Incremental Maintenance of Regression Models over Joins

Milos Nikolic and Dan Olteanu
Department of Computer Science, University of Oxford
{milos.nikolic,dan.olteanu}@cs.ox.ac.uk

ABSTRACT

This paper introduces a principled incremental view maintenance (IVM) mechanism for in-database computation described by rings. We exemplify our approach by introducing the covariance matrix ring that we use for learning linear regression models over arbitrary equi-join queries.

Our approach is a higher-order IVM algorithm that exploits the factorized structure of joins and aggregates to avoid redundant computation and improve performance. We implemented it in DBToaster, which uses program synthesis to generate high-performance maintenance code. We experimentally show that it can outperform first-order and fully recursive higher-order IVM as well as recomputation by orders of magnitude while using less memory.

1. INTRODUCTION

Many of today's applications require real-time analytics over dynamic datasets, from online retailers to sensor networks, retail planning and forecasting [25, 4, 6]. These applications typically have long-lived analysis queries that enable data analysts to promptly react to fast-changing events. Although collected datasets can be large in size, in most cases, datasets evolve through small changes. This observation motivates *incremental data analysis* whose aim is to achieve efficient maintenance of analytical models for such small-size updates [14, 23].

In modern data analysis, datasets are not just dynamic in nature but also large. There is an increasing need to analyze data acquired from many different sources, for instance, to enable prediction models with higher accuracy or at a more granular level [27]. Data practitioners build such models over a common relation representing a join of the collected data sources. However, this can increase the load by several orders of magnitude due to the blowup in the tabular representation of the join result. A more succinct, *factorized representation* exploits multi-valued dependencies in the join result to eliminate a high degree of redundancy in both computation and data representation [34].

Motivated by the need to run complex analytics over joins of data sources, both industry and academia have developed systems that offer a tighter integration of databases and machine learning [26, 36, 19, 35]. Several data systems also integrate with statistical packages like R to enable in-situ data processing using domain-specialized routines [42, 11].

In this paper, we study incremental computation of in-database analytics described by rings such as the covariance matrix ring that can be used for learning regression models. We exploit dependencies among variables of join queries to factorize the computation of ring operations over joins and contain the effects of data explosion in join results [34, 39].

We next highlight key observations behind our work by means of an example.

Running Example. Consider a database with three relations [39]: $\texttt{House}(z, s, p)$ records house prices p and living areas (in squared meters) s within locations given by zipcodes z; $\texttt{Tax}(s, t)$ relates city tax bands t with house living areas s; $\texttt{Shop}(z, h)$ list shops with zipcode z and opening hours h (our experiments consider an extended real dataset).

We first explain how to compute the factorized natural join of the three relations using the order on the query variables from Figure 1(left) [34]: We first intersect `Shop` and `House` on zipcode z, then for each qualifying zipcode we branch and compute the available opening hours h separately from areas s, tax bands t, and prices p. This branching exploits the conditional independence in the join result: given z, h is independent of s, t, and p. Similarly, given s, t is independent of p and of z. This factorization avoids the explosion in the flat representation of the join result by not explicitly pairing the opening hours h with house living areas s for each zipcode z, and each tax band with each price for each s. Figure 1(middle) gives a schematic presentation of the factorized join that uses Cartesian products and unions to encode the independence among variables and respectively the possible value assignments of variables.

We can use a slightly extended approach to compute aggregates over factorized joins [8]. For instance, to compute the number of tuples in the join result (`COUNT` or `SUM(1)`), we follow the above approach where each visited value is interpreted as 1, unions as summation, and Cartesian products as multiplication. The count can then be accumulated as we progress with the factorized join.

Furthermore, we can learn any ordinary least squares regression model over the factorized join using batch gradient descent [39]. For instance, the following model

$$f(x_z, x_h, x_s, x_t) = \theta_0 + \theta_z \cdot x_z + \theta_h \cdot x_h + \theta_s \cdot x_s + \theta_t \cdot x_t$$

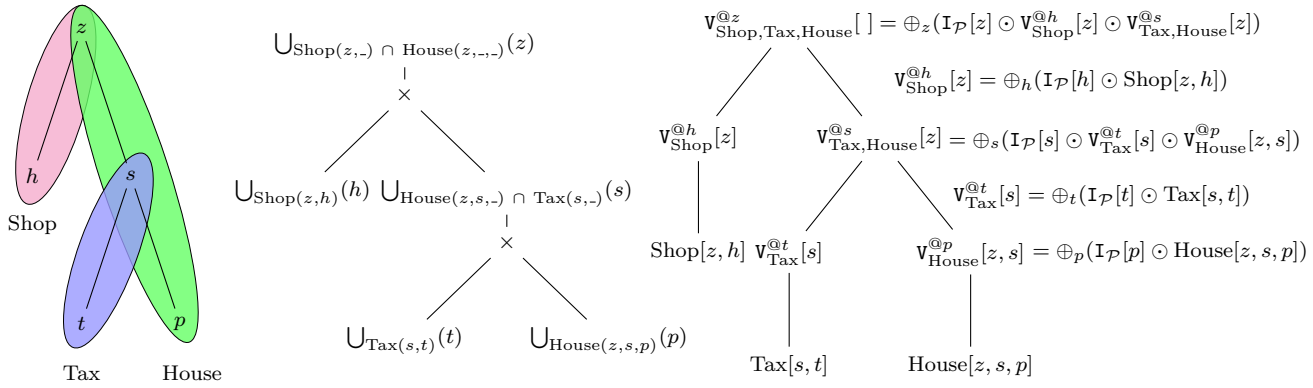


Figure 1: (left) Variable order ω of the natural join of the three relations House(z, s, p), Tax(s, t), Shop(z, h); (middle) Factorized join over ω; (right) View tree over ω and payload ring $\mathcal{P}$.

with parameters $\theta_0, \theta_z, \theta_h, \theta_s, \theta_t$ predicts the price p given the other variables as input features (for simplicity, we assume the features are continuous or already one-hot encoded; all values are numeric). The gradient vector of the square loss function with respect to the model parameters requires the computation of a covariance matrix that consists of sum aggregates of the form $\texttt{SUM}(1)$ and $\texttt{SUM}(a \cdot b)$ for every pair $(a, b) \in \{z, h, s, t, p\}^2$. We can share the computation across the aggregates and compute at each node in the factorized join a triple of aggregates $(C, \boldsymbol{S}, \boldsymbol{Q})$, where C is the count of tuples in the relation represented by the subtree rooted at that node, $\boldsymbol{S}$ is a vector with one linear sum of values per query variable within that subtree, and $\boldsymbol{Q}$ is a matrix of any product of two linear sums of values of query variables within that subtree.

We observe that all of the above computations follow the same structure, which is given by the variable order. Variable orders play the role of query plans here as they dictate the order of the joins in the query. Different variable orders may lead to different complexities in the above processing examples. We can capture all of the above computations using the abstraction of a so-called *view tree* that follows the structure of the variable order, cf. Figure 1(right). Each node n in this tree has a map $\mathtt{V}^{@n}_{rels}[vars]$ whose keys are tuples of values for variables $vars$ from relations $rels$ and payloads can be tuples of aggregates; we show in Section 3 how to formalize the payload operations using sum $\uplus$, product $\odot$, and $\oplus_z$ marginalization of variable z in specific rings. For instance, the top map is defined by the product of the payloads of the maps at its two children and a new map $\mathtt{I}_{\mathcal{P}}[z]$ for each z-value as map key, followed by a sum that marginalizes over z. The new map is specific to the ring $\mathcal{P}$ and lifts values that are assignments of variables in the key to elements of the ring.

To compute the `COUNT` aggregate over the factorized join, the payload at a node would be the number of tuples in the view's result, $\odot$ arithmetic multiplication, and $\uplus$ addition. To compute the covariance matrix, the payload is a triple of aggregates $(C, \boldsymbol{S}, \boldsymbol{Q})$ with slightly more involved definitions of the ring operations (Section 3).

Incremental view maintenance (IVM) of any computation within a ring $\mathcal{P}$ over factorized joins can now be phrased naturally within our framework. Whereas for non-incremental computation we only materialize the top view in the tree, for IVM we materialize all views. An update (tuple insert or delete) to a relation R triggers changes to all views at nodes from the leaf R to the root of the view tree: To such a view V we now add δV, which is computed using classical delta rules. For instance, an update δShop triggers the computation of the following delta views:

$$\delta\mathtt{V}^{@h}_{\text{Shop}}[z] = \oplus_h(\mathtt{I}_{\mathcal{P}}[h] \odot \delta\text{Shop}[z,h])$$
$$\delta\mathtt{V}^{@z}_{\text{Shop,Tax,House}}[\,] = \oplus_z(\mathtt{I}_{\mathcal{P}}[z] \odot \delta\mathtt{V}^{@h}_{\text{Shop}}[z] \odot \mathtt{V}^{@s}_{\text{Tax,House}}[z])$$

Our approach is an instance of higher-order delta-based IVM, since an update may trigger maintenance of several views. Higher-order IVM naturally follows from our factorized computation approach, which relies on variable orders.

The benefit of factorized IVM lies in efficiency. For our variable order, if each relation has size N, then there are databases for which the relational join result has size $\Theta(N^3)$ yet the factorized join can be computed in time and has size $\Theta(N)$. The time complexity remains $O(N)$ for computing sum aggregates over the factorized join and $O(m^2N)$ for computing the covariance matrix, where the additional factor $O(m^2)$ is due to the quadratically many aggregates in the number m of query variables.

Under updates to relations `Shop` and/or `House`, the time complexity for IVM becomes $O(1)$ for scalar aggregate payloads and $O(m^2)$ for the covariance matrix, so independent of the database size! Under updates to `Tax`, the time complexity for IVM stays linear, yet a finer analysis shows that the other views on the path to the root are only updated for those z-values that appear with the s-value in the update. Updates to `Tax` can be supported in $O(1)$ by a different variable order (where s and z are swapped and h and t remain children of z and s respectively), but then updates to `Shop` require linear-time support in worst case! An insight of this paper is a characterization of which relation updates can be supported in constant time within the same workload.

If we turn to the space complexity of our approach, we note that by decomposing the input join query into subqueries (the views), we essentially factorize its materialization and thereby reduce its size dramatically. The number of views to materialize depends linearly on the number of join variables and, at least for acyclic join queries, the size of each view is asymptotically upper bounded by the size of the factorized join result. For cyclic queries, such as the triangle query, we may materialize larger views to support constant-time updates to some relations.

The contributions of this paper are as follows:

- We introduce a higher-order incremental view maintenance (IVM) mechanism for aggregate computation over arbitrary equi-join queries. It relies on an order of the query variables to create a tree of interrelated, factorized views that decompose the query and aggregates.
- Our IVM mechanism can support various aggregate data rings. In particular, we introduce a ring that captures covariance matrices over factorized joins, which is used for learning linear regression models.
- Its complexity is captured by a new parameter called the dynamic factorization width, which draws on connections to widths for join queries.
- We implemented it in DBToaster, which uses recursive delta processing and program synthesis to generate high-performance maintenance code. The code represents an in-memory standalone stream processor.
- We benchmarked it against first-order and fully recursive higher-order IVM and recomputation. The experimental results show that our approach can outperform its competitors by orders of magnitude while using less memory.

2. PRELIMINARIES

Queries. We consider (equi-)join queries and families of SUM aggregates over them. Views are query definitions where we also allow to aggregate away variables in the query. The set of variables of a query or view Q is denoted by $\sigma(Q)$. The set of *free* variables of a query Q is denoted by $free(Q)$ and consists of all variables that appear in the schema of the relation representing the result of Q; for equi-join queries, this is the set of all variables in the query.

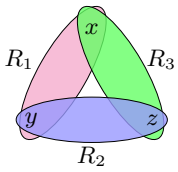
We let D be the domain of values for query variables; for simplicity, we assume all variables have the same domain.

Size bounds. For a join query Q, its hypergraph $H(Q)$ has one node per variable in Q and one hyperedge per relation in Q. Figures 1(left) depicts a query hypergraph.

An edge cover is a subset of (hyper)edges of $H(Q)$ such that each node appears in at least one edge. Edge cover can be formulated as an integer programming problem by assigning to each edge R_i a weight w_{R_i} that can be 1 if R_i is part of the cover and 0 otherwise. The size of an edge cover upper bounds the size of the query result, since the Cartesian product of the relations in the cover includes the query result: $|Q(\mathbf{D})| \leq |R_1|^{w_{R_1}} \cdot \ldots \cdot |R_n|^{w_{R_n}}$, where the database $\mathbf{D}$ is $(R_1, \ldots, R_n)$. By minimizing the size of the edge cover, we can obtain a lower upper bound on the size of the query result. This bound becomes tight for fractional weights [7]. Minimizing the sum of the weights thus becomes the objective of a linear program.

Definition 1 ([7]). *Given a join query Q over a database $(R_1, \ldots, R_n)$, the* fractional edge cover number $\rho^*(Q)$ *is the cost of an optimal solution to the linear program with variables $(w_{R_i})_{i\in[n]}$ representing weights of $(R_i)_{i\in[n]}$:*

$$\begin{aligned} &\textit{minimize} \prod_{i\in[n]} |R_i|^{w_{R_i}} \\ &\textit{subject to} \sum_{R \text{ is relation of } x} w_R \geq 1 \quad \textit{for each variable } x \\ &\forall i \in [n] : \omega_{R_i} \geq 0. \end{aligned}$$



$$\begin{aligned} &\textit{minimize } \textstyle\prod_{i\in[3]} |R_i|^{w_{R_i}} \\ &\textit{subject to} \\ &x : w_{R_1} \qquad + w_{R_3} \geq 1 \\ &y : w_{R_1} + w_{R_2} \qquad \geq 1 \\ &z : \qquad w_{R_2} + w_{R_3} \geq 1 \end{aligned}$$

Figure 2: (left) Hypergraph for the triangle query $Q_\triangleleft$; (right) Linear program for computing the tight size bound on the query result.

Figure 2 gives the hypergraph of the triangle query

$$Q_\triangleleft(x, y, z) = R_1(x, y), R_2(y, z), R_3(x, z) \tag{1}$$

and its linear program. By taking $w_{R_i} = 1/2$ and $|R_i| = N$ ($i \in [3]$), we get the optimal solution $\rho^*(Q_\triangleleft) = N^{3/2}$. Consequently, the result of the triangle query has $O(N^{3/2})$ tuples. This bound is tight in the sense that there exist classes of databases for which the result size is at least $\Omega(N^{3/2})$. For the acyclic query Q in Section 1, we set the weights 1 for each of the three relations and thus $\rho^*(Q) = N^3$ if all relations have size N.

Cardinality constraints can be used to lower the size bounds of query results. For instance, if the number of distinct x-values in $R_1(x, y)$ is $k \ll N$, then we can refine $Q_\triangleleft$ as $R_1(x, y), R_2(y, z), R_3(x, z), X(x)$ with the new size bound $\rho^*(Q_\triangleleft) = N \cdot k$, where $w_{R_2} = 1$ and $w_X = 1$.

Join selectivities can also be incorporated to obtain a size *estimate* (in contrast to an upper bound). For instance, assume the selectivity of the join on x between R_1 and R_3 is very low: $sel(x) = \frac{|R_1(x,y),R_3(x,z)|}{|R_1|\cdot|R_3|} = \frac{k}{N}$. Then, we consider a relation $R_4(x, y, z) = R_1(x, y), R_3(x, z)$ whose size estimate is $k \cdot N$ and use this as a cardinality constraint to obtain an estimate of $k \cdot N$ for $Q_\triangleleft$'s size since the size of the join of R_2 and R_4 cannot exceed the size of R_4.

Factorized computation. The result of a join query exhibits multi-valued dependencies that can be exploited for succinct, so-called factorized representation, and to reduce the time complexity to compute it and subsequent aggregates on it. This is captured by *variable orders*, which are akin to query plans in that they provide a relative order on the joins of the given query.

Definition 2 ([34]). *Given a join query Q and two variables x and y in Q, x* depends *on y if they occur in the same relation symbol in Q. A* variable order *ω for Q is a pair of a rooted forest with one node per variable in Q and a function* key *mapping each variable x to the subset of its ancestor variables in ω on which the variables in the subtree rooted at x depend. It satisfies the following constraints:*

- *The variables of each relation symbol in Q lie along the same root-to-leaf path in ω.*
- *For every variable y that is a child of a variable x, $key(y) \subseteq key(x) \cup \{x\}$.*

We let $\Omega(Q)$ denote the possible variable orders of Q.

Figure 1(left) gives a variable order for the natural join query from Section 1, where $key(h) = \{z\}$, $key(t) = \{s\}$, $key(p) = \{z, s\}$, $key(s) = \{z\}$, and $key(z) = \emptyset$. For the triangle query $Q_\triangleleft$, any permutation of the variables x, y, z is a path variable order. For the variable order $x - y - z$, we have $key(z) = \{x, y\}$, $key(y) = \{x\}$, and $key(x) = \emptyset$.

Given a database $\mathbf{D}$, the grounding of the variable order ω for query Q wrt $\mathbf{D}$ is a factorized representation of the query result $Q(\mathbf{D})$. This can be computed as exemplified in Figure 1(middle) for our running example.

If two variables x and y depend on each other, then the choice for a value for x may restrict the choice for a value for y. If they are not dependent, we can represent the values for x separately from those for y instead of explicitly representing their Cartesian product.

Similarly to $\rho^*(Q)$, the *factorization width* $fw(Q)$ governs the sizes of the factorized results of a join query Q [34]. In a factorized join over a variable order ω, the values of a variable x depend on the tuples of values of its $key(x)$ variables and are independent of the values for other variables. A tight bound on this number is then given by the size of a join query that covers the variables in $key(x) \cup \{x\}$. We denote this restriction of Q by $Q_{key(x)\cup\{x\}}$. An upper bound on the size of the factorization is then given by the maximum over all variables in ω of their number of values. This can be improved by going over all possible variable orders of Q and taking the minimum upper bound. This is the factorization width of the query.

Definition 3. *Given a join query Q, the* factorization width *of Q is $fw(Q) = \min_{\omega\in\Omega(Q)} \max_{v\in vars(Q)} \rho^*(Q_{key(x)\cup\{x\}})$.*

Example 1. For acyclic queries Q over relations $R_1, \ldots, R_n$, $fw(Q) = \max_{i\in[n]}(|R_i|)$ while $\rho^*(Q)$ can be as much as $\prod_{i\in[n]} |R_i|$ as in our running example. Here are examples of restrictions of our natural join Q in Section 1: $key(t)\cup\{t\} = \{s,t\}$ is covered by the query restriction $Q_{\{s,t\}}$ that is the relation Tax; $key(s) \cup \{s\} = \{s,z\}$ is covered by the query restriction $Q_{\{s,z\}}$ that is the relation House. For the triangle query $Q_\triangleleft$ and variable order $x{-}y{-}z$: $key(z)\cup\{z\} = \{x,y,z\}$ is covered by $Q_\triangleleft$, while $key(y) \cup \{y\} = \{x,y\}$ is covered by relation R. □

For any join query Q, its factorization width is the fractional hypertree width [34], a parameter that captures tractability for a host of computational problems [5].

Proposition 1. *Given a join query Q, for every database $\mathbf{D}$, the result $Q(\mathbf{D})$ admits*

- *a flat representation of size $O(\rho^*(Q))$ [7];*
- *a factorized representation of size $O(fw(Q))$ [34].*

There are classes of databases $\mathbf{D}$ for which the above size bounds are tight. The flat and factorized representations of $Q(\mathbf{D})$ can be computed worst-case optimally [29, 34].

A further result relevant here is that aggregates defined by arithmetic expressions over data values with operations summation and multiplication can be computed in one pass over factorized joins [8], as exemplified in Section 1.

3. FACTORIZED RING COMPUTATION

In this section, we introduce our framework for factorized computation over data rings. The main data structure used in our framework is a view that maps tuples to elements from a ring. The view is thus a map, where the keys are tuples from relations and the values, or the payload, carry useful computation. We accommodate the computation and the incremental maintenance of SUM aggregates or covariance matrices over joins in our framework by using hierarchies of interrelated views, also called view trees.

Definition 4. *Let $\mathcal{P}$ be a ring $(\mathcal{P}, +, *, \mathbf{0}, \mathbf{1})$. A* view $\mathsf{V}[key]$ *over $\mathcal{P}$ is a set of mappings from tuples of values for variables in key to elements in $\mathcal{P}$. Let $\mathsf{V}, \mathsf{V}_1, \mathsf{V}_2$ be views over the same ring $\mathcal{P}$. We define the following operations on views:*

- *join:* $\mathsf{V}[key] = \mathsf{V}_1[key_1] \odot \mathsf{V}_2[key_2]$
  Condition: $key = key_1 \cup key_2$.
  $\mathsf{V}[key] = v \leftarrow \mathsf{V}_1[key_1] = v_1, \mathsf{V}_2[key_2] = v_2, v = v_1 * v_2$.
- *union:* $\mathsf{V}[key] = \mathsf{V}_1[key_1] \uplus \mathsf{V}_2[key_2]$
  Condition: $key = key_1 = key_2$.
  $$\begin{aligned}\mathsf{V}[key] = v \leftarrow (&v = \mathsf{V}_1[key_1] + \mathsf{V}_2[key_2]);\\ &(v = \mathsf{V}_1[key_1],\ !\mathsf{V}_2[key_2] = v_2);\\ &(v = \mathsf{V}_2[key_2],\ !\mathsf{V}_1[key_1] = v_1).\end{aligned}$$
- *marginalization:* $\mathsf{V}[key] = \oplus_x \mathsf{V}_1[key_1]$
  Condition: $x \in key_1, key = key_1 - \{x\}$.
  $$\mathsf{V}[key] = \sum v_1 \leftarrow \mathsf{V}_1[key_1] = v_1.$$

For each ring $\mathcal{P}$, we define a special view $\mathtt{I}_\mathcal{P}[x]$ that lifts the domain values of any variable x to elements in the ring. These built-in views are necessary when marginalizing variables and casting the compute result as ring elements. Later in this section we define several rings and their lift view.

View Trees. We next define query plans for a given input join query Q. They mirror the tree structure of a variable order for Q and define at each node in this tree a view, which is a query over its children. The root of the tree corresponds to the entire query and the plan decomposes it into smaller queries in the spirit of (hyper)tree decompositions of join queries. We call such plans view trees.

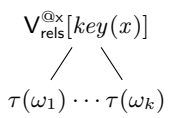
Figure 3 gives an algorithm that constructs a view tree $\tau(\omega)$ from a variable order ω of a given query Q. It is considered that all views are over the same ring $\mathcal{P}$. To construct complete query plans, the input variable order is extended by placing relation symbols at leaves under their lowest variable (as children or further below). The views are then defined on top of them. The base case is that of a relation symbol: We construct a view that is the relation itself. At an inner node x, we construct a view that is the marginalization over x of the natural join of the views at its children and of the lift view for x: we first join on x and then we aggregate it away. The schema of the view is given by the keys of x in the variable order ω. We use the notation $\mathsf{V}^{@x}_{rels}[key(x)]$ to state that the view V is (recursively) defined over relation symbols *rels* and corresponds to variable x in ω; for a view that is a relation symbol R, we may use the simplified notation $R[\sigma(R)]$ without *rels* and x, since they are just R.

Example 2. Figure 1 gives a variable order and a view tree for our example query. □

Our algorithm constructs one view per variable in the variable order ω. A wide relation (with many variables/columns) leads to long branches in ω with variables that are only local to this relation. This is, for instance, the case of our retailer dataset used in Section 6. Such long branches create long chains of views, with each view marginalizing a variable over the previous view in the chain. For practical reasons, we compose such long chains into a single view that marginalizes several variables at the same time.

τ (variable order ω)

switch ω:

R	$\mathsf{V}^{@\mathsf{R}}_{\mathsf{R}}[\sigma(\mathsf{R})]$
	 where

$\forall i \in [k]$: **let** $\mathsf{V}^{@\omega_i}_{\mathsf{rels_i}}[key(\omega_i)]$ be root of $\tau(\omega_i)$
$rels = \bigcup_{i\in[k]} rels_i$
$\mathsf{V}^{@x}_{\mathsf{rels}}[key(x)] = \oplus_x(\mathtt{I}_{\mathcal{P}}[x] \odot_{i\in[k]} \mathsf{V}^{@\omega_i}_{\mathsf{rels_i}}[key(\omega_i)])$

Figure 3: Algorithm for creating a view tree $\tau(\omega)$ from a variable order ω with views over ring $\mathcal{P}$.

Example 3. Consider an extension of our running example where House has two more variables, d_1 and d_2, all placed along a path under p in the variable order in Figure 1. Let $H[z,s,p,d_1,d_2] = \mathsf{V}^{@\text{House}}_{\text{House}}[z,s,p,d_1,d_2])$. The views for d_1 and d_2 are defined by:

$$\mathsf{V}^{@d_1}_{\text{House}}[z,s,p] = \oplus_{d_1}(\mathtt{I}_{\mathcal{P}}[d_1] \odot \mathsf{V}^{@d_2}_{\text{House}}[z,s,p,d_1])$$
$$\mathsf{V}^{@d_2}_{\text{House}}[z,s,p,d_1] = \oplus_{d_2}(\mathtt{I}_{\mathcal{P}}[d_2] \odot H[z,s,p,d_1,d_2]).$$

We can compose the two views into one equivalent view: $\mathsf{V}^{@d_1}_{\text{House}}[z,s,p] = \oplus_{d_1}(\mathtt{I}_{\mathcal{P}}[d_1] \odot \oplus_{d_2}(\mathtt{I}_{\mathcal{P}}[d_2] \odot H[z,s,p,d_1,d_2]))$.

Payload Rings. We next give examples of aggregate rings that can be used in our framework.

Count (#) ring. For `COUNT` aggregates over a view tree, the payload is from the ring of integer numbers $(\mathbb{Z}, +, *, \mathbf{0}, \mathbf{1})$ with the lift view $\mathtt{I}_{\#}[x] = v \leftarrow D(x), v = 1$, where D is a unary relation representing the value domain.

Sum (+) ring. For `SUM` aggregates over variables with values from $\mathbb{R}$, the payload is from the ring of real numbers $(\mathbb{R}, +, *, \mathbf{0}, \mathbf{1})$ with the lift view $\mathtt{I}_{+}[x] = v \leftarrow D(x), v = x$.

Average (μ) ring. Ring-based payloads can also represent compound types like tuples of aggregates. For `AVG` aggregates over variables with values from $\mathbb{R}$, the payload stores count and sum aggregates using elements from the ring $(\mathbb{Z} \times \mathbb{R}, +, *, \langle 0,0\rangle, \langle 1,0\rangle)$, where $+$ is vector-wise addition and $*$ is defined as $\langle c_1, s_1\rangle * \langle c_2, s_2\rangle = \langle c_1 * c_2, c_2 * s_1 + c_1 * s_2\rangle$. The lift view is $\mathtt{I}_{\mu}[x] = v \leftarrow D(x), v = \langle 1, x\rangle$.

Covariance matrix ($\square$) ring. This payload ring is used for learning linear regression models. Consider a training dataset that consists of n training examples with m features arranged into a design matrix $\boldsymbol{X}$ of size $(n \times m)$ and and output vector $\boldsymbol{y}$ of size $(n \times 1)$. The goal of linear regression is to learn the model parameters $\boldsymbol{\theta} = [\theta_1 \dots \theta_m]^{\mathrm{T}}$ best satisfying $\boldsymbol{X\theta} = \boldsymbol{y}$. The gradient vector of the square loss function with respect to the model parameters requires the computation of a covariance matrix $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ that quantifies the degree of correlation for each pair of features. Our goal is to compute $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ when $\boldsymbol{X}$ is the result of a join of database relations. Previous work [39] shows that computing a covariance matrix over a factorized join is possible using a triple of regression aggregates $(C, \boldsymbol{S}, \boldsymbol{Q})$ computed at each node in the factorized join, where C is the count of tuples in the relation represented by the subtree rooted at that node, $\boldsymbol{S}$ is a vector with one linear sum of values per query variable within that subtree, and $\boldsymbol{Q}$ is a matrix of any product of two linear sums of values of query variables within that subtree.

We introduce a ring that captures the covariance matrix computation using regression aggregates.

Definition 5. *Let $\mathcal{A}$ denote a set of triples $\{(C, \boldsymbol{S}, \boldsymbol{Q}) \in (\mathbb{Z}, \mathbb{R}^m, \mathbb{R}^{m\times m}), m \in \mathbb{N}\}$. Then, for $a = (C_a, \boldsymbol{S}_a, \boldsymbol{Q}_a) \in (\mathbb{Z}, \mathbb{R}^{m_1}, \mathbb{R}^{m_1\times m_1})$, $b = (C_b, \boldsymbol{S}_b, \boldsymbol{Q}_b) \in (\mathbb{Z}, \mathbb{R}^{m_2}, \mathbb{R}^{m_2\times m_2})$, $m = max(m_1, m_2)$, define $+^{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ as:*

$$a +^{\mathcal{A}} b = (C_a + C_b,\ \boldsymbol{S}_a^{pad} +^{\mathbb{R}^m} \boldsymbol{S}_b^{pad},\ \boldsymbol{Q}_a^{pad} +^{\mathbb{R}^{m\times m}} \boldsymbol{Q}_b^{pad})$$

where $\boldsymbol{S}_i^{pad}$ and $\boldsymbol{Q}_i^{pad}$, $i \in \{a,b\}$, are zero-padded up to the sizes $(m \times 1)$ and $(m \times m)$. Define $*^{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ as:

$$(C_a, \boldsymbol{S}_a, \boldsymbol{Q}_a) *^{\mathcal{A}} (C_b, \boldsymbol{S}_b, \boldsymbol{Q}_b) = (C, \boldsymbol{S}, \boldsymbol{Q}),\ \textit{where}$$

$$C = C_a \cdot C_b,\ \boldsymbol{S} = \begin{bmatrix} C_b \cdot \boldsymbol{S}_a \\ C_a \cdot \boldsymbol{S}_b \end{bmatrix},\ \boldsymbol{Q} = \begin{bmatrix} C_b \cdot \boldsymbol{Q}_a & \boldsymbol{S}_a \cdot \boldsymbol{S}_b^T \\ \boldsymbol{S}_b \cdot \boldsymbol{S}_a^T & C_a \cdot \boldsymbol{Q}_b \end{bmatrix}.$$

Let $[]_{0\times a}$ be an empty matrix of size $(0 \times a)$, define $\boldsymbol{0}$ as $(0, []_{0\times 1}, []_{0\times 0})$ and $\boldsymbol{1}$ as $(1, []_{0\times 1}, []_{0\times 0})$.

The algebra $(\mathcal{A}, +^{\mathcal{A}}, *^{\mathcal{A}}, \boldsymbol{0}, \boldsymbol{1})$ forms a ring called the covariance matrix ring denoted by $\square$.

The lift view is $\mathtt{I}_{\square}[x] = v \leftarrow D(x), v = \langle 1, x, x^2\rangle$.

To understand the intuition behind this ring, consider the covariance matrix computation over a join result expressed as a matrix $\boldsymbol{X}$. Then, C corresponds to the total number of tuples in $\boldsymbol{X}$, $\boldsymbol{S}$ contains the sum over each variable, and $\boldsymbol{Q}$ is the covariance matrix $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$. Consider now two matrices $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ for disjoint partitions of the join result, and their regression aggregates $(C_1, \boldsymbol{S_1}, \boldsymbol{Q_1})$ and $(C_2, \boldsymbol{S_2}, \boldsymbol{Q_2})$. Then, $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \end{bmatrix}$, the aggregates for $\boldsymbol{X}$ are:

$$\begin{aligned} C &= C_1 + C_2 \\ \boldsymbol{S} &= sum(\boldsymbol{X}) = sum(\boldsymbol{X}_1) + sum(\boldsymbol{X}_2) = \boldsymbol{S_1} + \boldsymbol{S_2} \\ \boldsymbol{Q} &= \boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{X}_1^{\mathrm{T}}\boldsymbol{X}_1 + \boldsymbol{X}_2^{\mathrm{T}}\boldsymbol{X}_2 = \boldsymbol{Q_1} + \boldsymbol{Q_2} \end{aligned}$$

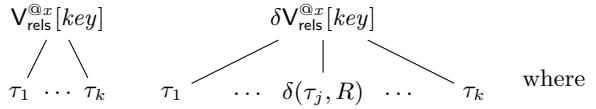
where *sum* returns a vector with the sum of each column. Now consider $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ for two vertical partitions whose product is the join result. Then, $\boldsymbol{X}$ has $\boldsymbol{X}_1$ duplicated C_2 times and $\boldsymbol{X}_2$ duplicated C_1 times, hence the rescaling of the sum and quadratic aggregates in the definition of $*^{\mathcal{A}}$. The product of $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ also forms new interactions between features from different datasets, captured via the product of their linear aggregates.

Example 4. Consider the computation of the covariance matrix on top of the join query from Example 1. At each node in the view tree, the view maps keys to a triple of aggregates; the leaf views map tuples to $\mathbf{1}$ from the covariance matrix ring. The view $\mathsf{V}^{@\mathsf{p}}[\mathsf{z},\mathsf{s}]$ maps each p-value to an aggregate $\langle 1, p, p^2\rangle$ and sums up those with the same (z,s)-value. For a fixed z-value, the view $\mathsf{V}^{@\mathsf{s}}[\mathsf{z}]$ first lifts each s-value into $\langle 1, s, s^2\rangle$ and then multiplies it with the aggregates computed over t and p, followed by marginalization of z. The regression aggregate in $\mathsf{V}^{@\mathsf{s}}[\mathsf{z}]$ for a fixed z-value has the covariance matrix of size (3×3) for the variables in $\{t, p, s\}$. Similarly, the root view multiplies these regression aggregates with those computed over h and z, followed by marginalization of z. The regression aggregate at the root consists of three parts: the constant aggregate that is the tuple count, linear aggregates that are sums over variables, and quadratic aggregates that are sums of products of each pair of variables. $\square$

δ (view tree τ, relation symbol R)

switch τ:

$\mathsf{V}^{@R}_{\mathsf{R}}[\sigma(R)]$ $\quad$ $\delta\mathsf{V}^{@R}_{\mathsf{R}}[\sigma(R)]$



$\forall i \in [k]$: **let** $\mathsf{V}^{@\tau_i}_{\mathsf{rels_i}}[key_i]$ be root of τ_i
let wlog j be such that $R \in rels_j$

$$\delta\mathsf{V}^{@x}_{\mathsf{rels}}[key] = \oplus_A(\mathtt{I}_{\mathcal{P}}[x] \odot \delta\mathsf{V}^{@\tau_j}_{\mathsf{rels_j}}[key_j] \odot_{i\in[k],i\neq j} \mathsf{V}^{@\tau_i}_{\mathsf{rels_i}}[key_i])$$

Figure 4: Algorithm for creating a delta view tree $\delta(\tau, R)$ for a given view tree τ over ring $\mathcal{P}$ to accommodate updates to a relation symbol R.

4. FACTORIZED HIGHER-ORDER IVM

In this section, we introduce incremental view maintenance in our factorized ring computation framework. In contrast to re-evaluation, incremental computation requires materialization and maintenance of views in the view tree. An update to a relation R triggers changes in all views from the leaf R to the root of the view tree.

Delta Views. For each view V affected by the update, a *delta view* δV defines the change in the view contents. If the view V is a relation symbol R, then $\delta\mathsf{V} = \delta R$ if there are updates to R and $\delta\mathsf{V} = \emptyset$ otherwise. If the view is defined using operations on other views, we derive $\delta\mathsf{V}$ using the following set of derivation rules:

$$\begin{aligned}
\delta(\mathsf{V}_1 \uplus \mathsf{V}_2) &= \delta\mathsf{V}_1 \uplus \delta\mathsf{V}_2 \\
\delta(\mathsf{V}_1 \odot \mathsf{V}_2) &= (\delta\mathsf{V}_1 \odot \mathsf{V}_2) \uplus (\mathsf{V}_1 \odot \delta\mathsf{V}_2) \uplus (\delta\mathsf{V}_1 \odot \delta\mathsf{V}_2) \\
\delta(\oplus_x(\mathsf{V})) &= \oplus_x(\delta\mathsf{V})
\end{aligned}$$

The correctness of the rules follows from the associativity of $\uplus$ and the distributivity of $\odot$ over $\uplus$, and $\oplus_x$ can be thought of as the repeated application of $\uplus$. The obtained deltas are subject to standard simplifications: If V_1 is not defined over the updated relation R, then its delta view $\delta\mathsf{V}_1$ is empty, and then we propagate this information using identities $\emptyset \uplus \mathsf{V}_2 = \mathsf{V}_2$ and $\emptyset \odot \mathsf{V}_2 = \emptyset$.

Delta Trees. Under updates to a relation symbol, a view tree becomes a delta tree, where the affected views become delta views. The algorithm in Figure 4 traverses the view tree τ top-down and replaces views with delta views on the path from the root to the updated relation symbol R.

Example 5. Consider the query in Example 1 and an update $\delta\mathsf{Tax}$. The update triggers delta computation at each view from the leaf Tax to the root of the view tree:

$$\begin{aligned}
\delta\mathtt{V}^{@t}_{\mathrm{Tax}}[s] &= \oplus_t(\mathtt{I}_{\mathcal{P}}[t] \odot \delta\mathrm{Tax}[s,t]) \\
\delta\mathtt{V}^{@s}_{\mathrm{Tax,House}}[z] &= \oplus_s(\mathtt{I}_{\mathcal{P}}[s] \odot \delta\mathtt{V}^{@t}_{\mathrm{Tax}}[s] \odot \mathtt{V}^{@p}_{\mathrm{House}}[z,s]) \\
\delta\mathtt{V}^{@z}_{\mathrm{Shop,Tax,House}}[\,] &= \oplus_z(\mathtt{I}_{\mathcal{P}}[z] \odot \mathtt{V}^{@h}_{\mathrm{Shop}}[z] \odot \delta\mathtt{V}^{@s}_{\mathrm{Tax,House}}[z])
\end{aligned}$$

We may also maintain each affected view:

$$\begin{aligned}
\mathrm{Tax}[s,t] &= \mathrm{Tax}[s,t] \uplus \delta\mathrm{Tax}[s,t] \\
\mathtt{V}^{@t}_{\mathrm{Tax}}[s] &= \mathtt{V}^{@t}_{\mathrm{Tax}}[s] \uplus \delta\mathtt{V}^{@t}_{\mathrm{Tax}}[s] \\
\mathtt{V}^{@s}_{\mathrm{Tax,House}}[z] &= \mathtt{V}^{@s}_{\mathrm{Tax,House}}[z] \uplus \delta\mathtt{V}^{@s}_{\mathrm{Tax,House}}[z] \\
\mathtt{V}^{@z}_{\mathrm{Shop,Tax,House}}[\,] &= \mathtt{V}^{@z}_{\mathrm{Shop,Tax,House}}[\,] \uplus \delta\mathtt{V}^{@z}_{\mathrm{Shop,Tax,House}}[\,]
\end{aligned}$$

An update to Tax binds s and t, so the computation of $\delta\mathtt{V}^{@t}_{\mathrm{Tax}}[s]$ is done in constant time. $\delta\mathtt{V}^{@s}_{\mathrm{Tax,House}}[z]$ requires to iterate over all possible z-values for a fixed s-value and has linear time maintenance cost. □

In case of a sequence of updates to distinct relation symbols, we obtain a chain of delta trees derived from the same input view tree to reflect the order of updates. Update sequences can also happen when inserting into a relation R that has several occurrences (i.e., relation symbols) in the query such as for self-joins. The relation symbols representing R are at different leaves, and we thus have changes along multiple leaf-to-root paths in the delta tree.

Our approach is a higher-order IVM as one update may trigger maintenance of several views. In contrast to the fully-recursive incremental view maintenance scheme [23], which also creates a hierarchy of views that support each other's maintenance, our approach relies on variable orders to decompose the query into views and factorize its computation and maintenance.

5. DYNAMIC FACTORIZATION WIDTH

As in the non-incremental case, different variable orders may lead to wildly different performance of our IVM approach. In this section, we settle the question of which variable orders can best support IVM under updates to a given set of relations and thereby pinpoint the complexity of maintaining query results under updates. This is captured by a novel notion called *dynamic factorization width*, which is a refinement of the factorization width recalled in Section 2.

We first recall the complexities in the non-incremental case. There, we only materialize the root view of a view tree over a variable order with the smallest factorization width, and we thus have the time data complexity $O(fw(Q))$ for computing factorized joins [34] and aggregates over them [8, 5]; for covariance matrices over factorized joins, there is an additional $O(m^2)$ factor, since the sizes of these matrices can be quadratic in the number m of variables (features) [39]. The space complexity is $O(1)$ or $O(m^2)$ to store the aggregate or covariance matrix in addition to the database (modulo logarithmic factors in the data size for data iterators).

We next discuss the IVM case.

Let Q be any join query. For any variable order $\omega \in \Omega(Q)$, let $\tau(\omega)$ be the view tree inferred from ω. This view tree has exactly one leaf for each relation symbol in Q.

We consider updates to relations whose relation symbols in Q form a set $\mathcal{U}$; a relation may have several relation symbols if it is involved in self-joins in Q, in which case all of them are in $\mathcal{U}$. For a relation symbol $R \in \mathcal{U}$, let $\Upsilon_{\tau(\omega)}(R)$ be the set of views that are ancestors of the leaf R in $\tau(\omega)$, i.e., it consists of all the views (recursively) defined using R.

The time needed to compute the delta for a view $\mathsf{V}^{\mathsf{x}}_{\mathsf{rels}}[\mathsf{key}]$ is upper bounded by that of a join query $Q^{\mathsf{rels}}_{key\cup\{x\}-\sigma(R)}$ over relations in rels that cover x and the variables in key but excluding the variables in R. The reason for the exclusion is that a single-tuple update to R binds the variables in R

to constants. The overall time to compute the deltas of all views in $\Upsilon_{\tau(\omega)}(R)$ is then

$$T(\omega, R) = \sum_{\mathsf{V}^{\mathsf{x}}_{\mathsf{rels}}[\mathsf{key}] \in \Upsilon_{\tau(\omega)}(R)} \rho^*(Q^{\mathsf{rels}}_{\mathsf{key} \cup \{\mathsf{x}\} - \sigma(R)}).$$

We are now ready to define the dynamic factorization width that captures the time complexity of incremental maintenance of Q under updates to relations in $\mathcal{U}$.

Definition 6. *Given a join query Q and a set of relation symbols $\mathcal{U}$ in Q. Then, the* dynamic factorization width *of Q and $\mathcal{U}$ is $dfw(Q,\mathcal{U}) = \min_{\omega \in \Omega(Q)} \max_{R \in \mathcal{U}} T(\omega, R)$.*

Theorem 1. *Given a query Q with m variables, database $\mathbf{D}$, a payload ring $\mathcal{P}$, and a set of relations $\mathcal{U}$ in $\mathbf{D}$. The time complexity of incrementally maintaining the result of Q over the ring $\mathcal{P}$ under single-tuple updates to relations in $\mathcal{U}$ is $O(dfw(Q,\mathcal{U}) \cdot T_{\mathcal{P}})$, where $T_{\mathcal{P}}$ is $O(1)$ for the sum ring and $O(m^2)$ for the covariance matrix ring.*

Example 6. For our query Q in Section 1 and database $\mathbf{D}$, the (static) factorization width is $fw(Q) = O(|\text{Tax}| + |\text{House}| + |\text{Shop}|)$. Under single-tuple updates to relations in a set $\mathcal{U}_1 \subseteq \{\text{Shop}, \text{House}\}$, the dynamic factorization width is $dfw(Q,\mathcal{U}_1) = 1$ since there are no free variables of the views over Shop or House in the variable order in Figure 1. This means that we can maintain the result of a sum aggregate over Q in $O(1)$ time under $\mathcal{U}_1$ updates. The same holds for $\mathcal{U}_2 \subseteq \{\text{Tax}, \text{House}\}$, i.e., $dfw(Q,\mathcal{U}_2) = 1$, as supported by the variable order $s\{t; z\{h; p\}\}$. However, $dfw(Q,\mathcal{U}_3) = O(|\mathbf{D}|)$ for $\mathcal{U}_3 = \{\text{Tax}, \text{House}, \text{Shop}\}$ since there is no variable order without free variables above all three relations and some variable orders have one free variable above at least one of the three relations. Under the variable order in Figure 1, $dfw(Q,\mathcal{U}_3) = \min(|\text{Shop}|, |\text{House}|)$.

The triangle query $Q_\triangleleft$ in Equation (1) has the (static) factorization width $fw(Q_\triangleleft) = \sqrt{|R_1| \cdot |R_2| \cdot |R_3|}$. For any relation R_i, $i \in [3]$, $dfw(Q, \{R_i\}) = 1$ as supported by a path variable order that has the variables in R_i as prefix. We can thus maintain an aggregate over the triangle query in $O(1)$ under single-tuple updates to exactly one of its three relations. For updates to at least two relations $\mathcal{U}_4$, $dfw(Q,\mathcal{U}_4) = O(|\mathbf{D}|)$. For instance, assume a variable order $x - y - z$. We need to cover: no variable under updates to R_1; one of the variables x or y under updates to R_2 or R_3 respectively (the case for other permutations of this variable order is analog). Maintenance has thus lower time cost than recomputation. □

We next analyze the space complexity $S(Q)$ of our approach. This is the sum of the sizes of the views in a view tree. The space needed by the keys of a view $\mathsf{V}^{\mathsf{x}}_{\mathsf{rels}}[\mathsf{key}]$ is given by the fractional edge cover of a join query built using relation symbols rels to cover the variables in key. To obtain the minimum size, we go over all variable orders of Q:

$$S(Q) = \min_{\omega \in \Omega(Q)} \sum_{\mathsf{V}^{\mathsf{x}}_{\mathsf{rels}}[\mathsf{key}] \in \tau(\omega)} \rho^*(Q^{\mathsf{rels}}_{\mathsf{key}}).$$

Theorem 2. *Given a query Q with m variables, database $\mathbf{D}$, a payload ring $\mathcal{P}$. The space complexity required by the materialization of a view tree for Q over the ring $\mathcal{P}$ is $O(S(Q) \cdot T_{\mathcal{P}})$, where $T_{\mathcal{P}}$ is $O(1)$ for the sum ring and $O(m^2)$ for the covariance matrix ring.*

There are three differences between the formula $S(Q)$ and Definition 3 of the factorization width $fw(Q)$: (1) the use of summation vs. maximum, though the gap between them is linear in m and thus independent of the database size; (2) the cover for $S(Q)$ can only use relation symbols of the view; (3) for $S(Q)$, we only need to cover *key* and not also the variable at the view as in the case of $fw(Q)$. The interplay of (2) and (3) can in fact make $S(Q)$ larger than $fw(Q)$. For acyclic queries, both complexities are linear if all relations have the same size and $S(Q)$ can be smaller than $fw(Q)$ in case some relations are asymptotically smaller than others. For cyclic queries, however, $S(Q)$ can be larger than $fw(Q)$. We show this for the triangle query in Equation (1) and relations of the same size N. Under any variable order, there is a view of size $O(N^2)$, whereas $fw(Q_\triangleleft) = N^{3/2}$. For instance, for the variable order $x - y - z$ we materialize the view $\mathsf{V}^{@z}_{\mathsf{R}_2,\mathsf{R}_3}[\mathsf{x},\mathsf{y}] = \oplus_z(\mathsf{I}_{\mathcal{P}}[\mathsf{z}] \odot \mathsf{R}_2[\mathsf{y},\mathsf{z}] \odot \mathsf{R}_3[\mathsf{x},\mathsf{z}])$, which may create $O(N^2)$ pairs (x, y) as we need both R_2 and R_3 to cover the variables x and y. To avoid the large intermediate result, we join all three relations at the same time [29], so as to cover (x, y) using R_1. That would, however, require recomputation of this 3-way join for each update. This takes $O(N)$ time since only two of the three variables are bound to constants. In contrast, our IVM approach trades off space for time: We need $O(N^2)$ space but then support $O(1)$ updates to one of the three relations (Example 6).

6. EXPERIMENTS

We benchmark our approach against DBToaster, a state-of-the-art IVM system for queries with joins and aggregates [23]. Our results can be summarized as follows:

- For maintaining a sum aggregate on top of a join query, our approach outperforms first-order and recursive higher-order IVM by pushing partial aggregates past joins and exploiting the factorization structure in the join result.
- Maintaining more complex aggregates in the form of covariance matrices widens the performance gap, resulting in several orders of magnitude better performance for our approach over the competitors.
- The performance of maintaining a covariance matrix depends on the update size. In our experiments, the throughput peaks for updates with $1,000$ - $10,000$ tuples.
- Compared with the competitors, our approach uses less memory for view maintenance over acyclic joins and yet delivers faster view maintenance. For cyclic queries, we analyze the trade-off between using more space and having faster maintenance.

Benchmarked systems. We compare our algorithm against first-order IVM and fully recursive higher-order IVM. The latter two techniques are supported by DBToaster, a system that uses recursive delta processing and program synthesis to generate high-performance maintenance code. DBToaster compiles a given SQL query into a set of triggers that keep the query result up to date for updates to base relations. The generated code represents an in-memory stream processor that is standalone and independent of any database system. The IVM performance of DBToaster on decision support and financial workloads is several orders of magnitude better than state-of-the-art commercial databases and stream processing systems [23].

	DF	DBT	IVM	DF-RE	DBT-RE
Retailer	2.84	1.18	2.31	21.30	17.54
Housing	20.67	20.59	2.42	37.58	0.17

Figure 5: The average throughput (M tuples/sec) of re-evaluation and incremental maintenance of a sum aggregate under updates of size $1,000$ to all base relations of the *Retailer* and *Housing* datasets.

We implement our factorization-based IVM algorithm as a relational program that processes a set of views materialized for a given variable order. We use the intermediate language of DBToaster to encode such a program, which then serves as input to the code generation phase performed by the backend of DBToaster. In our experiments, all the benchmarked approaches use the same code generator and runtime environment and materialize views as multi-indexed maps with memory-pooled records. The algorithms and record types used in these approaches, however, can greatly differ.

We use the following notations:

- **DF**, short for Dynamic Factorization (dynamic **F** [39]), is our factorized incremental maintenance approach.
- **DBT** and **IVM** are the fully recursive higher-order IVM and non-recursive IVM implemented in DBToaster.
- **SQL OPT** is an optimized SQL encoding of covariance matrix computation that improves on the SQL encoding reported in prior work [39]. The latter uses wide relations with one column per regression aggregate; recall there are quadratically many such aggregates in the number of query variables. The former *pivots* these aggregates into a single aggregate column indexed by the degree of each query variable. Both encodings take as input a variable order and construct one SQL query that intertwines join and aggregate computation by pushing (partial) regression aggregates (counts, sums, and cofactors) past joins [33].

In addition to these incremental approaches, we also benchmark two re-evaluation strategies: **DF-RE** represents reevaluation using factorization structures and payload rings (our implementation of **F** [39]), and **DBT-RE** denotes query reevaluation in DBToaster.

Workload. We run experiments over three datasets:

- *Retailer* is a real-world dataset from our industrial collaborator LogicBlox used by a retailer for business decision support and forecasting user demands. The dataset has a snowflake schema with one fact relation `Inventory` with 84M records, storing information about the inventory units for products in a location, at a given date. The `Inventory` relation joins along three dimension hierarchies: `Item` (on product id), `Weather` (on location and date), and `Location` (on location) with its lookup relation `Census` (on zip code). The training dataset is the acyclic natural join of these five relations and has 43 features. The regression task is to predict the inventory units based on all the features in the dataset. We consider a view tree in which the variables of each relation form a distinct root-to-leaf path, and the partial order on join variables is location{ date { product id }, zip }.
- *Housing* is a synthetic dataset modeling a house price market [39]. It consists of six relations: `House`, `Shop`, `Institution`, `Restaurant`, `Demographics`, and `Transport`, arranged into a star schema and with 1.4M tuples in total (scale factor 20). The training dataset is the acyclic natural join of all relations on the common attribute (postcode) and has 27 features. The regression task is to predict the housing price based on all the features in the dataset. We consider an optimal view tree that has each root-to-leaf path consisting of query variables for one relation.
- *Higgs Twitter* dataset represents friends/followers social relationships among users who were active on Twitter during the discovery of Higgs boson [2]. We split the first 3M records from the dataset into three binary relations of equal size, and use these relations to compute a covariance matrix on top of a triangle query. We consider a view tree that is a path of the three triangle variables.

We benchmark the performance of maintaining a sum aggregate and a covariance matrix for learning regression models over a natural join. We compute the covariance matrix over all variables of the join query (i.e., over all attributes of the input database), which suffices to learn linear regression models over *any label and set of features* that is a subset of the set of variables [32]. This is achieved by specializing the convergence step to the relevant restriction of the covariance matrix. In end-to-end learning of regression models over factorized joins, the convergence step takes negligible time compared to the data-dependent covariance matrix computation, which takes orders of magnitude more time [39].

We run the systems over data streams synthesized from the above databases by interleaving insertions to the base relations in a round-robin fashion. We group insertions into batches of different sizes and place no restriction on the order of records in input relations. All systems use payloads defined over rings with additive inverse – the sum and covariance matrix rings in **DF** and the sum ring in **DBT**– thus processing deletions is similar to that of insertions.

Experimental setup. We run all experiments on an EC2 m4.4xlarge instance (Intel(R) Xeon(R) CPU E5-2686 v4 2.30GHz, 64GB RAM, 500GB SSD) with Ubuntu Server 14.04. We use DBToaster v2.2 for the IVM competitors and code generation in our approach. The C++ generated code is single-threaded and compiled using g++ 6.3.0 and the -O3 optimization flag. We report wall-clock times by averaging three best results out of four runs of each query. We run experiments with a one-hour timeout on query execution, not counting loading of streams into memory.

Maintenance of sum aggregates. We start by analyzing different maintenance strategies on the task of maintaining a sum aggregate over one variable on top of a natural join. We measure the average throughput of re-evaluation and batched incremental maintenance under updates of size $1,000$ to all the relations of the *Retailer* and *Housing* datasets. For the former dataset, we aggregate the inventory units for products in `Inventory`; for the latter, we sum over the common join variable. Table 5 summarizes the results.

DF achieves the highest average throughput in the incremental maintenance of the sum aggregate. In the *Retailer* schema, the maintenance cost is dominated by the update on `Inventory`. **DBT**'s recursive delta compilation materializes 13 views representing connected sub-queries: five group-by aggregates over the base relations, `Inv`, `It`, `W`, `L`, and `C`; one group-by aggregate joining `L` and `C`; six views joining `Inv` with subsets of the others, namely {`It`}, {`It`, `W`}, {`It`, `W`, `L`}, {`W`}, {`W`, `L`}, and {`W`, `L`, `C`}; and the final aggregate. The two

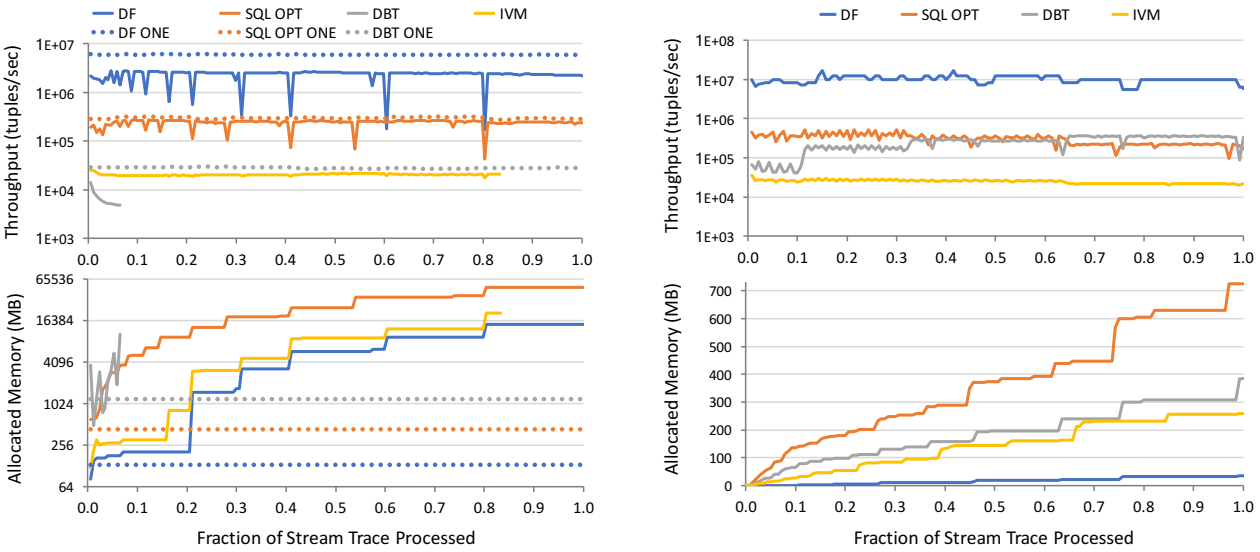


Figure 6: The performance of incremental maintenance of the covariance matrix over the *Retailer* (left) and *Housing* (right) datasets under updates of size $1,000$ to all base relations. The ONE plots consider updates to the largest Inventory relation only. For the *Retailer* dataset, DBT crashes after processing 6.5% of the input stream and IVM exceeds the one-hour time limit.

views joining `Inv` with { `W`, `L` } and { `It`, `W`, `L` } have linear maintenance for a single-tuple change in `Inventory`. **IVM** stores only the base relations with no aggregates on top of them. Its strategy of recomputing a delta from scratch on each update incurs modest overheads due to relatively small dimension relations. **DF** exploits the given variable order to materialize 8 views, four of them over `Inventory` (the base `Inv`, views over `Inv` and `It`, and over `Inv`, `It`, and `W`, and the final sum) but with constant maintenance for single-tuple updates to this relation. In contrast to **IVM**, our approach stores pre-computed relations in which all non-join variables are aggregated away. In the *Housing* schema, both **DF** and **DBT** benefit from this pre-aggregation, and since the query is a star join, both strategies materialize the same views.

For re-evaluation, **DF-RE** is capable of pushing the aggregate past joins towards the base relations, in contrast to **DBT-RE**. In the *Retailer* schema, this pre-aggregation has limited effects due to small dimension relations; in the *Housing* schema, however, joining the base relations first blows up the size of the intermediate result, making **DBT-RE** perform two orders of magnitude worse than **DF-RE**. Also, **DF-RE** achieves higher throughput than **DF** by avoiding the need to materialize (and maintain) intermediate views.

Maintenance of covariance matrices used in learning regression models. In addition to three incremental strategies from before, we now also benchmark **SQL OPT**. We consider two datasets, *Retailer* and *Housing*, and updates to all their relations.

The systems materialize greatly different numbers of views for incremental maintenance. In the *Retailer* schema, **DF** and **SQL OPT** rely on the given variable order in which the variables of `Inventory`, `Location`, and `Weather` form three separate root-to-leaf paths, and the other relations are as close to the root as possible. These two strategies materialize 9 views each: five views over the base relations, one view joining the views over `Inventory` and `Item` followed by a join with the view over `Weather`, one view joining the views over `Location` and `Census`, and the top-level view. In contrast, fully recursive higher-order IVM and first-order IVM in DBToaster fail to effectively share the computation of the regression aggregates, materializing linearly many views in the size of the covariance matrix: **DBT** and **IVM** use $3,425$ and respectively 951 views to maintain 946 covariance aggregates. In the *Housing* schema, where all relations join on a single variable, **DF** and **SQL OPT** materialize one view per base relations and the final result, so 7 in total, while **DBT** and **IVM** use 512 and 384 views to maintain 378 covariance aggregates.

Figure 6 shows the throughput of these techniques as the stream progresses. The *Retailer* stream consists of insertions into `Inventory` mostly, and since this relation lies along a root-to-leaf path in the variable order, processing a single-tuple update takes $O(1)$ time under **DF** and **SQL OPT**. The former outperforms the latter due to efficient encoding of triples of aggregates $(C, \boldsymbol{S}, \boldsymbol{Q})$ as payloads containing vectors and matrices. The recursive delta derivation in **DBT** creates a set of supporting views such that updating each covariance aggregate takes constant time for single-tuple changes in each relation. Although refreshing the result is cheap, maintaining these auxiliary views becomes a linear time operation. Maintaining a large number of views is a resource-intensive task, which causes **DBT** to crash after processing 6.5% of the input stream. For similar reasons, **IVM** was unable to process the entire stream within a one-hour time limit.

The join query in *Housing* is a star join with all relations joining on the common variable, which is the root in our variable order. Thus, **DF** and **SQL OPT** can process a single tuple in $O(1)$ time. **DBT** exploits the conditional independence in the derived deltas to materialize each base relation separately such that all non-join variables are aggregated away. Although each materialized view has $O(1)$ maintenance cost per update tuple, the large number of such views in **DBT** is the main reason for its poor performance.

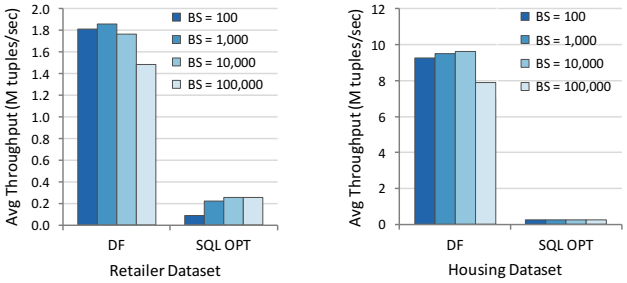


Figure 7: The performance of maintaining a covariance matrix under updates of different sizes.

In contrast, **IVM** stores entire tuples of the base relations including non-join variables. On each update to the database, **IVM** recomputes an aggregate on top of the join of these base relations and the update. Since an update tuple binds the value of the common join attribute, the delta query consists of disconnected components. DBToaster optimizes such a delta query by placing an aggregate around each component, which means that re-computing a delta now involves on-the-fly pre-aggregation of each relation followed by a join. Thus, **IVM** in *Housing* takes linear time, which explains its poor performance.

Memory consumption. Figure 6 shows that **DF** achieves the lowest memory utilization on both datasets while providing orders of magnitude better performance than its competitors! The reason behind the memory efficiency of our approach is twofold. First, it uses complex aggregates and factorization structures to express the covariance matrix computation over a much smaller set of views compared to **DF** and **IVM**. Second, it encodes regression aggregates implicitly using vectors and matrices rather than explicitly using variable degrees, like in **SQL OPT**. The occasional throughput hiccups in the plot are due to expansion of the underlying data structures used for storing views.

The effect of batch size on IVM. In this experiment, we evaluate the performance of maintaining a covariance matrix for batch updates of different sizes. Figure 7 shows the throughput of batched incremental processing for batch sizes varying from 100 to 100,000 on the *Retailer* and *Housing* datasets for updates to all relations. To show the desired behavior, we use a linear scale on the y-axis and omit **DBT** and **IVM** from the plot due to their poor performance.

We observe that using very large or small batch sizes can have negative performance effects: Iterating over large batches invalidates previously cached data resulting in future cache misses, whereas using small batches cannot offset the overhead associated with processing each batch. Using batch sizes of $1,000-10,000$ delivers best performance in our experiments. Previous work [30] presented similar findings in batched delta processing of sum and count aggregates in decision support workloads. This experiment confirms that the same also holds for the covariance matrix computation.

The effect of update workload on IVM. Our next experiment studies the effect of different update workloads on the performance of incremental processing. We consider the *Retailer* dataset and two possible update scenarios: the first one assumes all relations can change, which requires materialization of every view in the view tree; the second scenario assumes that only `Inventory` can change, while all others are static (denoted as ONE in Figure 6). The latter allows

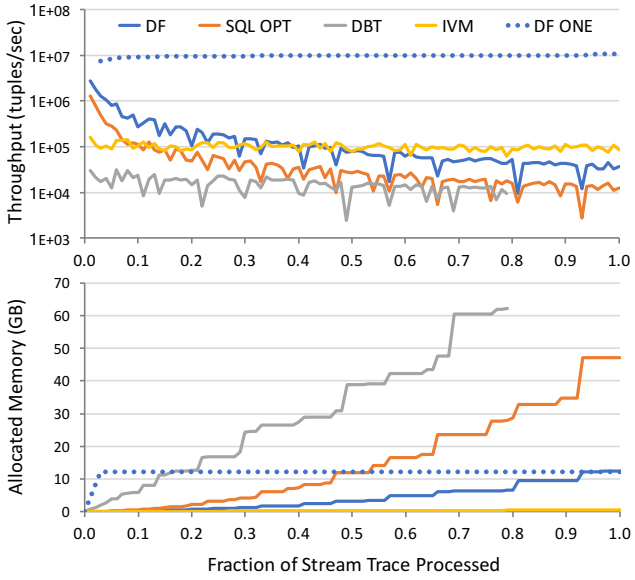


Figure 8: The performance of incremental maintenance of the covariance matrix on top of the triangle query over the *Twitter* dataset under updates of size $1,000$ to all base relations.

us to precompute the views that are unaffected by changes in `Inventory` and avoid materialization of those views that do not directly join with that relation. Thus, restricting updates to only one relation leads to materializing fewer views, which in turn reduces the maintenance overhead.

Figure 6 shows the throughput of processing updates for incremental maintenance of the covariance matrix in these two scenarios. If we restrict updates only to `Inventory`, then we can avoid materializing all views on the leaf-to-root path covered by that relation. This corresponds more to a streaming scenario where we compute a continuous query and do not store the stream. Restricting updates to only one relation improves the average throughput, 3.2x in **DF** and 1.6x in **SQL OPT**, and also reduces memory requirements (note the log scale on the y-axis). The latter is also reflected in smoother throughput curves for the ONE variants. In **DBT**, restricting updates brings constant time maintenance per view, yet the number of materialized views is still large.

Batched incremental processing is also beneficial for recomputing the entire covariance matrix from scratch. Using medium-sized batches of updates brings better performance, cf. Figure 7, but can also lower memory requirements and improve cache locality during query processing. For instance, incrementally processing the entire *Retailer* dataset in chunks of $1,000$ tuples can bring up to 50% performance improvements compared to the re-evaluation approach that loads and processes the entire dataset at once.

Covariance matrix over cyclic joins. We evaluate the performance of batched incremental processing of the covariance matrix on top of the triangle query for updates of size $1,000$ to all three relations, R, S, and T. The number of updates to each relation is the same. Alternatively, if R, S, and T were representing the same relation, we would consider three trigger executions instead of one per update.

For **DF** and **SQL OPT**, we consider the view tree where S and T are placed at the leaves so that both strategies materialize the join of S and T of size $O(N^2)$. Their time

complexity for single-tuple update to R is $O(1)$, but updating the join of S and T takes $O(N)$. **DBT** materializes 21 views in total (to maintain 6 covariance aggregates), out of which 12 views represent joins of two relations. Its time complexity for maintaining the covariance matrix upon a single-tuple update to either of the three relations is $O(N)$. The **IVM** strategy maintains just the base relations and recomputes the delta upon each update in linear time.

The throughput rate of the strategies that materialize views of quadratic size declines as the input stream progresses. **DBT** exhibits the highest processing and memory overheads caused by storing 12 auxiliary views of quadratic size. **DF** outperforms **SQL OPT** due to its efficient encoding of the covariance aggregates, which also results in 3.8x lower peak memory utilization. **IVM** exhibits a 10% decline in performance after processing the entire trace, which is due to the linear time maintenance. Overall, the extent of this decrease is much lower compared to the other approaches with the quadratic space complexity. Restricting updates to R only makes **DF-ONE** requiring just a lookup in the materialized view joining S and T per update. This strategy has two orders of magnitude higher throughput than **IVM** at the expense of using 30 times more memory.

Clique queries like triangles provide no factorization opportunities. Materializing auxiliary views to speed up incremental view maintenance increases memory and processing overheads, in which case classical IVM techniques are more appropriate. The **DF** approach can be tuned, if necessary, to trade space for time and skip materialization of supporting views for cliques in large queries.

7. RELATED WORK

There is a wealth of work in the ML community on incremental or online learning over *arbitrary* relations, e.g., [40]. Our approach learns over *joins* and crucially exploits the join factorization of the underlying training dataset to improve the performance; as such, it is specific to a database setting. We next consider immediately related work on incremental maintenance of in-database analytics.

To the best of our knowledge, ours is the first approach to propose factorized IVM of regression models. It builds on a new combination of two distinct lines of prior work: higher-order delta-based incremental view maintenance (IVM) and factorized computation of in-database analytics.

IVM. IVM is a well-studied area spanning more than three decades, we refer the interested reader to a recent survey [14]. Prior work extensively studied IVM for various query languages and showed that the time complexity of IVM is lower than that of re-computation. We go beyond prior work as we adapt DBToaster, a state-of-the-art higher-order IVM for queries with joins and aggregates [23], to factorized computation of aggregates over joins [8] and to learning regression models over factorized joins [39]. DBToaster uses one materialization hierarchy per relation in the query, whereas we use one view tree for all relations. One effect is that DBToaster has much larger space requirements and update times. Furthermore, the current implementation of DBToaster does not primarily target the maintenance of covariance matrices or large sets of aggregates over joins. This is also observed experimentally in Section 6.

DBToaster uses generalized multiset relations in which tuples have associated elements from an arbitrary ring, and the addition and multiplication operations over such generalized relations form a ring of databases [22]. In our framework, the data model is that of generalized multiset relations, and the view language is a subset of that used in DBToaster. This allowed us to adapt DBToaster to perform factorized IVM on computation-specific rings.

We are aware of ongoing, independent work on factorized IVM for acyclic joins [41]. This is strictly subsumed by our general framework when the payload ring is a ring of databases defined over a set of generalized multiset relations mapping tuples to integer multiplicities [22]. The so-called q-hierarchical join queries (such as the Housing query in our experiments) are exactly those self-join-free conjunctive queries that admit constant time update [9]. Recent work on in-database maintenance of linear regression models shows how to compute such models using previously computed models over distinct sets of features [18]. Its contribution is complementary to our algorithmic and complexity contributions and it shares a similar goal with our prior work on reusing gradient computation to efficiently explore the space of possible regression models [32]. Exploiting key attributes to enable succinct delta representations and accelarate maintenance can also complement our approach [21]. LINVIEW shares the goal of our system to incrementally maintain regression models, but it requires the relational materialization of the full join result [31]. Applying its idea of matrix factorization in R or MATLAB also depends on the full join result, comes with greater costs, and is not scalable; in some of our experiments, the full join result needs space beyond the available memory and time longer than re-computing the regression model over the factorized join.

Most commercial database systems, e.g., Oracle [3] and SQLServer [1], support incremental view maintenance for restricted classes of queries. LogicBlox supports higher-order IVM for Datalog (meta)programs [6, 17]. Trill is a streaming engine that supports incremental processing of relational-style queries but no covariance matrix computation [13].

In-DB analytics. Beyond IVM, there is a solid body of related work at the intersection of databases and machine learning, cf. a SIGMOD 2015 panel [36]. Our work follows a recent line of research on marrying databases and machine learning [19, 15, 10, 24, 26, 28, 20, 12, 38, 36, 35] and in particular builds on *static* factorized in-database learning [16, 37, 39, 32]. Our factorization approach is that from prior work [39, 32]. Limited forms of factorized learning have been also used by Rendle [37] and Kumar et al. [24]. The former considers zero-suppresed design matrices for high-degree regression models called factorization machines. The latter proposes a framework for learning generalized linear models over key-foreign key joins in a distributed environment.

Most efforts in the database community are on designing systems to support large-scale machine learning libraries on distributed architectures [16], e.g., MLLib [26] and DeepDist [28] on Spark [42], GLADE [35], MADlib [19] on PostgreSQL, SystemML [20, 10], system benchmarking [12] and sample generator for cross-validate learning [38].

8. CONCLUSION

We introduce a principled approach to factorized incremental maintenance of in-database learning of learning regression models over joins, which relies on a new ring that captures the computation of covariance matrices. Our approach is applicable beyond the rings discussed in the paper. Following earlier work showing that factorized computation

of aggregates over joins captures a host of core computational problems [5], we are currently investigating the applicability of our approach to incremental maintenance of inference in probabilistic graphical models, matrix chain computation, and count-SAT. The goal is to provide one IVM mechanism to such core problems of diverse interest.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 682588. The authors also acknowledge awards from Amazon Cloud Credits for Research, Google Research, and Fondation Wiener Anspach.

9. REFERENCES

[1] Create Indexed Views. `http://msdn.microsoft.com/en-us/library/ms191432.aspx`.

[2] Higgs Twitter Dataset. `https://snap.stanford.edu/data/higgs-twitter.html`.

[3] Materialized View Concepts and Architecture. `http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmview.htm`.

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, and et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, volume 5, pages 277–289, 2005.

[5] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.

[6] M. Aref and et al. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.

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

[8] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[9] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *PODS*, 2017. to appear. also arXiv:1702.06370.

[10] M. Boehm and et al. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[11] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.

[12] Z. Cai and et al. A comparison of platforms for implelementing and running very large scale machine learning algorithms. In *SIGMOD*, pages 1371–1382, 2014.

[13] B. Chandramouli, J. Goldstein, and et al. Trill: A high-performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4):401–412, 2014.

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

[15] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer. Machine learning for big data. In *SIGMOD*, 2013.

[16] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*, pages 325–336, 2012.

[17] T. J. Green, D. Olteanu, and G. Washburn. Live programming in the LogicBlox system: A MetaLogiQL approach. *PVLDB*, 8(12):1782–1791, 2015.

[18] P. Gupta, N. Koudas, E. Shang, R. Johnson, and C. Zuzarte. Processing analytical workloads incrementally. *CoRR*, abs/1509.05066, 2015.

[19] J. M. Hellerstein and et al. The MADlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[20] B. Huang and et al. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

[21] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing IDs to accelerate incremental view maintenance. In *SIGMOD*, 2015.

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

[23] 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.

[24] A. Kumar, J. F. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[25] S. R. Madden and et al. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *TODS*, 30(1):122–173, 2005.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, and et al. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, 2016.

[27] R. Menich and N. Vasiloglou. The future of LogicBlox machine learning. LogicBlox User Days, 2013.

[28] D. Neumann. Lightning-fast deep learning on Spark via parallel stochastic gradient updates, `www.deepdist.com`, 2015.

[29] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[30] M. Nikolic, M. Dashti, and C. Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *SIGMOD*, pages 511–526, 2016.

[31] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD*, pages 253–264, 2014.

[32] D. Olteanu and M. Schleich. F: regression models over factorized views. *PVLDB*, 9(13):1573–1576, 2016.

[33] D. Olteanu and M. Schleich. Factorized Databases. *SIGMOD Rec.*, 45(2):5–16, 2016.

[34] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015.

[35] C. Qin and F. Rusu. Speculative approximations for terascale distributed gradient descent optimization. In *DanaC*, pages 1–10, 2015.

[36] C. Ré and et al. Machine learning and databases: The sound of things to come or a cacophony of hype? In *SIGMOD*, pages 283–284, 2015.

[37] S. Rendle. Scaling factorization machines to relational data. *PVLDB*, 6(5):337–348, 2013.

[38] S. Schelter and et al. Efficient sample generation for scalable meta learning. In *ICDE*, 2015.

[39] M. Schleich, D. Olteanu, and R. Ciucanu. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, pages 3–18, 2016.

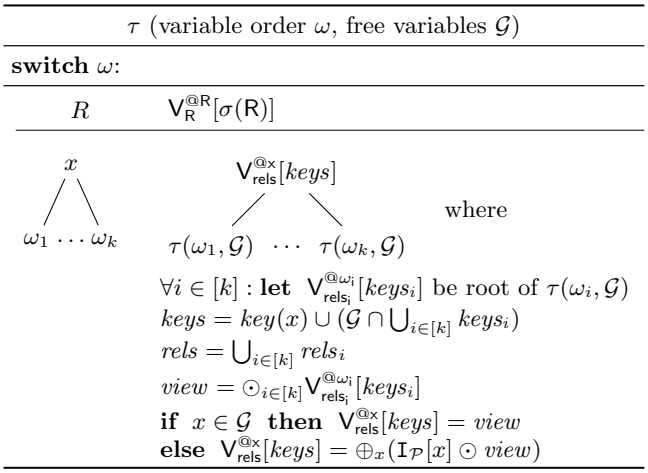


Figure 9: Algorithm for creating a view tree $\tau(\omega, \mathcal{G})$ from a variable order ω and a set of free variables $\mathcal{G}$ with views over ring $\mathcal{P}$.

[40] S. Shalev-Shwartz. Online learning and online convex optimization. *Found. & Trends in ML*, 4(2), 2012.

[41] S. Vansummeren. Personal communication, 2017.

[42] M. Zaharia and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

APPENDIX

Appendix A gives an extension of our factorized IVM framework to queries with *group-by* aggregates over joins.

Appendix B further strengthens the framework by showing how the views in a view tree can be factorized; this is in addition to the factorization of the computation and maintenance as provided by variable orders and view trees. View factorization can effectively lower the space and time complexities for maintenance and recomputation.

Our framework is modular thanks to its use of payload rings. Appendix C shows how it can capture the maintenance of arbitrary conjunctive queries and matrix chain multiplication.

A. GROUP-BY AGGREGATES

We generalize our framework to group-by aggregates over joins of relations that map tuples to payloads from the same ring. The development in the main body of this paper considered queries that marginalize over all variables, that is, there are no free variables. In contrast, group-by aggregates have free variables. We write such queries as functional aggregate queries or FAQ expressions for short [5]:

$$Q(y_1, \ldots, y_g) = \sum_{x_1} \cdots \sum_{x_m} \prod_{i \in [n]} \mathbf{1}_{R_i(\sigma_i)}$$

where $\mathbf{1}_E$ is the Kronecker delta that is 1 in case E holds and 0 otherwise, σ_i is the set of variables in the relation symbol R_i, the variables $(y_j)_{j \in [g]}$ in the head of Q are the free variables while the variables $(x_j)_{j \in [m]}$ are bound. Assuming the payloads are from a ring $\mathcal{P}$, the above FAQ expression is stated as follows in our framework:

$$Q[y_1, \ldots, y_g] = \oplus_{x_1, \cdots, x_m} (\odot_{i \in [n]} R_i[\sigma_i] \odot_{j \in [m]} \mathtt{I}_{\mathcal{P}}[x_j])$$

To evaluate such queries, we extend the view tree construction procedure shown in Figure 3 to consider free variables. Figure 9 gives a generalized algorithm that constructs a view tree $\tau(\omega, \mathcal{G})$ for a variable order ω and a set of free variables $\mathcal{G}$ of a given query Q. Each view in the view tree retains free variables in its schema and marginalizes over bound variables. The top-level view in a view tree defines the query result. For queries without free variables, the two algorithms for view tree construction from Figures 3 and 9 yield the same view tree. For queries with free variables, several views may be identical: This is the case when all variables in their keys are free. In this case, we only need to materialize one of the several identical views while the other identical views remain non-materialized.

The generalized algorithm operates on *any* variable order that is valid for a given query Q. Following prior work on variable orders for the evaluation of group-by aggregates over joins [8, 5], the variable orders that ensure lowest space and time complexities have the free variables above the bound variables in the variable order. This leads to view trees where marginalization over bound variables happens as early as possible in a bottom-up traversal of the view tree.

Example 7. Consider a path query Q that computes a group-by aggregate within a given ring $\mathcal{P}$ over a natural join of the input relations $R[a, b]$, $S[b, c]$, $T[c, d]$, and $U[d, e]$, where the free variables are $\mathcal{G} = \{b, c, d\}$:

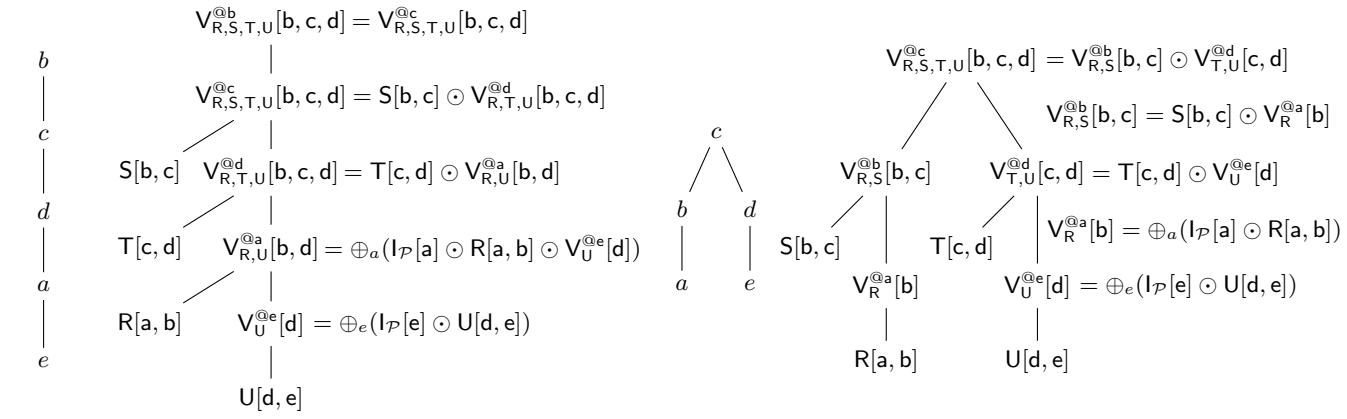
$$Q[b, c, d] = \\ \oplus_{a,e} (R[a, b] \odot S[b, c] \odot T[c, d] \odot U[d, e] \odot \mathtt{I}_{\mathcal{P}}[\mathsf{a}] \odot \mathtt{I}_{\mathcal{P}}[\mathsf{e}])$$

For the `COUNT` ring, the values for a and e are mapped to 1 and the query Q computes the count for each tuple of free variables. For the `SUM` ring, the values are mapped to reals and then the query Q computes the sum of products of values for a and e for each distinct tuple of free variables.

We next show how to compute and maintain this query using two distinct variable orders. We first consider the variable order ω_1 in Figure 10(a) and then a second variable order ω_2 in Figure 10(c). Both variable orders have the free variables $\{b, c, d\}$ above the bound variables. For simplicity in the complexity analysis below, we assume all relations have size N.

Figure 10(b) shows the view tree $\tau(\omega_1, \{b, c, d\})$ derived using the algorithm from Figure 9. Its views marginalize over the bound variables a and e. The view $\mathsf{V}^{@\mathsf{a}}_{\mathsf{R,U}}[\mathsf{b}, \mathsf{d}]$ at variable a is a product of the marginalization over a and the view at e. The use of a product is suboptimal and suggests there may be a different better variable order for this query or we can keep the view non-materialized (cf. Appendix B).

For both recomputation and incremental maintenance, we need $O(N^2)$ space since we need at least two relations to cover the free variables $\{b, c, d\}$, while all bounded variables are marginalized as soon as possible. Whereas for recomputation we only store the top view, for maintenance we need to store all views and the views $\mathsf{V}^{@\mathsf{a}}_{\mathsf{R,U}}[\mathsf{b}, \mathsf{d}]$, $\mathsf{V}^{@\mathsf{d}}_{\mathsf{R,T,U}}[\mathsf{b}, \mathsf{c}, \mathsf{d}]$, and $\mathsf{V}^{@\mathsf{c}}_{\mathsf{R,S,T,U}}[\mathsf{b}, \mathsf{c}, \mathsf{d}]$ can have $O(N^2)$ size. For recomputation, the time complexity is also $O(N^2)$ for the same reason. For incremental maintenance under single-tuple updates to any relation, incremental maintenance takes $O(N)$ time, since the key variables of each view that are not bound to a constant in the update can be covered by one relation. In this example, the root of the view tree need not be materialized as it is identical to its child.



(a) ω_1 (b) $\tau(\omega_1, \{b,c,d\})$ (c) ω_2 (d) $\tau(\omega_2, \{b,c,d\})$

Figure 10: For the query Q from Example 7 with free variables $\{b,c,d\}$ and payload ring $\mathcal{P}$, from left to right: variable order ω_1 for Q, view tree $\tau(\omega_1, \{b,c,d\})$, variable order ω_2, and view tree $\tau(\omega_2, \{b,c,d\})$.

We next turn to the variable order ω_2 in Figure 10(c), where the (conditionally) independent free variables b and c are in different branches. The time and space complexities of recomputation and incremental maintenance are asymptotically the same as for ω_1. However, incremental maintenance using $\tau(\omega_2, \{b,c,d\})$ achieves lower costs in practice as it requires maintenance of only one view of quadratic size, the top view. As discussed in Appendix B, we can factorized the top view to further reduce the maintenance time to $O(1)$ and required space to $O(N)$. □

B. FACTORIZED VIEWS

Our factorized IVM framework uses variable orders as plans for query evaluation and to avoid the size explosion for intermediate query results. So far, we have considered the factorization of the query by means of a view tree. This suffices for queries without free variables, such as for aggregates without group-by clauses as considered in the main body of the paper. There, the variable order (and consequently the view tree) ensures that each bounded variable is marginalized over as soon as possible.

However, for queries with free variables our approach does not effectively exploit the conditional independence among the free variables to achieve smaller views that require lower maintenance cost. In this section, we are after factorizing the keys of the views. For instance, the top-level view in Figure 10(d) uses a relational (flat) representation of the quadratically many tuples over its keys (b,c,d) of quadratic size (since we need two relations to cover all these key variables). However, b and d are independent given c and the view could be factorized as a join on c of one subview over (c,b) and a second subview over (c,d), with both subviews of linear size.

Example 8. Consider the variable order and view tree from Figure 10(c-d). The top view is not materialized but expressed as a pair of its children, $(V^{@b}_{R,S}[b,c], V^{@b}_{T,U}[c,d])$. Each of these two views is materialized as a multi-indexed map. Querying the result for given b, c, and d values requires two map lookups, thus $O(1)$ operation. We can enumerate tuples in the result of the factorized view with constant delay, following prior results on enumeration for factorized representations of query results [34]. The space complexity of storing these maps is linear. Incremental maintenance for single-tuple updates in S and T takes $O(1)$ time and in R and U takes $O(N)$ time. By factorizing this view, we can thus reduce the required space for view materialization while also improving the single-tuple update time! □

τ (variable order ω, free variables $\mathcal{G}$)

switch ω:

R	$V^{@R}_{R}[\sigma(R)]$

x with children $\omega_1 \cdots \omega_k$ — $V^{@x}_{rels}[keys]$ with children $\tau(\omega_1, \mathcal{G}) \cdots \tau(\omega_k, \mathcal{G})$ where

$\forall i \in [k]$: **let** $V^{@\omega_i}_{rels_i}[keys_i]$ be root of $\tau(\omega_i, \mathcal{G})$
$keys = key(x) \cup (\mathcal{G} \cap \bigcup_{i\in[k]} keys_i)$
$rels = \bigcup_{i\in[k]} rels_i$
if $x \notin \mathcal{G}$ **then**
  $V^{@x}_{rels}[keys] = \oplus_x(I_{\mathcal{P}}[x] \odot_{i\in[k]} V^{@\omega_i}_{rels_i}[keys_i])$
else
  if $keys \subseteq \mathcal{G}$ **then**
    $V^{@x}_{rels}[keys] = (V^{@\omega_i}_{rels_i}[keys_i])_{i\in[k]}$
  else
    $V^{@x}_{rels}[keys] = \odot_{i\in[k]} V^{@\omega_i}_{rels_i}[keys_i]$

Figure 11: Algorithm for creating a factorized view tree $\tau(\omega, \mathcal{G})$ from a variable order ω and a set of free variables $\mathcal{G}$ with views over ring $\mathcal{P}$.

Figure 11 shows the algorithm for creating a view tree $\tau(\omega, \mathcal{G})$ for a given variable order ω and a set of free variable $\mathcal{G}$, where views are selectively materialized based on their keys. The algorithm materializes views that marginalize over a bound variable to support incremental maintenance. It also materializes views that have a mix of free and bound keys in order to speed up subsequent marginalization (summation) operations further up in the view tree. However, the algorithm avoids materialization of views whose keys are only free keys, denoted by a tuple of views $(V_1, \ldots, V_n)$, as in such cases child views are conditionally independent

on some of the free variables.

Our refinement to factorize those views whose keys only have free variables is at least as good as non-factorization in case of acyclic queries, but it can lead to lower space and time complexities. However, this is not the case for clique queries, as discussed in the next example.

Example 9. Consider the triangle query from Equation (1) in which x, y, and z are free variables. Assume the variable order $x-y-z$. The view at node z representing the join between R_2 and R_3 is *not* materialized since its keys x, y, and z are free variables; instead, the algorithm keeps this view in factorized form as a pair $(\mathsf{R}_2[\mathsf{y},\mathsf{z}], \mathsf{R}_3[\mathsf{x},\mathsf{z}])$. The view at node y representing the join of the three relations has the same set of keys, so it is also not materialized but represented as a pair $(\mathsf{R}_1[\mathsf{x},\mathsf{y}], (\mathsf{R}_2[\mathsf{y},\mathsf{z}], \mathsf{R}_3[\mathsf{x},\mathsf{z}]))$. Postponing view materialization until the tree root allows us to join the three relations using a worst-case optimal join algorithm [29], thus avoiding the explosion of intermediate results, in $O(N^{3/2})$ time. The space complexity of incremental maintenance is linear in the relation size, $O(N)$, and processing single-tuple updates to any relation requires $O(N)$ time. □

Factorized views avoid materialization and thus introduce another dimension to the search space of possible view trees when computing dynamic factorization width.

C. APPLICATIONS

In this section, we show how two further problems can be expressed in our framework: maintaining the results of conjunctive queries and of matrix chain computations. Further applications as possible, such as computing marginal probabilities and MAP in probabilistic graphical models.

C.1 Conjunctive queries

Consider conjunctive queries Q over relations $R_i(\sigma_i)_{i\in[n]}$:

$$Q(y_1,\ldots,y_g) = R_1(\sigma_1),\ldots,R_n(\sigma_n)$$

To incrementally maintain the result of Q under input updates, we construct a view tree over the COUNT (#) ring using the algorithm from Figure 11 on any variable order for Q. We thus maintain a count for every tuple in the query result that states the number of derivations of that tuple from the input tuples. The tuple is in the result if its count is greater than 0.

Example 10. Given the schema from Figure 1, we want to compute the count aggregate over the natural join of the three relations grouped by z and s.

$$\mathsf{Q}[\mathsf{z},\mathsf{s}] = \oplus_{p,h,t}(\mathtt{I}_{\#}[p] \odot \mathtt{I}_{\#}[h] \odot \mathtt{I}_{\#}[t] \odot \mathsf{House}[z,s,p] \odot \mathsf{Shop}[z,h] \odot \mathsf{Tax}[s,t])$$

We fix a variable order, say $z-s-\{h,t,p\}$. The input relations are located below leaf variables. The view tree derived from this variable order consists of the following views (from bottom to top):

$$\mathsf{V}^{@\mathsf{h}}_{\mathrm{Shop}}[z] = \oplus_h(\mathtt{I}_{\#}[h] \odot \mathrm{Shop}[z,h])$$
$$\mathsf{V}^{@\mathsf{t}}_{\mathrm{Tax}}[s] = \oplus_t(\mathtt{I}_{\#}[t] \odot \mathrm{Tax}[s,t])$$
$$\mathsf{V}^{@\mathsf{p}}_{\mathrm{House}}[z,s] = \oplus_p(\mathtt{I}_{\#}[p] \odot \mathrm{House}[z,s,p])$$
$$\mathsf{V}^{@\mathsf{s}}_{\mathrm{Shop,House,Tax}}[z,s] = \mathsf{V}^{@\mathsf{h}}_{\mathrm{Shop}}[z] \odot \mathsf{V}^{@\mathsf{t}}_{\mathrm{Tax}}[s] \odot \mathsf{V}^{@\mathsf{p}}_{\mathrm{House}}[z,s]$$
$$\mathsf{V}^{@\mathsf{z}}_{\mathrm{Shop,House,Tax}}[z,s] = \mathsf{V}^{@\mathsf{s}}_{\mathrm{Shop,House,Tax}}[z,s]$$

For incremental maintenance, the views at h, t, and p are materialized, while the views at s and z are factorized and not materialized. Let us assume that each relation has size N. Each of the former three views takes $O(N)$ space. The latter two views are equivalent and represent the query result. Incremental maintenance under single-tuple updates to House takes $O(1)$ time since all free variables are bound to update values. For single-tuple updates to Shop and Tax, incremental maintenance requires $O(N)$ time. If we would materialize the view $\mathsf{V}^{@\mathsf{s}}_{\mathrm{Shop,House,Tax}}[z,s]$, then we would need $O(N)$ space for it, yet without any improvement in the update times. □

C.2 Matrix Chain Multiplication

Consider the problem of computing a product of a series of matrices $\mathbf{A}_1,\ldots,\mathbf{A}_n$ over some ring $\mathcal{P}$, where matrix $\mathbf{A}_i[x_i, x_{i+1}]$ has the size of $p_i \times p_{i+1}$, $i \in [n]$. The product $\mathbf{A} = \mathbf{A}_1 \cdots \mathbf{A}_n$ of size $p_1 \times p_{n+1}$ can be formulated as:

$$\mathbf{A}[x_1, x_{n+1}] = \sum_{x_2\in[p_2]} \cdots \sum_{x_n\in[p_n]} \prod_{i\in[n]} \mathbf{A}_i[x_i, x_{i+1}]$$

In our framework, we express matrix $\mathbf{A}_i$ as a relation $A_i[x_i, x_{i+1}]$ and compute $\mathbf{A}$ as a `SUM` aggregate over a path join grouped by (x_1, x_{n+1}). For any variable order, we then construct a view tree over the SUM $(+)$ ring.

Example 11. Consider a chain of matrix multiplications $\mathbf{A} = \mathbf{A}_1 \cdots \mathbf{A}_4$, where all matrices are of equal size $p \times p$. Each matrix is represented as a relation $\mathbf{A}_i[x_i, x_{i+1}]$. Let $\mathcal{G} = \{x_1, x_5\}$ be the set of free variables and ω be the variable order $x_1 - x_5 - x_3 - \{x_2, x_4\}$, with the matrices being placed below the leaf variables in ω. Then, the view tree $\tau(\omega, \mathcal{G})$ has the following views (from bottom to top):

$$\mathsf{V}^{@\mathsf{x}_2}_{\mathsf{A}_1,\mathsf{A}_2}[\mathsf{x}_1,\mathsf{x}_3] = \oplus_{x_2}(\mathsf{I}_{+}[\mathsf{x}_2] \odot A_1[x_1,x_2] \odot A_2[x_2,x_3])$$
$$\mathsf{V}^{@\mathsf{x}_4}_{\mathsf{A}_3,\mathsf{A}_4}[\mathsf{x}_3,\mathsf{x}_5] = \oplus_{x_4}(\mathsf{I}_{+}[\mathsf{x}_4] \odot A_3[x_3,x_4] \odot A_4[x_4,x_5])$$
$$\mathsf{V}^{@\mathsf{x}_3}_{\mathsf{A}_1,\mathsf{A}_2,\mathsf{A}_3,\mathsf{A}_4}[\mathsf{x}_1,\mathsf{x}_5] = \oplus_{x_3}(\mathsf{I}_{+}[\mathsf{x}_3] \odot \mathsf{V}^{@\mathsf{x}_2}_{\mathsf{A}_1,\mathsf{A}_2}[\mathsf{x}_1,\mathsf{x}_3] \odot \mathsf{V}^{@\mathsf{x}_4}_{\mathsf{A}_3,\mathsf{A}_4}[\mathsf{x}_3,\mathsf{x}_5])$$

The view $\mathsf{V}^{@\mathsf{x}_3}_{\mathsf{A}_1,\mathsf{A}_2,\mathsf{A}_3,\mathsf{A}_4}[\mathsf{x}_1,\mathsf{x}_5]$ is equivalent to its ancestor views at x_1 and x_5 and represents the query result. The time complexity of computing $\mathbf{A}$ is $O(p^3)$. Materializing the views at x_2, x_3, and x_4 for incremental maintenance takes $O(p^2)$ space. Processing single-tuple changes to any input matrix takes $O(p^2)$ time.

Next we generalize this example to a product of n matrices of equal size $p \times p$. Consider a variable order that yields a binary view tree of the lowest depth with x_1 and x_{n+1} at the top: $x_1 - x_{n+1} - \{V_1, V_2\}$, where the variable orders V_1 and V_2 are balanced binary trees with roots $x_{n/4}$ and respectively $x_{3n/4}$. It takes $O(p^3 \cdot n)$ time to compute $\mathbf{A}$ from scratch. For incremental maintenance, the space needed to materialize each of the n views, except for the top two views, is $O(p^2)$. Maintaining $\mathbf{A}$ upon a single-tuple update to $\mathbf{A}_1$ or $\mathbf{A}_n$ requires $O(p^2 \log n)$ time, whereas single-tuple updates to all other matrices take $O(p^3 \log n)$ time. □

Different variable orders lead to different evaluation plans for matrix chain multiplication. The optimal variable ordering corresponds to the optimal sequence of matrix multiplications that minimizes the overall multiplication cost, which is the textbook Matrix Chain Multiplication problem solved by dynamic programming.