Insert-Only versus Insert-Delete in Dynamic Query Evaluation

MAHMOUD ABO KHAMIS, RelationalAI, USA
AHMET KARA, OTH Regensburg, Germany
DAN OLTEANU, University of Zurich, Switzerland
DAN SUCIU, University of Washington, USA

We study the dynamic query evaluation problem: Given a full conjunctive query Q and a sequence of updates to the input database, we construct a data structure that supports constant-delay enumeration of the tuples in the query output after each update.

We show that a sequence of N insert-only updates to an initially empty database can be executed in total time $O(N^{\mathrm{w}(Q)})$, where $\mathrm{w}(Q)$ is the fractional hypertree width of Q. This matches the complexity of the static query evaluation problem for Q and a database of size N. One corollary is that the amortized time per single-tuple insert is constant for α-acyclic full conjunctive queries.

In contrast, we show that a sequence of N inserts and deletes can be executed in total time $\widetilde{O}(N^{\mathrm{w}(\widehat{Q})})$, where $\widehat{Q}$ is obtained from Q by extending every relational atom with extra variables that represent the "lifespans" of tuples in the database. We show that this reduction is optimal in the sense that the static evaluation runtime of $\widehat{Q}$ provides a lower bound on the total update time for the output of Q. Our approach achieves amortized optimal update times for the hierarchical and Loomis-Whitney join queries.

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

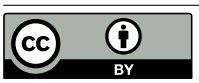
Additional Key Words and Phrases: incremental view maintenance; optimality; intersection joins

ACM Reference Format:
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 (PODS), Article 219 (November 2024), 26 pages. https://doi.org/10.1145/3695837

1 Introduction

Answering queries under updates to the database, also called dynamic query evaluation, is a fundamental problem in data management, with recent work addressing it from both systems and theoretical perspectives. There have been recent efforts on building systems for dynamic query evaluation, such as DBToaster [28], DynYannakakis [18, 19], F-IVM [26, 37, 38], and CROWN [44]. By allowing tuples to carry payloads with elements from rings [27], such systems can maintain complex analytics over database queries, such as linear algebra computation [36], collection programming [29], and machine learning models [26]. There has also been work on dynamic computation for intersection joins [42] and for expressive languages such as Datalog [32], Differential

Authors' Contact Information: Mahmoud Abo Khamis, mahmoudabo@gmail.com, RelationalAI, Berkeley, CA, USA; Ahmet Kara, ahmet.kara@oth-regensburg.de, OTH Regensburg, Department of Computer Science & Mathematics, Regensburg, Germany; Dan Olteanu, olteanu@ifi.uzh.ch, University of Zurich, Department of Informatics, Zurich, Switzerland; Dan Suciu, suciu@cs.washington.edu, University of Washington, Department of Computer Science & Engineering, Seattle, WA, USA.



ACM 2836-6573/2024/11-ART219
https://doi.org/10.1145/3695837

Datalog [1], and DBSP [11]. The descriptive complexity of various recursive Datalog queries under updates, such as reachability, has also been investigated [13, 40]. Dynamic query evaluation has also been utilized for complex event recognition [15, 16].

While this problem attracted continuous interest in academia and industry over the past decades, it is only relatively recently that the first results on the fine-grained complexity and in particular on the optimality of this problem have emerged. These efforts aim to mirror the breakthrough made by the introduction of worst-case optimal join algorithms [7, 34, 43]. Beyond notable yet limited explorations, understanding the optimality of maintenance for the entire language of conjunctive queries remains open. Prime examples of progress towards the optimality of query maintenance are the characterizations of queries that admit (worst-case or amortized) constant time per single-tuple update (insert or delete): the q-hierarchical queries [9, 18, 26, 44], queries that become q-hierarchical in the presence of free access patterns [24] or by rewriting under functional dependencies [26], or queries on update sequences whose enclosureness is bounded by a constant [45]. The δ_1-hierarchical queries [23–25] and the triangle queries [21, 22] admit optimal, albeit non-constant, update time conditioned on the Online Matrix-Vector Multiplication (OMv) conjecture [17].

In this paper, we introduce a new approach to incremental view maintenance for full conjunctive queries (or queries for short). Our approach complements prior work in four ways.

First, we give reductions from the dynamic query evaluation problem for a query Q to the static query evaluation problem of a derived query $\widehat{Q}$ (called the multivariate extension of Q), and vice versa. Our reductions link the complexities of the two problems and allow to transfer *both* algorithms *and* lower bounds from one problem to the other. Specifically, we give a reduction from dynamic query evaluation to a wide class of algorithms for static query evaluation, namely algorithms that meet the *fractional hypertree width*, thus allowing us to use those algorithms for dynamic query evaluation. Moreover, we give a reduction from static query evaluation to *any* algorithm for dynamic query evaluation, thus allowing us to use lower bounds on static query evaluation to infer lower bounds on dynamic query evaluation. Both reductions use a translation of the time dimension in the dynamic problem into a spatial dimension in the static problem, so that the maintenance under a stream of updates corresponds to taking the intersection of intervals representing the "lifespans" of tuples in the update stream.

Second, we devise new dynamic query evaluation algorithms that use our reductions. We call them MVIVM (short for MultiVariate IVM). Their single-tuple update times are *amortized*. For any α-acyclic query in the insert-only setting, MVIVM take amortized constant single-tuple update time. This recovers a prior result [44], when restricted from free-connex to full conjunctive queries. For Loomis-Whitney queries in the insert-delete setting, MVIVM take amortized $\widetilde{O}(|\mathcal{D}|^{1/2})$ update time, where $|\mathcal{D}|$ is the database size at update time; this is optimal up to a polylogarithmic factor in the database size. This recovers prior work on the triangle query, which is the Loomis-Whitney query with three variables [22]. For hierarchical queries in the insert-delete setting, MVIVM take amortized constant single-tuple update time; this is weaker than prior work [9], which showed that worst-case constant single-tuple update time can be achieved for q-hierarchical queries.

Third, MVIVM support (worst-case, not amortized) constant delay enumeration of the tuples in the query output (full enumeration) or in the change to the query output (delta enumeration) *after each update*. In particular, MVIVM map an input stream of updates to an output stream of updates, so they are closed under updates. In practice, the change to the output after an update is often much smaller than the full output, making delta enumeration more efficient than full enumeration.

Fourth, we pinpoint the complexity gap between the insert-only and the insert-delete settings. In the insert-only setting, we show that one can readily use a worst-case optimal join algorithm like GenericJoin [34] and LeapFrogTrieJoin [43] to execute a stream of inserts one insert at a time in

the same overall time as if the entire stream was executed as one bulk update (which corresponds to static evaluation). This implies an upper bound on the amortized update time for the dynamic evaluation of Q in the insert-only setting. In contrast, in the insert-delete setting, a stream of both inserts and deletes can be executed one update at a time in the same overall time as the static evaluation of a query $\widehat{Q}$, which is obtained by extending every relational atom in Q with extra variables representing the "lifespans" of tuples. This query $\widehat{Q}$ may have a higher complexity than Q, thus leading to an upper bound on the amortized update time for Q in the insert-delete setting that may be higher than in the insert-only setting. Nevertheless, our reduction from the maintenance of Q in the insert-delete setting to the static evaluation problem for $\widehat{Q}$ is optimal: Lower bounds on the latter problem imply lower bounds on the former. This shows that maintenance in the insert-delete setting can be more expensive than in the insert-only setting.

Motivating Examples

We exemplify our results for the following three join queries:

$$\begin{aligned} Q_{3p}(A,B,C,D) &= R(A,B) \wedge S(B,C) \wedge T(C,D) \\ Q_{\triangle}(A,B,C) &= R(A,B) \wedge S(B,C) \wedge T(A,C) \\ Q_4(A,B,C,D) &= R(B,C,D) \wedge S(A,C,D) \wedge T(A,B,D) \wedge U(A,B,C) \end{aligned} \tag{1}$$

We would like to perform single-tuple updates to each query, while ensuring constant-delay enumeration of the tuples in the query output after each update. These queries are not q-hierarchical, hence they cannot admit worst-case constant time per single-tuple update [9]: The OMv conjecture implies that there is no algorithm that takes $O(|\mathcal{D}|^{1/2-\gamma})$ time for single-tuple updates, for any $\gamma > 0$, to any of these queries. In fact, for a database $\mathcal{D}$, systems like DBToaster [28], DynYannakakis [18], F-IVM [37], and CROWN [44] need at least worst-case $O(|\mathcal{D}|)$ time per single-tuple update to each of these queries. For the triangle query $Q_{\triangle}$, IVM$^{\epsilon}$ takes $O(|\mathcal{D}|^{1/2})$ amortized time for single-tuple updates [22]. In the insert-only setting, CROWN maintains Q_{3p} in amortized constant time per insert. However, it cannot handle non-acyclic queries like $Q_{\triangle}$ and Q_4 (with update time better than recomputation from scratch). Appendix A explains how each prior approach maintains Q_{3p} and $Q_{\triangle}$ in the insert-delete setting, and compares their update times to our approach, MVIVM.

We show that, for Q_{3p}, MVIVM performs single-tuple updates: in amortized constant time in the insert-only setting; and in amortized $\widetilde{O}(|\mathcal{D}|^{1/2})$ time in the insert-delete setting ($\tilde{O}$ notation hides a polylog$(|\mathcal{D}|)$ factor). Both times are optimal, the latter is conditioned on the OMv conjecture. The amortized constant time per insert matches that of CROWN [44] and does not contradict the non-constant lower bound for non-hierarchical queries without self-joins [9]: that lower bound proof relies on a reduction from OMv that requires *both* inserts *and* deletes. Achieving this amortized constant time, while also allowing constant-delay enumeration after each insert, is non-trivial. The lazy maintenance approach, which only updates the input data, is *not* sufficient to achieve this result. This is because, while it does indeed take constant insert time, it still requires the computation of the query output *before* the enumeration of the first output tuple. This computation takes linear time using a factorized approach [26] and needs to be repeated for *every* enumeration request.

MVIVM takes $O(|\mathcal{D}|^{1/2})$ and $O(|\mathcal{D}|^{1/3})$ amortized single-tuple insert time for the triangle query $Q_{\triangle}$ and, respectively, the Loomis-Whitney-4 query Q_4. Given a sequence of $|\mathcal{D}|$ inserts, the total times of $O(|\mathcal{D}|^{3/2})$ and $O(|\mathcal{D}|^{4/3})$, respectively, match the worst-case optimal times for the static evaluation of these two queries [34]. In the insert-delete setting, single-tuple updates take $\widetilde{O}(|\mathcal{D}|^{1/2})$ amortized time for both queries, and this is optimal based on the OMv conjecture. Figure 4 in Appendix A shows the update times of MVIVM and prior work for further queries.

2 Preliminaries

In this section, we review some preliminaries, most of which are standard in database theory.

We use $\mathbb{N}$ to denote the set of natural numbers including 0. For $n \in \mathbb{N}$, we define $[n] \stackrel{\text{def}}{=} \{1, 2, \ldots, n\}$. In case $n = 0$, we have $[n] = \emptyset$.

Data Model. Following standard terminology, a *relation* is a finite set of tuples and a *database* is a finite set of relations [2]. The size $|R|$ of a relation R is the number of its tuples. The size $|\mathcal{D}|$ of a database $\mathcal{D}$ is given by the sum of the sizes of its relations.

Queries. We consider natural join queries or full conjunctive queries. We refer to them as *queries* for short. A *query* has the form

$$Q(\boldsymbol{X}) = R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k), \tag{2}$$

where each R_i is a relation symbol, each $\boldsymbol{X}_i$ is a tuple of variables, and $\boldsymbol{X} = \bigcup_{i\in[k]} \boldsymbol{X}_i$. We refer to $\boldsymbol{X}_i$ as the *schema* of R_i and treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. We call each $R_i(\boldsymbol{X}_i)$ an *atom* of Q. We denote by $\text{at}(Q) = \{R_i(\boldsymbol{X}_i) \mid i \in [k]\}$ the set of all atoms of Q and by $\text{at}(X)$ the set of atoms $R_i(\boldsymbol{X}_i)$ with $X \in \boldsymbol{X}_i$. The set of all variables of Q is denoted by $\text{vars}(Q) = \bigcup_{i\in[k]} \boldsymbol{X}_i$. Since the set $\boldsymbol{X}$ is always equal to $\text{vars}(Q)$ for all queries we consider in the paper, we drop $\boldsymbol{X}$ from the query notation and abbreviate $Q(\boldsymbol{X})$ with Q in Eq. (2). We say that Q is without *self-joins* if every relation symbol appears in at most one atom.

The domain of a variable X is denoted by $\text{Dom}(X)$. A value tuple $\boldsymbol{t}$ over a schema $\boldsymbol{X} = (X_1, \ldots, X_n)$ is an element from $\text{Dom}(\boldsymbol{X}) \stackrel{\text{def}}{=} \text{Dom}(X_1) \times \cdots \times \text{Dom}(X_n)$. We denote by $\boldsymbol{t}(X)$ the X-value of $\boldsymbol{t}$.

A tuple $\boldsymbol{t}$ over schema $\boldsymbol{X}$ is in the result of Q if there are tuples $\boldsymbol{t}_i \in R_i$ for $i \in [k]$ such that for all $X \in \text{vars}(Q)$, the set $\{\boldsymbol{t}_i(X) \mid i \in [k] \wedge X \in \boldsymbol{X}_i\}$ consists of the single value $\boldsymbol{t}(X)$. Given a query Q and a database $\mathcal{D}$, we denote by $Q(\mathcal{D})$ the result of Q over $\mathcal{D}$.

A *union of queries* is of the form $Q = \bigvee_{i\in[n]} Q_i$ where each Q_i, called a component of Q, is a query over the same set of variables as Q. The result of Q is the union of the results of its components.

Query Classes. A query is *(α-)acyclic* if we can construct a tree, called *join tree*, such that the nodes of the tree are the atoms of the query (*coverage*) and for each variable, it holds: if the variable appears in two atoms, then it appears in all atoms on the path connecting the two atoms (*connectivity*) [10].

Hierarchical queries are a sub-class of acyclic queries. A query is called hierarchical if for any two variables X and Y, it holds $\text{at}(X) \subseteq \text{at}(Y)$, $\text{at}(Y) \subseteq \text{at}(X)$, or $\text{at}(X) \cap \text{at}(Y) = \emptyset$ [41].

Loomis-Whitney queries generalize the triangle query from a clique of degree 3 to higher degrees [30]. A Loomis-Whitney query of degree $k \geq 3$ has k variables $X_1, \ldots, X_k$ and has, for each subset $\boldsymbol{Y} \subset \{X_1, \ldots, X_k\}$ of size $k - 1$, one distinct atom with schema $\boldsymbol{Y}$.

A union of queries is acyclic (hierarchical, Loomis-Whitney) if each of its components is acyclic (hierarchical, Loomis-Whitney).

Single-tuple Updates and Deltas. We denote an *insert* of a tuple $\boldsymbol{t}$ into a relation R by $+R(\boldsymbol{t})$ and a *delete* of $\boldsymbol{t}$ from R by $-R(\boldsymbol{t})$. The insert $+R(\boldsymbol{t})$ inserts $\boldsymbol{t}$ into R if $\boldsymbol{t}$ *is not* contained in R, and the delete $-R(\boldsymbol{t})$ deletes $\boldsymbol{t}$ from R if it *is* contained in R. Each insert or delete has a *timestamp* τ, which is a natural number that indicates the specific time at which the insert or delete occurs. An *update* $\delta_\tau R$ to R at time τ is the set of all inserts and deletes to R at time τ. An *update* $\delta_\tau R$ is applied to R by first applying all the deletes in $\delta_\tau R$ and then applying all the inserts. We use $R \uplus \delta_\tau R$ to denote the result of applying an update $\delta_\tau R$ to a relation R. We use $R^{(\tau)}$ to refer to the specific version of a relation R at timestamp τ. In particular, $R^{(\tau)}$ is the result of applying $\delta_\tau R$ to the previous version $R^{(\tau-1)}$, i.e. $R^{(\tau)} = R^{(\tau-1)} \uplus \delta_\tau R$. An *update* $\delta_\tau \mathcal{D}$ to a database $\mathcal{D}$ is the set of all inserts and deletes to the relations in $\mathcal{D}$ at time τ. Similarly, we use $\mathcal{D}^{(\tau)}$ to refer to the specific version of $\mathcal{D}$ at time τ,

i.e., $\mathcal{D}^{(\tau)} = \mathcal{D}^{(\tau-1)} \uplus \delta_\tau \mathcal{D}$. If $\delta_\tau \mathcal{D}$ consists of a single insert or delete, i.e., $|\delta_\tau \mathcal{D}| = 1$, it is called a *single-tuple update*. In particular, if it consists of a single insert (or delete), we call it a *single-tuple insert* (or *single-tuple delete*). Given a query Q over a database $\mathcal{D}$, we use $\delta_\tau Q(\mathcal{D})$ to denote the change in $Q(\mathcal{D})$ at time τ. Specifically, $\delta_\tau Q(\mathcal{D})$ is the difference between $Q(\mathcal{D}^{(\tau)})$ and $Q(\mathcal{D}^{(\tau-1)})$:

$$\begin{aligned}\delta_\tau Q(\mathcal{D}) \stackrel{\text{def}}{=} &\{+Q(\boldsymbol{t}) \mid \boldsymbol{t} \in Q(\mathcal{D}^{(\tau)}) \text{ and } \boldsymbol{t} \notin Q(\mathcal{D}^{(\tau-1)})\} \\ &\cup \{-Q(\boldsymbol{t}) \mid \boldsymbol{t} \in Q(\mathcal{D}^{(\tau-1)}) \text{ and } \boldsymbol{t} \notin Q(\mathcal{D}^{(\tau)})\} \end{aligned} \tag{3}$$

When the time τ is clear from the context, we drop the annotation (τ) and write $R, \delta R, \mathcal{D}, \delta\mathcal{D}, Q(\mathcal{D})$, and $\delta Q(\mathcal{D})$. In the insert-only setting, since all deltas are positive, we ignore the signs and see a single-tuple update as just a tuple, δR as just a relation, and so on.

Constant-Delay Enumeration. Given a query Q and a database $\mathcal{D}$, an enumeration procedure for $Q(\mathcal{D})$ (or $\delta_\tau Q(\mathcal{D})$) outputs the elements in $Q(\mathcal{D})$ (or $\delta_\tau Q(\mathcal{D})$) one by one in some order and without repetition. The *enumeration delay* of the procedure is the maximum of three times: the time between the start of the enumeration process and the output of the first element, the time between outputting any two consecutive elements, and the time between outputting the last element and the end of enumeration [14]. The enumeration delay is *constant* if it does not depend on $|\mathcal{D}|$.

Width Measures. Consider the following linear program for a query Q and a database $\mathcal{D}$:

$$\begin{aligned} \min \quad & \sum_{R(\boldsymbol{X}) \in \text{ at}(Q)} \log(|R|) \cdot \lambda_{R(\boldsymbol{X})} \\ \text{s.t.} \quad & \sum_{R(\boldsymbol{X}) \in \text{ at}(Q) \text{ s.t. } Y \in \boldsymbol{X}} \lambda_{R(\boldsymbol{X})} \geq 1 && \text{for all } Y \in \text{vars}(Q) \text{ and} \\ & \lambda_{R(\boldsymbol{X})} \in [0,1] && \text{for all } R(\boldsymbol{X}) \in \text{at}(Q) \end{aligned}$$

Every feasible solution $(\lambda_{R(\boldsymbol{X})})_{R(\boldsymbol{X}) \in \text{at}(Q)}$ to the above program is called a *fractional edge cover* of Q with respect to $\mathcal{D}$. The optimal objective value of the program is called the *fractional edge cover number* $\rho^*(Q, \mathcal{D})$ of Q with respect to $\mathcal{D}$. The fractional edge cover number gives an upper bound on the size of $Q(\mathcal{D})$: $|Q(\mathcal{D})| \leq 2^{\rho^*(Q,\mathcal{D})}$ [7]. This bound is known as the *AGM-bound* and denoted as $\text{AGM}(Q, D)$. We denote by $\rho^*(Q)$ the optimal objective value of the linear program that results from the above program by replacing the objective function with $\sum_{R(\boldsymbol{X}) \in \text{at}(Q)} \lambda_{R(\boldsymbol{X})}$.

Definition 2.1 (Tree Decomposition). A *tree decomposition* of a query Q is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree with vertices $V(\mathcal{T})$ and $\chi : V(\mathcal{T}) \to 2^{\text{vars}(Q)}$ maps each node t of the tree $\mathcal{T}$ to a subset $\chi(t)$ of variables of Q such that the following properties hold:

(1) for every atom $R(\boldsymbol{X}) \in \text{at}(Q)$, the schema $\boldsymbol{X}$ is a subset of $\chi(t)$ for some $t \in V(\mathcal{T})$, and
(2) for every variable $X \in \text{vars}(Q)$, the set $\{t \mid X \in \chi(t)\}$ is a non-empty connected subtree of $\mathcal{T}$. The sets $\chi(t)$ are called the *bags* of the tree decomposition.

We use $\text{TD}(Q)$ to denote the set of tree decompositions of a query Q.

Definition 2.2 (The restriction Q_Y of a query Q). For a query Q and a subset $\boldsymbol{Y} \subseteq \text{vars}(Q)$, we define the *restriction of Q to $\boldsymbol{Y}$*, denoted by Q_Y, to be the query that results from Q by restricting the schema of each atom to those variables that appear in $\boldsymbol{Y}$. Formally:

$$\begin{aligned} R_Y(\boldsymbol{X} \cap \boldsymbol{Y}) &\stackrel{\text{def}}{=} \pi_{\boldsymbol{X} \cap \boldsymbol{Y}} R(\boldsymbol{X}), \qquad \text{for all } R(\boldsymbol{X}) \in \text{at}(Q) \\ Q_Y(\boldsymbol{Y}) &\stackrel{\text{def}}{=} \bigwedge_{R(\boldsymbol{X}) \in \text{at}(Q)} R_Y(\boldsymbol{X} \cap \boldsymbol{Y}) \end{aligned} \tag{4}$$

Definition 2.3 (Fractional Hypertree Width). Given a query Q and a tree decomposition $(\mathcal{T}, \chi)$ of Q, the *fractional hypertree width* of $(\mathcal{T}, \chi)$ and of Q are defined respectively as follows:

$$\mathrm{w}(\mathcal{T}, \chi) \stackrel{\text{def}}{=} \max_{t \in V(\mathcal{T})} \rho^*(Q_{\chi(t)}), \qquad \mathrm{w}(Q) \stackrel{\text{def}}{=} \min_{(\mathcal{T},\chi) \in \mathrm{TD}(Q)} \mathrm{w}(\mathcal{T}, \chi) \tag{5}$$

We call a tree decomposition $(\mathcal{T}, \chi)$ of Q *optimal* if $\mathrm{w}(\mathcal{T}, \chi) = \mathrm{w}(Q)$. The *fractional hypertree width* of a union $Q = \bigcup_{i \in [n]} Q_i$ of join queries is $\mathrm{w}(Q) \stackrel{\text{def}}{=} \max_{i \in [n]} \mathrm{w}(Q_i)$.

If a query is a union of join queries, its AGM-bound (or fractional hypertree width) is the maximum AGM-bound (or fractional hypertree width) over its components.

Computational Model. We consider the RAM model of computation where schemas and data values are stored in registers of logarithmic size and operations on them can be done in constant time. We assume that each materialized relation with schema $\boldsymbol{X}$ is implemented by a data structure of size $O(|R|)$ that can: (1) look up, insert, and delete tuples in constant time, and (2) enumerate the tuples in R with constant delay. Given $\boldsymbol{Y} \subseteq X$ and $\boldsymbol{t} \in \mathrm{Dom}(\boldsymbol{Y})$, the data structure can: (1) check $\boldsymbol{t} \in \pi_{\boldsymbol{Y}} R$ in constant time; and (2) enumerate the tuples in $\sigma_{\boldsymbol{Y}=\boldsymbol{t}} R$ with constant delay.

3 Problem Setting

We consider natural join queries, or equivalently, full conjunctive queries and refer to them as *queries*. We investigate the incremental view maintenance (IVM) of a query Q under *single-tuple updates* to an initially empty database $\mathcal{D}$. A single-tuple update is an insert or a delete of a tuple into a relation in $\mathcal{D}$. We consider four variants of the IVM problem depending on the update and enumeration mode. With regard to updates, we distinguish between the *insert-only* setting, where we only allow single-tuple inserts, and the *insert-delete* setting, where we allow both single-tuple inserts and single-tuple deletes. With regard to enumeration, we distinguish between the *full* enumeration and the *delta* enumeration, where, after each update, the full query result and respectively the change to the query result at the current time can be enumerated with constant delay. The four IVM variants are parameterized by a query Q and take as input a database $\mathcal{D}$, whose relations are initially empty, and a stream $\delta_1\mathcal{D}, \delta_2\mathcal{D}, \ldots, \delta_N\mathcal{D}$ of N single-tuple updates to $\mathcal{D}$. Neither N nor the updates are known in advance. The updates arrive one by one. Let $\mathcal{D}^{(0)} \stackrel{\text{def}}{=} \emptyset, \mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(N)}$ be the sequence of database versions, where for each $\tau \in [N]$, the database version $\mathcal{D}^{(\tau)}$ results from applying the update $\delta_\tau\mathcal{D}$ to the previous version $\mathcal{D}^{(\tau-1)}$.

The task of the first IVM variant is to support constant-delay enumeration of the full query result $Q(\mathcal{D}^{(\tau)})$ after each update:

Problem:	$\mathrm{IVM}^{\pm}[Q]$
Parameter:	Query Q
Given:	An initially empty database $\mathcal{D}$ and a stream $\delta_1\mathcal{D}, \ldots, \delta_N\mathcal{D}$ of N single-tuple updates to $\mathcal{D}$ where N is *not* known in advance.
Task:	Support constant-delay enumeration of $Q(\mathcal{D}^{(\tau)})$ after each update $\delta_\tau\mathcal{D}$

The task of the second IVM variant is to support constant-delay enumeration of the *change* to the query result after each update, given by Eq. (3):

Problem:	$\text{IVM}^{\pm}_{\delta}[Q]$
Parameter:	Query Q
Given:	An initially empty database $\mathcal{D}$ and a stream $\delta_1\mathcal{D}, \ldots, \delta_N\mathcal{D}$ of N single-tuple updates to $\mathcal{D}$ where N is *not* known in advance.
Task:	Support constant-delay enumeration of $\delta_\tau Q(\mathcal{D})$ after each update $\delta_\tau\mathcal{D}$

The other two variants, denoted as $\text{IVM}^{+}[Q]$ and $\text{IVM}^{+}_{\delta}[Q]$, are identical to $\text{IVM}^{\pm}[Q]$ and $\text{IVM}^{\pm}_{\delta}[Q]$, respectively, except that the updates $\delta_\tau\mathcal{D}$ are restricted to be single-tuple *inserts* (no deletes are allowed). In all four variants, we are interested in the *update time*, i.e., the time to process a single-tuple update.

Example 3.1. Consider the triangle query $Q_\triangle$ from Eq. (1). Suppose we have the update stream of length 8 given in the second column of Table 1. The third and fourth column show the full result and the delta result, respectively, after each update. We will use this example as a running example throughout the paper.

τ	$\delta_\tau\mathcal{D}$	$Q(\mathcal{D}^{(\tau)})$	$\delta_\tau Q(\mathcal{D})$
1	$+R(a_1, b_1)$	$\{\}$	$\{\}$
2	$+S(b_1, c_1)$	$\{\}$	$\{\}$
3	$+T(a_1, c_1)$	$\{(a_1, b_1, c_1)\}$	$\{+Q(a_1, b_1, c_1)\}$
4	$+S(b_2, c_1)$	$\{(a_1, b_1, c_1)\}$	$\{\}$
5	$-S(b_1, c_1)$	$\{\}$	$\{-Q(a_1, b_1, c_1)\}$
6	$-S(b_2, c_1)$	$\{\}$	$\{\}$
7	$-T(a_1, c_1)$	$\{\}$	$\{\}$
8	$-R(a_1, b_1)$	$\{\}$	$\{\}$

Table 1. An update sequence for the query $Q_\triangle$ in Eq. (1). The last two columns show the full and delta result after each update.

We obtain lower bounds on the update time for our IVM problems by reductions from the following static query evaluation problem:

Problem:	Eval$[Q]$
Parameter:	Query Q
Given:	Database $\mathcal{D}$
Task:	Compute $Q(\mathcal{D})$

For this problem, we are interested in the time to compute $Q(\mathcal{D})$.

Definition 3.2 ($\mathsf{w}(Q)$ and $\omega(Q)$). Given a query Q, let $\mathsf{w}(Q)$ denote the *fractional hypertree width* of Q (see Section 2). Let $\omega(Q)$ denote the *smallest exponent* κ such that Eval$[Q]$ has an algorithm with runtime $O(|\mathcal{D}|^{\kappa+o(1)} + |Q(\mathcal{D})|)$ on any input database $\mathcal{D}$. Given a union of queries $Q = \bigvee_i Q_i$, we define $\mathsf{w}(Q)$ and $\omega(Q)$ to be the maximum of $\mathsf{w}(Q_i)$ and $\omega(Q_i)$, respectively, over all i. [1]

The function $\omega(Q)$ is not known in general. However, we know that $\omega(Q) \leq \mathsf{w}(Q)$; see e.g. [4]. More strongly, $\omega(Q)$ is upper bounded by the *submodular width* of Q [5, 6, 31][2]. For acyclic queries, we have $\omega(Q) = 1$ [46]. For the triangle query, $\omega(Q_\triangle) \geq \frac{4}{3}$ modulo the 3SUM conjecture [39]. In

[1]The function $\omega(Q)$ is defined analogously to the constant ω which is the best exponent for matrix multiplication.
[2][5, 6] provide a query evaluation algorithm with runtime $O(|\mathcal{D}|^{\text{subw}(Q)} \cdot \text{polylog}|\mathcal{D}| + |Q(\mathcal{D})|) = O(|\mathcal{D}|^{\text{subw}(Q)+o(1)} + |Q(\mathcal{D})|)$, thanks to the extra $+o(1)$ in the exponent. $\text{subw}(Q)$ denotes the submodular width of Q.

this paper, we prove lower bounds on the IVM problems in terms of the function $\omega(Q)$. These lower bounds are not conditioned on a fine-grained complexity conjecture, like the OMv conjecture [17]. Instead, they apply no matter what the value of $\omega(Q)$ turns out to be. In that sense, they can be thought of as unconditional. However, they are relatively straightforward to prove and are mostly meant to justify the matching upper bounds.

Complexity Measures. For all problems introduced above, we consider the query to be fixed. Moreover, we say that the *amortized* update time for the τ-th single-tuple update is $f(N, |\mathcal{D}^{(\tau)}|)$ for some function f, if the total time to process all N updates in the stream is upper bounded by $\sum_{\tau \in [N]} f(N, |\mathcal{D}^{(\tau)}|)$. In the insert-only setting, the database size $|\mathcal{D}^{(\tau)}|$ at time τ is always equal to τ. Hence, for $\text{IVM}^+[Q]$ and $\text{IVM}^+_\delta[Q]$, we ignore the database size and only measure the update time as a function of the length N of the update stream. In the insert-delete setting, the database size at time τ is upper bounded by τ and could be much smaller than τ. As a result, for $\text{IVM}^\pm[Q]$ and $\text{IVM}^\pm_\delta[Q]$, it is more natural to ignore N and only measure the update time as a function of $|\mathcal{D}^{(\tau)}|$. For the four IVM problems, the constant enumeration delay is *not* allowed to be amortized, regardless of whether the update time is amortized or not. For the problem Eval, we measure the computation time as a function of the size of the input database $\mathcal{D}$. We use the O-notation to state worst-case bounds and $\widetilde{O}$-notation to hide a polylogarithmic factor in N, $|\mathcal{D}^{(\tau)}|$, or $|\mathcal{D}|$.

Multivariate Extensions. For the purpose of stating our results, we introduce the following central concept, which gives our approach its name, MultiVariate IVM (MVIVM for short).

Definition 3.3 (Multivariate extension $\widehat{Q}$ of a query Q). Consider a query $Q(\boldsymbol{X}) = R_1(\boldsymbol{X}_1) \wedge \ldots \wedge R_k(\boldsymbol{X}_k)$ where $\boldsymbol{X} = \bigcup_{i \in [k]} \boldsymbol{X}_i$, and fresh variables $Z_1, \ldots, Z_k$ that do *not* occur in Q. (In this paper, we drop head variables $\boldsymbol{X}$ and write Q instead of $Q(\boldsymbol{X})$ for brevity.) Let Σ_k be the set of permutations of the set $[k]$. For any permutation $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_k) \in \Sigma_k$, we denote by $\widehat{Q}_{\boldsymbol{\sigma}}$ the query that results from Q by extending the schema of each atom $R_{\sigma_i}(\boldsymbol{X}_{\sigma_i})$ with the variables $Z_1, \ldots, Z_i$:

$$\widehat{Q}_{\boldsymbol{\sigma}} = \widehat{R}_{\sigma_1}(Z_1, \boldsymbol{X}_{\sigma_1}) \wedge \widehat{R}_{\sigma_2}(Z_1, Z_2, \boldsymbol{X}_{\sigma_2}) \wedge \cdots \wedge \widehat{R}_{\sigma_k}(Z_1, \ldots, Z_k, \boldsymbol{X}_{\sigma_k}) \tag{6}$$

The *multivariate extension* $\widehat{Q}$ of Q is the union of $\widehat{Q}_{\boldsymbol{\sigma}}$ over all permutations $\boldsymbol{\sigma} \in \Sigma_k$:

$$\widehat{Q} = \bigvee_{\boldsymbol{\sigma} \in \Sigma_k} \widehat{Q}_{\boldsymbol{\sigma}} \tag{7}$$

We call each $\widehat{Q}_{\boldsymbol{\sigma}}$ a *component* of $\widehat{Q}$.

Example 3.4. The multivariate extension $\widehat{Q}_\triangle$ of $Q_\triangle$ from Eq. (1) consists of 6 components $\widehat{Q}_{123}, \widehat{Q}_{132}, \widehat{Q}_{213}, \widehat{Q}_{231}, \widehat{Q}_{312}$ and $\widehat{Q}_{321}$. The first component, $\widehat{Q}_{123}$, orders the input relations as (R, S, T) and applies Eq. (6) by adding $\{Z_1\}$ to R, $\{Z_1, Z_2\}$ to S, and $\{Z_1, Z_2, Z_3\}$ to T, thus resulting in $\widehat{R}(Z_1, A, B)$, $\widehat{S}(Z_1, Z_2, B, C)$, and $\widehat{T}(Z_1, Z_2, Z_3, A, C)$ respectively. Below, we add the subscript 123 to each relation to indicate the component for later convenience:

$$\widehat{Q}_{123} = \widehat{R}_{123}(Z_1, A, B) \wedge \widehat{S}_{123}(Z_1, Z_2, B, C) \wedge \widehat{T}_{123}(Z_1, Z_2, Z_3, A, C) \tag{8}$$

4 Overview of Main Results

In this section, we overview our main results and discuss their implications.

4.1 Insert-Only Setting

In the insert-only setting, we show that off-the-shelf worst-case optimal join algorithms, e.g., LeapFrogTrieJoin [43], can be used to achieve the best known amortized update time for the

dynamic query evaluation of arbitrary join queries, so including the *cyclic* queries, while supporting constant delay for full/delta enumeration. Specifically, we give an upper bound on the single-tuple update time in terms of the fractional hypertree width $\mathrm{w}(Q)$ of Q:

Theorem 4.1. *For any query Q, both $IVM^+[Q]$ and $IVM^+_\delta[Q]$ can be solved with $O(N^{\mathrm{w}(Q)-1})$ amortized update time and non-amortized constant enumeration delay, where N is the number of single-tuple inserts.*

The upper bound in Theorem 4.1 is met by our algorithm outlined in Sec. 5. It uses worst-case optimal join algorithms and tree decompositions. Amortization is necessary as some inserts may be costly, while many others are necessarily relatively cheaper, so the average insert time matches the cost of computing a factorized representation of the query result divided by the number of inserts.

Since every acyclic query has a fractional hypertree width of one, Theorem 4.1 implies that every acyclic query can be maintained with amortized constant time per insert:

Corollary 4.2 (Theorem 4.1). *For any acyclic query Q, $IVM^+[Q]$ and $IVM^+_\delta[Q]$ can be solved with $O(1)$ amortized update time and non-amortized constant enumeration delay.*

Corollary 4.2 recovers the amortized constant update time for acyclic joins from prior work [44]. It also shows that the insert-only setting can be computationally cheaper than the insert-delete setting investigated in prior work [9]: In the insert-delete setting, the non-hierarchical acyclic queries cannot admit $O(|\mathcal{D}|^{1/2-\gamma})$ update time (while keeping the enumeration delay constant) for any database $\mathcal{D}$ and $\gamma > 0$ [9], conditioned on the OMv conjecture.

We accompany the upper bound from Theorem 4.1 with a lower bound on the insertion time in terms of the lower bound $\omega(Q)$ on the static evaluation of Q (Definition 3.2). Unlike prior lower bounds [9, 21], this lower bound is not conditioned on a fine-grained complexity conjecture, such as the OMv conjecture [17]. However, its proof is straightforward by viewing static evaluation as a stream of inserts. It is meant to show that Theorem 4.1 is optimal up to the gap between $\mathrm{w}(Q)$ and $\omega(Q)$. [3] It also applies to *both* amortized and non-amortized update time and enumeration delay. [4]

Proposition 4.3. *For any query Q and any constant $\gamma > 0$, neither $IVM^+[Q]$ nor $IVM^+_\delta[Q]$ can be solved with $\widetilde{O}(N^{\omega(Q)-1-\gamma})$ (amortized) update time and (non-amortized) constant enumeration delay.*

4.2 Insert-Delete Setting

In the insert-delete setting, our approach can maintain arbitrary join queries, and in particular any cyclic join query, with update times that can be asymptotically lower than recomputation. In particular, we give an upper bound on the update time for a query Q in terms of the fractional hypertree width of the multivariate extension $\widehat{Q}$ of Q (Definition 3.3):

Theorem 4.4. *For any query Q, both $IVM^\pm[Q]$ and $IVM^\pm_\delta[Q]$ can be solved with $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})-1})$ amortized update time and non-amortized constant enumeration delay, where $\widehat{Q}$ is the multivariate extension of Q, and $|\mathcal{D}^{(\tau)}|$ is the current database size at update time τ.*

Sec. 7.1 overviews our algorithm that meets the upper bound in Theorem 4.4. It involves *intersection joins* (Sec. 6). The following statements shed light on the relationship between a query Q and its multivariate extension $\widehat{Q}$ (their proofs can be found in [8]):

- Q is hierarchical if and only if its multivariate extension $\widehat{Q}$ is acyclic, or equivalently $\mathrm{w}(\widehat{Q}) = 1$.

[3] More broadly, our lower bounds are meant to introduce a new framework to assess the optimality of algorithms in database theory, in terms of the function $\omega(Q)$.

[4] Note that a lower bound on the amortized time also implies the same lower bound on the worst-case time. This is because if an amortized algorithm does not exist, then a worst-case algorithm does not exist either.

- If Q is non-hierarchical, then $\mathsf{w}(\widehat{Q}) \geq \frac{3}{2}$.
- If Q is Loomis-Whitney of *any* degree, then $\mathsf{w}(\widehat{Q}) = \frac{3}{2}$.
- For *any* query Q, we have $\mathsf{w}(Q) \leq \mathsf{w}(\widehat{Q}) \leq \mathsf{w}(Q) + 1$.

Immediate corollaries of these statements are that, in the insert-delete setting, our approach needs: (1) amortized $\widetilde{O}(1)$ update time for hierarchical queries; and (2) amortized $\widetilde{O}(|\mathcal{D}|^{1/2})$ update time for the triangle query, which is the Loomis-Whitney query of degree 3. These update times mirror those given in prior work on q-hierarchical queries [9] and the full triangle query [22], albeit our setting is more restricted: it considers full conjunctive queries and initially empty databases and the update times are amortized and have a polylog factor.

A close analysis of the lower bound proof for non-hierarchical queries in prior work [9] reveals that for any non-hierarchical query Q and any $\gamma > 0$, there is no algorithm that solves $\mathrm{IVM}^{\pm}[Q]$ or $\mathrm{IVM}^{\pm}_{\delta}[Q]$ with *amortized* $\widetilde{O}(N^{1/2-\gamma})$ update time, unless the OMv-conjecture fails [8]. Following Theorem 4.4 and the fact that $\mathsf{w}(\widehat{Q}) = \frac{3}{2}$ for any Loomis-Whitney query Q, we conclude that both $\mathrm{IVM}^{\pm}[Q]$ and $\mathrm{IVM}^{\pm}_{\delta}[Q]$ can be solved with $\widetilde{O}(|\mathcal{D}|^{1/2})$ amortized update time, which is optimal, unless the OMv conjecture fails [8].

First-order IVM, i.e., delta queries, and even higher-order IVM, i.e., delta queries with materialized views, cannot achieve the update time of our approach. It was already discussed in prior work that for the triangle join query, both IVM approaches need linear update time per single-tuple update [21, 22]. See Appendix A. The IVM^{ϵ} approach resorts to a heavy/light partitioning argument that is tailored to the triangle query and does not generalize to other cyclic queries [21, 22]. Instead, our approach solves this problem more systematically and for *any* query Q by translating the temporal dimension (the tuple lifespan as defined by its insert and possible delete) into spatial attributes (the multivariate encoding of the tuple lifespan), taking a tree decomposition of the multivariate extension $\hat{Q}$ of Q, and by materializing and maintaining the bags of this tree decomposition.

We complement the upper bound in Theorem 4.4 with a lower bound in terms of the lower bound $\omega(\widehat{Q}_{\boldsymbol{\sigma}})$ for the static evaluation of any component $\widehat{Q}_{\boldsymbol{\sigma}}$ of $\widehat{Q}$. It is not conditioned of a complexity conjecture. However, it only applies to the $\mathrm{IVM}^{\pm}_{\delta}[Q]$ problem. In particular, it is meant to show that our upper bound from Theorem 4.4 is tight for $\mathrm{IVM}^{\pm}_{\delta}[Q]$ up to the gap between $\mathsf{w}(\widehat{Q})$ and $\omega(\widehat{Q}_{\boldsymbol{\sigma}})$. It also applies to both amortized and non-amortized update time and enumeration delay. Sec. 7.2 gives the high-level idea:

Theorem 4.5. *Let Q be a query and $\widehat{Q}_{\boldsymbol{\sigma}}$ any component of its multivariate extension. For any constant $\gamma > 0$, $\mathrm{IVM}^{\pm}_{\delta}[Q]$ cannot be solved with (amortized) update time $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\omega(\widehat{Q}_{\boldsymbol{\sigma}})-1-\gamma})$ and (non-amortized) constant enumeration delay.*

5 IVM: Insert-Only Setting

In this section, we give an overview of our algorithm for $\mathrm{IVM}^{+}[Q]$ that meets the upper bound of Theorem 4.1. We leave the details and proofs to [8]. We start with the following lemma (Recall notation from Sec. 2):

Lemma 5.1. *Given a query Q, an initially empty database $\mathcal{D}^{(0)}$, and a stream of N single-tuple inserts, we can compute the* new *output tuples $\delta_\tau Q(D)$ after every insert $\delta_\tau \mathcal{D}$, where the total computation time over* all *inserts is $O(N + \mathsf{AGM}(Q, \mathcal{D}^{(N)}))$.*

The total computation time above is the same as the AGM bound [7] of Q over the final database $\mathcal{D}^{(N)}$. In particular, even in the static setting where $\mathcal{D}^{(N)}$ is given upfront, the output size $|Q(\mathcal{D}^{(N)})|$ can be as large as $\mathsf{AGM}(Q, \mathcal{D}^{(N)})$ in the worst-case. Moreover, worst-case optimal join algorithms cannot beat this runtime [33, 34, 43]. The above lemma is proved based on the *query decomposition*

lemma [33, 35], which says the following. Let Q be a query and let $\boldsymbol{Y} \subseteq \mathsf{vars}(Q)$. Then, the AGM bound of Q can be decomposed into a sum of AGM bounds of "residual" queries: one query $Q \ltimes \boldsymbol{y}$ for each tuple $\boldsymbol{y}$ over the variables $\boldsymbol{Y}$.

Lemma 5.2 (Query Decomposition Lemma [33, 35]). *Given a query Q and a subset $\boldsymbol{Y} \subseteq \mathsf{vars}(Q)$, let $\left(\lambda_{R(\boldsymbol{X})}\right)_{R(\boldsymbol{X}) \in \mathsf{at}(Q)}$ be a fractional edge cover of Q. Then, the following inequality holds:*

$$\sum_{\boldsymbol{y} \in \mathsf{Dom}(\boldsymbol{Y})} \underbrace{\prod_{R(\boldsymbol{X}) \in \mathsf{at}(Q)} |R(\boldsymbol{X}) \ltimes \boldsymbol{y}|^{\lambda_{R(\boldsymbol{X})}}}_{\text{AGM-bound of } Q \ltimes \boldsymbol{y}} \leq \underbrace{\prod_{R(\boldsymbol{X}) \in \mathsf{at}(Q)} |R|^{\lambda_{R(\boldsymbol{X})}}}_{\text{AGM-bound of } Q} \tag{9}$$

In the above, $\boldsymbol{y} \in \mathsf{Dom}(\boldsymbol{Y})$ indicates that the tuple $\boldsymbol{y}$ has schema $\boldsymbol{Y}$. Moreover, $R(\boldsymbol{X}) \ltimes \boldsymbol{y}$ denotes the *semijoin* of the atom $R(\boldsymbol{X})$ with the tuple $\boldsymbol{y}$.

Example 5.3 (for Lemma 5.1). Consider the triangle query $Q_\triangle$ from Eq. (1). Lemma 5.1 says that given a stream of N single-tuple inserts into R, S, and T, we can compute the new output tuples after every insert in a total time of $O(N^{3/2})$. To achieve this, we need to maintain two indices for $R(A, B)$: one index that sorts R first by A and then by B, while the other sorts R first by B and then by A. For every insert to R, we update the two indices simultaneously. Similarly, we maintain two indices for each of S and T.

Following notation from Sec. 2, let $R^{(0)} \stackrel{\text{def}}{=} \emptyset, R^{(1)}, \ldots, R^{(N)}$ be the sequence of versions of R after each insert, and the same goes for S and T. Suppose that the τ-th insert in the stream is inserting a tuple (a, b) into $R(A, B)$. The new output tuples that are added due to this insert correspond to the output of the following query: (Below, we drop the database instance $\mathcal{D}$ from the notation $\delta_\tau Q_\triangle(\mathcal{D})$ since $\mathcal{D}$ is clear from the context.)

$$\delta_\tau Q_\triangle(A, B, C) \stackrel{\text{def}}{=} \sigma_{B=b} S^{(\tau)}(B, C) \wedge \sigma_{A=a} T^{(\tau)}(A, C)$$

The output size of this query is upper bounded by:

$$\min(|\sigma_{B=b} S^{(\tau)}(B, C)|, |\sigma_{A=a} T^{(\tau)}(A, C)|) \leq \sqrt{|\sigma_{B=b} S^{(\tau)}(B, C)| \cdot |\sigma_{A=a} T^{(\tau)}(A, C)|} \tag{10}$$

Moreover, the above query can be computed in time within $O(1)$ factor from the quantity in Eq. (10). To achieve this time, we need to use the index for $S(B, C)$ that is indexed by B first, and we also need the index for $T(A, C)$ that is indexed by A first. Because $S^{(\tau)} \subseteq S^{(N)}$ and $T^{(\tau)} \subseteq T^{(N)}$, the quantity in Eq. (10) is bounded by:

$$\sqrt{|\sigma_{B=b} S^{(N)}(B, C)| \cdot |\sigma_{A=a} T^{(N)}(A, C)|} \tag{11}$$

The query decomposition lemma says that the sum of the quantity in Eq. (11) over all $(a, b) \in R^{(N)}$ is bounded by $\sqrt{|R^{(N)}| \cdot |S^{(N)}| \cdot |T^{(N)}|} \leq N^{3/2}$. Hence, all inserts into R take time $O(N^{3/2})$.

Now suppose that the τ-th insert in the stream is inserting a tuple (b, c) into $S(B, C)$. To handle this insert, we compute the query:

$$\delta_\tau Q_\triangle(A, B, C) \stackrel{\text{def}}{=} \sigma_{B=b} R^{(\tau)}(A, B) \wedge \sigma_{C=c} T^{(\tau)}(A, C)$$

To compute this query in the desired time, we need to use the index for $T(A, C)$ that is indexed by C first. This is why we need to maintain two indices for $T(A, C)$. The same goes for R and S.

Instead of the AGM-bound, the upper bound in Theorem 4.1 is given in terms of the fractional hypertree width of Q. To achieve this bound, we take an optimal tree decomposition of Q and maintain a materialized relation for every bag in the tree decomposition using Lemma 5.1. In order

to support constant-delay enumeration of the output, we "calibrate" the bags by semijoin reducing adjacent bags with one another. The following example illustrates this idea.

Example 5.4 (for Theorem 4.1, IVM$^+$). Suppose we want to solve IVM$^+[Q]$ for the following query Q consisting of two adjacent triangles: One triangle over $\{A, B, C\}$ and the other over $\{B, C, D\}$.

$$Q(A, B, C, D) = R(A, B) \wedge S(B, C) \wedge T(A, C) \wedge U(B, D) \wedge V(C, D) \tag{12}$$

The above query Q has a fractional hypertree width of $\frac{3}{2}$. In particular, one optimal tree decomposition of Q consists of two bags: One child bag $B_1 = \{A, B, C\}$ and another root bag $B_2 = \{B, C, D\}$. Theorem 4.1 says that given a stream of N single-tuple inserts, Q can be updated in a total time of $O(N^{3/2})$ where we can do constant-delay enumeration of the output after every insert. To achieve this, we maintain the following query plan:

$$\begin{aligned} Q_1(A, B, C) &= R(A, B) \wedge S(B, C) \wedge T(A, C) \\ P_1(B, C) &= Q_1(A, B, C) \\ Q_2'(B, C, D) &= P_1(B, C) \wedge S(B, C) \wedge U(B, D) \wedge V(C, D) \end{aligned}$$

By Lemma 5.1, Q_1 above can be updated in a total time of $O(N^{3/2})$, as shown in Example 5.3. Also, P_1 can be updated in the same time because it is just a projection of Q_1. Regardless of the size of P_1, the query Q_2' can be updated in a total time of $O(N^{3/2})$ because its AGM bound is upper bounded by the input relations S, U and V. To do constant-delay enumeration of $Q(A, B, C, D)$, we enumerate (b, c, d) from Q_2', and for every (b, c), we enumerate the corresponding A-values from Q_1. Note that at least one A-value must exist because Q_2' includes P_1, which is the projection of Q_1.

In the above example, we only calibrate bottom-up from the leaf Q_1 to the root Q_2'. However, this is not sufficient for IVM$^+_\delta[Q]$. Instead, we also need to calibrate top-down, as the following example demonstrates.

Example 5.5 (for Theorem 4.1, IVM$^+_\delta$). Consider again the query (12) from Example 5.4. Suppose now that we want to extend our solution from Example 5.4 to the IVM$^+_\delta[Q]$ problem. In particular, suppose we have an insert of a tuple (a, b) into R above. In order to enumerate the new output tuples corresponding to (a, b), we have to start our enumeration from Q_1 as the root. For that purpose, Q_1 also needs to be calibrated with Q_2'. To that end, in addition to Q_1, P_1 and Q_2' that are defined in Example 5.4, we also need to maintain the following relations:

$$\begin{aligned} P_1''(B, C) &= Q_2'(B, C, D) \\ Q_1''(A, B, C) &= Q_1(A, B, C) \wedge P_1''(B, C) \end{aligned}$$

The AGM bound of Q_1'' is still $N^{3/2}$, and so is the AGM bound of P_1''. Hence, we can maintain all inserts into them in a total time of $O(N^{3/2})$. Moreover, note that by definition, Q_1'' must be the projection of $Q(A, B, C, D)$ onto $\{A, B, C\}$.

Now, suppose we have an insert of a tuple $\boldsymbol{t}$ into either one of the relations S, U or V. Then, we can enumerate the new output tuples that are added due to this insert by starting from Q_2', just like we did in Example 5.4. However, if we have inserts into R or T, then we have to start our enumeration from Q_1''. For each output tuple (a, b, c) of Q_1'' that joins with $\boldsymbol{t}$, we enumerate the corresponding D-values from Q_2'. There must be at least one D-value because Q_1'' is the projection of the output Q onto $\{A, B, C\}$.

6 Technical Tools: Intersection Joins

Before we explain our upper and lower bounds for IVM in the insert-delete setting in Sec. 7, we review in this section some necessary background on intersection joins in the *static* setting.

Definition 6.1 (Intersection queries [3]). An *intersection query* is a (full conjunctive) query that contains *interval variables*, which are variables whose domains are discrete intervals over the natural numbers. For instance, the discrete interval $[3, 7]$ consists of the natural numbers from 3 to 7. For better distinction, we denote an interval variable by $[X]$ and call the variables that are not interval variables *point variables*. A value of an interval variable $[X]$ is denoted by $[x]$, which is an interval. Two or more intervals *join* if they *intersect*. In particular, the semantics of an intersection query $Q = R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k)$ is defined as follows. A tuple $\boldsymbol{t}$ over the schema $\text{vars}(Q)$ is in the result of Q if there are tuples $\boldsymbol{t}_i \in R_i$ for $i \in [k]$ such that for all $X \in \text{vars}(Q)$, it holds:

- if X is a point variable, then the set of points $\{\boldsymbol{t}_i(X) \mid i \in [k] \wedge X \in \boldsymbol{X}_i\}$ consists of a single value, which is the point $\boldsymbol{t}(X)$;
- if $[X]$ is an interval variable, then the intervals in the set $\{\boldsymbol{t}_i([X]) \mid i \in [k] \wedge [X] \in \boldsymbol{X}_i\}$ have a non-empty intersection, which is the interval $\boldsymbol{t}([X])$.

In this paper, we are only interested in a special class of intersection queries, defined below:

Definition 6.2 (The time extension $\overline{Q}$ of a query Q). Let $Q = R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k)$ be a query where $\text{vars}(Q)$ are all point variables, and let $[Z]$ be a new interval variable. The *time* extension $\overline{Q}$ of Q is an intersection query that results from Q by adding $[Z]$ to the schema of each atom:

$$\overline{Q} = \overline{R}_1([Z], \boldsymbol{X}_1) \wedge \cdots \wedge \overline{R}_k([Z], \boldsymbol{X}_k) \tag{13}$$

A query $\overline{Q}$ is called a *time extension* if it is the time extension of some query Q.

We call $\overline{Q}$ the "time extension" of Q because in Sec. 7.1, we use the interval $[Z]$ in $\overline{Q}$ to represent the "lifespan" of tuples in Q.

Example 6.3. Consider this intersection query, which is the time extension of $Q_\triangle$ from Eq. (1):

$$\overline{Q}_\triangle([Z], A, B, C) = \overline{R}([Z], A, B) \wedge \overline{S}([Z], B, C) \wedge \overline{T}([Z], A, C) \tag{14}$$

In the above, $[Z]$ is an interval variable while A, B, C are point variables. Consider the database instance $\overline{\mathcal{D}}$ depicted in Figure 1. The output of $\overline{Q}_\triangle$ on this instance is depicted in Figure 1d.

$[Z]$	A	B
$[1, 8]$	a_1	b_1

(a) $\overline{R}$

$[Z]$	B	C
$[2, 5]$	b_1	c_1
$[4, 6]$	b_2	c_1

(b) $\overline{S}$

$[Z]$	A	C
$[3, 7]$	a_1	c_1

(c) $\overline{T}$

$[Z]$	A	B	C
$[3, 5]$	a_1	b_1	c_1

(d) $\overline{Q}_\triangle$

Fig. 1. A database instance $\overline{\mathcal{D}}$ for the query $\overline{Q}_\triangle$ in Eq. (14).

Let $\overline{Q}$ be the time extension of a query Q. Prior work reduces the evaluation of $\overline{Q}$ to the evaluation of $\widehat{Q}$, where $\widehat{Q}$ is the multivariate extension of Q (Eq. (7)), whose variables are all point variables [3].

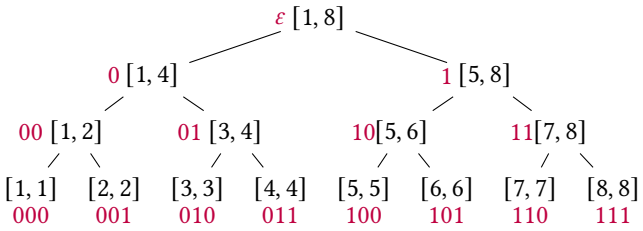


Fig. 2. A segment tree $\mathcal{T}_8$. Each node is identified by a bitstring.

Moreover, it is shown in [3] that this reduction is *optimal* (up to a $\widetilde{O}(1)$ factor). In particular, the query $\overline{Q}$ is exactly as hard as the hardest component $\widehat{Q}_{\sigma}$ in $\widehat{Q}$. This is shown by a backward reduction from (the static evaluation of) each component $\widehat{Q}_{\sigma}$ to $\overline{Q}$. We summarize both reductions below and defer the details to Appendix B and [3]. We rely on both reductions to prove upper and lower bounds for IVM in the insert-delete setting (Sec. 4.2).

6.1 Reduction from $\overline{Q}$ to $\widehat{Q}$

Suppose we want to evaluate $\overline{Q}$ on a given database instance $\overline{\mathcal{D}}$. We construct a *segment tree*, $\mathcal{T}_N$, over the interval variable $[Z]$, where the parameter N is roughly the maximum number of different Z-values. A segment tree $\mathcal{T}_N$ is a balanced binary tree with N leaves. Each node corresponds to an interval. The root corresponds to the interval $[1, N]$, its left and right child correspond to the intervals $[1, N/2]$ and $[N/2 + 1, N]$ respectively, and so on. Each node can be represented by a bitstring of length at most $\log_2 N$. Figure 2 depicts the segment tree $\mathcal{T}_8$. Each interval $[z]$ can be broken down into at most $O(\log N)$ nodes in $\mathcal{T}_N$. We refer to these nodes as the *canonical partition* of $[z]$, and denote them by $\mathrm{CP}_N([z])$. For example, the canonical partitions of $[2, 8]$ and $[2, 5]$ in the segment tree $\mathcal{T}_8$ are:

$$\mathrm{CP}_8([2, 8]) = \{001, 01, 1\}, \qquad \mathrm{CP}_8([2, 5]) = \{001, 01, 100\} \tag{15}$$

The canonical partition of the intersection of some intervals is roughly the intersection of their canonical partitions [3]. Two (or more) nodes in the segment tree correspond to overlapping intervals if and only if one node is an ancestor of the other, which can only happen if one of the two corresponding bitstrings is a prefix of the other [3]. For example, the nodes labeled by 1 and 100 in Figure 2 correspond to two overlapping intervals, namely $[5, 8]$ and $[5, 5]$, because the string 1 is a prefix of 100. Therefore, testing intersections can be reduced to testing whether some bitstrings form a chain of prefixes. Armed with this idea, we convert $\overline{\mathcal{D}}$ into a database instance $\widehat{\mathcal{D}}$ for $\widehat{Q}$, called the *canonical partition* of $\overline{\mathcal{D}}$ and denoted by $\mathrm{CP}_N(\overline{\mathcal{D}})$. There is a mapping between the outputs of $\overline{Q}(\overline{\mathcal{D}})$ and $\widehat{Q}(\widehat{\mathcal{D}})$, as the following example shows.

Example 6.4. Suppose we want to reduce $\overline{Q}_{\triangle}$ from Eq. (14) to $\widehat{Q}_{\triangle}$ from Example 3.4. In particular, we are given the database instance $\overline{\mathcal{D}}$ from Figure 1 and want to compute the corresponding instance $\widehat{\mathcal{D}}$ for $\widehat{Q}_{\triangle}$. We show how to construct $\widehat{R}_{123}, \widehat{S}_{123}, \widehat{T}_{123}$ for $\widehat{Q}_{123}$ in Eq. (8). The remaining $\widehat{Q}_{132}, \ldots, \widehat{Q}_{321}$ are similar. The query $\widehat{Q}_{123}$ is meant to test whether $\overline{R}, \overline{S}, \overline{T}$ contain three intervals $[z^R], [z^S], [z^T]$ whose canonical partitions contain bitstrings z^R, z^S, z^T where z^R is a prefix of z^S which is a prefix of z^T. This can only happen if there are bitstrings z_2, z_3 where $z^S = z^R \circ z_2$ and

$z^T = z^S \circ z_3$, where $\circ$ denotes string concatenation. To that end, we define:

$$\begin{aligned}
\widehat{R}_{123} &\stackrel{\text{def}}{=} \{(z_1, a, b) \mid \exists [z] : ([z], a, b) \in \overline{R} \wedge z_1 \in \mathrm{CP}_N([z])\} \\
\widehat{S}_{123} &\stackrel{\text{def}}{=} \{(z_1, z_2, b, c) \mid \exists [z] : ([z], b, c) \in \overline{S} \wedge (z_1 \circ z_2) \in \mathrm{CP}_N([z])\} \\
\widehat{T}_{123} &\stackrel{\text{def}}{=} \{(z_1, z_2, z_3, a, c) \mid \exists z : ([z], a, c) \in \overline{T} \wedge (z_1 \circ z_2 \circ z_3) \in \mathrm{CP}_N([z])\}
\end{aligned} \tag{16}$$

Note that $|\widehat{T}_{123}| = O(|\overline{T}| \cdot \mathrm{polylog}(N))$ and can be constructed in time $O(|\overline{T}| \cdot \mathrm{polylog}(N))$, because the height of the segment tree is $O(\log N)$. The same goes for $\widehat{R}_{123}$ and $\widehat{S}_{123}$. Let ε denote the empty string. Applying the above to $\overline{\mathcal{D}}$ from Figure 1, $\widehat{R}_{123}, \widehat{S}_{123}, \widehat{T}_{123}$ respectively contain the tuples: (ε, a_1, b_1), $(\varepsilon, 01, b_1, c_1)$, $(\varepsilon, 01, \varepsilon, a_1, c_1)$, which join together producing the output tuple $(\varepsilon, 01, \varepsilon, a_1, b_1, c_1)$ of $\widehat{Q}_{123}$. This is a witness to the output tuple $([3, 5], a_1, b_1, c_1)$ of $\overline{Q}_\triangle$. The full answer to $\overline{Q}_\triangle$, given by Figure 1d, can be retrieved from the union of the answers to the six queries $\widehat{Q}_{132}, \ldots, \widehat{Q}_{321}$.

6.2 Reduction from $\widehat{Q}$ to $\overline{Q}$

We now summarize the backward reduction from the static evaluation of each component $\widehat{Q}_\sigma$ of $\widehat{Q}$ back to $\overline{Q}$, based on [3]. We will utilize this reduction later to establish our lower bound in Sec. 7.2.

Example 6.5. Continuing with Example 6.3, let us consider the query $\widehat{Q}_{123}$ from (8). Let $\widehat{\mathcal{D}}'_{123}$ be an *arbitrary* database instance for $\widehat{Q}_{123}$. (Unlike Example 6.3, here we cannot make any assumption about how $\widehat{\mathcal{D}}'_{123}$ was constructed.) We show how to use an oracle for the intersection query $\overline{Q}_\triangle$ from (14) in order to compute $\widehat{Q}_{123}(\widehat{\mathcal{D}}'_{123})$ in the same time. WLOG we can assume that each value x that appears in $\widehat{\mathcal{D}}'_{123}$ is a bitstring of length exactly ℓ for some constant ℓ. (If a bitstring has length less than ℓ, we can pad it with zeros.) We construct a segment tree $\mathcal{T}_{N'}$ where $N' \stackrel{\text{def}}{=} 2^{3\ell}$, i.e. a segment tree of depth 3ℓ. (We chose 3 in this example because it is the number of atoms in $\widehat{Q}_{123}$.) Each node v in the segment tree corresponds to an interval that is contained in $[N']$ and is identified by a bitstring of length at most 3ℓ. Given a bitstring b of length at most 3ℓ, let $\mathrm{seg}_{N'}(b)$ denote the corresponding interval in the segment tree $\mathcal{T}_{N'}$. We construct the following database instance $\overline{\mathcal{D}}'$ for $\overline{Q}_\triangle$:

$$\begin{aligned}
\overline{R}' &= \{(\mathrm{seg}_{N'}(z_1), a, b) \mid (z_1, a, b) \in \widehat{R}'_{123}\} \\
\overline{S}' &= \{(\mathrm{seg}_{N'}(z_1 \circ z_2), b, c) \mid (z_1, z_2, b, c) \in \widehat{S}'_{123}\} \\
\overline{T}' &= \{(\mathrm{seg}_{N'}(z_1 \circ z_2 \circ z_3), a, c) \mid (z_1, z_2, z_3, a, c) \in \widehat{T}'_{123}\}
\end{aligned}$$

Following [3], we can show that there is a one-to-one mapping between the output tuples of $\overline{Q}_\triangle(\overline{\mathcal{D}}')$ and the output tuples of $\widehat{Q}_{123}(\widehat{\mathcal{D}}'_{123})$. This follows from the observation that the three intervals $\mathrm{seg}_{N'}(z_1^R)$, $\mathrm{seg}_{N'}(z_1^S \circ z_2^S)$, $\mathrm{seg}_{N'}(z_1^T \circ z_2^T \circ z_3^T)$ overlap if and only if $z_1^R = z_1^S = z_1^T$ and $z_2^S = z_2^T$. (Recall that the bitstrings $z_1^R, z_1^S, \ldots$ all have the same length ℓ.) Moreover note that $\overline{R}', \overline{S}', \overline{T}'$ have the same sizes as $\widehat{R}'_{123}, \widehat{S}'_{123}, \widehat{\mathcal{T}}'_{123}$ respectively and they can be constructed in linear time.

7 IVM: Insert-Delete Setting

In this section, we give a brief overview of how we obtain our upper and lower bounds for IVM in the insert-delete setting from Theorem 4.4 and 4.5, respectively. Details are deferred to Appendix C.

7.1 Upper bound for IVM$^{\pm}[Q]$

In order to prove Theorem 4.4, we start by describing an algorithm that meets a weaker version of the upper bound in Theorem 4.4, where the current database size $|\mathcal{D}^{(\tau)}|$ is replaced by the number of single-tuple updates N: (Note that $|\mathcal{D}^{(\tau)}| \leq N$ and could be unboundedly smaller.)

Lemma 7.1. *For any query Q, both IVM$^{\pm}[Q]$ and IVM$^{\pm}_{\delta}[Q]$ can be solved with $\widetilde{O}(N^{\mathrm{w}(\widehat{Q})-1})$ amortized update time and non-amortized constant enumeration delay, where $\widehat{Q}$ is the multivariate extension of Q and N is the number of single-tuple updates.*

In particular, we can show that we can use the algorithm from Lemma 7.1 as a black box in order to meet the stronger upper bound in Theorem 4.4. The main reason why the algorithm from Lemma 7.1 does not immediately meet the upper bound from Theorem 4.4 is that N can grow much larger than the database size $|\mathcal{D}|$, especially in the scenario where the update stream contains many deletes. However, whenever that happens, we can "reset" the algorithm from Lemma 7.1 by restarting from scratch and inserting all the tuples in the current database $\mathcal{D}$ as a stream of $|\mathcal{D}|$ inserts. If this is done carefully, then we can ensure that the total number of updates since the last reset is not significantly larger than $|\mathcal{D}|$.

We now focus on demonstrating our algorithm that proves Lemma 7.1 for IVM$^{\pm}[Q]$.

Example 7.2 (for Lemma 7.1). Suppose we want to solve the problem IVM$^{\pm}[Q_{\triangle}]$ for $Q_{\triangle}$ in Eq. (1), where we have a stream of N single-tuple inserts/deletes into R, S and T. For simplicity, assume that N is known in advance. (If N is not known in advance, we can initially assume N to be a constant and keep doubling N every time we exceed it.) We introduce a new interval variable $[Z]$ and use it to represent *time*. Specifically, $[Z]$ will represent the *lifespan* of every tuple in the database, in the spirit of temporal databases [20]. By adding $[Z]$ to every atom, we obtain the time extension $\overline{Q}_{\triangle}$ of $Q_{\triangle}$ in Eq. (14). Suppose that the τ-th update is an insert of a tuple (a, b) into R, i.e. $+R(a, b)$. Then, we apply the insert $+\overline{R}([\tau, \infty], a, b)$ into the time extension, indicating that the tuple (a, b) lives in R from time τ on (since we don't know yet its future deletion time). Now suppose that the τ'-th update (for some $\tau' > \tau$) is a delete of the same tuple (a, b) in R. Then, we replace the tuple $([\tau, \infty], a, b)$ in $\overline{R}$ with $([\tau, \tau'], a, b)$. We can extract the result of $Q_{\triangle}$ at any time τ by selecting output tuples $([z], a, b, c)$ of $\overline{Q}_{\triangle}$ where the interval $[z]$ contains the current time τ. Therefore, if we can efficiently maintain $\overline{Q}_{\triangle}$, we can efficiently maintain $Q_{\triangle}$. Table 2 shows the same stream of 8 updates into $Q_{\triangle}$ as Table 1 but adds the corresponding updates to $\overline{Q}_{\triangle}$. The final database $\overline{\mathcal{D}}$ after all updates have taken place is the same as the one shown in Figure 1.

τ	$\delta_\tau \mathcal{D}$	$\delta_\tau \overline{\mathcal{D}}$
1	$+R(a_1, b_1)$	$+\overline{R}([1, \infty], a_1, b_1)$
2	$+S(b_1, c_1)$	$+\overline{S}([2, \infty], b_1, c_1)$
3	$+T(a_1, c_1)$	$+\overline{T}([3, \infty], a_1, c_1)$
4	$+S(b_2, c_1)$	$+\overline{S}([4, \infty], b_2, c_1)$
5	$-S(b_1, c_1)$	$-\overline{S}([2, \infty], b_1, c_1), +\overline{S}([2, 5], b_1, c_1)$
6	$-S(b_2, c_1)$	$-\overline{S}([4, \infty], b_2, c_1), +\overline{S}([4, 6], b_2, c_1)$
7	$-T(a_1, c_1)$	$-\overline{T}([3, \infty], a_1, c_1), +\overline{T}([3, 7], a_1, c_1)$
8	$-R(a_1, b_1)$	$-\overline{R}([1, \infty], a_1, b_1), +\overline{R}([1, 8], a_1, b_1)$

Table 2. Updates to $Q_{\triangle}$ (1) (same as Table 1) along with the corresponding updates to $\overline{Q}_{\triangle}$ (14). At the end (i.e. $\tau = 8$), the database $\overline{\mathcal{D}}$ is the same as the one shown in Figure 1.

To maintain $\overline{Q}_\triangle$, we use the six components $\widehat{Q}_{123}, \ldots, \widehat{Q}_{321}$ of the multivariate extension $\widehat{Q}_\triangle$ of $Q_\triangle$ from Example 3.4. In particular, we construct and maintain the relations $\widehat{R}_{123}$, $\widehat{S}_{123}$ and $\widehat{T}_{123}$ from Eq. (16) and forth. We use them along with Theorem 4.1 to maintain $\widehat{Q}_{123}$ (and the same goes for $\widehat{Q}_{132}, \ldots$). For example, suppose that the number of updates N is 8, and consider the insert $+S(b_1, c_1)$ at time 2 in Table 2. This insert corresponds to inserting the following tuples into $\widehat{S}_{123}$: (Recall Eq. (15). Below, z_1 and z_2 are bitstrings and $z_1 \circ z_2$ is their concatenation.)

$$+\{(z_1, z_2, b_1, c_1) \mid (z_1 \circ z_2) \in \mathsf{CP}_8([2, 8])\}$$

The corresponding delete $-S(b_1, c_1)$ at time 5 corresponds to deleting and inserting the following two sets of tuples from/to $\widehat{S}_{123}$:

$$-\{(z_1, z_2, b_1, c_1) \mid (z_1 \circ z_2) \in \mathsf{CP}_8([2, 8])\} \qquad +\{(z_1, z_2, b_1, c_1) \mid (z_1 \circ z_2) \in \mathsf{CP}_8([2, 5])\}$$

Out of the box, Theorem 4.1 is only limited to inserts. However, we can show that these pairs of inserts/deletes have a special structure that allows us to maintain the guarantees of Theorem 4.1. In particular, at time 5, when we truncate a tuple $\boldsymbol{t}$ from $[2, 8]$ to $[2, 5]$, even after truncation, $\boldsymbol{t}$ still joins with the same set of tuples that it used to join with before the truncation. This is because the only intervals that overlap with $[2, 8]$ but not with $[2, 5]$ are the ones that start after 5. But these intervals are in the *future*, hence they don't exist yet in the database. We use this idea to amortize the cost of truncations over the inserts.

For $\mathrm{IVM}^{\pm}[Q_\triangle]$, our target is to do constant-delay enumeration of the full output $Q_\triangle$ at time τ. To that end, we enumerate tuples $([z], a, b, c)$ of $\overline{Q}_\triangle$ where the interval $[z]$ contains the current time τ. And to enumerate those, we enumerate tuples (z_1, z_2, z_3, a, b, c) from $\widehat{Q}_{123}, \ldots, \widehat{Q}_{321}$ where $z_1 \circ z_2 \circ z_3$ corresponds to an interval in $\mathsf{CP}_N([z])$ that contains τ. This is a *selection* condition over (z_1, z_2, z_3). By construction, the tree decomposition of any component, say $\widehat{Q}_{123}$, must contain a bag W that contains the variables $\{z_1, z_2, z_3\}$. This is because, by Definition 3.3, $\widehat{Q}_{123}$ contains an atom $\widehat{T}_{123}$ that contains these variables. We designate W as the root of tree decomposition of $\widehat{Q}_{123}$ and start our constant-delay enumeration from W. The same goes for $\widehat{Q}_{132}, \ldots, \widehat{Q}_{321}$. Moreover, the enumeration outputs of $\widehat{Q}_{123}, \ldots, \widehat{Q}_{321}$ are *disjoint* in this case.

For $\mathrm{IVM}^{\pm}_{\delta}[Q_\triangle]$, our target is to enumerate the change to the output at time τ. To that end, we enumerate tuples $([z], a, b, c)$ of $\overline{Q}_\triangle$ where the interval $[z]$ has τ *as an endpoint*. Appendix C.2 explains the technical challenges and how to address them.

7.2 Lower bound for $\mathrm{IVM}^{\pm}_{\delta}[Q]$

In order to prove the lower bound from Theorem 4.5, we prove a stronger lower bound where $|\mathcal{D}^{(\tau)}|$ is replaced by the number of single-tuple updates N. The following example demonstrates our lower bound for $\mathrm{IVM}^{\pm}_{\delta}[Q]$, which is based on a reduction from the static evaluation of a component $\widehat{Q}_{\boldsymbol{\sigma}}$ of the multivariate extension of Q to $\mathrm{IVM}^{\pm}_{\delta}[Q]$.

Example 7.3 (for Theorem 4.5). Consider $Q_\triangle$ from Eq. (1) and $\widehat{Q}_{123}$ from Eq. (8). Suppose that there exists $\gamma > 0$ such that $\mathrm{IVM}^{\pm}_{\delta}[Q_\triangle]$ can be solved with amortized update time $\widetilde{O}(|\mathcal{D}^{(\tau)}|^\kappa)$ where $|\mathcal{D}^{(\tau)}|$ is the database size, $\kappa \stackrel{\text{def}}{=} \omega(\widehat{Q}_{123}) - 1 - \gamma$, and ω is given by Definition 3.2. (Note that $\widetilde{O}(|\mathcal{D}^{(\tau)}|^\kappa) = \widetilde{O}(N^\kappa)$ since $|\mathcal{D}^{(\tau)}| \leq N$.) We show that this implies that $\mathrm{Eval}[\widehat{Q}_{123}]$ can be solved in time $\widetilde{O}(|\widehat{\mathcal{D}}_{123}|^{\omega(\widehat{Q}_{123})-\gamma} + |\widehat{Q}_{123}(\widehat{\mathcal{D}}_{123})|)$ on any database instance $\widehat{\mathcal{D}}_{123}$, thus contradicting the definition of $\omega(\widehat{Q}_{123})$.

Let us take an arbitrary database instance $\widehat{\mathcal{D}}_{123} = (\widehat{R}_{123}, \widehat{S}_{123}, \widehat{T}_{123})$ and evaluate $\widehat{Q}_{123}$ over $\widehat{\mathcal{D}}_{123}$ using the oracle that solves $\text{IVM}^{\pm}_{\delta}[Q_{\triangle}]$. Example 6.5 shows how to reduce $\text{Eval}[\widehat{Q}_{123}]$ over $\widehat{\mathcal{D}}_{123}$ to the static evaluation of $\overline{Q}_{\triangle}$ from Eq. (14) over a database instance $\overline{\mathcal{D}} = (\overline{R}, \overline{S}, \overline{T})$ that satisfies: $|\overline{\mathcal{D}}| = |\widehat{\mathcal{D}}_{123}|$ and $|\overline{Q}_{\triangle}(\overline{\mathcal{D}})| = |\widehat{Q}_{123}(\widehat{\mathcal{D}}_{123})|$. Moreover, $\overline{\mathcal{D}}$ can be constructed in linear time. Next, we reduce the evaluation of $\overline{Q}_{\triangle}$ over $\overline{\mathcal{D}}$ into $\text{IVM}^{\pm}_{\delta}[Q_{\triangle}]$.

We interpret the interval $[Z]$ of a tuple in $\overline{Q}_{\triangle}$ as the "lifespan" of a corresponding tuple in $Q_{\triangle}$, similar to Section 7.1. In particular, we initialize empty relations R, S, T to be used as input for $Q_{\triangle}$, which will be maintained under a sequence of updates for the $\text{IVM}^{\pm}_{\delta}[Q_{\triangle}]$ problem. We also sort the tuples $([\alpha, \beta], a, b)$ of $\overline{R}$ based on the end points α and β of their $[Z]$-intervals. Each tuple $([\alpha, \beta], a, b)$ appears twice in the order: once paired with its beginning α and another with its end β. Similarly, we add the tuples $([\alpha, \beta], b, c)$ of $\overline{S}$ and $([\alpha, \beta], a, c)$ of $\overline{T}$ to the same sorted list, where each tuple appears twice. Now, we go through the list in order. If the next tuple $([\alpha, \beta], a, b)$ is paired with its beginning α, then we insert the tuple (a, b) into R. However, if the next tuple $([\alpha, \beta], a, b)$ is paired with its end β, then we delete (a, b) from R. The same goes for S and T. The number of updates, N, is $2|\overline{\mathcal{D}}|$, and the total time needed to process them is $\widetilde{O}(|\overline{\mathcal{D}}|^{\kappa+1})$.

To compute the output of $\overline{Q}_{\triangle}$, we do the following. After every insert or delete in $Q_{\triangle}$, we use the oracle for $\text{IVM}^{\pm}_{\delta}[Q_{\triangle}]$ to enumerate the *change* in the output of $Q_{\triangle}$. Whenever the oracle reports an insert of an output tuple (a, b, c) at time τ and the delete of the same tuple at a later time $\tau' > \tau$, then we know that $([\tau, \tau'], a, b, c)$ is an output tuple of $\overline{Q}_{\triangle}$. Therefore, the overall time needed to compute the output of $\overline{Q}_{\triangle}$ (including the total update time from before) is $\widetilde{O}(|\overline{\mathcal{D}}|^{\kappa+1} + |\overline{Q}_{\triangle}(\overline{\mathcal{D}})|) = \widetilde{O}(|\widehat{\mathcal{D}}_{123}|^{\kappa+1} + |\widehat{Q}_{123}(\widehat{\mathcal{D}}_{123})|)$. This contradicts the definition of $\omega(\widehat{Q}_{123})$.

8 Conclusion

This paper puts forward two-way reductions between the dynamic and static query evaluation problems that allow us to transfer a wide class of algorithms and lower bounds between the two problems. For the dynamic problem, the paper characterizes the complexity gap between the insert-only and the insert-delete settings. The proposed algorithms recover best known amortized update times and produce new ones. The matching lower bounds justify why our upper bounds are natural. The lower bound function $\omega(Q)$ that we define can be used as a general tool to assess the optimality of some algorithms in database theory.

The assumptions that the queries have all variables free and the input database be initially empty can be lifted without difficulty. To support arbitrary conjunctive queries, our algorithms need to consider free-connex tree decompositions, which ensure constant-delay enumeration of the tuples in the query result. To support non-empty initial databases, we can use a preprocessing phase to construct our data structure, where all initial tuples get the same starting timestamp. The complexities stated in the paper need to be changed to account for the size of the initial database as part of the size of the update sequence.

Although the proposed dynamic algorithms MVIVM are designed to maintain base relations and the query output as sets, they can be extended to maintain bags. As in DBToaster [28] and F-IVM [26], MVIVM can be extended to maintain the multiplicity of each tuple in a base relation or view and enumerate each tuple in the query output together with its multiplicity.

Acknowledgments

This work was partially supported by NSF-BSF 2109922, NSF-IIS 2314527, NSF-SHF 2312195, and UZH Global Strategy and Partnerships Funding Scheme, and was conducted while some of the authors participated in the Simons Program on Logic and Algorithms in Databases and AI.

References

[1] Martín Abadi, Frank McSherry, and Gordon D. Plotkin. 2015. Foundations of Differential Dataflow. In *FoSSaCS*. 71–83. https://doi.org/10.1007/978-3-662-46678-0_5

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

[3] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. 2022. The Complexity of Boolean Conjunctive Queries with Intersection Joins. In *PODS*. 53–65. https://doi.org/10.1145/3517804.3524156

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

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

[6] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2024. PANDA: Query Evaluation in Submodular Width. *arXiv* (2024). https://doi.org/10.48550/arXiv.2402.02001

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

[8] Anonymous Author(s). 20xx. Insert-Only versus Insert-Delete in Dynamic Query Evaluation. (20xx).

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

[10] Johann Brault-Baron. 2016. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.* 49, 3 (2016), 54:1–54:26. https://doi.org/10.1145/2983573

[11] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614. https://doi.org/10.14778/3587136.3587137

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

[13] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. 2018. Reachability Is in DynFO. *J. ACM* 65, 5 (2018), 33:1–33:24. https://doi.org/10.1145/3212685

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

[15] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. 2017. Complex Event Recognition in the Big Data Era. *Proc. VLDB Endow.* 10, 12 (2017), 1996–1999. https://doi.org/10.14778/3137765.3137829

[16] Alejandro Grez and Cristian Riveros. 2020. Towards Streaming Evaluation of Queries with Correlation in Complex Event Processing. In *ICDT*. 14:1–14:17. https://doi.org/10.4230/LIPIcs.ICDT.2020.14

[17] 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. https://doi.org/10.1145/2746539.2746609

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

[19] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General Dynamic Yannakakis: Conjunctive Queries with Theta Joins under Updates. *VLDB J.* 29, 2-3 (2020), 619–653. https://doi.org/10.1007/S00778-019-00590-9

[20] Christian S. Jensen and Richard T. Snodgrass. 2018. Temporal Database. In *Encyclopedia of Database Systems, Second Edition*. Springer. https://doi.org/10.1007/978-1-4614-8265-9_395

[21] 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. https://doi.org/10.4230/LIPICS.ICDT.2019.4

[22] 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. https://doi.org/10.1145/3396375

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

[24] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. Conjunctive Queries with Free Access Patterns Under Updates. In *ICDT*. 17:1–17:20. https://doi.org/10.4230/LIPICS.ICDT.2023.17

[25] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *Log. Methods Comput. Sci.* 19, 3 (2023). https://doi.org/10.46298/LMCS-19(3:11)2023

[26] 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. https://doi.org/10.1007/S00778-023-00817-W

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

[28] 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. https://doi.org/10.1007/S00778-013-0348-4

[29] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *PODS*. 75–90. https://doi.org/10.1145/2902251.2902286

[30] L. H. Loomis and H. Whitney. 1949. An Inequality Related to the Isoperimetric Inequality. *Journal: Bull. Amer. Math. Soc.* 55, 55 (1949), 961–962. DOI: 10.1090/S0002-9904-1949-09320-5.

[31] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6, Article 42 (2013). https://doi.org/10.1145/2535926

[32] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of Datalog Materialisations Revisited. *Artif. Intell.* 269 (2019), 76–136. https://doi.org/10.1016/J.ARTINT.2018.12.004

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

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

[35] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (feb 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[36] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. 2014. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD*. 253–264. https://doi.org/10.1145/2588555.2610519

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

[38] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: Learning over Fast-Evolving Relational Data. In *SIGMOD*. 2773–2776. https://doi.org/10.1145/3318464.3384702

[39] Mihai Patrascu. 2010. Towards Polynomial Lower Bounds for Dynamic Problems. In *STOC*. 603–610. https://doi.org/10.1145/1806689.1806772

[40] Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. 2020. Sketches of Dynamic Complexity. *SIGMOD Rec.* 49, 2 (2020), 18–29. https://doi.org/10.1145/3442322.3442325

[41] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00362ED1V01Y201105DTM016

[42] Yufei Tao and Ke Yi. 2022. Intersection Joins under Updates. *J. Comput. Syst. Sci.* 124 (2022), 41–64. https://doi.org/10.1016/J.JCSS.2021.09.004

[43] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. https://doi.org/10.5441/002/ICDT.2014.13

[44] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation without Joins. *Proc. VLDB Endow.* 16, 5 (2023), 1046–1058. https://doi.org/10.14778/3579075.3579080

[45] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *SIGMOD*. 1225–1239. https://doi.org/10.1145/3318464.3380586

[46] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94.

A Comparison to Prior IVM Approaches

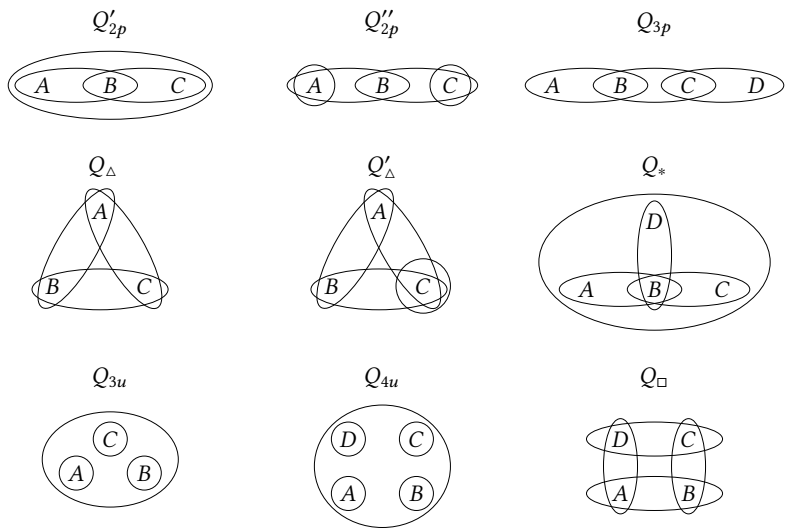


Fig. 3. Hypergraphs of the queries analyzed in this section. Atoms are represented by hyperedges.

In this section, we compare our MVIVM approach with existing IVM approaches. We later focus on two specific queries, namely Q_{3p} and $Q_\triangle$, and explain in detail how they are maintained by all considered approaches. We focus on the problem IVM$^\pm$, where the task is to process single-tuple inserts *and* deletes and enumerate the *full* query result upon each enumeration request. We consider the following existing approaches:

- Naïve: This approach recomputes the query result from scratch after each update.
- Delta [12]: This is a *first-order IVM* approach that materializes the query result and computes the delta or the change to the query result for each update. It updates the query result with the delta result.
- F-IVM [26, 38]: This is a *higher-order IVM* approach that speeds up the delta computation using a view tree whose structure is modelled on a variable order. Its update times are shown to be at least as good as the prior higher-order IVM approach DBToaster [28].
- IVM$^\epsilon$ [21, 22]: This is an *adaptive IVM* methodology that uses delta computation and materialized views and also takes the degrees of data values into account. It is adaptive in the sense that it treats updates referring to values with high degree differently from those with low degree. The specific attributes whose values are partitioned with regard to their degrees and the partitioning threshold that lead to the best update time can vary depending on the query structure. So far, there is no algorithm that produces the optimal partitioning strategy for arbitrary queries. Therefore, IVM$^\epsilon$ is not a concrete algorithm for arbitrary queries but rather a methodology. It was shown in [21, 22] how to manually tailor this methodology to the triangle query and several non-hierarchical queries to achieve the worst-case optimal update time.
- CROWN (**C***hange p***RO***pagation* **W***ithout joi***N***s*) [44]: This is a higher-order IVM approach tailored to free-connex (α-)acyclic queries. Its maintenance approach is in line with F-IVM, when applied to free-connex acyclic queries, yet it provides a more refined complexity analysis that takes the structure of the update sequence into account. For certain update sequences,

such as *insert-only* or *first-in-first-out*, the amortized update time is constant. For the purpose of comparing against the other IVM approaches, which do not consider such a fine-grained analysis, we consider the complexity results of CROWN for arbitrary update sequences.

Another higher-order IVM approach is DynYannakakis [18, 19], which can maintain acyclic queries. It achieves the same asymptotic update times as F-IVM and CROWN for acyclic queries.

We consider the following non-hierarchical join queries (all variables are free). They are visualized in Figure 3:

$$Q'_{2p} = R(A,B,C) \wedge S(A,B) \wedge T(B,C) \tag{17}$$
$$Q_{3p} = R(A,B) \wedge S(B,C) \wedge T(C,D) \tag{18}$$
$$Q_{\triangle} = R(A,B) \wedge S(B,C) \wedge T(A,C) \tag{19}$$
$$Q'_{\triangle} = R(A,B) \wedge S(B,C) \wedge T(A,C) \wedge U(C) \tag{20}$$
$$Q_{*} = R(A,B,C,D) \wedge S(A,B) \wedge T(B,C) \wedge U(B,D) \tag{21}$$
$$Q_{3u} = R(A,B,C) \wedge S(A) \wedge T(B) \wedge U(C) \tag{22}$$
$$Q_{4u} = R(A,B,C,D) \wedge S(A) \wedge T(B) \wedge U(C) \wedge V(D) \tag{23}$$
$$Q''_{2p} = R(A) \wedge S(A,B) \wedge T(B,C) \wedge U(C) \tag{24}$$
$$Q_{\square} = R(A,B) \wedge S(B,C) \wedge T(C,D) \wedge U(A,D) \tag{25}$$

Figure 4 summarizes the update times of all aforementioned IVM approaches, give in data complexity. The update times of Naïve, Delta, and F-IVM are worst-case and those of CROWN, IVM$^\epsilon$, and MVIVM are amortized. The update times of MVIVM have an extra polylog($|\mathcal{D}|$) factor, which is hidden by the $\widetilde{O}$-notation. Since CROWN is designed for free-connex acyclic queries, we skip its complexity for the acyclic ones, which are marked "NA". It is not clear how to manually tailor the IVM$^\epsilon$ approach to all queries, thus some entries are marked "Open". In terms of query complexity, MVIVM has an extra factor of $k!$, where k is the number of input relations, due to the use of multivariate extensions (Definiton 3.3).

We observe that, besides the last two queries, MVIVM is on par or outperforms the existing approaches. For the last two queries, it performs only worse than IVM$^\epsilon$. However, in contrast to IVM$^\epsilon$, MVIVM is a systematic algorithm that works for all queries, whereas IVM$^\epsilon$ is a general methodology where a heavy-light partitioning strategy needs to be manually tailored to a specific query. Such a strategy defines (i) on which tuples of variables in each relation to partition into heavy and light, and (ii) what is the heavy-light threshold for each relation partition. There are infinitely many possible partitioning thresholds and hence partitioning strategies. It is not possible to try them all out in order to find the best one. Therefore, so far there does not exist an algorithm that produces an optimal IVM$^\epsilon$ strategy for any query.[5]

B Missing Details in Section 6: Reduction from $\overline{Q}$ to $\widehat{Q}$

Example 6.4 introduces the high-level idea of the reduction from the intersection join query $\overline{Q}$ to the multivariate extension $\widehat{Q}$, based on [3]. Extrapolating from that example, we introduce the following definition:

[5]At a technical level, the reason why MVIVM has a worse update time for a query like $Q_{\square}$ than IVM$^\epsilon$ is because MVIVM is limited to the fractional hypertree width, while for $Q_{\square}$, it is known that the *submodular width* is strictly smaller than the fractional hypertree width [5, 6, 31]. MVIVM does not immediately generalize to the submodular width because the query decomposition lemma (Lemma 5.2) does not. Known algorithms that meet the submodular width [5, 6] are limited to the static setting and do not seem to generalize to the dynamic setting. We leave this generalization as an open problem.

	Naïve	Delta	F-IVM	CROWN	IVM$^\epsilon$ (manual)	MVIVM
Q'_{2p}	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	*Open*	$\widetilde{O}(\lvert\mathcal{D}\rvert^{1/2})$
Q_{3p}	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert^{1/2})$	$\widetilde{O}(\lvert\mathcal{D}\rvert^{1/2})$
$Q_\triangle$	$O(\lvert\mathcal{D}\rvert^{3/2})$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	NA	$O(\lvert\mathcal{D}\rvert^{1/2})$	$\widetilde{O}(\lvert\mathcal{D}\rvert^{1/2})$
$Q'_\triangle$	$O(\lvert\mathcal{D}\rvert^{3/2})$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	NA	$O(\lvert\mathcal{D}\rvert^{2/3})$	$\widetilde{O}(\lvert\mathcal{D}\rvert^{2/3})$
Q_*	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	*Open*	$\widetilde{O}(\lvert\mathcal{D}\rvert^{2/3})$
Q_{3u}	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	*Open*	$\widetilde{O}(\lvert\mathcal{D}\rvert^{2/3})$
Q_{4u}	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	*Open*	$\widetilde{O}(\lvert\mathcal{D}\rvert^{3/4})$
Q''_{2p}	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert^{1/2})$	$\widetilde{O}(\lvert\mathcal{D}\rvert)$
$Q_\square$	$O(\lvert\mathcal{D}\rvert^2)$	$O(\lvert\mathcal{D}\rvert)$	$O(\lvert\mathcal{D}\rvert)$	NA	$O(\lvert\mathcal{D}\rvert^{2/3})$	$\widetilde{O}(\lvert\mathcal{D}\rvert)$
	worst-case				amortized	

Fig. 4. Update times of different IVM approaches for the problem IVM$^\pm$, given in data complexity. The update times of the approaches Naïve, Delta, and F-IVM are worst-case, while those of CROWN, IVM$^\epsilon$, and MVIVM are amortized. $\widetilde{O}$ hides a polylog($|\mathcal{D}|$) factor. Queries are depicted in Figure 3 and are given by Eq. (17)-(25). MVIVM is our approach, introduced in this paper. CROWN is not applicable to cyclic queries, thus some entries are marked "NA". IVM$^\epsilon$ is not a concrete algorithm but rather a general methodology. It is not clear how to manually set it up for every query, thus some entries are marked "Open". In query complexity, MVIVM has a query-dependant factor of $k!$, where k is the number of input relations.

Definition B.1 (Canonical partition of a relation $\mathsf{CP}_N^{(i)}(R)$*).* Let $\boldsymbol{t} = ([z], x_1, \ldots, x_m)$ be a tuple where $[z]$ is an interval and $x_1, \ldots, x_m$ are points. Given natural numbers N and i, the *canonical partition* of $\boldsymbol{t}$ is defined as the following set of tuples

$$\mathsf{CP}_N^{(i)}([z], x_1, \ldots, x_m) \stackrel{\text{def}}{=} \{(z_1, \ldots, z_i, x_1, \ldots, x_m) \mid (z_1 \circ \cdots \circ z_i) \in \mathsf{CP}_N([z])\} \tag{26}$$

We lift the definition of a canonical partition $\mathsf{CP}_N^{(i)}$ to relations and define $\mathsf{CP}_N^{(i)}(R)$ as the union of the canonical partitions of the tuples in the given relation R. We also lift the definition to deltas and define $\mathsf{CP}_N^{(i)}(\delta R)$ as the corresponding delta for $\mathsf{CP}_N^{(i)}(R)$.

Definition B.2 (Canonical partition of a database instance $\mathsf{CP}_N(\overline{\mathcal{D}})$*).* Let $Q = R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k)$ be a query and $\overline{Q} = \overline{R}_1([Z], \boldsymbol{X}_1) \wedge \cdots \wedge \overline{R}_k([Z], \boldsymbol{X}_k)$ its time extension. Given a permutation $\boldsymbol{\sigma} \in \Sigma_k$, let $\widehat{Q}_{\boldsymbol{\sigma}}$ be the corresponding component of its multivariate extension $\widehat{Q}$:

$$\widehat{Q}_{\boldsymbol{\sigma}} = \widehat{R}_{\sigma_1}(Z_1, \boldsymbol{X}_{\sigma_1}) \wedge \widehat{R}_{\sigma_2}(Z_1, Z_2, \boldsymbol{X}_{\sigma_2}) \wedge \cdots \wedge \widehat{R}_{\sigma_k}(Z_1, \ldots, Z_k, \boldsymbol{X}_{\sigma_k})$$

Given a database instance $\overline{\mathcal{D}}$ for $\overline{Q}$, the *canonical partition* of $\overline{\mathcal{D}}$ using $\boldsymbol{\sigma}$, denoted by $\mathsf{CP}_N^{\sigma}(\overline{\mathcal{D}})$, is a database instance $\widehat{\mathcal{D}}_{\boldsymbol{\sigma}}$ for $\widehat{Q}_{\boldsymbol{\sigma}}$ that is defined as:

$$\widehat{R}_{\sigma_i} \stackrel{\text{def}}{=} \mathsf{CP}_N^{(i)}(\overline{R}_{\sigma_i}), \qquad \text{for } i \in [k] \tag{27}$$

Moreover, define $\mathsf{CP}_N(\overline{\mathcal{D}})$ to be a database instance for $\widehat{Q}$ which is a combination of $\mathsf{CP}_N^{\sigma}(\overline{\mathcal{D}})$ for every $\boldsymbol{\sigma} \in \Sigma_k$:

$$\mathsf{CP}_N(\overline{\mathcal{D}}) \stackrel{\text{def}}{=} \left(\mathsf{CP}_N^{\sigma}(\overline{\mathcal{D}})\right)_{\boldsymbol{\sigma}\in\Sigma_k}$$

We lift the definition of CP_N^{σ} to deltas and define $\mathsf{CP}_N^{\sigma}(\delta\overline{\mathcal{D}})$ to be the corresponding delta for $\mathsf{CP}_N^{\sigma}(\overline{\mathcal{D}})$. We define $\mathsf{CP}_N(\delta\overline{\mathcal{D}})$ similarly.

(16) is a special case of (27) above. The following theorem basically says that evaluating $\overline{Q}(\overline{\mathcal{D}})$ is equivalent to evaluating $\widehat{Q}$ on the canonical partition of $\overline{\mathcal{D}}$.

Theorem B.3 (Implicit in [3]). *Given a tuple $(z_1, \ldots, z_k, x_1, \ldots, x_m)$, define*

$$G_k(z_1, \ldots, z_k, x_1, \ldots, x_m) \stackrel{def}{=} (z_1 \circ \cdots \circ z_k, x_1, \ldots, x_m) \tag{28}$$

We lift the definition of G_k to relations and define $G_k(R)$ to be the relation resulting from applying G to every tuple in R. Then,

$$\mathsf{CP}_N^{(1)}(\overline{Q}(\overline{\mathcal{D}})) = G_k(\widehat{Q}(\mathsf{CP}_N(\overline{\mathcal{D}}))) \tag{29}$$

C Missing Details from Section 7

C.1 Algorithm for Lemma 7.1: The IVM$^{\pm}[Q]$ problem

Example 7.2 introduces the high-level idea of our algorithm for IVM$^{\pm}[Q]$ that proves Lemma 7.1. We present the general algorithm in Algorithm 1. The correctness proof and runtime analysis are technically involved and are deferred to the full version [8].

C.2 Algorithm for Lemma 7.1: The IVM$^{\pm}_{\delta}[Q]$ problem

In the previous section, we showed our algorithm that proves Lemma 7.1 for the IVM$^{\pm}[Q]$ problem for a given query Q. In this section, we describe how to extend this algorithm in order to prove Lemma 7.1 for the IVM$^{\pm}_{\delta}[Q]$ problem. The proof can be found in [8].

Here are the main changes to Algorithm 1 in order to support IVM$^{\pm}_{\delta}[Q]$, i.e. support constant-delay enumeration of $\delta_\tau Q(\mathcal{D})$ after every update $\delta_\tau\mathcal{D}$:

- **Insertion** $\delta_\tau\mathcal{D} = \{+R_j(\boldsymbol{t})\}$: In addition to applying the insert $\{+\overline{R}_j([\tau,\infty],\boldsymbol{t})\}$ into $\overline{\mathcal{D}}$, we now also insert $+\overline{R}_j([\tau,\tau],\boldsymbol{t})$.
- **Deletion** $\delta_\tau\mathcal{D} = \{-R_j(\boldsymbol{t})\}$: In addition to applying the update $\{-\overline{R}_j([\tau',\infty],\boldsymbol{t}), +\overline{R}_j([\tau',\tau],\boldsymbol{t})\}$ to $\overline{\mathcal{D}}$, we now also apply the insert $+\overline{R}_j([\tau,\tau],\boldsymbol{t})$.
- **Enumeration** Suppose we want to enumerate $\delta_\tau Q(\mathcal{D})$ after an update $\delta_\tau\mathcal{D} = sR_j(\boldsymbol{t})$ where $s \in \{+,-\}$. Then, we enumerate the following:

$$\delta_\tau Q(\mathcal{D}) = s\left(\pi_{\mathsf{vars}(Q)} \bigcup_{\substack{z_1,\ldots,z_k\in\{0,1\}^*\\ z_1\circ\cdots\circ z_k=\mathsf{CP}([\tau,\tau])}} \sigma_{Z_1=z_1,\ldots,Z_k=z_k}\widehat{Q}(\widehat{\mathcal{D}})\right) \tag{30}$$

We call the above modified version of Algorithm 1 the *delta version* of Algorithm 1.

C.3 Proof of Theorem 4.4 using Lemma 7.1

The upper bound in Lemma 7.1 is written in terms of the length N of the update stream. In this section, we show how to use Lemma 7.1 as a black box in order to prove Theorem 4.4, which gives a stronger bound where N is replaced by the current database size $|\mathcal{D}^{(\tau)}|$ at the τ-th update.

Algorithm 1 Algorithm of IVM$^{\pm}$[Q]

Inputs
- $Q = R_1(\boldsymbol{X}_1) \wedge \cdots \wedge R_k(\boldsymbol{X}_k)$ ▷ Input query to IVM$^{\pm}$
- An initially empty database $\mathcal{D}$ ▷ Database instance for Q
- N: The length of the update stream

Initialization
- Construct the time extension $\overline{Q}$ ▷ Definition 6.2
- Initialize $\overline{\mathcal{D}}$ to be empty ▷ Database instance for $\overline{Q}$
- Construct the multivariate extension $\widehat{Q}$ ▷ Eq. (7)
- Initialize $\widehat{\mathcal{D}}$ to be empty ▷ Database instance for $\widehat{Q}$
- For each component $\widehat{Q}_{\boldsymbol{\sigma}}$ of $\widehat{Q}$
  - Initialize a tree decomposition using Theorem 4.1

Invariants
- $\widehat{\mathcal{D}} = \mathsf{CP}_N(\overline{\mathcal{D}})$ ▷ Definition B.2

Insertion $\delta_\tau\mathcal{D} = \{+R_j(\boldsymbol{t})\}$ ▷ The τ-th update inserts a tuple $\boldsymbol{t}$ into R_j
- $\delta_\tau\overline{\mathcal{D}} \leftarrow \{+\overline{R}_j([\tau,\infty],\boldsymbol{t})\}$ ▷ Insert $([\tau,\infty],\boldsymbol{t})$ into $\overline{R}_j$
- $\overline{\mathcal{D}} \leftarrow \overline{\mathcal{D}} \uplus \delta_\tau\overline{D}$ ▷ Apply $\delta_\tau\overline{\mathcal{D}}$ to $\overline{\mathcal{D}}$
- $\delta_\tau\widehat{\mathcal{D}} \leftarrow \mathsf{CP}_N(\delta_\tau\overline{\mathcal{D}})$ ▷ Definition B.2
- $\widehat{\mathcal{D}} \leftarrow \widehat{\mathcal{D}} \uplus \delta_\tau\widehat{\mathcal{D}}$ ▷ Use Theorem 4.1

Deletion $\delta_\tau\mathcal{D} = \{-R_j(\boldsymbol{t})\}$ ▷ The τ-th update deletes a tuple $\boldsymbol{t}$ from R_j
- Find $([\tau',\infty],\boldsymbol{t}) \in \overline{R}_j$
- $\delta_\tau\overline{\mathcal{D}} \leftarrow \{-\overline{R}_j([\tau',\infty],\boldsymbol{t}),+\overline{R}_j([\tau',\tau],\boldsymbol{t})\}$ ▷ Replace $([\tau',\infty],\boldsymbol{t})$ with $([\tau',\tau],\boldsymbol{t})$ in $\overline{R}_j$
- $\overline{\mathcal{D}} \leftarrow \overline{\mathcal{D}} \uplus \delta_\tau\overline{D}$
- $\delta_\tau\widehat{\mathcal{D}} \leftarrow \mathsf{CP}_N(\delta_\tau\overline{\mathcal{D}})$
- $\widehat{\mathcal{D}} \leftarrow \widehat{\mathcal{D}} \uplus \delta_\tau\widehat{\mathcal{D}}$

Enumeration
- Enumerate (using Theorem 4.1)

$$\pi_{\mathsf{vars}(Q)} \bigcup_{\substack{z_1,\ldots,z_{k+1}\in\{0,1\}^*\\ z_1\circ\cdots\circ z_{k+1}=\mathsf{CP}([\tau,\tau])}} \sigma_{Z_1=z_1,\ldots,Z_k=z_k}\widehat{Q}(\widehat{\mathcal{D}})$$

Theorem 4.4. *For any query Q, both IVM$^{\pm}$[Q] and IVM$^{\pm}_{\delta}$[Q] can be solved with $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\mathsf{w}(\widehat{Q})-1})$ amortized update time, where $\widehat{Q}$ is the multivariate extension of Q, and $|\mathcal{D}^{(\tau)}|$ is the current database size at update time τ.*

Recall that having an amortized update time of $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\mathsf{w}(\widehat{Q})-1})$ means that all N updates combined take time $\widetilde{O}(\sum_{\tau\in[N]}|\mathcal{D}^{(\tau)}|^{\mathsf{w}(\widehat{Q})-1})$.

Proof. The proof uses Lemma 7.1 as a black box. In particular, we will use the same maintenance algorithm from the proof of Lemma 7.1. However, every now and then, we will "reset" the algorithm

by taking the current database $\mathcal{D}^{(\tau)}$, creating a new empty database from scratch, and inserting every tuple in $\mathcal{D}^{(\tau)}$ into the new database as a single-tuple update. After that, we process the next $\frac{1}{2}|\mathcal{D}^{(\tau)}|$ updates[6] from the original update sequence, and then we apply another reset. The total number of updates between the last two resets is thus $\frac{3}{2}|\mathcal{D}^{(\tau)}|$. By Lemma 7.1, these $\frac{3}{2}|\mathcal{D}^{(\tau)}|$ updates together take time $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})})$. Moreover, note that between the two resets, the database size cannot change by more than a factor of 2. This is because we only have $\frac{1}{2}|\mathcal{D}^{(\tau)}|$ updates from the original stream that are applied to $\mathcal{D}^{(\tau)}$. In the two extreme cases, all of these updates are inserts or all of them are deletes.

Consider the original update stream of length N. Initially, $|\mathcal{D}^{(0)}| = 0$. While using the above scheme to process the stream, suppose that we end up performing k resets in total at time stamps $\tau_1 \stackrel{\text{def}}{=} 0 < \tau_2 < \cdots < \tau_k \leq N$. WLOG assume that $\tau_k = N$. From the above construction, for any $i \in [k-1]$, we have the following:

$$\tau_{i+1} - \tau_i = \frac{1}{2}|\mathcal{D}^{(\tau_i)}| \tag{31}$$

$$\frac{1}{2}|\mathcal{D}^{(\tau_i)}| \leq |\mathcal{D}^{(\tau)}| \leq \frac{3}{2}|\mathcal{D}^{(\tau_i)}|, \quad \forall \tau \in [\tau_i, \tau_{i+1}) \tag{32}$$

Eq. (32) holds because between τ_i and τ_{i+1}, we are making $\frac{1}{2}|\mathcal{D}^{(\tau_i)}|$ updates to $\mathcal{D}^{(\tau_i)}$, which in the two extreme cases are either all inserts or all deletes. As noted before, from the beginning of the τ_i-reset until the beginning of the τ_{i+1}-reset, we process $\frac{3}{2}|\mathcal{D}^{(\tau_i)}|$ updates, which together take time $\widetilde{O}(|\mathcal{D}^{(\tau_i)}|^{\mathrm{w}(\widehat{Q})})$, thanks to Lemma 7.1. To bound the total runtime, we sum up over $i \in [k-1]$: (We don't actually need to perform the last reset at time $\tau_k = N$ since there are no subsequent updates anyhow.)

$$\sum_{i\in[k-1]} |\mathcal{D}^{(\tau_i)}|^{\mathrm{w}(\widehat{Q})} = \sum_{i\in[k-1]} \sum_{\tau\in[\tau_i,\tau_{i+1})} \frac{1}{\tau_{i+1}-\tau_i} \cdot |\mathcal{D}^{(\tau_i)}|^{\mathrm{w}(\widehat{Q})} \tag{33}$$

$$= 2 \cdot \sum_{i\in[k-1]} \sum_{\tau\in[\tau_i,\tau_{i+1})} |\mathcal{D}^{(\tau_i)}|^{\mathrm{w}(\widehat{Q})-1} \tag{34}$$

$$\leq 2^{\mathrm{w}(\widehat{Q})} \cdot \sum_{i\in[k-1]} \sum_{\tau\in[\tau_i,\tau_{i+1})} |\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})-1} \tag{35}$$

$$= O(1) \cdot \sum_{\tau\in[N]} |\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})-1} \tag{36}$$

Equality (34) follows from Eq. (31), while inequality (35) follows from Eq. (32). In particular, the total time needed to process all N updates in the stream is $\widetilde{O}(\sum_{\tau\in[N]} |\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})-1})$. This implies that the amortized time per update is $\widetilde{O}(|\mathcal{D}^{(\tau)}|^{\mathrm{w}(\widehat{Q})-1})$, as desired. □

Received May 2024; revised August 2024; accepted September 2024

[6]For simplicity, we assume that $|\mathcal{D}^{(\tau)}|$ is a positive even integer.