ in $V$. Assuming $\text{git}_V(O|I)$ is opened for a relation $ctx$, fetching the output value $o$ and computing the set of tuples for $o$ each take $O(|ctx|)$ time. Thus, the next() method also runs in $O(|ctx|)$ time.

Enumeration Procedure. The function BUILDITERATORS from Figure 16 builds a list of generalised view iterators for a given view tree of a CQAP query $Q$ with access pattern $O|I$. Each generalised view iterator comes paired with a support relation that provides the context for any variable with no binding. The support provided in the initial call to BUILDITERATORS is the singleton relation with the empty tuple (the identity for the join operation).
```plaintext
Figure 13: Open the generalised view iterator \( \text{git}_V(O|I) \) with the relation \( ctx \) over schema \( I \) as context.

```plaintext
```plaintext
Figure 14: Fetch the next distinct value from a list of iterators.

The function recursively constructs generalised view iterators, traversing the view tree \( T \) in a top-down fashion. Consider the root view \( V_X(X) \) of \( T \) constructed at variable \( X \) in the corresponding variable order. If \( X \not\in X \), then \( V_X \) is an auxiliary view that allows for efficient maintenance under updates (c.f. Figure 11) but has no role in enumeration, thus we recur on its child. The function creates a generalised view iterator over \( V_X \) if \( X \) is a free variable. Otherwise, if \( X \) is a bound variable, it uses \( V_X \) as the support relation for any generalised view iterator created for a free variable below \( X \). The function recursively creates iterators in each subtree and concatenates them into a list of iterators with their support relation.

**Example 36.** Consider the view tree from Figure 3 (second from left) for the query \( Q_1(B, C, D | A_1) = R(A_1, B, C), S(A_1, B, D) \). BuildIterators returns the following generalised view iterators for this view tree: \( \text{git}_{V_{A_1}}(A_1|A_1), \text{git}_{V_B}(B|A_1), \text{git}_{V_C}(C|A_1, B), \) and \( \text{git}_{V_D}(D|A_1, B) \), each paired with the support \{\}. 

Consider now the view tree from Figure 4 (bottom-right), which is created for the connected component \( Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D) \). BuildIterators returns the following iterators for this view tree: \( \text{git}_{V_{A_1}}(A_1|A_1) \) with the support \{\}, \( \text{git}_{V_C}(C|A_1, B, C) \) with the support \( V_B(A_1, B) \), and \( \text{git}_{V_D}(D|A_1, B) \) with the support \( V_B(A_1, B) \).

The returned support relations define the context to be used when opening each generalised view iterator, as shown in Figure 6.

**E.1 Proof of Proposition 15**

We want to show that for any CQAP query, its distinct output tuples given an input tuple can be enumerated with \( O(1) \) delay.
Figure 15: Fetch the next output value from the generalised view iterator \( \text{git}_V(O|I) \) together with the set of tuples over schema \( I \) that are paired with that output value in \( V \).

```plaintext
let \{ t_1 \cdot \text{it}_1, \ldots, t_n \cdot \text{it}_n \} = \text{git}_V(O|I).\text{iterators}

o := \text{UNION}(\text{it}_1, \ldots, \text{it}_n)
ctx_o := \{ t_i \mid i \in [n], \text{it}_i.\text{contains}(o) \}
return (o, ctx_o)
```

Figure 16: Create a list of generalised view iterators with support for the access pattern \((O|I)\) in a view tree \( T \). The first call to \text{BuildIterators} uses the support \{()\}.

```plaintext
\text{BuildIterators}(\text{view tree } T, \text{access pattern } (O|I), \text{relation supp})

switch T:

\[ R(Y) \]
1 return []

\[ V_X(X) \]
2 if \( X \notin X' \) // skip auxiliary maintenance views
3 return \text{BuildIterators}(T_1, (O|I), supp)
\[ T_1 \cdots T_k \]
4 \( \text{it}_X = \begin{cases} \{ \text{new git}_V(X|X), \text{supp} \} & \text{if } X \in I \\ \{ \text{new git}_V(X|X \setminus \{X\}), \text{supp} \} & \text{if } X \in O \\ [] & \text{otherwise} \end{cases} \)
5 \text{supp}_\text{child} = \begin{cases} \text{supp} & \text{if } X \in (I \cup O) \\ V_X(X) & \text{otherwise} \end{cases}
6 \text{it}_\text{child}_i = \text{BuildIterators}(T_i, (O|I), \text{supp}_\text{child}), \forall i \in [k]
7 return \text{it}_X ++ \text{it}_\text{child}_1 ++ \ldots ++ \text{it}_\text{child}_k
```

The fracture of any CQAP\(_0\) query with access pattern \((O|I)\) is hierarchical, \((O \cup I)\)-dominant, and \(I\)-dominant, per Definition 7. For each connected component of the fracture, we can construct a variable order where the free variables are above the bound variables and the input variables are above the output variables, see the \( \Omega \) function from Figure 10. For the view tree constructed following that variable order, we can create a list of view iterators by doing a pre-order traversal of the view tree such that the iterators for input variables precede those for output variables in the list. By forming a nesting chain of these iterators, we can enumerate the distinct output tuples for the given input tuple with constant delay.

If the fracture consists of several connected components, we concatenate the list of iterators constructed for each connected component and form a nesting chain for the enumeration from their view trees.

**E.2 Proof of Proposition [17]**

We now sketch the proof that for any hierarchical CQAP query \( Q \), database of size \( N \), and \( \epsilon \in [0, 1] \), the distinct output tuples given an input tuple can be enumerated with \( O(N^{1-\epsilon}) \) delay.

Consider a CQAP query \( Q \) with hierarchical fractures. If \( Q \) is in CQAP\(_0\), the distinct output tuples can be enumerated with \( O(1) \) delay, per Proposition [15]. Otherwise, there exists a variable \( X \) such that either \( X \) is a bound variable and above a free variable or \( X \) is an output variable and above an input variable in the canonical variable order of \( Q \). For each such case, we partition the relations in the subtree rooted at \( X \)
and create different evaluation strategies over the heavy and light relation parts, see Figure 10.

In the light case, the created view trees follow access-top variable orders, thus admitting constant delay enumeration of the output tuples for a given input tuple. In the heavy case, the view defined $X$ consists of at most $N^{1-\epsilon}$ heavy values, which define the support for the enumeration from child views. This size of the support determines the enumeration delay.

**F Missing Details in Section 6**

We present our strategy for maintaining the view trees returned by the function $\text{VIEWTrees}(\omega, (O|L))$ (Figure 12) for a canonical variable order $\omega$ of a hierarchical CQAP query $Q((O|L))$ under updates to base relations. We write $\delta R = \{x \to m\}$ to denote a single-tuple update to a base relation $R$ mapping the tuple $x$ to the non-zero multiplicity $m \in \mathbb{Z}$ and any other tuple to 0; i.e., $|\delta R| = 1$. Inserts and deletes are updates represented as relations in which tuples have positive and negative multiplicities, respectively. We assume that after applying an update, all relations and views contain no tuples with negative multiplicities.

The function $\text{VIEWTrees}(\omega, (O|L))$ in Figure 12 constructs view trees that have relation parts at their leaves. In Section F.1 we describe how to determine the part of a base relation that is affected by an update. Several view trees can refer to the same relation part. To simplify the reasoning about the maintenance task, we assume that each view tree has a copy of its relation parts. We explain in Section F.2 how to apply a single-tuple update to a set of view trees. As the database evolves under updates, we periodically rebalance the relation partitions and views to account for new database sizes and updated degrees of values. Section F.3 describes how to intertwine a sequence of single-tuple updates with rebalancing steps.

**F.1 Determining the Relation Part of a Tuple**

Given an update $\delta R = \{x \to m\}$, we have to find out which part of relation $R$ is affected by the update. That is, we need to compute the HL-signature of the part of $R$ on which the update is to be applied. The function $\text{TRANSIENTHLs}(x)$ in Figure 17 constructs an HL-signature by checking in which relation parts the values in $x$ are contained. The set $\text{PARTITIONKEYS}$ in the definition of the function consists of all keys the input relations are partitioned on. $\text{PARTITIONKEYS}$ consists of $A$ and $(AB)$. The function first creates an HL-signature $\{k_1 \to s_1, \ldots, k_n \to s_n\}$ where each $k_i$ is included in $\text{PARTITIONKEYS}$ and is a subset of the schema of $x$. If there exists a relation part $K^{\text{sig}}$ such that $x[k_i]$ is included in the projection of $K^{\text{sig}}$ onto $k_i$, $s_i$ is defined as the symbol the key $k_i$ is mapped to in $\text{sig}$ (first case in Line 3). Otherwise, $x[k_i]$ does not exist in the database yet, so it is light. Thus, in this case $s_i$ is defined as $L$ (first case in Line 3). Recall that our preprocessing stage does not further partition a relation on a key $k$ if the relation is already light on a subset of $k$. Hence, the function $\text{REMOVEHEAVYTAIL}$ called in the last line of $\text{TRANSIENTHLs}$ and defined
RemoveHeavyTail(HL-signature sig) : HL-signature

1. let \( \{k_1 \rightarrow s_1, \ldots, k_n \rightarrow s_n\} = \text{sig} \)
2. heavyTail = \( \emptyset \)
3. foreach \( i \in [n] \)
4. if \( \exists j \in [n] \) s.t. \( s_j = L \) and \( k_j \subset k_i \)
5. \( \text{heavyTail} = \text{heavyTail} \cup \{k_i \rightarrow s_i\} \)
6. return \( \text{sig} \setminus \text{heavyTail} \)

Figure 18: Deletion of the heavy tail from an HL-signature \( \text{sig} \). If \( k \rightarrow L \) and \( k' \rightarrow H \) are included in \( \text{sig} \) and \( k \) is a proper subset of \( k' \), then \( k' \rightarrow H \) is deleted from \( \text{sig} \).

ActualHLS(tuple x, threshold \( \theta \)) : HL-signature

1. let \( \{k_1, \ldots, k_n\} = \{k \mid k \in \text{PartitionKeys}, k \subseteq \text{Sch(x)}\} \)
2. let \( s_i = \begin{cases} L, & \text{if } \forall K \in D: |\sigma_{k_i = x[K]} | < \theta \text{ for } i \in [n] \\ H, & \text{otherwise} \end{cases} \)
3. return RemoveHeavyTail(\( \{k_1 \rightarrow s_1, \ldots, k_n \rightarrow s_n\} \))

Figure 19: Computing a HL-signature for tuple \( x \) by checking the degrees of the values in \( x \) based on the threshold \( \theta \).

in Figure 18 removes from \( \text{sig} \) all pairs \( k \rightarrow s \) such that there is \( k' \rightarrow L \) in \( \text{sig} \) with \( k' \subset k \). We call the HL-signature constructed by TransientHLS(\( x \)) the transient HL-signature of \( x \).

When constructing relation parts from scratch, we determine the part a tuple needs to be included based on the degrees of the values in the tuple. Given a tuple \( x \) and a threshold \( \theta \), the function ActualHLS(\( x, \theta \)) in Figure 19 computes an HL-signature \( \text{sig} \) based on \( \theta \). If the degree of the projection of \( x \) onto a partition key is below \( \theta \) in all input relations, \( \text{sig} \) maps the partition key to \( L \) (first case in Line 2). Otherwise, the partition key is mapped to \( H \) (second case in Line 2). The HL-signature constructed by ActualHLS(\( x, \theta \)) is called the transient HL-signature of \( x \) based on \( \theta \).

F.2 Processing a Single-Tuple Update

Given a set \( \mathcal{T} \) of view trees and an update \( \delta R = \{x \rightarrow m\} \), the procedure UpdateTrees(\( \mathcal{T}, \delta R \)) in Figure 20 maintains the view trees under the update. It first computes the transient HL-signature \( \text{sig} \) of \( x \) (Line 2). Then, it applies \( \delta R^{\text{sig}} = \{x \rightarrow m\} \) to the view trees in \( \mathcal{T} \) (Line 2). There might be several view trees constructed in our preprocessing stage that refer to \( R^{\text{sig}} \).

The function Apply(\( T, \delta R^{\text{sig}} \)) in Figure 21 propagates the update \( \delta R^{\text{sig}} \) in the view tree \( T \) from the leaf \( R^{\text{sig}} \) to the root view. For each view on this path, it updates the view result with the change computed using the standard delta rules [7]. If \( T \) does not refer to \( R^{\text{sig}} \), the procedure has no effect.

We next state the complexity of a single-tuple update in our approach.

Proposition 37. Given a hierarchical CQAP query \( Q(O|I) \) with dynamic width \( \delta \), a canonical variable order \( \omega \) for \( Q \), a database of size \( N \), and \( c \in [0,1] \), maintaining the view trees in ViewTrees(\( \omega, (O|I) \)) under a single-tuple update to any input relation takes \( O(N^{3c}) \) time.

Proof. The proof is an extension of the proof of Proposition 6.2 in [20]. In the preprocessing stage, for a CQAP query \( Q \) with input variables \( I \), output variables \( O \), canonical variable order \( \omega \) and delta width \( \delta \), we construct variable orders \( \Omega(\omega, (O|I)) \) and then construct view trees following these variable orders using the
that joins the child views. By construction, the schema \( X \) of the sibling views. Overall, propagating the update through the view tree constructed for a CQAP query takes constant time.

While propagating an update through the view tree, the delta for each view \( X \) constructed variable orders to \( X \) and same as discussed above, if \( X \) is already included in a part of \( R \), all view trees referring to that part are updated. Otherwise, the HL-signature \( sig \) of \( x \) is computed and all view trees referring to \( R \) are updated.

**Figure 20:** Updating a set \( T \) of view trees for a single-tuple update \( \delta R = \{ x \to m \} \) to relation \( R \). If \( x \) is already included in a part of \( R \), all view trees referring to that part are updated. Otherwise, the HL-signature \( sig \) of \( x \) is computed and all view trees referring to \( R \) are updated.

**Figure 21:** Updating views in a view tree \( T \) for a single-tuple update \( \delta R \) to relation part \( R \). If \( R \) is a leaf of \( T \), the function updates \( R \) and its ancestor views in a bottom-up fashion and returns the change of the root view. Otherwise, the empty set is returned.

procedure \( \tau \). The procedure \( \Omega \) traverses the variable order \( \omega \) in a top-down manner. Consider any subtree \( \omega' \) of \( \omega \) rooted at \( X \) and the residual query \( Q_X \) at \( X \) in \( \omega \). The procedure \( \Omega \) distinguishes different cases.

In case the residual query \( Q_X \) is in CQAP\( _0 \), \( \Omega \) creates an access-top variable order \( \omega'_{at} \) for \( \omega' \). At each node \( X \) of \( \omega'_{at} \), \( \tau \) creates a view \( V_X \) with schema \( \{ X \} \cup \text{dep}_{\omega'_{at}}(X) \) that joins the child views below. By construction, if \( X \) has only one child \( Y \) in \( \omega'_{at} \), the child view \( V_Y \) created at \( Y \) below \( V_X \) has the schema \( \{ X, Y \} \cup \text{dep}_{\omega'_{at}}(X) \) and \( V_X \) is computed by variable marginalisation, otherwise, i.e., \( V_X \) has multiple child views, these child views have the same schema \( \{ X \} \cup \text{dep}_{\omega'_{at}}(X) \) as \( V_X \). Consider an update \( \delta R \) to a relation \( R \). The update \( \delta R \) fixes the values of all variables on the path from the leaf \( R \) to the root to constants. While propagating an update through the view tree, the delta for each view \( V_X \) requires joining the update with the sibling child views of \( X \). Each of these sibling child views (if exists) has the same schema as view at \( X \), as discussed above. Thus, computing the delta at each node makes only constant-time lookups in the sibling views. Overall, propagating the update through the view tree constructed for a CQAP\( _0 \) residual query takes constant time.

We now discuss the case \( Q \) is not in CQAP\( _0 \). If \( X \) is an input variable, or \( X \) is an output variable and its ancestors have no input variables, the \( \Omega \) procedure traverses to the subtrees of \( \omega' \) and attaches the constructed variable orders to \( X \). The \( \tau \) procedure creates a view \( V_X \) at \( X \) with the schema \( \{ X \} \cup \text{dep}_{\omega'}(X) \) that joins the child views. By construction, the schema \( \{ X \} \cup \text{dep}_{\omega'}(X) \) is covered by the any atom of \( \omega' \), and same as discussed above, if \( X \) has only one child \( Y \) in \( \omega'_{at} \), the child view \( V_Y \) created at \( Y \) below \( V_X \) has the schema \( \{ X, Y \} \cup \text{dep}_{\omega'_{at}}(X) \) and \( V_X \) is computed by variable marginalisation, otherwise, i.e., \( V_X \)
are distributed in \( \delta \) with \( \delta \) for \( V \).

Major Rebalancing.

As the database size and degrees of data values change under updates, we periodically rebalance the relation parts with the new threshold \( M \), followed by strictly repartitioning the relation parts with the new threshold \( M \) and recomputing the views. Figure 22 shows the major rebalancing procedure. For any base relation \( K \) and tuple \( x \) contained in \( K \), the procedure computes the HL-signature \( \text{sig} \) of \( x \) based on the threshold \( \theta \) and inserts \( x \) into \( K^{\text{sig}} \) (Line 3). It then recomputes all views in the view trees (Line 4).

**Proposition 38.** Given a hierarchical CQAP query \( Q(O|I) \) with static width \( w \), a canonical variable order \( \omega \) for \( Q \), a database of size \( N \), and \( \epsilon \in [0,1] \), major rebalancing of the views in the view trees in \( \text{VIEWTREES}(\omega,(O|I)) \) takes \( O(N^{1+(w-1)\epsilon}) \) time.

---

**F.3 Processing a Sequence of Single-Tuple Updates**

As the database size and degrees of data values change under updates, we periodically rebalance the relation partitions. The cost of rebalancing is amortised over a sequence of updates. Rebalancing and amortisation follow closely prior work [20]. In [20], an update to a relation \( R \) can require to move tuples between the parts of \( R \) only. The partitioning strategy this work takes the degrees of values in all relations into account. This means that an update to \( R \) that refers to a tuple \( t \) can require to move tuples that share partition keys with \( t \) but are not necessarily contained in \( R \). For the sake of completeness, we summarise our rebalancing and amortisation strategies.

**Major Rebalancing.** We loosen the partition threshold to amortise the cost of rebalancing over multiple updates. Instead of the actual database size \( N \), the threshold now depends on a number \( M \) for which the invariant \( \frac{1}{4}M \leq N < M \) always holds. If the database size falls below \( \frac{1}{4}M \) or reaches \( M \), we perform major rebalancing, where we halve or respectively double \( M \), followed by strictly repartitioning the relation parts with the new threshold \( M \) and recomputing the views. Figure 22 shows the major rebalancing procedure. For any base relation \( K \) and tuple \( x \) contained in \( K \), the procedure computes the HL-signature \( \text{sig} \) of \( x \) based on the threshold \( \theta \) and inserts \( x \) into \( K^{\text{sig}} \) (Line 3). It then recomputes all views in the view trees (Line 4).
**Proof.** Consider the major rebalancing procedure from Figure 22. The relation parts can be computed in $O(N)$ time. Proposition 12 implies that the affected views can be recomputed in time $O(N^{1+(w-1)\epsilon})$. □

The cost of major rebalancing is amortised over $O(M)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}M$ (after doubling), or $N = \frac{1}{4}M - \frac{1}{2}$ or $N = \frac{1}{4}M - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}M$. The amortised $O(N^{(w-1)\epsilon})$ time of major rebalancing. By Proposition 25, we have $\delta = w$ or $\delta = w - 1$; hence, the amortised major rebalancing time is $O(M^w)$.

**Minor Rebalancing.** After an update $\delta R = \{x \rightarrow m\}$ to relation $R$, we check the degrees of the values in $x$. Consider a partition key $k$ that is included in the schema of $x$ and the projection $v$ of $x$ onto $k$. If $v$ is included in a relation part that is light on $k$ but the degree of $v$ is not below $\frac{1}{2}M'$ in at least one base relation, all tuples including $v$ are moved to relation parts that are heavy on $v$. Likewise, if $v$ is in a relation part that is heavy on $k$ but the degree of $v$ is below $\frac{1}{2}M'$ in all base relations, all tuples including $v$ are moved to relation parts that are light on $v$. Figure 23 shows the minor rebalancing procedure that moves tuples including $v$ to relation parts whose HL-signature matches the degree of $v$ in the base relations. For each tuple $x$ in a relation part $K^{sig}$, it first computes the actual HL-signature $sig'$ of $x$ based on the threshold $\theta$ (Line 4). It then inserts $x$ into $K^{sig'}$ (Line 5) and deletes it from $K^{sig}$ (Line 6).

**Proposition 39.** Given a hierarchical CQAP query $Q(O[I])$ with dynamic width $\delta$, a canonical variable order $\omega$ for $Q$, a database of size $N$, and $\epsilon \in [0,1]$, minor rebalancing of the views in the view trees in $\text{VIEWTREES}(\omega, (O[I]))$ takes $O(N^{(\delta+1)\epsilon})$ time.

**Proof.** Figure 23 shows the procedure for minor rebalancing of tuples containing the given value $v$ to relation parts whose signature matches the degree of $v$ in base relations. Minor rebalancing either moves $O(\frac{1}{2}M')$ tuples that have $v$ to relation parts that are heavy on $v$ (light to heavy) or $O(\frac{1}{2}M')$ tuples that have $v$ to relation parts that are light on $v$ (heavy to light). Each move is by a insert followed by a delete, which takes $O(N^{(\delta+1)\epsilon})$ time, as discussed in the proof of Proposition 37. Since there are $O(M')$ such moves and the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ holds, the total time is $O(N^{(\delta+1)\epsilon})$. □

The cost of minor rebalancing is amortised over $O(M')$ updates. This lower bound on the number of updates is due to the gap between the two thresholds in the heavy and light part conditions. The amortised $O(N^{(\delta+1)\epsilon})$ time of minor rebalancing.

Figure 24 gives the trigger procedure ONUPDATE that maintains a set $T$ of view trees under a sequence of single-tuple updates to input relations. It first applies an update $\delta R = \{x \rightarrow m\}$ to the view trees from $T$ using $\text{UPDATETREES}$ from Figure 20 (Line 1). If this update leads to a violation of the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$, it invokes $\text{MAJORREBALANCING}$ to recompute the relation parts and views (Lines 2-7). Otherwise, it computes the transient HL-signature $\{k_1 \rightarrow \sigma_1, \ldots, k_n \rightarrow \sigma_n\}$ of $x$ (Line 10). If for any $s_i$, we have $s_i = L$ but there exists a relation such that the degree of $x[k_i]$ is at least $\frac{1}{2}M'$; or it holds $s_i = H$ but the degree of $x[k_i]$ is below $\frac{1}{2}M'$ in all relations, it invokes $\text{MINORREBALANCING}$ to move all tuples.
**ONUPDATE(view trees \( T \), update \( \delta R \))**

1. **UpdateTrees**\((T, \delta R)\)
2. if \(|D| = M\)
   3. \(M = 2M\)
3. **MajorRebalancing**\((T, M')\)
4. else if \(|D| < \lfloor \frac{1}{2}M \rfloor\)
   5. \(M = \lfloor \frac{1}{2}M \rfloor - 1\)
6. **MajorRebalancing**\((T, M')\)
7. else
   8. let \(\delta R = \{ x \rightarrow m \}\)
   9. let \(\{ k_1 \rightarrow s_1, ..., k_n \rightarrow s_n \} = \text{TransientHLS}(x)\)
10. foreach \(i \in [n]\) do
   11. if \((s_i = L \text{ and } \exists K \in D: |\sigma_{k_i = x[k_i]}K| \geq \frac{3}{2}M'\) or \((s_i = H \text{ and } \forall K \in D: |\sigma_{k_i = x[k_i]}K| < \frac{1}{2}M')\)
   12. **MinorRebalancing**\((T, x[k_i], M')\)

Figure 24: Updating a set of view trees \( T \) under a sequence of single-tuple updates to base relations. \( D \) is the database. The global variable \( M \) is set to \( 2|D| + 1 \) in the preprocessing stage.

containing \(x[k_i]\) to the relation parts whose HL-signature matches the degree of \(x[k_i]\) in base relations (Lines 11-14).

We state the amortised maintenance time of our approach under a sequence of single-tuple updates.

**Proposition 40.** Given a hierarchical CQAP query \(Q(O|I)\) with dynamic width \(\delta\), a canonical variable order \(\omega\) for \(Q\), a database of size \(N\), and \(\epsilon \in [0, 1]\), maintaining the views in the view trees in \(\text{ViewTrees}(\omega, (O|I))\) under a sequence of single-tuple updates takes \(O(N^{\delta \epsilon})\) amortised time per single-tuple update.

**Proof.** By Proposition \(38\) a major rebalancing step requires \(O(N^{1+(w-1)\epsilon})\) time. This time is amortised over \(\Omega(N)\) updates executed before the rebalancing step. Hence, the amortised time of major rebalancing is \(O(N^{(w-1)\epsilon})\). Since \(\delta = w\) or \(\delta = w - 1\), we conclude that the amortised time for major rebalancing is \(O(N^{\delta \epsilon})\). By Proposition \(39\) a minor rebalancing step requires \(O(N^{(\delta+1)\epsilon})\) time, which is amortised over \(\Omega(N)\) previous updates. This results in \(O(N^{\delta \epsilon})\) amortised minor rebalancing time. The formal proof for the amortised time upper bound is a straightforward extension of the amortisation proof in [20]. In [20], an update to a relation \(R\) can trigger a rebalancing step in which tuples are moved between the different parts of \(R\) only. Our partitioning strategy takes the degrees of values in all relations into account (see Section 2). Hence, an update to a relation can require to move tuples in parts of other relations. This, however, adds only a constant factor to the overall amortised time. \(\Box\)