

CONJUNCTIVE QUERIES WITH FREE ACCESS PATTERNS UNDER UPDATES

AHMET KARA ^a, MILOS NIKOLIC ^b, DAN OLTEANU ^c, AND HAOZHE ZHANG ^c

^a OTH Regensburg
e-mail address: ahmet.kara@oth-regensburg.de

^b University of Edinburgh
e-mail address: milos.nikolic@ed.ac.uk

^c University of Zurich
e-mail address: dan.olteanu@uzh.ch, haozhe.zhang@uzh.ch

ABSTRACT. We study the problem of answering conjunctive queries with free access patterns (CQAPs) 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 that works for all CQAPs and is optimal for two classes of CQAPs. This approach recovers prior work on the dynamic evaluation of conjunctive queries without access patterns.

We first give a syntactic characterisation of all CQAPs that admit constant time per single-tuple update and whose output tuples can be enumerated with constant delay given a tuple of values over the input variables.

We further chart the complexity trade-off between the preprocessing time, update time and enumeration delay for a class of CQAPs. For some of these CQAPs, our approach achieves optimal, albeit non-constant, update time and delay. This optimality is predicated on the Online Matrix-Vector Multiplication conjecture.

We finally adapt our approach to the dynamic evaluation of tractable CQAPs over probabilistic databases under updates.

1. INTRODUCTION

We consider the problem of dynamic evaluation for conjunctive queries with access restrictions. Restricted access to data is commonplace [NL04a, NL04b, LC01]: 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 (CQAPs 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.

Key words and phrases: fully dynamic algorithm, enumeration delay, complexity trade-off, dichotomy, probabilistic databases.

Example 1.1. Assume that a flight booking company has a (simplified) database consisting of the two relations **Flight** and **Airport**. The relation **Flight** contains information about flights, including flight numbers, the departure and arrival airports, and the date of the flights. The relation **Airport** contains the names of airports and the cities in which they are located. Assume that the company provides a user-interface where users can search for flights by specifying a departure city **depCity**, an arrival city **arrCity**, and a departure date **date**. Given such a triple of inputs, the user-interface queries the database and lists all flight numbers **flightNo** together with the corresponding departure and arrival airports, **depAirport** and **arrAirport**, that match the user request. We formalise this data access using the following CQAP:

$$\begin{aligned} \text{FlightSearch}(\text{flightNo}, \text{depAirport}, \text{arrAirport} \mid \text{depCity}, \text{arrCity}, \text{date}) = \\ \text{Flight}(\text{flightNo}, \text{depAirport}, \text{arrAirport}, \text{date}), \\ \text{Airport}(\text{depAirport}, \text{depCity}), \text{Airport}(\text{arrAirport}, \text{arrCity}) \end{aligned}$$

The variables in the query head that appear after the symbol \mid , i.e., **depCity**, **arrCity**, and **date**, are input variables. The other variables in the head, i.e., **flightNo**, **depAirport**, and **arrAirport**, are output variables. If a user is interested in all flights from Edinburgh to Zurich on the 1st of January 2024, the user interface runs the query **FlightSearch** after setting the input variables to the values "Edinburgh", "Zurich", and "2024-01-01". \square

In database systems, CQAPs formalise the notion of parameterized queries (or prepared statements) [AHV95]. In probabilistic graphical models, they correspond to conditional queries [KF09]: 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 [FLMS99, YLUGM99, DLN07, BLT15, BTCT14]. 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 [NL04a, NL04b, LC01].

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 [DG07]. The *update time* is the time used to update the data structure¹ for one single-tuple update.

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

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

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 language of CQAPs. 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 forest of (rooted) trees with one node per variable, 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 is set to the corresponding given input value. Yet this may lead to structurally simpler queries whose fully dynamic evaluation admits lower complexity.

Section 4 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 possible access-top variable orders of the fracture of the query.

Section 5 overviews our main results on the complexity of dynamic CQAP evaluation.

Sections 6-8 introduce our approach for dynamic 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 9 discusses key decisions behind our approach.

Section 10 gives a syntactic characterisation of those CQAPs that admit linear-time preprocessing and constant-time update and enumeration delay. We call this class of well-behaved queries CQAP_0 . All queries outside CQAP_0 and without repeating relation symbols do not admit constant-time update and delay regardless of the preprocessing time, unless the widely held Online Matrix-Vector Multiplication conjecture [HKNS15] fails. This dichotomy generalises a prior dichotomy for q -hierarchical queries *without access patterns* [BKS17]. The q -hierarchical queries are in CQAP_0 and have no input variables. The class CQAP_0 further contains cyclic CQAPs with input variables.

Example 1.2. The following triangle detection problem is in CQAP_0 : Given three nodes in a graph, we ask whether the nodes form a triangle. This problem can be expressed by the following query in CQAP_0 :

$$Q(\cdot | A, B, C) = E(A, B), E(B, C), E(C, A),$$

where E is the edge relation of the graph. All variables of the query are free and input. The dot (\cdot) in the query head signals that the query does not have output variables. \square

The smallest query patterns not in CQAP_0 strictly include the non- q -hierarchical ones and also others that are sensitive to the interplay of the output and input variables.

Section 11 charts the preprocessing time - update time - enumeration delay trade-off for the dynamic evaluation of CQAPs whose fractures are hierarchical. It shows that by

increasing the preprocessing and update times, we can decrease the enumeration delay. Our trade-off reveals the optimality for a particular class of CQAPs with hierarchical fractures, called CQAP_1 , which lies outside CQAP_0 .

Example 1.3. The following edge triangle listing problem is in CQAP_1 : Given an edge in a graph, the task is to list all triangles containing this edge. This problem can be expressed by the following query in CQAP_1 :

$$Q(C|A, B) = E(A, B), E(B, C), E(C, A). \quad \square$$

The complexity of CQAP_1 for both the update time and the enumeration delay matches the (conditional) 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 there can be no tighter upper bounds for *both* the update time and the enumeration delay (though it does not rule out the possibility that one of the two times can be lowered). 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 every $\epsilon \in [0, 1]$. 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 (Section 11.5). Our approach recovers the complexity of the well-studied dynamic set intersection problem [KPP15]:

Example 1.4. The dynamic set intersection problem is defined as follows. We are given sets S_1, \dots, S_m that are 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 the sets S_i and S_j is empty. Consider a relation S that consists of the tuples $\{(i, x) \mid x \in S_i \text{ and } i \in \{1, \dots, m\}\}$. The dynamic set intersection problem is expressed by the following query in CQAP_1 :

$$Q(\cdot|B, C) = S(B, A), S(C, A).$$

The variables B and C are free and input. The query has no output variables. Prior work designs a randomised algorithm for this problem that uses expected $\mathcal{O}(N^{\frac{1}{2}})$ update time and enumeration delay, where N is the size of the sets [KPP15]. Our approach recovers these complexities using our deterministic algorithm and $\epsilon = \frac{1}{2}$. \square

Our dynamic evaluation approach for CQAPs can be applied to further domains. Section 12 discusses three possible semantics for updates in probabilistic databases: set, bag, and expectation-variance. In these probabilistic settings, an update can be an insertion or a deletion of a tuple with an arbitrary probability. Section 13 shows how to maintain hierarchical conjunctive queries without repeating relation symbols with constant update time and enumeration delay under the set semantics and the expectation-variance semantics for updates to the underlying probabilistic database.

Example 1.5. The CQAP language naturally expresses conditional queries over probabilistic databases, asking for the probability of a certain outcome *given* specific values for the input variables. Consider the flight search query in Example 1.1 and a probabilistic version of the relation **Flight**, which encodes the probability of each flight taking place based on historical evidence. The query returns tuples of flight number, departure and arrival airports, given date and departure and arrival cities, *together* with the probability for the flight to happen.

Consider now a probabilistic graph, where each edge has a probability for being in the graph. The edge relation of the graph has one tuple per probabilistic edge. Then, the CQAP in Example 1.2 returns the probability that three given vertices (a, b, c) form a triangle in

the graph. The CQAP in Example 1.3 gives, for each input edge (a, b) , each C -node c with which it forms a triangle (a, b, c) and the probability of that triangle. \square

A prior version of this work appeared in ICDT 2023 [KNOZ23a]. This article extends the prior version as follows. We illustrate CQAPs using Examples 1.1-1.4 and variable orders using Examples 4.7 and 4.10. We include the proof of Theorem 5.1, which states the complexity of our approach for the dynamic evaluation of arbitrary CQAPs. Theorem 5.1 is now implied by the new Propositions 6.4, 7.2, and 8.2. We also include the proof of Theorem 5.2, which characterises the class of CQAPs that admit linear preprocessing time, constant update time, and constant enumeration delay. Furthermore, we explain in more detail the adaptive evaluation strategy that achieves the preprocessing - update - enumeration trade-offs for CQAPs with hierarchical fractures (Sections 11.2 – 11.4). Finally, we introduce three update semantics for probabilistic databases (Section 12) and show that our approach maintains queries in CQAP₀ over probabilistic databases with constant update time and enumeration delay under two of these update semantics (Section 13). Due to lack of space, some proofs and technical details are deferred to the technical report [KNOZ25].

2. PRELIMINARIES

Data Model. A schema $\mathcal{X} = (X_1, \dots, X_n)$ is a tuple of distinct variables. Each variable X_i has a discrete domain $\text{Dom}(X_i)$. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple \mathbf{x} of values has schema $\mathcal{X} = \text{Sch}(\mathbf{x})$ and is an element from $\text{Dom}(\mathcal{X}) = \text{Dom}(X_1) \times \dots \times \text{Dom}(X_n)$. A relation R over schema \mathcal{X} is a function $R : \text{Dom}(\mathcal{X}) \rightarrow \mathbb{Z}$ such that the multiplicity $R(\mathbf{x})$ 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 $\mathbf{t} \in \text{Dom}(\mathcal{S})$: $\sigma_{\mathcal{S}=\mathbf{t}}R = \{\mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{S}] = \mathbf{t}\}$ 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 [KNR16]:

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

That is, the indicator projection $I_{\mathcal{Y}}R$ is a relation mapping the tuples from $\pi_{\mathcal{Y}}R$ to 1.

Updates. An update is a relation, where tuples with positive multiplicities represent inserts and tuples with negative multiplicities represent deletes. Consider a relation R and an update δR over the same schema \mathcal{X} . To apply the update δR to R means to compute their *union* $R \uplus \delta R$ defined as:

$$(R \uplus \delta R)(\mathbf{x}) = R(\mathbf{x}) + \delta R(\mathbf{x}), \text{ for } \mathbf{x} \in \text{Dom}(\mathcal{X}).$$

A single-tuple update to relation R is a singleton relation $\delta R = \{\mathbf{x} \rightarrow m\}$, where the multiplicity $m = \delta R(t)$ of the tuple t in δR is non-zero.

Updates to input relations may cause changes to indicator projections. Applying the single-tuple update δR to R triggers a single-tuple update $\delta I_{\mathcal{Y}}R = \{\mathbf{x}[\mathcal{Y}] \rightarrow k\}$ to $I_{\mathcal{Y}}R$ in

the following two cases. If δR is an insertion and $\mathbf{x}[\mathcal{Y}]$ is a value not already in $\pi_{\mathcal{Y}}R$, then the new update $\delta I_{\mathcal{Y}}R$ is triggered with $k = 1$. If δR is a deletion and $\pi_{\mathcal{Y}}R$ does not contain $\mathbf{x}[\mathcal{Y}]$ after applying the update to R , then the new update $\delta I_{\mathcal{Y}}R$ is triggered with $k = -1$.

Computational Model. We consider the RAM model of computation where schemas and data values are stored in registers of logarithmic size and operations on them can be done in constant time². We assume that each relation 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 $\mathbf{t} \in \text{Dom}(\mathcal{S})$ 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

$$Q(\mathcal{O}|\mathcal{I}) = R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n). \quad (3.1)$$

We denote by: $(R_i)_{i \in [n]}$ the relation symbols; $(R_i(\mathcal{X}_i))_{i \in [n]}$ the atoms; $\text{vars}(Q) = \bigcup_{i \in [n]} \mathcal{X}_i$ the set of variables; $\text{atoms}(X)$ the set of the atoms containing the variable X ; $\text{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of all atoms; and $\text{free}(Q) = \mathcal{O} \cup \mathcal{I} \subseteq \text{vars}(Q)$ the set of *free* variables, which are partitioned into *input* variables \mathcal{I} and *output* variables \mathcal{O} . An empty set of input or output variables is denoted by a dot (\cdot). All variables in $\text{vars}(Q) \setminus \text{free}(Q)$ are called *bound*. We call $R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n)$ the *body* of Q .

The hypergraph of a query Q is $\mathcal{H} = (\mathcal{V} = \text{vars}(Q), \mathcal{E} = \{\mathcal{X}_i \mid i \in [n]\})$, 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 fresh 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 $\text{atoms}(A) \subset \text{atoms}(B)$. 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 $R(\mathcal{X}) \in \text{atoms}(B)$, there is an atom $S(\mathcal{Y}) \in \text{atoms}(B)$, possibly different from $R(\mathcal{X})$, 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 $A, B \in \text{vars}(Q)$, either $\text{atoms}(A) \subseteq \text{atoms}(B)$, $\text{atoms}(B) \subseteq \text{atoms}(A)$, or $\text{atoms}(B) \cap \text{atoms}(A) = \emptyset$. The class of hypergraphs of hierarchical queries is strictly contained in the class of γ -acyclic (hence, α - and β -acyclic) hypergraphs. The class of Berge-acyclic hypergraphs and the class of hypergraphs of hierarchical queries are incomparable: there are Berge-acyclic queries that

²In this article, we use data complexity: The complexity factors that only depend on the query, such as the number of variables in a query atom and the number of query atoms, are considered constant.

are not hierarchical, e.g., $Q() = R(A), S(A, B), T(B)$, and hierarchical queries that are not Berge-acyclic, e.g., $Q() = R(A, B), S(A, B)$. For the precise definitions of these acyclicity notions, we refer to a recent overview [Bra16]. A query is q -hierarchical if it is hierarchical and free-dominant.

Definition 3.1. 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 [BKS17].

Example 3.2. The query $Q_0(B, C \mid \cdot) = R(A, B), S(A, C)$ is hierarchical and input-dominant. It is not free-dominant: The bound variable A dominates the free variables B and C .

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 $Q_1(A, C \mid B_1, B_2, D_1, D_2) = R(A, B_1), S(B_2, C), T(C, D_1), U(A, D_2)$ 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 variables. 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 $Q_2(A \mid B_1, B_2) = S(A, B_1), T(B_2)$ 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 $Q(\cdot \mid A) = R(A, B), S(B)$. \square

Query Semantics. We give the semantics of CQAPs using the function $\llbracket \cdot \rrbracket$ defined on the structure of CQAPs:

$$\begin{aligned} \llbracket (Q(\mathcal{O} \mid \mathcal{I}) = \text{body}) \rrbracket &= \{(t \mapsto m) \mid J = \llbracket \text{body} \rrbracket, (t_2 \mapsto m_2) \in J, t = t_2[\mathcal{O}], \text{in} = t_2[\mathcal{I}] \\ m &= \sum_{(t_1 \mapsto m_1) \in J, t = t_1[\mathcal{O}], \text{in} = t_1[\mathcal{I}]} m_1\} \end{aligned} \quad (3.2)$$

$$\begin{aligned} \llbracket Q_1(\mathcal{X}_1), Q_2(\mathcal{X}_2) \rrbracket &= \{(t \mapsto m) \mid (t_1 \mapsto m_1) \in \llbracket Q_1(\mathcal{X}_1) \rrbracket, (t_2 \mapsto m_2) \in \llbracket Q_2(\mathcal{X}_2) \rrbracket, \\ &\quad t \in \text{Dom}(\mathcal{X}_1 \cup \mathcal{X}_2), t_1 = t[\mathcal{X}_1], t_2 = t[\mathcal{X}_2], m = m_1 \cdot m_2\} \end{aligned} \quad (3.3)$$

$$\llbracket R(\mathcal{X}) \rrbracket = \{(t \mapsto m) \mid (t \mapsto m) \in R\} \quad (3.4)$$

Eq. (3.2) computes the set of mappings of the tuples over the output variables \mathcal{O} to their multiplicities under a specific tuple in of constants assigned to the input variables \mathcal{I} . It recursively invokes the semantics function applied to the body of the query. Eq. (3.3) computes the set of mappings $(t \mapsto m)$ defining the join of two subqueries Q_1 and Q_2 . The tuple t is the result of joining the tuple t_1 in the output of Q_1 and the tuple t_2 in the output of Q_2 , while its multiplicity m is the product of the multiplicities of t_1 and t_2 . Eq. (3.4) is the base case of one relation atom. Its semantics is the set of key-value mappings represented by the corresponding relation.

Delta Queries. Updates to input relations can change the query output. A delta query captures this change for updates to one input relation. The derivation of delta queries follows the standard delta rules [CY12]. Consider a CQAP as in Eq. (3.1) and an update δR_i to a relation R_i . If there is a single atom using the relation symbol R_i in Q , then the delta query expressing the change in the query output is:

$$\delta Q(\mathcal{O}|\mathcal{I}) = R_1(\mathcal{X}_1), \dots, R_{i-1}(\mathcal{X}_{i-1}), \delta R_i(\mathcal{X}_i), R_{i+1}(\mathcal{X}_{i+1}), \dots, R_n(\mathcal{X}_n)$$

If there are several atoms using R_i in Q , then we issue one delta query for each such atom.

4. VARIABLE ORDERS AND WIDTH MEASURES

In this section, we introduce the notions of variable orders and width measures for CQAPs.

4.1. Variable Orders. Variable orders are used as logical plans for the evaluation of conjunctive queries [OZ15]. 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*, or VO for short, ω for a CQAP Q is a pair (T_ω, dep_ω) , where:

- T_ω is a forest of (rooted) trees with one node per variable. For each atom $R(\mathcal{X})$ in Q , \mathcal{X} is a subset of the set of variables on a 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.

For convenience, we sometimes omit the index ω in (T_ω, dep_ω) when ω is clear from the context. A VO always exists for a query, e.g., by having all variables on a single path. In the remainder of this paper, we consider VOs in which atoms corresponding to relations and their indicator projections are added as new leaves. Each atom in the query is added as a child of its variable placed lowest in the VO. We explain next how the indicator projections are added to a VO ω . Indicator projections can reduce the asymptotic complexity of cyclic queries [KNR16].

Given a CQAP Q and a VO ω , where the atoms of Q have been already added, the function `indicators` in Figure 1 extends ω with indicator projections. It processes ω recursively in a bottom-up manner (Lines 1-2). At each variable X in ω , we compute the set \mathcal{I} of indicator projections (Line 4). Such indicator projections $I_{\mathcal{Y} \cap \mathcal{S}} R$ are for relations R whose atoms $R(\mathcal{Y})$ are not included in the subtree rooted at X but have schema \mathcal{Y} that shares a non-empty set of variables with $\mathcal{S} = \{X\} \cup dep_\omega(X)$. We choose from this set those indicators that together with the atoms in the subtree rooted at X form a cyclic query (Line 5). We achieve this using a variant of the GYO reduction [BFMY83]. 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 (α) -acyclic. Otherwise, the input hypergraph is cyclic and the GYO's output is a cyclic hypergraph. 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. This set of chosen indicator projections, which is empty if the input hypergraph is (α) -acyclic, are added as children of X (Line 6).

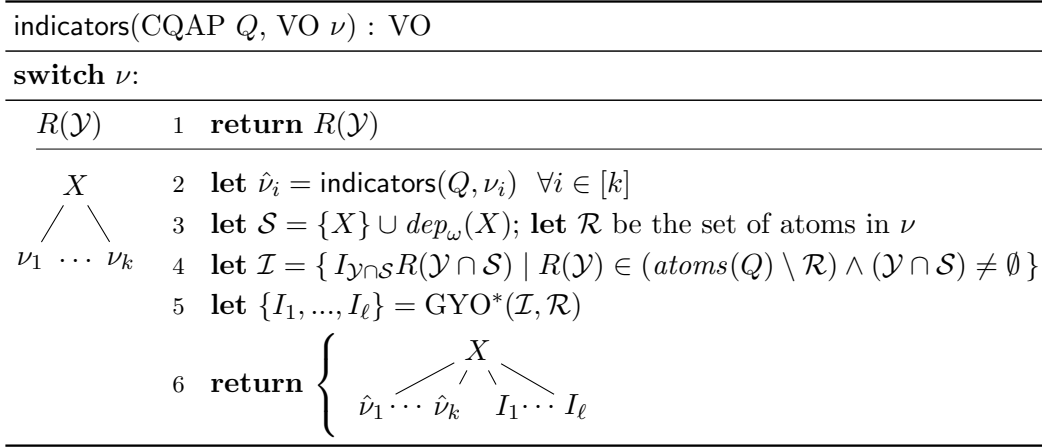


Figure 1: Extending a VO ω of a CQAP Q with indicator projections by calling $\text{indicators}(Q, \nu = \omega)$. The function indicators is defined using pattern matching (left column under **switch**) on the structure of ω , which can be a leaf (relation atom) or an inner node (query variable). Each 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 cyclic query with those that occur. GYO^* (defined in Section 4.1) is based on the GYO reduction [BFMY83].

The next proposition states that joining a query with the indicator projections constructed by the function indicators in Figure 1 does not change the result of the query. The proof of the proposition is in Appendix B.1 of the technical report [KNOZ25].

Proposition 4.1. *For any CQAP $Q(\mathcal{O}|\mathcal{I})$ and VO ω for Q , $Q(\mathcal{O}|\mathcal{I})$ is equivalent to a CQAP $Q'(\mathcal{O}|\mathcal{I})$, whose body is the conjunction of the atoms of Q and the indicator projections at the leaves of the VO returned by $\text{indicators}(Q, \omega)$.*

The following example illustrates the construction of indicator projections as described by the function indicators in Figure 1. Examples 4.9 and 4.10 show that indicator projections can reduce the preprocessing and update time for CQAPs.

Example 4.2. Consider the triangle CQAP

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

The fracture Q_\dagger of Q is the query itself. Figure 2 depicts a VO ω for Q . The input variable A is on top of the output variables B and C . The atoms $S(B, C)$ and $T(C, A)$ are included in the subtree of ω rooted at C but the atom $R(A, B)$ is not. We apply GYO^* to the atoms $S(B, C)$ and $T(C, A)$ and the indicator projection $I_{A,B}R(A, B)$ and obtain $\text{GYO}^*(\{I_{A,B}R(A, B)\}, \{S(B, C), T(C, A)\}) = \{I_{A,B}R(A, B)\}$, which means that the indicator projection $I_{A,B}R(A, B)$ and the atoms $S(B, C)$ and $T(C, A)$ form a cyclic query. For this reason, $I_{A,B}R(A, B)$ is added as a new child of C in ω . \square

For the following development, we need additional notation. Given a VO ω , its subtree rooted at X is denoted by ω_X . The sets $\text{vars}(\omega)$ and $\text{anc}_\omega(X)$ consist of all variables of ω and respectively the variables on the path from X to the root excluding X . We denote by $\text{atoms}(\omega)$ all atoms and indicators at the leaves of ω and by Q_X the query that is the join of all atoms $\text{atoms}(\omega_X)$ and where all variables are free.

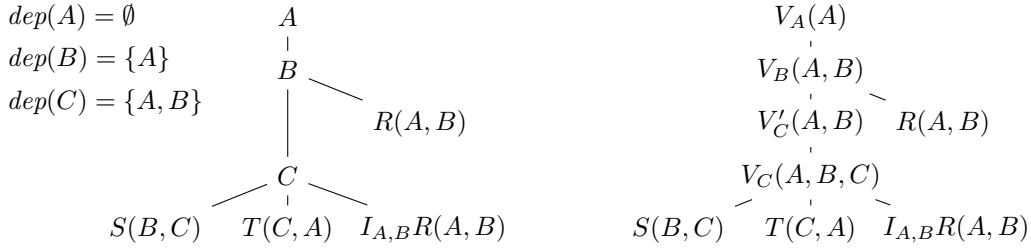


Figure 2: Left: (Access-top) VO for the query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. Right: The view tree constructed from this VO. Note the indicator $I_{A,B}R(A, B)$ added below the variable C (left) and below the view V_C (right).

We next introduce classes of VOs for CQAPs. A VO ω is *canonical* if the variables of the leaf atom of each root-to-leaf path are *exactly* 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 $\text{free-top}(Q)$ and $\text{input-top}(Q)$, respectively. A VO is called *access-top* if it is free-top and input-top³: $\text{acc-top}(Q) = \text{free-top}(Q) \cap \text{input-top}(Q)$.

Example 4.3. The query $Q(B|A) = R(A, B), S(B)$ admits the VO $B - \{A - R(A, B), S(B)\}$ (notation-wise, “-” represents the parent-child relationship), where the variable B has two children: the variable A and the atom $S(B)$; and the variable A has one child: the atom $R(A, B)$. 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 , the VO becomes $A - B - \{R(A, B), S(B)\}$ with the dependency sets $dep(A) = \emptyset$ 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 trivially 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 Q_1 in Example 3.2 admits the access-top VO consisting of the following 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 $dep(D_1) = \{B_2\}$. \square

³Although our approach in this work uses variable orders, it could also be phrased in terms of hypertree decompositions [GLS99], while preserving the same complexities. Every variable order ω can be translated into a hypertree decomposition by replacing each node X by a bag consisting of the variables $\{X\} \cup dep_\omega(X)$; conversely, every hypertree decomposition can be transformed into a variable order by replacing each bag B by a path that consists of all variables in B that do not appear in bags that are ancestor of B [OZ15]. An access-top variable order corresponds to a hypertree decomposition that contains a connected subtree consisting of all free variables and also a connected subtree consisting of the input variables. Since all input variables are free, the subtree consisting of the input variables must be subsumed by the subtree consisting of the free variables. For a discussion on the usefulness of variable orders for our approach, see Section 9.

4.2. Width Measures. Given a conjunctive query Q and $\mathcal{F} \subseteq \text{vars}(Q)$, a *fractional edge cover* of \mathcal{F} is a solution $\lambda = (\lambda_{R(\mathcal{X})})_{R(\mathcal{X}) \in \text{atoms}(Q)}$ to the following linear program [AGM13]:

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

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 $R(\mathcal{X}) \in \text{atoms}(Q)$. 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 $\rho_Q(\mathcal{F})$.

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

Lemma 4.4 (Lemma D.1 in [KNOZ20]). *For any hierarchical query Q and $\mathcal{F} \subseteq \text{vars}(Q)$, it holds $\rho^*(\mathcal{F}) = \rho(\mathcal{F})$.*

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 4.5. The static width $w(\omega)$ and dynamic width $\delta(\omega)$ of a VO ω are:

$$\begin{aligned} w(\omega) &= \max_{X \in \text{vars}(\omega)} \rho_{Q_X}^*(\{X\} \cup \text{dep}_\omega(X)) \\ \delta(\omega) &= \max_{X \in \text{vars}(\omega)} \max_{R(\mathcal{Y}) \in \text{atoms}(\omega_X)} \rho_{Q_X}^*((\{X\} \cup \text{dep}_\omega(X)) \setminus \mathcal{Y}) \end{aligned}$$

Q_X is the join of all atoms under X in the VO ω . For a query Q_X , the set of variables $\mathcal{X} = \{X\} \cup \text{dep}_\omega(X)$, and a database of size N , N^e is an upper bound on the worst-case output size of the query $Q_X(\mathcal{X})$, where $e = \rho_{Q_X}^*(\mathcal{X})$ is the fractional edge cover number of \mathcal{X} . The static width w of a VO ω is defined by the maximum over the fractional edge cover numbers of the queries Q_X for the variables X in ω . 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: $(\delta_1, w_1) \leq (\delta_2, w_2)$ if $\delta_1 < \delta_2$ or $\delta_1 = \delta_2$ and $w_1 \leq w_2$. Given a set \mathcal{S} of VOs, we define $\min_{\omega \in \mathcal{S}} (\delta(\omega), w(\omega)) = (\delta, w)$ such that $\forall \omega \in \mathcal{S} : (\delta(\omega), w(\omega)) \leq (\delta, w)$.

Definition 4.6. The dynamic width $\delta(Q)$ and static width $w(Q)$ of a CQAP Q are:

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

Since we are interested in dynamic evaluation, Definition 4.6 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.

Example 4.7. We show how to compute the widths for the variable order of the fractured 4-cycle query in Example 4.3: For the bag at variable A , we have $\rho^*(\{A\} \cup \text{dep}(A)) =$

$\rho^*({A, D_2, B_1}) = 2$, 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 $\{A\} \cup \text{dep}(A) = \{A, D_2, B_1\}$, 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. \square

Example 4.8. 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:

- $w = 1$ and $\delta = 0$ for $Q(C, D, E \mid A, B)$, $Q(A, B, C, D, E \mid \cdot)$, $Q(\cdot \mid A, B, C, D, E)$ and $Q(B, C, D, E \mid A)$;
- $w = 1$ and $\delta = 1$ for $Q(A, C, D, E \mid B)$;
- $w = 2$ and $\delta = 1$ for $Q(A, C, D \mid B, E)$;
- $w = 2$ and $\delta = 2$ for $Q(A, E \mid B, C, D)$;
- $w = 3$ and $\delta = 2$ for $Q(A, B \mid C, D, E)$.

The next example illustrates that the indicator projections constructed by the function indicators in Figure 1 can lower the static width of the VO of a query. Lower static width implies lower preprocessing time as stated in Theorem 5.1.

Example 4.9. Recall the triangle CQAP $Q(B, C \mid A) = R(A, B), S(B, C), T(C, A)$ from Example 4.2 and its access-top VO ω in Figure 2. The indicator projection $I_{A,B}R(A, B)$ is below C in ω , since the output of $\text{GYO}^*({I_{A,B}R(A, B)}, \{R(A, B), T(C, A)\})$ is $\{I_{A,B}R(A, B)\}$.

Assume first that $I_{A,B}R(A, B)$ is not included in ω . In this case, the query Q_C is defined as the join of $S(B, C)$ and $T(C, A)$, which means $\rho_{Q_C}^*({C} \cup \text{dep}(C)) = \rho_{Q_C}^*({A, B, C}) = 2$.

Assume now that $I_{A,B}R(A, B)$ is included in ω . In this case, the query Q_C is defined as the join of $S(B, C)$, $T(C, A)$, and $I_{A,B}R(A, B)$, which means $\rho_{Q_C}^*({C} \cup \text{dep}(C)) = \rho_{Q_C}^*({A, B, C}) = \frac{3}{2}$. Hence, the fractional edge cover number reduces from 2 to $\frac{3}{2}$. This fractional edge cover number dominates the static width of ω , so the static width of ω is $\frac{3}{2}$.

The dynamic width of ω (including $I_{A,B}R$) is dominated by the fractional edge cover number $\rho_{Q_C}^*({C} \cup \text{dep}(C)) - \mathcal{S} = \rho_{Q_C}^*({A, B, C}) - \mathcal{S}$, 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, therefore, the dynamic width of ω is 1. \square

The next example demonstrates that indicator projections are inevitable when we want to construct VOs with minimal dynamic width for cyclic CQAPs. As stated in Theorem 5.1, low dynamic width implies low update time for CQAPs.

Example 4.10. Consider the following query:

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

It is a triangle query with three tails. The fracture of the query is the same as the query. Figure 3 shows the hypergraph (top-left) of the query and three access-top VOs for the query. No other VO for the query has better static or dynamic width than these VOs.

Consider the VO in the top right of Figure 3. The subtree of the variable order rooted at C has the atoms $\mathcal{R} = \{R_3(C, A), R_2(B, C), R_8(C, H), R_9(H, J), R_5(D, E), R_4(A, D)\}$. The

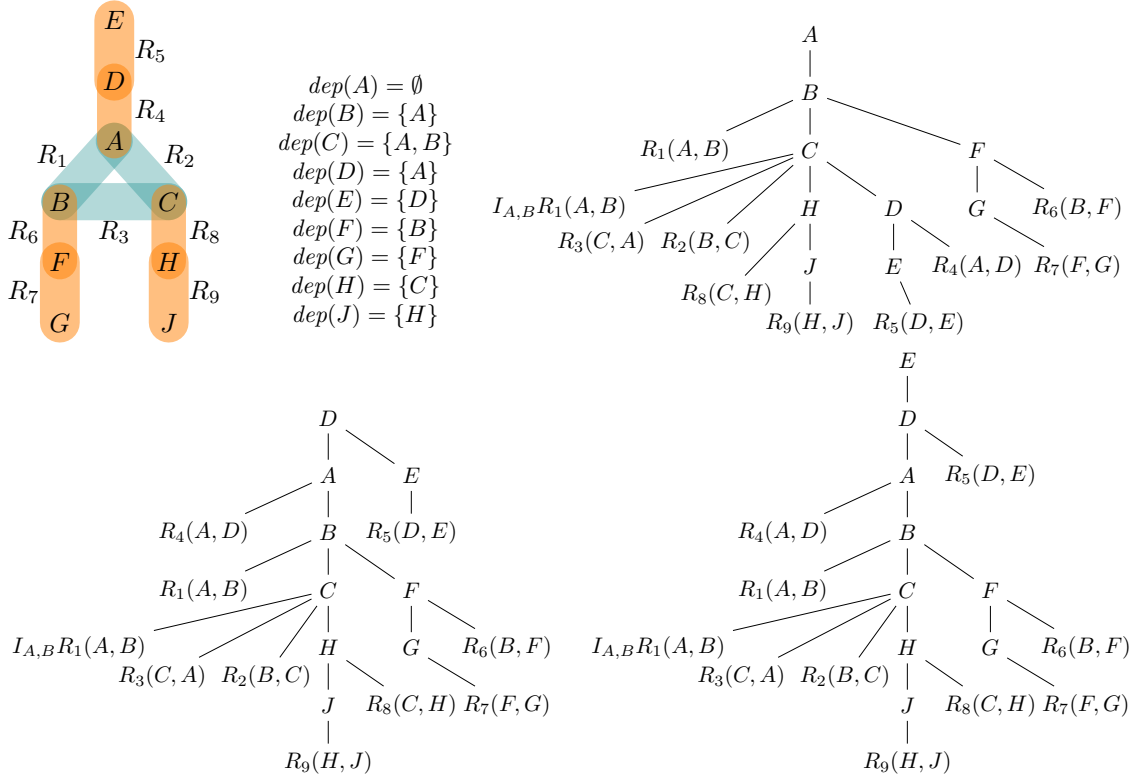


Figure 3: Top left: The hypergraph of the query Q in Example 4.10. Remaining three: the optimal access-top VOs of the query Q with the roots A , D and E , respectively. All other access-top VOs are analogous to these three VOs. The dependent sets of the two VOs in the second row are omitted.

atoms that are not in the subtree but whose schemas intersect with $\{C\} \cup dep(C) = \{A, B, C\}$ are $R_1(A, B)$ and $R_6(B, F)$. Hence, we consider the indicator projections $\mathcal{I} = \{I_{A,B}R_1(A, B), I_B R_6(B, F)\}$. We have $GYO^*(\mathcal{I}, \mathcal{R}) = \{I_{A,B}R_1(A, B)\}$. Therefore, $I_{A,B}R_1(A, B)$ becomes a child of C . This indicator projection reduces the dynamic width of the variable order from 2 to $\frac{3}{2}$, as explained next.

It holds $\{C\} \cup dep(C) = \{A, B, C\}$. We compute the maximal $\rho^*(\{A, B, C\} \setminus \mathcal{S})$, where \mathcal{S} is the schema of any atom in the subtree of the variable order rooted at C . If we choose \mathcal{S} to be the schema of $R_9(H, J)$, we obtain $\{A, B, C\} \setminus \mathcal{S} = \{A, B, C\}$. In case $I_{A,B}R_1$ is not included in the subtree rooted at C , we have $\rho^*(\{A, B, C\} \setminus \mathcal{S}) = 2$. Otherwise, we have $\rho^*(\{A, B, C\} \setminus \mathcal{S}) = \frac{3}{2}$ (by assigning a weight of $\frac{1}{2}$ to the indicator projection $I_{A,B}R_1$ and to each of the atoms R_3 and R_2). For any other variable X and atom $R(\mathcal{S})$ below X , the fractional edge cover number $\rho^*((\{X\} \cup dep(X)) \setminus \mathcal{S})$ is not greater than 1. Hence, we conclude that the dynamic width of the VO is $\frac{3}{2}$.

The two VOs in the second row of Figure 3 are similar to the aforementioned VO: Each of them has the variables A , B , and C on one root-to-leaf path, followed by the atom R_9 , which has no intersection with $\{A, B, C\}$. 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. \square

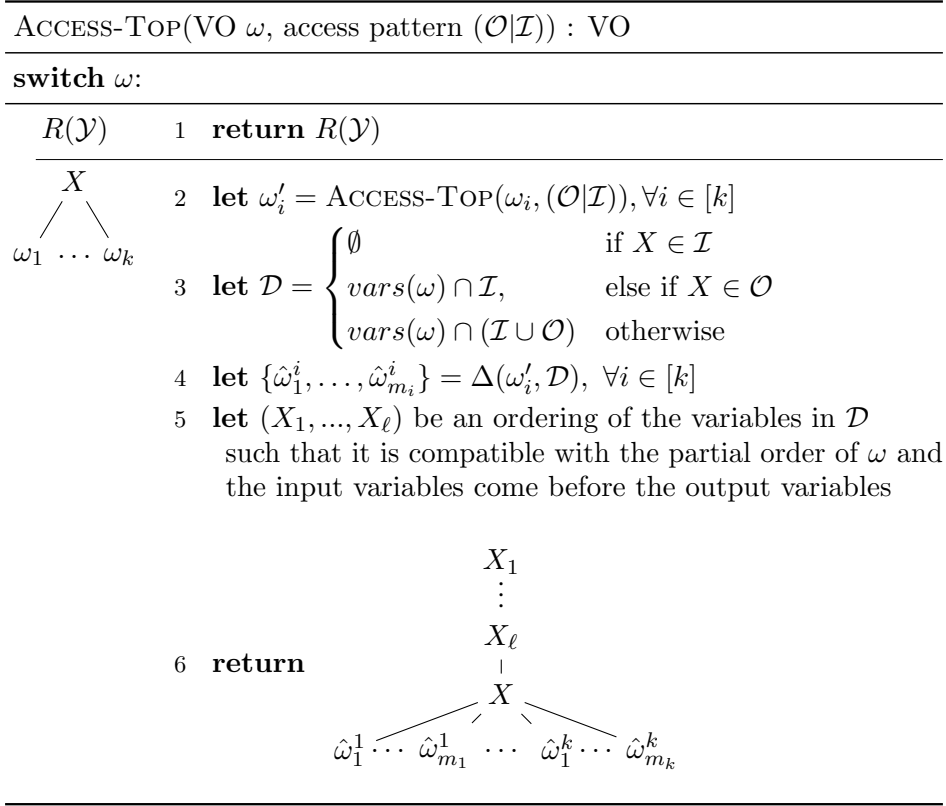


Figure 4: Construction of an access-top VO from a canonical VO ω of a hierarchical CQAP with access pattern $(\mathcal{O}|\mathcal{I})$. The function $\Delta(\omega', \mathcal{D})$, defined in Figure 5, deletes the variables in \mathcal{D} from the VO ω' .

Prior work defined the static and the dynamic width of conjunctive queries without access patterns [KNOZ20]. It was shown that for any hierarchical conjunctive query with static width w and dynamic width δ , it holds $\delta = w$ or $\delta = w - 1$ (Proposition 3.7 in [KNOZ20]). 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 4.11 (Corollary of Proposition 3.7 in [KNOZ20]). *For any CQAP with hierarchical fracture, static width w and dynamic width δ , it holds either $\delta = w$ or $\delta = w - 1$.*

4.3. 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 $\text{ACCESS-TOP}(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 4 returns an access-top VO for Q whose static and dynamic widths equal the corresponding widths of Q .

First, we give the high-level idea of the construction. At each variable X , the function *pulls up* some variables from the subtree rooted at X , which means that it deletes these variables from the subtree and puts them on a path on top of X . If X is an output variable, all input variables in the subtree are pulled up. If it is a bound variable, all free variables in

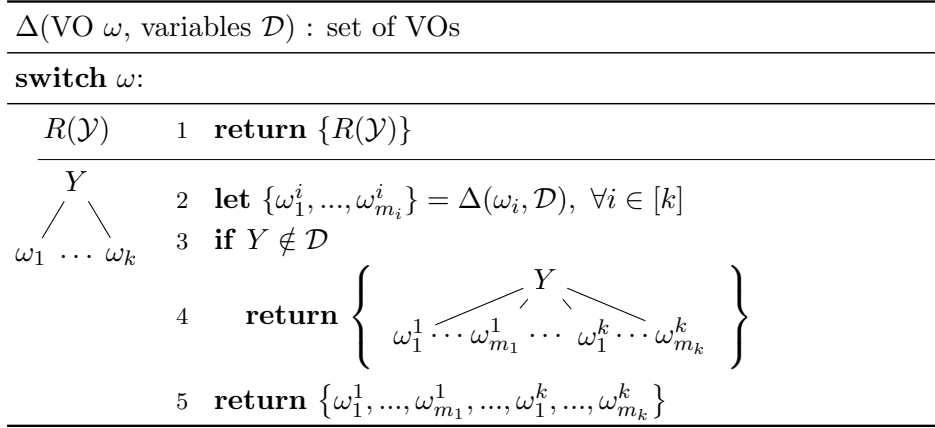


Figure 5: Deletion of a set \mathcal{D} of variables from a VO ω . In case ω has a root variable Y , the variables in \mathcal{D} are first deleted from the child trees of Y . If Y is included in \mathcal{D} , the child trees of Y become a forest of trees without any common root.

the subtree are pulled up. In the newly constructed path on top of X , the input variables are placed on top of the output variables.

We explain the function $\text{ACCESS-TOP}(\omega, (\mathcal{O}|\mathcal{I}))$ in more detail. It proceeds recursively on the structure of ω . Consider a variable X in ω and assume that the child trees of X are already access-top. The function defines a set \mathcal{D} of variables (Line 3) that are going to be deleted from the subtree ω_X rooted at X and put on a path on top of X . If X is an input variable (Case 1 in Line 3), then \mathcal{D} is empty, which means that we do not need to pull up any variable from ω_X . If X is an output variable (Case 2 in Line 3), then \mathcal{D} consists of all input variables in ω_X . If X is a bound variable (Case 3 in Line 3), then \mathcal{D} consists of all free variables in ω_X . The deletion of the variables in \mathcal{D} from ω_X (Line 4) is implemented by the function Δ in Figure 5, which we explain in more detail further below. The top-down ordering of the new path constructed from the variables in \mathcal{D} respects the partial ordering defined by ω_X and has the input variables on top of the output variables (Line 5). Observe that this is possible, since the child trees of X are already access-top.

Given a variable order ω' and a set \mathcal{D} of variables to be deleted from ω' , the function $\Delta(\omega', \mathcal{D})$ in Figure 5 traverses recursively over each variable Y in ω' with child trees $\omega_1, \dots, \omega_k$. First, the function deletes the variables in \mathcal{D} from the child trees of Y and obtains the trees $\mathcal{T} = \{\Delta(\omega_1, \mathcal{D}), \dots, \Delta(\omega_k, \mathcal{D})\}$ (Line 2). If Y is not included in \mathcal{D} , the function returns the tree with root Y and child trees \mathcal{T} (Lines 3-4). Otherwise, it returns the forest \mathcal{T} (Line 5).

Example 4.12. Consider the query

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

which is hierarchical but not free-dominant. Figure 6 shows the hypergraph and the canonical VO ω of the query (top row). We explain an intermediate and the final step of the function $\text{ACCESS-TOP}(\omega(\{C, D\}|\{E\}))$ in Figure 4 that transforms ω into an access-top VO.

At variable B in ω , the function determines that B is bound and its two children are free. Hence, the function moves C and D on a path above B . Figure 4 (bottom row, left) shows the VO ω' that arises from this step. At variable A in ω' , the function determines that A is bound and the children C , D , and E are free. Thus, it puts the latter variables on a path on top of A such that the input variable E sits on top of the output variables C and

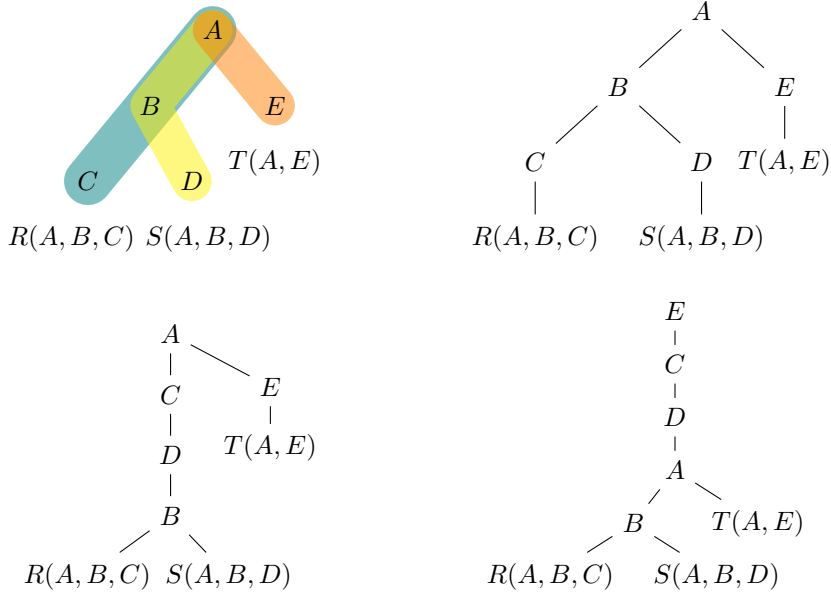


Figure 6: Top row: Hypergraph of the query from Example 4.12 and its canonical VO ω . Bottom row: An intermediate and the final VO constructed by the function $\text{ACCESS-TOP}(\omega, (\{C, D\}, \{E\}))$ in Figure 4.

D. Figure 4 (bottom row, right) shows the resulting access-top VO, which is the final VO returned by the function $\text{ACCESS-TOP}(\omega(\{C, D\}|\{E\}))$. \square

The function ACCESS-TOP in Figure 4 turns canonical VOs into optimal VOs. The proof of the following proposition is given in Appendix B.2 of the technical report [KNOZ25].

Proposition 4.13. *Given a CQAP Q , whose fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I})$ is hierarchical, and a canonical VO ω for Q_{\dagger} , $\text{ACCESS-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top VO for Q_{\dagger} with static width $w(Q)$ and dynamic width $\delta(Q)$.*

5. COMPLEXITY OF DYNAMIC CQAP EVALUATION

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

Theorem 5.1. *Given a CQAP with static width w and dynamic width δ and a database of size N , the query can be evaluated with $\mathcal{O}(N^w)$ 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 explained in Sections 6, 7, and 8, respectively. Given a CQAP with static width w and dynamic width δ and a database of size N , we construct in the preprocessing stage a set of view trees in $\mathcal{O}(N^w)$ time that represent the result of the query (Proposition 6.4). Using these view trees, we can enumerate with constant delay the tuples over the output variables, given any tuple over the input variables (Proposition 7.2). The view trees can be maintained

with $\mathcal{O}(N^\delta)$ update time under single-tuple updates to the base relations (Proposition 8.2). The full proof of Theorem 5.1 is given in Appendix F of the technical report [KNOZ25].

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 5.2. *Let any CQAP 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 prove Theorem 5.2 in Section 10. The hardness result in the theorem is based on the following OMv problem:

Definition 5.3 (Online Matrix-Vector Multiplication (OMv) [HKNS15]). We are given an $n \times n$ Boolean matrix \mathbf{M} and receive n Boolean column vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ 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 5.4 (OMv Conjecture, Theorem 2.4 [HKNS15]). 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 10.2). Our approach from Theorem 5.1 achieves linear preprocessing time, constant update time and enumeration delay for such queries, 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 of these queries 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 [BKS17]. To prove the hardness statement in Theorem 5.2, we show in Section 10 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 .

For CQAPs with hierarchical fractures, the complexities in Theorem 5.1 can be parameterised to uncover trade-offs between preprocessing, update, and enumeration.

Theorem 5.5. *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 $\mathcal{O}(N^{1+(w-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

We illustrate in Section 11 the core ideas of our algorithm achieving the trade-offs in Theorem 5.5. The full proof of the theorem can be found in Appendix F of the technical report [KNOZ25]. The trade-off continuum in Theorem 5.5 can be obtained using one algorithm parameterised by ϵ . In Section 11.5, we show that this algorithm either recovers or has lower complexity than prior approaches. Using $\epsilon = 1$, we recover the complexities in Theorem 5.1 and therefore also the constant update time and delay for queries in CQAP_0 in Theorem 5.2.

Theorem 5.5 can be refined for CQAP_1 , since $\delta = 1$ and $w \leq 2$ for queries in this class.

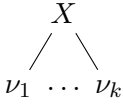
$\tau(\text{VO } \nu) : \text{view tree}$	
switch ν :	
$R(\mathcal{Y})$	1 return $R(\mathcal{Y})$
	2 let $T_i = \tau(\nu_i), \forall i \in [k]$ 3 let $\mathcal{S} = \{X\} \cup \text{dep}_\omega(X)$ 4 let $V_X(\mathcal{S}) = \text{join of roots of } T_1, \dots, T_k \text{ with variables not in } \mathcal{S} \text{ marginalised out}$ 5 if X has no sibling return $\begin{cases} V_X(\mathcal{S}) \\ T_1 \dots T_k \end{cases}$ 6 let $V'_X(\mathcal{S} \setminus \{X\}) = V_X(\mathcal{S})$ with variable X marginalised out 7 return $\begin{cases} V'_X(\mathcal{S} \setminus \{X\}) \\ V_X(\mathcal{S}) \\ T_1 \dots T_k \end{cases}$

Figure 7: The function τ constructs a view tree for a given VO ω . It is defined using pattern matching on the structure of ω , which can be a leaf or an inner node (cf. left column under **switch**). At each variable X in ω , the function defines a new view V_X whose free variables \mathcal{S} are X and the dependency set of X ; its body is the join of the views defined at the variables that are roots of the child VOs of X . If X has siblings, it defines a new view on top of V_X so that X becomes bound in V_X (so it is marginalised). Note that when X has an atom $R(\mathcal{S})$ as its only child in ω , the new view $V_X(\mathcal{S})$ is redundant; for simplicity, we retain this view.

Corollary 5.6 (Theorem 5.5). *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.*

The proof of the corollary is given in Appendix F of the technical report [KNOZ25]. For $\epsilon = 0.5$, the amortised update time and the delay for queries in CQAP_1 match the lower bound in Theorem 5.2 for all queries outside CQAP_0 . This makes our approach weakly Pareto optimal for CQAP_1 , as lowering both the amortised update time and the delay would violate the OMv conjecture.

6. PREPROCESSING

In this section, we describe the preprocessing stage of our approach for the dynamic evaluation of arbitrary CQAPs. Consider in the following a CQAP Q , its fracture Q_\dagger , and a database of size N .

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 [NO18] is a (rooted) tree with one view per node. It is a logical project-join plan in the classical database systems literature,

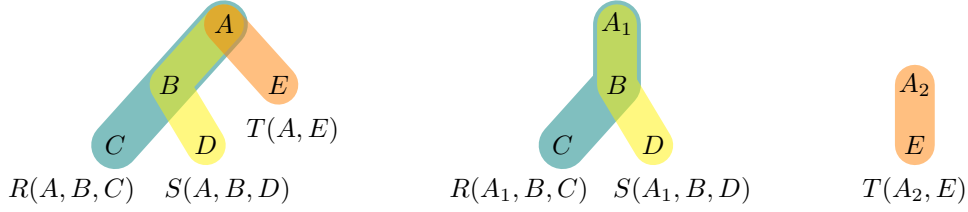


Figure 8: (Left) Hypergraph of the two queries with the same body but different access patterns, as used in Examples 6.1 and 6.2; (middle and right) hypergraph of their fractures.

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 for Q_{\dagger} , the function $\tau(\omega)$ in Figure 7 returns a view tree constructed from ω . The function recursively traverses ω bottom-up (Line 2) and creates at each variable X , a view V_X defined over the join of the views created for the children 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 to aggregate away (or marginalise out) X from V_X (Line 6). This view allows to efficiently maintain the ancestor views of V_X under updates to the views created for the siblings of X .

The next example demonstrates the construction of the view trees for a query in CQAP₀. The construction time is linear in the database size.

Example 6.1. Figure 8 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 9 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 $\{B, A_1\} = \{B\} \cup \text{dep}(B)$. 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. \square

The next example considers a query in CQAP₁ where the view tree construction time is quadratic in the database size.

Example 6.2. Consider the query $Q(E, D | A, C) = R(A, B, C), S(A, B, D), T(A, E)$ in CQAP₁ 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 8) of Q and its fracture are the same as for the query in Example 6.1. Figure 10 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

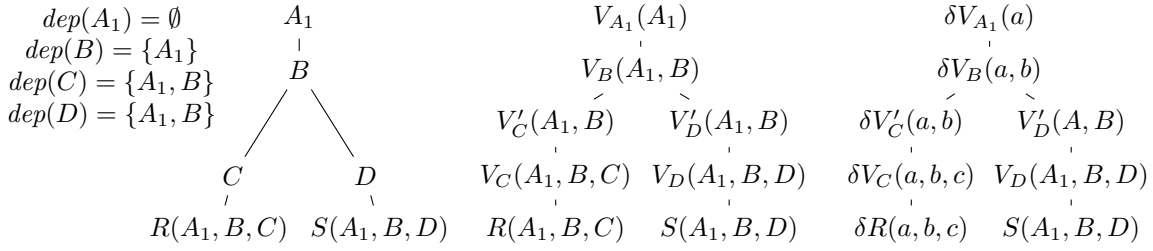


Figure 9: (Left) Access-top VO for $Q_1(B, C, D | A_1) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree constructed from the VO; (right) the delta view tree under a single-tuple update to R .

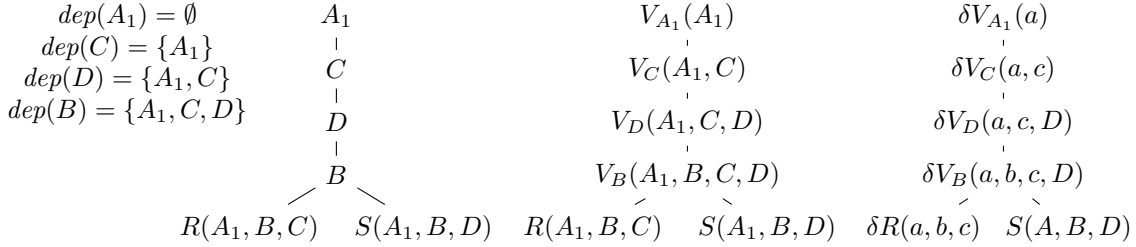


Figure 10: (Left) Access-top VO for $Q_1(B, D | A_1, C) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree corresponding to the VO; (right) the delta view tree under a single-tuple update to R .

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 6.1 and can be constructed in linear time. \square

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

Example 6.3. Figure 2 depicts a VO and the view tree constructed from it for the triangle CQAP $Q(B, C | A) = R(A, B), S(B, C), T(C, A)$ from Example 4.2. 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. \square

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 $\mathcal{O}(N^w)$ time, where $w = \rho_{Q_X}^*(\{X\} \cup dep_\omega(X))$. The definition of the static width of ω implies that the view tree $\tau(\omega)$ can be constructed in $\mathcal{O}(N^{w(\omega)})$ time, as stated in the next proposition. By choosing an access-top VO ω for Q_\dagger with $w(\omega) = w(Q)$, we obtain the preprocessing time from Theorem 5.1.

Proposition 6.4. *Given a VO ω of static width w and a database of size N , the view tree $\tau(\omega)$ can be constructed in $\mathcal{O}(N^w)$ time.*

Proof. Consider a CQAP Q , a VO ω for Q with static width $w(\omega) = w$, and a database of size N . 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 . We show by induction on the structure of $T = \tau(\omega)$ that every node in T can be materialised in $\mathcal{O}(N^w)$ time, where τ is the procedure given in Figure 7.

Base Case: Each leaf atom or indicator projection in T can be materialised in $\mathcal{O}(N)$ time. Since $w \geq 1$, the complexity bound holds in the base case.

Induction Step: Consider an auxiliary view $V'_X(\mathcal{S}')$ in T with $X \in \text{vars}(\omega)$ and $\mathcal{S}' = \text{dep}_\omega(X)$. By construction, this view results from its single child view $V_X(\mathcal{S} \cup \{X\})$ by marginalising out variable X . By induction hypothesis, the view V_X can be computed in $\mathcal{O}(N^w)$ time, hence its size has the same asymptotic 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 $\mathcal{O}(N^w)$ overall time.

Consider now a view $V_X(\mathcal{S})$ in T with $X \in \text{vars}(\omega)$ and $\mathcal{S} = \{X\} \cup \text{dep}_\omega(X)$. Let $V_{X_1}(\mathcal{S}_1), \dots, V_{X_k}(\mathcal{S}_k)$ be the child nodes of $V_X(\mathcal{S})$. Each child node can be a view, an atom, or an indicator projection. By induction hypothesis, the child nodes of $V_X(\mathcal{S})$ can be materialised in $\mathcal{O}(N^w)$ time.

Consider any variable Y that occurs in the schemas of at least two child nodes of $V_X(\mathcal{S})$. It follows from the construction of view trees that $Y \in \mathcal{S} = \{X\} \cup \text{dep}_\omega(X)$: Consider two child views $V_{X_i}(\mathcal{S}_i)$ and $V_{X_j}(\mathcal{S}_j)$ of $V_X(\mathcal{S})$ such that $Y \in \mathcal{S}_i \cap \mathcal{S}_j$ and the variables X_i and X_j are children of X in ω . The two views V_{X_i} and V_{X_j} are siblings, so they are constructed as in Line 6 of τ , which means that $\mathcal{S}_i = \text{dep}_\omega(X_i)$ and $\mathcal{S}_j = \text{dep}_\omega(X_j)$. Thus $Y \in \text{dep}_\omega(X_i) \cap \text{dep}_\omega(X_j)$. Since Y must be a common ancestor of X_i and X_j in ω , Y is either X or an ancestor of X that is in the dependency set of X . Hence, Y is in $\mathcal{S} = \{X\} \cup \text{dep}_\omega(X)$.

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 $\mathcal{O}(N^w)$ time. Let $V'_1(\mathcal{S}'_1), \dots, V'_k(\mathcal{S}'_k)$ be the resulting views. The view $V_X(\mathcal{S})$ can now be written as $V_X(\mathcal{S}) = V'_1(\mathcal{S}'_1), \dots, V'_k(\mathcal{S}'_k)$, where $\bigcup_{i=1}^k \mathcal{S}'_i = \mathcal{S}$. We use a worst-case optimal join algorithm to compute the view V_X . The size of V_X is upper-bounded by $\mathcal{O}(N^p)$ where $p = \rho_{Q_X}^*(\mathcal{S})$ and Q_X is the query that joins all atoms and indicator projections in ω_X [NRR13]. By definition of w , the quantity p is upper-bounded by w . This means that the view V_X can be computed in $\mathcal{O}(N^w)$ time [NPRR18].

Overall, we conclude that the desired complexity bound holds for the induction step. \square

7. ENUMERATION

We describe the enumeration procedure of our approach for the dynamic evaluation of arbitrary CQAPs. Consider a CQAP $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_\dagger(\mathcal{O}|\mathcal{I}_\dagger)$, and an access-top VO ω for Q_\dagger . Recall from Section 6 that in the preprocessing stage, our approach uses the procedure τ in Figure 7 to construct view trees T_j following ω for the connected components $Q_j(\mathcal{O}_j|\mathcal{I}_j)$ of the fracture Q_\dagger , as explained in Section 6. These view trees are maintained under updates (Section 8). Consider an input tuple \mathbf{i} over \mathcal{I} for Q . We enumerate the output tuples of each $Q_j(\mathcal{O}_j|\mathcal{I}_j)$ and concatenate them to obtain the output tuples of $Q(\mathcal{O}|\mathcal{I})$.

We first describe the enumeration for a single connected component $Q_j(\mathcal{O}_j|\mathcal{I}_j)$. We enumerate tuples over \mathcal{O}_j for the input tuple $\mathbf{i}_j = \mathbf{i}[\mathcal{I}_j]$ over \mathcal{I}_j from the view tree T_j . We traverse in preorder the views in T_j that are constructed for the free variables of Q_j . At each view $V_X(\mathcal{S})$, we do the following: If X is an input variable, we check if $\mathbf{i}_j[\mathcal{S}]$ is in V_X ; if $\mathbf{i}_j[\mathcal{S}]$ is not in V_X , this means that $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ is empty, so we stop. Otherwise, we continue with the traversal. If X is an output variable, we retrieve from V_X an X -value that is paired the values in \mathbf{i}_j and the values retrieved from the views above V_X . Once all these views are visited, we report the tuple consisting of the retrieved values. Reporting each tuple takes constant time, since lookup and retrieval of values are constant-time operations as discussed in Section 2.

The tuples in $Q(\mathcal{O}|\mathbf{i})$ are the Cartesian product of the tuples in $Q_1(\mathcal{O}_1|\mathbf{i}_1), \dots, Q_n(\mathcal{O}_n|\mathbf{i}_n)$. We enumerate the tuples in $Q(\mathcal{O}|\mathbf{i})$ by interleaving the enumeration procedures for $Q_1(\mathcal{O}_1|\mathbf{i}_1), \dots, Q_n(\mathcal{O}_n|\mathbf{i}_n)$. This gives us a constant-delay enumeration procedure for Q . We demonstrate the enumeration procedure in the following example.

Example 7.1. Consider the query $Q(B, C, D, E|A)$ from Example 6.1 and the two connected components $Q_1(B, C, D|A_1)$ and $Q_2(E|A_2)$ of its fracture. Figure 9 (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. \square

The following proposition states that our approach achieves constant-delay enumeration of the tuples in the query output. This matches the enumeration delay stated in Theorem 5.1.

Proposition 7.2. *Let $Q(\mathcal{O}|\mathcal{I})$ be a CQAP and ω an access-top VO ω for the fracture of Q that consists of the trees $(\omega_j)_{j \in [n]}$. Given any tuple \mathbf{i} over \mathcal{I} , the tuples in $Q(\mathcal{O}|\mathbf{i})$ can be enumerated from the view trees $(\tau(\omega_j))_{j \in [n]}$ with constant delay.*

Proof. Consider a CQAP $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$, and an access-top VO ω for Q_{\dagger} . Assume that ω consists of the trees $\omega_1, \dots, \omega_n$ and let $T_1 = \tau(\omega_1), \dots, T_n = \tau(\omega_n)$ be the view trees constructed by the procedure τ in Figure 7. 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 T_1, \dots, T_n .

For $j \in [n]$, let $Q_j(\mathcal{O}_j|\mathcal{I}_j)$ with $\mathcal{O}_j = \mathcal{O} \cap \text{vars}(\omega_j)$ and $\mathcal{I}_j = \mathcal{I}_{\dagger} \cap \text{vars}(\omega_j)$ 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, there

is no view V_Y with Y being bound (output) that is above a view V_X with X being free (input). 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, $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ 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 a first X -value in $\sigma_{\mathcal{S}=\mathbf{t}'}V_X$, where \mathcal{S} is the schema of V_X without 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.

```

1  foreach  $\mathbf{o}_1 \in Q_1(\mathcal{O}_1|\mathbf{i}_1)$ 
2      ...
3      foreach  $\mathbf{o}_n \in Q_n(\mathcal{O}_n|\mathbf{i}_n)$ 
4          report  $\mathbf{o}_1 \cdots \mathbf{o}_n$ 

```

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 signal the end of the enumeration after the last tuple is reported is constant. \square

8. UPDATES

In this section, we explain how our approach maintains the view trees constructed in the preprocessing stage under single-tuple updates to the base relations.

Consider a CQAP Q , an access-top VO ω for the fracture Q_\dagger and the view trees T_1, \dots, T_n constructed from ω by the procedure τ in Figure 7. Let $\delta R = \{\mathbf{x} \rightarrow m\}$ be a single-tuple update to an input relation R ; m is positive in case of insertion and negative in case of deletion. We first update each view tree T_j 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 standard delta rules [CY12]. The update δR may also trigger single-tuple updates to indicator projections $I_{\mathcal{Z}}R$, as discussed in Section 2. These updates to indicators are propagated up to the root of each view tree, like for δR .

Example 8.1. Figure 9 (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 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 10 (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 $O(N)$ 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 $O(1)$ time. Updates to S are handled analogously. Overall, maintaining the view tree under a single-tuple update to any relation takes $O(N)$ time. \square

The following proposition states the time to maintain the view trees under single-tuple updates to the base relations. This matches the update time in Theorem 5.1.

Proposition 8.2. *Given a VO ω consisting of the trees $(\omega_j)_{j \in [n]}$ and a database of size N , the view trees $(\tau(\omega_j))_{j \in [n]}$ can be maintained under single-tuple updates to the base relations with $O(N^{\delta(\omega)})$ update time.*

Proof. Consider a VO ω that consists of the trees $(\omega_j)_{j \in [n]}$ and a database of size N . Let $(T_j = \tau(\omega_j))_{j \in [n]}$ be the view trees constructed by the procedure τ in Figure 7. We show that the view trees can be maintained with $O(N^{\delta(\omega)})$ update 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 T_j referring to an atom of the form $R(\mathcal{X})$. Updating T_j 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 in the proof of Proposition 6.4 that for each variable X in ω , the views V_X and V'_X can be materialised in $O(N^p)$ time, where $p = \rho_{Q_X}^*(\{X\} \cup \text{dep}_\omega(X))$. Since the update fixes the values in \mathcal{X} , the time to compute the delta of these views under the update becomes $O(N^d)$, where $d = \rho_{Q_X}^*((\{X\} \cup \text{dep}_\omega(X)) \setminus \mathcal{X})$. A single-tuple update to R can trigger a single-tuple update to each indicator view of the form $I_{\mathcal{Z}}(R(\mathcal{Z}))$. Following a similar argument as above, we conclude that the time to compute the deltas of the views under such updates is $O(N^{d'})$, where $d' = \rho_{Q_X}^*((\{X\} \cup \text{dep}_\omega(X)) \setminus \mathcal{Z})$. It follows from the definition of the dynamic width of VOs that the exponents d and d' are upper-bounded by $\delta(\omega)$. This implies that the overall update time is $O(N^{\delta(\omega)})$. \square

9. DISCUSSION OF OUR APPROACH

Sections 6-8 explain our approach to evaluating arbitrary CQAPs. We next discuss key decisions behind our approach.

1. Variable orders. Our approach can be rephrased to use hypertree decompositions [GLS99] instead of VOs, since they are different syntaxes for the same query decomposition class [OZ15]. Indeed, the set consisting of a variable and its dependency set in a VO can be interpreted as a bag of a hypertree 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 [Vel14] and their use for constructing factorised representations of query results [OZ15]: 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 5.1, as explained next. The maintenance procedure for view trees ensures that each view is calibrated⁴ 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 6.3: $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. 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

⁴A 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 .

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 6.3.

10. A DICHOTOMY FOR CQAPs

In this section, we prove our dichotomy result in Theorem 5.2, which states that the queries in the class CQAP_0 are precisely those queries that can be evaluated with constant update time and enumeration delay:

Theorem 5.2. *Let any CQAP 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 OMv conjecture was introduced in Section 5 (Conjecture 5.4). Before proving Theorem 5.2, we introduce an auxiliary lemma and a proposition.

The next lemma states that the evaluation complexity of the fracture Q_{\dagger} of a CQAP Q is upper-bounded by the evaluation complexity of Q .

Lemma 10.1. *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 for databases of size N and some functions f_p , f_e , and f_u , then the fracture Q_{\dagger} can be evaluated with the same asymptotic complexities.*

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 Q_1, \dots, Q_n be the connected components of Q_{\dagger} and C_1, \dots, C_n sets of database relations such that each C_i consists of the relations that are referenced in Q_i . We create from \mathcal{D} the databases $\mathcal{D}_1, \dots, \mathcal{D}_n$, where each \mathcal{D}_i is constructed as follows. The database \mathcal{D}_i contains each relation R in \mathcal{D} , modified as follows: (1) If $R \in C_i$ and R has a variable A in its schema that is a representative of a variable A' , then 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 $\mathcal{D}_1, \dots, \mathcal{D}_n$ 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} over \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 $\mathcal{D}_1, \dots, \mathcal{D}_n$. 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 $\times_{i \in [n]} \pi_{\mathcal{O}_i}(Q(\mathcal{O}|\mathbf{t}_i), \mathcal{D}_i)$, where \mathcal{O}_i is the set of output variables of Q contained in C_i . Now, assume that we want to enumerate the tuples in $(Q_{\dagger}(\mathcal{O}|\mathbf{t}_{\dagger}), \mathcal{D})$. We start the enumeration procedure for each $Q(\mathcal{O}|\mathbf{t}_i), \mathcal{D}_i$ with $i \in [n]$. For each $\mathbf{t}'_1 \in Q(\mathcal{O}|\mathbf{t}_1), \mathcal{D}_1$, \dots , $\mathbf{t}'_n \in Q(\mathcal{O}|\mathbf{t}_n), \mathcal{D}_n$, we return the tuple $\pi_{\mathcal{O}_1} \mathbf{t}'_1 \circ \dots \circ \pi_{\mathcal{O}_n} \mathbf{t}'_n$. Hence, the tuples in $(Q_{\dagger}(\mathcal{O}|\mathbf{t}_{\dagger}), \mathcal{D})$ 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 $\mathcal{D}_1, \dots, \mathcal{D}_n$, 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 with $i \in [n]$. 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. \square

The next proposition is essential for the complexity upper bound in Theorem 5.2.

Proposition 10.2. *Every query in CQAP_0 has dynamic width 0 and static width 1.*

Proof. Consider a query Q in CQAP_0 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 ; and \mathcal{Y} contains X and all ancestor variables of X . Hence, we have $\rho_{Q_X}^*((\{X\} \cup \text{dep}_{\omega}(X)) \setminus \mathcal{Y}) = \rho_{Q_X}^*((\{X\} \cup \text{anc}_{\omega}(X)) \setminus \mathcal{Y}) = \rho_{Q_X}^*(\emptyset) = 0$. 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 4.11 that the static width of Q is 1⁵. \square

In the following, we first prove the complexity upper bound and then the complexity lower bound stated in Theorem 5.2.

10.1. Complexity Upper Bound. We prove the first statement in Theorem 5.2. Assume that Q is in CQAP_0 . By Proposition 10.2, Q 's dynamic width is 0. By the definition of CQAP_0 , the fracture Q_{\dagger} must be hierarchical. From Proposition 4.11, the static width of Q_{\dagger} , hence the static width of Q , is at most 1. It follows from Theorem 5.1 that Q can be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.

10.2. Complexity Lower Bound. We prove the second statement in Theorem 5.2. The proof is based on a reduction of the Online Matrix-Vector Multiplication (OMv) problem (Definition 5.3) to the evaluation of CQAPs that are not in CQAP_0 .

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

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

⁵To simplify the presentation, we assume that Q contains at least one variable, so it has the static width at least 1. Otherwise, it can be trivially evaluated with constant preprocessing time, update time, and enumeration delay.

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 [BKS17]. 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 Q_3 or Q_4 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 any $\gamma > 0$, can be used to solve the OMv problem in subcubic time, which rejects the OMv conjecture. We then show that, given any CQAP Q that is not in CQAP_0 and does not have repeating relation symbols, we can reduce the evaluation of one of the queries Q_1 , Q_2 , Q_3 and Q_4 to the evaluation of Q .

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} , $\mathbf{v}_1, \dots, \mathbf{v}_n$ be an input to the OMv problem, where \mathbf{M} is an $n \times n$ Boolean Matrix and $\mathbf{v}_1, \dots, \mathbf{v}_n$ 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 $\mathbf{v}_1, \dots, \mathbf{v}_n$. The database domain is $[n]$. Algorithm \mathcal{B} first 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 . Each update can be processed in $\mathcal{O}((n^2)^{\frac{1}{2}-\gamma})$ amortised update time. Hence, the overall 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 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 [BKS17]. By Lemma 10.1, 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 $\text{atoms}(A) \subset \text{atoms}(B)$. This means 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 10.1, 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 referenced 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 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, 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.

Overall, we obtain that CQAPs that are not in CQAP_0 and do not have repeating relation symbols cannot be evaluated with $\mathcal{O}(N^{0.5-\gamma})$ enumeration delay and $\mathcal{O}(N^{0.5-\gamma})$ amortised update time for any $\gamma > 0$, unless the OMv conjecture fails. This concludes the proof of the lower bound statement in Theorem 5.2.

11. TRADE-OFFS FOR CQAPS WITH HIERARCHICAL FRACTURES

For CQAPs with hierarchical fractures, we can parameterise the complexities in Theorem 5.1 to obtain trade-offs between preprocessing time, update time, and enumeration delay. We first restate our main result on such trade-offs from Section 5:

Theorem 5.5 *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 $\mathcal{O}(N^{1+(w-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortised update time for single-tuple updates.*

We achieve these trade-offs by following two core ideas from prior work [KNOZ23c]. 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 during query evaluation, caused by high-degree values in the database.

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 6 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 .

The enumeration faces a new challenge: the tuples encoded in the constructed view trees may overlap, yet we need to enumerate only distinct tuples, i.e., tuples that have not been reported before. To address this challenge, we adapt the union algorithm from prior work [DS11], which is originally designed to enumerate distinct elements from a union of sets. We modify this algorithm to enumerate distinct tuples from multiple view trees.

Handling updates also faces a new challenge: although propagating updates in the constructed view trees follows the procedure from Section 8, updates may change the degrees of values, causing previously light tuples to become heavy and vice versa. In such cases, we need to rebalance the data partitioning and possibly recompute some views. While such rebalancing steps may take longer than a single-tuple update, they happen periodically, and their amortised cost remains the same as that of a single-tuple update.

Sections 11.1-11.4 elaborate our technique and algorithmic ideas that achieve the trade-offs in Theorem 5.5. The full details of our approach are given in Appendices C–E of the technical report [KNOZ25]. Section 11.5 compares our maintenance strategy achieving these trade-offs with typical eager and lazy approaches.

11.1. 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 $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ is a *partition* of R on \mathcal{S} with threshold θ if it satisfies the conditions:

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

We call $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ 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:

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

The relation $R^{\mathcal{S} \rightarrow H}$ is called *heavy*, and the relation $R^{\mathcal{S} \rightarrow L}$ is called *light* on the partition key \mathcal{S} , as they consist of all \mathcal{S} -tuples in R that are heavy and respectively light. Due to the domain partition, the relations $R^{\mathcal{S} \rightarrow H}$ and $R^{\mathcal{S} \rightarrow L}$ are disjoint. For $|\mathcal{D}| = N$ and a strict partition $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ of R on \mathcal{S} with threshold $\theta = N^\epsilon$ for $\epsilon \in [0, 1]$, we have two bounds:

$$(1) \forall \mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S} \rightarrow L} : |\sigma_{\mathcal{S}=\mathbf{t}} R^{\mathcal{S} \rightarrow L}| < \theta = N^\epsilon, \quad \text{and} \quad (2) |\pi_{\mathcal{S}} R^{\mathcal{S} \rightarrow H}| \leq \frac{N}{\theta} = N^{1-\epsilon}.$$

The first bound follows directly from the strict light part condition. The second bound follows from the strict heavy part condition, which says that for each tuple $\mathbf{t} \in \pi_{\mathcal{S}} R^{\mathcal{S} \rightarrow H}$, there exists a relation K such that $|\sigma_{\mathcal{S}=\mathbf{t}} K| \geq N^\epsilon$. Assume now that there exists more than $N^{1-\epsilon}$ such tuples. Then, the database contains more than $N^{1-\epsilon} N^\epsilon = N$ tuples, which contradicts our assumption that the database is of size N .

Disjoint relation parts can be further partitioned independently of each other on different partition keys. We write $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n}$ to denote the relation part obtained after partitioning $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_{n-1} \rightarrow s_{n-1}}$ on \mathcal{S}_n , where $s_i \in \{H, L\}$ for $i \in [n]$. The domain of $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n}$ is the intersection of the domains of $R^{\mathcal{S}_i \rightarrow s_i}$, for $i \in [n]$. We refer to $\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n$ 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 $R^{A \rightarrow L}$, $R^{A \rightarrow H, AB \rightarrow L}$, and $R^{A \rightarrow H, AB \rightarrow H}$. The union of these relation parts constitutes the relation R . In our approach described in Sections 11.2-11.4, the partition keys $\mathcal{S}_1, \dots, \mathcal{S}_n$ in a signature $\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n$ form a strict inclusion chain, i.e., $\mathcal{S}_1 \subset \dots \subset \mathcal{S}_n$. In general, partition keys can be disjoint.

11.2. 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 Figure 7. We illustrate the idea in the following example.

Example 11.1. We explain the construction of the view trees for the connected component from Figure 8 (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 9 (left), the bound variable B dominates the free variables C and D . We create a strict partition of 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

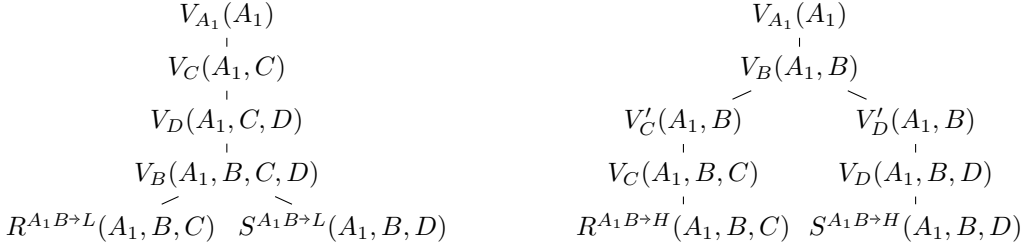


Figure 11: View trees constructed for $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$ from Example 11.1 using the VOs: $A_1 - C - D - B - \{R^{A_1 B \rightarrow L}(A_1, B, C), S^{A_1 B \rightarrow L}(A_1, B, D)\}$ (left) and $A_1 - B - \{C - R^{A_1 B \rightarrow H}(A_1, B, C), D - S^{A_1 B \rightarrow H}(A_1, B, D)\}$ (right).

Figure 11 (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 $R^{A_1 B \rightarrow L}(A_1, B, C)$ of R , we fetch the D -values in $S^{A_1 B \rightarrow L}(A_1, B, D)$ that are paired with (a, b) . The iteration in $R^{A_1 B \rightarrow L}(A_1, B, C)$ takes $\mathcal{O}(N)$ time and for each (a, b) , there are at most N^ϵ D -values in $S^{A_1 B \rightarrow L}(A_1, B, D)$. 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 11 right). The VO and view tree are the same as in Figure 8, except that the leaves are the heavy parts of R and S . We can materialise this view tree in $\mathcal{O}(N)$ time, cf. Example 6.1.

Overall, we can compute the two view trees in $\mathcal{O}(N^{1+\epsilon})$ time. \square

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. Figure 12 shows the construction procedure for a canonical VO ω and an access pattern $(\mathcal{O}|\mathcal{I})$. 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 (Line 5). 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 using the procedure ACCESS-TOP in Figure 4 (Lines 6-7). Queries in CQAP_0 admit a canonical access-top VO. Hence, for such queries, this restructuring does not increase the static width of ω_X .

If Q_X is not in CQAP_0 , then ω_X contains a problematic variable, which is either a bound variable that dominates a free variable or an output variable that dominates an input variable. If X is *not* a problematic variable, we recur on each subtree and combine the constructed VOs (Line 9). Otherwise, we form evaluation strategies that compute different parts of the result of Q_X over its input relations partitioned on *key*, which is the set of variables on the path from X to the root of the canonical VO for Q , including X . We create two sets of VOs: *htrees* and *ltree*. For the former, for each subtree ν_i of ω_X , we construct a VO $\nu_i^{key \rightarrow H}$ by extending the heavy-light signature of each atom in ν_i with $key \rightarrow H$, and we recur on $\nu_i^{key \rightarrow H}$ (Line 10). This ensures that the evaluation of Q_X is over relation parts that are heavy on *key*. The VO *ltree* is obtained by extending the heavy-light signature of each atom in ω_X with $\{key \rightarrow L\}$ and turning $\omega_X^{key \rightarrow L}$ into an access-top VO (Line 11); this restructuring of the VO may increase its static width.

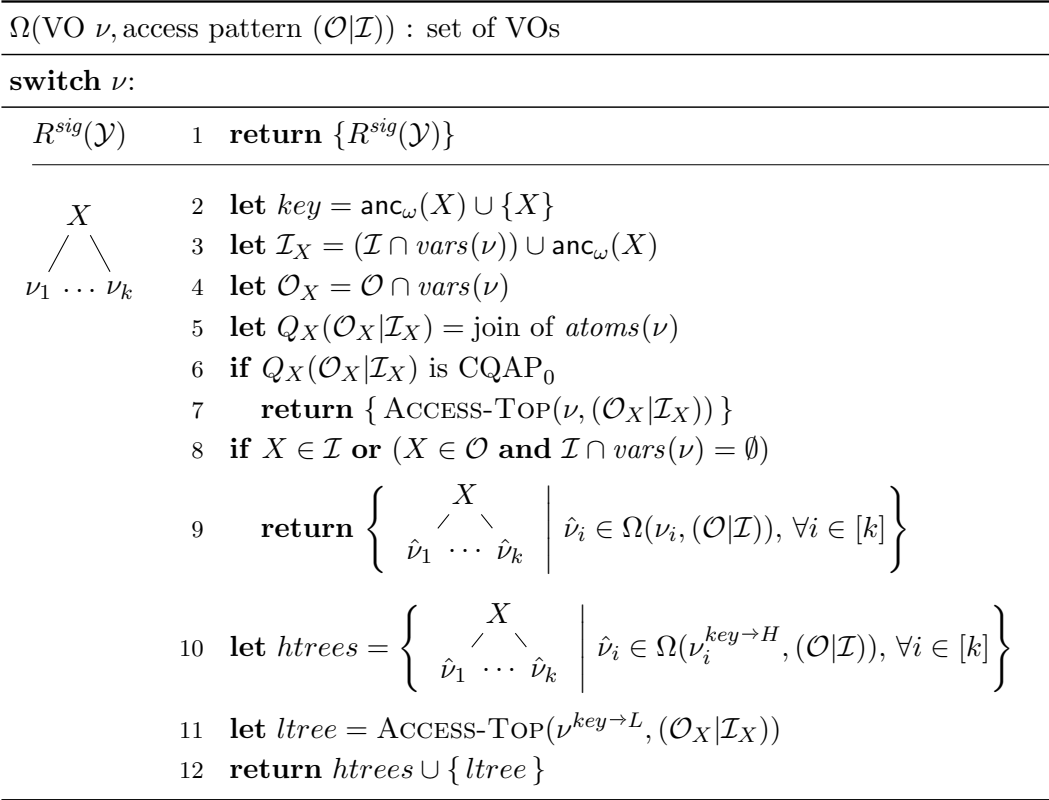


Figure 12: Construction of a set of VOs from a canonical VO ω of a hierarchical CQAP with access pattern $(\mathcal{O}|\mathcal{I})$. Each constructed VO corresponds to an evaluation strategy of some part of the query result. The VO $\nu^{key \rightarrow s}$ for $s \in \{H, L\}$ has the structure of ν but the heavy-light signature of each atom is extended by $key \rightarrow s$. The procedure ACCESS-TOP is given in Figure 4.

We construct a view tree for each VO formed in the previous step. For each view tree, we create a strict partition of 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.

We next discuss the complexity of view tree materialisation. The view trees constructed for the evaluation of queries in CQAP₀ 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 $\mathcal{O}(N^{1+(w-1)\epsilon})$ time, where w is the static width of the query. We give the intuition for this complexity. Consider a view $V(\mathcal{S})$ in a view tree over light relation parts and the set \mathcal{A} of leaf atoms in the subtree rooted at $V(\mathcal{S})$. Since for each hierarchical query, the integral edge cover number is the same as the fractional edge cover number (Lemma 4.4), w must be a natural number. By definition of w , we can cover all variables in \mathcal{S} using at most w atoms from \mathcal{A} . Let $\mathcal{A}_1 \subseteq \mathcal{A}$ be a set of at most w atoms that cover the variables in \mathcal{S} . First, we compute the join of the atoms in \mathcal{A}_1 as follows: We choose one atom $R(\mathcal{X})$ in \mathcal{A}_1 , iterate over the tuples in R , and for each such tuple, we iterate over the matching tuples in the relations of the other at most $w - 1$ atoms in \mathcal{A}_1 . Since all leaf relations are light, there are at most N^ϵ

UNION(view trees T_1, \dots, T_n) : tuple

```

1  if ( $n = 1$ ) return  $T_n.next()$ 
2  if ( $t_{[n-1]} := \text{UNION}(T_1, \dots, T_{n-1}) \neq \mathbf{EOF}$ )
3    if ( $T_n.lookup(t_{[n-1]}) = \text{True}$ )
4       $t_n := T_n.next()$ 
5      return  $t_n$ 
6  return  $t_{[n-1]}$ 
7  if ( $t_n := T_n.next() \neq \mathbf{EOF}$ )
8    return  $t_n$ 
9  return  $\mathbf{EOF}$ 

```

Figure 13: Report the next tuple in a union of view trees.

matching tuples in each relation. Thus, the join can be computed in $\mathcal{O}(N^{1+(w-1)\epsilon})$ time. Let T be the resulting relation and \mathcal{Y} the set of all variables of the atoms in \mathcal{A}_1 . We can rewrite the view $V(\mathcal{S})$ as:

$$V(\mathcal{S}) = T(\mathcal{Y}), R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n),$$

where $R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n)$ are the atoms in $\mathcal{A} \setminus \mathcal{A}_1$. The above query is free-connex α -acyclic [BDG07], since its body is α -acyclic and the free variables \mathcal{S} are included in the schema \mathcal{Y} of one atom. Hence, using Yannakakis' algorithm, V can be computed in time linear in the input plus the output size of the query [BFMY83]. The input size is upper-bounded by the worst-case size of $T(\mathcal{Y})$, which is $\mathcal{O}(N^{1+(w-1)\epsilon})$. The output is a subset of T projected onto \mathcal{S} , hence, its size is also $\mathcal{O}(N^{1+(w-1)\epsilon})$. Thus, $V(\mathcal{S})$ can be computed in $\mathcal{O}(N^{1+(w-1)\epsilon})$ time. Overall, the view tree materialisation takes $\mathcal{O}(N^{1+(w-1)\epsilon})$ time, as stated in Theorem 5.5.

11.3. Enumeration. We next discuss how to enumerate output tuples from the view trees constructed for a CQAP with hierarchical fractures. Our approach builds upon the enumeration procedure for hierarchical queries from prior work [KNOZ23c]. For queries in CQAP₀, the preprocessing stage constructs view trees from access-top VOs. Such view trees admit constant enumeration delay, as discussed in Section 7.

For queries not in CQAP₀, the preprocessing stage constructs view trees from VOs that are not access-top. Enumerating distinct tuples from these view trees poses two challenges: (1) for view trees built from VOs that are not access-top, the enumeration approach from Section 7 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; and (2) the tuples encoded in the constructed view trees may overlap, while we need to enumerate distinct tuples. We rely on the union algorithm [DS11] to handle these challenges.

The UNION algorithm is given in Figure 13. It takes as input n view trees that represent possibly overlapping sets of tuples and returns a tuple that is distinct from all tuples returned before. Each view tree supports two operations: *next()* returns the next tuple in the view tree or **EOF** if the view tree is exhausted, and *lookup(t)* checks whether the tuple t is present in the view tree.

We first explain the algorithm on two view trees T_1 and T_2 that represent possibly overlapping sets of tuples. Each call returns one tuple or **EOF**. The algorithm returns the next tuple t_1 in T_1 only if it is not present in T_2 ; otherwise, it returns the next tuple in T_2 (Lines 2-6). In case T_1 is exhausted, the algorithm returns the next tuple in T_2 , or **EOF** in case T_2 is also exhausted.

In the case of more than two view trees ($n > 2$), we consider the union of the first $n - 1$ view trees as one view tree and T_n as another view tree. This reduces the general case to the previous case of two view trees.

The UNION algorithm performs $\mathcal{O}(n)$ *lookup* and *next* operations over n view trees before reporting a tuple. Thus, its runtime is $\mathcal{O}(n(\text{delay} + \text{lookup}))$, where *delay* is the time to retrieve the next tuple in a view tree and *lookup* is the cost of a lookup into a view tree.

We use the UNION algorithm to address the two aforementioned challenges in enumerating from the view trees constructed from VOs that are not access-top. For the first challenge, let A be a variable that violates the free-dominance or input-dominance condition. The constructed non-access-top view trees are over relation parts where A -values are heavy. We instantiate a view tree for each A -value and use the union algorithm to report only the distinct tuples. The number of instantiated view trees is upper-bounded by the number of heavy A -values, i.e., $n = \mathcal{O}(N^{1-\epsilon})$. Since A is fixed in each instantiated view tree, A is effectively an input variable and the view tree is as if constructed from an access-top VO, and thus supports constant-delay enumeration using the enumeration approach from Section 7. The lookup operation can be performed using the enumeration procedure, where all free variables are considered as input variables and set to the values of the tuple to be looked up, which thus takes constant time. Hence, the delay of the union algorithm is $\mathcal{O}(N^{1-\epsilon})$.

For the second challenge, we use the union algorithm to report only distinct tuples from the set of view trees. As explained in the first challenge, the view trees admit enumeration delay $\mathcal{O}(N^{1-\epsilon})$ and thus $\mathcal{O}(N^{1-\epsilon})$ lookup time. The number of constructed view trees is constant. Overall, the delay of the union algorithm is $\mathcal{O}(N^{1-\epsilon})$.

Example 11.2. We explain the enumeration procedure for the view trees from Figure 11, constructed for the query $Q_1(D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$. The view tree on the left, constructed over the light parts of R and S , corresponds to an access-top VO. For a fixed (A_1, C) -value, enumerating the matching D -values from V_D takes constant time per output value. The view tree on the right, however, corresponds to a VO where B violates the free-dominance and input-dominance conditions. This view tree comprises the heavy parts of R and S partitioned on (A_1, B) . The number of distinct (A_1, B) -values in each part is at most $N^{1-\epsilon}$, meaning that the size of the view $V_B(A_1, B)$ built on top of these parts is also at most $N^{1-\epsilon}$. To resolve the issue with violating B , we find the B -values that are paired with the input A_1 -value in V_B and also paired with the input (A_1, C) -value in $R^{A_1 B \rightarrow H}$, and instantiate for each such B -value a view tree. The number of such view trees is at most $N^{1-\epsilon}$, and each view tree supports constant-time lookup and constant-delay enumeration of the D -values in $S^{A_1 B \rightarrow H}$. To report only distinct D -values, we employ the union algorithm over the iterators instantiated from the right tree and the iterator over the left tree. The cost of de-duplication using this algorithm is proportional to the number of instantiated view tree iterators; thus, the enumeration delay is $\mathcal{O}(N^{1-\epsilon})$. \square

Appendix D of the technical report provides a complete proof of the enumeration delay $\mathcal{O}(N^{1-\epsilon})$ from Theorem 5.5, along with our enumeration procedure, which uses the standard iterator interface with *open* and *next* methods [KNOZ25].

11.4. 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 8.

Example 11.3. We consider the maintenance of the view trees from Figure 11 under a single-tuple update $\delta R(a, b, c)$ to R . The update affects the heavy part $R^{A_1 B \rightarrow H}$ if $(a, b) \in \pi_{A_1, B} R^{A_1 B \rightarrow H}$; otherwise, it affects the light part $R^{A_1 B \rightarrow L}$. For the former, we propagate the update from $R^{A_1 B \rightarrow H}$ 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 $\delta V_B(a, b, c, D) = \delta R^{A_1 B \rightarrow L}(a, b, c), S^{A_1 B \rightarrow L}(a, b, D)$ in $\mathcal{O}(N^\epsilon)$ time because there are at most N^ϵ D -values paired with (a, b) in $S^{A_1 B \rightarrow L}$. 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. \square

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. We give the intuition behind the amortised cost of rebalancing. The full proof is in Appendix E of the technical report [KNOZ25].

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 $\lfloor \frac{1}{4}M \rfloor \leq N \leq M$ 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 recreating a strict partition of the input relations with the new threshold M^ϵ and recomputing the views.

A major rebalancing requires $\mathcal{O}(N)$ time to repartition the relations and $\mathcal{O}(N^{1+(w-1)\epsilon})$ time to recompute the view trees using the procedure from Section 11.2. This cost is amortised over $\Omega(M)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}M$ (after doubling), or $N = \frac{1}{2}M - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N \leq M$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}M$. The amortised time of major rebalancing is thus $\mathcal{O}(N^{(w-1)\epsilon})$. By Proposition 4.11, we have $\delta = w$ or $\delta = w - 1$; hence, the amortised major rebalancing cost is $\mathcal{O}(N^{\delta\epsilon})$.

Minor Rebalancing. After an update $\delta R = \{\mathbf{x} \rightarrow m\}$ to relation R , we check the degrees of the values in \mathbf{x} . Consider a partition key key that is included in the schema of \mathbf{x} and the projection \mathbf{v} of \mathbf{x} onto key . If \mathbf{v} is included in the light part of the partition of R on key but the degree of \mathbf{v} is not below $\frac{3}{2}M^\epsilon$ in at least one input relations, all tuples containing \mathbf{v} are moved to the relation parts that are heavy on \mathbf{v} . Likewise, if \mathbf{v} is in a relation part that is heavy on key but the degree of \mathbf{v} is below $\frac{1}{2}M^\epsilon$ in all input relations, all tuples containing \mathbf{v} are moved to the relation parts that are light on \mathbf{v} .

A minor rebalancing step requires $\mathcal{O}(N^{(\delta+1)\epsilon})$ time: It either moves $\mathcal{O}(\frac{3}{2}M^\epsilon)$ tuples that contain \mathbf{v} to relations parts that are heavy on \mathbf{v} (light to heavy) or $\mathcal{O}(\frac{1}{2}M^\epsilon)$ tuples that contain \mathbf{v} to relation parts that are light on \mathbf{v} (heavy to light). Each move is by an insert and a delete operation, which takes $\mathcal{O}(N^{\delta\epsilon})$ time. The total cost $\mathcal{O}(N^{(\delta+1)\epsilon})$ of minor rebalancing is amortised over $\Omega(M^\epsilon)$ updates. This lower bound on the number of updates

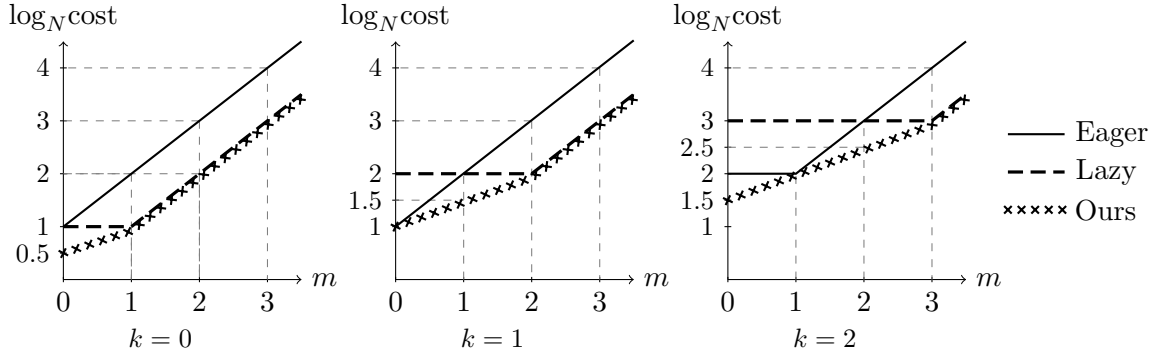


Figure 14: Plotting the exponents in the complexities of three maintenance approaches (ours, eager, and lazy) as piecewise linear functions in the parameters m and k , for processing a batch of $\mathcal{O}(N^m)$ single-tuple updates followed by the enumeration of $\mathcal{O}(N^k)$ output tuples. Our approach is asymptotically faster or the same as the best of the eager and lazy approaches.

comes from the gap between the two thresholds in the heavy and light part conditions. The amortised cost of minor rebalancing is $\mathcal{O}(N^{\delta\epsilon})$.

Overall, even though the cost of rebalancing steps 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.

11.5. Comparison with Prior Approaches. We compare our adaptive maintenance strategy with the mainstream eager and lazy approaches in an IVM scenario where either all or a fraction of the output tuples are reported after a batch of updates. We show in the following examples that our approach has at most the same overall time complexity as these mainstream approaches.

Example 11.4. Let us consider the running example with the query from Example 11.1:

$$Q_1(D \mid A_1, C) = R(A_1, B, C), S(A_1, B, D)$$

Assume the relations have size $\mathcal{O}(N)$. The query result has size $\mathcal{O}(N^2)$ for all pairs of input (A_1, C) -values and $\mathcal{O}(N)$ for one such pair.

We can recover the complexities for typical eager and lazy maintenance approaches using our approach by setting $\epsilon = 1$ and respectively $\epsilon = 0$ (except for the complexity of the 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 in $\mathcal{O}(N^2)$ time. 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 D -values for any input pair of $\{A_1, C\}$ -values with constant delay.

The lazy approach has no precomputation and only updates each relation, without propagating the changes to the query output. For the first enumeration request for a pair (a_1, c) of input values, it needs to calibrate the relations in the residual query $Q_1(D) = R(a_1, B, c), S(a_1, B, D)$. This takes linear time. After that, it can enumerate the D -values for that input pair with constant delay. For another pair of input values, it needs again to recalibrate the relations, which also takes linear time. Its delay is linear in worst-case.

Consider now an IVM scenario where, after every $\mathcal{O}(N^m)$ single-tuple updates, we request the enumeration of $\mathcal{O}(N^k)$ output tuples (in the lexicographic order A_1, C, D , for some pairs of input values), for $m \geq 0$ and $0 \leq k \leq 2$. After the initial preprocessing, our approach then takes $\mathcal{O}(N^{m+\epsilon} + N^{k+1-\epsilon}) = \mathcal{O}(N^{\max\{m+\epsilon, k+1-\epsilon\}})$ overall time to accommodate the batch of updates followed by the enumeration requests. In contrast, the eager and lazy approaches need time $\mathcal{O}(N^{m+1} + N^k) = \mathcal{O}(N^{\max\{m+1, k\}})$ and $\mathcal{O}(N^m + N^{k+1}) = \mathcal{O}(N^{\max\{m, k+1\}})$ respectively. For any value for $m \geq 0$, the time complexity of our approach is at most that of the other approaches and it can be asymptotically better. Figure 14 shows the complexity of the three approaches for different values of m and k .

For $k = 0$, our approach has the time complexity $\mathcal{O}(N^{\frac{m+1}{2}})$ for $m \leq 1$ and $\epsilon = \frac{1-m}{2}$, and the complexity $\mathcal{O}(N^m)$ for $m > 1$ and $\epsilon = 0$. In contrast, the eager and lazy approaches take time $\mathcal{O}(N^{m+1})$ and $\mathcal{O}(N^{\max\{m, 1\}})$ respectively.

For $k = 1$, our approach has the time complexity $\mathcal{O}(N^{1+\frac{m}{2}})$ for $m \leq 2$ and $\epsilon = \frac{2-m}{2}$, and the complexity $\mathcal{O}(N^m)$ time for $m > 2$ and $\epsilon = 0$. In contrast, the eager and lazy approaches take time $\mathcal{O}(N^{m+1})$ and $\mathcal{O}(N^{\max\{m, 2\}})$ respectively.

For $k = 2$, our approach has the time complexity $\mathcal{O}(N^{\frac{m+3}{2}})$ for $m \leq 3$ and $\epsilon = \frac{3-m}{2}$, and the complexity $\mathcal{O}(N^m)$ for $m > 3$ and $\epsilon = 0$. In contrast, the eager and lazy approaches take time $\mathcal{O}(N^{\max\{m+1, 2\}})$ and $\mathcal{O}(N^{\max\{m, 3\}})$ respectively.

The 4-cycle query from Example 3.2:

$$Q_2(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D)$$

exhibits the same trade-offs and complexities as the above acyclic query Q_1 for all three approaches: ours, eager, and lazy. Therefore, the analysis and conclusion are the same. \square

12. SEMANTICS FOR UPDATES IN PROBABILISTIC DATABASES

In this section and the following section, we extend our dynamic evaluation approach to probabilistic databases. Here, we discuss possible semantics of updates in probabilistic databases: Given a single-tuple insertion or a deletion, how to update the probabilistic database to incorporate this update? In the next section, we show how to maintain the result of any query in CQAP₀ over probabilistic databases under updates.

We first recall the notion of tuple-independent probabilistic databases and the semantics of query evaluation over such databases. We then contrast several update semantics.

12.1. Tractable Query Evaluation over Probabilistic Databases. A probabilistic database is a relational database in which the tuples are pairwise independent probabilistic events [SORK11]. We interpret a tuple t as being in the database with probability $p(t)$ and out of the database with probability $1 - p(t)$. Since each tuple can be in or out of the database, a probabilistic database of n such tuples represents 2^n possible worlds, one world for each relational database representing a subset of the set of tuples in the database. Let S

be the set of tuples in a probabilistic database \mathcal{W} and $W \in \mathcal{W}$ be one of its possible worlds, i.e., $W \subseteq S$. The probability $P(W)$ of W is the product of (1) the probability of each tuple in W , and (2) one minus the probability of each tuple in S and not in W :

$$P(W) = \prod_{t \in W} p(t) \cdot \prod_{t \in S \setminus W} (1 - p(t))$$

Given a Boolean conjunctive query Q and a probabilistic database \mathcal{W} , the semantics of Q is to compute Q in each possible world W of \mathcal{W} and sum up the probabilities of those possible worlds where its answer is true:

$$P(Q) = \sum_{W \in \mathcal{W}: Q(W) = \text{true}} P(W)$$

For a non-Boolean conjunctive query $Q(\mathcal{F})$ with free variables \mathcal{F} and any tuple of values $\mathbf{f} \in \text{Dom}(\mathcal{F})$ in the active domain of the tuple of free variables \mathcal{F} , we define the residual Boolean query $Q_{\mathbf{f}}$ where we set the variables in \mathcal{F} to their respective values in \mathbf{f} . Then, the probability for \mathbf{f} to be in the output of Q is $P(Q_{\mathbf{f}})$.

For a CQAP $Q(\mathcal{O}|\mathcal{I})$ and a given tuple $\mathbf{in} \in \text{Dom}(\mathcal{I})$ of values for the input variables, the query $Q(\mathcal{O}|\mathbf{in})$ is a (possibly non-Boolean) conjunctive query. Therefore, the probability for a tuple $\mathbf{out} \in \text{Dom}(\mathcal{O})$ in the active domain of the output variables \mathcal{O} is given by $P(Q_{\mathbf{out} \circ \mathbf{in}})$, where $Q_{\mathbf{out} \circ \mathbf{in}}$ is the residual Boolean query obtained by setting the free output and input variables to their respective values in the tuple of values $\mathbf{out} \circ \mathbf{in}$.

The query semantics does not lead to a practical query evaluation, as it requires to iterate over all possible worlds. Instead, state-of-the-art query evaluation techniques (1) exploit the query structure to compute directly on the probabilistic database, without the need to iterate over possible worlds, or (2) derive the so-called query lineage, which is a Boolean function tracing the possible derivations of the query answer from the input tuples, and then use knowledge compilation techniques to compile the lineage into a tractable form that allows efficient probability computation [SORK11].

A remarkable result is the following computational dichotomy [DS04]: Let Q be a Boolean conjunctive query without repeating relation symbols and \mathcal{W} any probabilistic database. If Q is hierarchical, then its data complexity is polynomial time. If Q is non-hierarchical, then its data complexity is hard for $\#P$. An immediate generalization holds for non-Boolean conjunctive queries, by checking whether their residual Boolean queries (obtained by fixing the free variables to constants) are hierarchical [OHK09].

An implication of this dichotomy is that CQAPs with hierarchical fracture can be computed in polynomial time data complexity over probabilistic databases. A natural question is whether they can be also maintained with constant update time and constant enumeration delay. As shown in Section 13, to achieve these maintenance desiderata, we need the three properties from Definition 3.1: the query fracture is hierarchical, free-dominant, and input-dominant. Tractability in the static setting only requires the hierarchical property.

12.2. Probabilistic Update Semantics. Prior work [BM21] considers an update semantics for probabilistic databases that is deterministic in case of deletions and probabilistic in case of insertions. Given an insertion of a tuple t with probability $p(t)$ in a probabilistic relation R , the tuple is inserted into R as an independent event t with probability $p(t)$. Given a deletion of a tuple t from a probabilistic relation R , the tuple is removed from R if it exists in R , regardless of its probability; if t does not exist in R , no action is taken.

A natural interpretation of single-tuple updates, which agrees with the possible worlds semantics of probabilistic databases, is that of independent probabilistic events: Given an insertion (deletion) of a tuple t with probability p , we insert t in (delete t from) the database with probability p and ignore the update with probability $1 - p$.

Example 12.1. Consider a probabilistic database consisting of a tuple t with probability $1/2$. We consider two scenarios: We either insert or delete t with probability $1/4$.

The insertion case. We have four possible worlds, depending on whether each of the two events holds: (1) t is in the database and we ignore the insert; this world consists of the tuple t and has the probability $1/2 \cdot (1 - 1/4) = 3/8$; (2) t is in the database and the insertion is triggered; this world consists of t and has probability $1/2 \cdot 1/4 = 1/8$; (3) t is not in the database and the insertion is ignored; this world is empty and has probability $(1 - 1/2) \cdot (1 - 1/4) = 3/8$; (4) t is not in the database and the insertion is triggered; this world consists of t and has probability $(1 - 1/2) \cdot 1/4 = 1/8$. As expected, the sum of the probabilities of all four worlds is 1. The third world is empty, all other worlds consist of t . The probability of t is the sum of the probabilities of all worlds except the third world, or equivalently 1 minus the probability of the third world: $1 - 3/8 = 5/8$.

The deletion case. We again have four possible worlds, depending on whether each of the two events holds: (1) t is in the database and we ignore the delete; this world consists of the tuple t and has the probability $1/2 \cdot (1 - 1/4) = 3/8$; (2) t is in the database and the deletion is triggered; this world is empty and has probability $1/2 \cdot 1/4 = 1/8$; (3) t is not in the database and the deletion is ignored; this world is empty and has probability $(1 - 1/2) \cdot (1 - 1/4) = 3/8$; (4) t is not in the database and the deletion is triggered, albeit with no effect; this world has probability $(1 - 1/2) \cdot 1/4 = 1/8$. As expected, the sum of the probabilities of all four worlds is 1. Out of them, only the first world has t , so the probability that t is in the database after the update is the probability of this world, which is $3/8$. \square

We can generalise Example 12.1. Given a single-tuple update $t \mapsto p$, we update the probabilistic database as follows. If the update is an insertion and t is already in the database with probability p' , then the updated database contains t with probability $p + p' - p \cdot p' = 1 - (1 - p)(1 - p')$; this corresponds to the probability of those worlds where at least one of the two holds: (i) the tuple is inserted and (ii) the tuple is in the database. If the database has no event $t \mapsto p'$ before the update, or equivalently $p' = 0$, then after the insertion the database contains t with probability p . If the update is a deletion and t is already in the database with probability p' , then the updated database contains t with probability $p' \cdot (1 - p)$; this corresponds to the probability of the world where t is in the database and the deletion is not triggered. If the database has no event $t \mapsto p'$ before the update, or equivalently $p' = 0$, then before and after the deletion the database does not contain t (and the deletion has no effect). The above behaviour holds regardless of other possible tuples in the database, since they are independent of both t and the update.

We call this semantics the *probabilistic set semantics*: It interprets each update as an independent probabilistic event and uses set semantics (no duplicates) within each world. The query maintenance mechanism put forward in Section 13 can propagate updates from the input relations up the view trees constructed for any query in CQAP_0 using this probabilistic set semantics for updates in constant time, while allowing for constant-delay enumeration of the query result after each update.

A shortcoming of the probabilistic set semantics, as already apparent in Example 12.1, is that the order of updates matters: Given two updates, one deleting t and one inserting t ,

then the two possible orders of updates yield different databases. Furthermore, the semantics ignores the multiplicity of a tuple in a possible world, so this semantics does not generalise the relational case discussed in the previous sections, where we maintain tuple multiplicities to ensure correct maintenance and accommodate out-of-order updates, to the probabilistic setting. In particular, this means that a possible world, where we trigger several insertions of the same tuple t followed by one deletion of t , is empty. Also, if we were to first delete and then insert, then the deletion is lost and therefore has no effect.

The probabilistic set semantics can be generalised to avoid the two aforementioned pitfalls: Instead of maintaining the probability of a tuple being in (and missing from) the database, we maintain the discrete probability distribution over its possible multiplicities: $\{(i, p_i) \mid i \in \mathbb{Z}\}$, where p_i is the probability that the tuple has multiplicity i , $p_i \neq 0$ for finitely many i values, and $\sum_i p_i = 1$. Like in the relational case in the previous sections, the multiplicity is an integer and captures the number of insertions and deletions of a tuple in the database; for derived tuples in views defined over the database, it captures the number of derivations from the input tuples. This generalisation is the *probabilistic bag semantics*.

Example 12.2. Consider now a probabilistic database that contains a tuple t whose probability distribution over its multiplicities is: $\{(2, p_2), (1, p_1), (0, p_0), (-1, p_{-1})\}$. That is, tuple t has multiplicity i with probability p_i , for $-1 \leq i \leq 2$. We again consider two scenarios: We either insert or delete t with probability p .

The insertion case. The new probability distribution over the multiplicities of t becomes: $\{(3, p_2 \cdot p), (2, p_2 \cdot (1-p) + p_1 \cdot p), (1, p_1 \cdot (1-p) + p_0 \cdot p), (0, p_0 \cdot (1-p) + p_{-1} \cdot p), (-1, p_{-1} \cdot (1-p))\}$. The tuple has multiplicity i after the insertion if either (1) it had multiplicity i before the insertion and the insertion is not triggered, or (2) it had multiplicity $i - 1$ before the insertion and the insertion is triggered. In the first case, the probability is the product of the probability p_i that the tuple is in the database with multiplicity i and of the probability $1 - p$ that the insertion is not triggered. The product here is correct since the two events are independent, and they must both occur. Similarly, in the second case, the two cases are mutually exclusive events, so their joint probability is the sum of their probabilities.

The deletion case. If we delete t with probability p , then the new probability distribution over the multiplicities of t becomes: $\{(2, p_2 \cdot (1-p)), (1, p_2 \cdot p + p_1 \cdot (1-p)), (0, p_1 \cdot p + p_0 \cdot (1-p)), (-1, p_0 \cdot p + p_{-1} \cdot (1-p)), (-2, p_{-1} \cdot p)\}$. The reasoning is similar to that of insertion. \square

We can generalise Example 12.2 to formally define probability distributions over multiplicities and the operations on them, as detailed in Appendix G of the technical report [KNOZ25]. A drawback of the probabilistic bag semantics is that the probability distribution associated with each tuple in an input relation can grow linearly with the number of updates; for tuples in views, their probability distributions can grow polynomially (in data complexity) with the number of updates. As a result, both the update and the enumeration steps are expensive.

To alleviate the computational complexity brought by the probabilistic bag semantics for updates, we can maintain the expectation and variance of the probability distributions over tuple multiplicities, instead of storing and maintaining the full distributions. We then associate each tuple with a pair of the expected value and the variance of its multiplicity. We refer to this refinement as the *expectation-variance update semantics*.

Under the expectation-variance update semantics, an insertion of a tuple t with probability p is a random event, where the multiplicity of t is a random variable X with the probability distribution $\{(0, 1-p), (1, p)\}$. By definition, the expectation of X is given by $E[X] = 0 \cdot (1-p) + 1 \cdot p = p$ and the variance of X is given by $\text{Var}[X] = E[X^2] - E[X]^2 =$

$1^2 \cdot p - p^2 = p(1 - p)$. Similarly, for a deletion of a tuple t with probability p , the multiplicity of t is a random variable Y with the probability distribution $\{(0, 1 - p), (-1, p)\}$. Then, by definition, $E[Y] = -1 \cdot p = -p$ and $\text{Var}[Y] = E[Y^2] - E[Y]^2 = (-1)^2 \cdot p - (-p)^2 = p(1 - p)$.

To compute the expectation and variance of the multiplicity of tuple in a view, we exploit properties of the sum and product of two independent random variables X and Y :

$$\begin{aligned} E[X + Y] &= E[X] + E[Y] \\ \text{Var}[X + Y] &= \text{Var}[X] + \text{Var}[Y] \\ E[XY] &= E[X] E[Y] \\ \text{Var}[XY] &= \text{Var}[X] \text{Var}[Y] + \text{Var}[X] E[Y]^2 + \text{Var}[Y] E[X]^2 \end{aligned}$$

Inserting (deleting) a tuple with some probability increases (decreases) the expected value of the tuple's multiplicity. Note that the expected multiplicity can be negative.

Example 12.3. Consider a probabilistic database under the expectation-variance update semantics, where each tuple is paired with the expected value and variance of its multiplicity. Since updates are independent probabilistic events, the expected value and variance after an update can be computed using the above properties of the sum of two independent random variables. When a tuple is updated with probability p , its expected multiplicity increases by p if the update is an insertion and decreases by p if the update is a deletion, while the variance of its multiplicity increases by $p(1 - p)$ in both cases. \square

We can define two binary operations, \oplus and \odot , to compute the expected value and variance of the sum and respectively product of two independent random variables, given the (expected value, variance) pairs of the two variables.

Definition 12.4. Define binary operations $\oplus : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$ and $\odot : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^2$ as:

$$\begin{aligned} (a, b) \oplus (c, d) &= (a + c, b + d) \\ (a, b) \odot (c, d) &= (ac, bd + a^2d + bc^2) \end{aligned}$$

Both operations execute in constant time.

Proposition 12.5. *The structures $(\mathbb{R}^2, \oplus, (0, 0))$ and $(\mathbb{R}^2, \odot, (1, 0))$ are commutative monoids.*

The query maintenance mechanism in Section 13 can propagate updates from the input relations to the result of any query in CQAP₀ using the \oplus and \odot operations from Definition 12.4 under the probabilistic expectation-variance semantics for updates.

13. DYNAMIC EVALUATION FOR CQAP₀ OVER PROBABILISTIC DATABASES

We here show that any query in CQAP₀ without repeating relation symbols can be maintained over a probabilistic database with constant update time and enumeration delay under both the set semantics and the expectation-variance semantics for updates in probabilistic databases. We conclude with a discussion on the maintenance under the probabilistic bag semantics.

Our main insight is that the exact same maintenance approach used for queries in CQAP₀ over relational databases is also applicable for such queries over probabilistic databases. There are two main reasons for this. (1) The probability of any Boolean hierarchical conjunctive query without repeating relation symbols can be computed using two operators, independent-project and independent-join [SORK11]. Independent-project computes the disjunction of independent events, as discussed in Section 12.2 for the various update semantics considered.

Independent-join computes the conjunction of independent events, which is possible if we only join distinct probabilistic relations (no self-joins). (2) For queries in CQAP_0 over probabilistic databases, we can construct trees of views that are tuple-independent relations and constructed only using projections, which employ the independent-project operator, and one-to-one joins, which employ the independent-join operator.

Example 13.1. Recall the query $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ in CQAP_0 , whose hypergraph is depicted in Figure 8 (middle), its canonical access-top variable order is in Figure 9 (left), and its view tree is in Figure 9 (middle). The probabilistic relations R and S consist of pairwise independent tuples. The view $V'_C(A_1, B)$ is created by projecting away C from the relation R . The projection may create duplicates, i.e., tuples $(a_1, b) \mapsto p_i$ with the same pair of values (a_1, b) for the variables (A_1, B) in V'_C and $i \in [n]$. Since these tuples are pairwise independent, we can replace them by one tuple $t \mapsto 1 - \prod_{i \in [n]} (1 - p_i)$ under the probabilistic set semantics. The tuples in V'_C remain pairwise independent, even after removing the duplicates as explained above. The same treatment applies to the view $V'_D(A_1, B)$, which is created by projecting away D from the relation S . The view $V_B(A_1, B)$ is the intersection of the views V'_C and V'_D . Each resulting tuple appears in both views. Its probability is the product of its probabilities in the two views. Since the tuples in V_B result from distinct tuples in the child views, they are pairwise independent. Finally, the view $V_{A_1}(A_1)$ is created by projecting away B from V_B . The duplicates are merged into a single tuple whose probability is the probability of the disjunction of its duplicates. Again, the distinct tuples in V_B are pairwise independent, since they originate from disjoint sets of tuples in the child view V_B . \square

The following statement captures the property that the view trees for queries in CQAP_0 have independent tuples.

Proposition 13.2. *Given a query Q in CQAP_0 without repeating relation symbols, a canonical access-top variable order ω for Q , and the view tree $\tau(\omega)$ for Q over a probabilistic database D . Then the tuples in each view of $\tau(\omega)$ are pairwise independent.*

Proof. We show this using induction on the structure of the view tree built for Q .

Base case: By definition, the tuples in the input relations are pairwise independent. Updates to the input relations also preserve the independence of the tuples under both probabilistic set and bag semantics, as discussed in Section 12.2.

Inductive step: We assume the tuples in the child views are pairwise independent and show this to be the case also for parent views.

For a query in CQAP_0 , the view tree construction in Figure 7 has three cases of inner views: (1) It either constructs a parent view that is a copy of the child view (this is not needed but keeps the algorithm and its analysis simpler); (2) Alternatively, it creates a parent view that is a projection of the child view; (3) The last case is a parent view that is the intersection of several child views. We analyse each of these cases next.

(1) This holds trivially by the induction hypothesis.

(2) A projection may create duplicates, which can be merged into one common tuple whose probability is that of the disjunction of the duplicates. Since the tuples in the view can be partitioned into disjoint sets of duplicates, such that within each set and across sets the tuples are independent, the output tuples for the different sets are pairwise independent.

(3) An intersection of several child views is a 1-1 join. Since the input relations are distinct (no self-join), the tuples across all the child views are pairwise independent. The

intersection then yields the subset of these tuples that appear in all child views, so these tuples remain pairwise independent. The probability of each such tuple is the product of the probabilities of the tuple in each of the child views. \square

Proposition 13.2 essentially states that the view trees we create in case of queries in CQAP_0 correspond to so-called safe plans used for efficient probability computation [DS04, SORK11]. The safe plans are however not enough for efficient maintenance, i.e., for constant update time and constant enumeration delay. Indeed, CQAP_0 further constraints the queries to be free-dominant and input-dominant.

Theorem 13.3. *Given a query Q in CQAP_0 without repeating relation symbols and a probabilistic database D of size N , then Q can be maintained with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ time for single-tuple updates, and $\mathcal{O}(1)$ enumeration delay, using the probabilistic set semantics or the probabilistic expectation-variance semantics for updates.*

Proof. We first prove the theorem under the probabilistic set semantics and then extend the proof to the probabilistic expectation-variance semantics.

Consider a query $Q(\mathcal{O}|\mathcal{I})$ in CQAP_0 without repeating relation symbols, its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$, and a database D of size N . Theorem 5.2 states that Q can be maintained with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay, in case the database D consists of relations that map tuples to multiplicities, as defined in our data model in Section 2. These complexities are achieved by our approach described in Sections 6-8. In case D is a probabilistic database, we can achieve the same complexities using the same maintenance approach with two twists, which we explain next.

We distinguish the following cases when handling a single-tuple update $t \mapsto p$:

- (1) *Updating a base relation.* Assume that the probability of tuple t being in the database is p^{old} , then we compute the probability p^{new} of tuple t being in the database after the update as discussed in Section 12.2:

$$p^{new} = \begin{cases} 1 - (1 - p^{old}) \cdot (1 - p), & \text{if the update is an insertion} \\ p^{old} \cdot (1 - p), & \text{if the update is a deletion.} \end{cases}$$

Computing p^{new} takes constant time, assuming that the basic arithmetic operations can be performed in constant time. We then propagate the information that the probability of tuple t has changed from p^{old} to p^{new} further up in each affected view tree.

- (2) *Updating a view V .* Since the query Q is in CQAP_0 , each view represents the intersection of its child views, possibly followed by an aggregation that projects away variables. A single-tuple update coming from one child yields a change containing at most one tuple whose probability is the product of the probabilities of the joined tuples.

While in the relational case, the aggregation amounts to summing up the multiplicities of duplicates, in the probabilistic case the aggregation amounts to computing the probability of the disjunction of independent events corresponding to k duplicate tuples, as per Proposition 13.2.

Consider k such independent events with probabilities p_1, \dots, p_k . Their joint probability is $p = 1 - \prod_{i \in [k]} (1 - p_i)$. Our goal is to maintain p whenever any p_i changes. If any of the duplicates is certain (i.e., $p_i = 1$), then the joint probability becomes 1, effectively disregarding all other probabilities. This does not encode how many of these duplicates are certain. To ensure we maintain the correct joint probability p under changes to any of the probabilities p_1, \dots, p_k , we associate the tuple t in the view V resulting from

the aggregation of the k duplicates with a pair (q, m) , where q is the product term $\prod_{i \in [k]: p_i < 1} (1 - p_i)$, which involves only the probabilities of the uncertain duplicates, and m is the number of the certain duplicates. Then, the joint probability p is 1 when $m > 0$, i.e., when there is at least one certain duplicate, and $1 - q$ when $m = 0$, i.e., when there is no certain duplicate and then $p = 1 - q = 1 - \prod_{i \in [k]} (1 - p_i)$.

Assume now that the probability p_i changes from p_i^{old} to p_i^{new} . We compute for the tuple t the new pair (q^{new}, m^{new}) from the current pair (q^{old}, m^{old}) as follows:

$$(q^{new}, m^{new}) = \begin{cases} (q^{old}, m^{old}) & \text{if } p^{old} = 1 \wedge p^{new} = 1 \\ (q^{old} \cdot (1 - p^{new}), m^{old} - 1) & \text{if } p^{old} = 1 \wedge p^{new} < 1 \\ (\frac{q^{old}}{1 - p^{old}}, m^{old} + 1) & \text{if } p^{old} < 1 \wedge p^{new} = 1 \\ (\frac{q^{old}}{1 - p^{old}} \cdot (1 - p^{new}), m^{old}) & \text{if } p^{old} < 1 \wedge p^{new} < 1 \end{cases}$$

In each case computing (q^{new}, m^{new}) takes constant time. From (q^{old}, m^{old}) and (q^{new}, m^{new}) , computing the aggregated probability before and after the update, needed for subsequent propagation in the view tree, also takes constant time. Overall, the overhead added to the maintenance cost by the probability computation is constant. The correctness of the probabilities assigned to tuples in views follows from Proposition 13.2.

Our enumeration procedure from Section 7 reports for any input tuple, all tuples in the query result with constant delay. In case of probabilistic databases, we need to report for each output tuple also its probability. We explain in the following how to compute these probabilities. Assume that our view trees constructed in the preprocessing stage follow an access-top VO ω for $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$ that consists of the trees $\omega_1, \dots, \omega_n$. Let $T_1 = \tau(\omega_1), \dots, T_n = \tau(\omega_n)$ be the view trees constructed using the procedure τ in Figure 7. For each $j \in [n]$, let $Q_j(\mathcal{O}_j | \mathcal{I}_j)$ with $\mathcal{O}_j = \mathcal{O} \cap \text{vars}(\omega_j)$ and $\mathcal{I}_j = \mathcal{I}_{\dagger} \cap \text{vars}(\omega_j)$ be the CQAP that joins the atoms appearing at the leaves of T_j . In Section 7, we 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 . For each such tuple $\mathbf{t}_j \in Q_j(\mathcal{O}_j|\mathbf{i}_j)$, we can traverse T_j to compute its probability as follows. We first check whether the schema of \mathbf{t}_j is equal to the schema of the root view V . If yes, the probability of \mathbf{t}_j is $V(\mathbf{t}_j)$. Otherwise, let $\hat{T}_1, \dots, \hat{T}_k$ be the child trees of the root view V , and let $\hat{\mathbf{t}}_j^1, \dots, \hat{\mathbf{t}}_j^k$ be the restrictions of \mathbf{t}_j onto the variables of $\hat{T}_1, \dots, \hat{T}_k$, respectively. We recursively compute the probabilities p_1, \dots, p_k of the tuples $\hat{\mathbf{t}}_j^1, \dots, \hat{\mathbf{t}}_j^k$, and set the probability of \mathbf{t}_j to $\prod_{i \in [k]} p_i$.

Consider now a tuple \mathbf{i} over \mathcal{I} . The set of tuples in $Q(\mathcal{O}|\mathbf{i})$ is equal to the Cartesian product $\times_{j \in [n]} Q_j(\mathcal{O}_j|\mathbf{i}_j)$, where $\mathbf{i}_j[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q . Section 7 explains how to enumerate the tuples in this Cartesian product with constant delay, given that the tuples in each $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ can be enumerated with constant delay. It remains to explain how to obtain the probability of a tuple \mathbf{t} that is the concatenation of tuples $\mathbf{t}_1 \in Q_1(\mathcal{O}_1|\mathbf{i}_1), \dots, \mathbf{t}_n \in Q_n(\mathcal{O}_n|\mathbf{i}_n)$. Since the query is without self-joins, the constructed view trees are over disjoint sets of relations, so the tuples in the views of one view tree are independent of the tuples in the views of another view tree. Thus, the probability of \mathbf{t} is the product of the probabilities of $\mathbf{t}_1, \dots, \mathbf{t}_n$.

The maintenance approach for queries in CQAP₀ extends to updates under the probabilistic expectation-variance semantics. Each tuple t in the database is associated with a pair representing the expectation and variance of the tuple's multiplicity. An update of a

tuple t with probability p changes the expected multiplicity by p for an insertion and by $-p$ for a deletion, and increases the variance by $p(1 - p)$. Updating a base relation involves summing up the expectation-variance pairs for t using the \oplus operation from Definition 12.4. The correctness of this approach follows from the linearity of expectation and the linearity of variance given that all updates are independent probabilistic events, see Section 12.

We then propagate the new expectation-variance pair further up in each affected view tree. A single-tuple update from a child results in at most one affected tuple, whose expectation-variance pair is computed using the sum \oplus and product \otimes operations from Definition 12.4 in case of projection and respectively join. These operations take constant time as they only combine two expectation-variance pairs. Thus, updates are propagated in each affected view tree in constant time. The enumeration procedure under the expectation-variance semantics follows a similar approach to that used in set semantics. \square

The same maintenance approach can be applied to queries in CQAP_0 over probabilistic databases under the probabilistic bag semantics for updates. Each tuple is mapped to a probability distribution from a set S of probability distributions, and the constructed views are evaluated using two binary operation, addition and multiplication, defined over S ; Appendix G of the technical report provides further details of these operations [KNOZ25].

Theorem 13.3 can be extended to the probabilistic bag semantics, albeit at a higher cost for maintenance and enumeration. When a tuple is updated, the support of its associated probability distribution grows, making the distribution's size dependent on the database size. Consequently, both addition and multiplication operations over distributions require non-constant time, leading to non-constant time for update propagation and enumeration.

14. 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 [AHV95]. 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 [DK18]. 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 [DHK21]. Further works on queries with access patterns [FLMS99, YLUGM99, DLN07, BLT15, BTCT14] 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 [NL04a, NL04b, LC01].

Dynamic evaluation. Our work generalises the dichotomy for q -hierarchical queries under updates [BKS17] and the complexity trade-offs for queries under updates [KNN⁺19, KNN⁺20, KNOZ20]. The IVM approaches Dynamic Yannakakis [IUV17] and F-IVM [NO18], which is implemented on top of DBToaster [K⁺14], 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 5.1 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 [KNOZ23c] of dynamic query evaluation techniques and how they are recovered by the trade-off [KNOZ20] extended in our work.

Our CQAP₀ dichotomy strictly generalises the one for q -hierarchical queries [BKS17]: The set of q -hierarchical queries is a strict subset of CQAP₀, while there are hard patterns of non-CQAP₀ beyond those for non- q -hierarchical queries.

There are key technical differences between the prior framework for dynamic evaluation trade-off [KNOZ20] 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 [Pea89] and cutset sampling [BD07] 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.

15. CONCLUSION AND OUTLOOK

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.

The work presented in this article can be extended naturally in a number of ways.

Adaptive maintenance. The computational complexity of static query evaluation can be asymptotically improved by combining several execution strategies (query plans, hypertree decompositions, variable orders, or view trees) for the same query, where each strategy is adapted to a different part of the data. Such adaptive strategies can also benefit dynamic query evaluation. Our optimality results for CQAP₀ and CQAP₁ rely in fact on maintaining several view trees for one query, each view tree adapted to a different (light or heavy) part of the data. Extending our optimality results to queries beyond CQAP₀ and CQAP₁ is likely to require data partitioning and adaptive maintenance. Yet the view trees used by our approach to maintain one query can be derived from *one* variable order. Using several variable orders may lead to improved complexity for queries beyond CQAP₀ and CQAP₁. There are several technical challenges to overcome when translating existing adaptive approaches for static query evaluation, e.g., [KNS17, ZDK23, Hu24], to dynamic query evaluation, including: Can the cost of regular rebalancing of heavy-light partitions be amortised over the sequence of updates so that it does not increase the single-tuple update time? Can the multiplicities of each tuple in the views and input relations be maintained as efficiently as the maintenance of the tuple itself? To appreciate the difficulty of addressing the latter question, note the

blow-up in the number of variable orders needed by the PANDA adaptive strategy for static query evaluation [KNS17]: This is exponential in the query size and poly-logarithmic in the data size, so the poly-logarithmic factor in the data size carries over to the enumeration delay. A possible solution is to consider a restriction of PANDA [KCM⁺20], which ensures that different variable orders yield disjoint sets of query output tuples, albeit with a higher complexity than PANDA.

Beyond hierarchical queries. An open research question is the generalisation of our maintenance trade-off for *all* CQAPs as well as of the optimality for *all* CQAPs. The recent trade-off between preprocessing time and enumeration delay for α -acyclic conjunctive queries in the static setting [KNOZ23b] can be extended to also consider the update time and also to apply to arbitrary conjunctive queries and CQAPs.

Support for aggregates. Our approach requires the maintenance of the multiplicities (the number of derivations or counts) of tuples in each view and input relation. Section 13 also shows how to maintain tuple probabilities in case of queries in the CQAP₀ class. More generally, our approach can support any aggregate expressible using the sum and product operations of a ring, as detailed in the F-IVM system [KNOZ24].

Beyond probabilistic databases. Our maintenance approach introduced in Section 13 can be extended beyond probabilistic databases. A special case of probabilistic databases is when all probabilities are $\frac{1}{2}$, so the probability distributions are uniform. This corresponds to the *model counting* problem [GSS21]: Given a Boolean query, in case of arbitrary probability distributions we compute the probability of the query to be true, whereas in case of uniform probability distributions we compute the fraction of those possible worlds where the query is true. An immediate corollary of the dichotomy for conjunctive queries without repeating relation symbols in probabilistic databases [DS04, SORK11] is that model counting can be computed efficiently for hierarchical queries. Our work complements this result in the static setting with a similar result in the dynamic setting: A corollary of Theorem 13.3 is that model counting for CQAP₀ can be maintained with linear preprocessing time, constant update time, and constant enumeration delay. This also immediately implies that further tasks, which can be expressed using model counting, can immediately benefit from the efficient maintenance approach put forward in our work. Prime examples are the computation of the Shapley and Banzhaf values of database tuples, whose computation in relational databases is polynomial-time equivalent to model counting [KOS24] and is in particular tractable for hierarchical queries [LBKS21, DFKM22, DFKM22, ADF⁺23].

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable suggestions, which have significantly improved this article. The authors would also like to acknowledge the UZH Global Strategy and Partnerships Funding Scheme and EPSRC grant EP/T022124/1. This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 682588.

REFERENCES

- [ADF⁺23] Omer Abramovich, Daniel Deutch, Nave Frost, Ahmet Kara, and Dan Olteanu. Banzhaf Values for Facts in Query Answering. *CoRR*, abs/2308.05588, 2023.
- [AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
- [AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [BD07] Bozhena Bidyuk and Rina Dechter. Cutset Sampling for Bayesian Networks. *J. Artif. Intell. Res.*, 28:1–48, 2007.
- [BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*, pages 208–222, 2007.
- [BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.
- [BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017.
- [BLT15] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with Access Patterns and Integrity Constraints. *VLDB*, 8(6):690–701, 2015.
- [BM21] Christoph Berkholz and Maximilian Merz. Probabilistic Databases under Updates: Boolean Query Evaluation and Ranked Enumeration. In *PODS*, pages 402–415, 2021.
- [Bra16] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3):1–26, 2016.
- [BTCT14] Michael Benedikt, Balder Ten Cate, and Efthymia Tsamoura. Generating Low-cost Plans from Proofs. In *PODS*, pages 200–211, 2014.
- [CY12] Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012.
- [DFKM22] Daniel Deutch, Nave Frost, Benny Kimelfeld, and Mikaël Monet. Computing the Shapley Value of Facts in Query Answering. In *SIGMOD*, pages 1570–1583, 2022.
- [DG07] Arnaud Durand and Etienne Grandjean. First-Order Queries on Structures of Bounded Degree are Computable with Constant Delay. *TOCL*, 8(4):21, 2007.
- [DHK21] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Space-Time Tradeoffs for Answering Boolean Conjunctive Queries. *CoRR*, abs/2109.10889, 2021.
- [DK18] Shaleen Deep and Paraschos Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.
- [DLN07] 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.
- [DS04] Nilesch N. Dalvi and Dan Suciu. Efficient Query Evaluation on Probabilistic Databases. In *VLDB*, pages 864–875, 2004.
- [DS11] Arnaud Durand and Yann Strozecki. Enumeration Complexity of Logical Query Problems with Second-Order Variables. In *CSL*, pages 189–202, 2011.
- [FLMS99] 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.
- [GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *PODS*, pages 21–32, 1999.
- [GSS21] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting. In *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 993–1014. IOS Press, 2021.
- [HKNS15] 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.
- [Hu24] Xiao Hu. Output-Optimal Algorithms for Join-Aggregate Queries. *CoRR*, abs/2406.05536, 2024.
- [IUV17] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017.
- [K⁺14] Christoph Koch et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014.
- [KCM⁺20] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Functional Aggregate Queries with Additive Inequalities. *TODS*, 45(4):17:1–17:41, 2020.

- [KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.
- [KNN⁺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.
- [KNN⁺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.
- [KNOZ20] 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.
- [KNOZ23a] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive Queries with Free Access Patterns Under Updates. In *ICDT*, volume 255, pages 17:1–17:20, 2023.
- [KNOZ23b] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Evaluation Trade-Offs for Acyclic Conjunctive Queries. In *CSL*, pages 29:1–29:20, 2023.
- [KNOZ23c] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *LMCS*, 2023.
- [KNOZ24] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. F-IVM: Analytics over Relational Databases under Updates. *VLDB J.*, 33(4):903–929, 2024.
- [KNOZ25] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive Queries with Free Access Patterns under Updates. *CoRR*, arxiv.org/abs/2206.09032v5, 2025.
- [KNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.
- [KNS17] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What Do Shannon-Type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another? In *PODS*, pages 429–444, 2017.
- [KOS24] Ahmet Kara, Dan Olteanu, and Dan Suciu. From Shapley Value to Model Counting and Back. *PODS (PACMOD Journal)*, 2(2):79, 2024.
- [KPP15] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Dynamic Set Intersection. In *WADS*, pages 470–481, 2015.
- [LBKS21] Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. The Shapley Value of Tuples in Query Answering. *LMCS*, 17(3), 2021.
- [LC01] Chen Li and Edward Chang. On Answering Queries in the Presence of Limited Access Patterns. In *ICDT*, pages 219–233, 2001.
- [NL04a] Alan Nash and Bertram Ludäscher. Processing First-Order Queries under Limited Access Patterns. In *PODS*, pages 307–318, 2004.
- [NL04b] Alan Nash and Bertram Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *EDBT*, pages 422–440, 2004.
- [NO18] Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018.
- [NPRR18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-Case Optimal Join Algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.
- [NRR13] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.
- [OHK09] Dan Olteanu, Jiewen Huang, and Christoph Koch. SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *ICDE*, pages 640–651, 2009.
- [OZ15] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *TODS*, 40(1):2:1–2:44, 2015.
- [Pea89] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.
- [SORK11] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
- [Vel14] Todd L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*, pages 96–106, 2014.
- [YLUGM99] Ramana Yerneni, Chen Li, Jeffrey Ullman, and Hector Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In *ICDT*, pages 348–364, 1999.
- [ZDK23] Hangdong Zhao, Shaleen Deep, and Paraschos Koutris. Space-Time Tradeoffs for Conjunctive Queries with Access Patterns. In *PODS*, pages 59–68, 2023.