Factorized and Vectorized Execution: Optimizing Analytical and Semantic Queries over Relations

SUNNY YASSER, Polytechnique Montréal, Canada

ANAS DORBANI, Polytechnique Montréal, Canada

AMINE MHEDHBI, Polytechnique Montréal, Canada

Many-to-many joins are central to analytical and semantic workloads such as fraud detection, network analysis, and recommendation, where insights arise from relationships between entities. These workloads often suffer from an explosion of intermediate results, sometimes orders of magnitude larger than the inputs. Factorized representations address this problem by exploiting conditional independence among attributes to encode intermediates more compactly. In some cases, they can reduce the output size asymptotically below the worst-case output size. However, adopting factorization in modern vectorized query processors remains challenging: factorized representations are hierarchical, whereas vectorized execution is built around flat, block-oriented processing. Prior approaches either rely on full materialization or support only restricted factorization layouts, sacrificing much of the benefits of both factorization and vectorization.

We present FFX, a novel engine for **F**ast **F**actorized e**X**ecution. FFX is the first pipelined engine to support arbitrary factorization schemes while preserving full vectorization. The engine introduces packed factorized vectors and operators that maintain cache-friendly, contiguous layouts. Beyond analytics, FFX also co-optimizes semantic operators by serializing factorized intermediates into compact prompts for large language models (LLMs), substantially reducing token usage and inference cost while maintaining output quality and, in some cases, improving it. Together, these contributions enable efficient execution of join-heavy analytical queries, including queries augmented with semantic operators.

CCS Concepts: • **Information systems → Query operators**; **Join algorithms**; **DBMS engine architectures**.

Additional Key Words and Phrases: Factorization, Vectorization, Execution, Analytics, Semantic Processing, Language Models, Operators

1 Introduction

Many-to-many joins underpin a wide range of analytical workloads, in which insights arise from the structure of relationships between entities. A common analytical task is the enumeration of network substructures, such as common neighbourhoods and paths. For example, follower patterns inform social recommendations [14], while path queries trace traffic flow in computer networks [35]. Beyond analytics, these substructures are increasingly used as context for LLM inference in tasks such as recommendation [28] and network diagnosis [47].

Most networks are stored as records in RDBMSs, where enumerating substructures translates into complex multi-way joins. However, join evaluation often becomes a scalability bottleneck because

Authors' Contact Information: Sunny Yasser, Polytechnique Montréal, Montréal, QC, Canada, sunny.yasser@polymtl.ca; Anas Dorbani, Polytechnique Montréal, Montréal, QC, Canada, anas.dorbani@polymtl.ca; Amine Mhedhbi, Polytechnique Montréal, Montréal, QC, Canada, amine.mhedhbi@polymtl.ca.

intermediate results can exceed input sizes by orders of magnitude. As a result, this large size limits the efficiency of both traditional RDBMSs and even specialized graph analytics frameworks [41]. The problem is even more severe in emerging systems that interleave data analytics with semantic reasoning using LLMs [10–12, 19, 42]. In these systems, intermediate relations must be serialized as text and included in LLM prompts, making inference cost directly proportional to their size. This motivates our focus on join-heavy queries, including those augmented with LLM-powered semantic operators.

A promising approach to mitigating the explosion in intermediate result size is to use factorized representations [36]. They rewrite a multiset of flat tuples as a nested expression over unions and products by exploiting the distributivity of product over union, along with the commutativity of both operators. This representation factors tuples according to shared attribute-value prefixes: unions enumerate alternative values, while products combine subexpressions that vary independently under a shared prefix. For example, the tuples $(a_1, a_2, a_3) \in \{(0, 1, 4), (0, 2, 4), (0, 3, 4), (0, 1, 5),$ $(0, 2, 5), (0, 3, 5), (0, 1, 6), (0, 2, 6), (0, 3, 6)\}$ can be rewritten more succinctly as $(\cup_{a_1}\{0\}\times\cup_{a_2}\{1, 2, 3\}\times$ $\cup_{a_3}\{4, 5, 6\})$. Factorization can yield representations whose size is asymptotically smaller than the maximum possible output size, known as the AGM bound [2].

Despite these theoretical advantages, adopting factorization in vectorized query engines remains challenging. Factorization organizes results hierarchically, grouping variable-length sets of values, whereas vectorized execution processes flat blocks of tuples for CPU-efficient execution over contiguous memory. A second challenge is maintaining state dynamically. Selections at one level may propagate to dependent values, causing *cascading updates* across multiple levels of the hierarchy. This behaviour contrasts with vectorized execution, where tuple reduction is typically confined to a single selector. Existing systems such as FDB [3, 4] sidestep cascading updates by fully materializing fresh factorized results at each operator, at the cost of higher memory utilization and the loss of pipelining and cache-efficient processing. Other systems restrict factorization to simplify the engine, sacrificing much of the potential performance benefit [15, 18].

1.1 Existing Vectorized Approaches

Prior systems demonstrate the benefits of factorization, but remain limited in how they adopt it in vectorized pipelines. Kuzu [18] and Graphflow [21] adopt list-based processing (LBP) [15], which is pipelined but supports only restricted factorization layouts and yields sparsely populated vectors with high interpretation overhead. DuckPGQ [45, 46] exploits factorization for specialized optimizations (*e.g.*, early aggregation), rather than as a general intermediate representation propagated across operators. What is missing is a unified execution model that treats intermediates as native factorized, packed vectors within the pipeline. Such a model preserves both compactness (smaller intermediates) and vectorization (cache-efficient CPU execution). *Bridging these paradigms for join-heavy analytical queries with semantic operators is the central goal of this work.*

Predicate-transfer techniques [6, 44, 48, 50] make Yannakakis-style processing practical [49] by pruning dangling tuples early, thereby improving the robustness of query processing. However, they still enumerate large join results as flat tuples and do not propagate factorized intermediates between operators. Factorization and predicate-transfer are complementary: pruning reduces intermediates, while factorization compactly represents the output without flattening.

We argue that *passing factorized intermediates is essential for end-to-end co-optimization*. In this work, this approach optimizes semantic processing by allowing factorized join intermediates to flow directly into LLM-powered semantic operators. More broadly, the same idea could support other workloads, such as gradient boosting [17], group-by-over-join analytics for decision trees and Rk-means [43], and join-free feature engineering [23].

	Flat	FDB	LBP	FFX
Pipelined execution	✓	✗	✓	✓
Fully packed vectors	✓	✗	✗	✓
Factorized intermediates	✗	✓	✓	✓
Supports arbitrary f-trees	✗	✓	✗	✓

Table 1. Comparison of FFX with prior execution models. We use ✓/✗ to denote support.

1.2 Contributions

We present FFX[1], a novel query engine for **F**ast **F**actorized e**X**ecution. FFX co-optimizes join-heavy analytics and LLM-powered semantic operators through the following contributions:

- *Packed factorized vectors.* (§4) We propose a new representation for intermediates that supports arbitrary factorization while enabling fully packed, contiguous in-memory vectors.
- *Vectorization with graceful fallback.* (§5.1) We adapt operators to execute over packed factorized vectors in tight loops without pointer chasing. When factorization offers limited benefit, FFX falls back to standard vectorized execution without overhead.
- *Cascade update operator.* (§5.2) We propose a new operator to propagate updates, such as tuple reductions, across the factorized hierarchy without breaking vectorized execution.
- *Factorized semantic processing.* (§6) We design LLM-powered operators that consume serialized factorized vectors and materialize Cartesian combinations as part of their output. This substantially reduces input token counts and, in turn, inference cost.

Table 1 compares FFX with prior work. Our experiments (§7) demonstrate that factorized and vectorized execution are complementary. FFX achieves major speedups on heavy-join workloads over state-of-the-art systems and maintains comparable or improved accuracy for semantic operators over flat representations while reducing inference cost.

2 Background

FFX builds on three foundations: vectorized execution, factorized query processing, and semantic query operators. We provide the necessary background on each below.

2.1 Vectorized Execution

Modern analytical DBMSs widely employ *vectorized execution* for query processing, as implemented in systems such as Photon [5], Apache DataFusion [25], Velox [38], DuckDB [39], and DB2 [40]. Unlike traditional tuple-at-a-time processing in Volcano [13] or full materialization in MonetDB [7], vectorized engines follow a *block-at-a-time* model in which operators process small, fixed-size batches of tuples in a columnar layout. Pioneered by MonetDB/X100 [8] and VectorWise [51], vectorized execution evaluates fixed-size contiguous vectors in tight loops. This design is well suited to modern superscalar CPUs: it yields cache-friendly execution, amortizes interpretation overhead, hides memory latency, reduces branching, and thereby improves instructions per cycle [22].

2.2 Factorized Query Processing

Factorized representations mitigate the intermediate-result explosion in many-to-many joins by compactly representing join outputs [37]. They exploit the conditional independence induced by joins to express results as algebraic combinations of products and unions or, equivalently, as vertical and horizontal partitions. By avoiding or delaying full Cartesian expansion, factorized representations eliminate redundancy and can reduce intermediate result size below the worst-case output bound, namely, the AGM bound. We introduce f-representations, the specific factorized

---

representation used in this work, and then provide an overview of representative factorized query engines. We also note that more compact representations exist, namely d-representations. However, such representations require materialization, whereas our focus is on pipelined execution.

2.2.1 F-representations. A relation over n attributes is typically represented as a multiset of flat n-ary tuples, *i.e.*, a union of tuples. An *f-representation* compresses this form into a nested algebraic expression by exploiting the distributivity of product over union and the commutativity of both operators. Specifically, an f-representation is a nested expression over unions, products, and attribute values. The expression has a trie-like structure that factors tuples by their shared attribute-value prefixes. Unions enumerate alternative values of an attribute, while products over unions combine subexpressions that vary independently given the current prefix of values (*i.e.*, along any root-to-leaf path, values under the same union share an attribute-value prefix). Flattening distributes products over unions, producing the Cartesian combinations needed to recover flat tuples.

We say that an f-representation is over an *f-tree* $\mathcal{T}$. An f-tree is a rooted tree specifying how attributes are grouped: each node corresponds to one attribute, and each parent-child relationship, indicated by an edge $x - y$, means that values of attribute y (the child) are stored grouped by a given value of attribute x (the parent), together with all ancestors of x. Equivalently, every root-to-leaf path $a_{i_1} - a_{i_2} - \cdots - a_{i_k}$ defines a nesting order in which a union of $a_{i_{j+1}}$ values appears under each fixed prefix assignment $(a_{i_1}, \ldots, a_{i_j})$. Branching in f-trees captures conditional independence: if a node x has children y and z, then the subrelations rooted at y and z are represented independently for each fixed assignment to the attributes on the path from the root to x. Intuitively, branching avoids enumerating Cartesian products between sibling substructures, because their combinations are implied by the product in the factorized expression rather than listed as flat tuples.

We illustrate f-representations and f-trees with a concrete example. For a formal treatment, we refer the reader to Olteanu *et al.* [37]. Consider the base relation $R(src, dst)$ in Fig. 1a and the query $Q_{2H} = R(a_1, a_2) \bowtie R(a_2, a_3)$. The output of Q_{2H} contains 13 flat tuples, for a total of 39 atomic values, with substantial repetition.

Figures 1d and 1f show factorized outputs of Q_{2H} over the *f-trees* $\mathcal{T}_1$ and $\mathcal{T}_2$. An f-tree specifies the grouping layout. For instance, $\mathcal{T}_1$ groups values as $a_1 - a_2 - a_3$, mirroring flat tuple enumeration except that repeated prefixes are removed, resulting in 22 atomic values. In contrast, $\mathcal{T}_2$ exploits conditional independence and groups values as $a_2 - a_1$ and $a_2 - a_3$, yielding a more compact intermediate encoding of 15 values. In general, more branching and smaller height imply greater compactness. We use the arrow notation and the visual tree diagrams interchangeably.

These are examples of data-independent factorization layouts: the same f-tree applies regardless of the specific input instance. Valid f-trees must satisfy the *path constraint* [37], which requires that any two dependent attributes, *i.e.*, attributes in the same relation or linked via a chain of join predicates with all join keys projected out, appear along the same root-to-leaf path.

2.2.2 Factorized Engines — FDB [3, 4]. FDB was the first engine to evaluate queries over f-representations. Like MonetDB [7], it fully materializes operator outputs. Each operator produces a fresh factorized result, represented as a multi-level trie, for the next operator to consume. This design avoids complex in-place hierarchical updates during joins and filters, but requires reconstructing full factorized intermediates at every operator. While factorization reduces the size of intermediates, the full-materialization model forfeits cache locality and pipelined execution, leading to excessive memory traffic from iterating over, copying, and rebuilding factorized intermediates.

Subsequent work further optimized FDB by refining the underlying data structures to reduce pointer chasing and improve locality through more contiguous memory layouts [29]. While these optimizations improve the efficiency of reading and writing factorized representations, the operators' execution model still relies on fully materializing intermediate results.
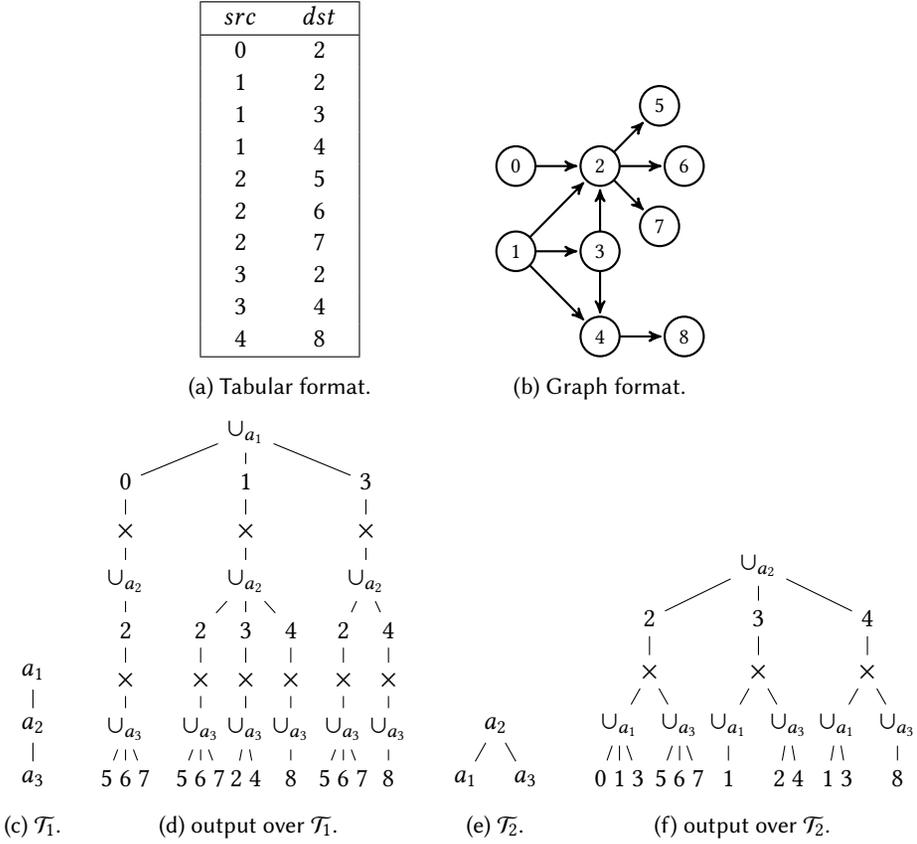
src	dst
0	2
1	2
1	3
1	4
2	5
2	6
2	7
3	2
3	4
4	8

(a) Tabular format.

(b) Graph format.

(c) $\mathcal{T}_1$.

(d) output over $\mathcal{T}_1$.

(e) $\mathcal{T}_2$.

(f) output over $\mathcal{T}_2$.

Fig. 1. Example relation $R(src, dst)$ in tabular and graph formats, and factorized outputs of $R(a_1, a_2) \bowtie R(a_2, a_3)$ over f-trees $\mathcal{T}_1$ and $\mathcal{T}_2$.

2.2.3 Factorized Engines — List-based Processing (LBP) [15].

LBP improves on FDB by adopting factorization into pipelined execution. Graph DBMSs such as Graphflow [21] and Kuzu [18] operate on *factorized vectors* [30] aligned to adjacency lists [31].

Existing implementations use worst-case-optimal join plans. Each plan is defined by a query attribute ordering. Query evaluation follows that ordering: it first enumerates candidate values for the first attribute, namely those common to the base relations that contain it, and then extends the bindings one attribute at a time by applying a binary or multi-way join over the relations involving the next attribute, conditioned on the bound values.

LBP allocates fixed-size vectors (*e.g.*, of size 2048), and each vector has a state containing its size, a position field (pos), and a selection array (sel). We explain the vector state shortly through a concrete example. Consider $R(a_1, a_2) \bowtie R(a_2, a_3)$ and the ordering (a_2, a_1, a_3), which yields the pipeline Scan[a_2]→Join[$R(a_1, a_2)$]→Join[$R(a_2, a_3)$]. After one call to Scan and each subsequent Join, a subset of the output relation, also called a *chunk*, is stored in the output vectors shown in Fig. 2. We next explain step by step how these vector states are obtained.

The Scan produces a vector of a_2 values common to $R(a_1, a_2)$ and $R(a_2, a_3)$, namely {2, 3, 4}, with pos = −1, marking the vector as a list of values, and size = 3. The output vector of a Scan thus represents a one-attribute relation, equivalent to an f-representation over a single-node f-tree rooted at a_2. When pos ≥ 0, the vector is single-valued and bound to the a_2 value at position pos.

Fig. 2. First output vectors produced by list-based processing [15] for $R(a_1, a_2) \bowtie R(a_2, a_3)$ and ordering (a_2, a_1, a_3) evaluated on the base relation R in Fig. 1a.

The first $\text{Join}[R(a_1, a_2)]$ is many-to-many, *i.e.*, each a_2 value can match multiple tuples in $R(a_1, a_2)$. Thus, LBP binds a_2 to one value at a time by iterating $\text{pos} \in \{0, 1, 2\}$, *i.e.*, $a_2 \in \{2, 3, 4\}$, until it finds non-empty matches. This iteration advances pos from -1 to 0, thereby binding $a_2 = 2$, and produces a vector of a_1 matches, namely $\{0, 1, 3\}$, with $\text{pos} = -1$ and $\text{size} = 3$. Logically, the intermediate relation represented by these two vectors is $(a_2, a_1) \in \{\cup_{a_2}\{2\} \times \cup_{a_1}\{0, 1, 3\}\}$, which is equivalent to the flat tuples $(a_2, a_1) \in \{(2, 0), (2, 1), (2, 3)\}$. Since a_1 depends on a_2, the two output vectors are equivalent to an f-representation over an f-tree with a_2 as the root and a_1 as its child.

The second $\text{Join}[R(a_2, a_3)]$ proceeds similarly, but does not need to iterate because a_2 is already single-valued, bound to 2 with $\text{pos} = 0$. This produces a vector of a_3 matches, namely $\{5, 6, 7\}$, with $\text{pos} = -1$ and $\text{size} = 3$. Logically, the intermediate relation represented by these three vectors is $(\cup_{a_2}\{2\} \times \cup_{a_1}\{0, 1, 3\} \times \cup_{a_3}\{5, 6, 7\})$, which is equivalent to the flat tuples $(a_2, a_1, a_3) \in \{(2, 0, 5), (2, 0, 6), (2, 0, 7), (2, 1, 5), (2, 1, 6), (2, 1, 7), (2, 3, 5), (2, 3, 6), (2, 3, 7)\}$. Since a_3 depends on a_2, the three output vectors are equivalent to an f-representation over the f-tree $\mathcal{T}_2$ in Fig. 1e. These tuples form the first output chunk of the relation. Execution then backtracks to $\text{Join}[R(a_1, a_2)]$ and resumes by advancing pos from 0 to 1.

As the walkthrough and Fig. 2 illustrate, LBP materializes each node of $\mathcal{T}_2$ (*i.e.*, each output attribute) as a separate factorized vector. These vectors maintain the f-tree's parent-child groupings implicitly through their state: internal nodes become single-valued ($\text{pos} \geq 0$) because join evaluation iterates over one parent binding at a time, while child nodes are produced as lists of values ($\text{pos} = -1$) conditioned on the current single-valued parent. As a result, child vectors are populated at the granularity of join matches, typically tens to hundreds of values in network datasets, rather than being filled to the fixed vector width. Fig. 2 shows most slots in the 8-wide vectors empty, which is a major shortcoming: sparsely populated vectors incur high interpretation overhead and underutilize caches relative to fully packed flat vectors [30]. Moreover, orderings equivalent to flat tuple evaluation can degenerate into a few-tuples-at-a-time execution. For example, the ordering (a_1, a_2, a_3) yields an output over $\mathcal{T}_1$ in Fig. 1c, which offers no factorization benefit and therefore performs worse than vectorized engines because of higher interpretation cost.

Finally, LBP supports only a restricted class of f-trees. Because parent-child groupings are maintained implicitly through the vector state, LBP supports only f-trees with limited branching, *i.e.*, at any node, at most one child branch can extend beyond a height of 1. For example, for the query $R(a_1, a_2) \bowtie R(a_2, a_3) \bowtie R(a_3, a_4) \bowtie R(a_4, a_5)$, the most compact f-tree $\mathcal{T}_3$ in Fig. 3a yields f-representations with worst-case size N^2: branching at a_3 separates independent subqueries conditioned on a_3 values, so the representation size becomes the sum of the sizes of $R(a_1, a_2) \bowtie R(a_2, a_3)$ and $R(a_3, a_4) \bowtie R(a_4, a_5)$ rather than their Cartesian product. In contrast, LBP can realize only $\mathcal{T}_4$ or $\mathcal{T}_5$, shown in Fig. 3b and Fig. 3c, respectively, both of which have worst-case size N^3. Thus, because of LBP's intermediate layouts, substantial compression benefits remain unattainable.

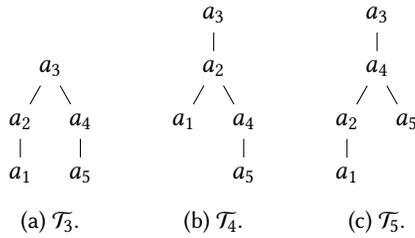(a) $\mathcal{T}_3$.   (b) $\mathcal{T}_4$.   (c) $\mathcal{T}_5$.

Fig. 3. Valid f-trees for $R(a_1, a_2) \bowtie R(a_2, a_3) \bowtie R(a_3, a_4) \bowtie R(a_4, a_5)$.

2.3 Semantic LLM Operators

A new wave of analytical-semantic DBMSs (TAG [10], PALIMPSEST [11], LOTUS [12], THALAMUSDB [19], and GALOIS [42]) integrate LLMs into query execution via semantic operators over relations, such as summarization, classification, and extraction. In these systems, LLMs are invoked within scalar or aggregate functions applied in Projection operators over intermediate relations. However, LLMs consume text, so intermediates must be serialized as text and included in prompts.

Consider the relations Paper(id, title, abstract) and Citation(src, dst), where src and dst reference Paper.id. A tuple Citation(src, dst) denotes a directed citation from src to dst (src→dst). Suppose we want to apply an LLM operator to triples of papers connected by two consecutive citations, $P_1 \rightarrow P_2 \rightarrow P_3$, for example, to extract technical terms common to all three papers. The query below joins the two citation relations with the corresponding paper records and then applies an LLM map operator to attributes in the resulting intermediate relation:

```
SELECT llm_map({'model': '...', 'prompt': '...',
    'relevant_columns': ['P1.title','P1.abstract',
    'P2.title','P2.abstract','P3.title','P3.abstract']})
FROM Citation AS C1
JOIN Citation AS C2 ON C2.src = C1.dst
JOIN Paper AS P1 ON P1.id = C1.src
JOIN Paper AS P2 ON P2.id = C1.dst
JOIN Paper AS P3 ON P3.id = C2.dst;
```

Semantic operators (*e.g.*, llm_map) declare a set of relevant columns whose values are serialized into the prompt. To amortize prompt overhead, such as a shared task description and instructions, these operators batch multiple tuples into a single LLM call using a shared prompt template. In current systems, each batch is typically serialized as a flat table (Markdown, JSON, or XML) with one row per tuple. This layout mirrors the underlying flat-tuple intermediates and therefore inherits their redundancy under intermediate-result explosion.

3 System Overview

We revisit the fundamental question of how to unify factorized and vectorized execution in a single engine. Our point of departure is list-based processing (LBP) [15], which adopts factorization in vectorized execution by aligning vectors with adjacency lists. While promising, as we showed in §2.2.3, LBP underutilizes modern CPUs due to sparse vectors and interpretation overhead. It is also limited in the factorization layouts it can realize. At the same time, recent semantic DBMSs invoke LLMs on join outputs. Since these intermediates are flat tuples and are serialized as text for prompting, they inherit the redundancy of the relational model. This motivates a broader systems question that guides FFX: *Can we preserve factorization in a fully vectorized pipeline, including as input to semantic operators?*

FFX addresses these challenges with three core mechanisms: (i) *packed factorized vectors* that encode f-tree groupings while preserving fully packed, contiguous vectors (§4); (ii) a *cascade update* operator that propagates tuple reductions across the grouping hierarchy within the vectorized execution model (§5.2); and (iii) a *structure-aware iterator and prompt serializer* that emits text derived from factorized intermediates for semantic operators without flattening (§6). When factorization provides no benefit, FFX falls back to standard flat vectorized execution with no overhead.

4 Packed Factorized Vectors

FFX introduces factorized vectors as an extension of the standard vector layout, also called *data chunks*. By piggybacking on the existing vector abstraction for intermediates, we implement factorized vectors as an overloaded vector type, following the same approach used to support multiple vector types in DuckDB [39] and Kuzu [18]. This deliberately constrains factorized vectors to fixed-size, contiguous memory blocks, preserving cache locality and enabling vectorized execution.

In LBP, parent-child groupings in the f-tree are maintained implicitly through the shared vector state. Join evaluation therefore binds each internal-node vector to one parent value at a time (pos ≥ 0), while producing each child vector as a list of matches (pos $= -1$) conditioned on that current binding. This design induces two limitations: (i) child vectors are populated at the granularity of adjacency lists, *i.e.*, join matches, yielding sparse vectors and high interpretation overhead; and (ii) the implicit encoding restricts the supported f-trees to limited branching, *i.e.*, only one branch of a node can extend beyond height 1.

The challenge is then twofold: (i) how to support many-to-many join expansion without degenerating into LBP; and (ii) how to encode an arbitrary f-tree layout within a physically flat, packed vector. Addressing (i) requires preserving fully packed vectors at all times, whereas addressing (ii) requires capturing hierarchy explicitly within contiguous memory blocks. We refer to our solution as *packed factorized vectors*, in contrast to the unpacked LBP approach, which cannot express arbitrary hierarchy or maintain packed vectors.

4.1 Vector Representation

To support fully packed vectors, a join-key vector must remain as a list of values after a many-to-many join while preserving the parent-child groupings implied by the f-tree. Our solution is a packed, offset-based representation with a new State. A parent-child grouping is encoded by an offset array (off) that maps each parent entry to a range in the child vector. Concretely, the values associated with the parent at position i occupy the slice (off[i], off[i+1]) in the child vector.

The State additionally stores the start and end positions to bound the active slice of a parent vector, as joins enumerate multiple child matches per parent value and are therefore iterated over in slices. Finally, we represent the selector as a bit array: it supports constant-size, in-place updates, which better suit our cascade operator (§5.2). The slice length is end − start, while the number of active values is the number of set bits in the selector over that slice.

The fields of the packed factorized vector State are as follows:

```
struct state {
   uint16_t start;
   uint16_t end;
   alignas(64) uint64_t  sel[MAX_SIZE / 64]; // a single bit per position
   alignas(64) uint16_t off[MAX_SIZE];     // offset array
};
```

The vector packages the values, which are stored as a contiguous byte array, together with a pointer to a State. The byte array supports multiple data types. For variable-length data types,

entries store the size, an inline prefix, and a pointer into an overflow block that carries the remaining bytes. Some scans and joins produce multiple vectors that share the same State, *e.g.*, dependent attributes of the same relationship that must be filtered or iterated over in lockstep, while other vectors carry their own state. We allocate State objects from an arena to ensure stable lifetimes and good locality; values and their associated state are co-located when possible, and shared-state vectors simply point to the same State.

We order the State fields to match operator access patterns so that operators can scan contiguous bytes in tight loops without the overhead of virtual-function dispatch. Algorithm 1 sketches the common iteration pattern for scanning active positions using count trailing zeros (CTZ).

Algorithm 1 Iterating over a packed factorized vector

Require: vec – *packed factorized vector*
1: $sel \leftarrow$ vec.state.sel
2: $start \leftarrow$ vec.state.start, $end \leftarrow$ vec.state.end
3: **while** pos $\leftarrow$ next_valid($sel, start, end$) **do**
4:      ... /* *op-specific processing of vec.vals[pos]* */

4.1.1 Evaluation Example. We use the running example from §2.2.1: $Q_{2H} = R(a_1, a_2) \bowtie R(a_2, a_3)$, evaluated with the pipeline Scan[a_2]→Join[$R(a_1, a_2)$]→Join[$R(a_2, a_3)$] under query attribute ordering (a_2, a_1, a_3). After one call to Scan and the two subsequent Joins, the first output chunk is materialized in the output vectors of Fig. 4, omitting sel for readability.

The Scan produces the output vector a_2 with values $\{2, 3, 4\}$, so it initializes start = 0 and end = 3. This vector represents the one-attribute relation over a_2, which is equivalent to an f-representation over a single-node f-tree rooted at a_2. We note that the offset array (off) is unused for the root of an f-tree and is therefore set to NULL, because the root has no grouping with a parent node to encode.

The first Join[$R(a_1, a_2)$] produces a_1 matches for each a_2 value and encodes the grouping using the offset array. The resulting intermediate relation represented by the two vectors is the f-representation $(\cup_{a_2}\{2\} \times \cup_{a_1}\{0, 1, 3\}) \cup (\cup_{a_2}\{3\} \times \cup_{a_1}\{1\}) \cup (\cup_{a_2}\{4\} \times \cup_{a_1}\{1, 3\})$, where the offset array (off) identifies, for each a_2 entry, the contiguous slice of a_1 values that belongs to it.

The second Join[$R(a_2, a_3)$] similarly produces the a_3 matches for each a_2 value and encodes the grouping using the offset array. The resulting output in the three vectors is the f-representation $(\cup_{a_2}\{2\} \times \cup_{a_1}\{0, 1, 3\} \times \cup_{a_3}\{5, 6, 7\}) \cup (\cup_{a_2}\{3\} \times \cup_{a_1}\{1\} \times \cup_{a_3}\{2, 4\}) \cup (\cup_{a_2}\{4\} \times \cup_{a_1}\{1, 3\} \times \cup_{a_3}\{8\})$. This representation is over the f-tree $\mathcal{T}_2$ in Fig. 1e, where both a_1 and a_3 are grouped under a_2.

4.1.2 Packed Vectors. Fig. 4 encodes the output as packed factorized vectors: each attribute a_2, a_1, and a_3 is stored in a fixed-size, contiguous vector, and the offset array (off) is used to recover the grouping required by $\mathcal{T}_2$. Unlike LBP, FFX does not bind a parent to a single value and materialize its matches as a separate child list; instead, it keeps both parent and child values packed and uses the offset array to delimit each parent's child slice. Operators then scan these contiguous ranges, with invalidations tracked by sel, while preserving the f-tree-defined associations.

For the evaluation example (§4.1.1), LBP would require backtracking to process the three a_2 bindings and therefore would require 3× as many operator calls. On larger datasets, this repeated backtracking translates into orders of magnitude higher interpretation overhead.

4.1.3 Arbitrary Factorizations. Packed factorized vectors also remove the branching restriction of LBP. Because parent-child groupings are stored explicitly in the offset array (off), rather than implicitly through pos, FFX can represent intermediates over any valid f-tree and propagate
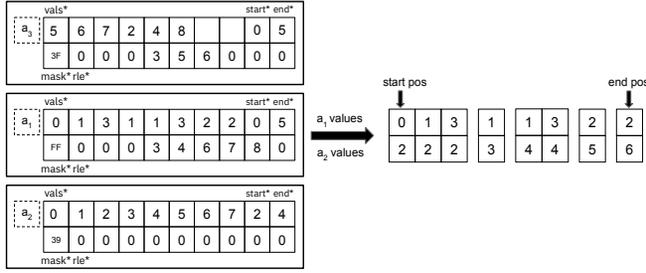
Fig. 4. First output vectors produced by FFX for $R(a_1, a_2) \bowtie R(a_2, a_3)$ and ordering (a_2, a_1, a_3), evaluated on the base relation R in Fig. 1a.

(a) Packed factorized vectors.   (b) F-representation.
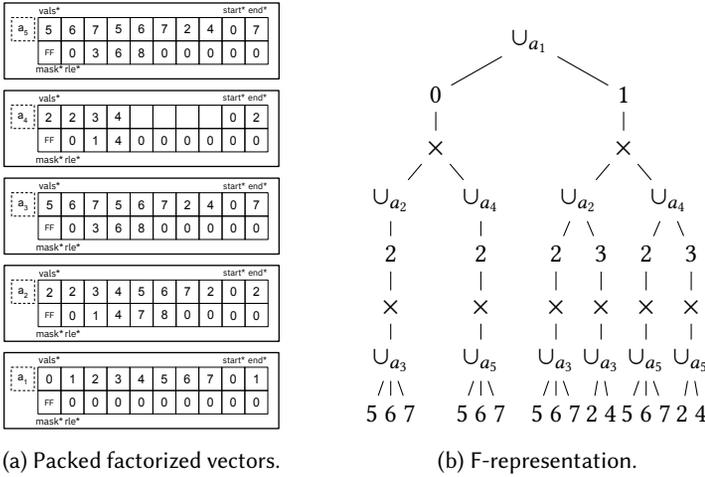
Fig. 5. First output vectors produced by FFX for $R(a_1, a_2) \bowtie R(a_2, a_3) \bowtie R(a_1, a_4) \bowtie R(a_4, a_5)$ and ordering $(a_1, a_2, a_3, a_4, a_5)$, evaluated on the base relation R in Fig. 1a.

them across operators in the same packed layout. For example, Fig. 5 shows the first output for $R(a_1, a_2) \bowtie R(a_2, a_3) \bowtie R(a_1, a_4) \bowtie R(a_4, a_5)$ under the ordering $(a_1, a_2, a_3, a_4, a_5)$, leading to a compact f-tree. The left subfigure shows the first output chunk as packed factorized vectors, while the right shows the equivalent logical f-representation. This is an illustration of branching layouts that FFX supports without sacrificing vector packing or requiring full materialization.

5 Query Execution

FFX evaluates queries using vectorized operators over packed factorized vectors (§4.1). Operators read and write fixed-size, contiguous vectors, yet the logical representations they consume and produce are f-representations, each defined over a particular f-tree. These intermediate f-trees are determined by the chosen query attribute ordering $(a_i, \ldots, a_m)$ and evolve as evaluation extends the current binding prefix. The first attribute a_i is scanned and yields a one-attribute f-representation, i.e., an f-tree rooted at a_i. Each subsequent join extends the current intermediate by introducing the next attribute and selecting a *grouping attribute* in the input f-tree under which the new values will be nested. We choose this grouping attribute as high as possible while respecting the root-to-leaf

path constraint induced by the query and the chosen ordering, thereby yielding a valid f-tree for each ordering prefix and maximizing compactness under that ordering.

When a binary join produces a new attribute a_k, the input join attribute is always an ancestor of a_k in the output f-tree and is the parent of a_k exactly when a_k can be attached directly under it. Multi-way joins follow the same principle: all input join keys must lie on a single root-to-leaf path in the current f-tree because of the dependency. All input join keys are ancestors of a_k, and the highest node to which a_k can be attached is the lowest input join key.

We adapt join operators in FFX to our vector representation and introduce a `Cascade Update` operator that propagates tuple reductions across the grouping hierarchy.

5.1 Join Evaluation

Each join consumes an input join-key vector (probe side) and produces one or more output-attribute vectors. Joins fall into two classes: (i) *non-expanding* joins produce at most one match per active key value; and (ii) *fanout-expanding* joins may produce multiple matches per key. FFX supports both *binary* and *multi-way* joins; a multi-way join is non-expanding if it matches at most one tuple across the joined relations. These classes differ in how they handle and update the vector `State`.

5.1.1 Non-expanding Join. For 1-to-1 and many-to-1 joins, each active key yields at most one matching tuple, so the operator emits output values in lockstep with the input join-key vector. Implementation-wise, it does not write to the offset array (`off`); the join-key vector and the produced attribute vectors share the same `State`. The operator updates only the selector (`sel`), invalidating keys that produce no matches.

5.1.2 Fanout-expanding Join. For 1-to-many and many-to-many joins, each active key may yield multiple matching tuples. As described earlier, the join operator has a specific attribute chosen as a grouping attribute, possibly one of the input join keys or one of their descendant attributes in the input f-tree. Given the chosen grouping attribute, the operator appends produced matches contiguously and modifies the offset array so that each grouping entry at index i maps to its child slice (`off[i]`, `off[i+1]`) in the child vector. If the grouping attribute is the join key, this mapping is direct. If the grouping attribute is instead a descendant in the input f-tree, the join applies the produced matches to each value in the descendant slice; that is, it applies a Cartesian product.

Fanout expansion sets not only the offset array of the output vector but can also update the active slices of vectors: packed output may shrink the effective input slice (`start`, `end`), and keys with no matches are invalidated in the selector (`sel`). These reductions must propagate through the grouping hierarchy: if a slice becomes fully inactive, its ancestors may become inactive as well. FFX propagates active-slice (`start`, `end`) changes during join evaluation and uses `Cascade Update` (§5.2) to propagate `selector` updates, which may also be triggered by filters.

5.1.3 Multi-way Joins. FFX evaluates a multi-way join as a join-at-a-time pipeline that alternates between (i) *extensions*, which produce candidate bindings for the next attribute via a binary join; and (ii) *intersections*, which enforce the remaining relations by intersecting the corresponding key vectors. This follows a worst-case-optimal join-at-a-time strategy [9], adapted to packed factorized vectors. As a practical optimization, we fix the extension and intersection order at compile time, rather than dynamically at run time in the order of the smallest lists, which enables reuse of intersections on shared ancestor attributes [32, 34] and cleanly supports filter operators interleaved between extension and intersection steps.

Our execution model gracefully falls back to standard vectorized execution on non-expanding joins. The input and output vectors simply share the same `State`, and no offset array updates are required. In these cases, factorization offers no additional benefit.

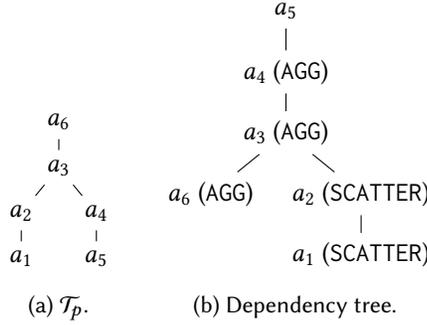(a) $\mathcal{T}_p$.   (b) Dependency tree.

Fig. 6. Dependency Tree for $R(a_1, a_2) \bowtie R(a_2, a_3) \bowtie R(a_3, a_4) \bowtie R(a_4, a_5) \bowtie R(a_6, a_3)$, derived from the f-tree $\mathcal{T}_p$ and with a_5 as the source of the reduction to be propagated.

5.2 Cascade Update

Reducing operators (filters and joins) can invalidate values within a packed factorized vector. Because vectors encode an f-representation, these invalidations are not local: if all values in a child slice become inactive, the corresponding parent binding must be invalidated. Once a parent binding becomes inactive, the invalidation can eliminate entire sibling slices and may propagate further upward and to siblings. Cascading these invalidations is required for correctness.

5.2.1 Operator Overview. FFX propagates reductions in place in two ways: (i) *slice update*: shrink active slices, *i.e.*, start and end positions, common in fanout-expanding joins. As join evaluation descends the f-tree and enumerates larger numbers of child values, it binds the ancestor vectors to progressively smaller slices because each join contracts the active slice during iteration; and (ii) *selector update*: propagate selector invalidations across dependent groupings when a join or filter invalidates entries in the input vector.

We implement a dedicated Cascade Update (CU) to propagate state changes induced by reducing operators. It can be configured to update only the active slice (start, end), only the selector, or both. It takes as input the modified State and applies the implied slice contraction or selector invalidations to all dependent vectors. To do so, CU is initialized from a dependency tree derived from the f-tree of the current intermediate. This tree specifies which ancestor and sibling States must be updated, and in what order, when a given vector changes. For correctness, the execution plan must ensure that every state modification produced by a reducing operator is propagated to the other dependent States before downstream operators need to consume them.

5.2.2 Dependency Tree. Cascade Update propagates reductions by traversing a *dependency tree* derived from the f-tree of the current intermediate. The tree is rooted at the f-tree node whose State changed, *e.g.*, the join key whose selector was updated. We construct the dependency tree by walking from this node up to the f-tree root and, at each step, adding exactly those other attributes whose validity may change as a consequence. We label dependency-tree nodes by their type of propagation: AGG nodes rely on a child slice to update an ancestor binding, while SCATTER nodes push newly invalidated bindings into dependent subtrees. For each visited f-tree node v:

(1) *Add the parent (AGG)*. If v has a parent in $\mathcal{T}$, we add parent(v) as a child of v in the dependency tree and label that node AGG.
(2) *Add side descendants (SCATTER)*. We add nodes for all descendants of v that are not on the path from v to the f-tree root as children of v and label each such node SCATTER.
(3) Set $v \leftarrow$ parent(v) and repeat until reaching the root.

Algorithm 2 Selector propagation from a dependency-tree parent node (p_node) to a child node (c_node) based on the parent's newly invalidated positions (p_inv_pos)

1: **function** SELECTOR_PROPAGATE(p_node, c_node, p_inv_pos)
2:     $p\_state \leftarrow p\_node.state$; $c\_state \leftarrow c\_node.state$;
3:     $new\_inv\_pos \leftarrow \emptyset$ ▷ Set to accumulate new invalidations due to propagation.
4:     **if** c_node.label = AGG **then**
5:          ▷ For AGG, an f-tree child node propagates invalidations to its parent node. Note that the node relationship in the dependency tree is reversed.
6:         **while** $i \leftarrow$ next_valid(c_state.sel, c_state.start, c_state.end) **do**
7:             **if** no_set_bits(p_state.sel, p_state.off[i], p_state.off[i+1]) **then**
8:                  CLEAR_BIT(c_state.sel, i)
9:                  $new\_inv\_pos \leftarrow new\_inv\_pos \cup \{i\}$
10:     **else** ▷ SCATTER: For each child slice, add active pos as new invalidations → clear the slice.
11:         **for each** i in p_inv_pos **do**
12:             **while** $j \leftarrow$ next_valid(c_state.sel, c_state.off[i], c_state.off[i+1]) **do**
13:                  $new\_inv\_pos \leftarrow new\_inv\_pos \cup \{j\}$
14:              CLEAR_BITS(c_state.sel, c_state.off[i], c_state.off[i+1])
15:     TRIM_SLICE(c_state)   ▷ Trim leading and trailing inactive positions → update start/end.
16:     **if** $new\_inv\_pos \neq \emptyset$ **then**
17:         **for each** $child$ in c_node.children **do**
18:             SELECTOR_PROPAGATE(c_node, $child$, new_inv_pos)

An AGG node denotes an ancestor update: Cascade Update checks the parent-child grouping and invalidates a parent binding whenever all values in the corresponding child slice become inactive. A SCATTER node denotes a descendant update: Cascade Update pushes newly introduced parent invalidations into the dependent subtree, pruning all slices grouped by the invalidated bindings.
Example. Fig. 6a shows an f-tree rooted at a_6 with child a_3 and two branches under a_3: one rooted at a_4 with child a_5, and one rooted at a_2 with child a_1. Suppose a filter or join updates the selector of a_5; Fig. 6 shows the corresponding dependency tree rooted at a_5.

Cascade Update first propagates AGG: logically, it checks whether each a_4 binding has any active values in its a_5 slice and invalidates the binding if the slice is empty, then repeats the same check from a_4 to a_3 and from a_3 to a_6. If some a_3 bindings are invalidated, it then propagates SCATTER from a_3 into the other branch, pruning the slices under a_2, and consequently under a_1, that are grouped by those newly invalidated a_3 bindings.

5.2.3 Operator Evaluation. Cascade Update propagates updates along the dependency tree using a preorder traversal. In effect, it first applies all AGG steps toward the root and then applies SCATTER to the affected subtrees. At each visited node, it updates the corresponding vector State with tight loops, avoiding recursion and tuple-at-a-time control flow. The traversal supports early stopping: if a node produces no new invalidations, propagation terminates for that subtree because no descendant can be affected. Alg. 2 shows the selector propagation.

5.2.4 Optimizations. Cascade Update (CU) represents additional overhead introduced by factorized execution. To minimize this overhead, we apply three complementary optimizations.
Placement. While correctness allows inserting a CU after every reducing operator, FFX's planner inserts CU operators lazily to amortize repeated propagation. Specifically, we delay selector propagation until it is required: (i) before a sink operator; or (ii) before a vector is consumed

as input when its `State` may be stale. Moreover, we avoid inserting `selector` cascades along frontier-only extensions, where reductions cannot yet affect sibling subtrees.

Operator fusion. CU can be inserted as a standalone operator or fused into the preceding reducing operator to reduce call overhead.

Delta-driven `AGG`. A naïve `AGG` scans all active parent bindings and checks whether each child slice is empty, even if only a few child positions have changed. We instead drive `AGG` from the delta: newly invalidated child positions identify their invalidated parent bindings via the offset array (`off`), forming a smaller set of parent bindings that may need to be invalidated. This is an optimization for Alg. 2, lines 6–7. `AGG` then tests slice emptiness only for parents in this workset, making the cost scale with the number of new invalidations.

Together, these optimizations make the cost of the `Cascade Update (CU)` operator largely proportional to the number of newly introduced invalidations while still linear in the size of the input f-representation: FFX's planner amortizes their insertion, fusion reduces per-invocation overhead, and the refinements to `AGG` avoid scanning unchanged parent slices unless a child slice actually becomes empty. As a result, CU remains lightweight on low-selectivity joins yet still prunes correctly when reductions are selective and cascade.

6 Factorized Semantic Processing

FFX extends semantic operators to execute directly over factorized intermediates. The main challenge is that LLM-powered operators consume text representations of flat tuples, whereas FFX represents intermediates as packed factorized vectors over an f-tree. A straightforward implementation would flatten the intermediate into tuples and then serialize those tuples into prompts, but doing so would forfeit the compactness benefits of factorization.

Instead, FFX preserves factorization throughout prompt construction. It traverses the relevant portion of the input using windowed iteration, serializes the corresponding f-representation in a structure-aware format, and instructs the LLM to perform the implied Cartesian expansion while producing one prediction per logical tuple. As we show in this section, this design reduces redundancy in serialized prompts with little to no loss in quality.

6.1 Semantic Operator Model

We focus on the `llm_map` primitive, which we implement as a dedicated operator. Other semantic primitives, such as `llm_reduce`, `llm_filter`, and `llm_rerank`, are left to future work.

The `llm_map` operator takes as input a factorized intermediate together with a set of relevant attributes and logically produces one prediction per logical input tuple. When the relevant attributes lie on a single root-to-leaf path of the current f-tree, the operator serializes that path directly in hierarchical form and produces one prediction for each resulting logical tuple. When they span multiple branches, FFX avoids flattening the input: it serializes the factorized structure and instructs the model to apply the implied Cartesian product and return one prediction per resulting tuple. This preserves the compactness benefits of factorization on the input side, although the logical output cardinality remains unchanged. Empirically, and somewhat surprisingly, our evaluation in §7.5 shows that even non-reasoning models can often perform this Cartesian expansion accurately while still carrying out the semantic task. Errors become noticeable mainly at large batch sizes, where flat representations already exceed the context-window limit.

Because each prediction depends on all relevant attributes, the predicted attribute value and all relevant attributes must lie on the same root-to-leaf path in the output f-representation. FFX reconstructs the output without full materialization: it inserts each tuple enumerated by the model,

together with its prediction, at the appropriate position in the output factorized vectors. The resulting output is then passed to the next operator in the pipeline.

6.2 Windowing Over Factorized Intermediates

Prompt construction begins with a deterministic iterator over the relevant portion of the current f-tree. For each root-to-leaf path containing relevant attributes, the iterator maintains a *window* at the corresponding relevant leaf, *i.e.*, a relevant attribute with no relevant descendant on that path. Each window tracks a bounded set of bindings for that leaf together with the shared ancestor prefix above it. One iterator step returns a factorized fragment formed by the current windows across all relevant paths. This fragment corresponds to a bounded set of logical tuples without flattening the full intermediate.

Window sizes control the trade-off between compactness and flatness. If all window sizes are 1, the iterator degenerates to ordinary flat-tuple iteration. Larger windows preserve common prefixes and expose multiple logical tuples together, so repeated context is emitted once rather than duplicated across rows. During iteration, windows advance in a fixed order and only one window moves at a time while earlier windows remain fixed. As a result, consecutive prompts share long prefixes, which improves prefix reuse, controls context size, and stabilizes output quality.

6.3 Structure-Aware Prompt Serialization

Each factorized fragment emitted by the iterator is serialized from the underlying packed factorized vectors into a format that follows the f-tree rather than a flat row-per-tuple layout. FFX supports JSON and XML, but the particular surface syntax is orthogonal to the core idea. In all cases, shared ancestor bindings are emitted once, and descendant bindings are nested under their corresponding parent, so the prompt directly mirrors the factorized structure of the input. The serializer projects only the relevant attributes and embeds the result inside a meta-prompt that specifies the task, the expected output schema, the f-tree structure, and the required Cartesian expansion.

The model is instructed to return output tuples in a fixed enumeration order. FFX uses the same order for its enumeration and aligns each returned prediction with its corresponding tuple. When predictions are missing for some tuples due to an LLM failure with the Cartesian expansion, FFX assigns `NULL`; alternatively, one could retry the LLM invocation. Thus, `llm_map` consumes a factorized input, avoids flattening for prompt construction, and flattens after LLM invocation because the relevant input attributes and the prediction are dependent and must therefore lie on the same root-to-leaf path in the output f-representation. The remaining attributes remain factorized.

7 Evaluation

We evaluate the key design decisions underlying FFX. Our empirical study addresses three questions: (i) How does our vector representation compare with prior state-of-the-art approaches such as LBP [15]? (ii) What overhead, if any, does our vector representation introduce when factorization provides no benefit? (iii) How effective is our factorized semantic processing at reducing input token usage, and what is the impact on accuracy?

We also compare FFX against DuckDB [39], a relational vectorized baseline, and Kuzu [18], an LBP-based system, and we evaluate multi-core scalability on large datasets.

7.1 Setup

7.1.1 Hardware. Unless otherwise stated, we run all experiments on a Google Cloud `c4-standard-8` instance with 8 vCPUs, 30 GB of main memory, and an `INTEL(R) XEON(R) PLATINUM 8581C CPU@2.30GHz`. The machine runs Debian v6.1.148.

Domain	Name	Nodes	Edges
Social	Epinions	76K	509K
	LiveJournal	4.8M	69M
	Twitter	41.6M	1.46B
Web	Google	876K	5.1M
Product	Amazon	403K	3.5M
Citation	ArXiv	169K	1.17B

Table 2. Datasets used.

	Query
Path	$Q1 = R(a,b) \bowtie R(b,c) \bowtie R(c,d) \bowtie R(d,e)$
	$Q2 = R(a,b) \bowtie R(b,c) \bowtie R(c,d) \bowtie R(d,e) \bowtie R(e,f)$
Star	$Q3 = R(a,b) \bowtie R(a,c) \bowtie R(a,d) \bowtie R(a,e)$
	$Q4 = R(a,b) \bowtie R(a,c) \bowtie R(a,d) \bowtie R(a,e) \bowtie R(a,f)$
	$Q5 = R(a,b) \bowtie R(a,c) \bowtie R(a,d) \bowtie R(a,e) \bowtie R(a,f) \bowtie R(a,g)$
Tree	$Q6 = R(a,b) \bowtie R(a,c) \bowtie R(b,d) \bowtie R(c,e)$
	$Q7 = R(a,b) \bowtie R(a,c) \bowtie R(a,d) \bowtie R(b,e) \bowtie R(c,f)$
	$Q8 = R(a,b) \bowtie R(b,c) \bowtie R(b,d) \bowtie R(c,e) \bowtie R(d,f)$
	$Q9 = R(a,b) \bowtie R(a,c) \bowtie R(b,d) \bowtie R(e,d)$

Table 3. Queries used, grouped by join query graph structure.

7.1.2 Compiler. All binaries are compiled with `clang v15.0.6` using the `-O3` flag. Build files are generated using `CMake v3.25.1`.

7.1.3 Datasets. The datasets used in our evaluation are listed in Table 2. They are drawn from the Open Graph Benchmark (OGB) [16], a 2010 Twitter crawl [24], and the SNAP collection [27]. Each dataset is represented as a base relation $R(\text{src}, \text{dst})$. In Table 2, *Edges* denotes the number of tuples in R, and *Nodes* denotes the number of distinct values across src and dst. These datasets span diverse domains, including citation, social, web, and product networks, and exhibit varying structural characteristics such as scale, neighbourhood degree distributions, and skew.

7.1.4 Queries. The queries used are listed in Table 3 in Datalog-style conjunctive form. Each query consists exclusively of equi-joins and self-joins over the same base relation R, where an equi-join is expressed by reusing the same query variable across multiple atoms, *e.g.*, $R(a, b) \bowtie R(b, c)$.

We use queries with varying join-graph shapes, including path, star, and tree queries, to cover a range of factorized representations and execution challenges, from deep chain groupings in paths to high-branching layouts in stars and trees. Our main focus is on acyclic join graphs, where factorization yields asymptotic size reductions and where query performance is most sensitive to intermediate-result explosion. To complement these workloads, we also evaluate FFX on the LSQB [33] and JOB [26] benchmarks in our end-to-end evaluation, which stress the system along additional dimensions, including cyclic queries and string predicate evaluation. These choices are consistent with those from prior work [1, 6, 20, 34].
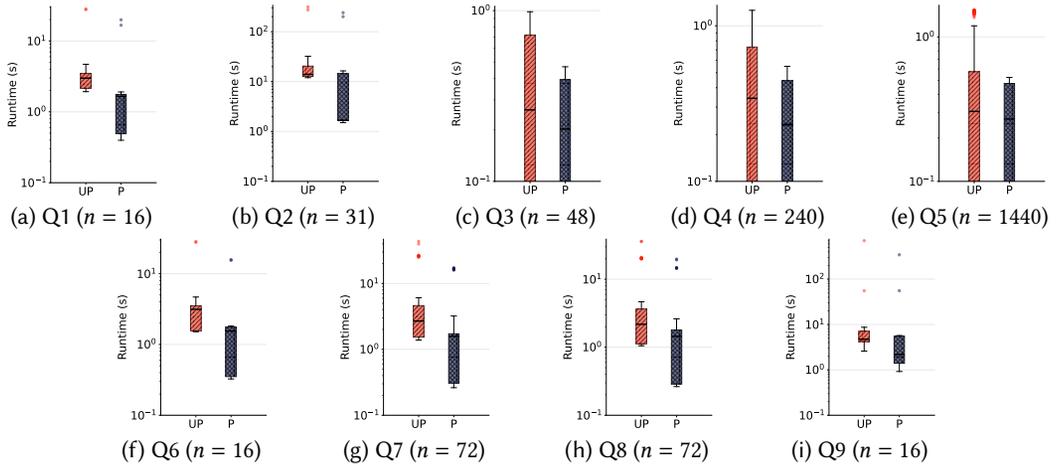
Fig. 7. Performance comparison on `Google` for Q1–Q9 using (i) LBP's unpacked vectors (UP) and (ii) packed factorized vectors (P), both implemented within FFX. The figure reports run time in seconds (log scale) across all plans for each query, where n denotes the number of plans.

7.1.5 Query Execution. We run each query three times and report the median unless otherwise stated. After each query, the cache is manually cleared to minimize bias from execution order using the following command: `sudo sync && echo 3 | sudo tee /proc/sys/vm/drop_caches`

7.2 Factorization and Vectorization Benefits

We compare our packed factorized vector representation with that of list-based processing (LBP) [15]. To ensure a fair comparison, both representations are implemented in FFX. We denote ours by P and LBP's by UP, standing for packed and unpacked, respectively. We evaluate the nine queries in Table 3 using P and UP across three datasets: `Amazon`, `Google`, and `Epinions`. Each query is evaluated over all possible plans, *i.e.*, all query attribute orderings. Some plans are equivalent to processing flat tuples since their intermediates are encoded over a root-to-leaf f-tree; for example, plan (a, b, c, d, e) for Q1 produces intermediates over $a - b - c - d - e$ rooted at a. Other plans yield more compact representations with intermediates over f-trees with branches and smaller heights; for example, plan (c, b, a, d, e) for Q1 produces intermediates over $c - b - a$ and $c - d - e$, rooted at c.

Fig. 7 shows the results on `Google`; we observe similar trends on `Epinions` and `Amazon`. On `Google`, P outperforms UP on 1883 plans (96.53%) and underperforms on only 68 plans (3.47%). The largest observed speedup occurs on Q2 for plan (d, c, e, f, b, a), where run time drops from 16.5 to 1.76 seconds (9.39×). Across all queries, the three highest average speedups are for Q2, Q7, and Q8, at 5.72×, 4.11×, and 3.32×, respectively. Overall, the mean speedup on `Google` across all plans and queries is 2.08×, and the 68 slowdowns all correspond to Q5 plans with run times below 100 ms.

We attribute these improvements to two factors: (i) vectorization, which lowers interpretation cost, with, on average, 78.77× fewer virtual function calls, and improves microarchitectural efficiency, as reflected in reductions in cycles, instruction count, branch-misprediction rate, and cache-miss rate; and (ii) factorization, which leads to more compact representations, as measured by the number of processed tuples, and hence less redundant work. These results highlight that factorization and vectorization offer complementary benefits. We next isolate and assess the contribution of each.

7.2.1 Vectorization Benefits Analysis. To isolate the contribution of vectorization and quantify the overhead of our packed factorized vectors, we need queries and plans for which factorization

Plan		RT (s)	# Calls	# I	# C	IPC	# BM	# CM	# Proc. T
$P_{Q1,1}$	UP	27663	913.0M	131.41B	86.74B	1.52	869.4M	1.46B	15.89B
	P	16671	8.00M	89.14B	55.23B	1.61	591.8M	1.29B	945.54M
		(1.66×)	(114.13×)	(1.47×)	(1.57×)	(1.06×)	(1.47×)	(1.13×)	(16.80×)
$P_{Q2,1}$	UP	294089	15.38B	2.20T	905.72B	2.18	7.24B	12.37B	282.28B
	P	200793	141.84M	1.46T	668.47B	2.43	6.17B	11.75B	15.89B
		(1.46×)	(108.43×)	(1.52×)	(1.35×)	(1.11×)	(1.17×)	(1.05×)	(17.76×)
$P_{Q1,2}$	UP	2348	56.91M	9.45B	8.11B	1.15	29.62M	133.17M	11.48B
	P	441	82.76K	2.60B	2.26B	1.17	15.86M	73.90M	141.79M
		(5.32×)	(687.69×)	(3.63×)	(3.59×)	(1.02×)	(1.87×)	(1.80×)	(80.96×)
$P_{Q2,2}$	UP	16502	812.1M	113.1B	54.20B	1.32	415.6M	608.55M	138.11B
	P	1757	0.59M	8.81B	6.66B	1.32	84.57M	275.59M	1.12B
		(9.39×)	(1378.84×)	(12.84×)	(8.14×)	(1.58×)	(4.91×)	(2.21×)	(122.60×)

Table 4. Comparison of FFX packed (P) and LBP unpacked (UP) representations on Google for queries Q1 and Q2 in Table 3. We use two plans for each: $P_{Q1,1} = (a, b, c, d, e)$, $P_{Q1,2} = (c, d, b, a, e)$, $P_{Q2,1} = (a, b, c, d, e, f)$, and $P_{Q2,2} = (d, c, e, f, b, a)$. Metrics include run time (RT) in seconds, and the numbers of operator function calls (# Calls), instructions (# I), CPU cycles (# C), instructions per cycle (IPC), branch misses (# BM), cache misses (# CM), and number of processed tuples (# Proc. T). Improvements are shown in parentheses.

provides no benefit. To this end, we focus on plans for path queries Q1 and Q2, whose intermediates are over root-to-leaf f-trees and are thus equivalent to processing flat tuples. For these queries, plans $P_{Q1,1} = (a, b, c, d, e)$ and $P_{Q2,1} = (a, b, c, d, e, f)$ yield no asymptotic factorization benefits, with intermediates over flat representations: $a - b - c - d - e$ and $a - b - c - d - e - f$, both rooted at a.

Table 4 reports results on the Google dataset, where packed execution consistently outperforms the unpacked baseline. Using packed factorized vectors, FFX achieves 1.66× and 1.46× speedups on $P_{Q1,1}$ for Q1 and on $P_{Q2,1}$ for Q2, while reducing operator function calls by up to 108.4×. Microarchitecturally, packed execution reduces total CPU cycles by 1.35–3.59× and improves IPC by 1.02–1.58×. These gains arise from both reduced work, with 1.47–12.84× fewer instructions and 1.17–4.91× fewer branch mispredictions, and higher per-cycle throughput, which compound to drive end-to-end speedups. In contrast, LBP's representation cannot realize these benefits, since it continues to pass unpacked vectors even when factorization offers no benefit.

Overall, our analysis shows that even when packed factorized vectors offer no compactness benefit, FFX retains substantial speedups over LBP. The packed layout enables cache-efficient processing and allows the compiler to exploit autovectorization more effectively when possible.

7.2.2 Factorization Benefits Analysis. Our goal here is to isolate the contribution of factorization, *i.e.*, of compactness, on top of vectorization. We therefore focus on plans whose intermediates are over branched f-trees rather than root-to-leaf paths. For Q1 and Q2, examples of such plans are $P_{Q1,2} = (c, d, b, a, e)$ and $P_{Q2,2} = (d, c, e, f, b, a)$. Their factorized representations contain branching and are over the f-trees $c - d - e$, $c - b - a$, rooted at c, and $d - c - b - a$, $d - e - f$, rooted at d. These branched representations eliminate heavily repeated enumerations. They are the most compact representations for Q1 and Q2 under f-representations [36].

Table 4 reports the results on the Google dataset. Packed factorized execution (P) delivers large gains over the unpacked baseline (UP) on these plans: 5.32× for Q1 and 9.39× for Q2. These improvements are due to the substantial elimination of redundant computation, as reflected in the

3.63× and 12.84× reductions in executed instructions. This elimination yields much larger benefits than those obtained from vectorization alone.

Branching reduces f-tree height, leading to greater compactness. By propagating packed factorized vectors, FFX can exploit these benefits. In contrast, Kuzu's unpacked, list-based representation cannot achieve the same reduction in computation and effectively linearizes these branches, yielding the f-trees $c - d$, $c - b - a - e$ for $P_{Q1,2}$ and $d - c$, $d - e - f - b - a$ for $P_{Q2,2}$. This increases the effective f-tree height and leads to redundant computation.

Across the nine queries, 22% (2/9: Q1, Q9) have the same best ordering under packed and unpacked execution, yet packed execution is still faster by 4.94× on average (Q1: (c, b, d, a, e), 397 vs. 2106 ms, 5.30×; Q9: (b, d, a, c, e), 921 vs. 4216 ms, 4.58×). For the remaining 78% (7/9: Q2–Q8), the optimal orderings differ slightly, and packed execution achieves a 5.47× average speedup when comparing the best plan under each representation. The largest gap is on Q2, where the packed best ordering (c, b, d, e, a, f) is 7.98× faster than the unpacked best ordering (c, b, d, e, f, a) (1.50 vs. 11.99 seconds); the smallest is on Q4, where packed execution under (a, e, b, c, f, d) improves by 1.22× over unpacked execution under (a, b, f, c, d, e) (49 vs. 60 ms).

Optimal plans diverge because the same ordering induces different factorization schemes under packed and unpacked execution. Unpacked execution is restricted to a limited class of f-trees (Figure 3), which can force less compact layouts and lead to additional redundant work. By supporting arbitrary f-trees, FFX can realize the compact layout implied by the chosen ordering, reducing computation and shifting the plan trade-offs. This effect is most pronounced for deep path queries (5.30×–7.98×), whereas shallow star queries see smaller gains (1.22×–1.26×) because flatter f-trees incur lower per-tuple overhead.

In summary, when branching enables compact factorized intermediates, FFX preserves vectorization and exploits compactness, reducing redundant computation. In contrast to prior approaches, vectors remain fully packed, interpretation overhead drops, microarchitectural efficiency increases, and FFX obtains the benefits of both compact representations and CPU-efficient execution.

7.3 End-to-End Join Benchmark Evaluation

We compare FFX against two state-of-the-art systems: DuckDB [39] and Kuzu [18]. DuckDB is a high-performance analytical RDBMS with a vectorized execution engine, while Kuzu is a graph DBMS with factorized execution based on a list-based representation.

While both DuckDB and Kuzu are mature, feature-rich production systems, FFX is a research prototype optimized for the core problem studied here. Hence, the absolute numbers are not directly comparable, but they are indicative of the potential efficiency gains achievable by tightly integrating factorization and vectorization.

7.3.1 System Setup. We ensure that table statistics are computed so that the DBMS optimizers can select join orders based on cost. We run all systems single-threaded in in-memory mode, with disk spilling disabled and sufficient memory provisioned. We perform one warm-up run, which is discarded from measurement. We then report the median of three timed runs and measure execution time only, excluding compilation time.

7.3.2 Workload. We evaluate the three systems on the LSQB [33] and JOB [26] benchmarks, as well as on the nine queries in Table 3 over the Amazon dataset.

7.3.3 LSQB Results. Table 5 reports single-threaded run times at scale factor 10 for Q1–Q6. We omit Q7–Q9 because FFX does not support outer and anti-joins. Overall, FFX is consistently competitive and often substantially faster than both Kuzu and DuckDB, with the largest gains on join patterns that admit compact factorization.

Query	FFX	Kuzu	DuckDB
Q1	1.28	591.10 (**545.5×**)	18.43 (**17.0×**)
Q2	2.71	15.16 (**5.6×**)	2.72 (**1.0×**)
Q3	3.32	105.61 (**31.8×**)	6.14 (**1.9×**)
Q4	1.26	7.48 (**5.9×**)	9.86 (**7.8×**)
Q5	5.56	10.08 (**1.8×**)	6.94 (**1.2×**)
Q6	5.84	49.42 (**8.6×**)	153.39 (**26.3×**)

Table 5. Run time comparison in seconds for queries Q1–Q6 (single-threaded execution) on LSQB. Improvement factors relative to Kuzu and DuckDB are shown in parentheses.

(a) J.a run time   (b) J.a speedup   (c) J.b run time   (d) J.b speedup   (e) J.c run time   (f) J.c speedup
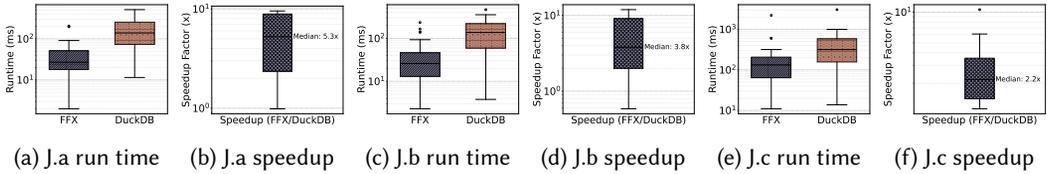
Fig. 8. Run time (ms) and speedup of FFX over DuckDB on the JOB [26] (J) variants a, b, and c.

For FFX, we pick orderings to maximize factorization, favouring short and branching f-trees; otherwise, we pick plans arbitrarily among equally compact f-trees. This strategy is most effective for path queries such as Q1 and Q6, where a compact hierarchy substantially reduces redundant work. In contrast, cyclic join queries lead to taller f-trees and reduce compactness. In these cases (*e.g.*, Q2), we evaluate FFX under DuckDB's chosen ordering.

7.3.4 JOB Results. Fig. 8 compares FFX and DuckDB on the JOB benchmark variants a, b, and c. Across all variants and queries, FFX is fastest when the ordering yields succinct f-trees, often star-like patterns, allowing it to keep intermediates factorized and avoid enumerating a large number of flat tuples.

We highlight variant c because it has less selective predicates, which increase intermediate sizes. As a result, speedups are smaller because of increased computation during join processing: lower selectivity causes more index nested-loop probes (FFX's core join operator) and more irregular join-index lookups, which in turn raise cache-miss rates. For example, from Q18b to Q18c, the cache-miss rate increases from 2.19% to 3.89%.

7.3.5 Nine Queries Results. Following the JOB benchmark methodology, for the nine queries in Table 3, we also project the MIN value for all attributes to isolate join execution performance and avoid heavy tuple copying or enumeration in the sink operator.

Fig. 9 reports run time in seconds across all queries in Table 3. FFX achieves substantial speedups, ranging from $10^2×$ to $10^5×$ over both DuckDB and Kuzu. The advantage is particularly pronounced for star-shaped queries such as Q5, where FFX is $10^5×$ faster than Kuzu. In such queries, data associated with the central join key (a) is repeatedly reused across multiple joins, and our factorized vectors allow FFX to eliminate potential redundancy by processing shared join keys once.

To better understand the performance gap, we inspect the query plans using EXPLAIN-like utilities and compare them with FFX's execution strategy. We find that FFX's main advantage arises when it chooses a join order that yields the most compact intermediate representation and therefore minimizes redundant computation. For example, on Q1, DuckDB and Kuzu evaluate the logical plan $(R(a, b) \bowtie R(b, c)) \bowtie (R(c, d) \bowtie R(d, e))$. Executing this plan requires them to
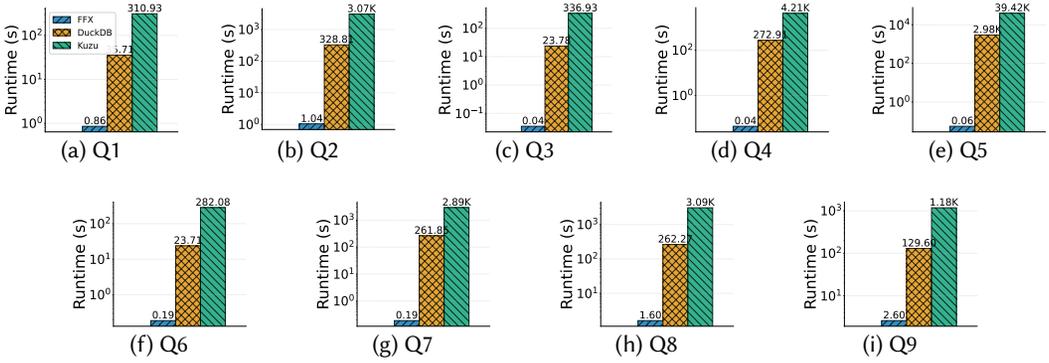
Fig. 9. Comparison of FFX, DuckDB, and Kuzu on `Amazon` for all nine queries, from `Q1` to `Q9`, with `MIN` aggregation. Each system is run in single-threaded mode. FFX achieves speedups: 10^2–$10^5\times$ across queries.
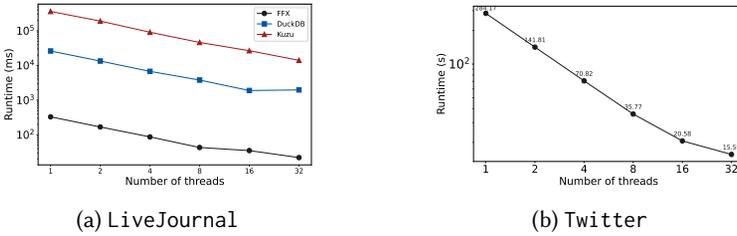

Fig. 10. Multi-threaded performance comparison for query $R(a, b) \bowtie R(a, c)$ on `LiveJournal` and `Twitter` datasets. All systems ran in in-memory mode.

materialize 2.98B tuples before the final projection applies `MIN` to all attributes. In contrast, FFX uses its factorized representation to avoid generating this large result. Instead, it pushes projection and aggregation down to smaller intermediates corresponding to the subqueries $R(a, b) \bowtie R(b, c)$ and $R(c, d) \bowtie R(d, e)$, which together contain only 63.2M tuples.

7.4 Multi-core Scalability

To evaluate FFX's scalability, we execute the query $R(a, b) \bowtie R(a, c)$ on the `LiveJournal` and `Twitter` datasets. Because FFX operates fully in memory, without disk I/O, query execution can be embarrassingly parallel. We compare FFX against DuckDB and Kuzu on `LiveJournal`; comparisons on `Twitter` are omitted because of its larger scale and frequent timeouts in the baselines. Both DuckDB and Kuzu are run in in-memory mode.

Since the `Twitter` dataset does not fit on the hardware described in §7.1.1, we use a machine equipped with 64 vCPUs, 64 GB of main memory, and an `AMD EPYC 9655 (Zen 5) @ 2.7 GHz` CPU. The query is evaluated using up to 32 cores.

Fig. 10 presents the results. All systems exhibit near-linear speedup. FFX remains faster than both Kuzu and DuckDB because of its combined factorized and vectorized execution. On `LiveJournal` and `Twitter`, FFX scales by up to 13.2× and 13.8×, respectively. We note that as the number of cores increases beyond 16, fewer scalability benefits are observed on this specific workload.

7.5 Evaluation of Factorized Semantic Processing

7.5.1 Setup. We evaluate the benefits of factorized semantic processing on the `llm_map` operator. All experiments run on the citation `ArXiv` dataset in Table 2 and apply `llm_map` to the output of a many-to-many join query, allowing us to compare different semantic operator implementations with different serialized data representations.

7.5.2 Dataset. We construct a dataset of academic-paper triplets from 2-hop citations by evaluating:

$$Q_p = Citation(a, b) \bowtie Citation(b, c) \bowtie Paper(a, -, abs_a) \bowtie Paper(b, -, abs_b) \bowtie Paper(c, -, abs_c)$$

where a cites b and b cites c, and `Citation` and `Paper` are base tables defined in §2.3. To keep inference costs manageable while enabling sweeps over batch sizes, operator implementations, and models, we sample the query result by randomly selecting six distinct b values. This yields 2028 (a, b, c) triplets spanning 232 unique papers, including 153 distinct a papers and 77 distinct c papers. We use one separately selected value of b, not among the six, to create a training set to iterate on and improve our prompts. We report the results on the 2028 triplets as a dev set.

7.5.3 Task. For each output tuple of Q_p, we apply `llm_map` to generate five representative keywords from the abstracts to summarize the 2-hop citation chain (a, b, c).

7.5.4 Metrics. Because there is no ground-truth keyword set, we evaluate keyword generation through retrieval. We use the concatenated keywords as a query and attempt to retrieve the corresponding triplet from the full set of 2028 abstract triplets using a hybrid search method that includes BM25. For each query, we report Recall@5 and Recall@10, that is, whether the correct triplet is retrieved in the top 5 or top 10 results, respectively, and we aggregate the results.

7.5.5 Baselines. Our primary external baseline is Lotus [12], a recent system for LLM-powered data processing. For an apples-to-apples comparison, we also compare against an `llm_map` implementation within `FFX` that serializes data as flat tuples for inclusion in prompts (`FFX-Flat`). The operator takes factorized vectors as input, fully enumerates the flat tuples by setting all window sizes to 1, and serializes each row's relevant attributes independently. This baseline reflects the standard way LLM operators are applied to relational intermediates, but it exposes the model to substantial redundancy since many output tuples in Q_p share the same value of b.

7.5.6 Models. We ran our evaluation using `GPT-5.2`, `GPT-4o`, and `GPT-4o-mini`, and observed the same overall trends across all three models. For brevity, we report only the results for `GPT-5.2`. We also experimented with smaller LLMs such as `Qwen3-8B`, which we found to be less reliable at simultaneously applying the Cartesian expansion and at performing the prediction, and therefore we omit them.

7.5.7 Analysis. We compare our approach from §6, denoted here as `FFX-Fact`, against the baselines. `FFX-Fact` serializes prompts directly from packed factorized vectors without flattening the intermediates and adds instructions that describe the f-tree structure and the required Cartesian expansion. In all executions, we use the attribute ordering (b, a, c), which yields the f-tree $b - a, b - c$, *i.e.*, the f-tree rooted at b with children a and c. Under this ordering, each attribute of paper b appears once, while the attributes of the associated papers a and c are emitted as separate lists beneath it rather than repeated across flattened tuples. We use *JSON* serialization.

Table 6 reports the results, grouped by the number of tuples in each batch (# T) processed across successive LLM invocations. For each batch, we report the number of missing output tuples (# MT), caused by failures to produce the full Cartesian-product output, and the corresponding ratio (MR) relative to the expected total of 2028 triplets, which provides an upper bound on recall. We also

# T	System	# MT	MR	R@5	R@10	In. T	Out. T	Run time (s)
1	FFX-Flat	0	0.000	0.7170	0.8023	2.41M	108.44K	88.69
4	FFX-Flat	0	0.000	0.7416	0.8333	1.65M	103.77K	58.14
	FFX-Fact	0	0.000	0.7569	0.8333	0.97M (**1.70×**)	110.51K	63.01
8	FFX-Flat	0	0.000	0.7801	0.8688	1.53M	102.50K	59.67
	FFX-Fact	0	0.000	0.7554	0.8388	0.61M (**2.51×**)	107.60K	61.40
16	FFX-Flat	0	0.000	0.8057	0.8861	1.46M	100.53K	63.45
	FFX-Fact	0	0.000	0.7643	0.8521	0.39M (**3.74×**)	103.62K	71.89
32	FFX-Flat	2	0.001	0.8215	0.8856	1.43M	97.59K	75.16
	FFX-Fact	4	0.002	0.7505	0.8185	0.27M (**5.29×**)	102.57K	98.69
64	FFX-Flat	33	0.016	0.8210	0.8861	1.42M	95.07K	105.48
	FFX-Fact	0	0.000	0.7525	0.8304	0.18M (**7.88×**)	97.79K	112.40
128	FFX-Flat	347	0.171	0.6800	0.7446	1.41M	79.93K	148.04
	FFX-Fact	75	0.037	0.6805	0.7608	0.12M (**11.75×**)	95.56K	174.06
256	FFX-Flat	1199	0.591	0.3319	0.3669	1.41M	39.07K	165.38
	FFX-Fact	517	0.255	0.5439	0.6154	0.09M (**15.67×**)	70.86K	213.80
512	FFX-Fact	976	0.481	0.4004	0.4329	70.14K (**20.10×**)	52.60K	303.78
2048	FFX-Fact	1112	0.548	0.3319	0.3782	62.56K (**22.54×**)	40.29K	164.16
−	Lotus	0	0.000	0.6824	0.7978	2.26M	95.95K	92.68

Table 6. Keyword generation with GPT-5.2. Comparison of factorized semantic processing and semantic processing over flat tuples (FFX-Flat) for the task in §7.5.3. Results are grouped by the number of tuples in each batch (# T) processed across successive LLM invocations. For each batch, we report the number of missing output tuples (# MT) caused by errors when the LLM applies Cartesian products, and their ratio (MR) relative to the total expected output of 2,028, which provides an upper bound on recall. We also report Recall@5 (R@5) and Recall@10 (R@10), the total number of input tokens (In. T), output tokens (Out. T), and run time in seconds under asynchronous execution. Values in parentheses report the achieved input-token reduction. For batch sizes larger than 256 tuples, flat representations lead to errors because the prompts exceed the context window; therefore, reductions are computed relative to FFX-Flat at batch size 256. We note that beyond batch sizes of 128, both approaches lead to degradation.

report Recall@5 (R@5), Recall@10 (R@10), the total number of input tokens (In. T), output tokens (Out. T), and run time in seconds under asynchronous execution.

For the same batch size, FFX-Fact consistently reduces input tokens relative to FFX-Flat, often substantially, while remaining competitive in retrieval accuracy. Across batch sizes for which both representations produce results, FFX-Fact reduces input tokens by 1.70× to 15.67× relative to FFX-Flat; for example, at batch size 256, it uses 0.09M input tokens versus 1.41M for FFX-Flat.

At moderate batch sizes (8 to 64), flat prompting achieves higher recall, with gaps of up to about 0.07 for both Recall@5 and Recall@10. However, at larger batch sizes, FFX-Flat degrades sharply: Recall@5 and Recall@10 fall from 0.8210 and 0.8861 at batch size 64 to 0.3319 and 0.3669 at batch size 256. This drop is driven largely by missing output tuples, which arise when the LLM fails to return the full output tuples for larger inputs. FFX-Fact also degrades beyond batch size 128, but more gradually, reaching 0.5439 Recall@5 and 0.6154 Recall@10 at batch size 256.

`FFX-Fact` also scales beyond the largest batch supported by `FFX-Flat`. While `FFX-Flat` cannot be evaluated beyond batch size 256 because the prompts exceed the context window, `FFX-Fact` continues to operate at batch sizes up to 2048 while keeping prompt sizes low: 70.14K input tokens at batch size 512 and 62.56K at batch size 2048. That said, at these larger batch sizes, the number of missing output tuples also rises substantially, indicating that output generation under large Cartesian expansions remains challenging. Thus, `FFX-Fact` significantly extends the usable batch-size range, but both approaches degrade beyond batch size 128, with factorized prompting degrading more gracefully.

Overall, these results show that, by serializing shared structure only once, factorized semantic processing achieves substantial token savings and avoids the much sharper quality collapse observed with flat prompting at large batch sizes. The main remaining bottleneck as input size grows is the reliable generation of the full Cartesian-product output.

Comparison to Lotus. Under the same `GPT-5.2` setup, `FFX-Fact` at batch size 128 achieves essentially the same Recall@5 as `Lotus` and lower Recall@10 (0.6805 vs. 0.6824 for Recall@5 and 0.7608 vs. 0.7978 for Recall@10), while using substantially fewer input tokens (0.12M vs. 2.26M, an 18.83× reduction). At batch size 64, `FFX-Fact` achieves higher recall than `Lotus` (0.7525 vs. 0.6824 for Recall@5 and 0.8304 vs. 0.7978 for Recall@10) while using 0.18M input tokens, which is about 12.55× fewer than `Lotus`. At no evaluated batch size does `FFX-Fact` use more input tokens.

8 Conclusion

This work tackles a core systems challenge: how to unify factorized and vectorized execution in a single engine. We presented FFX to address this challenge through three main ideas: *packed factorized vectors* that encode arbitrary f-trees while preserving fully packed, cache-friendly layouts; a *cascade update* operator that propagates tuple reductions across dependent bindings without reconstruction or pointer chasing; and factorized semantic processing, which enables semantic operators to serialize factorized intermediates and predict over their implied Cartesian products, producing flat output tuples and predictions without first serializing flat tuples.

Our results show that factorization and vectorization are complementary design choices. Across join-heavy workloads, FFX preserves tight inner loops, reduces interpretation overhead, and achieves substantial speedups over prior systems. When factorization offers little or no benefit, FFX gracefully falls back to standard vectorized execution without additional overhead. For semantic query processing, FFX reduces prompt sizes substantially by up to 18.83× compared to `Lotus` with little to no degradation in output quality. Overall, FFX demonstrates that factorized representations can be propagated end-to-end to optimize join-heavy analytical queries, including those augmented with LLM-powered semantic operators.

Acknowledgments

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. *SIGMOD* (2016).

[2] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. *FOCS* (2008).

[3] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* (2013).

[4] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. 2012. FDB: A Query Engine for Factorised Relational Databases. *PVLDB* (2012).

[5] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD*.

[6] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *PVLDB* (2024).

[7] Peter Boncz. 2002. Monet; a next-Generation DBMS Kernel For Query-Intensive Applications. *PhD Thesis, Universiteit van Amsterdam, Amsterdam* (2002).

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. (2005).

[9] Radu Ciucanu and Dan Olteanu. 2016. *Worst-Case Optimal Join at a Time*. Technical Report. Department of Computer Science, University of Oxford. https://www.cs.ox.ac.uk/dan.olteanu/papers/co-tr16.pdf First version: November 2015. Latest version: March 2016..

[10] Asim Biswal et al. 2025. Text2SQL is Not Enough: Unifying AI and Databases with TAG. *CIDR* (2025).

[11] Chunwei Liu et al. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. *CIDR* (2025).

[12] Liana Patel et al. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *CoRR* abs/2407.11418 (2024).

[13] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE* (1994).

[14] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: the who to follow service at Twitter. *WWW* (2013).

[15] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *PVLDB* (2021).

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2021. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR* abs/2005.00687 (2021).

[17] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees Over Normalized Data Using Only SQL. *PVLDB* (2023).

[18] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. *CIDR* (2023).

[19] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *SIGMOD* (2024).

[20] Hasara Kalumin and Amol Deshpande. 2025. Optimizing Queries with Many-to-Many Joins. In *ICDE*.

[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD*.

[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* (2018).

[23] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join?: Thinking Twice about Joins before Feature Selection. In *SIGMOD*.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? *WWW* (2010).

[25] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. *SIGMOD* (2024).

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* (2015).

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Jiahao Liu, Xueshuo Yan, Dongsheng Li, Guangping Zhang, Hansu Gu, Peng Zhang, Tun Lu, Li Shang, and Ning Gu. 2025. Improving LLM-powered Recommendations with Personalized Information. (2025).

[29] Antonio Lombardo. 2016. *Storage layer for factorized databases*. MSc thesis. University of Oxford. https://fdbresearch.github.io/papers/AntonioLombardo_Thesis.pdf New College.

[30] Amine Mhedhbi, Amol Deshpande, and Semih Salihoglu. 2024. Modern Techniques For Querying Graph-structured Databases. *FnT DB* (2024).

[31] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2021. A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems. In *ICDE*.

[32] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. *TODS* (2021).

[33] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. *GRADES-NDA* (2021).

[34] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB*. (2019).

[35] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. *NSDI* (2016).

[36] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. (2012).

[37] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *TODS* (2015).

[38] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *PVLDB* (2022).

[39] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. *SIGMOD* (2019).

[40] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* (2013).

[41] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDBJ* (2020).

[42] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. 2025. Logical and Physical Optimizations for SQL Query Execution over Large Language Models. *SIGMOD* (2025).

[43] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An Engine for Batches of Group-By Aggregates. *PVLDB* (2020).

[44] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. *PVLDB*. (2025).

[45] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS. *CIDR* (2023).

[46] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *PVLDB* (2023).

[47] Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, Weirong Jiang, and Yi Wang. 2024. NetAssistant: Dialogue Based Network Diagnosis in Data Center Networks. *NSDI* (2024).

[48] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. *CIDR* (2024).

[49] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. *PVLDB* (1981).

[50] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *SIGMOD* (2025).

[51] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *ICDE*.