# 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.ICDT.2023.18

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

## 1 Introduction

We consider the problem of dynamic evaluation for conjunctive queries with access restrictions. Restricted access to data is commonplace [28, 29, 27]: For instance, the flight information behind a user-interface query can only be accessed by providing values for specific input fields such as the departure and destination airports in a flight booking database.

We formalise such 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. In database systems, CQAPs formalise the notion of parameterized queries (or prepared statements) [1]. In probabilistic graphical models, they correspond to conditional queries [25]: Such inference queries ask for (the probability of) each possible value of a tuple of random variables (corresponding to CQAP output variables) given specific values for a tuple of random variables (corresponding to CQAP input variables). Prior work on queries with access patterns considered 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 [15, 34, 12, 5, 6]. In this more general setting, and in sharp contrast to our simpler setting, a fundamental question is whether the query can even be answered for a given access pattern to each relation [28, 29, 27].

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 database. It computes a data structure that supports the enumeration of the distinct output tuples for any values of the input variables and maintains this data structure under updates to the input database.

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 [13]. The *update time* is the time used to update the data structure[1] for one single-tuple update. 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 [19, 21], 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 database, 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 and independent preparation for each enumeration request, e.g., to remove dangling tuples and possibly create a data structure to support enumeration. In contrast, the data structure constructed by our approach shares this preparation across the enumeration 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 forest of view trees, which is a data structure that represents compactly 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 call this class $\text{CQAP}_0$.

[1] We do not allow updates during the enumeration; this functionality is orthogonal to our contributions and can be supported using a versioned data structure.

All queries outside $\text{CQAP}_0$ do not admit constant-time update and delay regardless of the preprocessing time, unless the widely held Online Matrix-Vector Multiplication conjecture [17] fails. Our dichotomy generalises a prior dichotomy for $q$-hierarchical queries *without access patterns* [7]. The $q$-hierarchical queries are in $\text{CQAP}_0$, yet they have no input variables. The class $\text{CQAP}_0$ further contains cyclic queries with input variables. For instance, the edge triangle detection problem is in $\text{CQAP}_0$: Given an edge $(u, v)$, check whether it participates in a triangle. The smallest query patterns not in $\text{CQAP}_0$ 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 $\text{CQAP}_1$, which lies outside $\text{CQAP}_0$: The complexity of $\text{CQAP}_1$ for both the update time and the enumeration delay matches the lower bound $\Omega(N^{\frac{1}{2}})$ for queries outside $\text{CQAP}_0$, 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 $\text{CQAP}_1$ 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 $\epsilon$, 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 $\text{CQAP}_1$ is the Dynamic Set Intersection problem [26]: 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 [26] for this problem using $\epsilon = 0.5$.

## 2 Preliminaries

**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 [2]:

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

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 $\delta R$ 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 (amortised) 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 (amortised) constant time.

We next exemplify a data structure that conforms to the above computational model. 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.

## 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 $atoms(A) \subset atoms(B)$. 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 $R(\mathcal{X}) \in atoms(B)$, there is another atom $S(\mathcal{Y}) \in atoms(B)$ 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 $A, B \in vars(Q)$, either $atoms(A) \subseteq atoms(B)$, $atoms(B) \subseteq atoms(A)$, or $atoms(B) \cap atoms(A) = \emptyset$. 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 [7].

▶ **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 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 [31]. 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) $\omega$ for a CQAP $Q$ is a pair $(T_\omega, dep_\omega)$, where:

- $T_\omega$ 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 $T_\omega$.
- The function $dep_\omega$ maps each variable $X$ to the subset of its ancestor variables in $T_\omega$ on which the variables in the subtree rooted at $X$ depend.

An *extended* VO is a VO where we add as new leaves atoms corresponding to relations and their indicator projections. We add each atom in the query as child of its variable placed lowest in the VO. We explain next how the indicator projections are added to a VO $\omega$. The role of the indicators is to reduce the asymptotic complexity of cyclic queries [2].

Given a CQAP $Q$ and a VO $\omega$, where the atoms of $Q$ have been already added, the function indicators in Figure 1 extends $\omega$ with indicator projections. At each variable $X$ in

```
indicators(CQAP $Q$, VO $\omega$) : extended VO
switch $\omega$:
$R(\mathcal{Y})$
    1 return $R(\mathcal{Y})$
```

$$\begin{tikzcd}[column sep=tiny] & X \arrow[dl, dash] \arrow[dr, dash] & \\ \omega_1 & \cdots & \omega_k \end{tikzcd}$$

```
    2 let $\hat{\omega}_i$ = indicators($\omega_i$)  $\forall i \in [k]$
    3 let $\mathcal{S} = \{X\} \cup dep_\omega(X)$ and $\mathcal{R}$ be the set of atoms in $\omega$
    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 $\{I_1, ..., I_\ell\}$ = GYO$^*(\mathcal{I}, \mathcal{R})$
    6 return
```

$$\begin{tikzcd}[column sep=tiny] & & & X \arrow[dlll, dash] \arrow[dl, dash] \arrow[dr, dash] \arrow[drrr, dash] & & & \\ \hat{\omega}_1 & \cdots & \hat{\omega}_k & & I_1 & \cdots & I_\ell \end{tikzcd}$$

**Figure 1** Adding indicator projections to a VO $\omega$ of a CQAP $Q$. The function indicators is defined using pattern matching on the structure of the VO $\omega$, which can be a leaf or an inner node (cf. left column under **switch**). Each variable $X$ in $\omega$ 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. GYO$^*$ (Section 3.1) is based on the GYO reduction [4].

$\omega$, 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 $\omega$ rooted at $X$ (Line 5). We achieve this using a variant of the GYO reduction [4]. Given the hypergraph formed by the hyperedges representing these indicators $\mathcal{I}$ and the atoms $\mathcal{R}$, GYO repeatedly applies two rules until it reaches a fixpoint: (1) Remove a node that only appears in one hyperedge; (2) Remove a hyperedge that is included in another hyperedge. If the result of GYO is a hypergraph with no nodes and one empty hyperedge, then the input hypergraph is ($\alpha$-)acyclic. Otherwise, the input hypergraph is cyclic and the GYO's output is a hypergraph with cycles. Our GYO variant, dubbed GYO$^*$ in Figure 1, returns the hyperedges that originated from the indicator projections in $\mathcal{I}$ and contribute to this non-empty output hypergraph. The chosen indicator projections become children of $X$ (Line 6).

In the rest of this paper, whenever we refer to a variable order, we always assume an extended VO.

▶ **Example 3.** 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 2 depicts a VO $\omega$ 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 $T$. ⌟

We introduce notation for an extended VO $\omega$. Its subtree rooted at $X$ is denoted by $\omega_X$. The sets $vars(\omega)$ and $\mathsf{anc}_\omega(X)$ consist of all variables of $\omega$ 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 $\omega$ and by $Q_X$ the join of all atoms $atoms(\omega)$ (all variables are free).

We next introduce classes of VOs for CQAP queries. A VO $\omega$ is *canonical* if the variables of the leaf atom of each root-to-leaf path are the inner nodes of the path. Hierarchical queries

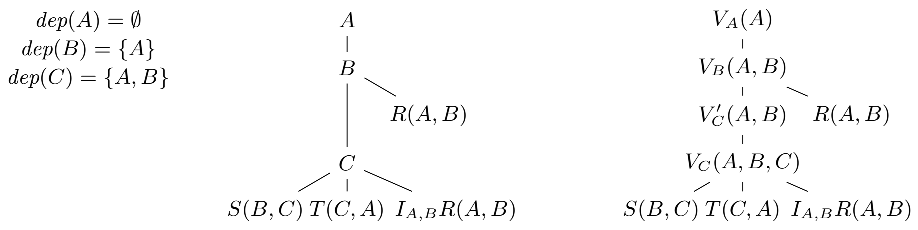

**Figure 2** Left: (Access-top extended) VO for the query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. Right: The view tree constructed from this VO. Note the indicator $I_{A,B}R(A, B)$ added below the variable $C$ (left) and below the view $V_C$ (right).

are precisely those conjunctive queries that admit canonical variable orders. A VO $\omega$ is *free-top* if no bound variable is an ancestor of a free variable. It is *input-top* if no output variable is an ancestor of an input variable. The sets of free-top and input-top 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 4.** 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 $\omega$ and CQAP $Q$. They capture the complexity of computing and maintaining the output of $Q$.

▶ **Definition 5.** *The static width* $\mathsf{w}(\omega)$ *and dynamic width* $\delta(\omega)$ *of a VO* $\omega$ *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 [3] $\rho^*_{Q_X}(\mathcal{X})$ 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 $\omega$. The static width $\mathsf{w}$ of a VO $\omega$ is then defined by the maximum over the fractional edge cover numbers of the queries $Q_X$ for the variables $X$ in $\omega$. The dynamic width is defined similarly, with one simplification: We consider every case of a relation (or indicator projection) $R$ being replaced by a single-tuple update, so its variables $\mathcal{Y}$ are all set to constants and can be ignored 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 6.** *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 6 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. The extended technical report [22] further expands on the width measures with examples and properties.

▶ **Example 7.** Consider the query $Q(\mathcal{O} \mid \mathcal{I}) = R(A, B, C), S(A, B, D), T(A, E)$. The static width $\mathsf{w}$ and the dynamic width $\delta$ 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 $\delta = 0$.

Recall the triangle CQAP query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$ from Example 3 and its access-top VO in Figure 2. By adding the indicator $I_{A,B}R$ below $C$, the fractional edge cover number $\rho^*(\{C\} \cup \mathit{dep}(C)) = \rho^*(\{A, B, C\})$ of the query $Q_C(A, B, C) = S(B, C), T(C, A), I_{A,B}R(A, B)$ 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 $\omega$ is $\frac{3}{2}$.

The dynamic width of $\omega$ is dominated by the fractional edge cover number $\rho^*(\{C\} \cup \mathit{dep}(C)) - \mathcal{S}) = \rho^*(\{A, B, C\} - \mathcal{S})$ of the query $Q_C$, where $\mathcal{S}$ is the schema of $S$, $T$, or $I_{A,B}R$. In each of these three cases, $\{A, B, C\} - \mathcal{S}$ consists of a single variable. Hence, the fractional edge cover number is 1 and then the dynamic width of $\omega$ is 1. ⌟

## 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 8.** *Given a CQAP with static width $\mathsf{w}$ and dynamic width $\delta$ 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. 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 [30] is a (rooted) tree with one view

$\tau(\text{VO } \omega)$ : view tree

**switch** $\omega$:

$R(\mathcal{Y})$

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

$$\begin{tikzcd}[column sep=tiny] & X \arrow[dl, dash] \arrow[dr, dash] & \\ \omega_1 & \cdots & \omega_k \end{tikzcd}$$

```
2 let $T_i = \tau(\omega_i)\ \ \forall i \in [k]$
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 $\left\{ \begin{tikzcd}[column sep=tiny] & V_X(\mathcal{S}) \arrow[dl, dash] \arrow[dr, dash] & \\ T_1 & \cdots & T_k \end{tikzcd} \right.$
5 let $V'_X(\mathcal{S} \setminus \{X\}) = V_X(\mathcal{S})$   return $\left\{ \begin{tikzcd}[column sep=tiny] & V'_X(\mathcal{S} \setminus \{X\}) \arrow[d, dash] & \\ & V_X(\mathcal{S}) \arrow[dl, dash] \arrow[dr, dash] & \\ T_1 & \cdots & T_k \end{tikzcd} \right.$
```

**Figure 3** Constructing a view tree following a VO $\omega$. The function $\tau$ is defined using pattern matching on the structure of the VO $\omega$, which can be a leaf or an inner node (cf. left column under **switch**). At each variable $X$ in $\omega$, the function creates a view $V_X$ whose schema consists of $X$ and the dependency set of $X$. If $X$ has siblings, it adds a view on top of $V_X$ that marginalises out $X$.

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 $\omega$ of $Q_\dagger$. In the following, we discuss the case of $\omega$ consisting of a single tree; otherwise, we apply the preprocessing stage to each tree in $\omega$.

Given an access-top VO $\omega$, the function $\tau(\omega)$ in Figure 3 returns a view tree constructed from $\omega$. The function traverses $\omega$ 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 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 $\omega$ 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 8.

The next example demonstrates the construction of a view tree for a CQAP$_0$ query.

▶ **Example 9.** Figure 4 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 5 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

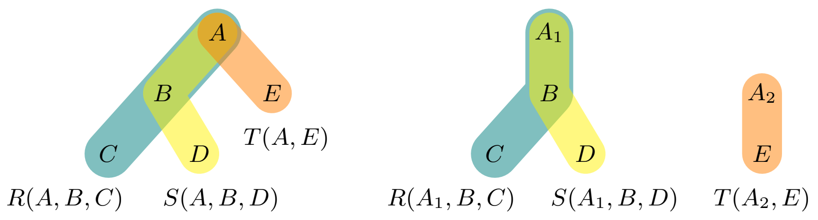

**Figure 4** (Left) Hypergraph of the two queries with the same body but different access patterns, as used in Examples 9 and 10; (middle and right) hypergraph of their fractures.

$$\begin{aligned} dep(A_1) &= \emptyset \\ dep(B) &= \{A_1\} \\ dep(C) &= \{A_1, B\} \\ dep(D) &= \{A_1, B\} \end{aligned}$$

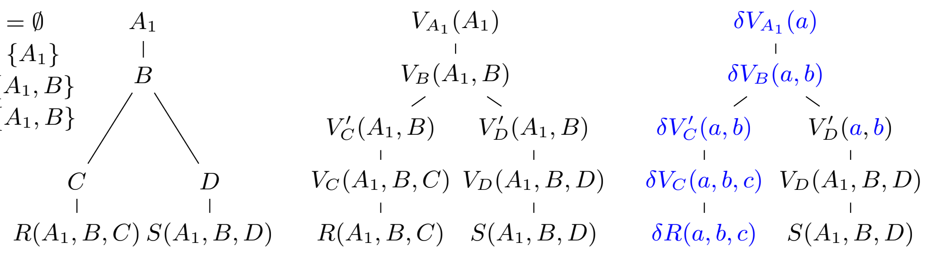

**Figure 5** (Left) Access-top VO for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree constructed from the VO; (right) the delta view tree under a single-tuple update to $R$.

$\{B, A_1\} = \{B\} \cup dep(B)$. 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 10.** 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 4) of $Q$ and its fracture are the same as for the query in Example 9. Figure 6 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 9 and can be constructed in linear time. ⌟

Finally, we exemplify the construction of a view tree for a cyclic query.

▶ **Example 11.** Figure 2 depicts a VO and the view tree constructed from it for the triangle CQAP query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$ from Example 3. 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. ⌟

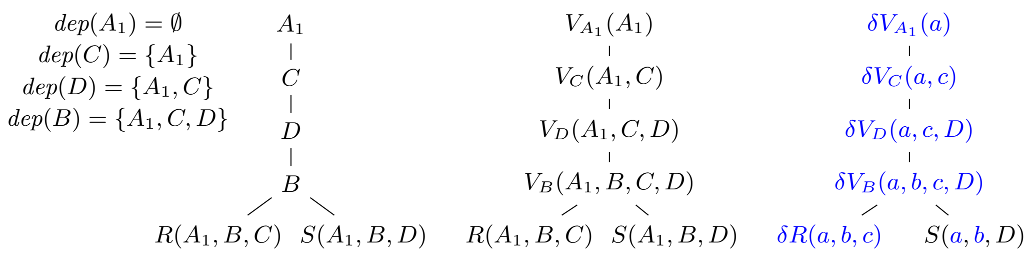

**Figure 6** (Left) Access-top VO for $Q_1(B, D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree corresponding to the VO; (right) the delta view tree under a single-tuple update to $R$.

## 4.2 Enumeration

The view trees constructed by the function $\tau$ 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 $\omega_i$ is a tree in the forest $\omega$ 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 $\mathcal{O}_i = \mathcal{O} \cap vars(\omega_i)$ and $\mathcal{I}_i = \mathcal{I}_\dagger \cap vars(\omega_i)$ 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 12.** Consider the query $Q(B, C, D, E|A)$ from Example 9 and the two connected components $Q_1(B, C, D|A_1)$ and $Q_2(E|A_2)$ of its fracture. Figure 5 (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 $\tau$ in Figure 3. 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 [9]. The update $\delta R$ 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 $\delta R$ 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 $\delta R$ 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 $\delta R$.

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 $\omega$ is $\delta(Q)$, we conclude that the update time of our approach is $\mathcal{O}(N^{\delta(Q)})$, as stated in Theorem 8.

▶ **Example 13.** Figure 5 (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 $\delta R$ (in blue) are the deltas of the corresponding views, computed by propagating $\delta R$ 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 6 (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 relation takes $O(N)$ time. ⌟

## 4.4 Discussion

So far in this section, we explained how our approach works. We conclude with a high-level discussion on key decisions behind our approach.

**1. Variable orders.** Our approach can be rephrased to use tree decompositions [16] instead of VOs, since they are different syntaxes for the same query decomposition class [31]. Indeed, the set consisting of a variable and its dependency set in a VO can be interpreted as a bag of a tree decomposition whose edges between bags reflect those between the variables in the VO. Variable orders are more natural for our algorithms for constructing view trees and for enumeration as well as worst-case optimal join algorithms such as the LeapFrog TrieJoin [33] and their use for constructing factorized representations of query results [31]: These algorithms proceed one variable at a time and not one bag of variables at a time. VO-based algorithms express more naturally computation by variable elimination.

**2. Access-top VOs.** Access-top VOs can have higher static and dynamic widths than arbitrary VOs. However, they are needed to attain the constant-delay enumeration in

Theorem 8, as explained next. The maintenance procedure for view trees ensures that each view is calibrated[2] with respect to all of its descendant views and relations, since the updates are propagated bottom-up from the relations to the top view. Since the views constructed for the input variables are above all other views in a view tree constructed from an access-top VO, these views are calibrated. For a given tuple of values over the input variables, the calibration of these views guarantees that if they do not agree with this tuple, then there is no output tuple associated with the input tuple. For constant-delay enumeration, we follow a top-down traversal of the view tree and use the constant-time lookup of the hash maps implementing the views. Furthermore, since the output variables are above the bound variables in the VO, tuples of values over the output variables can be retrieved from views whose schemas do not contain bound variables. Hence, we can enumerate the *distinct* tuples over the output variables for a given tuple over the input variables.

In case we would have used an arbitrary (and not access-top) VO, then the input variables may be anywhere in the VO; in particular, there may be views above the relations with the input variables that do not have input variables. On an enumeration request, the values given to the input variables act as selection conditions on the relations and may require the calibration of the views on top before the enumeration starts; this calibration may be as expensive as computing the query. Otherwise, we incur a non-constant cost for the enumeration of each output tuple. Either way, the enumeration delay may not be constant.

**3. Lazy approach using residual queries.** A simple CQAP evaluation approach is the lazy approach. On updates, the lazy approach just updates the input relations. On enumeration, where each input variable is given a value, it computes the residual query obtained by setting the input variables to the given values. The enumeration of the tuples in the output of a residual query cannot guarantee constant delay, since the parts of the input relations, which satisfy the selection conditions on the input variables, are not necessarily calibrated, and the calibration may take as much time as computing the residual query.

**4. Replacing each occurrence of an input variable by a fresh variable.** Although this query rewriting removes the joins on the input variables, it does not affect the correctness of query evaluation. For enumeration, all fresh variables are fixed to given values. In access-top VOs, these variables are above the other variables and are in views that are calibrated with respect to the relations in their respective connected component of the rewritten query. We can then check whether all view trees satisfy the assignment of values to the input values. If a view tree fails, then the query output is empty for the values given to the input variables.

**5. Query fractures.** The query rewriting in the previous discussion point is only the first step of query fracturing. The second step merges all fresh variables for an input variable into one variable in case they are in the same connected component. This does not affect correctness but may affect the complexity, as exemplified next. Consider the triangle query in Example 11: $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. If we were to replace $A$ by two fresh variables $A_1$ and $A_2$, then the rewritten query would be: $Q'(B, C|A_1, A_2) = R(A_1, B), S(B, C), T(C, A_2)$. It still has one connected component. An access-top VO for $Q'$ is $A_1 - A_2 - B - C$ ($A_1$ and $A_2$ may be swapped, same for $B$ and $C$). The static width of $Q'$ is 2. Yet by merging back $A_1$ and $A_2$, we obtain $Q$, which admits the access-top VO $A - B - C$ and static width 3/2 (same width can be obtained if $B$ and $C$ are swapped), as in Example 11.

[2] A relation $R$ is calibrated with respect to other relations in a query $Q$ if each tuple in $R$ participates to at least one tuple in the output of $Q$.

## 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 14.** *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 14 is based on the following OMv problem:

▶ **Definition 15** (Online Matrix-Vector Multiplication (OMv) [17]). *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 16** (OMv Conjecture, Theorem 2.4 [17]). *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 [22]. Our approach from Section 4 achieves linear preprocessing time, constant update time and enumeration delay for such queries (Theorem 8), 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. $Q_3$ 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 [7]. To prove the hardness statement in Theorem 14, 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 8 can be parameterised to uncover trade-offs between preprocessing, update, and enumeration.

▶ **Theorem 17.** *Let any CQAP $Q$ with static width $\mathsf{w}$ and dynamic width $\delta$, a database of size $N$, and $\epsilon \in [0,1]$. If $Q$'s fracture is hierarchical, then $Q$ admits $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

This trade-off continuum can be obtained using one algorithm parameterised by $\epsilon$. This algorithm either recovers or has lower complexity than prior approaches. Using $\epsilon = 1$, we recover the complexities in Theorem 8 and therefore also the constant update time and delay for queries in CQAP$_0$ in Theorem 14.

Theorem 17 can be refined for CQAP$_1$, since $\delta = 1$ and $\mathsf{w} \leq 2$ for queries in this class.

▶ **Corollary 18.** *(Theorem 17). 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 $\text{CQAP}_1$ match the lower bound in Theorem 14 for all queries outside $\text{CQAP}_0$. This makes our approach weakly Pareto optimal for $\text{CQAP}_1$, as lowering both the update time and delay would violate the $\mathsf{OMv}$ conjecture.

Our algorithm has two core ideas. (For lack of space, we defer the details to the extended technical report [22].) 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 $\epsilon$.

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 [14] and view tree iterators, as in prior work [23]. 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 $\epsilon$. 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 $\theta$, the pair $(R^{\mathcal{S}\to H}, R^{\mathcal{S}\to L})$ is a *partition* of $R$ on $\mathcal{S}$ with threshold $\theta$ if it satisfies the conditions:

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

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

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

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 $\mathcal{S}$, as they consist of all $\mathcal{S}$-tuples that are heavy and respectively light in $R$. 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 $R^{\mathcal{S}_1 \to s_1,\ldots,\mathcal{S}_n \to s_n}$ to denote the relation part obtained after partitioning $R^{\mathcal{S}_1 \to s_1,\ldots,\mathcal{S}_{n-1} \to s_{n-1}}$ on $\mathcal{S}_n$, where $s_i \in \{H, L\}$ for $i \in [n]$. The domain of $R^{\mathcal{S}_1 \to s_1,\ldots,\mathcal{S}_n \to s_n}$ is the intersection of the domains of $R^{\mathcal{S}_i \to s_i}$, for $i \in [n]$. We refer to $\mathcal{S}_1 \to s_1, \ldots, \mathcal{S}_n \to s_n$ 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 $\tau$ from the general case (Figure 3).

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

If $Q_X$ is in $\text{CQAP}_0$, we turn $\omega_X$ 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 $\text{CQAP}_0$, this restructuring does not increase their static width.

If $Q_X$ is not in $\text{CQAP}_0$, then $\omega_X$ 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 over relations parts that are heavy on *key*. For the second set of VOs, we turn $\omega_X$ 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 $\text{CQAP}_0$ 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 trees can be materialised in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$, where $\mathsf{w}$ is the static width of the query.

▶ **Example 19.** We explain the construction of the views tree for the connected component from Figure 4 (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 5 (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

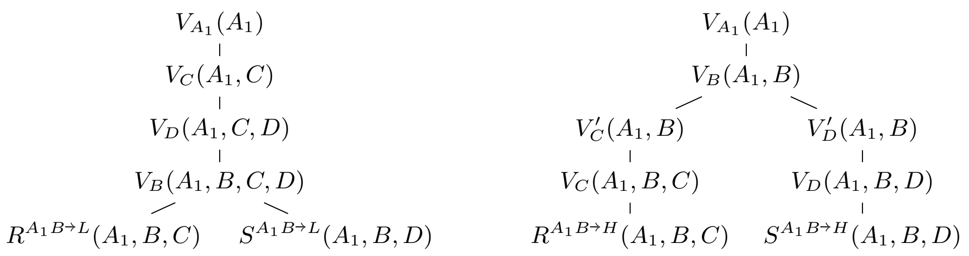

**Figure 7** View trees constructed for $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$ from Example 19 using the VOs: (left) $A_1 - C - D - B - \{R^{A_1 B \to L}(A_1, B, C), S^{A_1 B \to L}(A_1, B, D)\}$ and (right) $A_1 - B - \{C - R^{A_1 B \to H}(A_1, B, C), D - S^{A_1 B \to H}(A_1, B, D)\}$.

threshold $N^\epsilon$, 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 7 (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 $N^\epsilon$ $D$-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 7 right). The VO and view tree are the same as in Figure 4, 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 9. 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 20.** We consider the maintenance of the view trees from Figure 7 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 $N^\epsilon$ $D$-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.** Our notion of queries with free access patterns corresponds to parameterized queries [1]. These queries have selection conditions that set variables to parameter values to be supplied at query time. Prior work closest in spirit to ours investigates the space-delay trade-off for the static evaluation of full conjunctive queries with free access patterns [11]. It 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 [10]. Further works on queries with access patterns [15, 34, 12, 5, 6] consider the setting where *input* relations have input and output variables and there is no restriction on whether they are bound or free; also, a variable may be input in a relation and output in another. This poses the challenge of whether the query can be answered under specific access restrictions [28, 29, 27].

**Dynamic evaluation.** Our work generalises the dichotomy for $q$-hierarchical queries under updates [7] and the complexity trade-offs for queries under updates [19, 20, 21]. The IVM approaches Dynamic Yannakakis [18] and F-IVM [30], which is implemented on top of DBToaster [24], achieve (i) linear-time preprocessing, linear-time single-tuple updates, and constant enumeration delay for free-connex acyclic queries; and (ii) linear-time preprocessing, constant-time single-tuple updates, and constant enumeration delay for $q$-hierarchical queries. Theorem 8 recovers these results by noting that the static and dynamic widths are: 1 and respectively in $\{0, 1\}$ for free-connex acyclic queries and 1 and respectively 0 for $q$-hierarchical queries. We refer the reader to a comprehensive comparison [23] of dynamic query evaluation techniques and how they are recovered by the trade-off [21] extended in our work.

Our $\text{CQAP}_0$ dichotomy strictly generalises the one for $q$-hierarchical queries [7]: The set of $q$-hierarchical queries is a strict subset of $\text{CQAP}_0$, while there are hard patterns of non-$\text{CQAP}_0$ beyond those for non-$q$-hierarchical queries.

There are key technical differences between the prior framework for dynamic evaluation trade-off [21] 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. One advantage is a simpler complexity analysis for the views, since the variables orders and their view trees share the same width measures.

**Cutset optimisations.** Cutset conditioning [32] and cutset sampling [8] are used for efficient exact and approximate inference in Bayesian networks. The idea is to *choose* a cutset, which is a subset of variables, such that conditioning on the variables in the cutset, i.e., instantiating them with possible values, yields a network with a small treewidth that allows exact inference. The set of input variables of a CQAP can be seen as a *given* cutset, while fixing the input variables to given values is conditioning. Query fracturing, as introduced in our work, is a query rewriting technique that does not have a counterpart in cutset optimisations in AI.

## 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

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.

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

4 Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.

5 Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with Access Patterns and Integrity Constraints. *VLDB*, 8(6):690–701, 2015.

6 Michael Benedikt, Balder Ten Cate, and Efthymia Tsamoura. Generating Low-cost Plans from Proofs. In *PODS*, pages 200–211, 2014.

7 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017.

8 Bozhena Bidyuk and Rina Dechter. Cutset Sampling for Bayesian Networks. *J. Artif. Intell. Res.*, 28:1–48, 2007.

9 Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012.

10 Shaleen Deep, Xiao Hu, and Paraschos Koutris. Space-Time Tradeoffs for Answering Boolean Conjunctive Queries. *CoRR*, abs/2109.10889, 2021.

11 Shaleen Deep and Paraschos Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.

12 Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting Queries using Views with Access Patterns under Integrity Constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.

13 Arnaud Durand and Etienne Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *TOCL*, 8(4):21, 2007.

14 Arnaud Durand and Yann Strozecki. Enumeration Complexity of Logical Query Problems with Second-order Variables. In *CSL*, pages 189–202, 2011.

15 Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Rec.*, 28(2):311–322, 1999.

16 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *PODS*, pages 21–32, 1999.

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

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

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

20 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining Triangle Queries under Updates. *TODS*, 45(3):11:1–11:46, 2020.

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

22 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive Queries with Free Access Patterns under Updates. *CoRR*, abs/2206.09032, 2022.

23 Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *To appear in LMCS*, 2023.

24 Christoph Koch et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014.

25 Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

26 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Dynamic Set Intersection. In *WADS*, pages 470–481, 2015.

27 Chen Li and Edward Chang. On Answering Queries in the Presence of Limited Access Patterns. In *ICDT*, pages 219–233, 2001.

28 Alan Nash and Bertram Ludäscher. Processing First-Order Queries under Limited Access Patterns. In *PODS*, pages 307–318, 2004.

29 Alan Nash and Bertram Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *EDBT*, pages 422–440, 2004.

30 Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018.

31 Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *TODS*, 40(1):2:1–2:44, 2015.

32 Judea Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

33 Todd L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*, pages 96–106, 2014.

34 Ramana Yerneni, Chen Li, Jeffrey Ullman, and Hector Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In *ICDT*, pages 348–364, 1999.