Maintaining Queries under Updates Using Heavy-Light Partitioning of the Input Relations

MAHMOUD ABO-KHAMIS, Relational AI, USA
EDEN CHMIELEWSKI, University of Zurich, Switzerland
ANDREI DRAGHICI, University of Zurich, Switzerland
AHMET KARA, OTH Regensburg, Germany
DAN OLTEANU, University of Zurich, Switzerland

We study the classical incremental view maintenance problem: Given a query and a database, maintain the query output under single-tuple updates (inserts or deletes) to the database such that the tuples in the query output can be enumerated with constant delay after any update.

We introduce a maintenance approach whose update time matches or improves the best update time reported in prior work. Whereas prior approaches are manually tailored to each of a handful of queries, our approach generalizes to arbitrary join queries. It combines three techniques: delta queries, trees of materialized views, and heavy-light data partitioning. The overall update time incurred by our approach for a given join query is characterized by the maintenance width, a new measure that is parameterized by the heavy-light threshold for data partitioning. We show how to find the threshold that minimizes the maintenance width.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**; **Online algorithms**.

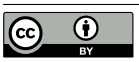
Additional Key Words and Phrases: incremental view maintenance; data partitioning; delta queries; join queries; materialized views

ACM Reference Format:
Mahmoud Abo-Khamis, Eden Chmielewski, Andrei Draghici, Ahmet Kara, and Dan Olteanu. 2026. Maintaining Queries under Updates Using Heavy-Light Partitioning of the Input Relations. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 109 (May 2026), 22 pages. https://doi.org/10.1145/3801905

1 Introduction

In this paper, we study the classical incremental view maintenance (IVM) problem for join (or full conjunctive) queries: Given a query and a database, we want to maintain the query output under single-tuple updates (inserts or deletes) to the database such that the tuples in the query output can be enumerated with constant delay after each update. This problem is central to databases and received attention from both database systems and theory communities over the past decades. As highlighted in a recent overview [27], there has been renewed interest in charting the complexity of the IVM problem [14, 16] and in developing IVM systems in academia [12, 17, 23, 30] and industry [6, 25, 29].

Authors' Contact Information: Mahmoud Abo-Khamis, mahmoudabo@gmail.com, Relational AI, Berkeley, CA, USA; Eden Chmielewski, edenviolet.chmielewski@uzh.ch, University of Zurich, Department of Informatics, Zurich, Switzerland; Andrei Draghici, andrei.draghici@uzh.ch, University of Zurich, Department of Informatics, Zurich, Switzerland; Ahmet Kara, ahmet.kara@oth-regensburg.de, OTH Regensburg, Department of Computer Science & Mathematics, Regensburg, Germany; Dan Olteanu, dan.olteanu@uzh.ch, University of Zurich, Department of Informatics, Zurich, Switzerland.



ACM 2836-6573/2026/5-ART109
https://doi.org/10.1145/3801905

Join Queries	Update Time	Update Time Upper Bounds				
	Lower Bounds	F-IVM [17]	IVM$^\epsilon$ [15, 16]	HHH [10]	MVIVM [1]	This Paper
hierarchical	-	$O(1)$	$O(1)$	N/A	$O(1)$	$O(1)$
3-path ($\nwarrow$)	$\Omega(N^{1/2-\gamma})$ [1, 5]	$O(N)$	$O(N^{1/2})^*$	$O(N^{1/2})^*$	$O(N^{1/2})$	$O(N^{1/2})$
4-path (Λ)	$\Omega(N^{1/2-\gamma})$ [1, 5]	$O(N)$	$O(N^{1/2})^*$	N/A	$O(N)$	$O(N^{1/2})$
triangle ($\triangle$)	$\Omega(N^{1/2-\gamma})$ [1, 5]	$O(N)$	$O(N^{1/2})$	$O(N^{1/2})^*$	$O(N^{1/2})$	$O(N^{1/2})$
LW-k	$\Omega(N^{1/2-\gamma})$ [1, 5]	$O(N)$	$O(N^{1/2})$	N/A	$O(N^{1/2})$	$O(N^{1/2})$
4-cycle ($\square$)	$\Omega(N^{2/3-\gamma})$ [1]	$O(N)$	N/A	$O(N^{2/3})^*$	$O(N)$	$O(N^{2/3})$
diamond ($\diamond$)	$\Omega(N^{2/3-\gamma})$ [1]	$O(N)$	N/A	$O(N^{2/3})^*$	$O(N)$	$O(N^{2/3})$
paw ($\dashv$)	$\Omega(N^{2/3-\gamma})$ [1]	$O(N)$	N/A	$O(N^{2/3})^*$	$O(N^{2/3})$	$O(N^{2/3})$
big paw ($\perp$)	$\Omega(N^{2/3-\gamma})$ [1]	$O(N)$	N/A	N/A	$O(N^{2/3})$	$O(N^{2/3})$
bow tie ($\bowtie$)	$\Omega(N^{3/4-\gamma})$ [1]	$O(N)$	N/A	N/A	$O(N)$	$O(N)$

Table 1. Comparison between the update times of our maintenance approach versus the best known combinatorial results for different join queries studied in the literature. Update times of all approaches are amortized, except for F-IVM which is worst-case. N is the size of the database at the time of the update. All results assume that after each update, constant delay enumeration of the query output is supported. (*) entries indicate update times shown only for the counting version of the problem; (N/A) entries indicate that the approach is not applicable; LW-k is the Loomis-Whitney query on k variables: $Q(X_1, \ldots, X_k) = \prod_{i \in [k]} R_i(\{X_1, \ldots, X_k\} - \{X_i\})$. All lower bounds assume the query has no self-joins and they hold for any $\gamma > 0$.

The classical IVM approach is based on delta queries [7]. More recently, two further techniques made their way into the IVM theory and systems: using a hierarchy, or tree, of materialized views [17, 23], and heavy-light data partitioning [14, 16]. View trees (and variants thereof) have been previously used for maintenance by several IVM systems. DBToaster [23] compiles the given query into a set of view trees, one for each updatable relation. Dynamic Yannakakis [12], F-IVM [13, 17], and Crown [30] compile the given query into one view tree, which is then maintained under updates. MVIVM translates the given query into a so-called multivariate extension query, which is maintained using view trees [1]. With the exception of q-hierarchical queries, which can be maintained using (a variant of) view trees with constant update time and constant enumeration delay as shown in a seminal work [5] and readily adopted by all subsequent approaches, the update time achieved by these approaches for arbitrary queries can be suboptimal.

Motivated by the suboptimality of the aforementioned IVM approaches, a distinct line of theoretical work [1, 10, 14–16] proposed adaptive maintenance approaches that partition the data and use different view trees for different data parts. This can lead to asymptotically lower and even optimal update times. These adaptive approaches were employed for a handful of queries, for which a careful crafting of view trees and the accompanying complexity analysis were made on a case-by-case basis, with no general approach in sight. One notable exception is the adaptive maintenance of hierarchical queries with arbitrary free variables [16]. IVM$^\epsilon$ is the first approach to achieve optimal (amortized) update time $O(N^{1/2})$ for the triangle count query [14] and for the full triangle query [15], where N is the database size at the time of update. A further optimality result is known for a subclass of hierarchical (but not q-hierarchical) queries [16], where the update time and enumeration delay are $O(N^{1/2})$. Yet tight bounds on the update time are not known beyond these notable cases. This is primarily due to the scarcity of the available lower bounds, which

are conditional on the Online Matrix-Vector-Multiplication (OMv) conjecture [5, 11][1] or on the conjectured optimality of the submodular width for static query evaluation [1].[2] Further approaches fall short of achieving the best known update times. For instance, the MVIVM approach [1], which reduces the IVM problem of a query to that of its multivariate extension, was shown [1, Fig. 4] to require $O(N)$ update time for the 4-cycle query, whereas the best known update time is $O(N^{2/3})$. Furthermore, MVIVM also needs $O(N)$ update time for the 4-path query, whereas the best update time is $O(N^{1/2})$. Table 1 overviews the update times (lower and upper bounds) achieved by representative *combinatorial*[3] approaches for queries studied in the literature.

In this paper, we put forward an adaptive maintenance approach that works for arbitrary join queries and whose update time matches or improves the best update time reported in prior work.

Our approach uses all three aforementioned techniques: delta queries, view trees, and data partitioning. The key challenge addressed by our approach is to algorithmically find the view trees and the heavy-light threshold parameter for data partitioning that minimize the update time for any given join query. This challenge was not addressed in prior works [10, 14–16], as the choices of view trees and threshold parameter were made manually for each of the considered queries. As shown in Table 1, our approach matches the best known upper bounds on the update time for the queries considered in the literature, while also providing a general approach for arbitrary join queries. It also achieves new non-trivial sub-linear update times for other queries, e.g., the optimal $O(N^{1/2})$ update time for the 4-path query.

This paper is organized as follows. Sec. 2 introduces preliminary notions used throughout the paper. Sec. 3 overviews our approach, exemplifies it on the 4-cycle query, and states our main theorem. Sec. 4 introduces the maintenance width, our new measure for the complexity of the maintenance cost. Sec. 5 compares the maintenance width and our evaluation strategy with width measures and strategies used in prior approaches. Sec. 6 discusses how to amortize the cost of occasional expensive updates over a sequence of updates. Sec. 7 reviews the constant-delay enumeration of tuples from the query output represented by the view trees. Sec. 8 concludes with thoughts on future work. Some proof details are deferred to the appendix. Further details on the amortization and enumeration procedures, as well as the application of our approach to the queries listed in Table 1, are provided in an extended version of this article.

2 Preliminaries

Data and Queries. A *schema* $\boldsymbol{X}$ is a tuple of attributes or variables $(X_1, \ldots, X_n)$, which we also conveniently see as a set to allow set operations on tuples. Each variable X_i draws its values from a set $\mathsf{Dom}(X_i)$. A tuple $\boldsymbol{x}$ of values over the schema $\boldsymbol{X}$ is an element of the set $\mathsf{Dom}(\boldsymbol{X}) = \mathsf{Dom}(X_1) \times \ldots \times \mathsf{Dom}(X_n)$. Following prior work on IVM, e.g., [17], a relation R over schema $\boldsymbol{X}$ is a function that maps tuples of values over $\boldsymbol{X}$ to multiplicities, which are integers. When applying set operations to R, we treat it as the set of tuples $\boldsymbol{x}$ with $R(\boldsymbol{x}) > 0$. For instance, the size of R, denoted

[1] In the OMv problem, we are given an $n \times n$ Boolean matrix $\mathbf{M}$ and receive n column vectors of size n denoted by $\mathbf{v}_1, \ldots, \mathbf{v}_n$, one by one; after seeing each $\mathbf{v}_i$, we output the product $\mathbf{M}\mathbf{v}_i$, before we see the next vector. The OMv conjecture states that for any $\gamma > 0$, there is no combinatorial algorithm that solves OMv in time $O(n^{3-\gamma})$ [11]. Unless the OMv conjecture fails, there is no dynamic algorithm that can enumerate the output of a non-q-hierarchical self-join-free conjunctive query on any database of size N with arbitrary pre-processing time and $O(N^{1/2-\gamma})$ delay and update time for any $\gamma > 0$ [5].

[2] This conjecture states that for every $\gamma > 0$ and every (full or Boolean) conjunctive query Q, there does not exist a combinatorial algorithm that for any database of size N can answer Q in time $O(N^{\mathsf{subw}(Q)-\gamma} + \mathsf{OUT})$, where $\mathsf{subw}(Q)$ is the submodular width of Q and OUT is the query output size [1]. This conjecture in the static query evaluation setting implies that there is no fully dynamic algorithm that can maintain Q with amortized update time $O(N^{\mathsf{subw}(\widehat{Q})-1-\gamma})$ and constant enumeration delay for any $\gamma > 0$, where $\widehat{Q}$ is the multivariate extension of Q [1].

[3] Using fast-matrix multiplication, a recently proposed non-combinatorial IVM algorithm [3] can achieve $O(N^{2/3-\gamma})$ for the 4-cycle query, where $\gamma = 0.009811$.

by $|R|$, is the number of tuples $\boldsymbol{x}$ for which $R(\boldsymbol{x}) > 0$. We specify queries using a syntax similar to that of functional aggregate queries over the $(\mathbb{Z}, +, \cdot, 0, 1)$ ring [21]:

$$Q(\boldsymbol{F}) \;=\; \sum_{\boldsymbol{B}} R_1(\boldsymbol{X_1}) \cdot \ldots \cdot R_k(\boldsymbol{X_k}), \tag{1}$$

where $\sum$ and $(\cdot)$ are the summation and respectively multiplication operation from the ring, $R_1, \ldots, R_k$ are *relation symbols*, each $\boldsymbol{X_i}$ is a schema, and each $R_i(\boldsymbol{X_i})$ is an *atom* of Q. We assume distinct relation symbols in a query. If several relation symbols correspond to the same physical database relation, which happens in case Q has self-joins, then we assume without loss of generality (i.e., without changes in the data complexities stated in the paper) that each such atom gets its own copy of the database relation. The set of variables of Q is $\mathsf{vars}(Q) \stackrel{\text{def}}{=} \bigcup_{i\in[k]} \boldsymbol{X_i}$. The *free* variables of Q are $\boldsymbol{F} \subseteq \mathsf{vars}(Q)$, while $\mathsf{vars}(Q) \setminus \boldsymbol{F}$ are the *bound variables*. If $\mathsf{vars}(Q) = \boldsymbol{F}$, then Q is a full (or join) query. By $\mathsf{at}(Q)$ and $\mathsf{at}(Y)$ we denote the set of all atoms of Q and the set of all atoms $R_i(\boldsymbol{X_i})$ with $Y \in \boldsymbol{X_i}$, respectively. For compactness, we write a set of variables as the concatenation of their names, e.g., $\{X, Y, Z\}$ becomes XYZ while $\{X\}$ becomes X. Each variable $A \in \boldsymbol{X}_i \cap \boldsymbol{X}_j$ expresses an equi-join between R_i and R_j, for $i \neq j$, and is called a *join variable*. Let $\mathcal{J}_Q$ be the tuple of all join variables in Q, ordered using a fixed total order on $\mathsf{vars}(Q)$.

For any variable $Y \in \mathsf{vars}(Q)$, $\mathsf{Dom}(Y)$ is defined as: $\mathsf{Dom}(Y) \stackrel{\text{def}}{=} \bigcup_{R_i(\boldsymbol{X}_i)\in\mathsf{at}(Y)} \pi_Y R_i$, where π is the standard projection operator in relational algebra. The marginalization of variables $\boldsymbol{Y} \subseteq \boldsymbol{X}$ from a relation R over variables $\boldsymbol{X}$, denoted by $S(\boldsymbol{Z}) = \sum_{\boldsymbol{Y}} R(\boldsymbol{X})$ for $\boldsymbol{Z} = \boldsymbol{X} \setminus \boldsymbol{Y}$, is defined by: $\forall \boldsymbol{z} \in \mathsf{Dom}(\boldsymbol{Z}) : S(\boldsymbol{z}) \stackrel{\text{def}}{=} \sum\{R(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathsf{Dom}(\boldsymbol{X}) \wedge \boldsymbol{z} = \boldsymbol{x}.\boldsymbol{Z}\}$, where $\boldsymbol{x}.\boldsymbol{Z}$ is the restriction of the tuple $\boldsymbol{x}$ to the values of the variables in schema $\boldsymbol{Z}$. The union of two relations R and S over the same variable set $\boldsymbol{X}$, denoted by $T = R \cup S$, is defined as: $\forall \boldsymbol{x} \in \mathsf{Dom}(\boldsymbol{X}) : T(\boldsymbol{x}) = R(\boldsymbol{x}) + S(\boldsymbol{x})$. The query Q in Eq. (1) defines the relation: $\forall \boldsymbol{f} \in \mathsf{Dom}(\boldsymbol{F}) : Q(\boldsymbol{f}) = \sum_{\boldsymbol{x}\in\mathsf{Dom}(\mathsf{vars}(Q)):\boldsymbol{f}=\boldsymbol{x}.\boldsymbol{F}} R_1(\boldsymbol{x}.\boldsymbol{X}_1)\cdot\ldots\cdot R_k(\boldsymbol{x}.\boldsymbol{X}_k)$.

Data Updates. Following prior work [17], we model database *updates* as a sequence of single-tuple inserts and deletes. The insert (delete) of a tuple $\boldsymbol{x}$ into (from) a relation R is expressed as a delta relation δR that maps $\boldsymbol{x}$ to 1 (and -1, respectively). The updated relation is the union of the old relation and the delta relation: $R := R \cup \delta R$. A delete, whose effect is a tuple with negative multiplicity in the updated relation, is rejected. Updates are defined for joins of relations using the classical delta rule: $\delta(V_1(\boldsymbol{Z}_1) \cdot V_2(\boldsymbol{Z}_2)) = (\delta V_1(\boldsymbol{Z}_1) \cdot V_2(\boldsymbol{Z}_2)) \cup (V_1(\boldsymbol{Z}_1) \cdot \delta V_2(\boldsymbol{Z}_2)) \cup (\delta V_1(\boldsymbol{Z}_1) \cdot \delta V_2(\boldsymbol{Z}_2))$. This generalizes to a join of arbitrary relations by taking V_2 to be the join of relations and applying recursively the delta rule to δV_2. If only V_1 is changed, then $\delta V_2 = \emptyset$ and $\delta(V_1(\boldsymbol{Z}_1) \cdot V_2(\boldsymbol{Z}_2)) = \delta V_1(\boldsymbol{Z}_1) \cdot V_2(\boldsymbol{Z}_2)$. Updates commute with variable marginalization: $\delta(\sum_Y V) = \sum_Y \delta V$. If a database relation has several copies due to our assumption on distinct relation symbols, then each of these copies needs to be updated and triggers a delta query.

Delta View Trees. Our maintenance approach is supported by a tree of materialized views.

Definition 1 ((Delta) View Tree)**.** *A* view tree *T for a query Q is a rooted tree with the properties:*

- *There is a one-to-one mapping between the leaves of T and the atoms of Q.*
- *Each inner node is a view over some variables of Q.*
- *If a node $V'(\boldsymbol{Y})$ has a single child node $V(\boldsymbol{X})$, then it is a* projection view *defined by marginalizing variables of $V(\boldsymbol{X})$, i.e., $V'(\boldsymbol{Y}) = \sum_{\boldsymbol{X}\setminus\boldsymbol{Y}} V(\boldsymbol{X})$. Furthermore, every atom of Q with a variable from $\boldsymbol{X} \setminus \boldsymbol{Y}$ occurs in the subtree rooted at $V(\boldsymbol{X})$.*
- *If a node $V(\boldsymbol{X})$ has several children $V_1(\boldsymbol{X_1}), \ldots, V_1(\boldsymbol{X_n})$, for $n \geq 2$, then it is a* join view *defined by the natural join of the child views, i.e., $V(\boldsymbol{X}) = V_1(\boldsymbol{X_1}) \cdot \ldots \cdot V_k(\boldsymbol{X_n})$.*

Under an update to a relation R, the view tree T becomes a delta view tree*, denoted by δT_R, where R is replaced by δR, and each view V along the path from δR to the root view is replaced by δV.*

For a view tree T, an update to a relation R, and a view $V(\boldsymbol{X}) \in T$, let $\mathsf{leaves}(\delta V(\boldsymbol{X}), \delta T_R)$ denote the set of all leaves of the subtree of δT_R rooted at $\delta V(\boldsymbol{X})$. When δT_R is clear from context, it is omitted. We use $\mathcal{T}(Q)$ to denote the set of all view trees of Q. Fig. 1 depicts view trees for the 4-cycle query.

A procedure enumerates the query output with *constant delay* if the time is constant between: (i) the start of the enumeration process and the output of the first tuple; (ii) outputting any two consecutive tuples; and (iii) outputting the last tuple and the end of the enumeration process [8]. Any view tree for a join query allows for the constant-delay enumeration of the query output [17].

Heavy-Light Data Partitioning. For a join variable Y, we partition the Y-values in the database in *light* and *heavy* according to a *threshold parameter* $\epsilon \in [0, 1]$ and database size N:

$$Light(Y) := \{y \in \mathsf{Dom}(Y) \mid \sum_{R_i(\boldsymbol{X_i}) \in \mathsf{at}(Y)} |\sigma_{Y=y} R_i| \leq N^\epsilon\} \qquad Heavy(Y) := \mathsf{Dom}(Y) \setminus Light(Y).$$

That is, a light Y-value y occurs in at most N^ϵ tuples across all relations, while there are at most $N^{1-\epsilon}$ heavy Y-values. For an atom $R(Y, \boldsymbol{Z})$, the relation R is the disjoint union of its fragment where Y is light and its fragment where Y is heavy.

Definition 2 (Degree Configuration)**.** *Given a query Q with the tuple of join variables $(Y_1, \ldots, Y_n)$, a* degree configuration *is a tuple $(d_1, \ldots, d_n)$, where $d_i = L$ in case Y_i is light and $d_i = H$ in case Y_i is heavy, for $i \in [n]$.*

There are 2^n degree configurations for a query Q with n join variables. We denote by $\mathcal{D}(Q)$ the set of all such degree configurations.

Definition 3 (Relation Restriction)**.** *Given a query Q with the tuple of join variables $(Y_1, \ldots, Y_n)$, degree configuration $\boldsymbol{d} = (d_1, ..., d_n)$, and atom $R(\boldsymbol{X})$ in Q, the* $\boldsymbol{d}$-restriction *of relation R is:*

$$\{\boldsymbol{r} \mid \boldsymbol{r} \in R \wedge \forall i \in [n] : Y_i \in \boldsymbol{X} \rightarrow (d_i = L \wedge \boldsymbol{r}.Y_i \in Light(Y_i) \vee d_i = H \wedge \boldsymbol{r}.Y_i \in Heavy(Y_i))\}.$$

Degree Constraints. Using the degree configurations and data updates, we can derive constraints on the degrees for the values in the database.

Definition 4 (Degree Constraint)**.** *[26, Def. 1] A* degree constraint *on a database of size N is a tuple $(\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}})$, where $\boldsymbol{Y} \subsetneq \boldsymbol{Z}$ and $p_{Z|Y} \in \mathbb{Q}_{\geq 0}$. Let Q be a query and $R_i(\boldsymbol{X}_i)$ an atom of Q. Then $R_i(\boldsymbol{X}_i)$* guards *the degree constraint $(\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}})$ if $\boldsymbol{Z} \subseteq \boldsymbol{X}_i$ and $\max_{\boldsymbol{t}} |\pi_{\boldsymbol{Z}}(\sigma_{\boldsymbol{Y}=\boldsymbol{t}.\boldsymbol{Y}} R_i)| \leq N^{p_{Z|Y}}$.*

We denote the set of all variables appearing in a set C of degree constraints by $\mathsf{vars}(C) \stackrel{\text{def}}{=} \bigcup_{(\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}}) \in C} \boldsymbol{Z}$. A *projection* of a set C of degree constraints onto a set $\boldsymbol{V}$ of variables, denoted by $C[\boldsymbol{V}]$, is defined as the set $\{(\boldsymbol{Z} \cap \boldsymbol{V}|\boldsymbol{Y}, N^{p_{Z|Y}}) \mid (\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}}) \in C \wedge \boldsymbol{Y} \subsetneq \boldsymbol{Z} \cap \boldsymbol{V}\}$. Projections may not only shrink the set of variables covered by a degree constraint ($\boldsymbol{Z} \cap \boldsymbol{V}$), but also remove a constraint from C altogether. This happens when $\boldsymbol{Y} \subsetneq \boldsymbol{Z} \cap \boldsymbol{V}$ is violated. For instance, the projection $(\boldsymbol{Y}|\boldsymbol{Y}, N^{p_{Z|Y}})$ of the constraint $(\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}})$ onto $\boldsymbol{Y}$ is uninformative and therefore discarded.

For a database of size N, delta view $\delta V(\boldsymbol{X})$ in a delta view tree T_{R_j} for an update δR_j, and degree configuration $\boldsymbol{d}$, we define the set of degree constraints that are guarded by the database relations at the leaves $\mathcal{L} = \mathsf{leaves}(\delta V(\boldsymbol{X}))$ of $\delta V(\boldsymbol{X})$ in T_{R_j} as:

$$\begin{aligned}
\mathrm{DC}(\mathcal{L}, \boldsymbol{d}) \stackrel{\text{def}}{=} &\{(\boldsymbol{X}_i|\emptyset, N) \mid R_i(\boldsymbol{X}_i) \in \mathcal{L} \wedge i \neq j\} \cup && \text{(size constr.)}\\
&\{(\boldsymbol{X}_i|Y, N^\epsilon) \mid R_i(\boldsymbol{X}_i) \in \mathcal{L} \wedge Y \in \boldsymbol{X}_i \wedge Y \text{ is light in } \boldsymbol{d} \wedge i \neq j\} \cup && \text{(light constr.)}\\
&\{(Y|\emptyset, N^{1-\epsilon}) \mid R_i(\boldsymbol{X}_i) \in \mathcal{L} \wedge Y \in \boldsymbol{X}_i \wedge Y \text{ is heavy in } \boldsymbol{d} \wedge i \neq j\} \cup && \text{(heavy constr.)}\\
&\{(A|\emptyset, 1) \mid A \in \boldsymbol{X}_j\} && \text{(update constr.)}
\end{aligned}$$

Note that this set does not contain constraints that are guarded by the relation R_j itself, since this relation is not in δT_{R_j}. If a set C of degree constraints is guarded by a relation, then this also holds for its projection $C[X]$ onto any set of variables $X \subseteq \text{vars}(C)$.

Definition 5 (Acyclic Sets of Degree Constraints)**.** *For any set C of degree constraints, associate a directed graph G_C with a node for every variable in* $\text{vars}(C)$ *and with a directed edge $(y, z) \in Y \times (Z - Y)$ for every degree constraint $(Z \mid Y, N^{p_{Z|Y}}) \in C$. If G_C is acyclic, then C is called* acyclic*. We denote by $\mathcal{A}(C)$ the set of all maximal acyclic subsets of C.*

All maximal acyclic subsets of a set C of degree constraints contain all size, heavy, and update constraints from C, as these constraints have the form $(Z|\emptyset, N^{p_{Z|\emptyset}})$ and do not create edges in the constraint graph. It is only the light constraints that create edges and therefore cycles in this graph.

RAM Model of Computation. We assume that each relation R_i is implemented by a data structure of size $O(|R_i|)$ that can: (i) look up, insert, and delete tuples in R_i in amortized constant time, and (ii) enumerate all tuples in R_i with constant delay. For a set $S \subsetneq X_i$, we use an index data structure that, for any tuple x_S over the variables in S, (iii) can enumerate all tuples in $\sigma_{S=x_S} R_i$ with constant delay, and (iv) insert and delete index entries in amortized constant time. We also need indices to (v) enumerate with constant delay the distinct tuples in each relation constructed by marginalizing any subset of variables of R_i. We report the time complexity as a function of the database size N only, where the query is considered fixed and of constant size (data complexity). Therefore, the enumeration delay is constant when it does not depend on the database size.

3 Overview of Our Adaptive IVM approach

In this section, we overview our maintenance approach. Given a join query Q, a database of size N, and a single-tuple update (tuple insert or delete), our approach updates the query output and allows for constant-delay enumeration of the tuples in the query output. Our approach is adaptive: For different heavy-light partitioning of the database on the columns corresponding to the join variables in Q (as described in Sec. 2), it may use a different maintenance strategy.

Example 6. *We use as running example throughout this section the 4-cycle query:*

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

The query has 4 join variables, $2^4 = 16$ degree configurations. For instance, the degree configuration $\boldsymbol{d} = (L, L, H, H)$ for the tuple of join variables (A, B, C, D) corresponds to the relation restrictions where A and B are light while C and D are heavy. Under an update δR, the delta query is: $\delta Q(A, B, C, D) = \delta R(A, B) \cdot S(B, C) \cdot T(C, D) \cdot U(D, A)$. For the degree configuration $\boldsymbol{d}$, threshold parameter ϵ, and database size N, we have the following degree constraints:

$$\begin{aligned}\text{DC}(\text{at}(\delta Q), \boldsymbol{d}) = \{&(BC|\emptyset, N), (CD|\emptyset, N), (AD|\emptyset, N), (AD|A, N^{\epsilon}), (BC|B, N^{\epsilon}),\\ &(C|\emptyset, N^{1-\epsilon}), (D|\emptyset, N^{1-\epsilon}), (A|\emptyset, 1), (B|\emptyset, 1)\}.\end{aligned}$$

The first three constraints are size constraints (relations S, T, and U have sizes at most N). The next two constraints are light constraints, e.g., there are at most N^{ϵ} D-values for a given A-value. The first two constraints in the second line are heavy constraints, e.g., there are at most $N^{1-\epsilon}$ C-values, while the last two constraints express that each of A and B is set to one value (due to the update δR).

We maintain the output of a join query using trees of materialized views (Def. 1).

Example 7. *Fig. 1 depicts six possible view trees for the 4-cycle query. The leaves are atoms that correspond to the four relations, while the intermediate nodes are join or projection views.*

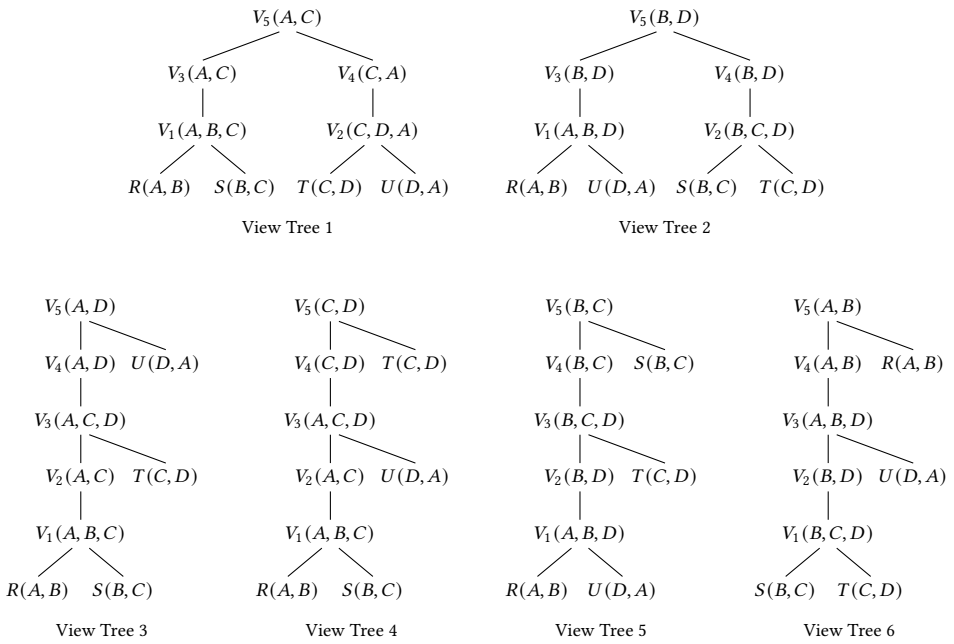


Fig. 1. The six view trees used to maintain the 4-cycle query.

Each view tree admits a simple maintenance mechanism [13, 17]. Given an update to a relation, all views along the path from the leaf corresponding to the updated relation to the root may be affected by the update and we compute deltas for them. We refer to the modified tree, where the updated relation and the views along the path to the root are replaced by their deltas, as the delta view tree. The enumeration of the tuples in the query output proceeds top-down in the view tree and needs constant delay per tuple.

Example 8. *An update $\delta R : \{(a,b) \mapsto m\}$ to relation R, where $m = +1$ for an insert and $m = -1$ for a delete, in the first view tree in Fig. 1 triggers the computation of updates for the views along the path from the leaf R to the root of the view tree:*

$$\begin{aligned} \delta V_1(A,B,C) &= \delta R(A,B) \cdot S(B,C) & V_1 &:= V_1 \cup \delta V_1 \\ \delta V_3(A,C) &= \sum_B \delta V_1(A,B,C) & V_3 &:= V_3 \cup \delta V_3 \\ \delta V_5(A,C) &= \delta V_3(A,C) \cdot V_4(C,A) & V_5 &:= V_5 \cup \delta V_5 \end{aligned}$$

This bottom-up propagation of the updates ensures that the views are calibrated top-down. For instance, all pairs (a,c) in V_5 are also in all views and relations below V_5; furthermore, the B-values (D-values) paired with (a,c) in V_1 (V_2) are also in R and S (respectively T and U). Consequently, the tuples in the query output can be enumerated with constant delay. We enumerate with constant delay: the pairs (a,c) in V_5, the B-values paired with (a,c) in V_1 and the D-values paired with (a,c) in V_2.

View trees (and variants thereof) have been previously used by several IVM systems [5, 12, 13, 17, 23, 30]. For instance, F-IVM [17] compiles a given query into one view tree, which is then maintained under updates as shown in Ex. 8. As discussed in the introduction, the update time achieved by these approaches for arbitrary queries can be suboptimal.

Our approach can use several view trees. It considers a heavy-light partitioning of the data and can use different view trees for different degree configurations. This adaptivity can lead to lower update times than when using a single view tree. The challenge brought by adaptivity is to algorithmically find (1) an asymptotically best view tree for each degree configuration and (2) the heavy-light threshold parameter ϵ that minimizes the update time for any given query. We address this challenge as follows.

In our approach, the update time is a function of the threshold parameter ϵ. In particular, given a delta view tree and a degree configuration, each delta view is computed under the degree constraints parameterized by ϵ. By parameterization, we mean that for a degree constraint $(\boldsymbol{Z}|\boldsymbol{Y}, N^{p_{Z|Y}})$, the exponent $p_{Z|Y}$ is a linear function of ϵ, which evaluates to a positive rational number for a given value for ϵ. The compute time for a delta view is given by $O(N^s)$, where N is the database size and s is the optimal solution of a linear program that computes the polymatroid bound under degree constraints [22, 26]. This polymatroid bound is a generalization of the well-known AGM bound [4] from size constraints to more general degree constraints that also include the light constraints. The linear program of such bounds assigns a positive weight to each constraint such that for each query variable, the sum of the weights of the constraints that cover the variable is at least one. The objective is to minimize the sum of all weights, where each weight is multiplied by the exponent $p_{Z|Y}$ in the corresponding constraint. A key observation is that the inequalities of the linear program do not depend on ϵ. Therefore, the set of vertices of the polyhedron given by the feasible region of the linear program are independent of ϵ. This also means that the optimal solution, which is given by one of these vertices, can be expressed as the minimum over all vertices of the objective instantiated for each vertex.

Example 9. *Let us consider View Tree 4 (Fig. 1), the degree configuration $\boldsymbol{d} = (L, L, H, H)$ for join variables (A, B, C, D), and an update δR. This update triggers updates to the views along the path from R to the root of the view tree. We first consider $\delta V_1(A, B, C) = \delta R(A, B) \cdot S(B, C)$. The degree constraints that hold at the leaves of $\delta V_1(A, B, C)$ in the delta view tree are:*

$$C_1 = \mathrm{DC}(\mathsf{leaves}(\delta V_1(A, B, C)), \boldsymbol{d}) = \{(BC|\emptyset, N), (BC|B, N^{\epsilon}), (C|\emptyset, N^{1-\epsilon}), (A|\emptyset, 1), (B|\emptyset, 1)\}.$$

We obtain the exponent s of an upper bound $O(N^s)$ on the time to compute $\delta V_1(A, B, C)$ using the following linear program that assigns a weight (positive number) to each constraint. We have the following vector of weights: $\boldsymbol{w} = (w_1, w_2, w_3, w_4, w_5)$. We assume that the order of the weights follows the order of the above constraints. The linear program is as follows:

$$\begin{aligned}
&\textit{minimize} && w_1 \cdot 1 + w_2 \cdot \epsilon + w_3 \cdot (1 - \epsilon) + w_4 \cdot 0 + w_5 \cdot 0 && \\
&\textit{subject to} && w_4 \geq 1 && \textit{// covers } A \\
& && w_1 + w_5 \geq 1 && \textit{// covers } B \\
& && w_1 + w_2 + w_3 \geq 1 && \textit{// covers } C \\
& && w_1, \ldots, w_5 \geq 0 &&
\end{aligned}$$

The objective is the sum of all weights, each multiplied by the base-N logarithm of the bound in the corresponding constraint; note the coefficient of w_4 and w_5 is $\log_N 1 = 0$, so these weights do not contribute to the objective. The program has one inequality per variable: the sum of the weights of those constraints that cover the variable must be at least 1. For the optimal solution, it is enough to consider the weight vectors that are the vertices of the convex polyhedron given by the feasible region of the above linear program: $(1, 0, 0, 1, 0), (0, 1, 0, 1, 1), (0, 0, 1, 1, 1)$.

None of these vertices depend on ϵ, yet the program solutions may be parameterized by ϵ. For instance, the solution given by the 3rd vertex, which sets $w_3 = w_4 = w_5 = 1$ and all other weights to 0, is $1 - \epsilon$, whereas the solution given by the 2nd vertex, which sets $w_2 = w_4 = w_5 = 1$ and all other

weights to 0, is ϵ. The minimal solution is therefore at most $\min(\epsilon, 1-\epsilon)$. *The delta view δV_1 is a join query that can be computed using an adaptation of a worst-case optimal join algorithm [26] in time* $O(N^{\min(\epsilon,1-\epsilon)})$.

We next consider $\delta V_2(A,C) = \sum_B \delta V_1(A,B,C)$. The time to compute δV_2 is asymptotically the same as for δV_1, since the former can be computed in one pass over the latter. In the paper, we introduce a systematic approach to upper bound the time to compute conjunctive queries with bound variables, such as δV_2. For this, we consider the set of constraints that hold at the leaves of δV_2, which is C_1, projected onto different supersets of the set of its free variables $\{A,C\}$, i.e., $C_2 = C_1[AC] = \{(C|\emptyset, N), (C|\emptyset, N^{1-\epsilon}), (A|\emptyset, 1)\}$ and $C_1 = C_1[ABC]$. We next discuss each of these two cases.

Using C_2, we can define the join query Q_{C_2} that over-approximates δV_2 in the sense that all tuples in δV_2 are also in Q_{C_2} (multiplicities are ignored): $Q_{C_2}(A,C) = \delta R'(A) \cdot S'(C)$, where $\delta R'$ is the projection of δR onto A and S' is the projection of S onto C (Def. 13). The query Q_{C_2} can be computed in time $O(N^{1-\epsilon})$: C_2 states that we have one A-value and at most $N^{1-\epsilon}$ C-values. The output of δV_2 can be recovered by semi-join reducing Q_{C_2} with δV_1. The time to compute δV_2 is therefore asymptotically the same as for Q_{C_2}.

In case we use C_1, we retain B and with it the light constraint and the update constraint on B. The join query defined by C_1 is precisely δV_1, which is an over-approximation of δV_2 in the sense that we can recover the tuples of δV_2 from δV_1 in the same $O(N^{\min(\epsilon,1-\epsilon)})$ time as for computing δV_1.

We next consider $\delta V_3(A,C,D) = \delta V_2(A,C) \cdot U(D,A)$. We take the set $C_3 = \mathrm{DC}(\mathrm{leaves}(\delta V_3), \boldsymbol{d})$ of constraints that hold at the leaves of δV_3 and project it onto the two possible supersets of the set of its free variables: $C_3' = C_3[ACD] = \{(C|\emptyset, N), (C|\emptyset, N^{1-\epsilon}), (A|\emptyset, 1), (D|\emptyset, N^{1-\epsilon}), (AD|\emptyset, N), (AD|A, N^{\epsilon})\}$ and $C_3'' = C_3[ABCD] = C_1 \cup \{(D|\emptyset, N^{1-\epsilon}), (AD|\emptyset, N), (AD|A, N^{\epsilon})\}$.

In case of C_3', we use the join query $Q_{C_3'}$ that over-approximates δV_3: $Q_{C_3'}(A,C,D) = \delta R'(A) \cdot S'(C) \cdot U(D,A)$. We use that: There is one A-value that is also light, D is heavy, and C is heavy. This yields a $N^{\min(\epsilon,1-\epsilon)}$ bound on the number of D-values. The time is then $O(N^{\min(\epsilon,1-\epsilon)+1-\epsilon}) = O(N^{\min(1,2-2\epsilon)})$.

In case of C_3'', we use the join query $Q_{C_3''}(A,B,C,D) = \delta R(A,B) \cdot S(B,C) \cdot U(D,A)$. This query over-approximates δV_3 in the sense that for every tuple $\boldsymbol{t}$ in δV_3 there is a tuple $\boldsymbol{t'}$ in $Q_{C_3'}$ such that $\boldsymbol{t'}.(ACD) = \boldsymbol{t}$. To compute this query, we observe that the number of C-values or of D-values is upper bounded by $N^{\min(\epsilon,1-\epsilon)}$, since both C and D are heavy and there is one A-value and one B-value and both values are light. This yields the compute time $O(N^{2\min(\epsilon,1-\epsilon)})$. The analysis follows similarly for $\delta V_4(C,D) = \sum_A \delta V_3(A,C,D)$ and $\delta V_5(C,D) = \delta V_4(C,D) \cdot T(C,D)$.

Our approach needs to account for all such evaluation strategies for each delta view, as their time may depend on ϵ and it is only clear which ones support the lowest overall update time once we find the value of ϵ that minimizes the update time across all degree configurations and relation updates.

The update time for a view tree and degree configuration is given by the maximum time to compute any delta view in the view tree. For a degree configuration, the update time is the minimum update time over all possible view trees. The overall update time is the maximum update time over all degree configurations. We call the base-N logarithm of the overall update time for a query Q the *maintenance width* of Q and denote it by $\mathrm{mw}(Q)$ (Def. 20). This width is given by a nesting of minimizations and maximizations of linear functions in the threshold parameter ϵ. Its minimal value, and ϵ that gives its minimal value, can be computed using a set of linear programs (Sec. 4.4).

Example 10. *Table 2 gives the base-N logarithm of the update time for each degree configuration, when using the best view tree for that configuration. Each of the six view trees from Fig. 1 is used by at least one configuration. No further view trees are needed to achieve the overall lowest update time. The maintenance width* $\mathrm{mw}(Q)$ *of the 4-cycle query Q is then the minimum over all expressions in the table column for the base-N logarithm of the update time (we ignore wlog the expressions for all remaining view trees as they do not yield a smaller maintenance width). This width can be computed*

Configuration (A, B, C, D)	***View Tree***	$\log_N$ ***UpdateTime***	***Configuration*** (A, B, C, D)	***View Tree***	$\log_N$ ***UpdateTime***
$*, L, *, L$	1	ϵ	H, L, L, H	3	$f(\epsilon)$
L, L, L, H	2	ϵ	H, L, H, H	1	$1-\epsilon$
L, L, H, H	4	$f(\epsilon)$	H, H, L, L	6	$f(\epsilon)$
$L, H, L, *$	2	$f(\epsilon)$	H, H, L, H	2	$1-\epsilon$
L, H, H, L	5	$f(\epsilon)$	$H, H, H, *$	1	$1-\epsilon$
L, H, H, H	2	$1-\epsilon$			

Table 2. The base-N logarithm of the update time for each degree configuration and a specific view tree from Fig. 1. Here, $f(\epsilon) \stackrel{\text{def}}{=} \max(\min(2\epsilon, 2-2\epsilon)), \min(2\epsilon, 1))$; $(*)$ in the degree configuration indicates that the join variable can be either heavy or light.

as the minimization of optimal solutions of linear programs:

$$
\begin{aligned}
\mathrm{mw}(Q) &= \min_{\epsilon} \max(\epsilon, 1-\epsilon, \max(\min(2\epsilon, 2-2\epsilon), \min(2\epsilon, 1))) \\
&= \min_{\epsilon} \max(\epsilon, 1-\epsilon, \min(2\epsilon, 2-2\epsilon), \min(2\epsilon, 1)) \\
&\stackrel{*}{=} \min_{\epsilon} \min(\max(\epsilon, 1-\epsilon, 2\epsilon, 2\epsilon), \max(\epsilon, 1-\epsilon, 2\epsilon, 1), \\
&\qquad\qquad \max(\epsilon, 1-\epsilon, 2-2\epsilon, 2\epsilon), \max(\epsilon, 1-\epsilon, 2-2\epsilon, 1)) \\
&\stackrel{+}{=} \min(\min_{\epsilon} \max(\epsilon, 1-\epsilon, 2\epsilon, 2\epsilon) \ \textit{s.t.}\ 0 \le \epsilon \le 1 \\
&\qquad \min_{\epsilon} \max(\epsilon, 1-\epsilon, 2-2\epsilon, 1) \ \textit{s.t.}\ 0 \le \epsilon \le 1 \\
&\qquad \min_{\epsilon} \max(\epsilon, 1-\epsilon, 2-2\epsilon, 2\epsilon) \ \textit{s.t.}\ 0 \le \epsilon \le 1 \\
&\qquad \min_{\epsilon} \max(\epsilon, 1-\epsilon, 2-2\epsilon, 1) \ \textit{s.t.}\ 0 \le \epsilon \le 1)
\end{aligned}
$$

The equality (*) holds due to the distributivity of max *over* min*, while the equality (+) is due to the commutativity of the two* min *functions. We then have to take the minimum of the optimal solutions of four optimization problems, which can be encoded as linear programs. We show the equivalent linear program for the first optimization problem above:*

$$
\min_{\epsilon} \ q \quad \textit{s.t.} \qquad q \ge \epsilon \qquad q \ge 1-\epsilon \qquad q \ge 2\epsilon \qquad 0 \le \epsilon \le 1.
$$

The optimal solution is $2/3$ *and obtained for* $\epsilon = 1/3$.

Once we know the value of ϵ*, we can decide which evaluation strategy is best for each delta view in each view tree and for each degree configuration.*

We are now ready to state the main technical result of this paper. Our incremental view maintenance approach follows the setting of prior work [14–16]. It has a preprocessing phase, in which the heavy-light partitioning happens and the views of the used view trees are materialized. Then, it receives a sequence of single-tuple updates (inserts and deletes) and processes one update at a time. After each update, it can resolve requests to enumerate the tuples in the query output with constant delay. After some updates, the heavy/light degree assignment of some values may become invalid as: (1) light (heavy) values become heavy (light) according to the current threshold N^ϵ; or (2) the threshold itself changes significantly as N changes [14]. In the first case, we need to move values between the light and heavy parts of relations; this is called minor rebalancing. In the

second case, we need to recompute the partitioning and then the views from scratch; this is called major rebalancing. Major and minor rebalancing only need to be performed after a sufficiently large number of updates so that the partitioning still guarantees the desired asymptotic complexity for the update time. By amortizing the cost of rebalancing over many updates, the update time remains the same, albeit amortized.

THEOREM 11. *Any join query Q can be maintained with $O(N^{1+\mathrm{mw}(Q)})$ preprocessing time, amortized $O(N^{\mathrm{mw}(Q)})$ single-tuple update time, and $O(1)$ enumeration delay, where N is the size of the database at the time of update and $\mathrm{mw}(Q)$ is the maintenance width of Q.*

4 The Maintenance Width

In this section, we introduce the *maintenance width* and show how to compute it. This measure is central to our approach as it is the exponent of the update time incurred by our approach for maintaining the query output under database updates.

Consider a view $V(\boldsymbol{X})$ defined over a database that satisfies a set of degree constraints. Given an update δR_i, we bound the time needed to compute the delta view δV using a formulation [22, 26] as a linear optimization problem that generalizes the well-known AGM bound [4] to incorporate the degree constraints.

Adapting this framework to our setting presents two challenges. First, the original algorithm [22, 26] is designed for join queries (all variables are free); we must extend it to support views, where some variables are marginalized out. Second, our degree constraints depend on the threshold parameter ϵ, which is not fixed in advance. Consequently, we cannot solve the bounding linear program numerically. Instead, we solve it symbolically to obtain an objective value expressed as an explicit function of ϵ. We then select the optimal ϵ that minimizes the worst-case maintenance cost across all degree configurations.

4.1 The Polymatroid Bound under Degree Constraints

In this section, we first revisit prior work on the polymatroid bound [26]. This bound is essential to our analysis of the maintenance time under different degree configurations and updates.

Definition 12 (Polymatroid Bound under Degree Constraints)**.** *[26, Eq. 48] Given a set $C = \{c_1, \ldots, c_m\}$ of degree constraints, we define the* Polymatroid Bound under Degree Constraints*, denoted by $\mathrm{PBD}(C)$, as the optimal value of the following linear program:*

$$\textit{minimize} \quad \sum_{c_i=(\boldsymbol{Z}|\boldsymbol{Y},N^{p_i})\in C} w_i \cdot p_i \tag{2}$$

$$\textit{subject to} \quad \sum_{\substack{c_i=(\boldsymbol{Z}|\boldsymbol{Y},N^{p_i})\in C \\ A\in\boldsymbol{Z}\setminus\boldsymbol{Y}}} w_i \geq 1 \qquad \forall A \in \mathrm{vars}(C) \tag{3}$$

$$w_i \geq 0 \qquad \forall c_i = (\boldsymbol{Z}|\boldsymbol{Y}, N^{p_i}) \in C \tag{4}$$

Prior work [26] showed that join queries can be computed worst-case optimally in the presence of acyclic degree constraints. In particular, given a join query Q and an acyclic set C of constraints, where each constraint is guarded by database relations used in Q, then Q can be evaluated in time $O\left(|\mathrm{vars}(Q)| \cdot |C| \cdot \left[N + N^{\mathrm{PBD}(C)}\right] \cdot \log N\right)$ [26]. The $\log N$ factor in the runtime is due to the use of B-tree indices and can be dropped by using hash maps to represent the relations. The additive $N \log N$ factor is due to pre-computation. Notice that Inequalities 3, 4 are trivially satisfied by setting all variables w_i to 1, so $\mathrm{PBD}(C)$ is always defined.

4.2 Upper Bounding the Time to Compute a Delta View

In our work we need to compute delta views that are not necessarily join queries. For this, we extend the prior work [26] to compute queries with arbitrary bound variables under degree constraints. We do this in several steps. First, we show how to derive possible join queries that over-approximate a given delta view. We tailor the set of constraints that are satisfied by the database to those relations and variables that are relevant to the over-approximation join queries. We then use the algorithm from prior work [26] to compute the over-approximation join queries under specific constraints and take the over-approximation with the lowest time complexity. Below, we make this plan concrete.

Definition 13 (Guarding Query)**.** *Let $\boldsymbol{d}$ be a degree configuration, δT_R be a delta view tree for an update δR_j, $\delta V(\boldsymbol{X})$ be a delta view in δT_{R_j}, and $C = \mathrm{DC}(\mathsf{leaves}(\delta V(\boldsymbol{X})), \boldsymbol{d})$. For any set $\boldsymbol{Y}$ of variables such that $\boldsymbol{X} \subseteq \boldsymbol{Y} \subseteq \mathsf{vars}(C)$ and any acyclic set $C' \in \mathcal{A}(C[\boldsymbol{Y}])$ of degree constraints, we define the C'-guarding query of δV as the join query:*

$$Q_{C'}(\boldsymbol{Y}) = R'_1(\boldsymbol{X}'_1) \cdot \ldots \cdot R'_k(\boldsymbol{X}'_k)$$

where $\{R_1(\boldsymbol{X}_1), \ldots, R_k(\boldsymbol{X}_k)\}$ is the set of atoms at the leaves of $\delta V(\boldsymbol{X})$ in δT_{R_j} that guard the constraints in C' and includes $\delta R_j(\boldsymbol{X}_j)$, and $R'_i(\boldsymbol{X}'_i) = \sum_{\boldsymbol{X}_i \setminus \boldsymbol{Y}} R_i(\boldsymbol{X}_i)$ and $\boldsymbol{X}'_i = \boldsymbol{X}_i \cap \boldsymbol{Y}$ for $i \in [k]$.

Ex. 9 gives guarding queries for several delta views.

Remark 14. *In Def. 13, C' is a maximal acyclic subset of $C[\boldsymbol{Y}]$. It contains all size, heavy, and update constraints from $C[\boldsymbol{Y}]$, since these constraints are of the form $(\boldsymbol{Z}|\emptyset, N^{p_{\boldsymbol{Z}|\emptyset}})$ and cannot be part of a cycle in the constraint graph. Therefore, C' may only miss some of the light constraints from $C[\boldsymbol{Y}]$. Yet the variables in these light constraints are also covered by a size constraint. The implication is twofold. First, the atoms that guard the constraints in C' are also those that guard the constraints in $C[\boldsymbol{Y}]$. Second, $\mathsf{vars}(C') = \mathsf{vars}(C[\boldsymbol{Y}])$.*

Example 15. *We discuss the computation of the delta view $\delta V_5(A, C)$ in the delta view tree based on View Tree 1 (Fig. 1) under the update δR and the degree configuration $\boldsymbol{d} = (L, L, \cdot, \cdot)$, so where the join variables A and B are light and where – for simplicity here – we do not partition on the remaining join variables C and D. The following degree constraints are guarded by the leaves of $\delta V_5(A, C)$:*

$$\begin{aligned} C &= \mathrm{DC}(\mathsf{leaves}(\delta V_5), \boldsymbol{d}) \\ &= \{(A|\emptyset, 1), (B|\emptyset, 1), (BC|B, N^\epsilon), (BC|\emptyset, N), (CD|\emptyset, N), (AD|A, N^\epsilon), (AD|\emptyset, N)\}. \end{aligned}$$

The C-guarding query is $Q_C(A, B, C, D) = \delta R(A, B) \cdot S(B, C) \cdot T(C, D) \cdot U(A, D)$ and its output can be computed in worst-case optimal time $O(N^{\mathrm{PBD}(C)})$ [26]. Since the two queries Q_C and δV_5 have the same body, it follows that if we project the output of $Q_C(A, B, C, D)$ onto the variables $\{A, C\}$ we obtain the output of δV_5. However, $\mathrm{PBD}(C)$ evaluates to 1 or 2ϵ (by using the first, second, and fifth constraints or by using the first, second, third, and sixth constraints to cover all variables) which gives the upper bounds $O(N)$ and $O(N^{2\epsilon})$. Depending on the value of ϵ, both of these bounds can be tight upper bounds on the time needed to compute the output of Q_C, but they are loose on the time needed to compute the output of δV_5. As we show in later examples in this section, the output of δV_5 can be computed in time $O(N^\epsilon)$. The reason is that the linear program for $\mathrm{PBD}(C)$ needs to cover all variables, including the bound variables B and D, and this increases the cost of its optimal solution.

Any guarding query of a delta view *over-approximates* the delta view in the sense that we can recover the output of a delta view from the output of any of its guarding queries.

Lemma 16. *For any delta view $\delta V(\boldsymbol{X})$ and any of its $C[\boldsymbol{Y}]$-guarding queries $Q_{C[\boldsymbol{Y}]}(\boldsymbol{Y})$, where C is the set of constraints that are guarded by the leaves of $\delta V(\boldsymbol{X})$ and $\boldsymbol{X} \subseteq \boldsymbol{Y}$, it holds: For any tuple $\boldsymbol{x}$ in the output of δV over any database, there is a tuple $\boldsymbol{y}$ in the output of $Q_{C[\boldsymbol{Y}]}$ over the same database such that $\boldsymbol{x} = \boldsymbol{y}.\boldsymbol{X}$.*

Proof. Let $\{R_1(\boldsymbol{X}_1), \ldots, R_k(\boldsymbol{X}_k)\} = \text{leaves}(\delta V(\boldsymbol{X}))$. The delta view is thus defined by

$$\delta V(\boldsymbol{X}) = \sum_{(\bigcup_{i\in[k]} \boldsymbol{X}_i)\setminus \boldsymbol{X}} R_1(\boldsymbol{X}_1) \cdot \ldots \cdot R_k(\boldsymbol{X}_k).$$

Let C be the set of constraints that are guarded by the database relations $R_1, \ldots, R_k$. Since C always contains the size constraints of these relations, it holds that $\text{vars}(C) = \bigcup_{i\in[k]} \boldsymbol{X}_i$. By definition of the $C[Y]$-guarding query $Q_{C[Y]}(\boldsymbol{Y})$ (Def. 13), we have that $\boldsymbol{X} \subseteq \boldsymbol{Y} \subseteq \text{vars}(C)$ and:

$$Q_{C[Y]}(\boldsymbol{Y}) = R'_1(\boldsymbol{X}'_1) \cdot \ldots \cdot R'_k(\boldsymbol{X}'_k), \text{ where } R'_i(\boldsymbol{X}'_i) = \sum_{\boldsymbol{X}_i\setminus \boldsymbol{Y}} R_i(\boldsymbol{X}_i) \text{ and } \boldsymbol{X}'_i = \boldsymbol{X}_i \cap \boldsymbol{Y} \text{ for } i \in [k].$$

The set of tuples in the output of δV is:

$$\left\{ \boldsymbol{t}.\boldsymbol{X} \mid \sum_{\boldsymbol{t}\in \text{Dom}(\text{vars}(C))} R_1(\boldsymbol{t}.\boldsymbol{X}_1) \cdot \ldots \cdot R_k(\boldsymbol{t}.\boldsymbol{X}_k) > 0 \right\}, \tag{5}$$

whereas the set of tuples in the output of $Q_{C[Y]}$ is:

$$\left\{ \boldsymbol{y} \in \text{Dom}(\boldsymbol{Y}) \mid R'_1(\boldsymbol{y}.\boldsymbol{X}'_1) \cdot \ldots \cdot R'_k(\boldsymbol{y}.\boldsymbol{X}'_k) > 0 \right\}. \tag{6}$$

If the output of δV is empty, then the statement of the lemma follows trivially. Otherwise, consider a tuple $\boldsymbol{x}$ in the output of δV given in Eq. (5). Since the sum in Eq. (5) is positive, it must contain at least one positive term. Thus, there exists a tuple $\boldsymbol{t} \in \text{Dom}(\text{vars}(C))$ with $\boldsymbol{t}.\boldsymbol{X} = \boldsymbol{x}$ such that $R_1(\boldsymbol{t}.\boldsymbol{X}_1), \ldots, R_k(\boldsymbol{t}.\boldsymbol{X}_k)$ are all positive. It then also follows that $R'_1(\boldsymbol{t}.\boldsymbol{X}'_1) \cdot \ldots \cdot R'_k(\boldsymbol{t}.\boldsymbol{X}'_k) > 0$, since $\boldsymbol{X}'_i \subseteq \boldsymbol{X}_i$ and R'_i is defined by marginalizing $\boldsymbol{X}_i \setminus \boldsymbol{Y}$ from R_i, for $i \in [k]$. We conclude that for any tuple $\boldsymbol{x}$ such that $\delta V(\boldsymbol{x}) > 0$ there exists at least one tuple $\boldsymbol{y} \in \text{Dom}(\boldsymbol{Y})$ with $\boldsymbol{y}.\boldsymbol{X} = \boldsymbol{x}$ such that $Q_{C[Y]}(\boldsymbol{y}) > 0$. □

Guarding queries for a delta view are join queries, so we can compute them efficiently under an acyclic set of degree constraints [26]. Furthermore, we can compute the delta view in time proportional to the time need to compute any of its guarding queries.

Lemma 17. *Given a database of size N, a delta view tree δT_R, a delta view $\delta V(\boldsymbol{X})$ in δT_R, and any of the C-guarding queries of $\delta V(\boldsymbol{X})$, where C is an acyclic set of constraints, then δV can be computed in time $O(N^{\text{PBD}(C)})$ given that the child views of $\delta V(\boldsymbol{X})$ in δT_R are already computed.*

Proof. We compute δV in two steps. In the first step, we compute the join query Q_C in time $O(N^{\text{PBD}(C)})$ using the algorithm from prior work [26]. Its output is, however, an over-approximation of the output of δV as stated in Lemma 16. In the second step, we first create a relation V' that is the projection of Q_C's output onto the set of variables of the child views of δV in T. We next semi-join reduce V' with each of the child views of δV. The multiplicity of each tuple $\boldsymbol{t}$ in V' becomes the product of the multiplicities of the tuples in the child views whose join make $\boldsymbol{t}$ in V'. By performing the computation of the delta views bottom-up in the delta view tree, we ensure that the correct multiplicities of the child views are computed before those of the parent views. Finally, we marginalize out all variables of V' except $\boldsymbol{X}$ to obtain the output of δV. The second step also takes time proportional to the size of Q_C's output, so in $O(N^{\text{PBD}(C)})$ time. □

There are two immediate implications of Lemma 17. First, it gives a maintenance strategy for a delta view tree and an update δR_j: We proceed bottom-up from δR_j and first compute its parent delta view, then the parent of the parent, and so on until the delta view at the root. Second, it gives an upper bound on the update time for a delta view: We can take the minimum of the computation time over all its guarding queries.

A question remains: Why should we consider all guarding queries of a delta view in order to upper bound the compute time for the delta view? Recall there is a guarding query for each set of variables that is a superset of the set of free variables of the delta view and a subset of the set of variables at the leaves of the delta view in the delta view tree. The key observation is that a guarding query with more variables does not necessarily have a higher computation time than another guarding query with less variables.

Example 18. *Returning to Ex. 15, let us consider an over-approximation of* $\delta V_5(A,C) = \sum_{B,D} \delta R(A,B) \cdot S(B,C) \cdot T(C,D) \cdot U(D,A)$. *Projecting the set* $\boldsymbol{C}$ *of degree constraints onto* $\{A,C\}$ *gives*

$$\boldsymbol{C}[AC] = \{(A|\emptyset,1), (C|\emptyset,N), (A|\emptyset,N)\}$$

with the $\boldsymbol{C}[AC]$*-guarding query* $Q_{\boldsymbol{C}[AC]}(A,C) = \delta R'(A) \cdot S'(C) \cdot T'(C) \cdot U'(A)$. *By Lemma 16, we know that this join query is an over-approximation of the conjunctive query* $\delta V_5(A,C)$. *We can bound the time needed to compute* $Q_{\boldsymbol{C}[AC]}(A,C)$ *by* $O(N^{\mathrm{PBD}(\boldsymbol{C}[AC])})$ *using Lemma 17. This bound is* $O(N)$ *and obtained by covering the variable* A *with the first constraint and the variable* C *with the second constraint. While correct, this bound is loose since it does not exploit the lightness information on* B.

The previous example highlights a trade-off: projecting the set $\boldsymbol{C}$ of constraints onto a subset of its variables guarantees the correctness of the bound, but it may discard useful constraints (like those conditioned on B). Although this requires covering more variables in the linear program, it allows us to retain more constraints in the projection. Thus, we need to look at all the subsets $\boldsymbol{X}'$ with $\boldsymbol{X} \subseteq \boldsymbol{X}' \subseteq \mathrm{vars}(\boldsymbol{C})$.

Example 19. *We continue Ex. 18 and now consider an over-approximation of* $\delta V_5(A,C)$ *by projecting the set* $\boldsymbol{C}$ *of degree constraints onto the superset* $\{A,B,C\}$. *This new projection preserves a light constraint conditioned on* B *and gives*

$$\boldsymbol{C}[ABC] = \{(A|\emptyset,1), (B|\emptyset,1), (BC|\emptyset,N), (C|\emptyset,N), (A|\emptyset,N), (BC|B,N^\epsilon)\}$$

By Lemma 16, we conclude that $Q_{\boldsymbol{C}[ABC]} = \delta R(A,B) \cdot S(B,C) \cdot T'(C) \cdot U'(A)$ *is an over-approximation of* $\delta V_5(A,C)$. *The time needed to compute* $Q_{\boldsymbol{C}[ABC]}$ *is bounded by* $O(N^{\mathrm{PBD}(\boldsymbol{C}[ABC])})$, *which is* $O(N^\epsilon)$ *(by covering* A,B,C *using the first two and the last constraints). This is tighter than* $O(N)$ *for all* $\epsilon < 1$.

Yet, if we would project $\boldsymbol{C}$ *onto the full set* $\{A,B,C,D\}$ *of variables, then we would regain the constraints on* D *at the price of having to cover* D *in the linear program. As shown in Ex. 15, this gives the upper bound of* $O(N^{\min(1,2\epsilon)})$, *which is also worse than* $O(N^\epsilon)$ *for all* $\epsilon \in (0,1)$.

4.3 Symbolic Optimization

As illustrated in Ex. 18, the degree constraints can depend on ϵ (e.g., N^ϵ), while the value of ϵ is not fixed in advance. Consequently, in the linear program of Def. 12, we treat the coefficients p_i as functions of ϵ in order to obtain an explicit closed form expression for the optimal value as a function of ϵ. All the degree constraints obtained by the function $\mathrm{DC}(\cdot,\cdot)$ have the form $c_i = (\boldsymbol{Z} \mid \boldsymbol{Y}, N^{p_i} = N^{f_i(\epsilon)})$, where f_i is either affine in ϵ (it is ϵ or $1-\epsilon$) or constant (it is 0 or 1). For each constraint c_i, we view the coefficient p_i as the function $f_i(\epsilon)$.

Crucially, while the *objective function* in Eq. (2) varies with ϵ, the *feasible region* $\mathcal{P}(\boldsymbol{C})$ defined by Eq. (3) and (4) depends only on the sets $\boldsymbol{C}$ and $\boldsymbol{X}$, and is independent of ϵ. It is a standard result in linear programming that, if an optimum exists, then it is attained at a vertex of the feasible polyhedron $\mathcal{P}(\boldsymbol{C})$ [28]. Since $\mathcal{P}(\boldsymbol{C})$ is defined by a fixed set of constraints, it has a finite set of vertices (and independent on the database size), which we denote by $\mathcal{S}_{\boldsymbol{C}}$.

The vertices $\boldsymbol{w} = (w_i)_{c_i \in \boldsymbol{C}} \in \mathcal{S}_{\boldsymbol{C}}$ are constant vectors independent of ϵ. We can compute these weight vectors $\boldsymbol{w}$ once and reuse them to evaluate the cost for any ϵ. For a specific vertex $\boldsymbol{w} \in \mathcal{S}_{\boldsymbol{C}}$

and parameter ϵ, the symbolic cost is given by:

$$\text{obj}(\boldsymbol{w}, C)(\epsilon) \stackrel{\text{def}}{=} \sum_{c_i \in C} w_i \cdot f_i(\epsilon).$$

We are now ready to introduce the notion of *maintenance width*, which captures the maintenance cost for a given query. In order to find the lowest maintenance cost for a query, we look at the degree configuration that induces the most expensive set of degree constraints, for which we pick the cheapest view tree. For this view tree, we pick the most expensive view to maintain under the most expensive update.[4]

Definition 20 (Maintenance Width)**.** *For a join query Q, the maintenance width of Q is*

$$\text{mw}(Q) \stackrel{def}{=} \min_{\epsilon \in [0,1]} \max_{\boldsymbol{d} \in \mathcal{D}(Q)} \min_{T \in \mathcal{T}(Q)} \max_{\substack{V(\boldsymbol{X}) \in T \\ R(\boldsymbol{Y}) \in \text{leaves}(V(\boldsymbol{X}), T)}} \min_{\substack{C \subseteq \text{DC}(\text{leaves}(\delta V(\boldsymbol{X}), \delta T_R), \boldsymbol{d}) \\ \boldsymbol{X} \subseteq \boldsymbol{X}' \subseteq \text{vars}(C) \\ C' \in \mathcal{A}(C[\boldsymbol{X}']) \\ \boldsymbol{w} \in \mathcal{S}_{C'}}} \text{obj}(\boldsymbol{w}, C')(\epsilon). \quad (7)$$

Remark 21. *Def. 20 states that for every degree configuration $\boldsymbol{d}$ and view tree T, we take the maximum compute time over all views V of all its delta views δV subject to updates at each of its leaves δR. This computes the maximum update time per view over updates at any of its leaves in δT_R and then takes the maximum over all views in the view tree. In contrast, our maintenance approach propagates each update δR from a leaf along the path to the root of the delta view tree δT_R for δR and takes the maximum time to compute the delta views that are the ancestors of δR in δT_R. Yet both ways to account for the maintenance time for a given view tree yield the same maximum time to update all the views of T under all updates at the leaves. This justifies the equivalent formulation in the definition.*

We explain how Lemma 17 implies that any join query Q can be maintained with update time $O(N^{\text{mw}(Q)})$ as stated in Theorem 11. Fix an $\epsilon \in [0, 1]$. Given a degree configuration $\boldsymbol{d}$ and a view tree T for Q, let

$$\text{PBD}^*(\boldsymbol{d}, T)(\epsilon) = \max_{V(\boldsymbol{X}) \in T} \max_{R(\boldsymbol{Y}) \in \text{leaves}(V(\boldsymbol{X}), T)} \text{PBD}^*(\boldsymbol{d}, \delta V(\boldsymbol{X}), \delta T_R)(\epsilon) \text{ and}$$

$$\text{PBD}^*(\boldsymbol{d}, \delta V(\boldsymbol{X}), \delta T_R)(\epsilon) = \min_{C \subseteq \text{DC}(\text{leaves}(\delta V(\boldsymbol{X}), \delta T_R), \boldsymbol{d})} \min_{\boldsymbol{X} \subseteq \boldsymbol{X}' \subseteq \text{vars}(C)} \min_{C' \in \mathcal{A}(C[\boldsymbol{X}'])} \min_{\boldsymbol{w} \in \mathcal{S}_{C'}} \text{obj}(\boldsymbol{w}, C')(\epsilon).$$

Consider an atom $R(\boldsymbol{Y})$ in a view tree T for Q and an update δR to a relation R in Q. To maintain T under δR, we derive the delta view tree δT_R, and compute bottom-up all delta views $\delta V(\boldsymbol{X})$ along the path from $R(\boldsymbol{Y})$ to the root of δT_R. By Lemma 17, each such delta view $\delta V(\boldsymbol{X})$ can be computed in time $O(N^{\text{PBD}^*(\boldsymbol{d}, \delta V(\boldsymbol{X}), \delta T_R)(\epsilon)})$. The view tree T can then be maintained in time $O(N^{\text{PBD}^*(\boldsymbol{d}, T)(\epsilon)})$ by taking the maximum over all views in T and for each such view V the maximum over all atoms R at the leaves of this view of the time to compute the δV under the update δR. By Def. 20, we conclude that Q can be maintained in time $O(N^{\text{mw}(Q)})$ under any update to its input relations.

4.4 Computing the Maintenance Width

The computability of the maintenance width relies on the observation that all domains in Def. 20, excluding the interval $[0, 1]$, are finite. Let $\mathcal{H}$ denote the set of affine functions $f(\epsilon) = \text{obj}(\boldsymbol{w}, C)(\epsilon)$ generated by the inner minimization over C and $\boldsymbol{w}$. The maintenance width can be viewed as the value of a logical expression involving finite min and max operations over functions in $\mathcal{H}$.

[4]Alternatively, the maintenance width can be seen as the outcome of a game where we choose ϵ and the view tree T to minimize cost, while an adversary chooses the data statistics $\boldsymbol{d}$, the update δR_i, and the specific view in T to maximize cost.

We rewrite the nested minimization and maximization steps into a canonical min-of-max form by iteratively using the distributivity of max over min: $\max(a, \min(b, c)) = \min(\max(a, b), \max(a, c))$. Thus, there exists a finite index set $\mathcal{M}$ and, for each $i \in \mathcal{M}$, a finite set of affine functions $\mathcal{F}_i \subseteq \mathcal{H}$ such that:

$$\mathrm{mw}(Q) = \min_{\epsilon \in [0,1]} \left(\min_{i \in \mathcal{M}} \left(\max_{f \in \mathcal{F}_i} f(\epsilon) \right) \right). \tag{8}$$

Since the min operator is commutative, we can swap the continuous minimization over ϵ with the discrete minimization over i. This yields:

$$\mathrm{mw}(Q) = \min_{i \in \mathcal{M}} \left(\min_{\epsilon \in [0,1]} \left(\max_{f \in \mathcal{F}_i} f(\epsilon) \right) \right). \tag{9}$$

Eq. (9) reduces the optimization problem to finding the minimum of $|\mathcal{M}|$ independent sub-problems. Each sub-problem aims to minimize the pointwise maximum of a finite set of affine functions over the unit interval. Consequently, the objective is a piecewise linear convex function of ϵ, which can be efficiently minimized by the following linear program:

$$\begin{aligned} \text{minimize} \quad & v \\ \text{subject to} \quad & v \geq f(\epsilon) \quad \forall f \in \mathcal{F}_i \\ & 0 \leq \epsilon \leq 1 \end{aligned}$$

By solving these linear programs, we obtain both the exact value of $\mathrm{mw}(Q)$ and the optimal parameter ϵ^* for which this value is obtained. Fixing ϵ^* determines the threshold for data partitioning. We then proceed to select the optimal view tree for each degree configuration $\boldsymbol{d}$ by choosing the view tree T that minimizes the maintenance width for this fixed ϵ^*. In case of ties, we deterministically select a canonical tree (e.g., in lexicographical order). We denote this selected tree by $T(\boldsymbol{d})$. The set of these selected trees constitutes the set of *active view trees* that we use in our maintenance algorithm. In practice, this set can be significantly smaller than the set of all possible view trees. For instance, we only need six view trees for the optimal maintenance of the 4-cycle query. Furthermore, for every update δR_j and view $V(\boldsymbol{X})$ we determine the optimal set $\boldsymbol{X}'$ of variable to use for computing a guarding query that is an over-approximation of δV under the update δR_j.

5 Comparison with Prior Width Measures and Maintenance Strategies

In this section, we compare the maintenance width with other common width measures and discuss our choice of view trees as the maintenance strategy of our approach.

5.1 Maintenance Width vs. Dynamic Width

The maintenance width generalizes the previously introduced notion of *dynamic width*, which defines the update time for maintaining queries under simple size constraints [18]. Any query Q can be maintained with $O(N^{\mathrm{dw}(Q)})$ update time, where $\mathrm{dw}(Q)$ denotes the dynamic with of Q. To the best of our knowledge, the dynamic width defines the best update time achieved by approaches that do not rely on heavy-light partitioning. The F-IVM column in Table 1 lists the update times for several queries, these times follow the dynamic width[5]. For any hierarchical query Q, we have $\mathrm{mw}(Q) = \mathrm{dw}(Q) = 0$. Hence, both our approach and F-IVM achieve $O(1)$ update time. For the bow tie query Q, we have $\mathrm{mw}(Q) = \mathrm{dw}(Q) = 1$, implying that both our approach and F-IVM achieve $O(N)$ update time. For all other queries in Table 1, we have $\mathrm{mw}(Q) < \mathrm{dw}(Q)$, which means that our approach outperforms F-IVM.

[5]Although the work on F-IVM [17] did not formally introduce the notion of dynamic width, the update time achieved by F-IVM for any query Q is of the form $O(N^{\mathrm{dw}(Q)})$, where $\mathrm{dw}(Q)$ denotes the dynamic width of Q. The notion of dynamic width was formally introduced in subsequent work [18].

Computing the dynamic width requires iterating over all view trees, then over the views in each view tree, and finally over the leaves under each view. For the maintenance width, we must further iterate over all degree configurations, which accounts for the increased complexity of the definition, yet which may yield a smaller width value. In the following, we introduce the dynamic width dw and show that $\mathrm{mw}(Q)$ is upper-bounded by $\mathrm{dw}(Q)$ for any join query Q.

We start by recalling the *fractional edge cover number* of a set of variables with regard to a query [4]. Given a query Q and a set $Y \subseteq \mathrm{vars}(Q)$ of variables, the fractional edge cover number $\rho^*_Q(Y)$ of Y with regard to Q is the cost of the optimal solution of the following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{R(X)\in \mathrm{at}(Q)} w_{R(X)} \\ \text{subject to} \quad & \sum_{R(X):B\in X} w_{R(X)} \geq 1 && \text{for all } B \in Y \\ & w_{R(X)} \geq 0 && \text{for all } R(X) \in \mathrm{at}(Q) \end{aligned}$$

Given a set $\mathcal{L}$ of atoms of a query Q, we denote by $Q_{\mathcal{L}}$ the join query whose body is the conjunction of the atoms in $\mathcal{L}$. We can now define the dynamic width of a join query[6]:

Definition 22 (Dynamic Width)**.** *For any join query Q, the dynamic width of Q is*

$$\mathrm{dw}(Q) \stackrel{def}{=} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in \mathrm{leaves}(V(X),T)}} \rho^*_{Q_{\mathrm{leaves}(V(X),T)}}(X \setminus Y). \tag{10}$$

Proposition 23. *For any join query Q, it holds* $\mathrm{mw}(Q) \leq \mathrm{dw}(Q)$.

Proof. The proof is implied by the following chain of (in)equalities. We explain each step below.

$$\begin{aligned} \mathrm{mw}(Q) &\stackrel{\text{def}}{=} \min_{\epsilon\in[0,1]} \max_{d\in\mathcal{D}(Q)} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in\mathrm{leaves}(V(X),T)}} \min_{\substack{C\subseteq \mathrm{DC}(\mathrm{leaves}(\delta V(X),\delta T_R),d) \\ X\subseteq X'\subseteq \mathrm{vars}(C) \\ C'\in\mathcal{A}(C[X']) \\ w\in\mathcal{S}_{C'}}} \mathrm{obj}(w,C')(\epsilon) \\ &\stackrel{(1)}{\leq} \max_{d\in\mathcal{D}(Q)} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in\mathrm{leaves}(V(X),T)}} \min_{\substack{C\subseteq \mathrm{DC}(\mathrm{leaves}(\delta V(X),\delta T_R),d) \\ X\subseteq X'\subseteq \mathrm{vars}(C) \\ C'\in\mathcal{A}(C[X']) \\ w\in\mathcal{S}_{C'}}} \mathrm{obj}(w,C')(1) \\ &\stackrel{(2)}{=} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in\mathrm{leaves}(V(X),T)}} \min_{\substack{C\subseteq \mathrm{DC}(\mathrm{leaves}(\delta V(X),\delta T_R),L) \\ X\subseteq X'\subseteq \mathrm{vars}(C) \\ C'\in\mathcal{A}(C[X']) \\ w\in\mathcal{S}_{C'}}} \mathrm{obj}(w,C')(1) \\ &\stackrel{(3)}{\leq} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in\mathrm{leaves}(V(X),T)}} \min_{w\in\mathcal{S}_{\hat{C}}} \mathrm{obj}(w,\hat{C})(1) \\ &\stackrel{(4)}{\leq} \min_{T\in\mathcal{T}(Q)} \max_{\substack{V(X)\in T \\ R(Y)\in\mathrm{leaves}(V(X),T)}} \rho^*_{Q_{\mathrm{leaves}(V(X),T)}}(X \setminus Y) \\ &\stackrel{\text{def}}{=} \mathrm{dw}(Q), \end{aligned}$$

[6]We introduce here a simplified version of the dynamic width restricted to join queries, whereas the original definition [18] is for the more general conjunctive queries. This is because our maintenance width is defined here for join queries only.

where $\boldsymbol{L} = (L)_{|\mathcal{J}_Q|}$ is the degree constraint where all join variables are light and $\hat{C} = \overline{C}[\boldsymbol{X}]$ with $\overline{C} = \{(\boldsymbol{Z}|\emptyset, N) | S(\boldsymbol{Z}) \in \text{leaves}(\delta V(\boldsymbol{X}), \delta T_R) \wedge S \neq R\} \cup \{(B|\emptyset, 1) | B \in \boldsymbol{Y})\}$.

Inequality (1) is obtained by fixing ϵ to 1. Equality (2) is implied by the following observation. Consider a degree configuration $\boldsymbol{d} \in \mathcal{D}(Q)$ and a vector $\boldsymbol{w} \in \mathcal{S}_{C'}$ in the definition of $\text{mw}(Q)$. Assuming that $\epsilon = 1$, any degree constraint resulting from a variable A that is heavy in $\boldsymbol{d}$ is of the form $(A \mid \emptyset, N^{1-\epsilon} = N^0)$. This implies that for any vector $\boldsymbol{w} \in \mathcal{S}_{C'}$, the additive factor added to $\text{obj}(\boldsymbol{w}, C')(1)$ by such a constraint is of the form $w_i \cdot (1 - \epsilon) = 0$. Hence, instead of maximizing $\text{obj}(\boldsymbol{w}, C')(1)$ over all possible degree configurations, it suffices to restrict to the configuration $\boldsymbol{L} = (L)_{|\mathcal{J}_Q|}$, where all variables are light.

Inequality (3) holds because $\overline{C} \subseteq \text{DC}(\text{leaves}(\delta V(\boldsymbol{X}), \delta T_R), \boldsymbol{L})$ and both $\overline{C}$ and $\hat{C} = \overline{C}[\boldsymbol{X}]$ are acyclic, since the graph associated with $\overline{C}$ does not have any edge.

For Inequality (4), consider a view $\delta V(\boldsymbol{X})$ in a delta view tree for an update $\delta R(\boldsymbol{Y})$. Assume that $\hat{C} = \{c_1, \ldots, c_m\} \cup \{c^B\}_{B \in \boldsymbol{X} \cap \boldsymbol{Y}}$, where each c_i is of the form $(\boldsymbol{Z}|\emptyset, N)$ and each c^B is of the form $(B|\emptyset, 1)$. The linear program determining $\min_{\boldsymbol{w} \in \mathcal{S}_{\hat{C}}} \text{obj}(\boldsymbol{w}, \hat{C})(1)$ is as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{c_i = (\boldsymbol{Z}|\emptyset, N)} w_i \cdot 1 + \sum_{B \in \boldsymbol{X} \cap \boldsymbol{Y}} w^B \cdot 0 & & \\
\text{subject to} \quad & \sum_{\substack{c_i = (\boldsymbol{Z}|\emptyset, N) \\ A \in \boldsymbol{Z}}} w_i \geq 1 & & \forall A \in \boldsymbol{X} \setminus \boldsymbol{Y} \\
& \sum_{\substack{c_i = (\boldsymbol{Z}|\emptyset, N) \\ B \in \boldsymbol{Z}}} w_i + w^B \geq 1 & & \forall B \in \boldsymbol{X} \cap \boldsymbol{Y} \qquad (11) \\
& w_i \geq 0 & & \forall i \in [m] \\
& w^B \geq 0 & & \forall B \in \boldsymbol{X} \cap \boldsymbol{Y}
\end{aligned}
$$

The above program simplifies to the linear program determining $\rho^*_{\mathcal{Q}_{\text{leaves}(V(\boldsymbol{X}),T)}}(\boldsymbol{X} \setminus \boldsymbol{Y})$ due to the following two observations. Firstly, each constraint in $\hat{C}$ is of the form $(\boldsymbol{Z}|\emptyset, N)$ or of the form $(B|\emptyset, 1)$ with $B \in \boldsymbol{X} \cap \boldsymbol{Y}$. Secondly, the weights w^B associated with constraints of the form $c^B = (B|\emptyset, 1)$ do not have any effect on the objective function of the above linear program, since they are multiplied with 0 in the definition of the objective function. Hence, the constraints in Line (11) can be easily satisfied by setting such a weight w^B to 1. This implies that constraints in Line (11) can be omitted. □

5.2 Comparison with Further Width Measures

The maintenance width does not come with a simple syntactic check. Indeed, to find this width for a given query Q, one needs to iterate over all view trees and degree configurations for Q and solve a linear program to cost the update of each view in a view tree triggered by an update to any input relation. Yet this is conceptually not different from well-established width measures, such as the fractional hypertree width [24] or the submodular width [2]. These widths are defined by iterating over all hypertree decompositions [9] of Q and by solving a linear program to cost each bag of a hypertree decomposition. Furthermore, the submodular width also requires adaptive computation by data partitioning. There are two differences here: (i) The data partitioning for the submodular width is fine-grained as it yields (poly-logarithmically many in N) database parts of uniform degrees, whereas for the maintenance width it is coarse-grained as it yields (constantly many in N) database parts with either heavy or light degrees. (ii) The data partitioning for the

maintenance width is on the input relations only, whereas for the submodular width it can also be on the intermediate results (so also on the materialized views in our setting).

There are two aspects of our maintenance width which are distinct from the aforementioned widths. (i) Due to our dynamic setting, we also need to consider the cost of view updates as opposed to the cost of computing the view only. (ii) We do not know the concrete cost of each view update until we fix the threshold ϵ, which can only be done after constructing the function in ϵ that defines the width. This is novel to our setting.

5.3 View Trees vs. Hypertree Decompositions

Each view tree of a query Q can be mapped to a hypertree decomposition of Q (possibly with redundant bags), where each view (relation) becomes a bag consisting of the view variables. Also, from each hypertree decomposition we can construct a view tree, with one view for each bag of the decomposition, possibly additional projection views, and one leaf per relation in Q. So both view trees and hypertree decompositions allow us to explore the same space of structural decompositions of Q, albeit the view trees are more refined in that they use redundant information in the form of materialized projection views to allow for a faster propagation of updates in the view trees.

6 Major and Minor Rebalancing of Data Partitioning

In the previous sections we showed that for any join query Q, the view trees constructed by our approach can be maintained in $O(N^{\mathrm{mw}(Q)})$ time under *one* single-tuple update, where $\mathrm{mw}(Q)$ denotes the maintenance width of the query Q. In this section, we extend this analysis to *sequences* of single-tuple updates. We show that, given a *sequence* of single-tuple updates, the *amortized* single-tuple update time remains $O(N^{\mathrm{mw}(Q)})$. Our proof is based on an adaptation of the amortization technique previously developed for the triangle query [15]. For clarity, we outline the differences from that prior work and state the central ideas of the argument.

Each update may affect both the size of the database and the degrees of data values. Whenever the database size exceeds specified bounds, we recompute the view trees for all degree configurations by taking the new database size into account. We refer to this operation as *major rebalancing*. Similarly, if a light value becomes heavy, or vice-versa, as a result of an update, we move the affected tuples between the corresponding relation fragments and update the view trees evaluated over these fragments. We call this operation *minor rebalancing*. The cost of both types of rebalancing can be amortized over the number of updates between two consecutive rebalancing steps, yielding an amortized rebalancing cost of $O(N^{\mathrm{mw}(Q)})$ per single-tuple update.

One difference between the maintenance strategy for the triangle query in prior work [15] and the approach proposed here is the definition of light and heavy values. In the prior work, lightness and heaviness were defined with respect to each individual relation; that is, an X-value x is considered light in a relation R if $|\sigma_{X=x}R| \leq N^{\epsilon}$, and heavy otherwise. In contrast, in this work we define lightness and heaviness globally, with respect to *all* relations in the database (see Sec. 2). This distinction does not change the analysis of the amortized rebalancing time. The detailed analysis of the major and minor rebalancing steps is given in the extended version of this article.

7 Constant-Delay Enumeration of the Query Output

For any join query, our approach constructs a set of view trees and maintains them under single-tuple updates to the input relations. As discussed in this section, we can also enumerate the query output from these view trees with constant delay.

Consider a join query $Q(\boldsymbol{X}) = R_1(\boldsymbol{X}_1) \cdot \ldots \cdot R_k(\boldsymbol{X}_k)$ with degree configurations $\mathcal{D}(Q)$. Let OUT denote the set of output tuples of Q, and $\mathsf{OUT}_{\boldsymbol{d}}$ the set of output tuples of Q under the degree

configuration $\boldsymbol{d} \in \mathcal{D}(Q)$. The set OUT is the disjoint union of the sets $\mathsf{OUT}_{\boldsymbol{d}}$. Given a constant-delay enumeration procedure for each tuple set $\mathsf{OUT}_{\boldsymbol{d}}$, we can therefore enumerate the tuples in OUT with constant delay by invoking the procedures one after the other. For each tuple $\boldsymbol{t} \in \mathsf{OUT}$, its multiplicity is $\sum_{i\in[k]} R_i(\boldsymbol{t}.\boldsymbol{X}_i)$, which can be computed in constant time.

We next discuss such a constant-delay enumeration procedure for a single set $\mathsf{OUT}_{\boldsymbol{d}}$, for any $\boldsymbol{d} \in \mathcal{D}(Q)$. Recall that for each degree configuration $\boldsymbol{d} \in \mathcal{D}(Q)$, our approach maintains a view tree $T^{\boldsymbol{d}}$. It then suffices to enumerate $\mathsf{OUT}_{\boldsymbol{d}}$ from $T^{\boldsymbol{d}}$ with constant delay. Prior work shows how from any view tree, the tuples in the join of the views in the tree can be enumerated with constant delay [13, Prop. 11]. The extended version of this article gives further details, illustrates the enumeration procedure for the 4-cycle query, and explains how to slightly change the approach to maintain the count version of the query with the same update time and constant-delay enumeration.

8 Conclusion

In this paper we introduced an approach to adaptive maintenance of join queries under database updates (inserts and deletes) that exploits the constraints that come with partitioning the database using the heavy and light degrees of the values of the join variables. Our approach matches the best known update times stated in the literature, while also generalizing to arbitrary join queries.

There are several promising directions of future work, we list below a few:

- Our complexity results hold for join queries only, but they can be generalized in a standard way to arbitrary conjunctive queries by taking view trees that correspond to free-connex hypertree decompositions of such queries. This free-connex restriction is necessary to ensure constant delay enumeration of the query output. The machinery developed in this paper to upper bound the sizes and compute times for delta views already works for arbitrary conjunctive queries.
- We can extend our approach to richer constraints beyond degree constraints: Using ℓ_p-norms on the degree sequences of join columns, we can obtain tighter upper bounds on the size of the query output and on the runtime to compute it [19, 20]. Adopting such statistics to the dynamic setting requires to maintain them efficiently under updates.
- We currently partition the data on each join variable. It remains open whether partitioning on tuples of variables and subsets of such tuples can improve the update time. For instance, for a query with body $R(A, B, C) \cdot S(A, B, D) \cdot T(A, E) \cdot U(B, F)$ we currently only partition on A and B separately, although we could also partition on the tuple A, B. Our framework extends immediately to this more general setting.
- We can lower the update time by using fast matrix multiplication: The $O(N^{2/3})$ barrier for the update time of the 4-cycle query can be broken using a non-combinatorial IVM approach [3]. It is unclear however how to generalize this non-combinatorial approach to arbitrary queries.
- When using a maintenance approach based on single view tree, inserts-only updates can require a lower update time than the general case with both inserts and deletes [1]. It is open whether this restriction can also lower the update time in our approach.

Acknowledgments

This work is partially supported by SNSF 200021-231956.

References

[1] Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2024. Insert-Only versus Insert-Delete in Dynamic Query Evaluation. *Proc. ACM Manag. Data* 2, 5, Article 219 (Nov. 2024). doi:10.1145/3695837

[2] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2025. PANDA: Query Evaluation in Submodular Width. *TheoretiCS* Volume 4, Article 12 (Apr 2025). doi:10.46298/theoretics.25.12

[3] Sepehr Assadi and Vihan Shah. 2025. An Improved Fully Dynamic Algorithm for Counting 4-Cycles in General Graphs Using Fast Matrix Multiplication. *Proc. ACM Manag. Data* 3, 2 (2025), 91:1–91:24. doi:10.1145/3725228

[4] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *FOCS*. 739–748. doi:10.1109/FOCS.2008.43

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

[6] Mihai Budiu, Leonid Ryzhyk, Gerd Zellweger, Ben Pfaff, Lalith Suresh, Simon Kassing, Abhinav Gyawali, Matei Budiu, Tej Chajed, Frank McSherry, and Val Tannen. 2025. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *VLDB J.* 34, 4 (2025). doi:10.1007/S00778-025-00922-Y

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

[8] Arnaud Durand and Etienne Grandjean. 2007. First-Order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic* 8, 4 (Aug. 2007), 19 pages. doi:10.1145/1276920.1276923

[9] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM* 56, 6 (2009), 30:1–30:32. doi:10.1145/1568318.1568320

[10] Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. 2022. Fully Dynamic Four-Vertex Subgraph Counting. In *SAND*. 18:1–18:17. doi:10.4230/LIPICS.SAND.2022.18

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

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

[13] Ahmet Kara, Zheng Luo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2025. Tractable Conjunctive Queries over Static and Dynamic Relations. In *ICDT*. 12:1–12:21. doi:10.4230/LIPIcs.ICDT.2025.12

[14] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT*. 4:1–4:18. doi:10.4230/LIPICS.ICDT.2019.4

[15] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Maintaining Triangle Queries under Updates. *ACM Trans. Database Syst.* 45, 3 (2020), 11:1–11:46. doi:10.1145/3396375

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

[17] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2024. F-IVM: Analytics over Relational Databases under Updates. *VLDB J.* 33, 4 (2024), 903–929. doi:10.1007/S00778-023-00817-W

[18] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2025. Conjunctive Queries with Free Access Patterns under Updates. *LMCS* Volume 21, Issue 2, Article 23 (Jun 2025). doi:10.46298/lmcs-21(2:23)2025

[19] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2024. Join Size Bounds using $\mathrm{l_p}$-Norms on Degree Sequences. *Proc. ACM Manag. Data* 2, 2 (2024), 96. doi:10.1145/3651597

[20] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2025. Information Theory Strikes Back: New Development in the Theory of Cardinality Estimation. *SIGMOD Rec.* 54, 1 (2025), 7–15. doi:10.1145/3733620.3733623

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

[22] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2016. Computing Join Queries with Functional Dependencies. In *PODS*. 327–342. doi:10.1145/2902251.2902289

[23] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.* 23, 2 (2014), 253–278. doi:10.14778/2336664.2336670

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

[25] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (2016), 75–83. doi:10.1145/2983551

[26] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *PODS*. 111–124. doi:10.1145/3196959.3196990

[27] Dan Olteanu. 2024. Recent Increments in Incremental View Maintenance. In *PODS*. 8–17. doi:10.1145/3635138.3654763

[28] Alexander Schrijver. 1986. *Theory of linear and integer programming*. John Wiley & Sons, Inc., USA.

[29] Daniel Sotolongo, Daniel Mills, Tyler Akidau, Anirudh Santhiar, Attila-Péter Tóth, Botong Huang, Boyuan Zhang, Igor Belianski, Ling Geng, Matt Uhlar, Nikhil Shah, Olivia Zhou, Saras Nowak, Sasha Lionheart, Vlad Lifliand, Wendy Grus, Yiwen Zhu, Ankur Sharma, Dzmitry Pauliukevich, Enrico Sartorello, Ilaria Battiston, Ivan Kalev, Lawrence Benson, Leon Papke, Niklas Semmler, Till Merker, and Yi Huang. 2025. Streaming Democratized: Ease Across the Latency Spectrum

with Delayed View Semantics and Snowflake Dynamic Tables. In *SIGMOD*. 622–634. doi:10.1145/3722212.3724455
[30] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation without Joins. *Proc. VLDB Endow.* 16, 5 (2023), 1046–1058. doi:10.14778/3579075.3579080

A Proof of Theorem 11

In this section we prove:

Theorem 11. *Any join query Q can be maintained with $O(N^{1+\mathrm{mw}(Q)})$ preprocessing time, amortized $O(N^{\mathrm{mw}(Q)})$ single-tuple update time, and $O(1)$ enumeration delay, where N is the size of the database at the time of update and $\mathrm{mw}(Q)$ is the maintenance width of Q.*

Given a join query Q, we analyze the time complexity of each of the three stages of our algorithm: preprocessing, maintenance, and enumeration.

Preprocessing. We compute the maintenance width $\mathrm{mw}(Q)$ and the optimal parameter ϵ^* as detailed in Sec. 4.4. Next, we partition the active domain into heavy and light values based on the thresholds defined by ϵ^*. We then compute the set of active view trees and assign the optimal tree $T^{\boldsymbol{d}}$ to each degree configuration $\boldsymbol{d}$.

To compute the views of our active view trees, we start with an empty database and then insert one by one each tuple from the initial database of size N. Since the time to process a single update is $O(N^{\mathrm{mw}(Q)})$, the overall time to compute the views is $O(N^{1+\mathrm{mw}(Q)})$). The number of views is only dependent on the query, so independent of the database size.

Maintenance. Consider a single tuple update $\delta R = \{\boldsymbol{t} \mapsto \pm 1\}$. The maintenance procedure follows two steps:

(1) **Selecting the configuration and delta view tree:** We inspect the values in the tuple $\boldsymbol{t}$. If a value a in $\boldsymbol{t}$ is encountered for the first (i.e., it is not in the active domain), we initialize it as a light value. Otherwise, we use its degree in the data to decide whether it is heavy or light. We then determine the degree configuration $\boldsymbol{d}$ corresponding to the degrees of values in $\boldsymbol{t}$ based on the current heavy-light threshold and select the corresponding view tree $T^{\boldsymbol{d}}$.
(2) **View Updates:** We compute $\delta T_R^{\boldsymbol{d}}$ for the update δR. By Lemma 17 and the definition of the maintenance width, the cost of this operation is bounded by $O(N^{\mathrm{mw}(Q)})$ time in data complexity. We the use the computed $\delta T_R^{\boldsymbol{d}}$ to update the view tree $T^{\boldsymbol{d}}$.

This bound on update time holds strictly when the degree constraints for $\boldsymbol{d}$ remain satisfied. However, a sequence of updates may alter value frequencies, violating the constraints. In such cases, a minor or major rebalancing step is triggered. As detailed in Sec. 6, the cost of these rebalancing steps can be amortized over the update sequence, yielding the same (now amortized) update time.

Enumeration. Upon each enumeration request, we enumerate the distinct tuples in the query output, along with their multiplicities, from the active view trees with constant delay, as described in Sec. 7.

Received December 2025; accepted February 2026