Logical Methods in Computer Science
Volume 19, Issue 3, 2023, pp. 11:1–11:50
https://lmcs.episciences.org/

Submitted Sep. 13, 2022
Published Aug. 09, 2023

TRADE-OFFS IN STATIC AND DYNAMIC EVALUATION OF HIERARCHICAL QUERIES

AHMET KARA [a], MILOS NIKOLIC [b], DAN OLTEANU [a], AND HAOZHE ZHANG [a]

[a] University of Zurich
e-mail address: kara@ifi.uzh.ch, olteanu@ifi.uzh.ch, zhang@ifi.uzh.ch

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

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 under single-tuple inserts or deletes to the database.

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

We define the preprocessing time, the update time, and the enumeration delay as functions of the light/heavy threshold. By appropriately choosing this threshold, our approach recovers a number of prior results when restricted to hierarchical queries.

We show that for a restricted class of hierarchical queries, our approach achieves worst-case optimal update time and enumeration delay conditioned on the Online Matrix-Vector Multiplication Conjecture.

1. Introduction

The problems of static evaluation, i.e., computing the result of a query [Yan81, OZ15, KNS17, NPRR18], and dynamic evaluation, i.e., maintaining the result of a query under inserts and deletes of tuples to the input relations [Koc10, CY12, K$^+$14, BKS17a, IUV17, KNN$^+$19a], are fundamental to relational databases.

We consider a refinement of these two problems that decomposes the overall evaluation time into the *preprocessing time*, which is used to compute a data structure that represents 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 is the time between the start of the enumeration process and the output of the first tuple in the query result, the time between outputting any two consecutive tuples, and the time between outputting the last tuple and the end of the enumeration process [DG07]. In this paper we investigate

Key words and phrases: adaptive evaluation, incremental maintenance.

the relationship between preprocessing, update, and delay and answer questions such as, how much preprocessing time is needed to achieve sublinear enumeration delay.

We consider the static and dynamic evaluation of a subclass of α-acyclic queries called hierarchical queries:

Definition 1.1 [SORK11, BKS17a]**.** 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 enjoy highly desirable tractability properties in a variety of computational settings, making them an important yardstick for database computation. The notion of hierarchical queries used in this paper has been initially introduced in the context of probabilistic databases [SORK11]. 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 [SORK11]. This dichotomy was extended to non-Boolean queries with negation [FO16]. 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 [OZ12]. For hierarchical queries without self-joins, this read-once factorized representation explains their tractability for exact probability computation over probabilistic databases. In the Massively Parallel Computation (MPC) model, the hierarchical queries admit parallel evaluation with one communication step [KS11b]. 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 [HY19]. Hierarchical queries also admit one-step streaming evaluation in the finite cursor model [GGL$^+$09]. Under updates, the q-hierarchical queries are the conjunctive queries that admit constant-time update and delay [BKS17a]. The q-hierarchical queries are a proper subclass of both the free-connex α-acyclic and hierarchical queries. Besides being hierarchical, the following condition holds on the free variables of a q-hierarchical query: if the set of atoms of a free variable is strictly contained in the set of another variable, then the latter must also be free.

In this paper we characterize trade-offs in the static and dynamic evaluation of hierarchical queries. In the static setting, we are interested in the trade-off between preprocessing time and enumeration delay. In the dynamic case, we additionally consider the update time. Section 2 states our main result in the static setting and explains how it recovers prior results on static query evaluation. Section 3 gives our main result in the dynamic setting and discusses its implications. These two sections also overview prior work on static and dynamic query evaluation. Section 4 introduces the basic notions underlying our approach. Sections 5-7 detail the preprocessing, enumeration, and update stages of our approach. Section 8 shows that for a restricted class of hierarchical queries, our approach achieves worst-case optimal update time and enumeration delay, conditioned on the Online Matrix-Vector Multiplication Conjecture. We illustrate our approach using two detailed examples in Section 9 and conclude in Section 10. The proofs of the main theorems in

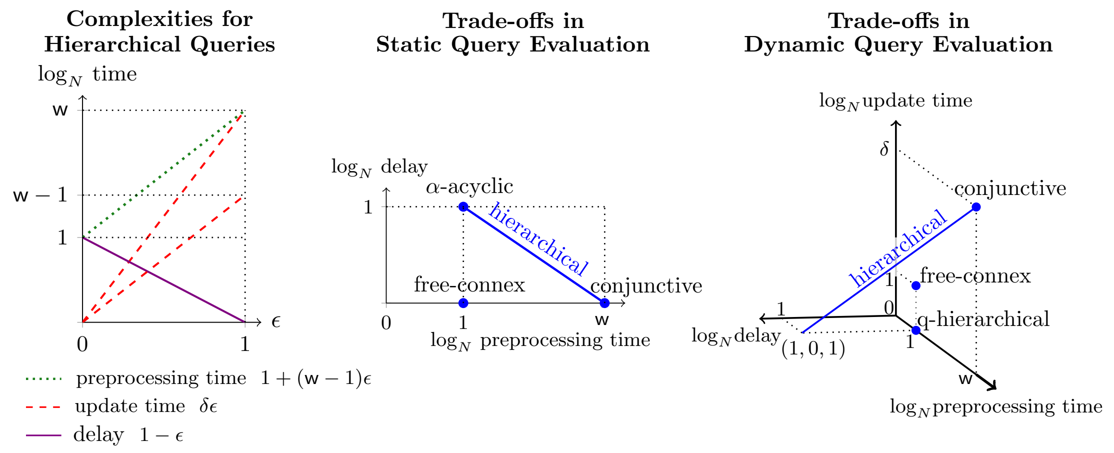


FIGURE 1. Left: Preprocessing time, enumeration delay, and amortized update time for a hierarchical query with static width w and dynamic width δ (δ can be w or $\mathsf{w}-1$, hence the two red lines for the update time). Middle and right: Trade-offs in static and dynamic evaluation. Our approach achieves each blue point and each point on the blue lines. Prior approaches are represented by the blue points.

Sections 2 and 3 and the propositions in Sections 5-7 are deferred to Appendices A-E. The proofs of the propositions in Sections 4 and 8 can be found in the technical report [KNOZ19].

A preliminary version of this work appeared in PODS 2020 [KNOZ20]. We extended it as follows. We overviewed in greater depth and breadth the related work for a more complete picture of the state of the art (Sections 1-3). We added new motivating examples to demonstrate that our approach achieves better overall evaluation time than existing approaches both in the static and dynamic cases (Sections 2 and 3). We included necessary background on the computational model and width measures (Section 4). We added a detailed description of the algorithms (UNION and PRODUCT) used by the enumeration procedure of our approach (Section 6). We included the procedures for major and minor rebalancing in case of updates and gave the procedure for the maintenance of a query result under sequences of updates (Section 7). Finally, we included complete proofs of the main results and the main statements on the preprocessing, enumeration, and update stages of our approach (Appendices A-E).

2. TRADE-OFFS IN STATIC QUERY EVALUATION

Our main result for the static evaluation of hierarchical queries is stated next.

Theorem 2.1. *Given a hierarchical query with static width* w*, a database of size* N*, and* $\epsilon \in [0,1]$*, the query result can be enumerated with* $\mathcal{O}(N^{1-\epsilon})$ *delay after* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *preprocessing time.*

The measure w, previously introduced as $s^{\uparrow}$ [OZ15], generalizes the fractional hypertree width [Mar10] from Boolean to arbitrary conjunctive queries. This is equivalent to the FAQ-width in case of Functional Aggregate Queries over a single semiring [AKNR16]. In this paper, we refer to this measure as the static width of the query (Definition 4.6).

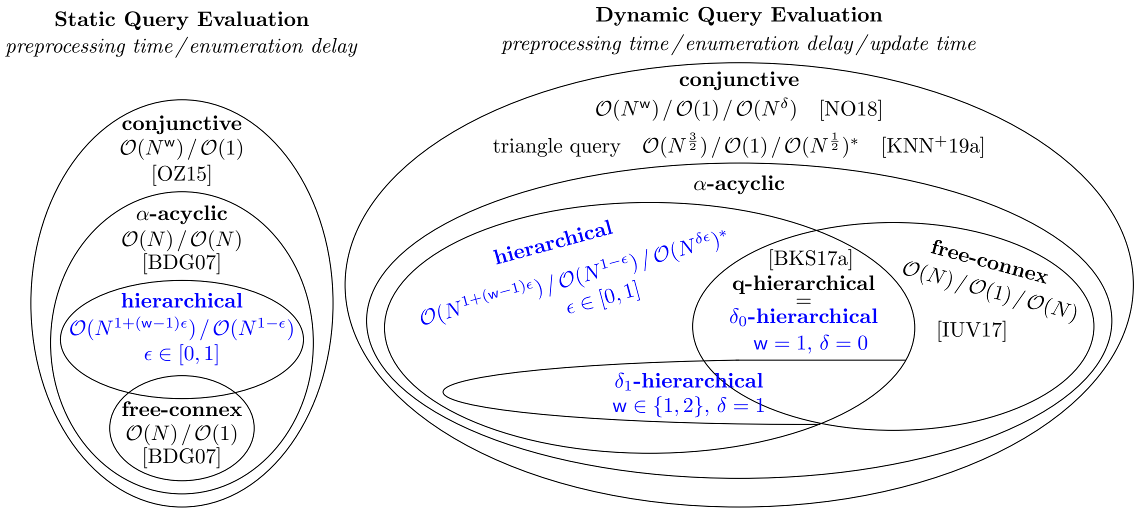

Static Query Evaluation
preprocessing time / enumeration delay
conjunctive
$\mathcal{O}(N^{\mathsf{w}}) / \mathcal{O}(1)$
[OZ15]
α-acyclic
$\mathcal{O}(N) / \mathcal{O}(N)$
[BDG07]
hierarchical
$\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon}) / \mathcal{O}(N^{1-\epsilon})$
$\epsilon \in [0,1]$
free-connex
$\mathcal{O}(N) / \mathcal{O}(1)$
[BDG07]
Dynamic Query Evaluation
preprocessing time / enumeration delay / update time
conjunctive
$\mathcal{O}(N^{\mathsf{w}}) / \mathcal{O}(1) / \mathcal{O}(N^{\delta})$ [NO18]
triangle query $\mathcal{O}(N^{\frac{3}{2}}) / \mathcal{O}(1) / \mathcal{O}(N^{\frac{1}{2}})^*$ [KNN+19a]
α-acyclic
hierarchical
$\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon}) / \mathcal{O}(N^{1-\epsilon}) / \mathcal{O}(N^{\delta\epsilon})^*$
$\epsilon \in [0,1]$
[BKS17a]
q-hierarchical
=
δ_0-hierarchical
$\mathsf{w} = 1,\ \delta = 0$
free-connex
$\mathcal{O}(N) / \mathcal{O}(1) / \mathcal{O}(N)$
[IUV17]
δ_1-hierarchical
$\mathsf{w} \in \{1,2\},\ \delta = 1$

FIGURE 2. Landscape of static and dynamic query evaluation. w: static width; δ: dynamic width; *: amortized time.

Theorem 2.1 expresses the runtime components as functions of a parameter ϵ. The dotted green line and the purple line in the left plot in Figure 1 depict the preprocessing time and respectively the enumeration delay. The middle plot in Figure 1 visualizes the trade-off between the two components. Our approach achieves each blue point and each point on the blue line. Prior approaches are represented by the blue points in the trade-off space. By appropriately setting ϵ, our approach recovers prior results restricted to hierarchical queries. For $\epsilon = 0$, both the preprocessing time and the delay become $O(N)$, as for α-acyclic queries [BDG07]. For $\epsilon = 1$, we obtain $\mathcal{O}(N^{\mathsf{w}})$ preprocessing time and $O(1)$ delay as for conjunctive queries [OZ15]. Free-connex acyclic queries are a special class of queries that enjoy linear preprocessing time and constant delay [BDG07]. We recover this result as follows. First, we observe that any free-connex hierarchical query has static width $\mathsf{w} = 1$. This means that the preprocessing time remains $O(N)$ regardless of ϵ; we then choose $\epsilon = 1$ to obtain $O(1)$ delay. For bounded-degree databases, i.e., where each value appears at most c times for some constant $c = N^{\beta}$, first-order queries admit $O(N)$ preprocessing time and $O(1)$ delay [DG07, KS11a]. We recover the $O(1)$ delay using $\epsilon = 1$. The preprocessing time becomes $\mathcal{O}(N \cdot (N^{\beta})^{\mathsf{w}-1}) = \mathcal{O}(N)$ if our approach uses the constant upper bound c instead of the upper bound N^{ϵ} on the degrees. The left Venn diagram in Figure 2 depicts the relationship of our result in Theorems 2.1 with prior results.

The next example demonstrates how the complexities of our approach in the static case imply lower overall evaluation time than existing approaches.

Example 2.2. Consider the hierarchical query $Q(A, C) = R(A, B), S(B, C)$. Let us assume that the input relations are of size N. Then, it takes quadratic time to compute the list of tuples in the query result of Q. (As it will become clearer later, this query has static width 2, which explains the $O(N^2)$ time complexity for the evaluation of Q.)

An *eager* evaluation does just this: It readily computes the list of tuples in the query result of Q. This requires quadratic preprocessing time, after which the tuples in the query result can be enumerated with constant delay [OZ12].

In contrast, a *lazy* evaluation approach computes the first tuple in the query result then the second tuple and so on. This can be done using linear preprocessing time followed by linear enumeration delay for each tuple in the result [BDG07]. It is conjectured that the delay cannot be lowered to constant after linear-time preprocessing for Q [BDG07]. (The explanation is that Q is not free-connex, a notion we will introduce in Section 4.)

Our approach achieves $\mathcal{O}(N^{1+\epsilon})$ preprocessing time and $\mathcal{O}(N^{1-\epsilon})$ enumeration delay for any $\epsilon \in [0,1]$. The complexities of the eager, lazy, and our approach are as follows:

approach	preprocessing	delay
lazy	$\mathcal{O}(N)$	$\mathcal{O}(N)$
eager	$\mathcal{O}(N^2)$	$\mathcal{O}(1)$
ours	$\mathcal{O}(N^{1+\epsilon})$	$\mathcal{O}(N^{1-\epsilon})$

Our approach recovers the lazy approach at $\epsilon = 0$ and the eager approach at $\epsilon = 1$. For any $\epsilon \in (0,1)$, it achieves new trade-offs between preprocessing time and enumeration delay.

Given that the input relations have size N, the AGM bound [AGM13] implies that the result of the query has at most N^2 tuples. Assume that we want to enumerate N^γ tuples from the result, for some $0 \leq \gamma \leq 2$. In the following table, the second to fourth rows give the exponents of the overall evaluation times achieved by the lazy, eager, and our approaches for different values of γ. The last row gives the ϵ values at which we achieve the complexities of our approach.

γ	0	$\frac{1}{2}$	1	$1\frac{1}{2}$	2
lazy	1	$1\frac{1}{2}$	2	$2\frac{1}{2}$	3
eager	2	2	2	2	2
ours	1	$1\frac{1}{4}$	$1\frac{1}{2}$	$1\frac{3}{4}$	2
ϵ	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1

For instance, if $\gamma = 1\frac{1}{2}$, the lazy approach requires $\mathcal{O}(N + N^{1+\frac{1}{2}} \cdot N) = \mathcal{O}(N^{2+\frac{1}{2}})$, the eager approach requires $\mathcal{O}(N^2 + N^{1+\frac{1}{2}} \cdot 1) = \mathcal{O}(N^2)$, and our approach needs only $\mathcal{O}(N^{1+\frac{3}{4}} + N^{1+\frac{1}{2}}N^{\frac{1}{4}}) = \mathcal{O}(N^{1+\frac{3}{4}})$ time at $\epsilon = \frac{3}{4}$. In case γ is equal to $\frac{1}{2}$, 1, or $1\frac{1}{2}$, the overall computation time of our approach (highlighted in green) is strictly lower than the eager and lazy approaches. For the other two cases shown in the table, our approach recovers the lower complexity of the prior approaches (highlighted in yellow). □

2.1. **Further Prior Work on Static Query Evaluation.**

We complement our discussion with further prior work on static query evaluation. Figure 3 gives a taxonomy of works in this area.

Prior work exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [DK18]. 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 projection, as addressed in our work. It also states Example 9.1 as an open problem.

The result of any α-acyclic conjunctive query can be enumerated with constant delay after linear-time preprocessing if and only if it is free-connex. This is under the conjecture that Boolean multiplication of $n \times n$ matrices cannot be done in $O(n^2)$ time [BDG07]. More recently, this was shown to hold also under the hypothesis that the existence of a triangle in

Class of Queries	Preprocessing	Delay	Extra Space	Source
f.c. α-acyclic CQ$^{\neq}$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	[BDG07]
f.c. β-acyclic negative CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[BB13, BB12]
f.c. signed-acyclic CQ	$\mathcal{O}(N\,(\log N)^{\lvert Q\rvert})$	$\mathcal{O}(1)$	–	[BB13]
Acyclic CQ$^{\neq}$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	[BDG07]
CQ$^{\neq}$ of f.c. treewidth k	$\mathcal{O}(\lvert\mathrm{Dom}\rvert^{k+1}+N)$	$\mathcal{O}(1)$	–	[BDG07]
CQ	$\mathcal{O}(N^{\mathsf{w}(Q)})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{\mathsf{w}(Q)})$	[OZ15, AKNR16]
Full CQ with access patterns	$\mathcal{O}(N^{\rho^*(Q)})$	$\mathcal{O}(\tau)$	$\mathcal{O}(N+N^{\rho^*(Q)}/\tau)$	[DK18]
CQ on $\underline{\text{X}}$-structures (trees, grids)	$\mathcal{O}(N)$	$\mathcal{O}(N)$	–	[BDFG10]
FO on bound. degree	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[DG07, KS11a]
FO on bound. expansion	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[KS13a]
FO on local bounded expansion	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	–	[SV17]
FO on low degree	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{2+\gamma})$	[DSS14]
FO on nowhere dense	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\gamma})$	[SSV18]
MSO on Bounded treewidth	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[Bag06, KS13b]

Figure 3. 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 [OZ15] and fractional edge cover number ρ^* [AGM13]; 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 –).

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 decided in linear time [BB13]. The free-connex characterization generalizes in the presence of functional dependencies [CK18]. An in-depth pre-2015 overview on constant-delay enumeration is provided by Segoufin [Seg15].

There are also enumeration algorithms for document spanners [ABMN19] and satisfying valuations of circuits [ABJM17].

3. Trade-offs in Dynamic Query Evaluation

Our main result for the dynamic query evaluation generalizes the static case.

Theorem 3.1. *Given a hierarchical query with static width* w *and dynamic width* δ*, a database of size* N*, and* $\epsilon \in [0,1]$*, the query result can be enumerated with* $\mathcal{O}(N^{1-\epsilon})$ *delay after* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *preprocessing time and* $\mathcal{O}(N^{\delta\epsilon})$ *amortized update time for single-tuple updates.*

The left plot in Figure 1 depicts the preprocessing time (dotted green line), the update time (dashed red lines), and the enumeration delay (purple line) of our approach in the dynamic case. For hierarchical queries, the dynamic width δ can be equal to either the static width w or $\mathsf{w}-1$ (Proposition 4.8). The plot hence shows two dashed red lines for the update time. The right plot in Figure 1 depicts the trade-off between the three components. Our approach can achieve sublinear amortized update time and delay for hierarchical queries with arbitrary free variables (Figure 1 left and right). For any $\epsilon = \frac{1}{\delta+\alpha} > 0$ with $\alpha > 0$, our algorithm has update time $\mathcal{O}(N^{1-\alpha\cdot\frac{1}{\delta+\alpha}})$ and delay $\mathcal{O}(N^{1-\frac{1}{\delta+\alpha}})$.

The update time for a single tuple is at most the preprocessing time: $\delta\epsilon \leq \mathsf{w}\epsilon \leq \epsilon + (\mathsf{w}-1)\epsilon \leq 1 + (\mathsf{w}-1)\epsilon$. If $\delta = \mathsf{w} - 1$, then $\delta\epsilon = (\mathsf{w}-1)\epsilon$, i.e., the update time is an $\mathcal{O}(N)$ factor less than the preprocessing time. The complexity of preprocessing thus amounts to inserting N tuples in an initially empty database using our update mechanism. If $\delta = \mathsf{w}$, then inserting N tuples would need $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon+\epsilon})$ time, which is an $\mathcal{O}(N^{\epsilon})$ factor more than the complexity of one bulk update using our preprocessing algorithm. This suggests a gap between single-tuple updates and bulk updates. A similar gap highlighting a fundamental limitation of single-tuple updates has been shown for the Loomis-Whitney query that generalizes the triangle query from a join of three binary relations to a join of n $(n-1)$-ary relations: The amortized update time for single-tuple updates is $\mathcal{O}(N^{1/2})$, which is worst-case optimal unless the Online Matrix-Vector Multiplication conjecture fails [KNN+19b]. Inserting N tuples in the empty database would cost $\mathcal{O}(N^{3/2})$, yet the query can be computed in the static setting in time $\mathcal{O}(N^{\frac{n}{n-1}})$ [NPRR18].

Amortized $\mathcal{O}(N^{\delta\epsilon})$ update time means that, given any sequence of updates, the *average* cost of a single update is $\mathcal{O}(N^{\delta\epsilon})$. Since updates can change the data structure, our approach needs to do a rebalancing step whenever the data structure gets out of balance. The time needed for a single update without rebalancing is $\mathcal{O}(N^{\delta\epsilon})$ in the worst case. A rebalancing step can require super-linear time (Propositions 7.3 and 7.4). We show that for any update sequence, the overall time needed for the updates and rebalancing steps, when averaged over the number of updates in the sequence, remains $\mathcal{O}(N^{\delta\epsilon})$ in the worst case (Proposition 7.5). Using classical de-amortization techniques [KP98], we can adapt our update mechanism to obtain non-amortized $\mathcal{O}(N^{\delta\epsilon})$ update time. The de-amortization strategy is analogous to the one used for the update mechanism of triangle queries (Section 10 in [KNN+20]), which performs more frequent but less time-consuming rebalancing steps.

Theorem 3.1 recovers prior work on conjunctive queries [NO18], free-connex acyclic queries[IUV17], and q-hierarchical queries [BKS17a] by setting $\epsilon = 1$ (Figure 1 right). For hierarchical queries in general, our approach achieves the same complexities as prior work on conjunctive queries when restricted to hierarchical queries. For free-connex queries, we obtain linear-time preprocessing and update and constant-time delay since $\mathsf{w} = 1$ (Proposition 4.9) and then $\delta \in \{0, 1\}$ (Proposition 4.8) for these queries. For q-hierarchical queries, we obtain linear-time preprocessing and constant-time update and delay since $\mathsf{w} = 1$ and $\delta = 0$. Existing maintenance approaches, e.g, classical first-order IVM [CY12] and higher-order recursive IVM [K+14], DynYannakakis [IUV17], and F-IVM [NO18], can achieve constant delay for general hierarchical queries yet after at least linear-time updates. The right Venn diagram in Figure 2 relates Theorem 3.1 with prior results.

The next example illustrates that our approach achieves better overall evaluation time than existing approaches when considering a sequence of updates.

Example 3.2. Let us consider the (free-connex hierarchical) query $Q(A) = R(A, B), S(B)$. The query has static and dynamic width 1. We assume that the input relations are of size N and consider the dynamic setting.

A *lazy* evaluation approach requires no preprocessing: For each single-tuple update, it only updates the input relations without propagating the changes to the query result. Before enumerating the A-values in the query result, it first scans the relation R to collect all A-values that are paired with B-values contained in S. This takes linear time. Afterwards, the approach can enumerate the A-values with constant delay.

An *eager* evaluation approach precomputes the initial result in linear time. On a single-tuple update, it computes the delta query obtained by fixing the variables of one relation to constants. For an update $\delta R(a, b)$ to R, the delta query $\delta Q(a) = \delta R(a, b), S(b)$ can be computed in constant time. For an update $\delta S(b)$ to S, the delta query $\delta Q(A) = R(A, b), \delta S(b)$ can be computed in linear time. In general, the update time is linear. Since the query result is materialized and eagerly maintained, the A-values in the result can be enumerated after an update with constant delay.

For this query, our approach achieves $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(N^{\epsilon})$ update time, and $\mathcal{O}(N^{1-\epsilon})$ enumeration delay for any $\epsilon \in [0, 1]$. The following table summarizes the preprocessing-update-delay trade-off achieved by the three approaches:

approach	preprocessing	update	delay
lazy	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$
eager	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$
ours	$\mathcal{O}(N)$	$\mathcal{O}(N^{\epsilon})$	$\mathcal{O}(N^{1-\epsilon})$

Our approach recovers the lazy and eager approaches by setting ϵ to 0 and respectively 1, with one exception: it cannot recover the constant preprocessing time in the lazy approach as it requires one pass over the input data.

Consider now a sequence of N^m updates, each followed by one access request to enumerate N^{γ} values out of the at most N A-values in the query result, for $m, \gamma \in [0, 1]$. With the eager and lazy approaches, this sequence takes time (excluding preprocessing) $\mathcal{O}(N^m(N + N^{\gamma}))$, which is $\mathcal{O}(N^{m+1})$ since $\gamma \leq 1$. With our approach, the sequence takes $\mathcal{O}(N^m(N^{\epsilon} + N^{\gamma}N^{1-\epsilon})) = \mathcal{O}(N^{m+\epsilon} + N^{m+\gamma+1-\epsilon})$. Depending on the values of m and γ, we can tune our approach (by appropriately setting ϵ) to minimize the overall time to execute the bulk of updates and access requests. For $\gamma < 1$ and any m, our approach has consistently lower complexity than the lazy/eager approaches, while for $\gamma = 1$ and any m it matches that of the lazy/eager approaches. The complexity of processing the sequence of updates and access requests is shown in the next table for various values of m and γ:

	our approach					eager/lazy approaches				
m \ γ	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1
0	$\frac{1}{2}$	$\frac{5}{8}$	$\frac{3}{4}$	$\frac{7}{8}$	1	1	1	1	1	1
$\frac{1}{2}$	1	$1\frac{1}{8}$	$1\frac{1}{4}$	$1\frac{3}{8}$	$1\frac{1}{2}$	$1\frac{1}{2}$	$1\frac{1}{2}$	$1\frac{1}{2}$	$1\frac{1}{2}$	$1\frac{1}{2}$
1	$1\frac{1}{2}$	$1\frac{5}{8}$	$1\frac{3}{4}$	$1\frac{7}{8}$	2	2	2	2	2	2
ϵ	$\frac{1}{2}$	$\frac{5}{8}$	$\frac{3}{4}$	$\frac{7}{8}$	1					

The middle five columns (highlighted by green and yellow) show the complexities for our approach. The last row states the values of ϵ for which the complexities in the same columns are obtained. The rightmost five columns show the complexities for the lazy/eager approaches for $\gamma \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$. They are all higher than for our approach, except for the last column for which $\gamma = 1$: Regardless of m, the complexity gap is $\mathcal{O}(N^{\frac{1}{2}})$ for $\gamma = 0$, $\mathcal{O}(N^{\frac{3}{8}})$ for $\gamma = \frac{1}{4}$, $\mathcal{O}(N^{\frac{1}{4}})$ for $\gamma = \frac{1}{2}$, and $\mathcal{O}(N^{\frac{1}{8}})$ for $\gamma = \frac{3}{4}$ For $\gamma = 1$, our approach defaults to the eager approach and achieves the lowest complexities for $\epsilon = 1$. □

Class of Queries	Preprocessing	Update	Delay	Extra Space	Source
q-hierarchical CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[BKS17a, IUV17]
Triangle count	$\mathcal{O}(N^{\frac{3}{2}})$	$\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$	[KNN$^+$19a]
Full triangle query	$\mathcal{O}(N^{\frac{3}{2}})$	$\mathcal{O}(N^{\frac{1}{2}})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(N^{\frac{3}{2}})$	[KNN$^+$20]
q-hierarchical UCQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[BKS18]
FO+MOD on bound. degree	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[BKS17b]
MSO on Strings	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$	–	[NS18]

Figure 4. 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 –). $\dagger$: amortized update time.

3.1. **Further Prior Work on Dynamic Query Evaluation.**

We discuss further prior work on dynamic query evaluation. Figure 4 gives a taxonomy of works in this field.

The q-hierarchical queries are the conjunctive queries that admit linear-time preprocessing and constant-time update and delay [BKS17a, IUV17]. If a conjunctive query without repeating relation symbols is not q-hierarchical, there is no $\gamma > 0$ such that the query result can be enumerated with $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ delay and update time, unless the Online Matrix Vector Multiplication conjecture fails. The constant delay and update time carry over to first-order queries with modulo-counting quantifiers on bounded degree databases, unions of q-hierarchical queries [BKS18], and q-hierarchical queries with small domain constraints [BKS17b].

Prior work characterizes the preprocessing-space-update trade-off for counting triangles under updates [KNN$^+$19a]. A follow-up work generalizes this approach to the triangle queries with arbitrary free variables, adding the enumeration delay to the trade-off space [KNN$^+$20]. In this work, we consider arbitrary hierarchical queries instead of the triangle queries, and we use a less trivial adaptive maintenance technique, where the same relation may be subject to partition on different tuples of variables and where the overall number of cases for each partition is reduced to only two: the all-light case and the at-least-one-heavy case.

MSO queries on strings admit 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 [LM14, ABM18].

DBToaster [K$^+$14], F-IVM [NO18], and DynYannakakis [IUV17, IUV$^+$18] are recent systems implementing incremental view maintenance approaches.

4. Preliminaries

Data Model. A schema $\mathcal{X} = (X_1, \dots, X_n)$ is a non-empty tuple of distinct variables. Each variable X_i has a discrete domain $\mathsf{Dom}(X_i)$. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple $\mathbf{x}$ of data values over schema $\mathcal{X}$ is an element from $\mathsf{Dom}(\mathcal{X}) = \mathsf{Dom}(X_1) \times \dots \times \mathsf{Dom}(X_n)$.

A relation R over schema $\mathcal{X}$ is a function $R : \mathsf{Dom}(\mathcal{X}) \to \mathbb{Z}$ such that the multiplicity $R(\mathbf{x})$ is non-zero for finitely many tuples $\mathbf{x}$. A tuple $\mathbf{x}$ is in R, denoted by $\mathbf{x} \in R$, if $R(\mathbf{x}) \neq 0$. The notation $\exists R$ denotes the use of R with set semantics: $\exists R(\mathbf{x})$ equals 1 if $\mathbf{x} \in R$ and 0 otherwise; also, $\nexists R(\mathbf{x}) = 1 - \exists R(\mathbf{x})$. The size $|R|$ of R is the size of the set $\{\mathbf{x} \mid \mathbf{x} \in R\}$. A database is a set of relations and has size given by the sum of the sizes of its relations.

Given a tuple $\mathbf{x}$ over schema $\mathcal{X}$ and $\mathcal{S} \subseteq \mathcal{X}$, $\mathbf{x}[\mathcal{S}]$ denotes the restriction of $\mathbf{x}$ to $\mathcal{S}$ such that the values in $\mathbf{x}[\mathcal{S}]$ follow the ordering in $\mathcal{S}$. For instance, $(a, b, c)[(C, A)] = (c, a)$ for the tuple (a, b, c) over the schema (A, B, C). For a relation R over $\mathcal{X}$, schema $\mathcal{S} \subseteq \mathcal{X}$, and tuple $\mathbf{t} \in \mathsf{Dom}(\mathcal{S})$, $\sigma_{\mathcal{S}=\mathbf{t}} R = \{\, \mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{S}] = \mathbf{t} \,\}$ denotes the set of tuples in R that agree with $\mathbf{t}$ on the variables in $\mathcal{S}$, while $\pi_{\mathcal{S}} R = \{\, \mathbf{x}[\mathcal{S}] \mid \mathbf{x} \in R \,\}$ denotes the set of restrictions of the tuples in R to the variables in $\mathcal{S}$.

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

In an idealized setting, the above requirements can be ensured using hashing. In practice, hashing can only achieve *amortized* constant time for some of the above operations. In our paper, whenever we claim constant time for hash operations, we mean amortized constant time. We give a hash-based example data structure that supports the above operations in amortized constant time. Consider a relation R over schema $\mathcal{X}$. A hash table with chaining stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple $\mathbf{x}$ over $\mathcal{X}$ with $R(\mathbf{x}) \neq 0$. The entries are doubly linked to support enumeration with constant delay. The hash table can report the number of its entries in constant time and supports lookups, inserts, and deletes in amortized constant time. To support index operations on a schema $\mathcal{F} \subset \mathcal{X}$, we create another hash table with chaining where each table entry stores an $\mathcal{F}$-value $\mathbf{t}$ as key and a doubly-linked list of pointers to the entries in R having $\mathbf{t}$ as $\mathcal{F}$-value. Looking up an index entry given $\mathbf{t}$ takes amortized constant time, 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 amortized constant time. For efficient deletion of index entries, each entry in R also stores back-pointers to its index entries (one back-pointer per index for R). When an entry is deleted from R, locating and deleting its index entries in doubly-linked lists takes constant time per index. An alternative data structure that can meet our requirements is a tree-structured index such as a B$^+$-tree. This would, however, require *worst-case* logarithmic time and imply an additional logarithmic factor in our complexity results.

Modeling Updates Using Multiplicities. We restrict multiplicities of tuples in the input relations to be strictly positive. Multiplicity 0 means the tuple is not present. A single-tuple update to a relation R is expressed as $\delta R = \{\mathbf{x} \to m\}$. The update is an insert of the tuple $\mathbf{x}$ in R if the multiplicity m is strictly positive. It is a delete of $\mathbf{x}$ from R if m is negative. Such a delete is rejected if the existing multiplicity of $\mathbf{x}$ in R is less than $|m|$. A batch update may consist of both inserts and deletes. Applying δR to R means creating a new version of R that is the union of δR and R.

Partitioning. The number of occurrences of a value in a relation is called the degree of the value in the relation. We partition relations based on value degree.

Definition 4.1. Given a relation R over schema $\mathcal{X}$, a schema $\mathcal{S} \subset \mathcal{X}$, and a threshold θ, the pair (H, L) of relations is a *partition* of R on $\mathcal{S}$ with threshold θ if it satisfies the following four conditions:

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

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

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

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 $\mathcal{S}$ with threshold $\theta = N^{\epsilon}$ for $\epsilon \in [0, 1]$, we have: $\forall \mathbf{t} \in \pi_{\mathcal{S}} L : |\sigma_{\mathcal{S}=\mathbf{t}} L| < \theta = N^{\epsilon}$; and $|\pi_{\mathcal{S}} H| \leq \frac{|R|}{\theta} = N^{1-\epsilon}$. We subsequently denote the light part of R on $\mathcal{S}$ by $R^{\mathcal{S}}$.

Queries. A conjunctive query (CQ) has the form

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

We denote by: $(R_i)_{i\in[n]}$ the relation symbols; $(R_i(\mathcal{X}_i))_{i\in[n]}$ the atoms; $\mathit{vars}(Q) = \bigcup_{i\in[n]} \mathcal{X}_i$ the set of variables; $\mathit{free}(Q) = \mathcal{F} \subseteq \mathit{vars}(Q)$ the set of *free* variables; $\mathit{bound}(Q) = \mathit{vars}(Q) - \mathit{free}(Q)$ the set of *bound* variables; $\mathit{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of the atoms; and $\mathit{atoms}(X)$ the set of the atoms containing X. The query Q is *full* if $\mathit{free}(Q) = \mathit{vars}(Q)$.

The hypergraph $G = (\mathit{vars}(Q), \mathit{atoms}(Q))$ of a query Q has one node per variable and one hyperedge per atom that covers all nodes representing its variables. A *join tree* for Q is a tree with the following properties: (1) Its nodes are exactly the atoms of Q; (2) if any two nodes have variables in common, then all nodes along the path between them also have these variables. The query Q is called *α-acyclic* if it has a join tree. It is *free-connex* if it is α-acyclic and remains α-acyclic when we add to its body a fresh atom over its free variables [BB13]. It is *hierarchical* if for any two of its variables, either their sets of atoms are disjoint or one is contained in the other. It is *q-hierarchical* if it is hierarchical and for every variable $A \in \mathit{free}(Q)$, if there is a variable B such that $\mathit{atoms}(A) \subset \mathit{atoms}(B)$ then $B \in \mathit{free}(Q)$ [BKS17a].

Example 4.2. The following query is α-acyclic:

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

A 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: $U(AEG) - T(AEF) - Q(ACF) - R(ABC) - S(ABD)$.

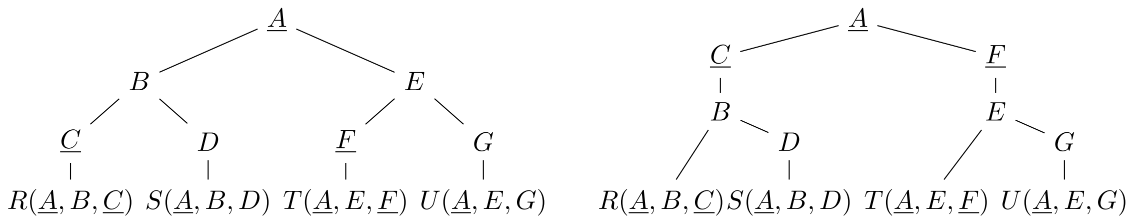


Figure 5. Canonical (left) and non-canonical but free-top (right) variable order for the query $Q(A,C,F) = R(A,B,C)$, $S(A,B,D)$, $T(A,E,F)$, $U(A,E,G)$ in Example 4.2. Free variables are underlined.

It is also hierarchical but not q-hierarchical: The bound variables B and E dominate the free variables C and respectively F. □

Variable Orders. Two variables *depend* on each other if they occur in the same atom.

Definition 4.3 (adapted from [OZ15])**.** A *variable order* ω for a conjunctive query Q is a pair (T, dep_ω) such that the following holds:

- T is a rooted forest with one node per variable in Q. The variables of each atom in Q lie along the same root-to-leaf path in T.
- 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, $\mathit{dep}_\omega(Y) \subseteq \mathit{dep}_\omega(X) \cup \{X\}$.

An *extended* variable order is a variable order where we add as new leaves the atoms corresponding to relations. We add each atom as the child of its variable placed lowest in the variable order. Whenever we refer to a variable order, we mean its extension with atoms at leaves. For ease of presentation, we often use ω to refer to the tree of ω.

The subtree of a variable order ω rooted at X is denoted by ω_X. The sets $\mathit{vars}(\omega)$, $\mathit{atoms}(\omega)$, and $\mathsf{anc}(X)$ consist of all variables of ω, the atoms at the leaves of ω, and the variables on the path from X to the root excluding X, respectively. The flag $\mathsf{has\_sibling}(X)$ is true if X has siblings. The variable order ω is *free-top* if no bound variable is an ancestor of a free variable (called d-tree extension [OZ15]). It is *canonical* if the variables of the leaf atom of each root-to-leaf path are the inner nodes of the path. The sets $\mathsf{freeTopVO}(Q)$, $\mathsf{canonVO}(Q)$, and $\mathsf{VO}(Q)$ consist of free-top, canonical, and all variable orders of Q.

Example 4.4. The left variable order in Figure 5 is a canonical variable order for the query from Example 4.2. This variable order is not free-top since the bound variables B and E sit on top of the free variables C and respectively F. The right variable order in Figure 5 is a free-top variable order for the query. This variable order 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, while q-hierarchical queries admit canonical free-top variable orders. The canonical variable order of a hierarchical query is unique up to orderings of variables sharing the same set of atoms.

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

$$\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{R(\mathcal{X}) \in\, atoms(Q)} \lambda_{R(\mathcal{X})} & \\
\text{subject to} & \displaystyle\sum_{R(\mathcal{X}) \in\, atoms(Q) \text{ s.t. } X \in \mathcal{X}} \lambda_{R(\mathcal{X})} \geq 1 & \text{for all } X \in \mathcal{F} \text{ and} \\
& \lambda_{R(\mathcal{X})} \in [0,1] & \text{for all } R(\mathcal{X}) \in atoms(Q)
\end{array}$$

The optimal objective value of the above program is called the *fractional edge cover number* of the variable set $\mathcal{F}$ and is denoted as $\rho^*_Q(\mathcal{F})$. An *integral edge cover* of $\mathcal{F}$ is a feasible solution to the variant of the above program with $\lambda_{R(\mathcal{X})} \in \{0,1\}$ for each $R(\mathcal{X}) \in atoms(Q)$. The optimal objective value of this program is called the *integral edge cover number* of $\mathcal{F}$ and is denoted as $\rho_Q(\mathcal{F})$. If Q is clear from the context, we omit the index Q in the expressions $\rho^*_Q(\mathcal{F})$ and $\rho_Q(\mathcal{F})$. For a database of size N, the result of the query Q can be computed in time $\mathcal{O}(N^{\rho^*})$ [NPRR18].

For hierarchical queries, the integral and fractional edge cover numbers are equal. The proofs of the following propositions in this section are given in the technical report [KNOZ19] (Appendices B and C).

Proposition 4.5. *For any hierarchical query Q and $\mathcal{F} \subseteq vars(Q)$, it holds $\rho^*(\mathcal{F}) = \rho(\mathcal{F})$.*

Definition 4.6. The *static width* of a conjunctive query Q is

$$\mathsf{w}(Q) = \min_{\omega \in \mathsf{freeTopVO}(Q)} \mathsf{w}(\omega)$$
$$\mathsf{w}(\omega) = \max_{X \in vars(Q)} \rho^*(\{X\} \cup dep_\omega(X))$$

If Q is Boolean, then w is the *fractional hypertree width* [Mar10]. *FAQ-width* generalizes w to queries over several semirings [AKNR16] [1].

Definition 4.7. The *dynamic width* of a conjunctive query Q is

$$\delta(Q) = \min_{\omega \in \mathsf{freeTopVO}(Q)} \delta(\omega)$$
$$\delta(\omega) = \max_{X \in vars(Q)} \max_{R(\mathcal{Y}) \in atoms(\omega_X)} \rho^*((\{X\} \cup dep_\omega(X)) - \mathcal{Y})$$

While the static width of a free-top variable order ω is defined over the sets $\{X\} \cup dep_\omega(X)$ with $X \in vars(Q)$, the dynamic width of ω is defined over restrictions of these sets obtained by dropping the variables in the schema of one atom. For any canonical variable order ω, variable X in ω, and atom $R(\mathcal{Y})$ in $atoms(\omega_X)$, the set $(\{X\} \cup dep_\omega(X)) - \mathcal{Y}$ is empty. Hence, queries that admit *canonical* free-top variable orders have dynamic width 0.

Proposition 4.8. *Given a hierarchical query with static width* w *and dynamic width δ, it holds that $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$.*

Free-connex hierarchical queries have static width 1.

[1]To simplify presentation, we focus on queries that contain at least one atom with non-empty schema. This implies that the static width of queries is at least 1. Queries where all atoms have empty schemas obviously admit constant preprocessing time, update time, and enumeration delay.

Proposition 4.9. *Any free-connex hierarchical query has static width 1.*

We give a syntactic classification of hierarchical queries based on their dynamic width.

Definition 4.10. A hierarchical query is δ_i-*hierarchical* for $i \in \mathbb{N}$ if i is the smallest number such that for each bound variable X and atom $R(\mathcal{Y})$ of X, there are i atoms $R_1(\mathcal{Y}_1), \dots, R_i(\mathcal{Y}_i)$ such that all free variables in the atoms of X are included in $\mathcal{Y} \cup \bigcup_{j \in [i]} \mathcal{Y}_j$.

For instance, the query $Q(Y_0, \dots, Y_i) = R_0(X, Y_0), \dots, R_i(X, Y_i)$ is a δ_i-hierarchical query for $i \in \mathbb{N}$. The class of hierarchical queries can be partitioned into subclasses of δ_i-hierarchical queries for $i \in \mathbb{N}$. Then, the δ_0-hierarchical queries are precisely the q-hierarchical queries from prior work [BKS17a].

Proposition 4.11. *A query is q-hierarchical if and only if it is δ_0-hierarchical.*

As depicted in Figure 2 (right), all free-connex hierarchical queries are either δ_0- or δ_1-hierarchical.

Proposition 4.12. *Any free-connex hierarchical query is δ_0- or δ_1-hierarchical.*

The following proposition relates δ_i-hierarchical queries to their dynamic width.

Proposition 4.13. *A hierarchical query is δ_i-hierarchical for $i \in \mathbb{N}$ if and only if it has dynamic width i.*

Proposition 4.13 and Theorem 3.1 imply the following corollary.

Corollary 4.14. *Given a δ_i-hierarchical query with $i \in \mathbb{N}$ and static width w, a database of size N, and $\epsilon \in [0,1]$, the query result can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay after $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ preprocessing time and $\mathcal{O}(N^{i\epsilon})$ amortized time for single-tuple updates.*

5. 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 view trees, where each view tree computes one part of the query result. A view tree is a tree-shaped hierarchy of materialized views with input relations as leaves and upper views defined in terms of their child views. The construction of view trees exploits the structure of the query and the degree of data values in base relations. We construct different sets of view trees for the static and dynamic evaluation of a given hierarchical query.

We next assume that the canonical variable order of the given hierarchical query consists of a single connected component. For several connected components, the preprocessing procedure is executed on each connected component separately.

5.1. View Trees Encoding the Query Result.

Given a hierarchical query $Q(\mathcal{F})$ and a canonical variable order ω for Q, the function BuildVT in Figure 6 constructs a view tree that encodes the query result. The function proceeds recursively on the structure of ω and constructs a view over schema $\mathcal{F}_X$ at each inner node X; the leaves correspond to the atoms in the query. The view is defined over the join of its child views projected onto $\mathcal{F}_X$ (Figure 7). The schema $\mathcal{F}_X$ includes the ancestors of X in ω since they are needed for joins at nodes above X. Each constructed view has a name to help us identify the place and purpose of the view in the view tree.

```
BuildVT(string $prefix$, variable order $\omega$, schema $\mathcal{F}$) : view tree
switch $\omega$:
$R(\mathcal{Y})$          return $R(\mathcal{Y})$

     $X$           let $T_i$ = BuildVT($V, \omega_i, \mathcal{F}$), $\forall i \in [k]$
    /   \          let $viewname = prefix$ + "_" + $X.name$
$\omega_1 \cdots \omega_k$     if $(\mathsf{anc}(X) \cup \{X\}) \subseteq \mathcal{F}$
                      let $\mathcal{F}_X = \mathsf{anc}(X) \cup \{X\}$
                      let $subtrees$ = { AuxView(root of $\omega_i, T_i$) }$_{i\in[k]}$
                      return NewVT($viewname, \mathcal{F}_X, subtrees$)
                   let $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega))$
                   let $subtrees = \{T_i\}_{i\in[k]}$
                   return NewVT($viewname, \mathcal{F}_X, subtrees$)
```

FIGURE 6. Construction of a view tree for a canonical variable order ω of a hierarchical query with free variables $\mathcal{F}$. View names share a given *prefix*.

```
NewVT(string $viewname$, schema $\mathcal{S}$, view trees $T_1, \ldots, T_k$) : view tree
let $V_i(\mathcal{S}_i)$ = root of $T_i$, $\forall i \in [k]$
let $V(\mathcal{S})$ = join of $V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ projected onto $\mathcal{S}$
$V.name := viewname$
return { $T_1$                 ,  $k = 1 \wedge \mathcal{S} = \mathcal{S}_1$
       {    $V(\mathcal{S})$
       {    /   \              ,  otherwise
       { $T_1 \cdots T_k$
```

FIGURE 7. Construction of a view tree with a given root view name, root schema $\mathcal{S}$, and children $T_1, \ldots, T_k$.

If X is free, then it is included in the schema of the view constructed at X (and not included if bound). It is also kept in the schemas of the views on the path to the root until it reaches a view whose schema does not have bound variables. The constructed view tree has the upper levels only with views over the free variables. The hierarchy of such views represents the query result and allows its enumeration with constant delay.

In the dynamic case, at each child Z of X we construct a view with schema $\mathsf{anc}(Z)$ on top of the view created at Z (Figure 8). This auxiliary view aggregates away Z from the latter view. The children of the view created at X then share the same schema $\mathcal{F}_X$. This property enables the efficient maintenance of the view at X since processing a change coming from any child view requires only constant-time lookups into that child's sibling views.

Our preprocessing is particularly efficient for free-connex hierarchical queries in the static case and for their strict subclass of δ_0-hierarchical queries in the dynamic case.

For a canonical variable order of a hierarchical query, the free-connex property fails if there are free variables such that they are below a bound join variable and are not covered

```
AuxView(node $Z$, view tree $T$) : view tree
1 let $V(\mathcal{S}) =$ root of $T$
2 let $viewname = V.name$ + "'"
3 if $mode =$ 'dynamic' $\wedge$ has_sibling($Z$) $\wedge$ anc($Z$) $\subset \mathcal{S}$
4     return NewVT($viewname$, anc($Z$), $\{T\}$)
5 return $T$
```

Figure 8. A tree T constructed at variable Z is extended with a new root view that aggregates away Z.

by one atom. 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.

For δ_0-hierarchical queries, there is no bound variable whose set of atoms strictly contains the atoms of a free variable. Such queries thus admit canonical free-top variable orders where all free variables occur above the bound ones.

For any free-connex hierarchical query, each view created by BuildVT is 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 [BDG07] and dynamic [BKS17a, IUV17] evaluation.

Example 5.1. Consider the free-connex query

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

and its canonical variable order in Figure 9. We construct the view tree bottom-up as follows. At C, we create the view $V_C(A, B)$ that aggregates away the bound variable C but keeps its ancestors A and B to define views up in the tree. Since D is free and has only one child, we skip creating a view at D; see the first case in Line 4 of NewVT from Figure 7. Similarly, no view is created at E. At B, we create the view $V_B(A, D) = V_C(A, B), S(A, B, D)$, which keeps D as it is free and A as the ancestor of B. At A, we create the views $V_A(A) = V_B(A, D), T(A, E)$ in the static case and $V_A(A) = V'_B(A), T'(A)$ in the dynamic case, where $V'_B(A) = V_B(A, D)$ and $T'(A) = T(A, E)$. Each view can be computed in linear time by aggregating away variables and semi-join reduction. The result of Q can be enumerated using $V_A(A)$, $V_B(A, D)$, and $T(A, E)$ with constant delay. □

5.2. **Skew-Aware View Trees.** For free-connex queries, the procedure BuildVT constructs in linear time a data structure that allows for constant-time enumeration delay (Proposition 6.1 and Lemma C.4). For δ_0-hierarchical queries, it also admits constant-time updates (Lemma E.1). We now focus on the bound join variables that violate the free-connex property in the static case or the δ_0-hierarchical property in the dynamic case. For each such violating bound variable X, we use two evaluation strategies.

The first strategy materializes a subset of the query result obtained for the *light* values over the set of variables $\mathsf{anc}(X) \cup \{X\}$ in the variable order. It also aggregates away the bound variables in the subtree rooted at X. Since the light values have a bounded degree, this materialization is inexpensive.

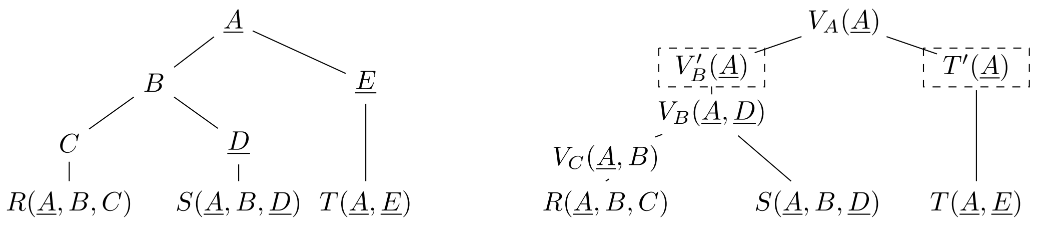


FIGURE 9. Canonical variable order and view tree for $Q(A, D, E) = R(A, B, C), S(A, B, D), T(A, E)$ in Example 5.1. The views V'_B and T' are created in the dynamic case. Free variables are underlined.

The second strategy computes a compact representation of the rest of the query result obtained for those values over $\mathsf{anc}(X) \cup \{X\}$ that are *heavy* (i.e., have 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 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 [DS11] where the delay is given by the number of heavy values of the variables we partitioned on and by the number of strategies.

Heavy and Light Indicators. We consider a bound join variables that violates the free-connex property in the static case or the δ_0-hierarchical property in the dynamic case. We compute heavy and light indicator views consisting of disjoint sets of values for each such variable X. The heavy indicator has the values that exist in all relations and are heavy in at least one relation. The light indicator has the values that exist in all relations and are light in all relations. Indicator views have set semantics. They allow us to rewrite the query into an equivalent union of two queries.

Partitioning the 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 heavy values over $\mathsf{anc}(X) \cup \{X\}$.

Figure 10 shows how to construct a triple of view trees for computing the indicators for $\mathsf{anc}(X) \cup \{X\}$, where X is the root of a variable order ω that is a subtree in the variable order of a hierarchical query (thus $\mathsf{anc}(X)$ may be non-empty). We first construct a view tree that computes the tuples of values for variables $keys = \mathsf{anc}(X) \cup \{X\}$ over the join of the relations from ω. We then build a similar view tree for the light indicator for $keys$ using a modified variable order ω^{keys} of the same structure as ω but with each relation R replaced by the light part of R partitioned on $keys$. Finally, the view tree for the heavy indicator computes the difference of all $keys$-values and those from the light indicator.

View Trees with Indicators. Figure 11 gives the algorithm for constructing the view trees for a variable order ω of a hierarchical query $Q(\mathcal{F})$. The algorithm traverses the variable

```
IndicatorVTs(variable order $\omega$) : triple of view trees
1  let $X$ = root of $\omega$
2  let $keys = \mathsf{anc}(X) \cup \{X\}$
3  let $alltree$ = BuildVT("All", $\omega$, $keys$)
4  let $ltree$ = BuildVT("L", $\omega^{keys}$, $keys$)
5  let $allroot$ = root of $alltree$
6  let $lroot$ = root of $ltree$
7  let $htree$ = NewVT("H_" + $X.name$, $keys$, $\{allroot, \nexists lroot\}$)
8  return ($alltree, ltree, htree$)
```

Figure 10. Construction of the heavy and light indicator view trees for a canonical variable order ω of a hierarchical query. The variable order ω^{keys} has the structure of ω but each atom $R(\mathcal{Y})$ is replaced with the light part $R^{keys}(\mathcal{Y})$ of relation R partitioned on *keys*. The view $\nexists lroot$ maps all tuples contained in *lroot* to 0 and all other tuples to 1.

order ω 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 values are heavy).

The free variables at node X are the ancestors of X and the free variables in the subtree rooted at X (Line 3). If the residual query Q_X at node X (Line 4) is free-connex in the static case or δ_0-hierarchical in the dynamic case, we return a view tree for Q_X (Lines 5-7). If X is free, we recursively compute a set of view trees for each child of X. We may extend the root of each child tree with an auxiliary view in the dynamic mode to support constant-time propagation of updates coming via the siblings of X. 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 8-11). If X is bound, we create two evaluation strategies for the residual query Q_X based on the degree of values of X and its ancestors in the relations of Q_X. We construct the indicator view trees for X and its ancestors (Line 12). The heavy indicator restricts the joins of the child views to only heavy values for the tuple of X and its ancestors (Lines 13-15). We also construct a view tree over the light parts of the relations in ω (Line 16).

The algorithm from Figure 11 uses different criteria for the static and dynamic cases (Lines 5-6) to decide on whether to stop recursively traversing the variable order. Since the class of δ_0-hierarchical queries is a proper subset of the class of free-connex queries, the algorithm may partition input relations on more attributes and create more view trees in the dynamic case than in the static case for the same variable order and free variables.

We next showcase our approach on a non-free-connex query. Section 9 provides additional examples with δ_1-hierarchical queries.

Example 5.2. Figure 12 shows the view trees for the query

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

We start from the root A in the variable order. Since Q is not free-connex (and also not δ_0-hierarchical) and A is bound, we create the view trees for the indicators $H_A(A)$ and $L_A(A)$. Materializing the views in these view trees takes linear time.

```
$\tau$(variable order $\omega$, free variables $\mathcal{F}$) : set of view trees
**switch** $\omega$:

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

      $X$               **let** $keys = \mathsf{anc}(X) \cup \{X\}$
     /   \              **let** $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega))$
  $\omega_1 \cdots \omega_k$          **let** $Q_X(\mathcal{F}_X)$ = join of $atoms(\omega)$
                        **if** ($mode$ = 'static' $\wedge$ $Q_X(\mathcal{F}_X)$ is free-connex) $\vee$
                           ($mode$ = 'dynamic' $\wedge$ $Q_X(\mathcal{F}_X)$ is $\delta_0$-hierarchical)
                           **return** { BuildVT("V", $\omega$, $\mathcal{F}_X$) }
                        **if** $X \in \mathcal{F}$
                           **return** { NewVT("V_" + $X.name$, $keys$, $\{\hat{T}_1, ..., \hat{T}_k\}$)
                                      | $T_i \in \tau(\omega_i, \mathcal{F})_{i\in[k]}$,
                                        $\hat{T}_i$ = AuxView(root of $\omega_i, T_i)_{i\in[k]}$ }
                        **let** $(\_, \_, H_X)$ = roots of IndicatorVTs($\omega$)
                        **let** $htrees$ = { NewVT("V_" + $X.name$, $keys$, $\{\exists H_X, \hat{T}_1, ..., \hat{T}_k\}$)
                                          | $T_i \in \tau(\omega_i, \mathcal{F})_{i\in[k]}$,
                                            $\hat{T}_i$ = AuxView(root of $\omega_i, T_i)_{i\in[k]}$ }
                        **let** $ltree$ = BuildVT("V", $\omega^{keys}$, $\mathcal{F}_X$)
                        **return** $htrees \cup \{ ltree \}$
```

FIGURE 11. 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 the evaluation mode. The variable order ω^{keys} has the structure of ω but each atom $R(\mathcal{Y})$ is replaced by the light part $R^{keys}(\mathcal{Y})$ of relation R partitioned on $keys$. The view $\exists H_X$ maps all tuples contained in H_X to 1 and all other tuples to 0.

In the light case for A, we create a view tree with the root $V_A(C, D, E, F)$ and the leaves being the light parts of the input relations partitioned on A (bottom-left). Computing $V_G(A, C)$ and $V_C(A, C, F)$ takes linear time. We compute the view $V_B(A, D, E)$ in time $\mathcal{O}(N^{1+\epsilon})$: For each (a, b, d) tuple in R^A, we iterate over at most N^ϵ (a, b, e) values in S^A. The view $V_B(A, D, E)$ contains at most $N^{1+\epsilon}$ tuples. Similarly, we compute $V_A(C, D, E, F)$ in time $\mathcal{O}(N^{1+2\epsilon})$: For each (a, d, e) tuple in V_B, we iterate over at most N^ϵ (a, c, f) values in V_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 δ_0-hierarchical, thus we compute its view tree with the root $V_C(A, C)$ in the static case and the root $V'_C(A)$ in the dynamic case (view trees in the second row) 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 δ_0-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 a

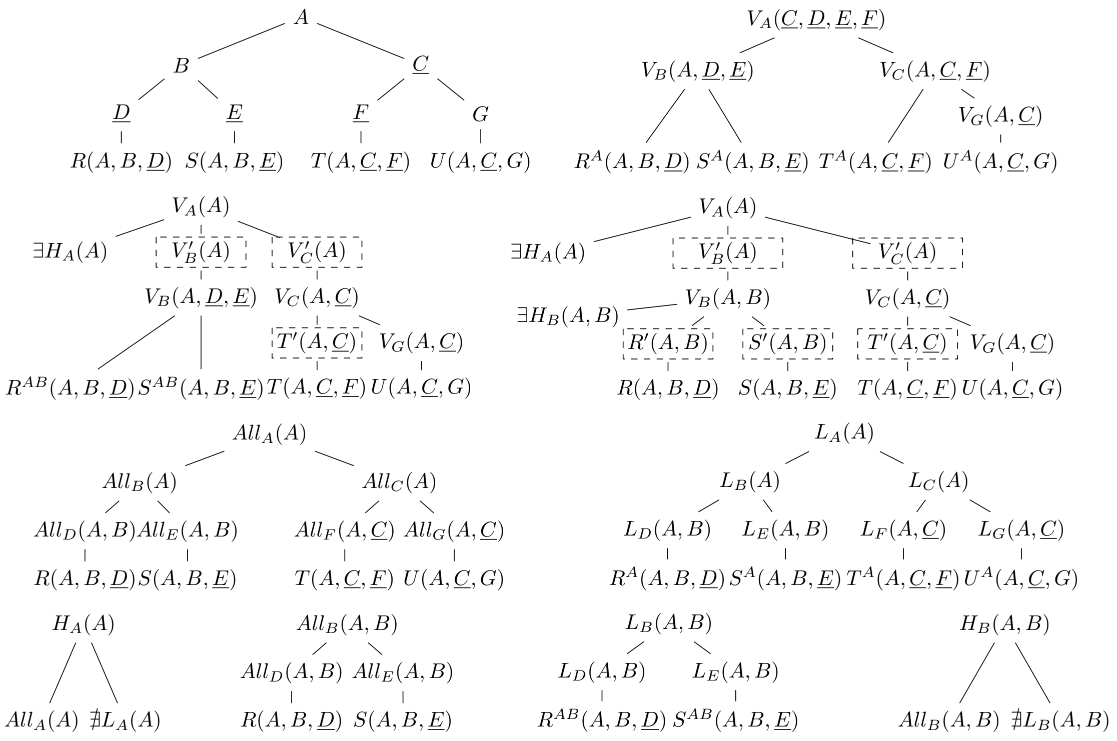


Figure 12. Canonical variable order for the query $Q(C,D,E,F) = R(A,B,D), S(A,B,E), T(A,C,F), U(A,C,G)$ (top left). The three view trees constructed for the query (top right and second row). The indicator view trees for computing H_A and H_B (third and fourth row). The views with a dashed box are only needed for dynamic query evaluation.

view tree with the root $V_B(A,D,E) = R^{AB}(A,B,D), S^{AB}(A,B,E)$ (second row left) and compute $V_B(A,D,E)$ in time $\mathcal{O}(N^{1+\epsilon})$ by iterating over R^{AB} and, for each (a,b,d), iterating over at most N^ϵ E-values in S^{AB}. In the heavy case for (A,B), we process the subtrees of B considering B as free variable. The two subqueries, $Q_D(A,B,D) = R(A,B,D)$ and $Q_E(A,B,E) = S(A,B,E)$, are δ_0-hierarchical.

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 $\mathcal{O}(N^{1+2\epsilon})$. □

Given a hierarchical query, our algorithm effectively rewrites it into an equivalent union of queries, with one query defined by the join of the leaves of a view tree.

Proposition 5.3. *Let $\{T_1,\ldots,T_k\} = \tau(\omega,\mathcal{F})$ be the set of view trees constructed by the algorithm in Figure 11 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 T_i, $\forall i \in [k]$. Then, $Q(\mathcal{F}) \equiv \bigcup_{i\in[k]} Q^{(i)}(\mathcal{F})$.*

The preprocessing time of our approach is given by the time to materialize the views in the view trees.

```
T.open(tuple ctx)
let $V(\mathcal{S})$ = root of $T$
$V.open(ctx)$
$T.buckets := \emptyset$
let $\{T_1, \ldots, T_k\}$ = children of $T$
if $\exists i \in [k]$ such that $T_i = \exists H$                // heavy indicator as child
   $\exists H.open(ctx)$
   while $(h := \exists H.next()) \neq$ EOF
      $T'$ := shallow copy of $T$ without $\exists H$
      $T'.open(h)$
      $T.buckets := T.buckets \cup \{T'\}$
else if $\mathcal{S} \subset$ free variables in $T$                // need to recurse
   $T.ctx := V.next()$                      // current context for entire tree $T$
   foreach $i \in [k]$ do $T_i.open(T.ctx)$
$T.next()$                   // initializes $T.tuple$ to first tuple to be returned
```

Figure 13. Open the view iterators in a view tree.

Proposition 5.4. *Given a hierarchical query $Q(\mathcal{F})$ with static 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 $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time.*

6. Enumeration

For any hierarchical query, Section 5 constructs a set of view trees that together represent the query result. We now show how to enumerate the distinct tuples in the query result with their multiplicity using the *open/next/close* iterator model for such view trees.

Each view in a view tree follows the iterator model. The function $V.open(ctx)$ initializes the iterator on view V using the tuple *ctx* as context, setting the range of the iterator to those tuples that are consistent with *ctx* in V, that is, *ctx* is part of each such tuple in V. The function $V.next()$ returns a tuple consistent with *ctx* in V; or it returns **EOF** if the tuples in the range of the iterator are exhausted. The tuples returned by $V.next()$ are distinct. Both functions operate in constant time, as per our computational model.

Given a subtree T of a view tree and the current tuple *ctx* in its parent view, the call $T.open(ctx)$ described in Figure 13 sets the range of the iterator of T to those tuples in its root view that agree with *ctx* and positions the iterator at the first tuple in this range. The *open* call is recursively propagated down the view tree with an increasingly more specific context tuple. A $T.close()$ call resets the iterators of tree T. Each subtree T has an attribute $T.tuple$ storing the next tuple to be reported. The *open* method ends with a call to $T.next()$ to set $T.tuple$ to the first tuple to be reported.

There are two cases that need special attention. If the schema of a view V includes all free variables in the subtree rooted at V, then there is no need to open the views in this subtree since V already has the tuples over these free variables; e.g., this is the case of the

```
$T.next(\,)$ : tuple

let $V(\mathcal{S})$ = root of $T$
if ($T$ has no children) $\vee$ ($\mathcal{S}$ = free variables in $T$)          // no need to recurse
    $t := T.tuple$; $T.tuple := V.next()$; return $t$
if $T.buckets \neq \emptyset$
    $t := T.tuple$; $T.tuple := \textsc{Union}(T.buckets)$; return $t$
let $\{T_1, \ldots, T_k\}$ = children of $T$
while ($T.ctx \neq$ EOF) do
    if ($n := \textsc{Product}(T_1, \ldots, T_k, T.ctx)) \neq$ EOF    // next tuple in Cartesian product
        $t := T.tuple$; $T.tuple := n$; return $t$
    $T.ctx := V.next()$                    // Cartesian product exhausted, next tree context
    foreach $i \in [k]$ do $T_i.close()$; $T_i.open(T.ctx)$
$t := T.tuple$; $T.tuple :=$ EOF; return $t$
```

Figure 14. Find the next tuple in a view tree.

view $V_A(C, D, E, F)$ in Figure 12. The views with heavy indicators, e.g., the views $V_A(A)$ and $V_B(A, B)$ in Figure 12, also require special treatment. If V has as child a heavy indicator $\exists H$, the tree T rooted at V represents possibly overlapping relations 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 current tuple *ctx* at the parent view of V and keep this iterator in a *shallow* copy of T. Creating a shallow copy of T means creating a tree of iterators of the same structure as T but without copying the content of views under T.

After the first *open* call for a view tree T, we can enumerate the distinct tuples from T with their multiplicity by calling $T.next()$, see Figure 14. The *next* call propagates recursively down T and observes the same cases as the *open* call. If a view V in T already covers all free variables in T, then it suffices to enumerate from V. If T has as child a heavy indicator, we return the next tuple and its multiplicity from the union of all its groundings using the Union algorithm (Section 6.1). Otherwise, we synthesize the returning tuple out of the tuples at the iterators of T's children. Given the current context at T's view, we return the next tuple and its multiplicity from the Cartesian product of the tuples produced by T's children using the Product algorithm (Section 6.2).

For a view tree with no heavy indicators, calling *open* and *next* on the view tree translates to calling *open* and *next* on its views, where each such call on a view takes constant time and the number of such calls is independent of the size of the database. Thus, calling *open* and *next* on a view tree with no heavy indicators takes constant time.

In the presence of heavy indicators, the time to initialize a view tree and produce the next tuple is dominated by the number of shallow view trees created in the grounding step. The delay of the Union algorithm is the sum of the delays of the grounded view trees. Their number is determined by the size of the heavy indicators in the view tree, which is $\mathcal{O}(N^{1-\epsilon})$. Thus, calling *open* and *next* on a view tree with heavy indicators takes $\mathcal{O}(N^{1-\epsilon})$ time.

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

```
UNION(view trees $T_1, \ldots, T_n$) : tuple
if $(n = 1)$ return $T_n.next()$
if $((t, m) := \text{UNION}(T_1, \ldots, T_{n-1})) \neq$ EOF
    if $T_n.lookup(t) \neq 0$
        $(t_n, m_n) := T_n.next()$
        return $(t_n, m_n + \sum_{i \in [n-1]} T_i.lookup(t_n))$
    return $(t, m)$
if $((t_n, m_n) := T_n.next()) \neq$ EOF
    return $(t_n, m_n + \sum_{i \in [n-1]} T_i.lookup(t_n))$
return EOF
```

FIGURE 15. Find the next tuple in a union of view trees.

components, i.e., it is a Cartesian product of hierarchical queries, we use the PRODUCT algorithm with an empty context.

The multiplicity for a tuple returned by the UNION algorithm is the sum of the multiplicities of its occurrences across the buckets, while for a tuple returned by the PRODUCT algorithm it is the multiplication of the multiplicities of the constituent tuples. Since all tuples in the database have positive multiplicities, the derived multiplicities are always strictly positive and therefore the returned tuple is part of the result.

We next explain the UNION and PRODUCT algorithms used by $T.next()$ in Figure 14.

6.1. The Union Algorithm.

The UNION algorithm is given in Figure 15. It is an adaptation of prior work [DS11]. It takes as input n view trees that represent possibly overlapping sets of tuples over the same relation and returns a tuple and its multiplicity in the union of these sets, where the tuple 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 return one tuple together with its multiplicity or **EOF**. We check whether the next tuple t_1 in T_1 is also present in T_2. If so, we return the next tuple in T_2 and its total multiplicity from T_1 and T_2; otherwise, we return t_1 and its multiplicity in T_1. If T_1 is exhausted, we return the next tuple in T_2 and its total multiplicity from T_1 and T_2, or **EOF** if 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.

The delay of this algorithm is given by the delay of iterating over each view, the cost of lookups into the views, and the cost of computing output multiplicities. The lookup costs are constant when using a hierarchy of materialized views for representing the query result [OZ15]. Given n views, computing an output multiplicity takes $\mathcal{O}(n)$ time. The overall delay is 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

```
Product(view trees $T_1,\dots,T_k$, tuple $ctx$) : tuple
1  while ($T_1.tuple \neq$ EOF) do
     ...
2      while ($T_{k-1}.tuple \neq$ EOF) do
3        while ($T_k.tuple \neq$ EOF) do
4          let $(t_i, m_i) = T_i.tuple, \forall i \in [k]$
5          let $(t_{ctx}, \_) = ctx$, where $t_{ctx}$ is over schema $\mathcal{S}$
6          $t := t_{ctx} \circ \bigcirc_{i\in[k]} \pi_{\text{free variables in } T_i - \mathcal{S}}\, t_i$
7          $m := \prod_{i\in[k]} m_i$
8          $T_k.next()$
9          return $(t, m)$
10       $T_k.close()$; $T_k.open(ctx)$; $T_{k-1}.next()$
     ...
11   $T_2.close()$; $T_2.open(ctx)$; $T_1.next()$
12 return EOF
```

Figure 16. Find the next tuple in a product of view trees. In case $k = 1$, the innermost loop is executed.

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.

6.2. The Product Algorithm.

The Product algorithm is given in Figure 16. It takes as input a set of view trees $T_1, \dots, T_k$ and a context, which is the current tuple in the parent view, and outputs the next tuple and its multiplicity in the Cartesian product of the tuples returned by the k view trees given the context. By construction, the parent view joins the roots of the k view trees and thus yields only contexts for which each of the view trees produces a non-empty result.

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 variables that are in common with the context tuple, retaining only the free variables in T_k. We concatenate this projection with the context tuple. The concatenation operator is $\circ$. Before we return this concatenated tuple and its multiplicity, we advance the iterator to the next tuple-multiplicity pair 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 tuple-multiplicity pairs for $T_1, \dots, T_{k-1}$ and iterate over T_k. Whenever T_k reaches **EOF**, we reset it and advance the iterator for T_{k-1}. We concatenate the context tuple and the current tuples of all iterators, projected onto the variables that are not in the schema of the context tuple (since those fields are already in the context). We multiply the current multiplicities of all iterators and advance the iterator for T_k before returning the concatenated tuple and its multiplicity.

The delay for a PRODUCT call is given by the sum of the delays of the k input view trees. In the worst case, the algorithms makes $k-1$ *open* calls and k *next* calls before returning the next tuple. 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 as 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 $\mathcal{O}(N^{1-\epsilon})$.

We next state the complexity of enumeration in our approach.

Proposition 6.1. *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 constructed by $\tau(\omega, \mathcal{F})$ for a canonical variable order ω for Q.*

7. UPDATES

We present our strategy for maintaining the views in the set of view trees $\tau(\omega, \mathcal{F})$ constructed for a canonical variable order ω of a hierarchical query $Q(\mathcal{F})$ under updates to input relations. We specify here the procedure for processing a single-tuple update to any input relation. Processing a sequence of such updates builds upon this procedure and occasional rebalancing steps (Section 7.2).

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; i.e., $|\delta R| = 1$. 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 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. In Figure 12, the view tree created for the case of heavy (A, B)-values (second row right) has five such additional views, marked with dashed boxes. These views enable an update to any leaf view to be propagated to the root view in constant time. For instance, the views R' and S' eliminate the need to iterate over the D-values in relation R for updates to relation S and $\exists H_B$ and respectively over the E-values in S for updates to R and $\exists H_B$. Figure 8 gives the rule for creating such views: If node Z has a sibling in the variable order, then we create an auxiliary view that aggregates away Z to avoid iterating over the Z-values for updates coming via the (auxiliary) views constructed for the siblings of Z.

7.1. 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 11.[2] 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 [CY12] (see Example 9.1). To simplify the reasoning about the maintenance task, we assume that each view tree has a

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

```
Apply(view tree $T$, update $\delta R$) : delta view
switch $T$:
$K(\mathcal{X})$
    if $K = R$
        $R(\mathcal{X}) := R(\mathcal{X}) + \delta R(\mathcal{X})$
        return $\delta R$
    return $\emptyset$

      $V(\mathcal{X})$
       /    \
    $T_1 \cdots T_k$
    let $V_i(\mathcal{X}_i)$ = root of $T_i$, for $i \in [k]$
    if $\exists\, j \in [k]$ such that $R \in T_j$
        $\delta V_j := \text{Apply}(T_j, \delta R)$
        let $\delta V(\mathcal{X})$ = join of $V_1(\mathcal{X}_1), \ldots, \delta V_j(\mathcal{X}_j), \ldots, V_k(\mathcal{X}_k)$ proj. onto $\mathcal{X}$
        $V(\mathcal{X}) := V(\mathcal{X}) + \delta V(\mathcal{X})$
        return $\delta V$
    return $\emptyset$
```

Figure 17. Updating views in a view tree T for a single-tuple update δR to relation R.

```
UpdateIndTree(indicator tree $T_{Ind}$, update $\delta R$) : indicator change
let $I(\mathcal{S})$ = root of $T_{Ind}$
let $key = \mathbf{x}[\mathcal{S}]$, where $\delta R = \{\mathbf{x} \to m\}$
let $\#before = I(key)$
Apply($T_{Ind}, \delta R$)
if $(\#before = 0) \wedge (I(key) > 0)$ return $\{key \to 1\}$
if $(\#before > 0) \wedge (I(key) = 0)$ return $\{key \to -1\}$
return $\emptyset$
```

Figure 18. Updating an indicator view tree T_{Ind} for a single-tuple update δR to relation R.

copy of its base relations. We use Apply$(T, \delta R)$ from Figure 17 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 heavy indicator $H_B(A, B)$ constructed over the view $All_B(A, B)$ and the light indicator $\nexists L_B(A, B)$ in Figure 12. An insert $\delta R = \{(a, b, d) \to 1\}$ into R may change the multiplicity $H_B(a, b)$ from 0 to non-zero, thus changing $\exists H_B(A, B)$ and its dependent views: $V_B(A, B)$, $V'_B(A)$, and $V_A(A)$. But if the multiplicity $H_B(a, b)$ stays 0 or non-zero after applying δR, then $\exists H_B$ also stays unchanged.

Figure 18 shows the function UpdateIndTree that applies an update δR to an indicator tree T_{Ind} with a root view $I(\mathcal{S})$. 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 $\mathcal{S}$ of I appear in each input relation from T_{Ind}, and δR fixes their values to constants; thus, $|\delta(\exists I)| \leq 1$.

```
UpdateTrees(view trees $\mathcal{T}$, indicator triples $\mathcal{T}_{Ind}$, update $\delta R$)
foreach $T \in \mathcal{T}$ do Apply($T, \delta R$)
foreach $(T_{All}, T_L, T_H) \in \mathcal{T}_{Ind}$ such that $R \in T_{All}$ do
   let $All(\mathcal{S})$ = root of $T_{All}$, $L(\mathcal{S})$ = root of $T_L$, $H(\mathcal{S})$ = root of $T_H$
   let $key = \mathbf{x}[\mathcal{S}]$, where $\delta R = \{\mathbf{x} \to m\}$
   let $\#before = All(key)$
   Apply($T_{All}, \delta R$)
   let $\#change = All(key) - \#before$
   let $\delta(\exists H)$ = UpdateIndTree($T_H, \delta All = \{key \to \#change\}$)
   foreach $T \in \mathcal{T}$ do Apply($T, \delta(\exists H)$)
   if $(key \notin \pi_{\mathcal{S}} R) \vee (key \in \pi_{\mathcal{S}} R^{\mathcal{S}})$
      foreach $T \in \mathcal{T}$ do Apply($T, \delta R^{\mathcal{S}} = \delta R$)
      let $\delta(\exists L)$ = UpdateIndTree($T_L, \delta R^{\mathcal{S}} = \delta R$)
      let $\delta(\exists H)$ = UpdateIndTree($T_H, \delta(\nexists L) = -\delta(\exists L)$)
      foreach $T \in \mathcal{T}$ do Apply($T, \delta(\exists H)$)
```

FIGURE 19. Updating a set $\mathcal{T}$ of view trees and a set $\mathcal{T}_{Ind}$ of triples of indicator trees for a single-tuple update δR to relation R.

Figure 19 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 1). Then, we consider the triples (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 $H(\mathcal{S}) = All(\mathcal{S}), \nexists L(\mathcal{S})$ for changes in both All and $\nexists L$. We apply δR to T_{All} (Line 6) and subsequently δAll to T_H (Line 8). 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 9). If the update δR belongs to the light part $R^{\mathcal{S}}$ (Line 10), we apply $\delta R^{\mathcal{S}}$ to the view trees from $\mathcal{T}$ and to the light indicator tree T_L (Lines 11-12). We then propagate the opposite change $\delta(\nexists L)$ in the support of the root L of T_L, if any, to T_H and further to the view trees from $\mathcal{T}$ (Lines 13-14).

Example 7.1. We analyze the time needed to maintain the views from Figure 12 under a single-tuple update to any input relation. For the view tree constructed for the case of heavy (A, B)-values (second row right), propagating an update from any relation to the root view takes constant time. For instance, an update δR to R changes the view $R'(A, B)$ with $\delta R'(a, b) = \delta R(a, b, d)$; the view $V_B(A, B)$ with $\delta V_B(a, b) = \exists H_B(a, b), \delta R'(a, b), S'(a, b)$; changes to the views $V'_B(A)$ and $V_A(A)$ are similar. The auxiliary views $S'(A, B)$ and $V'_C(A)$ enable the constant-time updates in this case by aggregating away the E-values in $S(A, B, E)$ and the C-values in $V_C(A, B)$.

Consider now the view tree defined over the light parts of input relations (bottom-left). The update $\delta R = \{(a, b, d) \to m\}$ affects the light part R^A of R when $(a, b, d) \notin R$ or $a \in \pi_A R^A$. If so, computing $\delta V_B(a, d, E) = \delta R^A(a, b, d), S^A(a, b, E)$ takes $\mathcal{O}(N^{\epsilon})$ time since a is light in S^A. The size of δV_B is also $\mathcal{O}(N^{\epsilon})$. Computing δV_A at the root requires pairing each E-value from δV_B with the (C, F)-values in V_C for the given a. Since a is light in T^A, the number of such (C, F)-values in T^A is $\mathcal{O}(N^{\epsilon})$. Thus, computing δV_A takes $\mathcal{O}(N^{2\epsilon})$ time.

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

Figure 20. Recomputing the light parts of base relations and affected views.

A similar analysis shows that updates to S^A and T^A also take $\mathcal{O}(N^{2\epsilon})$ time, while updates to U^A take $\mathcal{O}(N^{3\epsilon})$ time.

For the view tree constructed for the case of heavy A-values (bottom-middle), updates to R^{AB} and S^{AB} take $\mathcal{O}(N^{\epsilon})$ time, while updates to T and U take constant time. The indicator view trees (top and middle row) encode the results of δ_0-hierarchical queries, thus maintaining their views takes constant time per update.

The indicator views $\exists H_A(A)$ and $\exists H_B(A, B)$ may change under updates to any relation and respectively under updates to R and S. For instance, the update δR can trigger a new single-tuple change in $\exists H_A$ when the multiplicity $H_A(a)$ increases from 0 to non-zero or vice versa. Applying this change $\delta(\exists H_A)$ to the view trees containing $\exists H_A$ takes constant time; the same holds for propagating a change $\delta(\exists H_B)$ to the view trees containing $\exists H_B$.

In conclusion, maintaining the views from Figure 12 under a single-tuple update to any relation takes $\mathcal{O}(N^{3\epsilon})$ overall time. □

We next state the complexity of updates in our approach.

Proposition 7.2. *Given a hierarchical query $Q(\mathcal{F})$ with dynamic 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.*

7.2. 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 M for which the invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ always holds. If the database size falls below $\lfloor \frac{1}{4}M \rfloor$ or reaches M, we perform *major rebalancing*, where we halve or respectively double M, followed by strictly repartitioning the light parts of input relations with the new threshold M^{ϵ} and recomputing the views. Figure 20 shows the major rebalancing procedure.

Proposition 7.3. *Given a hierarchical query $Q(\mathcal{F})$ with static 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 $\Omega(M)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}M$ (after doubling), or $N = \frac{1}{2}M - \frac{1}{2}$ or $N = \frac{1}{2}M - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ and trigger another major rebalancing, the

```
MinorRebalancing(trees $\mathcal{T}$, tree $T_L$, tree $T_H$, source $R$, $key$, $insert$)
1 let $L(\mathcal{F})$ = root of $T_L$, $H(\mathcal{F})$ = root of $T_H$
2 foreach $\mathbf{x} \in \sigma_{\mathcal{F}=key} R$ do
3     let $cnt$ = if ($insert$) $R(\mathbf{x})$ else $-R(\mathbf{x})$
4     foreach $T \in \mathcal{T}$ do Apply($T, \delta R^{\mathcal{F}} = \{\mathbf{x} \to cnt\}$)
5     let $\delta(\exists L)$ = UpdateIndTree($T_L, \delta R^{\mathcal{F}} = \{\mathbf{x} \to cnt\}$)
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 21. Deleting heavy tuples from or inserting light tuples into the light part of relation R.

number of required updates is at least $\frac{1}{4}M$. In the extended technical report, we prove the amortized $\mathcal{O}(N^{(\mathsf{w}-1)\epsilon})$ time of major rebalancing [KNOZ19]. By Proposition 4.8, we have $\delta = \mathsf{w}$ or $\delta = \mathsf{w} - 1$; hence, the amortized major rebalancing time is $\mathcal{O}(M^{\delta\epsilon})$.

Minor Rebalancing. After an update $\delta R = \{\mathbf{x} \to m\}$ to relation R, we check the light part and heavy part conditions of each partition of R. Consider the light part $R^{\mathcal{S}}$ of R partitioned on a schema $\mathcal{S}$. If the number of tuples in $R^{\mathcal{S}}$ that agree with $\mathbf{x}$ on $\mathcal{S}$ exceeds $\frac{3}{2}M^{\epsilon}$, then we delete those tuples from $R^{\mathcal{S}}$. If the number of tuples that agree with $\mathbf{x}$ on $\mathcal{S}$ in $R^{\mathcal{S}}$ is zero and in R is below $\frac{1}{2}M^{\epsilon}$, then we insert those tuples into $R^{\mathcal{S}}$. Figure 21 shows this *minor rebalancing* procedure.

Proposition 7.4. *Given a hierarchical query $Q(\mathcal{F})$ with dynamic 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 $\Omega(M^{\epsilon})$ updates. This lower bound on the number of updates is due to the gap between the two thresholds in the heavy and light part conditions. The extended technical report proves the amortized $\mathcal{O}(N^{\delta\epsilon})$ time of minor rebalancing [KNOZ19].

Figure 22 gives the trigger procedure OnUpdate that maintains a set of view trees $\mathcal{T}$ and a set of indicator trees $\mathcal{T}_{Ind}$ 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 19. If this update leads to a violation of the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$, 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.

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

Proposition 7.5. *Given a hierarchical query $Q(\mathcal{F})$ with dynamic 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 sequence of single-tuple updates takes $\mathcal{O}(N^{\delta\epsilon})$ amortized time per single-tuple update.*

```
OnUpdate(view trees $\mathcal{T}$, indicator triples $\mathcal{T}_{Ind}$, update $\delta R$)
UpdateTrees($\mathcal{T}, \mathcal{T}_{Ind}, \delta R$)
if ($N = M$)
    $M = 2M$
    MajorRebalancing($\mathcal{T}, \mathcal{T}_{Ind}, M^{\epsilon}$)
else if ($N < \lfloor \frac{1}{4} M \rfloor$)
    $M = \lfloor \frac{1}{2} M \rfloor - 1$
    MajorRebalancing($\mathcal{T}, \mathcal{T}_{Ind}, M^{\epsilon}$)
else
    foreach ($T_{All}, T_L, T_H$) $\in \mathcal{T}_{Ind}$ such that $R \in T_{All}$ do
        let $R^{\mathcal{F}} \in T_L$ be light part of $R$ partitioned on $\mathcal{F}$
        let $key = \mathbf{x}[\mathcal{F}]$, where $\delta R = \{\mathbf{x} \to m\}$
        if ( $|\sigma_{\mathcal{F}=key} R^{\mathcal{F}}| = 0 \wedge |\sigma_{\mathcal{F}=key} R| < \frac{1}{2} M^{\epsilon}$)
            MinorRebalancing($\mathcal{T}, T_L, T_H, R, key$, true)
        else if ( $|\sigma_{\mathcal{F}=key} R^{\mathcal{F}}| \geq \frac{3}{2} M^{\epsilon}$ )
            MinorRebalancing($\mathcal{T}, T_L, T_H, R, key$, false)
```

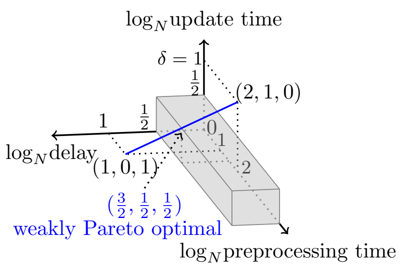
Figure 22. Updating a set of view trees $\mathcal{T}$ and a set of triplets of indicator view trees $\mathcal{T}_{Ind}$ under a sequence of single-tuple updates to base relations.

The proof of Proposition 7.5 is based on prior work (Section 4.1 in [KNN+19a]). The adapted proof is given in the technical report (Section F.4 in [KNOZ19]).

8. Matching Lower Bound for δ_1-Hierarchical Queries

Corollary 4.14 says that, given a database of size N and $\epsilon \in [0, 1]$, any δ_i-hierarchical query with $i \in \mathbb{N}$ can be evaluated with $\mathcal{O}(N^{i\epsilon})$ amortized update time and $\mathcal{O}(N^{1-\epsilon})$ enumeration delay. For δ_1-hierarchical queries, this upper bound is matched by a lower bound conditioned on the Online Matrix-Vector Multiplication Conjecture [HKNS15]. The following proposition extends the lower bound result from prior work [BKS18] to amortized update time. The adapted proof can be found in the technical report (Proposition 10 in [KNOZ19]).

Proposition 8.1. *Given a δ_1-hierarchical query without repeating relation symbols, $\gamma > 0$, and a database of size N, there is no algorithm that maintains the query with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, unless the Online Matrix-Vector Multiplication conjecture fails.*



The blue line connecting the points $(1, 0, 1)$ and $(2, 1, 0)$ in the left figure visualizes the trade-offs of our approach for δ_1-hierarchical queries. The gray cuboid is infinite in the dimension of preprocessing time. Each point strictly included in the gray cuboid corresponds to a combination of some preprocessing time and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time and delay for $\gamma > 0$. Following Proposition 8.1, this is not attainable, unless the Online Matrix-Vector Multiplication conjecture fails. Each point on the surface of the cuboid corresponds to Pareto worst-case optimality in the update-delay trade-off space. For $\epsilon = \frac{1}{2}$, our approach needs $\mathcal{O}(N^{\frac{1}{2}})$ amortized update time and delay, which is weakly Pareto worst-case optimal: there can be no tighter upper bounds for *both* the update time and delay. Since $\mathsf{w} \in \{1, 2\}$ for δ_1-hierarchical queries, the preprocessing time is $O(N^{\frac{3}{2}})$.

9. Examples Showcasing Our Approach

We exemplify our approach for the static and dynamic evaluation of two δ_1-hierarchical queries. We start with the query from Example 2.2.

Example 9.1. Consider the δ_1-hierarchical and non-free-connex query $Q(A, C) = R(A, B), S(B, C)$ from Example 2.2 whose relations have size at most N. We partition R and S on B: A B-value b is *light* in R if $|\{a \mid (a, b) \in R\}| \le N^{\epsilon}$ 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 either light or heavy in each of R and S. We can reduce them to two cases: either B is light in both relations, or B is heavy in at least one of them. We keep the light/heavy information in two indicator views: $L_B(B) = R^B(A, B), S^B(B, C)$, where R^B and S^B are the light parts of R and respectively S; and $H_B(B) = All_B(B), \nexists L_B(B)$, where $All_B(B) = R(A, B), S(B, C)$. The $\exists$ operator before indicators denotes their use with set semantics, i.e., the tuple multiplicities are 0 or 1. The $\nexists$ operator flips the multiplicity.

Figure 23 gives the evaluation and maintenance strategies for our query. 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.

If B is light, we compute the view $V_B(A, C)$ in time $\mathcal{O}(N^{1+\epsilon})$: We iterate over S^B and for each of its tuples (b, c), we fetch the A-values in R^B paired with b in R. The iteration over S^B 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's result.

If 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 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 for one B-value after those for 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 [DS11]. 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 enumerated 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. For each such tuple, we sum up the positive multiplicities of its occurrences in the buckets. This yields an overall $\mathcal{O}(N^{1-\epsilon})$ delay for the enumeration of the distinct tuples in the result of Q.

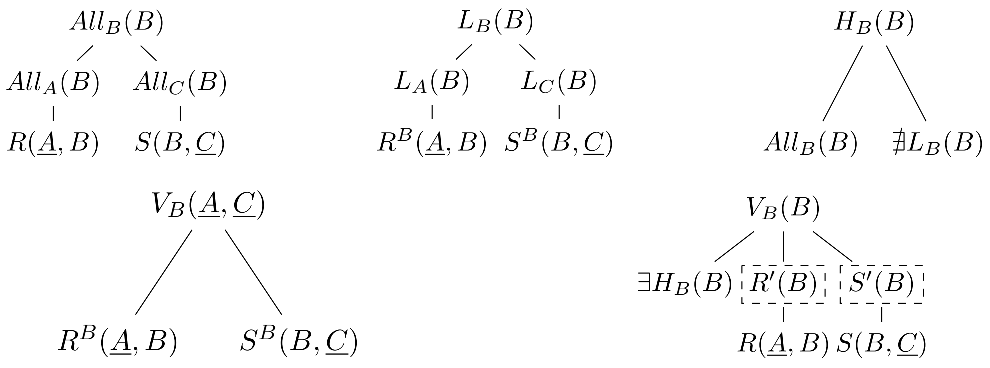


Figure 23. The view trees for $Q(A,C) = R(A,B), S(B,C)$ in Example 9.1. The dashed boxes enclose views that are only needed in the dynamic case.

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 in addition to the degree information of the B-values in the two relations. The multiplicity of a result tuple is the sum of the multiplicities of its duplicates across the $\mathcal{O}(N^{1-\epsilon})$ buckets. We also need two views to support efficient updates to R and S; these are marked with the dashed boxes in Figure 23. For simplicity, we assume that each view tree maintains copies of its base relations.

Consider a single-tuple update $\delta R = \{(a,b) \to m\}$ to relation R. We maintain each view affected by δR using the hierarchy of materialized views from Figure 23. The changes in those views are expressed using the classical delta rules [CY12]. We update the views $R'(B)$ and $V_B(B)$ in the bottom-right tree with $\delta R'(b) = \delta R(a,b)$ and $\delta V_B(b) = \exists H_B(b), \delta R'(b), S'(b)$ in constant time; the same holds for updating the views $All_A(B)$, $All_B(B)$, and $H_B(B)$.

The update δR affects the light part R^B of R if the B-value b already exists among the B-values in R^B or does not exist in R. For such change δR^B, we update $V_B(A,C)$ with $\delta V_B(a,C) = \delta R^B(a,b), S^B(b,C)$ in time $\mathcal{O}(N^{\epsilon})$ since b is light in S^B; updating $L_B(B)$ and $H_B(B)$ takes constant time.

The update δR may trigger a new single-tuple change in $\exists H_B$, affecting $V_B(B)$. The change $\delta(\exists H_B)$ is non-empty only when the multiplicity $H_B(b)$ changes from 0 to non-zero or vice versa. For such change $\delta(\exists H_B)$, we update $V_B(B)$ via constant-time lookups in $R'(b)$ and $S'(b)$.

The update δR may change the degree of b in R from light to heavy or vice versa. In such cases, we need to rebalance the partitioning 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 7). □

Next, we demonstrate our approach for the δ_1-hierarchical query from Example 3.2.

Example 9.2. Consider the δ_1-hierarchical free-connex query $Q(A) = R(A,B), S(B)$ from Example 3.2 whose relations have size at most N. Figure 24 shows the single view tree (bottom-left) that our approach constructs in the static case, and the other five view trees needed in the dynamic case. In the static case, since Q is free-connex, its result can be computed in $\mathcal{O}(N)$ time and then its tuples can be enumerated with $\mathcal{O}(1)$ delay. Our approach does not partition the relations in the static case. We compute the view $V_B(A)$ in

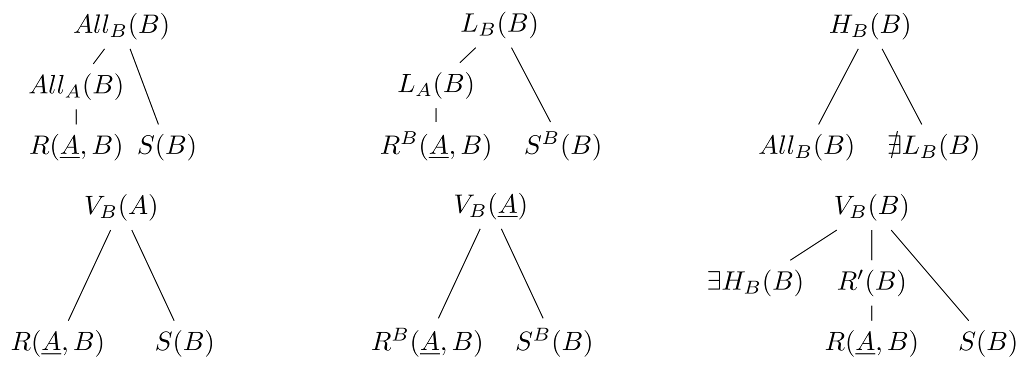


FIGURE 24. The view trees for $Q(A) = R(A, B), S(B)$ in Example 9.2. The bottom-left view tree is the only one needed in the static case, all others are needed in the dynamic case.

time $\mathcal{O}(N)$ by iterating over the tuples in R and looking up for each tuple (a, b) in R the multiplicity of b in S in $\mathcal{O}(1)$ time. The result can be enumerated from the view $V_B(A)$ with $\mathcal{O}(1)$ delay.

In the dynamic case, we partition relations R and S on the bound join variable B and create the indicators $L_B(B)$ and $H_B(B)$ as in Figure 24. In the light case, we compute the view $V_B(A)$ in $\mathcal{O}(N)$ time: For each (a, b) in the light part R^B of R, we check the multiplicity of b in the light part S^B of S using a constant-time lookup. In the heavy case, we compute the view $V_B(B)$ in $\mathcal{O}(N)$ time using the heavy indicator $\exists H_B$, the input relation S, and the projection $R'(B)$ of R on B.

We can enumerate the tuples in the query result with $\mathcal{O}(N^{1-\epsilon})$ 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 $\mathcal{O}(N^{1-\epsilon})$ 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. To obtain the multiplicity of each output tuple, we sum up the positive multiplicities of the duplicates of the tuple across the $\mathcal{O}(N^{1-\epsilon})$ buckets.

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 $\delta V_B(A) = R^B(A, b), \delta S^B(b)$, which requires $\mathcal{O}(N^\epsilon)$ time since b is light in R^B. We may need to rebalance the partitions, which gives an amortized update time of $\mathcal{O}(N^\epsilon)$. □

Since both queries in Examples 9.1 and 9.2 are δ_1-hierarchical and do not have repeating relation symbols, there is no algorithm that can maintain them under single-tuple updates with $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ delay for $\gamma > 0$ unless the Online Matrix-Vector Multiplication conjecture fails (Proposition 8.1). Our approach meets this lower bound for $\epsilon = \frac{1}{2}$.

10. Conclusion and Future Work

This paper investigates the preprocessing-update-delay trade-off for hierarchical queries and introduces an approach that recovers a number of prior results when restricted to hierarchical queries. There are several lines of future work. Of paramount importance is the generalization of our trade-off from hierarchical to conjunctive queries. The results of this paper can be immediately extended to hierarchical queries with group-by aggregates and order-by clauses. In particular, this extension would capture the prior result on constant-delay enumeration for such queries in the context of factorized databases [OS16]. An open problem is to find lower bounds for δ_i-hierarchical queries for $i > 1$. We conjecture our update/delay upper bounds $\mathcal{O}(N^{i\epsilon})/\mathcal{O}(N^{1-\epsilon})$ are worst-case optimal, as it is the case for $i = 0$ with $\epsilon = 1$ [BKS17a] and $i = 1$ with $\epsilon = \frac{1}{2}$ (Proposition 8.1).

Acknowledgment

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

References

[ABJM17] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A Circuit-Based Approach to Efficient Enumeration. In *ICALP*, pages 111:1–111:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.111.

[ABM18] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on Trees under Relabelings. In *ICDT*, pages 5:1–5:18, 2018. doi:10.4230/LIPIcs.ICDT.2018.5.

[ABMN19] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *ICDT*, pages 22:1–22:19, 2019. doi:10.1145/3422648.3422655.

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. doi:10.1109/FOCS.2008.43.

[AKNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016. doi:10.1145/2902251.2902280.

[Bag06] Guillaume Bagan. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *CSL*, pages 167–181, 2006. doi:10.1007/11874683_11.

[BB12] Johann Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *CSL*, pages 137–151, 2012. doi:10.4230/LIPIcs.CSL.2012.137.

[BB13] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

[BDFG10] Guillaume Bagan, Arnaud Durand, Emmanuel Filiot, and Olivier Gauwin. Efficient Enumeration for Conjunctive Queries over X-underbar Structures. In *CSL*, pages 80–94, 2010. doi:10.1007/978-3-642-15205-4_10.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*, pages 208–222, 2007. doi:10.1007/978-3-540-74915-8_18.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983. doi:10.1145/2402.322389.

[BKS17a] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017. doi:10.1145/3034786.3034789.

[BKS17b] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD Queries Under Updates on Bounded Degree Databases. In *ICDT*, pages 8:1–8:18, 2017. doi:10.1145/3232056.

[BKS18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *ICDT*, pages 8:1–8:19, 2018. doi:10.4230/LIPIcs.ICDT.2018.8.

[CK18] Nofar Carmeli and Markus Kröll. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. In *ICDT*, pages 11:1–11:17, 2018. doi:10.1007/s00224-019-09937-9.

[CY12] Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012. doi:10.1561/1900000020.

[DG07] Arnaud Durand and Etienne Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic*, 8(4):21, 2007. doi:10.1145/1276920.1276923.

[DK18] Shaleen Deep and Paraschos Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018. doi:10.1145/3196959.3196979.

[DS11] Arnaud Durand and Yann Strozecki. Enumeration Complexity of Logical Query Problems with Second-order Variables. In *CSL*, pages 189–202, 2011. doi:10.4230/LIPIcs.CSL.2011.189.

[DSS14] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating Answers to First-order Queries over Databases of Low Degree. In *PODS*, pages 121–131, 2014. doi:10.1145/2594538.2594539.

[FO16] Robert Fink and Dan Olteanu. Dichotomies for Queries with Negation in Probabilistic Databases. *ACM Trans. Datab. Syst.*, 41(1):4:1–4:47, 2016. doi:10.1145/2877203.

[GGL$^+$09] Martin Grohe, Yuri Gurevich, Dirk Leinders, Nicole Schweikardt, Jerzy Tyszkiewicz, and Jan Van den Bussche. Database Query Processing Using Finite Cursor Machines. *Theory Comput. Syst.*, 44(4):533–560, 2009. doi:10.1007/s00224-008-9137-7.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

[HY19] Xiao Hu and Ke Yi. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *PODS*, pages 450–463, 2019. doi:10.1145/3294052.3319698.

[IUV17] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017. doi:10.1145/3035918.3064027.

[IUV$^+$18] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive Queries with Inequalities Under Updates. *PVLDB*, pages 733–745, 2018. doi:10.14778/3192965.3192966.

[K$^+$14] Christoph Koch et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014. doi:10.1007/s00778-013-0348-4.

[KNN$^+$19a] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT*, pages 4:1–4:18, 2019. doi:10.4230/LIPIcs.ICDT.2019.4.

[KNN$^+$19b] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. *CoRR*, 2019. abs/1804.02780.

[KNN$^+$20] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020. doi:10.1145/3396375.

[KNOZ19] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *CoRR*, 2019. abs/1907.01988v2. doi:10.1145/3375395.3387646.

[KNOZ20] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. In *PODS*, pages 375–392, 2020. doi:10.1145/3375395.3387646.

[KNS17] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another? In *PODS*, pages 429–444, 2017. doi:10.1145/3034786.3056105.

[Koc10] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, pages 87–98, 2010. doi:10.1145/1807085.1807100.

[KP98] S. Rao Kosaraju and Mihai Pop. De-amortization of Algorithms. In *COCOON*, pages 4–14, 1998. doi:10.1007/3-540-68535-9_4.

[KS11a] Wojciech Kazana and Luc Segoufin. First-order Query Evaluation on Structures of Bounded Degree. *LMCS*, 7(2), 2011. doi:10.2168/LMCS-7(2:20)2011.

[KS11b] Paraschos Koutris and Dan Suciu. Parallel Evaluation of Conjunctive Queries. In *PODS*, pages 223–234, 2011. doi:10.1145/1989284.1989310.

[KS13a] Wojciech Kazana and Luc Segoufin. Enumeration of First-order Queries on Classes of Structures with Bounded Expansion. In *PODS*, pages 297–308, 2013. doi:10.1145/2463664.2463667.

[KS13b] Wojciech Kazana and Luc Segoufin. Enumeration of Monadic Second-order Queries on Trees. *ACM Trans. Comput. Logic*, 14(4):25:1–25:12, 2013. doi:10.1145/2528928.

[LM14] Katja Losemann and Wim Martens. MSO Queries on Trees: Enumerating Answers under Updates. In *CSL-LICS*, pages 67:1–67:10, 2014. doi:10.1145/2603088.2603137.

[Mar10] Dániel Marx. Approximating Fractional Hypertree Width. *ACM Trans. Alg.*, 6(2):29:1–29:17, 2010. doi:10.1145/1721837.1721845.

[NO18] Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018. doi:10.1145/3183713.3183758.

[NPRR18] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. doi:10.1145/2213556.2213565.

[NS18] Matthias Niewerth and Luc Segoufin. Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates. In *PODS*, pages 179–191, 2018. doi:10.1145/3196959.3196961.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized Databases. *SIGMOD Rec.*, 45(2):5–16, 2016. doi:10.1145/3003665.3003667.

[OZ12] Dan Olteanu and Závodný. Factorised Representations of Query Results: Size Bounds and Readability. In *ICDT*, pages 285–298, 2012. doi:10.1145/2274576.2274607.

[OZ15] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM TODS*, 40(1):2:1–2:44, 2015. doi:10.1145/2656335.

[Seg15] Luc Segoufin. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Rec.*, 44(1):10–17, 2015. doi:10.1145/2783888.2783894.

[SORK11] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. doi:10.1007/978-1-4899-7993-3_275-2.

[SSV18] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO Queries over Nowhere Dense Graphs. In *PODS*, pages 151–163, 2018. doi:10.1145/3196959.3196971.

[SV17] Luc Segoufin and Alexandre Vigny. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *ICDT*, pages 20:1–20:16, 2017. doi:10.4230/LIPIcs.ICDT.2017.20.

[Yan81] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *VLDB*, pages 82–94, 1981. doi:10.5555/1286831.1286840.

Appendix A. Proofs of the Results in Section 2

Theorem 2.1. *Given a hierarchical query with static width* w*, a database of size* N*, and* $\epsilon \in [0,1]$*, the query result can be enumerated with* $\mathcal{O}(N^{1-\epsilon})$ *delay after* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *preprocessing time.*

The theorem follows from Propositions 5.3, 5.4, and 6.1. Let $Q(\mathcal{F})$ be a hierarchical query and ω an arbitrary canonical variable order for Q. Without loss of generality, assume that ω consists of a single tree. The preprocessing stage materializes the views in the view trees $\{T_1, \ldots, T_k\}$ returned by $\tau(\omega, \mathcal{F})$ from Figure 11 in the static mode. By Proposition 5.4, these views can be materialized in $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$. By Proposition 5.3, $Q(\mathcal{F})$ 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 T_i. By Proposition 6.1, the result of $Q(\mathcal{F})$ can be enumerated from these materialized views with delay $\mathcal{O}(N^{1-\epsilon})$.

If the canonical variable order for Q consists of several trees $\omega_1, \ldots, \omega_m$, we construct a set $\mathcal{T}_i$ of view trees for each ω_i, where $i \in [m]$. 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 $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, the tuples in the Cartesian product can be enumerated with the same delay using the Product algorithm (Figure 16), since m is independent of the database size N.

Appendix B. Proofs of the Results in Section 3

Theorem 3.1. *Given a hierarchical query with static width* w *and dynamic width* δ*, a database of size* N*, and* $\epsilon \in [0,1]$*, the query result can be enumerated with* $\mathcal{O}(N^{1-\epsilon})$ *delay after* $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ *preprocessing time and* $\mathcal{O}(N^{\delta\epsilon})$ *amortized update time for single-tuple updates.*

The theorem follows from Propositions 5.3, 5.4, 6.1, 7.2, and 7.5. Let $Q(\mathcal{F})$ be a hierarchical query and ω an arbitrary canonical variable order for Q. Without loss of generality, assume that ω consists of a single tree. The preprocessing stage materializes the views in the set of view trees returned by $\tau(\omega, \mathcal{F})$ from Figure 11 in the dynamic mode. The preprocessing time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ follows from Proposition 5.4, which captures both the static and dynamic modes (see the proof). The equivalence between the constructed view trees and the query follows from Proposition 5.3. The delay $\mathcal{O}(N^{1-\epsilon})$ needed when enumerating the query result from these materialized views follows from Proposition 6.1. The time $\mathcal{O}(N^{\delta\epsilon})$ to maintain these materialized views under a single-tuple update follows from Proposition 7.2. By Proposition 7.5, the amortized maintenance time under a sequence of single-tuple updates is $\mathcal{O}(N^{\delta\epsilon})$.

If the canonical variable order consists of several view trees, the reasoning is analogous to the proof of Theorem 2.1.

Appendix C. Proofs of the Results in Section 5

C.1. Proof of Proposition 5.3.

Proposition 5.3. *Let* $\{T_1, \dots, T_k\} = \tau(\omega, \mathcal{F})$ *be the set of view trees constructed by the algorithm in Figure 11 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* T_i*,* $\forall i \in [k]$*. Then,* $Q(\mathcal{F}) \equiv \bigcup_{i\in[k]} Q^{(i)}(\mathcal{F})$*.*

We use two observations. (1) The procedure BuildVT constructs a view tree whose leaf atoms are exactly the same as the leaf atoms of the input variable order. (2) Each of the procedures NewVT and AuxView constructs a view tree whose set of leaf atoms is the union of the sets of leaf atoms of the input trees. For a variable order or view tree T and schema a set $\mathcal{S}$ of variables occurring in T, we define $Q_T(\mathcal{S}) = \Join_{R(\mathcal{X})\in atoms(T)} R(\mathcal{X})$.

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

$$Q_{\omega'}(\mathcal{F} \cap vars(\omega')) \equiv \bigcup_{T\in\tau(\omega',\mathcal{F})} Q_T(\mathcal{F} \cap vars(\omega')). \tag{C.1}$$

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

Inductive step: Assume that ω' has root variable X and subtrees $\omega'_1, \dots, \omega'_k$. Let $keys = \mathsf{anc}(X) \cup \{X\}$, $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega'))$, and $Q_X(\mathcal{F}_X) = \Join_{R(\mathcal{X}) atoms(\omega')} R(\mathcal{X})$. The procedure τ distinguishes 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* δ_0*-hierarchical).* The procedure τ returns a view tree T constructed by the procedure BuildVT$(\cdot, \omega', \mathcal{F}_X)$. The leaves of T are the atoms of ω'. This implies Equivalence C.1.

Case 1 does not hold and $X \in \mathcal{F}$: The set of view trees $\tau(\omega', \mathcal{F})$ is defined as follows: for each set $\{T_i\}_{i\in[k]}$ with $T_i \in \tau(\omega'_i, \mathcal{F})$, the set $\tau(\omega', \mathcal{F})$ contains the view tree $\textsc{NewVT}(\cdot, \mathit{keys}, \{\hat{T}_i\}_{i\in[k]})$ where $\hat{T}_i = \textsc{AuxView}(\text{root of } \omega'_i, T_i)$ for $i \in [k]$.

Using the induction hypothesis, we rewrite as follows:

$$Q_{\omega'}(\mathcal{F} \cap \mathit{vars}(\omega')) = \bowtie_{i\in[k]} Q_{\omega'_i}(\mathcal{F} \cap \mathit{vars}(\omega'_i)) \overset{\text{IH}}{\equiv} \bowtie_{i\in[k]} \Big(\bigcup_{T\in\tau(\omega'_i,\mathcal{F})} Q_T(\mathcal{F} \cap \mathit{vars}(\omega'_i)) \Big)$$

$$\equiv \bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} \bowtie_{i\in[k]} Q_{T_i}(\mathcal{F} \cap \mathit{vars}(\omega'_i)) \equiv \bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} Q_{\textsc{NewVT}(\cdot,\mathit{keys},\{\hat{T}_i\}_{i\in[k]})}(\mathcal{F} \cap \mathit{vars}(\omega'))$$

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

Case 1 does not hold and $X \notin \mathcal{F}$: The procedure τ creates the views $\mathit{All}_X(\mathit{keys}) = \bowtie_{R(\mathcal{X})\in \mathit{atoms}(\omega')} R(\mathcal{X})$, $L_X(\mathit{keys}) = \bowtie_{R(\mathcal{X})\in \mathit{atoms}(\omega')} R^{\mathit{keys}}(\mathcal{X})$, and $H_X(\mathit{keys}) = \mathit{All}_X(\mathit{keys}) \bowtie \nexists L_X(\mathit{keys})$. It then returns the view trees $\{\mathit{ltree}\} \cup \mathit{htrees}$ defined as follows:

– $\mathit{ltree} = \textsc{BuildVT}(\cdot, \omega^{\mathit{keys}}, \mathcal{F})$, where ω^{keys} has the same structure as ω' but each atom is replaced by its light part;
– for each set $\{T_i\}_{i\in[k]}$ with $T_i \in \tau(\omega'_i, \mathcal{F})$, htrees contains the view tree $\textsc{NewVT}(\cdot, \mathit{keys}, \{\exists H_X\} \cup \{\hat{T}_i\}_{i\in[k]})$ where $\hat{T}_i = \textsc{AuxView}(\text{root of } \omega'_i, T_i)$ for $i \in [k]$.

From $\mathit{ALL}_X(\mathit{keys}) = L_X(\mathit{keys}) \cup H_X(\mathit{keys})$, we derive the following equivalence. For simplicity, we skip the schemas of queries:

$$\bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} \bowtie_{i\in[k]} Q_{T_i} \equiv Q_{\mathit{ltree}} \cup \bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} Q_{\textsc{NewVT}(\cdot,\mathit{keys},\{\exists H_X\}\cup\{\hat{T}_i\}_{i\in[k]})} \tag{C.2}$$

Using Equivalence (C.2) and the induction hypothesis, we obtain:

$$\begin{aligned} Q_{\omega'} &= \bowtie_{i\in[k]} Q_{\omega'_i} \overset{\text{IH}}{\equiv} \bowtie_{i\in[k]} \Big(\bigcup_{T\in\tau(\omega'_i,\mathcal{F})} Q_T \Big) \equiv \bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} \bowtie_{i\in[k]} Q_{T_i} \\ &\overset{(C.2)}{\equiv} Q_{\mathit{ltree}} \cup \bigcup_{\forall i\in[k]: T_i\in\tau(\omega'_i,\mathcal{F})} Q_{\textsc{NewVT}(\cdot,\mathit{keys},\{\exists H_X\}\cup\{\hat{T}_i\}_{i\in[k]})} \\ &= Q_{\mathit{ltree}} \cup \bigcup_{T\in \mathit{htrees}} Q_T = \bigcup_{T\in\tau(\omega',\mathcal{F})} Q_T \end{aligned}$$

C.2. **Proof of Proposition 5.4.**

Proposition 5.4. *Given a hierarchical query $Q(\mathcal{F})$ with static 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 $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$ time.*

We analyze the procedure τ from Figure 11 for both of the cases mode = 'static' and mode = 'dynamic'. We show that in both cases the time to materialize the set of view trees $\tau(\omega, \mathcal{F})$ is $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$.

We explain the intuition behind the complexity analysis. If the procedure τ runs in 'static' mode and Q is free-connex, or it runs in 'dynamic' mode and Q is δ_0-hierarchical, the procedure constructs a view tree that can be materialized in $\mathcal{O}(N)$ time. Otherwise, there must be at least one bound variable X in ω such that the subtree ω_X rooted at X contains free variables. In this case, the algorithm partitions the relations at the leaves of ω_X into heavy and light parts and creates view trees for computing parts of the query. The time to materialize the views of the view trees where at least one leaf relation is heavy is $\mathcal{O}(N)$. The overall time to materialize the view trees $\tau(\omega, \mathcal{F})$ is dominated by the time to materialize the views of the view trees where all leaf relations are light. In the worst case, the root variable of ω is bound and we need to materialize a view that joins the light parts of all leaf relations in ω and has the entire set $\mathcal{F}$ as free variables. We can compute such a view $V(\mathcal{F})$ as follows. We first aggregate away all bound variables that are not ancestors of free variables in ω. By using the algorithm InsideOut [AKNR16], this can be done in $\mathcal{O}(N)$ time. Then, we choose one atom to iterate over the tuples of its relation (outer loop of the evaluation). For each such tuple, we iterate over the matching tuples in the relations of the other atoms (inner loops of the evaluation). 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 $\boldsymbol{\lambda}$ of $\mathcal{F}$. The schema of each atom that is mapped to 0 by $\boldsymbol{\lambda}$ must be subsumed by the schema of an atom mapped to 1. 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 during the iteration over the tuples of the other atoms. By exploiting the degree constraints on light relation parts, the view $V(\mathcal{F})$ can be materialized in $\mathcal{O}(N^{1+(\rho(\mathcal{F})-1)\epsilon})$ time. By Proposition 4.5, $\rho(\mathcal{F}) = \rho^*(\mathcal{F})$. Considering the time needed to aggregate away bound variables before computing $V(\mathcal{F})$, we get $\mathcal{O}(N^{\max\{1,1+(\rho^*(\mathcal{F})-1)\epsilon\}})$ overall time complexity. We show that $\max\{1, 1 + (\rho^*(\mathcal{F}) - 1)\epsilon\}$ is upper-bounded by $1 + (\mathsf{w}(Q) - 1)\epsilon$.

The proof is structured following the basic building blocks of the procedure τ. Lemmas C.2-C.6 give upper bounds on the times to materialize the views in the view trees returned by the procedures NewVT (Figure 7), AuxView (Figure 8) BuildVT (Figure 6), and IndicatorVTs (Figure 10). Lemma C.7 states the complexity of the procedure τ based on a measure ξ defined over canonical variable orders. The section closes with the proof of Proposition 5.4 that bridges the measure ξ and the static width of hierarchical queries.

We introduce the measure ξ. Let ω be a canonical variable order, $\mathcal{F} \subseteq \mathit{vars}(\omega)$, and X a variable or atom in ω. We denote by ω_X the subtree of ω rooted at X and by Q_X a query that joins the atoms at the leaves of ω_X. We define

$$\xi(\omega, X, \mathcal{F}) = \max_{\substack{Y \in \mathit{vars}(\omega_X) \\ (\mathsf{anc}(Y) \cup \{Y\}) \not\subseteq \mathcal{F}}} \{\rho^*_{Q_X}(\mathit{vars}(\omega_Y) \cap \mathcal{F})\}.$$

If ω_X does not contain a variable Y with $(\mathsf{anc}(Y) \cup \{Y\}) \not\subseteq \mathcal{F}$, then $\xi(\omega, X, \mathcal{F}) = 0$. If X has children $X_1, \ldots, X_k$, then

$$\xi(\omega, X, \mathcal{F}) \geq \max_{i \in [k]} \{\xi(\omega, X_i, \mathcal{F})\}. \tag{C.3}$$

We start with an observation that each view V constructed by the procedures BuildVT (Figure 6), NewVT (Figure 7), AuxView (Figure 8), IndicatorVTs (Figure 10), and τ (Figure 11) at some node X of a variable order ω contains in its schema all variables in the root path of X and no variables which are not in ω_X. Moreover, V results from the join of its child views. This can be shown by a straightforward induction over the structure of ω.

Observation C.1. Let ω be a canonical variable order and $V(\mathcal{F})$ a view constructed at some node X of ω by one of the procedures BuildVT, NewVT, AuxView, IndicatorVTs, and τ. It holds

(1) $\mathsf{anc}(X) \subseteq \mathcal{F} \subseteq \mathsf{anc}(X) \cup \mathit{vars}(\omega)$.
(2) If $V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$ are the child views of $V(\mathcal{F})$, then $V(\mathcal{F}) = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$.

The next lemma gives a bound on the time to materialize the views in a view tree returned by the procedure NewVT in Figure 7.

Lemma C.2. *Given a set $\{T_i\}_{i\in[k]}$ of view trees with root views $\{V_i(\mathcal{S}_i)\}_{i\in[k]}$, let M_i be the time to materialize the views in T_i, for $i \in [k]$. If the query $V(\mathcal{S}) = V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ with $\mathcal{S} \subseteq \bigcap_{i\in[k]} \mathcal{S}_i$ is δ_0-hierarchical, the views in the view tree* NewVT$(\cdot, \mathcal{S}, \{T_i\}_{i\in[k]})$ *can be materialized in $\mathcal{O}(\max_{i\in[k]}\{M_i\})$ time.*

Proof. The procedure NewVT defines the view $V(\mathcal{S}) = V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ (Line 2). The view tree T returned by NewVT is defined as follows (Line 4): If $k = 1$ and $\mathcal{S} = \mathcal{S}_1$, then $T = T_1$; otherwise, T is the view tree that has root $V(\mathcal{S})$ and subtrees $T_1, \ldots, T_k$. By assumption, the time to materialize the views in the trees $T_1, \ldots, T_k$ is $\mathcal{O}(\max_{i\in[k]}\{M_i\})$. Hence, the sizes of the materialized root views $V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$ must be $\mathcal{O}(\max_{i\in[k]}\{M_i\})$. Assume that the query defining $V(\mathcal{S})$ is δ_0-hierarchical. Hence, we can construct a free-top canonical variable order for the query. We materialize $V(\mathcal{S})$ as follows. Traversing the variable order bottom-up, we aggregate away all bound variables using the InsideOut algorithm [AKNR16]. Since the query defining $V(\mathcal{S})$ is α-acyclic, this aggregation phase can be done in time linear in the size of the views $V_1(\mathcal{S}_1), \ldots, V_k(\mathcal{S}_k)$. Thus, the aggregation phase requires $\mathcal{O}(\max_{i\in[k]}\{M_i\})$ time. It follows that the time to materialize the views in the tree returned by NewVT is $\mathcal{O}(\max_{i\in[k]}\{M_i\})$. □

We proceed with a lemma that gives a bound on the time to materialize the views in the view tree returned by the procedure AuxView in Figure 8.

Lemma C.3. *Let T be a view tree and M the time to materialize the views in T. The views in the view tree* AuxView$(\cdot, T)$ *can be materialized in time $\mathcal{O}(M)$.*

Proof. Assume that the parameters of the procedure AuxView are $\mathcal{Z}$ and T. Let $V(\mathcal{S})$ be the root of T. If the condition in Line 3 of the procedure AuxView does not hold, the procedure returns T (Line 5). Otherwise, it returns a view tree T' that results from T by adding a view $V'(\mathsf{anc}(Z))$ on top of $V(\mathcal{S})$ (Line 4). Since $\mathsf{anc}(Z) \subset \mathcal{S}$, $V'(\mathsf{anc}(Z))$ results from $V(\mathcal{S})$ by aggregating away the variables in $\mathcal{S} - \mathsf{anc}(Z)$. Since the size of $V(\mathcal{S})$ must be $\mathcal{O}(M)$ and the variables can be aggregated away in time linear in the size of $V(\mathcal{S})$, the overall time to materialize the views in the output tree T' is $\mathcal{O}(M)$. □

The following lemma says that if the input to the procedure BuildVT in Figure 6 represents a free-connex query, the procedure outputs a view tree whose views can be materialized in time linear in the database size.

Lemma C.4. *Let ω be a canonical variable order, X a node in ω, N the size of the leaf relations of ω, and $\mathcal{F}$ a set of variables. If the query $Q_X(\mathcal{F}') =$ join of atoms(ω_X) with $\mathcal{F}' = \mathcal{F} \cap (\mathsf{anc}(X) \cup \mathit{vars}(\omega_X))$ is free-connex, the views in the view tree* BuildVT$(\cdot, \omega_X, \mathcal{F})$ *can be materialized in $\mathcal{O}(N)$ time.*

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

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

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

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

$$Q_{X_i}(\mathcal{F}_i) = \text{join of } atoms(\omega_{X_i}) \text{ is free-connex.} \tag{C.4}$$

An α-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 α-acyclic [BB13]. A query is α-acyclic if it has a (not necessarily free-top) variable order with static width 1 [OZ15, BFMY83]. Let Q'_X be the query that results from Q_X by adding a new atom $R(\mathcal{F}')$. Likewise, let Q'_{X_i} be the query that we obtain from Q_{X_i} by adding a new atom $R_i(\mathcal{F}_i)$, for $i \in [k]$. Since Q_X is free-connex, there must be a variable order $\omega' = (T_{\omega'}, dep_{\omega'})$ for Q'_X, such that $\mathsf{w}(\omega') = 1$. In the following, we turn ω' into a variable order $\omega'_i = (T_{\omega'_i}, dep_{\omega'_i})$ for Q'_{X_i} with $\mathsf{w}(\omega'_i) = 1$, for $i \in [k]$. From this, it follows that Q_{X_i} is free-connex, for each $i \in [k]$. To obtain ω'_i, we traverse ω' bottom-up and eliminate all variables and atoms (including $R(\mathcal{F}')$) that do not occur in Q'_{X_i}. When eliminating a node Y with a parent node Z, we append the children of Y to Z. If Y does not have any parent node, the subtrees rooted at its children nodes become independent. Finally, we append $R_i(\mathcal{F}_i)$ under the lowest variable Y in the obtained variable order such that Y is included in $\mathcal{F}_i$. In the following we show that for $i \in [k]$:

(1) ω'_i is a valid variable order for Q'_{X_i}.
(2) $\mathsf{w}(\omega'_i) = 1$.

(1) ω'_i is a valid variable order for Q'_{X_i}, for $i \in [k]$: The following property follows from the construction of the variable order ω'_i:

($*$): Any two variables in $vars(Q'_{X_i})$ that are on the same root-to-leaf path in ω' remain on the same root-to-leaf path in ω'_i.

Each atom $K(\mathcal{X})$ in $atoms(Q'_{X_i}) - \{R_i(\mathcal{F}_i)\}$ is also an atom in $atoms(Q'_X)$. Hence, the variables in $\mathcal{X}$ must be on the same root-to-leaf path in ω'. Due to ($*$), they also must be on the same root-to-leaf path in ω'_i. It remains to show that all variables in $\mathcal{F}_i$ are on the same root-to-leaf path in ω'_i. In the canonical variable order ω, each variable in $\mathcal{F}' - \{X\}$ is either above or below X. Hence, X depends on all variables in $\mathcal{F}' - \{X\}$, which means that all variables in $\mathcal{F}'$ must be on the same root-to-leaf path in ω'. Due to $\mathcal{F}_i \subseteq \{X\} \cup \mathcal{F}'$ and Property ($*$), all variables in $\mathcal{F}_i$ must be on the same root-to-leaf path in ω'_i.

(2) $\mathsf{w}(\omega'_i) = 1$, for $i \in [k]$: Let $Y \in vars(\omega'_i)$ for some $i \in [k]$. We need to show that $\rho^*_{Q'_{X_i}}(\{Y\} \cup dep_{\omega'_i}(Y)) = 1$. By Proposition 4.5, it suffices to show that Q'_{X_i} contains an atom that covers $\{Y\} \cup dep_{\omega'_i}(Y)$, i.e., whose schema includes the latter set. By construction, Y must be included in ω'. First, observe that any two variables that are not dependent in Q'_X cannot be dependent in Q'_{X_i}. Moreover, each variable Z included in the root path of Y in ω'_i, is included in the root path of Y in ω'. Hence:

($**$) $\{Y\} \cup dep_{\omega'_i}(Y) \subseteq (\{Y\} \cup dep_{\omega'}(Y)) \cap vars(Q'_{X_i})$.

Due to $\mathsf{w}(\omega') = 1$ and Proposition 4.5, there must be an atom $K(\mathcal{X}) \in atoms(Q'_X)$ such that $\{Y\} \cup dep_{\omega'}(Y) \subseteq \mathcal{X}$. First, assume that $K(\mathcal{X}) \neq R(\mathcal{F}')$. Since $\mathcal{X}$ includes Y, the atom $K(\mathcal{X})$ must be under the variable Y in ω_{X_i}, which means that $atoms(Q'_{X_i})$ includes $K(\mathcal{X})$. Due to Property ($**$), $K(\mathcal{X})$ covers $\{Y\} \cup dep_{\omega'_i}(Y)$. Now assume that

$K(\mathcal{X}) = R(\mathcal{F}')$. This means that $\{Y\} \cup dep_{\omega'}(Y) \subseteq \mathcal{F}'$. By Property $(**)$, $\{Y\} \cup dep_{\omega'_i}(Y) \subseteq (\{Y\} \cup dep_{\omega'}(Y)) \cap vars(Q'_{X_i}) \subseteq \mathcal{F}' \cap vars(Q'_{X_i})$. Since $\mathcal{F}_i = \mathcal{F}' \cap vars(Q'_{X_i})$, $R_i(\mathcal{F}_i)$ covers $\{Y\} \cup dep_{\omega'_i}(Y)$.

This completes the proof of (C.4).

Let $T = \text{BuildVT}(\cdot, \omega_X, \mathcal{F})$. To construct the view tree T, the procedure BuildVT first constructs the view trees $\{T_i\}_{i\in[k]}$ with $T_i = \text{BuildVT}(\cdot, \omega_{X_i}, \mathcal{F})$ for each $i \in [k]$ (Line 2). By Property (C.4) and the induction hypothesis, the views in each view tree T_i can be materialized in $\mathcal{O}(N)$ time. In the following we show that all views in T can be materialized in $\mathcal{O}(N)$ time. We distinguish whether $\mathsf{anc}(X) \cup \{X\}$ is included in $\mathcal{F}$ (Lines 4-7) or not (Lines 8-10):

Case $\mathsf{anc}(X) \cup \{X\} \subseteq \mathcal{F}$*:* In this case, it holds $T = \text{NewVT}(\cdot, \mathcal{F}_X, subtrees)$, where $subtrees = \{\, \text{AuxView}(X_i, T_i)\,\}_{i\in[k]}$ and $\mathcal{F}_X = \mathsf{anc}(X) \cup \{X\}$. The procedures NewVT and AuxView are given in Figures 7 and 8, respectively. By the induction hypothesis and Lemma C.3, the views in *subtrees* can be materialized in $\mathcal{O}(N)$ time. Let $V'_1(\mathcal{F}'_1), \ldots, V_k(\mathcal{F}'_k)$ be the roots of the trees in *subtrees*. The overall size of these root views must be $\mathcal{O}(N)$. Observation C.1.(1) implies that for any $i, j \in [k]$ with $i \neq j$, it holds $\mathcal{F}'_i \cap \mathcal{F}'_i = \mathcal{F}_X$. Hence, the query $V_X(\mathcal{F}_X) = V_1(\mathcal{F}'_1), \ldots, V_k(\mathcal{F}'_k)$ is δ_0-hierarchical. Since $\mathcal{F}_X = \mathsf{anc}(X) \cup \{X\} \subseteq \bigcap_{i\in[k]} \mathcal{F}'_i$, it follows from Lemma C.2 that the views in T can be materialized in $\mathcal{O}(N)$ time.

Case $\mathsf{anc}(X) \cup \{X\} \not\subseteq \mathcal{F}$: n this case, we have $T = \text{NewVT}(\cdot, \mathcal{F}_X, subtrees)$, where $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap vars(\omega_X))$ and $subtrees = \{T_i\}_{i\in[k]}$. Let $V'_i(\mathcal{F}'_i)$ be the root of T_i, for $i \in [k]$. By the definition of the procedure NewVT, the tree T results from the trees $\{T_i\}_{i\in[k]}$ by adding a new root view defined by $V_X(\mathcal{F}_X) = V'_1(\mathcal{F}'_1), \ldots, V'_k(\mathcal{F}'_k)$. It follows from Observation C.1.(2), that $V_X(\mathcal{F}_X)$ can be rewritten as $V_X(\mathcal{F}_X) =$ join of $atoms(\omega_X)$. We show that the view $V_X(\mathcal{F}_X)$ can be materialized in $\mathcal{O}(N)$ time. The set $atoms(\omega_X)$ must contain an atom $R(\mathcal{Y})$ with $\mathcal{F}_X \subseteq \mathcal{Y}$ (Lemma 35 in [KNOZ19]). Hence, we can easily materialize the view $V_X(\mathcal{F}_X)$ by using the InsideOut algorithm [AKNR16] to aggregate away all variables that are not included in $\mathcal{F}_X$. Since the query defining the view $V_X(\mathcal{F}_X)$ is (α-)acyclic, the whole computation takes $\mathcal{O}(N)$ time. □

The next lemma upper bounds the time to materialize the views constructed by the procedure BuildVT in Figure 6 for a variable order ω_X^{keys}. This variable order has the same structure as ω_X yet each atom $R(\mathcal{Y})$ is replaced by the light part $R^{keys}(\mathcal{Y})$ of relation R partitioned on the variable set *keys* (cf. Section 5.2).

Lemma C.5. *Given a canonical variable order ω, a node X in ω, the size N of the leaf relations in ω, $keys = \mathsf{anc}(X) \cup \{X\}$, $\mathcal{F} \subseteq vars(\omega)$, and $\epsilon \in [0, 1]$. The view tree* $\text{BuildVT}(\cdot, \omega_X^{keys}, \mathcal{F})$ *can be materialized in* $\mathcal{O}(N^{\max\{1, 1+(\xi(\omega^{keys}, X, \mathcal{F})-1)\epsilon\}})$ *time.*

Proof. For a node X in ω^{keys}, we set

$$m_X = \max\{1, 1 + (\xi(\omega^{keys}, X, \mathcal{F}) - 1)\epsilon\}.$$

The proof is by induction on the structure of ω_X^{keys}.

Base case: If ω_X^{keys} is a single atom $R(\mathcal{X})$, the procedure BuildVT returns this atom, which can be materialized in $\mathcal{O}(N)$ time. Since $m_X \geq 1$, this completes the base case.

Inductive step: Assume $X \in vars(\omega^{keys})$ and has child nodes $X_1, \ldots, X_k$. The procedure first calls $\text{BuildVT}(\cdot, \omega_{X_i}^{keys}, \mathcal{F})$ for each $i \in [k]$ and produces the view trees $\{T_i\}_{i\in[k]}$ (Line 2).

By induction hypothesis, we need $\mathcal{O}(N^{m_{X_i}})$ time to materialize the views in each view tree T_i with $i \in [k]$. The procedure BUILDVT distinguishes whether $(\mathsf{anc}(X) \cup \{X\}) \subseteq \mathcal{F}$ (Lines 4-7) or not (Lines 8-10).

Case $(\mathsf{anc}(X) \cup \{X\}) \subseteq \mathcal{F}$: The view tree T returned by the procedure BUILDVT is NEWVT$(\cdot, \mathcal{F}_X, \mathit{subtrees})$, where *subtrees* is defined as $\{\,\text{AUXVIEW}(X_i, T_i)\,\}_{i\in[k]}$ and $\mathcal{F}_X = \mathsf{anc}(X) \cup \{X\}$. By induction hypothesis and Inequality (C.3), the overall time to materialize the views in $\{T_i\}_{i\in[k]}$ is $\mathcal{O}(N^{m_X})$. For each view tree T_i, AUXVIEW(X_i, T_i) adds at most one view with schema $\mathsf{anc}(X_i)$ on top of the root view of T_i. Then, $\mathsf{anc}(X_i)$ is a subset of the schema of the root view of T_i. Since the size of the root view of T_i must be bounded by $\mathcal{O}(N^{m_X})$, the view added by AUXVIEW can be materialized in $\mathcal{O}(N^{m_X})$ time. Assume that $V_1(\mathcal{F}_1), \dots, V_k(\mathcal{F}_k)$ are the roots of the view trees in *subtrees*. In case $k = 1$ and $\mathcal{F} = \mathcal{F}_i$, NEWVT$(V_X, \mathcal{F}_X, \mathit{subtrees})$ returns $V_1(\mathcal{F}_1)$; otherwise, it returns a view tree that has $V_X(\mathcal{F}_X) = V_1(\mathcal{F}_1), \dots, V_k(\mathcal{F}_k)$ as root view and *subtrees* as subtrees. By the definition of AUXVIEW, it holds $\mathcal{F}_i \cap \mathcal{F}_j = (\mathsf{anc}(X) \cup \{X\}) = \mathcal{F}_X$ for any $i, j \in [k]$. Hence, the view $V_X(\mathcal{F}_X)$ can be computed by iterating over the tuples in a view $V_i(\mathcal{F}_i)$ with $i \in [k]$ and filtering out those tuples that do not have matching tuples in all views $V_j(\mathcal{F}_j)$ with $j \in [k] - \{i\}$. Since the size of $V_i(\mathcal{F}_i)$ is $\mathcal{O}(N^{m_X})$ and materialized views allow constant time lookups, the view $V_X(\mathcal{F}_X)$ can be computed in $\mathcal{O}(N^{m_X})$ time. It follows that the view tree T returned by BUILDVT can be materialized in $\mathcal{O}(N^{m_X})$ time. This completes the inductive step for this case.

Case $(\mathsf{anc}(X) \cup \{X\}) \not\subseteq \mathcal{F}$: The procedure BUILDVT sets $\mathcal{F}_X = \mathsf{anc}(X) \cup (\mathcal{F} \cap \mathit{vars}(\omega_X^{keys}))$ and $\mathit{subtrees} = \{T_i\}_{i\in[k]}$. The view tree T returned by the procedure BUILDVT is NEWVT$(\cdot, \mathcal{F}_X, \mathit{subtrees})$. We show that all views in the view tree T can be materialized in $\mathcal{O}(N^{m_X})$ time. We analyze the steps in NEWVT. In case *subtrees* consists of a single tree T' such that the schema of the root view of T' is $\mathcal{F}_X$, the procedure NEWVT returns the view tree T'. By induction hypothesis and Inequality (C.3), the views in $T = T'$ can be materialized in $\mathcal{O}(N^{m_X})$ time. Otherwise, let $V(\mathcal{F}_i)$ be the root view of T_i, for $i \in [k]$. The tree T returned by NEWVT consists of the root view

$$V_X(\mathcal{F}_X) = V_1(\mathcal{F}_1), \dots, V_k(\mathcal{F}_k)$$

with subtrees $\{T_i\}_{i\in[k]}$. By induction hypothesis and Inequality (C.3), the views in the trees $\{T_i\}_{i\in[k]}$ can be materialized in $\mathcal{O}(N^{m_X})$ time. It suffices to show that $V_X(\mathcal{F}_X)$ can be materialized in $\mathcal{O}(N^{m_X})$ time. Using Observation C.1.(2), we rewrite the view $V_X(\mathcal{F}_X)$ using the leaf atoms of ω_X^{keys}:

$$V_X(\mathcal{F}_X) = \text{ join of } \mathit{atoms}(\omega_X^{keys}).$$

We materialize the view $V_X(\mathcal{F}_X)$ as follows. Using the InsideOut algorithm [AKNR16], we first aggregate away all variables in $\mathit{vars}(\omega_X^{keys}) - \mathcal{F}_X$ that are not above a variable from $\mathcal{F}_X$. Since the view V_X is defined by an α-acyclic query, the time required by this step is $\mathcal{O}(N)$. Let $V'_X(\mathcal{F}_X) = R_1(\mathcal{F}_1), \dots, R_k(\mathcal{F}_k)$ be the resulting query. We distinguish between two subcases.

Subcase 1: For all $R_i(\mathcal{F}_i)$, it holds $\mathcal{F}_i \cap \mathit{vars}(\omega_X^{keys}) = \emptyset$

This means that $\mathcal{F}_X$ and each $\mathcal{F}_i$ are contained in $\mathsf{anc}(X) \cup \{X\}$. Since ω^{keys} is canonical, the inner nodes of each root-to-leaf path are the variables of an atom. Hence, there is an $R_i(\mathcal{F}_i)$ with $i \in [k]$ such that $\mathcal{F}_i$ subsumes $\mathcal{F}_X$ and each $\mathcal{F}_j$ with $j \in [k]$. Thus, we can materialize the result of $V'_X(\mathcal{F}_X)$ in $\mathcal{O}(N)$ time by iterating over the tuples in R_i and doing constant-time lookups in the other relations.

Subcase 2: There is an $R_i(\mathcal{F}_i)$ with $\mathcal{F}_i \cap vars(\omega_X^{keys}) \neq \emptyset$

Let $\boldsymbol{\lambda} = (\lambda_{R_i(\mathcal{F}_i)})_{i\in[k]}$ be an edge cover of $\mathcal{F}_X \cap vars(\omega_X^{keys})$ with $\sum_{i\in[k]} \lambda_{R_i(\mathcal{F}_i)} = \rho^*_{V'_X}(\mathcal{F}_X \cap vars(\omega_X^{keys}))$. Since V'_X is hierarchical, we can assume that each $\lambda_{R_i(\mathcal{F}_i)}$ is either 0 or 1 (Proposition 4.5). There must be at least one $R_i(\mathcal{F}_i)$ with $\lambda_{R_i(\mathcal{F}_i)} = 1$, otherwise there cannot be any variable from $\mathcal{F}_X$ in ω_X^{keys} and we fall back to Subcase 1. Since ω_X^{keys} is canonical, 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 $\mathcal{F}_i \subseteq \mathcal{F}_j$. The atoms $R_1(\mathcal{F}_1), \ldots, R_k(\mathcal{F}_k)$ can still contain variables not included in $\mathcal{F}_X$. Each such variable appears above at least one variable from $\mathcal{F}_X$ in ω_X^{keys}. We first compute the result of the view $V''_X(\bigcup_{i\in[k]} \mathcal{F}_k) = R_1(\mathcal{F}_1), \ldots, R_k(\mathcal{F}_k)$ as follows. We choose an arbitrary atom $R_i(\mathcal{F}_i)$ with $\lambda_{R_i(\mathcal{F}_i)} = 1$ and iterate over the tuples in R_i. For each such tuple, we iterate over the matching tuples in the other atoms mapped to 1 by $\boldsymbol{\lambda}$. For atoms that are not mapped to 1, it suffices to do constant-time lookups while iterating over one of their witnesses. To obtain the result of V'_X from V''_X, we aggregate away all variables not included in $\mathcal{F}_X$. Recall that for each atom $R_i(\mathcal{F}_i)$, there is an atom in $atoms(\omega_X^{keys})$ that is the light part of a relation partitioned on $keys = \mathsf{anc}(X) \cup \{X\}$. Hence, each tuple in the relation of an atom mapped to 1 by $\boldsymbol{\lambda}$ can be paired with $\mathcal{O}(N^\epsilon)$ tuples in the relation of any other atom mapped to 1. This means that the time to materialize V''_X and hence V'_X is $\mathcal{O}(N^{m'})$ where $m' = 1 + (\rho^*_{V'_X}(\mathcal{F}_X \cap vars(\omega_X^{keys})) - 1)\epsilon$. Since V'_X results from V_X by aggregating away variables in $vars(\omega_X^{keys}) - \mathcal{F}_X$, we have $\rho^*_{V'_X}(\mathcal{F}_X \cap vars(\omega_X^{keys})) = \rho^*_{V_X}(\mathcal{F}_X \cap vars(\omega_X^{keys}))$. It follows from $\mathsf{anc}(X) \cup \{X\} \not\subseteq \mathcal{F}$ that $\rho^*_{V_X}(\mathcal{F}_X \cap vars(\omega_X^{keys})) = \xi(\omega^{keys}, X, \mathcal{F})$. Hence, the view V'_X can be materialized in $\mathcal{O}(N^{1+(\xi(\omega^{keys},X,\mathcal{F})-1)\epsilon})$ time.

We sum up the analysis for the case $(\mathsf{anc}(X) \cup \{X\}) \not\subseteq \mathcal{F}$: the initial aggregation step and the computation in Subcase 1 take $\mathcal{O}(N)$ time; the computation in Subcase 2 takes $\mathcal{O}(N^{1+(\xi(\omega^{keys},X,\mathcal{F})-1)\epsilon})$ time. Thus, given $m_X = \max\{1, 1 + (\xi(\omega^{keys}, X, \mathcal{F}) - 1)\epsilon\}$, the time to materialize the result of V_X is $\mathcal{O}(N^{m_X})$. This completes the inductive step in case $(\mathsf{anc}(X) \cup \{X\}) \not\subseteq \mathcal{F}$. □

The next lemma states that the view trees returned by the procedure IndicatorVTs from Figure 10 can be materialized in time linear in the database size.

Lemma C.6. *Let ω be a canonical variable order, X a variable in ω, and N the size of the leaf relations in the variable order ω. The views in the view trees returned by* IndicatorVTs(ω_X) *can be materialized in $\mathcal{O}(N)$ time.*

Proof. In Lines 3 and 4, the procedure constructs the view tree *alltree*, which is defined by BuildVT(“All”, ω_X, *keys*) and the view tree *ltree* = BuildVT(“L”, ω_X^{keys}, *keys*), where *keys* consists of the set $\mathsf{anc}(X) \cup \{X\}$. The variable order ω_X^{keys} results from ω_X by replacing each atom $R(\mathcal{X})$ by the atom $R^{keys}(\mathcal{X})$, which denotes the light part of relation R partitioned on *keys*. These light parts can be computed in $\mathcal{O}(N)$ time. The queries $Q_X(keys) =$ join of $atoms(\omega_X)$ and $Q_X^{keys}(keys) =$ join of $atoms(\omega_X)$ are free-connex. By using Lemma C.4, we derive that the views in *alltree* and *ltree* can be materialized in $\mathcal{O}(N)$ time. Hence, the roots *allroot* and *lroot* of *alltree* and *ltree*, respectively, can be materialized in $\mathcal{O}(N)$ time as well. It remains to analyze the time to materialize the views in the view tree $htree = \textsc{NewVT}(\cdot, \mathcal{F}, \{allroot, \neg lroot\})$ (Line 7). It follows from Observation C.1.(1) that $V(\mathcal{F}) = allroot, \nexists lroot$ is δ_0-hierarchical. By using Lemma C.2, we derive that the views in *htree* can be materialized in $\mathcal{O}(N)$ time. Overall, all views in the view trees $(alltree, ltree, htree)$ can be materialized in $\mathcal{O}(N)$ time. □

We use Lemmas C.2-C.6 to show an upper bound on the time to materialize the views in any tree produced by the procedure τ in Figure 11.

Lemma C.7. *Let ω be a canonical variable order, X a node in ω, $\mathcal{F} \subseteq vars(\omega)$, N the size of the leaf relations in ω, and $\epsilon \in [0,1]$. The views in the trees returned by $\tau(\omega_X, \mathcal{F})$ can be materialized in $\mathcal{O}(N^{\max\{1, 1+(\xi(\omega,X,\mathcal{F})-1)\epsilon\}})$ time.*

Proof. For simplicity, we set

$$m = \max\{1, 1 + (\xi(\omega, X, \mathcal{F}) - 1)\epsilon\}.$$

The proof is by induction on the structure of ω_X.

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

Inductive step: Assume that X is a variable with children nodes $X_1, \ldots, X_k$. 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 $\tau(\omega_X, \mathcal{F})$, 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 δ_0-hierarchical (Lines 5-7):

The procedure τ returns the view tree $\textsc{BuildVT}(\text{"V"}, \omega_X, \mathcal{F}_X)$ (Line 7). Since δ_0-hierarchical queries are in particular free-connex, it follows from Lemma C.4 that $\textsc{BuildVT}(\text{"V"}, \omega_X, \mathcal{F}_X)$ 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 8-11):
The set of view trees $\tau(\omega_X, \mathcal{F})$ is defined as follows: for each set $\{T_i\}_{i\in[k]}$ with $T_i \in \tau(\omega_{X_i}, \mathcal{F})$, the set $\tau(\omega_X, \mathcal{F})$ contains the view tree $\textsc{NewVT}(\cdot, keys, \{\hat{T}_i\}_{i\in[k]})$, where $\hat{T}_i = \textsc{AuxView}(X_i, T_i)$ for each $i \in [k]$. We consider one such set $\{T_i\}_{i\in[k]}$ of view trees. By induction hypothesis, the views in each T_i can be materialized in $\mathcal{O}(N^{\max\{1,1+(\xi(\omega,X_i,\mathcal{F})-1)\epsilon\}})$ time. It follows from Inequality (C.3), that the overall time to materialize the views in these view trees is $\mathcal{O}(N^m)$. By Lemma C.3, the views in each view tree $\hat{T}_i$ with $i \in [k]$ can be materialized in $\mathcal{O}(N^m)$ time. Let $V_i(\mathcal{F}_i)$ be the root view of $\hat{T}_i$, for $i \in [k]$. It follows from Observation C.1.(1) that *keys* is included in each $\mathcal{F}_i$ and the query $V_X(keys) = V_1(\mathcal{F}_1), \ldots, V_k(\mathcal{F}_k)$ is δ_0-hierarchical. Hence, it follows from Lemma C.2 that the views in the view tree $\textsc{NewVT}(\cdot, keys, \{\hat{T}_i\}_{i\in[k]})$ can be materialized in time $\mathcal{O}(N^m)$. This completes the inductive step in this case.

Case 3: Case 1 does not hold and $X \notin \mathcal{F}$ (Lines 12-17):
The procedure τ first calls $\textsc{IndicatorVTs}(\omega_X)$ (Line 12) given in Figure 10, which constructs the indicator view trees *alltree*, *ltree*, and *htree*. By Lemma C.6, the views in these view trees can be materialized in $\mathcal{O}(N)$ time. Let H_X be the root of *htree*. The only difference between the construction of the view trees returned in *Case 2* above and the view trees in the set *htrees* defined in Lines 13-15 is that the roots of the view trees in the latter set have

$\exists H_X$ as additional child view. By the same argumentation as in *Case 2*, it follows that the views in *htrees* can be materialized in $\mathcal{O}(N^m)$ time. Let $ltree = \text{BUILDVT}(\text{“V”}, \omega_X^{keys}, \mathcal{F}_X)$ as defined in Line 16, where ω_X^{keys} shares the same structure as ω_X, but each atom $R(\mathcal{X})$ is replaced with $R^{keys}(\mathcal{X})$ denoting the light part of relation R partitioned on *keys*. It follows from Lemma C.5 that the views in the view tree *ltree* can be materialized in $\mathcal{O}(N^m)$ time. Thus, all views of the views trees in the set $htrees \cup \{ltree\}$ can be materialized in $\mathcal{O}(N^m)$ time. This completes the inductive step for *Case 3*. □

Using Lemma C.7, we prove Proposition 5.4. Without loss of generality, assume that ω consists of a single connected component. Otherwise, we apply the same reasoning for each connected component. We also assume that Q contains at least one atom with non-empty schema. Otherwise, $\tau(\omega, \emptyset)$ returns a single atom with empty schema, which can obviously be materialized in constant time.

By Lemma C.7, the view trees generated by $\tau(\omega, \mathcal{F})$ can be materialized in time $\mathcal{O}(N^{\max\{1,1+(\xi(\omega,X,\mathcal{F})-1)\epsilon\}})$, where X is the root variable of ω. It remains to show:

$$\max\{1, 1 + (\xi(\omega, X, \mathcal{F}) - 1)\epsilon\} \le 1 + (\mathsf{w} - 1)\epsilon. \tag{C.5}$$

First, assume that $\xi(\omega, X, \mathcal{F}) = 0$. This means that $\max\{1, 1+(\xi(\omega, X, \mathcal{F})-1)\epsilon\} = 1$. Since Q contains at least one atom with non-empty schema, we have $\mathsf{w} \ge 1$. Thus, Inequality (C.5) holds. Now, let $\xi(\omega, X, \mathcal{F}) = \ell \ge 1$. We show that $\mathsf{w} \ge \ell$. It follows from $\xi(\omega, X, \mathcal{F}) = \ell$ that ω contains a bound variable Y such that $\rho_Q^*(B) = \ell$, where $B = vars(\omega_Y) \cap \mathcal{F}$. The inner nodes of each root-to-leaf path of a canonical variable order are the variables of an atom. Hence, for each variable $Z \in B$, there must be an atom in Q that contains both Y and Z. This means that Y and Z depend on each other. Let $\omega' = (T, dep_{\omega'})$ be an arbitrary free-top variable order for Q. Since all variables in B depend on Y, each of them must be on a root-to-leaf path with Y. Since Y is bound and the variables in B are free, the set B must be included in $\mathsf{anc}(Y)$. Hence, $B \subseteq dep_{\omega'}(Y)$. This means $\rho_Q^*(\{Y\} \cup dep_{\omega'}(Y)) \ge \ell$, which implies $\mathsf{w}(\omega') \ge \ell$. It follows $\mathsf{w} \ge \ell$.

Appendix D. Proofs of the Results in Section 6

Proposition 6.1. *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 constructed by $\tau(\omega, \mathcal{F})$ for a canonical variable order ω for Q.*

Following Proposition 5.3, the union of queries defined by the set of view trees constructed by $\tau(\omega, \mathcal{F})$ is equivalent $Q(\mathcal{F})$. 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 consisting of a hierarchy of views admits constant delay [OZ15]. In the static case, this holds for free-connex hierarchical queries; in the dynamic case, this holds for δ_0-hierarchical queries (Section 5.1).

The view subtrees constructed over the light parts of input relations only do not bring additional difficulty. By construction (Section 5), the root view V of such a subtree T contains all the free variables that are present in T. In this case, the *open* and *next* calls stop at V and do not explore the children of V. This means that for enumeration purposes, we can discard the descendants of V.

By grounding the heavy indicators in T, we obtain instances of T that may represent overlapping relations. We next analyze the enumeration delay in the presence of heavy indicators as a function of the view tree instances of a view tree created for Q.

Consider one heavy indicator. Since its size is $\mathcal{O}(N^{1-\epsilon})$, it may lead to that many view tree instances. From each instance, we can enumerate with constant delay, and we can also look up a tuple with schema $\mathcal{S}$ in constant time. Given there are $\mathcal{O}(N^{1-\epsilon})$ such tuples, we can enumerate from T with $\mathcal{O}(N^{1-\epsilon})$ delay.

Consider p heavy indicators $\exists H_1(\mathcal{X}_1), \ldots, \exists H_p(\mathcal{X}_p)$ whose parents $V_1(\mathcal{X}_1), \ldots, V_p(\mathcal{X}_p)$ 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 $\exists H_i$ 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 $\exists H_i$ is $\mathcal{O}(N^{1-\epsilon})$, it follows that the size of V_i is also $\mathcal{O}(N^{1-\epsilon})$. When grounding $\exists H_i$, we create an instance for each tuple t that is in both $\exists H_i$ and V_i: If t were not in V_i, then there would be at least one sibling of $\exists H_i$ that does not have it. When opening the descendants of V_i before enumeration, only these tuples in V_i that also occur in $\exists H_i$ and in all its siblings can be extended at the descendant views, including all views V_j for $j > i$. The overall number of groundings for the h heavy indicators is therefore $\mathcal{O}(N^{1-\epsilon})$. Let n_i be the number of instances of $\exists H_i$. Then, the delay for enumerating from the union of $\exists H_i$ 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 $\exists H_j$ that are descendants of $\exists H_i$. The overall delay is that for the union of instances for $\exists H_1$: $\sum_{1 \leq j \leq p} n_j = \mathcal{O}(p \times N^{1-\epsilon}) = \mathcal{O}(N^{1-\epsilon})$.

Consider again the p heavy indicators, but this time their parents $V_1, \ldots, V_p$ are *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 $\mathcal{O}(p \times N^{1-\epsilon}) = \mathcal{O}(N^{1-\epsilon})$.

So far we discussed the case of enumerating from one view tree. In case of a set of view trees we 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.

Appendix E. Proofs of the Results in Section 7

E.1. **Proof of Proposition 7.2.**

Proposition 7.2. *Given a hierarchical query $Q(\mathcal{F})$ with dynamic 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 give the maintenance time for the views constructed by BuildVT given a δ_0-hierarchical query (Lemma E.1). We then show the maintenance time for the views constructed by τ given a hierarchical query (Lemma E.2). The maintenance time uses a new measure, which we relate to dynamic width (Lemma E.3). We finally show the running times of UpdateIndTree and UpdateTrees.

Lemma E.1. *Given a δ_0-hierarchical query $Q(\mathcal{F})$, a canonical variable order ω for Q, and a database of size N, the views constructed by* BuildVT$(\cdot, \omega, \mathcal{F})$ *from Figure 6 in the dynamic mode can be maintained under a single-tuple update to any input relation in $\mathcal{O}(1)$ time.*

Proof. At each node X of a canonical variable order ω for a δ_0-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 the set $\mathcal{F} \cap \mathit{vars}(\omega_X)$ of free variables in ω_X is empty for δ_0-hierarchical queries. The functions AuxView and NewVT maintain the following invariant for δ_0-hierarchical queries 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, AuxView constructs an extra view with $\mathsf{anc}(X)$ as free variables.

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 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 makes only constant-time lookups in the sibling views. Overall, propagating the update through the view tree constructed for a δ_0-hierarchical query using BuildVT takes constant time. □

Consider now a canonical variable order ω for a hierarchical query and a set $\mathcal{F}$ of free variables. Given a node X in ω, let Q_X denote the join of $\mathit{atoms}(\omega_X)$. We define $\kappa(\omega, \mathcal{F})$ as:

$$\max_{X \in \mathit{vars}(\omega) - \mathcal{F}} \; \max_{R(\mathcal{Y}) \in \mathit{atoms}(\omega_X)} \{\rho^*_{Q_X}((\mathit{vars}(\omega_X) \cap \mathcal{F}) - \mathcal{Y})\},$$

The measure $\kappa(\omega, \mathcal{F})$ is the maximal fractional edge cover number of Q_X over the free variables occurring in the subtree ω_X of ω rooted at a bound variable X, when the variables of one atom $R(\mathcal{Y})$ in ω_X are excluded.

Lemma E.2. *Given a hierarchical query $Q(\mathcal{F})$, a canonical variable order ω for Q, a database of size N, and $\epsilon \in [0,1]$, the views constructed by $\tau(\omega, \mathcal{F})$ from Figure 11 in the dynamic mode can be maintained under a single-tuple update in $\mathcal{O}(N^{\kappa(\omega,\mathcal{F})\epsilon})$ time.*

Proof. If Q is δ_0-hierarchical, the function τ returns a view tree for Q that admits $\mathcal{O}(1)$ update time, per Lemma E.1.

Consider now a view tree created by τ for a non-δ_0-hierarchical query. Let us restrict this view tree such that the views created in the light case are treated as leaf views. This restricted view tree encodes the result of a δ_0-hierarchical query! As the procedure τ traverses the variable order in a top-down manner, every bound variable X with a free variable below is replaced by a set of view trees where X is free (heavy case) and by a view tree whose root view aggregates away X and includes only free variables (light case). Thus, single-tuple updates to the leaves of this restricted view tree take 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 might not be constant. The view tree *ltree* constructed by BuildVT at a bound variable X is defined over the light parts of relations partitioned on $\mathit{keys} = \mathsf{anc}(X) \cup \{X\}$ (Line 16 in Figure 11). Each view V_Z in *ltree* constructed at a variable Z includes all the free variables in ω_Z. A single-tuple update δR to any relation R in *ltree* fixes the values of the variables *keys*, thus reducing the size of other relations in *ltree* to $\mathcal{O}(N^{\epsilon})$. The maintenance cost for V_Z under the update δR with schema $\mathcal{Y}$ is $\mathcal{O}(N^{m_Z \epsilon})$, where $m_Z = \rho^*_{Q_Z}((\mathit{vars}(\omega_Z) \cap \mathcal{F}) - \mathcal{Y})$. The maintenance cost for *ltree* is dominated by the maintenance cost for its root V_X.

The change computed at V_X for the single-tuple update consist of $\mathcal{O}(N^{m_X \epsilon})$ 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. The overall time needed to maintain V_X and propagate the change at V_X up to the root is $\mathcal{O}(N^{m'_X \epsilon})$, where $m'_X = \max_{R(\mathcal{Y}) \in \mathit{atoms}(\omega_X)}\{\rho^*_{Q_X}((\mathit{vars}(\omega_X) \cap \mathcal{F}) - \mathcal{Y})\}$. In the worst case, the root variable of ω is bound; then, maintaining the root view and its descendants takes $\mathcal{O}(N^{\kappa(\omega,\mathcal{F})\epsilon})$ time.

The views constructed by τ in the light cases thus determine the overall maintenance $\mathcal{O}(N^{\kappa(\omega,\mathcal{F})\epsilon})$ time. □

We next relate the measure $\kappa(\omega, \mathcal{F})$ to dynamic width.

Lemma E.3. *Given a canonical variable order ω for a hierarchical query $Q(\mathcal{F})$ with dynamic width δ, it holds that $\kappa(\omega, \mathcal{F}) \leq \delta$.*

Proof. Given any variable order ω' for Q and a variable X in ω', we denote by $Q^{\omega'}_X$ the query that joins the atoms in $\mathit{atoms}(\omega'_X)$. To prove $\kappa(\omega, \mathcal{F}) \leq \delta$, we need to show that

$$\kappa(\omega, \mathcal{F}) \leq \delta(\omega^f) \tag{E.1}$$

for any free-top variable order ω^f for Q. It follows from the definition of $\kappa(\omega, \mathcal{F})$ that ω has a bound variable X and an atom $R(\mathcal{Y}) \in \mathit{atoms}(Q^{\omega}_X)$ such that

$$\kappa(\omega, \mathcal{F}) = \rho^*_{Q^{\omega}_X}(\mathcal{B})$$

where $\mathcal{B} = (\mathit{vars}(\omega_X) \cap \mathcal{F}) - \mathcal{Y}$. Since ω is canonical, it holds:

(∗) Each atom in Q containing a variable from $\mathcal{B}$ must contain X.

Let $\omega^f = (T, \mathit{dep}_{\omega^f})$ be a free-top variable order for Q. Property (∗) implies that the variables in $\mathcal{B}$ depend on X. Since X is bound and all variables in $\mathcal{B}$ are free, the latter variables cannot be below X in ω^f. Hence, $\mathcal{B} \subseteq \mathit{dep}_{\omega^f}(X)$. Since $R(\mathcal{Y})$ contains X, it must be included in $\mathit{atoms}(\omega^f_X)$. To prove Inequality (E.1), it thus suffices to show:

$$\rho^*_{Q}(\mathcal{B}) \geq \rho^*_{Q^{\omega}_X}(\mathcal{B}). \tag{E.2}$$

By Property (∗), each atom in Q covering a variable from $\mathcal{B}$ contains X. Hence, all such atoms are contained in $\mathit{atoms}(Q^{\omega}_X)$. This implies that any fractional edge cover $\boldsymbol{\lambda}'$ of $\mathcal{B}$ using atoms in Q can be turned into a fractional edge cover $\boldsymbol{\lambda}$ of B using atoms in Q^{ω}_X such that $\sum_{\lambda \in \boldsymbol{\lambda}} \lambda \leq \sum_{\lambda' \in \boldsymbol{\lambda}'} \lambda'$. This implies Inequality (E.2) and hence Inequality (E.1). □

Lemma E.4. *Given an indicator tree T_{Ind} constructed by* IndicatorVTs *from Figure 10 and a single-tuple update δR,* UpdateIndTree *from Figure 18 runs in $\mathcal{O}(1)$ time.*

Proof. The tree T_{Ind} encodes the result of a δ_0-hierarchical query and admits constant-time updates per Lemma E.1. The remaining operations in UpdateIndTree also take constant time. □

We next analyze the procedure UpdateTrees from Figure 19 under a single-tuple update. Applying the update to each view tree from $\mathcal{T}$ (Line 1) takes $\mathcal{O}(N^{\delta\epsilon})$ time, per Lemmas E.2 and E.3. We then apply the update to each triple (T_{All}, T_L, T_H) of indicator view trees. The tree T_{All} is a view tree of a δ_0-hierarchical query, thus updating it takes constant time (Line 6). The tree T_L is updated using UpdateIndTree in constant time (Line 12), per Lemma E.4. Both of these changes may trigger a change in $\exists T_H$, and propagating $\delta(\exists H)$ through each view tree from $\mathcal{T}$ (Lines 9 and 14) takes constant time since this change does not affect any view materialized in the light case. Updating each light part of relation R and the affected view trees (Line 11) takes $\mathcal{O}(N^{\delta\epsilon})$ time, per Lemmas E.2 and E.3.

Overall, the procedure UpdateTrees maintains the views constructed by τ under a single-tuple update in $\mathcal{O}(N^{\delta\epsilon})$ time.

E.2. **Proof of Proposition 7.3.**

Proposition 7.3. *Given a hierarchical query $Q(\mathcal{F})$ with static 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.*

Consider the major rebalancing procedure from Figure 20. The light relation parts can be computed in $\mathcal{O}(N)$ time. Proposition 5.4 implies that the affected views can be recomputed in time $\mathcal{O}(N^{1+(\mathsf{w}-1)\epsilon})$.

E.3. **Proof of Proposition 7.4.**

Proposition 7.4. *Given a hierarchical query $Q(\mathcal{F})$ with dynamic 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.*

Figure 21 shows the procedure for minor rebalancing of the tuples with the partitioning value *key* in the light part $R^{\mathcal{S}}$ of relation R. Minor rebalancing either inserts fewer than $\frac{1}{2}M^{\epsilon}$ tuples into $R^{\mathcal{S}}$ (heavy to light) or deletes at most $\frac{3}{2}M^{\epsilon}$ tuples from $R^{\mathcal{S}}$ (light to heavy). Each action updates the indicator trees T_L and T_H in constant time (lines 5 and 6), per Lemma E.4. Propagating the update to the light part of relation R through each view tree from $\mathcal{T}$ (line 4) takes $\mathcal{O}(N^{\delta\epsilon})$ time, per Lemmas E.2 and E.3. 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 7.2. Since there are $\mathcal{O}(M^{\epsilon})$ such operations and the size invariant $\left\lfloor \frac{1}{4}M \right\rfloor \leq N < M$ holds, the total time is $\mathcal{O}(N^{(\delta+1)\epsilon})$.