SPECIAL ISSUE PAPER

F-IVM: analytics over relational databases under updates

Ahmet Kara[1] · Milos Nikolic[2] · Dan Olteanu[1] · **Haozhe Zhang[1]**

Abstract

This article describes F-IVM, a unified approach for maintaining analytics over changing relational data. We exemplify its versatility in four disciplines: processing queries with group-by aggregates and joins; learning linear regression models using the covariance matrix of the input features; building Chow-Liu trees using pairwise mutual information of the input features; and matrix chain multiplication. F-IVM has three main ingredients: higher-order incremental view maintenance; factorized computation; and ring abstraction. F-IVM reduces the maintenance of a task to that of a hierarchy of simple views. Such views are functions mapping keys, which are tuples of input values, to payloads, which are elements from a ring. F-IVM supports efficient factorized computation over keys, payloads, and updates. It treats uniformly seemingly disparate tasks: While in the key space, all tasks require general joins and variable marginalization, in the payload space, tasks differ in the definition of the sum and product ring operations. We implemented F-IVM on top of DBToaster and show that it can outperform classical first-order and fully recursive higher-order incremental view maintenance by orders of magnitude while using less memory.

Keywords Incremental view maintenance · Factorized databases · Covariance matrix · Learning linear regression models · Mutual information · Chow-Liu tree · Commutative ring

1 Introduction

Supporting modern applications that rely on accurate and real-time analytics computed over large and continuously evolving databases is a challenging data management problem [4]. Special cases are the classical problems of incremental view maintenance (IVM) [15, 32] and stream query processing [1, 36].

Recent efforts studied the problem of computing machine learning (ML) tasks over *static* databases. The predominant approach loosely integrates the database systems with the statistical packages [26, 34, 40, 54, 55]: First, the database

system computes the input to the statistical package by joining the database relations. It then exports the join result to the statistical package for training ML models. This approach precludes real-time analytics due to the expensive export/import steps. Systems like Morpheus [35] and LMFAO [59] push the ML task inside the database and learn ML models over static normalized data. In particular, LMFAO and its precursors F [58] and AC/DC [30] decompose the task of learning classification and regression models over arbitrary joins into factorized computation of aggregates over joins and fixpoint computation of model parameters. This factorization may significantly lower the complexity by avoiding the computation of Cartesian products lurking within joins [5, 52]. Both the tight integration of the database computation step and of the statistical computation step as well as the factorized computation are pre-requisites for real-time analytics.

This article describes F-IVM,[1] a unified approach for maintaining analytics over changing relational data. We exemplify its versatility in four disciplines: processing queries with group-by aggregates and joins; learning linear regression models using the covariance matrix of the input features; building Chow-Liu trees using pairwise

✉ Dan Olteanu
olteanu@ifi.uzh.ch

Ahmet Kara
kara@ifi.uzh.ch

Milos Nikolic
milos.nikolic@ed.ac.uk

Haozhe Zhang
zhang@ifi.uzh.ch

[1] Department of Informatics, University of Zurich, Zurich, Switzerland

[2] School of Informatics, University of Edinburgh, Edinburgh, UK

---

[1] https://github.com/fdbresearch/FIVM.

mutual information matrix of the input features; and matrix chain multiplication.

F-IVM has three main ingredients: higher-order incremental view maintenance (IVM); factorized computation and data representation; and ring abstraction.

The first ingredient reduces the maintenance task to that of a hierarchy of simple views. Such views are functions mapping keys, which are tuples of input values, to payloads, which are elements from a ring. In contrast to classical (first-order) IVM, which computes changes to the query result on the fly and does not use extra views, F-IVM can significantly speed up the maintenance task and lower its complexity by using carefully chosen views. Yet F-IVM can use substantially fewer views than the fully recursive IVM, which is used by the state-of-the-art IVM system DBToaster [32]. In our experiments, F-IVM outperforms first-order and higher-order IVM by up to two orders of magnitude in both runtime and memory requirements.

The second ingredient supports efficient computation and representation for keys, payloads, and updates. F-IVM exploits insights from query evaluation algorithms with best known complexity and optimizations that push aggregates past joins [3, 5, 52]. It can process bulk updates expressed as low-rank decompositions [33, 61] and maintain a factorized representation of query results, which is essential to achieve low complexity for free-connex acyclic and q-hierarchical queries.

The third ingredient allows F-IVM to treat uniformly seemingly disparate tasks. In the key space, all tasks require joins and variable marginalization. In the payload space, tasks differ in the ring operations. To maintain linear regression models and Chow-Liu trees under updates, F-IVM uses a new ring that captures the maintenance of a covariance matrix over continuous and categorical features from the input database. Furthermore, it composes rings to capture the data-dependent computation for complex analytics. Thanks to the ring abstraction, F-IVM is highly extensible: efficient maintenance for new analytics over relational databases is readily available as long as they come with appropriate sum and product ring operations.

F-IVM was introduced in prior work [46]. This article complements it as follows. Section 2 overviews the F-IVM design. Section 7 shows how F-IVM achieves the best known complexity for q-hierarchical and free-connex acyclic queries. Section 8 introduces the covariance ring over continuous and categorical features. Section. 9 includes new experiments on: the covariance matrix for continuous and categorical features; end-to-end linear regression models; Chow-Liu trees; q-hierarchical queries with eager and lazy approaches and payloads carrying the listing or the factorized representation of the query result; and path queries of increasing length on graph data to stress-test the scalability of the IVM engines.

1.1 F-IVM by example

Consider the following SQL query over a database $\mathcal{D}$ with relations $R(A, B)$, $S(A, C, E)$, and $T(C, D)$:

```
Q := SELECT A, C, SUM(B*D*E)
       FROM R NATURAL JOIN S NATURAL JOIN T
       GROUP BY A, C;
```

A naïve query evaluation approach first computes the join and then the aggregate. This takes $\mathcal{O}(N^3)$ time, where N is the size of $\mathcal{D}$. An alternative approach exploits the distributivity of SUM over multiplication to partially push the aggregate past joins and then combine the partial aggregates. For instance, one such partial sum over S can be expressed as the view V_S:

```
VS := SELECT A, C, SUM(E) AS SE
       FROM S GROUP BY A, C;
```

In the view V_S, we identify keys, which are tuples over (A, C), and payloads, which are aggregate values S_E. Similarly, we compute partial sums over R and T as views V_R and V_T. These views are joined as depicted by the *view tree* in Fig. 1, which is akin to a query plan with aggregates pushed past joins. This view tree computes the result of Q in $\mathcal{O}(N)$ time.

Consider now the problem of learning, for each pair (a, c) of (A, C)-values in the natural join of R, S, and T, a linear function $f_{a,c}$ with parameters θ_0, θ_D and θ_E that predicts the label B given the features D and E:

$$f(D, E) = \theta_0 + \theta_D \cdot D + \theta_E \cdot E$$

Our insight is that the same view tree in Fig. 1 can compute the gradient vector used for learning $f_{a,c}$, where we replace the SQL SUM and * operators.

As shown in Sect. 8.1, the gradient of the square loss objective function needs the computation of three types of aggregates: the scalar c that is the count aggregate SUM(1); the vector $\boldsymbol{s}$ of linear aggregates SUM(i), for $i \in \{B, D, E\}$; and the matrix $\boldsymbol{Q}$ of quadratic aggregates SUM($i * j$), where
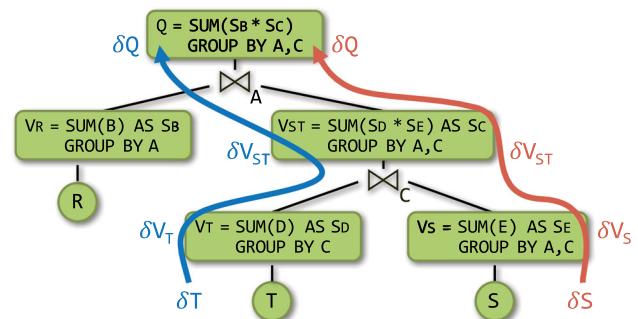
Fig. 1 View tree for the query in Ex. 1.1. The propagation paths for updates to S (right red) and to T (left blue)

$i, j \in \{B, D, E\}$. These aggregates capture the correlation between the features and the label.

We treat these aggregates as one compound aggregate $(c, \mathbf{s}, \mathbf{Q})$ so we can share computation across them. This compound aggregate can be partially pushed past joins similarly to the SUM aggregate discussed before. Its values are carried in the key payloads of views in the view tree from Fig. 1. For instance, the partial compound aggregate $(c_T, \mathbf{s}_T, \mathbf{Q}_T)$ at the view V_T computes, for each C-value, the count, sum, and sum of squares of the D-values in T. Similarly, the partial aggregate $(c_S, \mathbf{s}_S, \mathbf{Q}_S)$ at the view V_S computes, for each pair (A, C), the count, sum, and sum of squares of E-values in S. In the view V_{ST}, which is the join of V_T and V_S, each key (a, c) is associated with the multiplication of the payloads for the keys c in V_T and (a, c) in V_S. This multiplication works on compound aggregates: The scalar c_{ST} is the arithmetic multiplication of c_T and c_S; the vector of linear aggregates $\mathbf{s}_{ST}$ is the sum of the scalar-vector products $c_T \mathbf{s}_S$ and $c_S \mathbf{s}_T$; finally, the matrix $\mathbf{Q}_{ST}$ of quadratic aggregates is the sum of the scalar-matrix products $c_T \mathbf{Q}_S$ and $c_S \mathbf{Q}_T$, and of the outer products of the vectors $\mathbf{s}_T$ and the transpose of $\mathbf{s}_S$ and also of $\mathbf{s}_S$ and the transpose of $\mathbf{s}_T$. Our approach shares the computation across the aggregates: The scalar aggregates are used to scale up the linear and quadratic aggregates, while the linear aggregates are used to compute the quadratic aggregates.

We now turn to incremental view maintenance. F-IVM operates over view trees. Whereas for non-incremental computation we only materialize the top view in the tree and the input relations, for incremental computation we may materialize additional views to speed up the maintenance task. Our approach is an instance of higher-order IVM, where an update to one relation may trigger the maintenance of several views.

Figure 1 shows the leaf-to-root maintenance paths under changes to S and T. For updates δS to S, each delta view δV_S, δV_{ST}, and δQ is computed using delta rules:

```
δV_S   := SELECT A, C, SUM(E) AS S_E
             FROM δS GROUP BY A, C;
δV_ST  := SELECT A, C, SUM(S_D * S_E) AS S_C
             FROM V_T NATURAL JOIN δV_S GROUP BY A, C;
δQ     := SELECT A, C, SUM(S_B * S_C)
             FROM V_R NATURAL JOIN δV_ST GROUP BY A, C;
```

An update may consist of both inserts and deletes, which are encoded as keys with positive and respectively negative payloads. For the count aggregate, the payload is 1 for an insert and -1 for a delete. For the compound aggregate, the payload is $(1, \mathbf{0}_{5 \times 1}, \mathbf{0}_{5 \times 5})$ for an insert and $(-1, \mathbf{0}_{5 \times 1}, \mathbf{0}_{5 \times 5})$ for a delete, where $\mathbf{0}_{n \times m}$ is the n-by-m matrix with all zero values.

F-IVM materializes and maintains views depending on the update workload. For updates to all input relations, it materializes each view in the view tree. For updates to R only, it materializes V_{ST}; for updates to S only, it materializes V_R

and V_T; for updates to T only, it materializes V_R and V_S. F-IVM takes constant time for updates to S and linear time for updates to R and T; these complexities are in the number of distinct keys in the views. In contrast, the first-order IVM computes one delta query per each updated relation and without the use of extra views. It takes linear time for updates to any of the three relations for our example query. The fully recursive higher-order IVM constructs a view tree for each delta query, so overall more views, including the view materializing the join of V_R, V_S, and δT.

F-IVM thus needs the same view tree and views for our query with one SUM aggregate and even for the learning task with the ten SUM aggregates. In contrast, the first-order IVM needs to compute a distinct delta query for each of these aggregates for updates to any of the three relations. DBToaster, which is the state-of-the-art fully recursive IVM, computes 31 views, ten top views and 21 auxiliary ones. Whereas F-IVM shares the computation across these aggregates, the other IVM approaches do not. This significantly widens the performance gap between F-IVM and its competitors.

2 Overview of the F-IVM system

Figure 2 overviews the main components of F-IVM, annotated with the numbers of sections where they are discussed. Applications (Sect. 8), e.g., database analytics, training linear regression model and Chow-Liu trees, and linear algebra computation, rely on queries with natural joins and group-by aggregates, where each aggregate is expressed using the sum and product operations in a ring. Queries and rings serve as input to F-IVM, together with a stream of updates (tuple inserts and deletes) to the underlying database. Section 3 details the data model, the query language supported by F-IVM, and the ring algebraic structure.

The logical optimizer creates a *variable order* for the input query (Sect. 4). This is akin to a query plan, albeit expressed as a partial order on the query variables as opposed to a partial order on the relations to join. Classical query evaluation uses query plans that dictate the order in which the relations are joined. F-IVM uses variable orders that dictate the order in which the variables are marginalized. For each join variable,
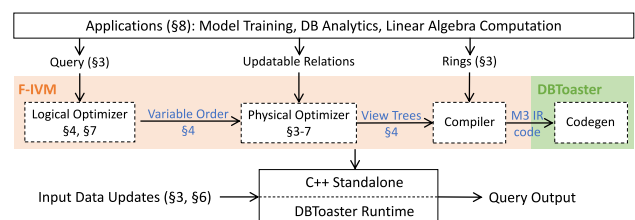

Fig. 2 Overview of the F-IVM system

all relations with that variable are joined. This choice is motivated by the observation that relation-at-a-time query plans is suboptimal in general, whereas the evaluation by variable orders is worst-case optimal [43].

Finding a good variable order is a hard problem. *Q-hierarchical* queries [9] admit asymptotically best variable orders that can be found efficiently (Sect. 7). This also applies to queries, which become q-hierarchical on databases that satisfy functional dependencies.

Given a variable order for a query, the physical optimizer creates a *view tree* (Sect. 4), which is a tree of views to support the maintenance and output enumeration of the query. Updates to base relations are propagated bottom-up in the tree, while output enumeration requires top-down access in the view tree. Depending on which base relations are updatable (dynamic) or non-updatable (static), F-IVM decides to materialize and maintain views in the view tree to support efficient propagation of the updates and avoid recomputation. Section 5 discusses the view materialization problem, whereas Sect. 6 discusses efficient update propagation.

Each view is accessed via indices with key-payload entries. Its primary index is a hash map over all its keys (Sect. 3). F-IVM may also need secondary and even tertiary indices, which are hash maps over different subsets of its keys. Such indices are updated lazily: the index updates are buffered and only executed when index access is required. The views for q-hierarchical queries require the primary indices to support updates that are propagated bottom-up in the view tree, and secondary indices to support output enumeration that proceeds top-down in the view tree (Sect. 7). F-IVM implements equality-based joins using in-memory hash-based join operators. Aggregation is performed using variable marginalization. To marginalize a variable, F-IVM enumerates the entries with the same key, except for the marginalized variable, and applies the aggregation on these entries on the fly.

For a view tree and ring specification for each variable to be marginalized, the compiler outputs code in DBToaster's intermediate representation language *M3*. DBToaster has its own optimizer and compiler that turns M3 code into highly optimized C++ code. This code takes the stream of input data updates, maintains the views, and enumerates the query output, relying on DBToaster's runtime library for data ingestion.

3 Data model and query language

The data model of F-IVM is based on relations over rings and its query language allows for natural joins and group-by aggregates over such relations.

Definition 1 A ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$ is a set $\mathbf{D}$ with two closed binary operations $+$ and $*$, the additive identity $\mathbf{0}$, and the multiplicative identity $\mathbf{1}$ such that for all $a, b, c \in \mathbf{D}$, the following axioms are satisfied:

1. $a + b = b + a$.
2. $(a + b) + c = a + (b + c)$.
3. $\mathbf{0} + a = a + \mathbf{0} = a$.
4. $\exists -a \in \mathbf{D} : a + (-a) = (-a) + a = \mathbf{0}$.
5. $(a * b) * c = a * (b * c)$.
6. $a * \mathbf{1} = \mathbf{1} * a = a$.
7. $a * (b + c) = a * b + a * c$ and $(a + b) * c = a * c + b * c$.

A semiring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$ satisfies all of the above properties except the additive inverse property (Property 4) and adds the axiom $\mathbf{0} * a = a * \mathbf{0} = \mathbf{0}$. A (semi)ring for which $a * b = b * a$ is commutative. □

Example 1 The number sets $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$, and $\mathbb{C}$ with arithmetic operations $+$ and $\cdot$ and numbers 0 and 1 form commutative rings. The set $\mathcal{M}$ of $(n \times n)$ matrices forms a non-commutative ring $(\mathcal{M}, \cdot, +, 0_{n,n}, I_n)$, where $0_{n,n}$ and I_n are the zero matrix and the identity matrix of size $(n \times n)$. The set $\mathbb{N}$ of natural numbers is a commutative semiring but not a ring because it has no additive inverse. Further examples are the max-product semiring $(\mathbb{R}_+, \max, \times, 0, 1)$, the Boolean semiring $(\{\text{true}, \text{false}\}, \vee, \wedge, \text{false}, \text{true})$, and the set semiring $(2^U, \cup, \cap, \emptyset, U)$ of all possible subsets of a given set U. □

Data. A schema $\mathcal{S}$ is a set of variables. Let $\mathsf{Dom}(X)$ denote the domain of a variable X. A tuple $\mathbf{t}$ over schema $\mathcal{S}$ has the domain $\mathsf{Dom}(\mathcal{S}) = \prod_{X \in \mathcal{S}} \mathsf{Dom}(X)$. The empty tuple () is the tuple over the empty schema.

Let $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$ be a ring. A *relation* R over schema $\mathcal{S}$ and the ring $\mathbf{D}$ is a function $\mathsf{R} : \mathsf{Dom}(\mathcal{S}) \rightarrow \mathbf{D}$ mapping tuples over schema $\mathcal{S}$ to values in $\mathbf{D}$ such that $\mathsf{R}[\mathbf{t}] \neq \mathbf{0}$ for finitely many tuples $\mathbf{t}$. The tuple $\mathbf{t}$ is called a *key*, while its mapping $\mathsf{R}[\mathbf{t}]$ is the *payload* of $\mathbf{t}$ in R. We use $\mathsf{sch}(\mathsf{R})$ to denote the schema of R. The statement $\mathbf{t} \in \mathsf{R}$ tests if $\mathsf{R}[\mathbf{t}] \neq \mathbf{0}$. The size $|\mathsf{R}|$ of R is the size of the set $\{\mathbf{t} \mid \mathbf{t} \in \mathsf{R}\}$, which consists of all keys with non-$\mathbf{0}$ payloads. A database $\mathcal{D}$ is a collection of relations over the same ring. Its size $|\mathcal{D}|$ is the sum of the sizes of its relations. This data model is in line with prior work on K-relations over provenance semirings [24], generalized multiset relations [31], and factors over semirings [3].

Each relation or materialized view R over schema $\mathcal{S}$ is implemented as a hash map or a multidimensional array that stores key-payload entries $(\mathbf{t}, \mathsf{R}[\mathbf{t}])$ for each tuple $\mathbf{t}$ with $\mathsf{R}[\mathbf{t}] \neq \mathbf{0}$. The data structure can: (1) look up, insert, and delete entries in amortized constant time, and (2) enumerate all stored entries in R with constant *delay*, i.e., the following times are constant: (i) the time between the start of the enumeration and outputting the first tuple, (ii) the time between outputting any two consecutive tuples, and (iii) the time

between outputting the last tuple and the end of the enumeration [19]. For a schema $\mathcal{X} \subset \mathcal{S}$, we use an index data structure that for any $\mathbf{t} \in \mathsf{Dom}(\mathcal{X})$ can: (4) enumerate all tuples in $\sigma_{\mathcal{X}=\mathbf{t}}\mathsf{R}$ with constant delay, (5) check $\mathbf{t} \in \pi_{\mathcal{X}}\mathsf{R}$ in amortized constant time; and (7) insert and delete index entries in amortized constant time.

We give a hash-based example data structure that supports the above operations with the stated complexities. Consider a relation R over schema $\mathcal{S}$. A hash table with chaining stores key value entries of the form $(\mathbf{t}, R(\mathbf{t}))$ for each tuple $\mathbf{t}$ over $\mathcal{S}$ with $R(\mathbf{t}) \neq \mathbf{0}$. The entries are doubly linked to support enumeration with constant delay. The hash table can report the number of its entries in constant time and supports lookups, inserts, and deletes in amortized constant time. To support index operations on a schema $\mathcal{X} \subset \mathcal{S}$, we create another hash table with chaining where each table entry stores an $\mathcal{X}$-value $\mathbf{t}$ as key and a doubly linked list of pointers to the entries in R having $\mathbf{t}$ as $\mathcal{X}$-value. Looking up an index entry given $\mathbf{t}$ takes amortized constant time, and its doubly linked list enables enumeration of the matching entries in R with constant delay. Inserting an index entry into the hash table additionally prepends a new pointer to the doubly linked list for a given $\mathbf{t}$; overall, this operation takes amortized constant time. For efficient deletion of index entries, each entry in R also stores back-pointers to its index entries (one back-pointer per index for R). When an entry is deleted from R, locating and deleting its index entries in doubly linked lists takes constant time per index.

Query Language. We consider queries with natural joins and group-by aggregates:

```
SELECT X₁,…,X_f , SUM(g_{f+1}(X_{f+1}) *…* g_m(X_m))
FROM  R₁ NATURAL JOIN … NATURAL JOIN R_n
GROUP BY  X₁,…,X_f
```

The group-by variables $X_1, \ldots, X_f$ are *free*, while the other variables $X_{f+1}, \ldots, X_m$ are *bound*. The SUM aggregate values are from a ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$. The SUM operator uses the addition $+$ from $\mathbf{D}$. Further aggregates can be expressed using the sum and product operations from the ring. A *lifting* function $g_k : \mathsf{Dom}(X_k) \to \mathbf{D}$, for $f < k \leq m$, maps X_k-values to elements in $\mathbf{D}$: when marginalizing X_k, we aggregate the values $g_k(x)$ from $\mathbf{D}$ and not the values x from $\mathsf{Dom}(X_k)$.

Instead of the verbose SQL notation, we use the following more compact encoding:

$$Q[X_1, \ldots, X_f] = \bigoplus_{X_{f+1}} \cdots \bigoplus_{X_m} \bigotimes_{i \in [n]} \mathsf{R_i}[\mathcal{S}_i]$$

where $\bigotimes$ is the join operator, $\bigoplus_{X_{f+1}}$ is the aggregation operator that marginalizes over the variable X_{f+1}, and each relation $\mathsf{R_i}$ is a function mapping keys over schema $\mathcal{S}_i$ to payloads in $\mathbf{D}$. We also need a union operator $\uplus$ to express updates (insert/delete) to relations.

Example 2 The SQL query

```
SELECT SUM(1) FROM R NATURAL JOIN S NATURAL
   JOIN T
```

over tables $R(A, B)$, $S(A, C, E)$, and $T(C, D)$ can be encoded as follows in our formalism. The table R is encoded as a relation $\mathsf{R} : \mathsf{Dom}(A) \times \mathsf{Dom}(B) \to \mathbb{Z}$ that maps tuples (a, b) to their multiplicity in R; similarly, we encode the tables S and T as relations S and T. We translate the SQL query into:

$$Q[\,] = \bigoplus_{A,B,C,D,E} \mathsf{R}[A, B] \otimes \mathsf{S}[A, C, E] \otimes \mathsf{T}[C, D]$$

where $\bigoplus_{A,B,C,D,E}$ abbreviates $\bigoplus_A \cdots \bigoplus_E$. The lifting functions used for marginalization map all values to 1. Recall that by definition R, S, and T are finite. The relation Q maps the empty tuple $()$ to the count. □

Given a ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$, relations R and S over schema $\mathcal{S}_1$ and relation T over schema $\mathcal{S}_2$, a variable $X \in \mathcal{S}_1$, and a lifting function $g_X : \mathsf{Dom}(X) \to \mathbf{D}$, we define the three operators as follows:

union:
$$\forall \mathbf{t} \in \mathsf{D}_1 : \ (\mathsf{R} \uplus \mathsf{S})[\mathbf{t}] = \mathsf{R}[\mathbf{t}] + \mathsf{S}[\mathbf{t}]$$

join:
$$\forall \mathbf{t} \in \mathsf{D}_2 : \ (\mathsf{S} \otimes \mathsf{T})[\mathbf{t}] = \mathsf{S}[\pi_{\mathcal{S}_1}(\mathbf{t})] * \mathsf{T}[\pi_{\mathcal{S}_2}(\mathbf{t})]$$

aggregation by marginalization:
$$\forall \mathbf{t} \in \mathsf{D}_3 : (\bigoplus_X \mathsf{R})[\mathbf{t}] = \sum \{ \mathsf{R}[\mathbf{t}_1] * g_X(\pi_{\{X\}}(\mathbf{t}_1)) \mid \\ \mathbf{t}_1 \in \mathsf{D}_1, \mathbf{t} = \pi_{\mathcal{S}_1\setminus\{X\}}(\mathbf{t}_1)\}$$

where $\mathsf{D}_1 = \mathsf{Dom}(\mathcal{S}_1)$, $\mathsf{D}_2 = \mathsf{Dom}(\mathcal{S}_1 \cup \mathcal{S}_2)$, and $\mathsf{D}_3 = \mathsf{Dom}(\mathcal{S}_1\setminus\{X\})$, and $\pi_{\mathcal{S}}(\mathbf{t})$ is a tuple representing the projection of tuple $\mathbf{t}$ on the schema $\mathcal{S}$.

Example 3 Consider relations over a ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$:

A	B	$\to \mathsf{R}[A, B]$
a_1	b_1	$\to r_1$
a_2	b_1	$\to r_2$

A	B	$\to [A, B]\mathsf{S}$
a_2	b_1	$\to s_1$
a_3	b_2	$\to s_2$

B	C	$\to [B, C]\mathsf{T}$
b_1	c_1	$\to t_1$
b_2	c_2	$\to t_2$

The values $r_1, r_2, s_1, s_2, t_1, t_2$ are non-$\mathbf{0}$ values from $\mathbf{D}$. The operators $\uplus, \otimes,$ and $\oplus$ are akin to union, join, and aggregation ($g_A : \mathsf{Dom}(A) \to \mathbf{D}$ is the lifting for A):

A	B	$\to (\mathsf{R} \uplus \mathsf{S})[A, B]$
a_1	b_1	$\to r_1$
a_2	b_1	$\to r_2 + s_1$
a_3	b_2	$\to s_2$

A	B	C	$\to ((\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T})[A, B, C]$
a_1	b_1	c_1	$\to r_1 * t_1$
a_2	b_1	c_1	$\to (r_2 + s_1) * t_1$
a_3	b_2	c_2	$\to s_2 * t_2$

B	C	$\to (\bigoplus_A (\mathsf{R} \uplus \mathsf{S}) \otimes \mathsf{T})[B, C]$
b_1	c_1	$\to r_1 * t_1 * g_A(a_1) + (r_2 + s_1) * t_1 * g_A(a_2)$
b_2	c_2	$\to s_2 * t_2 * g_A(a_3)$

Example 4 Let us consider the SQL query from Sect. 1.1, which computes SUM(R.B * T.D * S.E) grouped by A, C. Assume that B, D, and E take values from $\mathbb{Z}$. We model the tables R, S, and T as relations mapping tuples to their multiplicity, as in Ex. 2. The variables A and C are free, while B, D, and E are bound.

When marginalizing over the bound variables, we apply the same lifting function to these variables: $\forall x \in \mathbb{Z}$: $g_B(x) = g_D(x) = g_E(x) = x$. The SQL query can be expressed in our formalism as follows:

$$Q[A, C] = \bigoplus_{B, D, E} R[A, B] \otimes S[A, C, E] \otimes T[C, D]$$

The computation of the aggregate SUM(R.B * T.D * S.E) now happens over payloads. □

By using relations over rings, we avoid the intricacies of incremental computation under multiset semantics caused by the non-commutativity of inserts and deletes. We simplify delta processing by representing both inserts and deletes as tuples, with the distinction that they map to positive and respectively negative ring values. This uniform treatment allows for simple delta rules for the three operators of our query language.

4 Factorized ring computation

This section introduces a framework for query evaluation based on factorized computation and data rings. The next section extends it to incremental maintenance.

Variable Orders. Given a join query Q, a variable X *depends* on a variable Y if both are in the schema of a relation in Q.

Definition 2 (adapted from [52]) A *variable order* ω for a join query Q is a pair (F, dep), where F is a rooted forest with one node per variable in Q, and dep is a function mapping each variable X to a set of variables in F. It satisfies the following constraints:

– For each relation in Q, all of its variables lie along a root-to-leaf path in F.
– For each variable X, $dep(X)$ is the subset of its ancestors in F on which the variables in the subtree rooted at X depend.

Example 5 Consider the query from Ex. 2 that joins the relations R[A, B], S[A, C, E], and T[C, D]. Figure 3 gives a variable order (top left) for the query. Variable D has ancestors A and C, yet it only depends on C since C and D appear in the same relation T and D does not occur in any relation together with A. Thus, $dep(D) = \{C\}$. Given C, the variables D and E are independent of each other. □
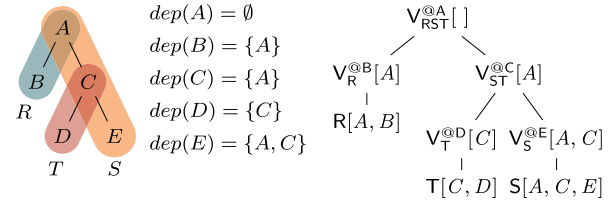
$$V_S^{@E}[A, C] = \bigoplus_E S[A, C, E]$$
$$V_T^{@D}[C] = \bigoplus_D T[C, D] \qquad V_{RST}^{@A}[\,] = \bigoplus_A (V_R^{@B}[A] \otimes V_{ST}^{@C}[A])$$
$$V_R^{@B}[A] = \bigoplus_B R[A, B] \qquad V_{ST}^{@C}[A] = \bigoplus_C (V_T^{@D}[C] \otimes V_S^{@E}[A, C])$$

Fig. 3 (Top left) variable order ω of the natural join of the relations R[A, B], S[A, C, E], and T[C, D]; (top right) View tree over ω and $\mathcal{F} = \emptyset$; (bottom) View definitions
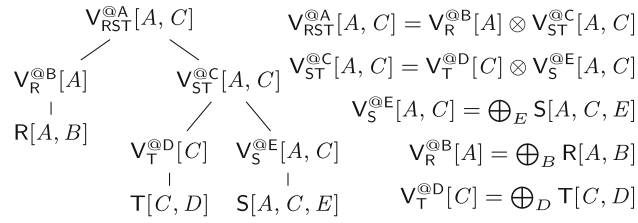

Fig. 4 Creating a view tree $\tau(\omega, \mathcal{F})$ for a variable order ω and a set of free variables $\mathcal{F}$

For a query Q with free variables, a variable order is *free-top* if no bound variable is an ancestor of a free variable [28]. Variable orders are a different syntax [52] for hypertree decompositions [23]. They are more natural for algorithms that proceed one variable at a time.

View Trees. Our framework relies on a variable order ω for the input query Q to describe the structure of the computation and indicate which variable marginalizations are pushed past joins. Based on ω, we construct a tree of views that represent F-IVM's data structure to support query maintenance and enumeration.

Figure 4 gives a function τ that constructs a view tree τ for a variable order ω and the set $\mathcal{F}$ of free variables of the query Q. Without loss of generality, we assume that ω is a single rooted tree. Otherwise, we apply τ to each tree in ω to obtain a set of view trees. For simplicity, we assume that ω was first extended with relations as children under their lowest variable.

$$V_{RST}^{@A}[A, C]$$

$$V_R^{@B}[A] \quad V_{ST}^{@C}[A, C]$$

$$R[A, B] \quad V_T^{@D}[C] \quad V_S^{@E}[A, C]$$

$$T[C, D] \quad S[A, C, E]$$

$$V_{RST}^{@A}[A, C] = V_R^{@B}[A] \otimes V_{ST}^{@C}[A, C]$$
$$V_{ST}^{@C}[A, C] = V_T^{@D}[C] \otimes V_S^{@E}[A, C]$$
$$V_S^{@E}[A, C] = \bigoplus_E S[A, C, E]$$
$$V_R^{@B}[A] = \bigoplus_B R[A, B]$$
$$V_T^{@D}[C] = \bigoplus_D T[C, D]$$

Fig. 5 (Left) view tree over the variable order ω in Fig. 3 and $\mathcal{F} = \{A, C\}$; (right) View definitions

A	B	$\to$ R$[A, B]$
a_1	b_1	$\to p_1$
a_1	b_2	$\to p_2$
a_2	b_3	$\to p_3$
a_3	b_4	$\to p_4$

A	$\to V_R^{@B}[A]$
a_1	$\to p_1 * g_B(b_1) + p_2 * g_B(b_2)$
a_2	$\to p_3 * g_B(b_3)$
a_3	$\to p_4 * g_B(b_4)$

A	C	E	$\to$ S$[A, C, E]$
a_1	c_1	e_1	$\to p_5$
a_1	c_1	e_2	$\to p_6$
a_1	c_2	e_3	$\to p_7$
a_2	c_2	e_4	$\to p_8$

A	C	$\to V_S^{@E}[A, C]$
a_1	c_1	$\to p_5 * g_E(e_1) + p_6 * g_E(e_2)$
a_1	c_2	$\to p_7 * g_E(e_3)$
a_2	c_2	$\to p_8 * g_E(e_4)$

C	D	$\to$ T$[C, D]$
c_1	d_1	$\to p_9$
c_2	d_2	$\to p_{10}$
c_2	d_3	$\to p_{11}$
c_3	d_4	$\to p_{12}$

C	$\to V_T^{@D}[C]$
c_1	$\to p_9 * g_D(d_1)$
c_2	$\to p_{10} * g_D(d_2) + p_{11} * g_D(d_3)$
c_3	$\to p_{12} * g_D(d_4)$

A	$\to V_{ST}^{@C}[A]$
a_1	$\to V_S^{@E}[a_1, c_1] * V_T^{@D}[c_1] * g_C(c_1) + V_S^{@E}[a_1, c_2] * V_T^{@D}[c_2] * g_C(c_2)$
a_2	$\to V_S^{@E}[a_2, c_2] * V_T^{@D}[c_2] * g_C(c_2)$

$() \to V_{RST}^{@A}[\,]$
$() \to V_{ST}^{@C}[a_1] * V_R^{@B}[a_1] * g_A(a_1) + V_{ST}^{@C}[a_2] * V_R^{@B}[a_2] * g_A(a_2)$

Fig. 6 Contents of the views in the view tree from Fig. 3 in case the relations R, S, and T are over a ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$ with $p_i \in \mathbf{D}$ for $i \in [12]$

The function τ maps the variable order to a view tree of the same tree structure, yet with each variable X replaced by a view $V_{rels}^{@X}[keys]$. This notation states that the view V is (recursively) defined over the input relations rels, has free variables $keys$, and it corresponds to the variable X in ω; in case of a view for an input relation R, we use the simplified notation R[sch(R)].

The base case (leaf in the extended variable order) is that of an input relation: We construct a view that is the relation itself. At a variable X (inner node), we distinguish two cases: If X is a bound variable, then we construct a view that marginalizes out X in the natural join of the views that are children of the current view; we thus first join on X, then apply the lifting function for X on its values, and aggregate X away. If X is a free variable, however, then we retain it in the view schema without applying the lifting function to its values. The schema of the view consists of $dep(X)$ and the free variables in the subtree of ω rooted at X.

Example 6 Figure 3 shows the view tree constructed by the function τ from Fig. 4 over the variable order ω and the empty set of free variables. Figure 5 depicts the view tree constructed over the same variable order but for the set $\mathcal{F} = \{A, C\}$ of free variables.

Figure 6 gives the contents of the views in the view tree from Fig. 3, where R, S, and T are relations over a ring $\mathbf{D}$ with payloads $p_i \in \mathbf{D}$ for $i \in [12]$. Assume that $\mathbf{D}$ is the $\mathbb{Z}$ ring, each tuple in these relations is mapped to 1, i.e., $p_i = 1$ for $i \in [12]$, and the lifting functions map all values to 1. Then, the view tree computes the COUNT query from Ex. 2 and the root view $V_{RST}^{@A}$ maps the empty tuple to the overall count 10. □

By default, the function τ in Fig. 4 constructs one view per variable in the variable order ω. A wide relation (with many variables) 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 Sect. 9. Such long branches create long chains of views, where each view marginalizes one bound variable over its child view in the chain. For practical reasons, we compose such long chains into a single view that marginalizes several variables at a time.

5 Factorized higher-order IVM

We introduce incremental view maintenance in our factorized ring computation framework. Unlike evaluation, the incremental maintenance of the query result may require the materialization and maintenance of views. An update to a relation R triggers changes in all views from the leaf R to the root of the view tree.

Updates. The insertion (deletion) of a tuple $\mathbf{t}$ into (from) a relation R is expressed as a delta relation δR that maps $\mathbf{t}$ to $\mathbf{1}$ (and respectively $-\mathbf{1}$). In general, δR can be a relation, thus a collection of tuples mapped to payloads. The updated relation is then the union of the old relation and the delta relation: R $:=$ R $\uplus$ δR.

Delta Views. For each view V affected by an update, a *delta view* δV defines the change in the view content. In case the view V represents a relation R, then δV $=$ δR if there are updates to R and δV $=$ $\emptyset$ otherwise. If the view is defined using operators on other views, δV is derived using the following delta rules:

$$\delta(V_1 \uplus V_2) = \delta V_1 \uplus \delta V_2$$
$$\delta(V_1 \otimes V_2) = (\delta V_1 \otimes V_2) \uplus (V_1 \otimes \delta V_2) \uplus (\delta V_1 \otimes \delta V_2)$$
$$\delta(\bigoplus_X V) = \bigoplus_X \delta V$$

The correctness of the rules follows from the associativity of $\uplus$ and the distributivity of $\otimes$ over $\uplus$; $\bigoplus_X$ is equivalent to the repeated application of $\uplus$ for the possible values of X. The derived delta views are subject to standard simplifications: If
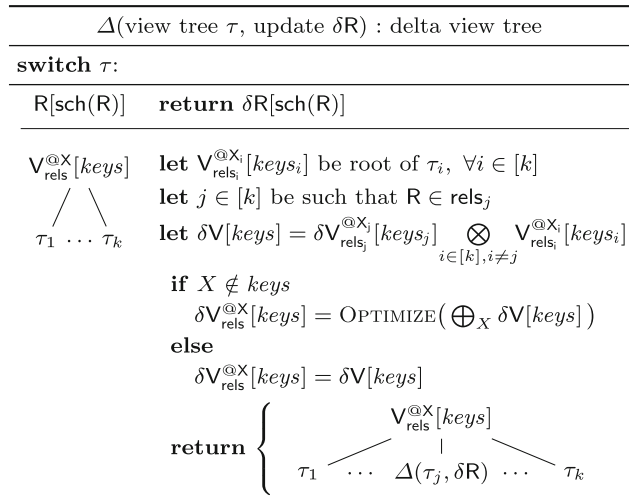
Fig. 7 Creating a delta view tree $\Delta(\tau, \delta R)$ for a view tree τ to process an update δR to relation R

V is not defined over the updated relation R, then its delta view δV is empty, and then we propagate this information using the identities $\emptyset \uplus V = V \uplus \emptyset = V$ and $\emptyset \otimes V = V \otimes \emptyset = \emptyset$.

Delta Trees. Under updates to one relation, a view tree becomes a delta tree where the affected views become delta views. The function Δ in Fig. 7 replaces the views along the path from the updated relation to the root with delta views. The OPTIMIZE method rewrites delta view expressions to exploit factorized updates by avoiding the materialization of Cartesian products and pushing marginalization past joins (see Sect. 6).

Example 7 Consider again the query from Ex. 2, its view tree in Fig. 3, and the same relations over the $\mathbb{Z}$ ring and the lifting functions that map all values to 1 as in Ex. 6. An update $\delta T[C, D] = \{(c_1, d_1) \rightarrow -1, (c_2, d_2) \rightarrow 3\}$ triggers delta computation at each view from the leaf T to the root of the view tree:

$$\delta V_T^{@D}[C] = \bigoplus_D \delta T[C, D]$$
$$\delta V_{ST}^{@C}[A] = \bigoplus_C \delta V_T^{@D}[C] \otimes V_S^{@E}[A, C]$$
$$\delta V_{RST}^{@A}[\,] = \bigoplus_A V_R^{@B}[A] \otimes \delta V_{ST}^{@C}[A]$$

Given that $V_S^{@E} = \{(a_1, c_1) \rightarrow 2, (a_1, c_2) \rightarrow 1, (a_2, c_2) \rightarrow 1\}$ and $V_R^{@B} = \{a_1 \rightarrow 2, a_2 \rightarrow 1, a_3 \rightarrow 1\}$, we obtain $\delta V_T^{@D}[C] = \{c_1 \rightarrow -1, c_2 \rightarrow 3\}$, $\delta V_{ST}^{@C}[A] = \{a_1 \rightarrow 1, a_2 \rightarrow 3\}$, and $\delta V_{RST}^{@A} = \{() \rightarrow 5\}$.

A single-tuple update to T fixes the values for C and D. Computing $\delta V_T^{@D}$ then takes constant time. The delta view $\delta V_{ST}^{@C}$ iterates over all possible A-values for a fixed C-value, which takes linear time; $\delta V_{RST}^{@A}$ incurs the same linear-time cost. A single-tuple update to R or S fixes all variables on a leaf-to-root path in the delta view tree, giving a constant view maintenance cost. □
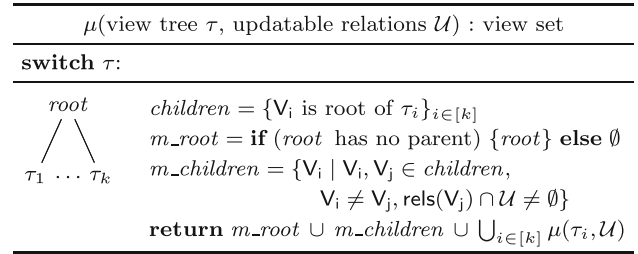
Fig. 8 Deciding which views in a view tree τ to materialize in order to support updates to a set of relations $\mathcal{U}$. The notation $rels(V_j)$ denotes the relations under the view V_j in τ

In contrast to classical (first-order) IVM that only requires maintenance of the query result [15], our approach is higher-order IVM as updates may trigger maintenance of several interrelated views. The fully recursive IVM scheme of DBToaster [31, 32] creates one materialization hierarchy per relation in the query, whereas we use one view tree for all relations. This view tree relies on variable orders to decompose the query into views and factorize its computation and maintenance.

Which Views to Materialize and Maintain? The answer to this question depends on which relations may change. The set of the updatable relations determines the possible delta propagation paths in a view tree, and these paths may use materialized views.

Propagating changes along a leaf-to-root path is computationally most effective if each delta view joins with sibling views that are already materialized. Figure 8 gives an algorithm that reflects this idea: Given a view tree τ and a set of updatable relations $\mathcal{U}$, the algorithm traverses the tree top-down to discover the views that need to be materialized. The root of the view tree τ is always stored as it represents the query result. Every other view V_i is stored only if there exists a sibling view V_j defined over an updatable relation.

Example 8 We continue with our query from Ex. 7. For updates to T only, i.e., $\mathcal{U} = \{T\}$, we store the root $V_{RST}^{@A}$ and the views $V_S^{@E}$ and $V_R^{@B}$ used to compute the deltas $\delta V_{ST}^{@C}$ and $\delta V_{RST}^{@A}$. Only the root view is affected: $V_{RST}^{@A}[\,] = V_{RST}^{@A}[\,] \uplus \delta V_{RST}^{@A}[\,]$. It is not necessary to maintain other views. To also support updates to R and S, we need to materialize $V_{ST}^{@C}$ and $V_T^{@D}$. If no updates are supported, then only the root view is stored. □

For queries with free variables, several views in their (delta) view trees may be identical: This can happen when all variables in their keys are free and thus cannot be marginalized. For instance, a variable order ω for the query from Ex. 4 may have the variables A and C above all other variables, in which case their views are the same in the view tree for ω. We then store only the top view out of these identical views.

IVM Triggers. For each updatable relation R, F-IVM constructs a trigger procedure that takes as input an update δR and

implements the maintenance schema of the corresponding delta view tree. This procedure also maintains all materialized views needed for the given update workload.

A bulk of updates to several relations is handled as a sequence of updates, one per relation. Update sequences can also happen when updating a relation R that occurs several times in the query. The instances representing the same relation are at different leaves in the delta tree and lead to changes along multiple leaf-to-root paths.

6 Factorizable updates

Our focus so far has been on supporting updates represented by delta relations. We next consider an alternative approach that decomposes a delta relation into a union of factorizable relations. The cumulative size of the decomposed relations can be much less than the size of the original delta relation. Also, the complexity of propagating a factorized update can be much lower than that of its unfactorized (listing) representation, since the factorization makes explicit the independence between query variables and enables optimizations of delta propagation such as pushing marginalization past joins. Besides the factorized view computation, this is the second instance where F-IVM exploits factorization.

Factorizable updates arise in many domains such as linear algebra and machine learning. Section 8 demonstrates how our framework can be used for the incremental evaluation of matrix chain multiplication, recovering prior work on this [44]. Matrix chain computation can be phrased in our language of joins and aggregates, where matrices are binary relations. Changes to one row/column in an input matrix may be expressed as a product of two vectors. In general, an arbitrary update matrix can be decomposed into a sum of rank-1 matrices, each of them expressible as products of vectors, using low-rank tensor decomposition methods [33, 61].

Example 9 Arbitrary relations can be decomposed into a union of factorizable relations. The relation $R[A, B] = \{(a_i, b_j) \to 1 \mid i \in [n], j \in [m]\}$ can be decomposed as $R_1[A] \otimes R_2[B]$, where $R_1[A] = \{(a_i) \to 1 \mid i \in [n]\}$ and $R_2[B] = \{(b_j) \to 1 \mid j \in [m]\}$. We thus reduced a relation of size nm to two relations of cumulative size $n + m$. If R were a delta relation, the delta views on top of it would now be expressed over $R_1[A] \otimes R_2[B]$ and their computation can be factorized as done for queries in Sect. 4. Product decomposition of relations can be done in linearithmic time in both the number of variables and the size of the relation [49].

Consider now $R'[A, B] = R[A, B] \uplus \{(a_{n+1}, b_j) \to 1 \mid j \in [m-1]\}$ with R as above. We can decompose each of the two terms in R' similarly to R, yielding overall $n + 2m$ values instead of $nm + m - 1$. A different decomposition with $n + m + 3$ values is given by a factorizable over-approximation

of R' compensated by a small product with negative payload: $\{(a_i) \to 1 \mid i \in [n+1]\} \otimes \{(b_j) \to 1 \mid j \in [m]\} \uplus \{(a_{n+1}) \to 1\} \otimes \{(b_m) \to -1\}$. □

The OPTIMIZE method used in the delta view tree algorithm in Fig. 7 exploits the distributivity of join $\otimes$ over marginalization $\bigoplus_X$ to push the latter past the former and down to the views with variable X. This optimization is reminiscent of pushing aggregates past joins in databases and variable elimination in probabilistic graphical models [3]. In case the delta views express Cartesian products, then they are not materialized but instead kept factorized.

Example 10 Consider the query Q from Example 7 and its view tree in Fig. 3. In the delta view tree derived for updates to S, the top-level delta is computed as:

$$\delta V_{RST}^{@A}[\,] = \bigoplus_A V_R^{@B}[A] \otimes \big(\underbrace{\bigoplus_C V_T^{@D}[C] \otimes \underbrace{\bigoplus_E \delta S[A, C, E]}_{\delta V_S^{@E}[A,C]}}_{\delta V_{ST}^{@C}[A]} \big)$$

A single-tuple update δS binds variables A, C, and E, and computing $\delta V_{RST}^{@A}$ requires $\mathcal{O}(1)$ lookups in $V_T^{@D}$ and $V_R^{@B}$. An arbitrary-sized update δS can then be processed in $\mathcal{O}(|\delta S|)$ time.

Assume now that δS is factorizable as $\delta S[A, C, E] = \delta S_A[A] \otimes \delta S_C[C] \otimes \delta S_E[E]$. In the construction of the delta view tree, the OPTIMIZE method exploits this factorization to push the marginalization past joins at each variable; for example, the delta at E becomes:

$$\delta V_S^{@E}[A, C] = \bigoplus_E \delta S_A[A] \otimes \delta S_C[C] \otimes \delta S_E[E]$$
$$= \delta S_A[A] \otimes \delta S_C[C] \otimes \bigoplus_E \delta S_E[E]$$

We also transform the top-level delta into a product of three views:

$$\delta V_{RST}^{@A}[\,] = \big(\bigoplus_A V_R^{@B}[A] \otimes \delta S_A[A] \big) \otimes$$
$$\big(\bigoplus_C V_T^{@D}[C] \otimes \delta S_C[C] \big) \otimes \big(\bigoplus_E \delta S_E[E] \big)$$

The computation time for this delta is proportional to the sizes of the three views representing the update: $\mathcal{O}(\min(|V_R^{@B}|, |\delta S_A|) + \min(|V_T^{@D}|, |\delta S_C|) + |\delta S_E|)$. □

7 F-IVM for special query classes

This section shows how F-IVM maintains *free-connex (α-)acyclic* queries [27] and *q-hierarchical* queries [9]. The analysis for these queries is refined into: (i) the preprocessing phase, where the view tree is constructed; (ii) the enumeration

phase, where we present the query result one tuple at a time; and (iii) the update phase, where we update the view tree. The following data complexity[2] claims assume that the ring operations require constant time; otherwise, the complexity results stated in this section have an extra multiplying factor to account for the complexity of the ring operations.

Theorem 1 *Let a query* Q *and a database of size* N.

F-IVM can maintain Q *with* $O(N)$ *preprocessing,* $O(1)$ *enumeration delay, and* $O(N)$ *single-tuple update in case* Q *is free-connex acyclic.*

F-IVM can maintain Q *with* $O(N)$ *preprocessing,* $O(1)$ *enumeration delay, and* $O(1)$ *single-tuple update in case* Q *is q-hierarchical.*

Remark. F-IVM has a special treatment for cyclic queries. Whereas for acyclic join queries the size of each view is asymptotically upper-bounded by the size of the query result, for cyclic queries views may be larger than the query result. In prior work [46], we show how to reduce the size of intermediate views for cyclic queries by adding indicator projections [3] to view trees. Such projections do not affect the query result but can constrain views (e.g., create cycles) and bring asymptotic savings in space and time. To decide which indicator projections to use, we apply a variant of the GYO reduction [7] that discovers cyclic parts in the query. □

7.1 Free-connex acyclic queries

We first introduce the class of free-connex acyclic queries and then explain how F-IVM maintains them.

Definition 3 ([10, 65]) A *join tree* for a query is a tree, where each node is a relation and if any two nodes have variables in common, then all nodes along the path between them also have these variables.

A query is *(α-)acyclic* if it admits a join tree. A query is *free-connex acyclic* if it is acyclic and remains acyclic after adding a new relation whose schema consists of the free variables of the query.

Example 11 Consider the query $Q[A, B, C] = \bigoplus_D \bigoplus_E$ $R[A, B] \otimes S[A, C, E] \otimes T[C, D]$. A possible join tree for Q is $R[A, B] - S[A, C, E] - T[C, D]$, where "$-$" denotes the parent–child relationship. Hence, Q is acyclic.

Consider the triangle query $Q_\triangle[\,] = \bigoplus_A \bigoplus_B \bigoplus_C$ $R[A, B] \otimes S[B, C] \otimes T[A, C]$. A possible tree built from the relations of $Q_\triangle$ is $R[A, B] - S[B, C] - T[A, C]$. The variable A occurs in the first and last relations but not in the middle relation; thus, this tree is not a join tree for $Q_\triangle$. One can show that any tree built from the relations of $Q_\triangle$ is not a join tree. Hence, $Q_\triangle$ is not acyclic.

The tree $R[A, B] - U[A, B, C] - S[A, C, E] - T[C, D]$ is a join tree of Q extended with the relation U whose schema consists of the free variables of Q. Hence, Q is free-connex acyclic. Consider now the variant Q' of Q where only the variables B and C are free. Adding a fresh relation U' with schema (B, C) to Q' turns it into a cyclic query Q'' that does not admit a join tree. □

We next detail how F-IVM achieves the complexity from Theorem 1 for a free-connex acyclic query Q.

Preprocessing. In the preprocessing phase, we create a view tree that compactly represent the result of Q. Given a variable order, the function τ in Fig. 4 constructs a view tree where the root view consists of all tuples over the free variables. While this view allows for constant enumeration delay, it may require superlinear computation and maintenance time as the free variables may originate from different input relations. We would like to avoid this super-linearity.

To keep the preprocessing and update times linear, we proceed as follows. We construct view trees such that the query result is kept and maintained factorized over several views at the top of the view tree. This approach still allows for constant enumeration delay, using a known enumeration approach for factorized representations [52]. We construct the view tree following a free-top variable order of the query Q and materialize a view over the schema $\{X\} \cup dep(X)$ for each variable X in the variable order. A key insight is that every free-connex acyclic query admits a free-top variable order where for each variable X, the set $\{X\} \cup dep(X)$ is covered by the variables of a single relation [8]. This ensures linear preprocessing and maintenance time for all views in view trees following such variable orders.

The function ν in Fig. 9 constructs a view tree for a given free-top variable order of a free-connex query. If a variable X has at least two children, it proceeds as follows. It creates at X a view $H_{rels}^{@X}$ with schema $\{X\} \cup dep(X)$ that joins the child views of X. If X has at least one sibling, it additionally creates a view $V_{rels}^{@X}$ on top of $H_{rels}^{@X}$ obtained from $H_{rels}^{@X}$ by marginalizing X. The first view enables efficient enumeration of X-values in the query result given a value tuple over $dep(X)$; the second view enables efficient updates coming from the subtrees rooted at siblings of X. If X has only one child, the creation of the view $H_{rels}^{@X}$ is not needed for efficient enumeration. In this case, the function creates a view $V_{rels}^{@X}$ marginalizing X in the child view if X has siblings.

Example 12 Consider the free-connex acyclic query Q from Ex. 11. Figure 3 gives a free-top variable order ω for Q. Figure 10 (left) depicts the view tree $\nu(\omega)$. The view $H_{ST}^{@C}$ can be computed by iterating over the (A, C)-tuples in $V_S^{@E}$ and multiplying the payload of each such tuple with the payload of the matching C-value in $V_T^{@D}$. Since each such (A, C)-tuple must be in S, we need to iterate over only linearly many such tuples. Similarly, the view $H_{RST}^{@A}$ can be computed by iter-

---

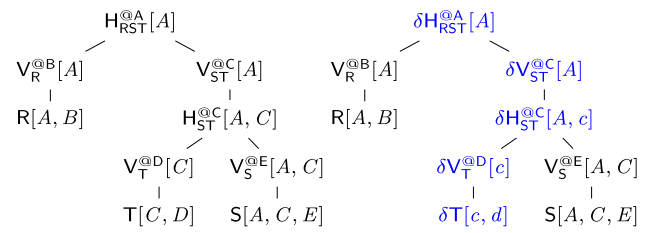[2] The *data complexity* is a function of the database size.

🖄 Springer

ν (free-top variable order ω) : view tree
switch ω:
R $\qquad$ **return** $\mathsf{R}[\mathsf{sch}(\mathsf{R})]$

For the X / $\omega_1 \cdots \omega_k$ branch:

let $T_i = \tau(\omega_i), \ \forall i \in [k]$
let $\mathsf{V}^{@\omega_i}_{\mathsf{rels}_i}[keys_i] = $ root of $T_i, \ \forall i \in [k]$
let $keys = \{X\} \cup dep(X)$
let $\mathsf{rels} = \bigcup_{i \in [k]} \mathsf{rels}_i$
let $\mathsf{H}^{@X}_{\mathsf{rels}}[keys] = \bigotimes_{i \in [k]} \mathsf{V}^{@\omega_i}_{\mathsf{rels}_i}[keys_i]$
let $\mathsf{V}^{@X}_{\mathsf{rels}}[keys \setminus \{X\}] = \bigoplus_X \mathsf{H}^{@X}_{\mathsf{rels}}[keys]$
if X has more than one child ($k \geq 2$)
$\quad$**if** X has no sibling
$\quad\quad$**return** $\begin{cases} \mathsf{H}^{@X}_{\mathsf{rels}}[keys] \\ \quad / \quad \backslash \\ T_1 \ \cdots \ T_k \end{cases}$
$\quad$**else**
$\quad\quad$**return** $\begin{cases} \mathsf{V}^{@X}_{\mathsf{rels}}[keys \setminus \{X\}] \\ \quad | \\ \mathsf{H}^{@X}_{\mathsf{rels}}[keys] \\ \quad / \quad \backslash \\ T_1 \ \cdots \ T_k \end{cases}$
else
$\quad$**if** X has no sibling
$\quad\quad$**return** T_1
$\quad$**else**
$\quad\quad$**return** $\begin{cases} \mathsf{V}^{@X}_{\mathsf{rels}}[keys \setminus \{X\}] \\ \quad | \\ T_1 \end{cases}$

Fig. 9 Creating a view tree for a free-top variable order

Fig. 10 (Left) view tree constructed by the function ν in Fig. 9 for the variable order ω in Fig. 3; (right) Delta view tree for a single-tuple update to T

ating over the A-values in one of the child views and doing lookups in the other child view to retrieve the payloads. For the computation of both views $\mathsf{H}^{@C}_{ST}$ and $\mathsf{H}^{@A}_{RST}$, we iterate over linearly many tuples and do a constant-time lookup for each such tuple. All other views are obtained by marginalizing one variable from their child views. Hence, all views can be computed in linear time. $\qquad\square$

Updates. The construction of delta view trees under single-tuple updates is exactly as described by the function Δ in Fig. 7 (Sect. 5). Since the view trees can be constructed in linear time, the delta view trees can also be constructed in linear time.

Example 13 Continuing Ex. 12, we consider a single-tuple update $\delta\mathsf{T}[c, d]$ to relation T. Figure 10 depicts the original view tree (left) and the delta view tree for updates to T (right). The difference is that along the path from T to the root, we now have delta views. The delta view $\delta\mathsf{V}^{@D}_T$ results from $\delta\mathsf{T}[c, d]$ by marginalizing D, which takes constant time since D is fixed to the constant d. To compute $\delta\mathsf{H}^{@C}_{ST}$, we iterate over all A-values paired with c in $\mathsf{V}^{@E}_S$. This operation takes

linear time with the support of an index on variable C built for this view. We obtain $\delta\mathsf{V}^{@C}_{ST}$ from $\delta\mathsf{H}^{@C}_{ST}$ by marginalizing the variable C. This requires constant time because C is fixed to the constant c. The top delta view $\delta\mathsf{H}^{@A}_{RST}$ is obtained by intersecting the two child views, e.g., by iterating over $\delta\mathsf{V}^{@C}_{ST}$ and doing lookups in $\mathsf{V}^{@B}_R$. This requires linear time. We conclude that the delta views can be computed in linear time. $\qquad\square$

Enumeration. Consider a view tree τ constructed using the function ν from Fig. 9 for a free-top variable order of a query Q. We first describe how to enumerate with constant delay the distinct tuples in the result of Q using τ. Then, we explain how to compute the payload of each result tuple in constant time.

Let $X_1, \ldots, X_n$ be an ordering of the free variables of the query that is compatible with a top-down traversal of the free-top variable order. We use the views $\mathsf{V}_1, \ldots, \mathsf{V}_n$ to enumerate the distinct tuples in the result of Q, where V_j is $\mathsf{H}^{@X_j}_{\mathsf{rels}}$ if X_j has at least two children and it is the child view of X_j otherwise. We retrieve from V_1 the first X_1-value in the result. When we arrive at a view V_j with $j > 1$, we have already fixed the values of the variables above X_j in the variable order. We retrieve from V_j the first X_j-value paired with these values. Once the values over all free variables are fixed, we have a complete result tuple that we output. Then, we iterate over the remaining distinct X_n-values in V_n paired with the fixed values over the ancestor variables of X_n and output a new tuple for each such value. After all X_n-values are exhausted, we backtrack, i.e., we move to the next X_{n-1}-value and restart the iteration of the matching X_n-values, and so on.

Given a complete tuple **t** constructed from the view tree τ, we use the function *payload* from Fig. 11 to compute its payload. The function first checks whether the schema of the root view is exactly the schema $\mathsf{sch}(\mathbf{t})$ of **t**. If so, it returns the payload of **t** in this view. Otherwise, the root view covers only a subset of the schema of the tuple. In this case, the function recursively computes the payload for each subtree τ_i of the root view and the projection of **t** onto the variables in τ_i. The final payload is the product of the payloads returned for the subtrees. The returned payloads are from the lowest views in the view tree whose schemas consist of free variables only.

$payload$(view tree τ, tuple $\mathbf{t}$): payload
switch τ:
R $\qquad$ **return** $R(\mathbf{t})$
$V[\mathcal{X}]$ $\qquad$ **if** $\mathcal{X} = \text{sch}(\mathbf{t})$
$\diagup\ \diagdown$ $\qquad$ **return** $V(\mathbf{t})$
$\tau_1 \cdots \tau_k$ $\quad$ **else** // $\mathcal{X} \subset \text{sch}(\mathbf{t})$
$\qquad$ **let** $\mathcal{V}_i$ = variables in τ_i, $\ \forall i \in [k]$
$\qquad$ **return** $\prod_{i \in [k]} payload(\tau_i, \pi_{\mathcal{V}_i} \mathbf{t})$

Fig. 11 Computing the payload of a tuple from a view tree

If all variables are free, then these lowest views are the input relations themselves.

Remark 1 The enumeration procedure needs the payloads of the lowest views whose schemas consist of free variables. The payloads from the views above these views thus need not be maintained, beyond keeping track of the multiplicities of each of their tuples. The maintenance of multiplicities is important for correctness, as it tells whether a tuple is to be removed from a view or still has at least one possible derivation from the input. For expensive payloads, such as those introduced in Sect. 8, it is therefore more efficient to only maintain them for the views from the input relations up to the views used to compute the payloads. Their ancestor views only need maintenance of tuple multiplicities. □

Example 14 We enumerate the distinct result tuples of the query Q[A, B, C] from Ex. 11 using the view tree in Fig. 10 (left). We iterate with constant delay over the A-values in $H_{RST}^{@A}[A]$. For each such A-value a, we iterate with constant delay over the B-values in $R[a, B]$ and over the C-values in $H_{ST}^{@C}[a, C]$. Each triple (a, b, c) obtained in this way is a result tuple of Q. Its payload is $R[a, b] \cdot H_{ST}^{@C}[a, c]$. □

Remark 2 To efficiently support enumeration and updates, we may need several indices for the views in a view tree for a free-connex acyclic query. Each view (and input relation) in the view tree in Fig. 10 (left) needs an index that can retrieve the payload for a given tuple of values over its variables. This is a primary index. For (top-down) enumeration, we may also need a secondary index per view to lookup for tuples that have as prefix a tuple of values over the variables shared with its parent view. Yet in case of some views, we may also need a tertiary index to support updates, which are propagated bottom-up. For instance, the view $V_S^{@E}[A, C]$ requires: a primary index to retrieve the payload for each (A, C)-tuple; a secondary index to enumerate the C-values paired with a given A-value fixed by the parent view; and a tertiary index to obtain all A-values paired with a given C-value c fixed by the delta of its left sibling $\delta V_T^{@D}[c]$. All other views only require primary and secondary indices and no tertiary index. □

7.2 *Q-hierarchical queries*

Q-hierarchical queries form a strict subclass of the free-connex acyclic queries. They admit linear preprocessing time, constant update time, and constant enumeration delay [9]. Under widely held complexity theoretic assumptions, there is no algorithm that achieves constant update time and enumeration delay for queries that are not *q*-hierarchical and have no repeating relation symbols [9]. F-IVM recovers the aforementioned complexities using exactly the same approach as for free-connex acyclic queries detailed in Sect. 7.1. This directly implies linear preprocessing time and constant enumeration delay. Constant update time follows from the following observation. Every *q*-hierarchical query admits a free-top variables order, where each root-to-leaf path consists of variables that represent precisely the schema of a relation in the query. A single-tuple update to that relation then sets all these variables to constants, effectively making each delta view along that path of constant size. Our view tree construction also ensures that the computation of each delta view only requires one constant-time lookup per child view.

We first define *q*-hierarchical queries and then show how F-IVM achieves constant-time update for them. For a variable X in a query, we denote by $\text{rels}(X)$ the set of relations that contain X in their schema.

Definition 4 ([9, 62]) A query is *hierarchical* if for any two variables X and Y, it holds $\text{rels}(X) \subseteq \text{rels}(Y)$, $\text{rels}(Y) \subseteq \text{rels}(X)$, or $\text{rels}(X) \cap \text{rels}(Y) = \emptyset$.

A query is *q-hierarchical* if it is hierarchical and for any two variables X and Y, it holds: if $\text{rels}(X) \supset \text{rels}(Y)$ and Y is free, then X is free.

Every *q*-hierarchical query admits a *canonical free-top* variable order, where (i) each root-to-leaf path consists of variables that form the schema of a relation and (2) no bound variable is above a free variable [28]. We can construct such a variable order in polynomial time in the query size as follows. We start with the empty variable order. For each relation R, we add to the variable order a root-to-leaf path made up of R's variables ordered as follows: a variable X is before a variable Y if (1) $\text{rels}(X) \supset \text{rels}(Y)$ or (2) $\text{rels}(X) \not\supset \text{rels}(Y)$, $\text{rels}(X) \not\subset \text{rels}(Y)$, X is free, and Y is bound.

Example 15 The free-connex acyclic query Q[A, B, C] $= \bigoplus_D \bigoplus_E R[A, B] \otimes S[A, C, E] \otimes T[C, D]$ from Ex. 11 is not hierarchical: the sets $\text{rels}(A) = \{R, S\}$ $\text{rels}(C) = \{S, T\}$ are not disjoint, nor one is included in the other. By extending the schema of T with A, we obtain the *q*-hierarchical query $Q_h[A, B, C] = \bigoplus_D \bigoplus_E R[A, B] \otimes S[A, C, E] \otimes T[A, C, D]$ whose canonical free-top variable order is given in Fig. 12 (left). The variant of the query, where variable A is bound, is hierarchical but not *q*-hierarchical because the set $\text{rels}(A) = \{R, S, T\}$ for the *bound* variable A is a strict superset of the set $\text{rels}(B) = \{R\}$ for the *free* variable B. □
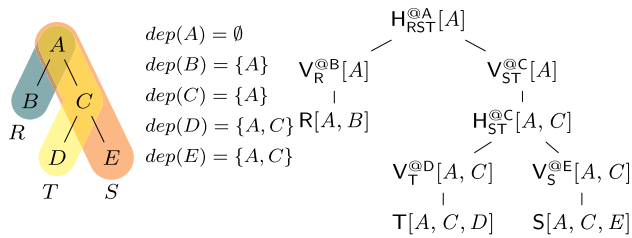
Fig. 12 (Left) canonical free-top variable order of the query Q_h from Ex. 15; (right) Corresponding view tree
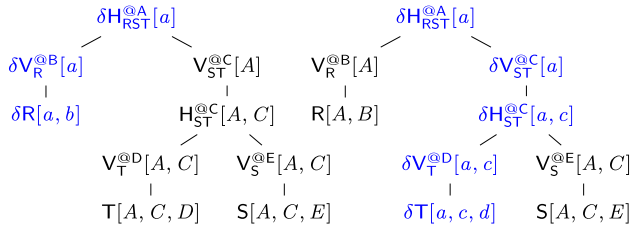
Fig. 13 Delta view trees derived from the view tree in Fig. 12 for single-tuple updates to relations R (left) and T (right)

We next exemplify how F-IVM achieves constant-time update for a q-hierarchical query.

Example 16 Fig. 12 shows the view tree (right) modeled on the canonical free-top variable order (left) of the q-hierarchical query Q_h in Ex. 15. Figure 13 shows the delta view trees under single-tuple updates to R and T.

In the delta view tree for R, the delta view $\delta H_{RST}^{@A}$ can be computed by a constant-time lookup in $V_{ST}^{@C}$. In the delta view tree for T, the delta views $\delta H_{ST}^{@C}$ and $\delta H_{RST}^{@A}$ can be computed by constant-time lookups in $V_S^{@E}$ and $V_R^{@B}$, respectively. All other delta views are computed by marginalizing a variable with a single value. □

Remark 3 Q-hierarchical queries admit view trees whose views only need primary indices to support payload lookup and updates and possibly secondary indices to support enumeration. Consider the view tree in Fig. 12. Enumeration proceeds top-down: We iterate over the A-values in the top view and for each such value a, we look up in $R[a, B]$ to enumerate over all the B-values paired with a, and also look up into $H_{ST}^{@C}[a, C]$ to enumerate over all C-values paired with a. All these look-ups require primary or secondary indices.

Figure 13 shows the delta view trees for single-tuple updates to R and T. To compute a delta view along the path from the delta relation to the root of the delta view tree, we either perform a projection on a delta view or a lookup in the primary index of a sibling view (so with all keys of the index set to constants). □

7.3 Queries under functional dependencies

Non-hierarchical queries may become hierarchical under functional dependencies (fds) [48].

Given a set Σ of fds, we denote by $\text{CLOSURE}_\Sigma(\mathcal{S})$ the closure of the set $\mathcal{S}$ of variables under Σ [2]. For instance, given the fds $\Sigma = \{A \rightarrow D; BD \rightarrow E\}$, we have $\text{CLOSURE}_\Sigma(\{A, B, C\}) = \{A, B, C, D, E\}$.

Definition 5 (adapted from [48]) Given a set Σ of fds and a query $Q[\mathcal{S}] = \bigoplus_\mathcal{B} R_1[\mathcal{S}_1] \otimes \cdots \otimes R_n[\mathcal{S}_n]$, the Σ-*reduct* of Q under Σ is:

$$Q[\text{CLOSURE}_\Sigma(\mathcal{S})] = \bigoplus_\mathcal{B} R_1[\text{CLOSURE}_\Sigma(\mathcal{S}_1)] \otimes \cdots \otimes R_n[\text{CLOSURE}_\Sigma(\mathcal{S}_n)]$$

The Σ-reduct of a query is thus another query, where the schema of each relation is extended to include all variables in the closure of this schema under Σ. Since the added variables are functionally determined by the original schema, they do not add more information. So, we could extend these schemas and the underlying database without increasing the number of tuples in the relations. For any database D with fds Σ and a query Q, the query result $Q(D)$ is the same as the result of its Σ-reduct over the extended database. The benefit of this rewriting is that queries may admit free-connex acyclic or even q-hierarchical Σ-reducts. We need not physically extend the database to reap this benefit. Instead, we use the Σ-reduct of Q to infer a free-top variable order or even a canonical free-top variable order *for* Q in case the Σ-reduct is free-connex acyclic or q-hierarchical, respectively. Using this variable order, we construct a view tree for Q that enjoys the preprocessing, update, and enumerate times as for its Σ-reduct.

Theorem 1 can be generalized to account for fds.

Theorem 2 *Let a query Q and a database of size N and with a set Σ of functional dependencies.*

F-IVM can maintain Q with $O(N)$ preprocessing, $O(1)$ enumeration delay, and $O(N)$ single-tuple updates in case the Σ-reduct of Q is free-connex acyclic.
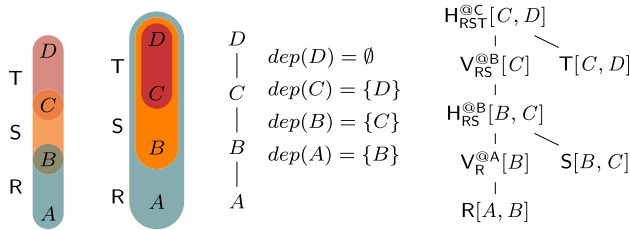
F-IVM can maintain Q with $O(N)$ preprocessing, $O(1)$ enumeration delay, and $O(1)$ single-tuple updates in case the Σ-reduct of Q is q-hierarchical.

Example 17 Consider $\Sigma = \{B \rightarrow C, C \rightarrow D\}$ and the free-connex acyclic but not hierarchical query
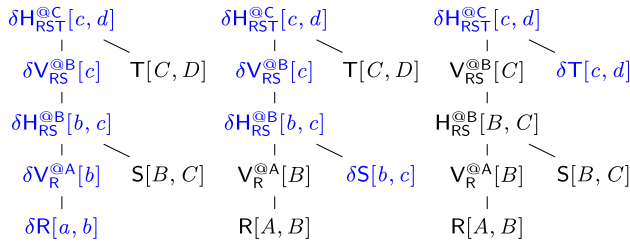
$$Q[A, B, C, D] = R[A, B] \otimes S[B, C] \otimes T[C, D].$$

The Σ-reduct of Q is

$$Q'[A, B, C, D] = R[A, B, C, D] \otimes S[B, C, D] \otimes T[C, D].$$

Fig. 14 From left to right: hypergraph of the query Q and its Σ-reduct for $\Sigma = \{B \rightarrow C, C \rightarrow D\}$ from Ex. 17; canonical variable order ω for Q; view tree modeled on ω

Fig. 15 Delta view trees derived from the view tree in Fig. 14 for single-tuple updates to R, S, and T (left to right). The values b and c functionally determine c and d, respectively

Figure 14 depicts the hypergraphs of Q and Q' (left), a free-top variable order for Q that is also canonical for Q' (middle), and the view tree for Q modeled on this variable order (right). Since Q is free-connex acyclic, we can compute the view tree in linear time and enumerate the result tuples of Q with constant delay, as explained in Sect. 7.1. We next describe how to achieve constant-time update by exploiting the fds. Figure 15 shows the delta view trees obtained from the view tree for Q for single-tuple updates to R, S, and T.

Consider first the update $\delta R[a, b]$ to relation R. The delta view $\delta V_R^{@A}[b]$ is just a projection of the update tuple. The delta view $\delta H_{RS}^{@B}[b, c]$ requires a lookup in $S[B, C]$ for $B = b$. In general, there may be many C-values paired with b. However, under the fd $B \rightarrow C$, there is at most one C-value c paired with b. Hence, the construction of this delta view takes constant time. Similarly, the delta view $\delta H_{RST}^{@C}[c, d]$ requires a lookup in $T[C, D]$ for $C = c$. Again, there may be many D-values paired with c, yet under the fd $C \rightarrow D$, there is at most one D-value d paired with c. Hence, the construction of this delta view takes constant time, too.

Similar reasoning applies to the update $\delta S[b, c]$. To compute the delta view $\delta H_{RS}^{@B}[c, b]$, we need a constant-time lookup in the view $V_R^{@A}[B]$ with $B = b$. Computing $\delta H_{RST}^{@C}[c, d]$ takes constant time due to the fd $C \rightarrow D$, as with updates to R. Processing the update $\delta T[c, d]$ takes constant time without exploiting the fds: it only requires a lookup in the view $V_{RS}^{@B}[C]$ with $C = c$. $\qquad\square$

8 Applications

This section highlights four applications of F-IVM, including learning regression models, building Chow-Liu trees, computing listing or factorized representations of the results of conjunctive queries, and multiplying a sequence of matrices. They behave the same in the key space, yet differ in the rings used to define the payloads.

8.1 Covariance matrix and linear regression

We next introduce the covariance matrix ring used for training linear regression models.

Linear Regression. Consider a training dataset that consists of k samples with $(X_i)_{i \in [m-1]}$ features and a label X_m arranged into a design matrix $\mathbf{M}$ of size $k \times m$; in our setting, this design matrix is the result of a join query. The goal of linear regression is to learn the parameters $\boldsymbol{\theta} = [\theta_1 \ldots \theta_m]^T$ of a linear function[3] $f(X_1, ..., X_{m-1}) = \sum_{i \in [m-1]} \theta_i X_i$ best satisfying $\mathbf{M}\boldsymbol{\theta} \approx \mathbf{0}_{k \times 1}$, where $\mathbf{0}_{k \times 1}$ is the zero matrix of size $k \times 1$.

We can solve this optimization problem using batch gradient descent. This method iteratively updates the model parameters in the direction of the gradient to decrease the squared error loss and eventually converge to the optimal value. Each convergence step iterates over the entire training dataset to update the parameters, $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \mathbf{M}^T \mathbf{M} \boldsymbol{\theta}$, where α is an adjustable step size. The complexity of each step is $\mathcal{O}(mk)$. The *covariance matrix* $\mathbf{M}^T \mathbf{M}$ quantifies the degree of correlation for each pair of features (or feature and label) in the data. Its computation can be done once for all convergence steps [58]. This is crucial for performance in case $m \ll k$ as each iteration step now avoids processing the entire training dataset and takes time $\mathcal{O}(m^2)$.

We next show how to compute the covariance matrix assuming all features have continuous domains; we consider the case with categorical features later on.

The covariance matrix $\mathbf{M}^T \mathbf{M}$ accounts for the interactions SUM(X*Y) of variables X and Y with continuous domains. We can factorize their computation over training datasets defined by arbitrary join queries [58]. We can further share their computation by casting the covariance matrix computation as the computation of one compound aggregate. This compound aggregate is a triple $(c, \boldsymbol{s}, \boldsymbol{Q})$, where c is the number of tuples in the training dataset (size k of the design matrix), $\boldsymbol{s}$ is an $m \times 1$ matrix (or vector) with one sum of values per variable, and $\boldsymbol{Q}$ is an $m \times m$ matrix of sums of products of values for any two variables. The covariance matrix computation can be captured by a ring.

---

[3] We consider wlog: θ_1 is the bias parameter and then $X_1 = 1$ for all tuples in the input data; θ_m remains fixed to -1 and corresponds to the label/response X_m in the data.

Definition 6 Fix a ring $(\mathbf{D}, +, *, \mathbf{0}, \mathbf{1})$ and $m \in \mathbb{N}$. Let C denote the set of triples $(\mathbf{D}, \mathbf{D}^m, \mathbf{D}^{m \times m})$, $\mathbf{0}^{\mathsf{C}} = (0, \mathbf{0}_{m \times 1}, \mathbf{0}_{m \times m})$, and $\mathbf{1}^{\mathsf{C}} = (1, \mathbf{0}_{m \times 1}, \mathbf{0}_{m \times m})$, where $\mathbf{0}_{m \times n}$ is an $m \times n$ matrix with all zeros from $\mathbf{D}$. For $a = (c_a, \mathbf{s}_a, \mathbf{Q}_a) \in \mathsf{C}$ and $b = (c_b, \mathbf{s}_b, \mathbf{Q}_b) \in \mathsf{C}$, define the operations $+^{\mathsf{C}}$ and $*^{\mathsf{C}}$ over C as:

$$a +^{\mathsf{C}} b = (c_a + c_b, \; \mathbf{s}_a + \mathbf{s}_b, \; \mathbf{Q}_a + \mathbf{Q}_b)$$
$$a *^{\mathsf{C}} b = (c_a c_b, \; c_b \mathbf{s}_a + c_a \mathbf{s}_b, \; c_b \mathbf{Q}_a + c_a \mathbf{Q}_b + \mathbf{s}_a \mathbf{s}_b^{\mathsf{T}} + \mathbf{s}_b \mathbf{s}_a^{\mathsf{T}})$$

using matrix addition, scalar multiplication, and matrix multiplication over $\mathbf{D}$. We refer to $(\mathsf{C}, +^{\mathsf{C}}, *^{\mathsf{C}}, \mathbf{0}^{\mathsf{C}}, \mathbf{1}^{\mathsf{C}})$ as the *covariance structure of degree m over $\mathbf{D}$*.

Theorem 3 *For $m \in \mathbb{N}$ and a ring $\mathbf{D}$, the covariance structure of degree m over $\mathbf{D}$ forms a commutative ring.*

Definition 7 The *continuous covariance ring of degree m* is the covariance structure of degree m over $\mathbb{R}$.

We next show how to use this ring to compute the covariance matrix over a training dataset defined by a join with relations $(\mathsf{R}_i)_{i \in [n]}$ over variables $(X_j)_{j \in [m]}$. The payload of each tuple in a relation is the identity $\mathbf{1}^{\mathsf{C}}$ from the continuous covariance ring of degree m. The query computing the covariance matrix is:

$$\mathsf{Q} = \bigoplus_{X_1} \cdots \bigoplus_{X_m} \bigotimes_{i \in [n]} \mathsf{R}_i[\mathsf{sch}(R_i)]$$

For each X_j-value x, the lifting function is $g_{X_j}(x) = (1, \mathbf{s}, \mathbf{Q})$, where $\mathbf{s}$ is an $m \times 1$ vector with all zeros except the value of x at position j, i.e., $\mathbf{s}_j = x$, and $\mathbf{Q}$ is an $m \times m$ matrix with all zeros except the value x^2 at position (j, j): $\mathbf{Q}_{(j,j)} = x^2$.

Example 18 We show how to compute the covariance matrix using the join and view tree from Fig. 3 and the database from Fig. 6. We assume alphabetical order of the five variables in the covariance matrix. The leaf relations R, S, and T map tuples to $\mathbf{1}^{\mathsf{C}}$ from the continuous covariance ring of degree 5.

In the view $\mathsf{V}_\mathsf{T}^{@\mathsf{D}}$, each D-value d is lifted to a triple $(1, \mathbf{s}, \mathbf{Q})$, where $\mathbf{s}$ is a 5×1 vector with one non-zero element $\mathbf{s}_4 = d$, and $\mathbf{Q}$ is a (5×5) matrix with one non-zero element $\mathbf{Q}_{(4,4)} = d^2$. Those covariance triples with the same key c are summed up, yielding:

$$\mathsf{V}_\mathsf{T}^{@\mathsf{D}}[c_1] = (1, \mathbf{s}_4 = d_1, \mathbf{Q}_{(4,4)} = d_1^2)$$
$$\mathsf{V}_\mathsf{T}^{@\mathsf{D}}[c_2] = (2, \mathbf{s}_4 = d_2 + d_3, \mathbf{Q}_{(4,4)} = d_2^2 + d_3^2)$$
$$\mathsf{V}_\mathsf{T}^{@\mathsf{D}}[c_3] = (1, \mathbf{s}_4 = d_4, \mathbf{Q}_{(4,4)} = d_4^2)$$

The views $\mathsf{V}_\mathsf{R}^{@\mathsf{B}}$ and $\mathsf{V}_\mathsf{S}^{@\mathsf{E}}$ are computed similarly. The view $\mathsf{V}_\mathsf{ST}^{@\mathsf{C}}$ joins $\mathsf{V}_\mathsf{T}^{@\mathsf{D}}$ and $\mathsf{V}_\mathsf{S}^{@\mathsf{E}}$ and marginalizes C. For instance, the payload for the key a_2 is:

$$\mathsf{V}_\mathsf{ST}^{@\mathsf{C}}[a_2]$$
$$= \mathsf{V}_\mathsf{T}^{@\mathsf{D}}[c_2] *^{\mathsf{C}} \mathsf{V}_\mathsf{S}^{@\mathsf{E}}[a_2, c_2] *^{\mathsf{C}} g_C(c_2)$$
$$= \mathsf{V}_\mathsf{T}^{@\mathsf{D}}[c_2] *^{\mathsf{C}} \left(1, \begin{vmatrix} 0 \\ 0 \\ 0 \\ 0 \\ e_4 \end{vmatrix}, \begin{vmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & e_4^2 \end{vmatrix} \right) *^{\mathsf{C}} \left(1, \begin{vmatrix} 0 \\ 0 \\ c_2 \\ 0 \\ 0 \end{vmatrix}, \begin{vmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ c_2^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix} \right)$$
$$= \left(2, \begin{vmatrix} 0 \\ 0 \\ 2c_2 \\ d_2 + d_3 \\ 2e_4 \end{vmatrix}, \begin{vmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2c_2^2 & c_2(d_2 + d_3) & 2c_2 e_4 \\ 0 & 0 & c_2(d_2 + d_3) & d_2^2 + d_3^2 & (d_2 + d_3) e_4 \\ 0 & 0 & 2c_2 e_4 & (d_2 + d_3) e_4 & 2e_4^2 \end{vmatrix} \right)$$

The root view $\mathsf{V}_\mathsf{RST}^{@\mathsf{A}}$ maps the empty tuple to the ring element $\sum_{i \in [2]} \mathsf{V}_\mathsf{R}^{@\mathsf{B}}[a_i] *^{\mathsf{C}} \mathsf{V}_\mathsf{ST}^{@\mathsf{C}}[a_i] *^{\mathsf{C}} g_A(a_i)$. This payload has aggregates for the entire join result: the count of tuples in the result, the vector with one sum of values per variable, and the covariance matrix. □

Linear Regression with Categorical Variables. Real-world datasets consists of both continuous and categorical variables. The latter take on values from predefined sets of possible values (categories). It is common practice to one-hot encode categorical variables as indicator vectors. This encoding can blow up the size of the covariance matrix and increase its sparsity.

Instead of blowing up the covariance matrix with one-hot encoding, we can capture the interactions between continuous and categorical variables as group-by queries: `SUM(X)` group by Y, when X is continuous and Y is categorical, and `SUM(1)` group by X and Y, when X and Y are categorical. Using the group-by queries ensures a compact representation of such interactions by considering only those categories and interactions that exist in the join result. We can encode those interactions as values from the relational data ring, introduced next.

Definition 8 Let $\mathbb{F}[\mathbb{R}]$ denote the set of relations over the $\mathbb{R}$ ring, the zero $\mathbf{0}$ in $\mathbb{F}[\mathbb{R}]$ is the empty relation $\{\}$, which maps every tuple to $0 \in \mathbb{R}$, and the identity $\mathbf{1}$ is the relation $\{() \to 1\}$, which maps the empty tuple to $1 \in \mathbb{R}$ and all other tuples to $0 \in \mathbb{R}$. The structure $(\mathbb{F}[\mathbb{R}], \uplus, \otimes, \mathbf{0}, \mathbf{1})$ forms the *relational data* ring.[4]

We generalize the continuous covariance ring from Definition 7 to uniformly treat continuous and categorical variables

---

[4] To form a proper ring, we need a generalization [31] of relations and join and union operators, where: tuples have their own schemas; union applies to tuples with possibly different schemas; join accounts for multiple derivations of output tuples. For our needs this generalization is not necessary.

as follows: we use relations from the relational data ring as values in c, s, and Q instead of scalars; we use union and join instead of scalar addition and multiplication; we use the empty relation $\mathbf{0}$ instead of the zero scalar. The operations $+^C$ and $*^C$ over triples (c, s, Q) remain unchanged.

Definition 9 The *generalized covariance ring of degree m* is the covariance structure of degree m over $\mathbb{F}[\mathbb{R}]$.

For clarity, we show the operations $+^C$ and $*^C$ of the generalized covariance ring C of degree m.

$$(c', s', Q') +^C (c'', s'', Q'') = (c, s, Q)$$

where $c = c' \uplus c''$, $s_j = s'_j \uplus s''_j$, $Q_{(i,j)} = Q''_{(i,j)} \uplus Q''_{(i,j)}$;

$$(c', s', Q') *^C (c'', s'', Q'') = (c, s, Q)$$

where $c = c' \otimes c''$, $s_j = (c'' \otimes s'_j) \uplus (c' \otimes s''_j)$, and $Q_{(i,j)} = (c'' \otimes Q'_{(i,j)}) \uplus (c' \otimes Q''_{(i,j)}) \uplus (s'_i \otimes s''_j) \uplus (s''_i \otimes s'_j)$.

The lifting function g_{X_j} now depends on whether X_j is continuous or categorical. For each X_j-value x, $g_{X_j}(x) = (\mathbf{1}, s, Q)$, where $\mathbf{1} = \{() \to 1\}$, s is an $m \times 1$ vector with all $\mathbf{0}$s except $s_j = \{() \to x\}$ if X_j is continuous and $s_j = \{x \to 1\}$ otherwise, and Q is an $m \times m$ matrix with all $\mathbf{0}$s except $Q_{(j,j)} = \{() \to x^2\}$ if X_j is continuous and $Q_{(j,j)} = \{x \to 1\}$ otherwise.

Example 19 We compute the covariance matrix using the view tree and database from Ex. 18 assuming that C is categorical.

Since B, D, and E are continuous, the contents of $V_R^{@B}$, $V_T^{@D}$, and $V_S^{@E}$ are similar to those of Ex. 18 except that every scalar value x in their payloads is replaced by the relation $\{() \to x\}$. The view $V_{ST}^{@C}$ marginalizes C, lifting every C-value c to $(\mathbf{1}, s_3 = \{c \to 1\}, Q_{(3,3)} = \{c \to 1\})$, and the other entries in s and Q are $\mathbf{0}$s. The payload $V_{ST}^{@C}[a_2]$ encodes the result of $\text{SUM}(1)$ group by C as $s_3 = Q_{(3,3)} = \{c_2 \to 2\}$, the result of $\text{SUM}(D)$ group by C as $Q_{(3,4)} = \{c_2 \to d_2 + d_3\}$, and the result of $\text{SUM}(E)$ group by C as $Q_{(3,5)} = \{c_2 \to 2e_4\}$. The remaining entries in the payload $V_{ST}^{@C}[a_2]$ are relations mapping the empty tuple to the same scalar value from $V_{ST}^{@C}[a_2]$ in Ex. 18. The root view $V_{RST}^{@A}$ computes the payload associated with the empty tuple in the same manner as in the continuous-only case but under the generalized covariance ring. □

Remark 4 For performance reasons, we only store as payloads blocks of matrices with non-zero values and assemble larger matrices as the computation progresses toward the root of the view tree. We further exploit the symmetry of the covariance matrix to compute only the entries above and including the diagonal. For the generalized covariance ring, we store relations, which have the empty tuple as key, as scalar values.

8.2 Mutual information and Chow-Liu tree

The mutual information (MI) of two random variables X and Y quantifies their degree of correlation [42]:

$$I(X, Y) = \sum_{x \in \text{Dom}(X)} \sum_{y \in \text{Dom}(Y)} p_{XY}(x, y) \log \frac{p_{XY}(x, y)}{p_X(x) p_Y(y)}$$

where $p_{XY}(x, y)$ is the joint probability of $X = x$ and $Y = y$, and $p_X(x)$ and $p_Y(y)$ are the marginal probabilities of $X = x$ and $Y = y$, respectively. A value close to 0 means the variables are almost independent, while a large value means they are highly correlated. It can be used to identify variables that predict a given label variable and can thus be used for model selection [42].

In our case, we are given the joint probability of several categorical variables as a relation, or the join of several relations. The probabilities defining the MI of any pair of variables can be computed as group-by aggregates over this relation. Let $C_\emptyset = \text{SUM}(1)$, $C_X = \text{SUM}(1)$ group by X, $C_Y = \text{SUM}(1)$ group by Y, and $C_{XY} = \text{SUM}(1)$ group by X, Y. Then, $p_{XY}(x, y) = \frac{C_{XY}(x,y)}{C_\emptyset}$, $p_X(x) = \frac{C_X(x)}{C_\emptyset}$, $p_Y(y) = \frac{C_Y(y)}{C_\emptyset}$, and

$$I(X, Y) = \sum_{x \in \text{Dom}(X)} \sum_{y \in \text{Dom}(Y)} \frac{C_{XY}(x, y)}{C_\emptyset} \log \frac{C_\emptyset C_{XY}(x, y)}{C_X(x) C_Y(y)}$$

The aggregates $C_\emptyset$, C_X, and C_{XY} define the covariance matrix over categorical variables, so we can use the generalized covariance ring to compute and maintain them (Sect. 8.1). To compute the MI for continuous variables, we first discretize their domains into finitely many bins, so we turn them into categorical variables.

Mutual information is used for learning the structure of Bayesian networks. Let a graph with one node per variable and one edge per pair of variables weighted by their MI, a Chow-Liu tree is a maximum weight spanning tree. The Chow-Liu algorithm [16] constructs such a tree in several rounds: it starts with a single node in the tree and in each round it connects a new node to a node already in the tree such that their pairwise MI is maximal among all pairs of variables not chosen yet.

8.3 Factorized representation of query results

Our framework can also support scenarios where the view payloads are themselves relations representing results of conjunctive queries, or even their factorized representations. Factorized representations can be much smaller than the listing representation of a query result [52], with orders of magnitude size gaps reported in practice [58]. They nevertheless remain lossless and support constant-delay enumeration

of the tuples in the query result as well as subsequent aggregate processing in one pass. Besides the factorized view computation and the factorizable updates, this is the third instance where our framework exploits factorization.

We store entire relations as payloads using a variant of the relational data ring (c.f. Definition 8) where values are relations over the $\mathbb{Z}$ ring. We denote this ring as $\mathbb{F}[\mathbb{Z}]$. When marginalizing a variable, we move its values from the key space to the payload space. The tuple payloads in a view are now relations over the same schema. These relations have themselves payloads in the $\mathbb{Z}$ ring used to maintain the multiplicities of their tuples.

We model conjunctive queries as count queries that marginalize *every* variable but use different lifting functions for the free and bound variables. For a free variable X and any of its values x, we define $g_X(x) = \{x \rightarrow 1\}$, i.e., the lifting function maps x to the unary relation that consists of the single value x whose payload is 1. In case X is bound, we define $g_X(x) = \mathbf{1} = \{() \rightarrow 1\}$, i.e., the lifting function maps x to the identity element $\mathbf{1}$ of the relational data ring. This element is the unique relation that consist of the empty tuple whose payload is 1. We have relational operations occurring at two levels: for keys, we join views and marginalize variables as before; for payloads, we interpret multiplication and addition of payloads as join and union of relations.

Example 20 Consider the conjunctive query

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

over the three relations from Fig. 6, where each tuple gets the identity payload $\{() \rightarrow 1\} \in \mathbb{F}[\mathbb{Z}]$. The corresponding view marginalizes all the variables:

$$Q[\,] = \bigoplus_A \ldots \bigoplus_E R[A, B] \otimes S[A, C, E] \otimes T[C, D]$$

The lifting function for E maps each value to $\{() \rightarrow 1\}$, while the lifting functions for all other variables map value x to $\{x \rightarrow 1\}$.

Figure 16 shows the contents of the views with relational data payloads (in black and red) for the view tree from Fig. 3 and the database from Fig. 6. The view keys gradually move to payloads as the computation progresses toward the root. The view definitions are identical to those of the COUNT query (but under a different ring!). The view $V_T^{@D}$ lifts each D-value d from T to the relation $\{d \rightarrow 1\}$ over schema $\{D\}$, multiplies (joins) it with the payload $\mathbf{1}$ of each tuple, and sums up (union) all payloads with the same c-value. The views at $V_R^{@B}$ and $V_S^{@E}$ are computed similarly, except the latter lifts e-values to $\mathbf{1}$ since E is a bound variable. The view $V_{ST}^{@C}$ assigns to each A-value a payload that is a union of Cartesian products of the payloads of its children and the lifted C-value. The root view $V_{RST}^{@A}$ similarly computes the

Fig. 16 Computing the query from Ex. 20 over the database in Fig. 6 and the relational ring, where $\forall i \in [12] : p_i = \{() \rightarrow 1\}$. The computation uses the view tree τ in Fig. 3. The red views (rightmost column) have payloads storing the listing representation of the intermediate and final query results. The blue views (top two views in the middle column) encode a factorized representation of these results distributed over their payloads. The remaining (black) views remain the same for both representations

payload of the empty tuple, which represents the query result (both views are at the right).

We next show how to construct a factorized representation of the query result. In contrast to the scenarios discussed above, this representation is *not* available as one payload at the root view, but *distributed* over the payloads of all views. This hierarchy of payloads, linked via the keys of the views, becomes the factorized representation. A further difference lies with the multiplication operation. For the listing representation, the multiplication is the Cartesian product. For a given view, it is used to concatenate payloads from its child views. For the factorized representation, we further project away values for all but the marginalized variable. More precisely, for each view $V_{rels}^{@X}[\mathcal{S}]$ and each of its keys $a_{\mathcal{S}}$, let $P[\mathcal{T}] = V_{rels}^{@X}[a_{\mathcal{S}}]$ be the corresponding payload relation. Then, instead of computing this payload, we compute $\bigoplus_{Y \in \mathcal{T} - \{X\}} P[\mathcal{T}]$ by marginalizing the variables in $\mathcal{T} - \{X\}$ and summing up the multiplicities of the tuples in $P[\mathcal{T}]$ with the same X-value.

Example 21 We continue Ex. 20. Figure 16 shows the contents of the views with factorized payloads (first two columns in black and blue). Each view stores relational payloads that have the schema of the marginalized variable. Together, these payloads form a factorized representation over the variable order ω used to define the view tree in Fig. 3. At the top of the factorization, we have a union of two A-values: a_1 and a_2. This is stored in the payloads of (middle) $V_{RST}^{A}[\,]$. The payloads of (middle) $V_{ST}^{@C}[A]$ store a union of C-values c_1 and c_2 under a_1, and a singleton union of c_2 under a_2. The payloads of $V_R^{@B}[A]$ store a union of B-values b_1 and b_2 under a_1

and a singleton union of b_3 under a_2. Note the (conditional) independence of the variables B and C given a value for A. This is key to succinctness of factorization. In contrast, the listing representation explicitly materializes all pairings of B and C-values for each A-value, as shown in the payload of (right) $\mathsf{V}^A_{\mathsf{RST}}[\ ]$. Furthermore, the variable D is independent of the other variables *given* C. This is a further source of succinctness in the factorization: Even though c_2 occurs under both a_1 and a_2, the relations under c_2, in this case the union of d_2 and d_3, is only stored once in $\mathsf{V}^{@D}_\mathsf{T}[C]$. Each value in the factorization keeps a multiplicity, that is, the number of its derivations from the input data. This is necessary for maintenance.

This factorization is over a variable order that can be used for all queries with same body and different free variables: As long as their free variables sit on top of the bound variables, the variable order is valid and so is the factorization over it. For instance, if the variable D were not free, then the factorization for the new query would be the same except that we would now discard the D-values from the payload of the view $\mathsf{V}^{@D}_\mathsf{T}$. □

8.4 Matrix chain multiplication

Consider the problem of computing a product of a sequence of matrices $A_1, \ldots, A_n$ over some ring $\mathbf{D}$, where matrix $A_i[x_i, x_{i+1}]$ has the size $p_i \times p_{i+1}$, $i \in [n]$. The product $A = A_1 \cdots A_n$ is a matrix of size $p_1 \times p_{n+1}$ and can be formulated as follows:

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

We model a matrix A_i as a relation $[X_i, X_{i+1}]A_i$ with the payload carrying matrix values. The query that computes the matrix A is:

$$\mathsf{A}[X_1, X_{n+1}] = \bigoplus_{X_2} \cdots \bigoplus_{X_n} \bigotimes_{i \in [n]} \mathsf{A}_i[X_i, X_{i+1}]$$

where each of the lifting functions $\{g_{X_j}\}_{j \in [2,n]}$ maps any key value to payload $\mathbf{1} \in \mathbf{D}$. Different variable orders lead to different evaluation plans for matrix chain multiplication. The optimal variable order corresponds to the optimal sequence of matrix multiplications that minimizes the overall multiplication cost, which is the textbook Matrix Chain Multiplication problem [17].

Example 22 Consider a multiplication chain of 4 matrices of equal size $p \times p$ encoded as relations $\mathsf{A}_i[X_i, X_{i+1}]$. Let $\mathcal{F} = \{X_1, X_5\}$ be the set of free variables and ω be the variable order $X_1 - X_5 - X_3 - \{X_2, X_4\}$, i.e., X_2 and X_4 are children of X_3, with the matrix relations placed below the leaf variables in ω. The view tree $\tau(\omega, \mathcal{F})$ has the following views

(from bottom to top; the views at X_5 and X_1 are equivalent to the view at X_3):

$$\mathsf{V}^{@X_2}_{\mathsf{A}_1\mathsf{A}_2}[X_1, X_3] = \bigoplus_{X_2} \mathsf{A}_1[X_1, X_2] \otimes \mathsf{A}_2[X_2, X_3]$$
$$\mathsf{V}^{@X_4}_{\mathsf{A}_3\mathsf{A}_4}[X_3, X_5] = \bigoplus_{X_4} \mathsf{A}_3[X_3, X_4] \otimes \mathsf{A}_4[X_4, X_5]$$
$$\mathsf{V}^{@X_3}_{\mathsf{A}_1\mathsf{A}_2\mathsf{A}_3\mathsf{A}_4}[X_1, X_5] = \bigoplus_{X_3} \mathsf{V}^{@X_2}_{\mathsf{A}_1\mathsf{A}_2}[X_1, X_3] \otimes \mathsf{V}^{@X_4}_{\mathsf{A}_3\mathsf{A}_4}[X_3, X_5]$$

Recomputing these views from scratch for each update to an input matrix takes $\mathcal{O}(p^3)$ time. A single-value change in any input matrix causes changes in one row or column of the parent view, and propagating them to compute the final delta view takes $\mathcal{O}(p^2)$ time. Updates to A_2 and A_3 change every value in A. In case of a longer matrix chain, propagating $\delta \mathsf{A}$ further requires $\mathcal{O}(p^3)$ matrix multiplications, same as recomputation.

We exploit factorization to contain the effect of such changes. For instance, if $\delta \mathsf{A}_2$ is a factorizable update expressible as $\delta \mathsf{A}_2[X_2, X_3] = \mathsf{u}[X_2] \otimes \mathsf{v}[X_3]$ (see Sect. 6), then we can propagate deltas more efficiently, as products of subexpressions:

$$\delta \mathsf{V}^{@X_2}_{\mathsf{A}_1\mathsf{A}_2}[X_1, X_3] = \underbrace{\left(\bigoplus_{X_2} \mathsf{A}_1[X_1, X_2] \otimes \mathsf{u}[X_2]\right)}_{\mathsf{u}_2[X_1]} \otimes \mathsf{v}[X_3]$$
$$\delta \mathsf{V}^{@X_3}_{\mathsf{A}_1\mathsf{A}_2\mathsf{A}_3\mathsf{A}_4}[X_1, X_5] = \mathsf{u}_2[X_1] \otimes \left(\bigoplus_{X_3} \mathsf{v}[X_3] \otimes \mathsf{V}^{@X_4}_{\mathsf{A}_3\mathsf{A}_4}[X_3, X_5]\right)$$

Using such factorizable updates enables the incremental computation in $\mathcal{O}(p^2)$ time. The final delta is also in factorized form, suitable for further propagation.

In general, for a chain of k matrices of size $p \times p$, using a binary view tree of the lowest depth, incremental maintenance with factorizable updates takes $\mathcal{O}(p^2 \log k)$ time, while reevaluation takes $\mathcal{O}(p^3 k)$ time. The space needed in both cases is $\mathcal{O}(p^2 k)$. □

The above example recovers the main idea of LINVIEW [44]: use factorization in the incremental computation of linear algebra programs where matrix changes are encoded as vector outer products, $\delta A = uv^\mathsf{T}$. Such rank-1 updates can capture many practical update patterns such as perturbations of one complete row or column, or even changes of the whole matrix when the same vector is added to every row or column. F-IVM generalizes this idea to arbitrary join-aggregate queries.

9 Experiments

This section reports our experimental findings with our system F-IVM and three competitors: first-order IVM (1-IVM),

DBToaster's higher-order IVM (DBT), and Apache Flink. We first summarize our findings.

(1) For maintaining covariance matrices over continuous variables, F-IVM outperforms DBT and 1-IVM by up to three orders of magnitude. This is primarily due to the use of the covariance ring in F-IVM, which can capture the maintenance for an entire covariance matrix of 100-800 entries with under ten views. In contrast, DBT requires 600-3,000 views, while 1-IVM needs as many delta queries as matrix entries (136 - 820). A similar conclusion holds for maintaining covariance matrices over continuous and categorical variables and also only over categorical variables, albeit the performance gap becomes smaller. Thanks to the covariance ring, F-IVM also has a low memory footprint, on par with 1-IVM and 4-16x less than DBT.

(2) Maintaining linear regression models over the covariance matrices takes insignificant time if the batch gradient descent resumes with the values for the model parameters computed after the previous update batch.

(3) Maintaining mutual information and Chow-Liu trees over the covariance matrices requires recomputation after every update batch and this can decrease the throughput of F-IVM by up to one order of magnitude.

(4) For q-hierarchical queries, F-IVM is the fastest approach in case the updates are followed occasionally by a request to enumerate the query result. F-IVM pushes the updates from the leaves to the root view in the view tree, yet keeps the result factorized. This ensures update time and enumeration delay per tuple proportional to the payload size. We confirmed experimentally that DBT and 1-IVM cannot achieve constant time for both update and enumeration.

(5) For path queries of up to 20 joins over the Twitter and TikTok graph datasets, F-IVM's throughput remains at least an order of magnitude larger than of competitors. 1-IVM and Apache Flink do not manage to process one 1K-batch within four hours for paths of more than 10 joins.

Further experiments are reported in prior work [45]: (1) F-IVM outperforms competitors for maintaining a sum aggregate over joins; (2) Using batches of $1k - 10k$ updates is best for maintaining the covariance matrix; (3) Factorized updates lead to two orders of magnitude speedup for F-IVM over competitors for matrix chain multiplication; and (4) For conjunctive query evaluation, factorized payloads can speed up view maintenance and reduce memory by up to two orders of magnitude compared to the listing representation of payloads.

9.1 Experimental settings

Competitors. The three maintenance strategies use DBToaster v2.3 [32], a system that compiles SQL queries into code that maintains the query result under updates to input relations. The generated code represents an in-memory stream processor that is standalone and independent of any database system. DBToaster's performance on decision support and financial workloads can be several orders of magnitude better than state-of-the-art commercial databases and stream processing systems [32]. DBToaster natively supports DBT and 1-IVM. We use the intermediate language of DBToaster to encode F-IVM that maintains a set of materialized views for a given variable order and a set of updatable relations. We feed this encoding into the code generator of DBToaster. Unless stated otherwise, all approaches use the same runtime and store views as multi-indexed maps with memory-pooled records. The algorithms and record types used in these approaches can differ greatly. We also report on the performance of Apache Flink v1.17.1 [12] (via Table API), configured to utilize all cores and main memory of our machine.

Datasets. Fig. 17 summarizes our datasets:

– *Housing* is a synthetic dataset modeling a house price market [58]. It consists of six relations: House, Shop, Institution, Restaurant, Demographics, and Transport, arranged into a star schema. The natural join of all relations is on the common variable (postcode) and has 26 non-join variables, 14 continuous and 12 categorical. We consider a variable order where each root-to-leaf path consists of variables of one relation.

– *Retailer* is a real-world dataset used by a retailer to inform decision-making and forecast user demands [58]. It has a snowflake schema with one large fact relation Inventory storing information about the inventory units for products in a location, at a given date. This 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). The natural join of these relations is acyclic and has 33 continuous and 6 categorical non-join variables. We use a variable order, where the variables of each relation form a distinct root-to-leaf path, and the

Dataset	#Tuples	#Relations	#JoinVars	# Non-JoinVars
Housing	1.4M	6	1	26
Retailer	85M	5	4	39
Favorita	125M	6	3	15
Twitter	1.7M	1	2	0
TikTok	0.6M	1	2	0

Fig. 17 Characteristics of the input datasets

partial order on join variables is: location - { date - { product id }, zip }.

– *Favorita* is a real-world dataset comprising sales data of items sold in grocery stores in Ecuador [21]. It has a star schema with one large fact relation `Sales` storing information on sales transactions, including the date, store, item, and item quantity. This relation joins with five dimension tables: `Stores` (on store id), `Item` (on item id), `Transaction` (on date and store id), `Holiday` (on date), and `Oil` (on date). The natural join has 3 continuous and 12 categorical non-join variables. We consider a variable order where the order on join variables is: date - store id - item id.

– *Twitter* [38] *and TikTok* [53] are publicly available graph datasets.

We evaluate the maintenance strategies over data streams synthesized from the above datasets by interleaving insertions to the input relations in a round-robin fashion. These insertions arrive sorted following a top-down order of F-IVM's variable orders. This leads to improved runtimes of all systems relative to out-of-order insertions. We group insertions into batches of 1000 tuples and place no restriction on the order of records in input relations. In all experiments, we use payloads defined over rings with additive inverse, thus processing deletions is similar to processing insertions.

Queries. We consider the following queries:

– *Covariance Matrix:* For F-IVM, we use one query per dataset to compute one covariance aggregate over the natural join of the input relations. For instance, the query over the *Retailer* schema is:

```
SELECT SUM(g₁(X₁) * ... * g₃₉(X₃₉))
FROM Inv NATURAL JOIN It NATURAL JOIN W
        NATURAL JOIN L NATURAL JOIN C;
```

where $\{X_i\}_{i \in [39]}$ are all the non-join variables from the *Retailer* schema. We consider three scenarios: (1) we treat all variables as continuous; (2) with a mix of continuous and categorical variables; and (3) with all categorical variables. For the first, we use the continuous covariance ring of degree 39 and the lifting function $g_i(x) = (c_i = 1, s_i = x, Q_{(i,i)} = x^2)$ for each variable X_i, as in Ex. 18. For the other two, we use the generalized covariance ring with relational values, as in Ex. 19. Similarly, the queries over *Housing* (*Favorita*) use the covariance rings of degree 26 (15).

For DBT and 1-IVM, we use queries that compute scalar sum aggregates in the covariance matrix. When considering all variables as continuous, we use one query per dataset to compute $1 + n + \frac{n(n+1)}{2}$ sums, where n is the number of variables; for *Housing*, *Retailer*, and *Favorita*, we compute 378, 820, and 136 sums, respectively. When

		F-IVM	DBT	1-IVM
CONT	Housing	11,570.2 (7)	953.7 (626)	0.7 (384)
	Retailer	3,818.8 (9)	9.1* (3,186)	28.8 (825)
	Favorita	1,411.3 (9)	33.2* (615)	182.0 (142)
MIXED	Housing	996.4 (7)	682.6 (599)	1.3 (375)
	Retailer	1,255.8 (9)	7.2* (3,144)	21.7* (819)
	Favorita	354.0 (9)	18.3* (535)	87.2 (130)

Fig. 18 The average throughput (in thousands of tuples/sec) and in parentheses the number of materialized views (including input relations) for the maintenance of the covariance matrix over datasets where features are treated as all continuous (CONT) and as a mix of continuous and categorical (MIXED). The symbol * denotes the one-hour timeout

considering continuous and categorical variables, we use a batch of group-by aggregate queries as input to DBToaster. For *Housing*, *Retailer*, and *Favorita*, the number of queries with distinct group-by variables is 46, 22, and 79, respectively.

– *Q-Hierarchical Queries:* We use the natural joins of all relations in each dataset. For *Housing*, this is a star join query. For *Favorita*, the relation `Stores` violates the *q*-hierarchical property: Its variables form a strict subset of a root-to-leaf path in the canonical variable order. To ensure constant time for single-tuple updates, we require `Stores` to be non-updatable. For *Retailer*, the query is *q*-hierarchical due to (1) the functional dependency `zip → location` in `Census` and (2) requiring the relation `Item` be non-updatable.

– *k-Path Queries:* These queries join *k* copies R_1 to R_k of the edge relation of the input graph:

```
SELECT A₁, Aₖ₊₁, SUM(W₁ * ... * Wₖ)
FROM R₁ NATURAL JOIN ... NATURAL JOIN Rₖ
GROUP BY A₁, Aₖ₊₁
```

Each relation R_i has schema (A_i, A_{i+1}, W_i) and can be seen as the adjacency matrix of the input graph, with rows indexed by A_i, columns indexed by A_{i+1}, and the value $W_i = 1$ in cell (A_i, A_{i+1}). The path query is then the *k* times multiplication of the adjacency matrix.

Experimental Setup. We run all experiments on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 188GB RAM, with Debian 10. F-IVM, DBT, and 1-IVM use the DBToaster v2.3 backend, which generates single-threaded C++ code compiled using g++ 8.3.0 with the -O3 flag. Unless stated otherwise, we set a one-hour timeout on each experiment and report wall-clock times by averaging three best results out of four runs. We profile memory utilization using gperftools, excluding the memory used for storing input streams.

9.2 Covariance matrix and linear regression

We benchmark the performance of maintaining a covariance matrix for learning regression models over natural joins. We consider updates to all input relations. We compute the covariance matrix over all non-join variables of the join query (i.e., over all non-join attributes in 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 [50]. This is achieved by specializing the convergence step in batch gradient descent to the relevant restriction of the covariance matrix. In our approach for learning linear regression models over database joins, the convergence step takes orders of magnitude less time compared to the data-dependent covariance matrix computation.

Figure 18 shows the number of views materialized by F-IVM, DBT, and 1-IVM for computing the covariance matrix. F-IVM computes one aggregate query with payloads from a covariance ring. For *Housing*, where all relations join on one variable, F-IVM materializes seven views: one view per relation to marginalize out all non-join variables, and the root view to join these views. For *Retailer*, F-IVM materializes five views over the input relations, three intermediate views, and the root view; similarly, for *Favorita*. These views have payloads from the continuous (generalized) covariance ring if all features are continuous (continuous and categorical).

DBT and 1-IVM maintain a batch of sum aggregate queries with scalar payloads. These materialization strategies fail to effectively share the computation of covariance aggregates, materializing linearly many views in the size of the covariance matrix: for instance, when considering all variables as continuous, DBT and 1-IVM materialize 626 and respectively 384 views to maintain 378 scalar aggregates for *Housing*; similar reasoning holds for the other datasets and the scenarios with both continuous and categorical variables.

Throughput. Fig. 19 shows the throughput of F-IVM, DBT, and 1-IVM as they process an increasing fraction of the stream of tuple inserts. Figure 18 shows their average throughput after processing the entire stream. The throughput is higher when all features are continuous than for a mix of continuous and categorical features. This is expected as the latter computes additional group-by aggregates for the categorical features; in this case, the number of computed aggregates is data-dependent. The occasional hiccups in the throughput of F-IVM are due to doubling the memory allocated to the underlying data structures used for the views.

The query for *Housing* joins all relations on the common variable, which is the root in our variable order; thus, the query is hierarchical. F-IVM computes the covariance matrix using the query with no free variables in both scenarios (CONT and MIXED) and can process a single-tuple update to any input relation in time linear in the size of the payload. In the continuous-only scenario, the update time is $\mathcal{O}(m^2)$, where m is the number of continuous features; in the mixed scenario, the update time depends on the size of the domain of the categorical features. DBT exploits the conditional independence in the derived deltas to materialize each input relation separately such that all non-join variables are aggregated away. In the case of all continuous features, each materialized view has $\mathcal{O}(1)$ maintenance cost per update tuple, but the large number of views in DBT is the main reason for its poor performance. 1-IVM stores entire tuples of the input relations including non-join variables. On each update, 1-IVM recomputes a batch of aggregates on top of the join of these input relations and the update tuple. Since the update tuple binds the value of the common join variable, the hypergraph of the delta query consists of disconnected components. DBToaster first aggregates over each relation and then joins together the partial aggregates on the common variable. Even with this optimization, 1-IVM takes time linear in the size of the dataset, which explains its poor performance.

For *Retailer*, the inserts are mostly into `Inventory`. Since the variables of this relation form a root-to-leaf path in the variable order, F-IVM can process single-tuple updates to this relation in $\mathcal{O}(1)$ time in data complexity in the continuous-only scenario. DBT maintains up to four views per scalar aggregate and fails to process the entire stream within a one-hour limit in both scenarios. 1-IVM maintains one view per scalar group-by aggregate but recomputes the delta query on each update, resulting in 132x (58x) lower throughput than F-IVM in the continuous-only (mixed) scenario.

For *Favorita*, 1-IVM achieves better performance than on *Retailer* but still 7.8x (4.1x) slower than F-IVM in the continuous (mixed) scenario. DBT fails to finish the entire stream within a one-hour timeout.

Memory consumption. Fig. 19 shows that F-IVM achieves lower or comparable memory utilization on the three datasets, while providing orders of magnitude better performance than its competitors. The reason behind this memory efficiency is that F-IVM uses compound aggregates and factorization structures to express the covariance matrix computation over fewer views compared to DBT and 1-IVM.

End-to-end training. We next analyze the cost of learning linear regression models from the computed covariance matrices. We consider the scenario where all variables are continuous and the target label is house price (*Housing*), inventory units (*Retailer*), and sold units (*Favorita*). Using batch gradient descent and the covariance matrix, the time needed to converge on the model parameters represents 0.24%, 0.2%, and 0.001% of the time needed to process all updates in the stream for *Housing*, *Retailer*, and *Favorita*, respectively.

Figure 20 illustrates the performance of F-IVM for maintaining the covariance matrix in three scenarios: 1) without training the linear regression model (baseline); 2) with train-
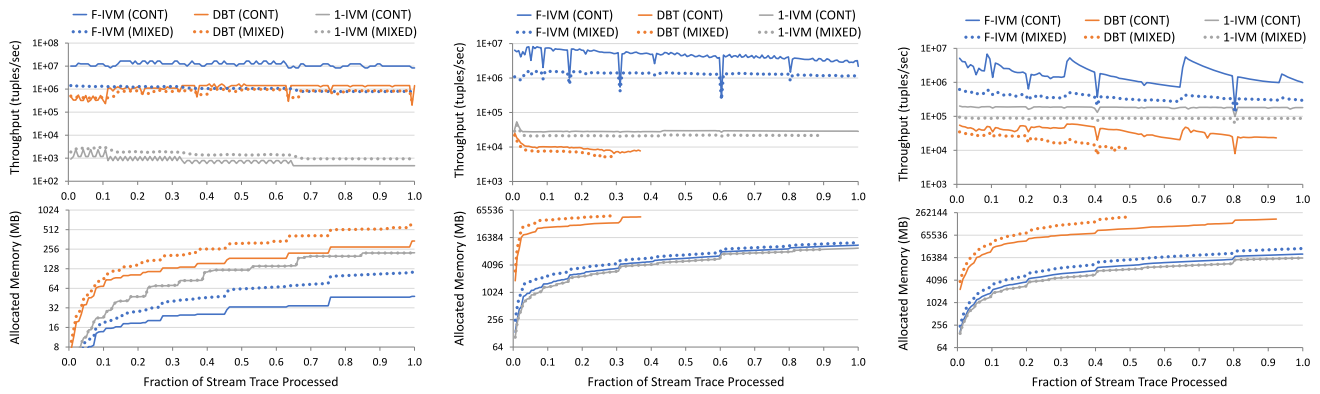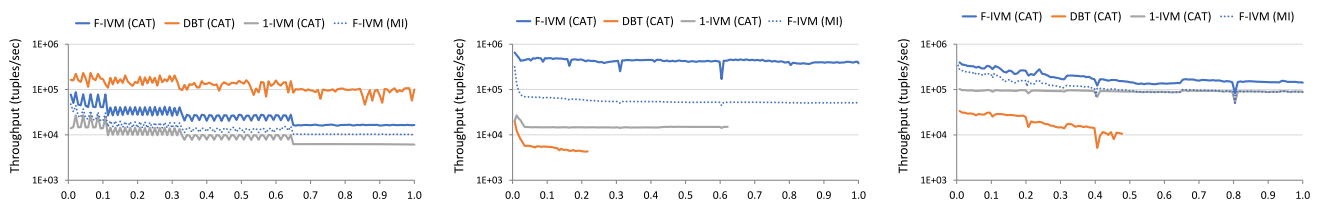
Fig. 19 Incremental maintenance of the covariance matrix over the *Housing* dataset (left), *Retailer* dataset (middle), and *Favorita* dataset (right) under updates of size 1000 to all relations with a one-hour time-out. The CONT plots consider all features as continuous, while the MIXED plots consider a mix of continuous and categorical features
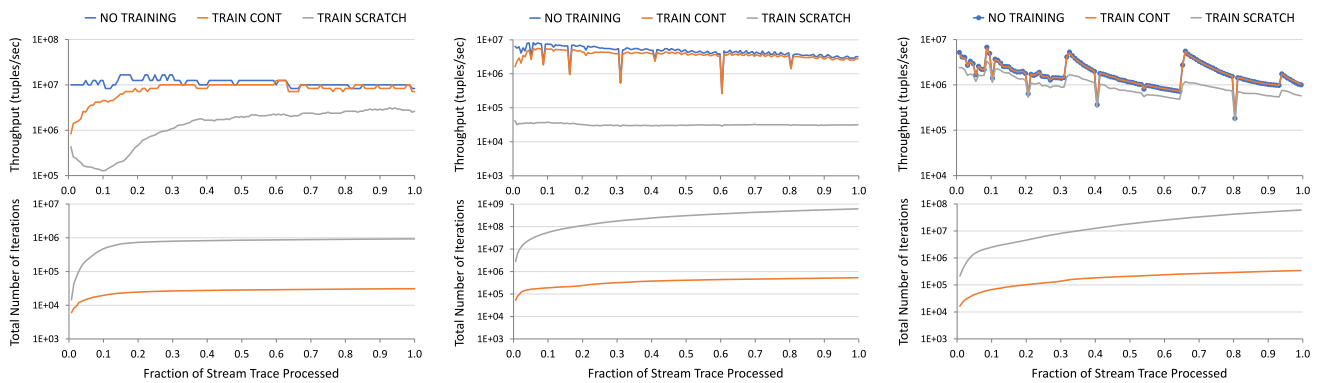
Fig. 20 Maintaining linear regression models over the *Housing* dataset (left), *Retailer* dataset (middle), and *Favorita* dataset (right) under updates of size 1000 to all relations using F-IVM. Batch gradient descent is invoked after every update using previously learned parameters (TRAIN CONT) and parameters set to 0 (TRAIN SCRATCH). The NO TRAINING plots show the time to compute the covariance matrices only. The bottom charts show the cumulative numbers of iterations used by the batch gradient descent during the training phase
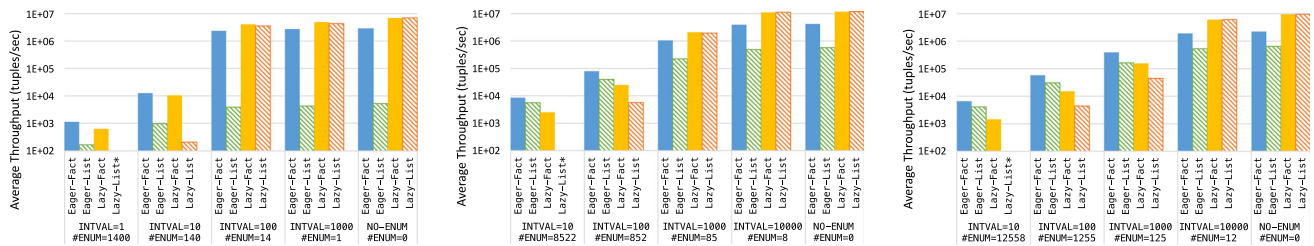

Fig. 21 Solid lines: incremental maintenance of the covariance matrix over the *Housing* (left), *Retailer* (middle) and *Favorita* (right) datasets under batches of 1000 updates to all relations with a one-hour time-out. All features are either categorical in the original dataset or made categorical by discretizing their domains into 100 buckets. Dotted line: computation of the mutual information matrix and the Chow-Liu tree on top of the covariance matrix after each batch of 1000 updates

ing after each batch update, starting from previously learned parameters (CONT); and 3) with training after every batch update, starting with value 0 for the parameters (SCRATCH). Continuously refreshing the model after every update reduces the throughput of baseline by 41%, 18%, and 2% for *Housing*, *Retailer*, and *Favorita*. In contrast, retraining from scratch after every update has significantly higher overheads and reduces the baseline throughput by 95%, 99%, and 42% for

the three datasets. Decreasing the training frequency brings the throughput closer to the baseline.

Figure 20 (bottom plots) shows for each dataset the cumulative number of iterations of batch gradient descent in the two training scenarios. Continuously improving learned parameters yields 30x, 1160x, and 175x fewer iterations compared to retraining from scratch for *Housing*, *Retailer*, and

Fig. 22 Incremental maintenance of the result of the *q*-hierarchical queries over the *Housing* (left), *Retailer* (middle), and *Favorita* (right) datasets under update batches and requests to enumerate all tuples in the query results after every INTVL update batches; #ENUM denotes the overall number of the enumeration requests. The symbol ∗ denotes the case where an IVM variant did not finish within the time limit (50 h) for this experiment. The throughput is not shown in this case

Favorita, respectively. This reflects in the throughput of the two training scenarios.

9.3 Mutual information and chow-liu trees

We benchmark the performance of maintaining the matrix of pairwise mutual information (MI) for the features representing the non-join variables in our datasets and Chow-Liu trees on top of the MI matrices.

As explained in Sect. 8.2, the MI matrix can be derived from the covariance matrix over categorical variables. We discretize the active domain of each continuous variable into 100 bins of equal size. The *Housing*, *Retailer*, and *Favorita* datasets have 26, 39, and 15 categorical variables, respectively. Insertions and deletions of values for a continuous variable are distributed into the appropriate bins, without changing the number of bins. Whereas the covariance matrix can be maintained incrementally under updates, the MI matrix needs to be recomputed from scratch after each update batch.

The view construction and maintenance are as in Sect. 9.2, except that all variables are now categorical. Figure 21 (solid lines) shows the throughput of F-IVM, DBT, and 1-IVM for maintaining the covariance matrix as they process an increasing fraction of the stream of tuple updates. F-IVM is 74x faster than DBT and 28x faster than 1-IVM for the *Retailer* dataset and 9.3x and 2.1x faster, respectively, for *Favorita*. For *Housing*, F-IVM is 2.8x faster than 1-IVM but 4.6x slower than DBT. This is because: (i) *Housing* is a relatively small dataset and the domain of the categorical variables is also small; (ii) DBT has specific optimizations for group-by count over star joins such as in this case.

Computing the MI matrix from the covariance matrix takes time linear in the number of categories of the variables. Computing the Chow-Liu tree takes time $\mathcal{O}(m \log m)$, where m is the number of variables. Figure 21 (dotted line) shows the throughput of F-IVM when the MI matrix and Chow-Liu tree are computed after each update batch. This throughput is 46%, 86%, and 35% smaller than the time to maintain the covariance matrix for *Housing*, *Retailer*, and *Favorita*, respectively.

9.4 *Q*-hierarchical queries

We would like to understand how different IVM variants perform for the three *q*-hierarchical queries from Sect. 9.1. We construct for each query a view tree modeled on the canonical free-top variable order. The query result is constructed and maintained in the payload space. We consider two dimensions for this experiment. One dimension is whether we push the updates all the way to the result (eager) or we only update the input relations and only construct the query result on an enumeration request (lazy). The other dimension is whether the query result has a listing representation (one tuple after the other) or a factorized representation. This defines four variants: eager-list (which is DBT), eager-fact (F-IVM's default strategy), lazy-list (1-IVM), and lazy-fact (a hybrid of F-IVM and 1-IVM).

Figure 22 shows the average throughput of the four variants for the three queries. We report the overall runtime for update batches as in the previous experiments, but where we also have requests for the enumeration of all tuples in the query result after every INTVAL batches of updates. We tried INTVAL values 1, 10, 100, 1000, and 10000. Each such value corresponds to different numbers of enumeration requests (#ENUM) as the datasets have different sizes. The lazy-list variant did not finish within the time limit of 50 h (denoted by ∗ in Fig. 22). The lazy-list variant has the lowest throughput among the four variants in our experiment.

The two lazy variants are clear winners in case of none or very few enumeration requests. In this case, there is almost no difference between their throughputs since they spend most of their time updating the input relations. In case of more enumeration requests, however, the eager variants are the winners, with eager-fact consistently outperforming eager-list.

Overall, the eager and lazy variants based on factorized representation outperform those based on listing representation in all but the trivial cases of none or few enumeration
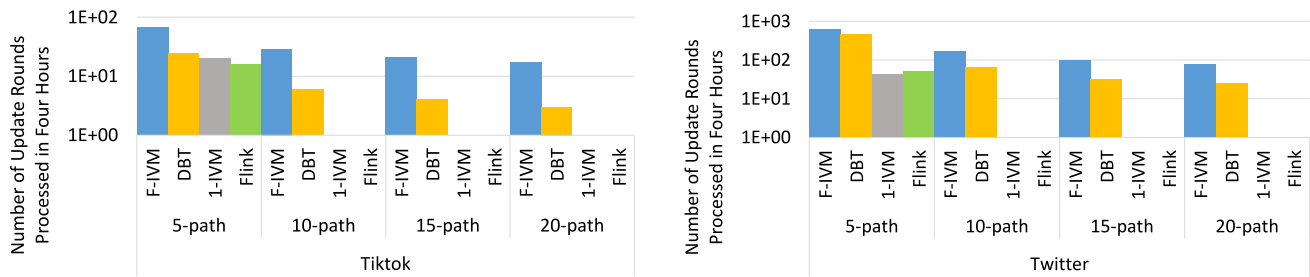
Fig. 23 The number of rounds of updates to all relations in the path queries processed within four hours over the *TikTok* (left) and *Twitter* (right) graph datasets under update batches of size 1000

requests, where the representation of the query result plays no role. This is as expected, since the enumeration delay and the update time can both remain constant for our queries only if the query result is kept factorized over the views in the view tree.

9.5 Path queries

We investigate the scalability of the maintenance approaches as we increase the number of joins in the query. We consider the path query with up to 20 self-joins of the edge relation in the *TikTok* and *Twitter* graphs. The edges are partitioned into batches of 1000 inserts. One round of updates processes one batch of inserts for each copy of the edge relation. Figure 23 shows the number of rounds of updates processed by each approach within four hours. F-IVM outperforms all other approaches on path queries of any length. All approaches are slower for *TikTok*, since it is more skewed than *Twitter*.

Flink and 1-IVM have a similar poor performance and do not scale for long path queries. Flink maintains the join result via a left-deep binary view tree and computes the aggregates at the root view. It projects away the join variable after each join. This reduces the number of columns but not the number of rows in the join result. For a delta to the bottom relation in the view tree, Flink joins it with all other $k-1$ relations in the query. This triggers $\mathcal{O}(N^{\lceil \frac{k}{2} \rceil})$ inserts to the join result, where N is the number of edges. 1-IVM computes the delta query by joining the batch of inserts with $k-1$ relations and has the same complexity as Flink.

DBT and F-IVM avoid the materialization of the large join result by pushing the aggregates past the joins at each view. Both of them need $\mathcal{O}(N^2)$ time to update each view. Like Flink, F-IVM uses a left-deep view tree. DBT uses one view tree per delta query, where the delta relation is a child of the top view and the two subqueries to the left and right of the delta relation have left-deep view trees. F-IVM constructs fewer views than DBT: For 20-path, F-IVM uses 19 views, while DBT uses 190 views. This explains the better performance of F-IVM.

10 Related work

To the best of our knowledge, ours is the first approach to propose factorized IVM for a range of distinct applications. It extends non-trivially two lines of prior work: higher-order delta-based IVM and factorized computation of in-database analytics.

Our view language is modeled on functional aggregate queries over semirings [3] and generalized multiset relations over rings [32]; the latter allowed us to adapt DBToaster to factorized IVM.

IVM. IVM is a well-studied area spanning more than three decades [15, 57, 63]. Prior work extensively studied IVM for various query languages and showed that the time complexity of IVM is lower than of recomputation. We go beyond prior work on higher-order IVM for queries with joins and aggregates, as realized in DBToaster [32], and propose a unified approach for factorized computation of aggregates over joins [5], factorized incremental computation of linear algebra [44], and in-database machine learning over database joins [58]. DBToaster uses one materialization hierarchy per relation in the query, whereas F-IVM uses one view tree for all relations. DBToaster can thus have much higher space requirements and update times. As we observed experimentally, it does not consider the maintenance of composite aggregates such as the covariance matrix. IVM over array data [67] targets scientific workloads but without exploiting data factorization.

F-IVM over the relational payload ring strictly subsumes prior work on factorized IVM for acyclic joins [27] as it can support arbitrary joins. F-IVM has efficient support for free-connex acyclic [27] and q-hierarchical queries [9]. Exploiting key attributes to enable succinct delta representations and accelerate maintenance complements our approach [29]. Our framework generalizes the main idea of the LINVIEW approach [44] for maintaining matrix computation over arbitrary joins. Unlike approaches that exploit the *append-only* nature of data streams [64], F-IVM allows for both data insertions and deletions. F-IVM can be used to improve the memory-efficiency of systems that integrate IVM into compilers to speed up the search in abstract syntax

trees [6]. Such systems suffer from the high storage overhead of systems such as DBToaster that maintain significantly more views than F-IVM.

Commercial DBMSs support IVM for restricted classes of queries, e.g., Oracle [37] and SQLServer [18]. LogicBlox supports higher-order IVM for Datalog meta-programs [4, 25]. Trill is a streaming engine that supports incremental processing of relational-style queries but no complex aggregates like covariance matrices [13]. Differential Dataflow [39] supports incremental processing for programs with recursion. There is a distinct line of work on maintenance for recursive Datalog [41].

Static In-DB analytics. The emerging area of in-database analytics has been overviewed in two tutorials [34, 54] and a recent keynote [47]. Several systems support analytics over normalized data via a tight integration of databases and machine learning [26, 34, 40, 54, 55]. Other systems integrate with R to enable in-situ data processing using domain-specialized routines [11, 66]. The closest in spirit to our approach is work on learning models over factorized joins [30, 50, 56, 58], pushing ML tasks past joins [22, 59] and on in-database linear algebra [14, 20, 51], yet they do not consider incremental maintenance.

Learning. There is a wealth of work in the ML community on incremental or online learning over *arbitrary* relations [60]. Our approach learns over *joins* and crucially exploits the join dependencies in the underlying training dataset to improve the runtime performance.

11 Conclusion and future work

This article introduces F-IVM, a system that unifies the task of maintaining a variety of analytics over normalized data under updates. We show its applicability to learning linear regression models, building Chow-Liu trees, and query evaluation with listing/factorized result representation. F-IVM recovers the best known complexities for free-connex acyclic and q-hierarchical queries. A prior version of this work [46] also discusses the application of F-IVM to matrix chain multiplication. These tasks use the same computation paradigm that factorizes the representation and the computation of the keys, the payloads, and the updates. Their differences are confined to the definition of the sum and product operations in a suitable ring. F-IVM is publicly available and was implemented as an extension of DBToaster [32], a state-of-the-art system for incremental maintenance, and shown to outperform competitors by orders of magnitude in both time and space.

Going forward, we would like to apply this approach to further tasks such as inference in probabilistic graphical models and more complex machine learning tasks.

F-IVM inherits the limitations of DBToaster, in particular it is single-threaded. A promising avenue of research is to build F-IVM on top of an open-source parallel and distributed framework such as Apache Flink. Another goal is to extend F-IVM to support further SQL operators such as theta joins, nested subqueries, and NULLs, which are relevant in practice.

> *If You Liked It, Then You Should Put A Ring On It.*
> *– Beyoncé.*

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., et al.: The design of the borealis stream processing engine. CIDR **5**, 277–289 (2005)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995). http://webdam.inria.fr/Alice/
3. Abo Khamis, M., Ngo, H.Q., Rudra, A.: FAQ: Questions asked frequently. In: PODS, pp. 13–28 (2016)
4. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the LogicBlox system. In: SIGMOD, pp. 1371–1382 (2015)
5. Bakibayev, N., Kociský, T., Olteanu, D., Závodnỳ, J.: Aggregation and ordering in factorised databases. PVLDB **6**(14), 1990–2001 (2013)
6. Balakrishnan, D., Nuessle, C., Kennedy, O., Ziarek, L.: TreeToaster: towards an IVM-optimized compiler. In: Li, G., Li, Z., Idreos, S., Srivastava, D. (eds) SIGMOD, pp. 155–167 (2021)
7. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. J. ACM **30**(3), 479–513 (1983)
8. Berkholz, C., Gerhardt, F., Schweikardt, N.: Constant delay enumeration for conjunctive queries: a tutorial. ACM SIGLOG News **7**(1), 4–33 (2020)
9. Berkholz, C., Keppeler, J., Schweikardt, N.: Answering conjunctive queries under updates. In: PODS, pp. 303–318 (2017)
10. Brault-Baron, J.: De la pertinence de l'énumération: Complexité en logiques propositionnelle et du premier ordre. Ph.D. thesis, Université de Caen (2013)
11. Brown, P.G.: Overview of SciDB: large scale array storage, processing and analysis. In: SIGMOD, pp. 963–968 (2010)

12. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink^{TM}: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)

13. Chandramouli, B., Goldstein, J., et al.: Trill: a high-performance incremental query processor for diverse analytics. PVLDB **8**(4), 401–412 (2014)

14. Chen, L., Kumar, A., Naughton, J.F., Patel, J.M.: Towards linear algebra over normalized data. Proc. VLDB Endow. **10**(11), 1214–1225 (2017)

15. Chirkova, R., Yang, J.: Materialized views. Found. Trends DB **4**(4), 295–405 (2012)

16. Chow, C., Liu, C.: Approximating discrete probability distributions with dependence trees. Trans. Inf. Theor. **14**(3), 462–467 (2006)

17. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (2009)

18. Create Indexed Views. http://msdn.microsoft.com/en-us/library/ms191432.aspx

19. Durand, A., Grandjean, E.: First-order queries on structures of bounded degree are computable with constant delay. TOCL **8**(4), 21 (2007)

20. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for large-scale machine learning. PVLDB **9**(12), 960–971 (2016)

21. Favorita Dataset. https://www.kaggle.com/competitions/favorita-grocery-sales-forecasting

22. Feng, X., Kumar, A., Recht, B., Ré, C.: Towards a unified architecture for in-RDBMS analytics. In: SIGMOD, pp. 325–336 (2012)

23. Gottlob, G., Leone, N., Scarcello, F.: Hypertree decompositions and tractable queries. In: PODS, pp. 21–32 (1999)

24. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS, pp. 31–40 (2007)

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

26. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., et al.: The MADlib analytics library or MAD Skills, the SQL. PVLDB **5**(12), 1700–1711 (2012)

27. Idris, M., Ugarte, M., Vansummeren, S.: The dynamic Yannakakis algorithm: compact and efficient query processing under updates. In: SIGMOD, pp. 1259–1274 (2017)

28. Kara, A., Nikolic, M., Olteanu, D., Zhang, H.: Trade-offs in static and dynamic evaluation of hierarchical queries. In: PODS, pp. 375–392 (2020)

29. Katsis, Y., Ong, K.W., Papakonstantinou, Y., Zhao, K.K.: Utilizing IDs to accelerate incremental view maintenance. In: SIGMOD, pp. 1985–2000 (2015)

30. Khamis, M.A., Ngo, H.Q., Nguyen, X., Olteanu, D., Schleich, M.: Learning models over relational data using sparse tensors and functional dependencies. ACM Trans. Database Syst. **45**(2), 7:1-7:66 (2020)

31. Koch, C.: Incremental query evaluation in a ring of databases. In: PODS, pp. 87–98 (2010)

32. Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., Shaikhha, A.: DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), 253–278 (2014)

33. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)

34. Kumar, A., Boehm, M., Yang, J.: Data management in machine learning: challenges, techniques, and systems. In: SIGMOD, pp. 1717–1722 (2017)

35. Kumar, A., Naughton, J.F., Patel, J.M.: Learning generalized linear models over normalized data. In: SIGMOD, pp. 1969–1984 (2015)

36. Madden, S.R., et al.: TinyDB: an acquisitional query processing system for sensor networks. TODS **30**(1), 122–173 (2005)

37. Materialized View Concepts and Architecture. http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmview.htm

38. McAuley, J., Leskovec, J.: Learning to discover social circles in ego networks. In: NIPS, pp. 539–547 (2012)

39. McSherry, F., Murray, D.G., Isaacs, R., Isard, M.: Differential dataflow. In: CIDR (2013)

40. Meng, X., et al.: MLlib: machine learning in apache spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)

41. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Maintenance of datalog materialisations revisited. Artif. Intell. **269**, 76–136 (2019)

42. Murphy, K.P.: Machine learning: a probabilistic perspective. MIT Press, Cambridge (2013)

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

44. Nikolic, M., Elseidy, M., Koch, C.: LINVIEW: incremental view maintenance for complex analytical queries. In: SIGMOD, pp. 253–264 (2014)

45. Nikolic, M., Olteanu, D.: Incremental view maintenance with triple lock factorization benefits. CoRR arXiv:1703.07484 (2017)

46. Nikolic, M., Olteanu, D.: Incremental view maintenance with triple lock factorization benefits. In: SIGMOD, pp. 365–380. ACM (2018)

47. Olteanu, D.: The relational data borg is learning. Proc. VLDB Endow. **13**(12), 3502–3515 (2020)

48. Olteanu, D., Huang, J., Koch, C.: SPROUT: Lazy vs. eager query plans for tuple-independent probabilistic databases. In: ICDE, pp. 640–651 (2009)

49. Olteanu, D., Koch, C., Antova, L.: World-set decompositions: expressiveness and efficient algorithms. Theor. Comput. Sci. **403**(2–3), 265–284 (2008)

50. Olteanu, D., Schleich, M.: F: Regression models over factorized views. PVLDB **9**(13), 1573–1576 (2016)

51. Olteanu, D., Vortmeier, N., Zivanovic, D.: Givens QR decomposition over relational databases. In: SIGMOD, pp. 1948–1961 (2022)

52. Olteanu, D., Závodnỳ, J.: Size bounds for factorised representations of query results. TODS **40**(1), 2:1-2:44 (2015)

53. Peña-Fernández, S., Ureta, A., Gras, J.: Current affairs on TikTok. Virality and entertainment for digital natives. El Profesional de la Informacion **31**, 12 (2022)

54. Polyzotis, N., Roy, S., Whang, S.E., Zinkevich, M.: Data management challenges in production machine learning. In: SIGMOD, pp. 1723–1726 (2017)

55. Qin, C., Rusu, F.: Speculative approximations for terascale distributed gradient descent optimization. In: DanaC, pp. 1–10 (2015)

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

57. Salem, K., Beyer, K.S., Cochrane, R., Lindsay, B.G.: How to roll a join: asynchronous incremental view maintenance. In: SIGMOD, pp. 129–140 (2000)

58. Schleich, M., Olteanu, D., Ciucanu, R.: Learning linear regression models over factorized joins. In: SIGMOD, pp. 3–18 (2016)

59. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: SIGMOD, pp. 1642–1659 (2019)

60. Shalev-Shwartz, S., et al.: Online learning and online convex optimization. Found. Trends ML **4**(2), 107–194 (2012)

61. Sidiropoulos, N.D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E.E., Faloutsos, C.: Tensor decomposition for signal processing and machine learning. Trans. Sig. Proc. **65**(13), 3551–3582 (2017)

62. Suciu, D., Olteanu, D., Ré, C., Koch, C.: Probabilistic databases. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2011)

63. Tang, D., Shang, Z., Elmore, A.J., Krishnan, S., Franklin, M.J.: Thrifty query execution via incrementability. In: SIGMOD, pp. 1241–1256 (2020)

64. Yang, Y., Golab, L., Özsu, M.T.: ViewDF: declarative incremental view maintenance for streaming data. Inf. Syst. **71**, 55–67 (2017)

65. Yannakakis, M.: Algorithms for acyclic database schemes. In: VLDB, pp. 82–94 (1981)
66. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)
67. Zhao, W., Rusu, F., Dong, B., Wu, K., Nugent, P.: Incremental view maintenance over array data. In: SIGMOD, pp. 139–154 (2017)