Conjunctive Queries with Free Access Patterns under Updates

Ahmet Kara
University of Zurich, Switzerland

Milos Nikolic
University of Edinburgh, United Kingdom

Dan Olteanu
University of Zurich, Switzerland

Haozhe Zhang
University of Zurich, Switzerland

Abstract

We study the problem of answering conjunctive queries with free access patterns under updates. A free 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 of values over the input variables.

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

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

Keywords and phrases fully dynamic algorithm, enumeration delay, complexity trade-off, dichotomy

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

We consider the problem of answering conjunctive queries with free access patterns under single-tuple updates to the input database. Restricted access to data is commonplace [?, ?, ?]: 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. Access patterns are also present due to built-in predicates, e.g., $a + b = c$ or $\text{fun}(a, b, c)$, where a and b are input variables, c is an output variable, and fun is a function mapping a and b to c.

We formalise such queries as **c**onjunctive **q**ueries with free **a**ccess **p**atterns (CQAP for short): The free variables of a CQAP are partitioned into *input* and *output*. The query yields tuples of values over the output variables *given* a tuple of values over the input variables. CQAPs in databases correspond to conditional queries in probabilistic graphical models [?]: The latter ask for (the probability of) each possible value of a tuple of random variables (corresponding to CQAP output variables) given specific values for another tuple of random variables (corresponding to CQAP input variables). Prior work on queries with access patterns considers a more general setting than CQAP: There, each relation in the query body may have input and output variables such that values for the latter can only be obtained if values for the former are supplied [?, ?, ?, ?, ?]. In this more general setting, and in sharp

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:55
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

contrast to our simpler setting, a fundamental question is whether the query can even be answered for a given access pattern to each relation [**?**, **?**, **?**].

We introduce a fully dynamic evaluation approach for CQAPs. It is fully dynamic in the sense that it supports both inserts and deletes of tuples to the input relations. Our approach computes a data structure that supports the enumeration of the output tuples and maintains it under single-tuple updates to the input data. Our analysis of the overall computation time is refined into three components. The *preprocessing time* is the time to compute the data structure before receiving any updates. Given a tuple over the input variables, the *enumeration delay* is the time between the start of the enumeration process and the output of the first tuple, the time between outputting any two consecutive tuples, and the time between outputting the last tuple and the end of the enumeration process [**?**]. The *update time* is the time used to update the data structure for one single-tuple update. (We do not allow updates during the enumeration; this functionality is orthogonal to our contributions and can be supported using a versioned data structure.) The preprocessing step may be replaced by a sequence of inserts to the initially empty database. However, as shown in prior work on conjunctive queries under updates [**?**, **?**], bulk inserts, as performed in the preprocessing step, may take asymptotically less time than a sequence of single-tuple inserts.

There are simple, albeit more expensive alternatives to our approach. For instance, on an update request we may only update the input relations, and on an enumeration request we may use an existing enumeration algorithm for the residual query obtained by setting the input variables to constants in the original query. However, such an approach needs time-consuming preparation for each enumeration request, e.g., to remove dangling tuples and possibly create a data structure to support enumeration. In contrast, our approach maintains state between requests and can readily serve enumeration requests for any values of the input variables.

The contributions of this paper are as follows.

Section 3 introduces the CQAP language. Two new notions account for the nature of free access patterns: *access-top variable orders* and *query fractures.*

An access-top variable order is a decomposition of the query into a rooted forest of variables, where: the input variables are above all other variables; and the free (input and output) variables are above the bound variables. This variable order is compiled into a tree of views, which is a data structure that compactly represents the query output.

Since access to the query output requires fixing values for the input variables, the query can be fractured by breaking its joins on the input variables and replacing each of their occurrences with fresh variables within each connected component of the query hypergraph. This does not violate the access pattern, since 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.

Section 3 also introduces the *static* and *dynamic* widths that capture the complexities of the preprocessing and respectively update steps. For a given CQAP, these widths are defined over the access-top variable orders of the fracture of the query.

Section 4 introduces our approach for CQAP evaluation. Computing and maintaining each view in the view tree accounts for preprocessing and respectively updates, while the view tree as a whole allows for the enumeration of the output tuples with constant delay.

Section 5 gives a syntactic characterisation of those CQAPs that admit linear-time preprocessing and constant-time update and enumeration delay. We called this class CQAP_0. All queries outside CQAP_0 do not admit constant-time update and delay regardless of the preprocessing time, unless the widely held Online Matrix-Vector Multiplication conjecture [**?**]

fails. Our dichotomy generalises a prior dichotomy for q-hierarchical queries *without access patterns* [?]. The q-hierarchical queries are in CQAP_0, yet they have no input variables. The class CQAP_0 further contains cyclic queries with input variables. For instance, the edge triangle detection problem is in CQAP_0: Given an edge (u, v), check whether it participates in a triangle. The smallest query patterns not in CQAP_0 strictly include the non-q-hierarchical ones and also contain others that are sensitive to the interplay of the output and input variables. Proving that they do not admit constant-time update and delay requires different and additional hardness reductions from the Online Matrix-Vector Multiplication problem.

Section 6 charts the preprocessing time - update time - enumeration delay trade-off for the dynamic evaluation of the class of CQAPs whose fractures are hierarchical. It shows that as the preprocessing and update times increase, the enumeration delay decreases. Our trade-off reveals the optimality for a particular class of CQAPs with hierarchical fractures, called CQAP_1, which lies outside CQAP_0: The complexity of CQAP_1 for both the update and delay matches the lower bound $\Omega(N^{\frac{1}{2}})$ for queries outside CQAP_0, where N is the size of the input database. This is weakly Pareto optimal as we cannot lower both the update time and delay complexities (whether one of them can be lowered remains open). Our approach for CQAP_1 exhibits a continuum of trade-offs: $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{\epsilon})$ amortized update time and $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, for $\epsilon \in [0, 1]$. By tweaking the parameter ϵ, one can optimise the overall time for a sequence of enumeration and update tasks and achieve an asymptotically lower compute time than prior work. A well-studied query in CQAP_1 is the Dynamic Set Intersection problem [?]: We are given sets $S_1, ..., S_m$ subject to element insertions and deletions. For each access request (i, j) with $i, j \in [m]$, we need to decide whether the intersection of S_i and S_j is empty. Our approach recovers the complexity given by prior work [?] for this problem using $\epsilon = 0.5$.

2 Preliminaries

We introduce the data and computation models. Further preliminaries are in Appendix A.

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 $\mathsf{Dom}(X_i)$. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple $\mathbf{x}$ of values has schema $\mathcal{X} = \mathsf{Sch}(\mathbf{x})$ and is an element from $\mathsf{Dom}(\mathcal{X}) = \mathsf{Dom}(X_1) \times \cdots \times \mathsf{Dom}(X_n)$. A relation R over schema $\mathcal{X}$ is a function $R : \mathsf{Dom}(\mathcal{X}) \to \mathbb{Z}$ such that the multiplicity $R(\mathbf{x})$ is non-zero for finitely many tuples $\mathbf{x}$. A tuple $\mathbf{x}$ is in R, denoted by $\mathbf{x} \in R$, if $R(\mathbf{x}) \neq 0$. The size $|R|$ of R is the size of the set $\{\mathbf{x} \mid \mathbf{x} \in R\}$. A database is a set of relations and has size given by the sum of the sizes of its relations. Given a tuple $\mathbf{x}$ over schema $\mathcal{X}$ and $\mathcal{S} \subseteq \mathcal{X}$, $\mathbf{x}[\mathcal{S}]$ is the restriction of $\mathbf{x}$ onto $\mathcal{S}$. For a relation R over schema $\mathcal{X}$, schema $\mathcal{S} \subseteq \mathcal{X}$, and tuple $\mathbf{t} \in \mathsf{Dom}(\mathcal{S})$: $\sigma_{\mathcal{S}=\mathbf{t}} R = \{\, \mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{S}] = \mathbf{t} \,\}$ is the set of tuples in R that agree with $\mathbf{t}$ on the variables in $\mathcal{S}$; $\pi_{\mathcal{S}} R = \{\, \mathbf{x}[\mathcal{S}] \mid \mathbf{x} \in R \,\}$ stands for the set of tuples in R projected onto $\mathcal{S}$, i.e., the set of distinct $\mathcal{S}$-values from the tuples in R with non-zero multiplicities. For a relation R over schema $\mathcal{X}$ and $\mathcal{Y} \subseteq \mathcal{X}$, the *indicator projection* $I_{\mathcal{Y}} R$ is a relation over $\mathcal{Y}$ such that [?]:

$$\text{for all } \mathbf{y} \in \mathsf{Dom}(\mathcal{Y}) : I_{\mathcal{Y}} R(\mathbf{y}) = \begin{cases} 1 & \text{if there is } \mathbf{t} \in R \text{ such that } \mathbf{y} = \mathbf{t}[\mathcal{Y}] \\ 0 & \text{otherwise} \end{cases}$$

An update is a relation where tuples with positive multiplicities represent inserts and tuples with negative multiplicities represent deletes. Applying an update to a relation means unioning the update with the relation. A single-tuple update to a relation R is a singleton relation $\delta R = \{\mathbf{x} \to m\}$, where the multiplicity $m = \delta R(t)$ of the tuple t in δR is non-zero.

Computational Model. We consider the RAM model of computation. Each relation or materialised view R over schema $\mathcal{X}$ is implemented by a data structure that stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ with $R(\mathbf{x}) \neq 0$ 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 $\mathcal{S} \subset \mathcal{X}$, we use an index data structure that for any $\mathbf{t} \in \mathsf{Dom}(\mathcal{S})$ can: (4) enumerate all tuples in $\sigma_{\mathcal{S}=\mathbf{t}}R$ with constant delay, (5) check $\mathbf{t} \in \pi_{\mathcal{S}}R$ in constant time; (6) return $|\sigma_{\mathcal{S}=\mathbf{t}}R|$ in constant time; and (7) insert and delete index entries in constant time.

3 Conjunctive Queries with Free Access Patterns

We introduce the queries investigated in this paper along with several of their properties. A *conjunctive query with free 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; $\mathit{vars}(Q) = \bigcup_{i\in[n]} \mathcal{X}_i$ the set of variables; $\mathit{atoms}(X)$ the set of the atoms containing the variable X; $\mathit{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of all atoms; and $\mathit{free}(Q) = \mathcal{O} \cup \mathcal{I} \subseteq \mathit{vars}(Q)$ 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 $(\cdot)$.

Given a database $\mathcal{D}$ and a tuple $\mathbf{i}$ over $\mathcal{I}$, the output of Q for the input tuple $\mathbf{i}$ is denoted by $Q(\mathcal{O}|\mathbf{i})$ and is defined by $\pi_{\mathcal{O}}\sigma_{\mathcal{I}=\mathbf{i}}Q(\mathcal{D})$: This is the set of tuples $\mathbf{o}$ over $\mathcal{O}$ such that the assignment $\mathbf{i} \circ \mathbf{o}$ to the free variables satisfies the body of Q.

The hypergraph of a query Q is $\mathcal{H} = (\mathcal{V} = \mathit{vars}(Q), \mathcal{E} = \{\{\mathcal{X}_i \mid i \in [n]\}\})$, whose vertices are the variables and hyperedges are the schemas of the atoms in Q. The *fracture* of a CQAP Q is a CQAP $Q_\dagger$ constructed as follows. We start with $Q_\dagger$ as a copy of Q. We 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.

We next define the notion of dominance for variables in a CQAP Q. For variables A and B, we say that B *dominates* A if $\mathit{atoms}(A) \subset \mathit{atoms}(B)$. The query Q is *free-dominant* (*input-dominant*) if for any two variables A and B, it holds: if A is free (input) and B dominates A, then B is free (input). The query Q is *almost free-dominant* (*almost input-dominant*) if: (1) For any variable B that is not free (input) and for any atom $R(\mathcal{X}) \in \mathit{atoms}(B)$, there is another atom $S(\mathcal{Y}) \in \mathit{atoms}(B)$ such that $\mathcal{X} \cup \mathcal{Y}$ cover all free (input) variables dominated by B; (2) Q is not already free-dominant (input-dominant). A query Q is *hierarchical* if for any $A, B \in \mathit{vars}(Q)$, either $\mathit{atoms}(A) \subseteq \mathit{atoms}(B)$, $\mathit{atoms}(B) \subseteq \mathit{atoms}(A)$, or $\mathit{atoms}(B) \cap \mathit{atoms}(A) = \emptyset$. A query is q-hierarchical if it is hierarchical and free-dominant.

▶ **Definition 1.** *A query is in CQAP$_0$ if its fracture is hierarchical, free-dominant, and input-dominant. A query is in CQAP$_1$ if its fracture is hierarchical and is almost free-dominant, or almost input-dominant, or both.*

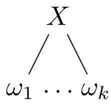
The subset of CQAP$_0$ without input variables is the class of q-hierarchical queries [**?**].

▶ **Example 2.** The query $Q_1(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D)$ is input-dominant, free-dominant, but not hierarchical. Its fracture $Q_\dagger(A, C \mid B_1, B_2, D_1, D_2) = R(A, B_1), S(B_2, C), T(C, D_1), U(A, D_2)$ is hierarchical but not input-dominant: C dominates indicators(CQAP Q, VO ω) : extended VO

switch ω:

$R(\mathcal{Y})$

1 **return** $R(\mathcal{Y})$



2 **let** $\hat{\omega}_i = \text{indicators}(\omega_i) \quad \forall i \in [k]$

3 **let** $\mathcal{S} = \{X\} \cup dep_\omega(X)$ and $\mathcal{R}$ be the set of atoms in ω

4 **let** $\mathcal{I} = \{\, I_{\mathcal{Z}} R(\mathcal{Z}) \mid R(\mathcal{Y}) \in (atoms(Q) \setminus \mathcal{R})$ and $\mathcal{Z} = \mathcal{Y} \cap \mathcal{S} \neq \emptyset \,\}$

5 **let** $\{I_1, ..., I_\ell\} = \text{GYO}(\mathcal{I} \cup \mathcal{R}) \setminus \mathcal{R}$

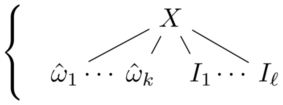
6 **return** 

Figure 1 Adding indicator projections to a VO ω of a CQAP Q. Each variable X in ω gets as new children the indicator projections of relations that do not occur in the subtree rooted at X but form a cycle with those that occur. The GYO reduction [?] eliminates from a set of relational schemas all schemas that do not take part in a cycle.

both B_2 and D_1 and A dominates both B_1 and D_2, yet A and C are not input. It is however almost input-dominant: A is not input 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. This means that Q_1 is in CQAP_1.

The query $Q_2(A \mid B) = S(A, B), T(B)$ is in CQAP_0, since its fracture $Q_\dagger(A \mid B_1, B_2) = S(A, B_1), T(B_2)$ is hierarchical, free-dominant, and input-dominant.

The query $Q_3(B \mid A) = S(A, B), T(B)$ is in CQAP_1. Its fracture is the query itself. It is hierarchical, yet not input-dominant, since B dominates A and is not input. It is, however, almost input-dominant: for each atom of B, there is one other atom such that together they cover A. Indeed, atom $S(A, B)$ already covers A, and it also does so together with $T(B)$; atom $T(B)$ does not cover A, but it does so together with $S(A, B)$.

The following are the smallest hierarchical queries that are not in CQAP_0 but in CQAP_1: $Q(A \mid \cdot) = R(A, B), S(B)$; $Q(B \mid A) = R(A, B), S(B)$; and $Q(\cdot \mid A) = R(A, B), S(B)$. ◀

3.1 Variable Orders

Variable orders are used as logical plans for the evaluation of conjunctive queries [?]. We next adapt them to CQAPs. Given a query, two variables *depend* on each other if they occur in the same query atom. A *variable order* (VO) ω for a CQAP Q is a pair (T_ω, dep_ω), where:

- T_ω is a (rooted) forest with one node per variable. The variables of each atom in Q lie along the same root-to-leaf path in T_ω.
- The function dep_ω maps each variable X to the subset of its ancestor variables in T_ω on which the variables in the subtree rooted at X depend.

An *extended* VO is a VO where we first add each atom as a child of its lowest variable and then atoms corresponding to the indicator projections of some relations, as explained next. The role of the indicators is to reduce the asymptotic complexity in case of cyclic queries [?].

Given a CQAP Q and a VO ω for Q, the function indicators in Figure 1 extends ω with indicator projections. It is assumed that the atoms of Q have been already added to ω. At each variable X in ω, we compute the set $\mathcal{I}$ of all possible indicator projections (Line 4).

Such indicators $I_{\mathcal{Z}}R$ are for relations R whose atoms are not included in the subtree rooted at X but share a non-empty set $\mathcal{Z}$ of variables with $\{X\} \cup dep_\omega(X)$. We choose from this set those indicators that form a cycle with the atoms in the subtree of ω rooted at X (Line 5), as determined by the GYO reduction procedure [?] that discards all atoms that do not take part in a cycle. The chosen indicator projections become children of X (Line 6). Appendix C illustrates the VO construction for a cyclic query.

We introduce notation for an extended VO ω. Its subtree rooted at X is denoted by ω_X. The sets $vars(\omega)$ and $\mathsf{anc}_\omega(X)$ consist of all variables of ω and respectively the variables on the path from X to the root excluding X. We denote by $atoms(\omega)$ all atoms and indicators at the leaves of ω and by Q_X the join of all atoms $atoms(\omega)$ (all variables are free).

In the rest of this paper, whenever we refer to a variable order, we always assume an extended VO. We next introduce classes of VOs for CQAP queries. A VO ω is *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. A VO ω 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 VOs for Q are denoted as $\mathsf{free\text{-}top}(Q)$ and $\mathsf{input\text{-}top}(Q)$, respectively. A VO is called *access-top* if it is free-top and input-top: $\mathsf{acc\text{-}top}(Q) = \mathsf{free\text{-}top}(Q) \cap \mathsf{input\text{-}top}(Q)$.

▶ **Example 3.** The query $Q(B|A) = R(A, B), S(B)$ admits the VO (in term notation; "-" represents the parent-child relationship): $B - \{A - R(A, B), S(B)\}$, where B has the variable A and the atom $S(B)$ as children and A has the atom $R(A, B)$ as child. The dependency sets are $dep(B) = \emptyset$ and $dep(A) = \{B\}$. This VO 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: $dep(A) = \emptyset$ and $dep(B) = \{A\}$.

The triangle query $Q(A, B|\cdot) = R(A, B), S(B, C), T(A, C)$ admits the VO $C - A - \{T(A, C), B - \{R(A, B), S(B, C), I_{AC}T(A, C)\}\}$, where one child of B is the indicator projection $I_{AC}T$ of T on $\{A, C\}$. The dependency sets are $dep(C) = \emptyset$, $dep(A) = \{C\}$, and $dep(B) = \{A, C\}$. The VO is input-top, since the query has no input variables; it is not free-top, since the bound variable C is on top of the free variables A and B.

The fracture of the 4-cycle query in Example 2 admits the access-top VO 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: $dep(A) = \{B_1, D_2\}$, $dep(D_2) = \{B_1\}$, $dep(B_1) = dep(B_2) = \emptyset$, $dep(C) = \{B_2, D_1\}$, and $dep(D_1) = \{B_2\}$. ◀

3.2 Width Measures

We next introduce two width measures for a VO ω and CQAP Q. They capture the complexity of computing and maintaining the output of Q.

▶ **Definition 4.** *The static width* $\mathsf{w}(\omega)$ *and dynamic width* $\delta(\omega)$ *of a VO* ω *are:*

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

$$\delta(\omega) = \max_{X \in vars(\omega)} \max_{R(\mathcal{Y}) \in atoms(\omega_X)} \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Y})$$

For a query Q_X and a set of variables $\mathcal{X} = \{X\} \cup dep_\omega(X)$, the fractional edge cover number [?] $\rho^*_{Q_X}(\mathcal{X})$ defines a worst-case upper bound on the time needed to compute $Q_X(\mathcal{X})$. Here, Q_X is the join of all atoms under X in the VO ω. The static width w of a VO ω is then defined as the maximum fractional edge cover number over all variables in ω. The dynamic width is defined similarly, with one simplification: We consider every case of a relation (or indicator projection) being replaced by a single-tuple update, so its variables $\mathcal{Y}$ are all set to constants and can be discarded in the computation of the fractional edge cover number.

We consider the standard lexicographic ordering $\leq$ on pairs of dynamic and static widths: $(\delta_1, \mathsf{w}_1) \leq (\delta_2, \mathsf{w}_2)$ if $\delta_1 \leq \delta_2$ or $\delta_1 = \delta_2$ and $\mathsf{w}_1 \leq \mathsf{w}_2$. Given a set $\mathcal{S}$ of VOs, we define $\min_{\omega \in \mathcal{S}}(\delta(\omega), \mathsf{w}(\omega)) = (\delta, \mathsf{w})$ such that $\forall \omega \in \mathcal{S} : (\delta, \mathsf{w}) \leq (\delta(\omega), \mathsf{w}(\omega))$.

▶ **Definition 5.** *The dynamic width $\delta(Q)$ and static width $\mathsf{w}(Q)$ of a CQAP Q are:*

$$(\delta(Q), \mathsf{w}(Q)) = \min_{\omega \in \mathsf{acc\text{-}top}(Q_\dagger)} (\delta(\omega), \mathsf{w}(\omega))$$

Since we are interested in dynamic evaluation, Definition 5 first minimises for the dynamic width and then for the static width. To determine the dynamic and the static width of a CQAP Q, we first search for the VOs of the fracture $Q_\dagger$ with minimal dynamic width and choose among them one with the smallest static width. Appendix B further expands on the width measures with examples and properties.

▶ **Example 6.** Consider the query $Q(\mathcal{O} \mid \mathcal{I}) = R(A, B, C), S(A, B, D), T(A, E)$. The static width w and the dynamic width δ of Q vary depending on the access pattern:
For $Q(\{C, D, E\} \mid \{A, B\})$, $\mathsf{w} = 1$ and $\delta = 0$. For $Q(\{A, C, D, E\} \mid \{B\})$, $\mathsf{w} = 1$ and $\delta = 1$.
For $Q(\{A, C, D\} \mid \{B, E\})$, $\mathsf{w} = 2$ and $\delta = 1$. For $Q(\{A, E\} \mid \{B, C, D\})$, $\mathsf{w} = 2$ and $\delta = 2$.
For $Q(\{A, B\} \mid \{C, D, E\})$, $\mathsf{w} = 3$ and $\delta = 2$. For $Q(\{A, B, C, D, E\}|\cdot)$, $Q(\cdot|\{A, B, C, D, E\})$ and $Q(\{B, C, D, E\}|\{A\})$, $\mathsf{w} = 1$ and $\delta = 0$. ◀

4 CQAP Evaluation

In this section, we introduce a fully dynamic evaluation approach for arbitrary CQAPs whose complexity is stated in the following theorem.

▶ **Theorem 7.** *Given a CQAP with static width w and dynamic width δ and a database of size N, the query can be evaluated with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(N^{\delta})$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

Our approach has three stages: preprocessing, enumeration, and updates. They are detailed in the following subsections. Our running examples consider queries with acyclic fractures. Examples with cyclic fractures are given in Appendix C. We consider in the following a fixed CQAP $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_\dagger(\mathcal{O}|\mathcal{I}_\dagger)$, and a database of size N.

4.1 Preprocessing

In the preprocessing stage, we construct a set of view trees that represent the result of $Q_\dagger$ over both its input and output variables. A view tree [?] is a (rooted) tree with one view per node. It is a logical project-join plan in the classical database systems literature, but where each intermediate result is materialised. The view at a node is defined as the join of the views at its children, possibly followed by a projection. The view trees are modelled following an access-top VO ω of $Q_\dagger$. In the following, we discuss the case of ω consisting of a single tree; otherwise, we apply the preprocessing stage to each tree in ω.

Given an access-top VO ω, the function $\tau(\omega)$ in Figure 2 returns a view tree constructed from ω. The function traverses ω bottom-up 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

τ(VO ω) : view tree		
switch ω:		
$R(\mathcal{Y})$	1	**return** $R(\mathcal{Y})$
X with children $\omega_1 \cdots \omega_k$	2	**let** $T_i = \tau(\omega_i) \quad \forall i \in [k]$
	3	**let** $\mathcal{S} = \{X\} \cup dep_\omega(X)$ and $V_X(\mathcal{S})$ = join of roots of $T_1, ..., T_k$
	4	**if** X has no sibling **return** $V_X(\mathcal{S})$ with children $T_1 \cdots T_k$
	5	**let** $V'_X(\mathcal{S} \setminus \{X\}) = V_X(\mathcal{S})$ **return** $V'_X(\mathcal{S} \setminus \{X\})$ — $V_X(\mathcal{S})$ with children $T_1 \cdots T_k$

Figure 2 Construction of a view tree following a VO ω. At each variable X in ω, 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 marginalises out X.

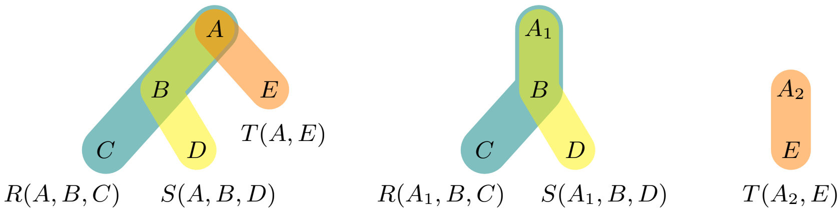


Figure 3 (Left) Hypergraph of the two queries with the same body but different access patterns, as used in Examples 8 and 9; (middle and right) hypergraph of their fractures.

dependency set of X (Line 3). This view allows to efficiently enumerate the X-values given a tuple of values for the variables in the dependency set. If X has siblings, the function creates an additional view V'_X on top of V_X whose purpose is to aggregate away (or marginalise out) X from V_X (Line 5). This view allows to efficiently maintain the ancestor views of V_X under updates to the views created for the siblings of X.

The time to construct the view tree $\tau(\omega)$ is dominated by the time to materialise the view V_X for each variable X. The auxiliary view V'_X above V_X can be materialised by marginalising out X in one scan over V_X. Each view V_X can be materialised in $\mathcal{O}(N^{\mathsf{w}})$ time, where $\mathsf{w} = \rho^*_{Q_X}(\{X \cup dep_\omega(X)\})$. The definition of the static width of ω implies that the view tree $\tau(\omega)$ can be constructed in $\mathcal{O}(N^{\mathsf{w}(\omega)})$ time. By choosing a VO whose static width is $\mathsf{w}(Q)$, the preprocessing time of our approach becomes $\mathcal{O}(N^{\mathsf{w}(Q)})$, as stated in Theorem 7.

The next example demonstrates our view tree construction for a query in CQAP_0.

▶ **Example 8.** Figure 3 shows the hypergraphs of the query $Q(B, C, D, E|A) = R(A, B, C)$, $S(A, B, D)$, $T(A, E)$ and its fracture $Q_\dagger(B, C, D, E|A_1, A_2) = R(A_1, B, C)$, $S(A_1, B, D)$, $T(A_2, E)$. The fracture has two connected components: $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E|A_2) = T(A_2, E)$. Figure 4 depicts an access-top VO (left) for Q_1 and its corresponding view tree (middle). The VO has static width 1. Each variable in the VO is mapped to a view in the view tree, e.g., B is mapped to $V_B(A_1, B)$, where $\{B, A_1\} = \{B\} \cup dep(B)$.

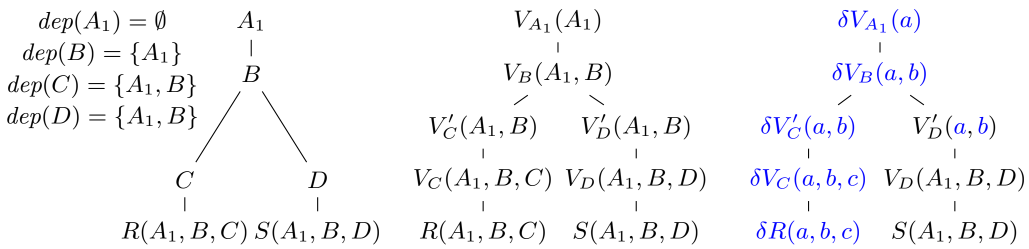


Figure 4 (Left) Access-top VO for $Q_1(B,C,D|A_1) = R(A_1,B,C), S(A_1, B,D)$; (middle) the view tree constructed from the VO; (right) the delta view tree under a single-tuple update to R.

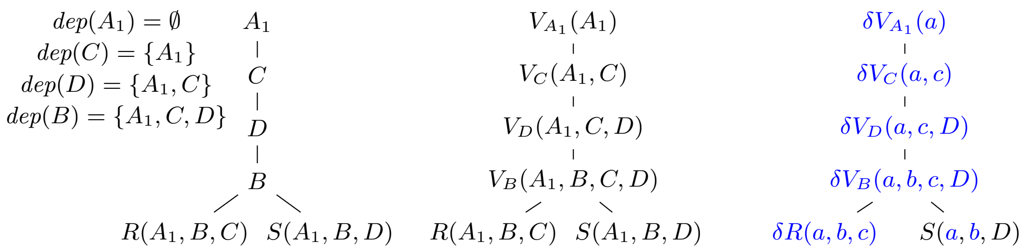


Figure 5 (Left) Access-top VO for $Q_1(B,D|A_1,C) = R(A_1,B,C), S(A_1, B,D)$; (middle) the view tree corresponding to the VO; (right) the delta view tree under a single-tuple update to R.

The views V'_C and V'_D are auxiliary views. The views V'_C, V'_D, and V_{A_1} marginalise out the variables C, D and respectively B from their child views. The view V_B is the intersection of V'_C and V'_D. Hence, all views can be computed in $\mathcal{O}(N)$ time. Since the query fracture is acyclic, the view tree does not contain indicator projections.

The only access-top VO for the connected component Q_2 of $Q_\dagger$ is the top-down path $A_2 - E - T(A_2,E)$. The views mapped to A_2 and E are $V_{A_2}(A_2)$ and respectively $V_E(A_2,E)$. They can obviously be computed in $\mathcal{O}(N)$ time. ◀

The next example considers a CQAP_1 whose preprocessing time is quadratic.

▶ **Example 9.** Consider the CQAP_1 $Q(E,D|A,C) = R(A,B,C), S(A,B,D), T(A,E)$ and its fracture $Q_\dagger(E,D|A_1,A_2,C) = R(A_1,B,C), S(A_1,B,D), T(A_2,E)$. The fracture has the two connected components $Q_1(B,D|A_1,C) = R(A_1,B,C), S(A_1,B,D)$ and $Q_2(E|A_2) = T(A_2,E)$. The hypergraphs (Figure 3) of Q and its fracture are the same as for the query in Example 8. Figure 5 depicts an access-top VO (left) for Q_1 and its corresponding view tree (middle). The VO has static width 2. The view V_B joins the relations R and S, which takes $\mathcal{O}(N^2)$ time. The views V_D, V_C, and V_A are constructed from V_B by marginalising out one variable at a time. Hence, the view tree construction takes $\mathcal{O}(N^2)$ time. The view tree for Q_2 is the same as in Example 8 and can be constructed in linear time. ◀

4.2 Enumeration

The view trees constructed by the function τ for any access-top VO for $Q_\dagger$ allow for constant-delay enumeration of the tuples in $Q(\mathcal{O}|\mathbf{i})$ given any tuple $\mathbf{i}$ over the input variables $\mathcal{I}$.

Assume that ω_i is a tree in the forest ω for which $\tau(\omega_i)$ constructs the view tree T_i, for $i \in [n]$. Let $Q_i(\mathcal{O}_i|\mathcal{I}_i)$ with $\mathcal{O}_i = \mathcal{O} \cap \mathit{vars}(\omega_i)$ and $\mathcal{I}_i = \mathcal{I}_\dagger \cap \mathit{vars}(\omega_i)$ be the CQAP that joins the atoms at the leaves of T_i. We first explain how to enumerate the tuples in $Q_i(\mathcal{O}_i \mid \mathbf{i})$

from T_i with constant delay, given an input tuple $\mathbf{i}$ over $\mathcal{I}_i$. We traverse the view tree T_i in preorder and execute at each view V_X the following steps. In case $X \in \mathcal{I}_i$, we check whether the projection of $\mathbf{i}$ onto the schema of V_X is in V_X. If not, the query output is empty and we stop. Otherwise, we continue with the preorder traversal. In case $X \in \mathcal{O}_i$, we retrieve in constant time the first X-value in V_X given that the values over the variables in the root path of X are already fixed to constants. After all views are visited once, we have constructed the first complete output tuple and report it. Then, we iterate with constant delay over the remaining distinct X-values in the last visited view V_X. For each distinct X-value, we obtain a new tuple and report it. After all X-values in V_X are exhausted, we backtrack.

Assume now that we have a procedure that enumerates the tuples in $Q_i(\mathcal{O}_i \mid \mathbf{i}_i)$ for any tuple $\mathbf{i}_i$ over $\mathcal{I}_i$ with constant delay. Consider a tuple $\mathbf{i}$ over the input variables $\mathcal{I}$ of Q. It holds $Q(\mathcal{O}|\mathbf{i}) = \times_{i\in[n]} Q_i(\mathcal{O}_i|\mathbf{i}_i)$ where $\mathbf{i}_i[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q. We can enumerate the tuples in $Q(\mathcal{O} \mid \mathbf{i})$ with constant delay by nesting the enumeration procedures for $Q_1(\mathcal{O}_1 \mid \mathbf{i}_1), \ldots, Q_n(\mathcal{O}_n \mid \mathbf{i}_n)$.

► **Example 10.** Consider the query $Q(B, C, D, E|A)$ from Example 8 and the two connected components $Q_1(B, C, D|A_1)$ and $Q_2(E|A_2)$ of its fracture. Figure 4 (middle) depicts the view tree for Q_1. Given an A_1-value a, we can use this view tree to enumerate the distinct tuples in $Q_1(B, C, D|a)$ with constant delay. We first check if a is included in the view V_{A_1}. If not, $Q_1(B, C, D|a)$ must be empty and we stop. Otherwise, we retrieve the first B-value b paired with a in V_B, the first C-value c paired with (a, b) in V_C, and the first D-value d paired with (a, b) in V_D. Thus, we obtain in constant time the first output tuple (b, c, d) in $Q_1(B, C, D|a)$ and report it. Then, we iterate over the remaining distinct D-values paired with (a, b) in V_D and report for each such D-value d', a new tuple (b, c, d'). After all D-values are exhausted, we retrieve the next distinct C-value paired with (a, b) in V_C and restart the iteration over the distinct D-values paired with (a, b) in V_D, and so on. Overall, we construct each distinct tuple in $Q_1(B, C, D|a)$ in constant time after the previous one is constructed.

Assume now that we have constant-delay enumeration procedures for the tuples in $Q_1(B, C, D|a)$ and the tuples in $Q_2(E|a)$ for any A-value a. We can enumerate with constant delay the tuples in $Q(B, C, D, E|a)$ as follows. We ask for the first tuple (b, c, d) in $Q_1(B, C, D|a)$ and then iterate over the distinct E-values in $Q_2(E|a)$. For each such E-value e, we report the tuple (b, c, d, e). Then, we ask for the next tuple in $Q_1(B, C, D|a)$ and restart the enumeration over the tuples in $Q_2(E|a)$, and so on. ◄

4.3 Updates

We now explain how to update the view trees constructed by the function τ in Figure 2. Consider a single-tuple update $\delta R = \{\mathbf{x} \to m\}$ to an input relation R; m is positive in case of insertion and negative in case of deletion. We first update each view tree that has an atom $R(\mathcal{X})$ at a leaf: We update each view on the path from that leaf to the root of the view tree using the classical delta rules [?]. The update δR may affect indicator projections $I_{\mathcal{Z}} R$. A new single-tuple update $\delta I_{\mathcal{Z}} R = \{\mathbf{x}[\mathcal{Z}] \to k\}$ to $I_{\mathcal{Z}} R$ is triggered in the following two cases. If δR is an insertion and $\mathbf{x}[\mathcal{Z}]$ is a value not already in $\pi_{\mathcal{Z}} R$, then the new update is triggered with $k = 1$. If δR is a deletion and $\pi_{\mathcal{Z}} R$ does not contain $\mathbf{x}[\mathcal{Z}]$ after applying the update to R, then the new update is triggered with $k = -1$. This update is propagated up to the root of each view tree, like for δR.

Recall that the time to compute a view V_X is $\mathcal{O}(N^{\mathsf{w}})$, where $\mathsf{w} = \rho^*_{Q_X}(\{X\} \cup dep_\omega(X))$. In case of an update to a relation or indicator R over schema $\mathcal{Y}$, the variables in $\mathcal{Y}$ are set to constants. The time to update V_X is then $\mathcal{O}(N^{\delta})$, where $\delta = \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Y})$.

Assuming that the dynamic width of ω is $\delta(Q)$, we conclude that the update time of our approach is $\mathcal{O}(N^{\delta(Q)})$, as stated in Theorem 7.

▶ **Example 11.** Figure 4 (right) shows the delta view tree for the view tree to the left under a single-tuple update $\delta R(a, b, c)$ to R. We update the relation $R(A, B, C)$ with $\delta R(a, b, c)$ in constant time. The ancestor views of δR (in blue) are the deltas of the corresponding views, computed by propagating δR from the leaf to the root. They can also be effected in constant time. Overall, maintaining the view tree under a single-tuple update to any relation takes $O(1)$ time.

Consider now the delta view tree in Figure 5 (right) obtained from the view tree to its left under the single-tuple update $\delta R(a, b, c)$. We update $V_B(A_1, B, C, D)$ with $\delta V_B(a, b, c, D) = \delta R(a, b, c), S(a, b, D)$ in $\mathcal{O}(N)$ time, since there are at most N D-values paired with (a, b) in S. We then update the views V_D, V_C, and V_{A_1} in $\mathcal{O}(1)$ time. Updates to S are handled analogously. Overall, maintaining the view tree under a single-tuple update to any input relation takes $O(N)$ time. ◀

5 A Dichotomy for CQAPs

The following dichotomy states that the queries in CQAP_0 are precisely those CQAPs that can be evaluated with constant update time and enumeration delay.

▶ **Theorem 12.** *Let any CQAP query Q and database of size N.*

- *If Q is in $CQAP_0$, then it admits $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ enumeration delay, and $\mathcal{O}(1)$ update time for single-tuple updates.*
- *If Q is not in $CQAP_0$ 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 $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time, for any $\gamma > 0$, unless the OMv conjecture fails.*

The hardness result in Theorem 12 is based on the following OMv problem:

▶ **Definition 13** (Online Matrix-Vector Multiplication (OMv) [?]). *We are given an $n \times n$ Boolean matrix $\mathbf{M}$ and receive n Boolean column vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ of size n, one by one; after seeing each vector $\mathbf{v}_i$, we output the product $\mathbf{M}\mathbf{v}_i$ before we see the next vector.*

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

▶ **Conjecture 14** (OMv Conjecture, Theorem 2.4 [?]). *For any $\gamma > 0$, there is no algorithm that solves the OMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

Queries in CQAP_0 have dynamic width 0 and static width 1 (Proposition 25, Appendix D). Our approach from Section 4 achieves linear preprocessing time, constant update time and enumeration delay for such queries (Theorem 7), so it is optimal for CQAP_0.

The smallest queries not included in CQAP_0 are: $Q_1(\mathcal{O}|\cdot) = R(A), S(A, B), T(B)$ with $\mathcal{O} \subseteq \{A, B\}$; $Q_2(A|\cdot) = R(A, B), S(B)$; $Q_3(\cdot|A) = R(A, B), S(B)$; and $Q_4(B|A) = R(A, B), S(B)$. Each query is equal to its fracture. Query Q_1 is not hierarchical; Q_2 is not free-dominant; and Q_3 and Q_4 are not input-dominant. Prior work showed that there is no algorithm that achieves constant update time and enumeration delay for Q_1 and Q_2, unless the OMv conjecture fails [?]. To prove the hardness statement in Theorem 12, we show that this negative result also holds for Q_3 and Q_4. Then, given an arbitrary CQAP Q that is not in CQAP_0, we reduce the evaluation of one of the four queries above to the evaluation of Q.

6 Trade-Offs for CQAPs with Hierarchical Fractures

For CQAPs with hierarchical fractures, the complexities in Theorem 7 can be parameterised to uncover trade-offs between preprocessing, update, and enumeration.

▶ **Theorem 15.** *Let any CQAP Q with static width w and dynamic width δ, a database of size N, and $\epsilon \in [0,1]$. If Q's fracture is hierarchical, then Q admits $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

This continuum of trade-offs can be obtained using one algorithm parameterised by ϵ. This algorithm either recovers or has lower complexity than prior approaches. Using $\epsilon = 1$, we recover the complexities in Theorem 7 and therefore also the constant update time and delay for queries in CQAP_0 in Theorem 12.

Theorem 15 can be refined for CQAP_1, since $\delta = 1$ and $\mathsf{w} \leq 2$ for queries in this class.

▶ **Corollary 16.** *(Theorem 15). Let any query in $CQAP_1$, a database of size N, and $\epsilon \in [0,1]$. Then Q admits $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\epsilon})$ amortised update time for single-tuple updates.*

For $\epsilon = 0.5$, the update time and delay for queries in CQAP_1 match the lower bound in Theorem 12 for all queries outside CQAP_0. This makes our approach weakly Pareto optimal for CQAP_1, as lowering both the update time and delay would violate the OMv conjecture.

Our algorithm has two core ideas. (For lack of space, we defer the details to Appendix E.) First, we partition the input relations into heavy and light parts based on the degrees of the values. This transforms a query over the input relations into a union of queries over heavy and light relation parts. Second, we employ different evaluation strategies for different heavy-light combinations of parts of the input relations. This allows us to confine the worst-case behaviour caused by high-degree values in the database during query evaluation.

We construct a set of VOs for the hierarchical fracture of a given CQAP. Each VO represents a different evaluation strategy over heavy and light relation parts. For VOs over light relation parts, we follow the general approach from Section 4 and construct view trees from access-top VOs. For VOs involving heavy relation parts, we construct view trees from VOs that are not access-top, thus yielding non-constant enumeration delay but better preprocessing and update times. This trade-off is controlled by the parameter ϵ.

Enumerating distinct tuples from the constructed view trees poses two challenges. First, these view trees may encode overlapping subsets of the query result. To enumerate only distinct tuples from these view trees, we use the union algorithm [?] and view tree iterators, as in prior work [?]. Second, for views trees built from VOs that are not access-top, the enumeration approach from Section 4 would report the values of bound variables before the values of free variables or the values of output variables before setting the values of input variables. To resolve this issue, we instantiate a view tree iterator for each value of the variable that violates the free-dominance or input-dominance condition. We then use the union algorithm to report only distinct tuples over the output variables. By partitioning input relations, we ensure that the number of instantiated iterators depends on ϵ. For view trees built from access-top VOs, we use the enumeration approach from Section 4.

6.1 Data Partitioning

We partition relations based on the frequencies of their values. For a database $\mathcal{D}$, relation $R \in \mathcal{D}$ over schema $\mathcal{X}$, schema $\mathcal{S} \subset \mathcal{X}$, and threshold θ, the pair $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ is a *partition* of R on $\mathcal{S}$ with threshold θ if it satisfies the conditions:

$$\begin{aligned}
&\text{(union)} && R(\mathbf{x}) = R^{\mathcal{S}\to H}(\mathbf{x}) + R^{\mathcal{S}\to L}(\mathbf{x}) \text{ for } \mathbf{x} \in \mathsf{Dom}(\mathcal{X})\\
&\text{(domain partition)} && \pi_{\mathcal{S}} R^{\mathcal{S}\to H} \cap \pi_{\mathcal{S}} R^{\mathcal{S}\to L} = \emptyset\\
&\text{(heavy part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to H}, \exists K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| \geq \tfrac{1}{2}\theta\\
&\text{(light part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L} \text{ and } \forall K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| < \tfrac{3}{2}\theta
\end{aligned}$$

We call $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ a *strict* partition of R on $\mathcal{S}$ with threshold θ if it satisfies the union and domain partition conditions and the strict versions of the heavy and light part conditions:

$$\begin{aligned}
&\text{(strict heavy part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to H}, \exists K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| \geq \theta\\
&\text{(strict light part)} && \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L} \text{ and } \forall K \in \mathcal{D}\text{: } |\sigma_{\mathcal{S}=\mathbf{t}} K| < \theta
\end{aligned}$$

The relation $R^{\mathcal{S}\to H}$ is called *heavy* and the relation $R^{\mathcal{S}\to L}$ is called *light* on the partition key $\mathcal{S}$. Due to the domain partition, the relations $R^{\mathcal{S}\to H}$ and $R^{\mathcal{S}\to L}$ are disjoint. For $|\mathcal{D}| = N$ and a strict partition $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ of R on $\mathcal{S}$ with threshold $\theta = N^{\epsilon}$ for $\epsilon \in [0, 1]$, we have: (1) $\forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S}\to L} : |\sigma_{\mathcal{S}=\mathbf{t}} R^{\mathcal{S}\to L}| < \theta = N^{\epsilon}$; and (2) $|\pi_{\mathcal{S}} R^{\mathcal{S}\to H}| \leq \frac{N}{\theta} = N^{1-\epsilon}$. The first bound follows from the strict light part condition. In the second bound, $\pi_{\mathcal{S}} R^{\mathcal{S}\to H}$ refers to the tuples over schema $\mathcal{S}$ with high degrees in some relation in the database. The database can contain at most $\frac{N}{\theta}$ such tuples; otherwise, the database size would exceed N.

Disjoint relation parts can be further partitioned independently of each other on different partition keys. We write $R^{\mathcal{S}_1\to s_1,\dots,\mathcal{S}_n\to s_n}$ to denote the relation part obtained after partitioning $R^{\mathcal{S}_1\to s_1,\dots,\mathcal{S}_{n-1}\to s_{n-1}}$ on $\mathcal{S}_n$, where $s_i \in \{H, L\}$ for $i \in [n]$. The domain of $R^{\mathcal{S}_1\to s_1,\dots,\mathcal{S}_n\to s_n}$ is the intersection of the domains of $R^{\mathcal{S}_i\to s_i}$, for $i \in [n]$. We refer to $\mathcal{S}_1 \to s_1, \dots, \mathcal{S}_n \to s_n$ as a heavy-light signature for R. Consider for instance a relation R with schema (A, B, C). One possible partition of R consists of the relation parts $R^{A\to L}$, $R^{A\to H, AB\to L}$, and $R^{A\to H, AB\to H}$. The union of these relation parts constitutes the relation R.

6.2 Preprocessing

The preprocessing has two steps. First, we construct a set of VOs corresponding to the different evaluation strategies over the heavy and light relation parts. Second, we build a view tree from each such VO using the function τ from the general case (Figure 2).

We next describe the construction of a set of VOs from a canonical VO ω of a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$. Without loss of generality, we assume that ω is a tree; in case ω is a forest, the reasoning below applies independently to each tree in the forest. The construction proceeds recursively on the structure of ω and forms the query $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ at each variable X. The query Q_X is the join of the atoms in ω_X, the set $\mathcal{O}_X$ consists of the output variables in ω_X, and the set $\mathcal{I}_X$ consists of the input variables in ω_X and all ancestor variables along the path from X to the root of ω. The next step analyses the query Q_X.

If Q_X is in CQAP_0, we turn ω_X into an access-top VO for Q_X by pulling the free variables above the bound variables and the input variables above the output variables. For queries in CQAP_0, this restructuring does not increase their static width.

If Q_X is not in CQAP_0, then ω_X contains a bound variable that dominates a free variable or an output variable that dominates an input variable. If X does not violate either of these conditions, we recur on each subtree and combine the constructed VOs. Otherwise, we create two sets of VOs, which encode different evaluation strategies for different parts of the result of Q_X. Let *key* be the set of variables on the path from X to the root of the canonical VO for Q, including X. For the first set of VOs, each leaf atom $R^{sig}(\mathcal{X})$ below X is replaced by $R^{sig,key\to H}(\mathcal{X})$ before recurring on each subtree, denoting that the evaluation of Q_X is

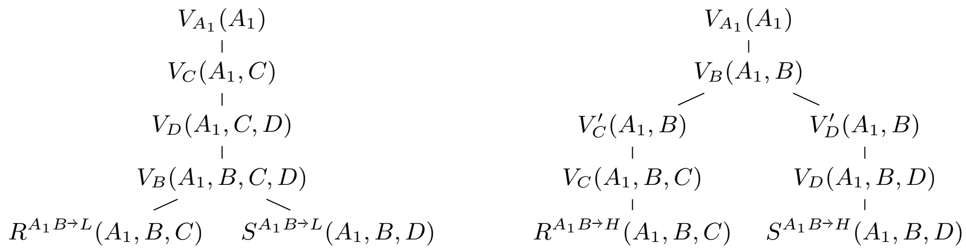


Figure 6 View trees constructed for $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$ from Example 17 using the VOs: (left) $A_1 - C - D - B - \{R^{A_1 B \to L}(A_1, B, C), S^{A_1 B \to L}(A_1, B, D)\}$ and (right) $A_1 - B - \{C - R^{A_1 B \to H}(A_1, B, C), D - S^{A_1 B \to H}(A_1, B, D)\}$.

over relations parts that are heavy on *key*. For the second set of VOs, we turn ω_X into an access-top VO over relations parts that are light on *key*; this restructuring of the VO may increase its static width.

We construct a view tree for each VO formed in the previous step. For each view tree, we strict partition the input relations based on their heavy-light signature and compute the queries defining the views. We refer to this step as view tree materialisation. The view trees constructed for the evaluation of queries in CQAP_0 or over heavy relation parts follow canonical VOs, meaning that they can be materialised in linear time. The view trees constructed for the evaluation of queries over light relation parts follow access-top VOs. Using the degree constraints in the input relations, each such view tree can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$, where w is the static width of the query.

▶ **Example 17.** We explain the construction of the views tree for the connected component from Figure 3 (middle) corresponding to the query $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$. In the canonical VO of this query, shown in Figure 4 (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^ϵ, where N is the database size. To evaluate the join over the light relation parts, we turn the subtree in the canonical VO rooted at B into an access-top VO and construct a view tree following this new VO, see Figure 6 (left). We compute the view $V_B(A_1, B, C, D)$ in time $\mathcal{O}(N^{1+\epsilon})$: For each (a, b, c) in the light part $R^{A_1 B \to L}(A_1, B, C)$ of R, we fetch the D-values in $S^{A_1 B \to L}(A_1, B, D)$ that are paired with (a, b). The iteration in $R^{A_1 B \to L}(A_1, B, C)$ takes $\mathcal{O}(N)$ time and for each (a, b), there are at most N^ϵ D-values in $S^{A_1 B \to L}(A_1, B, D)$. The views V_D, V_C, and V_A result from V_B by marginalising out one variable at a time. Overall, this takes $\mathcal{O}(N^{1+\epsilon})$ time.

To evaluate the join over the heavy parts of R and S, we construct a view tree following the canonical VO (Figure 6 right). The VO and view tree are the same as in Figure 3, except that the leaves are the heavy parts of R and S. The view tree can be materialised in $\mathcal{O}(N)$ time, cf. Example 8. Overall, the two view trees can be computed in $\mathcal{O}(N^{1+\epsilon})$ time. ◀

6.3 Updates

A single-tuple update to an input relation may cause changes in several view trees constructed for a given hierarchical CQAP. If the input relation is partitioned, we first identify which part of the relation is affected by the update. We then propagate the update in each view tree containing the affected relation part, as discussed in Section 4.

▶ **Example 18.** We consider the maintenance of the view trees from Figure 6 under a single-tuple update $\delta R(a, b, c)$ to R. The update affects the heavy part $R^{A_1 B \to H}$ if $(a, b) \in \pi_{A_1, B} R^{A_1 B \to H}$; otherwise, it affects the light part $R^{A_1 B \to L}$. For the former, we propagate the update from $R^{A_1 B \to H}$ to the root. For each view on this path, we compute its delta query and update the view in constant time for fixed (a, b, c). For the latter, we compute the delta $\delta V_B(a, b, c, D) = \delta R^{A_1 B \to L}(a, b, c), S^{A_1 B \to L}(a, b, D)$ in $\mathcal{O}(N^{\epsilon})$ time because there are at most N^{ϵ} D-values paired with (a, b) in $S^{A_1 B \to L}$. We then update $V_D(a, c, D)$ with $\delta V_D(a, c, D) = \delta V_B(a, b, c, D)$ in $\mathcal{O}(N^{\epsilon})$ time and update the views $V_C(A_1, C)$ and $V_{A_1}(A_1)$ in constant time. The case of single-tuple updates to S is analogous. Overall, maintaining the two view trees under a single-tuple update to any input relation takes $\mathcal{O}(N^{\epsilon})$ 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 $\mathcal{O}(N^{\delta\epsilon})$, they happen periodically and their amortised cost remains the same as for a single-tuple update.

7 Related Work

Our work is the first to investigate the dynamic evaluation for queries with access patterns.

Free Access Patterns. Prior work closest in spirit to ours investigated the space-delay trade-off for the static evaluation of full conjunctive queries with free access patterns [**?**]. This work constructs a succinct representation of the query output, from which the tuples that conform with value bindings of the input variables can be enumerated. It does not support queries with projection nor dynamic evaluation. Follow-up work considers the static evaluation for Boolean conjunctive queries with access patterns [**?**].

Dynamic evaluation. Our work generalises the dichotomy for q-hierarchical queries under updates [**?**, **?**] and the complexity trade-offs for queries under updates [**?**, **?**, **?**]. We refer the reader to a comprehensive comparison [**?**] of dynamic query evaluation techniques and how they are recovered by the trade-off [**?**] extended in our work.

Our CQAP$_0$ dichotomy strictly generalises the one for q-hierarchical queries [**?**]: The set of q-hierarchical queries is a strict subset of CQAP$_0$, while there are hard patterns of non-CQAP$_0$ beyond those for non-q-hierarchical queries.

There are key technical differences between the prior framework for dynamic evaluation trade-off [**?**] and ours: different data partitioning; new modular construction of view trees; access-top variable orders; new iterators for view trees modelled on any variable order. We 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.

Dissociation. Query fractures are central to our access pattern approach. Under certain conditions, they replace the input variables with fresh input variables. *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 [**?**]. *Query dissociation* serves the same purpose [**?**]. 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

This paper introduces a fully dynamic evaluation approach for conjunctive queries with free access patterns. It gives a syntactic characterisation of those queries that admit constant-time update and delay and further investigates the trade-off between preprocessing time, update time, and enumeration delay for such queries.

References

A Missing Details in Section 2

A.1 Example Data Structure Conforming to the Computational Model

We give an example data structure that conforms to the computational model from Section 2. Consider a relation (materialized view) R over schema $\mathcal{X}$. A hash table with chaining stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ over $\mathcal{X}$ with $R(\mathbf{x}) \neq 0$. 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 $\mathcal{F} \subset \mathcal{X}$, we create another hash table with chaining where each table entry stores a tuple $\mathbf{t}$ of $\mathcal{F}$-values as key and a doubly-linked list of pointers to the entries in R having the $\mathcal{F}$-values $\mathbf{t}$ as value. Looking up an index entry given $\mathbf{t}$ 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 $\mathbf{t}$; 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 doubly-linked lists takes constant time per index.

B Missing Details in Section 3

B.1 Width measures

Given a conjunctive query Q and $\mathcal{F} \subseteq \mathit{vars}(Q)$, a *fractional edge cover* of $\mathcal{F}$ is a solution $\boldsymbol{\lambda} = (\lambda_{R(\mathcal{X})})_{R(\mathcal{X}) \in \mathit{atoms}(Q)}$ to the following linear program [**?**]:

$$
\begin{aligned}
&\text{minimize} && \sum_{R(\mathcal{X}) \in\, \mathit{atoms}(Q)} \lambda_{R(\mathcal{X})} && \\
&\text{subject to} && \sum_{R(\mathcal{X}):\, X \in \mathcal{X}} \lambda_{R(\mathcal{X})} \geq 1 && \text{for all } X \in \mathcal{F} \text{ and} \\
& && \lambda_{R(\mathcal{X})} \in [0,1] && \text{for all } R(\mathcal{X}) \in \mathit{atoms}(Q)
\end{aligned}
$$

The optimal objective value of the above program is called the *fractional edge cover number* of $\mathcal{F}$ in Q 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 $\lambda_{R(\mathcal{X})} \in \{0,1\}$ for each $R(\mathcal{X}) \in \mathit{atoms}(Q)$. The optimal objective value of this program is called the *integral edge cover number* of $\mathcal{F}$, denoted as $\rho_Q(\mathcal{F})$. If Q is clear from the context, we omit the subscript Q in $\rho^*_Q(\mathcal{F})$ and $\rho_Q(\mathcal{F})$.

▶ **Example 19.** We show how to compute the widths for the variable order of the fractured 4-cycle query in Example 3: For the bag at variable A, we have $\rho^*(\{A\} \cup \mathit{dep}(A)) = \rho^*(\{A, D_2, B_1\}) = 2$, 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.

For the bag $\{A\} \cup dep(A) = \{A, D_2, B_1\}$, we get $\rho^*(\{A, D_2, B_1\} \setminus \{A, B_1\}) = 1$ for the atom $R(A, B_1)$ and $\rho^*(\{A, D_2, B_1\} \setminus \{A, D_2\}) = 1$ for the atom $U(A, = D_2)$. Overall, the dynamic width of this variable order is 1. ☐

For hierarchical queries, the integral and fractional edge cover numbers are the same.

▶ **Lemma 20** (Lemma D.1 in [?]). *For any hierarchical query Q and $\mathcal{F} \subseteq vars(Q)$, it holds $\rho^*(\mathcal{F}) = \rho(\mathcal{F})$.*

Prior work defined the static and the dynamic width of conjunctive queries without access patterns [?]. It was shown that for any hierarchical conjunctive query with static width w and dynamic width δ, it holds $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$ (Proposition 3.7 in [?]). The proof can easily be adapted to the width measures of CQAPs. The only change is that we argue over access-top variable orders for the fractures of CQAPs instead of free-top variable orders for conjunctive queries.

▶ **Proposition 21** (Corollary of Proposition 3.7 in [?]). *For any CQAP with hierarchical fracture, static width w and dynamic width δ, it holds either $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$.*

C Missing Details in Section 4

C.1 Proof of Theorem 7

▶ **Theorem 7.** *Given a CQAP with static width w and dynamic width δ and a database of size N, the query can be evaluated with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(N^{\delta})$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

Given a CQAP Q with static width $\mathsf{w}(Q) = \mathsf{w}$ and dynamic width $\delta(Q) = \delta$ and a database of size N, we show that our approach presented in Section 4 evaluates Q with $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time, $\mathcal{O}(N^{\delta})$ update time, and $\mathcal{O}(1)$ enumeration delay. Consider an access-top variable order ω for the fracture $Q_\dagger$ with $\mathsf{w}(\omega) = \mathsf{w}$ and $\delta(\omega) = \delta$. In the following, we analyse each of the three stages preprocessing, update, and enumeration.

Preprocessing

Without loss of generality, assume that ω consists of a single tree. Otherwise, we do the analysis below for each of the constantly many trees in ω. The preprocessing stage consists of materialising the view tree $T = \tau(\omega)$ where τ is the function given in Figure 2. We show by induction on the structure of T that every node in T can be materialised in $\mathcal{O}(N^{\mathsf{w}})$ time.

Base Case: Each leaf atom or indicator projection in T can be materialised in linear time.

Induction Step: Consider an auxiliary view V'_X in T for $X \in vars(\omega)$. By construction, this view results from its single child view V_X by marginalising out variable X. By induction hypothesis, the view V_X can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time, hence its size has the same complexity bound. We can compute V'_X by scanning over the tuples in V_X and maintaining during the scan the count $|\sigma_{\mathcal{S}=\mathbf{s}} V_X|$ for each tuple $\mathbf{s}$ in $\pi_{\mathcal{S}} V_X$. This can be done in $\mathcal{O}(N^{\mathsf{w}})$ overall time.

Consider now a view $V_X(\mathcal{S})$ in T with $X \in vars(\omega)$ and $\mathcal{S} = \{X\} \cup dep_\omega(X)$. Let $V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ be the child nodes of V_X. Each child node can be a view, an atom, or an indicator projection. By induction hypothesis, the child nodes of V_X can be materialised in $\mathcal{O}(N^{\mathsf{w}})$ time. Consider any variable Y that occurs in the schemas of at least two child nodes of V_X. This means that $Y \in \mathcal{S} = \{X\} \cup dep_\omega(X)$. Hence, any variable that does not occur in $\mathcal{S}$ cannot be a join variable for the child views of V_X. We first marginalise out the variables in the child views that do not occur in $\mathcal{S}$. This can be done in $\mathcal{O}(N^{\mathsf{w}})$ time. Let $V_1'(\mathcal{S}_1'), \dots, V_k'(\mathcal{S}_k')$ be the resulting views. The view V_X can now be rewritten as $V_X(\mathcal{S}) = V_1'(\mathcal{S}_1'), \dots, V_k'(\mathcal{S}_k')$. Since the views $V_1', \dots, V_k'$ result from joining the leaf atoms (and indicator projections) in ω_X, we can upper-bound the computation time for V_X by $\mathcal{O}(N^p)$ where $p = \rho^*_{Q_X}(\mathcal{S})$ [**?**]. Recall that Q_X is the query that joins all atoms and indicator projections in ω_X. It follows from the definition of w that p is upper-bounded by w. We conclude that the view V_X can be computed in $\mathcal{O}(N^{\mathsf{w}})$ time.

Enumeration

Assume that $\mathcal{I}$ and $\mathcal{O}$ are the input and respectively output variables of Q and let $\mathcal{I}_\dagger$ be the input variables of $Q_\dagger$. We show that for any input tuple $\mathbf{i}$ over $\mathcal{I}$, the tuples in $Q(\mathcal{O}|\mathbf{i})$ can be enumerated with constant delay using the view trees constructed in the preprocessing stage. Let $\omega_1, \dots, \omega_n$ be the trees in ω and $\tau(\omega_1) = T_1, \dots, \tau(\omega_n) = T_n$ the view trees constructed from the variable order ω. For $j \in [n]$, let $Q_j(\mathcal{O}_j|\mathcal{I}_j)$ with $\mathcal{O}_j = \mathcal{O} \cap \mathit{vars}(\omega_j)$ and $\mathcal{I}_j = \mathcal{I}_\dagger \cap \mathit{vars}(\omega_j)$ be the CQAP that joins the atoms appearing at the leaves of T_j. We first explain how for any $j \in [n]$ and $\mathbf{i}_j$ over $\mathcal{I}_j$, the tuples in $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ can be enumerated with constant delay using the view tree T_j. Since the view tree is constructed following an access-top variable order, it holds that all views V_X where X is free (input) are above the views V_Y where Y is bound (output). To construct the first output tuple in $Q_j(\mathcal{O}_j|\mathbf{i}_j)$, we traverse T_j in preorder and do the following at each view V_X, where X is free. If $X \in \mathcal{I}_j$, i.e., it is an input variable, we check if the projection of $\mathbf{i}_j$ onto the schema of V_X is included in V_X. If not, $Q_i(\mathcal{O}_j|\mathbf{i}_j)$ is empty and we stop the traversal. Otherwise, we continue with the traversal. When we arrive at a view V_X with $X \in \mathcal{O}_j$, we have already fixed a tuple $\mathbf{t}$ over the variables in the root path of X. We retrieve in constant time the first value in $\sigma_{\mathcal{S}=\mathbf{t}'}\pi_X V_X$, where $\mathcal{S}$ is the schema of V_X excluding X and $\mathbf{t}' = \mathbf{t}[\mathcal{S}]$. After all views V_X with free X are visited, we have fixed all values over the variables in $\mathcal{O}_i$, hence we report the tuple consisting of these values. Then, we iterate over the remaining distinct Y-values in the last visited view V_Y with constant delay (given that the values over the root path of Y are fixed). For each distinct Y-value, we obtain a new tuple that we report. After all Y-values are exhausted, we backtrack.

Assume that we can enumerate the tuples in $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ with constant delay for any $j \in [n]$ and tuple $\mathbf{i}_j$ over $\mathcal{I}_j$. Consider a tuple $\mathbf{i}$ over $\mathcal{I}$. It holds $Q(\mathcal{O}|\mathbf{i}) = \times_{j\in[n]} Q_j(\mathcal{O}_j|\mathbf{i}_j)$ where $\mathbf{i}_j[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q. We enumerate the tuples in $Q(\mathcal{O}|\mathbf{i})$ by interleaving the enumeration procedures for $Q_1(\mathcal{O}_1|\mathbf{i}_1), \dots, Q_n(\mathcal{O}_n|\mathbf{i}_n)$, as follows.

```
foreach o_1 ∈ Q_1(O_1|i_1)
  ···
    foreach o_n ∈ Q_n(O_n|i_n)
      report o_1···o_n
```

That is, we first retrieve the first complete tuple $\mathbf{o}_j$ from $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ for each $j \in [n]$ and report $\mathbf{o}_1 \cdots \mathbf{o}_n$. Then, we iterate over the remaining tuples in $Q_n(\mathcal{O}_n|\mathbf{i}_n)$. For each such tuple $\mathbf{o}_n'$, we report $\mathbf{o}_1 \cdots \mathbf{o}_n'$. After all tuples in $Q_n(\mathcal{O}_n|\mathbf{i}_n)$ are exhausted, we move to the next tuple in $Q_{n-1}(\mathcal{O}_{n-1}|\mathbf{i}_{n-1})$ and restart the enumeration for $Q_n(\mathcal{O}_n|\mathbf{i}_n)$, and so on.

We conclude that the time to report the first tuple in $Q(\mathcal{O}|\mathbf{i})$, the time to report a next tuple after the previous one is reported, and the time to signalise the end of the enumeration after the last tuple is reported is constant.

Updates

We show that the view trees constructed in the preprocessing stage can be updated in $\mathcal{O}(N^{\delta})$ time under single-tuple updates to the base relations. Consider a single-tuple update to a base relation R. We first update each view tree referring to an atom of the form $R(\mathcal{X})$. Updating a view tree amounts to computing the deltas of the views on the path from $R(\mathcal{X})$ to the root of the view tree. We have shown above that for each variable X, the views V_X and V'_X can be materialised in $\mathcal{O}(N^p)$ time where $p = \rho^*_{Q_X}(\{X\} \cup dep_\omega(X))$. Since the update fixes the values in $\mathcal{X}$, the time to compute the delta of these views under the update becomes $\mathcal{O}(N^d)$ where $d = \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{X})$. A single-tuple update to R can trigger a single-tuple update to each indicator view of the form $I_{\mathcal{Z}}(R(\mathcal{Z}))$. Analogously to the reasoning above, we conclude that the time to compute the deltas of the views under such updates is $\mathcal{O}(N^d)$ where $d = \rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Z})$. It follows from the definition of the dynamic width δ of ω, that in both cases the exponent d is upper-bounded by δ. This implies that the overall update time is $\mathcal{O}(N^{\delta})$.

C.2 Evaluation of Cyclic CQAPs

▶ **Example 22.** We show in this example that the indicator projections can reduce the update time for a query no matter which VO is chosen as the strategy for the dynamic evaluation. Consider the following query:

$$\begin{aligned} Q(A,B,C,D,E,F,G,H,J \mid \cdot) = & R_1(A,B), R_2(B,C), R_3(C,A), R_4(A,D), R_5(D,E), \\ & R_6(B,F), R_7(F,G), R_8(C,H), R_9(H,J) \end{aligned}$$

It is a triangle query with three tails. Its fracture is same as the query itself. Figure 7 shows the hypergraph (top-left) of the query and three access-top VOs of the query. They are the optimal VOs that are rooted at variables A, D and E. That is, other VOs rooted at the corresponding variable do not admit smaller static and dynamic widths. Since the query is symmetric, the optimal VOs rooted at other variables are analogous to these three VOs.

Consider the VO in the top right of Figure 7. The indicator projection $I_{A,B}R_1$ is created under variable C to reduce the dynamic width of the query: The induced query Q_C at C contains the variables $\{C\} \cup dep(C) = \{A,B,C\}$. The dynamic width of the subtree ω_C rooted at C is defined as the fractional edge cover number of these variables minus the schema of an atom below C. If we choose the atom to be $R_9(H,J)$, the remaining variables are still $\{A,B,C\}$. With the indicator projection $I_{A,B}R_1$, the fractional edge cover number is $\rho^*(A,B,C) = \frac{3}{2}$ (by assigning a weight of $\frac{1}{2}$ to each atom $I_{A,B}R_1$, R_3 and R_2). Without $I_{A,B}R_1$, the fractional edge cover number is $\rho^*(A,B,C) = 2$. Hence, the indicator projection $I_{A,B}R_1$ reduces the dynamic width of ω_C from 2 to $\frac{3}{2}$. Since ω_C is the only subtree that has a dynamic width greater than 1, the dynamic width of the query Q is $\frac{3}{2}$.

The two VOs in the second row of Figure 7 are similar to the aforementioned VO: all have the variables A, B, and C in one root-to-leaf path, followed by the atom R_9, which has no intersection with A, B, and C. The indicator projection $I_{A,B}R_1$ created under variable C reduces the dynamic width from 2 to $\frac{3}{2}$ in the same way. Hence, the indicator projections can reduce the dynamic width, and thus the update time of the query Q for all VOs. ◀

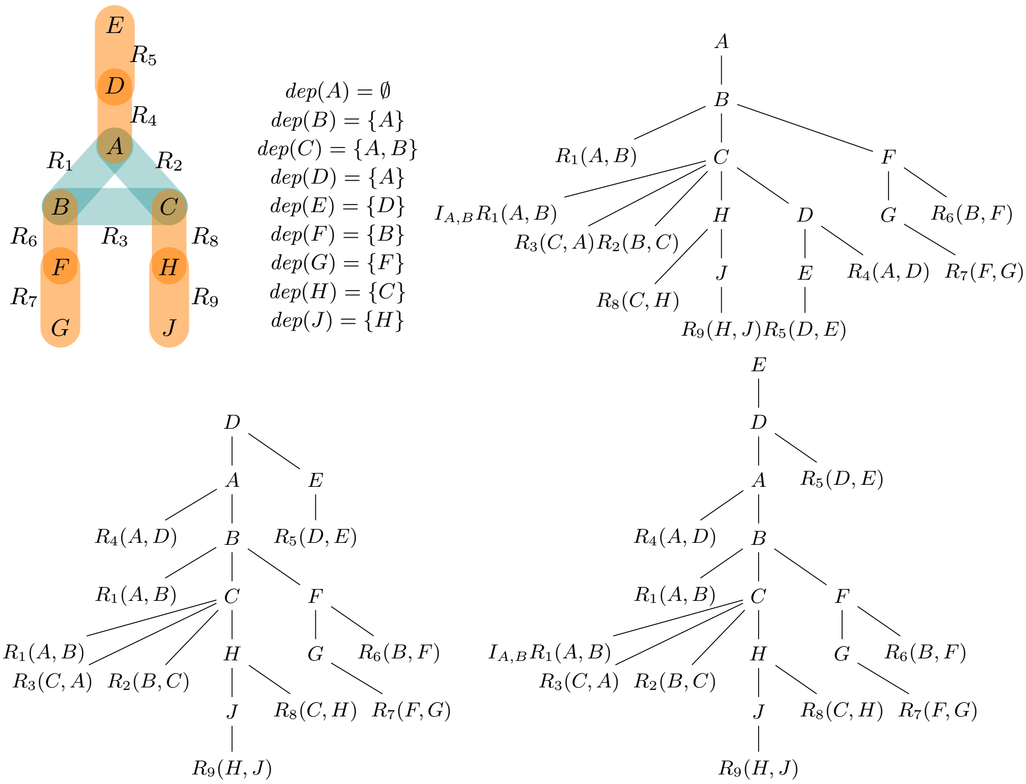


Figure 7 Top left: The hypergraph of the query Q in Example 22. Remaining three: the optimal access-top VOs of the query Q with the roots A, D and E, respectively. All other access-top VOs are analogous to these three VOs. The dependent sets of the two VOs in the second row are omitted.

▶ **Example 23.** Consider the triangle CQAP query

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

The fracture $Q_\dagger$ of Q is the query itself.

Figure 8 shows the access-top VO ω for Q. The input variable A is on top of the output variables B and C. At variable C, the function indicators from Figure 1 creates an indicator projection $I_{A,B}R$ since the relation R is not under C but forms a cycle with the relations S and T. By adding $I_{A,B}R$ below C, the fractional edge cover number $\rho^*(\{C\} \cup dep(C)) = \rho^*(\{A, B, C\})$ of the query Q_C reduces from 2 to $\frac{3}{2}$. This fractional edge cover number is the largest one among the fractional edge cover numbers of the queries induced by other variables, thus the static width of the VO ω is $\frac{3}{2}$.

In the preprocessing stage, we construct the view tree following the VO as shown in Figure 8 (second from left). The view V_C joins the relations R and S and the indicator projection $I_{A,B}R$, which can be computed in $\mathcal{O}(N^{\frac{3}{2}})$ time using a worst-case optimal join algorithm. The view V_B can be computed in linear time by looking up each tuple from V'_C in R. The views V'_C and V_A are constructed by marginalising out one variable at a time in time $\mathcal{O}(N^{\frac{3}{2}})$ and $\mathcal{O}(N)$ time, respectively. Hence, the view tree construction takes $\mathcal{O}(N^{\frac{3}{2}})$ time.

In the enumeration stage, we need to answer the query $Q(B, C|a)$, i.e., enumerate the tuples over the output variables B and C for an input value a over A from the view tree. We first check if a is in the root view V_A. If yes, we keep retrieving the next B-value b paired with a in V_B, and then the next C-value c paired with a and b in V_C, until all values are retrieved.

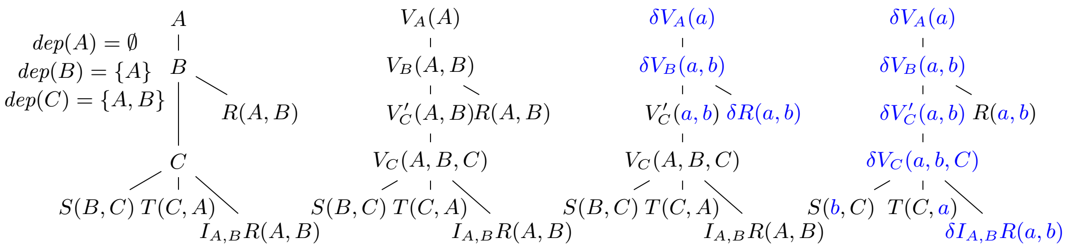


Figure 8 From left to right: Access-top VO for the query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$; the view tree constructed from the VO; the two delta view trees under a single-tuple update to R.

Each combination of the B- and C-values forms a new output tuple of $Q(B, C|a)$. These operations can be done in constant time per our data model (Section 2), so the enumeration delay is constant.

In the update stage, consider a single-tuple update $\delta R = \{(a, b) \to m\}$ to R, the base relation R and the indicator projection $I_{A,B}R$ are affected by the update. We compute two delta view trees shown on the right in Figure 8 for changes in R and respectively $I_{A,B}R$. In the delta view tree for changes to R (the left one), computing the delta $\delta V_B(a, b) = V'_C(a, b), \delta R(a, b)$ requires a constant lookup in V'_C; computing $\delta V_A(a) = \delta V_B(a, b)$ takes constant time. In the delta view tree for changes to $I_{A,B}R$ (the right one), computing the delta $\delta V_C(a, b, C) = S(b, C), T(C, a), \delta I_{A,B}R(a, b)$ requires intersecting the C-values that are paired with b in S and with a in T, which takes $\mathcal{O}(N)$ time; computing $\delta V'_C(a, b) = \delta V_C(a, b, C)$ requires aggregating away $\mathcal{O}(N)$ C-values; computing δV_B and δV_A takes constant time. Overall, a single-tuple update to R takes $\mathcal{O}(N)$ time. The delta view trees for changes to S and T are analogous. Hence, the update time of the query Q is $\mathcal{O}(N)$. ◀

D Missing Details in Section 5

D.1 Proof of Theorem 12

▶ **Theorem 12.** *Let any CQAP query Q and database of size N.*

- *If Q is in $CQAP_0$, then it admits $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ enumeration delay, and $\mathcal{O}(1)$ update time for single-tuple updates.*
- *If Q is not in $CQAP_0$ 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 $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time, for any $\gamma > 0$, unless the* OMv *conjecture fails.*

We start with an auxiliary lemma and a proposition.

▶ **Lemma 24.** *If a CQAP Q can be evaluated with $\mathcal{O}(f_p(N))$ preprocessing time, $\mathcal{O}(f_e(N))$ enumeration delay, and $\mathcal{O}(f_u(N))$ 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 $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_\dagger(\mathcal{O}|\mathcal{I}_\dagger)$, and a database $\mathcal{D}$ for $Q_\dagger$ of size N. We call a fresh variable A in $Q_\dagger$ 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 $Q_\dagger$. 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 $\mathbf{t}$ over $\mathcal{I}$, we denote by $(Q(\mathcal{O}|\mathbf{t}), \mathcal{D}_i)$ the result of Q for input $\mathbf{t}$ evaluated on $\mathcal{D}_i$. The result consists of the tuples over the output variables in C_i for the given input tuple $\mathbf{t}$, paired with the dummy value d_i over the output variables not in C_i. Intuitively, the result of $Q_\dagger$ 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 $\mathbf{t}_\dagger$ over $\mathcal{I}_\dagger$. We define for each $i \in [n]$, a tuple $\mathbf{t}_i$ over $\mathcal{I}$ such that $\mathbf{t}_i[A] = \mathbf{t}_\dagger[A']$ if A' is a representative of A. The result of $Q_\dagger(\mathcal{O}|\mathbf{t}_\dagger)$ on $\mathcal{D}$ is equal to the Cartesian product $\times_{i\in[n]}\pi_{\mathcal{O}_i}(Q(\mathcal{O}|\mathbf{i}), \mathcal{D}_i)$, where $\mathcal{O}_i$ is the set of output variables of Q contained in C_i. Now, assume that we want to enumerate the result of $(Q_\dagger(\mathcal{O}|\mathbf{t}_\dagger), \mathcal{D})$. We start the enumeration procedure for each $Q(\mathcal{O}|\mathbf{i}), \mathcal{D}_i)$ with $i \in [n]$. For each $\mathbf{t}'_1 \in Q(\mathcal{O}|\mathbf{t}_1), \mathcal{D}_1)$, $\ldots$, $\mathbf{t}'_n \in Q(\mathcal{O}|\mathbf{t}_n), \mathcal{D}_n)$, we return the tuple $\pi_{\mathcal{O}_1}\mathbf{t}'_1 \circ \ldots \circ \pi_{\mathcal{O}_n}\mathbf{t}'_n$. This implies that the result of $(Q_\dagger(\mathcal{O}|\mathbf{t}_\dagger), \mathcal{D})$ 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 $\{\mathbf{t} \mapsto m\}$ to a relation R that is contained in the connected component C_i for some $i \in [n]$. We apply the update $\{\mathbf{t}_\mathcal{I} \mapsto m\}$ to relation R in $\mathcal{D}_i$, where $\mathbf{t}_\mathcal{I}$ is the tuple over $\mathcal{I}$ defined as:

$$\mathbf{t}_\mathcal{I}[A] = \begin{cases} \mathbf{t}[A'] & \text{if } A' \text{ is a representative of } A \\ \mathbf{t}[A] & \text{otherwise} \end{cases}$$

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

Overall, we obtain an evaluation procedure for $Q_\dagger$ with $\mathcal{O}(f_p(N))$ preprocessing time, $\mathcal{O}(f_e(N))$ enumeration delay, and $\mathcal{O}(f_u(N))$ amortised update time. ◀

▶ **Proposition 25.** *Every $CQAP_0$ query has dynamic width* 0 *and static width* 1.

Proof. Consider a CQAP_0 query Q and its fracture $Q_\dagger$. We first show that the dynamic width of Q is 0. By definition, $Q_\dagger$ is hierarchical, free-dominant, and input-dominant. Hierarchical queries admit canonical VOs. In canonical VOs, it holds: If a variable A dominates a variable B, then, A is on top of B. Hence, $Q_\dagger$ admits a canonical VO that is access-top. Consider a variable X in ω and an atom $R(\mathcal{Y})$ in the subtree ω_X rooted at X. By the definition of canonical VOs, it holds: the dependency set of X consists of the ancestor variables of X; $\mathcal{Y}$ contains X and all ancestor variables of X. Hence, we have $\rho^*_{Q_X}((\{X\} \cup dep_\omega(X)) \setminus \mathcal{Y}) = \rho^*_{Q_X}((\{X\} \cup \mathsf{anc}_\omega(X)) \setminus \mathcal{Y}) = \rho^*_{Q_X}(\emptyset) = 0$. This implies that the dynamic width of ω is 0. This means that the dynamic width of $Q_\dagger$, hence, the dynamic width of Q is 0.

It follows from Proposition 21 that the static width of Q is 1[1]. ◀

We are ready to prove Theorem 12.

Complexity Upper Bound

We prove the first statement in Theorem 12. Assume that Q is in CQAP_0. By Proposition 25, Q has dynamic width 0. By definition of CQAP_0, the fracture $Q_\dagger$ of Q must be hierarchical. It follows from Proposition 21 that the static width of $Q_\dagger$, hence the static width of Q, is at most 1. Using Theorem 7, we conclude that Q can be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.

[1] To simplify the presentation, we assume that Q contains at least one variable, so it has static width at least 1. Otherwise, it can trivially be evaluated with constant preprocessing time, update time, and enumeration delay.

Complexity Lower Bound

We prove the second statement in Theorem 12. The proof is based on a reduction of the Online Matrix-Vector Multiplication (OMv) problem (Definition 13) to the evaluation of non-CQAP_0 queries.

We start with the high-level idea of the proof. Consider the following simple CQAPs, which are not in CQAP_0.

$$Q_1(\mathcal{O}|\cdot) = R(A), S(A,B), T(B) \quad \mathcal{O} \subseteq \{A, B\}$$
$$Q_2(A|\cdot) = R(A,B), S(B)$$
$$Q_3(\cdot|A) = R(A,B), S(B)$$
$$Q_4(B|A) = R(A,B), S(B)$$

Each query is equal to its fracture. 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 [?]. 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, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ 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 CQAP_0 and does not have repeating relation symbols.

In each of the following two reductions, our starting assumption is that there is an algorithm $\mathcal{A}$ that evaluates the given query with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay for some $\gamma > 0$. We then show that $\mathcal{A}$ can be used to design an algorithm $\mathcal{B}$ that solves the OMv problem in subcubic time.

Hardness for Q_3

Given $n \geq 1$, let $\mathbf{M}$, $\mathbf{v}_1$, ..., $\mathbf{v}_n$ be an input to the OMv problem, where $\mathbf{M}$ is an $n \times n$ Boolean Matrix and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are Boolean column vectors of size n. Algorithm $\mathcal{B}$ uses relation R to encode matrix $\mathbf{M}$ and relation S to encode the incoming vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$. The database domain is $[n]$. First, algorithm $\mathcal{B}$ 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 $\mathbf{M}(i,j) = 1$. Afterwards, it performs a round of operations for each incoming vector $\mathbf{v}_r$ 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 $\mathbf{v}_r(j) = 1$. Observe that $Q_3(\cdot|i)$ is true for some $i \in [n]$ if and only if $(\mathbf{M}\mathbf{v}_r)(i) = 1$. Algorithm $\mathcal{B}$ constructs the result vector $\mathbf{u}_r = \mathbf{M}\mathbf{v}_r$ as follows. It asks for each $i \in [n]$, whether $Q_3(\cdot|i)$ is true, i.e., i is in the result of Q_3. If yes, the i-th entry of the result of $\mathbf{u}_r$ is set to 1, otherwise, it is set to 0.

Time Analysis. The size of the database remains $\mathcal{O}(n^2)$ during the whole procedure. Algorithm $\mathcal{B}$ needs at most n^2 updates to encode $\mathbf{M}$ by relation R. Hence, the time to execute these updates is $\mathcal{O}(n^2(n^2)^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{3-2\gamma})$. In each round r with $r \in [n]$, algorithm $\mathcal{B}$ executes n updates to encode vector $\mathbf{v}_r$ into relation S and asks for the result of $Q_3(\cdot|i)$ for every $i \in [n]$. The n updates and requests need $\mathcal{O}(n(n^2)^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{2-2\gamma})$ time. Hence, the overall time for a single round is $\mathcal{O}(n^{2-2\gamma})$. Consequently, the time for n rounds is $\mathcal{O}(nn^{2-2\gamma}) = \mathcal{O}(n^{3-2\gamma})$. This means that the overall time of the reduction is $\mathcal{O}(n^{3-2\gamma})$ in worst-case, which is subcubic.

Hardness for Q_4

The reduction differs slightly from the case for Q_3 in the way algorithm $\mathcal{B}$ constructs the result vector $\mathbf{u}_r = \mathbf{M}\mathbf{v}_r$ in each round r. For each $i \in [n]$, it starts the enumeration process for $Q_4(B|i)$. If one tuple is returned, it stops the enumeration process and sets the i-th entry of $\mathbf{u}_r$ to be 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 $\mathbf{u}_r$ is the same as in case of Q_3. Hence, the overall time of the reduction stays subcubic.

Hardness in the General Case

Consider now an arbitrary CQAP query Q that is not in CQAP$_0$ and does not have repeating relation symbols. Since Q is not in CQAP$_0$, this means that its fracture $Q_\dagger$ is either not hierarchical, not free-dominant, or not input-dominant. If $Q_\dagger$ 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 $Q_\dagger$ with $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time for any $\gamma > 0$, unless the OMv conjecture fails [?]. By Lemma 24, no such algorithm can exist for Q. Hence, we assume that $Q_\dagger$ is hierarchical and consider two cases:

(1) $Q_\dagger$ is not free-dominant and all free variables are input
(2) $Q_\dagger$ is free-dominant but not input-dominant

Case (1). The query must contain an input variable A and a bound variable B such that $atoms(A) \subset atoms(B)$. 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 $\mathcal{A}$ that evaluates $Q_\dagger$ with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time for some $\gamma > 0$. We will design an algorithm $\mathcal{B}$ that evaluates Q_3 with the same complexities. This rejects the OMv conjecture. Hence, by Lemma 24, 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 $(\mathcal{S}, A = a, B = b, d)$ to denote a tuple over schema $\mathcal{S}$ that assigns the values a and b to the variables A and respectively B and all other variables in $\mathcal{S}$ to d. Likewise, $(\mathcal{S}, B = b, d)$ denotes a tuple that assigns value b to B and all other variables in $\mathcal{S}$ to d. Algorithm $\mathcal{B}$ first constructs from $\mathcal{D}$ a database $\mathcal{D}'$ for $Q_\dagger$ as follows. For each tuple (a, b) in relation 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 $\mathcal{B}$ executes the preprocessing for $Q_\dagger$ on $\mathcal{D}'$. Each single-tuple update $\{(a, b) \mapsto m\}$ to relation 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 $\{(\mathcal{S}', B = b, d) \mapsto m\}$ to all relations $\mathcal{S}'$ with $\mathcal{S}'(\mathcal{Y}') \in \mathcal{S}_{(\neg A,B)}$. Hence, the amortised update time is $\mathcal{O}(N^{0.5-\gamma})$. Each input tuple (a) for Q_3 is translated into an input tuple $(\mathcal{I}_\dagger, A = a, d)$ for $Q_\dagger$ where $\mathcal{I}_\dagger$ is the set of input variables for $Q_\dagger$. Recall that all free variables of $Q_\dagger$ are input. The answer of $Q_3(\cdot|a)$ is true if and only if the answer of $Q_\dagger(\cdot|(\mathcal{I}_\dagger, A = a, d))$ is true. The answer time is

$\mathcal{O}(N^{0.5-\gamma})$. We conclude that Q_3 can be evaluated with $\mathcal{O}(N^{0.5-\gamma})$ enumeration delay and $\mathcal{O}(N^{0.5-\gamma})$ amortised update time, a contradiction due to the OMv conjecture.

Case (2). We now consider the case that the query $Q_\dagger$ is free-dominant but not input-dominant. In this case, the we reduce the evaluation of Q_4 to the evaluation of $Q_\dagger$. 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 $Q_\dagger$ where all input variables besides A are assigned to d. Recall that $Q_\dagger$ might have several output variables besides B. By construction, they can be assigned only to d. Hence, all output tuples returned by $Q_\dagger$ have distinct B-values. These B-values constitute the result of $Q_4(B|a)$. We conclude that Q_4 can be evaluated with $\mathcal{O}(N^{0.5-\gamma})$ enumeration delay and $\mathcal{O}(N^{0.5-\gamma})$ amortised update time, which contradicts the OMv conjecture.

E Missing Details in Section 6

E.1 Comparison with Prior Approaches

We compare our adaptive maintenance strategy with typical eager and lazy approaches.

▶ **Example 26.** Let us consider the 4-cycle query from Example 2:

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

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

We can recover the complexities for typical eager and lazy approaches 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	$\mathcal{O}(N^{1+\epsilon})$	$\mathcal{O}(N^{\epsilon})$	$\mathcal{O}(N^{1-\epsilon})$

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 $\mathcal{O}(N^2)$ pairs of values. This sequence takes time (excluding preprocessing) $\mathcal{O}(m(N+k))$ in the eager and lazy approaches and $\mathcal{O}(m(N^{\epsilon}+kN^{1-\epsilon}))$ 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 that 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
	0	0.5	0.75	1	1.5	2	1	1
$\log_N m$	0.5	1	1.25	1.5	2	2.5	1.5	1.5
	1	1.5	1.75	2	2.5	3	2	2
	ϵ	0.5	0.75	1	1	1		

The middle five columns show the complexities for our general approach for various values of k. The last row states the value of ϵ, 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 $\mathcal{O}(N^{0.5})$ for $\log_N k = 0$ (with $\epsilon = 0.5$) and $\mathcal{O}(N^{0.25})$ 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 $\epsilon = 1$. ☐

E.2 Further Notation

We introduce some notation that will be useful in the following sections. Given a query and a variable X, we denote by $\mathit{vars}(\mathit{atoms}(X))$, $\mathit{free}(\mathit{atoms}(X))$, and $\mathit{in}(\mathit{atoms}(X))$, the sets of all, free and respectively input variables contained in $\mathit{atoms}(X)$. For a VO ω, $\mathit{bound}(\omega)$ and $\mathit{out}(\omega)$ are the sets of bound and respectively output variables in ω. Given a VO ω and a tuple $p = (X_1, \ldots, X_k)$ of variables, we denote by $(p \circ \omega)$ the VO defined as follows: X_1 is the root, X_{i+1} is the single child of X_i for $i \in [k-1]$, and ω is the single child tree of X_k. Consider the canonical VO ω of a hierarchical CQAP and the subtree ω_X of ω 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 ω_X. The set $\mathcal{I}_X$ consists of the input variables in ω_X and the variables on the path from X to a root of ω. The set $\mathcal{O}_X$ consists of the output variables in ω_X.

E.3 Preprocessing

Our query evaluation technique consists of three distinct, yet interdependent stages: preprocessing, updates and enumeration. This section addresses preprocessing, with the following two sections addressing updates and enumeration. Whenever we refer to the query in the three stages, we mean the hierarchical fracture of the input CQAP.

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 materialized view trees. Each view tree, which is modelled on a specific VO, 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 VOs corresponding to evaluation strategies for different parts of the query result. Each such VO is constructed from the canonical VO of the query by turning some of its subtrees into access-top VOs. Second, we construct from each VO a view tree. We obtain a view tree from a variable order by replacing each variable X by a view over X and its dependency set.

We describe the preprocessing stage in the following three subsections. In Section E.3.1 we give a function that turns canonical VOs into optimal access-top ones. In Section E.3.2 we explain how to obtain different VOs from the canonical VO of the hierarchical query by using the above function. In Section E.3.3 we describe the construction of view trees from VOs. To simplify the presentation, we assume in the following that the VO of the considered hierarchical query contains of a single tree. Otherwise, we apply the preprocessing stage to each tree in the VO.

AccessTop(VO ω, access pattern $(\mathcal{O}|\mathcal{I})$) : VO

switch ω:

$R(\mathcal{Y})$

1 **return** $R(\mathcal{Y})$

$\begin{array}{c} X \\ / \quad \backslash \\ \omega_1 \cdots \omega_k \end{array}$

2 **let** $\omega'_i = \text{AccessTop}(\omega_i, (\mathcal{O}|\mathcal{I})), \forall i \in [k]$

3 **let** $\mathcal{D} = \begin{cases} \emptyset & \text{if } X \in \mathcal{I} \\ vars(\omega) \cap \mathcal{I}, & \text{else if } X \in \mathcal{O} \\ vars(\omega) \cap (\mathcal{I} \cup \mathcal{O}) & \text{otherwise} \end{cases}$

4 **let** $\{\hat{\omega}^i_1, \ldots, \hat{\omega}^i_{m_i}\} = \Delta(\omega'_i, \mathcal{D}),\ \forall i \in [k]$

5 **let** $(X_1, ..., X_\ell) = \mathcal{D} \cap \mathcal{I} +\!\!+ \mathcal{D} \cap \mathcal{O}$ be an ordering that is compatible with the partial order of ω

6 **return** $\begin{array}{c} X_1 \\ \vdots \\ X_\ell \\ | \\ X \\ \hat{\omega}^1_1 \cdots \hat{\omega}^1_{m_1} \quad \cdots \quad \hat{\omega}^k_1 \cdots \hat{\omega}^k_{m_k} \end{array}$

Figure 9 Construction of an access-top VO from a canonical VO ω of a hierarchical CQAP with access pattern $(\mathcal{O}|\mathcal{I})$. The function $\Delta(\omega', \mathcal{D})$, defined in Figure 10, deletes the variables in $\mathcal{D}$ from the VO ω'.

E.3.1 From Canonical to Access-Top VOs

Given a canonical VO ω of a hierarchical CQAP Q with input variables $\mathcal{I}$ and output variables $\mathcal{O}$, the function AccessTop$(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 9 returns an access-top VO for Q with optimal static and dynamic width. The function proceeds recursively on the structure of ω. At a variable X, the function selects a set $\mathcal{D}$ of variables from the subtree ω' 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 defines $\mathcal{D}$ to be the input variables in ω', and 3) if X is bound, the function sets $\mathcal{D}$ to be the free variables in ω' (Line 3). The function then takes out $\mathcal{D}$ from ω' and puts them on top of X (Lines 4-6). Line 5 makes sure the input variables are put on top of the output variables.

The deletion of a set $\mathcal{D}$ of variables from a VO ω is implemented by the function $\Delta(\omega, \Delta)$ in Figure 10. The function traverses recursively over all variables in ω. If a variable X is not included in $\mathcal{D}$, the function does not change the structure of ω (Lines 3-4). In case $X \in \mathcal{D}$ and X has a parent Y, it appends the child trees of X to the variable Y (Lines 5-6). If $X \in \mathcal{D}$ and X has no parent, the child trees of X become independent (Line 7).

▶ **Example 27.** Figure 11 (left and middle) shows the hypergraphs of the query

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

and of its fracture

$$Q_\dagger(B, C, D, E \mid A_1, A_2) = R(A_1, B, C), S(A_1, B, D), T(A_2, E).$$

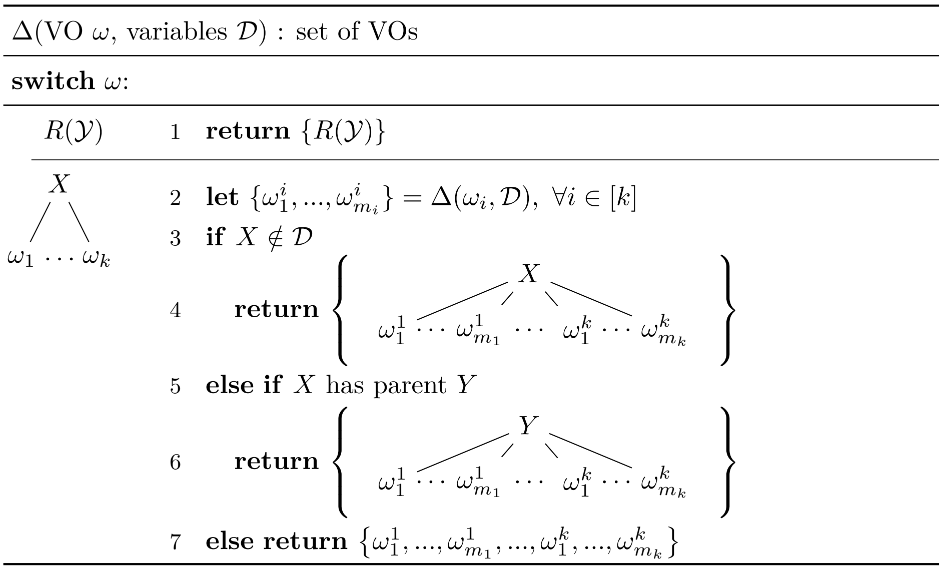


Figure 10 Deletion of a set $\mathcal{D}$ of variables from a VO ω. If $X \in \mathcal{D}$ and X has a parent Y, the child trees of X are appended to Y. If $X \in \mathcal{D}$ and X has no parent, the child trees of X become independent.

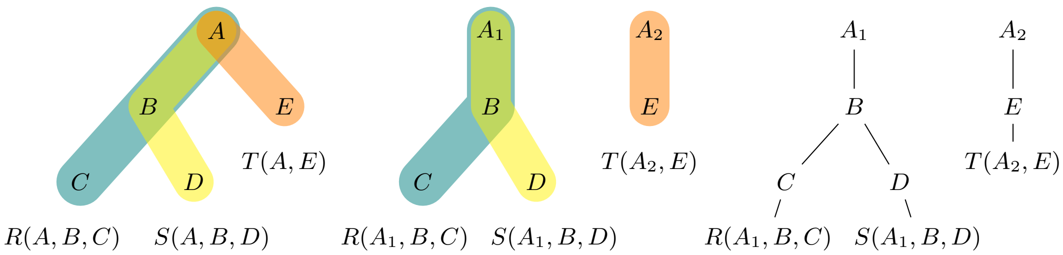


Figure 11 Left and middle: Hypergraphs of the query (left) and its fracture on input variable A (middle two) used in Example 27. Right two: The access-top VOs returned by AccessTop in Figure 9, which are the same as the canonical VOs.

The fracture is hierarchical, free-dominant and input-dominant. Hence, Q and $Q_\dagger$ are in CQAP_0. Figure 11 (right) depicts the access-top VOs for the queries whose bodies are the two connected components of the hypergraph of $Q_\dagger$, i.e., $Q_1(B,C,D|A_1) = R(A_1,B,C)$, $S(A_1,B,D)$ and $Q_2(E|A_2) = T(A_2,E)$. They are the canonical VOs of the two queries. ◀

▶ **Example 28.** Consider the query

$$Q(C, D \mid E) = R(A,B,C), S(A,B,D), T(A,E).$$

Figure 12 (left) shows the hypergraphs of the query. Its fracture is the query itself, which is hierarchical but not free-dominant. Figure 12 (middle) depicts the canonical VO of the query. Figure 12 (right) depicts the access-top VO for the query. The free variables C, D and E sit on top of the bound variables A and B. The input variable E sits on top of the output variables C and D. ◀

The function AccessTop in Figure 9 turns canonical VOs into optimal VOs.

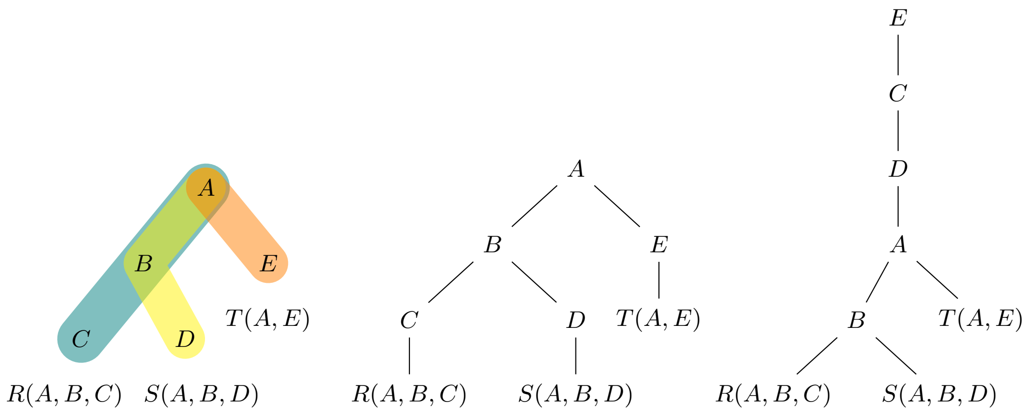


Figure 12 Left: Hypergraph of the query and its fracture used in Example 28. Middle: The canonical VO of the query. Right: The access-top VO returned by AccessTop in Figure 9.

▶ **Proposition 29.** *Given a CQAP Q, whose fracture $Q_\dagger(\mathcal{O}|\mathcal{I})$ is hierarchical, and a canonical VO ω for Q, $\text{AccessTop}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top VO for $Q_\dagger$ with static width $\mathsf{w}(Q)$ and dynamic width $\delta(Q)$.*

Before proving Proposition 29, we introduce some useful notation. Let ω be a canonical VO of a hierarchical CQAP. Let $\mathcal{F}$, $\mathcal{I}$, and $\mathcal{O}$ be the free, input, and respectively output variables of the query, and X a variable in ω. The following measures ξ and κ express the static and the dynamic width of ω_X without referring to access-top VOs.

$$\xi(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in bound(\omega_X)\\ Z \in out(\omega_X)}} \{\rho^*_{Q_X}(vars(\omega_Y) \cap \mathcal{F}), \rho^*_{Q_X}(vars(\omega_Z) \cap \mathcal{I})\}$$

$$\kappa(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in bound(\omega_X)\\ Z \in out(\omega_X)}} \max_{R(\mathcal{Y}) \in atoms(\omega_Y)}$$
$$\{\rho^*_{Q_X}((vars(\omega_Y) \cap \mathcal{F}) \setminus \mathcal{Y}), \rho^*_{Q_X}((vars(\omega_Z) \cap \mathcal{I}) \setminus \mathcal{Y})\}$$

In case ω_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 width of the variable orders returned by the function AccessTop in terms of the measures ξ and κ.

▶ **Lemma 30.** *Given a canonical VO ω of a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$, a variable X in ω, and the induced query Q_X at variable X, $\text{AccessTop}(\omega_X, (\mathcal{I}|\mathcal{O}))$ constructs a VO ω' such that $\omega^t = (\mathsf{anc}_\omega(X) \circ \omega')$ is an access-top VO for Q_X with $\mathsf{w}(\omega^t) = \max\{1, \xi(\omega_X, \mathcal{I}, \mathcal{O})\}$ and $\delta(\omega^t) = \kappa(\omega_X, \mathcal{I}, \mathcal{O})$.*

Proof. The function AccessTop traverses the given canonical VO and pulls up free variables such that the resulting VO 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 ω neither contains a bound variable above a free one nor an output variable above a bound one, the VO remains unchanged. Since a canonical VO has static width 1 and dynamic width 0, the statement in the lemma holds in this case.

Assume now that ω 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 ω that has free variables in its subtree. Let $\mathcal{F}$ be the set of free variables under X. Due to the structure of canonical VOs, 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 VO ω^t. Hence, the fractional edge cover number of $\{X\} \cup dep_{\omega^t}(X)$ 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 $\{Y\} \cup dep_{\omega^t}(Y)$ is upper-bounded by the fractional edge cover number of $\{X\} \cup dep_{\omega^t}(X)$.

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 $\{X\} \cup dep_{\omega^t}(X)$ is $\rho^*(\{X\} \cup \mathcal{V})$ and upper-bounds the fractional edge cover numbers at the other variables in the resulting VO ω^t.

Hence, the static width of ω^t is determined by the largest set of variables that is moved on top of a single variable by the function AccessTop.

For the dynamic width of ω^t, the reasoning is completely analogous. The dynamic width of ω^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 29.

Proof of Proposition 29. Consider a CQAP Q whose fracture $Q_\dagger(\mathcal{O}|\mathcal{I})$ is hierarchical. Let $\mathcal{F} = \mathcal{I} \cup \mathcal{O}$ and w and δ be the static and respectively dynamic width of Q. By the definition of static and dynamic width, $Q_\dagger$ must have static width w and dynamic width δ. Let ω be the canonical VO of $Q_\dagger$. Without loss of generality, assume that $Q_\dagger$ contains at least one atom with non-empty schema. Otherwise, AccessTop returns the set of atoms in $Q_\dagger$, which is already an optimal access-top VO for $Q_\dagger$. Assume also that ω consists of a single connected component. Otherwise, we apply the same reasoning for each connected component. By Lemma 30, $\text{AccessTop}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top VO ω^t for $Q_\dagger$ 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:

$$\max\{1, \xi(\omega, \mathcal{I}, \mathcal{O})\} \leq \mathsf{w} \tag{1}$$

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

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

We first consider Case (1.1). The inner nodes of each root-to-leaf path of a canonical VO are the variables of an atom. Hence, for each variable $Z \in \mathcal{F}'$, there must be an atom in $Q_\dagger$ that contains both Y and Z. This means that Y and Z depend on each other. Let $\omega' = (T, dep_{\omega'})$ be an arbitrary access-top VO for $Q_\dagger$. 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 $\mathsf{anc}_{\omega'}(Y)$. Thus, $\mathcal{F}' \subseteq dep_{\omega'}(Y)$. This means $\rho^*_{Q_Y}(\{Y\} \cup dep_{\omega'}(Y)) \geq \ell$, which implies $\mathsf{w}(\omega') \geq \ell$. It follows $\mathsf{w} \geq \ell$.

The reasoning for Case (1.2) is analogous. In any access-top VO $\omega' = (T, dep_{\omega'})$ for $Q_\dagger$, all variables in $\mathcal{I}'$ must be included in $\mathsf{anc}_{\omega'}(Y)$. Hence, $\mathcal{I}' \subseteq dep_{\omega'}(Y)$, which means $\rho^*_{Q_Y}(\{Y\} \cup dep_{\omega'}(Y)) \geq \ell$. This implies $\mathsf{w}(\omega') \geq \ell$, thus, $\mathsf{w} \geq \ell$.

It follows that the static width of the access-top VO $\textsc{AccessTop}(\omega, (\mathcal{I}|\mathcal{O}))$ must be $\mathsf{w}(Q)$.

Following similar steps, we can show:

$$\kappa(\omega, \mathcal{I}, \mathcal{O}) \leq \delta \tag{2}$$

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): ω contains a bound variable Y and an atom $R(\mathcal{Y})$ containing Y such that $\rho^*_Q(\mathcal{F}' \setminus \mathcal{Y}) = k$, where $\mathcal{F}' = vars(\omega_Y) \cap \mathcal{F}$
- Case (2.2): ω contains an output variable Y and an atom $R(\mathcal{Y})$ containing Y such that $\rho^*_Q(\mathcal{I}' \setminus \mathcal{Y}) = k$, where $\mathcal{I}' = vars(\omega_Y) \cap \mathcal{I}$.

We consider Case (2.1). Let $\omega' = (T, dep_{\omega'})$ be an arbitrary access-top VO for $Q_\dagger$. The atom $R(\mathcal{Y})$ must be included in $atoms(\omega'_Y)$, 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 $\mathcal{F}' \setminus \mathcal{Y}$ must be included in $\mathsf{anc}_{\omega'}(Y)$. Hence, $\mathcal{F}' \setminus \mathcal{Y} \subseteq dep_{\omega'}(Y)$. This implies that $\rho^*_{Q_Y}((\{Y\} \cup dep_{\omega'}(Y)) \setminus \mathcal{Y}) \geq k$. This means $\rho^*_{Q_Y}((\{Y\} \cup dep_{\omega'}(Y)) \setminus \mathcal{Y}) \geq k$. 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 $Q_\dagger(\mathcal{O}|\mathcal{I})$, $\textsc{AccessTop}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top VO with static width $\mathsf{w}(Q_\dagger) = \mathsf{w}(Q)$ and dynamic width $\delta(Q_\dagger) = \delta(Q)$. ◀

E.3.2 VOs Describing Evaluation Strategies

Each VO of a CQAP stands for an evaluation strategy for the query. In this section we show how we can derive from the canonical VO of a query to a set of VOs, which depict the evaluation strategies of the query result on different parts of the input relations.

We start with a high-level explanation of the construction. Consider the canonical VO ω of a hierarchical CQAP and a subtree ω' of ω 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 ω'. The $\mathcal{I}_X$ consists of the input variables in ω' and the root path of X. The set $\mathcal{O}_X$ consists of the output variables in ω'. Let ω'_{at} be an access-top VO of $Q_X(\mathcal{O}_X|\mathcal{I}_X)$. If Q_X is CQAP$_0$, we use ω'_{at} for the evaluation of Q_X. The view tree following ω'_{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, ω' must contain a bound or output variable Y such that Q_Y is not CQAP$_0$. If X is not this variable Y, we recursively process the subtrees of ω', otherwise, i.e., if X is this variable Y, we distinguish two cases based on the degree of values over $\mathsf{anc}_w(X) \cup \{X\}$. In the light case, we construct the view tree following the VO ω'_{at}. This view tree can be constructed and maintained under updates efficiently, since the values over $\mathsf{anc}_w(X) \cup \{X\}$ have bounded degree. In the heavy case, we use the VO ω'. The view tree following ω' allows for constant update time and an enumeration delay that depends on the number of distinct values over $\mathsf{anc}_w(X) \cup \{X\}$. Since these values have high degree, the number of distinct such values is bounded, which ensure efficient enumeration delay.

$\Omega(\text{VO } \omega, \text{access pattern } (\mathcal{O}|\mathcal{I}))$: set of VOs

switch ω:

$R^{sig}(\mathcal{Y})$

1 **return** $\{R^{sig}(\mathcal{Y})\}$

$$\begin{array}{c} X \\ / \quad \backslash \\ \omega_1 \cdots \omega_k \end{array}$$

2 **let** $key = \mathsf{anc}_\omega(X) \cup \{X\}$

3 **let** $\mathcal{I}_X = (\mathcal{I} \cap vars(\omega)) \cup \mathsf{anc}_\omega(X)$

4 **let** $\mathcal{O}_X = \mathcal{O} \cap vars(\omega)$

5 **let** $Q_X(\mathcal{O}_X|\mathcal{I}_X) =$ join of $atoms(\omega)$

6 **if** $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ is CQAP_0

7   **return** $\{\,\text{AccessTop}(\omega, (\mathcal{O}|\mathcal{I}))\,\}$

8 **if** $X \in \mathcal{I}$ **or** $(X \in \mathcal{O}$ **and** $vars(\omega) \cap \mathcal{I} = \emptyset)$

9   **return** $\left\{ \begin{array}{c} X \\ / \quad \backslash \\ \omega'_1 \cdots \omega'_k \end{array} \middle| \; \omega'_i \in \Omega(\omega_i, (\mathcal{O}|\mathcal{I})), \forall i \in [k] \right\}$

10 **let** $htrees = \left\{ \begin{array}{c} X \\ / \quad \backslash \\ \omega'_1 \cdots \omega'_k \end{array} \middle| \; \omega'_i \in \Omega(\omega_i^{key \to H}, (\mathcal{O}|\mathcal{I})), \forall i \in [k] \right\}$

11 **let** $ltree = \text{AccessTop}(\omega^{key \to L}, (\mathcal{O}|\mathcal{I}))$

12 **return** $htrees \cup \{\,ltree\,\}$

Figure 13 Construction of a set of VOs from a canonical VO ω of a hierarchical CQAP with access pattern $(\mathcal{O}|\mathcal{I})$. Each constructed VO corresponds to an evaluation strategy of some part of the query result. The VO $\omega^{key \to s}$ for $s \in \{H, L\}$ has the structure of ω but the HL-signature of each atom is extended by $key \to s$.

Given a canonical VO ω of a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$, the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 13 returns the set of all VOs for Q obtained from ω. The atoms at the leaves of these VOs are labelled by HL-signatures. When constructing view trees following these VOs, these atoms will be materialized with corresponding relation parts. That is, an atom $R^{sig}(\mathcal{Y})$ with $\mathcal{S} \to s \in sig$ will be materialized by a part of relation R that is heavy on $\mathcal{S}$ if $s = H$ and light on S if $s = L$. We assume that the atoms in the initial canonical VO ω passed as input to the function Ω are labelled by the empty HL-signature $\emptyset$.

We now describe the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in more detail. The function proceeds recursively on the structure of ω and considers at each variable X, the *induced query* $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ (Line 5). If Q_X is CQAP_0, the function returns an access-top VO constructed by the function $\text{AccessTop}(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 9 (Lines 6-7). If X is an input variable, or it is an output variable and ω 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 VOs for each child tree of X. For each combination of these VOs, it builds a new VO where X is on top of the child VOs (Line 9). Otherwise, if X is bound or an output variable and ω contains input variables, the function creates two evaluation strategies for Q_X based on the degree of values over $\{X\} \cup \mathsf{anc}(X)$. For the values over $\{X\} \cup \mathsf{anc}(X)$ 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 VO and to potentially fork into more strategies (Line 10). For the values over $\{X\} \cup \mathsf{anc}(X)$ that are *light*, the function constructs an access-top VO for ω

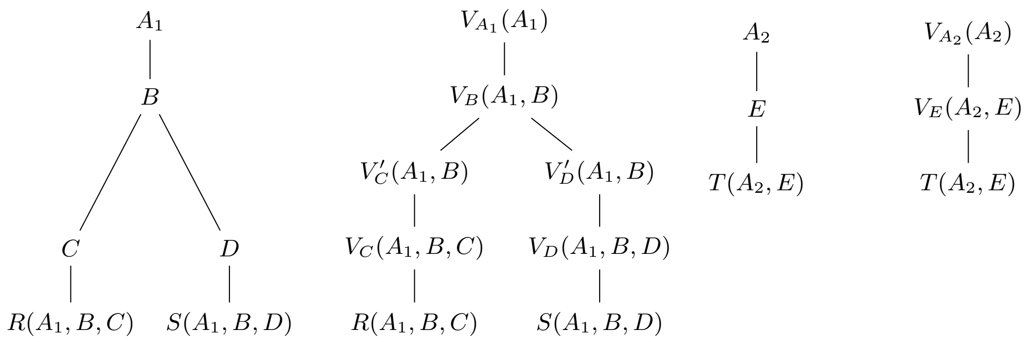


Figure 14 VOs constructed for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E|A_2) = T(A_2, E)$ in Example 27 and their corresponding view trees.

(Line 11).

▶ **Example 31.** Consider the CQAP_0 query

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

and the two queries from the decomposition of its fracture:

$$Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D) \text{ and } Q_2(E|A_2) = T(A_2, E)$$

from Example 27. Figure 14 (left and middle right) shows the VOs, i.e., the evaluation strategies, for the VOs of the two queries returned by Ω. Since Q is in CQAP_0, the VOs for evaluation are exactly the access-top VOs of the two queries. ◀

▶ **Example 32.** Consider the query

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

from Example 28. The canonical VO of the query is the same as in Figure 15 (middle). Figure 15 shows on the left column the three VOs returned by the function Ω in Figure 13.

We explain the construction of the VOs returned by Ω. We start from the root A in the canonical VO. The residual query $Q_A(\mathcal{O}_A|\mathcal{I}_A)$ is equal to $Q(\mathcal{O}|\mathcal{I})$. Since Q_A is not CQAP_0 and A is bound, we distinguish two cases based on the degree of A-values: In the light case for A, we create a access-top VO for Q_A whose leaves are the light parts of the input relations partitioned on A (top left in Figure 15).

In the heavy case for A, we recursively process the subtrees of A in the canonical VO and treat A as an input variable. The residual query $Q_E(\cdot|A, E) = T(A, E)$ is CQAP_0, thus we create a access-top VO for Q_E whose leaf is $T^{A \to H}(A, E)$, i.e., the heavy part of T partitioned on A (middle left and bottom left VOs in Figure 15). The residual query $Q_B(C, D|A) = R(A, B, C), S(A, B, D)$, however, is not CQAP_0. Since B is bound, we further distinguish two new cases based on the degree of the values over (A, B). In the light case for (A, B), we construct a VO whose leaves are $R^{A \to H, AB \to L}$ and $S^{A \to H, AB \to L}$, i.e., the parts of R and S that are heavy on A and light on (A, B) (middle left VO in Figure 15). In the heavy case for (A, B), we process the subtrees of B considering B as an input variable (bottom left VO in Figure 15). The residual queries $Q_C(C|A, B) = R(A, B, C)$ and $Q_D(D|A, B) = S(A, B, D)$, are CQAP_0. Overall, we create three VOs. ◀

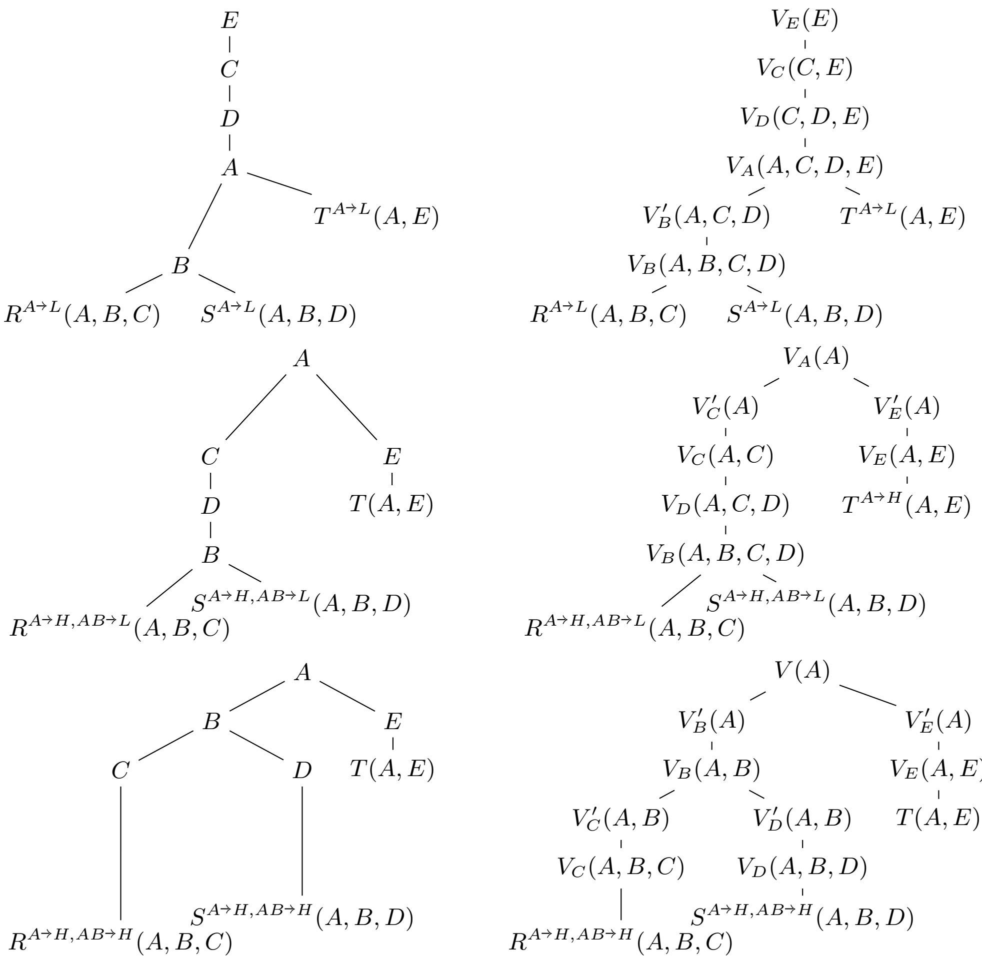


Figure 15 Left column: The VOs constructed for the query $Q(C,D \mid E) = R(A,B,C), S(A,B,D), T(A,E)$ in Example 28. Right column: The view trees constructed following the VOs on the left.

1185 ### E.3.3 View Trees Encoding the Query Result

1186 The translation from VOs for hierarchical CQAPs into view trees is the same as in our
1187 approach for arbitrary CQAPs (Section 4). Given a VO ω, the function $\tau(\omega)$ in Figure 2
1188 returns a view tree following ω. The function $\textsc{ViewTrees}(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 16 returns
1189 the set of all view trees for a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with canonical VO ω. For each VO
1190 ω' returned by $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ from Figure 13, the function creates the corresponding view tree
1191 by calling $\tau(\omega')$ from Figure 2.

1192 Materializing a view tree consists of computing the relation parts at the leaves and
1193 computing the joins defined by the views in the view tree. The preprocessing phase for a
1194 hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with canonical VO ω consists of materializing all view trees in
1195 $\textsc{ViewTrees}(\omega, (\mathcal{O}|\mathcal{I}))$.

1196 ▶ **Example 33.** Figure 14 (middle left and right) shows the view trees constructed from
1197 the corresponding VOs. Each variable in the VO is mapped to a view in the view tree, e.g.,
1198 B is mapped to $V_B(A_1,B)$, where $\{B, A_1\} = \{B\} \cup \mathit{dep}(B)$. The views V'_C, V'_D and V_{A_1}
1199 are auxiliary views that allow for efficient maintenance under updates to R and S: they

```
ViewTrees(canonical VO ω, access pattern (O|I)) : view trees
return {τ(ω′) | ω′ ∈ Ω(ω, (O|I))}
```

Figure 16 Construction of all view trees for a canonical VO ω of a hierarchical CQAP with access pattern $(\mathcal{O}|\mathcal{I})$.

marginalize out one variable from their child views. The view V_B is the intersection of V'_C and V'_D. Hence all views can be computed in linear time. ◀

▶ **Example 34.** Consider again the query

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

from Example 28. Figure 15 shows next to each VO for the query, the corresponding view tree. The query Q has static width 3. Computing the relation parts at the leaves of the view trees takes time linear in N, where N is the database size. We explain how the views in the view trees can be computed in $\mathcal{O}(N^{1+2\epsilon})$ time.

Consider the VO and view tree in the top row of Figure 15. At variable B, we create the view $V_B(A, B, C, D) = R^{A \to L}(A, B, C), S^{A \to L}(A, B, D)$, which joins the light parts of R and S partitioned on A. Computing $V_B(A, B, C, D)$ takes $\mathcal{O}(N^{1+\epsilon})$ time: For each value (a, b, c) in $R^{A \to L}$, we iterate over at most N^ϵ (a, b, d) values in $S_L^{A \to L}$. Since B has siblings in the VO, we also create the auxiliary view $V'_B(A, C, D)$ that aggregates away B in time linear in the size of V'_B. At A, we compute $V_A(A, C, D, E)$ in $\mathcal{O}(N^{1+2\epsilon})$ time: We iterate over $\mathcal{O}(N^{1+\epsilon})$ values (a, c, d) in $V'_B(A, C, D)$ and for each such value, iterate over at most N^ϵ values (a, e) in $T^{A \to L}$. We do not need to create an auxiliary view that aggregates away A, since A does not have siblings in the variable order. At each variable above A, we create a view that aggregates away the variable below. Aggregating a variable away takes time linear in the size of the view. Hence, computing $V_D(C, D, E)$ takes $\mathcal{O}(N^{1+2\epsilon})$ time, computing $V_C(C, E)$ takes $\mathcal{O}(N^{1+\epsilon})$ time, and computing $V_E(E)$ takes $\mathcal{O}(N)$ time. Overall, materializing this view tree takes $\mathcal{O}(N^{1+2\epsilon})$ time.

We now consider the VO and view tree in the second row. At B, we create the view $V_B(A, B, C, D) = R^{A \to H, AB \to L}(A, B, C),\ S^{A \to H, AB \to L}(A, B, D)$ in $\mathcal{O}(N^{1+\epsilon})$ time: For each value (a, b, c) in $R^{A \to H, AB \to L}$, we iterate over at most N^ϵ values (a, b, d) in $S^{A \to H, AB \to L}$. At E, we build $V_E(A, D, E)$ that aggregates away B in $\mathcal{O}(N^{1+\epsilon})$ time. At D, we build $V_D(A, D)$ and the auxiliary view $V'_D(A)$ in linear time. The other views can be computed in linear time by aggregating away variables and applying semi-join reduction. Hence, materializing the view tree in the second row takes $\mathcal{O}(N^{1+\epsilon})$ time.

Materializing the view tree in the bottom row takes linear time: All views are computed by aggregating away variables and applying semi-join reduction, which takes linear time.

Overall, we materialize the three view trees for Q in $\mathcal{O}(N^{1+2\epsilon})$ time. ◀

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

▶ **Proposition 35.** *Let $\{T_1, \ldots, T_k\}$ be the set of view trees in* ViewTrees$(\omega, (\mathcal{O}|\mathcal{I}))$ *for a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ and the canonical VO ω 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 an adaptation of the proof of Proposition 4.3. in [?] to CQAPs. For the sake of completeness, we give here the full proof.

The procedure ViewTrees calls Ω to construct from the input canonical VO ω a set of VOs $\omega_1, \ldots, \omega_k$ and constructs the set of view trees $T_1, \ldots, T_k$ following the VOs. The corresponding VO ω_i and view tree T_i for $i \in [k]$ have the same leaf atoms. We define $Q_{\omega'}(\mathcal{O}|\mathcal{I}) = \bowtie_{R(\mathcal{X}) \in atoms(\omega')} R(\mathcal{X})$ be the query defined by the conjunction of the leaf atoms in ω'.

The proof is by induction over the structure of the VO ω. We show that for any subtree ω' rooted at X of ω, it holds:

$$Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) \equiv \bigcup_{\omega'' \in \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))} Q_{\omega''}(\mathcal{O}_X|\mathcal{I}_X), \tag{3}$$

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

Base case: If ω' is an atom, the procedure Ω returns that atom and the base case holds trivially.

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

Case 1: $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ is CQAP$_0$. The procedure $\Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))$ constructs an access-top VO with leaves exactly the atoms of ω'. This implies Equivalence 3.

Case 1 does not hold and ($X \in \mathcal{O}$ or ($X \in \mathcal{O}$ and $vars(\omega') \cap \mathcal{I} = \emptyset$)): The procedure $\Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))$ constructs recursively a set of VOs for each subtree in $\omega'_1, \ldots, \omega'_k$ and returns a set of VOs, which are the combinations of the k sets of VOs attached to X. Using the induction hypothesis, we rewrite as follows:

$$\begin{aligned}
Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) &= \bowtie_{i \in [k]} Q_{\omega'_i}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \\
&\overset{\text{IH}}{\equiv} \bowtie_{i \in [k]} \Big(\bigcup_{\omega'' \in \Omega(\omega'_i, (\mathcal{O}_{X'}|\mathcal{I}_{X'}))} Q_{\omega''}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \Big) \\
&\equiv \bigcup_{\forall i \in [k]: \omega''_i \in \Omega(\omega'_i, (\mathcal{O}_{X'}|\mathcal{I}_{X'}))} \bowtie_{i \in [k]} Q_{\omega''_i}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \\
&= \bigcup_{T \in \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))} Q_T(\mathcal{O}_X|\mathcal{I}_X),
\end{aligned}$$

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

Cases 1 and 2 do not hold: The procedure Ω creates the VOs $htrees \cup \{ltree\}$ defined as follows:

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

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

$$\bigcup_{\forall i \in [k]: T_i \in \Omega(\omega'_i, (\mathcal{O}|\mathcal{I}))} \bowtie_{i \in [k]} Q_{T_i} \equiv Q_{ltree} \cup \bigcup_{\forall i \in [k]: T_i \in \Omega(\omega_i^{key \to H}, (\mathcal{O}|\mathcal{I}))} Q_{T_i} \tag{4}$$

Using the induction hypothesis, we obtain:

$$
\begin{aligned}
Q_{\omega'} = \bowtie_{i\in[k]} Q_{\omega'_i} &\overset{\text{IH}}{\equiv} \bowtie_{i\in[k]} \Big(\bigcup_{\omega''\in\Omega(\omega'_i,(\mathcal{O}|\mathcal{I}))} Q_{\omega''} \Big) \\
&\equiv \bigcup_{\forall i\in[k]:\omega''_i\in\Omega(\omega'_i,(\mathcal{O}|\mathcal{I}))} \bowtie_{i\in[k]} Q_{\omega''_i} \\
&\overset{(4)}{\equiv} Q_{ltree} \cup \bigcup_{\forall i\in[k]:\omega''_i\in\Omega(\omega_i^{key\to H},(\mathcal{O}|\mathcal{I}))} Q_{\omega''_i} \\
&= Q_{ltree} \cup \bigcup_{T\in htrees} Q_T = \bigcup_{T\in\Omega(\omega',(\mathcal{O}|\mathcal{I}))} Q_T
\end{aligned}
$$

◀

Given a hierarchical CQAP query $Q(\mathcal{O}|\mathcal{I})$ with static width w, the preprocessing time of our approach is given by the time to materialize the view trees in $\textsc{ViewTrees}(\omega, \mathcal{O}, \mathcal{I})$. The time to materialize these view tree is $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$.

▶ **Proposition 36.** *Given a hierarchical CQAP with static width* w*, a database of size* N*, and* $\epsilon \in [0, 1]$*, the view trees in the preprocessing stage can be computed in* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *time.*

The proof uses the auxiliary Lemma 37 given below. We first explain how Proposition 36 is implied by Lemma 37. Consider a CQAP Q with static width w and hierarchical fracture $Q_\dagger$ and an $\epsilon \in [0, 1]$. In the preprocessing stage, we apply for each connected component $Q'_\dagger(\mathcal{O}|\mathcal{I})$ of $Q_\dagger$ the following steps. Let ω be the canonical VO of $Q'_\dagger$. First, we call the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 13, which creates a set of VOs from ω. For each VO ω' in this set, we call the function $\tau(\omega')$ in Figure 2, which creates a view tree T following ω'. By Lemma 37, the view tree T can be materialised in $\mathcal{O}(N^{(\mathsf{w}(Q'_\dagger)-1)\epsilon})$ time. Since $\mathsf{w}(Q'_\dagger)$ is upper-bounded by w, this implies $\mathcal{O}(N^{(\mathsf{w}-1)\epsilon})$ overall preprocessing time.

It remains to prove Lemma 37.

▶ **Lemma 37.** *Let* ω *be a VO of a CQAP* $Q(\mathcal{O}|\mathcal{I})$*,* X *a variable in* ω*,* Q_X *the induced query at* X *in* ω*,* $\omega' \in \Omega(\omega_X, (\mathcal{O}, \mathcal{I}))$*,* $\omega^t = (\mathsf{anc}_\omega(X) \circ \omega')$*,* N *the size of the leaf relations in* ω'*, and* $\epsilon \in [0, 1]$*. The view tree* $\tau(\omega^t)$ *can be materialised in* $\mathcal{O}(N^{1+(\mathsf{w}(Q_X)-1)\epsilon})$ *time.*

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

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

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

Case (1): $Q_X(\mathcal{O}|\mathcal{I})$ *is a* $CQAP_0$ *query.*
In this case, the procedure Ω returns the VO $\omega' = \textsc{AccessTop}(\omega_X, (\mathcal{O}|\mathcal{I}))$. By Proposition 29, $\omega^t = (\mathsf{anc}_\omega(X) \circ \omega')$ is an access-top VO for Q_X with static width $\mathsf{w}(Q_X)$. Since

Q_X is in CQAP_0, its static width can be at most 1 (Proposition 25). This means that for every variable $Y \in \mathit{vars}(\omega^t)$, the set $\{Y\} \cup \mathit{dep}_{\omega^t}(Y)$ can be covered by a single atom in Q_X. Hence, each view $V_Y(\{Y\} \cup \mathit{dep}_{\omega^t}(Y))$ can be computed in $\mathcal{O}(N)$ time. This completes the inductive step for Case (1).

Case (2): Q_X is not in $CQAP_0$ and $\big(X \in \mathcal{I}$ or $(X \in \mathcal{O}$ and $\mathit{vars}(\omega) \cap \mathcal{I} = \emptyset)\big)$
The set of VOs returned by Ω is defined as follows: For each set $\{\omega_i\}_{i\in[k]}$ with $\omega_i \in \Omega(\omega_{X_i}, (\mathcal{O}|\mathcal{I}))$, the set contains a VO ω' with root node X and child trees $\omega_1, \ldots, \omega_k$. Consider for one such VO ω' the VO $\omega^t = (\mathsf{anc}_\omega(X) \circ \omega')$. By induction hypothesis, each view tree over ω_i can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}(Q_{X_i})-1)\epsilon})$ time. Since $\mathsf{w}(Q_{X_i}) \leq \mathsf{w}(Q_X)$ for any $i \in [k]$, it follows that each view tree over ω_i can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}(Q_X)-1)\epsilon})$ time. Consider now the view tree $\tau(\omega^t)$. The view at X is defined by $V_X(\mathcal{S}) = V_{X_1}(\mathcal{S}_1), \ldots, V_{X_k}(\mathcal{S}_k)$, where $\mathcal{S} = \{X\} \cup \mathit{dep}_\omega(X)$ 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_i} that are not included in $\mathcal{S}$ for each $i \in [k]$ and then computing the intersection of the remaining relations. This gives overall $\mathcal{O}(N^{1+(\mathsf{w}(Q_X)-1)\epsilon})$ 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 Ω constructs a set *htrees* of VOs and a single variable order *ltree*. The construction of the VOs in *htrees* differs from the VOs 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 VOs *htrees* can be computed in $\mathcal{O}(N^{1+(\mathsf{w}(Q_X)-1)\epsilon})$ time. The VO *ltree* is defined as $\mathit{ltree} = \textsc{AccessTop}(\omega_X^{key\to L}, (\mathcal{O}|\mathcal{I}))$, where $\omega_X^{key\to L}$ 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 29, *ltree* is an access-top VO for Q_X with optimal static width. Then, it follows from Lemma 38 (given below) that the view tree $\tau(\mathit{ltree})$ can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}(Q_X)-1)\epsilon})$ time. This completes the inductive step for *Case 3*. ◀

The next lemma gives the time to materialise view trees referring to light relation parts.

▶ **Lemma 38.** *Let ω be a VO, X a variable in ω such that $\mathsf{anc}_\omega(X)$ is included in the schemas of all leaf atoms in ω_X and $\omega^t = (\mathsf{anc}_\omega \circ \omega_X)$. If the leaf relations in ω_X are the light parts of a partition on $\{X\} \cup \mathsf{anc}_\omega(X)$ with threshold $\mathcal{O}(N^\epsilon)$ for some $\epsilon \in [0,1]$, the view tree $\tau(\omega^t)$ can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}(\omega^t)-1)\epsilon})$ time.*

Proof. Let $T = \tau(\omega^t)$ and $\mathsf{w} = \mathsf{w}(\omega^t)$. We show that every view in T can be computed in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon\}})$ time. The leaf atoms can obviously be materialised in $\mathcal{O}(N)$ time.

Consider any view $V_Y(\mathcal{S})$ in T with $\mathit{atoms}(\omega_Y^t) = \{R_i(\mathcal{X}_i)\}_{i\in[k]}$. The view V_Y is defined over the join of its child views and it holds $\mathcal{S} = \{Y\} \cup \mathit{dep}_\omega(Y)$. By the construction of our view trees, V_Y can be computed by joining the atoms $R_1(\mathcal{X}_1), \ldots, R_k(\mathcal{X}_k)$. Hence, we can write the view as

$$V_Y(\mathcal{S}) = R_1(\mathcal{X}_1), \ldots, R_k(\mathcal{X}_k).$$

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

$$V_Y(\mathcal{S}) = V(\mathcal{S}), R'_1(\mathcal{X}'_1), \ldots, R'_\ell(\mathcal{X}'_\ell), \tag{5}$$

where $R'_1(\mathcal{X}'_1)$, $\ldots$, $R'_\ell(\mathcal{X}'_\ell)$ are the atoms in $\mathcal{R}_0$. The query (5) is free-connex α-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 [**?**]. The input size is upper-bounded by $|V| = \mathcal{O}(N^{1+(m-1)\epsilon})$. The size of the output is also $\mathcal{O}(N^{1+(m-1)\epsilon})$. Hence, the overall time to compute V_Y is $\mathcal{O}(N^{1+(m-1)\epsilon})$. Since $m = \rho^*_{Q_Y}(\mathcal{S})$ is upper-bounded by w, we derive that the computation time for V_Y is $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. Each of the additional auxiliary views constructed in Line 8 of the procedure τ is obtained by marginalising away a variable from its child view. This does not blow up the overall asymptotic computation time. ◀

E.4 Enumeration

In the preprocessing stage, we construct view trees that represent the result of the query. In this section, we show how to enumerate from these view trees the distinct output tuples together with their multiplicity given a tuple of values over the input variables. The enumeration relies on iterators with access patterns created over materialized views. In this section, we first discuss the enumeration for CQAP_0 queries and then the enumeration for hierarchical CQAP queries in general.

E.4.1 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 $\mathtt{it}_V(O|\mathcal{I})$ to denote a view iterator $\mathtt{it}$ over a view V with schema $\{O\} \cup \mathcal{I}$, where O is the *output variable* and $\mathcal{I}$ 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 $\mathtt{it}_V(O|\mathcal{I}).\mathit{contains}(o)$ to check if the given value o can appear in the output of the $\mathtt{it}_V$ iterator; this is syntactic sugar for the membership test $\mathit{ctx} \circ (o) \in V$, where $\circ$ denotes tuple concatenation. All the three methods, *open*, *next*, and *contains*, take constant time as per the computational model from Section 2.

▶ **Example 39.** Consider a materialized view $V(A, B)$. The iterator $\mathtt{it}_V(B|A)$ enumerates the distinct B-values paired with a given A-value in V. The iterator $\mathtt{it}_V(B|A, B)$ returns the B-value in a given (A, B)-tuple if the tuple exists in V; otherwise, it returns **EOF**. The iterator $\mathtt{it}_V(A)$ is invalid as its output variable A and context schema $\emptyset$ do not match the schema of V, i.e., $\{A\} \cup \emptyset \neq \{A, B\}$. ◀

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 VO 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.

▶ **Example 40.** Figure 17 shows the enumeration procedure for the view tree from Figure 14 (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

let ctx_0 = input A_1-value
$\texttt{it}_{V_{A_1}}(A_1|A_1).open(ctx_0)$
while $(a := \texttt{it}_{V_{A_1}}(A_1|A_1).next()) \neq$ **EOF do**
  $\texttt{it}_{V_B}(B|A_1).open(a)$
  while $(b := \texttt{it}_{V_B}(B|A_1).next()) \neq$ **EOF do**
    $\texttt{it}_{V_C}(C|A_1, B).open(a, b)$
    while $(c := \texttt{it}_{V_C}(C|A_1, B).next()) \neq$ **EOF do**
      $\texttt{it}_{V_D}(D|A_1, B).open(a, b)$
      while $(d := \texttt{it}_{V_D}(D|A_1, B).next()) \neq$ **EOF do**
        output (b, c, d)
output EOF

Figure 17 Enumeration for $Q(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ using the second from left view tree from Figure 14.

iterators for this view tree top-down. At the root view V_A, we create $\texttt{it}_{V_{A_1}}(A_1|A_1)$ 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 $\texttt{it}_{V_B}(B|A_1)$ 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 $\texttt{it}_{V_C}(C|A_1B)$ and $\texttt{it}_{V_D}(D|A_1B)$, 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 $\mathcal{O}(1)$; this is also the enumeration delay of the procedure. ◀

Nesting view iterators, as in Figure 17, 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 the view iterators is not always unique; e.g., we can swap the two innermost loops in the procedure from Figure 17.

For any query in CQAP_0, the corresponding view trees follow access-top VOs 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 VOs is valid and allows constant delay enumeration.

For queries not in CQAP_0, nesting view iterators may be invalid. Assume for instance that the variable A_1 is bound in the query from Example 40. The query remains hierarchical but not free-dominant. The view iterators that enumerate B-, C-, and D-values have A_1 in their context schemas, yet there is no iterator for A_1-values. We say that such iterators are unsupported.

E.4.2 Generalised View Iterators

To support the enumeration for non-CQAP_0 queries, we generalise the above view iterators as follows. The context of a generalised view iterator $\texttt{git}_V(O|\mathcal{I})$ is a *relation* (instead of a tuple) over schema $\mathcal{I}$. The $open(ctx)$ method takes as input a relation ctx over $\mathcal{I}$ and instantiates a view iterator for each tuple in ctx. The $next()$ method uses the union algorithm [**?**] to report only distinct O-values, with the delay linear in the size of ctx. For each reported O-value o,

```
git_V(O|I).open(relation ctx)
---------------------------------------------------------------
git_V(O|I).iterators := empty map        // tuple ↦ view iterator
foreach t ∈ ctx do
    git_V(O|I).iterators[t] := new it_V(O|I)
    git_V(O|I).iterators[t].open(t)
```

Figure 18 Open the generalised view iterator $\texttt{git}_V(O|\mathcal{I})$ with the relation *ctx* over schema $\mathcal{I}$ as context.

```
UNION(iterators it_1, ..., it_n): value
---------------------------------------------------------------
if (n = 1)
    return it_n.next()
if (v_[n-1] := UNION(it_1, ..., it_{n-1})) ≠ EOF
    if it_n.contains(v_[n-1])
        return it_n.next()
    return v_[n-1]
if (v_n := it_n.next()) ≠ EOF
    return v_n
return EOF
```

Figure 19 Fetch the next distinct value from a list of iterators.

next() also returns a relation $ctx_o \subseteq ctx$ over schema $\mathcal{I}$ with the tuples that are paired with o in V. If there are no such tuples in V, the method returns $(\mathbf{EOF}, \emptyset)$.

Figures 18 shows the *open*(*ctx*) method, which takes as input a relation *ctx* over $\mathcal{I}$ 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, $\mathcal{O}(|ctx|)$.

The *next*() method uses the UNION algorithm from Figure 19 to fetch the next distinct output value from a list of iterators. The algorithm is an adaptation of prior work [**?**]. 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 $\texttt{it}_1$ and $\texttt{it}_2$. Given the next value v_1 of $\texttt{it}_1$, the algorithm calls $\texttt{it}_2.contains(v_1)$ to check if v_1 can be enumerated by $\texttt{it}_2$. If so, it returns the next value in $\texttt{it}_2$; otherwise, it returns v_1. If $\texttt{it}_1$ is exhausted, the function returns the next value in $\texttt{it}_2$ or **EOF** if $\texttt{it}_2$ 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 $\texttt{it}_n$ as the second iterator, and then reduces the general case to the previous case of two iterators. The algorithm invokes *next*() and checks for membership

$\texttt{git}_V(O|\mathcal{I}).next()$: (value, relation)

let $\{\, t_1 \rightarrow \texttt{it}_1, \ldots, t_n \rightarrow \texttt{it}_n \,\} = \texttt{git}_V(O|\mathcal{I}).iterators$
$o := \textsc{Union}(\texttt{it}_1, \ldots, \texttt{it}_n)$
$ctx_o := \{\, t_i \mid i \in [n],\ \texttt{it}_i.contains(o) \,\}$
return (o, ctx_o)

Figure 20 Fetch the next output value from the generalised view iterator $\texttt{git}_V(O|\mathcal{I})$ together with the set of tuples over schema $\mathcal{I}$ that are paired with that output value in V.

on n iterators, each taking constant time. Thus, fetching the next value takes $\mathcal{O}(n)$ time.

Figure 20 shows the $next()$ method. For each output value o obtained using the Union algorithm, $next()$ computes a set of tuples over schema $\mathcal{I}$ that are paired with o in V. Assuming $\texttt{git}_V(O|\mathcal{I})$ is opened for a relation ctx, fetching the output value o and computing the set of tuples for o each take $\mathcal{O}(|ctx|)$ time. Thus, $next()$ also runs in $\mathcal{O}(|ctx|)$ time.

▶ **Example 41.** Figure 21 shows the enumeration procedure for the view tree from Figure 6 (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 $\texttt{git}_{V_{A_1}}(A_1|A_1)$ serves to check if the given A_1-value exists in V_{A_1} (Lines 2-3). The iterator $\texttt{git}_{V_C}(C|A_1, B, C)$ 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 $\texttt{git}_{V_C}$ 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 $(\mathbf{EOF}, \emptyset)$. 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 $\texttt{git}_{V_{A_1}}$ is constant, while for $\texttt{git}_{V_C}$ and $\texttt{git}_{V_D}$ is at most the size of V_B. Given that V_B is over the heavy part $R^{A_1 B \rightarrow H}$ of R and the heavy part $S^{A_1 B \rightarrow H}$ of S, the number of distinct (A_1, B)-values in V_B is at most $N^{1-\epsilon}$. Thus, the enumeration delay is $\mathcal{O}(N^{1-\epsilon})$. ◀

E.4.3 Enumeration Procedure

The function BuildIterators from Figure 22 builds a list of generalised view iterators for a given view tree of a CQAP Q with access pattern $(\mathcal{O}|\mathcal{I})$. 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).

The function recursively constructs generalised view iterators, traversing the view tree T in a top-down fashion. Consider the root view $V_X(\mathcal{X})$ of T constructed at variable X in the corresponding VO. If $X \notin \mathcal{X}$, then V_X is an auxiliary view that allows for efficient maintenance under updates (c.f. Figure 2) 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

let $ctx_0(A_1, C) = \{(a_0, c_0)\}$, where a_0, c_0 are input values
$\texttt{git}_{V_{A_1}}(A_1|A_1).open(\pi_{A_1}(ctx_0))$
while $((a, ctx_a) := \texttt{git}_{V_{A_1}}(A_1|A_1).next()) \neq (\textbf{EOF}, \emptyset)$ **do**
  $\texttt{git}_{V_C}(C|A_1, B, C).open(V_B(A_1, B) \bowtie ctx_0)$
  while $((c, ctx_c) := \texttt{git}_{V_C}(C|A_1, B, C).next()) \neq (\textbf{EOF}, \emptyset)$ **do**
    $\texttt{git}_{V_D}(D|A_1, B).open(\pi_{A_1 B}(ctx_c))$
    while $((d, ctx_d) := \texttt{git}_{V_D}(D|A_1, B).next()) \neq (\textbf{EOF}, \emptyset)$ **do**
      output (d)
output EOF

Figure 21 Enumeration for $Q(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$ using the bottom-right view tree from Figure 6.

BuildIterators(view tree T, access pattern $(\mathcal{O}|\mathcal{I})$, relation *supp*)

switch T:

$R(\mathcal{Y})$
return []

$V_X(\mathcal{X})$ / \ $T_1 \cdots T_k$
if $X \notin \mathcal{X}$ // skip auxiliary maintenance views
  return BuildIterators$(T_1, (\mathcal{O}|\mathcal{I}), supp)$

$$it_X = \begin{cases} [\,(\textbf{new}\,\texttt{git}_{V_X}(X|\mathcal{X}),\, supp)\,] & \text{, if } X \in \mathcal{I} \\ [\,(\textbf{new}\,\texttt{git}_{V_X}(X|\mathcal{X} \setminus \{X\}),\, supp)\,] & \text{, if } X \in \mathcal{O} \\ [\,] & \text{, otherwise} \end{cases}$$

$$supp_{child} = \begin{cases} supp & \text{, if } X \in (\mathcal{I} \cup \mathcal{O}) \\ V_X(\mathcal{X}) & \text{, otherwise} \end{cases}$$

$it_{child_i} = \text{BuildIterators}(T_i, (\mathcal{O}|\mathcal{I}), supp_{child}), \forall i \in [k]$
return $it_X \mathbin{+\!\!+} it_{child_1} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} it_{child_k}$

Figure 22 Create a list of generalised view iterators with support for the access pattern $(\mathcal{O}|\mathcal{I})$ in a view tree T. The first call to BuildIterators uses the support $\{()\}$.

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.

Once we construct the iterators over the view tree, we generate the enumeration procedure by organizing 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. Each concatenation of the outputs of the iterators forms the values of an output tuple.

The time for an iterator to report an output tuple, i.e., the *next* method of the iterator, is determined by the size of its input context relation. That is, the size of the support relations. Hence, the enumeration delay of the procedure is upper-bounded by the size of the support relations.

▶ **Example 42.** Consider the view tree from Figure 14 (second from left) for the connected component $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$. BuildIterators returns the following union view iterators for this view tree: $\texttt{git}_{V_{A_1}}(A_1|A_1)$, $\texttt{git}_{V_B}(B|A_1)$, $\texttt{git}_{V_C}(C|A_1, B)$,

1 **let** $ctx_0 = \{e_0\}$ // where e_0 is the input E-value
2 $\texttt{git}_{V_E}(E|A,E).open(V_A(A) \bowtie ctx_0)$
3 **while** $((e, ctx_e) := \texttt{git}_{V_E}(E|A,E).next()) \neq$ **EOF do**
4   $\texttt{git}_{V_C}(C|A).open(ctx_e)$
5   **while** $((c, ctx_c) := \texttt{git}_{V_C}(C|A).next()) \neq$ **EOF do**
6     $\texttt{git}_{V_D}(D|A,C).open(ctx_c \bowtie \{c\})$
7   **while** $((d, ctx_d) := \texttt{git}_{V_D}(D|A,C).next()) \neq$ **EOF do**
8   **let** $m = \sum_{a \in \pi_A ctx_d} V_D(a,c,d) \cdot V_C(a,c) \cdot V_E(a,e)$
9   **output** $(c,d) \mapsto m$
10 **output EOF**

Figure 23 Enumeration procedure for the connected component $Q(C,D|E) = R^{A\to H, AB\to L}(A,B,C), S^{B\to H, AB\to L}(A,B,D), T^{A\to H}(A,E)$.

and $\texttt{git}_{V_D}(D|A_1,B)$, each paired with the support $\{()\}$. Figure 17 shows the enumeration procedure for these iterators. The multiplicity of the output tuple (b,c,d) for the input A_1-value a_1 is the product of the values in the base relations: $R(a_1,b,c) \cdot S(a_1,b,d)$. The enumeration delay is constant. ◀

▶ **Example 43.** Consider now the view tree from Figure 15 (left in the second row), created for $Q(C,D|E) = R^{A\to H, AB\to L}(A,B,C), S^{B\to H, AB\to L}(A,B,D), T^{A\to H}(A,E)$. BuildIterators returns the following iterators for this view tree:

- $\texttt{git}_{V_E}(E|A,E)$ with the support $V_A(A)$,
- $\texttt{git}_{V_C}(C|A)$ with the support $V_A(A)$, and
- $\texttt{git}_{V_D}(D|A,C)$ with the support $V_A(A)$.

Figure 23 shows the enumeration procedure for these iterators. The returned support relations define the context to be used when opening each union view iterator. As discussed in the next section, to compute the multiplicity of the output tuple (c,d) for the input E-value e_0, we sum over the multiplicities of the tuple concatenated with the A-values in the context relation ctx_d (Line 9).

◀

Multiplicity Computation. Once we get an output tuple from the enumeration procedure as shown above, we need to compute the multiplicity of the tuple in the view tree. Figure 24 shows the ComputeM function for computing the multiplicity of a tuple $\mathbf{t}$ in a view tree T. The parameter $context_{\mathbf{t}}$ contains the set of context relations returned by the *next* method of the union view iterators for the tuple $\mathbf{t}$, such as the relations ctx_e, ctx_c and ctx_d in Example 43.

The function traverses the view tree T based on a pre-order. At the root view $V(\mathcal{X})$ of T, there are three cases: (1) the view V has a variable A_1 that is not in the schema of the tuple $\mathbf{t}$ (Line 1). This corresponds to the case when A_1 is bound and has been aggregated away from the views below V in the view tree. In this case, we treat A_1 as if it is free, and sum over all the multiplicities of the concatenations of $\mathbf{t}$ and the A_1-values paired with $\mathbf{t}$ in the view tree: For each such A_1-value from the context set (Lines 2-3), the function concatenates the value to $\mathbf{t}$, and applies ComputeM to compute the multiplicity of the new tuple. The multiplicity of $\mathbf{t}$ is the sum of the multiplicities of these new tuples (Line 4). (2) The second case is the opposite of the first case: the schema of $\mathbf{t}$ has additional variables that are not in

ComputeM(view tree T, tuple $\mathbf{t}$, context relations $contexts_{\mathbf{t}}$): multiplicity

switch T:

$V_{\mathcal{X}}(\mathcal{X})$

$T_1 \cdots T_k$

1 **if** $\mathsf{Sch}(\mathbf{t}) \subsetneq \mathcal{X}$
2   **let** $\{A_1, \ldots, A_k\} = \mathcal{X} \setminus \mathsf{Sch}(\mathbf{t})$
3   $\mathcal{A}_1 := \pi_{A_1}(\bowtie_{ctx \in contexts_{\mathbf{t}}} ctx)$ // A_1-values that satisfy all context relations
4   **return** $\sum_{a \in \mathcal{A}_1}$ ComputeM$(T, \mathbf{t} \circ a, contexts_{\mathbf{t}} \cup \{\{a\}\})$
5 **else if** $\mathcal{X} \subsetneq \mathsf{Sch}(\mathbf{t})$
6   $\mathcal{V}_i :=$ variables in T_i
7   $contexts_i := \{\pi_{\mathcal{V}_i} R \mid R \in contexts_{\mathbf{t}}\}$
8   **return** $\prod_{i \in [k]}$ ComputeM$(T_i, \pi_{\mathcal{V}_i}\mathbf{t}, contexts_i)$
9 **else**   // $\mathcal{X} = \mathsf{Sch}(\mathbf{t})$
10   **return** $V[\mathbf{t}]$

Figure 24 Compute the multiplicity of the given tuple $\mathbf{t}$ in the view tree T. The input $contexts_{\mathbf{t}}$ contains all the context sets returned during the enumeration of $\mathbf{t}$.

the schema of V (Line 5). This means the tuple $\mathbf{t}$ is stored below V, possibly distributed in different branches. The function applies ComputeM recursively to each subtree and takes the product of the returned multiplicities (Lines 6-8). (3) When $\mathbf{t}$ is in V (Line 9), the function returns the multiplicity of $\mathbf{t}$ in V (Line 10).

The computation time of the multiplicity of a tuple $\mathbf{t}$ is upper-bounded by the time for enumerating $\mathbf{t}$ using the iterators. The time of the function ComputeM is determined by the number of multiplicities to be summed in the first case. That is, the size of the context relations. Since these context relations are all subsets of the support relations (as per the *next* method of union view iterators), their sizes are upper-bounded by the sizes of the support relations. Hence, ComputeM does not take time more than the time for the enumerating the tuple $\mathbf{t}$ using the iterators.

Enumeration from multiple connected components. We discussed how to enumerate tuples from one view tree. In case of queries with several connected components, we form a nesting chain for the enumeration from their view trees. To enumerate from view trees for different evaluation strategies, we use the union algorithm [?] and view tree iterators, as in prior work [?].

The enumeration for a query $Q(\mathcal{O}|\mathcal{I})$ is the enumeration for its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}')$: Given any tuple $\mathbf{t}$ over $\mathcal{I}$, let $\mathbf{t}'$ be the tuple over $\mathcal{I}'$ such that $\mathbf{t}[A] = \mathbf{t}'[A']$ for all fresh variables A' in $\mathcal{I}'$ that replace A in $\mathcal{I}$. Then the sets $Q(\mathcal{O}|\mathbf{t})$ and $Q_{\dagger}(\mathcal{O}|\mathbf{t}')$ are equal.

▶ **Proposition 44.** *For any CQAP$_0$ query, its distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(1)$ delay.*

Proof. We want to show that for any CQAP$_0$ query, its distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(1)$ delay.

The fracture of any CQAP$_0$ query with access pattern $(\mathcal{O}|\mathcal{I})$ is hierarchical, $(\mathcal{O} \cup \mathcal{I})$-dominant, and $\mathcal{I}$-dominant, per Definition 1. For each connected component of the fracture, we can construct a VO where the free variables are above the bound variables and the input variables are above the output variables, see the AccessTop function from Figure 9. For the view tree constructed following that VO, 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. ◀

▶ **Proposition 45.** *For any hierarchical CQAP Q, database of size N, and $\epsilon \in [0,1]$, the distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay.*

Proof. We give a sketch of the proof. Consider a CQAP Q with hierarchical fractures. If Q is in CQAP_0, the distinct output tuples can be enumerated with $\mathcal{O}(1)$ delay, per Proposition 44. 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 VO 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 the Ω function from Figure 13. In the light case, the created view trees follow access-top VOs, thus admitting constant delay enumeration of the output tuples for a given input tuple. In the heavy case, the view defined at X consists of at most $N^{1-\epsilon}$ heavy values, which define the support for the enumeration from child views. Using generalised view iterators, the time needed to fetch the next output tuple is linear in the size of the support used when opening those iterators. Hence, the overall enumeration delay is $\mathcal{O}(N^{1-\epsilon})$. ◀

E.5 Updates

We present our strategy for maintaining the views in the view trees returned by the function $\textsc{ViewTrees}(\omega, (\mathcal{O}|\mathcal{I}))$ (Figure 16) for a canonical VO ω of a hierarchical CQAP $Q((\mathcal{O}|\mathcal{I}))$ under updates to base relations. We write $\delta R = \{\mathbf{x} \to m\}$ to denote a single-tuple update to a base relation R mapping the tuple $\mathbf{x}$ 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[2].

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 $\mathbf{x}$ 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 δR 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 [**?**].

In Section E.5.1, 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 E.5.2 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. In Section E.5.3, we describe how to intertwine a sequence of single-tuple updates with rebalancing steps.

[2] We focus here on updates to queries without repeating relation symbols. In case a relation R occurs several times in a query, we represent an update to R as a sequence of updates to each occurrence of R.

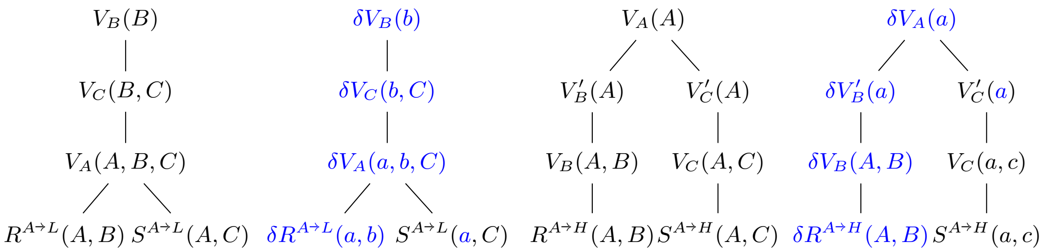


Figure 25 First and third from left: The view trees constructed for $Q(B,C) = R(A,B), S(A,C)$; The base relations are partitioned on the key A. Second and fourth from left: The delta view trees under a single-tuple update to R.

TransientHLs(tuple $\mathbf{x}$) : HL-signature

1 **let** $\{k_1, ..., k_n\} = \{k \mid k \in \text{PartitionKeys}, k \subseteq \mathsf{Sch}(\mathbf{x})\}$
2 **let** $\mathcal{K}$ = parts of base relations
3 **let** $s_i = \begin{cases} sig[k_i], & \text{if } \exists K^{sig} \in \mathcal{K} \text{ s.t. } \mathbf{x}[k_i] \in \pi_{k_i} K^{sig} \\ L, & \text{otherwise} \end{cases}$ for $i \in [n]$
4 **return** RemoveHeavyTail($\{k_1 \to s_1, \ldots, k_n \to s_n\}$)

Figure 26 Computing an HL-signature for tuple $\mathbf{x}$ by checking in which relation parts the values in $\mathbf{x}$ are contained. PartitionKeys consists of the set of all keys the base relations are partitioned on. $sig[k]$ returns the symbol the key k is mapped to in the HL-signature sig.

E.5.1 Determining the Relation Part of a Tuple

Given an update $\delta R = \{\mathbf{x} \to m\}$, 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.

▶ **Example 46.** Consider the query $Q(B,C) = R(A,B), S(A,C)$. Figure 25 (first and third from left) shows the view trees constructed for the query in the preprocessing stage; the base relations are partitioned on the key A. Let $\delta R = \{(a,b) \to m\}$ an update to the base relation R. We need to compute the HL-signature of the A-value a to find out which part of relation R is affected. If a exists in $R^{A\to L}$ or does not exist in the database, a is light on the partition key A and thus affects the part $R^{A\to L}$; otherwise, i.e., a is in $R^{A\to H}$, a is heavy and thus affects $R^{A\to H}$. ◀

The function TransientHLs($\mathbf{x}$) in Figure 26 constructs an HL-signature by checking in which relation parts the values in $\mathbf{x}$ are contained. The set PartitionKeys (in Line 1) consists of all keys on which the input relations are partitioned. In case of a triangle query, PartitionKeys consists of variables A, B and C. 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 PartitionKeys and is a subset of the schema of $\mathbf{x}$ (Line 1). If there exists a relation part K^{sig} such that $\mathbf{x}[k_i]$ is included in the projection of K^{sig} onto k_i, s_i is defined as the symbol the key k_i is mapped to in sig (first case in Line 3). Otherwise, $\mathbf{x}[k_i]$ 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, we apply the function RemoveHeavyTail (defined in Figure 27) to remove from sig all

RemoveHeavyTail(HL-signature *sig*) : HL-signature

1 **let** $\{k_1 \to s_1, ..., k_n \to s_n\} = sig$
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     $heavyTail = heavyTail \cup \{k_i \to s_i\}$
6 **return** $sig \setminus heavyTail$

Figure 27 Deletion of the heavy tail from an HL-signature *sig*. If $k \to L$ and $k' \to H$ are included in *sig* and k is a proper subset of k', then $k' \to H$ is deleted from *sig*.

ActualHLs(tuple $\mathbf{x}$, threshold θ) : HL-signature

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

Figure 28 Computing a HL-signature for tuple $\mathbf{x}$ by checking the degrees of the values in $\mathbf{x}$ based on the threshold θ.

pairs $k \to s$ such that there is $k' \to L$ in *sig* with $k' \subset k$ (Line 5). We call the HL-signature constructed by $\textsc{TransientHLs}(\mathbf{x})$ the transient HL-signature of $\mathbf{x}$.

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 $\mathbf{x}$ and a threshold θ, the function $\textsc{ActualHLs}(\mathbf{x}, \theta)$ in Figure 28 computes an HL-signature *sig* based on θ. If the degree of the projection of $\mathbf{x}$ onto a partition key is below θ in all input relations, *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 $\textsc{ActualHLs}(\mathbf{x}, \theta)$ is called the transient HL-signature of $\mathbf{x}$ based on θ.

E.5.2 Processing a Single-Tuple Update

Given a set $\mathcal{T}$ of view trees and an update $\delta R = \{\mathbf{x} \to m\}$, the procedure $\textsc{UpdateTrees}(\mathcal{T}, \delta R)$ in Figure 29 maintains the view trees under the update. It first computes the transient HL-signature *sig* of $\mathbf{x}$ (Line 2). Then, it applies $\delta R^{sig} = \{\mathbf{x} \to m\}$ to the view trees in $\mathcal{T}$ (Line 2). There might be several view trees constructed in our preprocessing stage that refer to R^{sig}.

The function $\textsc{Apply}(T, \delta R^{sig})$ in Figure 30 propagates the update δR^{sig} in the view tree T from the leaf R^{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 [**?**]. If T does not refer to R^{sig}, the procedure has no effect.

▶ **Example 47.** Figure 25 (second and fourth from left) shows the delta view trees for the corresponding view trees under the single-tuple update $\delta R = \{(a, b) \mapsto m\}$ 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 δR from the affected relation part

UPDATETREES(view trees $\mathcal{T}$, update δR)

let $\delta R = \{\mathbf{x} \to m\}$
let $sig = $ TRANSIENTHLS($\mathbf{x}$)
foreach $T \in \mathcal{T}$ **do** APPLY($T, \delta R^{sig} = \{\mathbf{x} \to m\}$)

Figure 29 Updating a set $\mathcal{T}$ of view trees for a single-tuple update $\delta R = \{\mathbf{x} \to m\}$ to relation R. If $\mathbf{x}$ is already included in a part of R, all view trees referring to that part are updated. Otherwise, the HL-signature sig of $\mathbf{x}$ is computed and all view trees referring to R^{sig} are updated.

APPLY(view tree T, update δR^{sig}) : delta view

switch T:

$K^{sig'}(\mathcal{X})$

if $K^{sig'} = R^{sig}$
  $R^{sig}(\mathcal{X}) = R^{sig}(\mathcal{X}) + \delta R^{sig}(\mathcal{X})$
  return δR
return $\emptyset$

$V(\mathcal{X})$ with children $T_1 \cdots T_k$

let $V_i(\mathcal{X}_i) =$ root of T_i, for $i \in [k]$
if $\exists j \in [k]$ s.t. $R^{sig} \in T_j$
  $\delta V_j = $ APPLY($T_j, \delta R^{sig}$)
  $\delta V(\mathcal{X}) =$ join of $V_1(\mathcal{X}_1), \ldots, \delta V_j(\mathcal{X}_j), \ldots, V_k(\mathcal{X}_k)$
  $V(\mathcal{X}) = V(\mathcal{X}) + \delta V(\mathcal{X})$
  return δV
return $\emptyset$

Figure 30 Updating views in a view tree T for a single-tuple update δR^{sig} to relation part R^{sig}. If R^{sig} is a leaf of T, the function updates R^{sig} and its ancestor views in a bottom-up fashion and returns the change of the root view. Otherwise, the empty set is returned.

to the root view. The update δR affects the light part $R^{A\to L}(A, B)$ of R if the tuple a, b is light on the partition key A. In this case, we update the relation part $R^{A\to L}(A, B)$ with $\delta R^{A\to A}(a, b) = \delta R(a, b)$, and propagate the change up the tree. We update $V_A(A, B, C)$ with $\delta V_A(a, b, C) = \delta R^{A\to L}(a, b), S^{A\to L}(a, C)$ in $\mathcal{O}(N^{\epsilon})$ time, since there are at most N^{ϵ} C-values paired with value a in $S^{A\to L}$. We then update $V_C(B, C)$ with $\delta V_C(b, C) = \delta V_A(a, b, C)$ in $\mathcal{O}(N^{\epsilon})$ time, and similarly for the view $V_B(B)$ with $\delta V_B(b) = \delta V_C(b, C)$ in $\mathcal{O}(1)$ time.

In case δR affects the heavy part $R^{A\to H}(A, B)$, i.e., (a, b) is heavy on A, we update $V_B(A, B)$ with $\delta V_B(a, b) = \delta R^{A\to H}(a, b)$ in $\mathcal{O}(1)$ time and then update the other views $V'_B(A)$ and V_A similarly in $\mathcal{O}(1)$ time.

Overall, maintaining the two view trees under a single-tuple update to any relation takes $\mathcal{O}(N^{\epsilon})$ time. ◀

▶ **Example 48.** Figure 31 shows the delta view trees for the middle right view tree in Figure 15 under the single-tuple update $\delta R = \{(a, b, c) \to m\}$ to R, $\delta S = \{(a, b, d) \to m\}$ to S, and $\delta T = \{(a, e) \to m\}$ to T.

For the delta view tree for the update δR, we update the view $V_B(A, B, C, D)$ with $\delta V_B(a, b, c, D) = \delta R^{A\to H, AB\to L}(a, b, c), S^{A\to H, AB\to L}(a, b, D)$ in $\mathcal{O}(N^{\epsilon})$ time. We then update $V_D(A, C, D)$ with $\delta V_D(a, c, D) = \delta V_B(a, b, c, D)$ with constant time and similarly for the

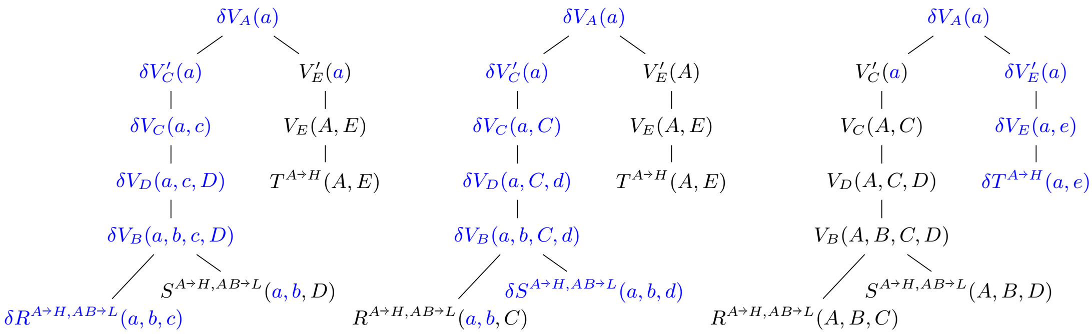


Figure 31 The delta view trees for the middle right view tree in Figure 15 under a single-tuple update to R, S, and T, respectively.

views $V_C(A,C)$, $V'_C(A)$ and $V_A(A)$. The computation of the delta view tree for the update δS is similar. For the update δT, we update the view $V_E(A,E)$ with $\delta V_E(a,e) = \delta T^{A \to H}(a,e)$ with constant time and similarly for the views $V'_E(A)$ and $V_A(A)$.

Overall, maintaining the view trees under a single-tuple update to any relation takes $\mathcal{O}(N^{\epsilon})$ time. ◀

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

▶ **Proposition 49.** *Given a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with dynamic width δ, 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 $\mathcal{O}(N^{\delta\epsilon})$ time.*

Proof. In the preprocessing stage, for a CQAP Q with input variables $\mathcal{I}$, output variables $\mathcal{O}$, canonical VO ω and delta width δ, we construct VOs $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ and then construct view trees following these VOs using the procedure τ. The procedure Ω traverses the VO ω in a top-down manner. Consider any subtree ω' of ω rooted at X and the residual query Q_X at X in ω. The procedure Ω distinguishes different cases.

In case the residual query Q_X is in CQAP$_0$, Ω creates an access-top VO ω'_{at} for ω'. At each node X of ω'_{at}, τ creates a view V_X with schema $\{X\} \cup dep_{\omega'_{at}}(X)$ that joins the child views below. By construction, if X has only one child Y in ω'_{at}, the child view V_Y created at Y below V_X has the schema $\{X,Y\} \cup dep_{\omega'_{at}}(X)$ and V_X is computed by variable marginalisation, otherwise, i.e., V_X has multiple child views, these child views have the same schema $\{X\} \cup dep_{\omega'_{at}}(X)$ as V_X. Consider an update δR to a relation R. The update δR 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 Ω procedure traverses to the subtrees of ω' and attaches the constructed VOs to X. The τ procedure creates a view V_X at X with the schema $\{X\} \cup dep_{\omega'}(X)$ that joins the child views. By construction, the schema $\{X\} \cup dep_{\omega'}(X)$ is covered by the any atom of ω', and same as discussed above, if X has only one child Y in ω'_{at}, the child view V_Y created at Y below V_X has the schema $\{X,Y\} \cup dep_{\omega'_{at}}(X)$ and V_X is computed by variable marginalisation, otherwise, i.e., V_X

MajorRebalancing(view trees $\mathcal{T}$, threshold θ)

```
1  let 𝒦 = parts of base relations
2  foreach K^sig ∈ 𝒦 do
3      K^sig = {x → K(x)
                  | x in base relation K, ActualHLs(x, θ) = sig}
4  foreach T ∈ 𝒯 do recompute views in T
```

Figure 32 Recomputing all relation parts and affected views in the view trees $\mathcal{T}$ based on the threshold θ.

has multiple child views, these child views have the same schema $\{X\} \cup dep_{\omega'_{at}}(X)$ as V_X. Since an update to any base relation in ω' fixes all variable in V_X, the delta for V_X can be computed in constant time by constant-time lookups.

If X is a bound variable and ω' has free variables, or X is an output variable and ω' has input variables, the Ω procedure partitions the base relations of ω' on $anc(X) \cup \{X\}$. In the heavy case, Ω traverses to the subtrees of ω' as in the previous case except the base relations are replaced by the heavy parts of the relations. The delta for the view constructed at X can be computed in constant time.

In the light case, Ω builds an access-top VO ω'_{at} of ω' with the light parts of the base relations as its leaves, and then τ constructs a view tree *ltree* following ω'_{at}. At variable X in ω'_{at}, τ creates a view V_X with schema $\mathcal{S}_X = \{X\} \cup dep_{\omega'_{at}}(X)$. Consider an update δR that affects the light part of relation R. While propagating the update up, at V_X, the update δR does not fix all variables in $\mathcal{S}_X$ and the unfixed variables are distributed in δ' views below V_X ($\delta' \leq \delta$ according to the definition of dynamic width). Computing the delta for V_X requires finding the values of these unfixed variables in the δ' views below V_X. Since the leaves of ω'_{at} are the light parts of the base relations, we can fetch the values of unfixed variables in each view in $\mathcal{O}(N^{\epsilon})$ time and $\mathcal{O}(N^{\delta'\epsilon})$ time in δ' views. In the worst case, δ' can be as large as δ, and therefore the update time is $\mathcal{O}(N^{\delta\epsilon})$.

Overall, the update time for a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ time. ◀

E.5.3 Processing a Sequence of Single-Tuple Updates

As the database evolves under updates, we periodically rebalance the relation partitions and views to account for a new database size and updated degrees of data values. The cost of rebalancing is amortised over a sequence of updates.

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 $\lfloor \frac{1}{4}M \rfloor \leq N < M$ always holds. If the database size falls below $\lfloor \frac{1}{4}M \rfloor$ 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^{ϵ} and recomputing the views. Figure 32 shows the major rebalancing procedure. For any base relation K and tuple $\mathbf{x}$ contained in K, the procedure computes the HL-signature *sig* of $\mathbf{x}$ based on the threshold θ and inserts $\mathbf{x}$ into K^{sig} (Line 3). It then recomputes all views in the views trees (Line 4).

$\textsc{MinorRebalancing}$(trees $\mathcal{T}$, value $\mathbf{v}$, threshold θ)

1 **let** $\mathcal{K}$ = parts of base relations
2 **foreach** $K^{sig} \in \mathcal{K}$ **do**
3   **foreach** $\mathbf{x} \in \sigma_{\mathsf{Sch}(\mathbf{v})=\mathbf{v}} K^{sig}$ **do**
4   **let** $sig' = \textsc{ActualHLs}(\mathbf{x}, \theta)$
5   **foreach** $T \in \mathcal{T}$ **do** $\textsc{Apply}(T, \delta K^{sig'} = \{\mathbf{x} \to K^{sig}(\mathbf{x})\})$
6   **foreach** $T \in \mathcal{T}$ **do** $\textsc{Apply}(T, \delta K^{sig} = \{\mathbf{x} \to -K^{sig}(\mathbf{x})\})$

Figure 33 Moving tuples $\mathbf{x}$ containing $\mathbf{v}$ to relation parts whose HL-signature matches the degree of $\mathbf{v}$ in base relations.

▶ **Proposition 50.** *Given a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with static width w, a canonical VO ω for Q, a database of size N, and $\epsilon \in [0,1]$, major rebalancing of the views in the view trees in $\textsc{ViewTrees}(\omega, (\mathcal{O}|\mathcal{I}))$ takes $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time.*

Proof. Consider the major rebalancing procedure from Figure 32. The relation parts can be computed in $\mathcal{O}(N)$ time. Proposition 36 implies that the affected views can be recomputed in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. ◀

The cost of major rebalancing is amortised over $\Omega(M)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}M$ (after doubling), or $N = \frac{1}{2}M - \frac{1}{2}$ or $N = \frac{1}{2}M - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}M \rfloor \le N < M$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}M$. The amortised major rebalancing time is then $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. By Proposition 21, we have $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$; hence, the amortised major rebalancing time is $\mathcal{O}(M^{\delta\epsilon})$.

Minor Rebalancing.

After an update $\delta R = \{\mathbf{x} \to m\}$ to relation R, we check the degrees of the values in $\mathbf{x}$. Consider a partition key k that is included in the schema of $\mathbf{x}$ and the projection $\mathbf{v}$ of $\mathbf{x}$ onto k. If $\mathbf{v}$ is included in a relation part that is light on k but the degree of $\mathbf{v}$ is not below $\frac{3}{2}M^\epsilon$ in at least one base relation, all tuples including $\mathbf{v}$ are moved to relation parts that are heavy on $\mathbf{v}$. Likewise, if $\mathbf{v}$ is in a relation part that is heavy on k but the degree of $\mathbf{v}$ is below $\frac{1}{2}M^\epsilon$ in all base relations, all tuples including $\mathbf{v}$ are moved to relation parts that are light on $\mathbf{v}$. Figure 33 shows the *minor rebalancing* procedure that moves tuples including $\mathbf{v}$ to relation parts whose HL-signature matches the degree of $\mathbf{v}$ in the base relations. For each tuple $\mathbf{x}$ in a relation part K^{sig}, it first computes the actual HL-signature sig' of $\mathbf{x}$ based on the threshold θ (Line 4). It then inserts $\mathbf{x}$ into $K^{sig'}$ (Line 5) and deletes it from K^{sig} (Line 6).

▶ **Proposition 51.** *Given a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with dynamic width δ, a canonical VO ω for Q, a database of size N, and $\epsilon \in [0,1]$, minor rebalancing of the views in the view trees in $\textsc{ViewTrees}(\omega, (\mathcal{O}|\mathcal{I}))$ takes $\mathcal{O}(N^{(\delta+1)\epsilon})$ time.*

Proof. Figure 33 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 $\mathcal{O}(\frac{3}{2}M^\epsilon)$ tuples that have $\mathbf{v}$ to relation parts that are heavy on $\mathbf{v}$ (light to heavy) or $\mathcal{O}(\frac{1}{2}M^\epsilon)$ tuples that have $\mathbf{v}$ to relation parts that are light on $\mathbf{v}$ (heavy to light). Each move is by an insert followed by a delete, which takes $\mathcal{O}(N^{\delta\epsilon})$ time, as discussed in the proof of Proposition 49. Since there are $\mathcal{O}(M^\epsilon)$ such moves and the size invariant $\lfloor \frac{1}{4}M \rfloor \le N < M$ holds, the total time is $\mathcal{O}(N^{(\delta+1)\epsilon})$. ◀

OnUpdate(view trees $\mathcal{T}$, update δR)

1 UpdateTrees($\mathcal{T}, \delta R$)
2 **if** ($|\mathcal{D}| = M$)
3    $M = 2M$
4   MajorRebalancing($\mathcal{T}, M^{\epsilon}$)
5 **else if** ($|\mathcal{D}| < \lfloor \frac{1}{4} M \rfloor$)
6    $M = \lfloor \frac{1}{2} M \rfloor - 1$
7   MajorRebalancing($\mathcal{T}, M^{\epsilon}$)
8 **else**
9   **let** $\delta R = \{\mathbf{x} \to m\}$
10   **let** $\{k_1 \to s_1, ..., k_n \to s_n\}$ = TransientHLs($\mathbf{x}$)
11   **foreach** $i \in [n]$ **do**
12   **if** ($s_i = L$ **and** $\exists K \in \mathcal{D}$: $|\sigma_{k_i = \mathbf{x}[k_i]} K| \geq \frac{3}{2} M^{\epsilon}$) **or**
13   ($s_i = H$ **and** $\forall K \in \mathcal{D}$: $|\sigma_{k_i = \mathbf{x}[k_i]} K| < \frac{1}{2} M^{\epsilon}$)
14   MinorRebalancing($\mathcal{T}, \mathbf{x}[k_i], M^{\epsilon}$)

Figure 34 Updating a set of view trees $\mathcal{T}$ under a sequence of single-tuple updates to base relations. $\mathcal{D}$ is the database. The global variable M is set to $2|\mathcal{D}| + 1$ in the preprocessing stage.

The cost of minor rebalancing is amortised over $\Omega(M^{\epsilon})$ 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. Hence, the amortised minor rebalancing time is $\mathcal{O}(N^{\delta\epsilon})$.

Figure 34 gives the trigger procedure OnUpdate that maintains a set $\mathcal{T}$ of view trees under a sequence of single-tuple updates to input relations. It first applies an update $\delta R = \{\mathbf{x} \to m\}$ to the view trees from $\mathcal{T}$ using UpdateTrees from Figure 29 (Line 1). If this update leads to a violation of the size invariant $\lfloor \frac{1}{4} M \rfloor \leq N < M$, it invokes MajorRebalancing to recompute the relation parts and views (Lines 2-7). Otherwise, it computes the transient HL-signature $\{k_1 \to s_1, \dots, k_n \to s_n\}$ of $\mathbf{x}$ (Line 10). If for any s_i, we have $s_i = L$ but there exists a relation such that the degree of $\mathbf{x}[k_i]$ is at least $\frac{3}{2} M^{\epsilon}$, or it holds $s_i = H$ but the degree of $\mathbf{x}[k_i]$ is below $\frac{1}{2} M^{\epsilon}$ in all relations, it invokes MinorRebalancing to move all tuples containing $\mathbf{x}[k_i]$ to the relation parts whose HL-signature matches the degree of $\mathbf{x}[k_i]$ in base relations (Lines 11-14).

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

▶ **Proposition 52.** *Given a hierarchical CQAP $Q(\mathcal{O}|\mathcal{I})$ with dynamic width δ, a canonical VO ω for Q, a database of size N, and $\epsilon \in [0, 1]$, maintaining the views in the view trees in* ViewTrees$(\omega, (\mathcal{O}|\mathcal{I}))$ *under a sequence of single-tuple updates takes $\mathcal{O}(N^{\delta\epsilon})$ amortised time per single-tuple update.*

Proof. By Proposition 50, a major rebalancing step requires $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time. This time is amortised over $\Omega(N)$ updates executed before the rebalancing step. Hence, the amortised time of major rebalancing is $\mathcal{O}(N^{(\mathsf{w}-1)\epsilon})$. Since $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$, we conclude that the amortised time for major rebalancing is $\mathcal{O}(N^{\delta\epsilon})$. By Proposition 51, a minor rebalancing step requires $\mathcal{O}(N^{(\delta+1)\epsilon})$ time, which is amortised over $\Omega(N)$ previous updates. This results in $\mathcal{O}(N^{\delta\epsilon})$ amortised minor rebalancing time. The formal proof for the amortised time upper bound is a straightforward extension of the amortisation proof in [**?**]. In [**?**], 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. ◀

E.6 Proof of Theorem 15

▶ **Theorem 15.** *Let any CQAP Q with static width* w *and dynamic width δ, a database of size N, and $\epsilon \in [0,1]$. If Q's fracture is hierarchical, then Q admits $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

Consider a CQAP query Q with static width w and dynamic width δ. Assume that the fracture $Q_\dagger$ of Q is hierarchical. In the preprocessing stage, we construct a set of view trees representing the result of $Q_\dagger$. These view trees can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time (Propositions 36) and can be maintained with $\mathcal{O}(N^{\delta\epsilon})$ amortised time under single-tuple updates (Proposition 52). Given any input tuple, the view trees allow for the enumeration of the result of Q with $\mathcal{O}(N^{1-\epsilon})$ enumeration delay (Proposition 45).

E.7 Proof of Corollary 16

▶ **Corollary 16.** *(Theorem 15). Let any query in $CQAP_1$, a database of size N, and $\epsilon \in$ $[0,1]$. Then Q admits $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\epsilon})$ amortised update time for single-tuple updates.*

We first show that CQAP_1 queries have dynamic width 1.

▶ **Lemma 53.** *Every $CQAP_1$ query has dynamic width* 1.

Proof. Consider a CQAP_1 query Q and its fracture $Q_\dagger$. We first show that the dynamic width of Q is at least 1. By definition, $Q_\dagger$ must be hierarchical and almost free-dominant or almost input-dominant. Assume first that $Q_\dagger$ is almost free-dominant. This means that $Q_\dagger$ contains a bound variable X and an atom $R(\mathcal{Y}) \in \mathit{atoms}(X)$ such that:

$$\mathit{free}(\mathit{atoms}(X)) \not\subseteq \mathcal{Y} \tag{6}$$

Let $\omega = (T_\omega, \mathit{dep}_\omega)$ be an arbitrary access-top variable order for $Q_\dagger$. Since the schema of each atom in $\mathit{atoms}(X)$ contains X, all variables in $\mathit{free}(\mathit{atoms}(X))$ depend on X. Hence, each variable in $\mathit{free}(\mathit{atoms}(X))$ must be on a root-to-leaf path with X. Since X is bound, the variables in $\mathit{free}(\mathit{atoms}(X))$ cannot be contained in ω_X. Hence, they must be contained in $\mathsf{anc}_\omega(X)$. This implies that $\mathit{free}(\mathit{atoms}(X)) \subseteq (\{X\} \cup \mathit{dep}_\omega(X))$. By Assumption (6), $\rho_{Q_X}((\{X\} \cup \mathit{dep}_\omega(X)) \setminus \mathcal{Y})$ must be at least 1. This implies that $\rho^*_{Q_X}((\{X\} \cup \mathit{dep}_\omega(X)) \setminus \mathcal{Y})$ must be at least 1 (Lemma 20). It follows that $\delta(\omega) \geq 1$. Since ω is an arbitrary access-top variable order for $Q_\dagger$, we derive that the dynamic width of Q is at least 1.

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

$$\mathit{in}(\mathit{atoms}(X)) \not\subseteq \mathcal{Y} \tag{7}$$

Consider any access-top variable order $\omega = (T_\omega, \mathit{dep}_\omega)$ for $Q_\dagger$. Since X is output, the variables in $\mathit{in}(\mathit{atoms}(X))$ must be contained in $\mathsf{anc}_\omega(X)$. This means that $\mathit{in}(\mathit{atoms}(X)) \subseteq$ $(\{X\} \cup \mathit{dep}_\omega(X))$. By Assumption (7), $\rho^*_{Q_X}((\{X\} \cup \mathit{dep}_\omega(X)) \setminus \mathcal{Y})$ must be at least 1. It follows that $\delta(\omega) \geq 1$. Therefore, the dynamic width of Q must be at least 1.

We now show that the dynamic width of Q is at most 1. Assume that $\mathcal{I}$ and $\mathcal{O}$ are the input and respectively the output variables of $Q_\dagger$. Let ω be a canonical variable order of $Q_\dagger$. By Lemma 30, the function $\textsc{AccessTop}(\omega, \mathcal{O}, \mathcal{I})$ in Figure 9 (Section E.3.1) constructs an access-top variable order ω^t for $Q_\dagger$ with dynamic width $\kappa(\omega, \mathcal{I}, \mathcal{O})$, where

$$\kappa(\omega, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in bound(\omega) \\ Z \in out(\omega)}} \max_{R(\mathcal{Y}) \in atoms(\omega_Y)} \{\rho^*_{Q_Y}((vars(\omega_Y) \cap \mathcal{F}) \setminus \mathcal{Y}), \rho^*_{Q_Z}((vars(\omega_Z) \cap \mathcal{I}) \setminus \mathcal{Y})\}$$

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

We are ready to prove Corollary 16. Consider a CQAP$_1$ query Q, a database of size N, and $\epsilon \in [0, 1]$. By Lemma 53, Q has dynamic width $\delta = 1$. By Proposition 21, the static width of Q is at most $\mathsf{w} = 2$. Using Theorem 15, we conclude that Q can be evaluated with $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon}) = \mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon}) = \mathcal{O}(N^{\epsilon})$ amortised update.