Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries

Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries

Ahmet Kara[1], Milos Nikolic[2], Dan Olteanu[1], Haozhe Zhang[1]

[1]University of Oxford [2]University of Edinburgh

July 4, 2019

Abstract

We investigate trade-offs in static and dynamic evaluation of hierarchical queries with arbitrary free variables. In the static setting, the trade-off is between the time to partially compute the query result and the delay needed to enumerate its tuples. In the dynamic setting, we additionally consider the time needed to update the query result in the presence of single-tuple inserts and deletes to the input database.

Our approach observes the degree of values in the database and uses different computation and maintenance strategies for high-degree and low-degree values. For the latter it partially computes the result, while for the former it computes enough information to allow for on-the-fly enumeration.

The main result of this work defines the preprocessing time, the update time, and the enumeration delay as functions of the light/heavy threshold and of the factorization width of the hierarchical query. By conveniently choosing this threshold, our approach can recover a number of prior results when restricted to hierarchical queries.

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

1 Introduction

The problems of static evaluation, i.e., computing the result of a query [49, 44, 32, 38, 39], and dynamic evaluation, i.e., maintaining the query result under data updates [33, 17, 34, 11, 26, 28], are fundamental to relational databases.

In this paper we consider a refinement of these problems that decomposes the overall computation time into the *preprocessing time*, which is used to compute a data structure that represents the tuples in the query result, the *update time*, which is the time to update the data structure under inserts and deletes to the input data, and the *enumeration delay*, which accounts for the time to list one distinct tuple after another has been listed from the data structure [19].

There is much prior work on the trade-off between preprocessing time, update time, and enumeration delay for various classes of databases and queries, cf. Figures 17 and 18 for a summary. We next highlight several lines of work that are particularly relevant to this paper.

For any conjunctive query and database of size N, we can construct its factorized result in time $\mathcal{O}(N^{\mathsf{w}})$ and then enumerate its tuples with $\mathcal{O}(1)$ delay [44]. The parameter w is the "factorization" width of the query (denoted $s^{\uparrow}$ in [44])[1].

The (α-)acyclic conjunctive queries admit linear-time preprocessing and delay [9]. The delay becomes constant by either increasing the preprocessing time or restricting the set of free variables. In the first case, the preprocessing time can be as large as the number of relations in the query (same upper bound as for arbitrary

[1]Factorization width is a generalization of the fractional hypertree width [37] from full (i.e., quantifier-free) to arbitrary conjunctive queries. It was subsequently generalized to FAQ-width to capture the complexity of Functional Aggregate Queries over several semirings [2].

conjunctive queries). In the second case, the free variables satisfy the free-connex property [9]. An acyclic conjunctive query admits linear-time preprocessing and constant delay if it is free-connex [9].[2] Assuming that Boolean multiplication of $n \times n$ matrices cannot be accomplished in time $\mathcal{O}(n^2)$, acyclic conjunctive queries that are not free-connex cannot be enumerated with constant delay after linear preprocessing time [9].

For the strict subclass of free-connex acyclic queries called q-hierarchical, maintenance can be achieved with linear-time preprocessing and constant-time update and enumeration delay [11, 26]. Queries that are not q-hierarchical cannot achieve this maintenance complexity [11], unless the Online Matrix-Vector Multiplication conjecture [24] fails. The classical delta processing [17] needs constant-time preprocessing and delay in exchange for update time that may be asymptotically as much as for evaluation from scratch.

These prior works investigated specific points in the static preprocessing-delay space or in the dynamic preprocessing-update-delay space. A natural question is what is the precise relationship between preprocessing, update, and delay. For instance, how much preprocessing time would be needed to achieve, say, $\mathcal{O}(N^{1/2})$ or any other sublinear delay?

1.1 Main Results

This paper characterizes the static and dynamic spaces for hierarchical queries. The class of hierarchical queries is a well-known subclass of acyclic queries:

Definition 1 ([48])**.** *A conjunctive query is* hierarchical *if for any two variables, their sets of atoms in the query are either disjoint or one is contained in the other.*

For instance, the query $Q(\mathcal{F}) = R(A,B), S(B,C)$ is hierarchical, while $Q(\mathcal{F}) = R(A,B), S(B,C), T(C)$ is not, for any $\mathcal{F} \subseteq \{A,B,C\}$. In our study, we do not set any restriction on the set of free variables of a hierarchical query.

Hierarchical queries play a key role in query evaluation in the probabilistic [48, 22], provenance [43], streaming [23, 11], and parallel [35, 25] settings. We express the preprocessing, update, and delay times as functions of a parameter ϵ.

We next state the two main results of this paper.

Theorem 2. *Given a hierarchical query with factorization width* w*, a database of size* N*, and* $\epsilon \in [0,1]$*, we can compute in time* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *a data structure that allows the enumeration of the query result with* $\mathcal{O}(N^{1-\epsilon})$ *delay.*

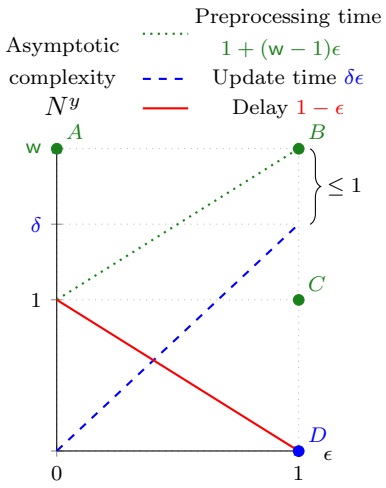
Theorem 2 recovers prior results restricted to hierarchical queries as shown in the right table in Figure 1. For $\epsilon = 1$, we obtain $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time and $\mathcal{O}(1)$ delay as for conjunctive queries on factorized databases [44] (second row). For $\epsilon = 0$, both the preprocessing time and the delay become linear, as for acyclic queries [9] (first row). For free-connex queries, $\mathsf{w} = 1$ and the preprocessing time remains linear regardless of ϵ. We can thus choose $\epsilon = 1$ to obtain constant delay [9] (third row). For bounded-degree databases, first-order queries admit linear-time preprocessing and constant delay [19, 29]. Theorem 2 recovers these complexities for hierarchical queries over bounded-degree databases (fourth row).

The dynamic case generalizes the static case.

Theorem 3. *Given a hierarchical query with factorization width* w *and delta width* δ*, a database of size* N*, and* $\epsilon \in [0,1]$*, we can compute in time* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *a data structure that allows the enumeration of the query result with* $\mathcal{O}(N^{1-\epsilon})$ *delay and can be maintained under a single-tuple update in* $\mathcal{O}(N^{\delta\epsilon})$ *amortized update time.*

Theorem 3 recovers prior works [11, 26] restricted to hierarchical queries as shown in the bottom row of the table in Figure 1: For q-hierarchical queries, we choose $\epsilon = 1$ to obtain linear-time preprocessing and constant-time update and delay.

[2] This result also follows from [44] and the fact that free-connex queries have the factorization width $\mathsf{w} = 1$.



ϵ	•	Queries/DBs	Preprocessing	Delay	Update	Source
0	A	Acyclic CQ/any DB	$\mathcal{O}(N)$	$\mathcal{O}(N)$	–	[9]
1	B	CQ/any DB	$\mathcal{O}(N^{\mathsf{w}})$	$\mathcal{O}(1)$	–	[44]
1	C	Free-connex/any DB ($\mathsf{w} = 1$)	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[9]
1	C	FO/bounded-degree DBs ($\mathsf{w} = 1$)	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[19, 29]
1	C D	q-hierarchical/any DB ($\mathsf{w} = 1$ and $\delta = 0$)	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	[11, 26]

Figure 1: Preprocessing time, amortized update time, and enumeration delay of a hierarchical query with factorization width w and delta width δ parameterized by ϵ (left). Our approach recovers prior results (column Source) restricted to hierarchical queries (right).

1.2 Example

Example 4. Consider the query $Q(A, C) = R(A, B), S(B, C)$ whose relations have size at most N. We can compute Q in time $\mathcal{O}(N^2)$ and then enumerate its tuples with $\mathcal{O}(1)$ delay. Since Q is acyclic, we can alternatively enumerate its tuples with $\mathcal{O}(N)$ delay after $\mathcal{O}(N)$ preprocessing time [9]. It is conjectured that its delay cannot be lowered to constant after linear-time preprocessing, since it is not free-connex.

Q admits $\mathcal{O}(N^{1-\epsilon})$ delay after $\mathcal{O}(N^{1+\epsilon})$ preprocessing time for $\epsilon \in [0, 1]$. We can recover the two cases mentioned above by conveniently choosing ϵ. In case $\epsilon = 1$, we obtain constant delay after quadratic preprocessing. In case $\epsilon = 0$, we obtain linear-time preprocessing and delay. A special case of Q is matrix multiplication for $n \times n$ matrices R and S. In this case, Q cannot be computed in time $o(n^3)$, unless the Combinatorial Matrix Multiplication conjecture (Conjecture 5 in [1]) fails. Our approach achieves $\mathcal{O}(n^3) = \mathcal{O}(N^{3/2})$ by taking $\epsilon = 1/2$: The preprocessing time is $\mathcal{O}(N^{3/2})$ followed by $\mathcal{O}(N^{1/2})$ delay for each of the N elements in the matrix Q.

We partition relations R and S on the join bound variable B: A B-value b is *light* in R if $|\{a \mid (a, b) \in R\}| \leq N^\epsilon$ and *heavy* otherwise (similar for S). Since each heavy B-value is paired with at least N^ϵ A-values in R, there are at most $N^{1-\epsilon}$ heavy B-values. There are four cases to consider: B is light/heavy in each of R and S. We can reduce them to two cases only: either B is light in both relations, or B is heavy in at least one of them. We keep the light and heavy information on B-values in two indicator views $L_B(B)$ and respectively $H_B(B)$: $L_B(B) = R^B(A, B), S^B(B, C)$, where R^B and S^B are the light parts of R and respectively S; $H_B(B) = All_B(B), \nexists L_B(B)$, where $All_B(B) = R(A, B), S(B, C)$.

Figure 2 gives the evaluation and maintenance strategies for the light and heavy cases and for the light/heavy indicators. A strategy is depicted by a view tree, with one view per node such that the head of the view is depicted at the node and its body is the join of its children.

To support light/heavy partitions, we need to keep the degree information of the B-values in the two relations. The light/heavy indicators can be computed in linear time, e.g., for L_B we start with the light parts of R and S, aggregate away A and respectively C and then join them on B. The $\exists$ operator before indicators denotes their use with set semantics.

In case B is light, we can compute the view $V_B(A, C)$ in time $\mathcal{O}(N^{1+\epsilon})$ as follows: We iterate over S and for each of its tuples (b, c), we first check $\exists L_B(b)$ and then fetch the A-values in R paired with b in R. The iteration over S takes linear time and for each b there are at most N^ϵ A-values in R. The view $V_B(A, C)$ is a subset of Q.

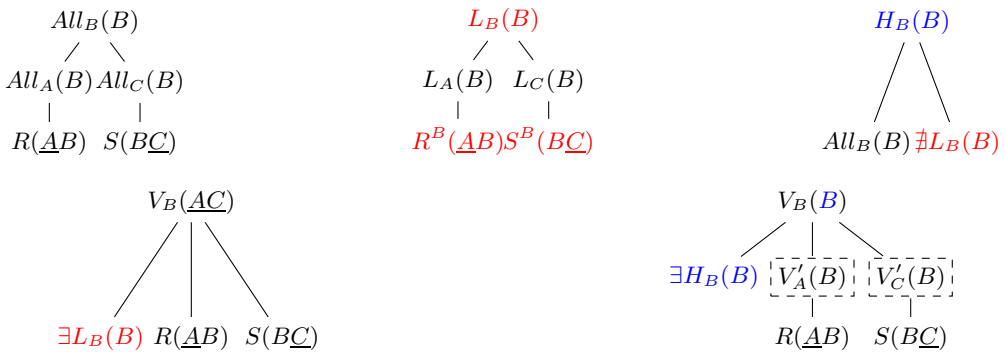


Figure 2: The view trees for $Q(A,C) = R(A,B), S(B,C)$ in Example 4. The dashed boxes enclose views that are only needed in the dynamic case.

In case B is heavy, we construct the view $V_B(B)$ with up to $N^{1-\epsilon}$ heavy B-values. For each such value b, we can trivially enumerate the distinct tuples (a,c) such that $R(a,b)$ and $S(b,c)$ hold. Distinct B-values may, however, have the same tuple (a,c). Therefore, if we were to enumerate such tuples of one B-value after the tuples of another B-value, the same tuple (a,c) may be output several times, which violates the enumeration constraint. To address this challenge, we use the union algorithm [21]. We use the $N^{1-\epsilon}$ buckets of (a,c) tuples, one for each heavy B-value, and an extra bucket $V_B(A,C)$ constructed in the light case. From each bucket of a B-value, we can enumerate the distinct (a,c) tuples with constant delay by looking up into R and S. The tuples in the materialized view $V_B(A,C)$ can be also enumerated trivially with constant delay. We then use the union algorithm to enumerate the distinct (a,c) tuples with delay given by the sum of the delays of the buckets. This gives $\mathcal{O}(N^{1-\epsilon})$ delay for the enumeration of the tuples in the result of Q.

We now turn to the dynamic case. The preprocessing time and delay remain the same as in the static case, while each single-tuple update can be processed in $\mathcal{O}(N^{\epsilon})$ amortized time. To support updates, we need to maintain tuple multiplicities[3] in addition to the degree information of the B-values in the two relations. We also need two additional views to support efficient updates to both R and S; these are marked with the dashed boxes in Figure 2. To simplify the reasoning about updates, we assume that each view tree maintains copies of its base relations.

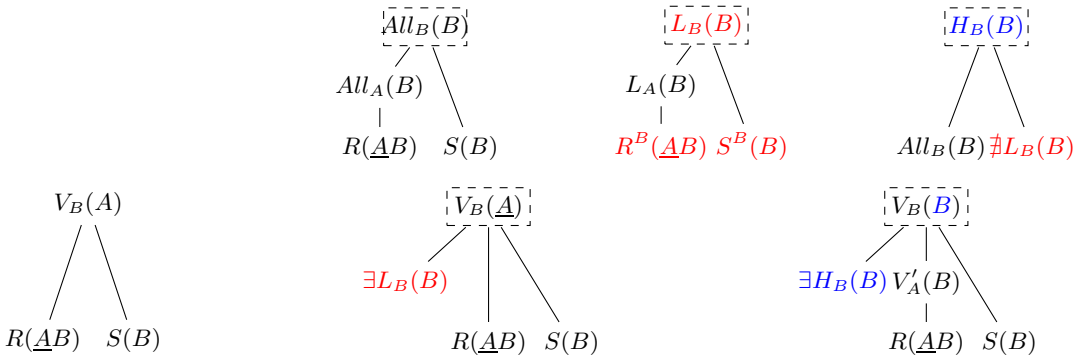
Consider a single-tuple update $\delta R(a,b)$ to R. We maintain each view affected by δR using the hierarchy of materialized views from Figure 2. The changes in those views are expressed using the classical delta rules [17]. We update $V_B(A,C)$ with $\delta V_B(a,C) = \exists L_B(b), \delta R(a,b), S(b,C)$ in time $\mathcal{O}(N^{\epsilon})$ since b is light in S. We update $V'_A(B)$ and $V_B(B)$ with $\delta V'_A(b) = \delta R(a,b)$ and $\delta V_B(b) = \exists H_B(b), \delta V'_A(b), V'_C(b)$ in constant time; the same holds for updating All_B, L_B, and H_B.

The update δR, however, may trigger a new single-tuple change in $\exists L_B$ and $\exists H_B$, affecting $V_B(A,C)$ and respectively $V_B(B)$. The change $\delta(\exists L_B(b))$ is non-zero only when the multiplicity of b in L_B changes from 0 to 1 or vice versa. A change from 0 to 1 can happen when we insert a tuple (a,b) in R with a new B-value b. Then, there is just one matching A-value in R, namely a, and computing $\delta V_B(a,C)$ takes $\mathcal{O}(N^{\epsilon})$ time. A change from 1 to 0 can happen when we delete the last tuple with the B-value b from R. Since there are no more matching A-values in R for the given b, there is no change in V_B. The change $\delta(\exists H_B(b))$ happens for similar reasons, and updating $V_B(B)$ requires constant-time lookups in $V'_A(b)$ and $V'_C(b)$.

The update δR may change the degree of b in R from light to heavy and vice versa. In such cases, we need to rebalance the light part of R and possibly recompute some of the views. Although such rebalancing steps may take time more than $\mathcal{O}(N^{\epsilon})$, they happen periodically and their amortized cost remains the same as for a single-tuple update (Section 5).

To enumerate the tuples in the updated query result, we can now use the same approach used in the static case.

[3]We restrict multiplicities of tuples in the input relations and views to be strictly positive. Multiplicity 0 means the tuple is not present. Deletes are expressed using negative multiplicities. A delete request for tuple t with multiplicity $-m$ is rejected if t's multiplicity in the relation is less than m.



(a) Static case

(b) Dynamic case

Figure 3: The view trees for $Q(A) = R(A, B), S(B)$ in Example 5. The left view tree is for the static case. The view trees on the right, whose root views are enclosed by dashed boxes, are for the dynamic case.

To conclude, in the static case we take $\mathcal{O}(N^{1+\epsilon})$ preprocessing and $\mathcal{O}(N^{1-\epsilon})$ delay. In the dynamic case, preprocessing time and delay remain same, while we take $\mathcal{O}(N^{\epsilon})$ update time.

□

We next discuss the simplest free-connex query that is not q-hierarchical.

Example 5. Consider the query $Q(A) = R(A, B), S(B)$ whose relations have size at most N. Figure 3a shows the single view tree that our approach constructs in the static case. The five view trees in Figure 3b are constructed in the dynamic case.

In the static case, since Q is free-connex, its result can be computed in linear time and then the result tuples can be enumerated with constant delay. Our approach does not partition the relations in the static case. We compute the view $V_B(A)$ in time $\mathcal{O}(N)$ as follows. We iterate over the tuples in relation R and, for each tuple (a, b) in relation R, we look up the multiplicity of b in relation S in constant time. The result can be enumerated from the view $V_B(A)$ with constant delay.

In the dynamic case, we partition relations R and S on the bound join variable B and create the two indicators L_B and H_B (top-middle and top-right in Figure 3b). In the light case, we compute the view $V_B(A)$ (bottom-left in Figure 3b), which is the subset of the query result originating in light B-values in relations R and S. It takes $\mathcal{O}(N)$ time to compute $V_B(A)$: For each (a, b) in R, we first check whether b is light (so in $\exists L_B(b)$) and then check the multiplicity of b in S by a constant-time lookup. In the heavy case, we compute the view $V_B(B)$ (bottom-right in Figure 3b) in linear time using the heavy indicator $\exists H_B$, the input relation S, and the projection $V'_A(B)$ of R on B.

We can enumerate the tuples in the query result with $O(N^{1-\epsilon})$ delay: Since there are at most $N^{1-\epsilon}$ heavy B-values in $V_B(B)$, each with its own list of A-values in R, we need $O(N^{1-\epsilon})$ delay to enumerate the distinct A-values paired with the heavy B-values. In addition, we can enumerate from the view $V_B(A)$ created for the light B-values with constant delay.

A single-tuple update to R triggers constant-time updates to all views. A single-tuple update to S triggers constant-time updates to the indicators and $V_B(B)$. In the light case, the update to $V_B(A)$ is given by $\delta V_B(a) = \exists L_B(b), R(A, b), \delta S(b)$, which requires $\mathcal{O}(N^{\epsilon})$ time since b is light in $\exists L_B$. As explained in Example 4, we may need to rebalance the partitions, which gives an amortized update time of $\mathcal{O}(N^{\epsilon})$.

To conclude, in the static case we take linear-time preprocessing and constant-time delay, while in the dynamic case, we take linear-time preprocessing, $\mathcal{O}(N^{1-\epsilon})$ delay, and $\mathcal{O}(N^{\epsilon})$ update time.

□

2 Preliminaries

2.1 Data Model

Each variable X has a discrete domain $\mathsf{Dom}(X)$ of data values. A tuple $\mathbf{x}$ of data values over schema $\mathcal{X}$ is an element from $\mathsf{Dom}(\mathcal{X}) = \prod_{X \in \mathcal{X}} \mathsf{Dom}(X)$.

A relation R over schema $\mathcal{X}$ is a function $R : \mathsf{Dom}(\mathcal{X}) \to \mathbb{Z}$ mapping tuples over $\mathcal{X}$ to integers such that $R(\mathbf{x}) \neq 0$ for finitely many tuples $\mathbf{x}$. The value $R(\mathbf{x})$ represents the multiplicity of $\mathbf{x}$ in R. 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 consists of relations, and the database size N is the sum of the sizes of the relations in the database.

Given a tuple $\mathbf{x}$ over schema $\mathcal{X}$ and a set $\mathcal{F} \subseteq \mathcal{X}$, we write $\mathbf{x}[\mathcal{F}]$ to denote the tuple of $\mathcal{F}$-values in $\mathbf{x}$. For a relation R over $\mathcal{X}$, a set $\mathcal{F} \subseteq \mathcal{X}$, and a tuple $\mathbf{t} \in \mathsf{Dom}(\mathcal{F})$, we use $\sigma_{\mathcal{F}=\mathbf{t}} R$ to denote the set of tuples in R whose $\mathcal{F}$-values are $\mathbf{t}$, $\sigma_{\mathcal{F}=\mathbf{t}} R = \{\, \mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{F}] = \mathbf{t} \,\}$. We write $\pi_{\mathcal{F}} R$ to denote the set of tuples of $\mathcal{F}$-values in R, $\pi_{\mathcal{F}} R = \{\, \mathbf{x}[\mathcal{F}] \mid \mathbf{x} \in R \,\}$.

2.2 Computational Model

We consider the RAM model of computation. Each relation (view) R over schema $\mathcal{X}$ is implemented by a data structure that stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ over $\mathcal{X}$ with $R(\mathbf{x}) \neq 0$ and needs space linear in the number of such tuples. We assume that this data structure supports (1) looking up, inserting, and deleting entries in constant time, (2) enumerating all stored entries in R with constant delay, and (3) computing $|R|$ in constant time. For instance, a hash table with chaining, where entries are doubly linked for efficient enumeration, can support these operations in constant time on average, under the assumption of simple uniform hashing.

Given a relation R over schema $\mathcal{X}$ and a non-empty set $\mathcal{F} \subset \mathcal{X}$ of variables, we assume there is an index structure on $\mathcal{F}$ that allows: for any $\mathbf{t} \in \mathsf{Dom}(\mathcal{F})$, (4) enumerating all entries in R matching $\sigma_{\mathcal{F}=\mathbf{t}} R$ with constant delay, (5) checking $\mathbf{t} \in \pi_{\mathcal{F}} R$ in constant time, and (6) returning $|\sigma_{\mathcal{F}=\mathbf{t}} R|$ in constant time; (7) inserting and deleting index entries in constant time. Such an index structure can be realized, for instance, as a hash table with chaining where each key-value entry stores a tuple $\mathbf{t}$ of $\mathcal{F}$-values and a doubly-linked list of pointers to the entries in R having the $\mathcal{F}$-values $\mathbf{t}$. Looking up an index entry given $\mathbf{t}$ takes constant time on average, and its doubly-linked list enables enumeration of the matching entries in R with constant delay. Inserting an index entry into the hash table additionally prepends a new pointer to the doubly-linked list for a given $\mathbf{t}$; overall, this operation takes constant time on average. For efficient deletion of index entries, each entry in R stores back-pointers to its index entries (as many back-pointers as there are index structures for R). When an entry is deleted from R, locating and deleting its index entries takes constant time per index.

2.3 Partitioning

We partition relations based on value degree.

Definition 6. *Given a relation R over schema $\mathcal{X}$, a non-empty set $\mathcal{F} \subset \mathcal{X}$ of variables, and a threshold θ, the pair (H, L) of relations is a* partition *of R on $\mathcal{F}$ with threshold θ if it satisfies the following conditions:*

$$\begin{aligned}
&\textit{(union)} && R(\mathbf{x}) = H(\mathbf{x}) + L(\mathbf{x}) \textit{ for } \mathbf{x} \in \mathsf{Dom}(\mathcal{X}) \\
&\textit{(domain partition)} && \pi_{\mathcal{F}} H \cap \pi_{\mathcal{F}} L = \emptyset \\
&\textit{(heavy part)} && \textit{for all } \mathbf{t} \in \pi_{\mathcal{F}} H\textit{: } |\sigma_{\mathcal{F}=\mathbf{t}} H| \geq \tfrac{1}{2}\theta \\
&\textit{(light part)} && \textit{for all } \mathbf{t} \in \pi_{\mathcal{F}} L\textit{: } |\sigma_{\mathcal{F}=\mathbf{t}} L| < \tfrac{3}{2}\theta
\end{aligned}$$

The pair (H, L) is called a strict partition of R on $\mathcal{F}$ with threshold θ if it satisfies the union and domain partition conditions and the following strict versions of the heavy part and light part conditions:

$$\begin{aligned}
&\textit{(strict heavy part)} && \textit{for all } \mathbf{t} \in \pi_{\mathcal{F}} H : \ |\sigma_{\mathcal{F}=\mathbf{t}} H| \geq \theta \\
&\textit{(strict light part)} && \textit{for all } \mathbf{t} \in \pi_{\mathcal{F}} L : \ |\sigma_{\mathcal{F}=\mathbf{t}} L| < \theta
\end{aligned}$$

The relations H and L are the heavy *and* light *parts of R.* □

Assuming $|R| = N$ and the strict partition (H, L) of R on $\mathcal{F}$ with threshold $\theta = N^\epsilon$ for $\epsilon \in [0, 1]$, we have that: $\forall \mathbf{t} \in \pi_{\mathcal{F}} L : |\sigma_{\mathcal{F}=\mathbf{t}} L| < \theta = N^\epsilon$; and $|\pi_{\mathcal{F}} H| \leq \frac{|R|}{\theta} = N^{1-\epsilon}$.

We subsequently denote the light part of R on $\mathcal{F}$ by $R^{\mathcal{F}}$.

2.4 Queries

A conjunctive query (CQ) has the form

$$Q(\mathcal{X}) = R_1(\mathcal{X}_1), \ldots, R_n(\mathcal{X}_n).$$

We denote by: $(R_i)_{i\in[n]}$ the relation symbols; $(R_i(\mathcal{X}_i))_{i\in[n]}$ the atoms; $\mathcal{S}(R_i(\mathcal{X}_i)) = \mathcal{X}_i$ the schema of an atom; $vars(Q) = \bigcup_{i\in[n]} \mathcal{X}_i$ the set of variables; $free(Q) = \mathcal{X}$ the set of *free* variables; $bound(Q) = vars(Q) \setminus free(Q)$ the set of *bound* variables; $njb(Q)$ the set of non-join bound variables; $atoms(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of the atoms; and $atoms(X)$ the (multi)set of the atoms containing X. The query Q is *full* if $free(Q) = vars(Q)$. For a variable set $\mathcal{V}$, the *restriction* of Q to $\mathcal{V}$ denoted by $Q|_{\mathcal{V}}$ is Q where the variables in $vars(Q) - \mathcal{V}$ are dropped from all atoms of Q and $free(Q|_{\mathcal{V}}) = \mathcal{V}$.

Example 7. We consider the query:

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

Let $\mathcal{V} = \{E, D, F\}$ and $\mathcal{V}' = \{A, B, C, E\}$. It holds $Q|_{\mathcal{V}}(E, D, F) = R(), S(D), T(E, F)$ and $Q|_{\mathcal{V}'}(A, B, C, E) = R(A, B, C), S(A, B), T(A, E)$. □

The hypergraph $G = (V = vars(Q), E = atoms(Q))$ of Q has one node per variable and one hyperedge per atom that covers all nodes representing its variables. The query Q is *(α)acyclic* if for every cycle in G the nodes in the cycle are covered by a hyperedge. In this case, Q admits a *join tree* where each node is an atom and if any two nodes have variables in common, then all nodes along the path between them also have these variables. The query Q is *free-connex* if it is acyclic and admits a join tree where the nodes with free variables form a connected subtree; alternatively, if we add the head atom as a node to a join tree the result of the addition is still a join tree [15]. The query Q is *hierarchical* if for any two of its variables, either their sets of atoms are disjoint or one is contained in the other (Definition 1) [48]. Hierarchical queries are acyclic but not necessarily free-connex. The query Q is *q-hierarchical* if it is hierarchical and for every variable $A \in free(Q)$, if there exists a variable B such that $atoms(A) \subset atoms(B)$ then $B \in free(Q)$ [11].

Example 8. The following query is acyclic:

$$Q(A, C, F) = R(A, B, C), S(A, B, D), T(A, E, F), U(A, E, G)$$

A possible join tree is the path $U(AEG) - T(AEF) - R(ABC) - S(ABD)$. It is free-connex since we can extend this join tree as follows: $U(AEG) - T(AEF) - Q(ACF) - R(ABC) - S(ABD)$. It is also hierarchical but not q-hierarchical: The bound variables B and E dominate the free variables C and respectively F. □

2.5 Variable Orders

A variable *depends* on another variable if they occur in the same atom of the query.

Definition 9 (adapted from [44])**.** *A* variable order ω *for a conjunctive query Q is a pair (T, dep_ω) such that:*

- *T is a forest with one node per variable or atom in Q. The variables of each atom in Q lie along the same root-to-leaf path in T. Each atom is a child of its lowest variable.*

- *The function dep_ω maps each variable X to the subset of its ancestor variables in T on which the variables in the subtree rooted at X depend, i.e., for every variable Y that is a child of variable X, $dep_\omega(Y) \subseteq dep_\omega(X) \cup \{X\}$.*

Given a variable order ω for a query Q, we denote by: $vars(\omega)$ all variables; $free(\omega)$ the free variables; $atoms(\omega)$ the set of atoms at the leaves; $\mathsf{anc}(X)$ the set of variables on the path from a variable X to the root excluding X; and $has\_sibling(X)$ whether X has siblings. The subtree of ω rooted at a variable X is denoted by ω_X. The query at X is denoted by $Q_X(\mathcal{F}_X)$: its body is the conjunction of $atoms(\omega_X)$ and $\mathcal{F}_X = free(\omega_X) \cup \mathsf{anc}(X)$. The set $bf(\omega) = \{X \mid X \in bound(Q), free(\omega_X) \neq \emptyset\}$ consists of the bound variables that are ancestors of free variables in ω. A variable order ω is *free-top* if $bf(\omega) = \emptyset$ (previously called d-tree extension [44]). It is *canonical* if the variables of the leaf atom of each root-to-leaf path are the inner nodes of the path. We denote by $\mathsf{freeTopVO}(Q)$, $\mathsf{canonVO}(Q)$, and $\mathsf{VO}(Q)$ the sets of free-top, canonical, and respectively all variable orders of Q.

Example 10. The query from Example 8 admits the canonical variable order $A-\{B-\{C-R(ABC); D-S(ABD)\}; E-\{F-T(AEF); G-U(AEG)\}\}$. This order is not free-top since the bound variables B and E sit on top of the free variables C and respectively F. A free-top order would be: $A-\{C-\{B-\{R(ABC); D-S(ABD)\}\}; F-\{E-\{T(AEF); G-U(AEG)\}\}\}$. This is not canonical: the atom at the leaf of the path $A-C-B-D-S(ABD)$ does not have the variable C. □

Hierarchical queries admit canonical variable orders and q-hierarchical queries admit canonical free-top variable orders.

2.6 Width Measures

Given a full conjunctive query Q, the *fractional* edge cover of Q is a feasible solution $\boldsymbol{\lambda} = (\lambda_R)_{R \in atoms(Q)}$ to the following linear program [6]:

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

The optimal objective value of the above program is denoted as $\rho^*(Q)$ and is called the *fractional edge cover number* of Q. An integral edge cover of Q is a feasible solution to the variant of the above program in which each λ_R with $R \in atoms(Q)$ is restricted to take integers from $\{0,1\}$. The optimal objective value of this program is called the *integral edge cover number* of Q and is denoted as $\rho(Q)$. For a full hierarchical query, its integral edge cover number is equal to its fractional one.

Lemma 11. *For any full hierarchical query Q, it holds $\rho^*(Q) = \rho(Q)$.*

For a query Q and database of size N, the query result can be computed in time $\mathcal{O}(N^{\mathsf{width}})$. If the result is represented as a set of tuples or factorized, then width is the *fractional edge cover number* ρ^* [6] or respectively the *factorization width* w [44]:

Definition 12. *The* factorization width *of a CQ Q is*

$$\begin{aligned} \mathsf{w}(Q) &= \min_{\omega \in \mathsf{freeTopVO}(Q)} \mathsf{w}(\omega) \\ \mathsf{w}(\omega) &= \max_{X \in vars(Q)} \rho^*(Q|_{\{X\} \cup dep_\omega(X)}) \end{aligned}$$

If Q is full, then w becomes the *fractional hypertree width* [37]. *FAQ-width* generalizes w to queries over several semirings [2].

Definition 13. *The* delta width *of a hierarchical query Q is*

$$\delta(Q) = \min_{\omega \in \mathsf{canonVO}(Q)} \delta(\omega)$$
$$\delta(\omega) = \max_{X \in bf(\omega)} \max_{R \in atoms(Q_X)} \rho^*(Q_X|_{vars(Q_X)-njb(Q_X)-\mathcal{S}(R)})$$

For a canonical variable order, the delta width is thus the maximum fractional edge cover number over a set of restriction queries at the bound variables that are above free variables. For each atom in the query, the restriction query is defined by dropping the non-join bound variables and the variables of that atom. By definition, queries that admit free-top canonical variable orders have delta width 0.

Example 14. We consider the hierarchical query

$$Q(\mathcal{F}) = R(A,B,C), S(A,B,D), T(A,E,F), U(A,E,G)$$

from Example 8 for different heads $\mathcal{F}$ and the canonical variable order $A - \{B - \{C - R(ABC); D - S(ABD)\}; E - \{F - T(AEF); G - U(AEG)\}\}$.

- If $\mathcal{F} = \{A, B, E\}$, the query is q-hierarchical and $bf(\omega) = \emptyset$. Hence, we have $\delta(\omega) = 0$.
- If $\mathcal{F} = \{A, C, F\}$, the query is not q-hierarchical but free-connex. We have $bf(\omega) = \{B, E\}$, $njb(Q_B) = \{D\}$, $njb(Q_E) = \{G\}$, the set of atoms in $atoms(Q_B)$ is $\{R, S\}$, and the set of atoms in $atoms(Q_E)$ is $\{T, U\}$. It holds $\rho^*(Q_B|_{vars(Q_B)-njb(Q_B)-\mathcal{S}(R)}) = 1$ for any atom $R \in atoms(Q_B)$. For any atom $R \in atoms(Q_E)$, we have $\rho^*(Q_E|_{vars(Q_E)-njb(Q_E)-\mathcal{S}(R)}) = 1$. It follows that $\delta(\omega) = 1$.
- If $\mathcal{F} = \{B, C, D, E, F, G\}$, the query is not free-connex. We have $bf(Q) = \{A\}$, $njb(Q_A) = \emptyset$, and $atoms(Q_A)$ consists of all atoms in the query. For any atom R, we have $\rho^*(Q_A|_{vars(Q_A)-njb(Q_A)-\mathcal{S}(R)}) = 3$. Hence, the delta width of ω is $\delta(\omega) = 3$.

□

Proposition 15. *Given a hierarchical query Q with factorization width w and delta width δ, it holds that $\mathsf{w} - 1 \leq \delta$.*

3 Preprocessing

In the preprocessing stage, we construct a data structure that represents the result of a given hierarchical query. The data structure consists of a set of materialized views that are organized as view trees, based on the structure of the query and the degree of data values in base relations. Each view tree defines a strategy for computing one part of the query result. We construct different sets of view trees for the static and dynamic evaluation of a given hierarchical query.

3.1 Query Factorization via View Trees

Given a hierarchical query $Q(\mathcal{F})$ and a canonical variable order ω for Q, the function FactVT in Figure 4 constructs a view tree that is a factorized representation of the query result. We proceed recursively on the structure of ω and construct a view $V_X(\mathcal{F}_X)$ at each inner node X; the leaves correspond to the atoms in the query and are preserved. This view is defined by the join of the atoms below that node, or equivalently by the joins of its child views that are each defined by the joins of the atoms below their nodes, cf. Figure 5. Its free variables $\mathcal{F}_X$ include the ancestors of X in ω, as they are needed for joins at nodes above X. In case X is free in Q, then $\mathcal{F}$ includes X as well. In case X is bound in Q, then it is aggregated away and $\mathcal{F}$ includes the free variables of Q below X in ω. This means that the free variables of Q are propagated through the views towards the root until they encounter an ancestor free variable, whereas the bound variables are not

FACTVT(symbol V, variable order ω, free variables $\mathcal{F}$) : view tree	
switch ω:	
$R(\mathcal{Y})$	1 **return** $R(\mathcal{Y})$
X $\omega_1 \ \cdots \ \omega_k$	2 **let** $\mathcal{F}_X = \begin{cases} \mathsf{anc}(X) \cup \{X\} & , \ X \in \mathcal{F} \\ \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega)) & , \ X \notin \mathcal{F} \end{cases}$ 3 **let** $subtrees = \{\text{FACTVT}(V, \omega_i, \mathcal{F})\}_{i \in [k]}$ 4 **return** $\text{NEWVT}(X, V_X, \mathcal{F}_X, subtrees)$

Figure 4: Construction of a factorized view tree for a variable order ω of a free-connex hierarchical query with free variables $\mathcal{F}$. V is used to prefix view names.

NEWVT(node X, symbol V, free variables $\mathcal{F}$, trees $\mathcal{T}$) : view tree
1 **let** $\mathcal{T} = \{T_1, \ldots, T_k\}$
2 **let** $V_i(\mathcal{F}_i) =$ root of $T_i, \forall i \in [k]$
3 **let** $V_X(\mathcal{F}) = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$
4 **let** $tree = \begin{cases} T_1 & , \ k = 1 \wedge \mathcal{F} = \mathcal{F}_1 \\ V_X(\mathcal{F}) \ / \ \backslash \ T_1 \ \cdots \ T_k & , \ \text{otherwise} \end{cases}$
5 **if** $mode =$ 'dynamic' $\wedge \ \mathsf{has\_sibling}(X) \wedge \mathsf{anc}(X) \subset \mathcal{F}$
6 **return** $\text{NEWVT}(X, V'_X, \mathsf{anc}(X), \{tree\})$
7 **return** $tree$

Figure 5: Construction of a view tree at node X with root symbol V, free variables $\mathcal{F}$, and children $\mathcal{T}$.

propagated. The obtained view tree would then have the upper levels only with views over the free variables of Q.

The presence of bound variables may thus harden the evaluation because the free-connex or q-hierarchical properties may no longer hold. For a canonical variable order of a hierarchical query, the free-connex property fails if, below a bound join variable, there are several free variables whose sets of atoms are not the same. Indeed, assume two branches out of a bound join variable X and with free variables Y and respectively Z. Then, there are two atoms in Q whose sets of variables include $\{X, Y\}$ and respectively $\{X, Z\}$, while $\{Y, Z\}$ are included in the head atom of Q. This creates a cycle in the hypergraph of Q, which means that Q is not free-connex. Furthermore, for q-hierarchical queries the free variables dominate the bound variables so the former can only occur above the latter in any canonical variable order, i.e., the variable orders are free-top. For any free-connex hierarchical query, each view created by FACTVT is thus defined over variables from one atom of the query and can be materialized in linear time. We can thus recover the linear-time preprocessing for such queries used for static [9] and dynamic [11, 26] evaluation. The function FACTVT is reminiscent of the factorized incremental view maintenance approach (F-IVM) [42].

Example 16. Consider the free-connex query $Q(B, D) = R(A, B, C), S(A, B, D), T(A, E)$ and its canonical variable order in Figure 6. We construct the view tree bottom-up as follows. At node C, we create the view $V_C(A, B)$ that aggregates away the bound variable C; A and B are its ancestors and kept by V_C to define views up the tree. At node D, we do not aggregate away since D is a free variable. The view at this node is the same as $S(A, B, D)$. At node B, we create the view $V_B(A, B, D) = V_C(A, B), S(A, B, D)$. It keeps both B and D as they are free and also A for joins higher in the tree. At node E, we create the view $V_E(A)$ that aggregates away the bound variable E. Finally, at node A, we aggregate away the bound variable A and keep the free variables B and D. Each view can be computed in linear time using two operations: aggregate away bound variables and semi-join reduction. □

Figure 6: Canonical variable order and view tree for the query $Q(B, D) = R(A, B, C), S(A, B, D), T(A, E)$ in Example 16.

IndicatorVTs(variable order ω) : pair of view trees

1 **let** X = root of ω
2 **let** $\mathcal{F} = \mathsf{anc}(X) \cup \{X\}$
3 **let** $alltree = \text{FactVT}(All, \omega, \mathcal{F})$
4 **let** $ltree = \text{FactVT}(L, \omega^{\mathcal{F}}, \mathcal{F})$
5 **let** $allroot$ = root of $alltree$
6 **let** $lroot$ = root of $ltree$
7 **let** $htree = \text{NewVT}(X, H_X, \mathcal{F}, \{allroot, \nexists lroot\})$
8 **return** $(htree, ltree)$

Figure 7: Construction of a pair of heavy and light indicator views for a variable order ω of a hierarchical query. The variable order $\omega^{\mathcal{F}}$ shares the same structure as ω, but each atom $R(\mathcal{Y})$ is replaced with $R^{\mathcal{F}}(\mathcal{Y})$ denoting the light part of relation R partitioned on $\mathcal{F}$.

3.2 Skew-Aware View Trees

The factorized query approach FactVT is sufficient in case of free-connex (q-hierarchical) queries as it constructs a linear-time data structure that allows for constant-time enumeration delay (and update). For arbitrary hierarchical queries, it however gives too high preprocessing time that cannot be traded for delay nor update time. We next introduce an adaptive approach that exhibits such a trade-off.

In case of a bound join variable X that violates the free-connex or q-hierarchical property, we create two evaluation strategies: The first strategy materializes a subset of the query result by considering only the light values over the set of variables $\mathsf{anc}(X) \cup \{X\}$ in the variable order. The bounded degree of these values justifies storing this result in the listing representation, where the bound variables in the subtree rooted at X in the variable order are aggregated away. The second strategy computes a compact representation of the rest of the query result corresponding to those values over $\mathsf{anc}(X) \cup \{X\}$ that have a high degree in at least one relation. This second strategy treats X as a free variable and proceeds recursively to resolve further bound variables located below X in the variable order and to potentially fork into more strategies. The union of all these strategies precisely cover the entire query result, yet not necessarily disjointly. To enumerate the distinct tuples in the query result, we then use an adaptation of the union algorithm where the delay is given by the number of high-degree values of the variables we partitioned on and by the number of strategies. Section 1.2 exemplifies this approach for two queries.

Heavy and Light Indicators. We construct alternative view trees based on the degree of values of bound variables. To facilitate this process, we compute heavy and light indicator views containing disjoint sets of values: the former contains values that exist in all relations and have a high degree in at least one relation; the latter contains values that exist in all relations and have a low degree in all relations. Indicator views have set semantics and allow us to rewrite the query as an equivalent union of two subqueries computing partial results for high-degree and low-degree data values.

We compute heavy and light indicators for each bound join variable X that violates the free-connex property in the static case or the q-hierarchical property in the dynamic case. However, partitioning the

τ(variable order ω, free variables $\mathcal{F}$) : set of view trees

switch ω:

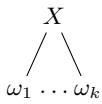
$R(\mathcal{Y})$	1	**return** $\{R(\mathcal{Y})\}$
	2	**let** $keys = \mathsf{anc}(X) \cup \{X\}$
	3	**let** $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega))$
	4	**let** $Q_X(\mathcal{F}_X) =$ join of $atoms(\omega)$
	5	**if** ($mode =$ 'static' $\wedge\ Q_X(\mathcal{F}_X)$ is free-connex) $\vee$
	6	($mode =$ 'dynamic' $\wedge\ Q_X(\mathcal{F}_X)$ is q-hierarchical)
	7	**return** $\{\textsc{FactVT}(V, \omega, \mathcal{F}_X)\}$
	8	**if** $X \in \mathcal{F}$
	9	**return** $\{\textsc{NewVT}(X, V_X, keys, \{T_1, ..., T_k\})$
	10	$\quad \mid T_i \in \tau(\omega_i, \mathcal{F})_{i \in [k]}\}$
	11	**let** $(H_X, L_X) =$ roots of $\textsc{IndicatorVTs}(\omega)$
	12	**let** $htrees = \{\textsc{NewVT}(X, V_X, keys, \{\exists H_X, T_1, ..., T_k\})$
	13	$\quad \mid T_i \in \tau(\omega_i, \mathcal{F})_{i \in [k]}\}$
	14	**let** $bvars =$ bound non-join vars in $atoms(\omega)$
	15	**let** $vrels = \{\textsc{NewVT}(R, V_R, \mathsf{anc}(R) \setminus bvars, \{R\})$
	16	$\quad \mid R \in atoms(\omega)\}$
	17	**let** $ltree = \textsc{NewVT}(X, V_X, \mathcal{F}_X, \{\exists L_X\} \cup vrels)$
	18	**return** $htrees \cup \{ltree\}$

Figure 8: Construction of skew-aware view trees for a canonical variable order ω of a hierarchical query with free variables $\mathcal{F}$. The global parameter $mode \in \{$'static', 'dynamic'$\}$ specifies their execution mode.

query result only based on the degree of X-values may blow up the enumeration delay: the path from X to the root may contain several bound join variables, each creating buckets of values per bucket of their ancestors, thus leading to an explosion of the number of buckets that need to be unioned together during enumeration. However, one remarkable property holds for hierarchical queries: each base relation located in the subtree rooted at X contains X but also all the ancestors of X. Thus, by partitioning each relation jointly on X and its ancestors, we can ensure the enumeration delay remains linear in the number of distinct high-degree tuple values of X and its ancestors.

Figure 7 shows how to construct a pair of view trees for computing the indicators for the root X of a variable order ω and the ancestors of X. This variable order ω may be a subtree in the variable order of a hierarchical query, thus $\mathsf{anc}(X)$ may be non-empty. We first construct a view tree for computing all tuples of values with variables $\mathcal{F} = \mathsf{anc}(X) \cup \{X\}$ over the join of the relations from ω. We then build a similar view tree for computing the light indicator for $\mathcal{F}$ using a modified variable order $\omega^{\mathcal{F}}$ of the same structure as ω but with each relation R replaced by the light part of R partitioned on $\mathcal{F}$. Finally, the view tree of the heavy indicator computes the difference of all $\mathcal{F}$-values and those from the light indicator.

Example 17. Figure 2 shows the view trees for computing the indicator views at the root B of a variable order of the query $Q(A, C) = R(A, B), S(B, C)$. Figure 3 gives similar indicator views for the query $Q(A) = R(A, B), S(B)$. Figure 9 has two pairs of indicators computed at nodes A and B of the given variable order for the query $Q(C, D, E, F) = R(A, B, D), S(A, B, E), T(A, C, F), U(A, C, G)$. □

View Trees with Indicators. Figure 8 gives the algorithm for constructing a set of view trees τ for a variable order ω of a hierarchical query Q with a set of free variables $\mathcal{F}$. The algorithm traverses the variable order top-down, maintaining the invariant that all ancestors of a node are free variables (or treated as such in case of bound join variables whose domain consists of high-degree values).

At node X, the set of free variables consists of all the ancestors of X and variables that appear in the

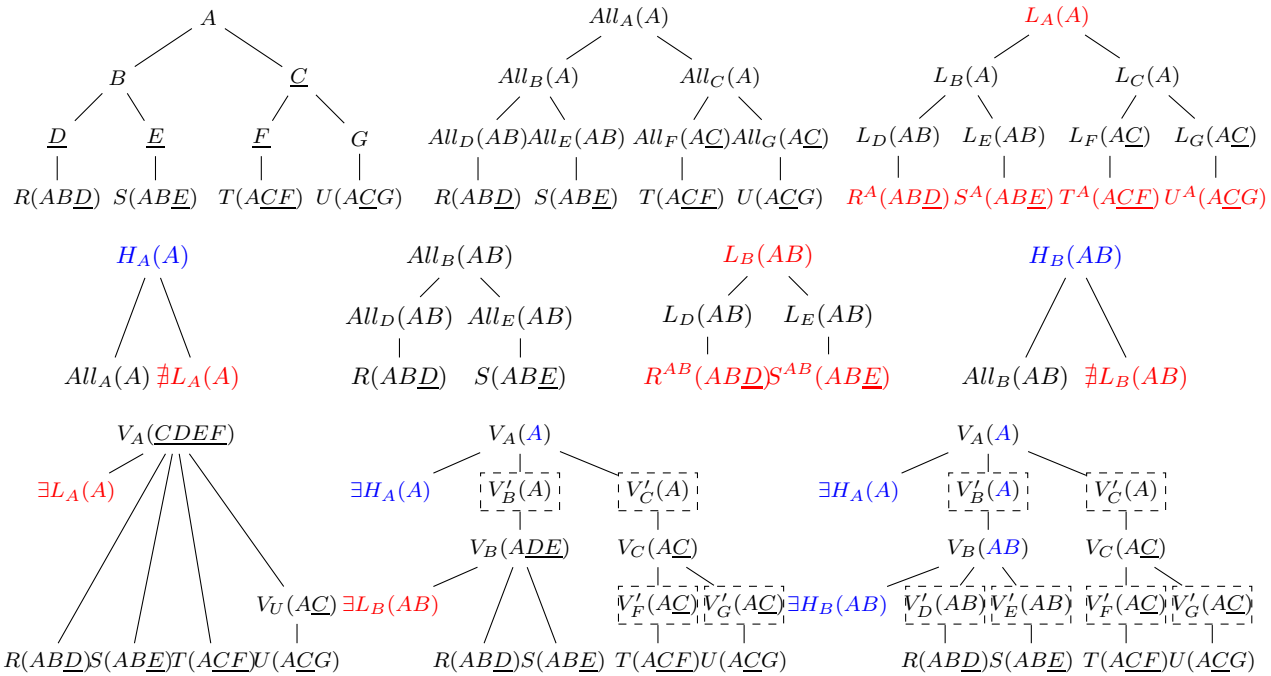


Figure 9: Canonical variable order for the query $Q(C, D, E, F) = R(A, B, D), S(A, B, E), T(A, C, F), U(A, C, G)$ (top left). Two pairs of indicator views (H_A, L_A) and (H_{AB}, L_{AB}) (top and middle row). The three view trees constructed for the query (bottom row). The views with a dashed box are only need for dynamic query evaluation.

subtree rooted at X (line 3). If the residual query Q_X at node X (line 4) is already free-connex in the static case or q-hierarchical in the dynamic case, we return a factorized view tree for computing the result of Q_X (lines 5-7). If X is a free variable, we recursively compute a set of view trees for each child of X (lines 8-10). For each combination of the child view trees, we form a new view joining the roots of the child view trees and using X and its ancestors as free variables (lines 8-10). If X is a bound variable, the algorithm creates two evaluation strategies for the residual query Q_X, based on the degree of values of X and its ancestors in the base relations of Q_X. We construct the heavy and light indicators for X and its ancestors (line 11). The heavy indicator restricts the joins of the child views to only high-degree tuples of values of X and its ancestors (lines 12-13). In the light case, we first aggregate away all bound non-join variables from each relation (lines 14-16). Then, we construct a single view tree joining the light indicator with these preprocessed relations (line 17).

Example 18. Consider the non-free-connex query

$$Q(C, D, E, F) = R(A, B, D), S(A, B, E), T(A, C, F), U(A, C, G).$$

Figure 9 shows the variable order of Q (top-left), the indicator relations created at nodes A and B (first and second rows), and the view trees constructed by our algorithm for both the static and dynamic evaluation of Q. The views with dashed boxes are created only in the dynamic case.

We start from the root A in the variable order. Since Q is not free-connex (thus non-q-hierarchical too) and A is bound, we create the view trees for the indicators $H_A(A)$ and $L_A(A)$. Computing H_A and L_A takes linear time.

In the light case for A, we create the view $V_A(C, D, E, F)$ by joining L_A with the preprocessed input relations (bottom-left). Computing $V_U(A, C)$ takes linear time. We compute $V_A(C, D, E, F)$ in time $\mathcal{O}(N^{1+2\epsilon})$: For each (a, b, d) tuple in R, if a exists in L_A, then we iterate over at most N^ϵ (a, b, e) values in S and at

most N^ϵ (a, c, f) values in T, and finally do a lookup in V_U with (a, c). The view $V_A(C, D, E, F)$ allows constant delay enumeration of its result.

In the heavy case for A, we recursively process the subtrees of A in ω and treat A as free. The right subquery, $Q_C(A, C, F) = T(A, C, F), U(A, C, G)$ is free-connex and q-hierarchical, thus we can compute its factorized view tree with the root $V_C(A, C)$ in the static case and the root $V'_C(A)$ in the dynamic case (bottom-middle, bottom-right) in linear time. The left subquery $Q_B(A, D, E) = R(A, B, D), S(A, B, E)$, however, is neither free-connex nor q-hierarchical. Since B is bound, we create the indicator relations $H_B(A, B)$ and $L_B(A, B)$ in linear time. We distinguish two new cases: In the light case for (A, B), we construct the view tree for $V_B(A, D, E) = L_B(A, B), R(A, B, D), S(A, B, E)$ (bottom-middle) and compute V_B in time $\mathcal{O}(N^{1+\epsilon})$ by iterating over R, checking in L_B, and iterating over at most N^ϵ E-values in S for each (a, b). In the heavy case for (A, B), we process the subtrees of B and consider B as free variable. The two subqueries, $Q_D(A, B, D) = R(A, B, D)$ and $Q_E(A, B, E) = S(A, B, E)$ need no extra views.

Overall, we create three view trees for Q and two sets of view trees for the indicator relations at A and B. The time needed to compute these view trees is $\mathcal{O}(N^{1+2\epsilon})$. □

Our algorithm constructs a set of view trees for a hierarchical query. Each of these view trees represents the execution plan for a query defined by the join of the leaves of the view tree. As stated next, our algorithm effectively rewrites Q into an equivalent union of queries.

Proposition 19. *Let $\{T_1, \ldots, T_k\} = \tau(\omega, \mathcal{F})$ be the set of view trees constructed by the algorithm in Figure 8 for a given hierarchical query $Q(\mathcal{F})$ and a canonical variable order ω for Q. Let $Q_i(\mathcal{F})$ be the query defined by the conjunction of the leaf atoms in $\mathcal{T}_i$, $\forall i \in [k]$. Then, $Q(\mathcal{F}) \equiv \bigcup_{i\in[k]} Q_i(\mathcal{F})$.*

We are now ready to state the complexity of materializing the views in the view trees for a given hierarchical query.

Proposition 20. *Given a hierarchical query $Q(\mathcal{F})$ with factorization width w, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0, 1]$, the views in the set of view trees $\tau(\omega, \mathcal{F})$ can be materialized in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$.*

4 Enumeration

For any hierarchical query, Section 3 constructs a data structure consisting of a set of view trees that compactly represent the query result. In this section, we show how to enumerate the distinct tuples in the query result using view trees that support the *open/next/close* iterator model.

The call $T.open(ctx)$ sets the range of the iterator of a node T to those tuples in its view that agree with the context *ctx* and also positions the iterator at the first tuple in this range, cf. Figure 10. This context is the current tuple in the view at the parent node. An invariant of the view tree construction is that the variable set at the parent view is included in the variable set at each child. The context tuple is therefore a prefix of the tuples at the children's views. By construction, each tuple of a view at a node can always be extended by tuples of views at children. The *open* call is recursively propagated down the view tree with an increasingly more specific context. There are two cases that need special attention. In case the variables of the view V at T are all the free variables of the query present in the tree rooted at T, then there is no need to open the nodes below, since V already provides the possible tuples over these variables. This is the case of the parents of light indicators, e.g., the view $V_A(CDEF)$ in Figure 9. Heavy indicators e.g., the views $V_A(A)$ and $V_B(A, B)$ in Figure 9, also require special treatment. If T has as child a heavy indicator $\exists H$, it represents possibly overlapping subsets of the query result in the contexts given by the different tuples $h \in \exists H$. We *ground* the heavy indicator by creating an iterator for each heavy tuple agreeing with the parent context *ctx* and keep this iterator in a shallow copy of T (the content of views under T is not copied).

The time for the *open* calls is dominated by grounding. Since the overall size of the heavy indicators in a view tree is $\mathcal{O}(N^{1-\epsilon})$, it takes $\mathcal{O}(N^{1-\epsilon})$ time to effect all *open* calls. Seeking a context in a view is a constant-time lookup.

A $T.close()$ call resets the iterators at node T.

```
T.open (tuple ctx)
1  let V(F) = root of T
2  V.seek(ctx)
3  T.value := V.nextValue()
4  if (T has children T_1, ..., T_k)
5    if (F = T.freeVars) return
6    if (∃i ∈ [k] : T_i = ∃H)
7      ∃H.seek(ctx)
8      foreach h ∈ ∃H do
9        T^(h) := shallow copy of T
10       T^(h).open(h)
11     T.buckets := (T^(h))_{h∈∃H}
12   else
13     foreach i ∈ [k] do T_i.open(T.value)
```

Figure 10: Open the view iterators in a view tree.

After the first open call to the root of a view tree T, we can enumerate the distinct tuples by calling $T.next()$, cf. Figure 11. The *next* call propagates recursively down T and observes the same cases as the *open* call. In case T's view V already covers all free variables in the subtree rooted at T, then it is sufficient to enumerate from V. In case T has as child a heavy indicator, we return the next tuple from the union of all its groundings using the UNION algorithm (Figure 12 in Section 4.1). Otherwise, we synthesize the returning tuple out of the tuples at the iterators of T's children. In the context given by the current tuple at T's view, we iterate over the Cartesian product of the tuples produced by T's children using the PRODUCT algorithm (Figure 12 in Section 4.1).

The time to produce the next tuple from a view tree is dominated by the UNION algorithm, whose delay is given by the sum of the delays of its input view trees. View iterators need constant delay, same for trees of such views in the absence of the grounding of heavy indicators. Since the overall size of the heavy indicators in a view tree is $\mathcal{O}(N^{1-\epsilon})$, the overall time taken by a next call is $\mathcal{O}(N^{1-\epsilon})$.

So far we discussed the case of enumerating from one view tree. In case of a set of view trees we again use the UNION algorithm. In case the query has several connected components, i.e., it is a Cartesian product of hierarchical queries, we use the PRODUCT algorithm with an empty context.

Proposition 21. *The tuples in the result of a hierarchical query $Q(\mathcal{F})$ over a database of size N can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay using the view trees $\tau(\omega, \mathcal{F})$ for a canonical variable order ω of Q.*

4.1 The Union and Product Algorithms

The UNION algorithm is given in Figure 12. It is an adaptation of prior work [21]. It takes as input n view trees that represent possibly overlapping sets of tuples over the same relation and returns a tuple in the union of these sets that is distinct from all tuples returned before.

We first explain the algorithm on two views T_1 and T_2 that have been already open and with their iterators positioned at the first respective tuples. On each call, we output one tuple or **EOF**. We check whether the next tuple t_1 in T_1 is also present in T_2. If it is, then we output the next tuple t_2 in T_2; otherwise, we output t_1. In case T_1 is exhausted, we output the next tuple in T_2 or **EOF** in case T_2 is also exhausted.

In case of $n > 2$ views, we consider one view defined by the union of the first $n-1$ views and a second view defined by T_n, and we then reduce the general case to the previous case of two views.

```
T.next ()
1  let V(F) = root of T
2  if (T has no children or F = T.freeVars)
3    t := T.value; T.value := V.nextValue(); return t
4  let {T_1, ..., T_k} = children of T
5  if (∃i ∈ [k] : T_i = ∃H)
6    while ((t := UNION(T.buckets)) ≠ EOF) do
7      T.value := t; return t
8  else
9    while (T.value ≠ EOF) do
10     t_[k] := PRODUCT(T_1, ..., T_k, T.value)
11     if (t_[k] ≠ EOF) return T.value ∘ t_[k]
12     T_1.close(); T.value := V.nextValue(); T_1.open(T.value)
13 return EOF
```

Figure 11: Find the next tuple in a view tree.

```
UNION(view trees T_1, ..., T_n) : tuple
1  if (n = 1)
2    return T_n.next()
3  while ( (t_[n-1] := UNION(T_1, ..., T_{n-1})) ≠ EOF)
4    if (T_n.lookup(t_[n-1]) ≠ EOF)
5      if ( (t_n = T_n.next()) ≠ EOF)
6        return t_n
7    else
8      return t_[n-1]
9  while ( (t_n := T_n.next()) ≠ EOF) do
10   return t_n
11 return EOF
```

```
PRODUCT(view trees T_1, ..., T_k, tuple ctx) : tuple
1  while (T_1.value ≠ EOF) do
   ...
2    while (T_{k-1}.value ≠ EOF) do
3      while (T_k.value ≠ EOF) do
4        t := ⨂_{i∈[k]} π_{T_i.freeVars − S(ctx)} T_i.value
5        T_k.next()
6        return t
7      T_k.close(); T_k.open(ctx); T_{k-1}.next()
   ...
8    T_2.close(); T_2.open(ctx); T_1.next()
9  return EOF
```

Figure 12: Left: Find the next tuple in a union of view trees. Right: Find the next tuple in a product of view trees. In case $k = 1$, the innermost loop is executed.

The delay of this algorithm is given by the delay of iterating over each view and of the cost of the lookup into the views. These per-views costs are constant for listing and factorized representations of these views [44]; in particular, the latter is relevant in the context of our work as the view trees are factorized representations of the query result (modulo the treatment of the heavy indicators, which is done differently as explained in the main body of the paper). The delay in this case is then the sum of the delays of the n views, which is $\mathcal{O}(n)$. In our paper, we employ the UNION algorithm in two cases: (1) on the set of view trees obtained after grounding the heavy indicators; and (2) on the set of view trees obtained by using skew-aware indicators in the preprocessing stage. In the first case, the number of the view trees is in $\mathcal{O}(N^{1-\epsilon})$, since the number of heavy tuples in any heavy indicator view is at most $N^{1-\epsilon}$. In the second case, the number of view trees does not depend on the database size N, but it may depend exponentially on the number of bound join variables in the input hierarchical query.

The PRODUCT algorithm is given in Figure 12. It takes as input a set of view trees $T_1, \ldots, T_k$ and a context, which is the current tuple in the parent view, and outputs the next tuple in the Cartesian product of the k views given the context.

In case $k = 1$, we execute the innermost loop for T_k: On a call, we take the current tuple in T_k and project away the fields that are in common with the context tuple. Before we return the tuple, we advance the iterator to the next tuple in T_k. Eventually, we reach the end of the iterator for T_k, in which case we return **EOF**.

In case $k > 1$, we hold the current values for $T_1, \ldots, T_{k-1}$ and iterate over T_k. Whenever T_k reaches **EOF**, we reset it and advance the iterator for T_{k-1}. We return the concatenation of the current values of all iterators, projected onto the variables that are not in the schema of the context tuple (since those fields are already as in the context). The concatenation operator is $\circ$.

The delay for a PRODUCT call is given by the sum of the delays of the k input view trees. We use this algorithm in two cases: (1) enumerating from a view with several children in a tree (in which case the context is given the current tuple in the view); (2) a collection of view trees, one per connected component of the input query (in which case the context is the empty tuple). In both cases, the number of parameters to the PRODUCT call is independent of the size of the database and only dependent on the number of atoms and respectively of connected components in the input query. This means that the delay (in data complexity) is the maximum delay of any of its parameter view trees, which is $\mathcal{O}(N^{1-\epsilon})$.

5 Updates

We present our strategy for maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ constructed for a variable order ω of a hierarchical query $Q(\mathcal{F})$ under updates to input relations. We first specify the procedure for processing a single-tuple update to any input relation and then specify the procedure for processing a sequence of such updates.

We write $\delta R = \{\mathbf{x} \to m\}$ to denote a single-tuple update δR mapping the tuple $\mathbf{x}$ to the non-zero multiplicity $m \in \mathbb{Z}$ and any other tuple to 0; that is, $|\delta R| = 1$. Inserts and deletes are updates represented as relations in which tuples have positive and negative multiplicities. We assume that after applying an update to the database, all input relations and views contain no tuples with negative multiplicities.

Compared to static evaluation, our strategy for dynamic evaluation may construct additional views to support efficient updates to *all* input relations. For instance, in Figure 2, the view tree created for the case of heavy B-values (bottom-right) contains the views V'_A and V'_C defined over R and respectively S. These views are needed only in the dynamic evaluation to avoid iteration over the A-values in R for updates to S and $\exists H_B$ and over the C-values in S for updates to R and $\exists H_B$. Figure 5 (lines 5-6) gives the rule for constructing such views: If node X has a sibling, then we create an extra view that aggregates away X to avoid iterating over the X-values for updates coming via the siblings of X.

5.1 Processing a Single-Tuple Update

An update δR to a relation R may affect multiple view trees in the set of view trees constructed by our algorithm from Figure 8.[4] We apply δR to each such view tree in sequence, by propagating changes along the path from the leaf R to the root of the view tree. For each view on this path, we update the view result with the change computed using the standard delta rules [17] (cf. Examples 4 and 5). To simplify the reasoning about the maintenance task, we assume that each view tree has a copy of its input relations. We use the procedure APPLY$(T, \delta R)$ to propagate an update δR in a view tree T; if T does not refer to R, the procedure has no effect.

Updates to indicator views, however, may trigger further changes in the views constructed over them. Consider, for instance, the light indicator $L_B(B)$ constructed over the light parts $R^B(A, B)$ and $S^B(B, C)$ in Figure 2. An insert $\delta R = \{(a, b) \to 1\}$ into R^B may change the multiplicity of b in L_B from 0 to non-zero, thus changing $\exists L_B$ and $V_B(A, C)$. But if the multiplicity $L_B(b)$ stays 0 or non-zero after applying the update δR, then $\exists L_B$ also stays unchanged.

[4] We focus here on updates to self-join free hierarchical queries. In case a relation R occurs several times in a query, we represent an update to R as a sequence of updates to each occurrence of R in the query.

UPDATEINDTREE(indicator tree T_{Ind}, update δR) : indicator change

```
1  let δR = {x → m}
2  let I(F) = root(T_Ind)
3  let key = x[F]
4  let #before = I(key)
5  APPLY(T_Ind, δK)
6  if (#before = 0 ∧ I(key) > 0) return {key → 1}
7  if (#before > 0 ∧ I(key) = 0) return {key → −1}
8  return ∅
```

Figure 13: Updating an indicator view tree T_{Ind} for a single-tuple update δR to relation R. Returns the change in the support of the root view caused by δR.

Figure 13 shows the function UPDATEINDTREE that applies an update δR to an indicator tree T_{Ind} with a root view $I(\mathcal{F})$. The function returns the change $\delta(\exists I)$ in the support of the indicator view I, to be further propagated to other views. The free variables $\mathcal{F}$ of I appear in each input relation from T_{Ind}, and δR fixes their values to constants; thus, $|\delta(\exists I)| \leq 1$.

Figure 14 gives our algorithm for maintaining a set of view trees $\mathcal{T}$ and a set of indicator tress T_{Ind} under an update δR. We first apply δR to the view trees from $\mathcal{T}$ (line 2). Then, we consider the triplets (T_{All}, T_L, T_H) of indicator trees from $\mathcal{T}_{Ind}$ that are affected by δR. We maintain the heavy indicator tree T_H with the root $H(\mathcal{F}) = All(\mathcal{F}), \nexists L(\mathcal{F})$ for changes in both All and $\nexists L$. We apply δR to T_{All} (line 7) and subsequently δAll to T_H (line 9). The latter may trigger a change $\delta(\exists H)$ in the support of H, which we apply to the view trees from $\mathcal{T}$ (line 10). If the update δR belongs to the light part $R^{\mathcal{F}}$, we apply δR to T_L and propagate any change $\delta(\exists L)$ in the support of the root L of T_L to the view trees from $\mathcal{T}$ (lines 11-13); we also propagate the opposite change $\delta(\nexists L)$, if any, to T_H and further to the view trees from $\mathcal{T}$ (lines 14-15).

Proposition 22. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ under a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ time.*

5.2 Rebalancing Partitions

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 amortized over a sequence of updates.

Major Rebalancing. We loosen the partition threshold to amortize the cost of rebalancing over multiple updates. Instead of the actual database size N, the threshold now depends on a number Θ for which the invariant $\lfloor \frac{1}{4}\Theta \rfloor \leq N < \Theta$ always holds. If the database size falls below $\lfloor \frac{1}{4}\Theta \rfloor$ or reach Θ, we perform *major rebalancing*, where we halve or, respectively, double Θ, followed by strictly repartitioning the light parts of input relations with the new threshold Θ^{ϵ} and recomputing the views. Figure 15 gives the procedure for major rebalancing.

Proposition 23. *Given a hierarchical query $Q(\mathcal{F})$ with factorization width w, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, major rebalancing of the views in the set of view trees $\tau(\omega, \mathcal{F})$ takes $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time.*

The cost of major rebalancing is amortized over $\Omega(\Theta)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}\Theta$ (after doubling), or $N = \frac{1}{2}\Theta - \frac{1}{2}$ or $N = \frac{1}{2}\Theta - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}\Theta \rfloor \leq N < \Theta$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}\Theta$.

UpdateTrees(view trees $\mathcal{T}$, indicator triples $\mathcal{T}_{Ind}$, update δR)

```
1  let δR = {x → m}
2  foreach T ∈ 𝒯 do Apply(T, δR)
3  foreach (T_All, T_L, T_H) ∈ 𝒯_Ind such that R ∈ T_All do
4    let All(ℱ) = root(T_All), L(ℱ) = root(T_L), H(ℱ) = root(T_H)
5    let key = x[ℱ]
6    let #before = All(key)
7    Apply(T_All, δR)
8    let #change = All(key) − #before
9    let δ(∃H) = UpdateIndTree(T_H, δAll = {key → #change})
10   foreach T ∈ 𝒯 do Apply(T, δ(∃H))
11   if (key ∉ π_ℱ R ∨ key ∈ π_ℱ R^ℱ)
12     let δ(∃L) = UpdateIndTree(T_L, δR^ℱ = {x → m})
13     foreach T ∈ 𝒯 do Apply(T, δ(∃L))
14     let δ(∃H) = UpdateIndTree(T_H, δ(∄L) = −δ(∃L))
15     foreach T ∈ 𝒯 do Apply(T, δ(∃H))
```

Figure 14: Updating a set of view trees $\mathcal{T}$ and a set of triplets of indicator view trees $\mathcal{T}_{Ind}$ for a single-tuple update δR to relation R. Apply$(T, \delta R)$ updates each view in a view tree T with the delta computed in a bottom-up fashion starting from a leaf R in the view tree T; if T does not refer to R, Apply has no effect.

Minor Rebalancing. After an update δR to relation R, we check the heavy and light conditions of each partition of R (cf. Definition 6). Consider the light part $R^{\mathcal{F}}$ of R partitioned on $\mathcal{F}$. If the number of tuples with the $\mathcal{F}$-values of δR in $R^{\mathcal{F}}$ exceeds $\frac{3}{2}\Theta^{\epsilon}$, then we delete those tuples from $R^{\mathcal{F}}$. If the number of tuples with the $\mathcal{F}$-values of δR in $R^{\mathcal{F}}$ is zero and in R falls below $\frac{1}{2}\Theta^{\epsilon}$, then we insert those tuples into $R^{\mathcal{F}}$. We call this procedure *minor rebalancing* (see Figure 15).

Proposition 24. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, minor rebalancing of the views in the set of view trees $\tau(\omega, \mathcal{F})$ takes $\mathcal{O}(N^{(\delta+1)\epsilon})$ time.*

The cost of minor rebalancing is amortized over $\Omega(\Theta^{\epsilon})$ updates. This lower bound on the number of updates is due to the gap between the two thresholds in the heavy and light part conditions.

Figure 16 gives the trigger procedure OnUpdate that maintains a set of view trees $\mathcal{T}$ and a set of indicator trees $\mathcal{T}_{Ind}$ and, if necessary, performs major and minor rebalancing under a sequence of single-tuple updates to input relations. We first apply an update δR to the view trees from $\mathcal{T}$ and indicator trees from $\mathcal{T}_{Ind}$ using UpdateTrees from Figure 14. If this update leads to a violation of the size invariant $\lfloor \frac{1}{4}\Theta \rfloor \leq N < \Theta$, we invoke MajorRebalancing to recompute the light parts of the input relations and affected views. Otherwise, for each triple of indicator trees from $\mathcal{T}_{Ind}$ with the light part $R^{\mathcal{F}}$ partitioned on $\mathcal{F}$, we check if the heavy or light condition is violated; if so, we invoke MinorRebalancing to move the R-tuples having the $\mathcal{F}$-values of the update δR either into or from the light part $R^{\mathcal{F}}$ of relation R.

As stated in the following proposition, the overall amortized update time is $\mathcal{O}(N^{\delta\epsilon})$.

Proposition 25. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ under a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ amortized time.*

6 Related Work

We overview prior work on static and dynamic query evaluation. Figures 17 and 18 give a survey on hierarchical queries and taxonomies of works on static and dynamic query evaluation.

MajorRebalancing(view trees $\mathcal{T}$, indicator triples $\mathcal{T}_{Ind}$, threshold θ)

1 **foreach** $(T_{All}, T_L, T_H) \in \mathcal{T}_{Ind}$ **do**
2 **foreach** $R^{\mathcal{F}} \in T_L, R \in T_{All}$ **do**
3 $R^{\mathcal{F}} = \{\mathbf{x} \to R(\mathbf{x}) \mid \mathbf{x} \in R,\ key = \mathbf{x}[\mathcal{F}],\ |\sigma_{\mathcal{F}=key} R| < \theta\}$
4 Recompute(T_L), Recompute(T_H)
5 **foreach** $T \in \mathcal{T}$ **do** Recompute(T)

MinorRebalancing(trees $\mathcal{T}$, tree T_L, tree T_H, source R, key, $insert$)

1 **let** $L(\mathcal{F}) = \mathsf{root}(T_L),\ H(\mathcal{F}) = \mathsf{root}(T_H)$
2 **foreach** $\mathbf{x} \in \sigma_{\mathcal{F}=key} R$ **do**
3 **let** $cnt =$ **if** $(insert)$ $R(\mathbf{x})$ **else** $-R(\mathbf{x})$
4 **let** $\delta(\exists L) =$ UpdateIndTree($T_L, \delta R^{\mathcal{F}} = \{\mathbf{x} \to cnt\}$)
5 **foreach** $T \in \mathcal{T}$ **do** Apply($T, \delta(\exists L)$)
6 **let** $\delta(\exists H) =$ UpdateIndTree($T_H, \delta(\nexists L) = -\delta(\exists L)$)
7 **foreach** $T \in \mathcal{T}$ **do** Apply($T, \delta(\exists H)$)

Figure 15: MajorRebalancing recomputes the light parts of base relations and affected views. MinorRebalancing deletes heavy tuples from or inserts light tuples into the light part of relation R.

Hierarchical queries. The notion of hierarchical queries used in this paper (Definition 1) has been initially introduced in the context of probabilistic databases [48].

The Boolean conjunctive queries without repeating relation symbols that can be computed in polynomial time on tuple-independent probabilistic databases are hierarchical; non-hierarchical queries are hard for #P [48]. This was extended to non-Boolean queries with negation [22].

Hierarchical queries are the conjunctive queries whose provenance admits a factorized representation where each input tuple occurs a constant number of times; any factorization of the provenance of a non-hierarchical query would require a number of occurrences of the provenance of some input tuple dependent on the input database size [43].

In the MPC model, the hierarchical queries admit parallel evaluation with one communication step [35]. The r-hierarchical queries, which are conjunctive queries that become hierarchical by repeatedly removing the atoms whose complete set of variables occurs in another atom, can be evaluated in the MPC model using a constant number of steps and optimal load on every single database instance [25].

Hierarchical queries also admit one-step streaming evaluation in the finite cursor model [23]. Under updates, the q-hierarchical queries are the conjunctive queries that admit constant-time update and delay [11]. The q-hierarchical queries are a proper subclass of both the free-connex acyclic and hierarchical queries. In addition to being hierarchical, a second condition holds for a q-hierarchical query: if the set of relation symbols of a free variable is strictly contained in the set of another variable, then the latter must also be free.

Static Evaluation. Prior seminal work exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [18]. It constructs a succinct representation of the query result that allows for enumeration of tuples over some variables under value bindings for all other variables. It does not support enumeration for queries with free variables, as addressed in our work. Example 4 is stated as an open problem in their work.

The result of any acyclic conjunctive query can be enumerated with constant delay after linear-time preprocessing if and only if it is free-connex [9]. This is based on the conjecture that Boolean multiplication of $n \times n$ matrices cannot be done in $O(n^2)$ time. A generalization of this result accommodates functional dependencies [16].

Acyclicity itself is necessary for having constant delay enumeration: A conjunctive query admits constant delay enumeration after linear-time preprocessing if and only if it is free-connex acyclic [15]. This is based

```
OnUpdate(view trees 𝒯, indicator triples 𝒯_Ind, update δR)
1  UpdateTrees(𝒯, 𝒯_Ind, δR)
2  if (N = Θ)
3     Θ = 2Θ
4     MajorRebalancing(𝒯, 𝒯_Ind, Θ^ϵ)
5  else if (N < ⌊¼Θ⌋)
6     Θ = ⌊½Θ⌋ − 1
7     MajorRebalancing(𝒯, 𝒯_Ind, Θ^ϵ)
8  else
9     foreach (T_All, T_L, T_H) ∈ 𝒯_Ind such that R ∈ T_All do
10       let R^ℱ ∈ T_L be light part of R partitioned on ℱ
11       let key = x[ℱ], where δR = {x → m}
12       if ( |σ_{ℱ=key} R^ℱ| = 0 ∧ |σ_{ℱ=key} R| < ½Θ^ϵ)
13          MinorRebalancing(𝒯, T_L, T_H, R, key, true)
14       else if ( |σ_{ℱ=key} R^ℱ| ≥ ¾Θ^ϵ )
15          MinorRebalancing(𝒯, T_L, T_H, R, key, false)
```

Figure 16: Updating a set of view trees $\mathcal{T}$ and a set of triplets of indicator view trees $\mathcal{T}_{Ind}$ under a sequence of single-tuple updates to base relations. The procedures MajorRebalancing and MinorRebalancing are given in Figure 15.

on a stronger hypothesis that the existence of a triangle in a hypergraph of n vertices cannot be tested in time $\mathcal{O}(n^2)$ and that for any k, testing the presence of a k-dimensional tetrahedron cannot be tested in linear time. An in-depth pre-2015 overview on constant-delay enumeration is provided by Segoufin [46].

The literature provides enumeration algorithms also for document spanners [5] and satisfying valuations of circuits [3].

Dynamic evaluation. The q-hierarchical queries are the conjunctive queries that admit linear-time preprocessing and constant-time update and delay [11, 26]. If a conjunctive query without repeating relation symbols is not q-hierarchical, there is no $\gamma > 0$ such that the result of the query can be enumerated with arbitrary preprocessing time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ delay and update time, unless the Online Matrix Vector Multiplication conjecture fails. The upper bound complexities for maintaining q-hierarchical queries are carried over to unions of q-hierarchical queries [13], q-hierarchical queries with small domain constraints, and first-order queries with modulo-counting quantifiers on bounded degree databases [12]. Similar lower bounds conditioned on the Online Matrix Vector Multiplication and the Orthogonal Vector Multiplication conjectures hold for unions of q-hierarchical queries and q-hierarchical queries with small domain constraints.

The work closest in spirit to ours characterizes the dynamic space for counting triangles [28]. Our approach furthers the adaptive maintenance techniques presented in that work. First, since our approach considers a class of queries and not only a single query, it employs a less trivial light/heavy partitioning scheme, where the same relation may be subject to partition on different tuples of variables and where the overall number of cases is reduced by considering the all-light case and at-leat-one-heavy case whenever a partition is made. Second, it uses view trees to capture the query evaluation and maintenance strategy, which is reminiscent of F-IVM [42]. Third, a key challenge in our work is to achieve sublinear delay for the enumeration problem. The triangle counting query from prior work has one result tuple (a scalar) and trivial enumeration.

Results of MSO queries on strings can be enumerated using linear-time preprocessing, constant delay and logarithmic update time. Here, updates can relabel, insert, or remove positions in the string. Further work considers MSO queries on trees under updates [40, 36, 4]. DBToaster [34], F-IVM [42], and DynYannakakis [26, 27] are recent systems for the incremental view maintenance of various classes queries.

Class of Databases	Class of Queries	Preprocessing	Delay	Extra Space	Source
All	f.c. α-acyclic CQ$^{\neq}$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	[9]
All	f.c. β-acyclic negative CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[15, 14]
All	f.c. signed-acyclic CQ	$\mathcal{O}(N\,(\log N)^{\lvert Q\rvert})$	$\mathcal{O}(1)$	–	[15]
All	Acyclic CQ$^{\neq}$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	[9]
All	CQ$^{\neq}$ of f.c. treewidth k	$\mathcal{O}(\lvert\text{Dom}\rvert^{k+1}+N)$	$\mathcal{O}(1)$	–	[9]
All	CQ	$\mathcal{O}(N^{\mathsf{w}(Q)})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{\mathsf{w}(Q)})$	[44, 2]
All	Full CQ with access patterns	$\mathcal{O}(N^{\rho^*(Q)})$	$\mathcal{O}(\tau)$	$\mathcal{O}(N+N^{\rho^*(Q)}/\tau)$	[18]
X-structures (trees, grids)	CQ	$\mathcal{O}(N)$	$\mathcal{O}(N)$	–	[8]
Bounded degree	FO	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[19, 29]
Bounded expansion	FO	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[30]
Local bounded expansion	FO	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	–	[47]
Low degree	FO	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{2+\gamma})$	[20]
Nowhere dense	FO	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\gamma})$	[45]
Bounded treewidth	MSO	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[7, 31]

Figure 17: Prior work on the trade-off between preprocessing time, enumeration delay, and extra space for different classes of queries (Conjunctive Queries, First-Order, Monadic Second-Order) and static databases under data complexity; f.c. stands for free-connex. Parameters: Query Q with factorization width w [44] and fractional edge cover number ρ^* [6]; database of size N; slack τ is a function of N and ρ^*; $\gamma > 0$. Most works do not discuss the extra space utilization (marked by –).

Class of Databases	Class of Queries	Preprocessing	Update	Delay	Extra Space	Source
All	q-hierarchical CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[11, 26]
All	Triangle count	$\mathcal{O}(N^{\frac{3}{2}})$	$\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$	[28]
All	q-hierarchical UCQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[13]
Bounded degree	FO+MOD	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[12]
Strings	MSO	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$	–	[41]

Figure 18: Prior work on the trade-off between preprocessing time, update time, enumeration delay, and extra space for different classes of queries (Conjunctive Queries, Count Queries, First-Order Queries with modulo-counting quantifiers, Monadic Second Order Logic) and databases under updates in data complexity. Parameters: Query Q; database of size N; $\epsilon \in [0, 1]$. Most works do not discuss the extra space utilization (marked by –). †: amortized update time.

References

[1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.

[2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.

[3] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017.

[4] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *ICDT*, pages 5:1–5:18, 2018.

[5] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.

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

[7] G. Bagan. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *CSL*, pages 167–181, 2006.

[8] G. Bagan, A. Durand, E. Filiot, and O. Gauwin. Efficient Enumeration for Conjunctive Queries over X-underbar Structures. In *CSL*, pages 80–94, 2010.

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

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

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

[12] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In *ICDT*, pages 8:1–8:18, 2017.

[13] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In *ICDT*, pages 8:1–8:19, 2018.

[14] J. Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *CSL*, pages 137–151, 2012.

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

[16] N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *ICDT*, 2018.

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

[18] S. Deep and P. Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.

[19] A. Durand and E. Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic*, 8(4):21, 2007.

[20] A. Durand, N. Schweikardt, and L. Segoufin. Enumerating Answers to First-order Queries over Databases of Low Degree. In *PODS*, pages 121–131, 2014.

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

[22] R. Fink and D. Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM Trans. Datab. Syst.*, 41(1):4:1–4:47, 2016.

[23] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. V. den Bussche. Database query processing using finite cursor machines. *Theory Comput. Syst.*, 44(4):533–560, 2009.

[24] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015.

[25] X. Hu and K. Yi. Instance and output optimal parallel algorithms for acyclic joins. In *PODS*, page To appear, 2019.

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

[27] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. Conjunctive queries with inequalities under updates. *PVLDB*, pages 733–745, 2018.

[28] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. In *ICDT*, pages 4:1–4:18, 2019.

[29] W. Kazana and L. Segoufin. First-order Query Evaluation on Structures of Bounded Degree. *LMCS*, 7(2), 2011.

[30] W. Kazana and L. Segoufin. Enumeration of First-order Queries on Classes of Structures with Bounded Expansion. In *PODS*, pages 297–308, 2013.

[31] W. Kazana and L. Segoufin. Enumeration of Monadic Second-order Queries on Trees. *ACM Trans. Comput. Logic*, 14(4):25:1–25:12, 2013.

[32] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, pages 429–444, 2017.

[33] C. Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, pages 87–98, 2010.

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

[35] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[36] K. Losemann and W. Martens. MSO queries on trees: enumerating answers under updates. In *CSL-LICS*, pages 67:1–67:10, 2014.

[37] D. Marx. Approximating fractional hypertree width. *ACM Trans. Alg.*, 6(2):29:1–29:17, 2010.

[38] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *PODS*, pages 111–124, 2018.

[39] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[40] M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, pages 769–778, 2018.

[41] M. Niewerth and L. Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, pages 179–191, 2018.

[42] M. Nikolic and D. Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018.

[43] D. Olteanu and Závodný. Factorised representations of query results: size bounds and readability. In *ICDT*, pages 285–298, 2012.

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

[45] N. Schweikardt, L. Segoufin, and A. Vigny. Enumeration for FO Queries over Nowhere Dense Graphs. In *PODS*, pages 151–163, 2018.

[46] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Rec.*, 44(10–17), 2015.

[47] L. Segoufin and A. Vigny. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *ICDT*, 2017.

[48] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[49] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

A Further Details for Section 1

A.1 Proof of Theorem 2

Theorem 2. *Given a hierarchical query with factorization width* w*, a database of size* N*, and* $\epsilon \in [0,1]$*, we can compute in time* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *a data structure that allows the enumeration of the query result with* $\mathcal{O}(N^{1-\epsilon})$ *delay.*

Proof. The theorem follows from Propositions 19, 20, and 21. Let $Q(\mathcal{F})$ be a hierarchical query and ω an arbitrary canonical variable order for $Q(\mathcal{F})$. Without loss of generality, assume that ω consists of a single tree. Our data structure consists of the materialized views in the view trees $\{T_1, \ldots, T_k\}$ returned by the procedure $\tau(\omega, \mathcal{F})$ in Figure 8 run in mode 'static'. By Proposition 20, these views can be materialized in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. Proposition 19 states that $Q(\mathcal{F})$ is equivalent to $\bigcup_{i\in[k]} Q_i(\mathcal{F})$, where $Q_i(\mathcal{F})$ is the query defined by the join of the leaves in $\mathcal{T}_i$ for $i \in [k]$. We can enumerate the tuples in the result of $Q(\mathcal{F})$ with delay $\mathcal{O}(N^{1-\epsilon})$ by using the materialized view trees in $\tau(\omega, \mathcal{F})$ (Proposition 21).

In case the canonical variable order of $Q(\mathcal{F})$ consists of several trees $\omega_1, \ldots, \omega_m$, we construct a set $\mathcal{T}_i$ of view trees for each ω_i with $i \in [m]$. The result of the query is the Cartesian product of the tuple sets obtained from each $\mathcal{T}_i$. Given that each set $\mathcal{T}_i$ of view trees admits enumeration with $\mathcal{O}(N^{1-\epsilon})$ delay, the Cartesian product can be enumerated with the same delay using the PRODUCT algorithm in Figure 12, since m is independent of the database size N. □

A.2 Proof of Theorem 3

Theorem 3. *Given a hierarchical query with factorization width* w *and delta width* δ*, a database of size* N*, and* $\epsilon \in [0,1]$*, we can compute in time* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *a data structure that allows the enumeration of the query result with* $\mathcal{O}(N^{1-\epsilon})$ *delay and can be maintained under a single-tuple update in* $\mathcal{O}(N^{\delta\epsilon})$ *amortized update time.*

Proof. The theorem follows from Propositions 20, 19, 21, and 22. Let $Q(\mathcal{F})$ be a hierarchical query and ω an arbitrary canonical variable order for $Q(\mathcal{F})$. Without loss of generality, we assume that ω consists of a single tree. To compute the data structure, we first run the procedure $\tau(\omega, \mathcal{F})$ from Figure 8 in mode 'dynamic'. The procedure returns a set $\mathcal{T} = \{T_1, \ldots, T_k\}$ of view trees. Our data structure consists of the materialized views of these view trees. Since the proof of Proposition 20 considers both *mode* = 'static' and *mode* = 'dynamic', the time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ to materialize the views in the view trees in $\mathcal{T}$ follows from that proof. Likewise, the equivalence between the materialized view trees and the query follows from Proposition 19. The delay $\mathcal{O}(N^{1-\epsilon})$ needed when enumerating the query result from the data structure is shown in the proof of Proposition 21. The amortized time $\mathcal{O}(N^{\delta\epsilon})$ to maintain the data structure under single-tuple updates follows from Proposition 25.

If the canonical variable order consists of several trees $\omega_1, \ldots, \omega_m$, we construct a data structure for each of the trees ω_i with $i \in [m]$. The result of $Q(\mathcal{F})$ is the Cartesian product of the tuple sets obtained from the data structures. The tuples in this Cartesian product can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay. □

B Further Preliminaries

We first give the proof of Lemma 11.

Lemma 11. *For any full hierarchical query Q, it holds $\rho^*(Q) = \rho(Q)$.*

Proof. We define an integral edge cover $\boldsymbol{\lambda} = (\lambda_R)_{R \in atoms(Q)}$ for Q and show that $\sum_{R \in atoms(R)} \lambda_R \leq \rho^*(Q)$. Let ω be an arbitrary canonical variable order for Q. Recall that each root-to-leaf path in ω corresponds to an atom R in Q, such that the leaf is R and the set of inner nodes is the schema of R. A maximal variable path p in ω is a path that starts at a root and ends at a variable X such that all children of X are atoms. We call X the end variable of p. We assume that ω has k maximal variable paths. For each maximal variable path p in ω with end variable X, we fix an arbitrary child atom R_p of X. For each atom R in Q, we define:

$$\lambda_R = \begin{cases} 1 & \text{, if } R = R_p \text{ for some maximal variable path } p \\ 0 & \text{, otherwise} \end{cases}$$

It follows from the definition of $\boldsymbol{\lambda}$ that $\sum_{R \in atoms(R)} \lambda_R = k$.

Let $\boldsymbol{\lambda'} = (\lambda_R)_{R \in atoms(Q)}$ be an arbitrary fractional edge cover for Q. Given any maximal variable path p where the end variable has the child atoms $R_1, \ldots, R_m$, we define $s_p = \sum_{i \in [m]} \lambda'_{R_i}$. We complete the proof by showing the following two statements:

1. $\boldsymbol{\lambda} = (\lambda_R)_{R \in atoms(Q)}$ is an integral edge cover for Q.
2. $k \leq \sum_{R \in atoms(R)} \lambda'_R$.

$\boldsymbol{\lambda} = (\lambda_R)_{R \in atoms(Q)}$ is an integral edge cover for Q: Let X be an arbitrary variable in Q. The variable X must be included in at least one maximal variable path p. Since $\lambda_{R_p} = 1$, the variable X is covered by the edge cover $\boldsymbol{\lambda}$. Since X was chosen arbitrary, it follows that $\boldsymbol{\lambda}$ is an integral edge cover for Q.

$k \leq \sum_{R \in atoms(Q)} \lambda'_R$: Let p be an arbitrary maximal variable path in ω with end variable X. Let $R_1, \ldots, R_m$ be the child atoms of X. Besides $R_1, \ldots, R_m$, no other atom has X in its schema. Hence, it must hold $s_p \geq 1$. Since there are k maximal variable paths, this implies $k \leq k \cdot s_p \leq \sum_{R \in atoms(Q)} \lambda'_R$. □

Next, we prove the following Proposition 15 from Section 2.

Proposition 15. *Given a hierarchical query Q with factorization width w and delta width δ, it holds $\mathsf{w} - 1 \leq \delta$.*

We first introduce restrictions of variable orders.

Restrictions of Variable Orders. Given a variable order ω and a set $\mathcal{X} \subseteq vars(\omega)$, we obtain the restriction $\omega|_{\mathcal{X}}$ of ω onto $\mathcal{X}$ by executing for each $X \in vars(\omega) - \mathcal{X}$ the following procedure: We eliminate X. If X has no children, we are done. Otherwise, let $\omega_1, \ldots, \omega_k$ be the subtrees of X. If X has a parent Y, then $\omega_1, \ldots, \omega_k$ become the subtrees of Y. If X has no parent, then $\omega_1, \ldots, \omega_k$ become independent trees.

The following lemma is key in the proof of Proposition 15

Lemma 26. *Let Q be a hierarchical query with factorization width $w \geq 2$. Any canonical variable order ω for Q must have a variable $X \in bf(\omega)$ such that $\rho^*(Q_X|_{vars(Q_X) - njb(Q_X)}) \geq w$.*

Proof. For the sake of contradiction assume that there is a canonical variable order $\omega^c = (T^c, dep^c)$ for Q such that for any $X \in bf(\omega^c)$, it holds $\rho^*(Q_X|_{vars(Q_X) - njb(Q_X)}) \leq w' < w$. We show that we can transform ω^c into a free-top variable order $\omega^{new} = (T^{new}, dep^{new})$ for Q such that $\rho^*(Q|_{\{X\} \cup dep^{new}(X)}) < w$ for all variables X. This contradicts the assumption that w is the factorization width of Q.

Let $X_1, \ldots, X_k \in bf(\omega^c)$ be all bound variables that are ancestors of free variables and have no bound variables as ancestors in ω^c. We obtain ω^{new} from ω^c by restructuring the subtrees in ω^c rooted at the variables $X_1, \ldots, X_k$ such that within each subtree the free variables are above the bound variables. Consider a subtree $\omega^c_{X_i}$ rooted at X_i for some $i \in [k]$. Let $F^i_1, \ldots, F^i_{n_i}$ be the free variables in $\omega^c_{X_i}$ in some fixed order. The new subtree $\omega^{new}_{F^i_1}$ that replaces the subtree $\omega^c_{X_i}$ starts with the path $F^i_1, \ldots, F^i_{n_i}$ followed by the restriction of $\omega^c_{X_i}$ that drops the variables $F^i_1, \ldots, F^i_{n_i}$. That is, the root of $\omega^{new}_{F^i_1}$ is F^i_1, the child of F^i_j is F^i_{j+1}, for $j \in [n_i - 1]$, and the child of $F^i_{n_i}$ is the root of the tree $\omega^c_{X_i}|_{vars(\omega^c_{X_i}) - \{F^i_1, \ldots, F^i_{n_i}\}}$. Let $\omega^{new} = (T^{new}, dep^{new})$ be the variable order that results from ω^c by replacing each subtree $\omega^c_{X_i}$ by $\omega^{new}_{F^i_1}$ for $i \in [k]$. It remains to show that the following three statements hold.

1. ω^{new} is free-top.
2. For any atom R in Q, the variables in $\mathcal{S}(R)$ are on the same root-to-leaf path in ω^{new}.
3. For each variable X in ω^{new}: $\rho^*(Q|_{\{X\} \cup dep^{new}(X)}) < w$.

Proof of (1): *ω^{new} is free-top.*
The variables $X_1, \ldots, X_k$ are the highest bound variables in ω^c that contain free variables in their subtrees. Hence, for each $i \in [k]$, all ancestors of X_i are free. When constructing ω^{new} from ω^c, the subtree $\omega^c_{X_i}$ is replaced by the free-top subtree $\omega^{new}_{F^i_1}$. Thus, all free variables in ω^{new} are at the top.

Proof of (2): *For any atom R in Q, the variables in $\mathcal{S}(R)$ are on the same root-to-leaf path in ω^{new}.*
We first observe two simple properties:

$(*)$ $\omega^c|_{vars(Q) - \bigcup_{i \in [k]} vars(\omega^c_{X_i})} = \omega^{new}|_{vars(Q) - \bigcup_{i \in [k]} vars(\omega^c_{X_i})}$

$(**)$ $vars(\omega^c_{X_i}) = vars(\omega^{new}_{F^i_1})$, for any $i \in [k]$.

Since ω^c is a valid variable order, the variables of each atom in Q must be on the same root-to-leaf path in ω^c. Using the two properties above, we show that any two variables X and Y that are on the same root-to-leaf path in ω^c, are on the same root-to-leaf path in ω^{new}. We make a case distinction on the positions of X and Y in ω^c.

- X and Y do not occur in any $\omega^c_{X_i}$, with $i \in [k]$. By property $(*)$, X and Y must be on the same root-to-leaf-path in ω^{new}.
- X does not occur in any $\omega^c_{X_i}$, with $i \in [k]$, and Y occurs in some $\omega^c_{X_j}$, with $j \in [k]$. This means that Y is an ancestor of X. By properties $(*)$ and $(**)$, Y must be in $\omega^{new}_{F^j_1}$ and X is an ancestor of Y in ω^{new}.
- X and Y occur in some $\omega^c_{X_i}$ with $i \in [k]$. By property $(**)$, X and Y are in $\omega^{new}_{F^i_1}$. If both X and Y are bound variables, then X and Y remain on the same root-to-leaf path by the construction of $\omega^{new}_{F^i_1}$. If X is free and Y is bound, then X must be above Y in $\omega^{new}_{F^i_1}$ by construction; thus, X and Y are on the same root-to-leaf path. If both X and Y are free, they belong to the same path $F^i_1, \ldots, F^i_{n_i}$ in ω^{new}.

Proof of (3): *For each X in ω^{new}, $\rho^*(Q|_{\{X\} \cup dep^{new}(X)}) < w$.*
We distinguish the following cases:

- X is not included in any $\omega^c_{X_i}$, for $i \in [k]$. When constructing ω^{new} from ω^c the set of variables in the root path of X as well as the set of variables in the subtree rooted at X do not change. This means that $dep^c(X) = dep^{new}(X)$. Since ω^c is canonical, all variables in $\{X\} \cup dep^c(X)$ are included in the schema of a single atom. Hence, $\rho^*(Q_{\{X\} \cup dep^c(X)}) = \rho^*(Q_{\{X\} \cup dep^{new}(X)}) \leq 1$.

- X is included in some $\omega^c_{X_i}$, for $i \in [k]$. In this case, X must be included in $\omega^{new}_{F^i_1}$.

  We recall that X_i is the root of $\omega^c_{X_i}$ in ω^c. The query Q_{X_i} consists of all atoms that occur at the leaves of ω^c. In the restricted query $Q_{X_i}|_{vars(Q_{X_i})-njb(Q_{X_i})}$, all non-join bound variables are dropped. Let $\boldsymbol{\lambda} = (\lambda_R)_{R \in atoms(Q_{X_i})}$ be a fractional edge cover for $Q_{X_i}|_{vars(Q_{X_i})-njb(Q_{X_i})}$. In the following we show that $\boldsymbol{\lambda}$ is a fractional edge cover for $Q|_{\{X\} \cup dep^{new}(X)}$ or $\rho^*(Q|_{\{X\} \cup dep^{new}(X)}) = 1$. From the assumption that $\rho^*(Q_{X_i}|_{vars(Q_{X_i})-njb(Q_{X_i})}) \leq w' < w$, and $w \geq 2$, it follows that $\rho^*(Q|_{\{X\} \cup dep^{new}(X)}) < w$.

  Recall that $\{X\} \cup dep^{\text{new}}(X) = \{X\} \cup dep^c(X) \cup \mathcal{F}$, where $\mathcal{F}$ is the set of free variables in ω^c_X that moved to the root path of X. Since $vars(Q_{X_i}) - njb(Q_{X_i})$ includes all variables in $(\{X\} \cup \mathsf{anc}^c(X) \cup vars(\omega^c_X)) - njb(Q_{X_i})$, $\boldsymbol{\lambda}$ is a fractional edge cover for $Q|_{(\{X\} \cup dep^{new}(X)) - njb(Q_{X_i})}$. We show that one of the following cases holds:

  - $\boldsymbol{\lambda}$ is a fractional edge cover for $Q|_{(\{X\} \cup dep^{new}(X)) \cap njb(Q_{X_i})}$. Since we already showed that $\boldsymbol{\lambda}$ is a fractional edge cover for $Q|_{(\{X\} \cup dep^{new}(X)) - njb(Q_{X_i})}$, it follows that $\boldsymbol{\lambda}$ is a fractional edge cover for $Q|_{\{X\} \cup dep^{new}(X)}$.
  - The fractional edge cover number of $Q|_{\{X\} \cup dep^{new}(X)}$ is 1.

  Let $Y \in (\{X\} \cup dep^{new}(X)) \cap njb(Q_{X_i})$. This means that Y is an ancestor of X in ω^{new}. It follows from the construction of ω^{new} that Y must be an ancestor of X in ω^c. We distinguish whether Y is an ancestor of free variables in ω^c or not.

  First assume that Y is an ancestor of free variables $\mathcal{F}'$ in ω^c. Since Y is a non-join variable, all free variables in $\mathcal{F}'$ must be non-join variables and $\{Y\} \cup \mathcal{F}'$ must be contained in the schema of a single atom R. In order to cover the free variables in $\mathcal{F}'$, the fractional edge cover $\boldsymbol{\lambda}$ must assign 1 to atom R. This means that $\boldsymbol{\lambda}$ covers Y.

  Now assume that Y is not an ancestor of free variables in ω^c. This means that X is not an ancestor of free variables in ω^c either. Since ω^c is canonical, $\rho^*(Q|_{\{X\} \cup dep^c(X)}) = 1$. Since X is not an ancestor of free variables in ω^c, we have $\{X\} \cup dep^c(X) = \{X\} \cup dep^{\text{new}}(X)$. it follows that the fractional edge cover number of $Q_{\{X\} \cup dep^{new}(X)}$ is 1.

The above analysis implies that ω^{new} is a free-top variable order for Q with factorization width $\max\{1, w'\}$ where $w' < w$. Since $w \geq 2$, the factorization width of ω^{new} must be less than w. This means however that the factorization width of Q must be less than w, which contradicts our initial assumption. □

We are now ready to prove Proposition 15.

Proof of Proposition 15. We first consider the case that the factorization width w of Q is 1. In this case, it holds $w - 1 = 0$. By definition, the delta width δ of Q must be at least 0. Hence, the statement in the lemma holds in this case.

We now consider the case that $w \geq 2$. For the sake of contradiction assume that $w - 1 > \delta$. Let ω^f be a free-top variable order for Q with factorization width $w(\omega^f) = w$ and let ω^c be a canonical variable order for Q with delta width $\delta(\omega^c) = \delta$. From our assumption $w - 1 > \delta$ follows that for any $X \in vars(Q)$ and any atom R in Q_X, we have

$$\rho^*(Q_X|_{vars(\omega^c)-njb(Q_X)-\mathcal{S}(R)}) < w - 1. \tag{1}$$

From Lemma 26, it follows that ω^c must have a node $X \in bf(\omega^c)$ such that

$$\rho^*(Q_X|_{vars(\omega^c)-njb(Q_X)}) \geq w. \tag{2}$$

We show that Inequalities (1) and (2) are contradicting, which completes the proof. Let $X \in vars(Q)$ and R any atom in Q_X. Let $\boldsymbol{\lambda} = (\lambda_K)_{K \in atoms(Q_X)}$ be a fractional edge cover for the query $Q_X|_{vars(\omega^c)-njb(Q_X)-\mathcal{S}(R)}$ such that

$$\sum_{K \in atoms(Q_X)} \lambda_K = \rho^*(Q_X|_{vars(\omega^c)-njb(Q_X)-\mathcal{S}(R)}) < w - 1.$$

Let $\boldsymbol{\lambda'} = (\lambda'_K)_{K \in atoms(Q_X)}$ be defined as

$$\lambda'_K = \begin{cases} 1 & \text{, if } K = R \\ \lambda_K & \text{, otherwise} \end{cases}$$

Clearly, $\boldsymbol{\lambda'}$ is a fractional edge cover for $Q_X|_{vars(\omega^c) - njb(Q_X)}$. Moreover, due to Inequality (1), it holds $\sum_{K \in atoms(Q_X)} \lambda'_K < w$. However, this contradicts Inequality (2). □

C Further Details for Section 3

In Section C.1 we prove Proposition 19 and in Section C.2 we give the proof of Proposition 20.

C.1 Proof of Proposition 19

Proposition 19. *Let $\{T_1, \ldots, T_k\} = \tau(\omega, \mathcal{F})$ be the set of view trees constructed by the algorithm in Figure 8 for a given hierarchical query $Q(\mathcal{F})$ and a canonical variable order ω for Q. Let $Q_i(\mathcal{F})$ be the query defined by the conjunction of the leaf atoms in $\mathcal{T}_i$, $\forall i \in [k]$. Then, $Q(\mathcal{F}) \equiv \bigcup_{i \in [k]} Q_i(\mathcal{F})$.*

Given a variable order or view tree T, we use $atoms(T)$ to denote the set of atoms at the leaves of T. We start with some observations. The procedure FACTVT in Figure 4 constructs a view tree whose leaf atoms are exactly the same as the leaf atoms of the input variable order. This is a direct implication of the behaviour of the procedure in the base case (line 1).

Remark 27. *Given a symbol V, a variable order ω and a set $\mathcal{F}$ of variables, the procedure* FACTVT$(V, \omega, \mathcal{F})$ *in Figure 4 returns a view tree T such that*

$$atoms(\omega) = atoms(T).$$

Likewise, the procedure NEWVT in Figure 5 outputs a view tree that is composed of the input trees. Hence, the set of leaf atoms of the output tree is the union of the sets of leaf atoms of the input trees.

Remark 28. *Given a variable X, a symbol V, a variable order ω, a set $\mathcal{F}$ of variables, and a set $\mathcal{T}$ of view trees, the procedure* NEWVT$(X, V, \mathcal{F}, \mathcal{T})$ *in Figure 5 returns a view tree T such that*

$$atoms(T) = \bigcup_{T' \in \mathcal{T}} atoms(T').$$

Proof of Proposition 19. For a variable order or view tree T and a set $\mathcal{F}'$ of variables, we write $Q_T(\mathcal{F}')$ to denote the query that has free variable set $\mathcal{F}'$ and joins the atoms in $atoms(T)$. The proof of Proposition 19 is by simple induction over the structure of ω. We show that for any subtree ω' of ω, it holds

$$Q_{\omega'}(\mathcal{F} \cap vars(\omega')) \equiv \bigcup_{T \in \tau(\omega', \mathcal{F})} Q_T(\mathcal{F} \cap vars(\omega')).$$

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

Inductive step: We assume that ω' has root variable X and subtrees $\omega_1, \ldots, \omega_k$. Let $keys = \mathsf{anc}(X) \cup \{X\}$, $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega_X))$, and $Q_X(\mathcal{F}_X) =$ join of $atoms(\omega)$. Following the control flow in the procedure τ, we distinguish the following cases:

Case 1: ($mode =$ 'static' $\wedge\, Q_X(\mathcal{F}_X)$ is free-connex) $\vee$ ($mode =$ 'dynamic' $\wedge\, Q_X(\mathcal{F}_X)$ is q-hierarchical). In this case, the algorithm returns a view tree T constructed by FACTVT$(V, \omega', \mathcal{F}_X)$. By Remark 27, T has at its leaves exactly the same set of atoms as ω'. Thus, the induction step holds in this case.

Case 1 does not hold and $X \in \mathcal{F}$: The set of view trees returned in this case contains for each set $\{T_i\}_{i\in[k]}$ of view trees with $T_1 \in \tau(\omega_1, \mathcal{F})$, ..., $T_k \in \tau(\omega_k, \mathcal{F})$, a view tree constructed by the procedure $\textsc{NewVT}(X, V_X, keys, \{T_i\}_{i\in[k]})$. Using Remark 28 and the induction hypothesis, we rewrite the query $Q_{\omega'}(\mathcal{F} \cap vars(\omega'))$ as follows:

$$
\begin{aligned}
& Q_{\omega'}(\mathcal{F} \cap vars(\omega')) = \text{ join of } atoms(\omega') \\
&= \text{ join of } atoms(\omega'_1), \ldots, atoms(\omega'_k) \\
&= Q_{\omega'_1}(\mathcal{F} \cap vars(\omega'_1)), \ldots, Q_{\omega'_k}(\mathcal{F} \cap vars(\omega'_k)) \\
&\overset{\text{IH}}{\equiv} \bigcup_{T\in\tau(\omega'_1,\mathcal{F})} Q_T(\mathcal{F} \cap vars(\omega'_1)) \ , \ldots, \ \bigcup_{T\in\tau(\omega'_k,\mathcal{F})} Q_T(\mathcal{F} \cap vars(\omega'_k)) \\
&= \bigcup_{\substack{T_1\in\tau(\omega'_1,\mathcal{F}),\ldots,\\ T_k\in\tau(\omega'_k,\mathcal{F})}} Q_{T_1}(\mathcal{F} \cap vars(\omega'_1)) \ , \ldots, Q_{T_k}(\mathcal{F} \cap vars(\omega'_k)) \\
&= \bigcup_{\substack{T_1\in\tau(\omega'_1,\mathcal{F}),\ldots,\\ T_k\in\tau(\omega'_k,\mathcal{F})}} \textsc{NewVT}(X, V_X, keys, \{T_i\}_{i\in[k]}) \\
&= \bigcup_{T\in\tau(\omega',\mathcal{F})} Q_T(\mathcal{F} \cap vars(\omega'))
\end{aligned}
$$

This completes the induction step for this case.

Case 1 does not hold and $X \notin \mathcal{F}$: Consider the case that X is a join variable. Let $\mathcal{Y} = X \cup \mathsf{anc}(X)$. Let $R_1(\mathcal{X}_1), \ldots, R_k(\mathcal{X}_k)$, for $k \leq n$, be the query atoms whose schemas include $\mathcal{Y}$.

The algorithm creates the view $All_X(\mathcal{Y}) = \bigwedge_{i\in[k]} R_i(\mathcal{X}_i)$, the light indicator $L_X(\mathcal{Y}) = \bigwedge_{i\in[k]} R_i^{\mathcal{Y}}(\mathcal{X}_i)$, and the heavy indicator $H_X(\mathcal{Y}) = All_X(\mathcal{Y}) \wedge \nexists L_X(\mathcal{Y})$. Here, $R_i^{\mathcal{Y}}$ is the light part of R_i, $\exists$ in front of a relation turns it into an indicator, i.e., the multiplicities of the tuples are either 0 or 1 (so any non-zero number becomes 1 under $\exists$), while $\nexists$ flips the 0/1 (Boolean) multiplicity. It follows $\exists L_X(\mathcal{Y}) \vee \exists H_X(\mathcal{Y}) = \exists All_X(\mathcal{Y})$.

The algorithm creates two extended view trees in this case: one that joins the original view tree joins with $\exists L_X(\mathcal{Y})$ and the other that joins again the original view tree with $\exists H_X(\mathcal{Y})$. We show that this transformation preserves equivalence in a strong sense, not only under set semantics but also under bag semantics so the multiplicity of the original query result.

As shown in the previous cases, the original view tree is an equivalent representation of the input query $Q(\mathcal{F}) = R_1(\mathcal{X}_1), \ldots, R_n(\mathcal{X}_n)$. The extended view trees thus correspond to the following queries:

$$
\begin{aligned}
Q_L(\mathcal{F}) &= \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) \wedge \exists L_X(\mathcal{Y}) \\
Q_H(\mathcal{F}) &= \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) \wedge \exists H_X(\mathcal{Y})
\end{aligned}
$$

The set of the two extended view trees encode the disjunction $Q'(\mathcal{F}) = Q_L(\mathcal{F}) \vee Q_H(\mathcal{F})$, with the query body

$$
Q'(\mathcal{F}) = \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) \wedge (\exists L_X(\mathcal{Y}) \vee \exists H_X(\mathcal{Y}))
$$

Since $\exists L_X(\mathcal{Y}) \vee \exists H_X(\mathcal{Y}) = \exists All_X(\mathcal{Y})$, we have

$$
Q'(\mathcal{F}) = \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) \wedge \exists All_X(\mathcal{Y})
$$

Since $All_X(\mathcal{Y}) = \bigwedge_{i\in[k]} R_i(\mathcal{X}_i)$, we have

$$\begin{aligned} Q'(\mathcal{F}) &= \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) \wedge \exists \bigwedge_{i\in[k]} \exists R_i(\mathcal{X}_i) \\ &= \bigwedge_{i\in[n]} R_i(\mathcal{X}_i) = Q(\mathcal{F}). \end{aligned}$$

Subsequent partition-based rewritings may apply to the queries $Q_L(\mathcal{F})$ and $Q_H(\mathcal{F})$ by adding new indicators to their view trees. At each such rewriting step, one can show that the equivalence is preserved. □

C.2 Proof of Proposition 20

Proposition 20. *Given a hierarchical query $Q(\mathcal{F})$ with factorization width w, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, the views in the set of view trees $\tau(\omega, \mathcal{F})$ can be materialized in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$.*

We analyze procedure τ from Figure 8 for both of the cases mode = 'static' and mode = 'dynamic'. Hence, the proof covers the corresponding parts in the dynamic case as well.

Without loss of generality we assume that Q has at least one atom $R(\mathcal{X})$ with nonempty $\mathcal{X}$. Otherwise, the factorization width w of the query is 0 and the data structure consists of a constant number of empty views, which obviously can be computed in constant time.

First, we explain the intuition behind the complexity analysis. We distinguish between two cases. If the algorithm is in static mode and Q is free-connex or it is in dynamic mode and Q q-hierarchical, the algorithm constructs a factorized view tree whose views can be materialized in linear time. Otherwise, there must be at least one bound variable X in ω such that the subtree ω' rooted at X has at least two free variables that are not covered by an atom. In this case, the algorithm partitions relations and creates view trees at X where a light or a heavy indicator hangs off the root. The dominating complexity is for the light case. A view tree with the light indicator has a view at its root that joins the atoms at the leaves of ω' using the light indicator. The schema of the view consists of all free variables in ω'. This corresponds to propagating the free variables in ω' upwards the tree. In worst case, the root view of ω is bound and we need to materialize a view whose schema is given by the entire set of free variables. We can compute such a view V as follows. We first aggregate away all bound variables that are not ancestors of free variables. By using the algorithm InsideOut [2], this can be done in time $\mathcal{O}(N)$. Then, we choose one atom to iterate over the tuples in its relation. For each such tuple, we first check in the light indicator whether it is included in the result. If yes, we iterate over the matching tuples in the relations of the other atoms. To decide which atom to take for the outer loop and which ones for the inner loops of our evaluation strategy, we use an optimal integral edge cover $\boldsymbol{\lambda}$ for $Q|_{\mathcal{F}}$, i.e., for the restriction of Q where all variables besides $\mathcal{F}$ are skipped. The schema of each atom that is mapped to 0 by λ must be subsumed by the schema of an atom mapped to one. Hence, we can take one of the atoms mapped to 1 to do the outer loop. The other atoms that are mapped to 1 are used for the inner loops. For the atoms that are mapped to 0, it suffices to do constant-time lookups while iterating over the others. By exploiting the degree constraints imposed by the light indicator, the time to materialize the view V is $\mathcal{O}(N^{1+(\rho(Q|_{\mathcal{F}})-1)\epsilon})$. By Lemma 11, $\rho(Q|_{\mathcal{F}}) = \rho^*(Q|_{\mathcal{F}})$. Taking into account the time we need for aggregating away the bound variables, we get $\mathcal{O}(N^{\max\{1,1+(\rho^*(Q|_{\mathcal{F}})-1)\epsilon\}})$ overall time complexity. In general, we do not need to materialize the join of all base relations, but can keep the representation as factorized as possible. We show that the overall complexity of this strategy is captured by the factorization width of Q.

The proof is structured following the basic building blocks of the main algorithm τ. Lemmas 30, 31, and 32 give upper bounds on the times to materialize the views in the view trees returned by the procedures NewVT (Figure 5), FactVT (Figure 4), and IndicatorVTs (Figure 7). Lemma 33 states the complexity of the main procedure τ. Finally, Lemma 34 builds the bridge of the complexity analysis to the factorization width of Q. Proposition 20 is a direct implication of Lemmas 33 and 34.

Let Q be a query with free variable set $\mathcal{F}$, ω a variable order of Q, and X a node, hence, a variable or relation symbol in ω. We recall that ω_X is the subtree in ω rooted at X and $bound(Q) = vars(Q) - \mathcal{F}$ is the set of bound variables in Q. We further introduce:

$$\xi(X, \mathcal{F}) = \max_{Y \in bound(Q) \cap vars(\omega_X)} \{\rho^*(Q|_{vars(\omega_Y) \cap \mathcal{F}})\}$$

is the maximal fractional edge cover we get when we restrict Q to the free variables that occur under a bound variable in ω_X. If there is no bound variable in the subtree rooted at X or there are no free variables under a bound variable, then $\xi(X, \mathcal{F}) = 0$.

We start with an observation that each view constructed at some node X of the variable order ω contains in its schema all variables in the root path of X and no variables which are not in ω_X. This can be shown by a straightforward induction over the structure of ω.

Remark 29. *Given a canonical variable order ω, a set $\mathcal{F}$ of variables, and a node X in ω, let T be a view tree that is returned by $\tau(\omega_X, \mathcal{F})$ and has root view $V(\mathcal{F}')$. It holds $\mathsf{anc}(X) \subseteq \mathcal{F}' \subseteq \mathsf{anc}(X) \cup \{X\} \cup vars(\omega)$.*

The next lemma states that the time to materialize the views in a view tree returned by the procedure NewVT in Figure 5 is linear in the time to materialize the views in the input trees.

Lemma 30. *Let X be a node in a canonical variable order, V a symbol, $\mathcal{F}' = \mathsf{anc}(X) \cup \{X\}$, and N the database size. Let $T_1 \ldots, T_k$ be view trees with root views $V(\mathcal{F}_1) \ldots, V(\mathcal{F}_k)$, respectively, and let $f_i : \mathbb{N} \to \mathbb{N}$ be a function such that T_i can be materialized in time $f_i(N)$ for $i \in [k]$. Let T be a view tree returned by the procedure* NewVT$(X, V, \mathcal{F}', \{T_i\}_{i \in [k]})$ *given in Figure 5. If the query $V(\mathcal{F}') = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$ is acyclic, the views in T can be materialized in time $\mathcal{O}(\max_{i \in [k]}\{f_i(N)\})$.*

Proof. The procedure first defines the view $V_X(\mathcal{F}') = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$ (line 3). We analyze the time to materialize this view. We first materialize the views in the trees $T_1, \ldots, T_k$. This takes time $\mathcal{O}(\max_{i \in [k]}\{f_i(N)\})$ time. The overall sizes of these views must be upper-bounded by the same amount. We can materialize the view $V_X(\mathcal{F}')$ by using the InsideOut algorithm [2]. Since the factorization width of the query defining the view is 1, the algorithm runs in time linear in the sizes of the materialized views $V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_1)$. Hence, the size and the time to materialize $V_X(\mathcal{F}')$ is bounded by $\mathcal{O}(\max_{i \in [k]}\{f_i(N)\})$. Let *tree* be the view tree that is defined as follows. If $k = 1$ and $\mathcal{F}' = \mathcal{F}_1$, then $tree = T_1$. Otherwise, tree is the view tree that has root $V_X(\mathcal{F}')$ and subtrees $T_1, \ldots, T_k$ (line 4). If the condition in line 5 does not hold, the procedure NewVT returns *tree* (line 7). Otherwise, the procedure returns a view tree that results from *tree* by adding a view $V'_X(\mathsf{anc}(X))$ on top of $V_X(\mathcal{F}')$ that aggregates away X (line 6). The materialization time for $V'_X(\mathsf{anc}(X))$ is not more than linear in the size of the materialized view $V_X(\mathcal{F}')$. Hence, in any case, the time to materialize the views in the tree returned by NewVT is $\mathcal{O}(\max_{i \in [k]}\{f_i(N)\})$. □

The following lemma says that if the input to the procedure FactVT in Figure 4 represents a free-connex query, the procedure outputs a view tree whose views can be materialized in time linear in the database size.

Lemma 31. *Let ω be a canonical variable order, X a node in ω, N the database size, and $\mathcal{F}' \subseteq \mathsf{anc}(X) \cup \{X\} \cup vars(\omega_X)$ with $\mathsf{anc}(X) \cup \{X\} \subseteq \mathcal{F}'$. Let T be the view tree returned by the procedure* FactVT$(V, \omega_X, \mathcal{F}')$ *given in Figure 4. If the query $Q_X(\mathcal{F}') =$ join of $atoms(\omega_X)$ is free-connex, the views in T can be materialized in time $\mathcal{O}(N)$.*

Proof. The proof is by induction over the structure of the variable order ω_X.

Base case: Assume that X is a single atom $R(\mathcal{Y})$. In this case, the procedure returns this atom, which can obviously be materialized in time $\mathcal{O}(N)$.

Inductive step: Assume that X is a variable with children $X_1, \ldots, X_k$ and $Q_X(\mathcal{F}') =$ join of $atoms(\omega_X)$ is a free-connex query. For each $i \in [k]$, let $\mathcal{F}_i = \mathsf{anc}(X_i) \cup \{X_i\} \cup (\mathcal{F}' \cap vars(\omega_X))$.

We first show for each $i \in [k]$:

$Q_{X_i}(\mathcal{F}_i) =$ **join of** $atoms(\omega_{X_i})$ **is free-connex:** An acyclic query is free-connex if and only if after adding an atom $R(\mathcal{X})$, where $\mathcal{X}$ is the set of free variables, the query remains acyclic [15]. A query is acyclic if it has a (not necessarily free-top) variable order ω' with $\mathsf{w}(\omega') = 1$ [44, 10]. Let Q'_X be the query that results from Q_X by adding a new atom $R(\mathcal{F}')$. Likewise, for any $i \in [k]$, let Q'_{X_i} be the query that we obtain from Q_{X_i} by adding a new atom $R(X_i)$. Let ω'_X be a variable order for Q'_X with $\mathsf{w}(\omega'_X) = 1$. This variable order can easily be turned into a variable order ω'_{X_i} for Q'_{X_i} with $\mathsf{w}(\omega'_{X_i}) = 1$. To achieve this, we traverse ω'_X bottom-up and eliminate all variables that are not in $\mathsf{anc}(X_i) \cup \{X_i\} \cup vars(\omega_i)$. When eliminating a variable Y with a parent node Z, we append the children of Y to Z. If an eliminated node has no parent node, its sub trees become independent. Note that after this elimination procedure, any two dependent variables in Q'_{X_i} are still on the same root-to-leaf path in ω'_{X_i}. Due to the definition of variable orders, it must hold that for any set of variables in ω'_{X_i} that is covered by a relation in Q'_X, there must be a covering relation in Q'_{X_i}. Thus, $\mathsf{w}(\omega'_{X_i}) = 1$, which means that Q'_{X_i} is acyclic, and, hence, Q_{X_i} is free-connex.

By induction hypothesis, it holds that the views in each $T_i \in \textsc{FactVT}(V, \omega_{X_i}, \mathcal{F}_i)$ with $i \in [k]$ can be materialized in time $\mathcal{O}(N)$. We now show that the root view V_X of the tree T returned by FactVT can be materialized in time $\mathcal{O}(N)$. We distinguish whether X is included in $\mathcal{F}'$ or not:

Case: $X \in \mathcal{F}'$**:** The tree T returned by FactVT is the output of $\textsc{NewVT}(X, V_X, \mathcal{F}_X, \{T_i\}_{i\in[k]})$, where NewVT is the procedure given in Figure 5 and $\mathcal{F}_X = \mathsf{anc}(X) \cup \{X\}$. Let $V_1(\mathcal{F}'_1), \ldots, V_k(\mathcal{F}'_k)$ be the roots of the trees $T_1, \ldots, T_k$, respectively. It follows from Remark 29 that the query $V_X(\mathcal{F}) = V_1(\mathcal{F}'_1), \ldots, V_k(\mathcal{F}'_k)$ cannot have relations covering variables from distinct subtrees ω_i and ω_j. Hence, the query is acyclic. Since the views in the subtrees $T_1, \ldots, T_k$ can be materialized in time $\mathcal{O}(N)$, it follows from Lemma 30 that the views in the view tree returned by $\textsc{NewVT}(X, V_X, \mathcal{F}_X, \{T_i\}_{i\in[k]})$ can also be materialized in time $\mathcal{O}(N)$.

Case: $X \notin \mathcal{F}'$**:** The procedure FactVT outputs the tree returned by $\textsc{NewVT}(X, V_X, \mathcal{F}'_X, \{T_i\}_{i\in[k]})$ with $\mathcal{F}'_X = \mathsf{anc}(X) \cup (\mathcal{F}' \cap vars(\omega_X))$. We cannot directly apply Lemma 30, since $\mathcal{F}'_X \neq \mathsf{anc}(X) \cup \{X\}$. Below we will show that Q_X must contain an atom $R(\mathcal{Y})$ with $\mathcal{F}' \subseteq \mathcal{Y}$. This implies $\mathcal{F}'_X \subseteq \mathcal{Y}$. Using the latter property, we can easily materialize the view $V_X(\mathcal{F}'_X) = V_1(\mathcal{F}'_1), \ldots, V_k(\mathcal{F}'_k)$ defined in NewVT as follows. We first use the InsideOut algorithm [2] to aggregate away all variables that are not included in $\mathcal{F}'_X$. Since the query $V_X(\mathcal{F}'_X)$ must be acyclic (by Remark 29) and the views $V_1, \ldots, V_k$ have size $\mathcal{O}(N)$, this aggregation step can be done in $\mathcal{O}(N)$ time. Let $V'_X(\mathcal{F}'_X) = V'_1(\mathcal{F}''_1), \ldots, V'_m(\mathcal{F}''_k)$ be the resulting query. Note that $\mathcal{F}'_X = \bigcup_{i\in[m]} \mathcal{F}''_i$. We iterate over the tuples in $R(\mathcal{Y})$. For each tuple $\mathbf{t}$, we check whether its restriction $\mathbf{t}_i$ to $\mathcal{F}''_1$ is contained in V'_i for each $i \in [m]$. If yes, the projection of $\mathbf{t}$ onto $\mathcal{F}'_X$ is included in the result of V'_X with multiplicity $V'_1(\mathbf{t}_m) \cdot \ldots \cdot V'_m(\mathbf{t}_m)$. Otherwise not. This procedure needs $\mathcal{O}(N)$ time. Then, by using an analogous argumentation as in the proof of Lemma 30, it follows that the views in the view tree returned by $\textsc{NewVT}(X, V_X, \mathcal{F}'_X, \{T_i\}_{i\in[k]})$ can be materialized in time $\mathcal{O}(N)$. It remains to show:

If $X \notin \mathcal{F}'$**,** Q_X **contains atom** $R(\mathcal{Y})$ **with** $\mathcal{F}' \subseteq \mathcal{Y}$**:** For the sake of contradiction, assume that $\mathcal{F}'$ contains two variables Y and Z such that there is no atom in Q_X that covers both. Let Q'_X result from Q_X by adding the atom $R(Y, Z)$ to Q_X. We show that Q'_X cannot be acyclic. This implies that Q_X is not free-connex [15]. For any pair of variables from $\{X, Y, Z\}$, the query Q'_X contains an atom that covers both, but there is no atom covering all three variables. Let $\omega' = (T, dep_{\omega'})$ be an arbitrary (not necessarily free-top) variable order for Q'_X. Due to their mutual dependencies, all three variables must be on the same root-to-leaf path in ω'. Without loss of generality, assume that Z is the lowest of these variables in ω'. It must hold $\{X, Y\} \subseteq dep_{\omega'}(Z)$. Since $\rho^*(Q_{\{X,Y,Z\}})$ must be greater than 1, $\mathsf{w}(\omega')$ cannot be 1. This means that Q'_X cannot be acyclic and, hence, Q_X is not free-connex, which contradicts our initial assumption. This completes the induction step. □

The next lemma states that the view trees returned by the procedure IndicatorVTs from Figure 7 can be materialized in time linear in the database size.

Lemma 32. *Let ω be canonical variable order, X a variable in ω, and N the database size. The views in the view trees returned by* IndicatorVTs(ω_X) *given in Figure 7 can be materialized in time $\mathcal{O}(N)$.*

Proof. The procedure first defines the view trees $alltree$ = FactVT($All, \omega_X, \mathcal{F}$) (line 3) and $ltree$ = FactVT($L, {\omega_X}^{\mathcal{F}}, \mathcal{F}$) (line 4), where $\mathcal{F}$ consists of the set $\mathsf{anc}(X) \cup \{X\}$. The variable order ${\omega_X}^{\mathcal{F}}$ results from ω_X by replacing each atom $R(\mathcal{Y})$ by an atom $R^{\mathcal{F}}(\mathcal{Y})$, which denotes the light part of relation R partitioned on $\mathcal{F}$. These light parts can be computed in linear time in the size of the input database. Note that the variables in $\mathcal{F}$ cannot occur in ω_X, which means that the query $Q_X(\mathcal{F}) =$ join of $atoms(\omega_X)$ is free-connex. By using Lemma 31, we derive that the views in $alltree$ and $ltree$ can be materialized in time $\mathcal{O}(N)$. Hence, the roots $allroot$ and $lroot$ of $alltree$ and $ltree$ and the complement $\neg lroot$ of $lroot$, can be materialized in $\mathcal{O}(N)$ as well. By Lemma 30, it follows that the views in the view tree $htree$ returned by NewVT($X, H_X, \mathcal{F}, \{allroot, \neg lroot\}$) called at (line 7) can be materialized in time $\mathcal{O}(N)$. Overall, all views in the pair of view trees ($htree, ltree$) returned by IndicatorVTs can be materialized in time $\mathcal{O}(N)$. □

We use Lemmas 31 and 32 to show an upper bound on the time to materialize the views in any tree produced by the procedure τ.

Lemma 33. *Let ω be a variable order, X a node in ω, $\mathcal{F}$ a set of variables in ω, N the database size, and $\epsilon \in [0,1]$. The views in each view tree returned by the procedure $\tau(\omega_X, \mathcal{F})$ given in Figure 8 can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X,\mathcal{F})-1)\epsilon\}})$.*

Proof. The proof is by induction on the structure of ω_X.

Base case: Assume that ω_X is a single atom $R(\mathcal{Y})$. In this case, the procedure τ returns the atom itself (Line 1). The atom can obviously be materialized in time $\mathcal{O}(N)$. It hods $\xi(X, \mathcal{F}) = 0$, since ω_X does not contain any node which is a variable. This means that $\max\{1, 1 + (\xi(X, \mathcal{F}) - 1)\epsilon\} = 1$. Then, the statement in the lemma holds for the base case.

Inductive step: Assume that ω_X is a variable order with root variable X and subtrees $\omega_1, \ldots, \omega_k$. Let $X_1, \ldots, X_k$ be the roots of these subtrees, $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega_X))$, and $keys = \mathsf{anc}(X) \cup \{X\}$. Following the control flow in $\tau(\omega_X, \mathcal{F})$, we make a case distinction.

Case 1 : mode = 'static' $\wedge$ $Q_X(\mathcal{F}_X)$ is free-connex or mode = 'dynamic' $\wedge$ $Q_X(\mathcal{F}_X)$ is q-hierarchical (lines 5-7): The procedure calls FactVT($V, \omega_X, \mathcal{F}_X$) (line 7). Since q-hierarchical queries are in particular free-connex, it follows from Lemma 31 that the latter procedure returns a view tree whose views can be materialized in time $\mathcal{O}(N)$. This completes the inductive step for Case 1.

Case 2 : Case 1 does not hold and $X \in \mathcal{F}$ (lines 8-10): In this case, the algorithm calls for each $T_1 \in \tau(\omega_1, \mathcal{F}), \ldots, T_k \in \tau(\omega_k, \mathcal{F})$, the procedure NewVT($X, V_X, keys, \{T_i\}_{i\in[k]}$). We consider one such tuple $T_1, \ldots, T_k$ of view trees. By induction hypothesis, the views in each T_i can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X_i,\mathcal{F})-1)\epsilon\}})$ for $i \in [k]$. Let $V_i(\mathcal{F}_i)$ be the root view of T_i for $i \in [k]$. By Remark 29, $keys$ is included in the schema of each root view and the query $V_X(keys) = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$ is acyclic. It follows from Lemma 30 that the views in the tree T produced by the procedure NewVT($X, V_X, keys, \{T_i\}_{i\in[k]}$) can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X_j,\mathcal{F})-1)\epsilon\}})$, where $j \in [k]$ is chosen such that $\xi(X_j, \mathcal{F})$ is maximal. Since X is free, $\xi(X, \mathcal{F})$ must be equal to $\xi(X_j, \mathcal{F})$. It follows that the views in T can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X,\mathcal{F})-1)\epsilon\}})$. This completes the inductive step in this case.

Case 3 : Case 1 does not hold and $X \notin \mathcal{F}$ (lines 11-18): We first give a lower bound on $\xi(X, \mathcal{F})$. Note that there cannot be any relation that covers variables in two distinct subtrees ω_i and ω_j of X. Moreover, X is bound. Therefore, $\xi(X, \mathcal{F}) \geq \sum_{i\in[k]} \xi(X_i, \mathcal{F})$.

The procedure τ first calls IndicatorVTs(ω_X) (line 11) given in Figure 7, which returns $htree$ and $ltree$. By Lemma 32, the views in these view trees can be materiaized in time $\mathcal{O}(N)$. Let H_X and L_X be the roots of $htree$ and $ltree$, respectively. For each $T_1 \in \tau(\omega_1, \mathcal{F}), \ldots, T_k \in \tau(\omega_k, \mathcal{F})$, τ calls NewVT

with the arguments $(X, V_X, keys, \{\exists H_X\} \cup \{T_i\}_{i\in[k]})$ (lines 12-13). Let $T_1, \ldots, T_k$ be one such tuple of view trees. By induction hypothesis, the views in each T_i with $i \in [k]$ can be materialized in time and have size $\mathcal{O}(N^{\max\{1,1+(\xi(X_i,\mathcal{F})-1)\epsilon\}})$. It follows from Remark 29 that the query $V_X(keys) = \exists H_X(keys), V_1(\mathcal{F}_1), \ldots V_k(\mathcal{F}_k)$, where each $V_i(\mathcal{F}_i)$ is the root view of T_i with $i \in [k]$ is acyclic. By Lemma 30, this means that the views in tree T returned by $\textsc{NewVT}(X, V_X, keys, \{\exists H_X\} \cup \{T_i\}_{i\in[k]})$ can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X_j,\mathcal{F})-1)\epsilon\}})$ where $j \in [k]$ such that $\xi(X_j, \mathcal{F}) = \max_{i\in[k]}\{\xi(X_i, \mathcal{F})\}$. It follows that the views in T can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(X,\mathcal{F})-1)\epsilon\}})$, since $\xi(X, \mathcal{F}) \geq \sum_{i\in[k]} \xi(X_i, \mathcal{F})$. In lines 14-16, the procedure τ uses the procedure NewVT to aggregate away all bound non-join variables in the atoms $R_1(\mathcal{F}_1), \ldots, R_k(\mathcal{F}_k)$ occurring at the leaves of ω_X. Using the InsideOut algorithm this can be done in time $\mathcal{O}(N)$. Let $vrels = \{R_1(\mathcal{F}'_1), \ldots, R_k(\mathcal{F}'_k)\}$ be the set of resulting atoms.

Time to materialize the join of $R_1(\mathcal{F}'_1), \ldots, R_k(\mathcal{F}'_k)$: The induction step is concluded by the analysis of the time to materialize the views in the tree T' returned by the procedure $\textsc{NewVT}(X, V_X, \mathcal{F}_X, \{\exists L_X\} \cup vrels)$ called in line 17. The materialization time for the views in T' is dominated by the time to materialize the view

$$V_X(\mathcal{F}_X) = \exists L_X(keys), R_1(\mathcal{F}'_1), \ldots, R_k(\mathcal{F}'_k)$$

with $keys = \mathsf{anc}(X) \cup \{X\}$ and $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega_X))$, as defined in the procedure NewVT. We materialize $V_X(\mathcal{F}_X)$ as follows. By using the InsideOut algorithm [2], we first aggregate away all variables in $vars(Q) - \mathcal{F}_X$ that are not above free variables in the variable order. Since the query is acyclic, the runtime for this procedure is linear in the database size. Let

$$V'_X(\mathcal{F}_X) = \exists L_X(keys), R_1(\mathcal{F}''_1), \ldots, R_k(\mathcal{F}''_k)$$

be the resulting query. If the atoms of V'_X do not contain any variable from ω_X, this means that $\mathcal{F}_X$ and each $\mathcal{F}''_i$ are contained in $\{X\} \cup \mathsf{anc}(X)$. Due to the definition of canonical variable orders, the set $\{X\} \cup \mathsf{anc}(X)$ must be contained in the schema of a single atom. Thus, we can materialize the result of $V'_X(\mathcal{F}_X)$ in linear time by iterating over the tuples in one of the atoms $R_i(\mathcal{F}''_i)$ and doing constant-time lookups in the remaining ones.

We now consider the case that the atoms of V'_X still contain variables from ω_X. In this case, each bound Y must be an ancestor node of a free variable Z in ω_X. Since Q is hierarchical, the restriction $Q|_{\mathcal{F}\cap vars(\omega_X)}$ must be hierarchical as well. Hence, there is an integral edge cover $\boldsymbol{\lambda} = \rho(Q|_{\mathcal{F}\cap vars(\omega_X)})$, which is equal to $\rho^*(Q|_{\mathcal{F}\cap vars(\omega_X)})$ (Lemma 11). Note that $\boldsymbol{\lambda}$ maps base atoms at the leaves of ω_X, which are the atoms $R_1(\mathcal{F}_1), \ldots, R_k(\mathcal{F}_k)$, to non-zero values, since other atoms cannot cover variables in ω_X. There must be at least one $i \in [k]$ with $\lambda_{R_i(\mathcal{F}_i)} = 1$, otherwise we fall back to the previous case where all variables in ω_X are aggregated away. For each atom $R_i(\mathcal{F}''_i)$ with $\lambda_{R_i(\mathcal{F}_i)} = 0$, there must be a *witness* atom $R_j(\mathcal{F}''_j)$ such that $\lambda_{R_j(\mathcal{F}_j)} = 1$ and $\mathcal{F}''_i \subseteq \mathcal{F}''_j$. We compute the full join of $\exists L_X(keys), R_1(\mathcal{F}''_1), \ldots, R_k(\mathcal{F}''_k)$ as follows. We choose an arbitrary atom $R_i(\mathcal{F}''_i)$ with $\lambda_{R_j(\mathcal{F}_j)} = 1$ and iterate over its tuples. For each such tuple, we check in constant time whether it is included in the light indicator L_X. If yes, we iterate over the matching tuples in the other atoms mapped to 1 by λ. For atoms that are not mapped to 1, it suffices to do constant-time lookups while iterating over one of their witnesses. Finally, we aggregate away all variables not included in $\mathcal{F}_X$. Since L_X is a light indicator, each tuple in $R_i(\mathcal{F}''_j)$ is paired with less than $\frac{3}{2}N^\epsilon$ tuples in each other relation. Hence, the computation time of the join is $\mathcal{O}(N^{1+(\rho^*(Q')-1)\epsilon})$, where $Q' = Q|_{\mathcal{F}\cap vars(\omega_X)}$. Taking into account the case that $\mathcal{F} \cap vars(\omega_X) = \emptyset$ and the computation of $V_X(\mathcal{F}_X)$ only requires that bound variables are aggregated away, the overall time complexity becomes $\mathcal{O}(N^{\max\{1,1+(\rho^*(Q')-1)\epsilon\}})$. The definition of $\xi(X, \mathcal{F})$ and the fact that $vars(\omega_X) \cap \mathcal{F}_X = vars(\omega_X) \cap \mathcal{F}$ imply that $\rho^*(Q') = \xi(X, \mathcal{F})$. □

The following lemma builds the bridge between the measure ξ and the factorization width w.

Lemma 34. *Let $Q(\mathcal{F})$ be a hierarchical query with factorization width w, ω a canonical variable order for $Q(\mathcal{F})$, and $\epsilon \in [0, 1]$. It holds*

$$\max\{1, 1 + (\xi(X, \mathcal{F}) - 1)\epsilon\} \leq 1 + (\mathsf{w} - 1)\epsilon.$$

Proof. We first consider the case that $\xi(X, \mathcal{F}) = 0$. This means that $\max\{1, 1 + (\xi(X, \mathcal{F}) - 1)\epsilon\} = 1$. Since we assume that Q contains at least one relation with non-empty schema, w must be at least 1. Thus, the inequality stated in the lemma holds in this case. Now, let $\xi(X, \mathcal{F}) = \ell \geq 1$. It suffices to show that $\mathsf{w} \geq \ell$. It follows from $\xi(X, \mathcal{F}) = \ell$ that there is a bound variable Y in ω such that there is a set $B \subseteq vars(\omega_Y) \cap \mathcal{F}$ with $\rho^*(Q|_B) = \ell$. Note that due to the definition of canonical variable orders, for each variable $Z \in B$, there must be a relation in Q that contains both Y and Z, which means that Y and Z depend on each other. Let $\omega' = (T, dep_{\omega'})$ be an an arbitrary free-top variable order for Q. Since Y is bound and the variables in B are free, all variables in B must be above Y in ω'. Moreover, $B \subseteq dep_{\omega'}(Y)$. By the definition of the factorization width, w must be at least $\rho^*(Q|_B) = \ell$. □

Proposition 20 follows from Lemmas 33 and 34.

D Further Details for Section 4

Proposition 21. *The tuples in the result of a hierarchical query $Q(\mathcal{F})$ over a database of size N can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay using the view trees $\tau(\omega, \mathcal{F})$ for a canonical variable order ω of Q.*

Proof. Following Proposition 19, the union of queries defined by the set of view trees constructed by $\tau(\omega, \mathcal{F})$ is equivalent to the query $Q(\mathcal{F})$. We enumerate the tuples over $\mathcal{F}$ from this set of view trees using the *next* calls of these trees in the set.

We first discuss the case of one view tree. In case there are no indicator views, then the view tree is a factorized representation that admits constant delay [44]. In the static case, this happens for input free-connex queries; in the dynamic case, this happens for q-hierarchical queries (Section 3.1).

The light indicators do not bring additional difficulty. By construction (Section 3), the free variables of the parent view T of such an indicator are the free variables of the query that are present in the view tree rooted at T. Furthermore, the *open* and *next* calls stop at T and do not explore the children of T, including the light indicator. This means that for enumeration purposes we can discard the descendants of the parent of a light indicator.

The heavy indicators require more careful treatment. For the purpose of enumeration, we ground a heavy indicator, that is, we replace the view tree rooted at its parent with a union of its instances, one per tuple in the indicator (the *open* procedure in Figure 10). This grounding does not lead however to a factorized representation in the sense of [44]: The relations represented by the different view tree instances replacing T may overlap. Constant delay is unlikely in this case: We create indicators in case of non-free-connex hierarchical queries in the static case (non-q-hierarchical queries in the dynamic case) and such queries do not admit constant-delay enumeration unless Boolean matrix multiplication conjecture fails [9].

We first consider the case of one heavy indicator. It can have at most $N^{1-\epsilon}$ tuples, so it leads to many view tree instances at its parent T. All instances represent relations over the same schema $\mathcal{F}_T$, which is the variables of the view at T. From each instance we can enumerate with constant delay and we can also look up a tuple with schema $\mathcal{F}_T$ in constant time. Given there are at most $N^{1-\epsilon}$ of them, we can enumerate from T with $\mathcal{O}(N^{1-\epsilon})$ delay.

We now consider the case of several heavy indicators $H_1(\mathcal{X}_1), \ldots, H_p(\mathcal{X}_h)$ whose parents $V_1(\mathcal{X}_1), \ldots, V_h(\mathcal{X}_p)$ are along the same path in the view tree. Let us assume V_i is an ancestor of V_j for $i < j$. By construction, there is a total strict inclusion order on their sets of variables, with the indicator above having less variables than at a lower depth: $\mathcal{X}_1 \subset \cdots \subset \mathcal{X}_p$. Each indicator draws its tuples from the input relations whose schemas include that of the indicator. There is also an inclusion between the parent views: $V_i \subseteq \pi_{\mathcal{X}_i} V_{i+1}, \forall i \in [p-1]$. This holds since V_i is defined by the join of the leaves underneath, so the view V_j that is a descendant of V_i is used to define V_i in joins with other views or relations. The size of V_i is at most that of H_i since they both have the same schema and the former is defined by the join of the latter with other views. Since the size of H_i is at most $N^{1-\epsilon}$, it follows that the size of V_i is also at most $N^{1-\epsilon}$. When grounding H_i, we create an instance for each tuple t that is both H_i and V_i: If t is not in V_i, it means that there is at least one sibling of H_i that does not have it. When opening the descendants of V_i before enumeration, only those

tuples in V_i can be extended with additional fields at its descendants, including all views V_j for $j > i$. This means that the overall number of groundings for the h heavy indicators is at most $N^{1-\epsilon}$. Let n_i be the number of instances of H_i. Then, the delay for enumerating from the union of H_i instances is $\sum_{i \leq j \leq p} n_j$ using the UNION algorithm, which also accounts for the delay incurred for enumeration from unions at instances of all H_j that are descendants of H_i. The overall delay is that for the union of instances for H_1: $\sum_{1 \leq j \leq p} n_j \leq p \times N^{1-\epsilon} = \mathcal{O}(N^{1-\epsilon})$.

We finally consider the case of heavy indicators whose parents $V_1, \ldots, V_p$ are now not all along the same path in the view tree. Each path is treated as in the previous case. We distinguish two cases. In the first case, there is no parent V_i that is an ancestor of several other parents in our list. Let W be a common ancestor of several parents. Then, the enumeration algorithm uses each tuple of W (possibly extended by descendant views) as context for the instances of these parents. A next tuple is produced *in sequence* at each of these parents over their corresponding schemas. These tuples are then composed into a larger tuple over a larger schema at their common ancestor using the PRODUCT algorithm. The number of branches is bounded by the number of atoms in the query, which means that the overall delay remains $\mathcal{O}(N^{1-\epsilon})$. In the second case, a parent V_i is a common ancestor of several other parents in our list. We reason similarly to the one-path case and obtain that the overall delay is less that $p \times N^{1-\epsilon}$.

So far we discussed the case of enumerating from one view tree. In case of a set of view trees we again use the UNION algorithm to enumerate the distinct tuples. In case the query has several connected components, i.e., it is a Cartesian product of hierarchical queries, we use the PRODUCT algorithm with an empty context. □

E Further Details for Section 5

In this section, we prove Propositions 22, 23, 24, and 25.

E.1 Proof of Proposition 22

Proposition 22. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ under a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ time.*

We first show the time needed to materialize the views constructed by FACTVT from Figure 4 given a q-hierarchical query. We then analyze the procedure APPLY$(T, \delta R)$ that maintains the views in any view tree T constructed by $\tau(\omega, \mathcal{F})$ under an update δR; APPLY is used in Figures 13, 14, and 15. We finally show the running times of UPDATEINDTREE from Figure 13 and UPDATEVIEWTREE from Figure 14.

Lemma 35. *Given a q-hierarchical query $Q(\mathcal{F})$, a canonical variable order of ω for Q, and a database of size N, the views in the view tree returned by the procedure* FACTVT$(\cdot, \omega, \mathcal{F})$ *from Figure 4 in the dynamic mode can be materialized in $\mathcal{O}(N)$ time and maintained under a single-tuple update in $\mathcal{O}(1)$ time.*

Proof. Lemma 31 shows that materializing the views constructed using FACTVT for free-connex queries, which also includes q-hierarchical queries, takes linear time.

We next show that the factorized view tree constructed for a q-hierarchical query using the function FACTVT from Figure 4 admits constant update time. At each node X of a variable order ω for a q-hierarchical query, the set $\mathcal{F}_X$ of free variables is either $\mathsf{anc}(X) \cup \{X\}$ if X is free, or $\mathsf{anc}(X)$ if X is bound because $\mathcal{F} \cap \mathit{vars}(\omega) = \emptyset$ for q-hierarchical queries. The function NEWVT maintains the following invariant in the dynamic mode: If X has a sibling node in ω, then the view created at node X has $\mathsf{anc}(X)$ as free variables. If X is bound, then already $\mathcal{F} = \mathsf{anc}(X)$; otherwise, NEWVT constructs an extra view with $\mathsf{anc}(X)$ as free variables (see lines 5-6 in Figure 5).

Now consider an update δR to a relation R. Due to the hierarchical property of the input query, the update δR fixes the values of all variables on the path from the leaf R to the root to constants. While propagating an update through the factorized view tree, the delta at each node X requires joining with the views constructed for the siblings of X. Each of the sibling views has $\mathsf{anc}(X)$ as free variables, as discussed

above. Thus, computing the delta at each node will require only constant-time lookups in the sibling views. Thus, propagating the update through the factorized view tree constructed for a q-hierarchical query using FACTVT takes constant time. □

Lemma 36. *Consider a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order of ω for Q, a database of size N, $\epsilon \in [0,1]$, and any view tree T produced by $\tau(\omega, \mathcal{F})$ in the dynamic mode. The procedure* APPLY$(T, \delta R)$ *that maintains the views of T under a single-tuple update δR runs in $\mathcal{O}(N^{\delta\epsilon})$ time.*

Proof. If Q is q-hierarchical, the procedure τ returns a factorized view tree for computing Q, which admits constant update time, per Lemma 35. If Q is not q-hierarchical, the procedure constructs two subtrees for each bound join variable X: one in the heavy case for computing Q_X but with X being free, and one in the light case for computing Q_X over the input relations below X, retaining only the free variables.

Consider now a view tree constructed by τ, where the views created in the light case are treated as leaf relations. This restricted view tree is a factorized view tree of a q-hierarchical query! As the procedure τ traverses the variable order in a top-down manner, every bound variable X is replaced by either a free variable (heavy case) or by a view that aggregates away X (light case) and now serves as input. Thus, single-tuple updates to the leaf relations of this restricted view tree can be done in constant time. That is, updates to the relations that are not part of the views materialized in the light case are constant.

However, updates to the relations that are part of the views materialized in the light case are not constant. For such a view V_X constructed at bound variable X, the cost of applying an update is captured by the delta width of the query Q_X at X joining the input relations from below X and the light indicator on X and its ancestors. Due to the hierarchical property of Q_X, an update to any input relation from below X fixes the values of X and its ancestors to constants in each input relation. Since these relations are joined with a light indicator, the size of each relation is reduced to $\mathcal{O}(N^\epsilon)$. The cost of computing the delta of Q_X for updates to input relations is determined by $\max_{R \in atoms(Q_X)} \rho^*(Q_X|_{vars(Q_X)-njb(Q_X)-\mathcal{S}(R)})$. This yields the maintenance cost of $\mathcal{O}(N^{\delta(Q_X)\epsilon})$. The change computed at V_X for a single-tuple update consist of $\mathcal{O}(N^{\delta(Q_X)\epsilon})$ tuples and needs to be propagated further up in the tree. Because there are no further light cases on the path from X to the root, the propagation cost is constant per tuple. By the definition of the delta width of the query Q, the cost of computing the delta of Q is the maximum of the maintenance costs of the views constructed at bound join variables in the light case. Thus, the overall time needed to maintain the views in the set of view trees created by τ for Q under a single-tuple update to any relation is $\mathcal{O}(N^{\delta(Q)\epsilon})$. □

Lemma 37. *Given an indicator tree T_{Ind} produced by the procedure* INDICATORVTS *and a single-tuple update δR to any input relation, the procedure* UPDATEINDTREE *from Figure 13 runs in $\mathcal{O}(1)$ time.*

Proof. The indicator tree T_{Ind} can be either the light or heavy indicator tree, both created using the function INDICATORVTS from Figure 7. Let T_{Ind} be constructed at the root X of a variable order with the set of free variables $\mathcal{F} = \mathsf{anc}(X) \cup \{X\}$. We distinguish two cases: 1) T_{Ind} is a tree with the root $H_X(\mathcal{F})$ joining children $All_X(\mathcal{F})$ and $\nexists L_X(\mathcal{F})$ (line 7 in Figure 7). Maintaining H_X under an update to either All_X or $\nexists L$ takes constant time. 2) T_{Ind} is a factorized view tree constructed for the query $Q_X(\mathcal{F})$ that joins the light parts of the relations in $\omega^{\mathcal{F}}$ (line 5 in Figure 7). The query Q_X is q-hierarchical, and the views in the factorized view tree constructed for Q_X admit constant update time, per Lemma 35. □

We next analyze the running time of the procedure UPDATETREES from Figure 14. We first apply the update to each view tree from $\mathcal{T}$ (line 2), which takes $\mathcal{O}(N^{\delta\epsilon})$ time, per Lemma 36. We then apply the update to each triple (T_{All}, T_L, T_H) of indicator view trees. The tree T_{All} is a factorized view tree of a q-hierarchical query, thus updating it takes constant time (line 7). The tree T_L is updated using UPDATEINDTREE in constant time (line 12), as per Lemma 37. Both of these changes may trigger an update to T_H. As discussed in the proof of Lemma 36, this update does not affect the views materialized in the light case by the procedure τ; thus, propagating the update $\delta\exists H$ through each view tree from $\mathcal{T}$ takes constant time (lines 10 and 15).

Propagating the update $\delta(\exists L)$, however, may require non-constant maintenance work. We consider two cases: 1) The multiplicity of a tuple $\mathbf{t}$ in L changed from 0 to non-zero. Then, we propagate this update through each view tree from $\mathcal{T}$, that is, through each materialized view constructed in the light cases by

τ (line 13). The reason why $\delta(\exists L)$ changed is because the multiplicity of a tuple containing $\mathbf{t}$ in the light part of an input relation increased from 0 to 1. Thus, when computing the change in each materialized view constructed for the light case, one input relation has exactly one tuple containing $\mathbf{t}$. Thus, computing this change takes $\mathcal{O}(N^{\delta\epsilon})$ time, same as when considering single-tuple updates to the relations of views materialized in the light case (see Lemma 36). 2) The multiplicity of a tuple $\mathbf{t}$ in L changed from non-zero to 0. Then, the light part of one of the input relations is already empty, thus applying this update to each view tree from $\mathcal{T}$ takes constant time.

Overall, the procedure UpdateTrees needs $\mathcal{O}(N^{\delta\epsilon})$ time to maintain the views constructed by τ under an update.

E.2 Proofs of Propositions 23 and 24

Proposition 23. *Given a hierarchical query $Q(\mathcal{F})$ with factorization width w, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, major rebalancing of the views in the set of view trees $\tau(\omega, \mathcal{F})$ takes $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time.*

Proof. We consider the major rebalancing procedure from Figure 15. The light relation parts can be computed in $\mathcal{O}(N)$ time. Proposition 20 implies that the affected views can be recomputed in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. □

Proposition 24. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, minor rebalancing of the views in the set of view trees $\tau(\omega, \mathcal{F})$ takes $\mathcal{O}(N^{(\delta+1)\epsilon})$ time.*

Proof. Figure 16 shows the procedure for minor rebalancing of the tuples with the partitioning value *key* in the light part $R^{\mathcal{F}}$ of relation R. Minor rebalancing either inserts fewer than $\frac{1}{2}\Theta^{\epsilon}$ tuples into $R^{\mathcal{F}}$ (heavy to light) or deletes at most $\frac{3}{2}\Theta^{\epsilon}$ tuples from $R^{\mathcal{F}}$ (light to heavy). Each action updates the indicator trees T_L and T_H in constant time (lines 4 and 6), per Lemma 37. The change $\delta(\exists L)$ is non-zero only after inserting the first tuple with the value *key* into $R^{\mathcal{F}}$ or deleting the last tuple with the value *key* from $R^{\mathcal{F}}$ (line 5). In the former case, $R^{\mathcal{F}}$ contains only one tuple (just inserted tuple), thus propagating the change $\delta(\exists L)$ to each view tree from $\mathcal{T}$ takes $\mathcal{O}(N^{\delta\epsilon})$ time. In the latter case, $R^{\mathcal{F}}$ has no more tuples with the value *key*, meaning that $\delta(\exists L)$ cannot cause any further changes in the view trees from $\mathcal{T}$. Propagating the change $\delta(\exists H)$ through each view tree from $\mathcal{T}$ takes constant time (line 7), as discussed in the proof of Proposition 22. Since there are $\mathcal{O}(\Theta^{\epsilon})$ such operations and the size invariant $\lfloor \frac{1}{4}\Theta \rfloor \leq N < \Theta$ holds, the total time is $\mathcal{O}(N^{(\delta+1)\epsilon})$. □

E.3 Proof of Proposition 25

Proposition 25. *Given a hierarchical query $Q(\mathcal{F})$ with delta width δ, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ under a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ amortized time.*

The proof of Proposition 25 is based on the proof from prior work (Section 4.1 in [28]). We start by defining the state that our approach initially creates and maintains on updates.

Definition 38. *Given a database of size N and $\epsilon \in [0,1]$, a state of the database is a tuple $\mathcal{Z} = (\epsilon, \Theta, \mathcal{T}, \mathcal{T}^{Ind})$, where:*

- *Θ is a natural number such that the size invariant $\lfloor \frac{1}{4}\Theta \rfloor \leq N < \Theta$ holds. Θ is called the threshold base.*
- *$\mathcal{T}$ is a set of view trees produced by the procedure τ from Figure 8.*
- *$\mathcal{T}^{Ind}$ is a set of triplets (T_{All}, T_L, T_H) of indicator view trees produced by the procedure τ from Figure 8.*

The initial state $\mathcal{Z}$ of the database has $\Theta = 2 \cdot N + 1$ and the view trees $\mathcal{T}$ and the indicator view trees $\mathcal{T}_{Ind}$ constructed in the preprocessing stage.

Proof of Proposition 25. Let $\mathcal{Z}_0 = (\epsilon, \Theta_0, \mathcal{T}_0, \mathcal{T}_0^{Ind})$ be the initial state of a database of size N_0 and $u_0, u_1, \ldots, u_{n-1}$ a sequence of arbitrary single-tuple updates.

The application of this update sequence to $\mathcal{Z}_0$ yields a sequence $\mathcal{Z}_0 \xrightarrow{u_0} \mathcal{Z}_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{n-1}} \mathcal{Z}_n$ of states, where $\mathcal{Z}_{i+1}$ is the result of executing the procedure OnUpdate$(\mathcal{T}_i, \mathcal{T}_i^{Ind}, u_i)$ from Figure 16, for $0 \leq i < n$. Let c_i denote the actual execution cost of OnUpdate$(\mathcal{T}_i, \mathcal{T}_i^{Ind}, u_i)$. For some $\Gamma > 0$, we can decompose each c_i as:

$$c_i = c_i^{apply} + c_i^{major} + c_i^{minor} + \Gamma, \qquad \text{for } 0 \leq i < n,$$

where c_i^{apply}, c_i^{major}, and c_i^{minor} are the actual costs of the subprocedures UpdateTrees, MajorRebalancing, and MinorRebalancing, respectively, in OnUpdate. If update u_i causes no major rebalancing, then $c_i^{major} = 0$; similarly, if u_i causes no minor rebalancing, then $c_i^{minor} = 0$. These actual costs admit the following worst-case upper bounds:

$$\begin{aligned}
c_i^{apply} &\leq \gamma \Theta_i^{\delta\epsilon} && \text{(by Proposition 22)},\\
c_i^{major} &\leq \gamma \Theta_i^{1+(\mathsf{w}-1)\epsilon} && \text{(by Proposition 23), and}\\
c_i^{minor} &\leq \gamma \Theta_i^{(\delta+1)\epsilon} && \text{(by Proposition 24)},
\end{aligned}$$

where γ is a constant derived from their asymptotic bounds, and Θ_i is the threshold base of $\mathcal{Z}_i$. The rebalancing steps have higher asymptotic costs than processing one update.

The crux of this proof is to show that assigning an *amortized cost* $\hat{c}_i$ to each update u_i accumulates enough budget to pay for such expensive but less frequent rebalancing procedures. For any sequence of n updates, we show that the accumulated amortized cost is no smaller than the accumulated actual cost:

$$\sum_{i=0}^{n-1} \hat{c}_i \geq \sum_{i=0}^{n-1} c_i. \tag{3}$$

The amortized cost assigned to an update u_i is $\hat{c}_i = \hat{c}_i^{apply} + \hat{c}_i^{major} + \hat{c}_i^{minor} + \Gamma$, where

$$\hat{c}_i^{apply} = \gamma \Theta_i^{\delta\epsilon}, \hat{c}_i^{major} = 4\gamma \Theta_i^{(\mathsf{w}-1)\epsilon}, \hat{c}_i^{minor} = \gamma \Theta_i^{\delta\epsilon}, \text{ and}$$

Γ and γ are the constants used to upper bound the actual cost of OnUpdate. In contrast to the actual costs c_i^{major} and c_i^{minor}, the amortized costs $\hat{c}_i^{major}$ and $\hat{c}_i^{minor}$ are always nonzero.

We prove that such amortized costs satisfy Inequality (3). Since $\hat{c}_i^{apply} \geq c_i^{apply}$ for $0 \leq i < n$, it suffices to show that the following inequalities hold:

$$\sum_{i=0}^{n-1} \hat{c}_i^{major} \geq \sum_{i=0}^{n-1} c_i^{major} \quad \text{and} \tag{4}$$

$$\sum_{i=0}^{n-1} \hat{c}_i^{minor} \geq \sum_{i=0}^{n-1} c_i^{minor}. \tag{5}$$

We prove Inequalities (4) and (5) by induction on the length n of the update sequence.

Major rebalancing.

- *Base case*: We show that Inequality (4) holds for $n = 1$. The preprocessing stage sets $\Theta_0 = 2 \cdot N_0 + 1$. If the initial database is empty, i.e., $N_0 = 0$, then $\Theta_0 = 1$ and u_0 triggers major rebalancing (and no minor rebalancing). The amortized cost $\hat{c}_0^{major} = 4\gamma \Theta_0^{(\mathsf{w}-1)\epsilon} = 4\gamma$ suffices to cover the actual cost

$c_0^{major} \leq \gamma \Theta_0^{1+(\mathsf{w}-1)\epsilon} = \gamma$. If the initial database is nonempty, u_0 cannot trigger major rebalancing (i.e., violate the size invariant) because $\lfloor \frac{1}{4}\Theta_0 \rfloor = \lfloor \frac{1}{2}N_0 \rfloor \leq N_0 - 1$ (lower threshold) and $N_0 + 1 < \Theta_0 = 2 \cdot N_0 + 1$ (upper threshold); then, $\hat{c}_0^{major} \geq c_0^{major} = 0$. Thus, Inequality (4) holds for $n = 1$.

- *Inductive step*: Assumed that Inequality (4) holds for all update sequences of length up to $n-1$, we show it holds for update sequences of length n. If update u_{n-1} causes no major rebalancing, then $\hat{c}_{n-1}^{major} = 4\gamma\Theta_{n-1}^{(\mathsf{w}-1)\epsilon} \geq 0$ and $c_{n-1}^{major} = 0$, thus Inequality (4) holds for n. Otherwise, if applying u_{n-1} violates the size invariant, the database size N_n is either $\lfloor \frac{1}{4}\Theta_{n-1} \rfloor - 1$ or Θ_{n-1}. Let $\mathcal{Z}_j$ be the state created after the previous major rebalancing or, if there is no such step, the initial state. For the former ($j > 0$), the major rebalancing step ensures $N_j = \frac{1}{2}\Theta_j$ after doubling and $N_j = \frac{1}{2}\Theta_j - \frac{1}{2}$ or $N_j = \frac{1}{2}\Theta_j - 1$ after halving the threshold base Θ_j; for the latter ($j = 0$), the preprocessing stage ensures $N_j = \frac{1}{2}\Theta_j - \frac{1}{2}$. The threshold base Θ_j changes only with major rebalancing, thus $\Theta_j = \Theta_{j+1} = \ldots = \Theta_{n-1}$. The number of updates needed to change the database size from N_j to N_n (i.e., between two major rebalancing) is at least $\frac{1}{4}\Theta_{n-1}$ since $\min\{\frac{1}{2}\Theta_j - 1 - (\lfloor \frac{1}{4}\Theta_{n-1} \rfloor - 1), \Theta_{n-1} - \frac{1}{2}\Theta_j\} \geq \frac{1}{4}\Theta_{n-1}$. Then,

$$
\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{major} &\geq \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} \hat{c}_i^{major} \ \textit{(ind. hyp.)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} 4\gamma\Theta_{n-1}^{(\mathsf{w}-1)\epsilon} \\
&\quad \textit{($\Theta_j = \ldots = \Theta_{n-1}$)} \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + \frac{1}{4}\Theta_{n-1}\, 4\gamma\Theta_{n-1}^{(\mathsf{w}-1)\epsilon} \\
&\quad \textit{(at least $\frac{1}{4}\Theta_{n-1}$ updates)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \gamma\Theta_{n-1}^{1+(\mathsf{w}-1)\epsilon} \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + c_{n-1}^{major} = \sum_{i=0}^{n-1} c_i^{major} \\
&\quad \textit{($c_j^{major} = \ldots = c_{n-2}^{major} = 0$).}
\end{aligned}
$$

Thus, Inequality (4) holds for update sequences of length n.

Minor rebalancing. When the degree of a tuple of values in a partition changes such that the heavy or light part condition no longer holds, minor rebalancing deletes heavy tuples from or inserts light tuples into the light part of the relation. To prove Inequality (5), we decompose the cost of minor rebalancing per triples of indicator trees, relation partitions, and data values of its partitioning key.

$$
c_i^{minor} = \sum_{(T_{All}, T_L, T_H) \in \mathcal{T}_{Ind}} \sum_{R^{\mathcal{F}} \in T_L} \sum_{key \in \mathsf{Dom}(\mathcal{F})} c_i^{R^{\mathcal{F}}, key} \quad \text{and}
$$

$$
\hat{c}_i^{minor} = \sum_{(T_{All}, T_L, T_H) \in \mathcal{T}_{Ind}} \sum_{R^{\mathcal{F}} \in T_L} \sum_{key \in \mathsf{Dom}(\mathcal{F})} \hat{c}_i^{R^{\mathcal{F}}, key}
$$

We write $c_i^{R^{\mathcal{F}}, key}$ and $\hat{c}_i^{R^{\mathcal{F}}, key}$ to denote the actual and respectively amortized costs of minor rebalancing caused by update u_i, for a light part $R^{\mathcal{F}}$ partitioned on $\mathcal{F}$ and a tuple key with the schema $\mathcal{F}$ whose value

comes from u_i. We denote the set of the light parts of a relation R by

$$\mathcal{R} = \{R^{\mathcal{F}} \mid (T_{All}, T_L, T_H) \in \mathcal{T}_{Ind}, R^{\mathcal{F}} \in T_L\}$$

Consider the update u_i of the form $\delta R = \{\mathbf{t} \to m\}$. If update u_i causes minor rebalancing, then $\sum_{R^{\mathcal{F}} \in \mathcal{R}} c_i^{R^{\mathcal{F}}, \pi_{\mathcal{F}}\mathbf{t}} = c_i^{minor}$; otherwise, $\sum_{R^{\mathcal{F}} \in \mathcal{R}} c_i^{R^{\mathcal{F}}, \pi_{\mathcal{F}}\mathbf{t}} = 0$. The amortized cost is $\sum_{R^{\mathcal{F}} \in \mathcal{R}} \hat{c}_i^{R^{\mathcal{F}}, \pi_{\mathcal{F}}\mathbf{t}} = \hat{c}_i^{minor}$ regardless of whether u_i causes minor rebalancing or not; otherwise, $\sum_{R^{\mathcal{F}} \in \mathcal{R}} \hat{c}_i^{R^{\mathcal{F}}, \pi_{\mathcal{F}}\mathbf{t}} = 0$. We prove that for a light part $R^{\mathcal{F}}$ of the partition of a relation R, and any $key \in \mathsf{Dom}(\mathcal{F})$ the following inequality holds:

$$\sum_{i=0}^{n-1} \hat{c}_i^{R^{\mathcal{F}}, key} \geq \sum_{i=0}^{n-1} c_i^{R^{\mathcal{F}}, key}. \tag{6}$$

Since the number of relation partitions of a relation is constant, Inequality (5) follows directly from Inequality (6). We prove Inequality (6) by induction on the length n of the update sequence.

- *Base case*: We show that Inequality (6) holds for $n = 1$. Assume that update u_0 is of the form $\delta R = \{\mathbf{t} \to m\}$; otherwise, $\hat{c}_0^{R^{\mathcal{F}}, key} = c_0^{R^{\mathcal{F}}, key} = 0$, and Inequality (6) follows trivially for $n = 1$. If the initial database is empty, u_0 triggers major rebalancing but no minor rebalancing, thus $\hat{c}_0^{R^{\mathcal{F}}, key} = \gamma \Theta_0^{\delta\epsilon} \geq c_0^{R^{\mathcal{F}}, key} = 0$. If the initial database is nonempty, each relation is partitioned using the threshold Θ_0^{ϵ}. For update u_0 to trigger the minor rebalancing of $R^{\mathcal{F}}$, the degree of the $\mathcal{F}$ tuple key in $R^{\mathcal{F}}$ has to either decrease from $\lceil \Theta_0^{\epsilon} \rceil$ to $\lceil \frac{1}{2}\Theta_0^{\epsilon} \rceil - 1$ (heavy to light) or increase from $\lceil \Theta_0^{\epsilon} \rceil - 1$ to $\lceil \frac{3}{2}\Theta_0^{\epsilon} \rceil$ (light to heavy). The former happens only if $\lceil \Theta_0^{\epsilon} \rceil = 1$ and update u_0 removes the last tuple with the $\mathcal{F}$-tuple key from $R^{\mathcal{F}}$, thus no minor rebalancing is needed; the latter cannot happen since update u_0 can increase $|\sigma_{\mathcal{F}=key} R^{\mathcal{F}}|$ to at most $\lceil \Theta_0^{\epsilon} \rceil$, and $\lceil \Theta_0^{\epsilon} \rceil < \lceil \frac{3}{2}\Theta_0^{\epsilon} \rceil$. In any case, $\hat{c}_0^{R^{\mathcal{F}}, key} \geq c_0^{R^{\mathcal{F}}, key}$, which implies that Inequality (6) holds for $n = 1$.
- *Inductive step*: Assuming that Inequality (6) holds for all update sequences of length up to $n-1$, we show it holds for update sequences of length n. Consider that update u_{n-1} is of the form $\delta R = \{\mathbf{t} \to m\}$ and causes minor rebalancing for the light part $R^{\mathcal{F}}$; otherwise, $\hat{c}_{n-1}^{R^{\mathcal{F}}, key} \geq 0$ and $c_{n-1}^{R^{\mathcal{F}}, key} = 0$, and Inequality (6) follows trivially for n. Let $\mathcal{Z}_j$ be the state created after the previous major rebalancing or, if there is no such step, the initial state. The threshold changes only with major rebalancing, thus $\Theta_j = \Theta_{j+1} = \ldots = \Theta_{n-1}$. Depending on the existence of minor rebalancing steps since state $\mathcal{Z}_j$, we have two cases:
  - *Case 1:* There is no minor rebalancing for $R^{\mathcal{F}}$ caused by an update of the form since state $\mathcal{Z}_j$; thus, $c_j^{R^{\mathcal{F}}, key} = \ldots = c_{n-2}^{R^{\mathcal{F}}, key} = 0$. From state $\mathcal{Z}_j$ to state $\mathcal{Z}_n$, the number of tuples with the $\mathcal{F}$ tuple key either decreases from at least $\lceil \Theta_j^{\epsilon} \rceil$ to $\lceil \frac{1}{2}\Theta_{n-1}^{\epsilon} \rceil - 1$ (heavy to light) or increases from at most $\lceil \Theta_j^{\epsilon} \rceil - 1$ to $\lceil \frac{3}{2}\Theta_{n-1}^{\epsilon} \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than $\frac{1}{2}\Theta_{n-1}^{\epsilon}$ since $\Theta_j = \Theta_{n-1}$ and $\min\{\lceil \Theta_j^{\epsilon} \rceil - (\lceil \frac{1}{2}\Theta_{n-1}^{\epsilon} \rceil - 1), \lceil \frac{3}{2}\Theta_{n-1}^{\epsilon} \rceil - (\lceil \Theta_j^{\epsilon} \rceil - 1)\} > \frac{1}{2}\Theta_{n-1}^{\epsilon}$. Then,

$$\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{R^{\mathcal{F}},key} &\geq \sum_{i=0}^{j-1} c_i^{R^{\mathcal{F}},key} + \sum_{i=j}^{n-1} \hat{c}_i^{R^{\mathcal{F}},key} \quad \textit{(ind. hyp.)} \\
&= \sum_{i=0}^{j-1} c_i^{R^{\mathcal{F}},key} + \sum_{i=j}^{n-1} \gamma\Theta_{n-1}^{\delta\epsilon} \\
&\quad (\Theta_j = \ldots = \Theta_{n-1}) \\
&> \sum_{i=0}^{j-1} c_i^{R^{\mathcal{F}},key} + \Theta_{n-1}^{\epsilon}\gamma\Theta_{n-1}^{\delta\epsilon} \\
&\quad \textit{(more than } \Theta_{n-1}^{\epsilon} \textit{ updates)} \\
&\geq \sum_{i=0}^{j-1} c_i^{R^{\mathcal{F}},key} + c_{n-1}^{R^{\mathcal{F}},key} \\
&= \sum_{i=0}^{n-1} c_i^{R^{\mathcal{F}},key} \quad (c_j^{R^{\mathcal{F}},key} = \ldots = c_{n-2}^{R^{\mathcal{F}},key} = 0).
\end{aligned}$$

– *Case 2:* There is at least one minor rebalancing step for $R^{\mathcal{F}}$ caused by an update of the form $\delta R = \{\mathbf{t}' \to m'\}$ where $\pi_{\mathcal{F}}\mathbf{t}' = key$ since state $\mathcal{Z}_j$. Let $\mathcal{Z}_\ell$ denote the state created after the previous minor rebalancing caused by an update of this form; thus, $c_\ell^{R^{\mathcal{F}},key} = \ldots = c_{n-2}^{R^{\mathcal{F}},key} = 0$. The minor rebalancing steps creating $\mathcal{Z}_\ell$ and $\mathcal{Z}_n$ inserts or deletes tuples with the $\mathcal{F}$ tuples *key*. From state $\mathcal{Z}_\ell$ to state $\mathcal{Z}_n$, the number of such tuples either decreases from $\lceil \frac{3}{2}\Theta_l^{\epsilon} \rceil$ to $\lceil \frac{1}{2}\Theta_{n-1}^{\epsilon} \rceil - 1$ (heavy to light) or increases from $\lceil \frac{1}{2}\Theta_l^{\epsilon} \rceil - 1$ to $\lceil \frac{3}{2}\Theta_{n-1}^{\epsilon} \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than Θ_{n-1}^{ϵ} since $\Theta_l = \Theta_{n-1}$ and $\min\{\lceil \frac{3}{2}\Theta_l^{\epsilon} \rceil - (\lceil \frac{1}{2}\Theta_{n-1}^{\epsilon} \rceil - 1), \lceil \frac{3}{2}\Theta_{n-1}^{\epsilon} \rceil - (\lceil \frac{1}{2}\Theta_l^{\epsilon} \rceil - 1)\} > \Theta_{n-1}^{\epsilon}$. Then,

$$\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{R^{\mathcal{F}},key} &\geq \sum_{i=0}^{\ell-1} c_i^{R^{\mathcal{F}},key} + \sum_{i=\ell}^{n-1} \hat{c}_i^{R^{\mathcal{F}},key} \quad \textit{(ind. hyp.)} \\
&= \sum_{i=0}^{\ell-1} c_i^{R^{\mathcal{F}},key} + \sum_{i=\ell}^{n-1} \gamma\Theta_{n-1}^{\delta\epsilon} \\
&\quad (\Theta_j = \ldots = \Theta_{n-1}) \\
&> \sum_{i=0}^{\ell-1} c_i^{R^{\mathcal{F}},key} + \Theta_{n-1}^{\epsilon}\gamma\Theta_{n-1}^{\delta\epsilon} \\
&\quad \textit{(more than } \Theta_{n-1}^{\epsilon} \textit{ updates)} \\
&> \sum_{i=0}^{\ell-1} c_i^{R^{\mathcal{F}},key} + c_{n-1}^{R^{\mathcal{F}},key} \\
&= \sum_{i=0}^{n-1} c_i^{R^{\mathcal{F}},key} \quad (c_\ell^{R^{\mathcal{F}},key} = \ldots = c_{n-2}^{R^{\mathcal{F}},key} = 0).
\end{aligned}$$

Cases 1 and 2 imply that Inequality (6) holds for update sequences of length n.

This shows that Inequality (3) holds when the amortized cost of OnUpdate$(\mathcal{T}_i, \mathcal{T}_i^{Ind}, u_i)$ is

$$\hat{c}_i = \gamma\Theta_i^{\delta\epsilon} + 4\gamma\Theta_i^{(\mathsf{w}-1)\epsilon} + \gamma\Theta_i^{\delta\epsilon} + \Gamma, \quad \text{for } 0 \leq i < n,$$

where Γ and γ are constants. The amortized cost $\hat{c}_i^{major}$ of major rebalancing is $4\gamma\Theta_i^{(\mathsf{w}-1)\epsilon}$, and the amortized cost $\hat{c}_i^{minor}$ of minor rebalancing is $\gamma\Theta_i^{\delta\epsilon}$. From the size invariant $\left\lfloor \frac{1}{4}\Theta_i \right\rfloor \leq N_i < \Theta_i$ follows that $N_i < \Theta_i < 4(N_i + 1)$ for $0 \leq i < n$, where N_i is the database size before update u_i. This implies that for any database of size N, the amortized major rebalancing time is $\mathcal{O}(N^{(\mathsf{w}-1)\epsilon})$ and the amortized minor rebalancing time is $\mathcal{O}(N^{\delta\epsilon})$. From $w - 1 \leq d$ in Proposition 15, it follows that the overall amortized update time is $\mathcal{O}(N^{\delta\epsilon})$. □