The Data World Is Not Flat: Efficient Factorized Execution for Relational Systems

Stefan Lehner
Technische Universität München
s.lehner@tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Relational database management systems (RDBMSs) struggle with efficient execution of graph workloads, particularly those involving many-to-many joins with large intermediate results. Such joins are common in modern applications such as social networks, recommendation engines, and knowledge graphs. Traditional RDBMSs are optimized for selective joins, while the typical result blow-up in graph analytics leads to exceptionally long query times or even system crashes due to out-of-memory errors. Since graph and relational data are often intertwined, a unified solution can simplify system architecture and improve performance.

To address this, we propose a novel code-generating engine with factorization, which represents tuples in a hierarchical form. Our approach enables intra-query-parallelized query execution on factorized representations and generates code to overcome their CPU-unfriendly layout. Additionally, we extend current factorized approaches by *bottom-inserts* to make joins reorderable.

Our factorized approach can outperform traditional tuple representations and state-of-the-art RDBMSs by orders of magnitude on synthetic workloads and benchmarks as our experiments show. Moreover, its potential integrability into traditional RDBMSs enables efficient processing of complex, highly redundant queries, offering a unified and scalable solution for modern workloads.

PVLDB Reference Format:
Stefan Lehner and Thomas Neumann. The Data World Is Not Flat: Efficient Factorized Execution for Relational Systems. PVLDB, 19(11): 3006 - 3019, 2026.
doi:10.14778/3836663.3836669

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lehners/FactDB.

1 INTRODUCTION

Graph processing has become ubiquitous, powering applications in social networks [2], recommendation systems [40], and knowledge graphs [19]. This has led to significant advances in graph algorithms and graph database management systems (GDBMSs). However, a recurring challenge in graph workloads is the prevalence of **many-to-many (m:n) join operations**, which often generate large intermediate results (cf. diamond problem [9]).

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836669

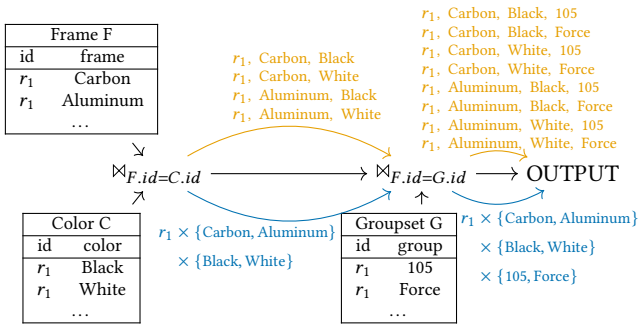

Figure 1: Reduction of processed tuples with factorization: Flat execution (orange) generates many duplicates, which can be avoided with factorized execution (blue).

These exploding intermediates are challenging for modern systems, as Figure 1 illustrates: Joining frame types and colors for a bike model (r_1) produces four intermediate tuples, each representing a unique combination. In a flat execution (orange path), these four tuples are joined with gear options and generate eight tuples in total. These tuples can be represented much more compactly by extracting common factors (blue path). This is the idea of **Factorization** [36]: Represent data compactly and nested in *f-representations* to eliminate duplication without information loss, which aligns better with the hierarchical nature of graph data. With factorization, we can reduce intermediate result size exponentially [36], which allows subsequent operators to process fewer values. E.g., in a factorized execution (blue path), the join with `Groupset` is evaluated only once on r_1 and not for each duplicate. Although factorization tries to reduce redundancy, it cannot fully avoid it; however, it guarantees not to exceed the flat redundancy. Thus, factorization significantly reduces the join costs by avoiding redundant evaluations of duplicates in graph workloads.

First GDBMSs introduced factorization with promising results [16, 21, 23]. However, graph and relational data are often intertwined but distributed across systems. Merging this distributed data leads to integration overhead, inconsistencies, and costly data movement. Meanwhile, **RDBMSs** are mature and widely deployed, but still lack efficient support for graph workloads and m:n joins.

This mismatch raises an important question: Why not enable graph processing directly within RDBMSs? Recent work [3, 4, 49, 50] has extended RDBMSs with graph query capabilities, but a key challenge remains: Relational engines are optimized for selective joins. They are designed for traditional workloads where tuples usually join with at most one other tuple, an assumption that breaks down in graph workloads with high fan-out joins. Years of research

have led to two predominant system designs: (1) Vectorized systems like DuckDB, where already first attempts with factorization were made, e.g., in worst-case optimal joins (WCOJ) [17] or supporting specific joins [22]. (2) Code-generating systems like Umbra [32], which produce query-specific executable code. In this work, we focus on the latter approach and combine it with the best from the factorized world by **performing factorization within a code-generating relational database engine** for SQL.

Early systems already proposed first factorized approaches and can be roughly divided into two groups: systems performing factorization on attribute-level as well as table-level. FDB pioneered the field using attribute-level factorization [6]. After multiple tree transformations, they can merge f-representations. Subsequent approaches focused successfully on table-level factorization, where base table rows are considered as indivisible [16, 21–23]. Some of these approaches focused on GDBMSs and *adjacency lists*, which are very efficient data structures for predefined join paths [16, 21, 23].

While factorization performs well in theory, efficient support in real systems seems challenging for three reasons:

(1) Tree-like f-representations require indirections leading to worse performance than flat, sequentially-stored tuples.
(2) The shape of the tree is highly dependent on the data and suboptimal for modern CPUs and branch prediction.
(3) Simultaneous optimization of f-representations and join ordering leads to a high-dimensional search space.

As a consequence, factorization only pays off under high duplication, while for low duplication the overheads often outweigh. Recent code-generating approaches address the first two problems [43–46], but either target aggregate queries in machine learning workloads or interleave data processing with its representation, leaving the third problem unaddressed. Therefore, we implement a relational engine and make the following contributions:

(1) We leverage code generation to express factorized data in dedicated data structures with minimal runtime overhead.
(2) We prototype a code-generating push-based query engine that adapts query execution to assemble f-representations and to evaluate factorized code generation.
(3) A novel join strategy makes factorized joins symmetric, reducing the search space for join ordering (cf. Section 4.2)

We overcome performance bottlenecks of complex factorized systems and extend FDBs' merging strategies for f-representations to support bag semantics. Our extensive experimental evaluation on synthetic and real-world workloads confirms the significant theoretical performance gains of factorization. It shows speedups of over 25× and outperforms traditional RDBMSs on over 19% of benchmarked CE queries [11].

We begin the paper by providing background on factorization (Section 2) and an example on factorized query execution (Section 3). Then, we outline our approach and describe the necessary adaptations to a code-generating query engine (Section 4). We discuss parallelization and optimizations (Section 5) before the evaluation (Section 6) and a discussion of related work (Section 7).

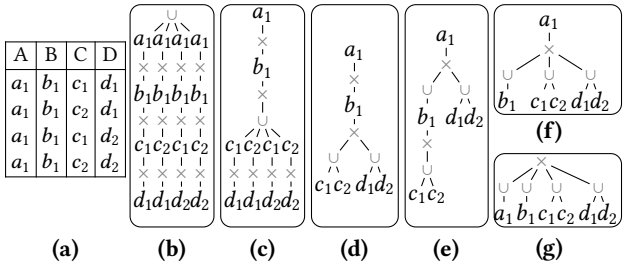

Figure 2: Different tuple representations. The flat representation (a) can be represented as an algebra tree (b). We factor out the common prefix (c), extract cross products (d), and push D (e) and C (f) up and A down (g).

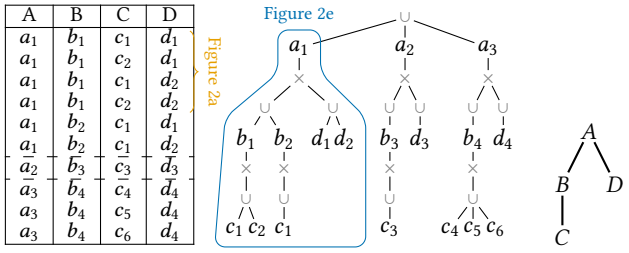

Figure 3: Representations of a set of tuples used as running example: flat representation (left), operator-based f-representation (mid), and the underlying f-tree t_{ex} (right).

2 BACKGROUND

In RDBMS, redundancy can be avoided in tuples with factorization, which extracts common factors [36]. We illustrate this now and introduce the most important definitions from [36].

2.1 From flat to f-representations

Figure 2 shows different representations of 4 tuples: A RDBMS stores them typically in *flat* form (cf. Figure 2a). These tuples can alternatively be expressed using algebraic constructs (cf. Figure 2b), where each tuple is represented as a Cartesian product of values, and the full relation is their union. This representation, known as an *f-representation* [36], uses:

- Atomic values (a_1, b_1, c_1, …) as tuples
- Cartesian products (×) to form tuples,
- Unions (∪) to express alternatives.

This initial f-representation requires 16 values, where many values are duplicates. To reduce this redundancy, we apply a series of transformations. We can observe that all tuples share the prefix a_1b_1. Hence, we apply distributivity of unions and Cartesian products and extract the prefix to represent the values with only 10 different values, but still with some redundancy (cf. Figure 2c).

Further, all pairs of $\{c_1, c_2\}$ and $\{d_1, d_2\}$ occur in the suffixes. These 4 suffixes can also be represented as a Cartesian product (cf. Figure 2d). Because the D values are independent of B values, they can be moved higher in the structure (cf. Figure 2e). Analogously, we can move C upwards (cf. Figure 2f). This step can be repeated for B, C, and D to obtain the f-representation in Figure 2g.

The first three transformations yield the blue subtree in Figure 3. By applying analogous transformations to the extended tuple set in Figure 3, we can derive the full f-representation. We can add singleton unions since they are idempotent.

The transformation reduces duplication of individual values and allows to represent the 4 tuples from Figure 2 with only 6 values. Using common structures (f-trees), we show how to implicitly capture algebraic operators in the following. I.e., we only store these 6 values in memory, which reduces the memory requirements for materialization by 62.5% (see Section 4.1 for its structure). Additionally, subsequent operators benefit from these compact representations: If an operator only needs values from column A, it processes the extracted A value once and remembers the cross products with other attributes afterward. This saves unnecessary evaluations.

2.2 Basics of factorization

Considering the f-representation from Figure 3, a global tree structure becomes apparent: Values from A are top-most, followed by descendants from B and D, and C tuples follow B tuples. The operator tree combines these tuples with Cartesian products followed by unions. This structure is possible for every set of tuples: Individual values must be connected to the corresponding tuples with cross products. If no union exists, it can be inserted at any point since it is idempotent for a single value (cf. c_3). Hence, every tuple can be transformed into f-representations satisfying this operator order.

F-trees [36] formalize the shape of f-representations. As exemplified in Figure 3, f-trees contain respective attributes and show their arrangement in the f-representation. The f-representation then presents a forest of such f-trees: Each tuple from A has child nodes containing B and D tuples and B nodes have, in turn, children containing tuples from C. Similar to previous work, we follow the approach that f-trees specify the structure of the f-representation in the execution engine [6, 18]. This uniform structure enables efficient processing of data and simplifies access patterns.

F-trees also model dependencies of attributes: An attribute is *dependent on* the attributes of its ancestor nodes [36]. E.g., B is dependent on A since A is its direct predecessor. Since C is a successor of B, B is not dependent on C. But, in turn, C depends on B.

The structure of f-trees reveals that the tuples of a node depend on the tuples of all its (transitive) ancestor nodes. E.g., the choice of the tuple from A limits the tuples in its descendant B. Hence, f-trees can naturally model domain restrictions. These dependencies capture, e.g., how joins and filters constrain the set of possible tuples in the child nodes. E.g., the dependency between the ancestors A and B can model the join condition $A.a = B.b$. In combination with dependencies from other operators, the whole hierarchical structure of the f-tree is generated (cf. Section 3).

Dependent attributes lead to the *Path constraint* [6]: Since each pair of dependent nodes must be ancestors in the f-tree, both nodes must reside on a common root-to-leaf path. In turn, a root-to-leaf path must exist for each pair of dependent variables in the f-representation. I.e., for each predicate, all included variables must occur on a root-to-leaf path in the f-tree. Conversely, Cartesian products represent independent attributes like B and D. However, both depend on A, the parent of the Cartesian product.

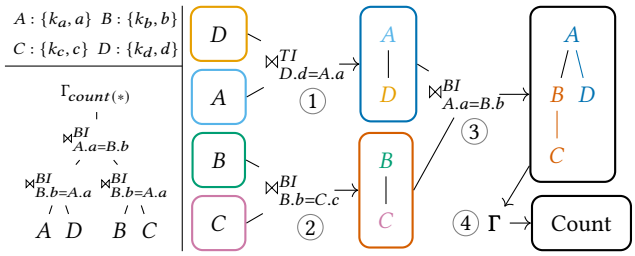

Figure 4: F-trees at different stages during query execution with top- (TI) and bottom-inserts (BI). The colors in the f-tree match the colors of the corresponding input f-tree's boxes.

3 QUERY EXAMPLE IN OUR ENGINE

Factorized execution requires some changes to traditional engines. To understand them better, we now illustrate a query execution in our engine (cf. Figure 4). Therein, our factorized engine keeps intermediate results in f-representations. We execute the given plan and do not consider transitivity for simplicity.

Let's assume that the tables A and D are joined first ①. While a flat engine reads both tables and creates a flat result, a factorized engine avoids potential duplication of the tuples by generating f-trees. Therefore, it places one side of the join below the other side. This enables two variants: ***top-* and *bottom-inserts*** (cf. Section 4.2) which **describe where the probe side is placed in the merged f-tree**. In the example, the engine reads the flat tables and uses a top-insert to create an f-tree with A as root.

In the plan, a bottom-insert between B and C is executed next ②. I.e., the probe side (C) becomes the lower part of the resulting f-tree. The different join types are necessary because the form of the f-tree significantly influences the duplication (cf. Section 2.1). In practice, the optimal position for inserting new attributes depends on the data distribution and must be determined by the optimizer. We prove in Section 4.2 that a top-insert with C on the build side generates the same f-tree. However, if C is large, the optimizer prefers to build the join index on B and uses a bottom-insert.

Continuing the example, the engine executes a third join ③, which merges the two input f-trees. By making B dependent of A in a bottom-insert join, it generates the f-tree from the introductory example. The join condition only requires tuples from A and B. So it suffices for the engine to scan the root nodes of the input f-trees and avoid iterating through the matches from the previous joins. I.e., it can join A with B without considering already retrieved matches with C or D. So each AB pair can be generated regardless of whether it already joins with 1 or 1000 values from C. After the join, the engine assembles the final representation with the respective C and D matches such that all tuples are stored in the representation.

The last operator is an aggregation ④ counting the number of tuples in the join output. This aggregation takes the f-representation and generates a flat value, which is the result of the query.

4 OUR FACTORIZED QUERY ENGINE

After exemplifying the functionality of our engine, we introduce it now in detail. We use f-representations to represent tuples during execution and only flatten them to match the flat output format

of relational engines. This necessitates some adaptations to the algebraic operators of traditional query engines. For the scope of this paper, we only consider table scans, inner joins with equality predicates, and simple count(*) aggregates. However, it is possible to adapt our approach also to other operators. We only consider an in-memory setting and leave spilling open for future work.

First, we describe our system on a conceptual level: In Section 4.1, we describe how factorized tuples can be stored efficiently in memory and develop a data structure for this. Next, we derive f-representations from the relational algebra tree (cf. Section 4.2), which makes tree transformations necessary in some cases (cf. Section 4.3). Then, we dive deeper into the actual implementation of joins (Section 4.4) and count(*) (Section 4.5). Section 4.6 finalizes this with the flattening of f-representations. In Section 4.7, we propose an approach to eliminate subtree duplication using d-trees. In Section 4.8, we briefly sketch the extensibility of our approach by adapting it to other join types. Note that the proposed join techniques for the usage of f-representations are not inherent to code generation and can also be applied in other systems.

4.1 Simplified f-representation in containers

Factorized tuples are stored in f-trees, which represent tuples in a hierarchical form. But this comes at a high cost in runtime systems: (1) Tree-like f-representations introduce numerous indirections, which result in high memory fragmentation and suboptimal prefetching. (2) The data-dependent structure of f-trees poses challenges for modern CPU features such as branch prediction. E.g., in Figure 3, A has two children, B has one, and leaf nodes have none. If each f-node is traversed only at runtime, the child counts appear for the CPU almost at random. Hence, recursive tree traversals lead to poor branch predictions and degrade execution efficiency.

While the former problem lies in the nature of factorization and is hardly solvable, we can completely eliminate the latter one with code generation: **Instead of determining at runtime for each factorized tuple in each algebra operator which node has which children, we do this once and embed the structure into the executed code.** When the generated code is compiled, it is even possible to apply some low-level optimizations to the structure. However, the generated code can also be interpreted. In addition, we tackle the former problem with code generation: By representing tuples in fixed structs, we only need one level of indirection to store the child nodes. If we have a variable number of children and values, we would need two levels of indirection.

The f-representations themselves are implemented similarly to the approach of Kalumin and Deshpande [22]: By representing unions and cross products implicitly, as in f-trees, we can construct a simplified f-representation that merges the values of a specific child node into a single list (cf. Figure 5). Instead of referring to the lists with offset values in one big list for each node as they do, we use a pointer-based structure as exemplified in Figure 5: The root node `NodeA` encompasses column A and two distinct lists for its child nodes B and D. Due to the independence of values in B and D, these values must be stored in two separate lists. Since B possesses only one child C within the f-tree, it comprises only one list. As C and D are leaves, they solely contain their respective columns. Then, the complete f-representation corresponds to the

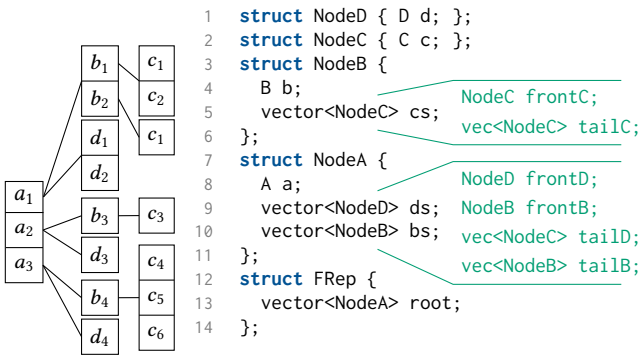

Figure 5: Simplified f-representation (left) and corresponding generated container (right). The simplified representation is derived from Figure 3. Attributes required for inlining (cf. Section 5.3) are shown in green.

root A (cf. `FRep`). Observe that this representation reduces to a list of tuples when the f-tree consists only of a single node. I.e., the representation is then equivalent to the flat representation.

With our pointer-based approach, we can insert elements at each child independently. The offset-based approach of Kalumin and Deshpande [22] only allows fast insertions to the last element of each node, since the different nodes are stored consecutively. FDB produces a sequential representation for serialization through a depth-first traversal of the f-representation, which merges union nodes and stores only the specific values, as we do [6, 8, 38]. Since this hinders efficient insertions, they use a pointer-based tree for the processing step, which even materializes unions and product nodes. With our approach, we can overcome these limitations and can support d-representations (cf. Section 4.7).

4.2 From relational algebra trees to f-trees

To utilize these f-representations within the query engine, we adapt some algebra operators. Since this representation can be flattened at any point, our system can extend traditional execution engines. We derive the different f-trees bottom-up along the algebra tree.

4.2.1 Table scan. Our factorized engine operates on flat base tables and interprets them as f-representations. As we apply table-level factorization, the corresponding f-tree consists only of one node with all attributes. As such f-representations are identical to flat tuples (cf. Section 4.1), **we can use a flat table scan and reinterpret the tuples** at the end. This equivalence eliminates the need for special handling of flat and factorized tuples in other operators.

4.2.2 Count(*) aggregates. The output of the count(*) aggregates is a scalar value, and so, we can again interpret the result as f-representations as for table scans. Additionally, these aggregates are often the topmost operator in the algebra tree, which should produce the flat query result anyway.

4.2.3 Inner joins. Relational equality joins find matching pairs between two relations. Thereby, one tuple from the build side may join with multiple tuples from the probe side. So, if we would just concat the tuples, the build tuple may be duplicated many

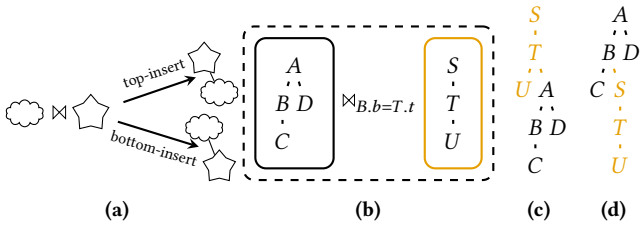

Figure 6: Factorized join variants: top- and bottom-insert. (a) shows the abstract joins, (b) the joining f-trees, and (c) and (d) show the f-tree t_{out} after a top- and bottom-insert.

times, which factorization should avoid. In turn, also tuples from the probe side may contain many matches, which would duplicate them. Hence, the tuples should not simply be concatenated, but rather create a hierarchical structure. Combinatorially, there are **two possibilities to assemble the f-representations**: the build side above the probe and vice versa. In the following, we introduce both approaches as *top-* and *bottom-insert*, depending on the insert position of the probe side (cf. Figure 6a).

Since **top-insert joins align** better **with traditional joins**, they have already been **implemented in some systems** [16, 22]. But, to our knowledge, **we are the first to propose bottom-insert joins**. FDB can also perform different join strategies [6, 12]. However, since their system works on set-semantics, they have no duplicates and can simply concatenate the matching tuples.

Top-insert. Top-insert joins follow the natural behaviour of joins: Usually, joins first build an index and probe it with each probe tuple. Thereby, the index must be somehow grouped by the probed tuples to find matching tuples quickly. In chaining hash tables, a bucket contains only values with the same hash. In sorting, the sorted relation contains ranges with equal values.

In top-inserts, the engine remembers in the subsequent operators that there is this set of matching values. So, the engine stores this set as a child of the probe tuple. The theory of factorization formalizes this: The join creates a dependency between the nodes B and T (cf. Figure 6b). When we insert the build tree below the probe tree, T must become an ancestor of B due to the path constraint. At the same time, B remains dependent on A. Due to the join, A may also get filtered as a consequence of B values without a join partner. So, we can insert A as a child of T or vice versa. But, to ensure good reusability of f-trees, we insert A like B below T. Analogously, C and D become dependent on T, and we insert the f-tree from the build side as child of T (cf. Figure 6c).

Technically, inserting the build tree below U also satisfies the path constraint. But then the build tree is also dependent on U, which is not covered by any join condition. So this corresponds to unnecessary flattening, yielding less succinct f-representations. This should be avoided. Hence, we always place the build tree in top-inserts as a child of the joined node from the probe side.

If the join condition involves multiple nodes, the build tree becomes a child of all required nodes in the probe f-tree. Thereby, a node is *required* if it is either part of the join condition or an ancestor of a required node. If all required nodes lie along a single root-to-leaf path, we can insert the build tree as before. However, if they are not, the f-tree must first be transformed to create such a path. Before exploring different transformation strategies, we first discuss bottom-insert joins, which also need this step.

Similarly, the build tree must also have a root-to-leaf path: When the join involves nodes from different paths within the build tree, these nodes become interdependent, as their domains now influence one another. Consequently, a root-to-leaf path is constructed for the build tree as well if none exists.

Bottom-insert. The second option for joining f-trees is to place the probe f-tree below the build side. Therein, the join again adds the dependencies between B and T (cf. Figure 6b). But now, the build f-tree becomes the tree's upper part. Analogously to the top-insert case, the probe tree becomes dependent on B and consequently its ancestor A, but not on C and D. So, we insert the probe tree as child of B (cf. Figure 6d). This analogy to top-inserts also holds for nodes with multiple required nodes. I.e., we first create a root-to-leaf path in the upper tree to insert the probe tree. As for top-inserts, the lower tree requires a root-to-leaf path to depict all dependencies.

Since both approaches differ only in which f-tree becomes the upper one, we can derive on the logical level

$$L \bowtie_{TI} R \equiv R \bowtie_{BI} L$$

Intuitively, L becomes in the top-insert the lower part of the tree. This corresponds to the definition of the probe tree of bottom-inserts. Thus, a top-insert join can be expressed as a bottom-insert join by swapping the build and probe sides.

This raises the question: *Why do we need bottom-insert joins at all?* The overall goal of factorization is to represent data more succinctly, which leads to performance improvements. Consider two relations B and P, where B is small with unique join-key values, while P is orders of magnitude larger and contains many duplicates. Since building an index is expensive, we prefer to build it on the smaller relation B and use P for probing [10]. In top-inserts, the probe f-tree becomes the upper tree. Because B is unique, each tuple in P will only join with at most one tuple from B. So, factorization cannot pay off. In contrast, B values may join with many P values due to its duplication. Storing shared B values in the upper tree would be highly beneficial. Hence, we rather want P as the lower part. This can be achieved either by swapping the build and probe sides of the top-insert or by using a bottom-insert. However, the former requires building an expensive index on P, which we want to avoid. So, the natural choice is a bottom-insert. This becomes even more attractive when an index on B already exists.

This symmetry decouples the problem of finding a good join order from the structure of the f-tree. Since both problems depend on the query and data distribution, they should be jointly optimized. With the symmetry, the optimizer can limit the search space to only top-inserts when using a symmetric cost function. In a second pass, it can decide between top- and bottom-inserts according to the equality. Afterwards, further optimizations like physical operator selection can be applied. As explained in Section 5.5, we leave the first step open for future work.

4.3 Generating common root-to-leaf paths

An essential step for the merging of two join trees in the join operator is the generation of root-to-leaf paths. As discussed in

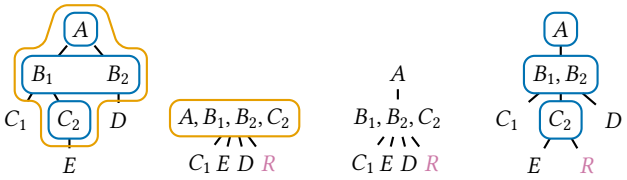

Figure 7: Merging of f-tree nodes for inserting R as dependent node of B_2 and C_2 in original f-tree (left). A naïve approach flattens all required nodes (orange), which can be improved by detecting the least common ancestor A (mid right). Our approach only flattens required nodes in the same tree level (blue).

```
1  void genRoot2LeafPath(FNode& lca) {
2    FNode mergedNode;
3    for (Node& child : lca.children) {
4      if (child.isRequired()) { // merge attr. & children
5        mergedNode.attributes.append(child->attributes);
6        mergedNode.children.append(child->children);
7        lca.children.remove(child); // rm child from LCA
8      } // else keep child nodes as they are
9    }
10   if (!mergedNode.empty()) {
11     lca.addChild(mergedNode);
12     genRoot2LeafPath(mergedNode);
13   }
14 }
```

Figure 8: Generating root-to-leaf path by merging nodes.

Section 4.2, all required nodes have to lie on this path. For instance, in a join involving B_2 and C_2 (cf. Figure 7), the nodes enclosed in the orange contour should be merged to a root-to-leaf path. A *naïve strategy* to generate such a path is to merge all required nodes into a single node. This flattening into a single node corresponds to destroying the factorization of the required nodes (cf. Figure 7).

In the original f-tree, the node A forms the longest common prefix along the paths to B_2 and C_2, and is therefore the *lowest common ancestor* (LCA) of these nodes. Beyond this point, the paths to the required nodes diverge. In turn, a common root-to-leaf path already exists up to A. Consequently, A can be excluded from flattening (cf. Figure 7), which preserves its factorization and reduces the scope of the transformation. This optimization does not affect the number of tuples in the flattened node $B_1B_2C_2$, but it can reduce the duplication of A values in the result. By identifying the LCA of required nodes, this improved strategy can be applied to any f-tree.

Building on this insight, we propose a recursive approach that flattens only required nodes on the same tree level. In each step, the algorithm combines the required children of the LCA into a single node by merging their attributes and children, while all other nodes remain separate children. Once this merging is complete, the newly created node represents the new LCA for the next recursive step. We repeat this process to gradually extend the LCA to the required root-to-leaf path. **So, only required nodes are flattened and other nodes remain factorized.**

Analogously to the improved naïve approach, the duplication of the lowest merged node (C_2) remains unchanged. However, for all preceding nodes, duplication can be reduced since they are only flattened within the same level and not merged with their children. Consequently, our approach is particularly beneficial when subsequent joins depend primarily on attributes in the upper nodes. For instance, we can reduce duplication of A because it is not flattened together with B_1, B_2, and C_2 as the naïve strategy.

4.4 Factorized joins

Having introduced the design of the f-tree structures in the preceding sections, we now turn to the operational aspect: Adapting the operators so they efficiently propagate factorized tuples. Our query engine follows the push-based, data-centric execution model introduced by Neumann [31], which underpins Umbra's state-of-the-art flat execution performance [32]. In this model, tuples are pipelined through operators while data is kept in CPU registers and materialization is deferred as long as possible.

We extend this execution strategy to f-representations. Importantly, these representations remain partially implicit in a pipeline for two reasons: (1) Required-column accessibility: All attributes needed by subsequent operators must be available in flat form, which requires partially flattening the f-representation by traversing all f-tree nodes that contain required attributes (including their ancestors), while non-required subtrees may remain factorized. (2) Merging of f-trees: Operators such as joins must merge tuples from different f-representations, but this merging is deferred until materialization is necessary. During pipelining, operators retain references to both flat and factorized fragments to provide the necessary access without eager materialization.

We adapt this approach for factorized equality joins, which requires construction of the resulting f-tree t_{out} from the build-side f-tree t_{build} and the probe-side f-tree t_{probe}. Our joins are implemented as hash joins with chaining hash tables, leveraging their proven efficiency in RDBMSs [10]. Our design of f-representations enables the support of other join types and algorithms if needed.

Following Leis et al. [26], the hash table is built in parallel by collecting entries in thread-local storage first and then allocating a table of perfect size. In a flat setting, tuples from the probe side can then be probed without synchronization and pushed to the next operator. For factorized joins, we adapt this scheme:

Top-Inserts. As already discussed in Section 4.2.3, top-insert joins are a natural fit for index-based joins: Tuples on the build side are materialized according to their current f-tree. In the flat case, this is just a tuple containing all attributes. In the factorized case, this follows the current f-tree t_{build}. We decouple the storage of the f-representations from the hash table by storing only the flat join keys along with references to the corresponding f-representations, for which we introduce iterators in Section 4.6.

During the probe phase, the join performs hash table lookups to find matching pairs of t_{build} and t_{probe} tuples analog to the flat case (cf. Figure 9, ll. 4-6). Before traversing the matches, the upper part of the output tree is generated from t_{probe} (l. 3). Then, each matching tuple from the build side is inserted into this tree.

Thereby, we want to preserve the factorization from the build side, even along the flattened nodes. Consider the introductory example in Figure 3 with join key C: The f-representations $a_1b_1c_1 \times \{d_1, d_2\}$ and $a_1b_2c_1 \times \{d_1, d_2\}$ share the root a_1 but have different

```
1  void probe(Hashtable ht, auto tuple, FRepOut& f) {
2    map<Prefix, OutFHandle> m;
3    auto rHandle = f.append(tuple); // create upper tree
4    for (auto cur=ht[tuple.key]; cur; cur=cur.next) {
5      if (tuple.key != cur.key) continue; // collision
6      auto [a, b, c] = cur.key;
7      ... // code for subequent pipeline operators
8      if (!m.contains(a)) {
9        m[a] = rHandle.insert(a);
10       m[a].insert(a.getSubtreeD());
11     }
12     auto aHandle = m[a];
13
14     if (!m.contains(a, b))
15       m[a, b] = aHandle.insert(b);
16     auto bHandle = m[a, b];
17     bHandle.insert(c);
18   }
19 }
```

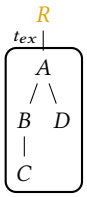

Figure 9: Generated lookup code with immediate materialization for top-inserts for t_{ex} joined with R on $C.c = R.r$.

B values. Since both tuples contain c_1, they join with the same R value to $r_1 \times \{a_1b_1c_1, a_1b_2c_1\} \times \{d_1, d_2\}$. Instead of duplicating a_1 in t_{out}, we want to detect such cases to store a_1 only once. Therefore, we cache all inserted elements with their location in t_{out} during a chain traversal (cf. Figure 9, l. 2). When a new match is found, the algorithm first checks if any prefix of the current match is already generated (ll. 8-16). If necessary, it creates and caches a new entry. For the last required node, we can omit this caching, since factorization avoids fully duplicate tuples. So, if only the root node of t_{build} is required, we can completely omit the cache. Since the key b_1 may occur multiple times with different prefixes, it is always necessary to look up the full prefix (ll. 14-16). In reality, this cache remains small because it persists only for one chain traversal.

Since we delay materialization as long as possible, matches can first be pipelined to subsequent operators (l. 7). This may necessitate flattening columns before proceeding to the next operator. From this perspective, we can interpret flat joins just as a special case of top-inserts, which immediately flatten all attributes.

Bottom-Inserts. In bottom-inserts, a different strategy is required. Unlike top-inserts, bottom-inserts do not align naturally with index-based joins, since the build side now forms the upper part of t_{out}. As a result, a single build-side tuple may join with multiple probe-side tuples, and the lower parts of t_{out} may originate from different probes. Consequently, we cannot guarantee that all relevant lower subtrees have been seen until all probes are processed. To handle this, **we implement bottom-inserts as full pipeline breakers**, which materialize the joined tuples immediately. If the flat columns form a subset of the columns required by the subsequent operators, bottom–inserts can still be half-pipeline breakers. However, we omitted this optimization in our query engine.

In the hash tables, bottom-inserts do not store t_{build} but directly generate t_{out}, where only the nodes stemming from t_{build} are filled (cf. Figure 10). Thereby, it suffices to store a pointer to the insert location of t_{probe} in the hashtable. The join probes the hash table as in the flat case (cf. Figure 11, l. 2-3) and uses the pointers to fill the

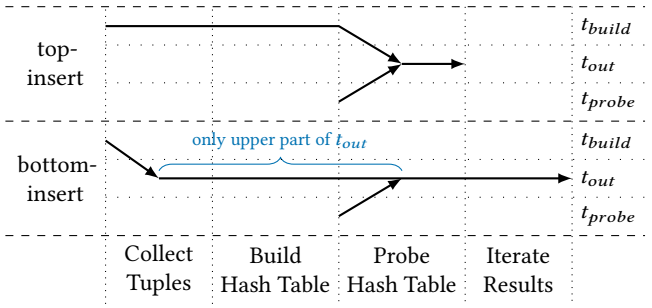

Figure 10: Used representations at different stages of the join implementations. t_{build}, t_{probe}, and t_{out} represent the build side, probe side, and output f-tree, respectively.

```
1  void probe(Hashtable ht, auto tuple){
2    for (cur=ht[tuple.key]; cur; cur=cur.next) {
3      if (tuple.key != cur.key) continue; // collision
4      cur.val.insert(tuple);
5    }
6  }
7  void produce(FRepOut frep) {
8    for (auto tuple : frep)
9      parent.consume(tuple);
10 }
```

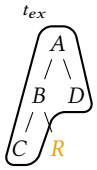

Figure 11: Generated code for lookup and tuple iteration for bottom-insert joins.

lower nodes for each match with the respective t_{probe} (l. 3). Once all probes are processed, the join iterates in a new pipeline through all required columns, and the subsequent operators consume the tuples (ll. 7-10). The materialization of iterators requires them to be valid even after further insertions into the lists. This is also the case for hash tables generated for caching in top-insert joins.

4.5 Factorized aggregations

In a traditional flat setting, the query engine computes the `count(*)` aggregate by simply counting all generated tuples, which directly corresponds to the result size. In a factorized setting, we compute this more efficiently by exploiting the structure of the f-tree and distributivity. E.g., the size of f_{ex} (cf. Figure 3) can be determined by $\#_{flatTuples} = \sum_{a \in A} \left(\left(\sum_{b \in B_a} |C_{a,b}| \right) \cdot |D_a| \right)$. We map this to a naïve dynamic programming approach, which maps the sums to for loops. By replacing explicit enumeration of all tuples with multiplications of subtree counts, we avoid unnecessary work and reduce runtime.

If the tuples are already partially flattened in the pipeline, flat nodes should be omitted and only factorized nodes need to be considered. E.g., in the running example, the for-loops can be omitted if A and B are already flat in the pipeline. D-representations enable even more optimizations for aggregations (cf. Section 5.4).

4.6 Iterating tuples in f-representations

To access attributes, the factorized engine must flatten tuples. Therefore, we can recursively fix entries at a node in the f-tree and iterate over the required entries in its child nodes until all required nodes

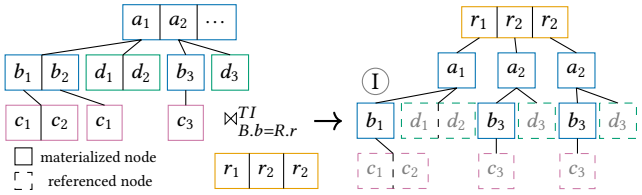

Figure 12: Constructing d-trees in joins. Reference nodes ($C^{\uparrow}$ and $D^{\uparrow}$) store only pointers to the respective nodes before the join and do not re-materialize their values. Nodes required by the join (A and B) must be materialized.

are flat. E.g., in f_{ex}, we iterate for each b entry over its dependent C_b entries. When a node has multiple children, we need to generate the cross product of those children. So, if we iterate over A, we generate all pairs of dependent B_a and D_a values. This can be done in any topological order, e.g., in a pre-order traversal.

For some operators, it suffices to flatten only parts of the f-tree, while keeping the remaining nodes unflattened. Thereby, special care must be taken on empty subtrees, as they occur in bottom-inserts. In particular, the engine must check whether a flattened tuple contains any empty subtrees and skip it if necessary.

4.7 Reduce duplication by d-representations

Our current design enforces copies of the entire tree whenever f-representations are materialized. E.g., when we perform a top-insert of f_{ex} with the relation R on $B.b = R.r$, unchanged C and D are copied. So, these subtrees are duplicated for duplicates in $R.r$.

To avoid this duplication, we use *d-representations*, which were introduced to reference common subtrees [36]. D-representations create aliases for duplicated tuples and refer to them instead of replicating subtrees. They can be described by *d-trees*, analog to f-trees. We implement d-representations with *reference nodes* that store only references to nodes from the input tree. E.g., the reference nodes $C^{\uparrow}$ and $D^{\uparrow}$ point to C and D from the input (cf. Figure 12). Since A and B are required by the join, they may change during the join and must be materialized (cf. Ⓘ). In the generated code, d-nodes use pointers to the subtree instead of lists (cf. Figure 5).

D-trees can reduce the materialization at multiple steps in our implementation: (1) After the lookup, some subtrees may remain unexpanded in the pipeline (e.g. C and D in Figure 12). (2) Not required nodes in t_{probe} can be reused in top-inserts instead of copying them after the hashtable lookup. (3) During transformations for common root-to-leaf paths, not required subtrees can be referenced (e.g., C_1, E, and D in Figure 7).

Note that reference nodes always refer to entire subtrees. I.e., the reference node $A^{\uparrow}$ would reference the subtree containing A, B, C, and D. Moreover, referenced trees may contain reference nodes themselves. This can, e.g., occur if the node A from the output tree in Figure 12 is referenced in subsequent joins. If an already referenced node is referenced again, we apply path compression.

4.8 Adaptation to other joins

The ideas behind our hash join implementations are not limited to inner joins. Outer joins extend tuples without a join partner with `null`, which is just another value in the domain. Hence, `null` handling is orthogonal to our approach. Naïvely, we can add `null` values into the respective f-tree nodes and adapt standard approaches [33]: For top-inserts, we can detect probe tuples without a join partner in the lookup, while build side tuples are detected with a marker afterward. In bottom-insert joins, we can create an upper f-tree with `null`-values and insert non-matching tuples therein for the probe side, while the build side can again use a marker.

For non-equality joins, Blockwise Nested Loop Joins are usually a good choice [33]. Consequently, we can create the upper part of t_{out} in top-inserts in the outer loop before iterating through the matches for a tuple. For bottom-inserts, the upper parts must be pre-generated as in the equi-join setting. Therefore, we can create a list containing all upper trees before we start the join itself. During the join, we could retrieve the corresponding prepared tree from the respective tuple index. However, these naïve approaches pave the way for various optimizations, which we leave as future work.

5 PARALLELIZATION AND OPTIMIZATIONS

Modern query engines rely heavily on intra-query parallelism [31], which is typically based on parallel processing of individual tuples. In the context of factorized execution, however, an f-representation encodes multiple flat tuples within a single factorized tuple, so adaptations are necessary for efficient parallel factorized processing.

Table scans can be adapted by reinterpreting the results of their parallel flat counterparts. For count(*) aggregates, we can use the flat parallel implementation if tuples are iterated in parallel. **Joins**, however, are a greater challenge, as they **require tuple processing and the materialization of f-representations** (i.e., writes). Despite this complexity, we can still primarily rely on the parallel enumeration of f-representations. Therefore, we describe in the following parallel iteration of factorized tuples and synchronized materialization. Afterwards, we discuss a few simple optimizations applied in our engine. All of the proposed ideas are orthogonal to code generation; however, inlining pays off most in generated data structures.

5.1 Load balancing for f-representations

Considering the running example from Figure 3, it is easy to see that parallel enumeration by simply splitting on a per-node basis may lead to large load imbalances during parallel tuple iteration. E.g., if the tuples are split based on A, one thread has to handle six tuples (a_1), whereas another processes only one tuple (a_2).

Hence, we distribute the tuples more evenly by assigning a specific ID to each element generated by the iterator. This ID corresponds to the position the tuple would occur in a sequential iteration. E.g., the tuple $a_1b_2 \times \{c_1\} \times d_1$ has ID 2 if the pipeline requires B and D, since the two tuples with root a_1 precede it in this specific iterator (only distinct ABD tuples are considered, since C is not required). Similar to morsel-driven parallelism, the engine processes the tuples in parallel by dividing them into chunks using these IDs [26]. The tuples in the chunks can be efficiently enumerated by the corresponding iterator. If we preprocess the f-representation to store in each node in an additional field how many enumerated children it contains, we can also quickly find

the start of a chunk. Then, a specific tuple is found by recursively searching for the ID in the tree nodes in the corresponding order.

We apply this in the running example (cf. Figure 3) to find the tuple with ID 2 while iterating over B and D. First, the node A is considered. Since a_1 holds four iterated tuples, it also contains ID 2, and we can recurse into B. In this step, we have to account for $|D_{a_1}| = 2$, which requires scaling the elements of each subtree of B by 2. Hence, b_1 contains 2 iterated tuples, namely the IDs 0 and 1 (C is not required) and, so, b_2 contains ID 2. Finally, D is considered, which does not have to be scaled since it is the last child of A. Then, d_1 represents only one iterated tuple, which is the searched ID 2.

Since the root node usually contains many elements, we optimize index access by pre-calculating the prefix sum of the sizes, which corresponds to the ID of these nodes. Then, we find an iterator by ID in $O(\log n)$ using binary search which only needs to be performed at the start of each chunk. So, if the chunks are sufficiently large, the number of different chunks and the impact on the total runtime is small. For inner nodes, we can apply a linear scan since they are usually small and fit into a couple of cache lines.

5.2 Factorized tuple materialization

Materializing tuples into f- or d-trees is done in parallel and may therefore require synchronization. Since top-inserts find all matches for a probe tuple in one chain traversal (cf. Section 4.4), they occur sequentially in our setting. So, only the generation of the upper part of t_{out}, which corresponds to t_{probe}, has to be synchronized since multiple threads probe different tuples in parallel.

In bottom-inserts, the upper parts of the d-trees are generated during the hash table build in parallel. So, probes require synchronization when different tuples map to the same key and the same upper f-trees, as concurrent insertions to the same nodes may occur.

To handle these synchronizations efficiently, we employ a combination of two strategies: (1) We associate a lock with each inner node that may be accessed concurrently. (2) We store the root nodes of the trees in thread-local storage. This is motivated by the observation that root nodes typically have high cardinality, while the lists in inner nodes remain small. So, contention on inner nodes is limited, and synchronization overhead is minimal. Since a factorized tuple can represent multiple flat tuples, we can accept the overhead for locking.

Note that we can construct the f-representations with only two functions: We have to create the upper and the lower parts of t_{out}, which correspond to t_{build} and t_{probe}. Depending on the join type, the input trees are mapped to the respective function. This can greatly simplify the implementation effort of factorization.

5.3 Inlining storage to reduce indirections

The tree-based storage of f-representations can offer significant performance benefits, particularly in scenarios where flat execution would result in substantial tuple duplication. However, flat execution still has advantages when no such duplication occurs. To address this trade-off, we optimize the f-representation by inlining the first element of each child list into the parent node (cf. Figure 5). Hence, materialization needs no indirections if tuple join only with one other tuple. The storage layout resembles a traditional flat, row-based format and can be interpreted as adaptive factorization.

5.4 Caching subtree results during aggregation

In d-representations, count(*) aggregates are calculated repeatedly on the same subtree if they are referenced multiple times. The same issue arises when a join produces multiple matches for a build-side tuple and passes them immediately to an aggregate. To address this, we cache the size of each subtree after its first computation. Since we only need to cache the sizes of unflat subtrees, we can reuse the existing ID field from the parallel iteration strategy.

5.5 Optimizing query plans

In a factorized query engine, new optimization objectives arise with the need for new cost models. Traditional cost-based optimizers in flat database systems, such as PostgreSQL, aim to minimize CPU and I/O costs [27]. However, f-representations alter the underlying cost dynamics. As a result, Bakibayev et al. [6] proposes optimizing for the size of the f-representations instead. We adopt a similar optimization strategy using true cardinalities for optimizer decisions. For a fair comparison, our baseline also uses true cardinalities. The full development of the optimizer is left for future work.

6 EVALUATION

We investigate whether factorization reduces query execution time compared to flat execution, how our compiled f-trees compare to existing engines, and which optimizations contribute most to performance. We compare our factorized engine against a flat baseline built on the same infrastructure, generating query-specific `C++` code compiled with `GCC 14.2.0` (`-O3`). This approach could also be applied to interpreting engines. We benchmark query performance and compare systems on (1) synthetic microbenchmarks joins under **Zipfian** and **uniform** distributions, (2) the CE benchmark [11], and (3) the SSB [37] for a full system comparison. The experiments were executed on an Intel Xeon Gold 6430 CPU (64 cores) with 256GB of RAM running Ubuntu 24.04.1 LTS. We executed 5 warmup runs and report the average runtime of 10 measured runs. We used bushy plans if not stated different. We exclude query compilation times from our measurements as prior work has shown that code-generation frameworks such as LLVM enable the generation of highly efficient machine code with small latency [31]. This latency can be hidden by a flying start in adaptive execution engines [24]. Our approach is fully compatible with such frameworks and can similarly achieve low compilation overhead. We compared against DuckDB 1.4.2 [39] and Umbra [32]. Microbenchmarks were also evaluated on FBench 1.0 (FDB) [36], LMFAO [43], and KÙZU 0.11.3 [21]. Since FDB requires manual specification of the factorized tree and Kuzu timed out during the execution of the CE benchmark, we did not include their results. LMFAO was omitted since it aggregates eagerly, which is not completely comparable.

6.1 Microbenchmarks

We conducted microbenchmarks measuring top- and bottom-insert performance. Each query consists of a left-deep join tree over three relations with three columns each, followed by aggregation (cf. Figure 13). The topmost join executes a top-insert to pipeline results into the final aggregation, while the preceding join uses the benchmark's respective join type. We compare our engine against

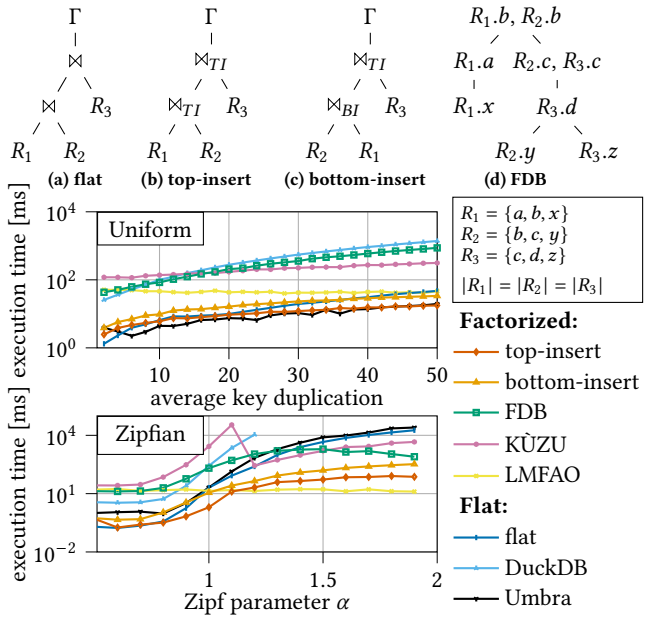

Figure 13: Performance on microbenchmarks. Each relation contained 100,000 (Uniform) and 10,000 (Zipfian) elements. Top-, bottom-insert and FDB executed the plans shown in b)-d). All other systems got SQL and Cypher statements equivalent to a). Top- and bottom-inserts show state-of-the-art performance and outperform flat systems.

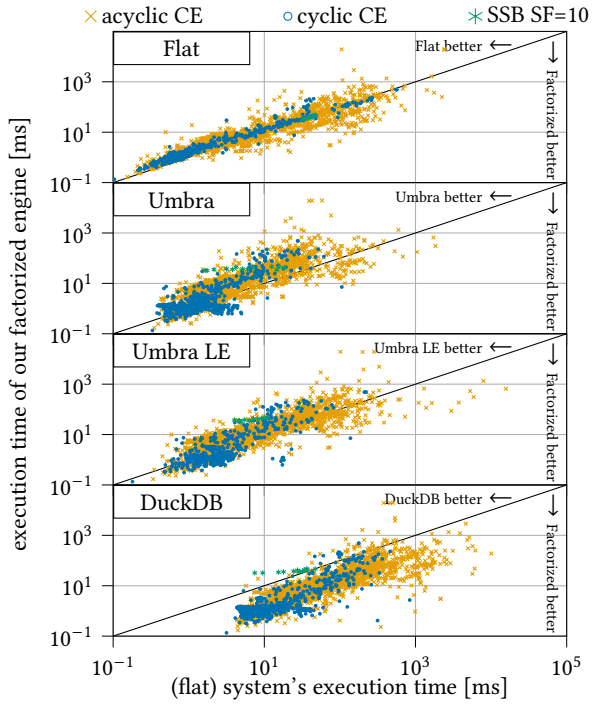

Figure 14: Factorized runtimes vs. flat execution, Umbra, Umbra with lookup-expand, and DuckDB, respectively. Factorization can significantly speed up execution times.

Umbra and DuckDB by executing equivalent SQL queries. For FDB, we model the factorization structure as shown in Figure 13. For KÙZU, we represent distinct keys as nodes and relations as edges.

Figure 13 present the evaluation results for *uniformly distributed* integers with growing duplication. In the case of no duplication, factorized approaches cannot reduce the number of tuples for subsequent operators. So factorization pays off especially for large duplication, as the performance gap between our flat and factorized engine shows. Since bottom-inserts require an additional materialization step, they are slightly more expensive (1.9×) than top-inserts. However, they can still outperform our flat implementation by over 13*ms* (1.4×). The expected small performance gap between our flat engine and Umbra confirms our goal to provide a simple yet comparable baseline. Umbra's speedup is primarily driven by its unchained hashtable, which reduces pointer chasing and improves cache locality for long chains [10]. Replacing it with a chaining hashtable, as used in our engine, degrades Umbra's runtime by up to 4.2×, falling below our flat engine. Since this optimization is orthogonal to our work, the gap reflects optimization space available to our baseline and, in particular, our factorized engine, which already slightly surpasses Umbra. DuckDB contains significant overhead in the uniform benchmarks compared to our approach, which we attribute primarily to bad parallelization for these queries. For the factorized engines FDB and KÙZU, we encounter significant overhead. Apart from poor parallelization, we outperform them with compiled f-trees by avoiding expensive runtime dispatching based on the tree shapes. Furthermore, FDBs' codebase reveals that it contains many indirections leading to numerous cache misses [8, 38], which our compiled f-trees also reduce (cf. Section 4.1).

Our factorized approaches still demonstrate competitive performance in scenarios with low duplication. For uniform data, top-inserts achieve performance comparable to flat joins with 10 duplicates on average, and begin to outperform them with 12 duplicates. Bottom-inserts surpass the flat implementation starting from 36 duplicates. As relation sizes increase, the break-even point shifts to lower duplication, since the absolute number of tuples saved by factorization continues to grow. Since higher levels of duplication cause intermediate results to explode (cf. Figure 1), the performance of all engines degrades with more duplicates. Factorization reduces this impact by tree-like f-representations. LMFAO can completely avoid this overhead by eliminating all duplicates in the relations. By repeating this for each operator, as in eager aggregation, the system achieves runtime independent of the degree of duplication. However, since the results are aggregated, it cannot print all tuples instead of the count, as the other systems can. This analogy to eager aggregation resembles the runtimes since this approach only pays off for a large degree of duplication.

Under a *Zipfian distribution*, which introduces skew, behavior changes notably. High skew leads to many matches for a few tuples, causing high duplication when they are materialized flat. This effect is evident in the flat execution engines. For DuckDB, this led to a timeout (15s). We observe problems with large intermediates also for KÙZU, since it executes the query flat for little skew ($\alpha \leq 1.1$). For high skew (large α), its optimizer selects a factorized plan, which drastically reduces runtime and enables KÙZU to outperform flat engines by over 5×. Still, we outperform the other systems with top- and bottom-inserts, while bottom-inserts

again perform a bit worse. Interestingly, FDB achieves for high skew (large α) comparable performance. We attribute this mainly to the attribute-level factorization, which factors out even more duplicates than table-level factorization in this specific benchmark. So, the number of tuples that have to be joined becomes even smaller, and FDB keeps up with our bottom-inserts. Nevertheless, our top-inserts even outperform the flat engines for low skew, where other factorized approaches without code generation are over 10× slower. We attribute these runtime advantages mainly to compiled f-trees and the inlined storage (cf. Section 5.3).

6.2 CE-Benchmark

The CE-Benchmark [11] contains many complex cyclic and acyclic pattern matching queries that stress the engines. Since it was initially designed for cardinality estimation, we exclude result sizes larger than 10^9. Evaluating all 2557 queries for which our oracle computed all estimates within 10s reveals significant improvements in acyclic queries and comparable performance on cyclic ones.

Using our factorized implementation, we observe a geometric mean slowdown of 15% over the flat implementation for all acyclic queries (cf. Figure 14). However, this is mainly driven by short-running queries, where factorization cannot pay off since the f-representation has the same size as the flat representation. For all 895 acyclic flat queries taking over 10ms, factorization achieves a geometric mean speedup of 6.9%; for queries taking over 100ms even 52%. 10 queries exhibit speedups exceeding one order of magnitude with factorization. The results validate that factorization pays off in the 1236 queries containing only m:n joins. Since 37% of all joins are foreign key joins, we see a small slowdown for the CE-benchmark in total. However, our factorized approach shows performance comparable to the flat baseline in queries with only 1:1 joins.

For DuckDB, we achieved speedups of over three orders of magnitude, primarily due to less efficient query plans and a substantially higher number of processed tuples. Umbra took up to 16× longer than our factorized engine. In total, we achieved a faster runtime with factorization in 294 acyclic queries than Umbra and in three queries over one magnitude. We outperformed DuckDB in over 97% of the acyclic queries. Since Umbra has many low-level optimizations, our simple engine has currently a geometric mean slowdown of 58%. However, we consistently outperform Umbra for long-running queries. When comparing our factorization to Umbra with lookup-expand joins (LE) [9], we experience similar results (33% slowdown, 559 queries faster with factorization). Since LE can be interpreted as a variant of top-inserts, this underlines the need for bottom-insert for efficient factorization.

In the cyclic case, the tree usually has to be transformed to close a cycle, which may include merging tree nodes, i.e., flattening their respective elements. Even though we reduce the number of flattened nodes as much as possible, this affects the performance in cyclic joins. Therefore, the geometric mean performance in cyclic cases is currently 32% slower than in the flat engine. This value is mainly driven by many queries that do not profit from factorization, as intermediates cannot be compacted. However, we still outperform the flat implementation in 35 queries and by up to 3.3×. Since the cyclic join problem can be seen as an orthogonal problem, further optimizations for the cyclic case, e.g., with worst-case-optimal joins, may close this performance gap [9, 15, 21, 25, 34].

We measured for our improved transformation strategy a geometric mean slowdown of 2.4% over the naïve approach, which seems counterintuitive at first glance. However, only 1104 queries benefit from more factorized intermediates, of which 736 are cyclic. So, for many queries, the improved approach cannot realize its advantages and suffers slightly from a deeper tree. We observe that the improved approach makes the execution more robust, and we see speedups over 6.2×, if its advantages can pay off. We therefore recommend checking during algebra tree generation whether subsequent operators use the attributes that remain factorized in the improved transformation. If not, use the naïve approach; otherwise, apply the improved approach for robust query processing.

These results on the CE benchmark highlight the practical impact of factorized execution, particularly in acyclic queries where structural redundancy can be effectively exploited. While cyclic queries show fewer performance benefits, the overall findings affirm the potential of factorized processing.

6.3 Star Schema Benchmark

We executed the SSB with scale factor 10 on slightly modified queries: Due to our reduced set of algebraic operators, we dropped the `group by` and `order by` clauses from all 13 queries and counted joined tuples. We constructed the tables without foreign key indexes on all systems for a fair comparison. Since the SSB models a typical analytical workload with many foreign key relationships, this benchmark is challenging for factorized execution.

Interestingly, we can slightly outperform the flat baseline with factorization. At first glance, this seems counterintuitive, but foreign key joins can also include some duplication, which factorization eliminates. This results in fewer processed tuples and improved runtimes with the factorized engine. This corresponds to the results of the CE-benchmark for foreign key joins. Since RDBMSs are optimized specifically for foreign key joins, we perform slightly worse than Umbra but comparable to DuckDB on the SSB.

6.4 Ablation study

We now analyze the different optimizations in detail on the CE benchmark and report speedups over the flat, single-threaded execution (cf. Figure 15). Since over 50% of the executed queries in the CE benchmark involve foreign key joins, factorization using only top-inserts and left-deep plans performs slightly worse than the single-threaded flat baseline for 1983 queries. Since bottom-inserts introduce an additional pipeline breaker, they may degrade performance when factorization does not pay off. However, when factorization is effective, bottom-inserts yield substantial speedups. They were used in 49% of the queries and in 2091 joins in total. 46 queries used exclusively bottom-inserts. It is noteworthy that these single-threaded left-deep factorized plans outperform the parallel bushy flat engine already in 185 queries.

Caching accelerates queries with many d-representations and high branching on average by 30%, while other queries encounter minimal slowdowns. Due to its similarity to the bottom-insert results, caching is omitted from the graph. Inlining writes the first child directly into its parent, which significantly reduces cache

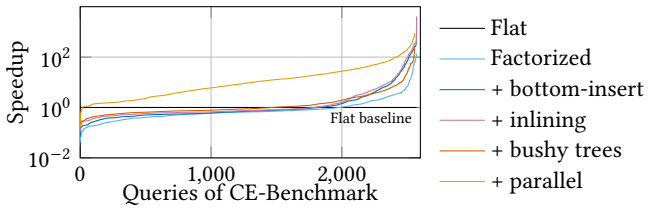

Figure 15: Distribution of speedup over flat, sequential baseline. The features were activated in the order of the labels.

misses and provides modest improvements. If our query engine uses bushy query-trees instead of left-deep plans, more succinct f-representations enable more improvements. The parallel factorized execution boosts the runtime consistently as shown in the graph. On average, it improves runtime by 18×.

7 RELATED WORK

FDB pioneered factorized execution using f-trees and later also d-trees [5, 6, 35, 36]. They demonstrated the potential of factorization by exploring the theoretical foundations. Since FDB performs factorization at the attribute level, it eliminates more duplication, but requires potentially expensive factorization of the base tables. FDB was not optimized for performance on large datasets or integration into modern query engines. The GDBMS Kùzu and its predecessor GraphflowDB introduced limited support for f-representations within a vectorized engine [18, 21, 23]. They store intermediate results in vector groups, where entries can be lists of values, but these structures cannot express arbitrarily nested f-representations like our system. They implement specialized join algorithms tailored to graph workloads, leveraging the adjacency list representation of graphs to exploit factorization. Similarly, the commercial graph engine GES [16] applies factorization in its Expand operator to group neighbors based on adjacency lists. This approach would incur expensive pre-computation steps in RDBMSs, since they usually lack adjacency lists [22]. GES currently supports only top-inserts and, like KÙZU, does not support d-representations.

Kalumin and Deshpande [22] adapt the optimizer for factorization and introduce f-representations into DuckDB's vectorized query engine. As KÙZU, they perform table-level factorization and use vector groups. The tree structure is recreated by referencing elements with offset values from the contiguous memory chunks. This works well for top-inserts, where no random inserts are required. However, the contiguous memory cannot be efficiently used for bottom-inserts, which are not considered in their work.

Factorization can be interpreted as a top-down dynamic programming approach, since it computes joining subresults once for each value and then memoizes them. In contrast, the FAQ approach follows the bottom-up counterpart [1]. This approach was implemented in LogicBlox to efficiently process Datalog queries and builds the basis for modern systems like RelationalAI [1, 3, 4].

3D-Hashjoins [14] can be seen as limited variant of factorization, which implements hash table chains hierarchically and groups equivalent keys in subchains. During the probe, they only return a pointer to this chain and flatten the subchains later by traversing them. They are closely related to our top-inserts, except that we can keep the factorized representation when required, whereas 3D-Hashjoins remain flattened.

Recent work adapted this approach in RDBMSs: Umbra's lookup-expand joins split the chain expansion from the hash table lookup [9]. Recently, this was also integrated in interpreted environments [7]. Groß et al. [17] implemented a *Linear-Chained* hash table adapting 3D-hash tables for linear chaining in DuckDB. They decide at runtime if the vectorized system uses factorization or flat execution based on statistics on the current data. Their usage of collision-free chains can be interpreted as d-representations. Other approaches perform factorized aggregation on top of established RDBMSs within SQL queries [42, 47]. LMFAO [43] is a multi-query engine that calculates batches of aggregates over relational data simultaneously in a code-generating engine. Its view-based approach, which pushes aggregates past joins, is inspired by factorization and eager aggregation. IFAQ [45, 46] extends this with a multi-layer compilation framework that additionally applies algebraic optimizations to extract and hoist loop-invariant aggregates before the aggregate-pushdown stage. It targets end-to-end relational machine learning workloads. SDQL [44] provides a unified representation based on semi-ring dictionaries capable of expressing both factorized joins and aggregates. While SDQL uses nested dictionaries uniformly for both input relations and computation, we maintain explicit f-tree data structures between operators, whose hierarchical shape is specialized into generated code and allows subsequent operators to process factorized tuples directly without flattening.

Whereas factorization uses normalized input relations, NF^2 (*non-first normal-form*) [20, 28] uses nested input relations. Subsequent papers proposed models to process these representations in the query engine, e.g., with limitations like only working on the outermost relations or involving flattening [48]. Others extended the algebra to process NF^2 [13, 41]. Dremel [29, 30], a more recent approach on NF^2, did not support joins of multiple tables in its initial version. Combined with the limited parallelization opportunities caused by sequential dependencies in the data representation, the resulting design goals differ from those of traditional RDBMSs.

8 CONCLUSION

Although previous work demonstrated that factorization can significantly reduce the number of tuples processed during query execution, deployment to RDBMSs remains limited. In this work, we proposed a novel approach that maintains factorization over the complete query execution and avoids flattening. By building a limited relational code-generating engine, we demonstrated that factorization can significantly improve performance.

By generating f-tree-specific code, we preserve the state-of-the-art performance of flat engines, as we can eliminate significant overhead from query runtime. We extend current relational factorized approaches with bottom-inserts, which make factorized joins fully reorderable and enable new opportunities in factorized query optimization. Our comprehensive evaluation across synthetic and real-world workloads underlines the potential of factorization for substantial performance gains, particularly for queries that exhibit high redundancy or data skew. This highlights the practical viability of factorized query processing in modern RDBMSs and opens new avenues for optimizing complex query workloads.

REFERENCES

[1] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 13–28. https://doi.org/10.1145/2902251.2902280

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[3] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoč, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 283–296. https://doi.org/10.1145/3722212.3724450

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[5] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and ordering in factorised databases. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[6] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: a query engine for factorised relational databases. *Proc. VLDB Endow.* 5, 11 (July 2012), 1232–1243. https://doi.org/10.14778/2350229.2350242

[7] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. *Proc. VLDB Endow.* 18, 8 (April 2025), 2413–2426. https://doi.org/10.14778/3742728.3742737

[8] Pierre-Yves Bigourdan. 2016. *Distributed and Multi-Threaded Learning of Regression Models.* MSc Thesis. University of Oxford, Oxford, UK.

[9] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[10] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3662010.3663442

[11] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2022. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proc. VLDB Endow.* 15, 8 (April 2022), 1533–1545. https://doi.org/10.14778/3529337.3529339

[12] Radu Ciucanu and Dan Olteanu. 2015. *Worst-Case Optimal Join at a Time.* Technical Report. University of Oxford, Department of Computer Science. Latest version March 2016.

[13] Latha S. Colby. 1989. A recursive algebra and query optimization for nested relations. *SIGMOD Rec.* 18, 2 (June 1989), 273–283. https://doi.org/10.1145/66926.66952

[14] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D hash join: Building on non-unique join attributes. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, Ca, USA, January 9-12, 2022*. CIDR, Chaminade, 1–9. https://madoc.bib.uni-mannheim.de/62365/ Online-Ressource.

[15] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[16] Sen Gao, Jianwen Zhao, Hao Zhang, Shixuan Sun, Chen Liang, Gongye Chen, Wenliang Zhang, Bo Ren, Chao Liu, Chenyi Zhang, Quan Chen, Chao Li, Jingwen Leng, and Minyi Guo. 2025. GES: High-Performance Graph Processing Engine and Service in Huawei. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 391–403. https://doi.org/10.1145/3722212.3724439

[17] Paul Groß, Daniel ten Wolde, and Peter Boncz. 2025. Adaptive Factorization Using Linear-Chained Hash Tables. In *15th Annual Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. https://vldb.org/cidrdb/papers/2025/p21-gro.pdf

[18] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proc. VLDB Endow.* 14, 11 (July 2021), 2491–2504. https://doi.org/10.14778/3476249.3476297

[19] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4, Article 71 (July 2021), 37 pages. https://doi.org/10.1145/3447772

[20] Gerhard Jaeschke and Hans-Jörg Schek. 1982. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California, USA*, Jeffrey D. Ullman and Alfred V. Aho (Eds.). ACM, 124–138. https://doi.org/10.1145/588111.588133

[21] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[22] Hasara Kalumin and Amol Deshpande. 2025. Optimizing Queries with Many-to-Many Joins. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3668–3681. https://doi.org/10.1109/ICDE65448.2025.00274

[23] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1695–1698. https://doi.org/10.1145/3035918.3056445

[24] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905. https://doi.org/10.1007/S00778-020-00643-4

[25] Ahmad Khazaie and Holger Pirk. 2023. SonicJoin: Fast, Robust and Worst-case Optimal. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 540–551. https://doi.org/10.48786/EDBT.2023.46

[26] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[28] Akifumi Makinouchi. 1977. A Consideration on Normal Form of Not-Necessarily-Normalized Relation in the Relational Data Model. In *Proceedings of the Third International Conference on Very Large Data Bases, October 6-8, 1977, Tokyo, Japan*. IEEE Computer Society, 447–453.

[29] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[30] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[31] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[32] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[33] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The Complete Story of Joins (in HyPer). In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 31–50.

[34] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[35] Dan Olteanu and Jakub Závodný. 2012. Factorised representations of query results: size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory* (Berlin, Germany) *(ICDT '12)*. Association for Computing Machinery, New York, NY, USA, 285–298. https://doi.org/10.1145/2274576.2274607

[36] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1, Article 2 (March 2015), 44 pages. https://doi.org/10.1145/2656335

[37] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[38] Ambros Petrou. 2015. *Single-Round vs. Multi-Round Distributed Query Processing in Factorised Databases*. MSc Thesis. University of Oxford, Oxford, UK.

[39] Mark Raasveldt and Hannes Muehleisen. 2024. DuckDB. https://github.com/duckdb/duckdb [Online; accessed 13. July 2026].

[40] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2022. *Recommender Systems: Techniques, Applications, and Challenges*. Springer US, New York, NY, 1–35. https://doi.org/10.1007/978-1-0716-2197-4_1

[41] Hans-Jörg Schek and Marc H. Scholl. 1986. The relational model with relation-valued attributes. *Inf. Syst.* 11, 2 (1986), 137–147. https://doi.org/10.1016/0306-4379(86)90003-7

[42] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2882903.2882939

[43] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1642–1659. https://doi.org/10.1145/3299869.3324961

[44] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. https://doi.org/10.1145/3527333

[45] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. 2020. Multi-layer optimizations for end-to-end data analytics. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO '20)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3368826.3377923

[46] Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. 2021. An intermediate representation for hybrid database and machine learning workloads. *Proc. VLDB Endow.* 14, 12 (July 2021), 2831–2834. https://doi.org/10.14778/3476311.3476356

[47] Bernhard Stöckl and Maximilian E. Schüle. 2025. Fast Factorized Learning: Powered by In-Memory Database Systems. arXiv:2512.09836 [cs.DB] https://arxiv.org/abs/2512.09836 [Online; accessed 13. July 2026].

[48] Stan J. Thomas and Patrick C. Fischer. 1986. Nested Relational Structures. *Adv. Comput. Res.* 3 (1986), 269–307.

[49] Kangfei Zhao, Jiao Su, Jeffrey Xu Yu, and Hao Zhang. 2021. SQL-G: Efficient Graph Analytics by SQL. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2021), 2237–2251. https://doi.org/10.1109/TKDE.2019.2950620

[50] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1165–1180. https://doi.org/10.1145/3035918.3035943