Maintaining Triangle Queries under Updates

Maintaining Triangle Queries under Updates

Ahmet Kara[1], Milos Nikolic[2], Hung Q. Ngo[3], Dan Olteanu[1], Haozhe Zhang[1]

[1]University of Oxford [2]University of Edinburgh [3]RelationalAI, Inc.

Abstract

We consider the problem of incrementally maintaining the triangle queries with arbitrary free variables under single-tuple updates to the input relations.

We introduce an approach called IVM$^\epsilon$ that exhibits a trade-off between the update time, the space, and the delay for the enumeration of the query result, such that the update time ranges from the square root to linear in the database size while the delay ranges from constant to linear time.

IVM$^\epsilon$ achieves Pareto worst-case optimality in the update-delay space conditioned on the Online Matrix-Vector Multiplication conjecture. It is strongly Pareto optimal for the triangle queries with zero or three free variables and weakly Pareto optimal for the triangle queries with one or two free variables.

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

1 Introduction

In this article we consider the problem of incrementally maintaining triangle queries under single-tuple updates to the input relations. We introduce an approach to this problem that expresses a trade-off between the update time, space, and enumeration delay. The *update time* is the time needed to maintain the data structure encoding the query result upon a single-tuple update. The *space* is the overall memory needed by the used data structure. The *enumeration delay* is the maximal time needed from starting the enumeration or reporting one result tuple to reporting the next result tuple or ending the enumeration.

We consider the triangle queries written in FAQ notation [2]. Let R, S, and T be relations that have schemas (A, B), (B, C), and (C, A), respectively, and are given as functions mapping tuples over their schemas to tuple multiplicities. The *ternary* triangle query

$$\triangle_3(a, b, c) = R(a, b) \cdot S(b, c) \cdot T(c, a)$$

returns each triangle and its multiplicity in the join of the three relations. The *binary* triangle query

$$\triangle_2(a, b) = \sum_{c \in \mathsf{Dom}(C)} R(a, b) \cdot S(b, c) \cdot T(c, a)$$

returns each (A, B)-pair that occurs in a triangle and its multiplicity. The *unary* triangle query

$$\triangle_1(a) = \sum_{b \in \mathsf{Dom}(B)} \sum_{c \in \mathsf{Dom}(C)} R(a, b) \cdot S(b, c) \cdot T(c, a)$$

returns each A-value that occurs in a triangle and its multiplicity. Finally, the *nullary* triangle query

$$\triangle_0() = \sum_{a \in \mathsf{Dom}(A)} \sum_{b \in \mathsf{Dom}(B)} \sum_{c \in \mathsf{Dom}(C)} R(a, b) \cdot S(b, c) \cdot T(c, a)$$

returns the number of triangles. There are further unary and binary triangle queries, e.g., $\triangle_1(b)$ or $\triangle_2(b,c)$, yet they can be treated similarly since the join of the three relations is symmetric in A, B, and C.

The ternary triangle query has served as a milestone for the worst-case optimality of join algorithms in the centralized and parallel settings. Likewise, the nullary triangle query is a working horse for randomized approximation schemes for data processing. They showcase the suboptimality of mainstream join algorithms used currently by virtually all commercial database systems. For a database $\mathbf{D}$ consisting of relations R, S, and T, standard binary join plans implementing these queries may take $O(|\mathbf{D}|^2)$ time, yet the ternary and nullary triangle queries can be solved in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ [32] and respectively $\mathcal{O}(|\mathbf{D}|^{1.41})$ time [3]. This observation motivated a new line of work on worst-case optimal algorithms for arbitrary join queries [32]. Triangle queries have also served as a yardstick for understanding the optimal communication cost for parallel query evaluation in the Massively Parallel Communication model [29]. They have witnessed the development of randomized approximation schemes with increasingly lower time and space requirements [18].

In our prior work we introduced a worst-case optimal approach for incrementally maintaining the exact result of the nullary triangle query [24]. This article extends that work with an investigation of Pareto worst-case optimality for the triangle queries in the update-delay space.

Incremental maintenance algorithms may benefit from a range of processing techniques whose combinations make it more challenging to reason about optimality. Such techniques include algorithms for aggregate-join queries with low complexity developed for the non-incremental case [32]; pre-materialization of views to reduce the maintenance of a query to that of subqueries [26]; and delta processing that allows to only compute the change to the result instead of the entire result [12].

1.1 Existing Incremental View Maintenance (IVM) Approaches

The problem of incrementally maintaining triangle queries has received a fair amount of attention. We next discuss the naïve approach, which recomputes the query result from scratch, and several IVM approaches.

We consider the single-tuple update $\delta R = \{(\alpha,\beta) \mapsto m\}$ to a binary relation R that maps the tuple (α,β) to a nonzero multiplicity m, which is positive for inserts and negative for deletes.

The naïve approach incurs constant-time updates: Each update is executed on a relation of the input database $\mathbf{D}$. Whenever we need the query result, we recompute it in time $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ [3, 32]. The number of distinct tuples in the result is at most $|\mathbf{D}|^{\frac{3}{2}}$ [30].

We next exemplify the classical first-order IVM [12] on the nullary triangle query $\triangle_0$ under the aforementioned single-tuple update δR; all other triangle queries are treated similarly. The classical IVM approach materializes the query result, computes on the fly a delta query $\delta\triangle_0$, and then updates the query result:

$$\delta\triangle_0() = \delta R(\alpha,\beta) \cdot \sum_{c \in \mathsf{Dom}(C)} S(\beta,c) \cdot T(c,\alpha), \qquad \triangle_0() = \triangle_0() + \delta\triangle_0().$$

The delta computation takes $\mathcal{O}(|\mathbf{D}|)$ time since it needs to intersect two lists of possibly linearly many C-values that are paired with β in S and with α in T (i.e., the multiplicity of such pairs in S and T is nonzero). Since the query result is materialized, it can be enumerated with constant delay.

The recursive IVM [26] speeds up the delta computation by precomputing three auxiliary views representing the update-independent parts of the delta queries:

$$V_{ST}(b,a) = \sum_{c \in \mathsf{Dom}(C)} S(b,c) \cdot T(c,a)$$

$$V_{TR}(c,b) = \sum_{a \in \mathsf{Dom}(A)} T(c,a) \cdot R(a,b)$$

$$V_{RS}(a,c) = \sum_{b \in \mathsf{Dom}(B)} R(a,b) \cdot S(b,c).$$

These three views take $\mathcal{O}(|\mathbf{D}|^2)$ space but allow to compute the delta query for single-tuple updates to the input relations in $\mathcal{O}(1)$ time. Computing the delta $\delta\triangle_0() = \delta R(\alpha,\beta) \cdot V_{ST}(\beta,\alpha)$ requires just a constant-time lookup in V_{ST}; however, maintaining the views V_{RS} and V_{TR}, which refer to R, still requires $\mathcal{O}(|\mathbf{D}|)$

time. The factorized IVM [33] materializes only one of the three views, for instance, V_{ST}. In this case, the maintenance under updates to R takes $\mathcal{O}(1)$ time, but the maintenance under updates to S and T still takes $\mathcal{O}(|\mathbf{D}|)$ time.

Further exact IVM approaches focus on acyclic conjunctive queries. For free-connex acyclic conjunctive queries, the dynamic Yannakakis approach allows for enumeration of result tuples with constant delay after single-tuple updates in linear time [20]. For databases with or without integrity constraints, it is known that a strict, small subset of the class of acyclic conjunctive queries admit constant-time update, while all other conjunctive queries have update times dependent on the size of the input database [6, 7].

A line of work relevant to our result unveils structure in the PTIME complexity class by giving lower bounds on the complexity of problems under various conjectures [19, 39].

Definition 1 (Online Matrix-Vector Multiplication (OMv) [19]). *We are given an $n \times n$ Boolean matrix $\mathbf{M}$ and receive n column vectors of size n, denoted by $\mathbf{v}_1, \ldots, \mathbf{v}_n$, one by one; after seeing each vector $\mathbf{v}_i$, we output the product $\mathbf{M}\mathbf{v}_i$ before we see the next vector.*

Conjecture 2 (OMv Conjecture, Theorem 2.4 in [19]). *For any $\gamma > 0$, there is no algorithm that solves OMv in time $\mathcal{O}(n^{3-\gamma})$.*

The OMv conjecture has been used to exhibit conditional lower bounds for many dynamic problems, including those previously based on other popular problems and conjectures, such as 3SUM and combinatorial Boolean matrix multiplication [19]. This also applies to the nullary triangle query: For any $\gamma > 0$ and database of domain size n, there is no algorithm that incrementally maintains the query result under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(n^{1-\gamma})$ update time, and $\mathcal{O}(n^{2-\gamma})$ answer time, unless the OMv conjecture fails [6]. All aforementioned prior approaches to maintaining triangle queries do not meet this (conditional) lower bound and are thus not worst-case optimal.

1.2 Contributions of This Article

This article introduces IVM$^\epsilon$, an IVM approach for triangle queries with arbitrary free variables that exhibits a trade-off between the update time, the space, and the enumeration delay.

Theorem 3. *Given a database $\mathbf{D}$ and $\epsilon \in [0, 1]$, IVM$^\epsilon$ incrementally maintains the triangle queries under single-tuple updates to $\mathbf{D}$ with $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ preprocessing time and $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon, 1-\epsilon\}})$ amortized update time. The space complexity and enumeration delay are given in Table 1:*

	$\triangle_0$	$\triangle_1$	$\triangle_2$	$\triangle_3$
Space	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{3}{2}})$
Enumeration delay	$\mathcal{O}(1)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{2\min\{\epsilon,1-\epsilon\}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{\epsilon,1-\epsilon\}})$	$\mathcal{O}(1)$

Table 1: IVM$^\epsilon$'s space and enumeration delay for maintaining triangle queries.

The preprocessing time is the time to compute the query result on the initial database before the updates; if we start with the empty database, then this is $\mathcal{O}(1)$. IVM$^\epsilon$ maintains triangle queries with repeating relation symbols with the same complexities from Theorem 3.

IVM$^\epsilon$ uses a data structure that partitions each input relation based on the degrees of data values. The degree of an A-value a in relation R is the number of B-values paired with a in R. The degree of B- and C-values is defined analogously. Depending on whether a combination of relation parts includes data values with high or low degrees, IVM$^\epsilon$ uses a different maintenance strategy. Thanks to this degree-based adaptive processing, the overall update time of IVM$^\epsilon$ is kept sublinear. As the database evolves under updates, IVM$^\epsilon$ needs to rebalance the relation partitions to account for updated degrees of data values. While this rebalancing may take superlinear time, it remains sublinear per single-tuple update. The overall update time is therefore amortized.

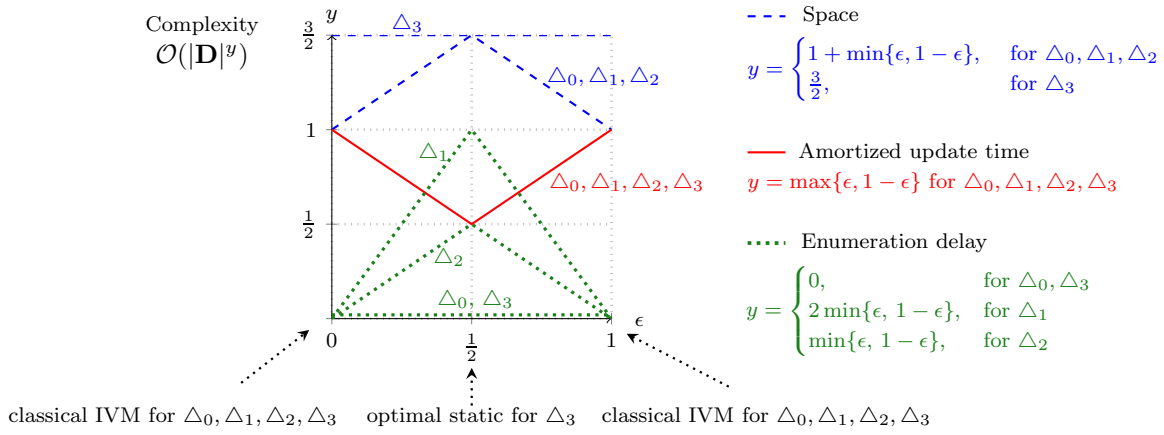


Figure 1: IVM$^\epsilon$'s amortized update time, space, and enumeration delay for maintaining triangle queries. $|\mathbf{D}|$ is the database size. The complexities are parameterized by ϵ. The space and enumeration delay depend on the arity of the query result. By setting ϵ to 0 or 1, IVM$^\epsilon$ recovers classical first-order IVM. For $\epsilon = \frac{1}{2}$, IVM$^\epsilon$ computes the ternary triangle query worst-case optimally.

We distinguish two types of relation partitioning. In *single partitioning*, relations are partitioned based on the degrees of data values in one column. In *double partitioning*, relations are partitioned based on the degrees of data values in two columns. Unary and binary triangle queries require double partitioning to obtain the complexity results in Theorem 3. For the nullary and ternary triangle queries, single partitioning suffices to obtain these complexity results. Nevertheless, double partitioning can lower the space complexity in case of the nullary triangle query, as stated next.

Proposition 4. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0, 1]$*, IVM*$^\epsilon$ *incrementally maintains the nullary triangle query under single-tuple updates to* $\mathbf{D}$ *with* $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ *preprocessing time,* $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon, 1-\epsilon\}})$ *amortized update time,* $\mathcal{O}(|\mathbf{D}|^{\max\{1, \min\{1+\epsilon, 2-2\epsilon\}\}})$ *space complexity, and* $\mathcal{O}(1)$ *enumeration delay.*

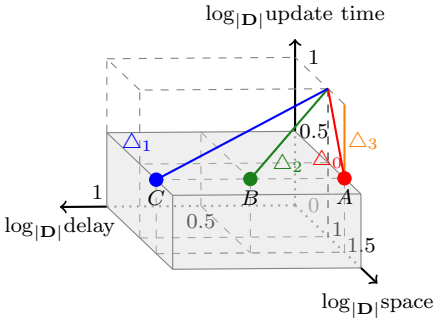
For $\epsilon = 0$ and $\epsilon \geq \frac{1}{2}$, the space complexity needed by IVM$^\epsilon$ to maintain the nullary triangle query becomes linear; its maximum is $\mathcal{O}(|\mathbf{D}|^{4/3})$ for $\epsilon = \frac{1}{3}$.

As depicted in Figure 1, IVM$^\epsilon$ defines a continuum of maintenance approaches that exhibit a trade-off between amortized update time, enumeration delay, and space based on the parameter ϵ, which ranges from 0 to 1. We can recover the classical first-order IVM for all triangle queries by setting ϵ to 0 or 1. For $\epsilon = \frac{1}{2}$, IVM$^\epsilon$ recovers the worst-case optimal time $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ of non-incremental algorithms for computing all tuples in the result of the ternary triangle query [32]. Whereas these static algorithms are monolithic and require processing the input data in bulk and joining all relations at once, IVM$^\epsilon$ achieves the same complexity by inserting $|\mathbf{D}|$ tuples one at a time in initially empty relations by using its update mechanism and binary join plans. Using binary join plans in the static case is suboptimal, since they can lead to intermediate results that are larger than the final result [32].

The following proposition shows that some combinations of update time and delay in the update-delay space are not possible, conditioned on the OMv Conjecture 2.

Proposition 5. *For any* $\gamma > 0$ *and database* $\mathbf{D}$*, there is no algorithm that incrementally maintains the result of any triangle query under single-tuple updates to* $\mathbf{D}$ *with arbitrary preprocessing time,* $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ *amortized update time, and* $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ *enumeration delay, unless the OMv conjecture fails.*

Figure 2 visualizes IVM$^\epsilon$'s trade-offs between space complexity, amortized update time, and enumeration delay for the maintenance of triangle queries. The preprocessing time is $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ for all triangle queries. The gray cuboid is infinite in the dimension of space complexity. Each point strictly included in the gray cuboid



ϵ	Query	Pareto optimality	Amortized update time	Enumeration delay
$\frac{1}{2}$	$\triangle_0$ and $\triangle_3$	strong (A)	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{1}{2}})$	$\mathcal{O}(1)$
$\frac{1}{2}$	$\triangle_2$	weak (B)	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{1}{2}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{1}{2}})$
$\frac{1}{2}$	$\triangle_1$	weak (C)	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{1}{2}})$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$

Figure 2: (left) IVM$^\epsilon$'s trade-offs between space complexity, amortized update time, and enumeration delay for the maintenance of triangle queries. The preprocessing time is $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ for all triangle queries. There is no algorithm that can maintain a triangle query with update time and enumeration delay representing a point in the gray cuboid, unless the OMv conjecture fails (Proposition 5). The surface of the gray cuboid corresponds to Pareto worst-case optimal combinations of amortized update time and enumeration delay. (right) IVM$^\epsilon$ is strongly Pareto optimal at point A for $\triangle_0$ and $\triangle_3$ and weakly Pareto optimal at point B and C for $\triangle_2$ and respectively $\triangle_1$. $\epsilon = \frac{1}{2}$ for points A, B, and C.

corresponds to a combination of some space complexity, $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ enumeration delay for $\gamma > 0$ (note that γ may be different for update and delay). Due to Proposition 5, there is no maintenance algorithm for triangle queries that admits a trade-off corresponding to a point in the gray cuboid, unless the OMv conjecture fails. Each point on the surface of the gray cuboid corresponds to a Pareto worst-case optimal trade-off between the amortized update time and enumeration delay. For $\epsilon = \frac{1}{2}$, IVM$^\epsilon$ needs $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ amortized update time and, depending on the query, an enumeration delay such that the trade-off between these two measures is Pareto optimal. For the nullary and ternary triangle queries, the delay is $\mathcal{O}(1)$ (Point A in Figure 2). IVM$^\epsilon$ is strongly Pareto worst-case optimal for these queries: There can be no tighter upper bound for any of the update time or delay measures without loosening the upper bound for the other measure. For the unary and binary triangle queries, the delay is $\mathcal{O}(|\mathbf{D}|)$ (Point C in Figure 2) and respectively $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ (Point B in Figure 2). IVM$^\epsilon$ is only weakly Pareto worst-case optimal for the unary and binary triangle queries: There are no tighter upper bounds for both the update time and delay measures. Nevertheless, either the update time or the delay may still be lowered for the unary query without contradicting the OMv conjecture. As for the binary query, only the update time may be lowered, since the delay is already below the $\mathcal{O}(|\mathbf{D}|)$ threshold from Proposition 5.

Corollary 6 summarizes the above discussion on the worst-case optimality of IVM$^\epsilon$.

Corollary 6 (Theorem 3 and Proposition 5)**.** *Under a single-tuple update to the database* $\mathbf{D}$*, IVM*$^\epsilon$ *with* $\epsilon = \frac{1}{2}$ *is strongly Pareto worst-case optimal for the nullary and ternary triangle queries and weakly Pareto worst-case optimal for the unary and binary triangle queries in the update-delay space, unless the* OMv *conjecture fails.*

1.3 Structure of This Article

Section 2 introduces the preliminaries. Sections 3 to 6 introduce IVM$^\epsilon$ for the nullary, ternary, binary, and unary triangle queries. IVM$^\epsilon$ for the nullary triangle query needs three techniques to achieve the complexities in Theorem 3: delta processing, materialization of auxiliary views, and adaptive maintenance strategy depending on the degree of values in one of the columns of the input relations. For the ternary triangle query IVM$^\epsilon$ additionally uses the concept of view trees. IVM$^\epsilon$ for unary and binary triangle queries exploits the degree of values in both columns of relations. It also uses two union algorithms: one for enumerating the distinct tuples in projections of views and one for enumerating the distinct tuples in unions of views. The lower bound in Proposition 5 is proven in Section 9. Section 10 details how IVM$^\epsilon$ recovers

existing dynamic and static approaches for triangle queries. Section 11 relates the results of this article to existing work. Section 12 discusses several extensions of IVM$^\epsilon$. Conclusion and future work are given in Section 13.

2 Preliminaries

2.1 Data Model and Query Language

A schema $\mathbf{X} = (X_1, \ldots, X_n)$ is a tuple of distinct variables. Each variable X_i has a discrete domain $\mathsf{Dom}(X_i)$. By $\mathbf{F} \subseteq \mathbf{X}$, we mean that $\mathbf{F}$ is a schema that consists of a subset of the variables in $\mathbf{X}$. A tuple $\mathbf{x}$ over schema $\mathbf{X}$ is an element from $\mathsf{Dom}(\mathbf{X}) = \mathsf{Dom}(X_1) \times \ldots \times \mathsf{Dom}(X_n)$. We use uppercase letters for variables and lowercase letters for data values. Likewise, we use bold uppercase letters for schemas and bold lowercase letters for tuples of data values.

A relation K over schema $\mathbf{X}$ is a function $K : \mathsf{Dom}(\mathbf{X}) \to \mathbb{Z}$ mapping tuples over $\mathbf{X}$ to integers such that $K(\mathbf{x}) \neq 0$ for finitely many tuples $\mathbf{x}$. A tuple $\mathbf{x}$ is in K, denoted by $\mathbf{x} \in K$, if $K(\mathbf{x}) \neq 0$. The value $K(\mathbf{x})$ represents the multiplicity of $\mathbf{x}$ in K. The size $|K|$ of K is the size of the set $\{\mathbf{x} \mid \mathbf{x} \in K\}$. A database $\mathbf{D}$ is a set of relations, and its size $|\mathbf{D}|$ is the sum of the sizes of the relations in $\mathbf{D}$.

Given a tuple $\mathbf{x}$ over schema $\mathbf{X}$ and $\mathbf{F} \subseteq \mathbf{X}$, we write $\mathbf{x}[\mathbf{F}]$ to denote the restriction of $\mathbf{x}$ onto the variables in $\mathbf{F}$ such that the values in $\mathbf{x}[\mathbf{F}]$ follow the ordering in $\mathbf{F}$. For instance, if the tuple (a, b, c) is over the schema (A, B, C), then it holds $(a, b, c)[(C, A)] = (c, a)$. For a relation K over $\mathbf{X}$, and a tuple $\mathbf{t} \in \mathsf{Dom}(\mathbf{F})$, $\sigma_{\mathbf{F}=\mathbf{t}}K$ denotes the set of tuples in K that agree with $\mathbf{t}$ on the variables in $\mathbf{F}$, that is, $\sigma_{\mathbf{F}=\mathbf{t}}K = \{\, \mathbf{x} \mid \mathbf{x} \in K \wedge \mathbf{x}[\mathbf{F}] = \mathbf{t} \,\}$. We write $\pi_{\mathbf{F}}K$ to denote the set of restrictions of the tuples in K onto $\mathbf{F}$, that is, $\pi_{\mathbf{F}}K = \{\, \mathbf{x}[\mathbf{F}] \mid \mathbf{x} \in K \,\}$.

Query Language We express queries and view definitions in the language of functional aggregate queries (FAQ) [2]. Compared to the original FAQ definition that uses several commutative semirings, we define queries over the single commutative ring $(\mathbb{Z}, +, \cdot, 0, 1)$ of integers with the usual addition and multiplication[1]. A query Q has one of the two forms:

1. Given a set $\{X_i\}_{i\in[n]}$ of variables and an index set $S \subseteq [n]$, let $\mathbf{X}_S$ denote a tuple $(X_i)_{i\in S}$ of variables and $\mathbf{x}_S$ denote a tuple of data values over the schema $\mathbf{X}_S$. Then,

$$Q(\mathbf{x}_{[f]}) = \sum_{x_{f+1}\in\mathsf{Dom}(X_{f+1})} \cdots \sum_{x_n\in\mathsf{Dom}(X_n)} \prod_{S\in\mathcal{M}} K_S(\mathbf{x}_S), \text{ where:}$$

   - $\mathcal{M}$ is a multiset of index sets.
   - For every index set $S \in \mathcal{M}$, $K_S : \mathsf{Dom}(\mathbf{X}_S) \to \mathbb{Z}$ is a relation over the schema $\mathbf{X}_S$.
   - $\mathbf{X}_{[f]}$ is the tuple of free variables of Q. The variables $X_{f+1}, \ldots, X_n$ are called bound.

2. $Q(\mathbf{x}) = Q_1(\mathbf{x}) + Q_2(\mathbf{x})$, where Q_1 and Q_2 are queries over the same tuple of free variables.

In the following, we use $\sum_{x_i}$ as a shorthand for $\sum_{x_i\in\mathsf{Dom}(X_i)}$.

Updates and Delta Queries. An update δK to a relation K is a relation over the schema of K. A single-tuple update, written as $\delta K = \{\mathbf{x} \mapsto m\}$, maps the tuple $\mathbf{x}$ to the nonzero multiplicity $m \in \mathbb{Z}$ and any other tuple to 0; that is, $|\delta K| = 1$. The data model and query language make no distinction between

[1]Previous work shows how the data-intensive computation of different applications can be captured by application-specific rings [33].

inserts and deletes – these are updates represented as relations in which tuples have positive and negative multiplicities[2].

Given a query Q and an update δK, the delta query δQ defines the change in the query result after applying δK to the database. The rules for deriving delta queries follow from the associativity, commutativity, and distributivity of the ring operations. Recall that relations and queries are functions mapping tuples of data values to multiplicities.

Query $Q(\mathbf{x})$	Delta query $\delta Q(\mathbf{x})$
$Q_1(\mathbf{x}_1) \cdot Q_2(\mathbf{x}_2)$	$\delta Q_1(\mathbf{x}_1) \cdot Q_2(\mathbf{x}_2) + Q_1(\mathbf{x}_1) \cdot \delta Q_2(\mathbf{x}_2) + \delta Q_1(\mathbf{x}_1) \cdot \delta Q_2(\mathbf{x}_2)$
$\sum_x Q_1(\mathbf{x}_1)$	$\sum_x \delta Q_1(\mathbf{x}_1)$
$Q_1(\mathbf{x}) + Q_2(\mathbf{x})$	$\delta Q_1(\mathbf{x}) + \delta Q_2(\mathbf{x})$
$K'(\mathbf{x})$	$\delta K(\mathbf{x})$ when $K = K'$ and 0 otherwise

2.2 Data Partitioning

Our maintenance approach partitions each input relation based on the degrees of its values and uses different maintenance strategies for values of high and low frequency.

Definition 7 (Single Relation Partition)**.** *Given a relation K over schema $\mathbf{X}$, a variable X from the schema $\mathbf{X}$, and a threshold θ, the pair (K^H, K^L) of relations is a* single partition *of K on X with threshold θ if it satisfies the following conditions:*

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

The pair (K^H, K^L) is called a strict partition of K on X with threshold θ if it satisfies the union and domain partition conditions and the following strict versions of the heavy and light part conditions:

$$\begin{aligned}
&\textit{(strict heavy part)} && \textit{for all } x \in \pi_X K^H : \ |\sigma_{X=x} K^H| \geq \theta \\
&\textit{(strict light part)} && \textit{for all } x \in \pi_X K^L : \ |\sigma_{X=x} K^L| < \theta
\end{aligned}$$

The relations K^H and K^L are called the heavy and light parts of K.

Definition 7 admits multiple ways to (non-strictly) partition a relation K with threshold θ. For instance, assume that $|\sigma_{X=x}K| = \theta$ for some X-value x in K. Then, all tuples in K with X-value x can be in either the heavy or light part of K; but they cannot be in both parts because of the domain partition condition. If the partition is strict, then all such tuples are in the heavy part of K. The strict partition of a relation K is unique for a given threshold and can be computed in time linear in the size of K.

To improve the time and space complexity of our maintenance approach, we may partition input relations based on the degrees of values of two variables.

Definition 8 (Double Relation Partition)**.** *Given a relation K over schema $\mathbf{X}$, distinct variables X and Y from the schema $\mathbf{X}$, and a threshold θ, let (K_X^H, K_X^L) and (K_Y^H, K_Y^L) be partitions of K on X and respectively on Y with threshold θ, and let $K^{HH} = K_X^H \cap K_Y^H$, $K^{HL} = K_X^H \cap K_Y^L$, $K^{LH} = K_X^L \cap K_Y^H$, and $K^{LL} = K_X^L \cap K_Y^L$. The tuple $(K^{HH}, K^{HL}, K^{LH}, K^{LL})$ is a* double partition *of K on (X, Y) with threshold θ.*

Let (K^H, K^L) be a single partition of a relation K on variable X and $(K^{HH}, K^{HL}, K^{LH}, K^{LL})$ a double partition of K on the pair (X, Y) with some threshold θ. We say that X is heavy in K^H, K^{HH} and K^{HL}

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

and light in K^L, K^{LH}, and K^{LL}. Similarly, Y is heavy in K^{HH} and K^{LH} and light in K^{HL} and K^{LL}. Observe the following implications of Definitions 7 and 8 to the heavy variables in relation parts. It holds $|\sigma_{X=x}K^H| \geq \frac{1}{2}\theta$ for any X-value x in K^H. However, if $K' \in \{K^{HH}, K^{HL}\}$ and x is an X-value in K', this means that $|\sigma_{X=x}K| \geq \frac{1}{2}\theta$, but not necessarily $|\sigma_{X=x}K'| \geq \frac{1}{2}\theta$. The same holds for the degrees of Y-values in K^{HH} and K^{LH}.

Notation. Our maintenance approach focuses on triangle queries and constructs auxiliary views over parts of relations R, S, and T. We use an indexing scheme for such views to capture which parts of R, S, and T are used in their definition. We write V^{rst} to denote a view V over the parts of R, S, and T specified by components r, s, and t, respectively. For component r, H means R^H; L means R^L; (HH) means R^{HH}; similarly for (HL), (LH), and (LL); and symbol $\boxminus$ means the entire relation R (i.e., the union of all parts of R). A similar convention holds for s and t .

For example, V^{HHH} denotes a view defined over the heavy parts of R, S, and T; $V^{\boxminus HL}$ denotes a view defined over R, S^H, and T^L; $V^{(LH)\boxminus H}$ denotes a view defined over R^{LH}, S, and T^H.

2.3 Computational Model

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

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

Computation Time Our maintenance approach first constructs a data structure that represents the result of a given triangle query on a database $\mathbf{D}$ and then maintains the data structure under a sequence of single-tuple updates. In our analysis, we consider the following computation times: (1) the *preprocessing time* is the time spent on initializing the data structure using $\mathbf{D}$ before any update is received, (2) the *update time* is the time spent on updating the data structure after one single-tuple update, and (3) the *enumeration delay* is the time spent until reporting the first tuple, the time between reporting two consecutive tuples, and the time between reporting the last tuple and the end of enumeration. For the nullary triangle query, the enumeration delay is the time spent on reporting the triangle count. We consider two types of bounds on the update time: *worst-case bounds*, which limit the time each individual update takes in the worst case, and *amortized worst-case bounds*, which limit the average worst-case time taken by a sequence of updates. When referring to sublinear time, we mean $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ for some $\gamma > 0$, where $|\mathbf{D}|$ is the database size.

```
UnionNext(iterators I_1, ..., I_n) : tuple
1  if (n = 1) return I_n.Next()
2  if ( (t = UnionNext(I_1, ..., I_{n-1})) ≠ EOF )
3     if ( I_n.Contains(t) )
4        return I_n.Next()
5     else
6        return t
7  return I_n.Next()
```

Figure 3: Given iterators $I_1, \ldots, I_n$ over (possibly non-disjoint) sets $S_1, \ldots, S_n$, UnionNext enumerates the distinct elements in $\bigcup_{i\in[n]} S_i$. Each iterator I_i supports two functions: I_i.Next() returns the next element in S_i if it exists and **EOF** otherwise; and I_i.Contains(t) checks whether element t exists in the set S_i.

2.4 Enumeration Algorithms

2.4.1 Iterators over Materialized Views

Each materialized view provides the iterator interface to allow the enumeration of its tuples. Each iterator maintains a pointer to the last reported tuple and supports two functions: Next() returns the next tuple in the view with a non-zero multiplicity if it exists or **EOF** otherwise; Contains($\mathbf{x}$) checks if a tuple $\mathbf{x}$ exists in the view without altering the iterator's pointer. The functions Next() and Contains($\mathbf{x}$) take constant time. Enumerating all tuples in a view amounts to repeatedly invoking the function Next() on its iterator until reaching **EOF**.

2.4.2 Enumerating Unions of Sets

Given possibly non-disjoint sets $S_1, \ldots, S_n$ the union algorithm enumerates the distinct elements in $\bigcup_{i\in[n]} S_i$ [17]. Figure 3 shows the function UnionNext that takes as input the iterators over $S_1, \ldots, S_n$ and based on the current iterator states (i.e., iterator pointers), returns the next element in $\bigcup_{i\in[n]} S_i$ or **EOF** if none. The case $n = 1$ simply returns the next element in S_n. For $n = 2$, the algorithm returns elements from S_1 only if they do not exist in S_2 (Line 6); otherwise, it returns the next element from S_2 (Line 4). The Next call in Line 4 always succeeds as it is made $|S_1 \cap S_2|$ times before exhausting S_1. After S_1 is exhausted, the algorithm returns the remaining elements from S_2. The case $n > 2$ is reduced to the binary case by treating $\bigcup_{i\in[n-1]} S_i$ as the first set and S_n as the second set.

Lemma 9. *Let $I_1, \ldots, I_n$ be iterators over sets $S_1, \ldots, S_n$, respectively, such that each iterator I_i allows lookups in S_i in time $\mathcal{O}(l)$ and enumeration of the elements in S_i with delay $\mathcal{O}(d)$. The function* UnionNext$(I_1, \ldots, I_n)$ *enumerates $\bigcup_{i\in[n]} S_i$ with $\mathcal{O}(nl + nd)$ delay.*

Proof. The case $n = 1$ follows trivially from the algorithm. We consider the case $n = 2$. Each element in $S_1 - S_2$ is reported from S_1 and all remaining elements from S_2; hence, each element from $S_1 \cup S_2$ is reported exactly once. In the worst case, we need one Contains() call in S_2 and two Next() calls before reporting the next element. Thus, the enumeration delay is $\mathcal{O}(l + d)$. The general case $n > 2$ follows by simple induction. □

An alternative method for enumerating the distinct elements in a union of sets uses skip pointers [7]. This method allows "jumping" over already reported values when iterating over these sets. To capture this idea, we first introduce the abstraction of a hop iterator, an extension of the classical iterator capable of invalidating values and omitting them during iteration. We then show how to enumerate the distinct elements in a union of sets using hop iterators.

```
NextHop( ): value
1  curr = Hop(C.Next(curr))
2  return curr
```

```
IsEmpty( ): bool
1  first = C.Next(BOF)
2  return Hop(first) = EOF
```

```
OpenHop( )
1  curr = BOF
```

```
Hop(value x): value
1  if (x ∈ skipTo)
2     return skipTo[x]
3  return x
```

```
HopBack(value x): value
1  if (x ∈ skippedFrom)
2     return skippedFrom[x]
3  return x
```

```
Exclude(value x)
1  if (not C.Contains(x)) return
2  to = Hop(C.Next(x))
3  from = HopBack(x)
4  skipTo[from] = to
5  skippedFrom[to] = from
```

Figure 4: Hop iterator over a collection **C** of values with no duplicates. The iterator maintains a pointer *curr* to the current value and two initially-empty dictionaries *skipTo* and *skippedFrom* mapping values to values. **BOF** and **EOF** represent special values before the first and after the last value in **C**. The collection **C** supports **C**.Contains(**x**) for checking the existence of x in **C** and **C**.Next(x) for finding the successor of x in **C**.

2.4.3 Hop Iterators over Collections

Consider a collection **C** of values with no duplicates. The collection supports **C**.Contains(x) for checking the existence of x in **C** and **C**.Next(x) for finding the successor of x in **C**. An iterator over **C** allows enumerating the values in **C** using the standard Volcano-style Open( ) and Next( ) functions. In addition to that, a *hop iterator* can invalidate an arbitrary value x in **C** using the Exclude(x) function. Such invalidated values are omitted during iteration. The hop iterator also ensures a constant amount of work per reported value.

Figure 4 defines the operations of a hop iterator over collection **C**. The hop iterator maintains a pointer *curr* to the current value in **C**. Upon opening the iterator via OpenHop( ), *curr* points to before the first element in **C**, denoted by **BOF**. The Next( ) function returns the next valid value from **C** if it exists or **EOF** otherwise. The Exclude(x) procedure invalidates $x \in$ **C** and records this information using dictionaries *skipTo* and *skippedFrom*. The former consists of (x, y) pairs encoding that x is invalid and its next value is y, while the latter is the inverse dictionary of the former. Exclude(x) computes a range of skipped values that includes x but potentially also values before and after x, ensuring there are no consecutive ranges of skipped values. This property guarantees that reporting the next valid value or **EOF** during iteration takes constant time.

Lemma 10. *Let* **C** *be a collection of values with no duplicates that allows lookups in time* $\mathcal{O}(l)$ *and returns the successor of a value in time* $\mathcal{O}(d)$*. Constructing a hop iterator over* **C** *takes constant time, and the hop iterator can exclude an arbitrary value from* **C** *in* $\mathcal{O}(l + d)$ *time and enumerate the non-excluded values from* **C** *with* $\mathcal{O}(d)$ *delay, using* $\mathcal{O}(|\mathbf{C}|)$ *space.*

Proof. Figure 4 defines the operations of a hop iterator. OpenHop( ), Hop(x), and HopBack(x) run in constant time, assuming constant-time dictionary operations over *skipTo* and *skippedFrom*. Next( ) looks for the valid successor of the current value in $\mathcal{O}(d)$ time. Exclude(x) checks if x exists in **C**, finds the valid successor of x in **C**, and stores the range of skipped elements in $\mathcal{O}(l + d)$ total time. The iterator state includes the pointer *curr* of constant size and two dictionaries, *skipTo* and *skippedFrom*, of size at most the size of **C**. The pointer *curr* is initialized to **BOF**, and the two dictionaries are initially empty. Thus, constructing the iterator state takes constant time. □

2.4.4 Enumerating Unions of Sets using Hop Iterators

We now design an iterator that uses hop iterators to enumerate the distinct elements in the union $\bigcup_{i\in[n]} S_i$ of possibly non-disjoint sets $S_1, \ldots S_n$. This union iterator first enumerates the elements from S_1, then those

Iterator state

$buckets[i]$ = iterator over elements of set $S_i, i \in [n]$
$I_{buckets}$ = iterator over $buckets$,
$I_{current}$ = iterator over elements of current bucket

```
OPEN( )
1  buckets = allocate iterators for sets {S_i}_{i∈[n]}
2  I_buckets = create iterator over buckets
3  I_buckets.OPENHOP( )
4  I_current = I_buckets.NEXTHOP( )
5  I_current.OPENHOP( )
```

```
NEXT( ): tuple
1   t = I_current.NEXTHOP( )
2   if (t = EOF)
3     I_current = I_buckets.NEXTHOP( )
4     if (I_current = EOF) return EOF
5     I_currrent.OPENHOP( )
6     t = I_currrent.NEXTHOP( )
7   foreach i ∈ CANDIDATEBUCKETS(t)
8     buckets[i].EXCLUDE(t)
9     if (buckets[i].ISEMPTY( ))
10      I_buckets.EXCLUDE(buckets[i])
11  return t
```

Figure 5: Iterator for enumerating the distinct elements in the union $\bigcup_{i\in[n]} S_i$ of (possibly non-disjoint) sets $S_1, \ldots, S_n$ using hop iterators. Each set S_i is an iterable collection (bucket) of values. The function CANDIDATEBUCKETS parameterizes the iterator and serves to restrict the set of buckets that may contain a given element t; the default implementation of this function returns the set $[n]$ for any element t.

from $S_2 - S_1$, then those from $S_3 - S_2 - S_1$, and so on. Using classical iterators, this strategy would incur an enumeration delay linear in the size of these sets. Using hop iterators, however, this strategy can skip over already reported elements, for example, omit the elements from S_2 that also exist in S_1 when enumerating $S_2 - S_1$. The enumeration delay in this case would depend on the time needed to exclude a just reported element from those sets containing that element.

Figure 5 defines the iterator for enumerating the distinct elements in the union of sets $S_1, \ldots S_n$. The iterator state includes a collection of hop iterators, one for each set S_i, called buckets, an iterator $I_{buckets}$ over this collection, and an iterator $I_{current}$ denoting the current hop iterator in this collection. The OPEN( ) procedure allocates the buckets and initializes $I_{current}$ with the hop iterator for S_1. The hop iterators are lazily initialized on their first access to allow OPEN( ) to run in constant time. The NEXT( ) function reports the next valid element using $I_{current}$. On exhausting the current iterator, $I_{current}$ moves on to the next bucket if it exists or returns **EOF** otherwise (Lines 2-6).

For each returned element t, NEXT( ) also excludes t from all the buckets containing t (Lines 7-10). The CANDIDATEBUCKETS(t) function identifies the set of buckets to be examined when excluding t. This function is a parameter of the union iterator. Its default implementation returns the set $[n]$ for any element t, as in prior work [7]. However, providing a context-specific implementation of this function may restrict the number of buckets that need to be examined to exclude t, further improving the enumeration delay, as demonstrated in Sections 5.4 and 6.4. Excluding t may leave a hop iterator with no valid elements. In this case, the hop iterator itself is also excluded from the collection of hop iterators (Lines 9-10).

Lemma 11. *Let $S_1, \ldots, S_n$ be collections of elements with no duplicates such that each collection S_i allows lookups in time $\mathcal{O}(l)$ and returns the successor of a value in time $\mathcal{O}(d)$. Let* CANDIDATEBUCKETS(t) *be a function that returns a set $B \subseteq [n]$ in time $\mathcal{O}(b)$, for any value t. Constructing an iterator as per Figure 5 takes constant time, and the iterator can enumerate the elements from $\bigcup_{i\in[n]} S_i$ with $\mathcal{O}(|B|l + |B|d + b)$ delay, using $\mathcal{O}(\sum_{i\in[n]} |S_i|)$ space.*

Proof. OPEN( ) creates a hop iterator $buckets[i]$ with a unique index i for each collection S_i. The hop iterators form an array with index-based constant-time lookup and successor operations. Each hop iterator is initialized on its first access. Opening the iterator $I_{buckets}$ and getting the first hop iterator from the array take constant time. Overall, OPEN( ) runs in constant time.

The NEXT( ) function gets the next tuple from $I_{current}$ in $\mathcal{O}(d)$ time, per Lemma 10 (Lines 1 and 6).

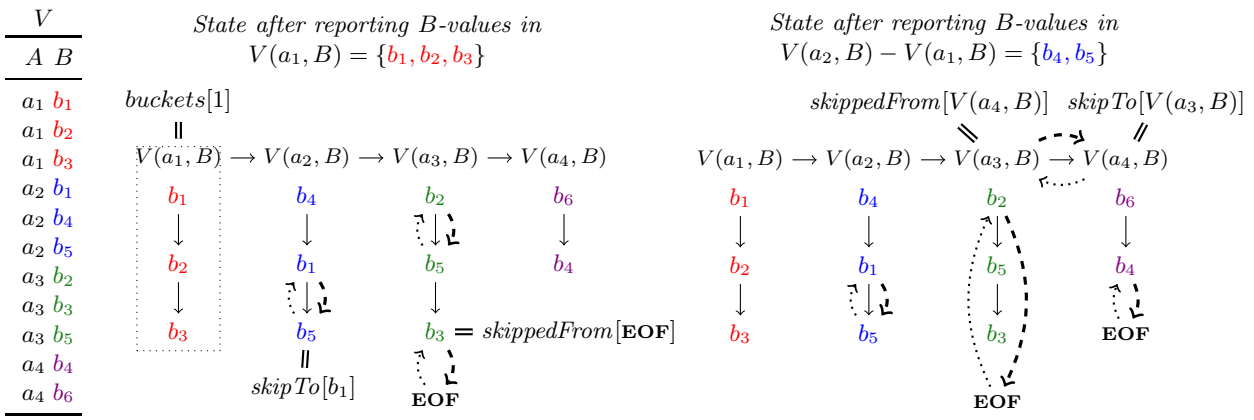


Figure 6: Using a hop-based iterator to enumerate the distinct B-values from the non-materialized view V over schema (A, B). Solid arrows represent the successor relationship among the values of V. Dotted and bold dashed arrows are hops and back hops added by the iterator during the enumeration of the distinct B-values in $\pi_B V$.

Moving on to the next bucket if it exists or returning **EOF** otherwise take constant time (Lines 3-5). The loop (Lines 7-10) runs $|B|$ times, and each loop iteration takes $\mathcal{O}(l+d)$ time to exclude t from a bucket (Line 8), $\mathcal{O}(d)$ time to check if the bucket is empty (Line 9), and constant time to exclude that bucket (Line 10), per Lemma 10. Given that CandidateBuckets runs in $\mathcal{O}(b)$ time, Next( ) takes $\mathcal{O}(|B|l + |B|d + b)$ total time. The overall space complexity directly follows from Lemma 10. □

Example 12. We illustrate the iterators for enumerating unions of sets using hop iterators described in Figures 4 and 5. Given the non-materialized view V with schema (A, B) presented in Figure 6, we show how a hop-based iterator can enumerate the distinct B-values in $\pi_B V$. We assume that the set $\{\pi_B \sigma_{A=a_i} V \mid a_i \in \pi_A V\}$ and each set $V(a_i, B) = \pi_B \sigma_{A=a_i} V$ of B-values for $i \in [4]$ support the operators Next(x) for returning the successor of x and Contains(x) for checking the existence of x.

Figure 6 visualizes two states of the hop-based iterator during the enumeration of the distinct B-values from the given view V. A vertical or horizontal solid arrow from x to y means Next(x) $= y$. Dotted and bold dashed arrows visualize hops: a dotted arrow from x to y represents $skipTo[x] = y$, while a bold dashed arrow from y to x represents $skippedFrom[y] = x$.

The B-values are reported in three stages. In Stage 1, the iterator for $\pi_B V$ reports all B-values paired with a_1; in Stage 2, it reports all B-values paired with a_2 but not with a_1; in Stage 3, it reports all B-values paired with a_4 but not with a_1, a_2, or a_3. Since all B-values paired with a_3 are also paired with a_1 or a_2, there is no stage for reporting B-values paired with a_3. The first state in Figure 6 visualizes the hop iterators at the end of Stage 1, and the second state shows the hop iterators at the end of Stages 2 and 3. We explain the three stages in more detail.

Stage 1: The Open procedure from Figure 4 initializes the iterator state by allocating an iterator $buckets[i]$ for each set in $\{V(a_i, B)\}_{i\in[4]}$ and positioning $I_{buckets}$ at $buckets[1]$ and $I_{current}$ before b_1 in the bucket for $V(a_1, B)$. The iterator then reports b_1, b_2, and b_3 from $V(a_1, B)$ and excludes b_1 from $buckets[2]$, and b_2 and b_3 from $buckets[3]$ by adding hops to their candidate buckets. At the end of Stage 1, $buckets[2]$ contains $skipTo[b_1] = b_5$ and $skippedFrom[b_5] = b_1$, and $buckets[3]$ contains $skipTo[b_2] = b_5$, $skippedFrom[b_5] = b_2$, $skipTo[b_3] =$ **EOF**, and $skippedFrom[$**EOF**$] = b_3$.

Stage 2: The iterator moves $I_{buckets}$ to $buckets[2]$ and $I_{current}$ to b_4 in $V(a_2, B)$. Then, it reports the values b_4 and b_5 in $V(a_2, B)$ but skips b_1 using the hop at this value. It excludes b_4 from $buckets[4]$ and b_5 from $buckets[3]$; for the latter, since b_5 has a hop back to b_2, and its successor b_3 has a hop to **EOF**, the iterator connects b_2 and **EOF**. Since all the B-values in $buckets[3]$ are now excluded, the iterator excludes $V(a_3, B)$ from $I_{buckets}$.

Materialized View Definition	Space Complexity
$\triangle_0() = \sum_{r,s,t \in \{H,L\}} \sum_{a,b,c} R^r(a,b) \cdot S^s(b,c) \cdot T^t(c,a)$	$\mathcal{O}(1)$
$V_{RS}(a,c) = \sum_b R^H(a,b) \cdot S^L(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V_{ST}(b,a) = \sum_c S^H(b,c) \cdot T^L(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V_{TR}(c,b) = \sum_a T^H(c,a) \cdot R^L(a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$

Figure 7: The definition and space complexity of the materialized views $\mathbf{V} = \{\triangle_0, V_{RS}, V_{ST}, V_{TR}\}$ for the nullary triangle query. The set $\mathbf{V}$ is part of an IVM$^\epsilon$ state of a database $\mathbf{D}$ partitioned for $\epsilon \in [0,1]$.

Stage 3: The iterator $I_{buckets}$ skips $V(a_3, B)$ and reaches $V(a_4, B)$. The iterator then reports b_6 while skipping b_4. The value b_6 does not appear under other A-value, hence, no hop has to be added. Since the set of A-values is exhausted, the iterator returns **EOF** and terminates.

3 Maintaining the Nullary Triangle Query

In this section, we present our strategy for maintaining the nullary triangle query

$$\triangle_0() = \sum_{a,b,c} R(a,b) \cdot S(b,c) \cdot T(c,a)$$

under a single-tuple update. We start with a high-level overview. Consider a database $\mathbf{D}$ consisting of three relations R, S, and T with schemas (A,B), (B,C), and (C,A), respectively. We partition R, S, and T on variables A, B, and C, respectively, for a given threshold. We then decompose the nullary triangle query into eight skew-aware views expressed over these relation parts:

$$\triangle_0^{rst}() = \sum_{a,b,c} R^r(a,b) \cdot S^s(b,c) \cdot T^t(c,a), \quad \text{for } r,s,t \in \{H,L\}.$$

The nullary triangle query is then the sum of these skew-aware views: $\triangle_0() = \sum_{r,s,t \in \{H,L\}} \triangle_0^{rst}()$.

IVM$^\epsilon$ adapts its maintenance strategy to each skew-aware view $\triangle_0^{rst}$ to allow for amortized update time that is sublinear in the database size. While most of these views may admit sublinear delta computation over the relation parts, few exceptions require linear-time maintenance in worst case. For these exceptions, IVM$^\epsilon$ precomputes the update-independent parts of the delta queries as *auxiliary materialized views* and then exploits these views to speed up the delta computation.

One such exception is the view $\triangle_0^{HHL}$. Consider a single-tuple update $\delta R^H = \{(\alpha,\beta) \mapsto m\}$ to the heavy part R^H of relation R, where α and β are fixed data values. Computing the delta view $\delta\triangle_0^{HHL}() = \delta R^H(\alpha,\beta) \cdot \sum_c S^H(\beta,c) \cdot T^L(c,\alpha)$ requires iterating over all the C-values c paired with β in S^H and with α in T^L; the number of such C-values can be linear in the size of the database. To avoid this iteration, IVM$^\epsilon$ precomputes the view $V_{ST}(b,a) = \sum_c S^H(b,c) \cdot T^L(c,a)$ and uses this view to evaluate $\delta\triangle_0^{HHL}() = \delta R^H(\alpha,\beta) \cdot V_{ST}(\beta,\alpha)$ in constant time.

Such auxiliary views, however, also require maintenance. All such views created by IVM$^\epsilon$ can be maintained in sublinear time under single-tuple updates to the input relations. Figure 7 summarizes these views used by IVM$^\epsilon$ to maintain the nullary triangle query: V_{RS}, V_{ST} and V_{TR}. They serve to avoid linear-time delta computation for updates to T, R, and S, respectively. IVM$^\epsilon$ also materializes the result of the nullary triangle query, which ensures constant enumeration delay.

We now describe our strategy in detail. We start by defining the state that IVM$^\epsilon$ initially creates and maintains upon each update. Then, we specify the procedure for processing a single-tuple update to any input

relation, followed by the space complexity analysis of IVM$^\epsilon$. Section 7 gives the procedure for rebalancing the partitions after a sequence of such updates.

Definition 13 (IVM$^\epsilon$ State)**.** *Let $\mathbf{D} = \{R, S, T\}$ be a database, $\triangle$ a triangle query and $\epsilon \in [0, 1]$. An IVM$^\epsilon$ state of $\mathbf{D}$ supporting the maintenance of $\triangle$ is a tuple $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$, where:*

- *N is a natural number such that the size invariant $\lfloor \frac{1}{4}N \rfloor \leq |\mathbf{D}| < N$ holds. N is called the threshold base.*
- *$\mathbf{P} = \mathcal{R} \cup \mathcal{S} \cup \mathcal{T}$ where $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ are partitions of the database relations R, S, and T, respectively, with threshold $\theta = N^\epsilon$.*
- *$\mathbf{V}$ is a set of materialized views.*

The initial state $\mathcal{Z}$ of $\mathbf{D}$ has $N = 2 \cdot |\mathbf{D}| + 1$ and the three partitions $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ are strict.

By construction, $|\mathbf{P}| = |\mathbf{D}|$. The size invariant implies $|\mathbf{D}| = \Theta(N)$ and, together with the heavy and light part conditions, it facilitates the amortized analysis of IVM$^\epsilon$ in Section 8.

For the nullary triangle query, the IVM$^\epsilon$ state has: the partitions $\mathbf{P} = \{R^H, R^L, S^H, S^L, T^H, T^L\}$ of R, S, and T on variables A, B, and C; and the set of materialized views $\mathbf{V} = \{\triangle_0, V_{RS}, V_{ST}, V_{TR}\}$ as defined in Figure 7. Definition 7 provides two essential upper bounds for each relation partition in an IVM$^\epsilon$ state: The number of distinct A-values in R^H is at most $\frac{N}{\frac{1}{2}N^\epsilon} = 2N^{1-\epsilon}$, that is, $|\pi_A R^H| \leq 2N^{1-\epsilon}$, and the number of tuples in R^L with an A-value a is less than $\frac{3}{2}N^\epsilon$, that is, $|\sigma_{A=a} R^L| < \frac{3}{2}N^\epsilon$, for any $a \in \mathsf{Dom}(A)$. The same bounds hold for B-values in $\{S^H, S^L\}$ and C-values in $\{T^H, T^L\}$.

3.1 Preprocessing Stage

The preprocessing stage for the nullary triangle query constructs the initial IVM$^\epsilon$ state given a database $\mathbf{D}$ and $\epsilon \in [0, 1]$.

Proposition 14. *Given a database $\mathbf{D}$ and $\epsilon \in [0, 1]$, constructing the initial IVM$^\epsilon$ state of $\mathbf{D}$ supporting the maintenance of the nullary triangle query takes $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time.*

Proof. We analyze the time to construct the initial state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ of $\mathbf{D}$. Retrieving the size $|\mathbf{D}|$ and computing $N = 2 \cdot |\mathbf{D}| + 1$ take constant time. Strictly partitioning the input relations from $\mathbf{D}$ using the threshold N^ϵ, as described in Definition 7, takes $\mathcal{O}(|\mathbf{D}|)$ time. Computing the result of the nullary triangle query on $\mathbf{D}$ (or $\mathbf{P}$) using the algorithms Leapfrog TrieJoin or Recursive-Join takes $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time [32]. Computing the auxiliary views V_{RS}, V_{ST}, and V_{TR} takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon, 1-\epsilon\}})$ time, as shown next. Consider the view $V_{RS}(a, c) = \sum_b R^H(a, b) \cdot S^L(b, c)$. To compute V_{RS}, one can iterate over all (a, b) pairs in R^H and then find the C-values in S^L for each b. The relation part S^L contains at most N^ϵ distinct C-values for any B-value, which gives an upper bound of $|R^H| \cdot N^\epsilon$ on the size of V_{RS}. Alternatively, one can iterate over all (b, c) pairs in S^L and then find the A-values in R^H for each b. The relation part R^H contains at most $N^{1-\epsilon}$ distinct A-values, which gives an upper bound of $|S^L| \cdot N^{1-\epsilon}$ on the size of V_{RS}. The number of steps needed to compute this result is upper-bounded by $\min\{\, |R^H| \cdot N^\epsilon, |S^L| \cdot N^{1-\epsilon} \,\} < \min\{\, N \cdot N^\epsilon, N \cdot N^{1-\epsilon} \,\} = N^{1+\min\{\epsilon, 1-\epsilon\}}$. From $|\mathbf{D}| = \Theta(N)$ follows that computing V_{RS} on the database partition $\mathbf{P}$ takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon, 1-\epsilon\}})$ time; the analysis for V_{ST} and V_{TR} is analogous. Note that $\max_{\epsilon \in [0,1]}\{1+\min\{\epsilon, 1-\epsilon\}\} = \frac{3}{2}$. Overall, the initial state $\mathcal{Z}$ of $\mathbf{D}$ can be constructed in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time. □

The preprocessing stage of IVM$^\epsilon$ happens *before* any update is received. In case we start from an empty database, the preprocessing cost of IVM$^\epsilon$ is $\mathcal{O}(1)$.

3.2 Space Complexity

We analyze the space complexity of the IVM$^\epsilon$ maintenance strategy for the nullary triangle query.

Proposition 15. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0,1]$*, an IVM*$^\epsilon$ *state of* $\mathbf{D}$ *supporting the maintenance of the nullary triangle query takes* $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$ *space.*

Proof. We consider a state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ of database $\mathbf{D}$. N and ϵ take constant space and $|\mathbf{P}| = |\mathbf{D}|$. Figure 7 summarizes the space complexity of the materialized views $\triangle_0$, V_{RS}, V_{ST}, and V_{TR} from $\mathbf{V}$. The result of $\triangle_0$ takes constant space. As discussed in the proof of Proposition 14, to compute the view $V_{RS}(a,c) = \sum_b R^H(a,b) \cdot S^L(b,c)$, we can use either R^H or S^L as the outer relation:

$$|V_{RS}| \leq \min\{\, |R^H| \cdot \max_{b \in \pi_B S^L} |\sigma_{B=b} S^L|,\ |S^L| \cdot \max_{b \in \pi_B R^H} |\sigma_{B=b} R^H| \,\} < \min\{\, N \cdot \frac{3}{2} N^\epsilon, N \cdot 2N^{1-\epsilon} \,\}$$

The size of V_{RS} is thus $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$. From $|\mathbf{D}| = \Theta(N)$ follows that V_{RS} takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$ space; the space analysis for V_{ST} and V_{TR} is analogous. Overall, the state $\mathcal{Z}$ of $\mathbf{D}$ supporting the maintenance of the nullary triangle query takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$ space. □

3.3 Processing a Single-Tuple Update

We describe the IVM$^\epsilon$ strategy for maintaining the nullary triangle query under a single-tuple update to the relation R. This update can affect either the heavy or light part of R partitioned on A, hence we write δR^r, where r stands for H or L. We can check in constant time whether the update affects R^H or R^L (cf. computational model in Section 2.3). The update is represented as a relation $\delta R^r = \{\, (\alpha, \beta) \mapsto m \,\}$, where α and β are data values and $m \in \mathbb{Z}$. Due to the symmetry of the nullary triangle query and auxiliary views, updates to S and T are handled similarly.

Figure 8 gives the procedure ApplyUpdate that takes as input a current IVM$^\epsilon$ state $\mathcal{Z}$ and the update δR^r, and returns a new state that results from applying δR^r to $\mathcal{Z}$. The procedure computes the deltas of the skew-aware views referencing R^r, which are $\delta\triangle_0^{rHH}$ (Line 3), $\delta\triangle_0^{rHL}$ (Line 4), $\delta\triangle_0^{rLH}$ (Line 5), and $\delta\triangle_0^{rLL}$ (Line 6), and uses these deltas to maintain the nullary triangle query (Line 7). These skew-aware views are not materialized, but their deltas facilitate the maintenance of the nullary triangle query. If the update affects the heavy part R^H of R, the procedure maintains V_{RS} (Line 9) and R^H (Line 12); otherwise, it maintains V_{TR} (Line 11) and R^L (Line 12). The view V_{ST} remains unchanged as it has no reference to R^H or R^L.

Figure 8 also gives the time complexity of computing these deltas and applying them to $\mathcal{Z}$. This complexity is either constant or dependent on the number of C-values for which matching tuples in the parts of S and T have nonzero multiplicities.

Proposition 16. *Given a database* $\mathbf{D}$*,* $\epsilon \in [0,1]$*, and an IVM*$^\epsilon$ *state* $\mathcal{Z}$ *of* $\mathbf{D}$ *supporting the maintenance of the nullary triangle query, IVM*$^\epsilon$ *maintains* $\mathcal{Z}$ *under a single-tuple update to any input relation in* $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ *time.*

Proof. We analyze the running time of the procedure from Figure 8 given a single-tuple update $\delta R^r = \{(\alpha,\beta) \mapsto m\}$ and a state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ of $\mathbf{D}$. Since the query and auxiliary views are symmetric, the analysis for updates to S and T is similar.

We first analyze the evaluation strategies for the deltas of the skew-aware views $\triangle_0^{rst}$:

- (Line 3) Computing $\delta\triangle_0^{rHH}$ requires summing over C-values (α and β are fixed). The minimum degree of each C-value in T^H is $\frac{1}{2}N^\epsilon$, which means the number of distinct C-values in T^H is at most $\frac{N}{\frac{1}{2}N^\epsilon} = 2N^{1-\epsilon}$. Thus, this delta evaluation takes $\mathcal{O}(N^{1-\epsilon})$ time.
- (Line 4) Computing $\delta\triangle_0^{rHL}$ requires constant-time lookups in δR^r and V_{ST}.

ApplyUpdate(update δR^r, state $\mathcal{Z}$)		Time
1	**let** $\delta R^r = \{(\alpha, \beta) \mapsto m\}$	
2	**let** $\mathcal{Z} = (\epsilon, N, \{R^H, R^L, S^H, S^L, T^H, T^L\}, \{\triangle_0, V_{RS}, V_{ST}, V_{TR}\})$	
3	$\delta\triangle_0^{rHH}() = \delta R^r(\alpha, \beta) \cdot \sum_c S^H(\beta, c) \cdot T^H(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
4	$\delta\triangle_0^{rHL}() = \delta R^r(\alpha, \beta) \cdot V_{ST}(\beta, \alpha)$	$\mathcal{O}(1)$
5	$\delta\triangle_0^{rLH}() = \delta R^r(\alpha, \beta) \cdot \sum_c S^L(\beta, c) \cdot T^H(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{\epsilon, 1-\epsilon\}})$
6	$\delta\triangle_0^{rLL}() = \delta R^r(\alpha, \beta) \cdot \sum_c S^L(\beta, c) \cdot T^L(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
7	$\triangle_0() = \triangle_0() + \delta\triangle_0^{rHH}() + \delta\triangle_0^{rHL}() + \delta\triangle_0^{rLH}() + \delta\triangle_0^{rLL}()$	$\mathcal{O}(1)$
8	**if** (r is H)	
9	$\quad V_{RS}(\alpha, c) = V_{RS}(\alpha, c) + \delta R^H(\alpha, \beta) \cdot S^L(\beta, c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
10	**else**	
11	$\quad V_{TR}(c, \beta) = V_{TR}(c, \beta) + T^H(c, \alpha) \cdot \delta R^L(\alpha, \beta)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
12	$R^r(\alpha, \beta) = R^r(\alpha, \beta) + \delta R^r(\alpha, \beta)$	$\mathcal{O}(1)$
13	**return** $\mathcal{Z}$	
	Total update time:	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\max\{\epsilon, 1-\epsilon\}})$

Figure 8: (left) Maintaining the nullary triangle query under a single-tuple update. ApplyUpdate takes as input an update δR^r to one of the parts R^H and R^L of relation R, hence $r \in \{H, L\}$, and the current IVM$^\epsilon$ state $\mathcal{Z}$ of a database $\mathbf{D}$ partitioned using $\epsilon \in [0, 1]$. It returns a new state that results from applying δR^r to $\mathcal{Z}$. Lines 3-6 compute the deltas of the affected skew-aware views, and Line 7 maintains $\triangle_0$. Lines 9 and 11 maintain the auxiliary views V_{RS} and V_{TR}, respectively. Line 12 maintains the affected part R^r. (right) The time complexity of computing and applying deltas. The evaluation strategy for computing $\delta\triangle_0^{rLH}$ in Line 5 may choose either S^L or T^H to bound C-values, depending on ϵ. The total time is the maximum of all individual times. The maintenance procedures for S and T are similar.

- (Line 5) Computing $\delta\triangle_0^{rLH}$ can be done in two ways, depending on ϵ: either sum over at most $2N^{1-\epsilon}$ C-values in T^H for the given α or sum over at most $\frac{3}{2}N^\epsilon$ C-values in S^L for the given β. This delta computation takes at most $\min\{2N^{1-\epsilon}, \frac{3}{2}N^\epsilon\}$ constant-time operations, thus $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ time.
- (Line 6) Computing $\delta\triangle_0^{rLL}$ requires summing over at most $\frac{3}{2}N^\epsilon$ C-values in S^L for the given β. This delta computation takes $\mathcal{O}(N^\epsilon)$ time.

Maintaining the nullary triangle query using these deltas takes constant time (Line 7). The views V_{RS} and V_{TR} are maintained for updates to distinct parts of R. Maintaining V_{RS} requires iterating over at most $\frac{3}{2}N^\epsilon$ C-values in S^L for the given β (Line 9); similarly, maintaining V_{TR} requires iterating over at most $2N^{1-\epsilon}$ C-values in T^H for the given α (Line 11). Finally, maintaining the part of R affected by δR^r takes constant time (Line 12). The total update time is $\mathcal{O}(\max\{1, N^\epsilon, N^{1-\epsilon}, N^{\min\{\epsilon, 1-\epsilon\}}\}) = \mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$. From the invariant $|\mathbf{D}| = \Theta(N)$ follows the claimed time complexity $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon, 1-\epsilon\}})$. □

3.4 Improving Space by Double Partitioning

We show how the space complexity of maintaining $\triangle_0$ can be improved to $\mathcal{O}(|\mathbf{D}|^{\max\{1, \min\{1+\epsilon, 2-2\epsilon\}\}})$ by double partitioning each input relation (cf. Proposition 4). This partitioning strategy allows us to obtain tighter bounds on the sizes of the materialized views. For $\epsilon = 0$ and $\epsilon \geq \frac{1}{2}$, the space complexity becomes linear; for $\epsilon = \frac{1}{3}$ it reaches its maximum $\mathcal{O}(|\mathbf{D}|^{4/3})$. Recall that the maximum space complexity under single partitioning is $\mathcal{O}(|\mathbf{D}|^{3/2})$ (Proposition 15).

We double partition the input relations R, S, and T on (A, B), (B, C), and (C, A), respectively, with the

Materialized View Definition	Space Complexity
$\triangle_0() = \sum_{r,s,t \in \{H,L\}^2} \sum_{a,b,c} R^r(a,b) \cdot S^s(b,c) \cdot T^t(c,a)$	$\mathcal{O}(1)$
$V_{RS}(a,c) = \sum_b R^{HL}(a,b) \cdot S^{LH}(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{1+\epsilon,2-2\epsilon\}})$
$V_{ST}(b,a) = \sum_c S^{HL}(b,c) \cdot T^{LH}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{1+\epsilon,2-2\epsilon\}})$
$V_{TR}(c,b) = \sum_a T^{HL}(c,a) \cdot R^{LH}(a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{1+\epsilon,2-2\epsilon\}})$

Figure 9: The definition and space complexity of the materialized views for the nullary triangle query under double partitioning. The set of views are part of an IVM$^\epsilon$ state of database $\mathbf{D}$ partitioned for $\epsilon \in [0,1]$.

threshold N^ϵ. We decompose the nullary triangle query into a union of skew-aware views:

$$\triangle_0^{rst}() = \sum_{a,b,c} R^r(a,b) \cdot S^s(b,c) \cdot T^t(c,a), \quad \text{for } r,s,t \in \{H,L\}^2.$$

Figure 9 gives the definitions of the materialized views under double partitioning. Under this refined partitioning strategy, each of the auxiliary views V_{RS}, V_{ST}, and V_{TR} has both of its free variables heavy in one of the relation parts defining the view. For instance, the view V_{RS} has the free variable A heavy in R^{HL} and the free variable C heavy in S^{LH}.

The IVM$^\epsilon$ state supporting the maintenance of the nullary triangle query under double partitioning has the partitions $\mathbf{P} = \{R^r, S^s, T^t\}_{r,s,t \in \{H,L\}^2}$ of R, S, and T on (A,B), (B,C), and (C,A), respectively; and the materialized views $\mathbf{V} = \{\triangle_0, V_{RS}, V_{ST}, V_{TR}\}$ defined in Figure 9.

The complexity analysis of maintaining the nullary triangle query under double partitioning is similar to that from the proofs of Propositions 14, 15, and 16. The preprocessing time and the maintenance time under a single-tuple update are the same as in the case of single partitioning. But the space complexity under double partitioning is improved.

Proposition 17. *Let $\mathbf{D}$ be a database and $\epsilon \in [0,1]$.*

- *The initial IVM$^\epsilon$ state with double partitioning for the maintenance of the nullary triangle query can be constructed in $\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{3}{2}})$ time.*
- *Any IVM$^\epsilon$ state with double partitioning for the maintenance of the nullary triangle query takes $\mathcal{O}(\lvert\mathbf{D}\rvert^{\max\{1,\min\{1+\epsilon,2-2\epsilon\}\}})$ space.*

Proof. Consider an IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ of $\mathbf{D}$ with double partitioning. Assume first that $\mathcal{Z}$ is the initial IVM$^\epsilon$ state. We analyze the time to construct $\mathcal{Z}$. Retrieving the database size $\lvert\mathbf{D}\rvert$ and computing $N = 2 \cdot \lvert\mathbf{D}\rvert + 1$ take constant time. For each input relation, strictly partitioning on both variables and then intersecting the relation parts to form the double partition (see Definition 8) take linear time. Thus, computing the partitions from $\mathbf{P}$ takes linear time. The materialized views in $\mathbf{V}$ can be computed in time $\mathcal{O}(N^{\frac{3}{2}})$ using the same strategies as in the proof of Proposition 14 and treating R, S, and T as partitioned only on A, B, and C, respectively.

Now, assume that $\mathcal{Z}$ is *any* IVM$^\epsilon$ state of $\mathbf{D}$. We investigate its space complexity. The components ϵ and N need constant space, and $\lvert\mathbf{P}\rvert = \lvert\mathbf{D}\rvert$. Figure 9 gives the definition and space complexity of each materialized view from $\mathbf{V}$. The size of $\triangle_0$ is constant.

We analyze the space complexity of the view $V_{RS}(a,c) = \sum_b R^{HL}(a,b) \cdot S^{LH}(b,c)$. From the proof of Proposition 15 follows that the size of V_{RS} under single partitioning is bounded by $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$. The double partitioning of R and S tightens this upper bound. Since A is heavy in R^{HL} and C is heavy in S^{LH}, the number of (A,C)-values in the result of V_{RS} is bounded by $2N^{1-\epsilon} \cdot 2N^{1-\epsilon} = 4N^{2-2\epsilon}$. Then, the size of V_{RS} is $\mathcal{O}(\min\{N^{1+\min\{\epsilon,1-\epsilon\}}, N^{2-2\epsilon}\})$, which simplifies to $\mathcal{O}(N^{\min\{1+\epsilon,2-2\epsilon\}})$ since $2-2\epsilon \leq 2-\epsilon$ for $\epsilon \in [0,1]$. The analyses for V_{ST} and V_{TR} are similar.

Considering all the components of state $\mathcal{Z}$, the size of $\mathcal{Z}$ is $\mathcal{O}(\max\{1, N, N^{\min\{1+\epsilon,2-2\epsilon\}}\})$, which simplifies to $\mathcal{O}(N^{\max\{1,\min\{1+\epsilon,2-2\epsilon\}\}})$.

From $|\mathbf{D}| = \Theta(N)$ follows the claimed preprocessing time and space complexity. □

Proposition 18. *Given a database* $\mathbf{D}$*,* $\epsilon \in [0,1]$*, and an IVM*$^\epsilon$ *state* $\mathcal{Z}$ *of* $\mathbf{D}$ *supporting the maintenance of the nullary triangle query with double partitioning, IVM*$^\epsilon$ *maintains* $\mathcal{Z}$ *under a single-tuple update to any input relation in* $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ *time.*

Proof. Consider an IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ and an update $\delta R^r = \{(\alpha, \beta) \mapsto m\}$, for $r \in \{H, L\}^2$. Most deltas of the skew-aware views can be computed in time $\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})$ using the same strategies as in the proof of Proposition 16 and treating the relations as single partitioned. The refined partitioning strategy splits the problematic case involving S^H and T^L into new cases involving S^{HH} and S^{HL} on one side and T^{LH} and T^{LL} on the other side. We next analyze the complexity of computing the deltas in these four cases:

- Computing $\delta\triangle_0^{r(HH)(LH)}$ and $\delta\triangle_0^{r(HH)(LL)}$ requires summing over at most $2N^{1-\epsilon}$ C-values paired with β in S^{HH}; thus, computing these deltas takes $\mathcal{O}(N^{1-\epsilon})$ time.
- Computing $\delta\triangle_0^{r(HL)(LL)}$ requires summing over less than $\frac{3}{2}N^\epsilon$ C-values paired with α in T^{LL}; thus, computing this delta takes $\mathcal{O}(N^\epsilon)$ time.
- Computing $\delta\triangle_0^{r(HL)(LH)}$ requires a constant-time lookup in the view V_{ST} from Figure 9.

From $|D| = \Theta(N)$ follows that $\mathcal{Z}$ can be maintained in time $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ under the single-tuple update δR^r. The analyses for updates to S and T are analogous. □

3.5 Summing Up

Materializing the query result in the IVM$^\epsilon$ state ensures constant-delay enumeration of the result. Then, our main result in Theorem 3 for the nullary triangle query follows from Propositions 14, 15, and 16 shown in the previous subsections, complemented by Proposition 33, which shows that the amortized rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$.

Proposition 4, which gives an improved space complexity for the maintenance of the nullary triangle query using double partitioning, follows from Propositions 17, 18, and 33.

4 Maintaining the Ternary Triangle Query

We now focus on the maintenance of the ternary triangle query

$$\triangle_3(a,b,c) = R(a,b) \cdot S(b,c) \cdot T(c,a)$$

under a single-tuple update. We employ a similar adaptive maintenance strategy as with the nullary triangle query. We first partition the relations R, S, and T on variables A, B, and C, respectively, with the threshold N^ϵ. We then decompose $\triangle_3$ into skew-aware views defined over the relation parts:

$$\begin{aligned}
\triangle_3^{HHH}(a,b,c) &= R^H(a,b) \cdot S^H(b,c) \cdot T^H(c,a),\\
\triangle_3^{LLL}(a,b,c) &= R^L(a,b) \cdot S^L(b,c) \cdot T^L(c,a),\\
\triangle_3^{\boxminus HL}(a,b,c) &= \sum_{r\in\{H,L\}} R^r(a,b) \cdot S^H(b,c) \cdot T^L(c,a),\\
\triangle_3^{L\boxminus H}(a,b,c) &= \sum_{s\in\{H,L\}} R^L(a,b) \cdot S^s(b,c) \cdot T^H(c,a),\\
\triangle_3^{HL\boxminus}(a,b,c) &= \sum_{t\in\{H,L\}} R^H(a,b) \cdot S^L(b,c) \cdot T^t(c,a).
\end{aligned}$$

Materialized View Definition	Space Complexity
$\triangle_3^{HHH}(a,b,c) = R^H(a,b) \cdot S^H(b,c) \cdot T^H(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{3}{2}})$
$\triangle_3^{LLL}(a,b,c) = R^L(a,b) \cdot S^L(b,c) \cdot T^L(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\frac{3}{2}})$
View tree for $\triangle_3^{HL\boxminus}(a,b,c) = \sum_{t\in\{H,L\}} R^H(a,b) \cdot S^L(b,c) \cdot T^t(c,a)$	
$V_{RS}(a,b,c) = R^H(a,b) \cdot S^L(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{RS}(a,c) = \sum_b V_{RS}(a,b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{HL\boxminus}(a,c) = \sum_{t\in\{H,L\}} \hat{V}_{RS}(a,c) \cdot T^t(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
View tree for $\triangle_3^{\boxminus HL}(a,b,c) = \sum_{r\in\{H,L\}} R^r(a,b) \cdot S^H(b,c) \cdot T^L(c,a)$	
$V_{ST}(b,c,a) = S^H(b,c) \cdot T^L(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{ST}(b,a) = \sum_c V_{ST}(b,c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{\boxminus HL}(a,b) = \sum_{r\in\{H,L\}} R^r(a,b) \cdot \hat{V}_{ST}(b,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
View tree for $\triangle_3^{L\boxminus H}(a,b,c) = \sum_{s\in\{H,L\}} R^L(a,b) \cdot S^s(b,c) \cdot T^H(c,a)$	
$V_{TR}(c,a,b) = T^H(c,a) \cdot R^L(a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{TR}(c,b) = \sum_a V_{TR}(c,a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{L\boxminus H}(b,c) = \sum_{s\in\{H,L\}} S^s(b,c) \cdot \hat{V}_{TR}(c,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$

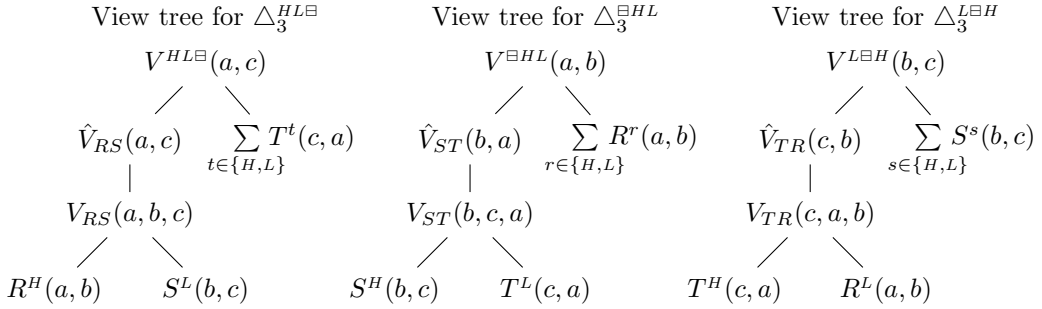


Figure 10: (top) The materialized views $\mathbf{V} = \{\triangle_3^{HHH}, \triangle_3^{LLL}, V_{RS}, \hat{V}_{RS}, V^{HL\boxminus}, V_{ST}, \hat{V}_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{L\boxminus H}\}$ supporting the maintenance of the ternary triangle query. The set $\mathbf{V}$ is part of an IVM$^\epsilon$ state of database $\mathbf{D}$. The views $\triangle_3^{HHH}$ and $\triangle_3^{LLL}$ are materialized, while the views $\triangle_3^{HL\boxminus}$, $\triangle_3^{\boxminus HL}$, and $\triangle_3^{L\boxminus H}$ allow for enumeration with constant delay using their auxiliary views denoted by indentation. (bottom) The view trees supporting the maintenance and enumeration of the results of $\triangle_3^{HL\boxminus}$, $\triangle_3^{\boxminus HL}$, and $\triangle_3^{L\boxminus H}$.

The result of $\triangle_3$ is the union of the disjoint results of these skew-aware views. To enumerate the result of $\triangle_3$, we can thus enumerate the results of these views one after the other.

As with the nullary triangle query, IVM$^\epsilon$ customizes the maintenance strategy for each of these skew-aware views and relies on auxiliary views to speed up the view maintenance.

The IVM$^\epsilon$ strategy for the nullary triangle query, however, fails to achieve sublinear maintenance time for most of these skew-aware views. Consider for instance the view $\triangle_3^{\boxminus HL}$ and a single-tuple update $\delta R^H = \{(\alpha,\beta) \mapsto m\}$ to the heavy part R^H of relation R. The delta $\delta\triangle_3^{\boxminus HL}(\alpha,\beta,c) = \delta R^H(\alpha,\beta) \cdot S^H(\beta,c) \cdot T^L(c,\alpha)$ iterates over linearly many C-values in the worst case. Precomputing the view $V_{ST}(b,c,a) = S^H(b,c) \cdot T^L(c,a)$ and rewriting the delta as $\delta\triangle_3^{\boxminus HL}(\alpha,\beta,c) = \delta R^H(\alpha,\beta) \cdot V_{ST}(\beta,c,\alpha)$ makes no improvement in the worst-case running time. In contrast, for the nullary triangle query, the view $V_{ST}(b,a) = S^H(b,c) \cdot T^L(c,a)$ enables computing $\delta\triangle_0^{HHL}$ in constant time.

The skew-aware views of the ternary triangle query can be maintained in sublinear time by avoiding

the listing (tabular) form of the view results. For that purpose, the result of a skew-aware view can be maintained in *factorized form*: Instead of using one materialized view, a hierarchy of materialized views is created such that each of them admits sublinear maintenance time and all of them together guarantee constant-delay enumeration of the result of the skew-aware view. Factorized evaluation has been previously used in the context of incremental view maintenance [6, 20, 33].

Figure 10 (top) presents the views used by IVM$^\epsilon$ to maintain the ternary triangle query under updates to the base relations. The results of the skew-aware views $\triangle_3^{HHH}$ and $\triangle_3^{LLL}$ are materialized in listing form. The remaining skew-aware views $\triangle_3^{HL\boxminus}$, $\triangle_3^{\boxminus HL}$, and $\triangle_3^{L\boxminus H}$ avoid materialization altogether but ensure constant-delay enumeration of their results using other auxiliary materialized views (denoted by indentation).

Figure 10 (bottom) shows for each of the skew-aware views $\triangle_3^{HL\boxminus}$, $\triangle_3^{\boxminus HL}$, and $\triangle_3^{L\boxminus H}$, the materialized auxiliary views needed to maintain the results of the skew-aware view in factorized form. These auxiliary views make a view tree with input relations as leaves and updates propagating in a bottom-up manner. The result of $\triangle_3^{HL\boxminus}$ is distributed among two auxiliary materialized views, $V^{HL\boxminus}$ and V_{RS}. The former stores all (a, c) pairs that would appear in the result of $\triangle_3^{HL\boxminus}$, while the latter provides the matching B-values for each (a, c) pair. The two views together provide constant-delay enumeration of the result of $\triangle_3^{HL\boxminus}$. In addition to them, the view $\hat{V}_{RS}$ serves to support constant-time updates to T^t. The view trees for $\triangle_3^{\boxminus HL}$ and $\triangle_3^{L\boxminus H}$ are analogous.

The IVM$^\epsilon$ state supporting the maintenance of the ternary triangle query has the partitions $\mathbf{P} = \{R^H, R^L, S^H, S^L, T^H, T^L\}$ of R, S, and T on variables A, B, and C; and the materialized views $\mathbf{V} = \{\triangle_3^{HHH}, \triangle_3^{LLL}, V_{RS}, \hat{V}_{RS}, V^{HL\boxminus}, V_{ST}, \hat{V}_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{L\boxminus H}\}$.

4.1 Preprocessing Stage

The preprocessing stage builds the initial IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, \mathbf{P}, \mathbf{V}, N)$ of database $\mathbf{D}$. This step partitions the input relations and computes the materialized views in $\mathbf{V}$ from Figure 10 before processing any update.

Proposition 19. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0, 1]$*, constructing the initial IVM$^\epsilon$ state of* $\mathbf{D}$ *supporting the maintenance of the ternary triangle query takes* $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ *time.*

Proof. Partitioning the input relations takes $\mathcal{O}(|\mathbf{D}|)$ time. The queries $\triangle_3^{HHH}$ and $\triangle_3^{LLL}$ can be computed using a worst-case optimal join algorithm like Leapfrog TrieJoin or Recursive-Join in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time [32]. The remaining skew-aware views $\triangle_3^{HL\boxminus}$, $\triangle_3^{\boxminus HL}$, and $\triangle_3^{L\boxminus H}$ are not materialized but represented using auxiliary views. Consider the views in the view tree for $\triangle_3^{HL\boxminus}$. Computing V_{RS} and $\hat{V}_{RS}$ takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon, 1-\epsilon\}})$ time, as explained in the proof of Proposition 14. The view $V^{HL\boxminus}$ is computed by intersecting $\hat{V}_{RS}$ and T in linear time. The same holds for the views in the view trees of $\triangle_3^{\boxminus HL}$ and $\triangle_3^{L\boxminus H}$. Overall, the preprocessing time is $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$. □

4.2 Space Complexity

We analyze the space complexity of the IVM$^\epsilon$ maintenance strategy for the ternary triangle query.

Proposition 20. *Given a database* $\mathbf{D}$*, an IVM$^\epsilon$ state of* $\mathbf{D}$ *supporting the maintenance of the ternary triangle query takes* $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ *space.*

Proof. Let $\mathcal{Z} = (\epsilon, \mathbf{P}, \mathbf{V}, N)$ be a state of $\mathbf{D}$. The size of ϵ and N is constant while the size of $\mathbf{P}$ is $\mathcal{O}(|\mathbf{D}|)$. Figure 10 summarizes the space complexities of the materialized views in $\mathbf{V}$. The size of each of the skew-aware views $\triangle_3^{HHH}$ and $\triangle_3^{LLL}$ is upper-bounded by $N^{\frac{3}{2}}$, the maximum number of triangles in a database of size N [30]. The space complexity of the auxiliary views V_{RS}, $\hat{V}_{RS}$, V_{ST}, $\hat{V}_{ST}$, V_{TR}, and $\hat{V}_{TR}$ is $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$, as discussed in the proof of Proposition 15. The sizes of the auxiliary views $V^{HL\boxminus}$, $V^{\boxminus HL}$, and $V^{L\boxminus H}$ are upper-bounded by the sizes of T, R, and S, respectively; hence, these auxiliary views take $\mathcal{O}(|\mathbf{D}|)$ space. From the invariant $|\mathbf{D}| = \Theta(N)$ follows the claimed space complexity $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$. □

ApplyUpdate(update δR^r, state $\mathcal{Z}$)		Time
1	**let** $\delta R^r = \{(\alpha, \beta) \mapsto m\}$	
2	**let** $\mathcal{Z} = (\epsilon, N, \{R^H, R^L, S^H, S^L, T^H, T^L\},$ $\{\triangle_3^{HHH}, \triangle_3^{LLL}, V_{RS}, \hat{V}_{RS}, V^{HL\boxminus}, V_{ST}, \hat{V}_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{L\boxminus H}\})$	
3	**if** (r is H)	
4	$\triangle_3^{HHH}(\alpha, \beta, c) = \triangle_3^{HHH}(\alpha, \beta, c) + \delta R^H(\alpha, \beta) \cdot S^H(\beta, c) \cdot T^H(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
5	$V_{RS}(\alpha, \beta, c) = V_{RS}(\alpha, \beta, c) + \delta R^H(\alpha, \beta) \cdot S^L(\beta, c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
6	$\hat{V}_{RS}(\alpha, c) = \hat{V}_{RS}(\alpha, c) + \delta R^H(\alpha, \beta) \cdot S^L(\beta, c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
7	$V^{HL\boxminus}(\alpha, c) = V^{HL\boxminus}(\alpha, c) + \sum_{t \in \{H,L\}} \delta R^H(\alpha, \beta) \cdot S^L(\beta, c) \cdot T^t(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
8	**else**	
9	$\triangle_3^{LLL}(\alpha, \beta, c) = \triangle_3^{LLL}(\alpha, \beta, c) + \delta R^L(\alpha, \beta) \cdot S^L(\beta, c) \cdot T^L(c, \alpha)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\epsilon})$
10	$V_{TR}(c, \alpha, \beta) = V_{TR}(c, \alpha, \beta) + T^H(c, \alpha) \cdot \delta R^L(\alpha, \beta)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
11	$\hat{V}_{TR}(c, \beta) = \hat{V}_{TR}(c, \beta) + T^H(c, \alpha) \cdot \delta R^L(\alpha, \beta)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
12	$V^{L\boxminus H}(\beta, c) = V^{L\boxminus H}(\beta, c) + \sum_{s \in \{H,L\}} T^H(c, \alpha) \cdot \delta R^L(\alpha, \beta) \cdot S^s(\beta, c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
13	$V^{\boxminus HL}(\alpha, \beta) = V^{\boxminus HL}(\alpha, \beta) + \hat{V}_{ST}(\beta, \alpha) \cdot \delta R^r(\alpha, \beta)$	$\mathcal{O}(1)$
14	$R^r(\alpha, \beta) = R^r(\alpha, \beta) + \delta R^r(\alpha, \beta)$	$\mathcal{O}(1)$
15	**return** $\mathcal{Z}$	
	Total update time:	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\max\{\epsilon, 1-\epsilon\}})$

Figure 11: (left) Maintaining an IVM$^\epsilon$ state under a single-tuple update to support constant-delay enumeration of the result of the ternary triangle query. ApplyUpdate takes as input an update δR^r to the heavy or light part of R, hence $r \in \{H, L\}$, and the current IVM$^\epsilon$ state $\mathcal{Z}$ of database $\mathbf{D}$. It returns a new state that results from applying δR^r to $\mathcal{Z}$. (right) The time complexity of computing and applying deltas. The procedures for updates to S and T are similar.

4.3 Processing a Single-Tuple Update

Figure 11 shows the procedure for maintaining a current state $\mathcal{Z}$ of the ternary triangle query under an update $\delta R^r(a, b)$. If the update affects the heavy part R^H of R, the procedure maintains $\triangle_3^{HHH}$ (Line 4) and propagates δR^H through the view tree for $\triangle_3^{HL\boxminus}$ (Lines 5-7). If the update affects the light part R^L of R, the procedure maintains $\triangle_3^{LLL}$ (Line 9) and propagates δR^L through the view tree for $\triangle_3^{L\boxminus H}$ (Lines 10-12). Finally, it updates $V^{\boxminus HL}$ (Line 13) and the part of R affected by δR^r (Line 14). The views V_{ST} and $\hat{V}_{ST}$ remain unchanged as they have no reference to R^H or R^L.

Proposition 21. *Given a database* $\mathbf{D}$*,* $\epsilon \in [0, 1]$*, and an IVM*$^\epsilon$ *state* $\mathcal{Z}$ *of* $\mathbf{D}$ *supporting the maintenance of the ternary triangle query, IVM*$^\epsilon$ *maintains* $\mathcal{Z}$ *under a single-tuple update to any input relation in* $\mathcal{O}(\lvert\mathbf{D}\rvert^{\max\{\epsilon, 1-\epsilon\}})$ *time.*

Proof. Figure 11 shows the time complexity of each maintenance statement in the ApplyUpdate procedure, for a given single-tuple update $\delta R^r = \{(\alpha, \beta) \mapsto m\}$ with $r \in \{H, L\}$ and a state $\mathcal{Z} = (\epsilon, \mathbf{P}, \mathbf{V}, N)$ of $\mathbf{D}$. This complexity is determined by the number of C-values that need to be iterated over during computing and applying the deltas of skew-aware views.

We first analyze the case when δR^r affects the heavy part R^H of R. The skew-aware view $\triangle_3^{HHH}$ (Line 4) is maintained by iterating over C-values paired with α in T^H and for each such C-value, doing constant-time lookups in the other relations and views in the maintenance statement. Since T^H is heavy on C, the number of distinct C-values iterated over in T^H is at most $2N^{1-\epsilon}$. Hence, the maintenance requires

$\mathcal{O}(N^{1-\epsilon})$ time. Each of the auxiliary views V_{RS}, $\hat{V}_{RS}$, and $V^{HL\boxminus}$ (Lines 5-7) is maintained by iterating over the C-values paired with β in S^L and doing constant-time lookups in the remaining relations and views in the corresponding maintenance statement. Since S^L is light on B, the B-value β is paired with less than $\frac{3}{2}N^\epsilon$ C-values in S^L. Thus, the auxiliary views V_{RS}, $\hat{V}_{RS}$, and $V^{HL\boxminus}$ are maintained in $\mathcal{O}(N^\epsilon)$ time.

We now consider the case when δR^r affects the light part R^L of R. Maintaining $\triangle_3^{LLL}$ (Line 9) requires iterating over less than $\frac{3}{2}N^\epsilon$ distinct C-values paired with β in S^L, which means that the maintenance requires $\mathcal{O}(N^\epsilon)$ time. Maintaining each of the auxiliary views V_{TR}, $\hat{V}_{TR}$, and $V^{L\boxminus H}$ (Line 10) requires iterating over at most $2N^{1-\epsilon}$ distinct C-values paired with α in T^H. Thus, these views can be maintained in time $\mathcal{O}(N^{1-\epsilon})$.

Maintaining $V^{\boxminus HL}$ and the part of R affected by δR^r takes constant time. Then, the total execution time of the procedure ApplyUpdate in Figure 11 is $\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})$. From the invariant $|\mathbf{D}| = \Theta(N)$ follows the claimed time complexity $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$. Due to the symmetry of the triangle query, the analysis for updates to parts of relations S and T is similar. □

4.4 Enumeration Delay

The materialized views stored in an IVM$^\epsilon$ state allow us to enumerate the tuples in the result of the ternary triangle query with constant delay.

Proposition 22. *Given an IVM$^\epsilon$ state $\mathcal{Z}$ supporting the maintenance of the ternary triangle query, IVM$^\epsilon$ enumerates the result of the query from $\mathcal{Z}$ with $\mathcal{O}(1)$ delay.*

Proof. The results of skew-aware views are disjoint, so the result of the ternary triangle query can be enumerated by enumerating the result of each skew-aware view, one after the other. Since the number of such skew-aware views is independent of the data size, it suffices to show that the result of each skew-aware view can be enumerated with constant delay to achieve an overall constant delay enumeration for the ternary triangle query.

The results of the skew-aware views $\triangle_3^{HHH}$ and $\triangle_3^{LLL}$ are materialized using the listing representation, so they admit constant-delay enumeration.

We next focus on the enumeration of the result of the skew-aware view $\triangle_3^{HL\boxminus}$. The remaining skew-aware views, $\triangle_3^{\boxminus HL}$ and $\triangle_3^{L\boxminus H}$, are treated similarly. The enumeration of the result of $\triangle_3^{HL\boxminus}$ is supported by the materialized views in its view tree from Figure 10 (left). The root $V^{HL\boxminus}$ materializes the set of all tuples (a, c) in the projection of the result of $\triangle_3^{HL\boxminus}$ onto (A, C). The view V_{RS} serves to retrieve all B-values in the result that are paired with a given tuple (a, c). Thus, enumerating the result of $\triangle_3^{HL\boxminus}$ requires iterating over the (A, C)-values in $V^{HL\boxminus}$, and for each such tuple (a, c), iterating over the B-values paired with (a, c) in V_{RS}. Based on our computational model (see Section 2.3), the B-values paired with (a, c) in V_{RS} are enumerable with constant delay. For each obtained triple (a, b, c), IVM$^\epsilon$ retrieves the correct multiplicity by looking up the multiplicities of the tuples (a, b), (b, c), and (c, a) in the leaf relations R^H, S^L, and T (i.e., the sum of the multiplicities of (c, a) in T^H and T^L), respectively, and multiplying them. These lookups are constant-time operations. Hence, the overall enumeration delay is constant. □

4.5 Summing Up

Our main result in Theorem 3 for the ternary triangle query follows from Propositions 19, 20, 21, and 22 shown in the previous subsections, complemented by Proposition 33, which shows that the amortized rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$.

5 Maintaining the Binary Triangle Query

We now consider the maintenance of the binary triangle query

$$\triangle_2(a, b) = \sum_c R(a, b) \cdot S(b, c) \cdot T(c, a)$$

Materialized View Definition	Space Complexity
$\triangle_2^{HHH}(a,b) = \sum_{s,t\in\{H,L\}} \sum_c R^H(a,b) \cdot S^{Hs}(b,c) \cdot T^{Ht}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{1,2-2\epsilon\}})$
$\triangle_2^{LLL}(a,b) = \sum_{s,t\in\{H,L\}} \sum_c R^L(a,b) \cdot S^{Ls}(b,c) \cdot T^{Lt}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\triangle_2^{H(LL)\boxminus}(a,b) = \sum_{t\in\{H,L\}^2} \sum_c R^H(a,b) \cdot S^{LL}(b,c) \cdot T^t(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\triangle_2^{L\boxminus(HH)(a,b)=\sum_{s\in\{H,L\}^2} \sum_c R^L(a,b)\cdot S^s(b,c)\cdot T^{HH}(c,a)}$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
View tree for $\triangle_2^{H(LH)\boxminus}(a,b) = \sum_{t\in\{H,L\}^2} \sum_c R^H(a,b) \cdot S^{LH}(b,c) \cdot T^t(c,a)$	
$V_{RS}(a,b,c) = R^H(a,b) \cdot S^{LH}(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{RS}(a,c) = \sum_b V_{RS}(a,b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{H(LH)\boxminus}(a,c) = \sum_{t\in\{H,L\}^2} \hat{V}_{RS}(a,c) \cdot T^t(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\hat{V}^{H(LH)\boxminus}(c) = \sum_a V^{H(LH)\boxminus}(a,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
View tree for $\triangle_2^{\boxminus HL}(a,b) = \sum_{r,s\in\{H,L\}} \sum_c R^r(a,b) \cdot S^{Hs}(b,c) \cdot T^L(c,a)$	
$V_{ST}(b,a) = \sum_{s,t\in\{H,L\}} \sum_c S^{Hs}(b,c) \cdot T^{Lt}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{\boxminus HL}(a,b) = \sum_{r\in\{H,L\}} R^r(a,b) \cdot V_{ST}(b,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
View tree for $\triangle_2^{L\boxminus(HL)}(a,b) = \sum_{s\in\{H,L\}^2} \sum_c R^L(a,b) \cdot S^s(b,c) \cdot T^{HL}(c,a)$	
$V_{TR}(c,a,b) = T^{HL}(c,a) \cdot R^L(a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{TR}(c,b) = \sum_a V_{TR}(c,a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{L\boxminus(HL)}(b,c) = \sum_{s\in\{H,L\}^2} S^s(b,c) \cdot \hat{V}_{TR}(c,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\hat{V}^{L\boxminus(HL)}(c) = \sum_b V^{L\boxminus H}(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$

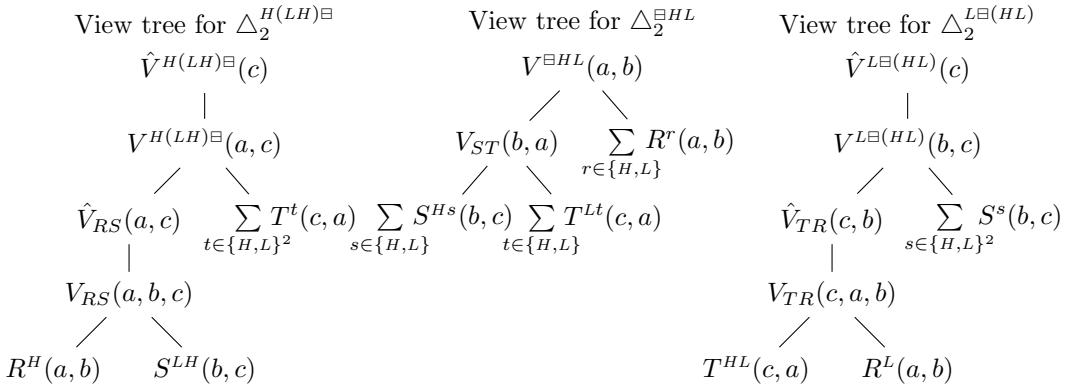


Figure 12: (top) The materialized views $\mathbf{V} = \{\triangle_2^{HHH}, \triangle_2^{LLL}, \triangle_2^{H(LL)\boxminus}, \triangle_2^{L\boxminus(HH)}, V_{RS}, \hat{V}_{RS}, V^{H(LH)\boxminus}, \hat{V}^{H(LH)\boxminus}, V_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{L\boxminus(HL)}, \hat{V}^{L\boxminus(HL)}\}$ supporting the maintenance of the binary triangle query. The set $\mathbf{V}$ is part of an IVM$^\epsilon$ state of database $\mathbf{D}$. (bottom) The view trees supporting the maintenance and enumeration of the results of $\triangle_2^{H(LH)\boxminus}$, $\triangle_2^{\boxminus HL}$, and $\triangle_2^{L\boxminus(HL)}$.

under a single-tuple update. Compared to the strategy for the ternary triangle query, the maintenance of the binary query faces two new challenges. First, the results of the skew-aware views are not disjoint anymore, which causes difficulties in the enumeration of distinct (A,B)-values with correct multiplicities. Second, among the view trees created for the ternary triangle query from Figure 10, only the view tree for

$\triangle_3^{\boxminus HL}$ allows constant-delay enumeration of (A,B)-values, while the view trees for $\triangle_3^{HL\boxminus}$ and $\triangle_3^{L\boxminus H}$ allow constant-delay enumeration of (A,C)- and respectively (B,C)-values but not (A,B)-values.

To overcome the first difficulty, we use the union algorithm [17] presented in Section 2.4.2. We modify this algorithm to report distinct tuples in the union of the skew-aware views together with their multiplicity. Since the number of skew-aware views is independent of the data size, the overall enumeration delay is the maximum delay of the individual skew-aware views.

To overcome the second difficulty, we observe that the view trees for $\triangle_3^{HL\boxminus}$ and $\triangle_3^{L\boxminus H}$ from Figure 10 both support constant-time lookups and constant-delay enumeration of (A,B)-values for a fixed C-value. Based on this observation, we can decompose each of the two view trees into a union of view trees instantiated for the distinct C-values appearing at its root view. For each union of instantiated view trees, we can use the union algorithm to enumerate the distinct (A,B) pairs with the delay that is linear in the number of these view trees, that is, the number of distinct C-values at the root view. In the view tree for $\triangle_3^{HL\boxminus}$, the number of distinct C-values at the root can be linear in the database size; thus, the enumeration delay for $\triangle_3^{HL\boxminus}$ is $\mathcal{O}(N)$. In the view tree for $\triangle_3^{L\boxminus H}$, the number of distinct C-values is at most $2N^{1-\epsilon}$ due to the heavy part condition on C in T^H; thus, the enumeration delay for $\triangle_3^{L\boxminus H}$ is $\mathcal{O}(N^{1-\epsilon})$. Overall, the enumeration delay in this case is linear.

We can improve this enumeration delay using the enumeration algorithm with hop iterators described in Section 2.4.4. In this case, this algorithm can enumerate the distinct (A,B) pairs with the delay determined by the CandidateBuckets function, see Lemma 11. The CandidateBuckets function takes any (A,B)-value and returns a set of indices that identify the instantiated view trees that may contain the given (A,B)-value. The default implementation of this function considers all such view trees, but exploiting the skew information can asymptotically reduce their number. For the view tree for $\triangle_3^{HL\boxminus}$ and a fixed (A,B)-value, CandidateBuckets can compute the matching C-values in the materialized view V_{RS} joining R^H and S^L and retain only those C-values that exist in the root $V^{HL\boxminus}$. For a fixed (A,B)-value, the number of such C-values is less than $\frac{3}{2}N^\epsilon$ due to the light part condition on B in S^L, which gives the $\mathcal{O}(N^\epsilon)$ enumeration delay for the view $\triangle_3^{HL\boxminus}$. Similarly, for the view tree for $\triangle_3^{L\boxminus H}$ and a fixed (A,B)-value, CandidateBuckets can compute the matching C-values in the materialized view V_{TR} joining T^H and R^L and retain only those C-values that exist in the root $V^{L\boxminus H}$. The number of such C-values is at most $2N^{1-\epsilon}$ due to the heavy part condition on C in T^H, which gives the $\mathcal{O}(N^{1-\epsilon})$ enumeration delay for the view $\triangle_3^{L\boxminus H}$. Overall, the enumeration algorithm with hop pointers in this case gives $\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})$ delay.

To further improve the enumeration delay to $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ in both cases, we refine our partitioning strategy to use double partitioning for S on (B,C) and for T on (C,A). This refinement allows us to further decompose the skew-aware view $\triangle_3^{HL\boxminus}$ into two parts: one part that involves S^{LH} and ensures the number of distinct C-values paired with any (A,B)-value, thus also the enumeration delay, is $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$; and another part that involves S^{LL} and ensures the number of B-values paired with any C-value in S^{LL} is $\mathcal{O}(N^\epsilon)$, which enables the materialization of this refined skew-aware view and enumeration with constant delay. Similarly, we decompose the skew-aware view $\triangle_3^{L\boxminus H}$ into one part that involves T^{HL} and guarantees $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ enumeration delay, and another part that involves T^{HH} and enables its materialization and constant-delay enumeration. Overall, our maintenance strategy for the binary triangle query that uses double partitioning for S and T achieves $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ enumeration delay.

We explain the IVM$^\epsilon$ strategy for the binary triangle query in more detail. The strategy uses single partitioning for relation R and double partitioning for relations S and T. The partition threshold is the same as for the nullary triangle query. Figure 12 shows the definition and space complexity of the views supporting the maintenance of the binary triangle query. The skew-aware views $\triangle_2^{HHH}$, $\triangle_2^{LLL}$, $\triangle_2^{H(LL)\boxminus}$, $\triangle_2^{L\boxminus(HH)}$, and $\triangle_2^{\boxminus HL}$ are materialized and enumerable with constant delay. The views $\triangle_2^{H(LH)\boxminus}$ and $\triangle_2^{L\boxminus(HL)}$ are represented as view trees consisting of auxiliary views that support the maintenance and enumeration of the results of $\triangle_2^{H(LH)\boxminus}$ and $\triangle_2^{L\boxminus(HL)}$.

The IVM$^\epsilon$ state supporting the maintenance of the binary triangle query has the partitions $\mathbf{P} = \{R^H, R^L, S^{HH}, S^{HL}, S^{LH}, S^{LL}, T^{HH}, T^{HL}, T^{LH}, T^{LL}\}$ of R on A, of S on (B,C), and of T on (C,A); $\mathbf{V} = \{\triangle_2^{HHH}, \triangle_2^{LLL}, \triangle_2^{H(LL)\boxminus}, \triangle_2^{L\boxminus(HH)}, V_{RS}, \hat{V}_{RS}, V^{H(LH)\boxminus}, \hat{V}^{H(LH)\boxminus}, V_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{L\boxminus(HL)}, \hat{V}^{L\boxminus(HL)}\}$.

The following complexity results follow mainly from the analysis of the IVM$^\epsilon$ algorithm for the ternary

triangle query in the proofs of Propositions 19, 20, and 21.

5.1 Preprocessing Stage

The preprocessing stage builds the initial IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, \mathbf{P}, \mathbf{V}, N)$ of database $\mathbf{D}$ supporting the maintenance of the binary triangle query. This step first partitions R on A, S on (B, C), and T on (C, A) and then computes the materialized views in $\mathbf{V}$ from Figure 12 before processing any update.

Proposition 23. *Given a database $\mathbf{D}$ and $\epsilon \in [0, 1]$, constructing the initial IVM$^\epsilon$ state of $\mathbf{D}$ supporting the maintenance of the binary triangle query takes $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time.*

Proof. Partitioning the input relations takes $\mathcal{O}(N)$ time. The materialized skew-aware views $\triangle_2^{H(LL)\boxminus}$ and $\triangle_2^{L\boxminus(HH)}$ can be computed in time $\mathcal{O}(N^{3/2})$ using Leapfrog TrieJoin or Recursive-Join [32]. All other materialized views can be computed using the same strategies as in the proof of Proposition 19 and ignoring that S and T are double partitioned. Overall, the initial IVM$^\epsilon$ state can be computed in time $\mathcal{O}(N^{\frac{3}{2}})$ and the result follows from $N = \Theta(|\mathbf{D}|)$. □

5.2 Space Complexity

We analyze the space complexity of the IVM$^\epsilon$ maintenance strategy for the binary triangle query.

Proposition 24. *Given a database $\mathbf{D}$ and $\epsilon \in [0, 1]$, an IVM$^\epsilon$ state of $\mathbf{D}$ supporting the maintenance of the binary triangle query takes $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon, 1-\epsilon\}})$ space.*

Proof. Figure 12 gives the space complexity of the materialized views. The space complexities of the auxiliary views follow from the proof of Proposition 20. The sizes of $V^{H(LH)\boxminus}$, $V^{\boxminus HL}$, and $V^{L\boxminus(HL)}$ are upper bounded by the sizes of T, R, and S, respectively, while the sizes of $\hat{V}^{H(LH)\boxminus}$ and $\hat{V}^{L\boxminus(HL)}$ are upper bounded by the number of distinct C-values in S^{LH} and respectively T^{HL}.

□

5.3 Processing a Single-Tuple Update

We analyze the time complexity of maintaining an IVM$^\epsilon$ state for the binary triangle query under a single-tuple update.

Proposition 25. *Given a database $\mathbf{D}$, $\epsilon \in [0, 1]$, and an IVM$^\epsilon$ state $\mathcal{Z}$ of $\mathbf{D}$ supporting the maintenance of the binary triangle query, IVM$^\epsilon$ maintains $\mathcal{Z}$ under a single-tuple update to any input relation in $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon, 1-\epsilon\}})$ time.*

Proof. Almost all the materialized views from Figure 12 can be maintained in time $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ under single-tuple updates by following the maintenance strategies described in the proof of Proposition 21. The only new challenge is to maintain the refined views $\triangle_2^{H(LL)\boxminus}$ and $\triangle_2^{L\boxminus(HH)}$.

We analyze the maintenance time for $\triangle_2^{H(LL)\boxminus}$. For updates to R^H, we need to iterate over less than $\frac{3}{2}N^\epsilon$ C-values in S^{LL} for a fixed B-value from δR^H and do lookups in T. For updates to T, we need to iterate over less than $\frac{3}{2}N^\epsilon$ B-values in S^{LL} for a fixed C-value from δT and do lookups in R^H. For updates to S^{LL}, we need to iterate over at most $2N^{1-\epsilon}$ distinct A-values in R^H and do lookups in T. Thus, $\triangle_2^{H(LL)\boxminus}$ can be maintained in $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ time.

We now consider the maintenance time for $\triangle_2^{L\boxminus(HH)}$. For updates to R^L, we need to iterate over at most $2N^{1-\epsilon}$ C-values in T^{HH} and do lookups in S. For updates to S, we need to iterate over at most $2N^{1-\epsilon}$ A-values in T^{HH} and do lookups in R^L. For updates to T^{HH}, we need to iterate over less than $\frac{3}{2}N^\epsilon$ B-values in R^L for a fixed A-value from δT^{HH} and do lookups in S. Thus, $\triangle_2^{L\boxminus(HH)}$ can be maintained in $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ time.

The proposition follows from the above analysis and the invariant $N = \Theta(|\mathbf{D}|)$. □

EnumerateBinary(state $\mathcal{Z}$)

1 **let** $\mathcal{Z} = (\epsilon, N, \{R^H, R^L, S^{HH}, S^{HL}, S^{LH}, S^{LL}, T^{HH}, T^{HL}, T^{LH}, T^{LL}\},$
$\{\triangle_2^{HHH}, \triangle_2^{LLL}, \triangle_2^{H(LL)\boxminus}, \triangle_2^{L\boxminus(HH)}, V^{\boxminus HL}\} \cup \mathbf{V})$
2 $\mathbf{I}_1 = \{\triangle_2^{HHH}.iter(), \triangle_2^{LLL}.iter(), \triangle_2^{H(LL)\boxminus}.iter(), \triangle_2^{L\boxminus(HH)}.iter(), V^{\boxminus HL}.iter()\}$
3 $\mathbf{I}_2 = \{\triangle_2^{H(LH)\boxminus}.iter\left(\text{CandidateBuckets}^{H(LH)\boxminus}\right), \triangle_2^{L\boxminus(HL)}.iter\left(\text{CandidateBuckets}^{L\boxminus(HL)}\right)\}$
4 **while** $(((\alpha, \beta) = \text{UnionNext}(\mathbf{I}_1 \cup \mathbf{I}_2)) \neq \mathbf{EOF})$
5 $m_1 = \triangle_2^{HHH}(\alpha, \beta) + \triangle_2^{LLL}(\alpha, \beta) + \triangle_2^{H(LL)\boxminus}(\alpha, \beta) + \triangle_2^{L\boxminus(HH)}(\alpha, \beta) + V^{\boxminus HL}(\alpha, \beta)$
6 $m_2 = \sum_{t \in \{H,L\}^2} \sum_c R^H(\alpha, \beta) \cdot S^{LH}(\beta, c) \cdot T^t(c, \alpha)$
7 $m_3 = \sum_{s \in \{H,L\}^2} \sum_c R^L(\alpha, \beta) \cdot S^s(\beta, c) \cdot T^{HL}(c, \alpha)$
8 **output** $(\alpha, \beta) \mapsto (m_1 + m_2 + m_3)$

Figure 13: Enumerating the result of the binary triangle query given an IVM$^\epsilon$ state of database $\mathbf{D}$. Line 2 creates iterators over materialized skew-aware views. Line 3 creates hop-based iterators over the non-materialized skew-aware views, parameterized by the CandidateBuckets$^{H(LH)\boxminus}$ and CandidateBuckets$^{L\boxminus(HL)}$ functions. Lines 5-7 compute the multiplicity of pair (α, β) reported by the union algorithm.

5.4 Enumeration Delay

We construct an iterator for each skew-aware view of the binary triangle query and use the union algorithm from Section 2.4.2 to enumerate the distinct tuples in the union of these views. For the materialized skew-aware views $\triangle_2^{HHH}$, $\triangle_2^{LLL}$, $\triangle_2^{H(LL)\boxminus}$, $\triangle_2^{L\boxminus(HH)}$, and $\triangle_2^{\boxminus HL}$ (materialized by $V^{\boxminus HL}$), we construct iterators with constant lookup time and enumeration delay (see Section 2.4.1). For each of the non-materialized views $\triangle_2^{H(LH)\boxminus}$ and $\triangle_2^{L\boxminus(HL)}$, we first instantiate its view tree for the distinct C-values appearing at its root and then construct a hop-based iterator (see Section 2.4.4) to enumerate the distinct (A, B)-values in the union of these instantiated view trees.

Given a materialized view V, we write $V.iter()$ to denote the iterator for V. We also call the function $\triangle_2^{H(LH)\boxminus}.iter(\text{CandidateBuckets}^{H(LH)\boxminus})$ to get the hop-based iterator for $\triangle_2^{H(LH)\boxminus}$ parameterized by the CandidateBuckets$^{H(LH)\boxminus}$ function. This function intersects the C-values from the root $\hat{V}^{H(LH)\boxminus}$ and the C-values paired with a given (A, B)-value in the view V_{RS}. Similarly, the hop-based iterator for $\triangle_2^{L\boxminus(HL)}$ uses the CandidateBuckets$^{L\boxminus(HL)}$ function that intersects the C-values from the root $\hat{V}^{L\boxminus(HL)}$ and the C-values paired with a given (A, B)-value in the view V_{TR}. Both functions return a set of indices that identify the view trees instantiated for the computed C-values.

The procedure EnumerateBinary from Figure 13 enumerates the result of the binary triangle query given an IVM$^\epsilon$ state $\mathcal{Z}$. The procedure first creates the iterators over the (possibly non-disjoint) results of the skew-aware views. The union algorithm from Figure 3 takes these iterators as input and reports distinct (A, B)-values as output. For each reported (a, b), EnumerateBinary computes the multiplicity of (a, b) by summing up the multiplicities in each skew-aware view.

Proposition 26. *Given a database $\mathbf{D}$, $\epsilon \in [0, 1]$, an IVM$^\epsilon$ state $\mathcal{Z}$ of $\mathbf{D}$ supporting the maintenance of the binary triangle query, IVM$^\epsilon$ enumerates the result of the query with $\mathcal{O}(|\mathbf{D}|^{\min\{\epsilon, 1-\epsilon\}})$ delay and $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon, 1-\epsilon\}})$ additional space.*

Proof. We analyze the procedure EnumerateBinary in Figure 13. Creating the iterators over materialized views takes constant time (Line 2); the same holds for the hop-based iterators in $\mathbf{I}_2$, per Lemma 10 (Line 3). The iterators in $\mathbf{I}_1$ allow constant-time lookups and constant-delay enumeration of (A, B)-values. The hop-based iterator for $\triangle_2^{H(LH)\boxminus}$ is over at most $2N^{1-\epsilon}$ view trees instantiated for the distinct C-values appearing

at the root $\hat{V}^{H(LH)\boxminus}$. Each view tree supports constant-time lookups and constant-delay enumeration of (A,B)-values. CandidateBuckets$^{H(LH)\boxminus}$ intersects at most $\min\{\frac{3}{2}N^\epsilon, 2N^{1-\epsilon}\}$ C-values from V_{RS} for a fixed (A,B)-value and at most $2N^{1-\epsilon}$ C-values from $\hat{V}^{H(LH)\boxminus}$; thus, the returned set of indices is of size at most $\min\{\frac{3}{2}N^\epsilon, 2N^{1-\epsilon}\}$. This function runs in $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ time. Per Lemma 11, the enumeration delay of the hop-based iterator for $\triangle_2^{H(LH)\boxminus}$ is $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$. A similar analysis for $\triangle_2^{L\boxminus(HL)}$ gives the same enumeration delay.

The iterators over materialized views need constant space during enumeration. The hop-based iterators over $\triangle_2^{H(LH)\boxminus}$ and $\triangle_2^{L\boxminus(HL)}$ need space linear in the total number of their (A,B)-values, per Lemma 10. This number is upper bounded by the size of V_{RS} for the former and by the size of V_{TR} for the latter. By Proposition 24, both of these views take $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$ space.

Computing the total multiplicity m of a pair (α,β) requires computing the multiplicity of (α,β) in the result of each skew-aware view. For the materialized views with schema (A,B), this operation takes constant time (Line 5). For the non-materialized views $\triangle_2^{H(LH)\boxminus}$ and $\triangle_2^{L\boxminus(HL)}$, computing the multiplicities of (α,β) requires iterating over the matching C-values in S^{LH} and respectively T^{HL} (Lines 6-7). In both cases, the number of distinct C-values for a fixed (α,β) is at most $\min\{\frac{3}{2}N^\epsilon, 2N^{1-\epsilon}\}$. Thus, the multiplicity of the pair (α,β) can be computed in $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ time.

Overall, EnumerateBinary enumerates the result of $\triangle_2$ from $\mathcal{Z}$ with $\mathcal{O}(N^{\min\{\epsilon,1-\epsilon\}})$ delay and $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$ additional space. The proposition follows from the invariant $N = \Theta(|\mathbf{D}|)$.

□

5.5 Summing Up

The additional space used during the enumeration of the result of the binary triangle query is linear in the size of the maintained views. Hence, our main result in Theorem 3 for the binary triangle query follows from Propositions 23, 24, 25, and 26 shown in the previous subsections, complemented by Proposition 33, which shows that the amortized rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$.

6 Maintaining the Unary Triangle Query

We now focus on the maintenance and enumeration of the unary triangle query

$$\triangle_1(a) = \sum_{b,c} R(a,b) \cdot S(b,c) \cdot T(c,a)$$

under a single-tuple update. As with the binary triangle query, the results of the skew-aware views in the unary case are not necessarily disjoint. To report only the distinct A-values in the union of skew-aware views, we again rely on the union algorithm, presented in Section 2.4.2.

We discuss the enumeration of distinct A-values in the result of skew-aware views that are not materialized but represented as view trees. As a starting point for our discussion, we consider the view trees created for the ternary triangle query, see Figure 10. The view trees for $\triangle_3^{HL\boxminus}$ and $\triangle_3^{\boxminus HL}$ contain A-values at the root, thus they can support the enumeration of A-values in constant time. The view tree T for $\triangle_3^{L\boxminus H}$, however, contains (B,C)-values at its root, meaning that we need to find the distinct A-values that occur under (B,C)-values. The number of distinct (B,C)-values paired with any given A-value can be linear, meaning that a hop-based iterator from Section 2.4.4 would enumerate distinct A-values with at least linear delay.

To improve the enumeration delay for the skew-aware view $\triangle_3^{L\boxminus H}$, we refine our partitioning strategy to get a tighter bound on the number of (B,C)-values paired with any given A-value. We double partition relation R on (A,B) and relation T on (C,A) while keeping S partitioned on B. This refinement further divides $\triangle_3^{L\boxminus H}$ into three skew-aware views. One skew-aware view involves R^{LH} and T^{HL} and ensures that the number of distinct (B,C)-values paired with any A-value is bounded by $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$ since A is light in both relation parts and each of the variables B and C is heavy in at least one of the relation parts. The other two skew-aware views either involve R^{LL} or involve R^{LH} and T^{HH}, which enables their materialization

Materialized View Definition	Space Complexity
$\triangle_1^{HHH}(a) = \sum_{r,t\in\{H,L\}} \sum_{b,c} R^{Hr}(a,b) \cdot S^H(b,c) \cdot T^{Ht}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
$\triangle_1^{LLL}(a) = \sum_{r,t\in\{H,L\}} \sum_{b,c} R^{Lr}(a,b) \cdot S^L(b,c) \cdot T^{Lt}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\triangle_1^{(LL)\boxminus H}(a) = \sum_{s,t\in\{H,L\}} \sum_{b,c} R^{LL}(a,b) \cdot S^s(b,c) \cdot T^{Ht}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
$\triangle_1^{(LH)\boxminus(HH)}(a) = \sum_{s\in\{H,L\}} \sum_{b,c} R^{LH}(a,b) \cdot S^s(b,c) \cdot T^{HH}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
View tree for $\triangle_1^{HL\boxminus}(a) = \sum_{r\in\{H,L\}} \sum_{t\in\{H,L\}^2} \sum_{b,c} R^{Hr}(a,b) \cdot S^L(b,c) \cdot T^t(c,a)$	
$V_{RS}(a,c) = \sum_{r\in\{H,L\}} \sum_b R^{Hr}(a,b) \cdot S^L(b,c)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{HL\boxminus}(a) = \sum_{t\in\{H,L\}^2} \sum_c V_{RS}(a,c) \cdot T^t(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1-\epsilon})$
View tree for $\triangle_1^{\boxminus HL}(a) = \sum_{r\in\{H,L\}^2} \sum_{t\in\{H,L\}} \sum_{b,c} R^r(a,b) \cdot S^H(b,c) \cdot T^{Lt}(c,a)$	
$V_{ST}(b,a) = \sum_{t\in\{H,L\}} \sum_c S^H(b,c) \cdot T^{Lt}(c,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$V^{\boxminus HL}(a) = \sum_{r\in\{H,L\}^2} \sum_b R^r(a,b) \cdot V_{ST}(b,a)$	$\mathcal{O}(\lvert\mathbf{D}\rvert)$
View tree for $\triangle_1^{(LH)\boxminus(HL)}(a) = \sum_{s\in\{H,L\}} \sum_{b,c} R^{LH}(a,b) \cdot S^s(b,c) \cdot T^{HL}(c,a)$	
$V_{TR}(c,a,b) = T^{HL}(c,a) \cdot R^{LH}(a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-\epsilon\}})$
$\hat{V}_{TR}(c,b) = \sum_a V_{TR}(c,a,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{1+\min\{\epsilon,1-2\epsilon\}})$
$V^{(LH)\boxminus(HL)}(b,c) = \sum_{s\in\{H,L\}} S^s(b,c) \cdot \hat{V}_{TR}(c,b)$	$\mathcal{O}(\lvert\mathbf{D}\rvert^{\min\{1,2-2\epsilon\}})$

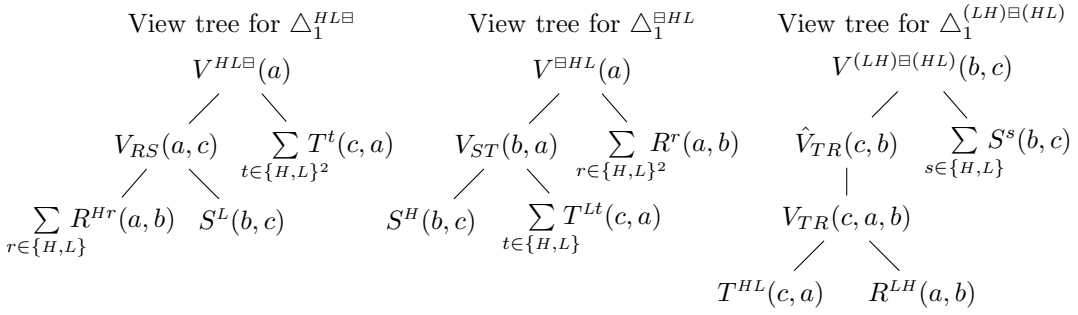


Figure 14: (top) The materialized views $\mathbf{V} = \{\triangle_1^{HHH}, \triangle_1^{LLL}, \triangle_1^{(LL)\boxminus H}, \triangle_1^{(LH)\boxminus(HH)}, V_{RS}, V^{HL\boxminus}, V_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{(LH)\boxminus(HL)}\}$ supporting the maintenance of the unary triangle query. The set $\mathbf{V}$ is part of an IVM$^\epsilon$ state of database $\mathbf{D}$. (bottom) The view trees supporting the maintenance of $\triangle_1^{HL\boxminus}$, $\triangle_1^{\boxminus HL}$, and $\triangle_1^{(LH)\boxminus(HL)}$.

and enumeration with constant delay. Overall, our maintenance strategy for the unary triangle query with double partitioning for R and T achieves $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$ enumeration delay, which is sublinear for $\epsilon \neq \frac{1}{2}$.

Figure 14 shows the definition and space complexity of the views supporting the maintenance of the unary triangle query. The IVM$^\epsilon$ state supporting the maintenance of the unary triangle query has the partitions $\mathbf{P} = \{R^{HH}, R^{HL}, R^{LH}, R^{LL}, S^H, S^L, T^{HH}, T^{HL}, T^{LH}, T^{LL}\}$ of R on (A,B), of S on B, and of T on (C,A); $\mathbf{V} = \{\triangle_1^{HHH}, \triangle_1^{LLL}, \triangle_1^{(LL)\boxminus H}, \triangle_1^{(LH)\boxminus(HH)}, V_{RS}, V^{HL\boxminus}, V_{ST}, V^{\boxminus HL}, V_{TR}, \hat{V}_{TR}, V^{(LH)\boxminus(HL)}\}$.

6.1 Preprocessing Stage

The preprocessing stage builds the initial IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, \mathbf{P}, \mathbf{V}, N)$ of database $\mathbf{D}$ supporting the maintenance of the unary triangle query.

Proposition 27. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0,1]$*, constructing the initial IVM*$^\epsilon$ *state of* $\mathbf{D}$ *supporting the maintenance of the unary triangle query takes* $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ *time.*

Proof. The proof is similar to the proof of Proposition 23. □

6.2 Space Complexity

We analyze the space complexity of the IVM$^\epsilon$ maintenance strategy for the unary triangle query.

Proposition 28. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0,1]$*, an IVM*$^\epsilon$ *state of* $\mathbf{D}$ *supporting the maintenance of the unary triangle query takes* $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$ *space.*

Proof. Figure 14 gives the definition and space complexity of the materialized views. The complexity results follow mainly from the proof of Proposition 20. The remaining views take either linear space because of their unary schema or sublinear space because of the heavy part condition on A in one of the relation parts. Two notable cases are the views $\hat{V}_{TR}$ and $V^{(LH)\boxminus(HL)}$. The size of $\hat{V}_{TR}$ is upper bounded by the size of V_{TR}, which is $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$ as discussed in the proof of Proposition 20, but also by at most $4N^{2-2\epsilon}$ (B,C)-values created by pairing the distinct heavy B-values from R^{LH} and the distinct heavy C-values from T^{HL}. Thus, the view $\hat{V}_{TR}$ takes $\mathcal{O}(N^{1+\min\{\epsilon,1-2\epsilon\}})$ space. The view view $V^{(LH)\boxminus(HL)}$ is further upper bounded by the size of S, which gives its $\mathcal{O}(N^{\min\{1,2-2\epsilon\}})$ space. The proposition follows from the invariant $N = \mathcal{O}(|\mathbf{D}|)$. □

6.3 Processing a Single-Tuple Update

We analyze the time complexity of maintaining an IVM$^\epsilon$ state for the unary triangle query under a single-tuple update.

Proposition 29. *Given a database* $\mathbf{D}$*,* $\epsilon \in [0,1]$*, and an IVM*$^\epsilon$ *state* $\mathcal{Z}$ *of* $\mathbf{D}$ *supporting the maintenance of the unary triangle query, IVM*$^\epsilon$ *maintains* $\mathcal{Z}$ *under a single-tuple update to any input relation in* $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ *time.*

Proof. Almost all materialized views in Figure 14 can be maintained following the same strategies as in the proof of Proposition 21 and by ignoring the double partitioning of R and T. The only notable cases are the refined skew-aware views $\triangle_1^{(LL)\boxminus H}$ and $\triangle_1^{(LH)\boxminus(HH)}$, considered next.

We analyze the time to maintain $\triangle_1^{(LL)\boxminus H}$. For updates to R^{LL}, we need to iterate over at most $2N^{1-\epsilon}$ C-values in T^H and do lookups in S. For updates to S, we need to iteration over less than $\frac{3}{2}N^\epsilon$ A-values in R^{LL} for a fixed B-value from δS and do lookups in T^H. For updates to T^H, we need to iterate over less than $\frac{3}{2}N^\epsilon$ B-values for a fixed A-value from δT^H and do lookups in S. Thus, maintaining $\triangle_1^{(LL)\boxminus H}$ takes $\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})$ time.

The maintenance strategies for $\triangle_1^{(LH)\boxminus(HH)}$ differ from the strategies above only in case of updates to S. For an update S, we iterate over at most $2N^{1-\epsilon}$ A-values in T^{HH} and do lookups in R^{LH}. This implies that the maintenance time is $\mathcal{O}(N^{1-\epsilon})$.

Hence, the overall maintenance time is $\mathcal{O}(N^{\max\{\epsilon,1-\epsilon\}})$. The result follows from $N = \mathcal{O}(|\mathbf{D}|)$. □

6.4 Enumeration Delay

The enumeration procedure for the unary triangle query is similar to that of the binary triangle query. The skew-aware views from Figure 14 are all materialized except $\triangle_1^{(LH)\boxminus(HL)}$. For each materialized view, we construct an iterator with constant lookup time and enumeration delay. For the non-materialized view $\triangle_1^{(LH)\boxminus(HL)}$, we first instantiate its view tree for the distinct (B,C)-values appearing at the root $V^{(LH)\boxminus(HL)}$ and then construct a hop-based iterator for enumerating the distinct A-values in the union of these view trees. The hop-based iterator is parameterized by the CandidateBuckets$^{(LH)\boxminus(HL)}$ function that restricts the set of instantiated view trees to be explored during enumeration for a fixed A-value. This function first

EnumerateUnary(state $\mathcal{Z}$)

1 **let** $\mathcal{Z} = (\epsilon, N, \{R^{HH}, R^{HL}, R^{LH}, R^{LL}, S^{H}, S^{L}, T^{HH}, T^{HL}, T^{LH}, T^{LL}\},$
$\{\triangle_1^{HHH}, \triangle_1^{LLL}, \triangle_1^{(LL)\boxminus H}, \triangle_1^{(LH)\boxminus(HH)}, V^{HL\boxminus}, V^{\boxminus HL}\} \cup \mathbf{V})$
2 $\mathbf{I}_1 = \{\triangle_1^{HHH}.iter(), \triangle_1^{LLL}.iter(), \triangle_1^{(LL)\boxminus H}.iter(), \triangle_1^{(LH)\boxminus(HH)}.iter(), V^{HL\boxminus}.iter(), V^{\boxminus HL}.iter(),\}$
3 $\mathbf{I}_2 = \{\triangle_1^{(LH)\boxminus(HL)}.iter\left(\text{CandidateBuckets}^{(LH)\boxminus(HL)}\right)\}$
4 **while** $((\alpha = \text{UnionNext}(\mathbf{I}_1 \cup \mathbf{I}_2)) \neq \textbf{EOF})$
5 $m_1 = \triangle_1^{HHH}(\alpha) + \triangle_1^{LLL}(\alpha) + \triangle_1^{(LL)\boxminus H}(\alpha) + \triangle_1^{(LH)\boxminus(HH)}(\alpha) + V^{HL\boxminus}(\alpha) + V^{\boxminus HL}(\alpha)$
6 $m_2 = \sum_{s\in\{H,L\}} \sum_{b,c} R^{LH}(\alpha, b) \cdot S^{s}(b, c) \cdot T^{HL}(c, \alpha)$
7 **output** $\alpha \mapsto (m_1 + m_2)$

Figure 15: Enumerating the result of the unary triangle query given an IVM$^\epsilon$ state of database $\mathbf{D}$. Line 2 creates six iterators over the results of materialized views with schema A. Line 3 creates a hop-based iterator over the non-materialized skew-aware view $\triangle_1^{(LH)\boxminus(HL)}$, parameterized by the CandidateBuckets$^{(LH)\boxminus(HL)}$ function. Lines 5 and 6 compute the multiplicity of α reported by the union algorithm.

computes the (B, C)-values that exist in both the materialized view V_{TR} for the given A-value and the root $V^{(LH)\boxminus(HL)}$, and then returns a set of indices that identify the view trees instantiated for those (B, C)-values.

The procedure EnumerateUnary from Figure 15 enumerates the result of the unary triangle query given an IVM$^\epsilon$ state $\mathcal{Z}$. The procedure first creates the iterators for all skew-aware views (Lines 2-3). The union algorithm (see Section 2.4.2) takes these iterators as input and reports distinct A-values as output. For each reported A-value α, EnumerateUnary sums up the multiplicity of α in each of the skew-aware views, which involves lookups in the materialized views with schema A (Line 5) and an aggregation of (B, C)-values over the relation parts from $\triangle_1^{(LH)\boxminus(HL)}$ (Line 6).

Proposition 30. *Given a database $\mathbf{D}$, $\epsilon \in [0, 1]$, an IVM$^\epsilon$ state $\mathcal{Z}$ of $\mathbf{D}$ supporting the maintenance of the unary triangle query, IVM$^\epsilon$ enumerates the query result from $\mathcal{Z}$ with $\mathcal{O}(|\mathbf{D}|^{2\min\{\epsilon,1-\epsilon\}})$ delay and $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$ additional space.*

Proof. Creating the iterators over materialized and the hop-based iterator over $\triangle_1^{(LH)\boxminus(HL)}$ takes constant time (Line 2-3), The iterators over the materialized views with schema A allow constant-time lookups and constant-delay enumeration of A-values. The hop-based iterator reports the distinct A-values from the union of at most $\min\{N, 4N^{2(1-\epsilon)}\}$ view trees instantiated for the distinct (B, C)-values in the root $V^{(LH)\boxminus(HL)}$. Each such a view tree allows constant-time lookups and constant-delay enumeration of A-values.

The CandidateBuckets$^{(LH)\boxminus(HL)}$ function, which parameterizes the hop-based iterator, first intersects the (B, C)-values from V_{TR} for a fixed A-value and from the root $V^{(LH)\boxminus(HL)}$. The number of (B, C)-values in V_{TR} is at most $4N^{2-2\epsilon}$ due to the heavy part conditions on B in R^{LH} and on C in T^{HL}, and less than $\frac{9}{4}N^{2\epsilon}$ for a fixed A-value due to the light part conditions on A in R^{LH} and on A in T^{HL}. The number of (B, C)-values in $V^{(LH)\boxminus(HL)}$ is further upper bounded by the size of S. Thus, computing the intersection and returning a set of indices that identify the matching view trees take $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$ time. The returned set of indices is of size at most $\min\{N, 4N^{2-2\epsilon}, \frac{9}{4}N^{2\epsilon}\}$. Per Lemma 11, the enumeration delay for the view $\triangle_1^{(LH)\boxminus(HL)}$ is $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$.

The iterators over materialized views require constant space during enumeration. The hop-based iterator over $\triangle_1^{(LH)\boxminus(HL)}$ requires space linear in the total number of its A-value, per Lemma 10. This number is upper bounded by the size of V_{TR}, which takes $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$ space by Proposition 28.

Computing the total multiplicity of each reported A-value α requires constant-time lookups in the materialized views with schema A (Line 5) and iteration over the distinct (B, C)-values appearing in the join

of R^{LH}, S, and T^{HL} (Line 6); since A is light in R^{LH} and T^{HL}, and each of the variables B and C is heavy in one of these relation parts, the number of such (B,C)-values is $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$. Thus, the multiplicity of the output value α can be computed in $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$ time.

Overall, EnumerateUnary enumerates the result of $\triangle_1$ from $\mathcal{Z}$ with $\mathcal{O}(N^{2\min\{\epsilon,1-\epsilon\}})$ delay and $\mathcal{O}(N^{1+\min\{\epsilon,1-\epsilon\}})$ additional space. The proposition follows from the invariant $|\mathbf{D}| = \Theta(N)$. □

6.5 Summing Up

The additional space used by the enumeration algorithm for the unary triangle query is linearly bounded by the overall space complexity of maintained views. We conclude that our main result in Theorem 3 for the unary triangle query follows from Propositions 27, 28, 29, and 30 shown in the previous subsections, complemented by Proposition 33, which shows that the amortized rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$.

7 Rebalancing Relation Partitions

The partition of a relation may change after updates. For instance, an insert $\delta R^L = \{(\alpha,\beta) \mapsto 1\}$ may violate the size invariant $\lfloor \frac{1}{4}N \rfloor \leq |\mathbf{D}| < N$ in an IVM$^\epsilon$ state or may violate the light part condition $|\sigma_{A=\alpha}R^L| < \frac{3}{2}N^\epsilon$ on data value α and require moving all tuples with A-value α from R^L to R^H. As the database evolves under updates, IVM$^\epsilon$ performs *major* and *minor* rebalancing steps to ensure that the size invariant and the heavy and light part conditions always hold. This rebalancing also ensures that the upper bounds on the number of data values, such as the number of B-values paired with α in R^L and the number of distinct A-values in R^H, are valid. The rebalancing cost is amortized over multiple updates.

The rebalancing procedures introduced in this section operate on IVM$^\epsilon$ states supporting any triangle query discussed in the previous sections. The maintenance procedure ApplyUpdate used by major and minor rebalancing is polymorphic in the sense that its definition depends on the maintained triangle query and used partitioning scheme (single or double partitioning). Sections 3.3 and 4.3 show the procedures ApplyUpdate for the nullary triangle query under single partitioning and respectively the ternary triangle query. Sections 3.4, 5.3, and 6.3 describe how to adapt these procedures for the nullary triangle query under double partitioning, the binary triangle query, and the unary triangle query, respectively.

Major Rebalancing If an update causes the database size to fall below $\lfloor \frac{1}{4}N \rfloor$ or reach N, IVM$^\epsilon$ halves or, respectively, doubles the threshold base N, and calls the procedure MajorRebalance shown in Figure 16. The procedure strictly repartitions the database relations with the new threshold N^ϵ (Line 2) and recomputes the materialized views using the new relation parts (Line 3).

Proposition 31. *Given a database* $\mathbf{D}$*, major rebalancing of an IVM$^\epsilon$ state of* $\mathbf{D}$ *supporting the maintenance of any triangle query takes* $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ *time.*

Proof. Let $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ be an IVM$^\epsilon$ state supporting the maintenance of any triangle query. Consider the procedure MajorRebalance from Figure 16. The procedure strictly repartitions the relations in $\mathbf{P}$ using the threshold N^ϵ and recomputes the materialized views in $\mathbf{V}$ based on the new relation partitions. Strictly partitioning the input relations takes $\mathcal{O}(|\mathbf{D}|)$ time. Propositions 14, 17, 19, 23, and 27 state that the computation of the initial IVM$^\epsilon$ state supporting the maintenance of any triangle query takes $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time. From the proofs of these propositions follows that the views in $\mathbf{V}$ can be recomputed in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time. □

The superlinear time of major rebalancing is amortized over $\Omega(N)$ updates. After a major rebalancing step, it holds that $|\mathbf{D}| = \frac{1}{2}N$ (after doubling), or $|\mathbf{D}| = \frac{1}{2}N - \frac{1}{2}$ or $|\mathbf{D}| = \frac{1}{2}N - 1$ (after halving, i.e., setting N to $\lfloor \frac{1}{2}N \rfloor - 1$; the two options are due to the floor functions in the size invariant and halving expression). To violate the size invariant $\lfloor \frac{1}{4}N \rfloor \leq |\mathbf{D}| < N$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}N$. Section 8 proves the amortized $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ time of major rebalancing.

```
MajorRebalance(state Z)
1 let Z = (ε, N, P, V)
2 P = StrictPartition(P, N^ε)
3 V = Recompute(V, P)
4 return Z
```

```
MoveTuples(variable X, value x, K_src → K_dst, state Z)
foreach x ∈ σ_{X=x} K_src do
    Z = ApplyUpdate(δK_dst = { x ↦ K_src(x) }, Z)
    Z = ApplyUpdate(δK_src = { x ↦ −K_src(x) }, Z)
return Z
```

```
MinorRebalance(relation K, variable X, value x, variable Y, value y, state Z)
1  if (K is single partitioned)
2     if ( x ∈ π_X K^H and |σ_{X=x} K^H| < ½N^ε)
3        Z = MoveTuples(X, x, K^H → K^L, Z)
4     else if ( x ∈ π_X K^L and |σ_{X=x} K^L| ≥ ¾N^ε)
5        Z = MoveTuples(X, x, K^L → K^H, Z)
6  else if (K is double partitioned)
7     if ( x ∈ (π_X K^HH ∪ π_X K^HL) and |σ_{X=x} K| < ½N^ε)
8        Z = MoveTuples(X, x, K^HH → K^LH, Z);   Z = MoveTuples(X, x, K^HL → K^LL, Z)
9     else if ( x ∈ (π_X K^LH ∪ π_X K^LL) and |σ_{X=x} K| ≥ ¾N^ε)
10       Z = MoveTuples(X, x, K^LH → K^HH, Z);   Z = MoveTuples(X, x, K^LL → K^HL, Z)
11    if ( y ∈ (π_Y K^HH ∪ π_Y K^LH) and |σ_{Y=y} K| < ½N^ε)
12       Z = MoveTuples(Y, y, K^HH → K^HL, Z);   Z = MoveTuples(Y, y, K^LH → K^LL, Z)
13    else if ( y ∈ (π_Y K^HL ∪ π_Y K^LL) and |σ_{Y=y} K| ≥ ¾N^ε)
14       Z = MoveTuples(Y, y, K^HL → K^HH, Z);   Z = MoveTuples(Y, y, K^LL → K^LH, Z)
15 return Z
```

Figure 16: MajorRebalance($\mathcal{Z}$) performs major rebalancing on a state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ supporting the maintenance of a triangle query. StrictPartition($\mathbf{P}, N^\epsilon$) strictly repartitions the relations in $\mathbf{P}$ with threshold N^ϵ, and Recompute($\mathbf{V}, \mathbf{P}$) recomputes the views in $\mathbf{V}$ using the partitions in $\mathbf{P}$. Given a relation K with schema (X, Y), an X-value x and a Y-value y, MinorRebalance($K, X, x, Y, y, \mathcal{Z}$) moves tuples between relation parts to ensure that the heavy and light part conditions on values x and y hold. MoveTuples($X, x, K_{src} \to K_{dst}, \mathcal{Z}$) uses ApplyUpdate to move all tuples with X-value x from relation part K_{src} to relation part K_{dst}. ApplyUpdate depends on the maintained triangle query, see Sections 3.3, 3.4, 4.3, 5.3, and 6.3.

Minor Rebalancing After each update $\delta R = \{(\alpha, \beta) \mapsto m\}$, IVM$^\epsilon$ checks whether the light and heavy part conditions still hold for α and β. If R is partitioned on variable A, the relation partition consists of the heavy part R^H and the light part R^L. By Definition 7, the heavy and light part conditions on α are $|\sigma_{A=\alpha} R^H| \geq \frac{1}{2} N^\epsilon$ and $|\sigma_{A=\alpha} R^L| < \frac{3}{2} N^\epsilon$, respectively. If the first condition is violated, all tuples in R^H with the A-value α are moved to R^L and the affected views are updated; similarly, if the second condition is violated, all tuples with the A-value α are moved from R^L to R^H, followed by updating the affected views.

If R is double partitioned on (A, B), the relation partition consists of the parts R^{HH}, R^{HL}, R^{LH}, and R^{LL}. Then, the heavy and light part conditions must be checked not only for the A-value α but also for the B-value β. From Definition 8, the heavy and light part conditions on α are $|\sigma_{A=\alpha} R| \geq \frac{1}{2} N^\epsilon$ and respectively $|\sigma_{A=\alpha} R| < \frac{3}{2} N^\epsilon$, where R is obtained by taking the union of the parts of R. If the update δR violates the first condition, all tuples with A-value α are moved from the relation parts in which A is heavy to the relation parts in which A is light, that is, from R^{HH} and R^{HL} to R^{LH} and R^{LL}, respectively. If the update violates the second condition, all tuples with A-value α are moved in the opposite direction, from R^{LH} and R^{LL} to R^{HH} and R^{HL}. In both cases, the affected views are updated. The heavy and light part conditions

on B-value β are ensured in a similar way. As a result of an update, both values α and β might change from light to heavy or vice-versa, but it is impossible that one value changes from light to heavy and the other one from heavy to light. The minor rebalancing steps followed by updates to the other relations S and T are analogous.

The procedure MinorRebalance in Figure 16 describes a minor rebalancing step on an IVM$^\epsilon$ state following an update $\delta K = \{(x, y) \mapsto m\}$ to a relation K over schema (X, Y). If K is single partitioned, the heavy and light part conditions are checked for X-value x only (Lines 1-5). If it is double partitioned, the conditions are checked for both X-value x and Y-value y (Lines 6-14). Tuples are moved between relation parts using the procedure MoveTuples in Figure 16. Given a variable X in the schema of relation K, an X-value x, a source relation part K_{src}, and a target relation part K_{dst}, the procedure MoveTuples moves all tuples with X-value x from K_{src} to part K_{dst}. A tuple $\mathbf{x}$ is moved from K_{src} to K_{dst} by using the procedure ApplyUpdate that updates the multiplicities of $\mathbf{x}$ in K_{dst} and K_{src} and maintains the materialized views in the IVM$^\epsilon$ state. Sections 3.3, 3.4, 4.3, 5.3, and 6.3 give the definition of ApplyUpdate for each triangle query. If K is single partitioned, MoveTuples is called at most once in MinorRebalance. If K is double partitioned, MoveTuples can be called up to four times, two times per x and y, to meet the heavy and light part conditions.

Proposition 32. *Given a database* $\mathbf{D}$ *and* $\epsilon \in [0, 1]$ *minor rebalancing of an IVM$^\epsilon$ state of* $\mathbf{D}$ *supporting the maintenance of any triangle query takes* $\mathcal{O}(|\mathbf{D}|^{\epsilon+\max\{\epsilon,1-\epsilon\}})$ *time.*

Proof. Consider an IVM$^\epsilon$ state $\mathcal{Z} = (\epsilon, N, \mathbf{P}, \mathbf{V})$ and an update $\delta R = \{(\alpha, \beta) \mapsto m\}$ to relation R. The analysis for updates to S and T is similar. If R is single partitioned, MinorRebalance calls MoveTuples at most once; if R is double partitioned, MinorRebalance calls MoveTuples at most four times. Consider the worst case when R is double partitioned and both values α and β change from heavy to light or vice-versa. If they change from heavy to light, the procedure moves fewer than $\frac{1}{2}N^\epsilon$ tuples with A-value α and fewer than $\frac{1}{2}N^\epsilon$ tuples with B-value β. If the two values change from light to heavy, the procedure moves fewer than $\frac{3}{2}N^\epsilon + 1$ tuples with A-value α and fewer than $\frac{3}{2}N^\epsilon + 1$ tuples with B-value β. Each tuple move performs one delete and one insert by executing ApplyUpdate. From Propositions 16, 18, 21, 25, and 29 follows that, regardless of the maintained triangle query, ApplyUpdate runs in time $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$. Since there are $\mathcal{O}(N^\epsilon)$ such operations, the procedure MinorRebalance requires $\mathcal{O}(|\mathbf{D}|^{\epsilon+\max\{\epsilon,1-\epsilon\}})$ time. As $|\mathbf{D}| = \Theta(N)$, minor rebalancing runs in time $\mathcal{O}(|\mathbf{D}|^{\epsilon+\max\{\epsilon,1-\epsilon\}})$. □

The (super)linear time of minor rebalancing is amortized over $\Omega(N^\epsilon)$ updates. This lower bound on the number of updates comes from the relation partition conditions (see Definition 7), namely from the gap between the two thresholds in these conditions. Section 8 proves the amortized $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ time of minor rebalancing.

Figure 17 gives the trigger procedure OnUpdate that maintains an IVM$^\epsilon$ state of a database $\mathbf{D}$ under a single-tuple update $\delta R = \{(\alpha, \beta) \mapsto m\}$ to relation R and, if necessary, rebalances partitions; the procedures for updates to S and T are analogous. The procedure first calls AffectedPart to determine in constant time which part R^r of R is affected by the update. We first consider the case when R is single partitioned. The update targets R^H if this relation part already contains a tuple with the same A-value α, or ϵ is set to 0; otherwise, the update targets R^L. When $\epsilon = 0$, all tuples are in R^H, while R^L remains empty. Although this behavior is not required by IVM$^\epsilon$ (without the condition $\epsilon = 0$, R^L would contain only tuples whose A-values have the degree of 1, and R^H would contain all other tuples), it allows us to recover existing IVM approaches, such as classical IVM for the nullary and ternary triangle queries; by setting ϵ to 0, IVM$^\epsilon$ ensures that all tuples are in R^H. The case when R is double partitioned is analogous. The update targets R^{HH} if R^{HH} contains the tuple (α, β) or $\epsilon = 0$; the update targets R^{HL} or R^{LH} if they already contain (α, β); otherwise, the update targets R^{LL}. The procedure OnUpdate then invokes ApplyUpdate. If the update causes a violation of the size invariant $\lfloor \frac{1}{4}N \rfloor \leq |\mathbf{D}| < N$, the procedure invokes MajorRebalance from Figure 16 to recompute the relation partitions and auxiliary views. Otherwise, if any heavy or light part condition is violated, it calls MinorRebalance from Figure 16 to move tuples between the parts of relation R and ensure that these conditions hold again.

```
OnUpdate(update δR, state Z)
1  let δR = {(α, β) ↦ m}
2  let Z = (ε, N, P, V)
3  let R^r = AffectedPart(δR, Z)
4  ApplyUpdate(δR^r = {(α, β) ↦ m}, Z)
5  if (|D| = N)
6     N = 2N
7     Z = MajorRebalance(Z)
8  else if (|D| < ⌊¼N⌋)
9     N = ⌊½N⌋ − 1
10    Z = MajorRebalance(Z)
11 else if (A is light in R^r and |σ_{A=α}R| ≥ ¾N^ε or
12          B is light in R^r and |σ_{B=β}R| ≥ ¾N^ε or
13          A is heavy in R^r and |σ_{A=α}R| < ½N^ε or
14          B is heavy in R^r and |σ_{B=β}R| < ½N^ε)
15    Z = MinorRebalance(R, A, α, B, β, Z)
16 return Z
```

```
AffectedPart(update δR, state Z)
1  let δR = {(α, β) ↦ m}
2  let Z = (ε, N, P, V)
3  if (R is single partitioned)
4     if (α ∈ π_A R^H or ε = 0)
5        return R^H
6     else
7        return R^L
8  else if (R is double partitioned)
9     if ((α, β) ∈ R^HH or ε = 0)
10       return R^HH
11    else if ((α, β) ∈ R^HL)
12       return R^HL
13    else if ((α, β) ∈ R^LH)
14       return R^LH
15    else
16       return R^LL
```

Figure 17: Maintaining an IVM$^\epsilon$ state supporting the maintenance of any triangle query under a single-tuple update and performing rebalancing. The procedure OnUpdate takes as input an update δR and an IVM$^\epsilon$ state $\mathcal{Z}$ of database $\mathbf{D}$ and returns a new state that results from applying δR to $\mathcal{Z}$ and, if necessary, rebalancing partitions. The procedure AffectedPart determines the relation part in $\mathcal{Z}$ affected by the update. ApplyUpdate depends on the maintained triangle query, see Sections 3.3, 3.4, 4.3, 5.3, and 6.3. MajorRebalance and MinorRebalance are given in Figure 16. The OnUpdate procedures for updates to S and T are analogous.

8 Amortizing Rebalancing Time

Sections 3-6 show that any IVM$^\epsilon$ state supporting the maintenance of a triangle query can be maintained in sublinear time under a single-tuple update. The sublinear maintenance time requires that the size invariant and the heavy and light part conditions are preserved for the relation partitions in IVM$^\epsilon$ states. To guarantee this, IVM$^\epsilon$ performs major and minor rebalancing steps, which can take superlinear time as stated in Propositions 31 and 32. We nevertheless show in this section that the amortized rebalancing costs and thus the overall amortized maintenance time over a sequence of updates remains sublinear.

Proposition 33. *Given a database $\mathbf{D}$, $\epsilon \in [0,1]$, and an IVM$^\epsilon$ state $\mathcal{Z}$ of $\mathbf{D}$ supporting the maintenance of any triangle query, IVM$^\epsilon$ maintains $\mathcal{Z}$ under a single-tuple update to any input relation and performs rebalancing in $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon, 1-\epsilon\}})$ amortized time.*

Proof. Let $\mathcal{Z}_0 = (\epsilon, N_0, \mathbf{P}_0, \mathbf{V}_0)$ be the initial IVM$^\epsilon$ state of a database $\mathbf{D}_0$ and $u_0, u_1, \ldots, u_{n-1}$ a sequence of arbitrary single-tuple updates. The application of this update sequence to $\mathcal{Z}_0$ yields a sequence $\mathcal{Z}_0 \xrightarrow{u_0} \mathcal{Z}_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{n-1}} \mathcal{Z}_n$ of IVM$^\epsilon$ states, where $\mathcal{Z}_{i+1}$ is the result of executing the procedure OnUpdate$(u_i, \mathcal{Z}_i)$ from Figure 17, for $0 \leq i < n$. Let c_i denote the actual execution cost of OnUpdate$(u_i, \mathcal{Z}_i)$. For some $\Gamma > 0$, we can decompose each c_i as:

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

where c_i^{apply}, c_i^{major}, and c_i^{minor} are the actual costs of the subprocedures ApplyUpdate, MajorRebalance, and MinorRebalance, respectively, in OnUpdate. If update u_i causes no major rebalancing, then

$c_i^{major} = 0$; similarly, if u_i causes no minor rebalancing, then $c_i^{minor} = 0$. These actual costs admit the following worst-case upper bounds:

$$\begin{aligned} c_i^{apply} &\le \gamma N_i^{\max\{\epsilon,1-\epsilon\}} && \text{(by Propositions 16, 18, 21, 25, 29),} \\ c_i^{major} &\le \gamma N_i^{\frac{3}{2}} && \text{(by Proposition 31), and} \\ c_i^{minor} &\le \gamma N_i^{\epsilon+\max\{\epsilon,1-\epsilon\}} && \text{(by Proposition 32),} \end{aligned}$$

where γ is a constant derived from their asymptotic bounds, and N_i is the threshold base of $\mathcal{Z}_i$. The costs of major and minor rebalancing can be superlinear in the database size.

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

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

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

$$\hat{c}_i^{apply} = \gamma N_i^{\max\{\epsilon,1-\epsilon\}}, \quad \hat{c}_i^{major} = 4\gamma N_i^{\frac{1}{2}}, \quad \hat{c}_i^{minor} = 4\gamma N_i^{\max\{\epsilon,1-\epsilon\}}, \quad \text{and}$$

Γ and γ are the constants used to upper bound the actual cost of OnUpdate. As it will be explained in more detail, the number of updates between a major rebalancing step caused by update u_i and the previous major rebalancing step can be as less as $\frac{1}{4}N_i$. In order to accumulate enough budget to pay for the major rebalancing cost triggered by update u_i, the amortized cost $\hat{c}_i^{major}$ is defined as $\gamma N_i^{\frac{3}{2}} / \frac{1}{4} N_i = 4\gamma N_i^{\frac{1}{2}}$. Given that u_i is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$ and invokes minor rebalancing for α, the number of updates since the previous minor rebalancing step for α can be as less as $\frac{1}{2}N^{\epsilon}$. Hence, to pay for the minor rebalancing step for α invoked by u_i, our budget must be at least $\gamma N_i^{\epsilon+\max\{\epsilon,1-\epsilon\}} / \frac{1}{2}N^{\epsilon} = 2\gamma N_i^{\max\{\epsilon,1-\epsilon\}}$. Since we also need to take the rebalancing costs for β into account, we define the amortized minor rebalancing cost $\hat{c}_i^{minor}$ as $4\gamma N_i^{\max\{\epsilon,1-\epsilon\}}$. In contrast to the actual costs c_i^{major} and c_i^{minor}, the amortized costs $\hat{c}_i^{major}$ and $\hat{c}_i^{minor}$ are always nonzero.

We prove that such amortized costs satisfy Inequality (1). Since $\hat{c}_i^{apply} \ge c_i^{apply}$ for $0 \le i < n$, it suffices to show that the following inequalities hold:

$$\textit{(amortizing major rebalancing)} \quad \sum_{i=0}^{n-1} \hat{c}_i^{major} \ge \sum_{i=0}^{n-1} c_i^{major} \qquad \text{and} \tag{2}$$

$$\textit{(amortizing minor rebalancing)} \quad \sum_{i=0}^{n-1} \hat{c}_i^{minor} \ge \sum_{i=0}^{n-1} c_i^{minor}. \tag{3}$$

We prove Inequalities (2) and (3) by induction on the length n of the update sequence.

Major rebalancing.

- *Base case*: We show that Inequality (2) holds for $n = 1$. The preprocessing stage sets $N_0 = 2 \cdot |\mathbf{D}_0| + 1$. If the initial database $\mathbf{D}_0$ is empty, then $N_0 = 1$ and u_0 triggers major rebalancing (and no minor rebalancing). The amortized cost $\hat{c}_0^{major} = 4\gamma N_0^{\frac{1}{2}} = 4\gamma$ suffices to cover the actual cost $c_0^{major} \le \gamma N_0^{1+\frac{1}{2}} = \gamma$. If the initial database is nonempty, u_0 cannot trigger major rebalancing (i.e., violate the size invariant) because $\lfloor \frac{1}{4} N_0 \rfloor = \lfloor \frac{1}{2} |\mathbf{D}_0| \rfloor \le |\mathbf{D}_0| - 1$ (lower threshold) and $|\mathbf{D}_0| + 1 < N_0 = 2 \cdot |\mathbf{D}_0| + 1$ (upper threshold); then, $\hat{c}_0^{major} \ge c_0^{major} = 0$. Thus, Inequality (2) holds for $n = 1$.

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

$$
\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{major} &\geq \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} \hat{c}_i^{major} && \textit{(by induction hypothesis)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} 4\gamma N_{n-1}^{\frac{1}{2}} && (N_j = \ldots = N_{n-1}) \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + \frac{1}{4} N_{n-1}\, 4\gamma N_{n-1}^{\frac{1}{2}} && \textit{(at least } \frac{1}{4} N_{n-1} \textit{ updates)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \gamma N_{n-1}^{\frac{3}{2}} \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + c_{n-1}^{major} = \sum_{i=0}^{n-1} c_i^{major} && (c_j^{major} = \ldots = c_{n-2}^{major} = 0).
\end{aligned}
$$

Thus, Inequality (2) holds for update sequences of length n.

Minor rebalancing. When the degree of a value in a partition changes such that the heavy or light part condition no longer holds, minor rebalancing moves the affected tuples between the relation parts. To prove Inequality (3), we decompose the cost of minor rebalancing per relation and data value over a variable in the schema of the relation.

$$
\begin{aligned}
c_i^{minor} &= \sum_{a \in \mathsf{Dom}(A)} (c_i^{R,a} + c_i^{T,a}) + \sum_{b \in \mathsf{Dom}(B)} (c_i^{R,b} + c_i^{S,b}) + \sum_{c \in \mathsf{Dom}(C)} (c_i^{T,c} + c_i^{R,c}) \\
\hat{c}_i^{minor} &= \sum_{a \in \mathsf{Dom}(A)} (\hat{c}_i^{R,a} + \hat{c}_i^{T,a}) + \sum_{b \in \mathsf{Dom}(B)} (\hat{c}_i^{R,b} + \hat{c}_i^{S,b}) + \sum_{c \in \mathsf{Dom}(C)} (\hat{c}_i^{T,c} + \hat{c}_i^{R,c})
\end{aligned}
$$

We write $c_i^{R,\alpha}$ and $\hat{c}_i^{R,\alpha}$ to denote the actual and respectively amortized costs of minor rebalancing caused by update u_i, for relation R and an A-value α. Recall that if update u_i is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$ and R is single partitioned, the update can cause minor rebalancing for A-value α. If R is double partitioned, the update can cause minor rebalancing for A-value α, or B-value β, or for both. Hence, if u_i is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$ and causes any rebalancing, we have $c_i^{R,\alpha} + c_i^{R,\beta} = c_i^{minor} \leq \gamma N_i^{\epsilon + \max\{\epsilon, 1-\epsilon\}}$; otherwise, $c_i^{R,\alpha} = c_i^{R,\beta} = 0$. If u_i is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$, we set $\hat{c}_i^{R,\alpha} = \hat{c}_i^{R,\beta} = \frac{1}{2} \hat{c}_i^{minor} = 2\gamma N_i^{\max\{\epsilon, 1-\epsilon\}}$ regardless of whether u_i causes minor rebalancing or not; otherwise, $\hat{c}_i^{R,\alpha} = \hat{c}_i^{R,\beta} = 0$. The actual costs $c_i^{S,b}$, $c_i^{S,c}$, $c_i^{T,c}$, and $c_i^{T,a}$ and the amortized costs $\hat{c}_i^{S,b}$, $\hat{c}_i^{S,c}$, $\hat{c}_i^{T,c}$, and $\hat{c}_i^{T,a}$ are defined similarly.

We prove that for R and any $a \in \mathsf{Dom}(A)$, the following inequality holds:

$$
\sum_{i=0}^{n-1} \hat{c}_i^{R,a} \geq \sum_{i=0}^{n-1} c_i^{R,a}. \tag{4}
$$

The proof of the inequality $\sum_{i=0}^{n-1} \hat{c}_i^{R,b} \geq \sum_{i=0}^{n-1} c_i^{R,b}$ for any $b \in \mathsf{Dom}(B)$ and the inequalities for the other two relations S and T are analogous. Inequality (3) follows directly from these inequalities.

We prove Inequality (4) for an arbitrary $a \in \mathsf{Dom}(A)$ by induction on the length n of the update sequence.

- *Base case*: We show that Inequality (4) holds for $n = 1$. Assume that update u_0 is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$; otherwise, $\hat{c}_0^{R,\alpha} = c_0^{R,\alpha} = 0$, and Inequality (4) follows trivially for $n = 1$. If the initial database is empty, u_0 triggers major rebalancing but no minor rebalancing, thus $\hat{c}_0^{R,\alpha} = 2\gamma N_0^{\max\{\epsilon, 1-\epsilon\}} \geq c_0^{R,\alpha} = 0$. If the initial database is nonempty, each relation is partitioned using the threshold N_0^ϵ. For update u_0 to trigger minor rebalancing for A-value α, the degree of α in R has to either decrease from $\lceil N_0^\epsilon \rceil$ to $\lceil \frac{1}{2} N_0^\epsilon \rceil - 1$ (heavy to light) or increase from $\lceil N_0^\epsilon \rceil - 1$ to $\lceil \frac{3}{2} N_0^\epsilon \rceil$ (light to heavy). The former happens only if $\lceil N_0^\epsilon \rceil = 1$ and update u_0 removes the last tuple with the A-value α from R, thus no minor rebalancing is needed; the latter cannot happen since update u_0 can increase $|\sigma_{A=\alpha} R|$ to at most $\lceil N_0^\epsilon \rceil$, and $\lceil N_0^\epsilon \rceil < \lceil \frac{3}{2} N_0^\epsilon \rceil$. In any case, $\hat{c}_0^{R,\alpha} \geq c_0^{R,\alpha}$, which implies that Inequality (4) holds for $n = 1$.

- *Inductive step*: Assumed that Inequality (4) holds for all update sequences of length up to $n-1$, we show that it holds for update sequences of length n. Consider that update u_{n-1} is of the form $\delta R = \{(\alpha, \beta) \mapsto m\}$ and causes minor rebalancing for α; otherwise, $\hat{c}_{n-1}^{R,\alpha} \geq 0$ and $c_{n-1}^{R,\alpha} = 0$, and Inequality (4) follows trivially for n. Let $\mathcal{Z}_j$ be the state created after the previous major rebalancing or, if there is no such step, the initial state. The threshold changes only with major rebalancing, thus $N_j = N_{j+1} = \ldots = N_{n-1}$. Depending on whether there exist minor rebalancing steps since state $\mathcal{Z}_j$, we distinguish two cases:

Case 1: There is no minor rebalancing caused by an update of the form $\delta R = \{(\alpha, \beta') \mapsto m'\}$ since state $\mathcal{Z}_j$; thus, we have $c_j^{R,\alpha} = \ldots = c_{n-2}^{R,\alpha} = 0$. From state $\mathcal{Z}_j$ to state $\mathcal{Z}_n$, the number of tuples with the A-value α either decreases from at least $\lceil N_j^\epsilon \rceil$ to $\lceil \frac{1}{2} N_{n-1}^\epsilon \rceil - 1$ (heavy to light) or increases from at most $\lceil N_j^\epsilon \rceil - 1$ to $\lceil \frac{3}{2} N_{n-1}^\epsilon \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than $\frac{1}{2} N_{n-1}^\epsilon$ since $N_j = N_{n-1}$ and $\min\{\lceil N_j^\epsilon \rceil - (\lceil \frac{1}{2} N_{n-1}^\epsilon \rceil - 1), \lceil \frac{3}{2} N_{n-1}^\epsilon \rceil - (\lceil N_j^\epsilon \rceil - 1)\} > \frac{1}{2} N_{n-1}^\epsilon$.

Case 2: There is at least one minor rebalancing step for α caused by an update of the form $\delta R = \{(\alpha, \beta') \mapsto m'\}$ since state $\mathcal{Z}_j$. Let $\mathcal{Z}_\ell$ denote the state created after the previous minor rebalancing for α caused by an update of this form; thus, $c_\ell^{R,\alpha} = \ldots = c_{n-2}^{R,\alpha} = 0$. The minor rebalancing steps creating $\mathcal{Z}_\ell$ and $\mathcal{Z}_n$ move tuples with the A-value a between the relation parts of R in opposite directions with respect to heavy and light. From state $\mathcal{Z}_\ell$ to state $\mathcal{Z}_n$, the number of such tuples either decreases from $\lceil \frac{3}{2} N_l^\epsilon \rceil$ to $\lceil \frac{1}{2} N_{n-1}^\epsilon \rceil - 1$ (heavy to light) or increases from $\lceil \frac{1}{2} N_l^\epsilon \rceil - 1$ to $\lceil \frac{3}{2} N_{n-1}^\epsilon \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than N_{n-1}^ϵ since $N_l = N_{n-1}$ and $\min\{\lceil \frac{3}{2} N_l^\epsilon \rceil - (\lceil \frac{1}{2} N_{n-1}^\epsilon \rceil - 1), \lceil \frac{3}{2} N_{n-1}^\epsilon \rceil - (\lceil \frac{1}{2} N_l^\epsilon \rceil - 1)\} > N_{n-1}^\epsilon$.

Let $k = j$ if Case 1 holds and $k = \ell$ if Case 2 holds. By the above analysis, there must be more than $\frac{1}{2} N_{n-1}^\epsilon$ updates between $\mathcal{Z}_k$ and $\mathcal{Z}_n$. Hence,

$$
\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{R,\alpha} &\geq \sum_{i=0}^{k-1} c_i^{R,\alpha} + \sum_{i=k}^{n-1} \hat{c}_i^{R,\alpha} && \textit{(by induction hypothesis)} \\
&= \sum_{i=0}^{k-1} c_i^{R,\alpha} + \sum_{i=k}^{n-1} 2\gamma N_{n-1}^{\max\{\epsilon,1-\epsilon\}} && (N_k = \ldots = N_{n-1}) \\
&> \sum_{i=0}^{k-1} c_i^{R,\alpha} + \frac{1}{2} N_{n-1}^{\epsilon} 2\gamma N_{n-1}^{\max\{\epsilon,1-\epsilon\}} && \textit{(more than } \tfrac{1}{2} N_{n-1}^{\epsilon} \textit{ updates)} \\
&\geq \sum_{i=0}^{k-1} c_i^{R,\alpha} + c_{n-1}^{R,\alpha} = \sum_{i=0}^{n-1} c_i^{R,\alpha} && (c_k^{R,\alpha} = \ldots = c_{n-2}^{R,\alpha} = 0).
\end{aligned}
$$

This implies that Inequality (4) holds for update sequences of length n.

The inductive analysis shows that Inequality (1) holds when the amortized cost of OnUpdate$(u_i, \mathcal{Z}_i)$ is

$$\hat{c}_i = \gamma N_i^{\max\{\epsilon,1-\epsilon\}} + 4\gamma N_i^{\frac{1}{2}} + 4\gamma N_i^{\max\{\epsilon,1-\epsilon\}} + \Gamma, \quad \text{for } 0 \leq i < n,$$

where Γ and γ are constants. The amortized cost $\hat{c}_i^{major}$ of major rebalancing is $4\gamma N_i^{\frac{1}{2}}$, and the amortized cost $\hat{c}_i^{minor}$ of minor rebalancing is $4\gamma N_i^{\max\{\epsilon,1-\epsilon\}}$. From the size invariant $\lfloor \frac{1}{4} N_i \rfloor \leq |\mathbf{D}_i| < N_i$ follows that $|\mathbf{D}_i| < N_i < 4(|\mathbf{D}_i| + 1)$ for $0 \leq i < n$, where $|\mathbf{D}_i|$ is the database size before update u_i. This implies that for any database $\mathbf{D}$, the amortized major rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$, the amortized minor rebalancing time is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$, and the overall amortized update time of IVM$^\epsilon$ is $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$. □

9 A Lower Bound on the Maintenance of Triangle Queries

In this section we prove Proposition 5, which states a lower bound on the trade-off between amortized update time and enumeration delay for the maintenance of triangle queries, conditioned on the OMv conjecture [19].

Proposition 5. *For any $\gamma > 0$ and database $\mathbf{D}$, there is no algorithm that incrementally maintains the result of any triangle query under single-tuple updates to $\mathbf{D}$ with arbitrary preprocessing time, $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ enumeration delay, unless the OMv conjecture fails.*

The proof relies on the Online Vector-Matrix-Vector Multiplication (OuMv) conjecture, which is implied by the OMv conjecture (Conjecture 2). First, we give the definition of the OuMv problem and state the corresponding conjecture.

Definition 34 (Online Vector-Matrix-Vector Multiplication (OuMv) [19])**.** *We are given an $n \times n$ Boolean matrix $\mathbf{M}$ and receive n pairs of Boolean column-vectors of size n, denoted by $(\mathbf{u}_1, \mathbf{v}_1), \ldots, (\mathbf{u}_n, \mathbf{v}_n)$; after seeing each pair $(\mathbf{u}_i, \mathbf{v}_i)$, we output the product $\mathbf{u}_i^T \mathbf{M} \mathbf{v}_i$ before we see the next pair.*

Conjecture 35 (OuMv Conjecture, Theorem 2.7 in [19])**.** *For any $\gamma > 0$, there is no algorithm that solves OuMv in time $\mathcal{O}(n^{3-\gamma})$.*

The following proof of Proposition 5 reduces the OuMv problem to the problem of incrementally maintaining a triangle query. This reduction implies that if there is an algorithm that incrementally maintains a triangle query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ enumeration delay for some $\gamma > 0$ and database $\mathbf{D}$, then the OuMv problem can be solved in subcubic time. This contradicts the OuMv conjecture and, consequently, the OMv conjecture.

Proof of Proposition 5. The proof is inspired by the lower bound proof for maintaining non-hierarchical Boolean conjunctive queries [6]. Let $\triangle$ be a triangle query of arbitrary arity. For the sake of contradiction,

```
SolveOuMv(matrix M, vectors u_1, v_1, ..., u_n, v_n)
1  let Z = initial IVM^ε state of the empty database
2  foreach (i, j) ∈ M do
3      δS = { (i, j) ↦ M(i, j) }
4      Z = OnUpdate(δS, Z)
5  foreach r = 1, ..., n do
6      foreach i = 1, ..., n do
7          δR = { (a, i) ↦ (u_r(i) − R(a, i)) }
8          Z = OnUpdate(δR, Z)
9          δT = { (i, a) ↦ (v_r(i) − T(i, a)) }
10         Z = OnUpdate(δT, Z)
11     output (△ ≠ ∅)
```

Figure 18: The procedure SolveOuMv solves the OuMv problem using an incremental algorithm that maintains a triangle query $\triangle$ of arbitrary arity under single-tuple updates. The state $\mathcal{Z}$ is the initial IVM$^\epsilon$ state of a database with empty relations R, S and T. The procedure OnUpdate is given in Figure 17 and maintains the triangle query under single-tuple updates.

assume that there is an incremental maintenance algorithm $\mathcal{A}$ that maintains $\triangle$ under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ enumeration delay, for some $\gamma > 0$. We show that this algorithm can be used to design an algorithm $\mathcal{B}$ that solves the OuMv problem in subcubic time, which contradicts the OuMv conjecture.

The reduction Figure 18 gives the pseudocode of the algorithm $\mathcal{B}$, which processes an OuMv input $(\mathbf{M}, (\mathbf{u}_1, \mathbf{v}_1), \ldots, (\mathbf{u}_n, \mathbf{v}_n))$. We denote the entry of $\mathbf{M}$ in row i and column j by $\mathbf{M}(i, j)$ and the i-th component of $\mathbf{v}$ by $\mathbf{v}(i)$. The algorithm first constructs the initial IVM$^\epsilon$ state $\mathcal{Z}$ from a database $\mathbf{D} = \{R, S, T\}$ with empty relations R, S, and T. Then, it executes at most n^2 updates to the relation S such that $S = \{ (i, j) \mapsto \mathbf{M}(i, j) \mid i, j \in [n] \}$. In each round $r \in [n]$, the algorithm executes at most $2n$ updates to the relations R and T such that $R = \{ (a, i) \mapsto \mathbf{u}_r(i) \mid i \in [n] \}$ and $T = \{ (i, a) \mapsto \mathbf{v}_r(i) \mid i \in [n] \}$, where a is some constant. By construction, $\mathbf{u}_r^{\mathrm{T}} \mathbf{M} \mathbf{v}_r = 1$ if and only if there exist $i, j \in [n]$ such that $\mathbf{u}_r(i) = 1$, $\mathbf{M}(i, j) = 1$, and $\mathbf{v}_r(j) = 1$, which is equivalent to $R(a, i) \cdot S(i, j) \cdot T(j, a) = 1$ at the end of round r. Thus, the algorithm outputs 1 at the end of round r if and only if the result of the triangle query is nonempty. Nonemptiness of the query result can be checked by triggering enumeration and checking whether at least one output tuple is reported.

Time analysis Constructing the initial state from a database with empty relations takes constant time. The construction of relation S from $\mathbf{M}$ requires at most n^2 updates. Given that the amortized time for each update is $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}-\gamma})$ and the database size $|\mathbf{D}|$ stays $\mathcal{O}(n^2)$, the overall time for constructing relation S is $\mathcal{O}(n^2 \cdot n^{2 \cdot (\frac{1}{2}-\gamma)}) = \mathcal{O}(n^{3-2\gamma})$. In each round, the algorithm performs at most $2n$ updates and needs $\mathcal{O}(|\mathbf{D}|^{1-\gamma})$ time to report the first result tuple or to signalize that the result is empty. Hence, the time to execute the updates in a single round is $\mathcal{O}(2n \cdot n^{2 \cdot (\frac{1}{2}-\gamma)}) = \mathcal{O}(n^{2-2\gamma})$. The time to report the first result tuple or signalize emptiness is $\mathcal{O}(n^{2 \cdot (1-\gamma)}) = \mathcal{O}(n^{2-2\gamma})$. Thus, the overall execution time is $\mathcal{O}(n^{2-2\gamma})$ per round and $\mathcal{O}(n^{3-2\gamma})$ for n rounds. Hence, algorithm $\mathcal{B}$ needs $\mathcal{O}(n^{3-2\gamma})$ time to solve the OuMv problem, which contradicts the OuMv conjecture and, consequently, the OMv conjecture.

□

10 Recovering Existing Dynamic and Static Approaches

We next show how IVM$^\epsilon$ recovers the classical first-order IVM [12] on triangle queries (Section 10.1) and the worst-case optimal time of non-incremental algorithms for computing the result of the ternary triangle query (Section 10.2).

10.1 Classical First-Order IVM

We start with a brief description of classical first-order IVM on the ternary triangle query $\triangle_3$. The other triangle queries are treated analogously. Classical first-order IVM materializes the query result. Given a single-tuple update $\delta R = \{(\alpha, \beta) \mapsto m\}$ to relation R, it maintains query $\triangle_3$ under the update by computing the delta query

$$\delta\triangle_3(\alpha, \beta, c) = \delta R(\alpha, \beta) \cdot S(\beta, c) \cdot T(c, \alpha)$$

and updating the query result by setting $\triangle_3(\alpha, \beta, c) := \triangle_3(\alpha, \beta, c) + \delta\triangle_3(\alpha, \beta, c)$ for each C-value c in $\delta\triangle_3$. The maintenance of the query requires the iteration over possibly linearly many C-values paired with β in relation S and with α in relation T. Hence, the update time is $\mathcal{O}(|\mathbf{D}|)$. The evaluation of updates to the relations S and T is analogous. The preprocessing phase uses a worst-case optimal join algorithm to compute the initial query result in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time [32]. Since the query result is materialized, the enumeration delay is constant. The space complexity is dominated by the size $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ of the query result [30].

IVM$^\epsilon$ becomes the classical first-order IVM algorithm by setting ϵ to 0 or 1.

We first consider the case $\epsilon = 1$ and explain it for the ternary triangle query; the other triangle queries are treated analogously. If $\epsilon = 1$, then all tuples are in the light parts of the relations and the results of all materialized views in Figure 10 become empty except for the skew-aware view

$$\triangle_3^{LLL}(a, b, c) = R^L(a, b) \cdot S^L(b, c) \cdot T^L(c, a),$$

whose result becomes exactly that of $\triangle_3$.

We next explain in more detail. The preprocessing stage sets the threshold base N of the initial IVM$^\epsilon$ state to $2 \cdot |\mathbf{D}| + 1$ and strictly partitions each relation with threshold $N^\epsilon = N$. Since for each relation $K \in \{R, S, T\}$, variable X in the schema of K, and value x in the domain of X, it holds $|\sigma_{X=x}K| < N$, all tuples in K end up in the light part of K. Consequently, all materialized views in Figure 10 besides $\triangle_3^{LLL}$ stay empty, since each of them refers to at least one heavy relation part. The only materialized view that is possibly non-empty is $\triangle_3^{LLL}$. This also means that the result of query $\triangle_3$ and $\triangle_3^{LLL}$ are equal. Given an update, the procedure OnUpdate in Figure 17 never performs minor rebalancing, since the degrees of data values can never reach $\frac{3}{2}N$, due to the size invariant $\lfloor \frac{1}{4}N \rfloor \leq |\mathbf{D}| < N$. The procedure MajorRebalancing, which might be invoked by OnUpdate, does not move tuples to the heavy relation parts, since the threshold for strict partitioning is always greater than the database size. This implies that the views in Figure 10 besides $\triangle_3^{LLL}$ stay empty after any update.

The case of $\epsilon = 0$ is symmetric and IVM$^\epsilon$ becomes the classical first-order IVM algorithm. In the preprocessing stage, the input relations are strictly partitioned with threshold $N^\epsilon = 1$, which means that all light relation parts and materialized views referring to these parts become empty. Only one skew-aware view is constructed and its result is equal to that of the triangle query under consideration. IVM$^\epsilon$ materializes this view and allows for constant-delay enumeration from it.

We next discuss in more detail the ternary triangle query. The result of the skew-aware view $\triangle_3^{HHH}(a, b, c) = R^H(a, b) \cdot S^H(b, c) \cdot T^H(c, a)$ is equal to the result of $\triangle_3$. The condition $\epsilon = 0$ in the third line of the procedure AffectedPart in Figure 17 avoids that any update affects the light relation parts. Since the degrees of data values in the heavy relation parts can never fall below $\frac{1}{2}N^\epsilon = \frac{1}{2}$, minor rebalancing is never invoked. Based on the threshold for strict relation partitioning, major rebalancing does not move tuples to the light relation parts.

10.2 Computing the Ternary Triangle Query in a Static Database

The worst-case optimal time to compute the result of the ternary triangle query over the database $\mathbf{D}$ is $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ [32]. IVM$^\epsilon$ recovers this computation time in the static case by using its update mechanism as follows. We fix $\epsilon = \frac{1}{2}$ and insert all tuples from $\mathbf{D}$, one at a time, into a database $\mathbf{D}'$ that is initially empty. For each insert, we call the procedure OnUpdate from Figure 17. The preprocessing time is constant. By Theorem 3, IVM$^\epsilon$ guarantees $\mathcal{O}(M^{\frac{1}{2}})$ amortized update time, where M is the size of $\mathbf{D}'$ at update time. Thus, the total time to insert all tuples into $\mathbf{D}'$ is

$$\mathcal{O}(\sum_{M=0}^{|\mathbf{D}|-1} M^{\frac{1}{2}}) = \mathcal{O}(|\mathbf{D}| \cdot |\mathbf{D}|^{\frac{1}{2}}) = \mathcal{O}(|\mathbf{D}|^{\frac{3}{2}}).$$

Finally, we enumerate the query result with constant delay. Since the number of tuples in the result is bounded by $|\mathbf{D}|^{\frac{3}{2}}$ [30], the overall enumeration takes $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time. Overall, we compute the result of the ternary triangle query in $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ time.

To avoid rebalancing while inserting the tuples into the empty database, we can preprocess the input relations in $\mathbf{D}$ to decide for each tuple its final relation part. For instance, if for an A-value a, it holds $|\sigma_{A=a}R| \geq |\mathbf{D}|^{\frac{1}{2}}$, the tuple is inserted to the heavy part of R, otherwise to the light part. Since we do not perform any rebalancing, the worst-case (and not only amortized) time of each insert is $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$.

11 Related Work

Triangle queries in the static setting The problems of finding, counting, and listing of given-length cycles in graphs have been extensively investigated since the 70s [21, 11, 40]. One important result that falls into the scope of this work is that, given a graph with n vertices and m edges, finding a triangle if one exists and counting all triangles can be done in time $\mathcal{O}(n^\omega)$ where $\omega < 2.373$ is the exponent of matrix multiplication [21]. The same problem can be solved in time $\mathcal{O}(m^{\frac{2\omega}{\omega+1}}) \leq \mathcal{O}(m^{1.41})$, which is better than the former time bound on sparse graphs [3]. The problem of computing for each edge the number of triangles using this edge can be solved in time $\mathcal{O}(m^{1.41})$ [16]. This problem corresponds to computing the result of the binary triangle query over the ring of integers. Given a number k, a flavor of the triangle listing problem asks for the listing of k triangles if the graph has at least k triangles and all triangles otherwise. This problem can be solved in time $\widetilde{\mathcal{O}}(n^{2.373} + n^{1.568}t^{0.478})$ on dense graphs and in time $\widetilde{\mathcal{O}}(m^{1.408} + m^{1.222}t^{0.186})$ on sparse graphs, where $\widetilde{\mathcal{O}}$ suppresses multiplicative factors of size $n^{o(1)}$ [8]. All time bounds mentioned above rely on algebraic fast matrix multiplication. IVM$^\epsilon$'s preprocessing phase relies on an algorithm like Leapfrog TrieJoin or Recursive-Join that does not use matrix multiplication and runs in time $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ [32] to compute the initial query result on a database $\mathbf{D}$. Further works approximate the triangle count in large graphs [37, 5, 27] and assess the practicability of triangle counting and listing algorithms in massive networks [13, 35].

Complexity gap between single-tuple and bulk updates Our main result states that for $\epsilon = \frac{1}{2}$, IVM$^\epsilon$ maintains the triangle count (unary triangle query) under single-tuple updates to a database $\mathbf{D}$ with $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ amortized update time and $\mathcal{O}(1)$ enumeration delay (Theorem 3), which is worst-case optimal under the OMv conjecture (Proposition 5). We also know that triangle counting on a graph with m edges can be solved in $\mathcal{O}(m^{1.41})$ time [3]. Corroborating these two results, we conclude that there is a gap in the worst-case complexity of counting triangles between the static and the dynamic case (or equivalently between bulk updates and single-tuple updates). If the tuples in $\mathbf{D}$ come as a stream of inserts and we do one insert at a time, the overall time to compute the triangle count on $\mathbf{D}$ is $\mathcal{O}(|\mathbf{D}| \cdot |\mathbf{D}|^{\frac{1}{2}}) = \mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$. This is worse than $\mathcal{O}(|\mathbf{D}|^{1.41})$, which is achieved by processing all tuples in $\mathbf{D}$ in bulk. For the ternary triangle query, however, IVM$^\epsilon$ recovers the worst-case optimal time to list all triangles in the static setting, cf. Section 10.2.

Dynamic set intersection A prior result [28] on the dynamic evaluation of a class of Boolean queries is closely related to the maintenance of the nullary triangle query. Assume that $\mathcal{F}$ is a family of sets that are subject to inserts and deletes and N is the overall size of these sets. Given two sets from $\mathcal{F}$, the emptiness query answers whether their intersection is empty. There is a dynamic algorithm that uses $\mathcal{O}(N)$ space, executes updates to the sets in $\mathcal{F}$ in $\mathcal{O}(N^{\frac{1}{2}})$ expected time, and answers emptiness queries in $\mathcal{O}(N^{\frac{1}{2}})$ expected time. The proof of this result reveals that the algorithm categorizes the sets in $\mathcal{F}$ into *small* and *large* sets using some threshold and maintains the intersection size for any two large sets in a lookup table. The emptiness query for two sets, where one of the sets is small, is answered by iterating over the elements in the small set and checking for each element its containment in the other set. For two large sets, the emptiness query is answered by using the intersection-size table. Although not stated in that work, the intersection-size table can be constructed in $\mathcal{O}(N^{\frac{3}{2}})$ expected time in the preprocessing phase. The algorithm can be adapted to allow for an unbounded number of sets in $\mathcal{F}$ and to return the intersection size for any two sets from $\mathcal{F}$. This prior work can be used to recover a restricted instance of IVM$^\epsilon$ for the nullary query. Given a database $\mathbf{D} = \{R(A,B), S(B,C), T(C,A)\}$, an A-value $a \in \pi_A R$, and a B-value $b \in \pi_B R$, we denote by $\mathcal{R}_a^B$ and $\mathcal{R}_b^A$ the set of B-values paired with a in R and respectively the set of A-values paired with b in R. The sets $\mathcal{S}_b^C$, $\mathcal{S}_c^B$, $\mathcal{T}_c^A$, and $\mathcal{T}_a^C$ are defined analogously. Let $\mathcal{F}$ consist of these sets for all data values in the database. Assuming that the current triangle count on $\mathbf{D}$ is materialized, we can obtain the new triangle count upon an insert of a tuple (a,b) to relation R as follows. If $\mathcal{R}_a^B$ is already contained in $\mathcal{F}$, we extend this set by b, otherwise we create a new set $\mathcal{R}_a^B = \{b\}$. The set $\mathcal{R}_b^A$ is updated or created analogously. Then, we ask for the intersection size $\mathcal{S}_b^C \cap \mathcal{T}_a^C$. The new triangle count is the sum of the previous count and the size of this intersection. Updating the sets in $\mathcal{F}$ and computing the intersection size require $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ expected time [28]. Deletes to R and updates to the other relations are handled analogously. Since the triangle count is materialized, it allows constant-time access. Hence, we obtain a maintenance strategy for the nullary triangle query with $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ expected preprocessing time, $\mathcal{O}(|\mathbf{D}|)$ space, $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ expected update time, and $\mathcal{O}(1)$ enumeration delay. While meeting the complexity bounds of Proposition 4 (for $\epsilon = \frac{1}{2}$), this alternative approach does not support tuple multiplicities or arbitrary rings beyond the ring of integers.

Fine-grained lower bounds Investigations on fine-grained complexity have led to important conjectures and hypotheses on finding and listing triangles in graphs that have served as conditional lower bounds for many other problems [34, 1]. The strong triangle conjecture states that in the word-RAM model with words of length $\mathcal{O}(\log n)$, there is no algorithm that decides whether a graph with n nodes and m edges contains a triangle in $\mathcal{O}(\min\{n^{\omega-\gamma}, m^{\frac{2\omega}{\omega+1}-\gamma}\})$ expected time for any $\gamma > 0$, where ω is the exponent of matrix multiplication. Moreover, there is no combinatorial algorithm that solves this problem in $\mathcal{O}(m^{\frac{3}{2}-\gamma})$ time, for any $\gamma > 0$. According to this conjecture, the best known algorithms for this problem, the combinatorial ones as well as those based on fast matrix multiplication, are optimal. The OMv conjecture (Conjecture 1) [19] was used to derive conditional lower bounds on the maintenance of conjunctive queries [6]. It states that for any $\gamma > 0$, there is no algorithm that solves the OMv problem (Definition 34) in $\mathcal{O}(n^{3-\gamma})$ time. The best known algorithm solving the OMv problem runs in $\mathcal{O}(\frac{n^3}{\log^2 n})$ time [38]. Let Q be a Boolean conjunctive query whose homomorphic core is not q-hierarchical [6]. Then, for any $\gamma > 0$ and database of domain size n, there is no algorithm that incrementally maintains the result of Q under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(n^{1-\gamma})$ update time, and $\mathcal{O}(n^{2-\gamma})$ answer time, unless the OMv conjecture fails [6]. Triangle queries are not q-hierarchical and their homomorphic cores are the queries themselves in case they do not have repeating relation symbols. Hence, the above lower bound holds for all triangle queries without repeating relation symbols. The proof of this lower bound is similar to that for the query $\varphi = \exists x \exists y (S(x) \wedge E(x,y) \wedge T(y))$, which is the simplest Boolean conjunctive query that is not q-hierarchical [6]. Our lower bound proof in Section 9 adapts the proof for φ to triangle queries, strengthens it to allow for amortized update time, and expresses complexities in terms of the database size.

Enumeration with skip pointers Skip pointers have been previously used for constant-delay enumeration of distinct elements in the union of a fixed number of sets [7]. Section 2.4.4 introduces this approach

using the abstraction of hop iterators. Our approach extends the original method [7] with second-level skip pointers and parameterizes it by a search function to enable tighter bounds on enumeration delay. We use iterators with skip pointers in the enumeration procedures for the binary and unary triangle queries.

Approximation schemes in the dynamic setting A distinct line of work investigates randomized approximation schemes with an arbitrary relative error for counting triangles in a graph given as a stream of edges [4, 22, 10, 31, 14]. Each edge in the data stream corresponds to a tuple insert, and tuple deletes are not considered. The emphasis of these approaches is on space efficiency, and they express the space utilization as a function of the number of nodes and edges in the input graph and of the number of triangles. The space utilization is generally sublinear but may become superlinear if, for instance, the number of edges is greater than the square root of the number of triangles. The update time is polylogarithmic in the number of nodes in the graph. There is also work estimating the number of triangles in graph streams with both edge inserts and deletes [9].

Dynamic descriptive complexity Further away from our line of work is the development of dynamic descriptive complexity, starting with the DynFO complexity class and the much-acclaimed result on FO expressibility of the maintenance for graph reachability under edge inserts and deletes, see a recent survey [36]. The k-clique query can be maintained under edge inserts by a quantifier-free update program of arity $k-1$ but not of arity $k-2$ [41].

12 Extensions

Relations over task-specific rings Different rings can be used as the domain of tuple multiplicities (or payloads). We used here the ring $(\mathbb{Z}, +, \cdot, 0, 1)$ of integers to support counting. Previous work shows how the data-intensive computation of many applications can be captured by application-specific rings, which define sum and product operations over data values [33]. The relational data ring supports payloads with listing and factorized representations of relations, and the degree-m matrix ring supports payloads that can be used for maintaining gradients of square loss functions for linear regression models [33].

IVM$^\epsilon$ variants IVM$^\epsilon$ can be used to maintain triangle queries with repeating relation symbols, the counting versions of any query built using three relations and the 4-path query [23] in worst-case optimal update time. The same conditional lower bound on the update time shown for the triangle count (nullary triangle query) applies for most of the mentioned queries, too. This leads to the striking realization that, while in the static setting the counting versions of the cyclic query computing triangles and the acyclic query computing paths of length 3 have different complexities and pose distinct computational challenges, they share the same complexity and can use a very similar approach in the dynamic setting.

Loomis Whitney queries The IVM$^\epsilon$ maintenance strategies also naturally extend from triangle to Loomis Whitney (LW) queries. LW queries generalize triangle queries from cliques of degree three to cliques of degree $n \geq 3$; they encode the Loomis Whitney inequality [30]. Let $A_1, \ldots, A_n$ be the query variables and $R_1, \ldots, R_n$ relations over schemas $\mathbf{X}_1, \ldots, \mathbf{X}_n$, where $\forall i \in [n] : \mathbf{X}_i = (A_{((i+j) \mod n)+1})_{-1 \leq j \leq n-3}$. That is, the schema of R_1 is $(A_1, \ldots, A_{n-1})$, whereas the schema of R_n is $(A_n, A_1, \ldots, A_{n-2})$. The n-ary LW query of degree n has the form

$$\Diamond_n(\mathbf{x}) = R_1(\mathbf{x}_1) \cdots R_n(\mathbf{x}_n),$$

where $\mathbf{x} = (a_j)_{j \in [n]}$ and for all $i \in [n]$, $\mathbf{x}_i = (a_{((i+j) \mod n)+1})_{-1 \leq j \leq n-3}$ is a value from the domain of the tuple $\mathbf{X}_i$ of variables. As for triangle queries, a LW query of degree n and arity $0 \leq k \leq n-1$ has the same body as for arity n but only keeps the first k values in the result. For instance, for $n = 4$ the binary LW query is

$$\Diamond_2(a_1, a_2) = \sum_{a_3, a_4} R_1(a_1, a_2, a_3) \cdot R_2(a_2, a_3, a_4) \cdot R_3(a_3, a_4, a_1) \cdot R_4(a_4, a_1, a_2).$$

In case $n = 3$, each LW query $\Diamond_k$ becomes the triangle query $\triangle_k$, for $0 \leq k \leq 3$.

IVM$^\epsilon$ achieves the following complexities for LW queries of degree n (stated without proof):

- The preprocessing and amortized update time are the same as for triangle queries: $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$ preprocessing time and $\mathcal{O}(|\mathbf{D}|^{\max\{\epsilon,1-\epsilon\}})$ amortized update time.
- In case all variables are free, the space complexity is the same as for the ternary triangle query, namely, $\mathcal{O}(|\mathbf{D}|^{\frac{3}{2}})$; otherwise, the space complexity is $\mathcal{O}(|\mathbf{D}|^{1+\min\{\epsilon,1-\epsilon\}})$.
- For the nullary and n-ary LW queries, the enumeration delay is constant; for k-ary LW queries where $0 < k < n$, the enumeration delay is $\mathcal{O}(|\mathbf{D}|^{\min\{1,(n-k)\cdot(1-\epsilon)\}})$. The delay hence improves with increasing arity. For $n = 3$, we get exactly the same enumeration delay as for the triangle queries.
- The lower bound on the update-delay trade-off for triangle queries stated in Proposition 5 carry over to LW queries. This means that at $\epsilon = \frac{1}{2}$, IVM$^\epsilon$ is strongly Pareto worst-case optimal for the nullary and n-ary LW queries and weakly Pareto worst-case optimal for all other LW queries.

The result of the n-ary LW query $\Diamond_n$ of degree n has size $\mathcal{O}(|\mathbf{D}|^{\frac{n}{n-1}})$ [30]. It can also be computed in the static setting in the same time, which is thus worst-case optimal [32]. IVM$^\epsilon$ cannot be used to recover the optimality in the static case, since it takes $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ amortized time per each single-tuple update and there are $|\mathbf{D}|$ tuples to insert. Since the combination of $\mathcal{O}(|\mathbf{D}|^{\frac{1}{2}})$ amortized time and $\mathcal{O}(1)$ delay is strongly Pareto worst-case optimal, it means that no dynamic algorithm can achieve a lower amortized single-tuple update time for the n-ary LW query. This shows the limitation of single-tuple updates. To achieve the overall $\mathcal{O}(|\mathbf{D}|^{\frac{n}{n-1}})$ time for $|\mathbf{D}|$ tuple inserts, one would need to process several inserts at the same time, that is, in bulk, such that the amortized time per insert should be $\mathcal{O}(|\mathbf{D}|^{\frac{1}{n-1}})$. A characterization of the difference between bulk updates and single-tuple updates remains an interesting open problem.

13 Conclusion and Future Work

This article introduces IVM$^\epsilon$, an incremental maintenance approach for triangle queries under updates that exhibits a trade-off between the update time on one hand and the space and enumeration delay on the other hand. IVM$^\epsilon$ captures classical first-order IVM as a special case that has suboptimal linear update time.

There are worst-case optimal algorithms for *join* queries in the *static* setting [32]. In contrast, IVM$^\epsilon$ is worst-case optimal for the nullary and ternary triangle join queries in the *dynamic* setting. The dynamic setting case poses challenges beyond the static setting. First, the optimality argument for static join algorithms follows from their runtime being linear(ithmic) in their output size; this argument does not apply to our nullary triangle query, since its output is a scalar and hence of constant size. Second, optimality in the dynamic setting requires a more fine-grained argument that exploits the skew in the data for different evaluation strategies, view materialization, and delta computation; in contrast, there are static worst-case optimal join algorithms that do not need to exploit skew, materialize views, nor delta computation.

We conclude with a discussion on possible directions for future work.

Worst-case optimal dynamic query evaluation This article opens up a line of work on worst-case optimal dynamic query evaluation algorithms. The goal is a complete characterization of the complexity of incremental maintenance for arbitrary functional aggregate queries [2]. We would first like to find a syntactical characterization of all queries that admit incremental maintenance in (amortized) sublinear time. Using known (first-order, fully recursive, or factorized) incremental maintenance techniques, cyclic and even acyclic joins require at least linear update time. Our intuition is that this characterization is given by a notion of diameter of the query hypergraph. This class strictly contains the q-hierarchical queries, which admit constant-time updates [6]. A first step towards this goal is a characterization of the update-delay trade-off for hierarchical queries with arbitrary free variables [25].

Space-delay trade-off IVM$^\epsilon$ does not admit any trade-off between the space complexity and the enumeration delay: for all queries, there is either no or positive correlation between the two measures (cf. Figure 1). Prior work investigates the trade-off between space and delay for the evaluation of conjunctive queries in the static setting [15]. An interesting future direction is to design a maintenance approach with focus on the space-delay trade-off.

Implementation of IVM$^\epsilon$ We would like to implement IVM$^\epsilon$ and benchmark against existing IVM systems. The implementation of IVM$^\epsilon$ may pose some challenges. For instance, maintaining the exact heavy-light partitions of relations is computationally expensive. One way to handle this problem is to loosen the partition thresholds so that relation partitions are rebalanced less frequently while accepting temporarily suboptimal maintenance strategies. A further challenge is the maintenance of the index structures of IVM$^\epsilon$. For each materialized view V with some schema $\mathbf{X}$ and sub-schema $\mathbf{Y} \subseteq \mathbf{X}$, IVM$^\epsilon$ assumes the existence of an index that allows to check containment of any tuple $\mathbf{y}$ over $\mathbf{Y}$ in $\pi_{\mathbf{Y}} V$ in constant time and to enumerate all tuples in V matching $\mathbf{y}$ with constant delay We need to address the trade-off between the cost of maintaining this indices and the cost of access times without them.

References

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

[2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016. DOI: 10.1145/2902251.2902280.

[3] N. Alon, R. Yuster, and U. Zwick. Finding and Counting Given Length Cycles. *Algorithmica*, 17(3):209–223, 1997. DOI: 10.1007/BF02523189.

[4] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. In *SODA*, pages 623–632, 2002.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient algorithms for large-scale local triangle counting. *TKDD*, 4(3):13:1–13:28, 2010. DOI: 10.1145/1839490.1839494.

[6] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017. DOI: 10.1145/3034786.3034789.

[7] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering UCQs Under Updates and in the Presence of Integrity Constraints. In *ICDT*, pages 8:1–8:19, 2018. DOI: 10.4230/LIPIcs.ICDT.2018.8.

[8] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing Triangles. In *ICALP*, pages 223–234, 2014. DOI: 10.1007/978-3-662-43948-7_19.

[9] L. Bulteau, V. Froese, K. Kutzkov, and R. Pagh. Triangle counting in dynamic graph streams. *Algorithmica*, 76(1):259–278, 2016. DOI: 10.1007/s00453-015-0036-4.

[10] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting Triangles in Data Streams. In *PODS*, pages 253–262, 2006. DOI: 10.1145/1142351.1142388.

[11] N. Chiba and T. Nishizeki. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985. DOI: 10.1137/0214017.

[12] R. Chirkova and J. Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012. DOI: 10.1561/1900000020.

[13] S. Chu and J. Cheng. Triangle Listing in Massive Networks. *TKDD*, 6(4):17:1–17:32, 2012. DOI: 10.1145/2382577.2382581.

[14] G. Cormode and H. Jowhari. A Second Look at Counting Triangles in Graph Streams (Corrected). *Theor. Comput. Sci.*, 683:22–30, 2017. DOI: 10.1016/j.tcs.2016.06.020.

[15] S. Deep and P. Koutris. Compressed representations of conjunctive query results. In *PODS*, pages 307–322, 2018. DOI: 10.1145/3196959.3196979.

[16] L. Duraj, K. Kleiner, A. Polak, and V. V. Williams. Equivalences between triangle and range query problems. In *SODA*, 2020. DOI: 10.1137/1.9781611975994.3.

[17] A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In *CSL*, pages 189–202, 2011. DOI: 10.4230/LIPIcs.CSL.2011.189.

[18] T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately Counting Triangles in Sublinear Time. In *FOCS*, pages 614–633, 2015. DOI: 10.1109/FOCS.2015.44.

[19] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015. DOI: 10.1145/2746539.2746609.

[20] M. Idris, M. Ugarte, and S. Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017. DOI: 10.1145/3035918.3064027.

[21] A. Itai and M. Rodeh. Finding a Minimum Circuit in a Graph. *SIAM J. Comput.*, 7(4):413–423, 1978. DOI: 10.1137/0207033.

[22] H. Jowhari and M. Ghodsi. New Streaming Algorithms for Counting Triangles in Graphs. In *COCOON*, pages 710–716, 2005. DOI: 10.1007/11533719_72.

[23] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. *CoRR*, abs/1804.02780, 2018.

[24] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. In *ICDT*, pages 4:1–4:18, 2019. DOI: 10.4230/LIPIcs.ICDT.2019.4.

[25] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. *CoRR*, abs/1907.01988, 2019. To appear in PODS 2020.

[26] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: Higher-Order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014. DOI: 10.1007/s00778-013-0348-4.

[27] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient Triangle Counting in Large Graphs via Degree-Based Vertex Partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012. DOI: 10.1080/15427951.2012.625260.

[28] T. Kopelowitz, S. Pettie, and E. Porat. Dynamic set intersection. In *WADS*, pages 470–481, 2015. DOI: 10.1007/978-3-319-21840-3_39.

[29] P. Koutris, S. Salihoglu, and D. Suciu. Algorithmic Aspects of Parallel Data Processing. *Found. & Trends DB*, 8(4):239–370, 2018. DOI: 10.1561/1900000055.

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

[31] A. McGregor, S. Vorotnikova, and H. T. Vu. Better Algorithms for Counting Triangles in Data Streams. In *PODS*, pages 401–411, 2016. DOI: 10.1145/2902251.2902283.

[32] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018. DOI: 10.1145/3180143.

[33] M. Nikolic and D. Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018. DOI: 10.1145/3183713.3183758.

[34] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610, 2010.

[35] T. Schank and D. Wagner. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *WEA*, pages 606–609, 2005. DOI: 10.1007/11427186_54.

[36] T. Schwentick and T. Zeume. Dynamic Complexity: Recent Updates. *SIGLOG News*, 3(2):30–52, 2016. DOI: 10.1145/2948896.2948899.

[37] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM*, pages 608–617, 2008. DOI: 10.1109/ICDM.2008.72.

[38] R. Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *SODA*, pages 995–1001, 2007.

[39] V. V. Williams. On Some Fine-Grained Questions in Algorithms and Complexity. In *ICM*, volume 3, pages 3431–3472, 2018. DOI: 10.1142/9789813272880_0188.

[40] R. Yuster and U. Zwick. Finding Even Cycles Even Faster. *SIAM J. Discrete Math.*, 10(2):209–222, 1997. DOI: 10.1137/S0895480194274133.

[41] T. Zeume. The Dynamic Descriptive Complexity of k-Clique. *Inf. Comput.*, 256:9–22, 2017. DOI: 10.1016/j.ic.2017.04.005.